# A Framework for Computing on Sparse Tensors based on Operator Properties

by

Rawn Tristan Henry

S.B., Massachusetts Institute of Technology (2019)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author ............................................................
Department of Electrical Engineering and Computer Science
May 18, 2020

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

Accepted by ........................................................
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# A Framework for Computing on Sparse Tensors based on Operator Properties

by

Rawn Tristan Henry

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Tensor operations have been traditionally limited to addition and multiplication operations. For operations of sparse tensors, these semantics were extended to account for the fact that tensors usually omit zero values. However, there are many operators with a rich semantics of operator properties that can be used in dense and sparse tensor computations.

This work addresses the problem of generating code for computing on a mix of sparse and dense tensors based on the properties of the operators on those tensors. I introduce the concept of a fill value to each tensor so that the data can be sparse on non-zeros. I show how to reason about the operator properties, along with the fill values of the input tensors in order to construct an IR describing how to iterate over these tensors. I show how we can take advantage of the operator properties to perform useful optimizations for both iterating over tensors and performing reductions. Lastly, I show how a user can leverage set notation to directly describe to a compiler how it should iterate over sparse tensors.

The ideas discussed in this work have been prototyped in the open-source TACO system. The API used makes operator properties and tensor fill values have to be explicitly provided by the user. However, it makes the TACO system much more flexible. I show how the primitives exposed in this work allows one to efficiently perform several graph algorithms by drawing on the literature about GraphBLAS. In the evaluation section, we benchmark this system against the SuiteSparse implementation of GraphBLAS on a variety of graph algorithms to demonstrate its performance.

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

# Acknowledgments

I would like to thank my research advisor, Saman Amarasinghe for all of his support over the last year and a half. Additionally, I like to thank Fredrik Kjolstad for his continued mentorship and helpful advice during our time working together. I also would like to thank Stephen Chou for all of the time spent discussing and fleshing out ideas with me. I also want to thank Stephen for seemingly always being available to answer my questions on slack! I consider myself immensely fortunate to be able to work with such devoted and talented individuals and I am very happy to have had the opportunity to learn from all of them. This work would also not have been possible without the discussions and reviews of the following people: Ryan Senanayake, Peter Ahrens, Shaoib Kamil and Amalee Wilson. Thank you so much for all of help!

I would like to thank my best friends, Dylan, Lucas and Bobby as well as my partner, Shalni. You have been the greatest friends and I will always cheerish the moments that we spent together.

But I am most grateful to my mother, Marlene Nicholas. I remember when you defended my desire to be a garbage collector when I was 6 despite fierce opposition from everyone else in our family. It reinforced what I have known my entire life - that I will always have your support. Your unwavering support allowed me to chase my dreams, culminating in me being at MIT, the school of my dreams pursuing a career I have been interested in since I was 15. Thank you for everything you have done for me. You mean everything to me.

# Contents

# List of Figures

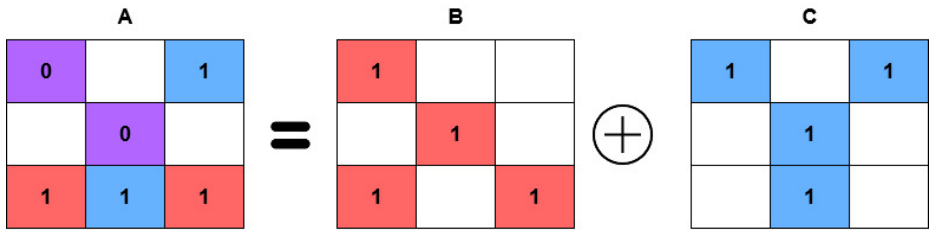
# List of Tables

# Chapter 1

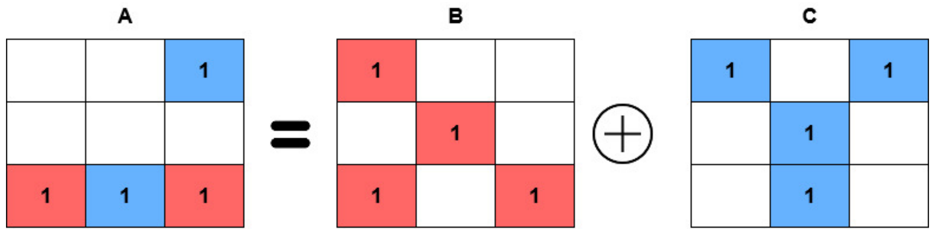
# Introduction

Graph algorithms have been traditionally performed using special frameworks such as Ligra [31], GraphIt [41] and Gunrock [38]. There has recently been a push to use linear algebra primitives to support graph algorithms [10, 19]. Unfortunately, the current work on sparse compilers cannot be leveraged to efficiently perform these algorithms due to three key missing features. The first is that current sparse compilers always assumes that tensors are sparse over zero. Secondly, sparse compilers are limited to using the $(+, \times)$ semi-ring which limits their applicability. Lastly, sparse compilers can only efficiently perform expressions that require iterating over the union or intersection of the operands of an operator.

In this thesis, I propose solutions for all three of the previously mentioned problems and also develop a framework for allowing computation outside of normal semi-rings. This will allow works by Kjolstad et al. on the Tensor Algebra Compiler (TACO) [21, 20, 12] to be used to tackle a broader set of problems. For example, this work allows Breadth First Search, Page Rank, Floyd-Warshall, Bellman-Ford and Triangle Counting graph algorithms to all be in expressed TACO by defining the appropriate semi-rings [19]. Some of these algorithms assume that the matrices and vectors are sparse over $\infty$ instead of 0 illustrating examples where non-zero sparsity is desired. Additionally, the Viterbi algorithm which is used extensively in communication and speech recognition can be expressed over a $(\min, +)$ semi-ring [35] and will now be expressible in TACO. This work, in addition to TACO's scheduling language [30] could allow users to achieve good performance on a wide variety of hardware for graph and machine learning applications.



(a) XOR operation as it would be performed in current TACO. The compiler would iterate over the Union of $B$ and $C$ and produce explicit zeros in the result matrix $A$. This is due to the fact that the loop structures the compiler can generate only supports unions and intersections.



(b) XOR operation as it would be performed by TACO after it is modified by this work. The compiler can support iterating over the union of $B$ and $C$ while excluding the intersection. This helps to reduce the memory footprint of the result tensor.

Figure 1-1: XOR with explicit zeros stored (top) vs without (bottom)

Current sparse compilers tie the optimizations they support to specific operations. An example of this is that when adding two sparse tensors $A$ and $B$, the compiler only needs to iterate over the union of $A$ and $B$ and can omit iterating over the space where both $A$ and $B$ are 0. While this is great for most cases it limits the generality of the current sparse compiler approach. Current compilers only support union and intersection operators meaning that they cannot cleanly support an operation such as an exclusive or. This leaves some performance on the table for such an operation since for a sparse result, one ends up storing needless explicit zeros in the result tensor as shown in Figure 1-1. As a result, I will first show how we can generalize the current compiler approach to generate code to iterate over any set expression. I will then show how we can decouple the implementation of the scalar operator with its iteration pattern. I will also demonstrate how a compiler can reason about the properties of scalar operations along with the fill values of the input tensors in order to automatically infer the iteration pattern it should use when computing the result. Lastly, I will generalize the ideas in the sparse compilation literature to allow for a tensor to be sparse on any value of the primitive types that the compiler supports. The value compressed out of the tensor will be referred to as the **fill value** from here onward. These new mechanisms will allow sparse compilers to be applied to a much larger variety of domains.

I have prototyped these ideas in the open-source TACO sparse tensor algebra compiler [21] which generates code for any sparse tensor algebra expression. The structure of the prototype is shown in Figure 1-3. I expose the abstractions described here by extending the tensor Index Notation in TACO to include a new generic operator class. This class takes in a scalar description of the operation along with its properties as well as an algebra to describe the iteration space. This provides a clean interface which decouples the scalar computation from the iteration pattern. Figure 1-2 shows the set patterns that TACO can now iterate over as a result of this work for binary operators. Prior to this work, only the two patterns above the red line in Figure 1-2 could be iterated over by the compiler for binary operators. However, it is now possible to generate code that iterates over all 16 of the possible subsets described in Figure 1-2 for binary operators. Further, this work generalizes to an arbitrary number of sets meaning that the prototype can now generate code that iterates over any subset of tensors given as operands to an operator.
My specific contributions are as follows:

1. Introduce the ability to define generic semi-ring and array operations in TACO,

2. Extend the TACO IR called merge lattices [21] or iteration lattices [22] so that any set can be represented in the new IR instead of only intersections and unions as shown in Figure 1-2,

3. Demonstrate that iteration patterns and optimizations such as short circuiting can be inferred from operator properties, and

