Sparse Tensor Algebra Compilation

by

Fredrik Berg Kjølstad

B.E., Høgskolen i Gjøvik (2005)
M.S., University of Illinois (2011)

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

February 2020

Author . . . . . .
Department of Electrical Engineering and Computer Science
January 30, 2020

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

Abstract

This dissertation shows how to compile any sparse tensor algebra expression to CPU and GPU code that matches the performance of hand-optimized implementations. A tensor algebra expression is sparse if at least one of its tensor operands is sparse, and a tensor is sparse if most of its values are zero. If a tensor is sparse, then we can store its nonzero values in a compressed data structure, and omit the zeros. Indeed, as the matrices and tensors in many important applications are extremely sparse, compressed data structures provide the only practical means to store them. A sparse tensor algebra expression may contain any number of operations, which must be compiled to fused sparse loops that compute the entire expression simultaneously. It is not viable to support only binary expressions, because their composition may result in worse asymptotic complexity than the fused implementation. I present compiler techniques to generate fused loops that coiterate over any number of tensors stored in different types of data structures. By design, these loops avoid computing values known to be zero due to the algebraic properties of their operators.

Sparse tensor algebra compilation is made possible by a sparse iteration theory that formulates sparse iteration spaces as set expressions of the coordinates of nonzero values. By ordering iteration space dimensions hierarchically, the compiler recursively generates loops that coiterate over tensor data structures one dimension at a time. By organizing per-dimension coiteration into regions based on algebraic operator properties, it removes operations that will result in zero. And by transforming the sparse iteration spaces, it optimizes the generated code. The result is the first sparse iteration compiler, called the Tensor Algebra Compiler (taco). Taco can compile any tensor algebra expressions, with tensors stored in different types of sparse and dense data structures, to code that matches the performance of hand-optimized implementations on CPUs and GPUs.

Acknowledgments

I thank Saman Amarasinghe, my advisor, mentor, and friend through eight years at the institute. His intuition, energy, and care for students are second to none, and he pushed me to raise my standards as high as I could live with (and then some). I cherish our long arguments, as we wrestled with problem formulations, unpolished ideas, and designs so vague that they were not even wrong. Saman treated me as if I was what I should be, and he made me a better researcher and person.

The work in this dissertation was a collaboration with brilliant researchers. Although almost all the writing in this dissertation is new, most of the ideas are based on the papers we wrote together [78, 41, 79, 114]. Saman was involved from the start, helping shape early ideas and emerging design. Stephen Chou was my office mate and close collaborator from the first paper. He led the design of format support beyond dense and compressed, as well as the design of the format abstraction. Shoaib Kamil and I have worked together since he arrived at MIT a year after me, and he was the first to join the taco project after Saman and I. David Lugato visited MIT for a year and helped us develop and evaluate taco. Peter Ahrens and I worked together on the concrete index notation intermediate representation and the precompute and reorder transformations. Ryan Senanayake developed the other optimizing transformations as a Masters student with me. I had so much fun working with these amazing people and am forever thankful to them.

I have also collaborated with other researchers on projects that are not covered in this dissertation. Simit was a programming language we designed for physical simulation. It was a collaboration with Shoaib, Jonathan Ragan-Kelley, David Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman. It taught me a lot about sparse linear algebra that led me to the ideas in this dissertation, which I originally began exploring because I wanted a flexible compiler for Simit. Other dear collaborators during my PhD studies include Timo Schneider, Torsten Hoefler, and Joel Emer. Finally, I would like to thank the undergraduate and Masters students I have had the pleasure to work with: Gurtej Kanwar, Junda Huang, Parker Tew, Sachin Shinde, Mengyuan Sun, Patricio Noyola, Ryan Senanayake, Suzy Mueller, Rawn Henry, and Ziheng Wang. The joy of these relationships encouraged me to profess as my vocation.

I have had many mentors who gave me invaluable advice and support, particularly Saman, Marc Snir, Danny Dig, David Lugato, Anne Elster, and Charles Leiserson. My thesis committee, of Saman, Charles, and Martin Rinard, have also provided valuable advice and comments. I cannot possibly measure the impact each has had in my life. I will pay them back with the currency they value, by endeavoring to make a similar difference in the lives of those I mentor in the future.

I want to thank my many friends during graduate school, especially David Hayden, Raj Sodhi, Adam and Miranda Smith, Chris Johnson, Brett Jones, Mallory Casperson, Aparna Chandramowlishwaran, Coleen Silva, Frank Arne Antonsen, Ethan Crawford, Marisa and Angela Puccini, James Brodman, Ludwig Schmidth, the people of social reading, of TGIT and GSB, of the COMMIT group, and many others. They made my life colorful, interesting, challenging, and fun!

But I am most grateful to my mother and sister who supported me on my journey. My sister and nephews were a safe haven to me, and when I spent Christmas in Cambridge, my sister collected and sent me presents from home. My mother believed her children should live our lives on our own terms. She always supported me, even when I made unconventional choices like leaving a job in Norway to study in America. Her attitude is perhaps best captured by her reply when I told her I had been accepted to MIT: “You sound happy, so I am very happy for you, but what is MIT?” Mom passed away during my graduate studies, and I dedicate my dissertation to her.

Dedicated to my mother, Turid Helland.

Contents

1 Introduction **10**
- 1.1 A Combinatorial View . . . 12
- 1.2 The Issue with Libraries . . . 13
- 1.3 The Sparse Tensor Algebra Compiler . . . 17
- 1.4 Contributions and Scope . . . 22
- 1.5 Dissertation Overview . . . 24

2 Data Structure Abstractions **25**
- 2.1 Coordinate Relations . . . 25
- 2.2 Coordinate Trees . . . 26
- 2.3 Level Abstraction . . . 27
- 2.4 Six Level Types . . . 31
- 2.5 Tensor Formats . . . 35
- 2.6 Conclusion . . . 39

3 Sparse Iteration Spaces **40**
- 3.1 Iteration Space Algebra . . . 41
- 3.2 Iteration Graphs . . . 43
- 3.3 Iteration Lattices . . . 50
- 3.4 Conclusion . . . 57

4 Tensor Notations **58**
- 4.1 Matrix Multiply Example . . . 58
- 4.2 Tensor Index Notation . . . 62
- 4.3 Concrete Index Notation . . . 63
- 4.4 Concretize Algorithm . . . 67
- 4.5 Conclusion . . . 68

5 Coiteration Code Generation **70**
- 5.1 Algorithm Overview . . . 70
- 5.2 Coiteration Code . . . 73
- 5.3 Derived Iteration Spaces . . . 78
- 5.4 Compute and Assembly . . . 80
- 5.5 Conclusion . . . 81

6 Optimizing Transformations **83**
6.1 Reorder . . . 84
6.2 Precompute . . . 86
6.3 Collapse . . . 89
6.4 Split . . . 89
6.5 Bound . . . 90
6.6 Iteration Space Mapping . . . 90
6.7 Conclusion . . . 92

7 Evaluation **94**
7.1 Experimental Setup . . . 95
7.2 Expressions Matter . . . 96
7.3 Formats Matter . . . 97
7.4 Optimizations Matter . . . 101
7.5 Kernels are Competitive . . . 103

8 Related Work **112**
8.1 Sparse Compilers . . . 114
8.2 Sparse Kernel Libraries . . . 119
8.3 Sparse Programming Systems . . . 122
8.4 Dense Programming Systems and Compilers . . . 124

9 Conclusion **127**

Bibliography **131**

List of Figures and Tables

1-1 Combinatorial explosion of implementations . . . 11
1-2 Amazon product review tensor . . . 13
1-4 Sampled Dense-Dense Matrix Multiplication . . . 15
1-3 Fused and Unfused SDDMM Performance Results . . . 15
1-5 The Tensor Algebra Compiler (taco) . . . 17
1-6 C++ taco API . . . 19
1-7 The taco code generation web tool . . . 19
1-8 Unscheduled GEMV C implementation . . . 20
1-9 Unscheduled CSR SpMV C implementation . . . 20
1-10 Unscheduled DCSR SpMV C implementation . . . 20
1-11 Unscheduled CSR SpMSpV C code . . . 20
1-12 Unscheduled coordinate SpMV C implementation . . . 20
1-13 Scheduled CPU SpMV OpenMP C implementation . . . 21
1-14 Scheduled GPU SpMV CUDA implementation . . . 21

2-1 Coordinate tree with duplicates . . . 26
2-2 Coordinate tree of a dense matrix . . . 27
2-3 Coordinate tree of a row-major sparse matrix . . . 27
2-4 Coordinate tree of a column-major sparse matrix . . . 27
2-5 Coordinate tree level . . . 27
2-6 Full tree level property . . . 28
2-7 Ordered tree level property . . . 28
2-8 Unique tree level property . . . 29
2-9 Branchless tree level property . . . 29
2-10 Compact tree level property . . . 29
2-11 The properties of the six level types. . . . 31
2-12 The capabilities of the six level types . . . 31
2-15 Data structure of dense levels . . . 32
2-16 Data structure of compressed levels . . . 32
2-17 Data structure of singleton levels . . . 32
2-18 Data structure of hashed levels . . . 32
2-13 Access functions of the six level types . . . 33
2-14 Assembly functions of the six level types . . . 34
2-19 Data structure of range levels . . . 35
2-20 Data structure of offset levels . . . 35
2-21 Data structures of vectors . . . 35
2-22 Data structures of row-major unstructured matrices . . . 36
2-23 Compressed Sparse Rows (CSR) format descriptor . . . 36

2-24 Data structures of column-major unstructured matrices . . . 37
2-25 Compressed Sparse Columns (CSC) format descriptor . . . 37
2-26 Data structures of structured matrices . . . 37
2-27 Data structures of blocked matrices . . . 38
2-28 Data structures of 3-order tensors . . . 38

3-1 Dense iteration space . . . 40
3-2 Sparse iteration space . . . 40
3-3 Sparse matrix addition . . . 40
3-4 Sparse iteration space of a coordinate tree . . . 41
3-5 Sparse iteration space examples . . . 42
3-8 Sparse iteration space of a broadcast expression . . . 42
3-6 Sparse iteration space of binary point-wise expressions . . . 42
3-7 Sparse iteration space of a tertiary point-wise expressions . . . 42
3-9 Sparse iteration graph of a $B_{ij} \cup C_{ij}$ . . . 43
3-10 Sparse iteration graph examples . . . 44
3-11 Sparse iteration graph with branches . . . 45
3-12 Sparse iteration space formed by concatenating spaces . . . 45
3-14 Derived index variable with a split relation . . . 48
3-15 Iteration over a coordinate tree before and after collapse . . . 48
3-16 Derived index variable with a collapse relation . . . 49
3-17 Three ways to split up a two-dimensional space . . . 50
3-18 Iteration regions of $b \cup c$ . . . 51
3-19 Iteration regions of $b \cap c$ . . . 51
3-20 Iteration regions of $(b \cup c) \cap d$ . . . 51
3-21 Iteration domain simplifications as segments run out of coordinates . . . 52
3-22 The iterate-and-locate strategy . . . 52
3-23 Iteration lattice of $(b \cup c) \cap d$ . . . 53
3-24 Lattice point segments divide into iterators and locators sets . . . 53
3-25 Iteration lattice and coiteration pseudocode . . . 53
3-27 Iteration lattice examples . . . 54
3-26 A sublattice rooted at a lattice point $\mathcal{L}_p$ . . . 54
3-28 An iteration lattice constructed from iteration subexpression . . . 54
3-29 Iteration lattices of a segment expression . . . 55
3-30 Iteration lattice of an iteration domain . . . 55
3-33 Iteration lattice intersection optimization . . . 55
3-31 Iteration lattice resulting from an intersection . . . 56
3-32 Iteration lattice resulting from a union . . . 56
3-34 Iteration lattice subset optimization . . . 56

4-1 The languages in the compiler overview . . . 58
4-2 Matrix multiplication iteration graph for the inner product algorithm . . . 59
4-3 Matrix multiplication iteration graph for the linear combination of rows algorithm 60
4-6 An $m \times n$ matrix and its sparse CSR index . . . 60
4-4 Sparse matrix multiplication code with CSR operands and dense result . . . 61
4-5 Sparse matrix multiplication code with CSR operands and CSR result . . . 61
4-7 Matrix multiplication iteration graph with a temporary row vector . . . 62
4-8 Tensor index notation example . . . 62

4-9 Grammar of tensor index notation . . . 63
4-10 Concrete index notation example . . . 63
4-11 Grammar of concrete index notation . . . 64
4-12 Assignment statement . . . 64
4-13 Forall statement . . . 65
4-14 Where statement . . . 65
4-15 Sequence statement . . . 65
4-16 Multi statement . . . 65

5-1 The lowering step in the compiler overview . . . 70
5-2 Algorithm to lower concrete notation to imperative IR . . . 71
5-3 A sublattice rooted at a lattice point $\mathcal{L}_p$ . . . 76
5-4 Advancing collapsed index variables . . . 77
5-5 Generated loop for lattice with a single dimension iterator . . . 77
5-6 Generated loop for lattice with a single position iterator . . . 77
5-7 Coordinate mapping between iteration spaces . . . 79

6-1 The schedule step in the compiler overview . . . 83
6-2 Reorder transformation example . . . 86
6-3 Precompute transformation SpGEMM example . . . 89

7-1 Combinatorial explosion of kernels . . . 94
7-2 Performance of fused and unfused implementations of SDDMM . . . 98
7-3 Format selection to fit matrix nonzero structures . . . 99
7-4 Format selection to balance data structure reorganization and computation costs . 100
7-5 Scheduling for load-balance . . . 102
7-6 SpMV on a CPU with the CSR format . . . 105
7-7 SpMV on a GPU with the CSR format . . . 105
7-8 MTTKRP CPU with the CSF format . . . 106
7-9 MTTKRP GPU with the CSF format . . . 106
7-10 Comparison of taco and MATLAB TTB on four tensor expressions . . . 107
7-11 SpMM CPU with the CSR format . . . 108
7-12 SpMM GPU with the CSR format . . . 108
7-13 SpMV on a CPU with the DIA format . . . 110
7-14 SpMV on a CPU with the ELL format . . . 110
7-15 Matrix Addition CPU with the CSR format . . . 111

8-2 Tensor algebra programming system with a kernel library . . . 113
8-3 Tensor algebra programming system with a compiler . . . 113
8-4 Compilation approach to tensor algebra . . . 115
8-5 Optimization approach to tensor algebra . . . 115

Chapter 1

Introduction

"The complexity of the problem will defeat us unless we find a simple way of writing it down, which lets us break it into smaller problems."

— Christopher Alexander

"A composition is always more than the sum of its parts."

— Yo-Yo Ma

1.1 A Combinatorial View

1.2 The Issue with Libraries

1.3 The Tensor Algebra Compiler (taco)

1.4 Contributions and Scope

1.5 Dissertation Overview

Thesis statement:
Sparse tensor algebra can be put on the same compiler transformation and code generation footing as dense tensor algebra and array codes.

The Fortran compiler [12] brought first-class support for dense loops and arrays to programming languages in 1957. Since then, we have made significant strides on compilation techniques for dense loop nests, leading to dependence analysis [96, 18], the polyhedral model [85, 5], and specialized compilers for stencil and dense tensor operations [9, 107]. The dense compilation techniques support sophisticated iteration space transformations, different data layouts, and portable compilation to general-purpose processors, accelerators, and domain-specific hardware. They also support fusion to improve temporal locality and keep data in caches.

Sparse operations go back almost as far, and the first sparse linear algebra implementations appeared in the 1960s [113, 124][1]. But there has not been a similar development of a first-class sparse language construct backed by general compilation techniques. I think the reason is the complexity of sparse operations that stem from the irregular data structures on which they operate. Many irregular data structures have been designed only for matrices and are optimized for different types of matrices, operations, and processors. Both the complexity and number of possible data structures grow significantly when we generalize to blocked matrices and tensors. Sparse operations may coiterate over several data structures at the same time, manage temporary data structures, and assemble results. The resulting code contains nested `while` loops that coiterate over nested data structures, with `if` statements to ensure that they compute values only at those data structure intersections and unions that actually contribute to the result. There is an almost biological complexity to many of these codes, with indirect memory references that access different data structures at different loop levels interleaved in a fine web. When we also consider loop optimizations, parallelization, vectorization, and compilation to accelerators, we face a substantial challenge.

As a result, current practice is either to implement sparse operations by hand on top of dense loop and array abstractions or to compose library functions.

But there is something broken about function composition that

[1] A linear or tensor algebra expression is sparse if at least one of its operands is sparse, and an operand is sparse if most of its values are zero. If a matrix, vector, or tensor is sparse, then we can store its nonzero values in a compressed data structure, and omit the zeros.

shows up in sparse tensor algebra. Modern sparse linear and tensor algebra libraries contain handwritten functions that each compute a single expression on operands stored in specific data structures. These functions are composed by programmers or a programming system to express compound operations. For example, to implement the linear algebra operation $A = B \odot (CD)$ from the data analytics literature, where B is a sparse matrix and $\odot$ is an element-wise multiplication, we can write

```
Matrix T = gemm(C, D);
Matrix A = spelmul(B, T);
```

where T is a dense temporary matrix variable. Its materialization leads to three issues:

1. Reduced temporal locality because values are produced long before they are consumed, which may cause them to be evicted from caches.

2. Forced data structure reorganization if the two functions require different data structures.

3. Increased asymptotic complexity when values produced by the first function are not used.

The final issue is alarming and unique to sparse codes. I discuss the issues with libraries in greater depth in Section 1.2.

Another challenge with hand-writing sparse linear and tensor algebra libraries is the combinatorial explosion of implementations that come from the Cartesian combination of expressions, data structures, processor types, and optimizations:

$$\text{expressions} \times \text{formats} \times \text{optimizations} \times \text{architectures}.$$

Figure 1-1 shows sufficient real-world examples to make it infeasible to hand-implement all combinations, and the number of possible combinations is infinite. We can implement any, but not all.

These issues with composing expressions with each other and with data structures, optimizations, and architectures point to a deeper problem of software abstraction. Our abstraction mechanisms impose friction at abstraction boundaries that shows up in force in sparse tensor algebra and requires us to resort to enumeration for performance. We need abstraction without friction.

This dissertation outlines how to implement abstractions without friction by moving the abstractions into the compiler. The compiler then weaves separate descriptions of tensor algebra expressions, data structures, and optimizations into bespoke and frictionless implementations for the given architecture.

To achieve sparse tensor algebra compilation, I present a sparse iteration theory, where sparse iteration spaces are described as set expressions of hierarchical data structures. I show how these iteration

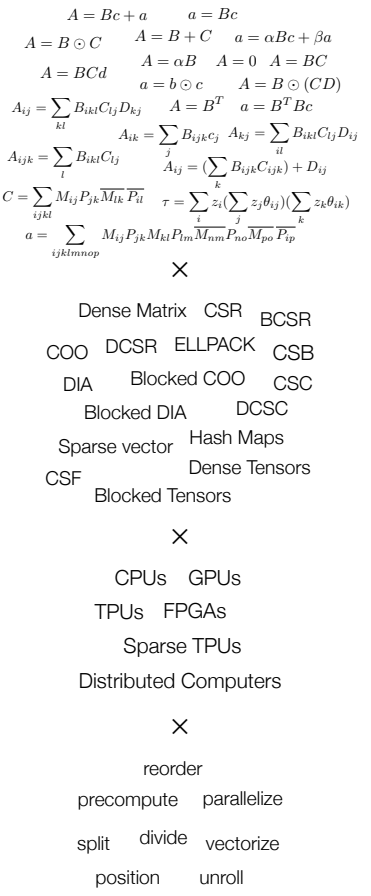


Figure 1-1: The Cartesian combination of expressions, data structures, optimizing transformations, and computer architectures produces a combinatorial explosion of implementations.

spaces can be expressed, optimized, and lowered to code that iterates over them by coiterating over intersections and unions of the irregular data structures in each dimension. I then extend the sparse iteration space descriptions to include tensor algebra expressions that compute tensor expressions and subexpressions in different iteration space regions. Finally, I demonstrate that these expressions can be optimized and compiled to efficient and portable code for CPUs and GPUs. In short, I show the following:

Thesis statement Sparse tensor algebra can be put on the same compiler transformation and code generation footing as dense tensor algebra and array codes.

1.1 A Combinatorial View

Tensors are often used to describe linear relationships between discrete objects, such as people, products, words, movies, webpages, and robotic limbs. One way to conceptualize a tensor as a set of relationships is to consider the tensor modes[2] as sets of objects and the entries as weighted relationships between these objects. For example, a matrix has two modes, and its entries are linear relationships between the objects of these modes. An example is a square matrix where the modes represent people, and where the entries measure the strength of their friendships. Other examples include how much a person likes a movie and the interaction between two strands of hair on a character in an animated movie. These relationships are often sparse, meaning that most objects have no relationship, leading to sparse tensors where most components are zero. Tensor sparsity thus comes from connections of the underlying system of objects. For example, most people do not know each other, most webpages do not connect to each other, and I have zero opinion on most movies because I have not seen them.

An example is a dataset of product reviews from the Amazon web store spanning 18 years up to March 2013, which was used by McAuley and Leskovec [91] to predict how users respond to new products. Figure 1-2 shows the dataset as a 3-order tensor where the tensor dimensions are Amazon customers, products, and words in the English language, and where a nonzero means that person used that word that many times to review the given product. We can factorize this tensor with one of the generalizations of singular value decomposition (SVD) [50, 58] to higher dimensions, such as the canonical polyadic decomposition (CP) [66]. This approach can provide us with insight into its latent components, such as product categories, which can help us make better product recommendations.[3] This tensor is extremely sparse. If we were to store it in a dense array, it would consume 13.4 exabytes, assuming that each component consumes a single byte. If, on the other hand, we store only the coordinates and values of nonzeros, then it fits in only 14.6 gigabytes.

[2] A tensor mode is a set, and the Cartesian combination of these sets identifies all the tensor components. A k-order tensor has k modes. For example, a matrix is a 2-order tensor with two modes, called rows and columns, and a vector is a one-order tensor with one mode.

[3] For instance, a key component of the winning submission of the Netflix Challenge for movie recommendations was an extension of the SVD algorithm [82] that incorporated a temporal dimension to the collaborative filtering of a (customer $\times$ movie) ranking matrix.

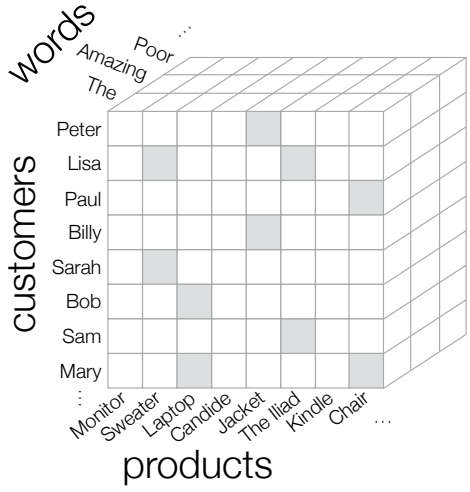


Figure 1-2: Amazon product review tensor with dimensions (customers × products × words), representing how many times that customer used that word to review that product. Since most customers never use most words to review most products, the tensor is extremely sparse with 8 billion zeros for every nonzero.

1.2 The Issue with Libraries

The state of the art for sparse linear and tensor algebra is to use libraries. These libraries are collections of hand-optimized functions that compute a single expression, on specific data structures, on a specific machine, with specific optimizations. Sometimes, these functions are written to support a class of expressions, such as all tensor-times-vector multiplications. In the absence of code generators, we have found that the generality of these functions comes at a cost, such as the functions of the MATLAB Tensor Toolbox. Many of these functions can compute a class of expressions, but they can take up to twenty times as long to compute and more memory than functions written for specific operations. Examples of optimized sparse linear and tensor algebra libraries include Eigen [60], Intel MKL [44], and SPLATT [116]. There also exist programming systems, such as MATLAB [123], Julia [22], and the Cyclops Tensor Framework [119], that provide support for general classes of expressions built on top of hand-optimized functions. They must, therefore, map expressions and data structures to the implementations at hand and suffer a mapping cost in addition to the friction imposed by function composition.

Libraries of hand-written functions face the fundamental limitation that you can write only so many functions. An engineering team can push the boundary for important operations—Intel MKL [44] provides around 80 variants of the (sparse matrix)$\cdot$(dense vector) (SpMV) operation. But this paradigm breaks down in modern use cases, such as data analytics, where you require more than one sparse operand (e.g., a sparse vector), compound operations with many operands, higher-order tensors, and the ability to target different architectures (e.g., accelerators). Each of these features adds another dimension to the combinatorial space of functions that must be written. And due to the fact that some of these dimensions have an unlimited number of possible

variants, we must either choose a small subset to implement or move to a meta-programming approach, where a code generator produces implementations on demand.

The issue with libraries of hand-optimized functions boils down to our inability to easily build composable building blocks that perform well, using modern programming paradigms. Current sparse linear and tensor algebra libraries do not let us compose separate specifications of expressions, data structures, optimization strategies, and architectures without sacrificing performance. And functions that implement different expressions do not compose without losing performance compared to a fused implementation. The first performance loss is from lost temporal locality—a deficiency that is also present with dense operations. Second, sparse operations may operate on many different data structures, which are each designed to work well on one type of sparsity pattern. If two composed functions do not support the same data structure, then it becomes necessary to perform an expensive conversion between irregular data structures. But the most serious issue is that composing two sparse linear or tensor algebra functions may perform *asymptotically worse* than a fused function written to compute the entire expression at once.

The asymptotic complexity of the fused implementation of two tensor algebra operations is different from the asymptotic complexity of separate implementations when the result of the first operation is multiplied by a sparse tensor. The complexity then changes to a function of the number of nonzeros, instead of a function of the size of the tensor modes. The reason is that the fused operation can traverse the nonzeros of the sparse operand and compute only those values that are multiplied by those nonzeros. With separated operations, the first operation does not know what values will be needed and must, therefore, compute them all. This inefficiency may seem like a theoretical point that arises accidentally from the annihilation property of multiplication. But it is fundamental because the zeros of a sparse tensor themselves derive from the locality properties of the system the tensor represents. A sparse multiplication should, therefore, be interpreted as a scaled intersection. Instead of a computation that happens to be multiplied by a zero, a better interpretation is that the computation did not make sense in the first place.

The futility of these operations can be seen in an example from data analytics: Suppose that we want to compute a similarity metric between friends in a social network. A natural way to express this computation with linear algebra is by storing the properties of each person as rows in a dense matrix and the friendship relationships in a separate sparse matrix.From a database point of view, the dense matrix is a table of properties with one row per person, whereas the sparse matrix is a table of friend relationships with one row per friendship. To compute the similarity of friends with linear algebra, we first square the dense property matrix to compute the distance metrics between pairs of persons. Then, we element-wise multiply it by the sparse friend-

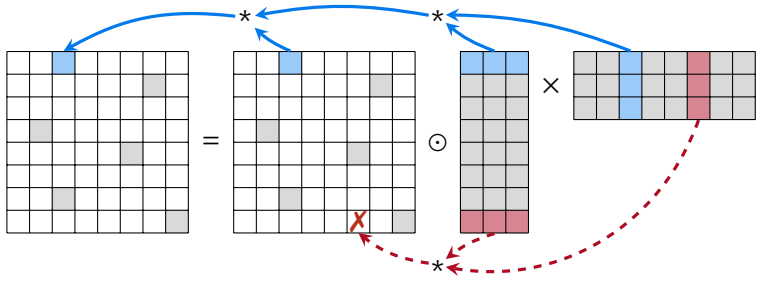


Figure 1-4: The SDDMM linear algebra expression $A = B \odot (CD)$, where A and B are sparse matrices, C and D are dense matrices, and $\odot$ denotes element-wise matrix multiplication. Also called Sampled Dense-Dense Matrix Multiplication (SDDMM), the expression is used in the alternating least squares matrix factorization algorithm [137]. To compute component (4, 5) of the result, that component from B is multiplied by the inner product of row 4 and column 5 from C and D. Wherever B is sparse, however, no operation is needed and computing those inner products is wasteful.

ship matrix to sample the distance metrics for pairs of friends. Clearly, we would do far too much work if we compute similarity metrics between all pairs of persons when we need them only between friends. In fact, if we assume that each person on average has a constant number of friends—perhaps Dunbar's number [49] due to the size of our neocortex—then we have done asymptotically too much work.[4]

The generalization of this linear algebra operation where different dense property matrices are multiplied is referred to as the sampled dense-dense matrix multiplication (SDDMM):

$$A = B \odot (CD).$$

It computes a sparse $n \times m$ matrix A as the element-wise multiplication ($\odot$) of a sparse $n \times m$ matrix B with the product of a dense $n \times k$ matrix C and a dense $k \times m$ matrix D. This is the core operation of the alternating least squares algorithms for matrix factorization [137]. Figure 1-4 demonstrates the redundant work. A tall and skinny matrix is squared and element-wise multiplied by a sparse matrix with six nonzeros. If the expression is computed as separate operations, the complexity is

$$\Theta(mnk)$$

due to the dominant cost of the dense matrix multiplication. If, on the other hand, it is computed in a fused implementation that iterates over the sparse matrix and computes only dot products the results of which are multiplied by a nonzero, then the complexity changes to

$$\Theta(nnz_B \cdot k),$$

where nnz_B is the number of nonzeros in B. If we assume that nnz_B grows at a slower rate than nm, then the fused operation has lower asymptotic complexity. The arrows demonstrate the redundant work that is avoided by the fused code. The blue arrows on the top show a dot product whose result is multiplied by a nonzero in B and that therefore produces a nonzero result. The red dashed arrows on the bottom, on the other hand, show a dot product whose result is multiplied by a zero. This operation does not contribute a nonzero to the result; hence, it does not need to be done in the first place. Finally, Figure 1-3 shows this effect empirically.

[4] In fact, we need only to assume that friends per person does not increase linearly with the size of the human race.

	unfused	fused	speedup
ca-HepPh	0.2	0.002	68
email-Enron	1.2	0.012	101
soc-Epinions1	5.3	0.018	291

Figure 1-3: Loop fusion on sparse operations result in asymptotically less work if work done in one loop is not used by the next. The table shows this effect for the SDDMM linear algebra operation, used in tensor factorization, on three sparse matrices constructed from the SNAP dataset [88]. Times are given in seconds. In these matrices, the sparsity increases with the size, and as the sparsity increases the speedup also increases. See Figure 7-2 for a full scaling experiment.

SDDMM is just one example of an expression where operations should be applied only to a subset. Other examples include:

Tensor Factorization: Zhang et al. [136] show that SDDMM is a special case of a class of operations they refer to as tensor times tensor products (TTTP). These operations have applications to the quadratic optimization step in the alternating least squares algorithm for tensor factorization. TTTP takes the form

$$A_{i_1,\ldots,i_N} = B_{i_1,\ldots,i_N} \sum_k \prod_{n=1}^{N} C^{(n)}_{i_n,k},$$

where N is the order of the result matrix and $C^{(n)}$ is matrix number n. That is, for an N-order tensor, we multiply N dense matrices and sample from the result. The notation is tensor index notation, where index variables used in subscripts—e.g., i_1 and k—index into tensor modes. Each index variable ranges over the values of the modes it indexes, which must be the same sets.

Breadth-first search: Yang et al. [135] show how a breadth-first search can be efficiently implemented with sparse matrix-vector multiplication. They use a masking vector b to avoid needless computation and thus reduce the operation's asymptotic complexity. The resulting linear algebra operation is

$$a = b \odot Cd,$$

where all operands are sparse.

Triangle queries: Azad et al. [10] show how to implement an efficient linear algebra algorithm for triangle counting by using a masking matrix. The algorithm finds wedges by multiplying the lower triangular part of the graph adjacency matrix by the upper triangular part. In graph terms, L directs the edges in one direction, and U directs them in the other. The algorithm then closes the wedges to form triangles by element-wise multiplying the LU result by the entire matrix. By computing the two operations with one linear algebra primitive, we avoid computing wedges that will not close. The resulting operation is

$$A = B \odot (LU),$$

where B is the adjacency matrix of a graph, L is the lower triangular part of B, and U is the upper triangular part of B.

I believe that many such operations will arise as we apply tensor algebra to data analytics, where we are often interested in computing information about a subset of a system. In fact, the GraphBLAS API [90, 36] for graph algorithms expressed as linear algebra makes extensive use of masking vectors and matrices to omit needless computation.

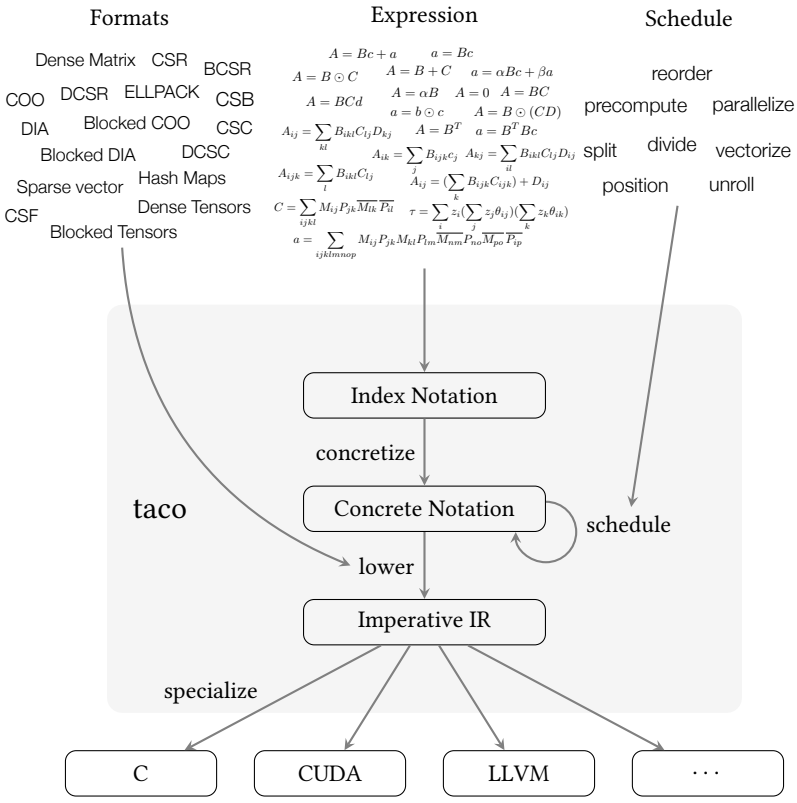


Figure 1-5: An overview of the Tensor Algebra Compiler (taco). The input to the compiler is an expression in the tensor index notation, formats that describe the data structure of each tensor, and a schedule that describes optimizing transformations to apply to the expression before generating code.

1.3 The Sparse Tensor Algebra Compiler

This dissertation describes a sparse iteration theory. I use this theory to create several intermediate representations (IRs) and a code generation lowering algorithm that make it possible to compile sparse tensor algebra expressions.[5] The IRs and the algorithm have been implemented in a compiler called the Tensor Algebra Compiler (abbreviated to taco). Taco is the first sparse iteration compiler, and it can compile any tensor index notation expression to code that computes it by coiterating over sparse and dense tensor data structures. In this section, I outline the rationale and design of the compiler, showing how the concepts described in this dissertation fit together and providing a guide to future implementers of systems built on these ideas.

As shown in Figure 1-5, taco has three types of inputs:

1. **a tensor algebra expression** to compute,
2. **tensor formats** that describe the tensor data structures, and
3. **a schedule** of optimizing transformations that change the order of computation, where temporaries are stored, and what parallel hardware to use.[6]

The compiler transforms the index notation expression to concrete notation, applies the schedule transformations to optimize it, and then

[5] A lowering transformation is a code generation algorithm that takes code in a higher-level abstract IR and produces code in a lower-level detailed IR.

[6] The separation of transformations and compiler internals was developed in the CHiLL compiler [39]. The Halide system [107] pioneered a clean API design that separates algorithms from schedules. This design lets users override the default schedules when they need more performance. An automatic scheduling system for taco is future work, perhaps using machine learning, optimization, or autotuning. The scheduling API and the compiler techniques described in this dissertation, however, enable such scheduling systems by automating the transformations and code generation.

lowers it to an imperative IR. This imperative code computes the expression by coiterating over the data structures described by the tensor formats. Finally, the imperative IR is specialized to C code, CUDA GPU code, or binary code through LLVM. These backends map the imperative IR to loops and parallel constructs, as described by the schedule.

The taco compiler is implemented inside a C++ library for tensor algebra. It can be used the same way as any other library; a user does not need to know that it generates code behind the scenes. Like the Eigen library [60], the taco C++ API makes extensive use of operator overloading and deferred execution to let users express operations cleanly. Taco also has a Python API, and it can even be used as a sparse linear and tensor algebra library generator—through the C++ API, a command-line tool, or a web tool. The code generation tools take descriptions of expressions and formats as inputs, and they produce a C or CUDA function that can be copied into an application.

I use a simple matrix-vector multiplication example to show how taco is used and how formats and schedules affect the generated code. The example comes from the linear algebra subset of tensor algebra. It can be written in linear algebra and tensor index notation as

$$a = Bc \text{ and } a_i = \sum_j B_{ij} c_j.$$

The tensor index notation uses subscripted index variables—i and j in this example—to implicitly range over the modes of the tensors they index, which requires modes indexed by the same index variable to be the same set. The sparse matrix-vector multiplication is likely the most important function in the sparse tensor algebra. It is simple compared to many tensor algebra expressions, but it is sufficient to show the profound impact of formats and schedules on the generated code. Figure 1-6 shows a small C++ code fragment that uses the taco library to multiply a sparse matrix in the compressed sparse row (CSR) format by a dense vector. The matrix is read from a file in the Matrix Market format (mtx) [97], whereas the vector is read from a file in the FROSTT tensor format (tns) [117]. Figure 1-7 shows a GUI for generating the same code, using the taco web tool.

The chosen formats dictate the generated code, which must iterate and coiterate over their data structures. Figure 1-8 through Figure 1-12 show code snippets, which I call compute **kernels**, that compute matrix-vector multiplications on matrices and vectors in different formats. These codes are unscheduled, which means that they use the default schedule that computes inner products without parallelization, strip-mining, vectorization, or unrolling. Their only difference is their formats. The dense matrix-vector multiplication (GEMV) code in Figure 1-8 is the simplest variant because the loops are independent of the matrix and vectors. They iterate over the entire $M \times N$ iteration space and randomly access the operands. The sparse matrix-vector multiplication (SpMV) code in Figure 1-9 shows how sparse expressions are

```
Tensor<double> a({A.getDimension(0)}, Format({dense}));
Tensor<double> B = read("pwtk.mtx",   Format({dense,compressed}));
Tensor<double> c = read("b.tns",      Format({dense}));

IndexVar i, j;
a(i) = B(i,j) * c(j);

write("a.tns", a);
```

Figure 1-6: C++ code that computes the sparse-matrix vector multiplication using the taco C++ deferred execution API.

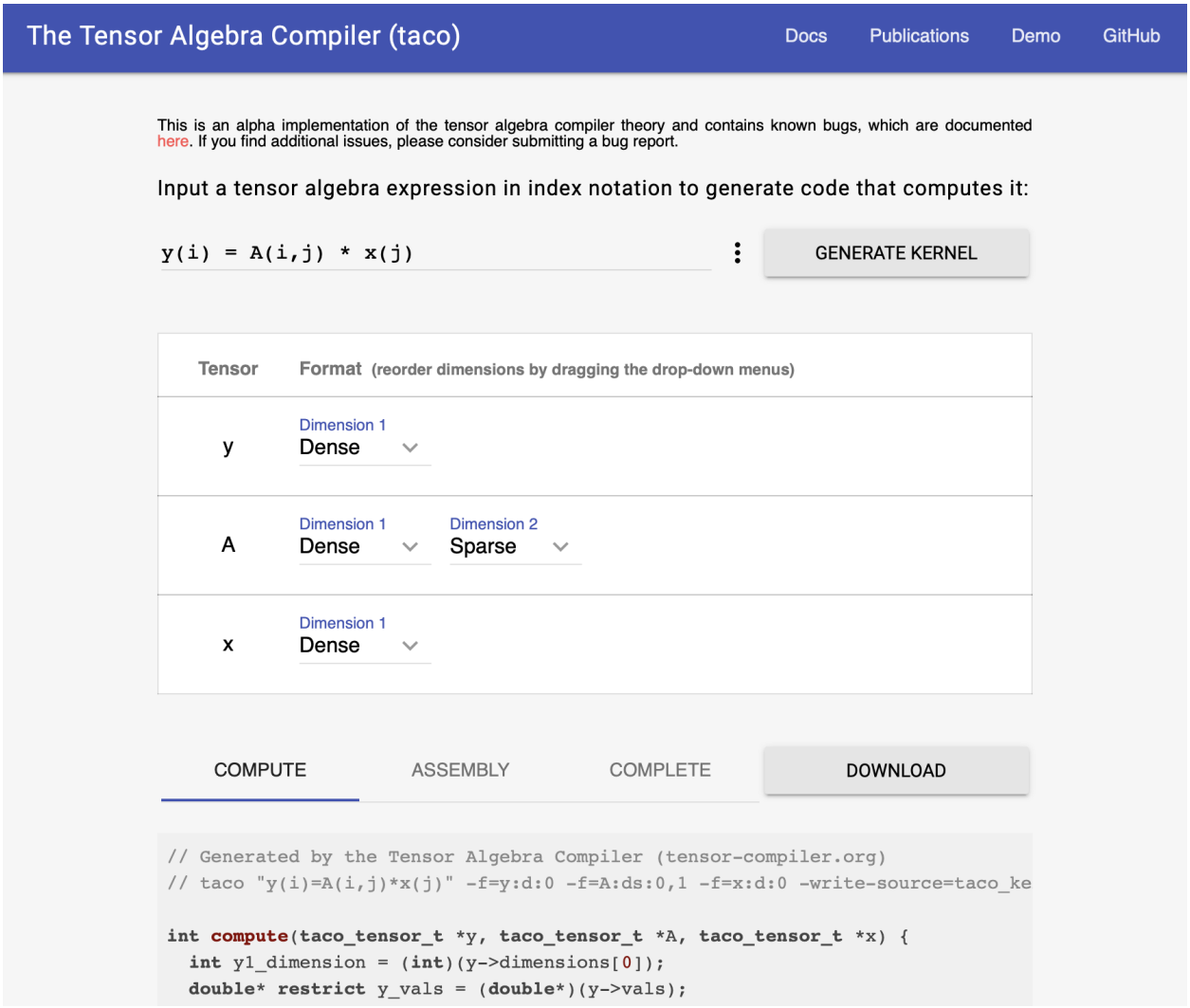


Figure 1-7: The taco web tool for generating sparse tensor C functions. The user enters an expression in the text box, selects formats from the drop-down boxes that appear, and presses the generate kernel button. A C header file with a function that implements the expression then shows up at the bottom of the page.

optimized by iterating over only the subset of components that are nonzeros. Because the coordinates of the nonzeros are stored as compressed rows in the CSR matrix, the loop iterates over them instead of over the entire row. The next SpMV code, in Figure 1-10, computes with a matrix in the doubly compressed sparse rows (DCSR) format.

This format compresses both the set of rows and each row; thus, the outer loop also iterates over a compressed data structure. The inner loop now obtains row locations from the outer loop's data structure. The sparse matrix-sparse vector multiplication (SpMSpV) code in Figure 1-11 shows how multiple sparse operands lead to code that coiterates over multiple compressed data structures. In this case, because the operation is multiplication, which is nonzero only if both operands are nonzero, the inner loop coiterates over the intersection of each compressed matrix row and the compressed vector.[7] Finally, Figure 1-12 computes SpMV with a matrix in the coordinate format. Both coordinates are stored for each component, requiring only one loop that iterates over the coordinates and the value.