4. Generalize tensors to allow them to be sparse over any value instead of just zero,

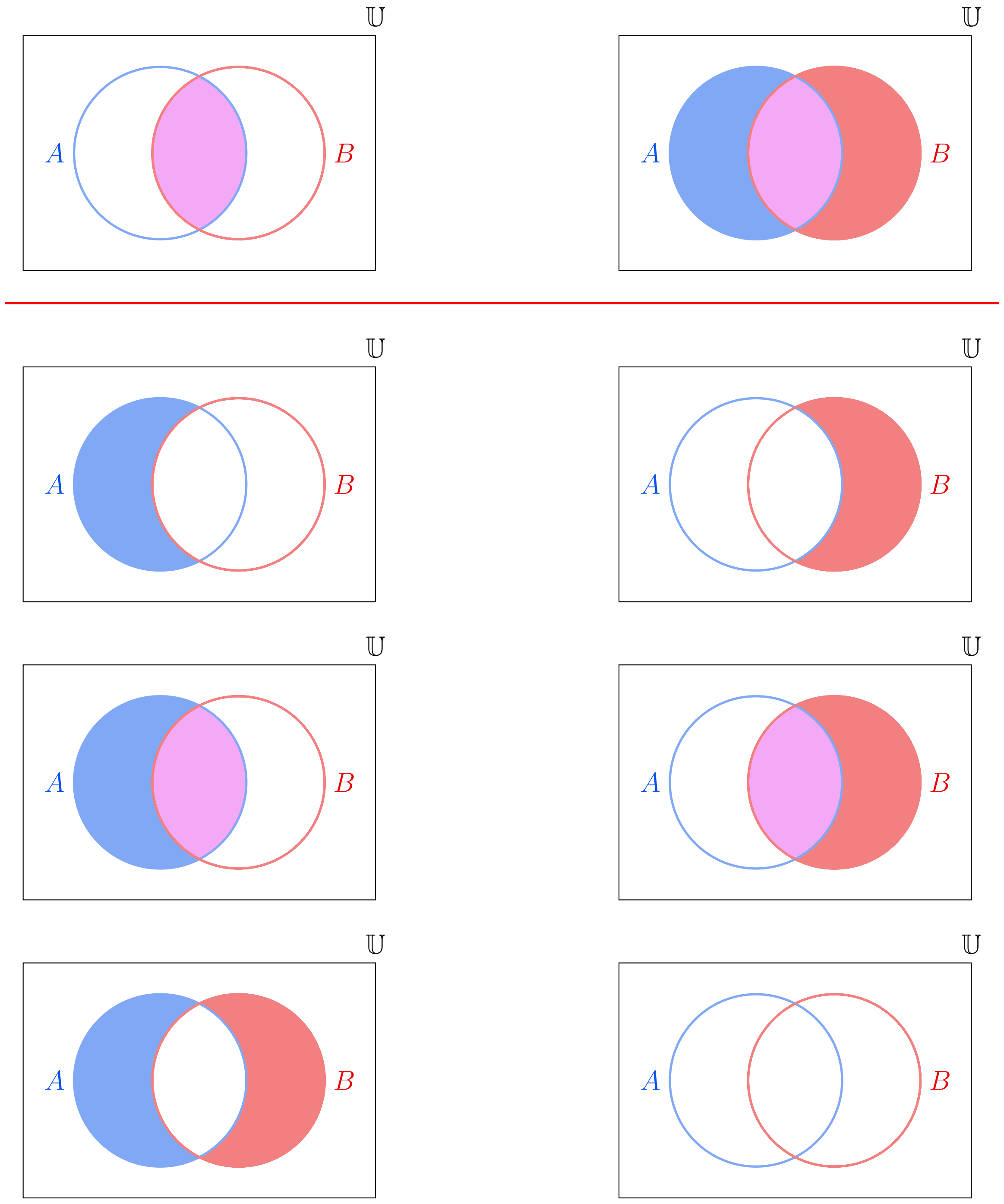


Figure 1-2: Figure showing binary sets one may iterate over excluding sets that include the universe. Previous work can only emit code to iterate over the sets above the red line. This work adds the 6 other set iteration patterns along with the 8 not shown that include the universe. In total, the remaining 14 of 16 binary iteration patterns can all be expressed with the addition of this work. The ideas expressed in this thesis also generalizes to an arbitrary number of sets.

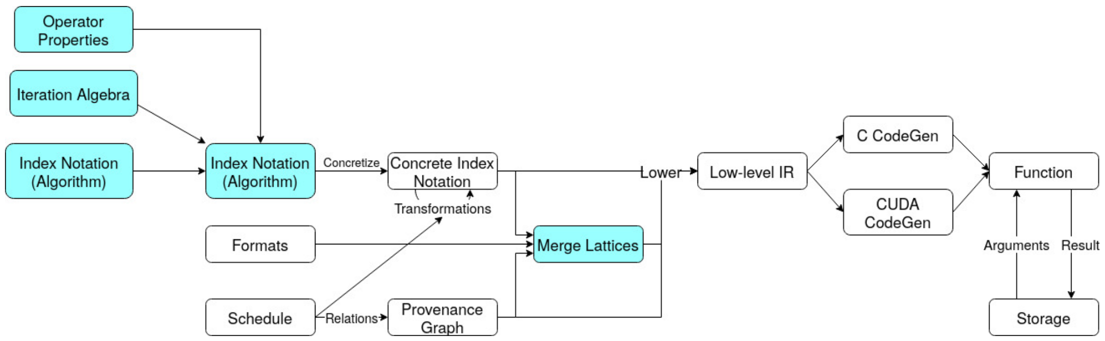


Figure 1-3: The structure of the prototype based on the Tensor Algebra Compiler (TACO). The user specifies the result in index notation along with a format and optionally a schedule. TACO transforms these through various intermediate representation before generating code that can be used to compute the expression on input tensors. The parts of TACO heavily modified in this work are highlighted in blue. This work adds two new components - operator properties and iteration algebra. The fact that the iteration algebra can be specified separately is what allows this work to separate the scalar computation from the actual loop structure generated.

The rest of this thesis is organized as follows:

**Chapter 2 - Motivating Example** describes two example problems and how the general ideas in this work allows TACO to be used to solve these problems. I describe a Direction Optimizing Breadth First Search [5, 40] and an algebraic bellman-ford formulation [19] and how this work allows TACO to solve these problems.

**Chapter 3 - Array Index Notation** describes a generalized language for computing on sparse arrays.

**Chapter 4 - Iteration Spaces** describes iteration spaces and the extensions to iteration space algebra and iteration lattices. These intermediate representations are crucial to the generality this work aims to achieve.

**Chapter 5 - Properties** describes the properties of operators supported in the prototype and how they influence iteration spaces. An explanation of how to infer iteration spaces from the scalar properties is described along with some limitations of only relying on properties. Demonstrates that there are some iteration spaces that cannot be derived from operator properties alone.

**Chapter 6 - Code Generation** describes extensions to the TACO code generator to support the new IR as well as the new construction rules for iteration lattices.

**Chapter 7 - Operator API** describes the interface provided for defining new generic operators to TACO.

**Chapter 8 - Evaluation** compares the performance of the generated code to a GraphBLAS library, SuiteSparse for the linear algebra graph algorithms and against hand-tuned works where available.

**Chapter 9 - Related Work** discusses this work in the context of previous work done with sparse tensor compilation.

**Chapter 10 - Conclusion and Future Work** concludes and provides guidance for future extensions of this work.

**Appendix A - Operator Implementations** provides the implementations of the operators used in this thesis.

# Chapter 2

# Motivating Example

In this chapter, I demonstrate a potential use case of this framework. I show how one can use the general ideas here to execute two sample graph algorithms - breadth first search and a single source shortest path (sssp) algorithm. This work enables taco to perform two new general optimizations needed to get good performance on graph algorithms performed using linear algebra. Firstly, this work allows TACO to perform masking on both sparse and dense tensors. This means that the compiler can skip performing computations based on the value of the mask. An example of this is seen in BFS allowing TACO to omit computing entire dot products. Secondly, this work allows TACO to realize when it can short circuit reductions. BFS also illustrates an example of this optimization. The ideas in this thesis also allow the expression of new algorithms that were not possible before (sssp) by taking advantage of the fact that the fill values of the tensors can be user specified.

## 2.1 Preliminaries

A **monoid** is a set $A$ with a single associative binary operator and an identity element. For example, the real numbers $\mathbb{R}$ under binary addition. From the definition, the set is the real numbers $\mathbb{R}$, the binary operator is addition and the identity element is 1. A **semiring** is a set $\mathbb{S}$ along with two binary operators $\oplus$ and $\odot$. These operators must have the following algebraic properties:

1. $(\mathbb{S}, \oplus)$ is a commutative monoid.

2. $(\mathbb{S}, \odot)$ is a monoid

3. $\odot$ distributes over $\oplus$

4. $\oplus$ is annihilated by the identity element of $\oplus$.

In this work, I use the notation $(\oplus, \odot, \mathbb{S})$ to denote a semi-ring. When $\mathbb{S}$ is omitted, I assume that it is the real numbers. Examples of semi-rings are $(+, \times)$, $(\min, +)$ and $(\max, +)$. An example of a semi-ring not over the reals is $(\wedge, \vee, \mathbb{B})$ where $\mathbb{B}$ is only contains the elements $\{0, 1\}$.

## 2.2 Linear Algebra Breadth First Search

I consider the breadth-first search (BFS) on graphs $G(V, E)$ where $V$ is the set of vertices in $G$ and $E$ is the set of all ordered pairs $(u, v)$ with $u, v \in V$. A BFS on a graph $G$ is an exploration of the graph that starts at some source node and explores nodes at a the current depth level before moving to nodes at the next depth level. The depth level is the number of edges needed to be traversed to get from the source to the current set of nodes. For example, the source has a depth level of 0 and all neighbors of the source have a depth of one since there is one edge the source and its neighbors them. The algorithm terminates when no new vertices can be reached from the current depth level.

There are two ways in which one can perform a breadth first search. The first is by "push" or "top-down" which is the traditional way a BFS is performed. This means that we start from a frontier of vertices and add all of the unvisited neighbors of the frontier to the next frontier. The second is "push" or "bottom-up" meaning that we start at all unvisited nodes and attach an unvisited node to the frontier if there is an edge from the frontier to that node.

Both versions of BFS can be expressed using sparse linear algebra. Graphs are usually stored as adjacency matrices of size $V \times V$ with a one in the cell $(u, v)$ if there is an outgoing edge from $u$ to $v$. So, the rows of the matrix contain the source nodes for each edge while the columns of the matrix consists of the destination nodes for the edges. Usually, most nodes only have a few neighbors so the adjacency matrix representation is usually sparse containing only $E$ edges where $E \ll |V|^2$. I will use $A$ to refer to the adjacency matrix representation of the graph from this point onward.

For the linear algebra formulation of the problem, a vector $f$ is used to represent the current frontier. Typically, we want the frontier vector $f$ to be sparse when there are only a few vertices available. This allows one to iterate over the vector to find the neighboring vertices of those in the frontier by querying the graph. That is, we want to find all of the destination nodes that can be reached from every vertex $u$ in the frontier. This is the push direction and corresponds to reading a row of the adjacency matrix $A$ since row $u$ has all the outgoing edges of the vertex $u$. Since $A$ is sparse, then for the push direction, we want $A$ to be stored as a CSR matrix so that the rows can be quickly queried. In the literature, the adjacency matrix $A$ is usually transposed and stored in the CSC format (which is the same as storing $A$ in CSR) so that the sparse vector times sparse matrix (SpVSpM) kernel can be applied to perform the push step.

However, once the frontier gets large it is beneficial to store its dense representation and to try to attach unvisited nodes to the frontier. Intuitively, this is because once the frontier is large enough, there may be more edges going into it than going out meaning that if we switch directions, less edges need to be traversed. In this scheme, we iterate over the graph to find the source of an edge $u$ given its destination vertex $v$. Since all the sources of a given vertex are in the columns of the adjacency matrix, it is beneficial to store $A$ as a CSC matrix for this scheme. This is the pull direction. Generally, graph frameworks keep both the CSC and CSR versions in memory when a graph is not symmetric. Again in the literature, the original adjacency matrix is often transposed and stored as a CSR matrix so that the well studied sparse matrix times dense vector (SpMV) kernel can be applied.

In matrix algebra, both directions perform the expression $v = A^T \times f$. The only difference is that in the push direction, the vectors $v$ and $f$ are sparse while $A^T$ is stored in the CSC format. However, in the pull direction, the vectors $v$ and $f$ are dense and $A^T$ is stored in the CSR format.

For both directions, BFS implementations use the $(\vee, \wedge, \mathbb{B})$ semi-ring [40]. Both versions also update an dense vector tracking the earliest depth at which each node was seen. A depth of 0 means the node is unvisited. This vector doubles as a mask that can be used to avoid doing work for vertices that have already been visited.

State of the art work uses both directions during the BFS iterations depending on heuristics to determine which would be more beneficial. TACO can already perform this optimization by using the same expression but with different formats. However, the pull direction leaves opportunities for optimization on the table. Namely:

1. There is no way to instruct the framework to only try to attach unvisited nodes to the frontier without explicitly storing a sparse mask of all the unvisited nodes. Current work already tracks the set of visited vertices in a dense vector and updates this set after each iteration. This work in this thesis introduces general iteration spaces allowing a user to express this masking optimizations in terms of iteration spaces.

2. Once we find that a node can be attached to the frontier, we can exit the inner product early. TACO has no way to currently recognize this and perform this optimization. This work solves this by allowing TACO to reason about operator properties.

The code TACO would currently generate to perform an iteration of BFS in the pull direction is shown in Figure 2-1a. The code in Figure 2-1b demonstrates optimization one which is referred to ask masking [40]. We avoid expanding any visited edges which wastes work on unneeded dot products while reusing the dense visited vector. Chapter 4 describes the concept of iteration spaces which allows a user to express concepts like this to the compiler.

The code in Figure 2-1c demonstrates optimization two applied to the kernel. This optimization allows early exit from the dot product and lead to the biggest speedup in the performance of the pull direction of BFS [40]. TACO can realize this if a

```
for(int32_t i = 0; i < f_dimension; i++) {
    for(int32_t jA = A_pos[i]; jA < A_pos[(i+1)]; jA++){
        int32_t j = A_crd[jA];
        v_vals[i] = v_vals[i] + A_vals[jA] * f_vals[j];
    }
}
```

(a) Normal SpMV

```
for(int32_t i = 0; i < f_dimension; i++) {
    if(!(visited_vals[i] != 0)) {
        for(int32_t jA = A_pos[i]; jA < A_pos[(i+1)]; jA++){
            int32_t j = A_crd[jA];
            v_vals[i] = v_vals[i] + A_vals[jA] * f_vals[j];
        }
    }
}
```

(b) SpMV with mask. Note that the dot product is performed only when visited_vals is 0 since we do not need to try to attach a node that has already been visited to the next frontier.

```
for(int32_t i = 0; i < f_dimension; i++) {
    if(!(visited_vals[i] != 0)) {
        for(int32_t jA = A_pos[i]; jA < A_pos[(i+1)]; jA++){
            int32_t j = A_crd[jA];
            v_vals[i] = v_vals[i] + A_vals[jA] * f_vals[j];
            if (v_vals[i] == 1) {
                break;
            }
        }
    }
}
```

(c) SpMV with masking and short circuiting

Figure 2-1: Code showing several ways SpMV can be used to perform one BFS step. All of the matrices are assumed to be sparse and stored in the CSR format. In the above, A is assumed to be the transposed adjacency matrix, v is assumed to be the next frontier and f is assumed to be the current frontier. Additionally, there is a dense visited vals vector which stores the lowest depth at which each vertex was seen.

user replaces $+$ with $|$ and $\times$ with & so that we perform the computation on the Boolean semi-ring. The graph interpretation of exiting early is that once we know an unvisited vertex can be attached to the frontier by one edge, we are sure that we can add it to the next frontier and can stop exploring other edges incident to that vertex. The linear algebra justification is that the OR operator has an annihilator of 1. An annihilator of an operator has the property that $a$ op $x = a\ \forall x$. Therefore once the reduction value ends up being 1, we can be sure the result of the reduction will be 1 since no other value can cause it to change.

While the code in Figure 2-1a can already be generated, the ideas presented in this thesis allows the compiler to generate the implementations shown in the rest of the code in Figure 2-1. Next, I describe another graph algorithm that leverages one more feature of this work.

## 2.3 Linear Algebra Bellman Ford

The Bellman-Ford algorithm solves the single source shortest path problem (sssp). Given a graph $G(V, E)$ with edge weights $w$ and a source $s \in V$, the algorithm determines the shortest path distances $\Delta(s, v)$ for all $v \in V$ and the corresponding paths assuming that there are no negative weight cycles.

Intuitively, Bellman-Ford proceeds by performing a series of edge relaxations. For each vertex $v$ it stores an estimate $d(v)$ for the shortest path distance maintaining the invariant that $d(v) \geq \Delta(s, v)$. Relaxing an edge $(u, v)$ means that $d(v) = \min(d(v), d(u) + W(u, v))$ where $W$ is the adjacency weight matrix for the graph. $|V|$ iterations are performed and every iteration relaxes every edge in the graph.

In algebra terms, relaxing all edges incident on a vector $v$ can be expressed as:

$$\Delta_k(s, v) = \min_u(\Delta_{k-1}(s, u) + W(u, v))$$

where $\Delta_k$ represents the shortest path using at most $k$ hops. This is kept in the distance estimate vector $d$. Therefore, the above can be translated to a dot product between $d$ and a row of the weight matrix $W$ where we reduce using min and replace the usual $\times$ with the $+$ operator. Unlike usual operators, $\infty$ annihilates $+$ and also is the identity of min. Therefore, our tensors must be sparse over $\infty$ for TACO to realize that it only needs to iterate over the defined values in $W$ when performing this dot product. The initial distance estimate $d$ is initialized as follows:

$$d_0(v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{otherwise} \end{cases}$$

The algorithm computes the SpMV expression $d_k = d_{k-1} \min .{+} W$ for $N$ iterations then terminates. The notation $\oplus.\odot$ implies that we are operating on a semiring with $\oplus$ as the usual plus operator and $\odot$ as the usual multiply operator. The structure of the code is exactly the same is shown in Figure 2-1a except the plus and multiply operators are replaces with min and + respectively in the code above. TACO now recognizes this by reasoning about fill values and operator properties. This shows the benefit of TACO being able to reduce and compute on arbitrary scalar operators based on their properties.