Changes to the schedules given to taco may also have a drastic effect on the generated code. Figure 1-13 shows a scheduled SpMV C code that does well on modern Intel CPUs, and Figure 1-14 shows a scheduled SpMV CUDA device kernel that does well on modern NVIDIA GPUs. These kernels are very different. The CPU code is not far re-

[7] A multiplication requires a single `while` loop, but expressions involving additions, as we will see in Section 3.3, result in more `while` loops.

```
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    int p = i*N + j;
    a[i] += B[p] * c[j];
  }
}
```

Figure 1-8: Unscheduled GEMV: $a = Bc$ with a dense matrix and dense vectors. Two nested loops iterate over the entire $M \times N$ iteration space, computes a location in B, and computes the product.

```
for (int i = 0; i < M; i++) {
  a[i] = 0.0;
}
for (int i = 0; i < M; i++) {
  for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
    int j = B_crd[p];
    a[i] += B[p] * c[j];
  }
}
```

Figure 1-9: Unscheduled CSR SpMV: $a = Bc$ with a CSR matrix and dense vectors. The dense outer loop iterates over every row, while the inner loop iterates over only columns with nonzero values.

```
for (int i = 0; i < M; i++) {
  a[i] = 0.0;
}
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int i = B1_crd[pB1];
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
    int j = B2_crd[pB2];
    a[i] += B[pB2] * c[j];
  }
}
```

Figure 1-10: Unscheduled DCSR SpMV: $a = Bc$ with a DSCR matrix and dense vectors. The outer loop now also iterates over a compressed data structure that dictates row positions.

```
int pa = 0;
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  a[pa] = 0.0;
  int pB2 = B2_pos[pB1];
  int pc  = c1_pos[0];
  while (pB2 < B2_pos[pB1+1] && pc < c1_pos[1]) {
    int jB = B2_crd[pB2];
    int jc = c1_crd[pc];
    int j = min(jB,jc);
    if (jB == j && jc == j) {
      a[pa] += B[pB2] * c[pc];
    }
    if (jB == j) pB2++;
    if (jc == j) pc++;
  }
  pa++;
}
```

Figure 1-11: Unscheduled DCSR SpMSpV: $a = Bc$ with a DCSR matrix and compressed vectors. The `while` loop coiterates over the intersection of each matrix row and the vector, as a multiplication is nonzero only if both operands are nonzero, in which case they are multiplied and appended to the result.

```
for (int i = 0; i < M; i++) {
  a[i] = 0;
}
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int i = B1_crd[pB1];

  int j = B2_crd[pB1];
  a[i] += B[pB1] * c[j];

}
```

Figure 1-12: Unscheduled COO SpMV: $a = Bc$ with a matrix in the coordinate format. Both i and j are stored for each component, so there is only one loop.

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < M; i++) {
  a[i] = 0.0;
}

#pragma omp parallel for schedule(dynamic, 1)
for (int i0 = 0; i0 < (M+31)/32; i0++) {
  for (int i1 = 0; i1 < 32; i1++) {
    int i = i0*32 + i1;
    if (i >= M) continue;

    for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
      int j = B2_crd[pB2];
      a[i] = B[pB2] * c[j];
    }
  }
}
```

Figure 1-13: Scheduled CSR CPU SpMV OpenMP C implementation. The schedule strip-mines and parallelizes the outer loop.

```
int ps = block*8192 + warp*512 + lane*16;
int i = search(B_pos, rows[block], rows[block+1], ps);
double w[16];

#pragma unroll
for (int k = 0; k < 16; k++) {
  int p = ps + k;
  if (p >= B_nnz) break;

  int j = B_crd[p];
  w[k] = B[p] * c[j];
}

double t = 0.0;
for (int k = 0; k < 16; k++) {
  int p = ps + k;
  if (p >= B_nnz) break;
  while (p == B_pos[i+1]) i++;

  t += w[k];
  if (p+1 == B_pos[i+1]) {
    atomicAdd(&a[i], t);
    t = 0.0;
  }
}
atomicAddWarp<double>(a, i, t);
```

Figure 1-14: Scheduled CSR GPU SpMV CUDA code. The schedule performs an index variable split with respect to the sparse matrix to load balance the computation. It then reorders the index variables so that consecutive threads load consecutive values. The schedule also uses the precompute transformation to store values into a temporary vector `w`. Finally, the schedule splits and parallelizes the index variables across GPU blocks, warps, and threads, using reduction strategies so that writes to the resulting vector `a` are synchronized. Chapter 6 describes these scheduling commands.

moved from the unscheduled CSR code from Figure 1-9; it merely parallelizes the outer loop and strip-mines it to expose more granular parallelism. This code performs on par with the Intel MKL library (see Figure 7-6). The GPU schedule does a lot to adapt the irregular SpMV computation to the exacting requirements of modern GPUs. To avoid thread divergence and load imbalance, it divides the nonzeros of the matrix into fixed-size blocks that are executed by different warps and threads. Thus, where the CPU code assigned each row to one thread, the GPU code assigns consecutive values to consecutive threads in a warp; hence, the GPU can coalesce loads. The result is a sophisticated GPU device kernel for a simple expression in a simple format. The kernel performs comparable to the SpMV implementation in the NVIDIA cuSPARSE library for large matrices, but is load balanced and therefore performs better on skewed matrices (see Figure 7-7).

As expressions, formats, and schedules get more complicated, it becomes difficult to write sparse tensor algebra code. Separate expression, format, and scheduling languages drastically reduce the complexity and let the compiler automate the sparse code generation. The end goal is, of course, to fully automate schedule and format generation.

But because automatic scheduling and automatic formatting systems will require a sparse code generator, a compiler like taco is their prerequisite. In the meantime, taco provides a far easier interface than writing irregular code by hand for CPUs and GPUs.

1.4 Contributions and Scope

In this dissertation, I present a comprehensive sparse iteration theory and its application to sparse tensor algebra compilation. I have used these ideas to build a compiler that can generate fused code, with performance comparable to handwritten code, for any sparse tensor algebra expression, on many data structures, with different optimizations, for CPUs and GPUS. My specific contributions are:

- **Hierarchical tensor formats** that describe sparse tensor storage as a composition of per-dimension data structures that store coordinates in a hierarchy.
- **An algebra of sparse iteration spaces** that let us reason about sparse iteration spaces and how to iterate over them.
- **A lattice formulation of general coiteration** that separates coiteration over the intersections and unions of multiple data structures into a sequence of coiterations over different regions of the sparse iteration space. This data structure coiteration omits operations that are structurally known to be zero.
- **Concrete index notation** that extend tensor index notation to describe the order of computation and where temporaries are stored.
- **Sparse optimizing transformations** that let us fuse, tile, and parallelize sparse iteration spaces, as we have long done for dense iteration spaces, and that lets us insert temporaries. The optimizing transformations can also tile a sparse iteration space with respect to an operand, producing load balanced sparse code.
- **A code generation algorithm** that lowers concrete index notation to imperative code that coiterates only over compressed tensor data structures to compute only those values that are required to produce the sparse tensor result.

These ideas were used in the design and implementation of taco. Chapter 7 shows that generated sparse code performs comparably to hand-optimized library implementations where those exist, but that the compiler approach generalizes to any expression. It also demonstrates the importance of compiling compound expressions to fused sparse code, the importance of support for many formats, and the importance of sparse scheduling transformations for performance and portability.

My dissertation puts sparse tensor algebra compilation on a firm foundation. It is the first step, however, and leaves several questions for future work (see also in Chapter 9). Specifically, it does not address the following topics:

- **Automatic format and schedule selection**, where an automated system determines the best formats and schedule for an expression on a given architecture. Instead, I present clean separate APIs to specify formats and schedules that can be used directly by a programmer or that can be the target for future automatic formatting and scheduling systems.
- **Matrix inversion and direct solvers**, where code is generated from expressions that include matrix inversions and direct solves.
- **Formats that interleave the storage of dimensions** beyond simple struct-of-array vs array-of-struct transformations. Examples of such formats include Morton and Hilbert space-filling curves, graph partitioning approaches, and quad-tree representations of tensors.
- **Symmetric formats**, where the data structure of a symmetric tensor stores only the subset of values required to reconstruct the tensor.
- **General tensor assembly**, where code is generated to assemble any data structures that appear on the left side of an index notation expression. This dissertation describes how to assemble tensors that are indexed by the free variables on the right side. It does not, however, cover general parallel assembly or the assembly of tensors the dimensions of which are not a direct match to the indices on the right hand side of the tensor algebra expression.
- **Distributed compilation**, where taco generates code to run on a distributed machine or a supercomputer. Together with collaborators, I have explored such compilation and believe that it is entirely possible with extensions to the abstractions I present.
- **Array operations beyond regular tensor algebra**, including any semiring, operations beyond semirings, and stencils.

But I stress that none of these topics is incompatible with the ideas in this dissertation. In fact, I think the ideas presented here are a prerequisite for exploring these topics. Finally, I believe that the sparse iteration space theory can be applied to computations beyond tensor algebra, such as general array operations, relational algebra, and graph computations. This dissertation, however, does not explore those extensions.

1.5 Dissertation Overview

The rest of this dissertation is organized as follows:

Chapter 2 - Data Structure Abstractions describes abstractions for sparse tensors and arrays as coordinate hierarchies that encode those positions that have values. Each level of a hierarchy encodes coordinates in one dimension, and the level can be represented by different data structures that are stacked.

Chapter 3 - Sparse Iteration Spaces shows how to form sparse iteration spaces from set expressions with coordinate hierarchies as operands. These iteration spaces can then be organized into iteration graphs and coiteration lattices to describe iteration over the sparse space.

Chapter 4 - Tensor Notations defines two languages for describing tensor algebra computations: a declarative tensor index notation and an operational concrete index notation.

Chapter 5 - Coiteration Code Generation describes an algorithm to lower concrete index notation to imperative code that coiterates over the iteration space of compressed tensor data structures.

Chapter 6 - Optimizing Transformations presents an optimization framework for transforming sparse iteration spaces to control the order of iteration, tiling, parallelization, and the introduction of temporary tensors into tensor computations. The transformations can also be used to target diverse hardware, such as CPUs and GPUs.

Chapter 7 - Evaluation evaluates the ideas of the previous chapters and shows that a compiler that implements them can generate code that performs comparable to hand-optimized code. It also demonstrates that a compiler must support compiling compound expressions to fused code, that different tensors benefit from different data structures, and that the optimization framework is important for performance and portability.

Chapter 8 - Related Work draws connections to prior work on libraries, programming systems, and compilers for sparse and dense tensor algebra.

Chapter 9 - Conclusion concludes with a discussion of some promising directions for future work.

"Abstraction is selective ignorance."
— Andrew Koenig

Chapter 2

Data Structure Abstractions

2.1 Coordinate Relations
2.2 Coordinate Trees
2.3 Level Abstraction
2.4 Six Level Types
2.5 Tensor Formats
2.6 Conclusion

Many data structures for storing sparse vectors, matrices, and tensors are described in the literature and used in practice. These data structures are often called formats and some examples are dense, COO [113], CSR [124], CSC, DCSR [34], ELL [110], DIA [112], BCSR, CSB [35], CSF [115], and hash maps.[8] Each is preferable in some situation and none is universally superior. The best format for an application depends on the pattern and sparsity of the tensor data, the computation, and the hardware. That is why it is important to support many formats in a sparse tensor algebra system.

Tensors can have any **order** (dimensionality), so it is intractable to enumerate all tensor formats. To support any-order tensors, it is necessary to construct formats from a bounded number of primitives. I describe how to define storage formats for tensors of any order by independently specifying the storage type of each **tensor mode** (dimension) and an ordering of the modes. This levelized formulation makes it possible to describe an unbounded number of tensor formats from a small number of composable building blocks. This leads to modular code generation that supports many formats of any order, as we will see in Chapter 5.

A tensor can be viewed abstractly as a coordinate relation with associated values which is concretely stored in one of many possible data structures. The data structure can be as simple as multidimensional dense arrays or as complex as irregular and hierarchical compressed data structures that store only nonzero coordinates. We will see how tensor coordinate relations can be thought of as coordinate trees and how to create types for those trees by independently typing and composing their levels. My goal in formalizing tensor storage as a type system is to make it possible to describe code generation and optimizing transformations independent of any specific data structure.

[8] I describe these formats in Section 2.5 and show how they can be built from the abstractions I outline in the rest of this chapter.

2.1 Coordinate Relations

Tensors can be viewed as relations over their coordinates and their values. A relation is a set whose elements are a subset of the Cartesian

product of other sets. A k-ary relation over k sets is a set of k-tuples whose elements each come from one of the k sets. I define two tensor relations, which I call "coordinate relation" and "component relation".

A k-order tensor has k modes, and I model each mode as a set of contiguous integers (a range). Tensor component locations are specified by k-tuples, where each element comes from a different mode. The tensor thus forms a Cartesian space where each mode is a dimension and the space is defined by the Cartesian product of the dimensions. The elements of the modes are coordinates of the space, and a k-tuple with one coordinate from each dimension is a point.

A **coordinate relation** is a relation over tensor modes. That is, its elements are a subset of points of the Cartesian space formed by the tensor modes. The relation may be complete and contain every point in that space, or it may contain a subset. It is often useful to store only the subset of points that correspond to nonzero tensor values, and to remove the coordinates that correspond to zeros.

Finally, a **component relation** is a binary relation over a coordinate relation and tensor values. It provides a unique mapping and every coordinate maps to exactly one value. Coordinate and component relations are abstractions for tensors that I use to describe sparse iteration spaces. In Section 2.2, I show how to specify them hierarchically as per-dimension storage trees that can be composed to describe many popular tensor storage formats and many other formats that have not been explored to date.

2.2 Coordinate Trees

A tensor and its coordinate relation can be viewed as a **coordinate tree**.[9] The root of the tree represents the tensor itself, internal nodes represent the coordinates, and the leaves represent the tensor component values. A path from the root to a leaf fully specifies a tensor component with coordinates and a value. A tree level consists of all the coordinates in a tensor mode, divided into segments of siblings that share a parent. Figure 2-1 shows that a level can have duplicates. The duplicates are clones of the coordinate that occur at multiple positions in a tree level, and the children of every duplicate are siblings. The figure also shows an unlabeled node, which is a node that takes space in the tree but that does not represent anything. Figure 2-2 shows a dense 3×3 matrix and the corresponding coordinate tree, where a path, a level, and a segment are marked in blue.

The coordinate tree representation is a convenient tool for reasoning about sparse tensor representations by removing from the tree zero-valued leaf nodes and coordinates with no children. Figure 2-3 shows a sparse 3×3 matrix and the corresponding coordinate tree. In contrast to the coordinate tree for the dense matrix in Figure 2-2, this tree is not full, as zero leaves and childless coordinates have been left out. By removing zeros from the coordinate tree, we have made

[9] The intuition that sparse tensors can be viewed as coordinate trees comes from the work of Smith and Karypis, who use it to describe the Compressed Sparse Fiber (CSF) tensor format [115].

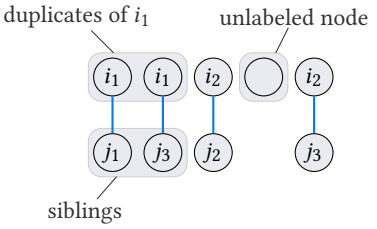


Figure 2-1: A coordinate tree with duplicates. Children of duplicated nodes are siblings.

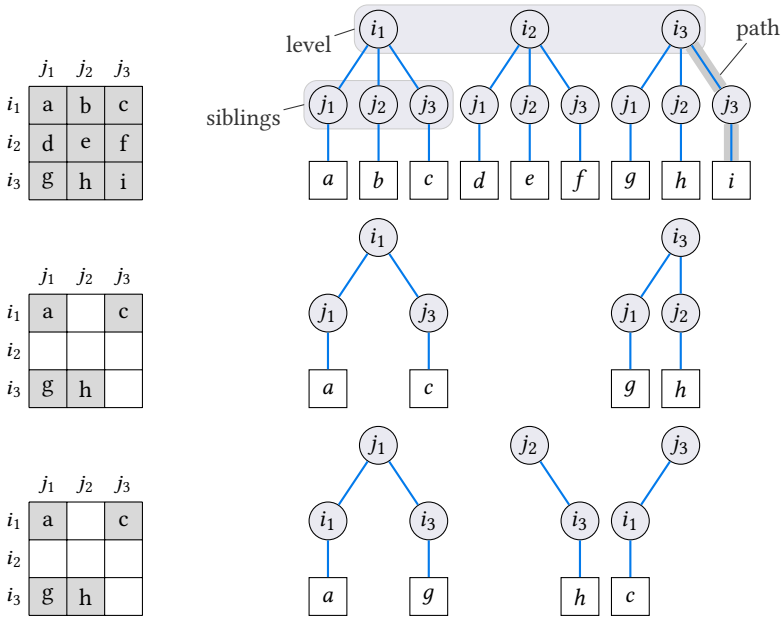


Figure 2-2: A dense 3×3 matrix and its full coordinate tree. A path, a tree level, and a set of siblings are marked.

Figure 2-3: A sparse matrix and its row-major coordinate tree. Zeros in the matrix are left out of the tree together with childless coordinates.

Figure 2-4: The same sparse matrix and its column-major coordinate trees. Tensor modes can be stored in any order by re-ordering tree levels.

the first step toward reasoning about sparse tensors in terms of data structures that compress out zeros. Finally, we can interchange tree levels to represent tensors whose modes are stored in a different order, such as a column-major matrix instead of a row-major matrix. Figure 2-4 shows a coordinate tree for the same matrix as Figure 2-3, but the tree levels have been interchanged to store the matrix in column-major order.

2.3 Level Abstraction

The coordinate tree abstraction must be represented somehow by values in memory. This representation may be regular or irregular. I define regular representations as representations that require at most a number of values proportional to the number of coordinate tree **levels**. Examples include a dense representation that encodes a full coordinate tree and a grid stencil that requires a constant number of values to encode the stencil offsets. Irregular representations, on the other hand, require a number of values proportional to the number of coordinate tree leaves. Examples include hash maps and segmented vectors.[10]

A key insight of our work is that coordinate trees can be described by composing separately defined data structures for each tree level into a hierarchy. The data structure of an internal tree level is an ordered set of maps, with one map for each segment of siblings (i.e., one map per node on the previous level). See Figure 2-5. The maps connect each coordinate to the position of its sibling segment in the next level. Finally, the tree leaves are represented by an ordered set of component values. Any physical data structure that can represent an ordered set of maps can be used. Furthermore, the maps need only encode coordinate-position pairs; they do not, for example, need to

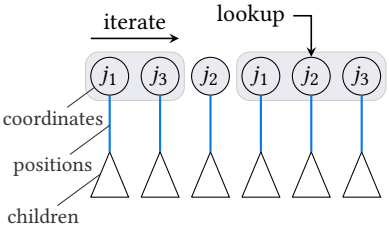


Figure 2-5: A coordinate tree level is an ordered set of maps from coordinates to the position of their children on the next level. Levels may support functionality such as traversals and looking up a position given a coordinate.

[10] A segmented vector is an ordered vector of ordered segments. Segmented vectors have been used in the CSR matrix format since 1967 [124]. Guy Blelloch describes a segmented vector as an independent data structure in his dissertation [29].

support $O(1)$ mapping from coordinates to positions.

Level storage can be described by an abstraction that defines the functionality and properties of a level without tying it to a specific underlying data structure. This abstraction can then be instantiated at compile time into many level types with different functionality and underlying physical storage. Since the level types are an abstraction inside the code generator, they do not appear in the generated code and thus have no performance cost.[11] The benefit of the level abstraction, as we will see in the next chapter, is that it separates the code generator from specific data structures. This design makes it possible to design a general code generator in terms of the abstraction's functionality and properties. The code generator does not need to know what level types exist. New level types can thus be designed without any need to change the code generator algorithms.

[11] I believe meta-abstractions will become more important in the future due to the cost of abstractions in the execution code. One goal of this dissertation is to provide *abstraction without friction* by moving abstractions to the code generator.

Level types must declare their *properties*, and they may support any one of five *capabilities* that will be used to generate code to iterate over and modify levels. Properties provide information that the code generator uses to optimize the generated code, such as whether a level is ordered. Capabilities, on the other hand, are abstractions that provide code to the code generator for iterating over and manipulating physical indices of tensor storage in a format-agnostic manner.

Level Properties

A coordinate tree **level property** describes a feature of a level type that code generators can use to optimize iteration code. We define five properties: full, ordered, unique, branchless, and compact. Each property describes an attribute of a level, such as whether coordinates are arranged in increasing order or not. The rows of a sorted CSR matrix, for example, are both ordered and unique, meaning they store every column coordinate just once and in increasing order. The code-generation technique relies on level properties to produce optimized code.

Full A 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 dense vector stores a component at every coordinate and is therefore full. By contrast, a sparse vector that stores only the nonzero coordinates is not full. Figure 2-6 shows a level that is full, assuming the full set of nodes is j_1, j_2, and j_3, and a level that is not full because it is lacking the j_1 node.

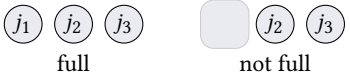


Figure 2-6: A *full* level and one that is not because it is lacking j_1.

Ordered A level is ordered if all coordinates that share any ancestor are arranged in a monotonically non-decreasing sequence. (Recall that two nodes that share duplicates of an ancestor are considered to share that ancestor.) Figure 2-7 shows a level that is ordered, from nodes

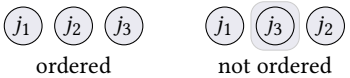


Figure 2-7: An *ordered* level and one that is not because j_3 comes before j_2.

with smaller indices to nodes with larger indices, and a level that is not ordered because j_3 occurs before j_2.

Unique A level is unique if no identical paths end at the level. Two paths are identical if the nodes at every step along both paths are the same nodes. Figure 2-8 shows a level that is unique and one that is not because paths i_1-j_1 (blue) and i_1-j_3 (green) appear twice. Note that it does not matter if both nodes along the path are duplicates (the blue path) or if one node is shared and the other a duplicate (the green path).

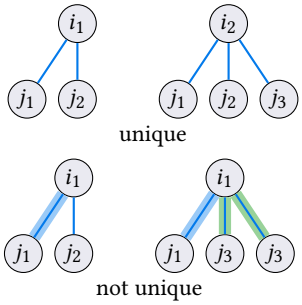


Figure 2-8: A *unique* level and one that is not because two paths appear twice.

Branchless A level is branchless if every node at the parent level, including each duplicate, has exactly one child. Knowledge of such one-to-one relationships between levels can be used to efficiently collapse their corresponding loops. Figure 2-9 shows one level that is branchless and one that is not because i_2 has more than one child.

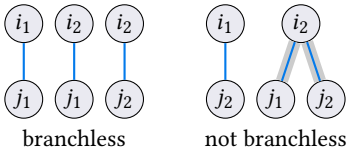


Figure 2-9: A *branchless* level and one that is not because i_2 has two children.

Compact A level is compact if no two coordinates are separated by an unlabeled node. As we will see in Section 2.5, some physical storage schemes, such as hash maps, have unlabeled nodes that represent empty buckets.

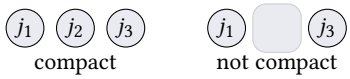


Figure 2-10: A *compact* level and one that is not because an unlabeled node appears between two nodes.

Level Capabilities

A coordinate tree **level capability** is exposed as a set of coordinate tree **level functions** with fixed interfaces that a level type may choose to implement. These function interfaces are designed to return code that implements the function. The code generator then emits this code as part of the function that implements an expression. We define five capabilities: coordinate iterate, position iterate, locate, append, and insert.

Coordinate Iterate The coordinate iteration capability iterates over coordinates and retrieves positions. It consists of two level functions: one that returns an iterator over coordinates (`coord_bounds`) and one that accesses the position of each coordinate (`coord_access`):

```
coord_bounds(i1, ..., ik−1) -> <ibeginₖ, iendₖ>
coord_access(pk−1, i1, ..., ik) -> <pₖ, found>
```

Given a list of ancestor coordinates $(i_1, \ldots, i_{k-1})$, `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 p_{k-1} that encodes i_k and returns `found` as true, or it returns `found` as false if the coordinate does not actually exist. In practice, the emitted code can often be optimized by removing

the `if` statement, because the value of `coord_access` is static in many level types.

Position Iterate The position iterate capability iterates over positions and retrieves coordinates. It consists of two level functions: one that returns an iterator over positions (`pos_bounds`) and one that accesses the coordinate encoded at each position (`pos_access`):

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

Given a coordinate at position p_{k-1}, `pos_bounds` returns the bounds of an iterator over positions that may have p_{k-1} as their parent. For each position p_k in those bounds, `pos_access` either returns the coordinate encoded at that position, or sets `found` to false if p_k is not a child of p_{k-1} or if it does not encode that coordinate (i.e., if p_k is unlabeled).

Locate The locate capability retrieves a coordinates position in a level.

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

`locate` has similar semantics as `coord_access`. Given a coordinate i_{k-1} at position p_{k-1}, `locate` attempts to locate among its children the coordinate i_k. If `locate` finds i_k, then it returns i_k's position p_k and returns `found` as true; otherwise it returns `found` as false. Traversing a path in a coordinate hierarchy to access a single tensor component can be done by successively calling `locate` at every level. As we will see in Chapter 3, having operands that implements the locate capability can lead to code that avoids accessing every nonzero.

Append Appends coordinates to a level and is also exposed as four level functions:

```
append_coord(p_k, i_k) -> void
append_edges(p_{k-1}, pbegin_k, pend_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 an output level (p_k). The function `append_edges` appends edges that connect all coordinates between positions $\texttt{pbegin}_k$ and $\texttt{pend}_k$ to the coordinate at position p_{k-1} in the previous level. By appending edges, inserted coordinates are attached to the rest of the coordinate hierarchy. In contrast to the insert capability, the append capability requires result coordinates to be appended in order. `append_init` and `append_finalize` serve identical purposes as `insert_init` and `insert_finalize`, and they take the same arguments.

Insert Inserts coordinates at any position and is exposed as four level functions:

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

Table 2-11: The properties of the six level types. A (✓) means that level type can be configured to either have or not have that property.

Level Types	Iterate Type	Locate	Assembly Type
Dense	✗	✓	Insert
Compressed	Position Iterate	✗	Append
Singleton	Position Iterate	✗	Append
Hashed	Position Iterate	✓	Insert
Range	Coordinate Iterate	✗	✗
Offset	Position Iterate	✗	✗

Table 2-12: The capabilities of the six level types. Dense does not support any iterate capability while range and offset do not support assembly.

```
insert_coord(p_k, i_k) -> void
size(sz_{k-1}) -> sz_k

insert_init(sz_{k-1}, sz_k) -> void
insert_finalize(sz_{k-1}, sz_k) -> void
```

The level function `insert_coord` inserts a coordinate i_k into an output level at position p_k given by `locate`. It requires the level to provide the locate capability. The level function `insert_init` initializes the data structures that encode an output level, while `insert_finalize` performs any post-processing required after all coordinates have been inserted. Both take, as inputs, the sizes of the level being initialized or finalized (sz_k) and its parent level (sz_{k-1}). For a level that provides an insert capability, its size is computed as a function of its parent's size by the level function `size`. For a level that supports append, its size is the number of coordinates that have been appended.

2.4 Six Level Types

In this section, I show how common variants of all the tensor formats examined in Section 2.5 can be expressed as compositions of just six level types that we have designed: dense, compressed, single, range, offset, and hashed.[12] Recall that a level type encodes the nodes in a level along with their grouping into sibling sets. Some level formats implicitly encode coordinates (e.g., as an interval), while others explicitly store them (e.g., in a segmented vector).

The properties of the six level types are shown in Table 2-11, and their capabilities are in Table 2-12. The properties of some level types can be configured depending on the application. Configurable properties reflect optional invariants that are not required by the way a physical index encodes coordinates. For example, the `crd` array in compressed levels typically stores coordinates in order when used in the CSR format, but the same data structure can also store coordinates

[12] These level types are just six of many possible level types that can be designed for the level abstraction. I expect that new and surprising level types will be designed in the future.

out of order. Declaring the ordered property, however, helps the code generator generate efficient code.

The six level types, their data structures, and their capabilities are described below. The source code returned by their iterate and locate level functions are listed in Table 2-13, while the source codes returned by their assembly functions are listed in Table 2-14. Most levels support position iterators, except the range iterator, which supports a coordinate iterator. The dense level type does not support iteration, as it is a full level with fast locate and can, therefore, be accessed by an iterator that iterates over an entire dimension.

Dense Level Type

Dense levels store a single number (`size`) that encodes every coordinate in the range `0:size` (inclusive-exclusive). Dense levels support very efficient `locate` and `insert` capabilities. They do not, however, support any iterate capabilities. As we shall see in Chapter 3, to iterate over an entire dense level we must iterate over all the coordinates in the full coordinate set and call locate to get positions in the dense level.

Figure 2-15: Dense data structure. Only one number is needed to encode every coordinate in the range `0:size` for each segment.

Compressed Level Type

Compressed levels store coordinates in a segmented vector consisting of a coordinate array (`crd`) and a position array (`pos`) which store segment bounds (Figure 2-16). Each sibling set is stored in one segment, so the children of the kth coordinate on the previous level are stored in the kth segment. They are the coordinates in range `pos[k]:pos[k+1]` in the `crd` array. Compressed levels support the `position iterate` and `append` capabilities. Compressed levels do not support $O(1)$ locate. Since they are ordered, however, it is possible to generate code to locate the position of a coordinate by emitting code to binary search the positions in $O(\log n)$ time.

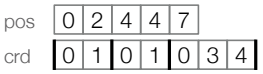


Figure 2-16: Compressed data structure. Each segment in the `crd` array is described by the range `pos[k]:pos[k+1]` and depicted with thick separators.

Singleton Level Type

Singleton levels store a single coordinate with no siblings for each node on the previous level in a coordinate array (`crd`). They are often stacked underneath compressed levels configured to have duplicates (not unique) to represent coordinates, as we shall see in Section 2.5.

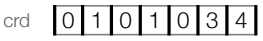


Figure 2-17: Singleton data structure. Each segment stores a single coordinate.

Hashed Level Type

Hashed levels store the coordinates of each segment in a hash map (`crd`). Figure 2-18 shows a hashed level that encodes a row vector,

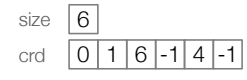


Figure 2-18: Hashed data structure. Each segment is stored as a hash map.

Table 2-13: Definitions of level functions of the six level types that implement access capabilities. All level types except dense levels provide coordinate or position iterators. In addition, dense and hash maps provide $O(1)$ locate functions.

Level Types	Level Function Definitions	
Dense	`locate(p`$_{k-1}$`, i`$_1$`, ..., i`$_k$`):` `  return <p`$_{k-1}$` * N`$_k$` + i`$_k$`, true>`	
Compressed	`pos_bounds(p`$_{k-1}$`):` `  return <pos[p`$_{k-1}$`], pos[p`$_{k-1}$` + 1]>`	`pos_access(p`$_k$`, i`$_1$`, ..., i`$_{k-1}$`):` `  return <crd[p`$_k$`], true>`
Singleton	`pos_bounds(p`$_{k-1}$`):` `  return <p`$_{k-1}$`, p`$_{k-1}$` + 1>`	`pos_access(p`$_k$`, i`$_1$`, ..., i`$_{k-1}$`):` `  return <crd[p`$_k$`], true>`

Range	`coord_bounds(i`$_1$`, ..., i`$_{k-1}$`):`
	`  return <max(0, -offset[i`$_{k-1}$`]),`
	`  min(N`$_k$`, M`$_k$` - offset[i`$_{k-1}$`])>`
		`coord_access(p`$_{k-1}$`, i`$_1$`, ..., i`$_k$`):`
		`  return <p`$_{k-1}$` * N`$_k$` + i`$_k$`, true>`

Offset	`pos_bounds(p`$_{k-1}$`):` `  return <p`$_{k-1}$`, p`$_{k-1}$` + 1>`	`pos_access(p`$_k$`, i`$_1$`, ..., i`$_{k-1}$`):` `  return <i`$_{k-1}$` + offset[i`$_{k-2}$`], true>`
Hashed	`pos_bounds(p`$_{k-1}$`):` `  return <p`$_{k-1}$` * W`$_k$`, (p`$_{k-1}$` + 1) * W`$_k$`>`	`pos_access(p`$_k$`, i`$_1$`, ..., i`$_{k-1}$`):` `  return <crd[p`$_k$`], crd[p`$_k$`] != -1>`

```
locate(p_{k-1}, i_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>
```

Table 2-14: Definitions of level functions of the six level types that implement assembly capabilities.

Level Types	Level Function Definitions	

Dense	`insert_coord(p_k, i_k):`
	`  // do nothing`
	
	`size(sz_{k-1}):`
	`  return sz_{k-1} * N_k`
		`insert_init(sz_{k-1}, sz_k):`
		`  // do nothing`
		
		`insert_finalize(sz_{k-1}, sz_k):`
		`  // do nothing`

Compressed	`append_coord(p_k, i_k):`
	`  crd[p_k] = i_k`
	
	`append_edges(p_{k-1}, pbegin_k, pend_k):`
	`  pos[p_{k-1} + 1] = pend_k - pbegin_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}, pbegin_k, pend_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`
	
	`size(sz_{k-1}):`
	`  return sz_{k-1} * W_k`
		`insert_init(sz_{k-1}, sz_k):`
		`  for (int p_k = 0; p_k < sz_k; ++p_k) {`
		`  crd[p_k] = -1`
		`  }`
		
		`insert_finalize(sz_{k-1}, sz_k):`
		`  // do nothing`

with empty buckets are marked by −1. Table 2-13 shows the level's hash function.

Range Level Type

Range levels encode coordinates in an interval with bounds computed from an offset array and dimension sizes N and M. Figure 2-19 shows the row dimension of a DIA matrix encoded as a range level with arrays. Given a parent coordinate 1, the level encodes coordinates between $\max(0, -\texttt{offset[1]}) = 1$ and $\min(4, 6 - \texttt{offset[1]}) = 4$.

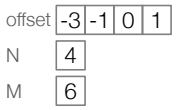


Figure 2-19: Range data structure. Each segment is stored as range offsets and sizes.

Offset Level Type

Offset levels encode for each parent a single child coordinate with no siblings. Each child is shifted from its parent coordinate by a value in the `offset` array. Figure 2-20 shows the column dimension of a DIA matrix encoded as an offset level. Given a parent coordinate 3 and an offset index 1, for instance, the level encodes the coordinate $(3 + \texttt{offset[1]} = 2)$.

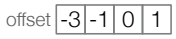


Figure 2-20: Offset data structure. Each segment is stored as offsets.

2.5 Tensor Formats

The six level types can be composed hierarchically to form tensor formats by assigning a level type to each tensor mode. Furthermore, an ordering of the levels is needed to determine the order in which the tensor modes will be stored. For example, storing matrix rows ahead of columns yields a row-major matrix, while storing columns ahead of rows yields a column-major matrix.

In this section I show how many vector, matrix, and tensor formats from the literature can be recreated by composing the six level types defined in Section 2.4. These level types, however, can be combined in an unbounded number of ways to express additional tensor formats that have not been explored yet. For instance, a variant of DIA for matrices that have only sparsely filled diagonals can be expressed as the combination (dense, compressed, offset) which replaces the range level that implicitly assumes diagonals are densely filled. Finally, I cast structured tensor formats, like the BCSR matrix format, as formats for higher-order tensors where the added dimensions expose sub-structures.

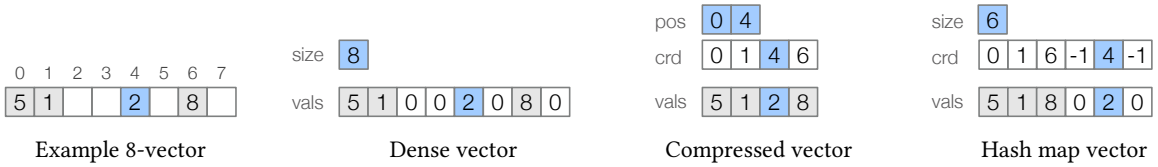


Figure 2-21: Vectors stored in data structures described by different formats. Elements shaded in blue show the coordinates and values corresponding to a single nonzero.

Figure 2-21 shows an 8-vector and three different data structures for storing its values: a dense vector, a compressed vector, and a hash map vector. These data structures are described as the dense, compressed, and hashed level types. Since these data structures have different capabilities, they work well in different circumstances. The dense vector has no meta-data apart from the dimension size. It supports fast iteration, random access, and insert, but it does not compress out zeros. The compressed vector must explicitly store the coordinates and the vector size (in `pos[1]`), but it compresses out zeros. Finally, the hash map compresses out zeros, and it supports random access and insert. But it is less efficient to iterate over it, and it stores some locations that are not used, shown as -1 in the coordinate (`crd`) array.

A straightforward way to store a k-order tensor is to use an k-dimensional dense array that explicitly stores all tensor components including zeros. A desirable feature of dense arrays is that the value at any coordinate can be accessed in constant time. Storing a sparse tensor in a dense array is inefficient, however, because 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 with many large dimensions, it may even be impossible to use a dense array due to a lack of memory.

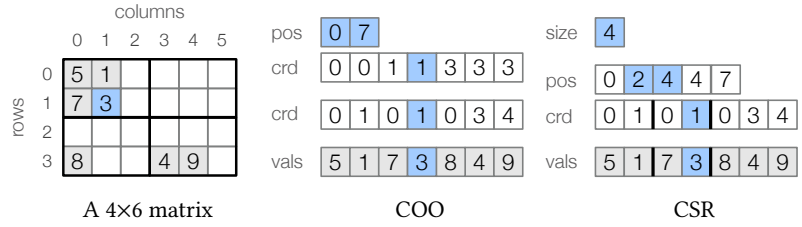


Figure 2-22: A 4×6 matrix stored in row-major unstructured data structures described by the Coordinate (COO) and Compressed Sparse Row (CSR) matrix formats. Elements shaded in blue show the coordinates and values corresponding to a single nonzero.

Figure 2-22 shows a 4×6 matrix and seven row-major data structures for storing its values. The simplest way to store a sparse matrix efficiently is to keep a list of all its nonzero coordinates and values. This data structure is known as the **Coordinate** format (COO) [113] and, in contrast to dense arrays, consumes only $\Theta(\text{nnz})$ memory. In addition, many common file formats for storing tensors, such as the Matrix Market exchange format [97] and the FROSTT sparse tensor format [117], closely mirror the COO format. Storing tensors in a file as a coordinate list minimizes the cost of file reads and writes, as inserting a coordinate and its nonzero value only requires appending them to the `crd` and `vals` arrays.

The **Compressed Sparse Rows** matrix format (CSR) stores every row compressed. This reduces memory footprint for many matrices over the COO format because it does not store redundant row coordinates. Figure 2-23 shows its level descriptor with a dense level type representing the rows (every row is stored) followed by a compressed level type representing the columns (each row is compressed). The CSR matrix in Figure 2-22 removes the duplicate row coordinates on the last row of the COO matrix. The auxiliary array (`pos`) keeps track

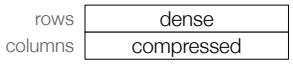


Figure 2-23: Compressed Sparse Rows (CSR) format descriptor.

of which nonzeros belong to each row.

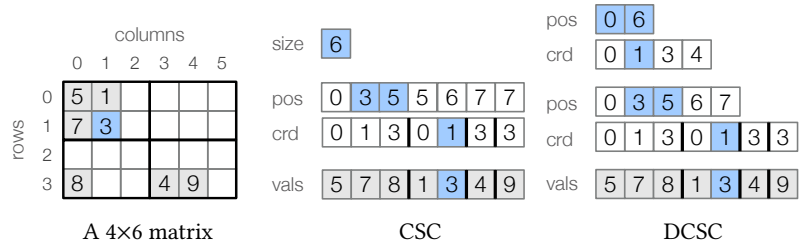


Figure 2-24: A 4×6 matrix stored in column-major unstructured data structures described by the Compressed Sparse Column (CSC) and Doubly Compressed Sparse Column (DCSC) matrix formats.

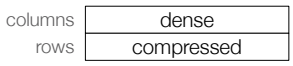


Figure 2-25: Compressed Sparse Columns (CSC) format descriptor.

The **Compressed Sparse Columns** format (CSC) is the column-major version of CSR. CSC is popular in linear solvers [46] because it stores every column compressed. Figure 2-24 shows the same 4×6 matrix with the column-major CSC data structures for storing its values. Figure 2-25 shows its level descriptor with a dense level above a compressed level. The ordering of the modes corresponding to these levels is the opposite of CSR, however, with the top dense level corresponding to the columns and the bottom compressed level corresponding to the rows.

The **Doubly Compressed Sparse Columns** format (DCSC) [34], also shown in Figure 2-24, achieves additional compression for hypersparse matrices by storing only the columns that contain nonzeros. In the example, the `pos` array of the DCSC format has fewer values than the CSC format. When there are many empty rows, it uses less memory, even though it stores additional column `pos` and `crd` arrays. In addition, the **Doubly Compressed Sparse Rows** format (DCSR) compresses the CSR format by storing only rows that have a nonzero.

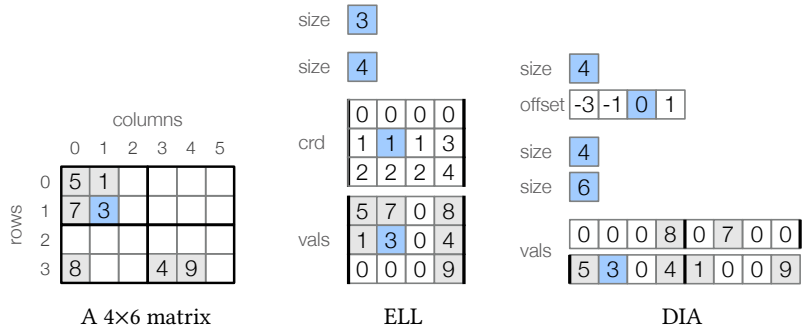


Figure 2-26: A 4×6 matrix stored in structured matrix data structures described by the ELLPACK (ELL) and diagonal (DIA) matrix formats.

Many important applications have matrices whose nonzero components have structure. Figure 2-26 shows two data structures that take advantage of this structure to further compress the representation. Matrices that encode vertex-edge connectivity of well-formed unstructured meshes, for instance, have a bounded number of nonzero components per row. The bound on nonzeros per row is exploited by the **ELLPACK** format (ELL), which stores the same number of components for each row [110]. Thus, it has to store only the column coordinates and nonzero values in the implicitly indexed row positions. The column coordinates and nonzero values are stored contiguously in memory, making it possible to efficiently vectorize the SpMV op-

eration [11]. If nonzeros are further restricted to a few dense diagonals, their coordinates can be computed from the offsets of the diagonals. This pattern is common in grid and image applications. It lets the Diagonal format (**DIA**) forgo storing the column coordinates altogether [112]. However, for matrices that do not conform to assumed structures, structured tensor formats may needlessly store many zeros and actually degrade performance.

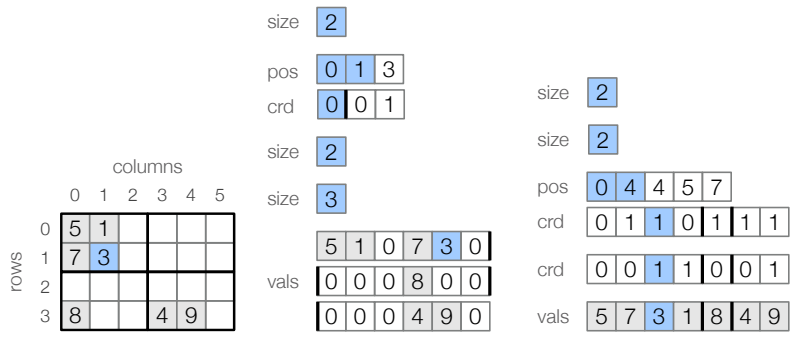


Figure 2-27: A 4×6 matrix stored in blocked matrix data structures described by the Block Compressed Sparse Row (BCSR) and Compressed Sparse Block (CSB) matrix formats.