Like BFS, it is also crucial that the algebraic Bellman-Ford changes direction. It uses a SpVSpM algorithm when the input vector is sparse then switches to SpMV when the number of non-zeros in the input vector is above some threshold. In order to switch correctly, TACO must use the general fill value of the sparse vector so that the dense vector is correctly defined everywhere its sparse representation was missing a value.

Finally, Bellman-Ford also benefits from the general concept of masking but in a different form to the one shown in BFS. In Bellman-Ford, active vertices are those whose shortest path weights changed during the previous iteration of the algorithm. We only need to consider active vertices when performing a relaxation since other edge weight would have been accounted for in previous iterations. This means that ideally, when performing $d_k = d_{k-1} \min . + W$ that the vector $d_{k-1}$ only consists of vertices that changed during the previous iteration since this is the only new information we need to take into consideration.

To make the discussion of masking clearer, I introduce a temporary $t$ so that $t = d_{k-1} \min . + W$. Now setting $d_k$ to be the active vertices is done as follows

$d_k = t \times (t < d_{k-1})$ so now $d_k$ only consists of vertices that were updated. I note again that having properties allows TACO to realize that for the mask sub-expression $(t < d_{k-1})$, it only needs to iterate over the union when $t$ and $d_{k-1}$ are sparse over $\infty$. Additionally, a new multiply operator can be defined to perform final masking multiply with $\infty$ as the annihilator so that TACO can set $d_k$ to the intersection of the mask and $t$.

The examples shown in this chapter demonstrate how the general ideas expressed in this work can be applied to graph algorithms. General iteration spaces and fill values are two key elements that allow TACO to be extended to this domain. The rest of this thesis describes how this generality is achieved in TACO and the APIs through which it can be used.

# Chapter 3

# Array Index Notation

**Tensor Index Notation** [21] 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 as shown in Figure 3-1. 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 in the corresponding tensor modes. For example, in Figure 3-1, the index variable $i$ is bound to coordinates in mode zero of $A$ and mode zero of $B$ while $l$ corresponds to coordinates in mode two of $A$ and mode one of $C$.

$$A_{ijl} = \sum_k B_{ijk} C_{kl} \qquad\qquad A_{0,3,1} = \sum_k B_{0,3,k}\, C_{k,1}$$

Figure 3-1: 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 $(0, 3, 1)$.

Array Index Notation is an extension to Tensor Index Notation [22]. It allows a user to perform arbitrary operations on sparse or dense arrays. The main difference between the two notations is that Tensor Index Notation bundles the iteration pattern with the operator and assumes that all input tensors are sparse over zero. An example

```c
int32_t pa1 = a1_pos[0];
int32_t pa1_end = a1_pos[1];
int32_t pb1 = b1_pos[0];
int32_t pb1_end = b1_pos[1];

while (pa1 < pa1_end && pb1 < pb1_end) {
  int32_t ia = a1_crd[pa1];
  int32_t ib = b1_crd[pb1];
  int32_t i = TACO_MIN(ia,ib);
  if (ia == i && ib == i) {
    c_vals[i] = a_vals[pa1] * b_vals[pb1];
  }
  pa1 += (int32_t)(ia == i);
  pb1 += (int32_t)(ib == i);
}
```

(a) Code for $c_i = a_i \times b_i$ where $a$ and $b$ are sparse over 0 and $c$ is dense

```c
int32_t pa1 = a1_pos[0];
int32_t pa1_end = a1_pos[1];
int32_t pb1 = b1_pos[0];
int32_t pb1_end = b1_pos[1];

while (pa1 < pa1_end && pb1 < pb1_end) {
  int32_t ia = a1_crd[pa1];
  int32_t ib = b1_crd[pb1];
  int32_t i = TACO_MIN(ia,ib);
  if (ia == i && ib == i) {
    c_vals[i] = a_vals[pa1] * b_vals[pb1];
  }
  else if (ia == i) {
    c_vals[i] = a_vals[pa1] * 1;
  }
  pa1 += (int32_t)(ia == i);
  pb1 += (int32_t)(ib == i);
}
while (pa1 < pa1_end) {
  int32_t i = a1_crd[pa1];
  c_vals[i] = a_vals[pa1] * 1;
  pa1++;
}
```

(b) Code for $c_i = a_i \times b_i$ where $a$ is sparse over 0 and $b$ is sparse over 1

Figure 3-2: Iteration patterns for the same scalar expressions $c_i = a_i \times b_i$ where the value compressed out is changed. Tensor Index Notation always generate the code in (a) whereas Array Index Notation can generate code in both (a) and (b).

of this is when performing a sparse vector-sparse vector multiply $c_i = a_i \times b_i$, the $\times$ operator always assumes that it only needs to iterate over the intersection of the two vectors. The code that would be emitted is shown in Figure 3-2a. This is because the result would be 0 when either of the input tensors is 0. However, tensors could be sparse over any value meaning that the assumptions made by the operators in Tensor Index Notation cannot generalize to cases where arbitrary values are missing. For example, if only $a$ was sparse over 0 but $b$ was sparse over 1, then the compiler would need to iterate over just $a$ instead of the intersection. The code for this case is shown in Figure 3-2b.

To solve this problem, Array Index Notation introduces a generic operator that decouples the computation from the iteration pattern. Thus, Array Index Notation is simply tensor index notation with at least one generic operator used in the expression. In its simplest form, an operator can be described by just a scalar computation to be done for each value of a sparse array including the value that has been compressed out. This computation is defined via a functor that is passed into the generic operator. For example, we can define and use a generic multiply operator as shown below. The last value in the tensor constructor is the value compressed out which defaults to zero.

```
struct MulImpl {
  ir::Expr operator()(const std::vector<ir::Expr> &v) {
    return ir::Mul::make(v[0], v[1]);
  }
};
IndexVar i;
Tensor<double> c("c", {10}, {Dense});
Tensor<double> a("a", {10}, {Compressed});
Tensor<double> b("b", {10}, {Compressed}, 1);
Op mulOp(MulImpl());
c(i) = mulOp(a(i), b(i));
```

Using this simple form causes the compiler to generate correct but inefficient code since it assumes it needs to perform computations at every coordinate including those where both $a$ and $b$ are missing values. In those cases, it generates the expression given but replaces $a$ and $b$ with their missing values. An example of the code that would be generated in this case is shown in Figure 3-3. In order for the operator to generate the code shown in Figure 3-2b one needs to add properties to the operator. These are explained in detail in Chapter 5 and Chapter 6 but I used them here for illustration. One would only need to change the instantiation of the `Op` to the following:

```
Op mulOp(MulImpl(), {Annihilator(0), Commutative});
```

From this, the compiler can infer that when any input expression is 0, the operator returns 0. It can use this knowledge to decide what iteration pattern is required. The compiler uses this information to enable it to generate the code shown in Figure 3-2b instead of the suboptimial code in Figure 3-3.

```
int32_t i = 0;
int32_t pa1 = a1_pos[0];
int32_t pa1_end = a1_pos[1];
int32_t pb1 = b1_pos[0];
int32_t pb1_end = b1_pos[1];

while (pa1 < pa1_end && pb1 < pb1_end) {
  int32_t ia = a1_crd[pa1];
  int32_t ib = b1_crd[pb1];
  if (ia == i && ib == i) {
    c_vals[i] = a_vals[pa1] * b_vals[pb1];
  }
  else if (ia == i) {
    c_vals[i] = a_vals[pa1] * 1;
  }
  else if (ib == i) {
    c_vals[i] = 0 * b_vals[pb1];
  } else {
    c_vals[i] = 0 * 1;
  }
  pa1 += (int32_t)(ia == i);
  pb1 += (int32_t)(ib == i);
  i++;
}
while (pa1 < pa1_end) {
  int32_t ia = a1_crd[pa1];
  if (ia == i) {
    c_vals[i] = a_vals[pa1] * 1;
  }
  else {
    c_vals[i] = 0 * 1;
  }
  pa1 += (int32_t)(ia == i);
  i++;
}
while (pb1 < pb1_end) {
  int32_t ib = b1_crd[pb1];
  if (ib == i) {
    c_vals[i] = 0 * b_vals[pb1];
  }
  else {
    c_vals[i] = 0 * 1;
  }
  pb1 += (int32_t)(ib == i);
  i++;
}
while (i < c1_dimension) {
  c_vals[i] = 0 * 1;
  i++;
}
```

Figure 3-3: Code showing unoptimized versions of sparse vector-sparse vector multiply. The compiler has applied no optimizations to the compute code in the sparse example above. Later chapters will discuss this snippet showing how the compiler can optimize this code given the properties and algebra defined in the operator.

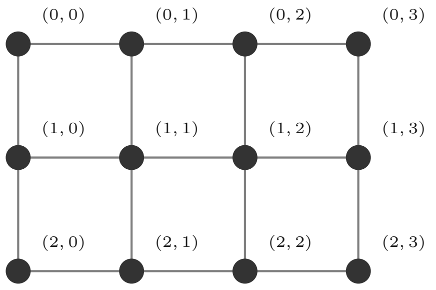
In addition to the properties, the generic operators introduced with array index notation also allows the user to specify an iteration algebra that describes the iteration pattern. This is described in Chapter 4 but I mention them here to give the reader a full overview of generic operators.

Array Index Notation extends Tensor Index Notation by introducing generic operators. These operators can be defined by just specifying a scalar operator. However, supplying properties and iteration algebra to the operator allows the compiler to emit efficient code over the scalar computation when computing on sparse tensors. In the rest of this thesis, I describe the iteration algebra and properties that are tagged with the generic operator's scalar definition and how it all comes together in order to generate code that computes over sparse arrays.

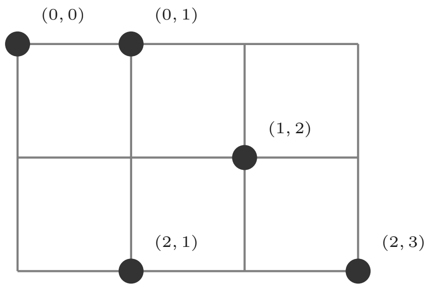
# Chapter 4

# Iteration Spaces

The iteration space of loops that iterate over dense tensors can be described as a hyper-rectangular grid of points by taking the Cartesian product of the iteration domain of each loop [22]. A sparse iteration space is a grid with missing points called holes [22]. In traditional sparse tensor algebra, the holes come from tensor components whose values are zero. Figure 4-1 illustrates this way of thinking about iteration spaces. In this work, the holes can be any value of the primitives types that the compiler supports which are integer, floating point and Boolean data types.



(a) Dense iteration space, with all points present.



(b) Sparse iteration space, with some points missing.

Figure 4-1: The Grid Representation of Iteration Spaces showing a dense and sparse iteration space for $4 \times 3$ matrix

In this chapter, we will extend the iteration space algebra from TACO [22] to include complements of iteration spaces instead of only unions and intersections. We will see a generalization of the iteration lattices from the tensor algebra compiler that allows the complements introduced in the iteration space algebra language to be expressed. This will allow any iteration pattern to be described to the compiler instead of only union and intersection patterns. The extension to the iteration lattice will include a new type of point called an **omitter point** which allows a user to skip computing when specific subsets of the input tensors have holes in their iteration spaces. The introduction of this new point requires a redefinition of the existing lattice points in the tensor algebra compiler. They will be redefined to be **producer points** meaning that they emit a compute statement whenever a specific subset of tensors all have a value defined at a given coordinate. Additionally, we will see a distinction between the new omitter points and **missing points** which are simply points that are not in the lattice. Lastly, I show how compute statements can be tied to lattice points to allow entirely different computations to be performed at each lattice point. The ideas in this chapter builds the representations needed to compile code to compute on sparse arrays and shows how to translate from an iteration algebra expression to an iteration lattice. Figure 4-2 gives a high level overview of this chapter. The properties, iteration algebra and region computations are bundled with the generic operator. The properties tell the compiler what mathematical properties the operator possesses such as its identity and whether it is commutative or associative. The properties are discussed in detail in Chapter 5. The iteration algebra is a set language for describing iteration spaces. It can be inferred from the properties and tensor fill values or it can be user specified. The region computations allow different computations to be done when a certain subset of the tensors that form the iteration space have a coordinate defined. For example, if the operator defines addition for $a, b$ and $c$, the region computation allows the user to specify that $a$ and $b$ should be multiplied if $c$ lacks a coordinate instead of being added. The iteration lattice is an IR used by TACO to reason about the loops that should be generated to compute on sparse iteration spaces. Finally, the imperative IR is another intermediate representation TACO uses before it generates code for its different backends. Chapter 6 demonstrates how these representations can be compiled into C code that iterates over and computes on any iteration space.

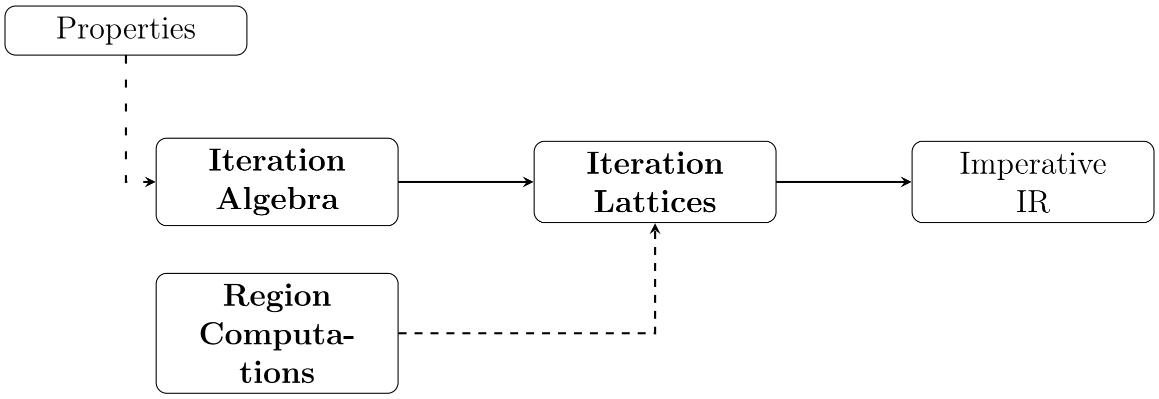


Figure 4-2: Iteration spaces overview that highlights the iteration algebra, iteration lattices and special region computations described in this chapter. Arrows in this diagram imply that one representation is used to build the other. Dashed lines imply that the prior representation is optional. For example, properties are optional when constructing iteration algebra since the algebra can be user specified. Likewise, region computations are not needed in order to construct an iteration lattice if the same expression is being performed in every lattice point.

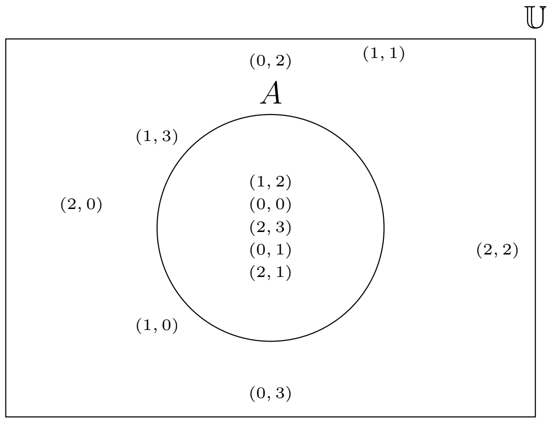
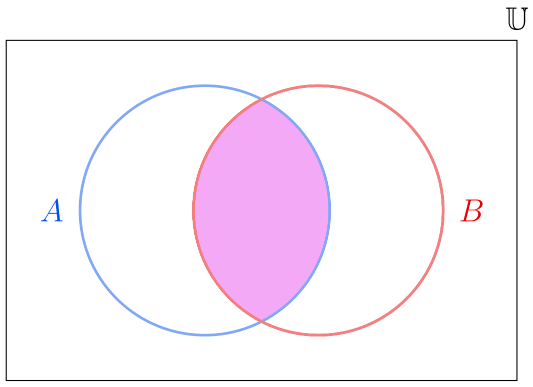


Figure 4-3: Venn Diagram showing which sets tensor coordinates belong to. All the coordinates belonging to A are in the subset A within the universe of coordinates. This is a Venn Diagram representation of Figure 4-1

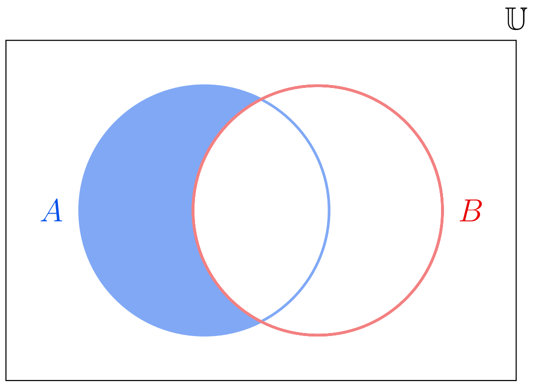
.

## 4.1 Iteration Space Algebra

One can also think about iteration spaces as a Venn Diagram of coordinates. The Universe of this Venn Diagram is equivalent to set of all points in a dense iteration space such as the one shown in Figure 4-1a. Since sparse tensors only define some of the possible coordinates in the dense space, they form subsets within the universe of points. This way of thinking naturally leads to a set expression language for describing



(a) Venn Diagram representation of $A_{ij} \cap B_{ij}$



(b) Venn Diagram representation of $A_{ij} \cap \overline{B_{ij}}$

Figure 4-4: The above shows the Venn Diagram regions of the iteration spaces two set expressions with tensors $A$ and $B$. The colours are mixed where the tensors overlap.

tensor iteration spaces called **iteration space algebra** [22]. Figure 4-3 illustrates this way of thinking about iteration spaces. Prior to this thesis, it was only possible to combine iteration spaces by taking the intersections and unions of their points. However, this work adds the capability to take the complement of point sets which opens up a much wider range of iteration spaces that can be considered.