Moreover, many applications have matrices with sub-structures that can be taken advantage of by storing them as blocked matrices. Figure 2-27 shows the data structures of two blocked formats. The **Block Compressed Sparse Rows** format (BCSR) [67] generalizes CSR by storing a dense block of nonzeros in the `vals` array for every nonzero coordinate. Its cousin, the **Blocked Compressed Sparse Columns** format (BCSC), generalizes CSC the same way. Since the BCSR and BCSC formats reduce storage and expose opportunities for vectorization, they are ideal for the inherently blocked matrices from FEM applications. By contrast, the **Compressed Sparse Blocks** (CSB) format, proposed by Buluç et al. [35], represents a matrix as a dense collection of sparse blocks stored in the COO format.

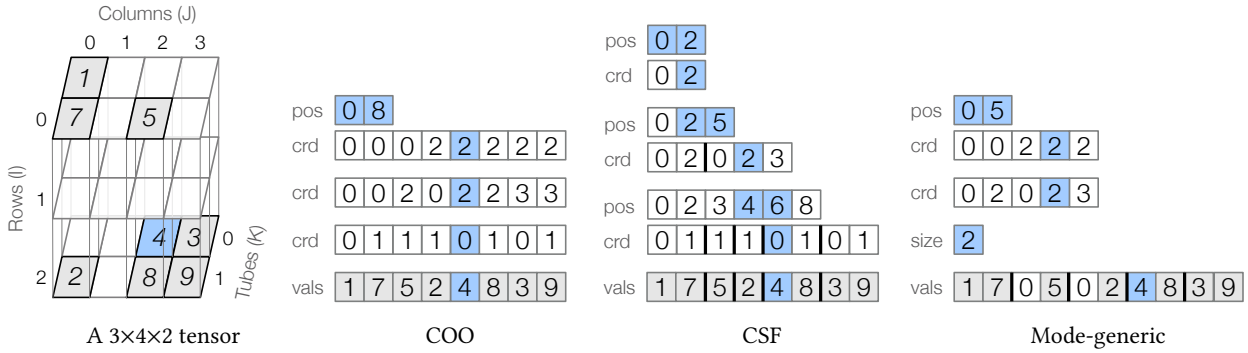


Figure 2-28: Tensors stored in data structures described by different formats. Elements shaded in blue show the coordinates and values corresponding to a single nonzero.

All the data structures we have described so far, including those described by unstructured, structured, and blocked formats, can be generalized to tensors of any order. Figure 2-28 shows a 3×4×2 tensor and three hierarchical data structures that store it. These data structures are generalizations of the matrix data structures above, and countless more generalizations are possible for tensors of any order. The

Coordinate format (COO) trivially generalizes to higher-order tensors by adding another coordinate array for every new mode. The **Compressed Sparse Fibers** (CSF) tensor format, designed by Smith and Karypis [115], is a generalization of CSR that compresses every dimension (Figure 2-28). Tensors stored in any of these compressed formats, however, are costly to assemble or modify. Finally, the mode-generic sparse tensor format, proposed by Baskaran et al. [20], generalizes the idea of BCSR to higher-order tensors. It stores a tensor as a sparse collection of any-order dense blocks, with the coordinates of the blocks stored in COO (i.e., the `crd` arrays).

2.6 Conclusion

In this chapter, I showed how to compose tensor formats from a hierarchy of per-mode data structures. I described coordinate trees and coordinate level abstractions that hide the details of each data structure behind a compiler abstraction. These abstractions are essential building blocks for a tensor algebra compiler because they enable modular reasoning, optimization, and code generation. The per-mode level is the smallest tensor building block, and from it, I formulate in Chapter 5 a simple recursive code generation algorithm that coiterates over tensors by coiterating over levels.

The coordinate trees and levels facilitate modular reasoning about tensors in both compilers and computer architectures. And they can generalize beyond tensors to any mathematical abstraction that is a sparse subset of a Cartesian coordinate space, such as database relations and graphs. In the next chapter, I show how to reason about sparse iteration spaces composed of coordinate relations represented as coordinate trees. I also show how to iterate over the spaces efficiently by coiterating over the trees. And Chapter 5 shows how to compile tensor algebra expressions to imperative code that coiterates over the physical data structures underneath coordinate trees. Finally, Section 7.3 of the Evaluation chapter provides empirical evidence for why a sparse tensor algebra compiler needs to support many different formats.

"Solving a problem simply means representing it so as to make the solution transparent."

— Herbert Simon

Chapter 3

Sparse Iteration Spaces

3.1 Iteration Space Algebra

3.2 Iteration Graphs

3.3 Iteration Lattices

3.4 Conclusion

We can describe the iteration space of loops that iterate over dense tensors as a hyperrectangular grid of points by taking the Cartesian product of the iteration domain of each loop. The polyhedral model takes this view for general affine loop nests [70, 85], and recent work has specialized it for dense tensor computations [9, 127]. Figure 3-1 shows the iteration space for iterating over a dense matrix. Because the iteration space is dense, the grid contains every point.

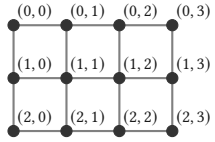


Figure 3-1: Dense iteration space, with all points present.

A **sparse iteration space** is a grid with holes, as shown in Figure 3-2. The holes are locations in the grid where points are missing and therefore should not be visited when iterating over the space. In sparse tensor algebra, the holes come from tensor components whose values are zero, and we can avoid iterating over them because $0+0=0$ and $a \cdot 0 = 0$. Figure 3-3 demonstrates how the property that $0+0=0$ results in sparse iteration in a sparse matrix addition. The locations in the addition iteration space in Figure 3-2—the union of the operand iteration spaces—where the resulting values are zero (colored white) are not visited. Sparse iteration spaces can be described by coordinate relations represented as coordinate trees or by combining relations using an iteration space algebra.

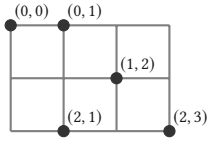


Figure 3-2: Sparse iteration space, with some points missing.

In this chapter, I describe an algebra for multi-dimensional sparse iteration spaces. I also describe an intermediate representation, called iteration graphs, that describes how to iterate through them. These graphs separate the iteration space into per-dimension iteration spaces, capture the dependencies between dimensions that come from iterating over coordinate trees, and describe the iteration through each dimension as a set expression of coordinate tree levels. Finally, I describe how to efficiently coiterate over coordinate tree levels by dividing the set expressions into ordered iteration lattices of subsets that can, in turn, be iterated over.

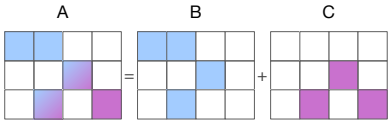


Figure 3-3: A sparse matrix addition, where zeros are white and nonzeros are colored by the matrix it comes from.

This chapter provides the theory and representations we will need to express, compile, and optimize sparse tensor algebra expressions. Chapter 4 will extend the two sparse iteration representations—iteration space algebra and iteration graphs—with arithmetic assignment statements to form two tensor languages called tensor algebra index no-

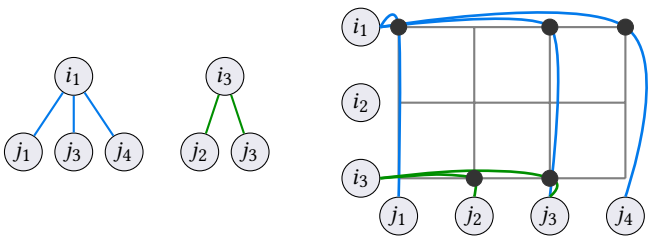


Figure 3-4: A coordinate tree and the corresponding sparse iteration space. The tree paths from the i_1 and i_3 nodes are shown both in the tree and overlaid on the sparse iteration space.

tation and concrete index notation. Chapter 5 shows how coordinate trees, iteration graphs, and iteration lattices can be used to generate efficient code to iterate over the sparse multi-dimensional iteration spaces. Finally, Chapter 6 presents several transformations on sparse iteration spaces that let us control the order of iteration, so that we can optimize the generated code.

3.1 Iteration Space Algebra

The simplest sparse iteration space is described by a coordinate relation. The full set of nodes at each internal tree level is a dimension of the space. Further, the nodes along each tree path, from the root to the leaves, are the coordinates of one point in the space; every other point is left out. By iterating over the coordinate tree, we iterate over the points of the sparse iteration space. Figure 3-4 shows a coordinate tree and its sparse iteration space. The coordinates are arranged along the dimensions of the space without duplicates, and tree paths are shown as graph edges. The sparse iteration space has a point if a tree path connects the point coordinates.

We can combine sparse iteration spaces by intersecting and unioning their points.[13] Let us define a sparse iteration space algebra by introducing index variables into set expressions where the variables are coordinate relations, represented by coordinate trees. The index variable index into the coordinate relations and thus controls what coordinates are compared in the set operations. For example, $B_{ij} \cup C_{ij}$ and $B_{ijk} \cap C_k$. The points in the variable sets are ordered k-tuples of coordinates, and index variables index into tuple locations determined by their position in the index lists. That is, j in B_{ijk} index the coordinates of the second dimension of coordinate relation B. The number of dimensions of the iteration space is determined by the total number of index variables in the expression. And the size of each dimension is the size of the coordinate sets each index variable ranges over. Coordinate sets indexed by the same, repeated, index variable are combined using the set operations that combine the coordinate relation variables they index. For example, the k coordinates of the third dimension of B and the second dimension of C are combined in $B_{ijk} \cap C_{jk}$.

One class of expressions index into every coordinate relation with

[13] The expressions in this chapter use only intersection and union operations; however, I believe that the concepts can be readily extended to other set operations.

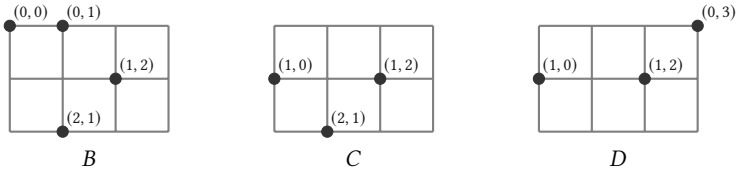


Figure 3-5: Three examples of two-dimensional sparse iteration space.

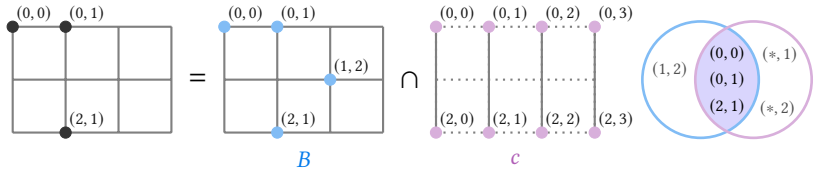


Figure 3-8: The sparse broadcast iteration space resulting from the expression $B_{ij} \cap c_i$, including the coordinate Venn diagram. The iteration space of c is broadcast over the coordinate universe of j, shown as dotted lines.

every index variable. We call these point-wise expressions, and two examples are $B_{ij} \cap C_{ij}$ and $B_{ij} \cap C_{ji}$. In the first expression, coordinates of B and C are intersected by comparing the first coordinates and the second coordinates of B and C with each other. The second expression, on the other hand, compares the first coordinate of B with the second of C and vice versa. Figure 3-5 shows three sparse iteration spaces and Figure 3-6 shows the sparse iteration spaces resulting from applying the two binary two-dimensional point-wise operations $B_{ij} \cap C_{ij}$ and $B_{ij} \cup C_{ij}$. The latter figure also shows coordinate Venn diagrams that illustrate the operations. Figure 3-7 shows a tertiary point-wise operation where the union of two iteration spaces is intersected by a third. We can form arbitrarily complicated operations by combining any number of iteration spaces this way.

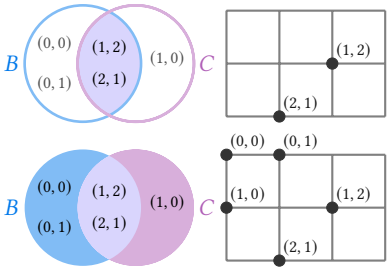


Figure 3-6: The sparse iteration spaces resulting from the intersection and union of B and C from Figure 3-5.

The other class of sparse iteration space expressions contain index variables that do not index into all the coordinate relation variables. I refer to these expressions as broadcast expressions, and examples include $B_{ijk} \cap c_k$, $B_{ij} \cup c_j$, $(B_{ij} \cap c_j) \cup d_i$, and $B_{ik} \cap C_{kj}$.[14] In these expressions, the combined iteration space has more dimensions than the number of coordinate sets in some or all of the coordinate relations. In fact, in the last example, it has more dimensions than *any* coordinate relation. As stated, the iteration space has as many dimensions as there are index variables. Point sets not indexed by an index variable are broadcast over that space, meaning that the lower-dimensional iteration space of the coordinate relation is replicated for every point in the missing dimensions. In set language, the iteration space of a coordinate relation is the flattened Cartesian product of its points and the sets the missing index variables range over. Figure 3-8 shows the sparse iteration space of the expression $B_{ij} \cap c_i$. The iteration space of c is one-dimensional; however, in the expression, it is broadcast over the set that i ranges over. A more complicated example is the three-dimensional iteration space of the expression $B_{ik} \cap C_{kj}$ where both B and C are broadcast, over the range sets of j and i respectively.

[14] I use lower-case letters to name sets whose points have one coordinate and upper-case letters to name sets whose points have two or more coordinates.

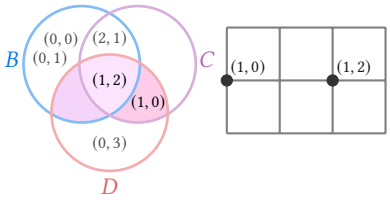


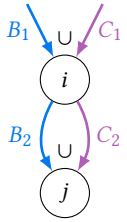
Figure 3-7: The sparse iteration space resulting from the iteration space expression $(B_{ij} \cup C_{ij}) \cap D_{ij}$ on the sparse iteration spaces from Figure 3-5.

3.2 Iteration Graphs

The **iteration graph** representation describes how to iterate over sparse iteration spaces. The iteration spaces were described as set expressions of coordinate relations. These relations, as we saw in Section 2.2, are represented as coordinate trees. An iteration graph describes how to iterate over the iteration spaces by iterating over a set expression of the coordinate trees. These trees impose a hierarchy on coordinates, and iterating over tree levels from top to bottom is faster than iterating in another order.[15] The coordinate trees and their order are reflected in iteration graphs, and they describe how to iterate over the iteration space by coiterating over coordinate trees.

An iteration graph represents a lexicographical iteration through an iteration space. The iteration through each dimension is represented by an index variable, and the index variables are organized in a tree structure. Nested index variables represent iteration over parents in a coordinate tree and their children, whereas sequenced index variables represent iteration through one coordinate tree segment followed by another. That is, if the index variable i is above j, then for each coordinate of i, we iterate through all the coordinates of j. And if j is left of k, then we first iterate through the coordinates of j and then through the coordinates of k. An iteration graph has directed paths going through the index variables that represent iteration over coordinate trees. These paths are ordered from the index variable that indexes into the top level to the one that indexes into the bottom level. Intuitively, each path represents a coordinate tree by symbolically collapsing all coordinate tree paths into a single path through index variables.

[15] For coordinate trees, whose levels support random access, such as those encoding dense tensors, iterating out of order only costs a constant amount of work to compute a strided formula, as well as reduced temporal locality.



$$\frac{\forall_i \forall_j \; B_{ij} \cup C_{ij}}{\begin{array}{l} i \in B_1 \cup C_1 \\ j \in B_2 \cup C_2 \end{array}}$$

Figure 3-9: The sparse iteration graph of $B_{ij} \cup C_{ij}$, represented both graphically and symbolically. The expressions under the line are iteration domains created from the iteration graph for dimensions i and j.

Representation

A graphical representation of iteration graphs lays out the nodes ordered by the tree embedding, and thus its iteration order, from top to bottom with children ordered from left to right. We label each edge with the coordinate tree level it represents and give each path a different color. For readability, we will often use the same colors for the corresponding operands in iteration space expressions. Finally, we place set operators between edges where they meet at dimension nodes to indicate how the tree levels they represent must be combined. Figure 3-9 shows the graphical iteration graph representation of the iteration space described by $B_{ij} \cup C_{ij}$, where the coordinate tree level ordering of both B and C is the i dimension followed by the j dimension. Note that the edges incoming on both dimensions are combined with the union operator, which turns a union of iteration space points into nested unions of coordinates (e.g., $B_1 \cup C_1$ at node i).

Although graphical representations are intuitive, a symbolic iteration graph representation is terser and will be easier to manipulate in a computer system. We represent iteration graphs symbolically as an ex-

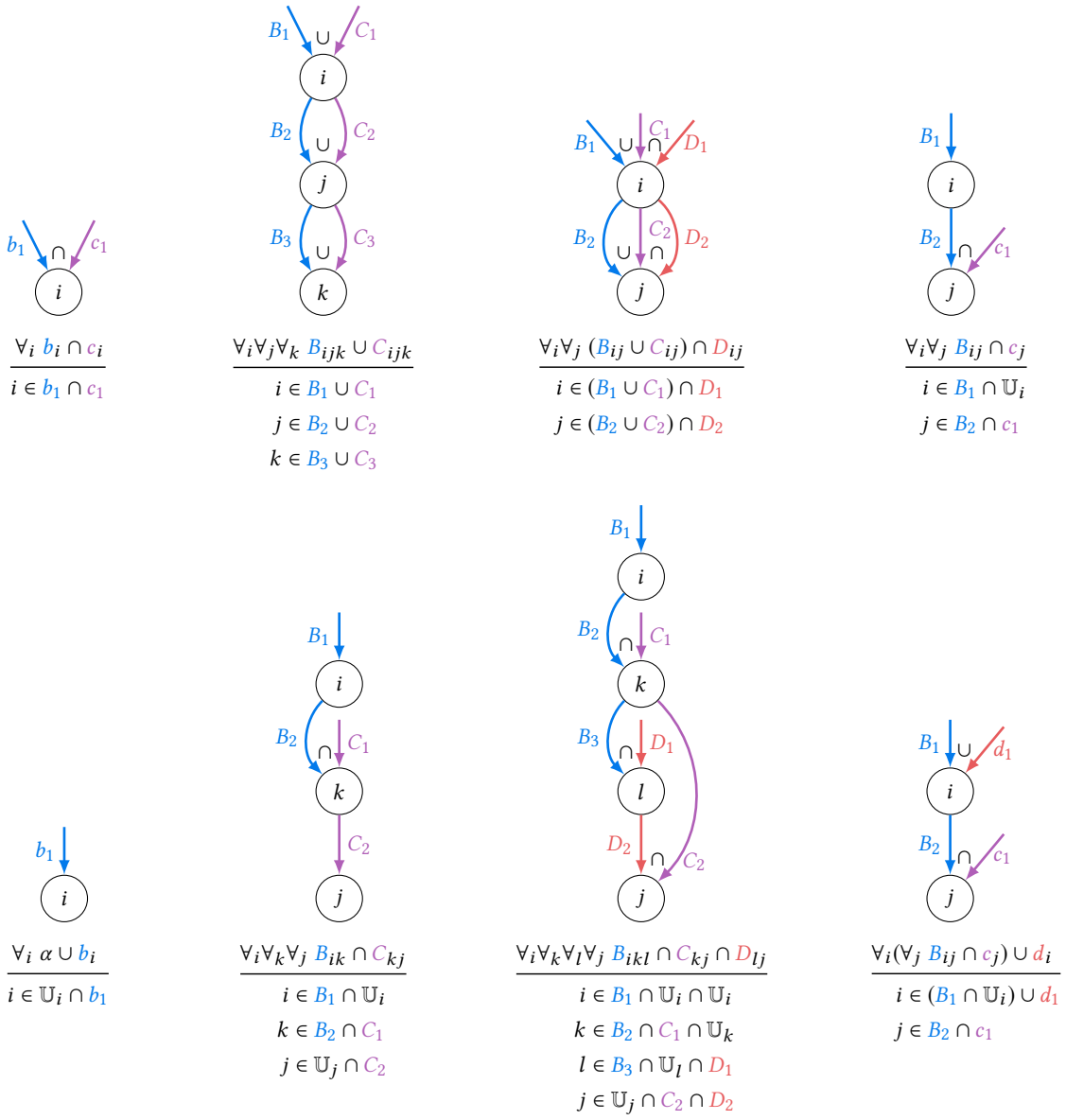


Figure 3-10: Eight sparse iteration graphs with index variable iteration domains computed from the iteration graph's symbolic expression for each index variable.

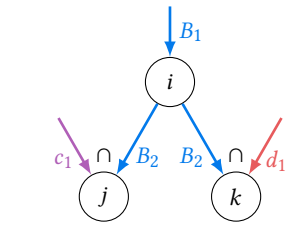
tension to the iteration space algebra where iteration over dimension nodes are represented as forall expressions that are nested according to the tree embedding. This extension gives the iteration graph expressions an operational semantics (iterate through dimensions in the given order), whereas iteration space expressions have declarative semantics (a point is in the iteration space if it is in the set described by the expression). The inner expressions are syntactically the same in the two algebras, but are in the iteration graph expressions reinterpreted so that indexed operands describe paths and operators describe how the coordinate tree levels should be combined.

Figure 3-10 provides eight examples of iteration graphs to help

us build a intuition for what they look like for different expressions. These graphs contain intersection and union operations, and some contain a mix of both. Iteration graphs can become arbitrarily complex when an arbitrary number of operations are combined.

Tree embeddings of iteration graphs can have branches, expressed symbolically as forall expressions nested inside set operators (Figure 3-11). Siblings in a tree embedding are concatenated iteration sub-spaces, nested inside their ancestors (Figure 3-12). The operator that combines the forall subexpressions also combines their iteration spaces, whereas the foralls nested inside it are concatenated.

We construct an iteration graph from an iteration space by wrapping its expression in one forall expression for each index variable. A choice must be made about the forall nesting order, and the construction machinery should aim to order them so that all iteration graph edges point down.[16] The reason is that coordinate trees, and their data structures, are organized to make it efficient to iterate over levels in order. Iterating over levels out of order requires locating positions by a call to their locate function or with search costing $O(\log(n))$ or $O(n)$. Iteration graphs may also have cycles, making it impossible to find an ordering without back edges. We must then either absorb the cost of out-of-order iteration or reorder the levels of the back edge's coordinate tree at the cost of a data structure transformation.



$$\forall_i \ (\forall_j B_{ij} \cap c_j) \cap (\forall_k B_{ik} \cap d_j)$$

$$i \in (B_1 \cap \mathbb{U}_i) \cap (B_1 \cap \mathbb{U}_i) = B_1 \cap \mathbb{U}_i$$

$$j \in B_2 \cap c_1$$

$$k \in B_2 \cap d_1$$

Figure 3-11: A sparse iteration graph with branches that describe concatenated iteration spaces.

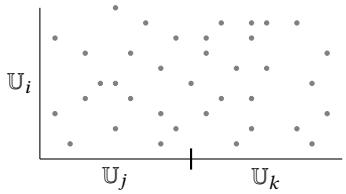


Figure 3-12: The sparse iteration space described by the iteration graph in Figure 3-11. The iteration spaces are concatenated on the x axis and intersected on the y axis.

Index Variable Iteration Universes and Domains

The index variables of an iteration graph each has an **iteration universe** and an **iteration domain**. Its iteration domain is the set of coordinates it iterates over, and its iteration universe is the universe of that set. In other words, an index variable's iteration universe is the set of coordinates that the iteration domain is drawn from. If the coordinate tree was full, say it comes from a dense tensor, then the domain would be the whole universes. The universe of index variable i is written as a blackboard bold $\mathbb{U}_i$. The universes considered in this dissertation are dense integer ranges, although future work includes generalizing them arbitrary sets. For example, the universe if i can be written as

$$\mathbb{U}_i \in [0, n),$$

where n is some integer, say the size of a tensor mode.

An index variable's iteration domain consists of the coordinates it iterates over, which is a non-strict subset of its universe. An iteration domain is expressed as a set expression over coordinate tree segments, where the segments are coordinate relation operands labeled with the coordinate tree level that contains the segment. For example, the index variables of the iteration graph $\forall_i \forall_j B_{ij} \cup C_{ij}$ has the following domains

[16] An ordering where all iteration graph edges point down can be achieved by forming a graph from the indexing expressions and topologically sorting the nodes ordered by the directed edges.

when the coordinate trees of B and C are both row-major:

$$i \in B_1 \cup C_1 \text{ and}$$
$$j \in B_2 \cup C_2.$$

(The expression has the domains $i \in B_2 \cup C_2$ and $j \in B_1 \cup C_1$ when the coordinate trees are column-major.) The notation does not indicate which segment is combined and the domains must therefore be interpreted recursively, with the domains of inner dimensions combining the segment given by the current index in outer dimensions. Hence, in the above expression the outer iteration domain of i describes iteration over the (single) segment of the top levels of the coordinate trees, whereas the inner iteration domain of j describes iteration over the segment of the second level indexed by the location of the current coordinate of i.

A broadcast is an operation where some operands are not indexed by some index variables. For example, the iteration graphs

$$\forall_i \forall_j B_{ij} \cap c_j \text{ and}$$
$$\forall_i \alpha \cup b_i,$$

where c and α are broadcast across the entire iteration space, even though they are not indexed by every index variable. Thus, the iteration domain of every index variable must include operands not indexed by its index variable. Because such operands do not have a tree level with coordinates to iterate over in that dimension, we insert instead an iterator over the full universe of the mode that i iterates over, labeled $\mathbb{U}_i$ for dimension i.[17] Thus, the expression $\forall_i \forall_j B_{ij} \cap c_j$ yields the iteration domains

$$i \in B_1 \cap \mathbb{U}_i \text{ and}$$
$$j \in B_2 \cap c_1,$$

while $\forall_i \alpha \cup b_i$ yields $i \in \mathbb{U}_j \cup b_1$.

[17] The effect on the graphical notation would be to add additional edges labeled by dimensions; however, for readability, we leave them out.

Derived Iteration Graphs

Iteration graphs constructed from an iteration space expression have exactly one forall statement per index variable that indexes into an operand. These forall statements iterate over an iteration space with the same dimensions as the original iteration space described by the iteration space expression. I refer to them as original iteration graphs and their index variable original index variables. But I also permit iteration graphs that iterate over index variables that derive from other index variables. I refer to such iteration graphs as derived iteration graphs, and they iterate over a derived iteration space. The derived index variables arise from optimizing transformations that add or remove dimensions from the iteration space. I discuss these transforma-

tions in Chapter 6, whereas this section describes the derived iteration graphs that they transform.

Derived index variables relate back to the original index variables through a sequence of relations. From these relations, the code generator generates code to map coordinates back and forth between the derived iteration space and the original iteration space. For example, it can map a coordinate in the derived iteration space, which we iterate over, to an original iteration space coordinate that we can use to index into a coordinate tree level. Or it can map an original iteration space coordinate that we read from a coordinate tree level, into a derived iteration space coordinate to compare it to a coordinate we are currently iterating over. There are two types of index variable relations: split and collapse.

The split relation splits an index variable j into two nested index variables j_0 and j_1. The outer variable j_0 iterates over an integer range that enumerates blocks, and its domain is identical to its universe. The inner variable j_1 coiterates over the same coordinate levels as j. In other words, j_0 iterates over blocks, while j_1 coiterates over the coordinate trees within the block. These coordinate trees are in different coordinate spaces—the universe of j_1 is the dense range from $[0, n/4)$, where n s the size of j's universe. The coordinates in the data structures B_2 and C_2, however, are in the universe of j. As I describe in Section 5.3, the coordinates of each space are mapped back to the original space, where they are compared to resolve the current combined coordinate value of j. The resolved j coordinate is then mapped back to the derived spaces. A split relation has three arguments: direction, size, and an optional coordinate tree. The direction—up or down—determines whether the number of blocks is constant (up) or whether the size of each block is constant (down). If the number of blocks is constant, then the size of each block is proportional to the size of j's universe. But if the size of each block is constant, then the number of blocks is proportional to the size of j's universe. The size argument is an integer that specifies the number of blocks or block size, depending on the direction. Finally, the optional coordinate tree argument creates blocks with the same number of coordinates from that tree, but that are variable-sized with respect to j's universe. Figure 3-13 shows the difference between a coordinate universe split (purple dashed) and a coordinate tree split (red)—a universe split evenly divides j's domain, while a tree split evenly divides the coordinate tree segments j indexes.[18]

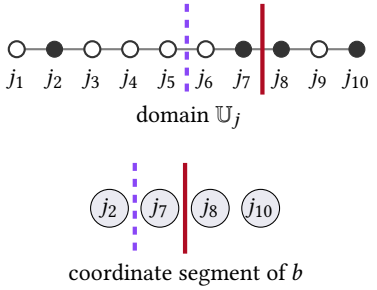


Figure 3-13: Two splits of the index variable j in $\forall_i\, b_i \cap c_i$—a universe split (purple dashed) and a b coordinate tree split (red).

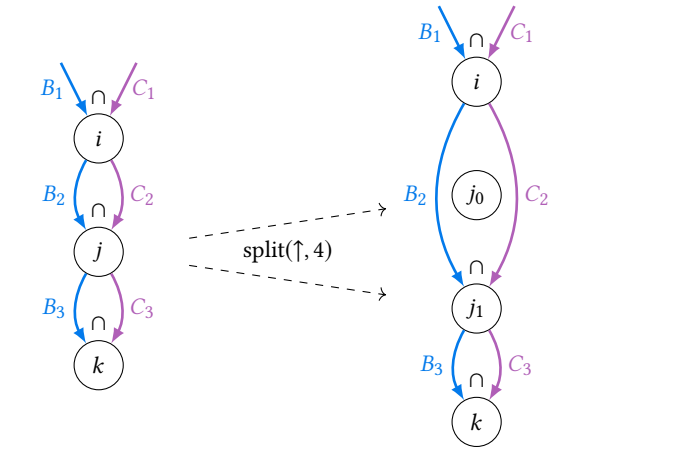
[18] A coordinate tree split results in an inner index variable with a dense domain, leading to code such as the generated CUDA SpMV implementation in Figure 1-14.

Figure 3-14 shows an example where an index variable j is split in the up direction with respect to its universe. The split changes the (original) iteration graph from

$$\forall_i \forall_j \forall_k\, B_{ijk} \cap C_{ijk},$$

which corresponds to an element-wise tensor multiplication, as

$$\forall_i \forall_{j_0} \forall_{j_1} \forall_k\, B_{ijk} \cap C_{ijk} : j \xrightarrow{\text{split}(\uparrow,4)} j_0 j_1.$$

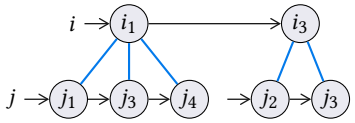


$$\frac{\forall_i \forall_j \forall_k \, B_{ijk} \cap C_{ijk}}{\begin{array}{l} i \in B_1 \cap C_1 \\ j \in B_2 \cap C_2 \\ k \in B_3 \cap C_3 \end{array}} \qquad \frac{\forall_i \forall_{j_0} \forall_{j_1} \forall_k \, B_{ijk} \cap C_{ijk} : j \xrightarrow{\text{split}(\uparrow,4)} j_0 j_1}{\begin{array}{l} i \in B_1 \cap C_1 \\ j_0 \in [0, 4) \\ j_1 \in B_2 \cap C_2 \\ k \in B_3 \cap C_3 \end{array}}$$

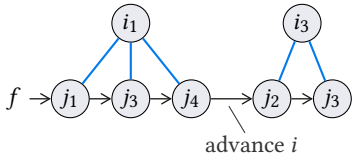
Figure 3-14: Two iteration graphs connected by a split relation. An original graph (left) that coiterates over two three-level coordinate hierarchies with three index variables, and a derived graph (right) whose middle index variables j_0 and j_1 derive from j in the original graph. Under the graphs there are iteration graph expressions, followed by the iteration domains of each index variable. The index variable j_0 iterates through equal-sized blocks of j's iteration domain, while each execution of j_1 coiterates through the coordinates of B_2 and C_2 within each block.

The derived iteration graph has two derived index variables, j_0 and j_1, that derive from j. These index variables have corresponding forall statements, and their universes are dimensions of the iteration space, but they do not index into the original coordinate trees. The relationship between j_0, j_1, and j is given as a relation after the : symbol, which should be read as "such that". Relations use an arrow annotated with the type of the relation and its arguments to map index variables to their derived counterparts. Figure 3-14 also contains the graphical representation of the derived iteration space. It shows how the relation maps between j in the original iteration graph and j_0 and j_1 in the derived iteration graph. It also shows the iteration graph expressions and index variable iteration domains for each iteration graph. The j_0 index variable iterates over the dense range $[0, 4)$ denoting four blocks. The j_1 index variable iterates over each block by coiterating over coordinate levels B_2 and C_2, intersected with the coordinates in the block. Each block contains one fourth of the coordinates in the universe of j's iteration domain—the first block contains the first fourth, the second block the second fourth, and so forth.

The collapse relation combines two nested index variables into a single index variable that iterates over the Cartesian combination of their coordinates. Collapse relations have no arguments. Figure 3-15 shows the effect fusion has on the iteration over a coordinate tree. Figure 3-15b shows the iteration after fusion: The j coordinates of every segment at the bottom level of the tree enumerates the subset of the i-j Cartesian combination that is present in the tree. Hence, the collapsed variable f can simply iterate over them to iterate over the Cartesian

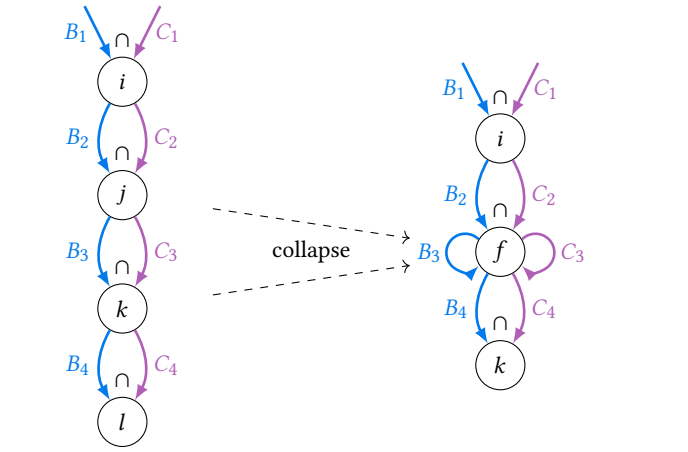


(a) The original index variables i and j iterate over their respective levels. For each iteration of i, j iterates over the corresponding segment.



(b) The collapsed index variable f iterates over the Cartesian combination of i and j by iterating over the bottom level of the coordinate tree. At each step it tracks the current j and i coordinates.

Figure 3-15: Iteration over a coordinate tree before and after fusing the i and j index variables into f.



$$\frac{\forall_i \forall_j \forall_k \forall_l \, B_{ijkl} \cap C_{ijkl}}{\begin{array}{l} i \in B_1 \cap C_1 \\ j \in B_2 \cap C_2 \\ k \in B_3 \cap C_3 \\ l \in B_4 \cap C_4 \end{array}} \qquad \frac{\forall_i \forall_f \forall_l \, B_{ijkl} \cap C_{ijkl} : jk \xrightarrow{\text{collapse}} f}{\begin{array}{l} i \in B_1 \cap C_1 \\ f \in (B_2 \times B_3) \cap (C_2 \times C_3) \\ k \in B_4 \cap C_4 \end{array}}$$

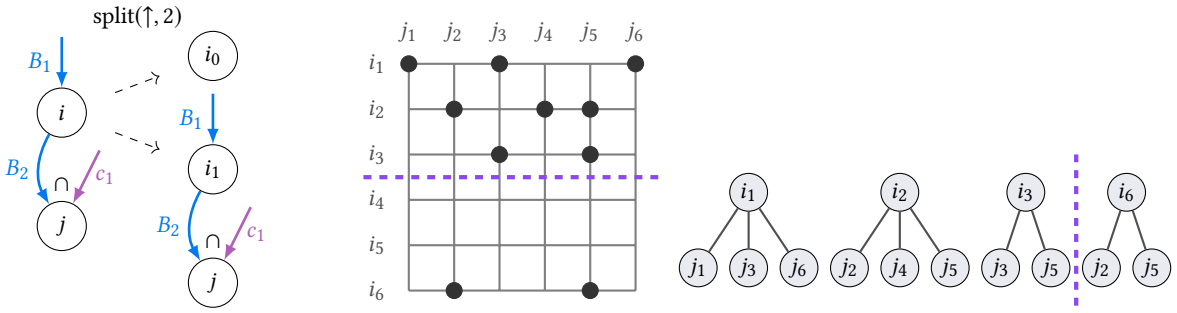
Figure 3-16: Two iteration graphs connected by a collapse relation. An original graph (left) that coiterates over two four-level coordinate tree with four index variables, and a derived graph (right) that coiterates over the same threes with three index variables. The middle collapsed index variable j coiterates over the cartesian combination of two coordinate tree levels from each of the coordinate trees. Under the graphs there are iteration graph expressions, followed by the iteration domains that highlight the iteration at each index variable.

combination. At each point in the iteration, the j and i coordinates can be recovered: the j coordinate can be read from the current position in the bottom level, and the i coordinate can be found by searching for the i coordinate that maps to the current j position. (If we collapse more index variables, then we must recover each index variable above the bottom through search.) Furthermore, we can improve on a search by tracking the current i coordinate and advancing it to the next when we come to the end of a row of j, as shown in the figure.

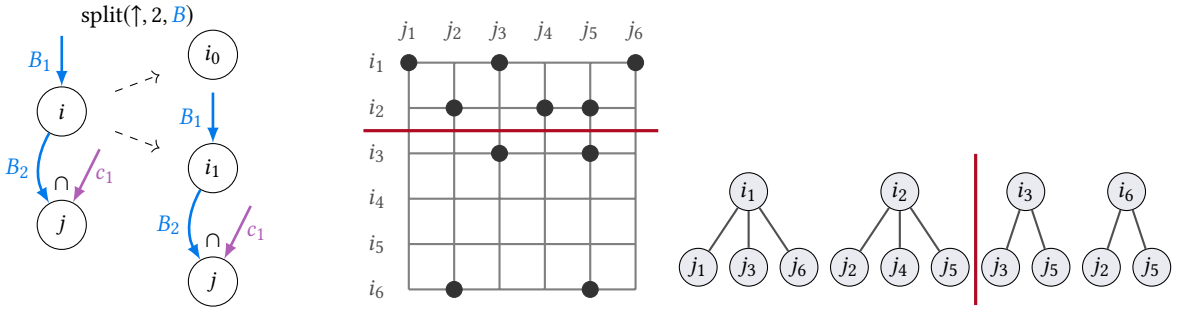
Figure 3-16 shows an example where a derived iteration graph (right) contains an index variable f that derives from two collapsed index variables j and k in an original iteration graph (left). The original iteration graph has four iteration variables that iterate over two coordinate trees with four levels each; whereas the derived iteration graph has only three index variables, it is still iterating over coordinate trees with four levels. The collapsed index variable coiterates over the Cartesian combination of the two levels by coiterating over the bottom levels—B_3 and C_3—and comparing both j and k.

Finally, Figure 3-17 compares three ways to divide into two pieces a two-dimensional iteration space formed by taking the intersection of a sparse coordinate relation B and a dense coordinate relation c. This iteration space appears in the doubly compressed sparse matrix-vector multiplication. Figure 3-17a and Figure 3-17c split the outer loop i with respect to its universe or the first level of B. The result is either an implementation that has the same number of rows in each block or one that has the same number of non-empty rows per block. Neither of these have the same amount of iteration space points, however, since

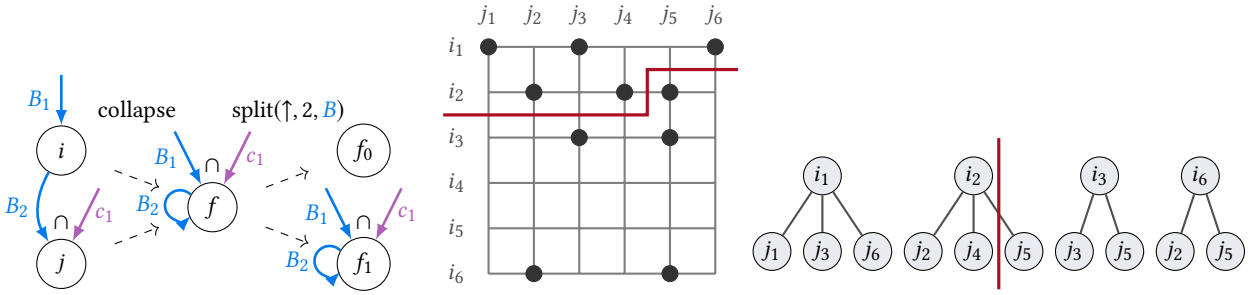
the rows may have a different number of points. Figure 3-17c combines an index variable collapse with a coordinate tree split to divide the iteration space into two pieces that have exactly the same number of non-empty points. This approach can be used to create statically load-balanced iteration spaces, and is the approach taken to create the load-balanced GPU SpMV implementation in Figure 1-14.



(a) By applying a universe split to the i dimension, we cut it into two halves with with the same number of rows and points, but with different numbers of of nonempty rows and points.



(b) By applying a coordinate tree split to the i dimension, we cut it into two halves with the same number of nonempty rows.



(c) By fusing the two dimensions followed by a coordinate tree split, we cut the iteration space into two halves with the same number of nonempty points. Observe that collapse and split are not inverse transformations.

Figure 3-17: Three ways to split up the two-dimensional iteration space of the iteration graph $\forall_i \forall_j B_{ij} \cap c_j$, which is the graph we would get from an SpMV operation $a_i = \sum_j B_{ij} c_j$. The three ways are a purple dashed universe split of the i dimension (a) and two red coordinate splits—one of the i dimension (b) and one of the collapsed i and j dimensions (c).

3.3 Iteration Lattices

An iteration graph describes iteration over a multidimensional iteration space as a hierarchy of iteration over index variables. Iteration domain expressions describe how to iterate over one index variable by

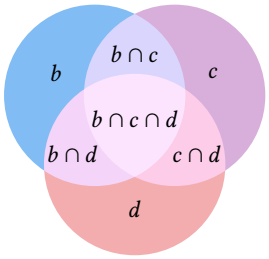


Figure 3-18: Venn diagram of iteration regions of $b \cup c \cup d$.

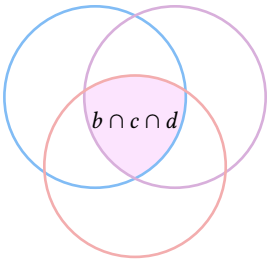


Figure 3-19: Venn diagram of the iteration regions of $b \cap c \cap d$.

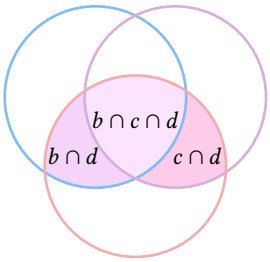


Figure 3-20: Venn diagram of the iteration regions of $(b \cup c) \cap d$.

coiterating over a set expression of coordinate tree segments. Since coiterating over segments can be expensive, I introduce a formulation that divides an iteration domain into smaller units that each coiterates over a subset of the segments. Iterating over only some of the segments is possible when the iteration algorithms determine that one or more of the segments has no more coordinates left to iterate over. It then jumps to a simpler iteration algorithm that ignores these segments.