The representation of the algebra remains the same as the one described in section 3.1 of Sparse Tensor Algebra Compilation [22] with the addition of set complement, which opens up all set combinations beyond intersections and unions. Briefly, the notation uses index variables to index tensor expressions in order to control what coordinates should be compared in the set operation. For example, one could write $A_{ij} \cap B_{ij}$ to indicate the intersection of two sparse matrices $A$ and $B$ as required for element-wise multiplication (assuming fill values of 0). However, with the introduction of complements, one can use expressions such as $A_{ij} \cap \overline{B_{ij}}$ to denote an operation that only uses results from a matrix $A$ whenever $B$ has no value defined at that coordinate. An algebra such as this is useful for expressing a filtering operation between two matrices. For example, if $A$ represents the set of MIT Computer Science students and $B$ represents the set of MIT students with glasses then the result of the set operation would return the set of MIT Computer Science students without glasses. These set expressions are shown in Figure 4-4.

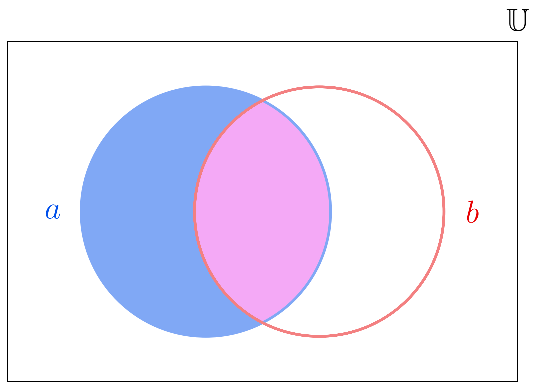


Figure 4-5: Venn diagram for $a$. The colors are mixed where both tensors need to be coiterated. Shows that outside of $a$ multiplication does not been to be performed since the result of the operation is known to be zero.

The Iteration Algebra is one of the extra pieces of information that can be tagged to the operators described in Chapter 3. This tells the compiler the exact set of values that needs to be iterated over to compute the final result. The compiler can then reason about this set to reduce the number of while loops and conditionals needed to iterate over sparse iteration spaces that arise from sparse tensor algebra.

Now, we go back to the sparse-sparse vector multiplication discussed in Chapter 3. Suppose our two input vectors were $a$ with a fill value of 0 and $b$ with a fill value of 1. Since the fill value of $a$ is 0, any time $a$ is missing a coordinate, the value of the final result is known to be 0 since $0 \cdot x = 0$ for all $x$. Therefore, this operator can be tagged with the algebra $a_i$ meaning results are only generated when $a$ has a defined coordinate. A graphical depiction of this algebra is shown in Figure 4-5. This algebra allows the generated code to be simplified from the original code shown in Figure 4-6a. For the simplified code shown in Figure 4-6b, assume the compiler has initialized the result vector $c$ to the correct value of 0. Figuring out the initialization value can be done by compile time constant propagation where tensor operands are replaced by their fill values.

```
int32_t i = 0;
int32_t pa1 = a1_pos[0];
int32_t pa1_end = a1_pos[1];
int32_t pb1 = b1_pos[0];
int32_t pb1_end = b1_pos[1];

while (pa1 < pa1_end && pb1 < pb1_end) {
  int32_t ia = a1_crd[pa1];
  int32_t ib = b1_crd[pb1];
  if (ia == i && ib == i) {
    c_vals[i] = a_vals[pa1] * b_vals[pb1];
  }
  else if (ia == i) {
    c_vals[i] = a_vals[pa1] * 1;
  }
  else if (ib == i) {
    c_vals[i] = 0 * b_vals[pb1];
  } else {
    c_vals[i] = 0 * 1;
  }
  pa1 += (int32_t)(ia == i);
  pb1 += (int32_t)(ib == i);
  i++;
}
while (pa1 < pa1_end) {
  int32_t ia = a1_crd[pa1];
  if (ia == i) {
    c_vals[i] = a_vals[pa1] * 1;
  }
  else {
    c_vals[i] = 0 * 1;
  }
  pa1 += (int32_t)(ia == i);
  i++;
}
while (pb1 < pb1_end) {
  int32_t ib = b1_crd[pb1];
  if (ib == i) {
    c_vals[i] = 0 * b_vals[pb1];
  }
  else {
    c_vals[i] = 0 * 1;
  }
  pb1 += (int32_t)(ib == i);
  i++;
}
while (i < c1_dimension) {
  c_vals[i] = 0 * 1;
  i++;
}
```

(a) Code for computing $c_i = a_i \times b_i$ where $a$ has a fill of 0 and $b$ has a fill of 1. In this example, the algebra is not used so the compiler unnecessarily iterates over the entire space.

```
int32_t pa1 = a1_pos[0];
int32_t pa1_end = a1_pos[1];
int32_t pb1 = b1_pos[0];
int32_t pb1_end = b1_pos[1];

while (pa1 < pa1_end && pb1 < pb1_end) {
  int32_t ia = a1_crd[pa1];
  int32_t ib = b1_crd[pb1];
  int32_t i = TACO_MIN(ia,ib);
  if (ia == i && ib == i) {
    c_vals[i] = a_vals[pa1] * b_vals[pb1];
  }
  else if (ia == i) {
    c_vals[i] = a_vals[pa1] * 1;
  }
  pa1 += (int32_t)(ia == i);
  pb1 += (int32_t)(ib == i);
}
while (pa1 < pa1_end) {
  int32_t i = a1_crd[pa1];
  c_vals[i] = a_vals[pa1] * 1;
  pa1++;
}
```

(b) Code after using algebra to iterate over the minimal part of the iteration space.

Figure 4-6: The above is shown to illustrate the difference in code size and complexity when the iteration algebra is used to reduce the iteration space. It shows an example of a code snippet the compiler generates when no iteration algebra is specified with the operator compared to the code generated when an iteration algebra is passed in. The code with the algebra simplifications is much shorter since there are less regions the compiler needs to iterate over.

## 4.2 Iteration Lattices

In this section, I discuss the iteration lattice representation while highlighting the contributions of this thesis. TACO uses iteration lattices (formerly merge lattices [21]) to generate code for sparse iteration spaces. I extend the iteration lattice representation by introducing a new point type called an omitter point. I also modified the existing lattice construction rules to account for the new representation and introduced a new rule when constructing lattices to allow for complements to be performed. Finally, I explain the difference between loop lattices and case lattices. In prior works, the iteration lattices are used both to generate loops as well as cases within those loops. However, with the extensions in this work, the two use cases of the iteration lattices need to be distinguished. This section discusses these modifications.

### 4.2.1 Iteration Lattice Background

This section is an adaptation of the description of iteration lattices provided in section 3.3 of Sparse Tensor Algebra Compilation [22].

The iteration space is divided into regions described by the tensors that intersect there. These regions are the powerset of the tensors that form the iteration space; that is, the set of all subsets. Thus, an iteration space with $k$ tensors divides into $2^k$ **iteration region**s (the last region is the empty set $\emptyset$ where no sets intersect). Figure 4-7 shows the Venn diagram of the union of three tensors with labels marking the seven non-empty regions where subsets of tensors 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 spaces, we must iterate through each of these regions. But if the iteration space contains intersections, we only need to iterate through some of the regions (a subset of the powerset). For example, in the intersection in Figure 4-8, we only need to iterate through the single region described by the intersection of all three tensors. Figure 4-9 shows that when the iteration space is a combination of union and intersection operations, we need to iterate over more than one but not every region.

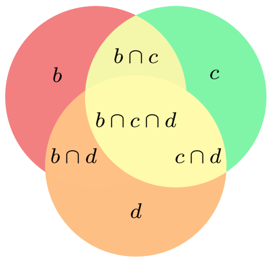


Figure 4-7: Venn diagram of iteration regions of $b \cup c \cup d$.

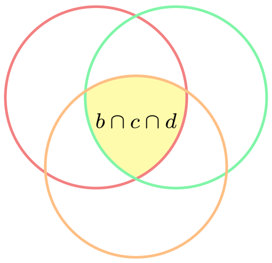


Figure 4-8: Venn diagram of the iteration regions of $b \cap c \cap d$.

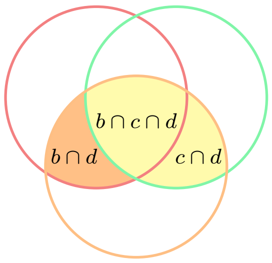


Figure 4-9: Venn diagram of the iteration regions of $(b \cup c) \cap d$.

The coiteration over an iteration space is organized so that when a region runs out of coordinates, the coiteration algorithm jumps to a simpler algorithm that excludes the tensors that ran out of coordinates (Figure 4-10). How, and whether, the algorithm can detect when segments run out of coordinates depends on the coiteration strategy. Two coiteration strategies are discussed over two segments and then generalized to any number of segments using a lattice formulation. The two-way coiteration strategies are called *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 tensors 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. 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 any region, but it requires that the sparse tensors defining that region have coordinates that are ordered. It coiterates through the tensors 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

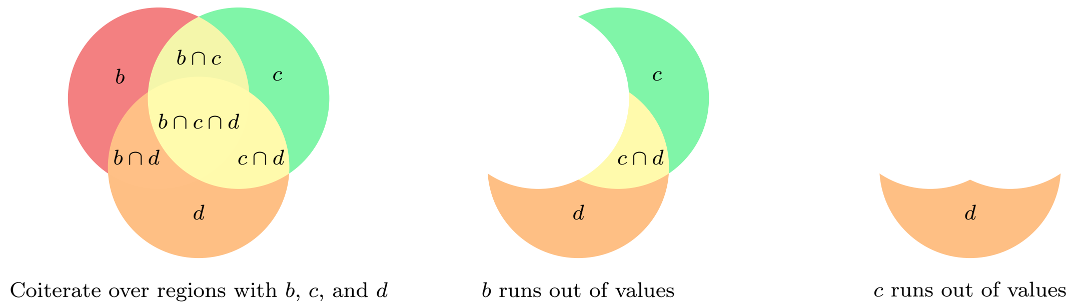


Figure 4-10: Coiterate over the coordinates of $b$, $c$, and $d$, considering all regions. When $b$ runs out of values the coiteration proceeds to coiterate over only $c$ and $d$, ignoring regions that contain $b$. And when $c$ also runs out of values the coiteration proceeds to only iterate over $d$, ignoring regions that contain $b$ or $c$.

considered next. It lies in the iteration region that contains only one tensor, and only that tensor's coordinate is advanced. The iteration terminates when either tensor 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 4-11, or a union where one of the regions is a superset of the other region. The two-way iterate-and-locate algorithm iterates over one of the region and locates coordinates from the other region if the tensors in that region support random access. Examples of tensors with this feature are dense arrays and hash maps.

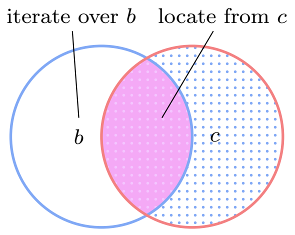


Figure 4-11: The iterate-and-locate strategy iterates over one operand and locates the coordinates in another.

An iteration lattice is a partial ordering of the powerset of some set of tensors $\mathbb{S}_t$ by the number of tensors in each subset. It is, in other words, a lattice of increasingly fewer tensors to coiterate over until all tensors run out of coordinates (Figure 4-12).

Each one of the sets in the $\mathbb{P}(\mathbb{S}_t)$ corresponds to one of the iteration regions in the Venn Diagram corresponding to the iteration lattice. The sets within the powerset are called iteration lattice points. The tensors of a point are divided into two sets called its *iterators* and *locators* (Figure 4-13). The iterators are the tensors 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.

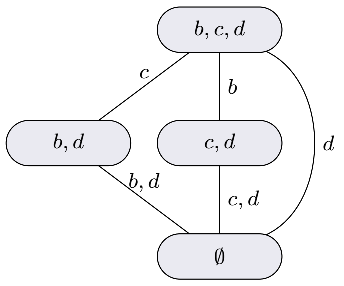


Figure 4-12: The iteration lattice of the iteration domain $(b \cup c) \cap d$ shown in Figure 4-9. Nodes are lattice points with tensors to coiterate over (the top coiterates over all tensors) and edges move to another point when a tensor runs out of coordinates.

A lattice can be viewed as a state machine that coiterates through subsets of regions as in Figure 4-10. Figure 4-15 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.

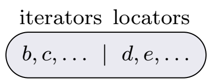


Figure 4-13: The tensors in a lattice point are divided into iterators to coiterate over coordinates, and locators to locate the coordinate in.

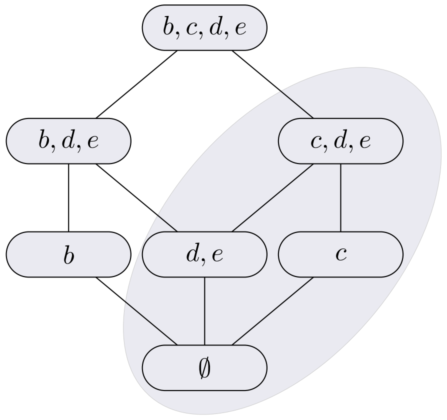


Figure 4-14: The sublattice of the lattice points below a lattice point $\mathcal{L}_p$. These are the lattice points that exclude segments that have run out of values.

2. Compute the candidate coordinate, which at each step is the smallest of the current coordinates of the tensors.

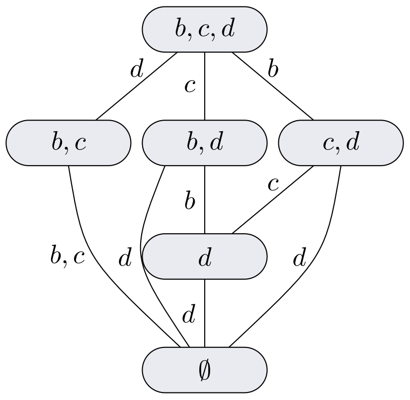
3. Determine which region the candidate coordinate is in by checking what tensors are currently at that coordinate. The only regions we need to consider are those of the lattice points underneath the current lattice point (Figure 4-14).

4. When any tensor 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 space made of unions and intersections. This code consists of a sequence of coiterating `while` loops that become simpler as we move down the lattice.

### 4.2.2 New Representation

I introduce the concept of an omitter point to the original iteration lattices described previously. An **omitter point** is simply a point where computation must be skipped inside of the if statement for that lattice point. We still need these points in the lattice because the code generator must still emit while loops that coiterate over the tensors defining that point. A pseudocode example of this is shown for performing the symmetric difference of two sparse tensors in Figure 4-18. To distinguish omitter



```
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 ...
```

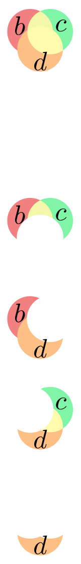
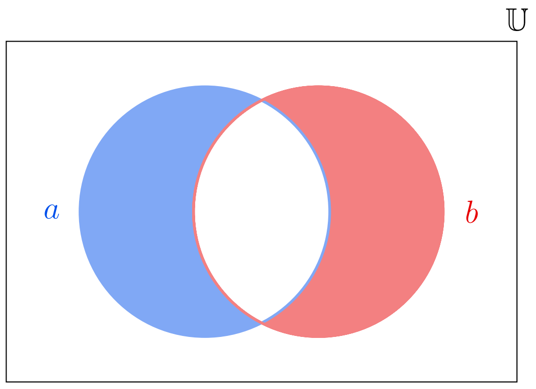


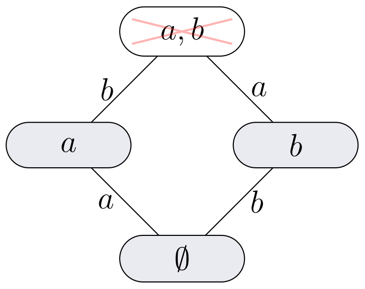
Figure 4-15: Iteration lattice and corresponding coiteration pseudocode for the iteration domain $(b \cap c) \cup d$. There is one `while` loop per lattice point and each `while` loop contains one `if` statement per sub-lattice point. Next to each `while` loop is a Venn diagrams showing the regions it coiterates over. The `while` loops iterate until a tensor 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.

points from the original lattice points, I rename the original lattice points to **producer points** since they always produce a computation inside of their if statements. Omitter points are semantically different from missing points since we do not iterate over the missing regions. However, omitter points are equivalent to missing points when the omitter has no non-empty set children points that are producers, since no loops or conditionals need to be emitted for the entire lattice sub-tree. The beauty of this extension is that all of the other concepts in the original iteration lattice remains unchanged while giving us the ability to describe any iteration space instead of just intersections and unions.

Figure 4-17b shows the lattice representation for the Venn Diagram shown in Figure 4-5. A lattice point is not explicitly needed to omit the region $b$ since it would have no children that are producers. However, Figure 4-16 illustrates an iteration space for the xor operator along with the corresponding iteration lattice for that space. The xor example is shown since the old lattice representation does not support this space and it illustrates the expressive power of omitter points in the new representation. The

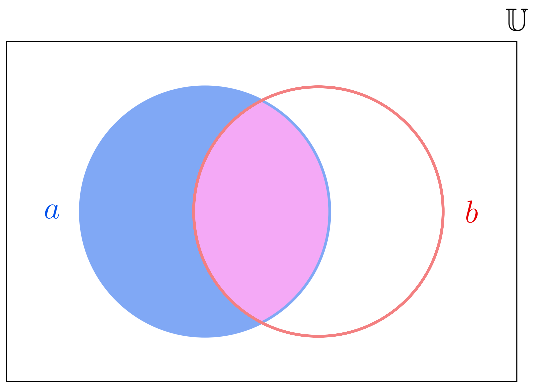


(a) Venn Diagram for computing the symmetric difference of $a$ and $b$.