We divide an iteration domain into regions described by the segments that intersect there. These regions are the powerset of the segments in an iteration domain; that is, the set of all subsets.[19] Thus, the iteration domain with k segments divide into 2^k **iteration region**s (the last region is the empty set $\emptyset$ where no sets intersect). Figure 3-18 shows the Venn diagram of the union of three segments with labels marking the seven non-empty regions where subsets of segments intersect (the last region is the background, where no segments intersect). For notational convenience, the regions in the Venn diagrams are labeled only with the sets that intersect there, leaving out of the expression a subtraction of regions where other sets also intersect. Thus, for example, the region where only c and d intersects is labeled

$$c \cap d,$$

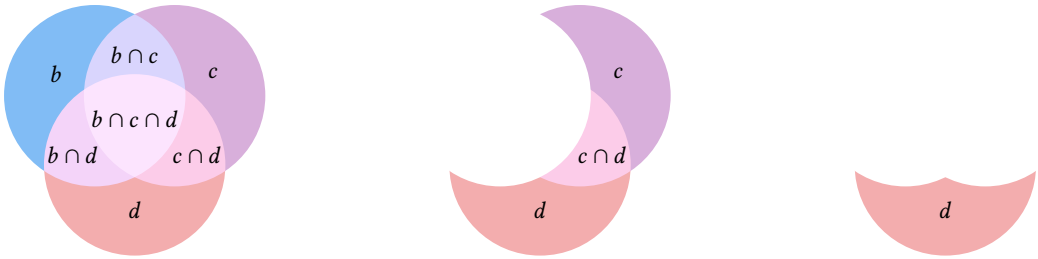
whereas the full expression is

$$c \cap d - b \cap c \cap d.$$

To iterate over union domains, we must iterate through each region, as shown in Figure 3-18. But if the iteration domain contains intersections, we only need to iterate through some of the regions (a subset of the powerset). For example, in the intersection in Figure 3-19, we only need to iterate through the single region described by the intersection of all three segments. Figure 3-20 shows that when the iteration domain is a combination of union and intersection operations, we need to iterate over more than one but not every region.

We organize the coiteration over an iteration domain so that when a region runs out of coordinates, the coiteration algorithm jumps to a simpler algorithm that excludes the segments that ran out coordi-

[19] A power set $\mathcal{P}(\mathcal{S})$ of a set $\mathcal{S}$ is the set of all subsets of $\mathcal{S}$, including $\mathcal{S}$ and the empty set.



Coiterate over regions with b, c, and d | b runs out of values | c runs out of values

Figure 3-21: Coiterate over the coordinate tree segments b, c, and d, considering all regions. When b runs out of values the coiteration proceeds to coiterate over only c and d, ignoring regions that contain b. And when c also runs out of values the coiteration proceeds to only iterate over d, ignoring regions that contain b or c.

nates (Figure 3-21). How, and whether, the algorithm can detect when segments run out of coordinates depends on the coiteration strategy. I discuss two coiteration strategies over two segments and then generalize to any number of segments using a lattice formulation. I call the two-way coiteration strategies *merge* and *iterate-and-locate*, and they demonstrate the coiteration patterns that the lattices implicitly use to construct n-way coiteration strategies for any number of segments combined with any combination of set intersection and union operators.

The two-way merge coiteration strategy is a common technique that is used in the merge sort and merge join algorithms.[20] It coiterates over two segments in $O(n)$ time, where n is the number of coordinates in the larger segment. A two-way merge can be used to coiterate over either an intersection or a union, but requires ordered segments. It coiterates through the segments in order until *either* runs out of coordinates. At each step, it compares their coordinates to determine whether or not they match. If they do, then that coordinate lies in their intersection, and both segments are advanced. If not, then the smaller coordinate is considered next. It lies in the iteration region that contains only one segment, and only that segment is advanced. The iteration terminates when either segment runs out of coordinates. At this point, the intersection variant of the merge strategy has completed, whereas the union variant must iterate through whichever segment still has coordinates left, if any.

The iterate-and-locate strategy is a simpler coiteration strategy that can be used to coiterate over either an intersection, shown in Figure 3-22, or a union where one of the segments is a superset of the other segment.[21] The two-way iterate-and-locate algorithm iterates over one of the segments and locates coordinates from the other segment using its locate capability.

The **iteration lattice**[22] of an iteration domain combines a set $\mathcal{S}$ of coordinate tree segments and is an ordered lattice of a non-strict subset of the power set of $\mathcal{S}$. It is inversely ordered on segment exclusion that occurs when a segment runs out of coordinates to iterate over. It is, in

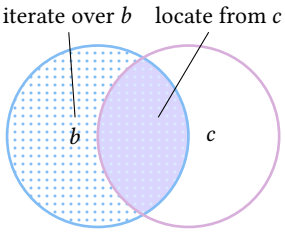


Figure 3-22: The iterate-and-locate strategy iterates over one operand and locates the coordinates in another.

[20] Knuth has a good description of the two-way merge algorithm in Chapter 5.2.4 of Volume 3 of The Art of Computer Programming [80].

[21] Databases sometimes use multi-phase union hash-join algorithms that I do not cover in this dissertation.

[22] In prior work, we referred to iteration lattices as merge lattices, but I take this opportunity to provide them with a more suitable name.

other words, a lattice of increasingly fewer segments to coiterate over until all segments run out of coordinates (Figure 3-23). The segment subsets are called iteration lattice points. The segments of a point are divided into two sets called its *iterators* and *locators* (Figure 3-24). The iterators are the segments to coiterate over in the lattice point, using a multi-way merge strategy, whereas the locators are segments we can simply locate from using the iterate-and-locate strategy.

A lattice can be viewed as a state machine that coiterates through subsets of regions as in Figure 3-21. Figure 3-25 shows an example with pseudocode. But we do not simply iterate over one region at a time. Instead, we coiterate over several regions until a segment runs out of values and then proceed to coiterate over the subset of regions that do not have that segment. Thus, the lattice points are used in two ways. First, they enumerate the regions we will successively exclude until all segments have run out of values; second, they enumerate the regions we must consider at the present moment. To iterate over an iteration lattice, we proceed in the following manner. We begin at the top lattice point:

1. Coiterate over the current lattice point's iterators until any of them runs out of values.
2. Compute the candidate coordinate, which at each step is the smallest of the current coordinates of the segments.
3. Determine which region the candidate coordinate is in by checking what segments are currently at that coordinate. The only regions we need to consider are those of the lattice points underneath the current lattice point (Figure 3-26).

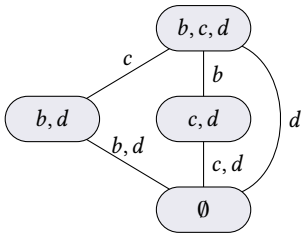


Figure 3-23: The iteration lattice of the iteration domain $(b \cup c) \cap d$ shown in Figure 3-20. Nodes are lattice points with segments to coiterate over (the top coiterates over all segments) and edges move to another point when a segment runs out of coordinates.

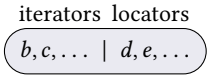
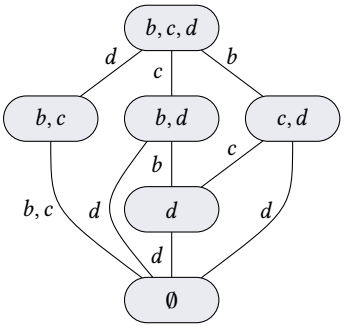


Figure 3-24: The segments in a lattice point are divided into iterators to coiterate over coordinates, and locators to locate the coordinate in.



while b, c and d have coordinates left **do**
  if in region $[b, c, d]$ **then** . . .
  else if in region $[b, c]$ **then** . . .
  else if in region $[b, d]$ **then** . . .
  else if in region $[c, d]$ **then** . . .
  else if in region $[d]$ **then** . . .

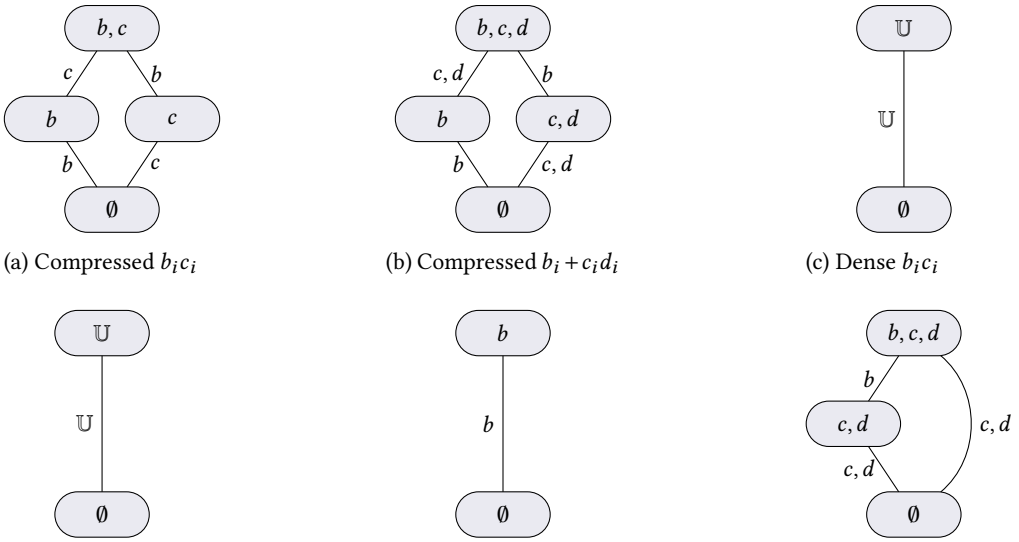
while b and c have coordinates left **do**
  if in region $[b, c]$ **then** . . .

while b and d have coordinates left **do**
  if in region $[b, d]$ **then** . . .
  else if in region $[d]$ **then** . . .

while c and d have coordinates left **do**
  if in region $[c, d]$ **then** . . .
  else if in region $[d]$ **then** . . .

while d has coordinates left **do**
  if in region $[d]$ **then** . . .

Figure 3-25: Iteration lattice and corresponding coiteration pseudocode for the iteration domain $(b \cap c) \cap d$. There is one `while` loop per lattice point and each `while` loop contains one `if` statement per sub-lattice point. Next to each `while` loop is a Venn diagrams showing the regions it coiterates over. The `while` loops iterate until a segment in its region runs out of values, and the `if` statements check which region a coordinate is in. Depending on the region, different actions are taken. Figure 5-2 in Chapter 5 will describe an algorithm that produces C coiteration code.



(a) Compressed $b_i c_i$ (b) Compressed $b_i + c_i d_i$ (c) Dense $b_i c_i$

(d) Dense $b_i + c_i$ (e) Mixed $b_i c_i$ (dense c) (f) Mixed $(b_i + c_i)d_i$

Figure 3-27: Iteration lattices for several expressions. (a)–(b) have only compressed operands, (c)–(d) have only dense operands, and (e)–(f) have mixed operands where c is dense while b and d are compressed. Lattices with dense operands are optimized.

4. When any segments run out of values, follow their lattice edge to a new lattice point and repeat the process until reaching the bottom.

This strategy leads to successively fewer segments to coiterate and regions to consider. We can use lattices, and this observation, to write code for any iteration domain that consists of a sequence of coiterating `while` loops that become simpler as we move down the lattice. Chapter 5 will show how to generate coiteration code from iteration lattices. Figure 3-27 provides several more lattice examples.

An iteration lattice can be constructed from an iteration domain. The construction algorithm proceeds bottom up, creating lattices at operands and merging them at operators (Figure 3-28). There are two types of operands (segments and dimensions) and two types of iterators (intersections and unions), resulting in four construction rules to create lattices that generalize the merge strategy. In addition, several optimizations can be employed to simplify the lattices and to move iterators to the locators set, thus creating lattices that combine the merge and iterate-and-locate strategies. The four iteration lattice construction rules are:

Segment rule The segment rule has two cases. If the level type of a segment supports an iterator capability, then the rule returns a lattice with a single non-bottom lattice point whose set of iterators contains the segment and whose set of locators is empty. If the level type does not support an iterator capability but instead supports locate, then the rule returns a lattice with a single non-bottom lattice point, the iterators set of which contains the

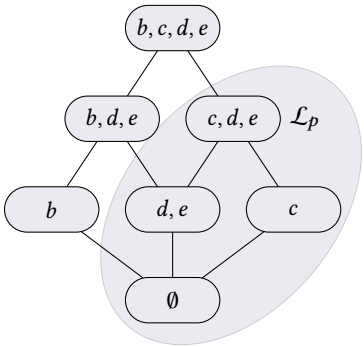


Figure 3-26: The sublattice of the lattice points below a lattice point $\mathcal{L}_p$. These are the lattice points that exclude segments that have run out of values.

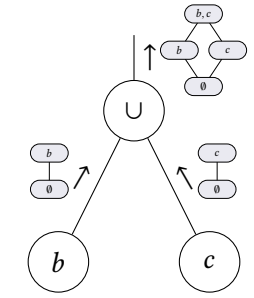


Figure 3-28: An iteration lattices constructed from the subexpressions of an iteration domain.

dimension of the segment and locators set contains the segment (Figure 3-29).

Dimension rule The dimension rule returns a lattice point with a single non-bottom lattice point whose iterators set contains the dimension and locators set is empty (Figure 3-30).

Intersection rule The intersection rule combines the lattices of its operands to produce a new lattice that describes iteration over the intersection of the operand iteration domains (Figure 3-31). To intersect two lattices, we first take the Cartesian product of their lattice points, which produces a set of ordered lattice point pairs. Next, we merge the lattice points in each pair: the union of the iterators and the union of the locators become the iterators and locators of the merged point. If there are multiple iterators in the merged iterators set and at least one of them is unordered, then we insert a dimension into the iterators set and move the unordered iterators to the locators set. Finally, we remove duplicates from the merged lattice points to produce the set of lattice points of the resulting lattice.

Union rule The union rule combines the lattices of its operands to produce a new lattice that describes iteration over the union of the operand iteration domains (Figure 3-32). To union two lattices, we first apply the intersection rule to produce an intersection lattice. We then take the union of the lattice points of the intersection lattice and the two operand lattices to produce the lattice points of the resulting lattice.

The four rules above produce correct iteration lattices; however, we can apply two optimizations during construction to produce lattices that coiterate over fewer segments:

Intersection optimization The intersection optimization takes advantage of the fact that the intersection of two sets is fully contained in either set. It is therefore sufficient to iterate over one of them to cover the entire intersection, provided there is a way to query whether coordinates exist in the other set and therefore is in the intersection (Figure 3-33).

The intersection optimization applied to the pre-merger lattice point pairs in the intersection rule, except when it is used to construct an initial lattice in the union rule. One of the locations in the pairs is chosen as the query side.[23] For each lattice point on the query side, we then move segments with the locate capability to the locators set.

Subset optimization The subset optimization takes advantage of the fact that the union of a subset and a superset is fully contained in the superset. It is therefore sufficient to iterate over the superset

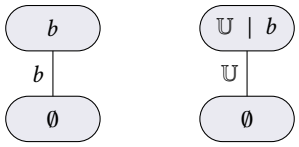


Figure 3-29: The iteration lattices of a segment expression, that supports an iterator capability (left) and that instead only supports locate (right).

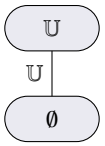


Figure 3-30: The iteration lattice of a dimension expression.

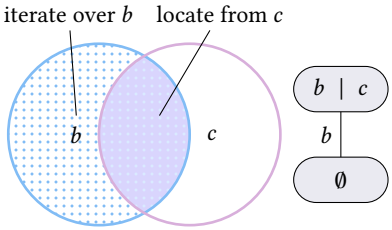


Figure 3-33: Venn diagram that shows the intersection optimization applied to $b \cap c$ where c supports locate. Instead of coiterating over b and c, we iterate over b and locate from c to determine if a coordinate is in the intersection.

[23] We can chose either side as the query side, but a reasonable heuristic is to select the side with more segments that can be moved to the locate set.

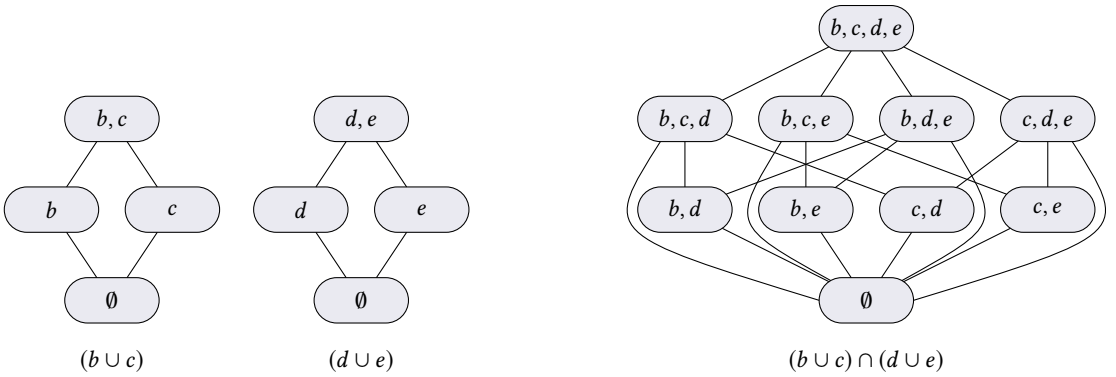


Figure 3-31: The iteration lattice intersection rule constructs a lattice from the Cartesian product of the lattice points of the operands, merging the resulting lattice point pairs.

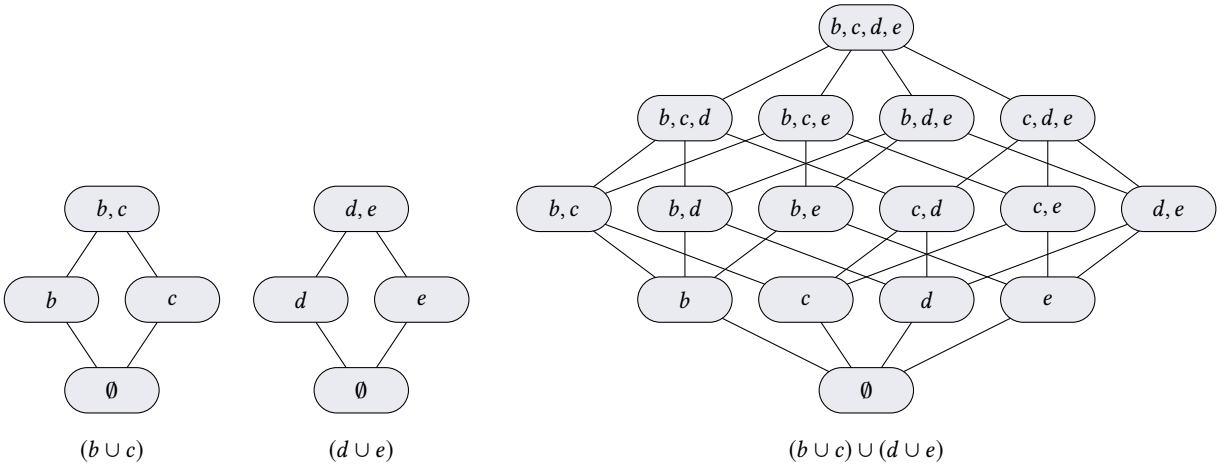


Figure 3-32: The iteration lattice union rule constructs a lattice from the union of the Cartesian product of the lattice points of the operands, merging the resulting lattice point pairs, and the lattice points of each operand.

to cover the entire union, provided the subset supports locate to query what region each coordinate lays in (Figure 3-34).

The subset optimization applies rules both to the pre-merger lattice point pairs and to the post-merger lattice points, in the union rule. One of the lattices of the union is first determined to be a subset of the other, and its location in the lattice point pairs is the subset side. Pre-merger, the optimization moves the segments with the locate capability of every lattice point on the subset side to the locators set. Post-merger, it remove all lattices whose iterators are all subsets of the iterators of a prior lattice point. For example, if the top lattice point has a full iterator, then we remove all iterators that do not have this lattice point.

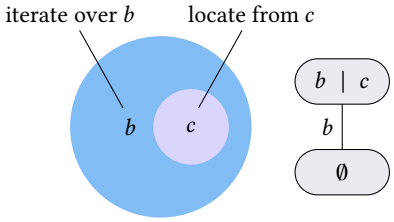


Figure 3-34: Venn diagram that shows the subset optimization applied to $b \cup c$ where $b \supseteq c$ and c supports locate. Instead of coiterating over b and c, we iterate over b and locate from c to determine the region of the coordinate.

3.4 Conclusion

In this chapter, I developed a comprehensive sparse iteration theory. This theory included an algebra that describes sparse iteration spaces as an indexed set operations of coordinate relations. It also included the iteration graph expression language that describes how to iterate through a sparse iteration space by recursively iterating over each dimension in turn. The iteration graphs can also express iteration over a derived space that is then mapped to the original space in order to enable fusion and tiling. Finally, I introduced an iteration lattice formulation that describes iteration over a single dimension of a space as a sequence of successively simpler iterations, ruling out coordinate trees when they run out of values.

The sparse iteration space theory puts sparse operations on a firm compiler foundation. It complements the polyhedral model for dense iteration spaces. Further, it lets us apply many of the traditional dense loop or iteration space transformations—reordering, loop collapsing, strip-mining, and tiling—to sparse loops and iteration spaces. Finally, the iteration lattice formalism enables us to reason about and generate code for data structure coiteration—joins!—over any number of data structures simultaneously. And it includes merge algorithms (e.g., merge join), locate algorithms (e.g., hash join), and any combination, with any number of data structures, all at once.

The sparse iteration space theory in this chapter is a foundation for reasoning about optimizing and compiling code that iterates over sparse iteration spaces. This includes tensor algebra, but I expect it to grow to also include relational algebra and graph operations. These operations also work on sets and can be formulated as iteration over sparse Cartesian coordinate spaces.

In Chapter 4, I extend the iteration space algebra and the iteration graphs to also include tensor computations. The resulting languages—tensor index notation and concrete notation—let us express tensor computations over sparse tensors. Chapter 5 shows how to compile the concrete notation to imperative code using iteration lattices. Because the concrete notation is an extension of iteration graphs, the code generation algorithm also works for iteration graphs in general. Finally, Chapter 6 shows how to transform concrete notation, and hence iteration graphs, to optimize iteration order.

"The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind. Economy of notation is therefore important."

— Ken Iverson

Chapter 4

Tensor Notations

4.1 Matrix Multiply Example

4.2 Tensor Index Notation

4.3 Concrete Index Notation

4.4 Concretize Algorithm

4.5 Conclusion

Tensor notations are powerful tools that describe relationships between tensors and how they should be combined. Mathematics advances with improvements in notation and the tensor index notation, since its development in the 1890s [109], has been used to express laws, relationships, and processes in physics, chemistry, engineering, data analytics, and machine learning. The general theory of relativity was first expressed as a tensor equation [51], quantum mechanics depends on tensors [54], they let us describe relationships between data sets [81], and tensor operations have now become the dominant language of neural networks [1].

We will examine two tensor languages—a generalization of the tensor index notation developed by Ricci-Curbastro and Levi-Civita [109] and a new language I call the concrete index notation. Tensor index notation is a declarative notation that describes the relationships between components of the result and the operands. The concrete index notation, one the other hand, is procedural and describes how to compute a result tensor from the operands. The power of the concrete index notation is that it lets us describe the order of operations, where temporary values are stored, and how iterations map to hardware—precisely the information we need to optimize computation. It does not, however, include the details of how to iterate over irregular data structures—`while` loops, `if` statements, and indirect memory accesses—and therefore requires no complicated analysis. These operations are introduced during the lowering of concrete index notation to imperative IR that I describe in Chapter 5.

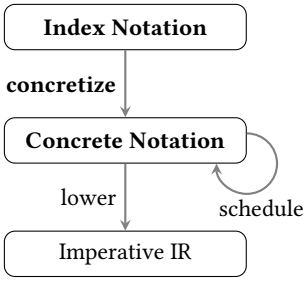


Figure 4-1: Compiler overview that highlights the tensor index notation, concrete notation, and concretize algorithm described in this chapter.

4.1 Matrix Multiply Example

We will first walk through a matrix multiplication example ($A = BC$) to get a sense of the notations and what the code generated from them looks like. We will also see how different transformations of the concrete index notation, described further in Chapter 6, can have a profound impact on the generated code. We will start out assuming that all matrices are dense but as we transform the intermediate repre-

sentation, we will change the data structures of first the operands and then also the result to the compressed sparse row (CSR) format (see Section 2.5 for a description of CSR). Although matrix multiplication comes from the linear algebra subset of tensor algebra, the language concepts, and compilation, applies the same way to higher-order tensor expressions. Matrix multiplication can be expressed in tensor index notation as

$$A_{ij} = \sum_k B_{ik} C_{kj}.$$

This notation is a declarative description that gives the relationship between each result component A_{ij} and the operands as an expression on the left-hand side.

The matrix multiplication can also be written in the concrete index notation.[24] Whereas tensor index notation is a declarative language that describes the relationship between results and operands, concrete index notation is a procedural language that describes computation that must be carried out to compute the results. The resulting statement, with dense pseudocode on the right, is

$$\forall_i \forall_j \forall_k \; A_{ij} \mathrel{+}= B_{ik} C_{kj}.$$

```
for i ∈ I
  for j ∈ J
    A_ij = 0
    for k ∈ K
      A_ij += B_ik * C_kj
```

The concrete index notation statement includes outer forall statements for the two free index variables i and j that index into the result. It also includes an inner forall statement for the summation variable k and replaces the assignment with a compound summation assignment +=. The statement has operational semantics, with the order of the forall statements giving the order of computation and the inner statement describing the computation that occurs for each iteration. Furthermore, result variables are implicitly initialized to zero. The pseudocode for the concretized matrix multiplication is given on the right. But, as we will see, the generated code may look very different depending on the storage formats of the tensors. Due to these storage formats, which describe coordinate hierarchies, the concrete index notation must iterate over the sparse iteration space of the operands that is described by an iteration graph. Figure 4-2 shows the iteration graph for the above concrete index notation statement, where the matrix C is stored in column-major order to avoid inefficient iteration through its data structure in the reverse direction.

The concrete index notation statements that result from concretization are often suboptimal. Reasons include traversing coordinate hierarchies in a different order than their level order, scattering into data structures that do not support random insert, or loop-invariant tensor subexpressions. In fact, in many cases, concretized sparse expressions are *asymptotically suboptimal*. By placing the reduction variable k in the innermost forall statement, the concretized matrix multiplication implements an inner product algorithm. This algorithm is typically the best algorithm for dense matrix multiplication because it avoids scat-

[24] In the taco compiler, users typically do not write concrete index notation. Instead, they write index notation and rely on taco to concretize it to concrete notation, where optimizations take place. The concrete index notation is thus used as an intermediate representation (IR).

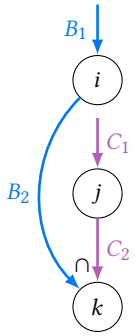


Figure 4-2: The iteration graph of the inner product matrix multiplication algorithm. The iteration graph assumes C is stored column-major. Because the reduction variable k is nested inside the free variables, there are no scattering behavior. But since there are no operand data structures connecting i and j, the algorithm requires iterating over the entire dense i-j iteration space.

tering into the result matrix[25] For sparse multiplications, however, it has two issues. The first issue is due to the (often asymptotically) high cost of iterating through the levels of a sparse coordinate hierarchy data structure out of order. If this forall statement ordering is to let us iterate over the coordinate hierarchies of matrices B and C in order, B must be stored in row-major order (e.g., CSR) and C must be stored in column-major order (e.g., CSC). The second issue is that the inner loop must coiterate over a sparse row and column. This coiteration requires an intersection merge algorithm that does work proportional to the number of components on the row with fewest components, instead of proportional to the intersection between them. Because the intersection can be smaller than the smallest of the two vectors, a merge algorithm may iterate over coordinate space points that do not contribute to the result.

By reordering the forall statements, we can change the matrix multiplication algorithm to a *linear combination of rows algorithm* that computes a row of A as a sum of rows from C scaled by a row from B. This algorithm does work only proportional to the number of multiplication that contributes to the result. We optimize concrete index notation statements by applying scheduling transformations, and one of these transformations is the reorder transformation. To obtain a linear combination of rows algorithm, we reorder the forall statements of j and k, yielding

$$\forall_i \forall_k \forall_j \; A_{ij} \mathrel{+}= B_{ik} C_{kj}.$$

```
A = 0
for i ∈ I
  for k ∈ K
    for j ∈ J
      A_ij += B_ik * C_kj
```

Figure 4-3 shows the corresponding iteration graph. The new order of the forall statements iterates through both operands in row-major order, which means a sparse row-major CSR data structure yields good performance. Furthermore, this algorithm does not require the merging of compressed data structures, the rows of B and C because the only merge happens at k, which indexes a compressed row and a dense column. The k forall statement, however, is now above the j forall statement resulting in scattering behavior into A because A is indexed by j and the k causes each coordinate in j to be visited multiple times. Scattering into the result matrix A is not an issue if its format supports efficient $O(1)$ insertion, such as a dense matrix.

Figure 4-4 shows C code generated from the linear combination of rows concrete index statement, where the operands are CSR matrices, and the result is a dense matrix. Figure 4-6 provides an example CSR data structure for the matrix B. Because it contains the subexpression B_{ik}, it iterates over B's sparse matrix data structure with the loops over i (line 2) and k (lines 3–4). The loop over i is dense because the CSR format stores every row, whereas the loop over k is sparse because each row is compressed. To iterate over the column coordinates of the ith row, the k loop iterates over [`B_pos[i]`, `B_pos[i+1]`) in `B_crd`.

In many applications, the result of a sparse matrix multiplication

[25] This coding pattern, where free variables are collected in the outer loops, computes results separately without any scattering behavior. It is often called owner-computes or output stationary and is particularly useful in dense computations, due to ease of vectorization and parallelization. In computations where the result is sparse, however, it can be problematic because one does not know a priori what result components are nonzero and therefore need to be computed.

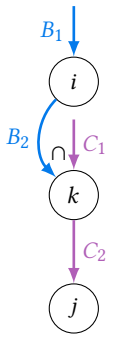


Figure 4-3: The iteration graph of the linear combination of rows matrix multiplication algorithm. The matrix C is now stored row-major. The iteration is now asymptotically efficient, but the last free variable j is nested under the reduction variable k resulting in values being scattered into the result matrix A. Since A is a compressed matrix, this is asymptotically inefficient.

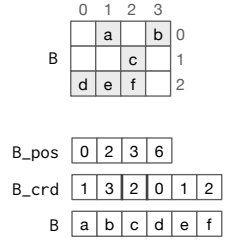


Figure 4-6: An $m \times n$ matrix B and its sparse CSR data structure.

```
1  memset(A, 0, A1_size * A2_size * sizeof(float));
2  for (int i = 0; i < I; i++) {
3    for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
4      int k = B_crd[pB];
5      for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++) {
6        int j = C_crd[pC];
7        A[i*n+j] += B[pB] * C[pC];
8      }
9    }
10 }
```

Figure 4-4: A sparse matrix multiplication, $A_{ij} = \sum_k B_{ik}C_{kj}$, where the operands are spares and the result is dense. The sparse matrices are stored with the CSR format and an example of the data structures of B is given in Figure 4-6.

```
1  memset(row, 0, row_size * sizeof(float));
2  for (int i = 0; i < I; i++) {
3    for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
4      int k = B_crd[pB];
5      for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++) {
6        int jp = C_crd[pC];
7        row[jp] += B[pB] * C[pC];
8      }
9    }
10
11   for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {
12     int jc = A_crd[pA];
13     A[pA] = row[jc];
14     row[jc] = 0.0;
15   }
16 }
```

Figure 4-5: A sparse matrix multiplication where the operands and the result are sparse. Sparse matrices do not support $O(1)$ insert so the code uses a dense row workspace. The index data structures of the result matrix A have been pre-assembled and all memory pre-allocated.

is sparse, and we may want the result matrix to also be stored in the CSR format. To gain intuition for when the result is sparse, consider a square matrix with one row and one column for every person in a social network. This matrix has a nonzero component wherever the person whose row the component is on is friends with the person whose column it is on. A multiplication of the matrix by itself acts like one step in a breadth-first search starting at every person. Thus, the resulting matrix will have nonzero components wherever two people are friends *and* wherever two people share a friend. The Milgram experiment estimated the degree of separation in the US population to be as high as six [94]; therefore, a single multiplication leaves most matrix components at zero. Other matrices, such as a finite element stiffness matrix of a mesh, have far higher degrees of separation, meaning the results of multiplying them are highly sparse.

A CSR result matrix complicates the kernel because the assignment on line 7 is nested inside the reduction loop k. This nesting order causes the inner loop j to iterate over and insert into each row of A several times. Compressed data structures, however, do not support fast random inserts (only appends), and inserting into the middle of a CSR matrix costs $O(\text{nnz})$ due to data movement.

The precompute transformation lets us sidestep the high cost of inserting into a compressed data structure, by rewriting the concrete index statement to accumulate results into a temporary row vector with $O(1)$ insertion, followed by a copy of the row to the end of the compressed result matrix. I refer to such temporary tensors as workspaces. Workspaces are often tensors of lower order than the result tensor, that support efficient random insertion, such as dense or a hash map. The concrete index notation that results from applying the precompute transformation to compute each row into a row workspace is

$$\forall_i \left(\forall_j \; A_{ij} = w_j \;\; \textbf{where} \;\; \forall_k \forall_j \; w_j \mathrel{+}= B_{ik} C_{kj} \right)$$

```
for i ∈ I
  w = 0
  for k ∈ K
    for j ∈ J
      w_j += B_ik * C_kj
  for j ∈ J
    A_ij = w_j
```

A where statement precomputes an expression and stores the result in a temporary workspace. In the concrete index statement above, one row at a time is computed and appended to A. The computation of the row into the workspace w is described on the right-hand side of the where statement. Figure 4-7 shows the iteration graph of the expression, where iteration domain over the workspace is concatenated to the second dimension.

Figure 4-5 shows C code generated from the above concrete index notation statement where both the operands and the result are stored in the CSR matrix format. The code assumes that A has been pre-assembled and the second loop can therefore replace the iteration over w with iteration over the ith row of A. Pre-assembling indices, common in simulation codes, increases performance when assembly can be moved out of inner loops, but it is also possible to simultaneously assemble and compute. Because values can be scattered efficiently into a dense workspace, the loop nest k, j (lines 3–9) looks similar to the kernel in Figure 4-4. Instead of storing values into the result matrix A, however, it stores them in a dense workspace vector. When a row of the result is fully computed in the workspace, it is appended to A in a second loop over j (lines 11–15).

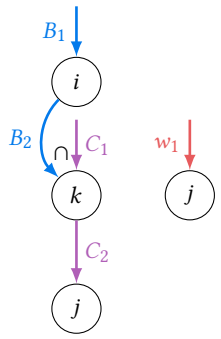


Figure 4-7: The iteration graph of the linear combination of rows matrix multiplication algorithm, with a row workspace vector w. This algorithm is efficient and avoids scattering into the compressed mode of the result A.

4.2 Tensor Index Notation

The **tensor index notation**[26] is a tensor language where an indexed assignment describes how each component on the left-hand side relates to an expression of the operands on the right-hand side (e.g., Figure 4-8). The expressions on the right-hand sides consist of tensor operands indexed by index variables, scalar operators that combine the indexed tensors, and reduction expressions that introduce an index variable that is reduced (e.g., summed) over. An **index variable** is a variable that is bound to a set of coordinate values termed its domain. These values correspond to values along tensor modes, and index variables can therefore be used to index into tensor modes. We call index variables that index into the result **free variables**, and the index variables that are bound to a reduction expression **reduction variables**. Figure 4-9 shows the grammar of index notation. It contains only a few common binary expressions and reduction expressions; however, it can readily be extended with new operators and reductions.

Reduction expressions consist of a binary operator $\oplus$, a reduction variable k, and an expression E. It is evaluated by calculating E for every value of k, combining the results with $\oplus$. Common reduction operators have special reduction symbols, such as addition (summation $\sum$)

[26] The tensor index notation I use in this dissertation is more general than the one used in the tensor algebra because it supports arbitrary nesting of reductions, additions inside reductions, and broadcast semantics for additions as well as multiplications. In Chapter 9, I describe other possible extensions such as support for operations in any semiring, support for general operations, and support for stencils.

$$A_{ijl} = \sum_k B_{ijk} C_{kl}$$

$$A_{0,3,1} = \sum_k B_{0,3,k} \, C_{k,1}$$

Figure 4-8: A tensor index notation example of a 3-order tensor multiplied by a matrix. The second expression shows, with indices separated by commas, the expression that describes the component of A at location $(0, 3, 1)$.

assignment :=	access "=" expr	operator :=	"-" expr
expr :=	reduction		expr "+" expr
	operator		expr "-" expr
	access		expr expr
	literal	access :=	$\text{tensor}_{\text{indices}}$
	"(" expr ")"	indices :=	index∗
reduction :=	$\text{"}\sum\text{"}_{\text{indices}}$expr		

Figure 4-9: The grammar of tensor index notation, where tensor, index and literal are identifiers. The notation can be extended with new operators and reductions.

and multiplication (product $\prod$). Other operators are simply indexed by the reduction variable, such as $\min_k$. Syntactically, the expression E is the next term after the reduction symbol unless parentheses are used to define scope, such as in $\sum_k (b_k + c_k)$.

The notation also supports powerful broadcast semantics when one or more of the operands are not indexed by one or more of the index variables, such as $a_i = \sum_j B_{ij} c_j$ or $A_{ij} = B_{ij} + c_j$ where the vector c is not indexed by i. When an index variable is missing from a tensor access expression, then that access expression is broadcast across the variable's index set.[27] Hence, in the above examples, the vector c is broadcast across i, meaning that it is respectively multiplied and added to every row of B.

[27] An access expression that lacks an index variable will be broadcast across the full index set of the index variable, even if the index variable indexes another operand that is sparse and therefore only has values at a subset of the coordinates of the index variable's index set.

4.3 Concrete Index Notation

The **concrete index notation**, or concrete notation for short, is a tensor language that specifies the order of computation and the storage location of intermediate computations (e.g., Figure 4-10). It is an excellent target for optimizing transformations that change the order of computation and that introduce temporary tensors to reduce work, increase locality, and take advantage of hardware features such as vectorization and parallelism. Although it is possible to express computations directly in concrete notation, such programs are not performance portable because they include optimization decisions that may only apply to one machine architecture. Instead, the concrete notation is intended to be an intermediate representation that is targeted by the optimizing transformations in Chapter 6.

$\forall_i \forall_j \forall_k \; A_{ijl} = t$ **where** $t \mathrel{+}= B_{ijk} C_{kl}$

Figure 4-10: A concrete index notation example of a 3-order tensor multiplied by a matrix ($A_{ijl} = \sum_k B_{ijk} C_{kl}$).

Grammar

The concrete index notation comprises five statements—assignment, forall, where, sequence and multi—and a scalar expression language (Figure 4-11). The assignment statement assigns the result of a scalar expression to a tensor component, the forall statement executes a statement over an index range inferred from tensor dimensions, the where statement creates temporaries that store subexpressions, the sequence

notation :=	stmt [":" environment]	expr :=	operator	environment :=	((relation | bound | tag) ",")∗
stmt :=	forall		access	relation :=	index index "$\xrightarrow{\text{"collapse"}}$" index
	where		literal		index "$\xrightarrow{\text{"split(" d "," s ")"}}$" index index
	sequence		"(" expr ")"	bound :=	"bound(" index "," b ")"
	multi	operator :=	"-" expr	tag :=	"parallelize(" index "," p "," r ")"
	assignment		expr "+" expr		"unroll("index "," u")"
forall :=	"∀"$_{\text{index}}$ stmt		expr "-" expr		
where :=	stmt "**where**" stmt		expr expr		
sequence :=	stmt ";" stmt	access :=	tensor$_{\text{indices}}$		
multi :=	stmt "|" stmt	indices :=	index∗		
assignment :=	access "=" expr				
	access "+=" expr				

Figure 4-11: The grammar of concrete index notation. I show only some incrementing assignments and binary expressions and others are possible. The grammar rules literal, tensor, index, d, s, b, p, r, and u are identifiers

statement reuses results, and the multi statement computes two results at the same time.

Concrete index notation expressions have two types of variables: index variables and tensors. An index variable is bound to a set of coordinate values by a forall statement and takes on each of these values at different iterations. They are also used in access expressions to access the tensor components to operate on in assignment statements. Index values could, in principle, be anything—numbers, strings, names, or webpages—but I only treat index values that are integers in this dissertation. Tensors are maps from coordinates to scalar components (see Chapter 2. They are only used in access expressions where they are indexed in each mode by an index variable or an index literal. The type of the index values must match the type of the corresponding tensor modes, and the index values must be a subset of the index values in the tensor mode. The tensor components are scalar values that can be a type that can be computed on by the operators they are used in. Common types are integers of different bit-widths, floats and doubles, complex numbers, and booleans.

The notation rule describes a complete concrete index notation statement. It consists of a statement body and an optional environment. The statement body can be either of the five statements, which in turn can be nested. The environment consists of relations, bounds, and tags. The relations relate derived index variables to the index variables they derived from. I describe them in Section 3.2, Section 6.3 (collapse), and Section 6.4 (split). The bounds place a constraint on the size of the universe of an index variable, and are describes in Section 6.5. And the tags specify how the index variable's forall statements should be lowered to imperative code, and are described in Section 6.6. Each index variable can be associated with at most one bound and one tag and can at most be the target of one relation.

An **assignment statement** computes the value of a single tensor component in the result tensor Figure 4-12). The result of an assign-

$$A_{i\ldots} = \text{expr}$$

Figure 4-12: Assignment statements compute a scalar expression and assign it to a component of a result tensor.

ment expression is one tensor indexed by index variables $i \ldots$ that are bound by enclosing forall statements and iterate through a set of index values. Expressions in concrete index notation cannot contain reductions. Rather, an assignment statement may be a **compound assignment**, which means that they use some operator to combine the result computed on the right-hand side. For example, an incrementing assignment += adds the value computed in each forall iteration to the result tensor.

A **forall statement** binds an index variable $i \in I$ to a set of index values I, called its range, and executes a statement stmt once for each index value (Figure 4-13). The range of the index variable must be the same set of index values as the mode of every tensor that it is used to index into. It can therefore be inferred from the tensor modes and, for ease of notation, we omit it when we write forall statements. A forall statement does not define an execution ordering and can be configured to be sequential, parallel, or vectorized (see Section iteration space mapping transformations).

$$\forall_i \text{ stmt}$$

Figure 4-13: Forall statements execute a statement for every value of an index variable's range.

A **where statement** consists of a producer statement stmt_p that computes temporary tensor variables that are consumed as operands in a consumer statement stmt_c (Figure 4-14). The temporary tensors, also referred to as workspaces, are defined and initialized on entering the where statement. They can be written to in the producer statement and exported as read-only tensors to the consumer side, where they can be used as operands. Their scope is the where statement, their lifetime is its duration of the where statement, and they are semantically recreated for each invocation (i.e., for each iteration of outer forall statements). A where statement can be used to introduce temporaries that can be scattered into, that are cheaper to coiterate over, and that can be used to hoist loop invariant code (see Section 6.2).

$$\text{stmt}_c \textbf{ where } \text{stmt}_p$$

Figure 4-14: Where statements produce one or more temporary tensors in a producer statement that are then consumed as operands in a consumer statement.

A **sequence statement** computes a tensor in two stages. It consists of a statement stmt_d that defines a tensor and a statement stmt_m that mutates it (Figure 4-15). Sequence statements can be nested to compute a tensor in multiple stages: stmt_d; stmt_{m_1}; stmt_{m_2}; .... A sequence statement can be used to remove temporaries by instead successively adding additional computations to the same result (see Section 6.2).

$$\text{stmt}_d;\ \text{stmt}_m$$

Figure 4-15: Sequence statements define one or more temporary tensors in a definition statement that are then mutated in a mutate statement.

A **multi statement** combines two statements stmt_l and stmt_r that compute different results (Figure 4-16). The statements may execute in any order and can share operands. A multi statement can be used to compute two tensors inside the same shared outer loop.

$$\text{stmt}_l \mid \text{stmt}_r$$

Figure 4-16: Multi statements combine statements with separate result.

Connection to Iteration Graphs

The concrete notation has a close connection to iteration graphs. Concrete statements compute tensor expressions with sparse tensors and may, due to the algebraic properties of their operators, iterate over only a sparse subset of the expression's full iteration. The concrete notation lowering machinery in Chapter 5 converts a concrete forall

statement to an iteration lattice, by way of iteration graphs and the forall index variable's iteration domain. I described how to convert iteration graphs to iteration domains, and iteration domains to iteration lattices in Chapter 3. This section describes how to convert a concrete notation statement into an iteration graph.

We can turn a concrete notation statement into an iteration graph that only iterates over a sparse iteration space, by taking advantage of the algebraic properties of the operators in the concrete expressions. Specifically, because sparse tensors compress out zeros, we will take advantage of the zero annihilators and zero identities.[28] An operator is annihilated (or absorbed) by zero if applying it to any value, and a zero yields a zero. For example,

[28] This dissertation does not cover the generalization of these ideas so that any function can be used in index and concrete notation expressions.

$$a \times 0 = 0$$

for all values of a. Owing to the fact that both operands must be nonzero for the result to be nonzero, it is sufficient to iterate over the intersection of two tensors combined by an operator with a zero annihilator. Furthermore, an operator has zero as its identity element if applying it to any value, and a zero yields the value. For example,

$$a + 0 = a$$

for all values of a. A consequence of this property is that applying the operator to two zeros yields a zero. Because at least one operand must be nonzero for the result to be nonzero, it is enough to iterate over the union of two tensors combined by an operator with a zero identity.

We can use these insights to define an algorithm to convert a concrete notation forall statement to an iteration graph expression. The algorithm recurs bottom up on the forall statement and applies different rules to the statements and expressions in its body based on their type. These rules combine the iteration graphs produced at substatements and subexpressions and are listed below.

Access Return an indexed coordinate relation with the same coordinates as the nonzero components of the tensor.

Literal Return a scalar coordinate relation.

Negation Return the iteration graph produced for the subexpression.

Parenthesis Return a parenthesis iteration graph expression that contains the iteration graph produced for the subexpression.

Addition and Subtraction Return the union of the iteration graphs produced for the left and right subexpressions.

Multiplication Return the intersection of the iteration graphs produced for the left and right subexpressions.

Assignment Statement Return the iteration graph expression produced for the right-hand side of the assignment.

Forall Return a forall iteration graph expression, the index variable of which is the same as the concrete notation forall and the body of which is the iteration graph produced for the concrete notation forall body.

Where Return the consumer iteration graph where the producer result (the temporary) is replaced by the producer iteration graph.

Sequence and Multi Return the union of the iteration graphs produced for the two substatements.

4.4 Concretize Algorithm

The concrete index notation was designed to be an intermediate representation for expressing program transformations and optimizations and not a primary language users would write their programs in. The reason is that programs written in concrete index notation are more difficult to read than equivalent programs in index notation and also interleave application code and optimization decisions. They therefore tend to be specific to one machine architecture and less performance portable.

The intended usage model is therefore that users write their programs in tensor index notation that is then transformed to concrete index notation through a process I call **concretization** (see Figure 4-1). This process is designed to be simple so that users can predict how the concretized code will look. Optimizations are then applied through a sequence of transformations of the concrete notation expressed with a scheduling language (Chapter 6).

The concretization algorithm converts an index notation assignment statement into a concrete notation statement. It replaces reduction expressions with compound assignments inside where statements and wraps the resulting statement in forall statement for the index variablefree variable. Each reduction expression is replaced by a where statement. The producer statement is a forall statement over the reduction variable that contains a compound assignment with the reduction operator. The left-hand side of the assignment is a scalar temporary variable, and the right-hand side is the reduced expression. The consumer statement is the original index statement with the reduction expression replaced by the temporary tensor. The following matrix-vector multiplication plus vector example ($a = Bc + d$) demonstrates this rewrite:

$$a_i = \sum_j \left(B_{ij} c_j\right) + d_i \implies \forall_i \left(a_i = t + d_i \textbf{ where } \forall_j \; t \mathrel{+}= B_{ij} c_j\right).$$

One simplification is applied to make the concrete index notation statements cleaner for the subsequent scheduling transformations. The simplification is applied if the assignment statement that consumes the temporary only has the temporary on its left-hand side and is either

not a compound assignment or is a compound assignment with the same operator as the compound assignment that produced the temporary. The simplification

1. hoists the loop in the producer statement out of the where statement,
2. replaces the temporary tensor on the consumer side with the right-hand side of the assignment that produced it and replaces the consumer assignment with a compound assignment with the same operator as the assignment that produced the temporary, and
3. replaces the where statement with its consumer side.

The step-by-step effect of this transformation is shown in the following matrix-vector multiplication example ($a = Bc$):

$$\forall_i \left(a_i = t \textbf{ where } \forall_j \; t \mathrel{+}= B_{ij}c_j\right)$$
$$\forall_i \forall_j \left(a_i = t \textbf{ where } t \mathrel{+}= B_{ij}c_j\right)$$
$$\forall_i \forall_j \left(a_i \mathrel{+}= B_{ij}c_j \textbf{ where } t \mathrel{+}= B_{ij}c_j\right)$$
$$\forall_i \forall_j \; a_i \mathrel{+}= B_{ij}c_j$$

Finally, the concretization algorithm wraps the assignment statement in one forall statement for each index variablefree variable in the index notation statement, nested by the order they index the assignment statement's left-hand side.

4.5 Conclusion

In this chapter, I introduced two tensor notations. The declarative tensor index notation describes the relationships between components of the result and the operands. It is intended to be used by the end user. The concrete notation, on the other hand, is primarily intended as an intermediate representation. It specifies how to compute results by iterating over an iteration space made sparse by the algebraic properties of scalar operators. I have also described a concretization algorithm that translates tensor index notation to concrete notation.

The tensor languages and the translation let us convert the tensor index notation provided by the user to computation over sparse iteration spaces. They therefore facilitate sparse imperative tensor algebra code generation. Furthermore, the concrete notation lets us reason about optimizing transformations on tensor algebra iteration spaces *before* we introduce sparse imperative constructs such as `while` loops, `if` statements, and indirect memory accesses. This separation of transformations from sparse code is an essential design feature because sparse code makes it substantially difficult to analyze and optimize sparse code.

In Chapter 5, I show how to generate imperative code from the concrete notation, using the connection to iteration graphs and iteration lattices. Further, in Chapter 6, I describe several transformations that apply to concrete notation to optimize its sparse iteration, its computations, and its mapping to imperative code and parallel hardware.

"Smart data structures and dumb code works a lot better than the other way around."

— Eric Raymond

Chapter 5

Coiteration Code Generation

In this chapter, I use the ideas and intermediate representations from the preceding chapters—coordinate trees, concrete notation, iteration graphs, and iteration lattices—to describe a code generation algorithm that lowers concrete notation statements to imperative code. Figure 5-1 highlights the lower algorithm in the compiler flow. It is applied after optimizing transformations have been applied to the concrete notation, as described in the next chapter. The lower algorithm takes in a concrete notation statement and produces imperative loops. These loops iterate over the concrete statement's sparse iteration space by coiterating over the data structures of its operands. The lower algorithm also produces code to compute result values and code to assemble result index data structures. It can be configured to emit compute code, assembly code, or both, which makes it possible to reuse the index data structure of the result when the values of the operands have changed but not their index data structures.

5.1 Algorithm Overview

5.2 Coiteration Code

5.3 Derived Iteration Spaces

5.4 Compute and Assembly

5.5 Conclusion

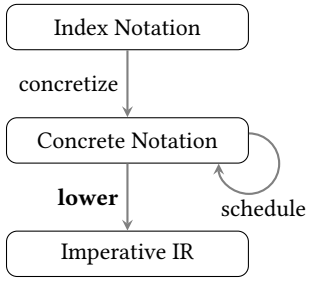


Figure 5-1: Compiler overview that highlights the lowering algorithms described in this chapter.

5.1 Algorithm Overview

Figure 5-2 shows the pseudocode for the lowering code generation algorithm. It operates recursively on statements in the concrete notation, and it has one lower function for each statement type. The figure uses color coding to separate the algorithm's control flow (black text) from the statements that emit code (colored text).

Most of the action is in the forall lower function. It generates code to iterate over one index variable, which describes one dimension of the iteration space. (The recursion composes per-dimension iteration into multi-dimensional iteration.) The forall lower function has two salient features: nested loops over iteration lattice points and statement simplification before the recursive call.

The first statement in the forall lower function constructs an iteration lattice from the forall loop by converting it to an iteration graph, extracting the index variable's iteration domain, and building a lattice from it. Section 4.3 describes how to convert concrete notation to an iteration graph, and Chapter 3 describes how to get iteration

```
function LOWER(forall statement S_forall of index variable i)
    let L be an iteration lattice constructed from S_forall
    Emit initialize iterators                                          ▷ Section 5.2
    for each lattice point L_p in L do
        Emit loop header                                               ▷ Section 5.2
        Emit access iterators                                          ▷ Section 5.2
        Emit map candidate coordinates to the original space           ▷ Section 5.3
        Emit resolve the coordinate of i                               ▷ Section 5.2
        Emit map resolved coordinate to each derived space             ▷ Section 5.3
        Emit locate from locators                                      ▷ Section 5.2
        for each lattice point L_q < L_p in L do
            Emit conditional header                                    ▷ Section 5.2
            let S_simplified be a statement constructed from
                the body of S_forall by removing operands
                that have run out of values in L_q
            LOWER(S_simplified)
            Emit assembly code                                         ▷ Section 5.4
            Emit conditional footer                                    ▷ Section 5.2
        end for
        Emit advance iterators                                         ▷ Section 5.2
        Emit loop footer                                               ▷ Section 5.2
    end for
end function

function LOWER(assignment statement S_assignment)
    Emit compute code                                                  ▷ Section 5.4
end function

function LOWER(where statement S_where)
    LOWER(producer statement of S_where)
    LOWER(consumer statement of S_where)
end function

function LOWER(sequence statement S_sequence)
    LOWER(definition statement of S_sequence)
    LOWER(mutation statement of S_sequence)
end function

function LOWER(multi statement S_multi)
    LOWER(left statement of S_multi)
    LOWER(right statement of S_multi)
end function
```

Figure 5-2: Algorithm that lowers concrete notation to imperative IR. The LOWER function recurs on concrete notation statements, and each call invokes the lower function that match the statement type. Black statements are the control flow of the lower algorithm, while colored statements emit code. Blue statements emit code to iterate over the concrete notation's sparse iteration space and green statements emit code to assemble result data structures and to compute values that go into those data structures.

domains from iteration graphs and iteration lattices from iteration domains. The lowering algorithm uses the lattice to generate `while` loops and `if` statements that coiterate over tensor coordinate trees. It generates these statements using two nested loops over lattice points: the outer loop generates `while` loops that coiterate over coordinate segments until they run out of values, and the inner loop generates if statements that execute different code in different regions of the iteration domain. The inner loop iterates over only the lattice points below the current point in the outer loop, because the regions of the other lattice points have already run out of coordinates when we get to this point. See Section 3.3 for more information on iteration lattices.

The current lattice point in the inner loop is used to simplify the body of the forall statement before it is passed down the recursion. The simplification sets tensor access expressions in the forall body to zero if they have run out of values in a prior lattice point, which is the case if the tensor is not included in the current lattice point. The simplification then further algebraically simplifies the concrete statement by propagating the zeros. For example, if B_1 runs out of values in

$$\forall_i \forall_j \forall_k (B_{ik} + C_{ik}) D_{kj},$$

then B_{ik} is set to zero and the expression simplifies to

$$\forall_i \forall_j \forall_k C_{ik} D_{kj}.$$

The simplification creates a peculiar recursion pattern, where the recursive call for each lattice point emits different code into each `if` statement. But the simplified code is cleaner and does not need to guard against out-of-bounds data structure accesses.

In the next sections, I describe the code emitted by the lowering algorithm. The lowering algorithm has several colored emit statements that emit blocks of imperative code with different functions. I divide the emit statements into statements that generate coiteration code (Section 5.2), statements that map index variables to the current index variable's universe (Section 5.3), and statements that compute values or assemble result data structures (Section 5.4). I give examples of emitted code in the C language, although the taco compiler emits an imperative IR that is specialized to C, CUDA, or LLVM.

The lowering algorithm has an environment with information about iterators. There is one level iterator for each coordinate tree level, as well as one universe iterator for each index variable. The iterators are compile-time concepts: they do not iterate over data structures. Instead, they provide imperative code to do the iteration. This code is retrieved from the level types in the tensor formats, which I described in Chapter 2. The iterators of a tensor coordinate tree are chained together, and the iterator at level $k + 1$ iterates over the segment at the position described by the iterator at level k. The iterator chaining thus affects the code to compute the beginning and end of coiteration loops.

5.2 Coiteration Code

The blue statements in Figure 5-1 emit imperative code that coiterates over the index data structures of the tensor coordinate trees. I describe each emit statement and give C examples that iterate over compressed levels and universe iterators.

Iteration Variable Naming Conventions

There are three types of induction variables involved. Each iterator that iterates over a coordinate tree segment has one variable that tracks its current position and one variable that tracks its current coordinate. The naming convention of the position variable of an iterator is the letter p followed by the name of the coordinate tree level of the segments that the iterator iterates over. For example, the position variable of an iterator over segments of B_2 is named pB2. The naming convention of the coordinate variable of an iterator is the name of the index variable whose domain the iterator contributes to, followed by the name of the coordinate tree whose segments the iterator iterates over. For example, the coordinate variable of an iterator that is part of index variable i's domain and that iterates over a coordinate tree B is named iB.

The current coordinate of each iterator is called a candidate coordinate. The candidate coordinates may differ from each other at any given iteration, depending on what coordinates are stored in their respective coordinate trees. They must, therefore, be combined to determine the coordinate of the current iteration, called its resolved coordinate. The merge strategy considers the smallest candidate coordinate to be the resolved coordinate, and it uses the min function to find it. The variable of the resolved coordinate is named after the iterator. For example, the candidate coordinate variable of iterator i is named i.

Emit initialize iterators

The first step is to emit code to define and initialize one loop induction variable for each segment in the iterator set of the entire lattice, which is the union of the iterator sets of its lattice point. These induction variables are integers, and the initialization expressions are the $\texttt{ibegin}_k$ results of the bounds level function in each iterator's iterator capability.

If the iterator capability is a position iterator, then the naming convention for the induction variable is a p followed by the level name, and the naming convention for the variable that holds the end of a segment is p followed by the level name followed by _end.[29] For a level B_2, where the i coordinate index into the parent level B_1, the generated C code is:

[29] Note that the end of segment variables are often collapsed into the loop bounds for code readability.

```
int pB2     = B2_pos[i];
int pB2_end = B2_pos[i+1];
```

If the iterator capability is a coordinate iterator, then the naming convention is the name of the forall's index variable followed by the

name of the coordinate relation of the level (e.g., `iB`). Universe iterator variables are named after the index variable, as are segment iterator induction variables if there is only a single level iterator. The generated initialization code for a universe induction variable over i is:

```
int i = 0;
```

If the resulting tensor is appended to, as opposed to inserted into, then this step also emits code to define the position variable of the last result level and initialize it to 0. For example:

```
int pA2 = 0;
```

Finally, suppose an index variable i is split into two nested index variables i_0 and i_1, where i_0 iterates over blocks and i_1 iterates through each block. Since each iteration of i_0 jumps to the next block, the iterators of i_1 must be initialized by searching for the block start. For example, suppose we split i into i_0 and i_1 in the computation $a_i = b_i + c_i$, where b and c are compressed vectors. Then the `i1b` and `i1c` iterators over i_1 are initialized to the start of the current block indexed by i_0. If b and c support the locate capability, then the emitted code can use it to find the block start and end positions in b and c. Compressed formats do not support locate, however, so the code must perform a search. If the compressed formats are ordered, then the code can use a binary search; otherwise it must use a linear search. For example, the initialization of the iterator of b in this example is:

```
int pb     = search(b_crd, b_pos[0], b_pos[1],  i0   *block_size);
int pb_end = search(b_crd, b_pos[0], b_pos[1], (i0+1)*block_size);
```

The `search` function that initializes `pb` searches the `b_crd` array, in the range [`b_pos[0]`:`b_pos[1]`), for the value of `i0*block_size`. It returns the position of `i0*block_size` in `b_crd` or, if `b_crd` does not contain `i0*block_size`, the position of the largest value that is smaller than it. The search function is used in the SpMV kernel optimized for GPUs given in Figure 1-14.

The search function is also used to initialize the upper variables of a collapsed set of index variables. Figure 3-15 shows how a collapse relation replaces the nested iteration over two nested index variables with the iteration over their Cartesian combination. In the generated code, this means that the two nested loops are replaced by a single loop that iterates across all the segments of the coordinate tree level(s) of the bottom variable. But the coordinates of the top variable or variables must also be computed. The search function can be used to compute the top variable's coordinates for each of the bottom variable's coordinates inside the generated loop for bottom variable. But, if the coordinate trees the top variable iterates over supports an iteration capability, then a tracking optimization can be applied. The tracking optimization is applied during code generation and produces code that initializes the upper variable(s) once before the loop of the inner variable is executed, and then advances them each time the inner variable

loop reaches the end of one of their segments. The initialization uses the search function, and the upper iterator advancement is described in the section below on emitting code to advance iterators.

Emit loop header

We saw in Section 3.3 that iteration lattices describe iteration through successive regions of an index variable's iteration domain. In code, this results in a sequence of `while` loops, one for each region. Each of them coiterates over segments or coordinate universes until one of them runs out of coordinates. The header of each `while` loop tests whether any of the iterators have run out of coordinates. This test is done by checking whether the iterator induction variables are still less than the value of the $\texttt{end}_k$ result of the bounds level function of the iterator's iterator capability. If two coiterating segments B_2 and C_2, where the i coordinate indexes into the parent levels B_1 and C_1 are both compressed, then the `while` loop header is:

```
while (pB2 < B2_pos[i+1] && pC2 < C2_pos[i+1) {
```

Emit access iterators

The next step is to emit code to access each iterator using its iterator `access` capability. This code defines and initializes the candidate coordinate variables of position iterators and the position variables of coordinate iterators. For a compressed iterator B_2 that partakes in iterating over the j index variable, the emitted access code is:

```
int jB = B2_crd[pB2];
```

Emit resolve the coordinate of i

Each iterator in the loop lattice point $\mathcal{L}_p$ produces a coordinate for each `while` loop iteration, either from its induction variable or from its `access` function. From these candidate coordinates, the generated code must compute a single **resolved coordinate** that is the coordinate of the current `while` loop iteration. Since the lattice point iterators are guaranteed to be ordered the same way, we choose the smallest of the iterator coordinates as the resolved coordinate for this iteration. The resolved coordinate is named after the current forall's index variable, and any segment whose coordinate is equal to the resolved coordinate is called an **active segment**. The emitted C code to resolve the current coordinate when coiterating over `ib`, `ic`, and `iD` is:

```
int i = min(ib, ic, iD);
```

Emit locate from locators

With the resolved coordinate computed, the next step is to retrieve its position in every locator segment in the lattice point by calling the locator's `locator` level function. Following this step, the emitted code

has computed the current coordinate and position of every active segment. The emitted C code to locate the position of a dense segment B_2, where m is the size of the i dimension, is:

```
int pB2 = (i * m) + j;
```

Emit conditional header

The code generation algorithm emits one `if` statement for every lattice point $\mathcal{L}_q$ underneath the loop lattice point $\mathcal{L}_p$ in the lattice structure (i.e., $\mathcal{L}_q < \mathcal{L}_p$). These `if` statements check which iteration domain region/intersection the resolved coordinate lies in by testing what segments have that coordinate and are therefore active. Since the lattice points are partially ordered, we can nest these `if` statements so that we test only whether the coordinate is in the region described by $\mathcal{L}_q$ if it is not in any of the regions described by lattice points above (greater than) $\mathcal{L}_q$. Thus, we emit an **`if`** statement for the first lattice point and an **`if`** statement nested inside an else statement for the others. The `if` statement's test checks whether each segment in the `if` statement's lattice point are at the resolved coordinate by comparing the segment's coordinates with the resolved coordinate. The emitted nested C **`if`** statements for the sub-lattice in Figure 5-3, whose lattice points are underneath $\mathcal{L}_p$ is:

```
if (ic == i && id == i && ie == i) {
// ...
}
else if (id == i && ie == i) {
// ...
}
else if (ic == i) {
// ...
}
```

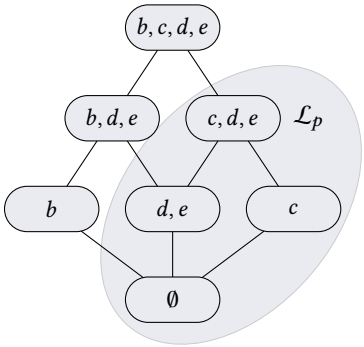


Figure 5-3: The sublattice of the lattice points below $\mathcal{L}_p$ whose lattice points result in `if` statements.

Emit advance iterators

The final step is to emit code to advance the lattice iterator induction variables. An iterator induction variable is advanced if its coordinate was consumed in the current iteration, meaning the coordinate value is equal to the resolved coordinate. The emitted C code to advance the position induction variable of a compressed segment B_2 is:

```
if (iB == i) pB2++
```

This code can be replaced with an optimized variant that avoids an `if` statement:

```
pB2 += (int)(iB == i);
```

A dense segment results in an increment of its coordinate induction variable. Since a dense segment has every coordinate (is full), we do not need to test whether it contains the resolved coordinate. Therefore, the emitted C code for a dense segment c_1 is just:

```
ic++;
```

As described in Section 3.2, a collapsed index variable iterates over the Cartesian combination of the coordinates of more than one original index variables. As shown in Figure 5-4, this means the the code iterates through and across all the segments of the innermost index variable and tracks the coordinate values of outer index variables. For each iteration step, when the generated code advances the inner variable, it must also advance the outer variables. This involves checking whether the inner variable has reached the end of its segment, if so, advancing the outer variable. Since the inner variable's data structure may have empty segments, meaning the outer variable has coordinates with no children, the emitted code must perform the segment check in a while loop. For example, the collapsed iteration shown in Figure 5-4 results in a single loop that iterates over the inner index variable j. When generating code to advance the (single) iterator of this loop, the code generator also generates code to advance the iterator over the outer index variable i if j has reached the end of a segment. Assuming the coordinate tree is stored in a CSR matrix B then the code to advance i is:

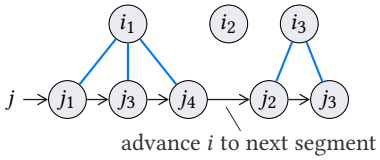


Figure 5-4: The collapsed index variable j iterates over the Cartesian combination of i and j coordinates by iterating over the bottom level of the coordinate tree. The variable j is advanced at each step, and the variable i is advanced at each segment boundary, to the next coordinate with a nonempty segment.

```
while (pB2 == B2_ptr[i+1]) i++;
```

Ranger and Merger Optimization

We can optimize the above code generation by dividing the iterators of each loop lattice point $\mathcal{L}_p$ into two sets called rangers and mergers. The **ranger iterators** are the iterators we iterate over until one runs out of values in the loop header emitted in Step (2), while the **merger iterators** are the iterators we combine to compute the resolved coordinate in Step (4). These sets both start out with the full set of iterators—and it would be correct to leave them like that—but we can potentially optimize the generated code by removing iterators from either set. We can remove an iterator from the ranger iterators if it is guaranteed to not be exhausted before the other rangers, which holds if the largest coordinate of the iterator is smaller than or the same as the largest coordinate of the other iterators. A common case is when the iterator is a subset of the other iterators, such as when one of the other iterators is full. We can remove an iterator from the merger iterator if it is a superset of the other iterators, since it will then always have the resolved coordinate. This condition holds when the iterator is full.

```
for (int i=0; i<m; i++) {
  // ...
}
```

Figure 5-5: A for loop generated from a lattice that has a single dimension iterator. No conditions are needed and the loop induction variable is advanced in the loop header.

Unary Loop Simplification

A lattice with a single iterator can be emitted as a `for` loop instead of `while` loops. A `for` loop improves code readability, makes it easier for compilers to analyze the code, and makes it possible to parallelize the loop with OpenMP [45] and Cilk [32]. Figure 5-5 shows the `for` loop of a lattice with a single dimension iterator, and Figure 5-6 shows one

```
for (int pB2 = B2_pos[i];
         pB2 < B2_pos[i+1];
         pB2++) {
  // ...
}
```

Figure 5-6: A for loop generated from a lattice that has a single position iterator. No conditions are needed and the loop induction variable is advanced in the loop header.

for a lattice with a single compressed iterator. The loop bodies are the code generated by recursive calls to CODEGEN in Figure 5-2.

5.3 Derived Iteration Spaces

Coordinates from different derived iteration spaces must be mapped to the same coordinate space so that they can be combined to compute the resolved coordinate. For example, consider an iteration graph

$$\forall_{i_0} \forall_{i_1} b_i \cap c_i : i \xrightarrow{\text{split}(\downarrow,4)} i^0 i^1,$$

where the original index variable i, with universe $[0, n)$, has been split into two index variables, i^0 and i^1. The universe of i^0 is the range $[0, n/4)$, and its domain is the same as its universe: $i^0 \in [0, n/4)$. The universe of i^1 is the range $[0, 4)$; however, its domain is the same as i:

$$i^1 \in b_1 \cap c_1,$$

The coordinates of b_1 and c_1 are in the universe of i, which we cannot change without transforming the data structure. Since all the coordinates must be resolved in the same space, the code generation algorithm emits code to map between coordinate spaces.

Figure 5-7 illustrates the situation when the outer index variable i^0 indexes the third block. The figure shows the sparse iteration spaces of i and i^1, and the iteration space of i is divided into tiles that are indexed by i^0. The inner index variable i^1 iterates through the coordinates in each tile by coiterating over the coordinate levels b_1 and c_1. The iteration strategy first maps every coordinate to the original iteration space of i, and it combines them there to compute the resolved coordinate. It then maps the resolved coordinate back, through any intermediate derived spaces, to the iteration space of i^1. For each step in the mapping the resolved coordinate back through a split relation, the iteration strategy checks that the resolved coordinate is inside the universe of the inner index variable (the tile). If it is outside that universe, then the iteration over i^1 exits and i^0 moves on to the next block.

Emit map candidate coordinates to the original space

The algorithm emits code to map every candidate coordinate variable in the index variable's domain to the original space. The algorithm tracks backward through the relations from the inner index variable of the derived candidate coordinate variable. For every split relation, it creates a new coordinate variable whose value is the sum of the value of its derived inner variable and the total size of the blocks preceding. This size is found by multiplying the inner index variable's universe by the current resolved coordinate value of the outer variable. For example, if in the above example we had accessed a temporary tensor t with the inner variable i^1, then `it` can be calculated from `i1t` as

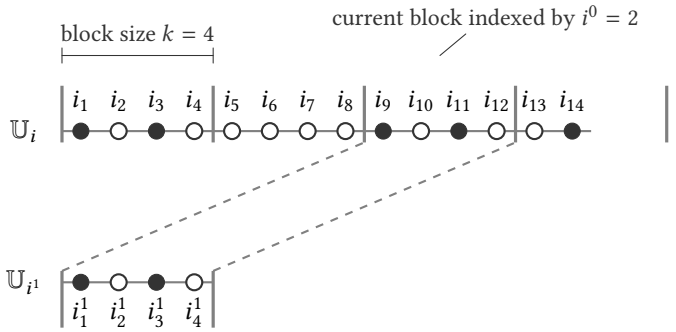


Figure 5-7: Coordinates in derived iteration spaces are mapped back to the original space, so that they can be combined to compute the resolved coordinate. This figure shows the mapping for the iteration graph $\forall_{i_0}\forall_{i_1} b_i \cap c_i : i \xrightarrow{\text{split}(\downarrow,4)} i^0 i^1$ when $i^0 = 2$.

follows:

```
int it = i0*4 + i1t;
```

And if the variable `it` in turn derived from another variable through a split, then a similar statement would map `it` to this variable.

Emit map resolved coordinate to each derived space

The second mapping step comes after the candidate coordinates in the original space have been combined to compute the resolved coordinate. In this step, the resolved coordinate is mapped back to each space that is involved in the fully derived inner index variable's domain. We need the resolved coordinate mapped back to each space for three reasons. First, the emitted code must check whether it is outside any of the derived inner loop's universes; in other words, whether it is outside any of the tiles. Second, any of the derived index variables might have been used to locate into a temporary tensor, and we need the resolved coordinate in that space to pass to the locate call. Third, we need the resolved variable in each space in the region tests in the **`if`** statements and in the advancement tests at the bottom of the loop.

The process to map the resolved coordinate to each space proceeds in steps. In the first step, it is mapped to the derived resolved coordinate of the inner index variable following the first split. Next, that resolved coordinate is mapped to the first inner index variable of the split that follows it, and so forth. After each resolved coordinate is mapped to its derived inner variable, a tile check tests whether coordinate landed outside the universe of the inner index variable. If so, the emitted code breaks out of the inner index variable's loop. In this section's running example, the emitted code to map the index variable i to its derived inner variable $i1$ and to perform a tile check is:

```
int i1 = i - i0*4;
if (i1 >= 4) break;
```

5.4 Compute and Assembly

The green emit statements in Figure 5-1 generate code to compute result values and to assemble result coordinate index structures. The computed results, coordinates, and positions that make up the result's tensor data structure can either be appended to these data structures or randomly inserted into them. The code generator supports only insertion into result coordinate levels that support the insert capability. It is an error to attempt to generate code from a concrete notation statement that scatters into a result that does not support insert, as seen by the presence of compound assignments (e.g., +=).

Emit compute code

In the concrete index notation, assignment statements compute results. The assignment lower function generates imperative compute code by traversing the concrete assignment and generating equivalent imperative expressions that compute scalar values. Each concrete access expression is rewritten to an array load or store expression indexed by the position variable of the last level of the accessed tensor's coordinate tree.[30] To give an example, the generated code for the assignment statement in

$$\forall_i \forall_j \forall_k \; A_{ij} \mathrel{+}= B_{ik} C_{kj},$$

where A and B are stored in the CSR and C in the CSC format, is

```
A[pA2] += B[pB2] * C[pC1];
```

The first forall loop iterates over the universe of i, while the second and third forall loops coiterate over the coordinate trees indexed by j and k. The iteration code generates one position variable for each coiterated or located data structure. The last coordinate level of A and B are the second tensor modes as they are stored row-major, while the last level of C is the first mode as it is stored column-major. The generated code uses the corresponding position variables to index into the array access expressions.

[30] An optimization is to treat scalar tensors as a special case that are stored in a variable and not loaded from an array.

Emit assembly code

The levels of the coordinate tree index data structure of the result tensor encode coordinate-position pairs. The lowering algorithm emits coordinate-position insertion code into the body of loops that iterate over free index variables. The pairs are inserted after the recursive code generation call for the body of the forall statement, and an optional optimization checks if any nonzero values were produced in the body's code and only then inserts the coordinate.

For example, the generated code for the i forall in

$$\forall_i \forall_j \; A_{ij} = B_{ij} C_{ij},$$

where all matrices are doubly-compressed row-major DCSR, is

```
A2_crd[pA2++] = j;
A2_pos[pA1+1] = pA2;
```

The code stores the resolved `i` coordinate to the current position of the `crd` index array of the compressed level `A2`. Since this kernel appends to the result, it then increments the position variable `pA2` to be ready to insert the next coordinate. It then increments the segment size to record that it has one more coordinate.[31] The array `A2_crd` is indexed by the position variable `pA2` of the second level, while the array `A2_pos` is indexed by the position variable `pA1` of the previous level.

[31] An optimization is to hoist the code that records the segment size to after the loop body, where the size of the whole segment can be stored once.

5.5 Conclusion

In this chapter I showed how to compile concrete notation to imperative code, using the concepts of iteration graphs and iteration lattices. The code generation algorithm recurs on the concrete index notation and turns forall statements into imperative `while` loops, by turning concrete notation to iteration graphs and then to iteration lattices. The algorithm also generates code that coiterates over different iteration spaces—original and derived spaces—and emits code to map between them. Finally, the algorithm emits code inside these loops that compute tensor results and assemble result data structures.

This algorithm is the first code-generation algorithm to compile the full sparse tensor algebra to sparse imperative code, parallel hardware, or accelerators. It also unifies sparse and dense tensor algebra by showing that both can be framed as loops that coiterate over data structures—compressed or dense—and coordinate universes. With a sparse tensor algebra compilation algorithm in hand, we can address the sparse composition problem I described in the Introduction. We can avoid asymptotic slow-down and poor temporal locality by compiling compound tensor algebra expressions to fused code that computes the whole expression at once. And we can avoid data structure reorganization and store tensors in the format that best fits them by compiling expressions to the data structures at hand. The code shapes to data, so data may rest.

Now that we have a code generator for sparse operations, we can start to expand its reach to encompass sparse operations beyond tensor algebra, like graph and mesh operations, robotics, and relational algebra. We can also explore sparse compilation to distributed systems and a plethora of interesting sparse data structures. And we can develop specialized computer architectures for sparse computations with some confidence that we will be able to compile sparse operations to them. That is, portable compilation of sparse operations is now in reach.

In the next chapter, I show how to optimize the concrete notation statements we just learned how to compile. And in Section 7.5 of the Evaluation chapter, I show empirical results demonstrating that the generated code performs on par with hand-optimized implementations

in sparse linear and tensor algebra libraries. Furthermore, Section 7.2 shows empirical results that demonstrate why support for compiling compound expressions matters. Finally, the Related Work chapter ties it all to the history of compilers and programming models for sparse and dense computations. It shows that this dissertation presents the first compiler for the entire sparse tensor algebra and the first comprehensive theory of sparse iteration spaces, their optimizing transformations, and their compilation.

"If a problem has no solution, it may not be a problem, but a fact—not to be solved, but to be coped with over time."

— Shimon Peres

Chapter 6

Optimizing Transformations

Optimizing transformations, which change iteration order to use caches and parallel hardware efficiently and which move computation to reduce work, are important for dense and sparse codes alike. But these transformations have additional effects on sparse codes. They can replace data structure accesses with asymptotically cheaper accesses, simplify coiterating code, and even create static load balance.

Concrete notation sits between the declarative index notation and the detailed imperative IR. It is operational, and it specifies both iteration order and where temporary results are stored. But it does not specify the low-level details of how to coiterate over compressed data structures—`while` loops, `if` statements, and indirect memory accesses. This makes concrete notation a great target for transformations that alter iteration order and introduce temporaries, as there is no need for sophisticated data-flow, dependence, or alias analysis. In Chapter 5, we saw how statements in concrete notation can be lowered to imperative coiterating code. This chapter shows how concrete notation can be optimized before lowering.

In the design I propose, a sequence of transformations called a **schedule**[32] are specified through a **scheduling API** and applied to concrete index notation (Figure 6-1). Specifying these scheduling transformations through a clean API separates the policy (what to do) from the mechanism (how to do it). The separation of policy and mechanism is an important design principle whose roots go back to early operating systems [62, 134]. In an optimizing compiler, the policy says what optimizing transformations to apply and in what order, while the mechanism applies them and generates code. Separating policy from mechanism—by a script [39] or an API [107]—leads to modular systems where they can be developed independently. Because a policy tends to change faster than the mechanisms, it is important to make the policy easy to replace without the need to also redevelop the mechanisms [108]. This dissertation describes how to design sparse tensor algebra mechanisms that apply schedules and generate code. The schedules can be specified by the user code or by some automatic scheduling system, but I leave the design of such systems as future

6.1 Reorder
6.2 Precompute
6.3 Collapse
6.4 Split
6.5 Bound
6.6 Iteration Space Mapping
6.7 Conclusion

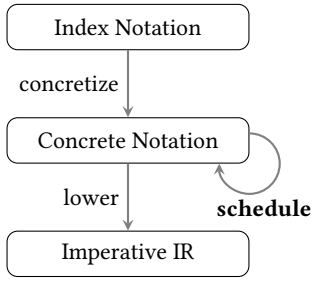


Figure 6-1: Compiler overview that highlights the schedule step that applies optimizing transformations described in this chapter.

[32] The convention to use schedule to mean a sequence of transformation comes from the Halide language [107], while the term itself comes from the polyhedral model literature where it refers to mapping statement instances in a polyhedral iteration domain to ordered execution instances [53]. The idea to specify a sequence of transformations from the outside comes from the CHiLL compiler [39].

work. That work, however, can now be built on top of the code generator presented here.

There are two types of transformations in this chapter: those that transform the iteration space by moving, adding, or removing index variables, and those that tag index variables with information about how they should be mapped to imperative IR. There are five iteration space transformations (reorder, precompute, collapse, split, and bound) and two mapping transformations (parallelize and unroll). These transformations are designed to preserve correctness, and I give preconditions that report an error if applying them changes the meaning of a concrete notation statement.[33]

The sparse iteration space transformations—reorder, precompute, collapse, split, and bound—transform concrete notation. They are verbs, and I described the nouns they act on in Section 3.2 and Section 4.3. I leave descriptions of the generated sparse code to those sections and focus in this chapter on the transformations and their preconditions.

[33] I assume that floating-point operations are associative, although it is not entirely true since reassociating floating-point numbers can give different results. The new results, however, are not any more wrong in principle.

6.1 Reorder

The reorder transformation takes a concrete statement S and two nested index variables i and j as arguments:

$$\text{reorder}(S, i, j).$$

It then rewrites S to move the forall statement $\forall_j$ of j past $\forall_i$ of i:

$$\ldots \forall_i \ldots \forall_j T \implies \ldots \forall_j \ldots \forall_i T,$$

where T is a statement. The forall statements may be directly nested or $\forall_j$ may be nested inside other statements, such as the consumer side of a where statement. The reorder transformation moves $\forall_j$ past $\forall_i$ in a sequence of steps, moving it past each statement on the way. The supported cases and their preconditions are:

1. The forall statement $\forall_j$ is directly nested inside another forall statement: $\forall_k \forall_j \ T$, where k may be equal to i. The $\forall_j$ statement can be moved past $\forall_k$ if the operator $\oplus$ of any compound assignment statement in T is associative.[34]

2. The forall statement $\forall_j$ is the producer of a where statement: $(C \textbf{ where } \forall_j \ P)$. Let t be a temporary tensor on the left hand side of a compound assignment statement A with operator $\oplus$ in P. The $\forall_j$ statement can be moved out of the where statement if, for every t in P, every operator $\odot$ that combines t with other expressions in C distributes over $\oplus$.

[34] Recall that a compound assignment is an assignment that combines the right hand side with the current value of the variable on the left hand side. For example, += *or* −=. The compound operator of a normal assignment statement = has no effect, and it is therefore both associative and commutative, and it distributes over every other operator.

The sparse matrix-vector multiplication (SpMV) is sufficient to demon-

strate both cases. In index notation, SpMV is expressed as

$$a_i = \sum_j B_{ij} c_j.$$

The simplest concrete notation statement that computes SpMV is

$$\forall_i \forall_j \; a_i \mathrel{+}= B_{ij} c_j,$$

which implements the inner product algorithm. We can reorder the i and j loops in this statement to turn it into a linear combination of columns algorithm, with the resulting expression:

$$\forall_j \forall_i \; a_i \mathrel{+}= B_{ij} c_j.$$

This transformation is legal because the + operator of the compound assignment is associative. Suppose we started out with an inner product concrete statement that accumulated the partial results into a temporary scalar, to avoid address calculations. The reorder algorithm applies a sequence of transformations that result in the following intermediate statements:

$$\begin{aligned}
&\forall_i \big(a_i = t \;\; \textbf{where} \;\; \forall_j \; t \mathrel{+}= B_{ij} c_j\big), \\
&\forall_i \forall_j \big(a_i = t \;\; \textbf{where} \;\; t \mathrel{+}= B_{ij} c_j\big), \\
&\forall_j \forall_i \big(a_i = t \;\; \textbf{where} \;\; t \mathrel{+}= B_{ij} c_j\big), \text{and} \\
&\forall_j \forall_i \; a_i \mathrel{+}= B_{ij} c_j.
\end{aligned}$$

The first step applies the second reorder case to move $\forall_j$ outside the where statement, and the second step applies the first case to move $\forall_j$ past $\forall_i$. The third step is a simplification step that is applied after every transformation to remove where statements that do nothing.

To demonstrate how the precondition of the second reorder case can fail, I extend the SpMV example to add a vector:

$$a_i = \big(\sum_j B_{ij} c_j\big) + d_i.$$

This is a common expression in sparse linear algebra libraries, and it can be implemented by the inner product algorithm with a temporary variable t:

$$\forall_i \big(a_i = t + d_i \;\; \textbf{where} \;\; \forall_j \; t \mathrel{+}= B_{ij} c_j\big).$$

The presence of d_i, however, prevents us from reordering $\forall_j$ and $\forall_i$ in this statement. The reason is that the + operator that combines t with d_i does not distribute over the compound assignment operator +=. Thus, moving $\forall_j$ out of the where statement causes d_i to be added to the result many times. If the operator was a multiplication, however, then the reordering would be fine. The multiplication distributes over +=, so applying it before or after accumulation makes no difference. Section 6.2 shows how to apply precompute and reorder in sequence

to rewrite this concrete statement to a linear combination algorithm, which can be useful if we are stuck with a column-major data structure.

Figure 6-2 shows the concrete notation statement and its iteration graph before and after reordering, assuming B is stored in row-major order. The reorder transformation does not take data structures into account, and it is possible to reorder forall statements so that they traverse up coordinate trees, which appear as back-edges in the iteration graph. When coordinate trees are stored with compressed data structures that do not support fast random access, such bottom-up traversals are asymptotically slow, as they require searching through levels.[35] And the iteration space transformations do not change the data layout of tensors, since this is the job of the format description language. Therefore, if you wanted an efficient linear combination of columns SpMV, for example, then you must store your data structure column-major. The user or an auto-scheduling system must, therefore, be aware of the storage of the tensor data structures and avoid lowering a concrete statement that traverses bottom-up a coordinate tree without random access. Such statements, however, are still legal concrete notation and are useful as intermediate scheduling steps.

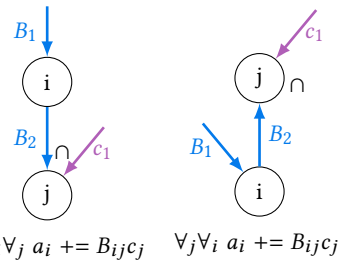


Figure 6-2: A concrete notation statement and its iteration graph before and after reordering $\forall_i$ and $\forall_j$.

[35] The lowering machinery in Chapter 5 does not support concrete notation statements with bottom-up traversals of coordinate trees without random access. Given a statement with a bottom-up traversal, the lowering machinery will report an error. A future version could add support for bottom-up traversals, but it is unclear whether they have any benefits.

6.2 Precompute

The precompute transformation takes a concrete statement S, a concrete expression e from S, a tensor t, and a sequence of index variable three-tuples $I = (i, i_c, i_p), (j, j_c, j_p) \ldots$:

$$\text{precompute}(S, e, t, I).$$

It then rewrites S by cloning the forall statements of the first index variables in each of the tuples, $\forall_i \forall_j \ldots$, into the two sides of a new where statement. The producer side precomputes e into a temporary tensor t, while the consumer side computes the original assignment statement with e replaced by t. Without loss of generality, assume there are only two index variable tuples $I = (i, i_c, i_p), (j, j_c, j_p)$ and that e is the leftmost expression. Then:

$$\begin{aligned} &\ldots \forall_i \forall_j\ a_{\ldots} \oplus\!= e \otimes f \implies \\ &\ldots \left(\forall_{i_c} \forall_{j_c}\ a_{\ldots} = t_{i_c j_c} \otimes f \ \textbf{where}\ \forall_{i_p} \forall_{i_p}\ t_{i_p j_p} \oplus\!= e\right), \end{aligned}$$

where $\oplus$ and $\odot$ are operators, e is an expression, and a is a result tensor. The forall statements of the index variables $i, j, \ldots$ in the original statement must be directly nested without any other forall statements in their body, and the operator $\otimes$ must distribute over $\oplus$. The transformation then does a three-step cleanup. First, it removes redundant forall statements and where statements. Second, it pushes forall statements outside the where statement, whose index variables are used only on one side, into that side. And third, it replaces where statements with sequence statements to reuse temporaries.

The precompute transformation has many uses, including enabling the reorder transformation, introducing temporaries that support random insert, simplifying costly coiteration code, hoisting loop-invariant work out of loops, and setting up efficient warp-level reductions on a GPU. In the following subsections, I show examples of some of these uses, to better illustrate the subtle effects of the precompute transformation and its cleanup step.

Facilitate Reorder

The first example shows how the precompute transformation can be used to work around the preconditions of the reorder transformation. Let us return to the SpMV plus a vector example from Section 6.1 and see how to turn it into a linear combination algorithm to compute with a column-major matrix:

$$a_i = \Big(\sum_j B_{ij} c_j\Big) + d_i.$$

The concrete statement for this expression produced by the concretization algorithm implements the inner product algorithm:

$$\forall_i \big(a_i = t + d_i \ \textbf{where} \ \forall_j \ t \mathrel{+}= B_{ij} c_j\big).$$

We wish to reorder the loops to iterate over B in column-major order. But the reorder algorithm cannot move $\forall_j$ out of the where statement, because the operator (+) that adds t to d_i does not distribute over the compound assignment operator (also +). To facilitate the reordering, we can apply the precompute transformation to precompute $t + d_i$ over i. This transformation, following the simplification step that removes a redundant where statement, yields the concrete statement

$$\big(\forall_{i_c} \ a_{i_c} = t_{i_c} + d_i \ \textbf{where} \ \forall_{i_p} \forall_j \ t_{i_p} \mathrel{+}= B_{i_p j} c_j\big).$$

We can now reorder $\forall_{i_p}$ with $\forall_j$, as the operation in the new statement takes place inside the producer side and does not need to move the j loop out of the where statement. The result is a linear combination algorithm that traverses B column-major:

$$\big(\forall_{i_c} \ a_{i_c} = t_{i_c} + d_i \ \textbf{where} \ \forall_j \forall_{i_p} \ t_{i_p} \mathrel{+}= B_{i_p j} c_j\big).$$

The downside of this algorithm is that it introduces a temporary vector that consumes memory and reduces temporal locality. It should therefore be used only when the cost of the temporary vector is less than the cost of transposing B. Finally, we must chose a format for vector t. Since it is scattered into, we need a format that supports random insert, such as a dense vector or a hash map.

Scatter into result

This section shows how to use precompute to transform a sparse matrix-matrix multiplication (SpGEMM)

$$A_{ij} = \sum_k B_{ik} C_{kj},$$

where each matrix is in row-major CSR format, into the linear combination of rows algorithm from Section 4.1. The initial concretized statement is

$$\forall_i \forall_j \forall_k \; A_{ij} \mathrel{+}= B_{ik} C_{kj},$$

and it implements an inner product algorithm. The first step is to reorder $\forall_j$ and $\forall_k$, so that we access C in row-major order. The resulting concrete statement implements a linear combination algorithm:

$$\forall_i \forall_k \forall_j \; A_{ij} \mathrel{+}= B_{ik} C_{kj}.$$

This algorithm computes one row of A at a time by scaling the rows of C with values from B and then adding them together. The row addition can be done in place by adding each computed value directly into the row of A. This leads to scattered writes, which you can see from the concrete statement by observing that $\forall_j$ of j, which indexes A, is nested inside $\forall_k$, which does not index A. Therefore, the algorithm inserts into the second mode of A once for each k. The second mode, however, is compressed in the CSR format and does not support $O(1)$ random insert. To get fast random inserts, we can apply the precompute transformation to precompute the $B_{ik} C_{kj}$ subexpression over j into a result temporary vector t that supports random insert (e.g., a dense vector or a hash map). The resulting concrete statement, after the simplification step has pushed $\forall_k$ into the producer side of the where statement, is:

$$\forall_i \big(\forall_{j_c} \; A_{ij_c} = w_{j_c} \; \textbf{where} \; \forall_k \forall_{j_p} \; w_{j_p} \mathrel{+}= B_{ik} C_{kj_p} \big).$$

The precompute transformation moved the compound assignment onto the producer side, where it scatters into the temporary t instead of A. After a full row has been computed and stored in t, the consumer side simply appends the row to A, which can be done efficiently with a compressed format.[36] Figure 6-3 shows the effect of the transformation on the iteration graph of the concrete notation. The j index variable is cloned and, since only j_p has an incoming arrow from k, j_c moves up underneath i.

[36] The linear combination sparse matrix multiplication algorithm with a temporary array is often called Gustavson's algorithm after Fred Gustavson, who proposed it in 1978 [61]. The precompute transformation recreates it, but also generalizes the underlying optimization so that it can be used on other tensor algebra expressions.

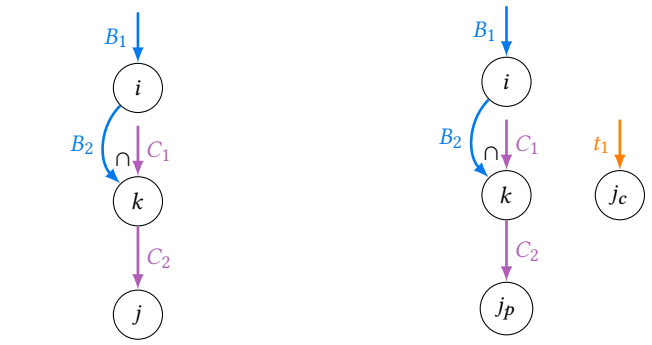


Figure 6-3: A concrete notation statement and its iteration graphs before and after precomputing $B_{ik}C_{kj}$ into t over j.

6.3 Collapse

The collapse transformation takes a concrete statement S and three index variables i, j, and f as arguments:

$$\text{collapse}(S, i, j, f).$$

It then rewrites S to collapse the two forall statements $\forall_i$ and $\forall_j$ into a single $\forall_f$, and it adds a collapse relation to its environment:

$$\ldots \forall_i \forall_j\ T \implies \ldots \forall_f\ T : ij \xrightarrow{\text{collapse}} f,$$

where T is a statement and a collapse relation describes the provenance of f. The collapse transformation requires that $\forall_i$ and $\forall_j$ are directly nested. The iteration domain of f is the Cartesian product of the domains of i and j. I described the collapse relation in Section 3.2. There I showed that when i and j iterate over two levels of a coordinate tree, the f iterates over their Cartesian product by iterating the bottom of those levels.

6.4 Split

The split transformation takes a concrete statement S, three index variables i, i_0, and i_1, a direction d, a split factor s, and an optional coordinate tree c as arguments:

$$\text{split}(S, i, i_0, i_1, d, s, c).$$

It then rewrites S to split $\forall_i$ into nested forall statements $\forall_{i_0}$ and $\forall_{i_1}$, and it adds a split relation to its environment:

$$\ldots \forall_i\ T \implies \ldots \forall_{i_0} \forall_{i_1}\ T : i \xrightarrow{\text{split}(d,s)} i_0 i_1,$$

where T is a statement and a split relation describes the provenance of i_0 and i_1. The outer forall $\forall_{i_0}$ iterates over a dense coordinate range, while the inner $\forall_{i_1}$ coiterates over whichever data structures the index

variable i coiterated over.

The bounds of the new index variables i_0 and i_1 depend on the direction, split factor, and coordinate tree arguments. The size of the domain of i is roughly equal to the size of the Cartesian product of the domains of i_0 and i_1.[37] The direction d—up or down—controls whether the domain of i_0 (up) or i_1 (down) has a fixed number of iterations, as determined by the split factor s. The domain of the other index variable will have a number of iterations determined by dividing the domain of i by s. That is, the up direction creates a loop nest that iterates over a fixed number of blocks the size of which is proportional to the size of i's domain, and the down direction creates a loop nest that iterates over fixed-size blocks, the number of which is proportional to the size of i's domain.

[37] The domain of the Cartesian product of i_0 and i_1 may be slightly larger than the domain of i if i's domain is not evenly divided by the split factor s. The additional iterations are removed by a tail strategy that changes the iteration over the last execution of i_1 to iterate over fewer values.

The optional coordinate tree argument c changes how the blocks of i_1 are determined. The index variable i_1 coiterates over the same data structures as i, within the dense coordinate range identified by the current value of i_0. These ranges are always the same size, but their size is with respect to different coordinate sets. When c is left out, the ranges are the same size with respect to the universe of i. But when c is given, the ranges are the same size with respect to the coordinates that are actually stored in c—a (non-strict) subset of the universe. The effect, as we saw when we discussed the corresponding derived index variable split relation Section 3.2, is to load-balance the computation with respect to that coordinate tree.

6.5 Bound

The bound transformation takes a concrete statement S, an index variable i, and a bound b as arguments:

$$\text{bound}(S, i, b).$$

It then rewrites S to add a constraint to the domain of i as specified by the bound. There are two types of bounds: upper bounds and strided bounds. Upper bounds restrict the domain's upper limit to a compile-time constant. Strided bounds restrict the domain's upper bound to be a multiple of a compile-time constant. Bound transformations give the compiler information about loop bounds that lets it unroll loops, vectorize them, or omit the tail strategy guards.

6.6 Iteration Space Mapping

The iteration space mapping transformations tag concrete forall statements with information about how they should be mapped to hardware features, such as CPU threads, GPU warps, and vector instructions, or to imperative code patterns, such as unroll. This information is then used by the lowering machinery to generate iteration code from

each forall statement.

Parallelize

The parallelize transformation takes a concrete index notation statement S, an index variable i, a parallel unit p, and a reduction strategy r as arguments:

$$\text{parallelize}(S, i, p, r).$$

It then rewrites S to add a tag to the environment, stating $\forall_i$ should be parallelized with the given parallel unit and the given reduction strategy:

$$\ldots \forall_i \; T \implies \ldots \forall_i \; T : \text{parallelize}(i, p, r),$$

where T is a statement and parallelize is a tag describing the mapping strategy of $\forall_i$. The parallelize tag changes the iteration order of an index variable and therefore requires every reduction in T to be associative and commutative. Furthermore, parallelize does not support forall statements that coiterate over more than one data structure. The parallelize transformation is therefore often combined with the split transformation, as an enabling transformation to create a dense outer loop that can be parallelized.

The parallel unit p specifies the parallel hardware feature to target the loop, such as CPU threads, GPU warps, or vector instructions. Some parallel units, such as GPU warps and vector instructions, can only be used on fixed-size loops. The split or bound transformations can be used to create fixed-sized loops to target. Supported parallel units are:

CPU Threads divides the loop's iteration domain between CPU threads.

CPU Vector Instructions divides the loop's iteration domain between lanes in a warp; it can be applied only to a dense inner loop.

GPU Threads divides the loop's iteration domain between GPU threads.

GPU Blocks divides the loop's iteration domain between GPU blocks; requires the loop to have an inner loop parallelized with GPU Threads and groups those threads into blocks.

GPU Warps divides a loop's iteration domain between GPU warps, requiring the loop to have an inner loop parallelized with GPU threads, and groups those threads into warps.

The reduction strategy r specifies how reduced values from compound assignment statements in T should be combined. Possible strategies include:

No Races asserts that there are no reductions in the loop; it raises a compile-time error if there are any.

Ignore Races states that reductions in the loop should be ignored. Any reduction will then be stored using regular store instructions at the cost of atomicity.

Atomics inserts atomic instructions or locks to ensure parallel writes are sequentialized.

Temporary is a shortcut for a precompute transformation to store the values to a temporary tensor with one component per thread to remove the reduction, followed by a serial reduction.

GPU Warp Reduction uses specialized warp-level reduction primitives on GPUs.

Unroll

The unroll transformation takes a concrete index notation statement S, an index variable i, and an unroll factor u as arguments:

$$\text{unroll}(S, i, u).$$

It then rewrites S to add a tag to the environment, stating that $\forall_i$ should be unrolled with the given unroll factor:

$$\ldots \forall_i \; T \implies \ldots \forall_i \; T : \text{unroll}(i, u),$$

where T is a statement and unroll is a tag describing the mapping strategy of $\forall_i$. The unroll transformation requires a fixed-size loop, and the split and bound transformations are therefore useful to facilitate the unroll transformation.

6.7 Conclusion

In this chapter, I described several optimizing transformations on concrete notation. Some of these transform the iteration space by reordering, fusing, and splitting index variables, while others introduce temporaries or map index variables to parallel hardware features. The transformations apply to dense and sparse loops alike, can tile sparse loops, and can even create statically load-balanced parallel sparse code.

I have described, for the first time, how to adapt these common dense loop transformations to sparse codes. It is thus the first comprehensive compiler transformation framework for sparse operations. Moreover, it also unifies compiler optimization for sparse and dense loops and iteration spaces. The transformations can form a starting point for much work on sparse code optimization. They are good targets for automatic sparse code optimizers and automatic scheduling systems. And their flexibility makes them suitable for transforming code to target the specialized sparse hardware that is likely to appear at high rates. Section 7.4 in the Evaluation provides empirical evidence for why these sparse transformations matter. And Section 7.5 shows

how they let us generate efficient sparse code for GPU accelerators and mixed sparse-dense operations that require tiling. These generated implementations are competitive with handwritten implementations in popular libraries.

This concludes the core of my dissertation on sparse tensor algebra compilation. I have described intermediate representations and algorithms that together provide, for the first time, a complete approach to compiling sparse linear and tensor algebra. The approach unifies sparse and dense computations by framing them both as coiteration over data structures. From here, we can apply these ideas more broadly to sparse operations beyond tensor algebra, such as relational algebra and graph operations. We can also start to explore interfaces between tensor algebra, relational algebra, and graph operations, as well as operator fusion across them. I, therefore, believe I have satisfied my thesis statement, and that sparse operations, like dense operations, are now on a solid compiler foundation.

"When you can measure what you are speaking about, and express it in numbers, you know something about it."

— Lord Kelvin

Chapter 7

Evaluation

7.1 Experimental Setup

7.2 Expressions Matter

7.3 Formats Matter

7.4 Optimizations Matter

7.5 Kernels are Competitive

The Cartesian combination of all possible expressions, formats, and architectures form a space with an infinite number of tensor algebra implementations. The focus of this dissertation has been on laying out ideas, intermediate representations, and algorithms to automatically generate these implementations. Figure 7-1 shows a part of the space, and the entire space can be expressed as

$$\text{expressions} \times \text{formats} \times \text{optimizations} \times \text{architectures},$$

where each factor represents the variants in each category (e.g., all possible expressions), and the expression comprises their Cartesian combinations. Each combination is a tuple

$$(\text{expression}, \text{formats}, \text{optimizations}, \text{architecture})$$

that describes an implementation that computes the expression, on the formats, on the given architecture, with the given optimizations.

Each factor represents a large number of variants, and many of these variants are necessary in practice.[38] In this chapter, I provide empirical evidence and arguments for the proposition that a large number of the expressions, formats, and optimizations are necessary in practice. Because the resulting practical space contains a large number of expressions—can implement any but not all— we need an automated approach to generate them. This dissertation supplies that approach.

[38] In fact, there are an infinite number of possible expressions, an exponential number of formats for a tensor of fixed order, and an unlimited number of possible architectures.

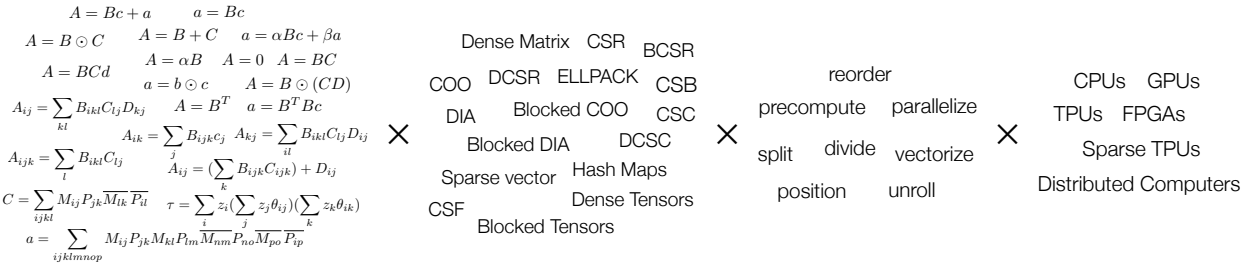


Figure 7-1: A subset of the combinatorial space of tensor algebra kernels described by the Cartesian combination of tensor algebra expressions, a format for each operand, target architectures, and sequences of optimizations.

Finally, in Section 7.5, I show that the Sparse Tensor Algebra Compiler (taco), described in Section 1.3, generates implementations that are comparable with hand-optimized implementations available in libraries such as Intel MKL [44], Eigen [60], MATLAB Tensor Toolbox [14], and SPLATT [116]. I focus on sparse linear/tensor algebra expressions where at least one operand is stored in a compressed data structure. Purely dense kernels, such as dense matrix multiplication (GEMM), have been explored extensively in prior work. I compare to libraries that each supports a subset of tensor algebra expressions, formats, architectures, and optimizations, whereas taco supports them all. Taco does not, however, do anything special for the expressions I evaluate in this chapter that does not generalize to other expressions. Therefore, by showing competitive performance to some of the hand-optimized implementations from state-of-the-art libraries, I provide evidence that the code generated by taco has good baseline performance across the board.

7.1 Experimental Setup

We ran the experiments on kernels generated by two different versions of taco: one that supports schedules and one that supports level formats beyond dense and compressed ones. These versions of taco are not merged yet; hence, kernels that use other level formats are not scheduled.

Most experiments use sparse matrices from the SuiteSparse matrix repository [47]. Many experiments show scatter plots with results for 1684 matrices, which include all real-valued SuiteSparse matrices except ones for which the number of nonzeros is too large to be indexed by a 32-bit integer.[39] Other experiments show bar charts with results for subsets of the real-valued SuiteSparse matrices. For example, the DIA experiment uses only SuiteSparse matrices that have few diagonals and that therefore benefits from this format. For experiments with higher-order kernels, we use tensor from the FROSTT sparse tensor repository [117]. Finally, we use synthesized matrices and tensors in some experiments and will describe the rationale in the text.

[39] We removed this matrix to save time because running experiments with it would have required us to recompile all the compared libraries to use 64-bit indexing. Note that a compiler-backed library such as taco can be extended to support custom bit-widths for all indices and to compile kernels accordingly.

Some CPU experiments in this chapter are new, whereas others were taken from the papers we have published on taco. Unless clearly stated in figure captions, the experiments are new and were run on dual-socket machines with Intel Xeon E5-2680 v3 CPUs running at 2.5 GHz with 12 cores, 24 threads, 30 MB L3 cache per socket, and 128 GB of main memory. The machines ran Ubuntu 18.04.3 LTS. We compiled the kernels with Intel ICC (version 19.1.0.166) using the `-O3`, `-DNDEBUG`, `-march=native`, `-mtune=native`, and `-ffast-math` compiler flags.

We ran all GPU experiments on a NVIDIA DGX machine with 8 V100 GPUs with 32 GB global memory, 6MB L2 cache, 128KB L1 cache per SM (80 SMs), and a bandwidth of 897 GB/s. We compiled the CUDA kernels with NVCC v9.0 with the `-O3` and `--use_fast_math` flags.

We ran each new experiment 25 times and report the median time. We cleared caches before each run; therefore, reported times are with cold caches. We used `numactl` to limit execution to one socket because taco does not yet support NUMA-aware code generation. We did not time data transfers between CPU and GPU memories.

The experiments adapted from our papers cite the appropriate paper in the figure caption. Those taken from the original paper on the taco compiler [78] were run on a two-socket, 12-core/24-thread 2.4 GHz Intel Xeon E5-2695 v2 machine with 32 KB of L1 data cache, 30 MB of L3 cache per socket, and 128 GB of main memory, running MATLAB 2016b and GCC 5.4. We report average cold cache performance (i.e. with the cache cleared of input and output data before each run), and results are for single-threaded execution unless otherwise stated. Multi-threaded results were obtained using 12 threads and using `numactl` to limit execution to one socket.

The experiments adapted from our paper on format extensions to the taco compiler [41] were run on a two-socket, 12-core/24-thread 2.4 GHz Intel Xeon E5-2695 v2 machine with 30 MB of L3 cache per socket and 128 GB of main memory, using GCC 5.4.0 and MATLAB 2016b. We ran each experiment between 10 (for longer-running benchmarks) to 100 times (for shorter-running benchmarks), with the cache cleared of input data before each run, and report average execution times. All results are for single-threaded execution.

7.2 Expressions Matter

In this evaluation, I describe two experiments to show why it is important to support many expressions in linear or tensor algebra systems, and hence the necessity of a compiler. There are two reasons why the number of possible expressions grows very large: a tensor can have any number of modes, and a compound expression can consist of many subexpressions. The alternative to a code generator or interpreter if we wish to support many expressions, is to convert them to the finite number of implementations we have available to us. This is the strategy taken by general programming systems such as MATLAB and CTF. Using this strategy, programming systems turn high-order tensors and tensor operations into matrices and matrix operations. They also divide compound expressions into binary or tertiary subexpressions for which they have implementations. The limitation with converting high-order expressions to matrix expressions is that converting tensors into matrices requires expensive data transformations. And the limitation with dividing compound expressions into binary subexpressions is that their composition may suffer from poor temporal locality or even have asymptotically worse performance than a fused implementation of the compound expression. In this section, I address the potential benefits of general support for compound expressions.

I described why the composition of separately implemented sparse expressions may suffer asymptotic slowdown from in Section 1.2. This is a serious issue for a library implementer because it means that we cannot simply implement the binary expressions and then compose these implementations to compute compound expressions without serious performance implications. I empirically demonstrate this effect using the SDDMM expression

$$A = B \odot (CD),$$

where A and B are square $n \times n$ sparse matrices, C is an $n \times k$ dense matrix, D is a $k \times n$ dense matrix, and $\odot$ is an element-wise operation. The asymptotic complexity of the fused version of this expression is

$$\text{nnz}_B \cdot k,$$

where nnz_B are the number of nonzeros of the matrix B. But the asymptotic complexity of the expression when computed as two separate operations $T = CD$; $A = B \odot T$, is

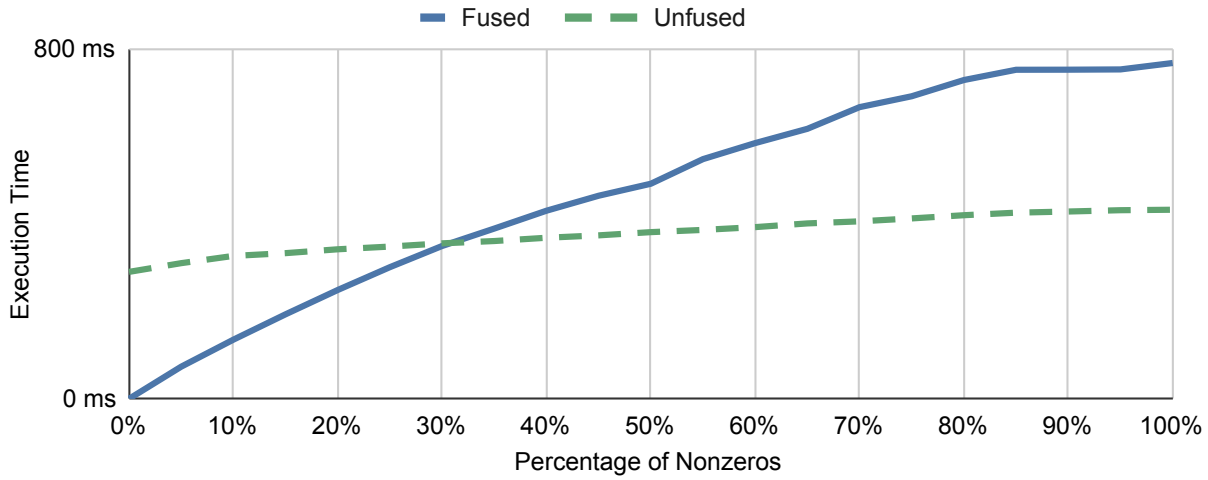
$$n^2 k.$$

These asymptotes grow at different rates and when the matrix B is sparse, then $nnz_B << n^2$.

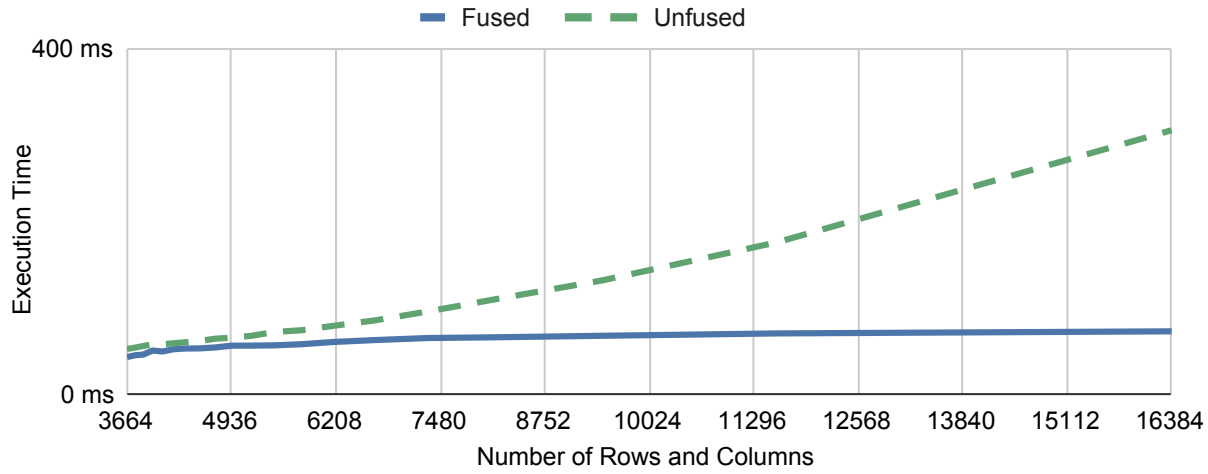
Figure 7-2 shows empirically the difference between the fused and unfused SDDMM implementations. Figure 7-2a plots their execution time as a function of nnz_B, whereas Figure 7-2b plots their execution time as a function of n. As shown in Figure 7-2a, the fused implementation is agnostic to the size of B and grows as a function of the necessary computations represented by its nonzeros. The unfused implementation performs, however, every operation in the dense multiplication whether or not they will be used, and it therefore only needs to perform one additional scalar multiplication for each new nonzero. Figure 7-2b shows what happens when we keep the number of nonzeros fixed and instead increase the matrix dimensions n. The fused implementation is unaffected, as the amount of necessary work does not change, whereas the unfused implementation performs increasingly more needless work.

7.3 Formats Matter

In this evaluation, I describe two experiments that show why it is important to support many sparse matrix or tensor data structures (formats) in linear or tensor algebra systems. These experiments provide further evidence for why we need a compiler. Two reasons for why we may want to choose a particular format for a tensor are that it is a good fit for the nonzero structure of the tensor, and that we want to avoid an expensive data structure reorganization. I show two experiments



(a) SDDMM performance as a function of number of nonzeros. The experiment uses a fixed-size 16,384 × 16,384 sparse matrix B with a varying percentage of nonzeros that varies along the x axis. The matrix B is then multiplied by the product of a 16,384 × 128 dense matrix C and a 128 × 16,384 matrix D. The fused implementation's performance is a function of nonzeros, while the unfused implementation is dominated by the fixed dense multiplication cost. For number of nonzero percentages below 30%, the unfused implementation performs better because it uses a cache-optimized dense matrix multiplication (GEMM) routine.



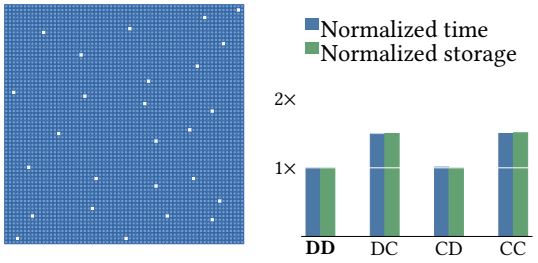
(b) SDDMM performance as a function of dimensions. The experiment uses a sparse square matrix B with a fixed number of 13,424,896 nonzeros, whose dimensions vary along the x axis. The fused implementation's performance is a function of the number of nonzeros and remains flat as the matrix dimensions increase, while the unfused implementation performs increasingly larger dense matrix multiplications.

Figure 7-2: Execution time of fused and separated implementations of Sampled Dense-Dense Matrix Multiplication (SDDMM), $A = B \cdot (CD)$, where A and B are sparse, whereas C and D are dense. Two scaling plots show execution time as functions of (a) the number of nonzeros in B and (b) the dimension sizes of B.

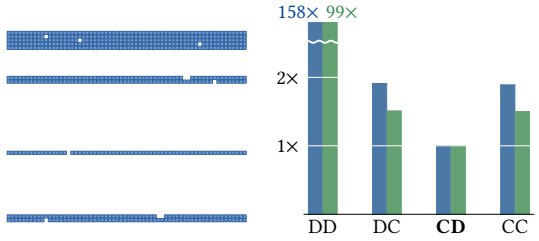
that demonstrate these two reasons.

Fit the format to the tensor

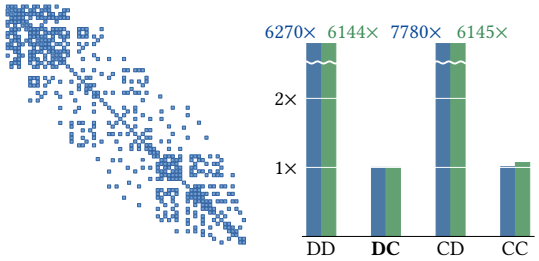
The first experiment shows the benefits of choosing a format that is a good fit for the nonzero structure of the tensor. Figure 7-3 (a–d) shows four matrices with different nonzero structures that appear in real-world applications. Each figure also shows the normalized SpMV execution time and normalized storage size for the matrix stored in each of the row-major formats from combining one dense and one compressed dimension. Each format is best for one of the matrices.



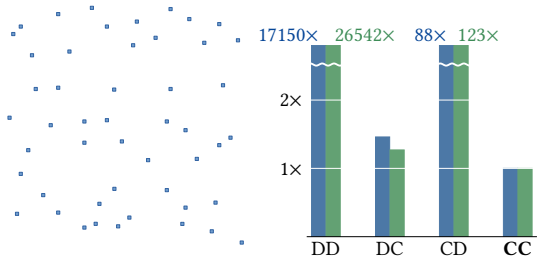
(a) A dense matrix.



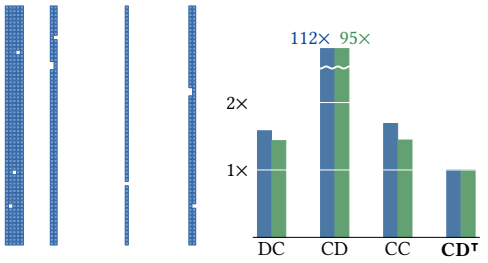
(b) A row-sliced matrix, with a sparse set of dense rows.



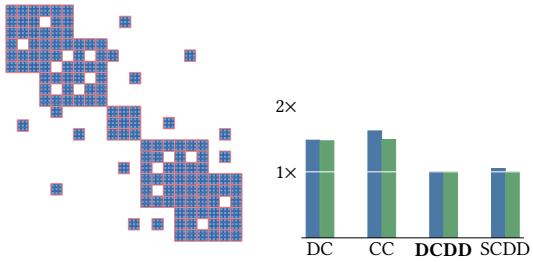
(c) A thermal matrix, where rows are sparse.



(d) A hypersparse matrix, where rows are sparse or empty.



(e) A column-slided matrix, with a sparse set of dense columns.



(f) An FEM matrix, where nonzeros come in 3×3 blocks.

Figure 7-3: SpMV execution time and storage size of six matrix stored in different formats. Each format is best for one matrix class and no format fits all matrices. The left half of each subfigure depicts the sparsity pattern of the matrix, while the right half shows the normalized average execution times (relative to the optimal format) of matrix-vector multiplication and the normalized storage costs using the storage formats labeled on the horizontal axis to store the matrix. The storage format labels describe the formats by describing their per-mode level types (D means dense and C means compressed). The dense matrix input has a density of 0.95, the hypersparse matrix has a density of 2.5×10^{-5}, the row-slicing and column-slicing matrices have densities of 9.5×10^{-3}, and the thermal and blocked matrices have densities of 1.0×10^{-3}. The SpMV kernels in this experiment are unscheduled—blocking the dense kernel would give better dense performance but not affect the conclusion. This figure was adapted from our original paper on the taco compiler [78].

And some of the formats are highly unsuitable for some of the matrices. Figure 7-3 (e) further demonstrates this point by showing that the transposed version of Figure 7-3 (b) benefits from a transposed version of the CD format. Finally, Figure 7-3 (f) goes further and shows that a blocked FEM matrix benefits from a blocked format.[40] These experiments show that the smallest storage size of a matrix and the best performance when computing with it can be achieved by choosing the format that best expresses the nonzero structure of the matrix.

[40] A blocked matrix format is equivalent to a higher-order tensor format. In this FEM matrix, a 4-tensor suffices to express the natural blocking, but further blocking can be achieved by adding more tensor modes.

Fit the code to the format

The second experiment illustrates the benefits of being able to compute with a given format (data structures). Often, libraries are used together, and it is important that they compose well. In Chapter 1, I mentioned the potential performance cost of composing libraries that require different data structures due to the cost of reorganizing data. It is therefore useful to have software components that can adapt to a given format so that data structures are only reorganized when that benefits performance. Assume that we need to compute an SpMV, but the matrix is given to us from another component in the COO format. We can compute the SpMV on the COO format data structures, or to pay a price to convert it to a more compressed CSR data structure to accelerate the SpMV. Figure 7-4 shows the cost of the CSR SpMV operation on ten matrices from the SuiteSparse repository when the cost of conversion from COO is included. The y axis is the SpMV execution time in multiples of COO. A CSR SpMV (blue) on these matrices is faster than a COO SpMV (the black line). But when conversion cost is included (red), the COO SpMV becomes faster. Unless the conversion cost can be amortized across multiple operations, we are better off computing with the COO matrix.

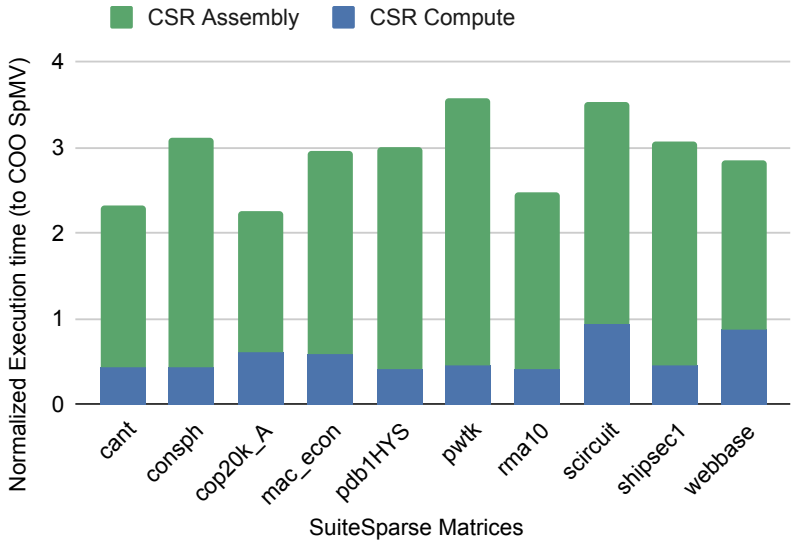


Figure 7-4: Execution time of CSR as a multiple of the execution time of COO, including the cost of converting the CSR matrix from an initial COO representation. The result shows that converting to and computing with CSR is faster only when the cost of conversion can be amortized across multiple uses. This figure was adapted from our paper on format extensions to the taco compiler [41].

7.4 Optimizations Matter

In this final evaluation section, I describe experiments to show that the optimizing transformations I proposed in Chapter 6 matter for performance. I demonstrate their importance by providing experimental results that show that we can often obtain better performance by scheduling, and that the best schedule depends on the situation.

Optimizing for GPUs

A good GPU schedule is different from a good CPU schedule. These processor architectures and their memories behave very differently, and kernels must be scheduled to iterate so that the machine at hand can work efficiently. For example, GPUs are sensitive to thread divergence and uncoalesced loads; hence, kernels should be scheduled to avoid these issues where it is practical. GPU schedules therefore tend to be more complicated than CPU schedules in order to ensure that operations are done in the right order. For instance, when we compile the best parallel SpMV schedule for our Intel CPU to our NVIDIA GPU, it performs 6.9x worse than a warp-per-row GPU schedule. This experiment was run on a matrix with four million nonzeros placed at random locations.

The best schedule for the SpMV operation on our Intel CPU is to split the outer i index variable by 16 into two variables i_1 and i_2 and to parallelize over i_1. This approach creates an outer parallel loop where each iteration processes 16 rows, which is sufficient work to amortize parallel overhead on this processor. This schedule does not work well on a GPU because they are much more sensitive to load imbalance. Furthermore, threads in GPU warps load different matrix rows and therefore cannot coalesce memory loads. Hence, the kernel only uses a fraction of the possible memory bandwidth.

In contrast, a schedule optimized for GPUs is carefully tiled to achieve coalesced loads and load balance and achieves excellent performance, even compared to hand-optimized kernels (Figure 7-7). It is tiled by the nonzeros of the matrix, and therefore assigns exactly the same number of matrix nonzeros to load to each thread. It also uses warp-level synchronization primitives to compute partial sums and atomic instructions to combine the partial sums. Finally, the inner loop is unrolled to increase instruction-level parallelism. On a matrix with four million randomly allocated nonzeros, this increased instruction-level parallelism provided a 36% performance improvement for our optimized schedule over the same schedule without the temporary or loop unrolling. All these transformations to target GPUs significantly increase performance over the obvious SpMV implementation. But they are difficult to carry out by hand, further motivating an automated system.

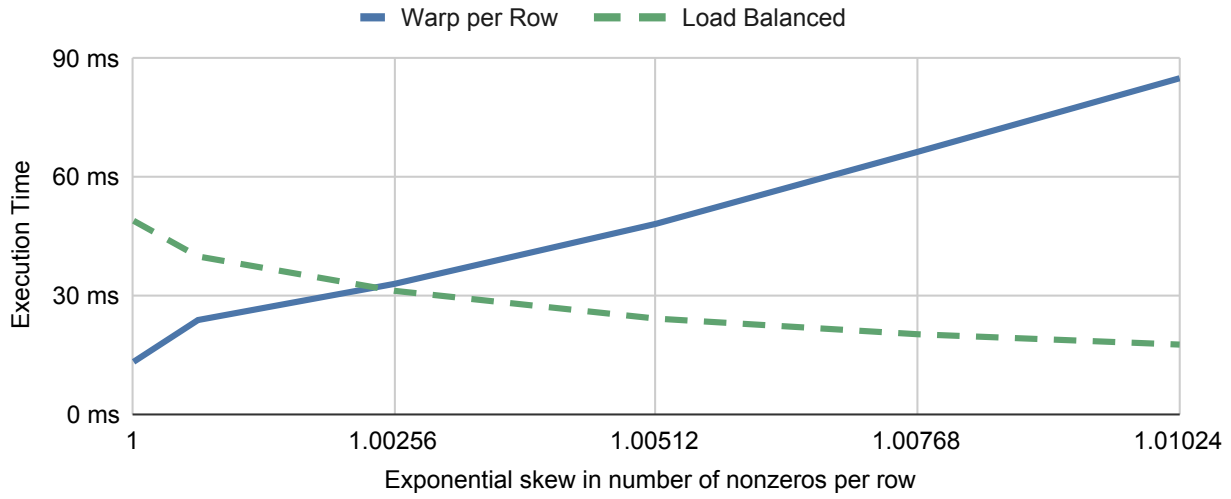


Figure 7-5: The execution time of an SpMV warp-per-row (blue) and load-balanced (green dashed) schedules, on matrices with a fixed number of nonzeros that become exponentially more skewed. The skew is controlled by the tuning knob c, whose value increases along the x axis.

Optimizing for Load Balance

This study shows that the best schedule to use also depends on how load-imbalanced the computation is because we can transform the iteration space to be more load balanced by fusing loops to create bottom-up iteration. Such iteration was described in Section 3.2 and trades improved load balance for increased overhead. I demonstrate the effect of scheduling on an SpMV kernel, but the observations generalize to general sparse tensor expressions.

In the previous section, I described a warp-per-thread GPU SpMV kernel that performs well across many matrices. It schedules threads in a warp to collectively work on a matrix row in order to coalesce loads. But if the pattern of nonzeros in the matrix is skewed so that some rows have a lot of nonzeros and others have very few or none, then this version suffers from load imbalance.[41] The schedule of the optimized GPU kernel in Figure 7-7 collapse the loops and then split them with respect to the matrix. This approach perfectly divides the multiplications across blocks and provides good load balancing at the cost of overhead from recovering the i coordinates by tracking the current row in B. The scatter plot in Figure 7-5 compares the performance of warp-per-thread and load-balanced schedules as the nonzero pattern becomes increasingly skewed according to an exponential formula. We use the formula

$$k * c^i,$$

where i is the row number, c is a skew tuning knob that varies along the x axis, and k is chosen relative to c to give the same overall number of nonzeros. In addition, the rows are randomly shuffled. The figure shows that the execution time of the warp-per-thread schedule (blue)

[41] Skewed matrices are common in data analytics, where we often observe power-law distributions, which have most of the nonzeros on a few of the rows.

increases as the skew increases, even though the number of nonzeros remains the same, due to thread divergence. The load-balanced schedule (green dashed) does not suffer from load imbalance, and it actually performs better because skew increases due to longer rows that decrease atomic conflicts at row ends. The execution times intersect when the value of the base c is approximately 1.00256. The load-balanced schedule is therefore more robust to skew, and it is preferable when the skew is high.

Optimizing for Increased Parallelism

We can also collapse loops to increase the amount of available parallelism. This approach does increase the overhead because there is less work per parallel task, but it can help in situations where there is little available parallelism in the first place. The schedule of the warp-per-row GPU SpMV kernel assigns a full row to each warp. Therefore, if the row has many nonzeros, then each thread gets to compute multiple values, and the startup overhead can be amortized across these computations. But if a matrix has few rows, then there may be too little parallelism to occupy a GPU, and fusing the two SpMV loops may be a better option. For example, we executed the SpMV kernel generated from each schedule on a short and wide $100 \times 100{,}000$ matrix with 10,000 nonzeros per row. This matrix had too few rows to occupy a GPU, and across 10 runs, the median execution time of the collapsed kernel was 4.5 times faster.

Optimizing for Locality

Finally, sparse tensor algebra expressions that iterate over compressed data structures can still have some dense loops that may be tiled for better temporal locality. An important example is the SpMM expression, where a sparse matrix is multiplied by a dense matrix. The resulting kernel has two dense loops that we can tile. We generated tiled and untiled versions and ran them on a $100{,}000 \times 100{,}000$ sparse matrix with an average of 1,000 randomly placed nonzeros per row that we multiplied by a $100{,}000 \times 32$ dense matrix. The tiled version performed 2 times better due to increased temporal locality; this result demonstrates the importance of tiling transformations in mixed dense-sparse loop nests.

7.5 Kernels are Competitive

The performance of taco-generated kernels is comparable with hand-optimized CPU and GPU implementations from leading linear and tensor algebra libraries. Sometimes, taco generates faster code (e.g., one taco-generated CSR SpMV kernel is on average 6.8% faster than MKL over 1677 SuiteSparse matrices), whereas sometimes the hand-optimized

code is faster (e.g., the SPLATT MTTKRP implementation is on average 14.6% faster than one taco-generated kernel).

But the point is *not* to beat these specific library implementations, with these specific tensor formats, on these specific architectures. The point is generality. Generality in the expressions we can compute. Generality in the formats we can use. Generality in the optimizations we can apply. And generality in the hardware we can target. The benefits of this generality—expressiveness and performance—are described in Section 7.2, Section 7.3, and Section 7.4.

The experiments in this section show how taco compares to leading implementations of the kernels that have received most attention from the performance optimization community. Because the taco compiler applies the same general optimizations—sparse coiteration and schedules—to any expression, the experiments provide evidence of its performance on all tensor expressions. For example, the reorder, collapse, split, precompute, and parallelize transformations are sufficient to optimize the expressions in this section, which range from linear algebra multiplications to higher-order tensor contractions and additions.

To provide a fair comparison, we made taco generate kernels that compute on the same tensor formats as the libraries we compare to. As we will see in Section 7.3, it is often possible to obtain better performance by changing the format to fit the tensor. The purpose of the experiments in this section, however, is to show that we can generate similar code to what performance engineers write by hand. Further, it is important to do well on these formats because the user cannot always afford the cost of converting between tensor data structures.

Sparse Matrix-Vector Multiplication (SpMV)

Figure 7-6 and Figure 7-7 plot the time it took taco-generated kernels and hand-optimized library implementations to compute sparse matrix-vector multiplication (SpMV) on a CPU and a GPU. We ran both experiments on 1677 sparse matrices from the SuiteSparse repository [47], and they are ordered by increasing number of nonzeros. The CPU results in Figure 7-6 compare two taco-generated kernels to the Eigen [60] and Intel MKL [44] libraries. The first taco-generated kernel has a parallel outer loop (taco). The second kernel also has a parallel outer loop, but the loop has been strip-mined to create coarse-grained parallel tasks (taco strip-mined). The first kernel performs better on small matrices, presumably because it has more parallel tasks. The Eigen library performs best on small matrices of the compared libraries. Its performance then degrades relative to the other libraries, until the matrices reach approximately 20,000 entries. At that point, it switches to a parallel implementation and performs on par with the other libraries. The experiment shows that the performance of the taco-generated kernels is similar to MKL and Eigen across many large matrices.

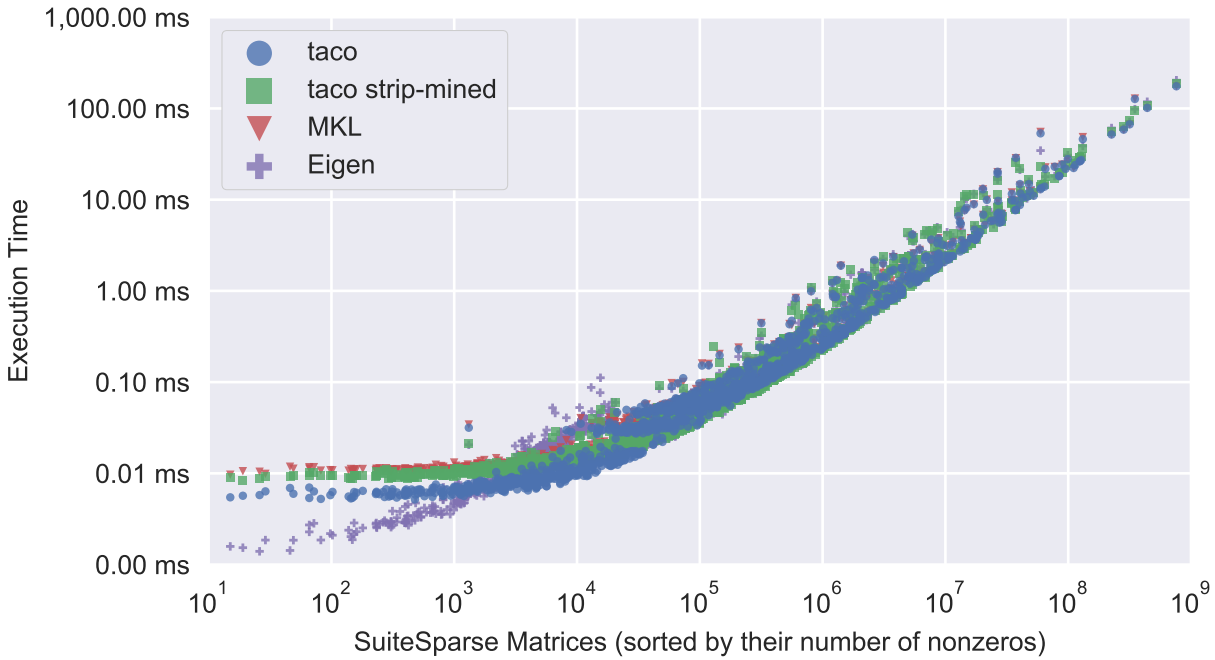


Figure 7-6: SpMV CPU results with the CSR format. I compare two taco-generated kernels to the Eigen [60] and Intel MKL [44] libraries. The first taco-generated kernel has a parallel outer loop (taco). The second kernel also has a parallel outer loop, but the loop has been strip-mined to create coarse-grained parallel tasks (taco strip-mined).

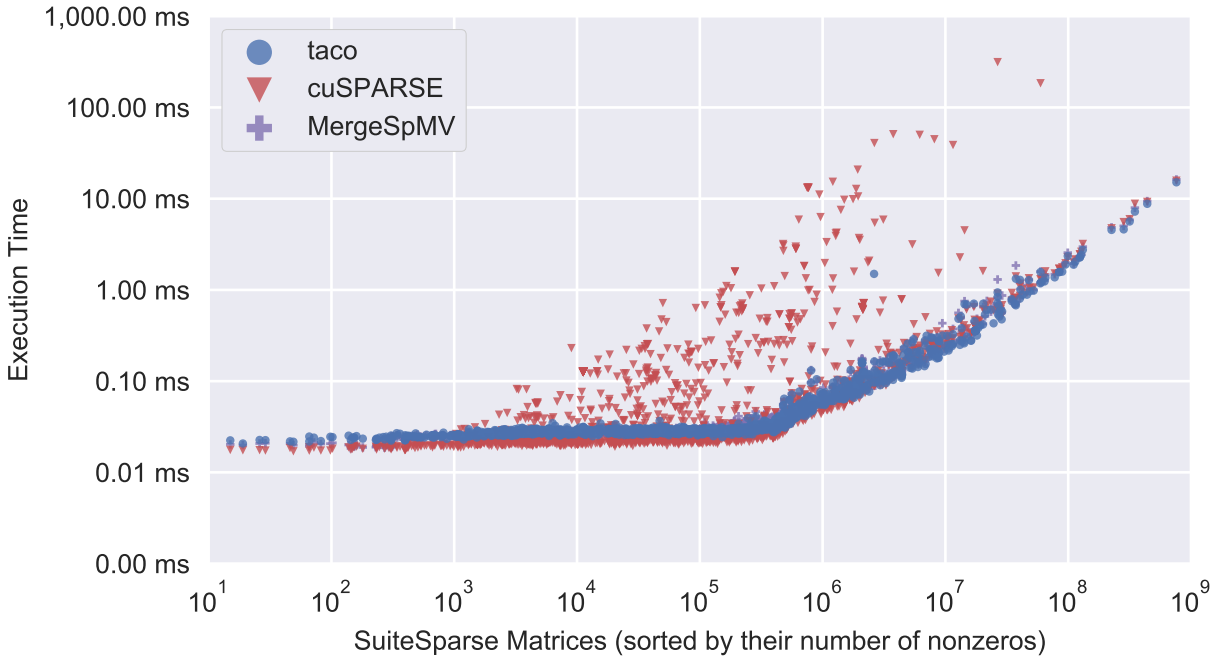


Figure 7-7: SpMV GPU results with the CSR format. I compare a taco-generated load-balanced kernel with the SpMV implementation in the NVIDIA cuSPARSE [125] library and with the Merge-based load-balanced SpMV implementation of Merrill and Garland [93].

The SpMV GPU results in Figure 7-7 compare a taco-generated kernel to the NVIDIA cuSPARSE library [125] and to the load-balanced merge-based SpMV implementation of Merrill and Garland [93]. The taco-generated kernel is generated using a collapse transformation followed by a split transformation that results in load-balanced execution. Thus, its execution time scales roughly with the number of nonzeros, similar to the MergeSpMV implementation. The NVIDIA cuSPARSE library performs better on small matrices, due to fewer kernel calls, but the results show that its performance is sensitive to load imbalance. For larger matrices, the taco-generated kernel typically performs better. These experiments demonstrate that taco can generate both CPU and GPU kernels that have competitive performance compared to hand-optimized libraries.

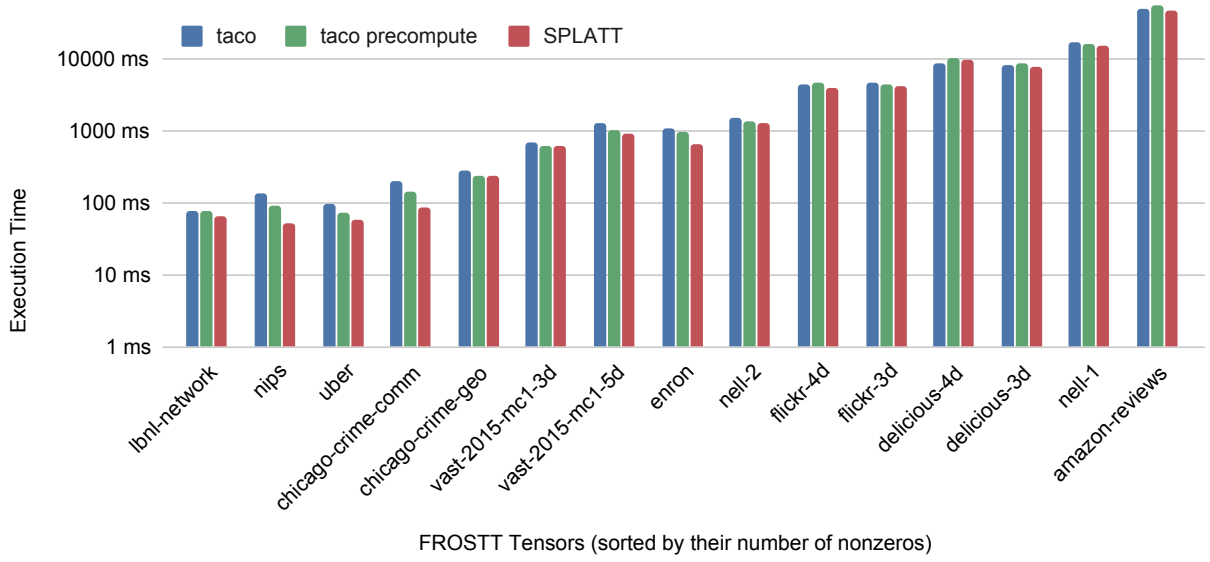


Figure 7-8: MTTKRP CPU with the CSF format

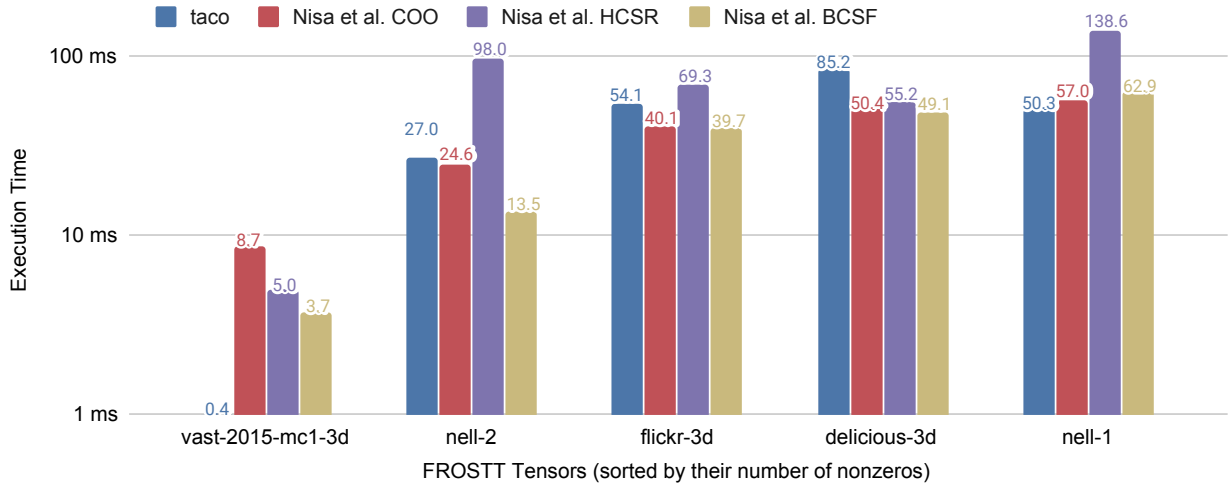


Figure 7-9: MTTKRP GPU with the CSF format

Matricized Tensor Times Khatri-Rao Product (MTTKRP)

Figure 7-8 shows a bar graph that compares the execution time of two matricized tensor times Khatri-Rao (MTTKRP) product implementations generated by taco with the hand-optimized SPLATT library [116]. The first taco-generated implementation has not been optimized beyond parallelizing the outer loop (taco), whereas the precompute transformation was applied to the second implementation (taco precompute). The SPLATT library outperforms the taco-generated precompute implementation on CPUs by an average 15%. Figure 7-9 depicts a bar graph that compares the execution time of a taco-generated MTTKRP GPU implementation with three implementations from a recent paper by Nisa et al. [100] that use three different tensor formats: COO, HCSR, and BCSR. The taco-generated implementation with the CSF format performs well across the five tensors, particularly on the `vast-2015-mc1-3d` tensor. On the nell-2 tensor, however, the BCSF format performs significantly better, which demonstrates the utility of supporting different formats.

Sparse-Dense Matrix Multiplication (SpMM)

Figure 7-11 and Figure 7-12 plot the time it took taco-generated and handwritten library implementations to compute sparse matrix-dense matrix multiplication (SpMM) on a CPU and a GPU. This experiment was run on 1677 sparse matrices from the SuiteSparse repository that were ordered by increasing number of nonzeros. The taco-generated unblocked CPU kernels perform better than Intel MKL on small matrices, and worse on large matrices. The taco-generated GPU kernel, however, performs significantly worse than the cuSPARSE kernel, particularly on small matrices. The primary reason is that the taco-generated kernels perform three kernel launches. I leave as future work to decrease the number of kernel launches. For matrices with more than 1 million nonzeros, however, the performance of the taco-generated GPU implementation is on average 89.5% of cuSPARSE's performance.

Table 7-10: Comparison of taco and MATLAB TTB on four tensor expressions with tensors stored in the coordinate format. taco performs up to 20 times faster than TTB, while supporting the entire tensor index notation. NA means TTB ran out of memory.

	TTV		TTM		Tensor Add		MTTKRP	
	taco	TTB	taco	TTB	taco	TTB	taco	TTB
vast-2015-mc1-3d	543	2950	10302	NA	1553	20895	1357	19706
nell-2	310	4762	2732	42816	3906	77757	4322	60356
flickr-3d	1059	9095	15739	NA	6018	90116	16066	103629
delicious-3d	1803	12695	26882	NA	8120	133167	20249	138943
nell-1	3377	12900	63272	NA	7682	139685	37200	199451

Figure 7-11: SpMM CPU with the CSR format

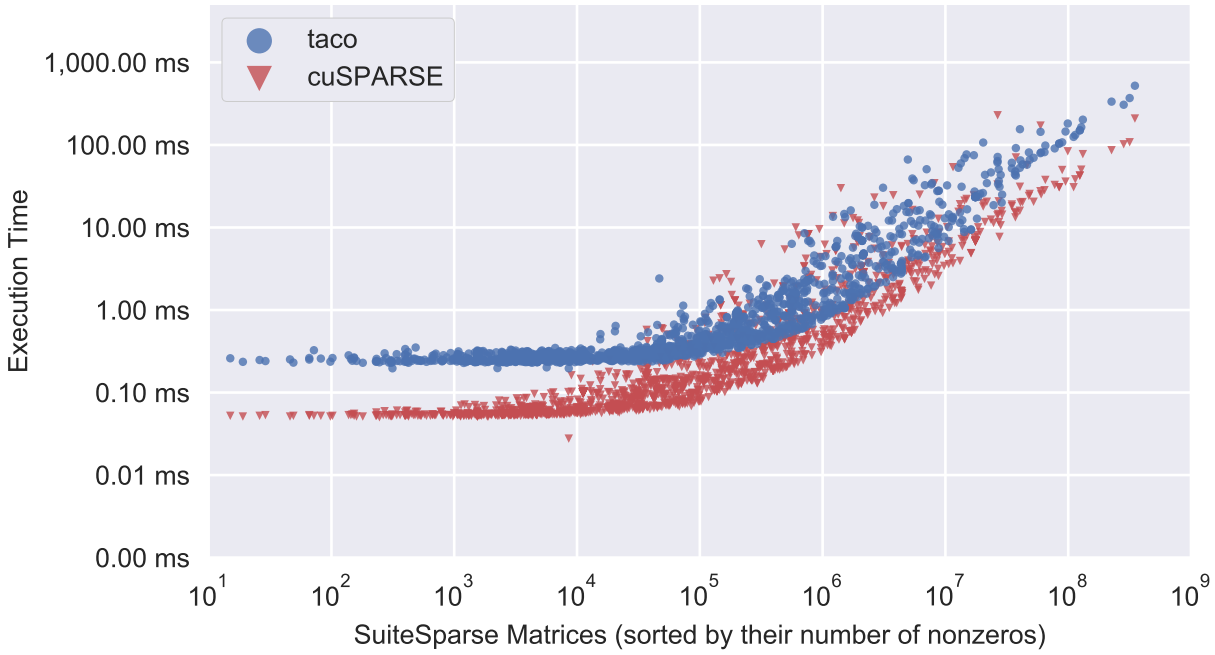


Figure 7-12: SpMM GPU with the CSR format

General High-Order Tensor Expression Support

I have shown that taco can generate code that performs better, the same, or within 15% of hand-optimized implementations on three of the expressions that are most studied in the performance literature, namely SpMV , SpMM, and MTTKRP. The number of expressions is unlimited, however, and many expressions have not yet been handwritten. For these expressions, there exists a few programming systems that support general classes of expressions. One such system is the MATLAB Tensor Toolbox (TTB) by Bader and Kolda [14]. TTB was an early system for sparse tensor computations that provided unprecedented generality. It is built on top of handwritten routines combined with data reorganization, and its generality comes at a performance cost. Table 7-10 shows the performance of taco compared to TTB on four higher-order tensor expressions with tensors stored in the coordinate format. The expressions are tensor times vector multiplication (TTV), tensor times matrix multiplication (TTM), tensor addition, and MTTKRP. Due to its code generator approach, taco is up to 20 times faster than TTB, while at the same time supporting more expressions. taco also supports GPU code generation that further increases performance when a GPU is available. Chapter 8 further argues the case for a compiler-backed programming system. With a code generation approach, we do not have to choose between the performance of handwritten routines and the generality of programming systems.

Specialized formats

Figure 7-13 and Figure 7-14 contain bar charts that show the performance of taco-generated SpMV implementations that compute on matrices stored in the DIA and ELL formats respectively. As I describe in Section 2.5, DIA and ELL are specialized formats for matrices with special structure. The DIA format is designed for matrices with only a few diagonals, such as those that arise from finite difference methods on structured grids and filters on images. The DIA results in Figure 7-13 compare a taco-generated implementation to handwritten implementations in Intel MKL [44] and SciPy [69]. The taco-generated implementation performs on par with SciPy's implementation, while Intel MKL performs better due to sophisticated cache blocking. I leave recreating these results, by combining taco's format support with the scheduling language, as future work. The ELL format optimizes matrix data structures by assuming every row has the same number of nonzeros. This assumption lets it omit the row sizes and potentially enables vectorization. Because the Intel MKL and SciPy libraries do not support the ELL format, I compare to the matrix template library (MTL4) [59]. The ELL results in Figure 7-14 show that taco also performs well on this format, further demonstrating that it performs well across expressions and formats.

Sparse Matrix Addition

Figure 7-15 plots the time it took taco-generated kernels and hand-optimized library implementations to compute a matrix addition. The x-axis lists 1677 matrices from the SuiteSparse repository. Each matrix was added to a synthetic matrix that was created by increasing the column coordinate of each nonzero by one, wrapping the last column around to zero. This approach yields two matrices with similar nonzero structure and with a mix of nonzeros that either overlap or do not. The plot shows that the taco-generated implementation performs similar to the best performing alternative, which is the SciPy implementation. The Matrix Template Library (MTL4) [59] perform less well on this particular kernel. The Intel MKL matrix addition kernel is a parallel kernel that we ran with one thread to get a direct comparison. Taco does not yet support generating parallel code when the result is sparse, which I leave as future work. When running MKL in parallel, it performs better than taco on large matrices, but worse on

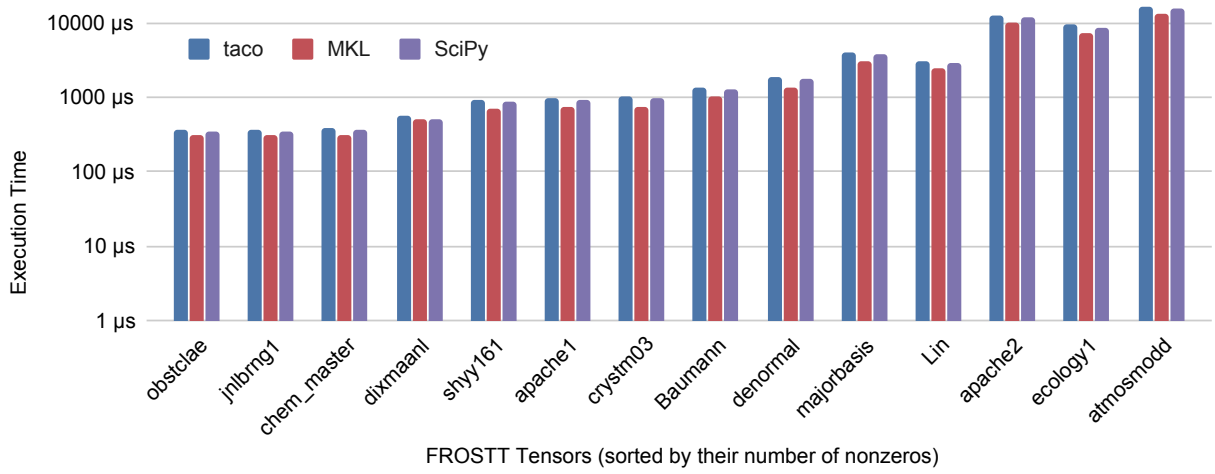


Figure 7-13: SpMV CPU results with the DIA format.

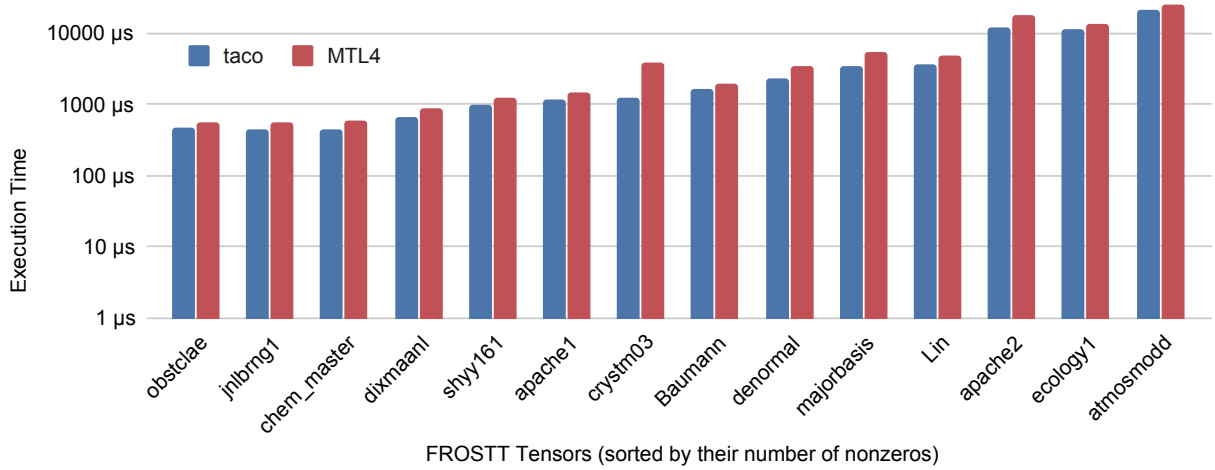


Figure 7-14: SpMV CPU results with the ELL format.

small matrices.

Concluding Remarks

Together, these experiments demonstrate that taco can generate fast CPU and GPU code across many different operations from both the sparse linear and tensor algebras. Further, it generates the code using general code generation techniques and optimizing transformations that apply across all tensor algebra expressions with tensors stored in many different formats. Section 7.2 shows why it is important to support the entire tensor algebra—not just a subset. Section 7.3 illustrates why it is important to support many different tensor formats. And Section 7.4 shows how the optimizing transformations can be used to adapt the generated code to the peculiarities of accelerators, by transforming its sparse iteration space, precomputing temporary tensors, and even tiling sparse loops for static load balance.

Figure 7-15: Matrix Addition CPU with the CSR format.

"Do not seek the footsteps of the wise; seek what they sought."
— Matsuo Bashō

Chapter 8

Related Work

In this chapter, I provide a history of the work on compilers, libraries, and programming systems for sparse linear and tensor algebra. Figure 8-1 shows a timeline of significant developments classified into three categories: sparse compilers, sparse kernel libraries, and sparse programming systems. Throughout this chapter, I describe and contrast these categories and discuss their pros and cons. To put these sparse developments in context, I then end with a brief overview of the longer history on language and compiler support for dense ar-

8.1 Sparse Compilers
8.2 Sparse Kernel Libraries
8.3 Sparse Programming Systems
8.4 Dense Programming Systems and Compilers

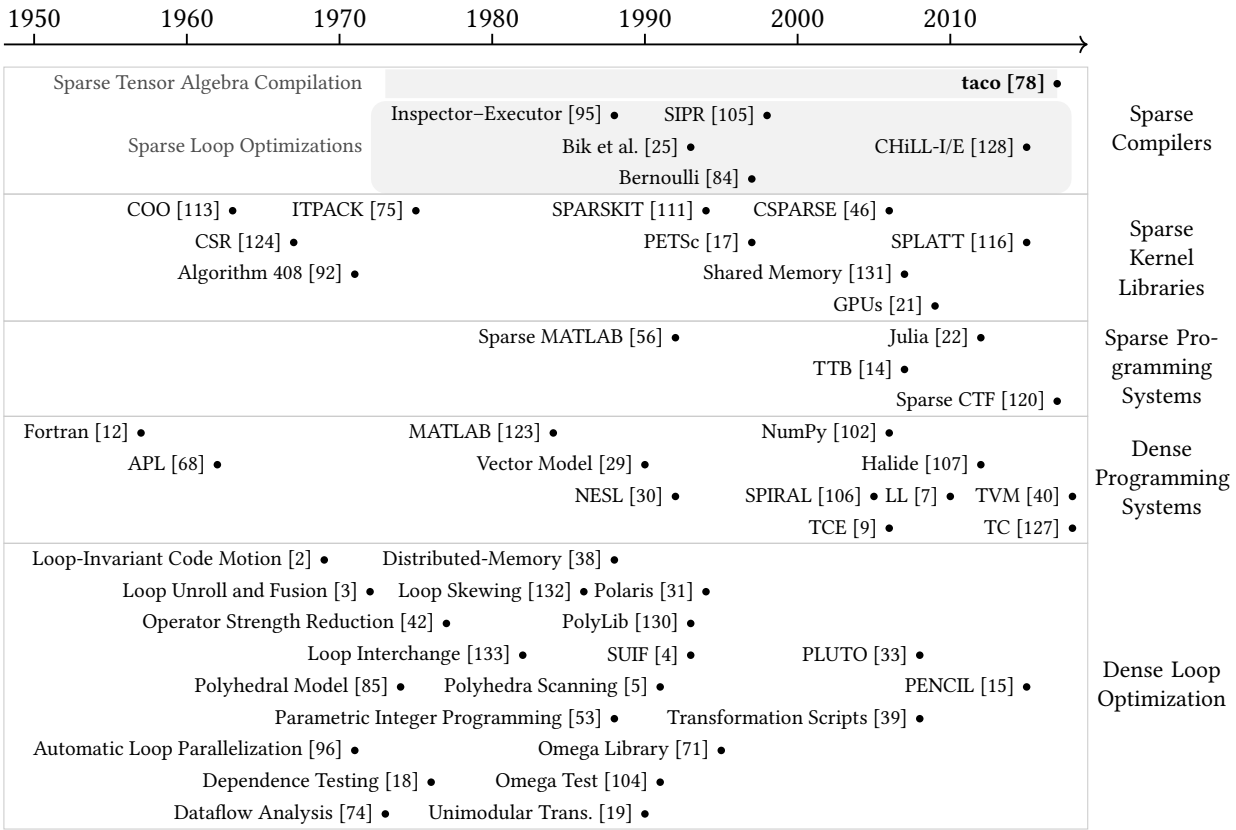


Figure 8-1: Timeline of efforts to provide library and programming system support for array and tensor computations.

ray and tensor algebra codes. I have chosen to make a distinction between libraries and programming systems, where I define a library as software that provides implementations for single operations, whereas programming systems offer general support for a class of operations. The programming systems vary in the type of operations they support. For example, NumPy supports dense array operations with broadcast semantics, the MATLAB Tensor Toolbox supports any binary pairwise sparse tensor contractions on coordinate tensors, and taco supports any n-ary tensor expression and tensors stored in many dense and sparse formats.

There are two ways to create a function that computes a tensor algebra expression (a kernel): it can be handwritten by a programmer or generated by a compiler. For any given kernel, it is, of course, easiest to write it by hand than to develop a compiler. In fact, how to generate general sparse tensor algebra kernels was, until this dissertation, an open problem. The downside of handwritten kernels is, however, that every expression incurs a development and optimization cost for each data structure and for each machine it is required for. Developing even one such kernel necessitates an intellectual effort that is often of such significance as to be published in a peer-reviewed conference or journal paper. Thousands of papers have been written on different sparse linear and tensor algebra kernels for different expressions, data formats, optimization strategies, and machines. Furthermore, as argued in Section 1.2, the number of possible kernels for sparse tensor algebra grows a the cartesian combination of these factors. Therefore, unless we move to a code-generation approach, such as the one described in this dissertation, we should expect thousands more.

A compiler approach promises to reduce overall implementation cost by enabling programmers to express their tensor expressions in a high-level tensor notation. These expressions are then combined with separate descriptions of data formats, optimization strategy, and machine to automatically generate an optimized kernel. This approach provides users with the best of both worlds: they write their expressions in high-level tensor notation and obtain the performance as if they wrote and optimized the kernel by hand in a low-level language such as C.

General programming systems for tensor algebra can be designed in two ways: they can either be built on top of libraries of handwritten kernels (Figure 8-2) or on top of a compiler (Figure 8-3).

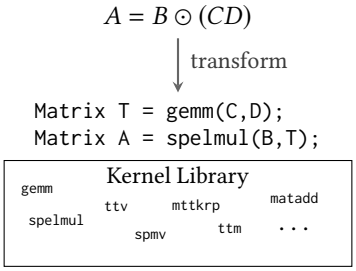


Figure 8-2: Programming system built on top of a kernel library. The system must transform expressions and tensors to fit available hand-written kernels.

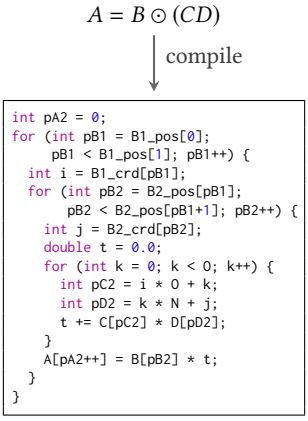


Figure 8-3: Programming system built on top of a compiler. The system can invoke the compiler to generate any kernel and is therefore not forced to transform expressions and tensors.

Programming systems built on top of kernel libraries must implement a strategy to map general expressions to a fixed number of kernels. These programming systems, such as MATLAB[123] and Cyclops [119], rewrite compound expressions as a sequence of sub-expressions and rearrange tensors to fit the available kernels. This strategy reduces their performance because sequences of unfused subexpressions may perform *asymptotically* more operations, may suffer from poor temporal locality, and may re-

quire data movement. And reorganizing tensors to look like matrices, so that tensor contractions can be expressed as linear algebra, requires costly data movement and transpose operations.

Programming systems built on top of compilers can generate implementations for any expression on demand and are not forced to rewrite expressions to subexpressions or reorganize data.[42] These programming systems are, however, still free to rewrite expressions when doing so improves performance. A programming system built on top of a compiler combines the performance of hand-optimized implementations with the generality of an expression language.

[42] As I am writing this dissertation, taco is the only compiler that can generate implementations for any tensor algebra expressions.

This dissertation describes how to build a sparse tensor algebra compiler. I have also outlined an implementation of a sparse tensor algebra compiler called taco (The Tensor Algebra Compiler), which we have made available as an open-source project under the MIT license.

In the following three sections, we will survey prior work on compiler techniques, kernel libraries, and programming systems. Most of this work has been on developing libraries of hand-optimized kernels for specific computations on specific formats on a specific machine and programming systems backed by those kernels. There has also been some work on compiler support for sparse tensor algebra; however, it has been limited to small classes of linear algebra expressions on a handful of matrix formats. The work in this dissertation, on the other hand, handles all of the basic linear as well as tensor algebra on a large class of vector, matrix, and tensor formats.

8.1 Sparse Compilers

This dissertation describes the compiler techniques that are needed to generate code for any sparse tensor algebra expression on tensors stored in the many formats that can be described by our format language. These techniques are the first to accomplish this level of generality. Prior to this work, code generation and optimization for sparse linear or tensor algebra was considered to be an open problem by the compiler community, as demonstrated by the following quote from Professor Sadayappan of the University of Utah, who is unaffiliated with this work [63]:

> "Many research groups over the last two decades have attempted to solve the compiler-optimization and code-generation problem for sparse-matrix computations [...] The recent developments from Fred and Saman represent a fundamental breakthrough on this long-standing open problem."

In the rest of this section, I describe the work by these groups and attempt to compare and contrast it to the work in this dissertation.

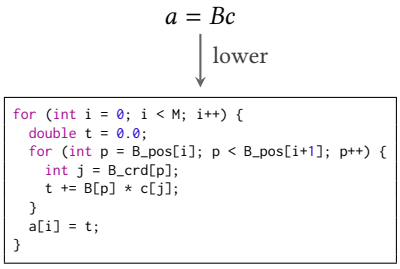


Figure 8-4: The top-down compilation approach to sparse tensor algebra compilation, where a high-level notation are lowered to an optimized kernel by a code generator.

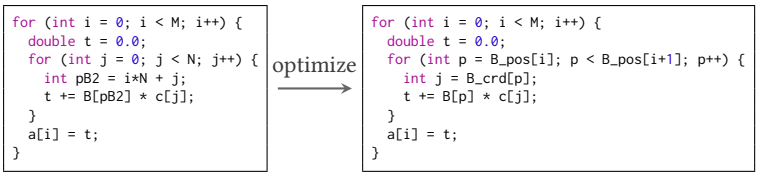


Figure 8-5: The sideways optimization approach to sparse tensor algebra compilation, where imperative loops are optimized and potentially converted to sparse loops that iterate over a different format. For example, a dense loop nest is converted to a sparse loop nest over a CSR data structure.

There are two approaches to sparse compiler techniques:

compilation of high-level language constructs, where the computation is expressed in high-level mathematical languages, such as tensor algebra, that is compiled down to optimized imperative code (Figure 8-4); and

optimization of sparse imperative code, where sparse imperative loops are optimized or where dense loops are transformed to sparse loops (Figure 8-5).

The work in this dissertation falls into the first category and explores how to compile and optimize a tensor algebra expression language on dense and sparse operands. This expression language can then be implemented as part of a language or as a library. The prior work, however, falls into the second category, exploring how to transform dense imperative linear algebra code to sparse code or how to optimize sparse code.

High-Level Language Construct Compilers

In the compilation approach, the linear and tensor algebras are viewed as programming languages, or part of programming languages, to be optimized and compiled to machine code, similar to the other programming language constructs. The history of compiler construction has been a march towards higher-level programming language constructs, from assemblers, through to the replacement of goto statements with half a dozen language constructs in structured programming, to modern high-level object-oriented and functional languages. As stated by Backus et al. [12], the goal of the FORTRAN Automatic Coding System

> "was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics and obtain automatically from this specification an efficient 704 program to carry out the procedure."

Support for tensor algebra—a mathematical language on multilinear collections of numbers—can be viewed as another step. Indeed, Parnas traced the history of automatic programming back to a machine that converted instructions to holes in punch cards, and concluded that

> "automatic programming always has been a euphemism for programming with a higher-level language than was then available to the programmer. Research in automatic programming is simply research in the implementation of higher-level programming languages." [103]

There are several benefits to viewing tensor algebra as a program to be optimized and compiled to machine code. First, and perhaps foremost, is that programmers obtain a high-level language to work in that is far more productive than an imperative language. Further, the language consists of simple expressions that can easily be presented to a programmer in the form of a library for easier adoption—indeed, the work in this dissertation was implemented by the author and collaborators as a C++ library. It is also easier to optimize programs written in tensor algebra than the corresponding program written in imperative code. They are closed expression languages and can be reasoned about algebraically, from mathematical properties, without the need for sophisticated control flow and dependence testing.

Sparse Imperative Code Optimizers

In the imperative code optimization approach, the sparse linear and tensor algebras are viewed as a compiler optimization problem on imperative code. These approaches take a loop nest that expresses a sparse or dense computation, such as a matrix multiplication, and optimizes it, either by transforming it to operate on a different data structure (e.g., from dense to CSR) or by changing loops to parallelize or tile dense sub-computations in mixed sparse and dense code.

The benefit of the code optimization approach is that it has the potential to generalize to a larger class of sparse computations than those computations that can be expressed in a high-level expression language. This generalization is compelling, and the history of dense computations has showed us that dense linear algebra was an excellent testbed for a large class of important loops. Many analysis, tiling, and auto-vectorization techniques developed from exploring such kernels, including dependence testing and the polyhedral mode. We should hope that the research on optimization of sparse code has similar success, despite the serious challenges related to analyzing code that depends on runtime values. Furthermore, the implicit use of linear and tensor algebra, where computations are linear but not phrased as operations on explicit tensors, is common. For example, the seven key numerical methods in science and engineering identified by Phil Collela in 2004 [43, 8] are predominately linear algebra in different guises.[43]

[43] The seven numerical methods are dense linear algebra, sparse linear algebra, spectral methods, N-body methods, structured grids, unstructured grids, and Monte Carlo. The first two are explicit linear algebra, and spectral methods, such as the FFT, are also predominately linear. The next three are typically either implicit matrix-free linear algebra, or turn into linear algebra through a matrix assembly. Finally, Monte Carlo random sampling methods are often a driver on top of a linear optimization or integration problem.

Compiler techniques that target general classes of sparse linear, multilinear, or even nonlinear computations that cannot yet be efficiently expressed in explicit algebraic form without loss of performance are therefore of significant value.

The drawback of the code optimization approach, however, is that it requires extensive control flow and dependence analysis, often involving expensive inspectors, that makes them more complicated and more fragile than compilation from high-level languages. Furthermore, the extensive use of linear and tensor algebra libraries and languages shows that programmers prefer to use high-level abstractions instead of writing imperative code by hand. Thus, I believe that compilers for high-level language constructs are preferable for computations that can be expressed in these high-level abstractions.

Prior Sparse Compilation Techniques

I discuss prior work on systems and techniques for sparse compilation in chronological order, starting with the work of Bik and Wijshoff in 1993 [25] and ending with the work of Venkat, Strout, Hall, and Olschanowsky on the CHiLL-I/E compiler [128] and the Sparse Polyhedral Framework [122].

The first research on compiler techniques for sparse linear algebra that is known to me was carried out by Bik and Wijshoff in the mid 1990s at Leiden University in the Netherlands. In a series of publications from 1993 to 1998, they presented a set of compiler transformations, implemented in their MT1 compiler, to convert dense FORTRAN implementations of some linear algebra primitives to sparse implementations [25, 26, 24]. In their approach, the programmer writes dense code and annotates the code to instruct the compiler regarding what matrix to sparsify. Alternatively, they also have a technique referred to as sublimation, published by Spek and Wijshoff in 2011 [126], that turns sparse code into dense code so that it can be sparsified into another data structure by the other techniques. In either case, the compiler then performs a dependence analysis on the dense loops and applies several incremental loop transformations that results in sparse code that uses a sparse data structure. The loop transformations they present are limited to matrix multiplication kernels, including a triagonal solve, where only one operand can be sparse. Their system can sometimes also apply a technique called loop distribution to densify a sparse array, similar to the temporaries in concrete index notation (Section 4.3). Their method does not work for addition expressions or for expressions that require coiteration over multiple sparse tensors. Their approach is also complicated because it must execute many fine-grained loop transformations, with preconditions that rely on loop dependence analysis, in exactly the right order to obtain to the correct sparse code.

The Bernoulli project took place at Cornell in the last half of the 1990s and was carried out by Kotlyar, Stodghill, and Pingali [84, 121,

83]. It presented a radically different approach to sparsifying dense loop nests than that of Bik and Wijshoff. Rather than applying a sequence of transformations on imperative loops, the Bernoulli system lifts the loops into a high-level relational algebra formulation that enumerates the dense iteration space as a cartesian product. Sparsity is then introduced as predicates in filters that are then pushed down into the cross products to turn them into relational joins. Finally, the system produces code similar to how traditional database systems work by selecting join algorithms from a menu of hand-coded join implementations. The Bernoulli system was demonstrated to work on many of the same matrix multiplication kernels as Bik and Wijshoff's approach, including triagonal solve. But it expanded the number of possible data structures and could generate sparse multiplication kernels with more than one sparse matrix if a suitable join implementation existed. By lifting the code to a higher level of abstraction than imperative code, the Bernoulli system substantially simplified the introduction of sparsity. The lifting still requires complex control flow and dependence analysis, however, and may therefore be fragile to minor changes in the loop nest. Furthermore, it is restricted to multiplication kernels, and there is no concept of temporaries because the relational algebra only describes the iteration space and not computation. Finally, it is limited by the need to map multiplications to hand-coded join implementations that, for binary operations, grows as the Cartesian product of supported data structures and machines increases. And for operations with more than two operands, such as the MTTKRP and SDDMM families of expressions, the number of join implementations grows exponentially. The techniques presented in this dissertation show how to instead generate code to coiterate over any combination of multiplications and additions over any number of tensor data structures stored in different formats.

The Sparse Intermediate Program Representation (SIPR) was developed by Pugh and Shpeisman as an IR for sparse compilers such as Bik's MT1 compiler and the Bernoulli compiler [105]. It was intended to separate the complications of sparse matrix data structures from computations and could express accesses and iteration over data structures as well as matrix format changes during program execution. Similar to the concrete index notation (Section 4.3), it represents programs as imperative code, but with sparsity predicates in the loop bounds. As straight-line imperative code, it has much looser semantics than the concrete index notation and does not type its operands. Furthermore, the compiler techniques are limited to those used in MT1 and Bernoulli: each loop can only iterate over one sparse data Structure, and additions are not supported. A particularly interesting feature of the SIPR system, however, is that it has a cost model that counts the number of operations of each sparse loop and thus provides a cost of the overall expression.

The LL language was developed by Arnold et al. to express and verify sparse matrix codes [7, 6]. It is a functional language with many

of the common functional constructs, such as map, filter, and reduce, and they show that it is expressive enough to describe assembly and SpMV kernels for many sparse formats, including COO, CSR, BCSR, and Jagged Diagonals. They also describe a compilation approach inspired by the flattening techniques and segmented vectors of the NESL language compiler [28]. Finally, they use the tight variable-free functional semantics of LL to verify, for the first time, the correctness of the assembly of SpMV programs. A limitation of the functional approach of the LL language is that programs are tied to the details of sparse formats; each program must therefore be written for each format. It is also not clear how well the language generalizes to more complicated programs, such as matrix-matrix multiplication and matrix addition. The ability to verify sparse matrix routines, however, is obviously appealing because they are used in applications such as nuclear simulations and bridge design optimization, where I think we would all prefer correct results.

Finally, Strout, Hall, and Olschanowsky developed the Sparse Polyhedral Framework in the far west of the USA in the 2010s [122]. It combines polyhedral compilation with an inspector-executor approach to data inspection. The framework shows how to generate and compose inspectors and how to optimize sparse data using polyhedral techniques with uninterpreted functions to model non-affine data accesses. This framework is being incorporated into the CHILL-I/E compiler by Venkat, Hall, and Strout [128], and it complements this compiler by providing optimizing transformations to move between dense and sparse similar to those of Bik, Spek, and Wijshoff. The result is a promising general sparse compilation framework.

8.2 Sparse Kernel Libraries

Kernel libraries have been the dominant approach to sparse linear and tensor algebra. The first use of sparse matrix data structures known to me is by Sato and Tinney in 1963, who use the coordinate format to store a network admittance matrix [113]. Tinney and Walker then described the compressed sparse row (CSR) data structure in a paper from 1967 [124]. Since then, an enormous number of papers has been written on different linear algebra kernels optimized for different sparse data structures, and this is still an active area of research. In the last two decades, we have seen a similar development for sparse tensor algebra, and many papers concerning sparse tensor data structures and optimized implementations of tensor algebra expressions are produced every year.

The advantage of kernel libraries is that every kernel can be manually optimized by a human. It is also easier to develop an optimized tensor algebra kernel than a sparse tensor algebra compiler. In fact, until this dissertation, it was not even known how to develop a compiler that could generate sparse kernels for expressions more complicated

than a linear algebra multiplication with a single sparse operand in one of a small number of formats. The disadvantage with a kernel library is that humans can, in principle, only handwrite a finite number of optimized kernels. In practice, however, we can only handwrite and hand optimize a small number of kernels, and kernel libraries typically restrict support to a small number of expressions, one sparse matrix data structure, and one type of machine.[44] Applications must thus be written to map their computations to this finite set of kernels, potentially leading to asymptotically worse performance, poor temporal locality, and data structure reorganization. This style of writing applications is particularly troublesome in tensor algebra, where tensors of any order lead to any number of tensor contractions. The standard solution, apart from developing a new hand-optimized kernel, is to transpose the tensor so that the contracted dimension is last, convert it to a sparse matrix, perform matrix multiplication, convert it back to a tensor, and then transpose the contracted dimension back in its place. This approach results in significant data movement, especially when multiple dimensions are contracted with different tensors such as in the MTTKRP operation [14].

[44] Some libraries, such as cuSPARSE [125] and SPARSKIT [111], have partial support for multiple formats. This partial support typically means that the SpMV operation works with multiple formats, and these libraries provide conversion routines between these formats and the format supported by the rest of the library.

Many sparse linear algebra libraries have been developed, and I will mention a few of particular significance. The first sparse linear algebra library known to me is referred to as Algorithm 408, developed by McNamee and published in 1971 [92]. The paper describes a number of operations on sparse matrices, including multiplications, transpositions, and permutations. Implementations that took advantage of special matrix structure came along soon thereafter. The ITPACK library was developed in the early 1970s, and the first publication describing its sparse functionality appeared in 1975 [75, 76]. It was an early comprehensive package that offered support for selecting among different matrix formats, including CSR [124] and ELLPACK [110]. A large number of matrix formats for special matrix structures has since been developed, typically intended to be used with the sparse matrix-vector multiplication (SpMV) operation. The SPARSKIT package [111] is notable for supporting SpMV on as many as 16 different sparse matrix formats and also provides conversion routines between these formats and the CSR format that is used by the rest of the package. This trend continues today, with recent developments of specialized formats for power-law matrices [37]. In the mid 2000s, shared-memory multicore processors became common, and Williams et al. wrote in depth about optimizing SpMV for these machines in 2007 [131]. The Intel MKL library [44] is a highly optimized dense and sparse multi-threaded linear algebra library that is widely used by applications on Intel processors. The last shared-memory library I describe is the Eigen library [60]. Eigen is an open-source header-only C++ library that has seen wide adoption in the last decade due to its ease of installation and use. It makes heavy use of template metaprogramming to generate specialized code for different data types, to fuse dense operations, and to inline vectorized code for common small dense operations into user code

(e.g., 3×3 matrix multiply).

In recent years, researchers have begun to investigate new compound sparse linear algebra kernels, such as the sampled dense-dense matrix multiplication (SDDMM) kernel used in factor analysis algorithms, such as alternating least squares [137]. In this kernel, a dense matrix multiplication is element-wise multiplied by a sparse matrix:

$$A = B \odot (CD),$$

where $\odot$ is an element-wise multiplication. If the operations are performed separately, then the element-wise multiplication discards results wherever the sparse matrix has a zero, whereas a custom compound kernel that computes the entire expression at once never needs to compute those results in the first place. The compound kernel is therefore asymptotically faster if the number of nonzeros in the sparse tensor is included in the asymptotic expression. The techniques in this paper were shown to generate an asymptotically superior compound version of this kernel [78], and an optimized version of the kernel was implemented on a multi-node GPU cluster by Nisa et al. [99].

Graphics processing units (GPUs) have become popular for computing applications over the last decade. They have an order of magnitude more floating-point-operation throughput than CPUs, making them excellent compute units for dense linear algebra. GPUs are sensitive to load locality and are therefore challenging to use for sparse linear algebra; however, because they have an order of magnitude more memory bandwidth than CPUs, they perform better than CPUs on many operations, including SpMV [93, 21]. The NVIDIA cuSPARSE library [125] is a popular GPU library written using the NVIDIA CUDA programming model [101]. It provides optimized GPU SpMV kernels for several sparse matrix formats [21], including COO, CSR, CSC and BCSR, as well as a statically load-balanced implementation of SpMV on the CSR format that uses a merge strategy to distribute work evenly between threads [93].

Many sparse linear algebra libraries have also been developed for distributed-memory supercomputers, typically using the MPI API [118]. The PETSc library from Argonne National Laboratory [16] was an early MPI library for sparse linear algebra basic operations and solvers. It pioneered an insightful object-oriented approach to the construction of supercomputing software, with sparse matrices modeled as distributed objects and with abstract matrix operations [17]. A more recent MPI library for sparse linear algebra is the Epetra linear algebra package in the Trilinos collection of supercomputing libraries from Sandia National Laboratories [65]. This library pushes the envelope on modular MPI software design and was built to make it easy to compose many different software packages to solve linear and non-linear systems.

Finally, the last years have seen a flourishing of kernel libraries for sparse tensor algebra, following the publication of the MATLAB

Tensor Toolbox (TTB) by Bader and Kolda [14]. These libraries typically focus on optimizing a handful of specific tensor kernels that are used to compute variants of the Canonical Polyadic Decomposition and Tucker Decomposition, which are generalizations of Singular Value Decomposition (SVD) for higher-order tensors. These operations include tensor-times-vector (TTV), tensor-times-matrix (TTM), and the matricized tensor times Khatri-Rao product (MTTKRP).[45] The MATLAB Tensor Toolbox proposed implementations of these operations for sparse tensors of any order stored in the Coordinate format. Baskaran et al. then described an early specialized tensor format for tensor factorization in data analytics, called the mode-generic format [20]. The performance of the MTTKRP operation was significantly improved by the SPLATT library of Smith et al. [116] that stores tensors in a high-order generalization of CSR termed CSF and provides hand-optimized MTTKRP functions for up to order-4 tensors. Furthermore, Li et al. proposed a sparse tensor library that uses a high-performance tensor format called HiCOO, a variant of Baskaran's mode-generic format, to further optimize MTTKRP [89]. Finally, Zhang et al. recently proposed a new class of tensor kernels that generalize the SDDMM linear algebra kernel, described above, for alternating least squares to higher-order tensor operations [136]. All of these kernels, except those that use the HiCOO format, can be generated by taco with performance that match or exceed the above implementations.

[45] See Table 7.1 in Bader and Kolda's paper for a comprehensive list of the operations they proposed [14].

8.3 Sparse Programming Systems

A programming system, as defined here, lets the user specify and compute general tensor expressions on sparse tensors. Many general programming systems have been developed for sparse linear algebra, including MATLAB [123] and Julia [23]. They are built on top of sparse linear algebra kernel libraries and work by splitting general linear algebra expression into a fixed set of sub-expressions that can be executed by available kernel functions. For example, the residual

$$r = b - Ax$$

may be split into two sequenced computations

$$t = Ax$$
$$r = b - t$$

if no kernel is available for the compound expression. As I described in Section 1.2, there are at least three problems with this approach:

1. values computed by early kernels may be discarded when later multiplied by zeros, causing asymptotically worse worst-case behavior;

2. loss of temporal locality and increased reuse distance as t is fully

materialized before it is consumed, which may cause values to be flushed from caches; and

3. a requirement that the two kernels agree on the data structure of t, or else its data structure must be changed on the fly.

Despite these issues, the approach of dividing linear algebra expressions to fit available kernels has been the dominant paradigm because code generation from sparse compound linear algebra expressions was an open problem until the approaches in this dissertation were presented.

Programming systems for general sparse tensor algebra are more challenging than programming systems for sparse linear algebra. There are two reasons: a tensor can have any order, and any subset of tensor dimensions can be contracted in a tensor expression. Thus, even the number of binary tensor expressions is limitless, whereas the number of binary expressions in linear algebra is finite. These sources of generality provide additional challenges to efficient implementation in addition to the, potentially asymptotic, inefficiency of composing handwritten kernels that we described above. In the absence of a code-generation technique as described in this dissertation, prior systems for sparse tensor algebra supported these sources of generality in two ways. They

interpret operations in a general routine that determine what operations to perform, and

permute tensors so that the modes to contract come last and then call matrix multiplication to perform the contraction.

An example of interpretation is the MATLAB Tensor Toolbox's support for tensor-times-tensor multiplications [13], and an example of permutation is Cyclops's support for general tensor contractions [119]. Both approaches add overheads to the computation that can be avoided if a compiler is available to generate custom kernels on demand for arbitrary operations.

The MATLAB Tensor Toolbox [14] provides a handful of kernel functions that each work for a small class of tensor expressions. For example, tensor-times-vector (ttv) works for tensors of any order, and tensor-times-tensor lets the user specify how many modes to contract when multiplying two tensors, as long as the contracted modes are at the beginning of the input tensors [13, Section 3.3]. Even this modest generality comes at a high cost in the absence of a compiler to generate custom kernels. As we saw in Chapter 7, the MATLAB Tensor Toolbox kernels are often two orders of magnitude slower than hand-optimized kernels or the kernels generated by the techniques in this dissertation.

The libtensor of Epifanovsky et al. [52] generalizes the limited tensor multiplications of the Tensor Toolbox to general tensor contractions of two tensors for block-sparse tensors. The general contraction lets two tensors of any order be contracted in any number of modes,

which is often required in quantum chemistry and quantum physics. The contractions are controlled by an indexing expression, as shown by the following example from their paper:

$$c_{ijkl} = \sum_{pq} a_{ipkq} b_{jplq}.$$

Their execution scheme interprets the contraction expression at the block level and dispatches block contractions, after appropriate tensor permutation (e.g., transpose), to the dense general matrix-matrix multiplication (GEMM) operation. Such an execution scheme requires interpretation and data transposition; however, when the dense blocks are large, this overhead can to some degree be amortized by the high cost of the dense multiplication.

The Cyclops Tensor Framework (CTF) from Solomonik et al. [119] computes general tensor algebra expressions in the Einstein summation notation on distributed machines using MPI. It was later extended to sparse matrices and tensors, including support for tensor algebra computations in several semirings [120]. Their system provides a function that can compute any binary tensor contraction with two operands by turning the tensor into a matrix and using GEMM for the contraction. They handle contractions with more than two operands by turning them into a sequence of binary contractions. Furthermore, terms of additions are independently computed and then added together in optimized kernels. Similar to libtensor, their execution scheme relies on matching general tensor expressions to handwritten kernels. This approach causes some overhead from data reorganization, from permutation and data structure conversion. Furthermore, when operands are sparse, as we discussed above, it can lead to asymptotic performance degradation when computed values are later multiplied by zeros. The Cyclops programming system, such as the MATLAB Tensor Toolbox and libtensor, can therefore benefit from a compiler to generate custom tensor algebra kernels on demand.

8.4 Dense Programming Systems and Compilers

The use of digital computers for dense array and tensor computations stretches back at least as far as the Manhattan project, where they were used to simulate nuclear reactions. Programming language and compiler support started appearing in the 1950s and has since received significant attention. Fortran, the first programming language, was designed to make it possible to specify numerical procedures with a language closer to mathematics and provided first-class support for expressions, do loops, and dense arrays [12]. Shortly thereafter came the APL language with first-class support for operations on arrays as a whole, instead of loops that access array components [68]. First-class programming language support for dense matrix codes was pioneered by the MATLAB programming system in the 1980s [123]. The

SPIRAL project explored mathematical descriptions of linear transforms as a programming model and showed how they could be compiled to autotuned implementations on data in dense arrays [106, 55]. The NumPy Python library, released in 2006, popularized operations on multi-dimensional dense arrays and the concept of implicit array broadcast semantics [102]. Dense tensor operations were introduced into programming systems in the same year by the Tensor Contraction Engine [9] and the MATLAB Tensor Toolbox [13]. In the early 2010s, the Halide system showed how deep stencil code pipelines could be separately expressed and optimized by two different languages [107]. Finally, the last decade has seen a significant amount of interest in dense tensor algebra compilers for deep learning following the release of the TensorFlow system [1], including XLA [87], TVM [40], and Tensor Comprehensions [127].

There have been two approaches to loop optimization: individual transformations applied directly to the loops and frameworks where the loops are modeled and manipulated as mathematical objects. Of the latter, the two main branches are referred to as the unimodular transformation framework and the polyhedral model.

The first published discussion on program optimization known to me was a book chapter by Allen from 1969, where she describes several optimizations, including loop-invariant code motion [2]. In 1971, Allen and Cocke put together the first systematic catalog of optimizing loop transformations, which includes the unrolling and fusion loop optimizations [3]. That same year in his dissertation, Muraoka was the first to describe a technique to automatically parallelize loops [96]. He generalized work by Bingham and Fischer [27], which computed dependencies between straight-line statements, to compute dependencies between statements in a loop to determine whether they can be run in parallel. Banerjee developed the GCD and Banerjee dependency tests in his 1976 Masters thesis that improves the efficiency of automatic parallelization and vectorization [18]. The next year, Cocke and Kennedy described what I believe was the first operator strength reduction algorithm for loops [42]. Wolfe introduced the loop interchange and showed how it can be composed with strip-mining to tile a loop (Figure 54) in his dissertation from 1982 [133]. He also introduced the loop skewing transformation in a later paper from 1986 [132]. Finally, in 2008, Chen et al. proposed a scripting language to expose compiler transformations, thus cleanly separating the implementation of the transformations and code generation from the machinery to decide what transformation to apply [39].

The unimodular transformation framework, proposed by Banerjee in 1990, is a unified loop transformation framework that describes many loop transformations as linear matrix transformations on vectors of loop indices [19].

The polyhedral model views loop nests as a multidimensional integer polyhedra that can be optimized through mathematical transformations. Its roots go back to Lamport's paper from 1974 on the

parallel execution of DO loops [85], which built on work by Karp et al. on organizing equations for parallel execution [70]. Two breakthroughs around 1990 made the model practical. The first was when Feautrier in 1988 introduced parametric integer programming that can be used for dependence analysis and scheduling [53]. The second came in 1991, when Ancourt and Irigoin showed how to efficiently compute loop bounds and thus generate code by scanning polyhedra [5]. One more polyhedral model development was the Omega, introduced by Pugh in 1990, that generalizes Fourier-Motzkin elimination to integer programming [104]. Since then, the polyhedral model has been used to develop many successful compilers, including SUIF that explored distributed-memory auto-parallelization [4], PLUTO that explored multicore auto-parallelization [33], and PENCIL that explored machine-neutral auto-parallelization [15]. There has also been work on tooling for working with polyhedra, such as PolyLib for rational polyhedra [130] and the Omega Library and isl for integer sets with affine constraints [71, 129].

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

— Alan Turing

"Most likely you go your way and I'll go mine."

— Bob Dylan

Chapter 9

Conclusion

At its heart, my dissertation is about sparse iteration spaces. I develop a comprehensive sparse iteration theory, including an algebra of sparse iteration spaces and an algebra of iteration over those spaces. I show how these spaces can be developed from coordinate relations and coordinate trees, how to transform them to optimize iteration order, and how to compile them to efficient sparse iteration algorithms. I extend the iteration algebra with the concrete index notation to include tensor algebra computation, with the result that we now know how to compile the sparse tensor algebra. The evaluation chapter demonstrated that sparse tensor algebra implementations generated with taco have comparable performance to handwritten implementations in modern sparse linear and tensor algebra libraries. I believe the concepts and techniques I have introduced fulfill the promise in my thesis statement and demonstrates that:

> Sparse tensor algebra can be put on the same compiler transformation and code generation footing as dense tensor algebra and array codes.

Sparse tensor algebra expressions, like their dense counterparts, can now be collapsed, tiled, parallelized, and compiled to code that uses different data layouts and runs on different machines.

Although this dissertation provides a foundation for reasoning about and optimizing sparse iteration spaces and sparse tensor algebra, there are many things left to do.

Sparse Automatic Scheduling

My dissertation shows how to compile sparse iteration spaces to code that iterates over different data structures, and how to transform them to optimize iteration order. But I have left open the question of what the right data structure and iteration order is for a given tensor and expression. That is, I have focused on the mechanism (how to do it) and left the policy (what to do) as future work. This focus was intentional because a mechanism is required to explore policy and because

the policy can be supplied by users or by simple heuristics while research into sophisticated policy methods takes place. The dissertation does, however, describe clean scheduling and format languages that will facilitate policy research. Such research is necessary both regarding how to find the best format for a given tensor (auto-formatting) and how to find the best schedule (auto-scheduling). This research may include heuristics, auto-tuners, hand-crafted models, learned models, and sampling approaches.

Sparse Hardware

The iteration space algebras are abstract descriptions that by design are independent of the physical layout of data and access order. This approach makes it possible to generate efficient algorithms for different architectures. I showed how to specialize code for CPUs and GPUs in this dissertation, but a promising direction is to build hardware that is specialized for computing sparse iteration spaces and for executing operations at their nonempty points. The ExTensor processor is an early architecture influenced by the ideas that went into this dissertation [64] and is the first domain-specific architecture to compute any sparse tensor algebra expression. In the future, I believe that there will be more processors for sparse computing problems and that the theory laid out in this dissertation will be directly applicable to the hardware itself and to the compilers that will target it.

Sparse Array Operations

The ideas of sparse iteration spaces presented in this dissertation generalize beyond tensor algebra, and I believe that they can serve as a foundation for compiler work to address sparse array computations in general. The first steps is to expand sparse iteration theory to apply to sparse array computations in general, including tensor algebra in other semirings[46], general array operations, transpositions, and stencils[47]. Another step is to incorporate symmetry descriptions into tensors and arrays, to reduce storage size, and to increase performance. Beyond basic array operations, I believe that the theory can generalize to many linear and nonlinear solvers that cannot be expressed as basic linear algebra operations, such as Cholesky factorization and LU decomposition.

Sparse Operations beyond Tensors and Arrays

Beyond sparse array and tensor operations, I believe sparse iteration theory can also be a foundation for sparse iteration compilers for related mathematical abstractions that model sparse systems, such as sets, relations, meshes, and graphs.

Viewed as relationships between sets of objects, tensors have a strong connection to both graphs and relational tables. The connection

[46] A **semiring** is an algebraic structure that bundles two binary operators $\oplus$ and $\odot$. These operators that must have the algebraic properties that $\oplus$ is a commutative monoid, $\odot$ is a monoid, $\odot$ distributes over $\oplus$, and $\odot$ is annihilated by the identity element of $\oplus$. Many graph algorithms can be expressed as linear algebra in different semirings. Kepner and Gilbert collect several examples in their edited collection of graph algorithms in the language of linear algebra [72]. A few common semirings are $(+, *)$, $(\max, +)$, $(\min, \max)$, and $(\vee, \wedge)$, which were included in a set of standard graph algorithm building blocks by Mattson et al. [90].

[47] A stencil is a function applied to an array to produce a new array, where each new array cell is a function of a specified set of neighbors of the corresponding cell in the original array.

comes from the fact that each of these abstractions provides a mathematical way to represent a system of interconnected objects. Viewed as a graph, each component of a k-order tensor is a weighted k-degree hyperedge between the objects of the tensor modes. Viewed as a relation, each component is a row of a table with $k + 1$ columns.

The operations on tensors, graphs, and relations have many similarities because they all, underneath the hood, boil down to operations on sets of objects [73]. A graph vertex program that computes a vertex value as the linear combination of values from the vertex's neighbors (e.g., PageRank or a step of a breadth-first search) is the same as a matrix-vector multiplication. And an element-wise multiplication of two tensors is similar to an inner join in relational algebra: the former multiplies values in the tensor's intersection, whereas the latter returns the intersection.

Do we need three abstractions—relations, graphs, and tensors—to describe systems of objects? I believe that we do because each provides a different conceptual model that makes it easy to express and reason about a different type of operation on systems:[48]

- **Relational algebra** lets us create systems by combining, filtering, and projecting relations.
- **Graph operations** let us operate locally on the parts of a system, by mapping functions to vertices and edges and by traversing from vertices to their neighbors.
- **Tensor algebra** lets us operate globally on the system as a whole, by viewing it as a single part in a high-dimensional configuration space.

[48] An analogy comes from structured programming, where the `goto` statement can be replaced by half a dozen statements that compose [48]. For example, we routinely use several loop variants with the same expressive power such as `do`, `while`, and `for` because each is better at expressing one type of iteration.

It would be difficult to express global operations, such as factorization or matrix addition, in terms of graph operations or relational algebra. But it would also be difficult to force operations whose control flow depends on local states, such as Dijkstra's algorithm, into an extension of tensor algebra operations.

To highlight the similarities between graph operations, relational algebra, and tensor algebra, I give one example that shows the same operation expressed in each. The problem of counting triangles (3-clique) in a graph is one example of an important class of graph structure queries from data analytics. There have been papers on expressing and optimizing triangle counting in each of the three abstractions.[49] Latapy [86] proposes a twelve-step graph algorithm called *compact-forward* that achieves the optimal bound of $\Theta(m^{3/2})$, where m is the number of edges in the graph. Ngo et al. [98] show that relational triangle queries must be implemented with a single 3-way join algorithm to achieve the optimal worst-case bound $\Theta(m^{3/2})$. Relational triangle queries can be represented as the relational joins

[49] Although triangle counting works well in any of the three abstraction, in my opinion structural queries are easiest to express as relational algebra operations.

$$Q_{\triangle} = R(A, B) \bowtie S(B, C) \bowtie T(A, C),$$

where $\bowtie$ denotes an inner (intersecting) join, A, B, and C are sets, and Q, R, S, and T are relations. Moreover, Godsil and Royle [57, Corollary 8.1.3] show how to count triangles using linear algebra operations, by dividing by 6 the trace of the cube of the adjacency matrix of a graph[50]:

$$\frac{1}{6}\text{trace}(A^3).$$

Finally, Azad et al. [10] show that triangle counting can be further optimized by computing and then closing wedges. They first direct the graph by multiplying the lower and upper triangular parts of the adjacency matrix. Then, they element-wise multiply the result by the entire adjacency matrix:

$$A \odot (LU),$$

where $\odot$ is an element-wise multiplication, L is the lower triangular part of A, and U is its upper triangular part.

Because of the underlying similarities between their operations and because each abstraction operates on sets and their relationships, I believe sparse iteration theory can be generalized to support the union of sparse array and tensor operations, relational algebra, and many graph operations. This approach would make it possible to not only individually compile the operations in each abstraction but to also compile algorithms that transition between them.[51] The resulting unified sparse iteration theory would thus provide us with a compiler approach for sparse computation in general.

[50] The intuition is that each matrix multiplication does one step of a breadth-first search from each vertex; hence, in two steps, you get back to yourself through triangles. The number of triangles each vertex v takes part in is half of A^3_{vv} because one can traverse each triangle in two directions from each vertex. By counting the number of triangles of each vertex, we obtain three times the number of total triangles because three vertices partake in each triangle. Thus, the number of triangles is $\frac{1}{3}\frac{1}{2}\text{trace}(A^3) = \frac{1}{6}\text{trace}(A^3)$.

[51] The Simit programming language [77] that I worked on during graduate school, but which is not covered in this dissertation, is a first step in this direction. It lets users write programs that transition between graph and linear algebra abstractions. It demonstrated we can get performance, productivity, and portability across abstractions by introducing new programming language constructs to express these transitions.

Bibliography

[1] Martin 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, Xiaoqiang Zheng, 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. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[2] Frances E. Allen. Program optimization. In Mark I. Halpern and Christopher J. Shaw, editors, *Annual Review in Automatic Programming*, volume 5, pages 239–307. Pergamon Press, Elmsford, NY, 1969.

[3] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[4] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. *ACM SIGPLAN Notices*, 28(6):126–138, June 1993. doi:10.1145/173262.155102.

[5] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. *Principles and Pratice of Parallel Programming*, 26(7):39–50, April 1991. doi:10.1145/109626.109631.

[6] Gilad Arnold. *Data-Parallel Language for Correct and Efficient Sparse Matrix Codes*. PhD thesis, University of California, Berkeley, 2011.

[7] Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. Specifying and verifying sparse matrix codes. In *ACM SIGPLAN international conference on Functional programming*, pages 249–260. ACM, 2010. doi:10.1145/1863543.1863581.

[8] Krste Asanović, Rastislav Bodík, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, UC Berkeley, 2006.

[9] 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. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006. doi:10.1080/00268970500275780.

[10] Ariful Azad, Aydın Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 804–811. IEEE, October 2015. doi:10.1109/IPDPSW.2015.75.

[11] Eduardo F. D. Azevedo, Mark R. Fahey, and Richard T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *International Conference on Computational Science*, pages 99–106, Atlanta, Georgia, 2005. Springer. doi:10.1007/11428831_13.

[12] John W. Backus, R. J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, Harold Stern, Irving Ziller, Robert A. Hughes, and Roy Nutt. The FORTRAN automatic coding system. In *Western Joint Computer Conference*, pages 188–198, Los Angeles, California, February 1957. doi:10.1145/1455567.1455599.

[13] Brett W. Bader and Tamara G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, 2006. doi:10.1145/1186785.1186794.

[14] Brett W. Bader and Tamara G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *Journal on Scientific Computing*, 30(1):205–231, 2007. doi:10.1137/060676489.

[15] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Svenvan V. Haastregt, A. Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149, San Fransisco, March 2015. IEEE. doi:10.1109/PACT.2015.17.

[16] S Balay, J Brown, K Buschelman, V Eijkhout, W Gropp, D Kaushik, M Knepley, L Curfman Mcinnes, B Smith, and H Zhang. PETSc users manual. Technical report, Argonne National Laboratory, 2019. URL `https://www.mcs.anl.gov/petsc`.

[17] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Effcient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 163–202. Birkhäuser, Boston, MA, 1997. ISBN 978-1-4612-7368-4. doi:10.1007/978-1-4612-1986-6_8.

[18] Utpal Banerjee. *Data dependence in ordinary programs.* Masters thesis, University of Illinois at Urbana-Champaign, November 1976.

[19] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1990.

[20] Muthu Baskaran, Benoît Meister, Nicolas Vasilache, and Richard Lethin. Efficient and scalable computations with sparse tensors. In *IEEE Conference on High Performance Extreme Computing*, pages 1–6, Waltham, MA, 2012. IEEE. doi:10.1109/HPEC.2012.6408676.

[21] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 18:1–18:11, Portland, Oregon, 2009. ACM. doi:10.1145/1654059.1654078.

[22] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. Technical Report, September 2012.

[23] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi:10.1137/141000671.

[24] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, 1996.

[25] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *International Conference on Supercomputing*, pages 416–424. ACM, July 1993. doi:10.1145/165939.166023.

[26] Aart J. C. Bik, Peter J. H. Brinkhaus, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. The automatic generation of sparse primitives. *ACM Transactions on Mathematical Software*, 24 (2):190–225, 1998. doi:10.1145/290200.287636.

[27] Harvey W. Bingham and Earl W. Reigel. Parallelism exposure and exploitation in digital computing systems. Technical report, Burroughs Corp, June 1969.

[28] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21 (1):4–14, April 1994. ISSN 07437315. doi:10.1006/jpdc.1994.1038.

[29] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. PhD thesis, Massachusetts Institute of Technology, 1990.

[30] Guy E Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, CMU, 1992.

[31] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 141–154, August 1994. doi:10.1007/BFb0025866.

[32] Robert D. Blumofe, Charles E. Leiserson, Christopher F. Joerg, Keith H. Randall, Bradley C. Kuszmaul, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996. doi:10.1006/jpdc.1996.0107.

[33] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146, Budapest, March 2008. Springer. doi:10.1007/978-3-540-78791-4_9.

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

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

[36] Aydın Buluç, Tim Mattson, Scott McMillan, Jose Moreira, and Carl Yang. Design of the GraphBLAS api for c. *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017*, 0(0):643–652, 2017. doi:10.1109/IPDPSW.2017.117.

[37] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai Ching Tuan. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *International Conference on Supercomputing*, pages 37:1–37:12, Istanbul, Turkey, June 2016. ACM. doi:10.1145/2925426.2926278.

[38] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–169, 1988. doi:10.1007/BF00128175.

[39] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical report, University of Southern California, 2008.

[40] 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. TVM: An automated end-to-end optimizing compiler for deep learning this paper is included in the proceedings of the. In *Symposium on Operating Systems Design and Implementation*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

[41] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA): 123:1–123:30, November 2018. doi:10.1145/3276493.

[42] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, 1977.

[43] Phil Colella. Defining software requirements for scientific computing. presentation, 2004.

[44] Intel Corporation. Intel math kernel library developer reference. Technical Report 25, 2019.

[45] Leonaxdo Dagum and Ramesh Menon. OpenMP: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January 1998. doi:10.1109/99.660313.

[46] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006. doi:10.1137/1.9780898718881.

[47] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. ISSN 0098-3500.

[48] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11 (3):147–148, March 1968. doi:10.1145/362929.362947.

[49] R. I. M. Dunbar. Neocortex size as a constraint size in primates on group ecologically. *Journal of Human Evolution*, 20:469–493, March 1992. doi:10.1016/0047-2484(92)90081-J.

[50] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936. doi:10.1007/BF02288367.

[51] Albert Einstein. The foundation of the general theory of relativity. *Annalen der Physik*, 354: 769–822, 1916.

[52] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. 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):2293–2309, 2013. doi:10.1002/jcc.23377.

[53] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988. doi:10.1051/ro/1988220302431.

[54] Richard Feynman, Robert B Leighton, and Matthew L Sands. *The Feynman Lectures on Physics*, volume 3. Addison-Wesley, 1963. ISBN 9780465025015.

[55] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and Jose M.F. Moura. *SPIRAL: Extreme Performance Portability*, volume 106. IEEE edition, 2018. doi:10.1109/JPROC.2018.2873289.

[56] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992. doi:10.1137/0613024.

[57] Chris Godsil and Gordon Royle. *Algebraic Graph Theory*. Graduate Texts in Mathematics. Springer, New York, 2001. ISBN 9780387952208.

[58] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970. doi:10.1007/BF02163027.

[59] Peter Gottschling, David S. Wise, and Michael D. Adams. Representation-transparent matrix algorithms with scalable performance. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 116–125, New York, NY, USA, June 2007. ACM. doi:10.1145/1274971.1274989.

[60] Gaël Guennebaud, Benoît Jacob, and Others. Eigen v3, 2010. URL `http://eigen.tuxfamily.org`.

[61] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3), September 1978.

[62] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970. doi:10.1145/362258.362278.

[63] Larry Hardesty. Faster big-data analysis. *MIT News*, 2017. URL http://news.mit.edu/2017/faster-big-data-analysis-tensor-algebra-1031.

[64] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. ExTensor: An accelerator for sparse tensor algebra. *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 319–333, October 2019. doi:10.1145/3352460.3358275.

[65] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu Williams, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of Trilinos. Technical report, Sandia National Laboratories, 2003.

[66] Frank L Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189, April 1927. doi:10.1002/sapm192761164.

[67] Eun-jin Im and Katherine Yelick. Model-based memory hierarchy optimizations for sparse matrices. In *Workshop on Profile and Feedback-Directed Compilation*, pages 1–10, 1998.

[68] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, 1962. ISBN 0-471430-14-5.

[69] Eric Jones, Travis E. Oliphant, Pearu Peterson, and Others. SciPy: Open source scientific tools for python, 2001. URL http://www.scipy.org/.

[70] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967. doi:10.1145/321406.321418.

[71] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega library. Technical report, March 1995.

[72] Jeremy Kepner and John R. Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011. doi:10.1137/1.9780898719918.

[73] Jeremy Kepner and Hayden Jananthan. *Mathematics of big data: Spreadsheets, databases, matrices, and graphs*. MIT Press, 2018. ISBN 9780262038393.

[74] Gary A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206. ACM, October 1973. doi:10.1145/512927.512945.

[75] David R. Kincaid and David M. Young. The development of a computer package for solving a class of partial differential equations by iterative methods. *Mathematics and Computers in Simulation*, 17(3):186–191, 1975. doi:10.1016/S0378-4754(75)80051-6.

[76] David R. Kincaid and David M. Young. A brief review of the ITPACK project. *Journal of Computational and Applied Mathematics*, 24(1-2):121–127, 1988. doi:10.1016/0377-0427(88)90347-0.

[77] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Transactions on Graphics*, 35(2):20:1–20:21, 2016. doi:10.1145/2866569.

[78] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA): 77:1–77:29, October 2017. doi:10.1145/3133901.

[79] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. In *International Symposium on Code Generation and Optimization*, pages 180–192, Washington, DC, February 2019. IEEE Press. ISBN 978-1-7281-1436-1. URL http://dl.acm.org/citation.cfm?id=3314872.3314894.

[80] Donald Ervin Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Pearson Education, 2nd edition, 1973. ISBN 0-201-89685-0.

[81] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009. doi:10.1137/07070111X.

[82] Yehuda Koren. Collaborative filtering with temporal dynamics. *Communications of the ACM*, 53(4):89–97, April 2010. doi:10.1145/1721654.1721677.

[83] Vladimir Kotlyar. *Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs*. PhD thesis, Cornell University, 1999.

[84] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Euro-Par Parallel Processing*, pages 318–327. Springer, Passau, Germany, 1997. doi:10.1007/BFb0002751.

[85] Leslie Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2): 83–93, February 1974. doi:10.1145/360827.360844.

[86] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1-3):458–473, November 2008. doi:10.1016/j.tcs.2008.07.017.

[87] Chris Leary and Todd Wang. XLA: Tensorflow, compiled! TensorFlow Dev Summit, February 2017.

[88] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[89] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. *International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 238–252, 2019. doi:10.1109/SC.2018.00022.

[90] Tim Mattson, David Bader, Jon Berry, Aydın Buluç, Jack Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Michael Stonebraker, Steve Wallach, and Andrew Yoo. Standards for graph algorithm primitives. In *IEEE High Performance Extreme Computing Conference*, pages 1–2. IEEE, 2013. doi:10.1109/HPEC.2013.6670338.

[91] Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172. ACM, 2013. doi:10.1145/2507157.2507163.

[92] John Michael McNamee. Algorithm 408: A sparse matrix package. *Communications of the ACM*, 14(4):265–273, 1971. doi:10.1145/362575.362584.

[93] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 58:1–58:12, Salt Lake City, Utah, 2016. IEEE Press. doi:10.1109/SC.2016.57.

[94] Stanley Milgram. The small world problem. *Psychology Today*, 2(1):60–67, 1967.

[95] Ravi Mirchandaney, Joel H. Saltz, Roger M. Smith, David M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. *Proceedings of the International Conference on Supercomputing*, pages 140–152, July 1988. doi:10.1145/55364.55378.

[96] Yoichi Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, University of Illinois at Urbana-Champaign, February 1971.

[97] National Institute of Standards and Technology. Matrix market file formats. File Format Specification, August 2013. URL `http://math.nist.gov/MatrixMarket/formats.html`.

[98] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 37–48, Scottsdale, Arizona, 2012. ACM. doi:10.1145/2213556.2213565.

[99] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. Sampled dense matrix multiplication for high-performance machine learning. In *IEEE International Conference on High Performance Computing*, pages 32–41, Bengaluru, India, December 2018. IEEE. doi:10.1109/HiPC.2018.00013.

[100] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P Sadayappan. Load-balanced sparse mttkrp on gpus. In *IEEE International Parallel and Distributed Processing Symposium*, pages 123–133. IEEE, May 2019. doi:10.1109/IPDPS.2019.00023.

[101] NVIDIA. CUDA programming model v10.1, 2019.

[102] Travis E. Oliphant. *A guide to NumPy*. Trelgol Publishing USA, 2006.

[103] David Lorge Parnas. Software aspects of strategic defense systems. *Communications of the ACM*, 28(12):1326–1335, December 1985. doi:10.1145/214956.214961.

[104] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 4–13. IEEE, November 1991. doi:10.1145/125826.125848.

[105] William Pugh and Tatiana Shpeisman. SIPR: A new framework for generating efficient code for sparse matrix computations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 213–229. Springer, August 1998. doi:10.1007/3-540-48319-5_14.

[106] Markus Püschel, José M.F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–273, 2005. doi:10.1109/JPROC.2004.840306.

[107] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):1–12, 2012. doi:10.1145/2185520.2335383.

[108] Eric Steven Raymond. Philosophy. In *The art of Unix programming*, chapter 1, pages 24–51. Addison-Wesley Professional, 2003. ISBN 0131429019.

[109] Gregorio Ricci-Curbastro and Tullio Levi-Civita. Méthodes de calcul différentiel absolu et leurs applications. *Mathematische Annalen*, 54(1-2):1900, 1900. doi:10.1007/BF01454201.

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

[111] Yousef Saad. Sparsekit: a basic tool kit for sparse matrix computations. Technical report, 1994.

[112] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003. doi:10.1137/1.9780898718003.

[113] Nobuo Sato and W. F. Tinney. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE Transactions on Power Apparatus and Systems*, 82(69):944–950, 1963. doi:10.1109/TPAS.1963.291477.

[114] Ryan Senanayake, Fredrik Kjolstad, Changwan Hong, Shoaib Kamil, and Saman Amarasinghe. A unified iteration space transformation framework for sparse and dense tensor algebra. Technical report, 2019. URL `http://arxiv.org/abs/2001.00532`.

[115] Shaden Smith and George Karypis. Tensor-matrix products with a compressed sparse tensor. In *Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–7. ACM, 2015. doi:10.1145/2833179.2833183.

[116] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium*, pages 61–70. IEEE, May 2015. doi:10.1109/IPDPS.2015.27.

[117] Shaden Smith, Jee W Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools. Open Dataset, 2017. URL `http://frostt.io/`.

[118] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI: The Complete Reference*. The MIT Press, 2 edition, 1998. ISBN 9780262692151.

[119] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014. doi:10.1016/j.jpdc.2014.06.002.

[120] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings International Conference for High Performance Computing, Networking, Storage, and Analysis*, pages 47:1–47:14, Denver, Colorado, 2017. ACM. doi:10.1145/3126908.3126971.

[121] Paul Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell, 1997.

[122] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018. doi:10.1109/JPROC.2018.2857721.

[123] The MathWorks Inc. MATLAB.

[124] William F Tinney and John W Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967. doi:10.1109/PROC.1967.6011.

[125] NVIDIA V10.1.243. cusparse software library, 2019.

[126] Harmen L. A. Van Der Spek and Harry A. G. Wijshoff. Sublimation: Expanding data structures to enable data instance specific optimizations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 106–120, Houston, Texas, 2010. Springer. doi:10.1007/978-3-642-19595-2_8.

[127] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. Technical report, June 2018. URL `http://arxiv.org/abs/1802.04730`.

[128] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 521–532, 2015. doi:10.1145/2737924.2738003.

[129] Sven Verdoolaege. isl: An integer set library for the polyhedral model. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6327 LNCS:299–302, 2010. ISSN 03029743. doi:10.1007/978-3-642-15582-6_49.

[130] Doran K. Wilde. *A Library for Doing Polyhedral Operations by*. Masters thesis, Oregon State University, December 1993.

[131] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, November 2007. IEEE. doi:10.1145/1362622.1362674.

[132] Michael Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, November 1986. doi:10.1007/BF01407876.

[133] Michael J Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, November 1982.

[134] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Fred Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974. doi:10.1145/355616.364017.

[135] Carl Yang, Aydın Buluç, and John D Owens. Implementing push-pull efficiently in GraphBLAS. In *International Conference on Parallel Processing*, pages 89:1–89:11, Eugene, OR, August 2018. ACM. doi:10.1145/3225058.3225122.

[136] Zecheng Zhang, Xiaoxiao Wu, Naijing Zhang, Siyuan Zhang, and Edgar Solomonik. Enabling distributed-memory tensor completion in python using new sparse tensor kernels. Technical report, October 2019. URL http://arxiv.org/abs/1910.02371.

[137] Huasha Zhao. *High Performance Machine Learning through Codesign and Rooflining*. PhD thesis, University of California, Berkeley, September 2014.