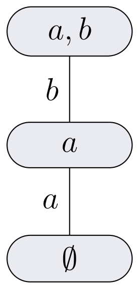


(b) Lattice for computing the symmetric difference of $a$ and $b$.

Figure 4-16: This shows the Venn Diagram for computing the XOR between two tensors $a$ and $b$ and the lattice corresponding to that diagram. This iteration space cannot be represented in the old representation since there is no way to instruct the compiler that it does not need to produce a value when both $a$ and $b$ have values defined at a particular coordinate.



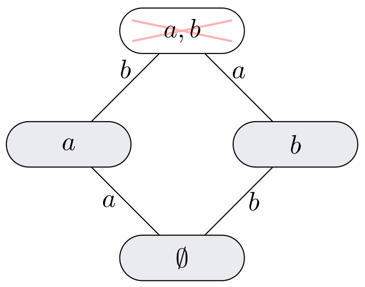
(a)



(b)

Figure 4-17: Lattice representation and Venn Diagram for the sparse vector - sparse vector multiplication where we only need to iterate over $a$. The lattice point b is absent because when $a$ is empty, we are done iterating.

omitter point is needed so TACO knows that it needs to co-iterate over the tensors $a$ and $b$ but it can avoid computing and storing values when both tensors are defined during the co-iteration.



```
while $a$ and $b$ have coordinates left do
    if in region $[a, b]$ then do nothing
    else if in region $[a]$ then ...
    else if in region $[b]$ then ...
while $a$ has coordinates left do
    if in region $[a]$ then ...
while $b$ has coordinates left do
    if in region $[b]$ then ...
```

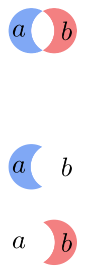


Figure 4-18: Iteration lattice and corresponding coiteration pseudocode for the iteration space describing a symmetric difference. The treatment of the omitter point is the same when emitting `while` loops. However, when emitting `if` statements, we do nothing in the intersection of $a$ and $b$ to ensure that it is skipped. Without this explicit skip, we may accidentally end up performing computations inside the region $a \cap b$.

### 4.2.3 Construction

Iteration Lattices are constructed directly from the iteration algebra discussed previously. Before a lattice is constructed, the iteration algebra for the entire expression is constructed then the lattice construction rules are applied to that algebra. Before discussing these rules formally, I will describe an intuitive way of thinking about constructing merge lattice. This will always provide the correct lattice but not necessarily the most optimized lattice.

The input to the lattice construction algorithm is an iteration algebra expression. An intuitive way of thinking about the result lattice is to visualize the space using a Venn Diagram. First, one can draw the Venn Diagram for the iteration algebra set expression. We can then list the powerset of the set of input tensors in an ordered lattice and include an extra point in that lattice for the universe. I refer to this lattice as a **template lattice** from here onward. An example of this structure is shown in Figure 4-19 for an arbitrary binary operation. After writing out this structure, we mark each point corresponding to an unshaded region in the Venn Diagram as an omitter point. Finally, to get the result lattice, we insert the universe into every point (except itself) if the universe was not marked as omitted.

I will use the expression $a \oplus b$, where $\oplus$ is the XOR operator that iterates over the symmetric difference of $a$ and $b$. The Venn Diagram for this operator is shown in Figure 4-16a as an example to demonstrate this intuitive framework. The actual iteration algebra would be $\overline{(a \cap b)} \cap (a \cup b)$. Since $a$ and $b$ are the inputs to the xor operation, we observe that $\mathbb{P}(\{a, b\}) = \{\{a, b\}, \{a\}, \{b\}\, \{\}\}$. The result lattice after including the universe and marking the necessary points as omitter points is shown in Figure 4-20. This lattice is correct but contains a point for every region in the Venn Diagram, which is unnecessary. It is also based on having the venn-diagram, which the machine would not have when constructing the lattice.

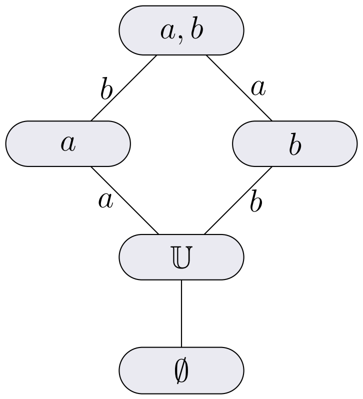


Figure 4-19: Lattice showing the structure a template lattice takes for an arbitrary binary operation before marking points as omitters. The diagram assumes the input arguments are regions $a$ and $b$.

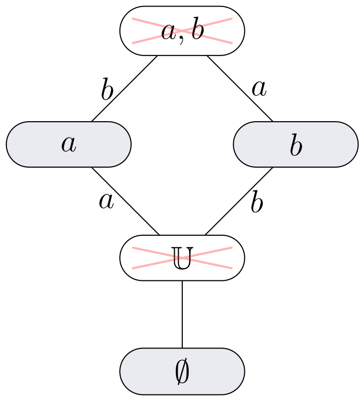
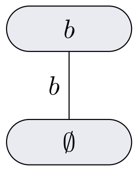


Figure 4-20: The lattice corresponding to the XOR Venn Diagram in Figure 4-16a with the unshaded regions marked as omitters. The universe is not included with the other points since it was marked as an omitter point in the template. Note that this lattice is different from the one in Figure 4-16b since it explicitly includes the universe as an omitter point.

Now that an intuitive framework has been established, I describe a revised bottom-up algorithm from the one detailed in sparse tensor algebra compilation [22]. I will introduce the new complement rule and then describe the revised union and intersection rules. I will also introduce a rule called the augmentation rule that is used to attach extra information to the iteration algebra in special cases to simplify the construction algorithm. The segment rule remains unchanged from the original work of TACO.

Before applying any of the rules to the input iteration algebra, we first apply

De Morgan's laws to simplify the algebra expressions. The primary reason for this is to push complements down the algebra expression so that all complements are applied to tensors and not complex algebra expressions. This preserves the meaning of the algebra while making the complement rule much simpler than it would have been otherwise. This simplicity is due to the fact that the algorithm would need to reason about intersections, unions and dimensions separately in the complement rule whereas now, the complement rule only needs to reason about segments containing single tensors. With the algebra simplified, we can apply the lattice construction rules below.



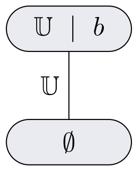


Figure 4-21: The iteration lattices of a segment expression, that supports an iterator capability (left) and that instead only supports locate (right).

**Segment Rule**: When we get to a tensor in the iteration algebra expression, we construct a segment for that tensor. This is the base case of the lattice construction algorithm. The segment rule has two cases:

**Case 1:** The tensor can be iterated over.

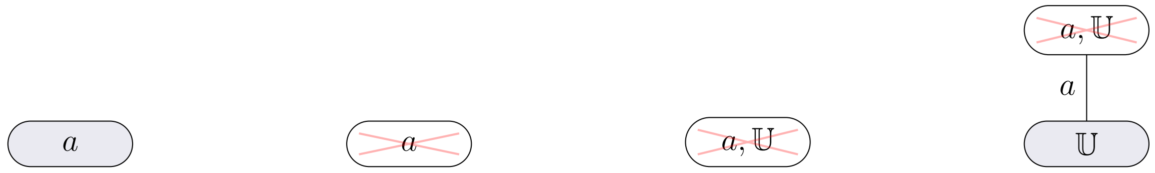
*Rule:* Return a lattice with a single non-bottom lattice point whose set of iterators contains the iterator for the tensor and whose set of locations is empty.

**Case 2:** The tensor cannot be iterated but can be located into.

*Rule:* Returns a lattice with a single non-bottom lattice points, the iterators set of which contains the dimension of the tensor and the locators set contains the tensor.

Figure 4-21 contains an example of the application of both rules on a tensor $b$.

**Complement Rule**: The complement rule only has one case. It can only complement a single lattice point representing an input tensor. When given a single lattice



(a) Lattice point for $a$.
(b) Mark point as omitter
(c) Insert dimension
(d) Add producer dimension

Figure 4-22: The stages of the complement rule. The dimension is needed so that TACO knows it needs to iterate over the entire space and must coiterate with $a$ and the dimension until $a$ runs out of values.

point, the complement rule marks that lattice point as an omitter, adds a dimension iterator to the new omitter point if one is not already present and inserts a lattice point with just the dimension iterator at the bottom. A flow of this is shown in Figure 4-22. The simplicity of this rule is due to the fact that De Morgan's laws are applied to the algebra before applying the lattice construction rules. Therefore, complements always appear around a single tensor expressions instead of arbitrary set expressions.

**Intersection Rule**: The intersection rule takes two lattices as its operands and combines the lattices to produce a new lattice that describes iteration over the intersection. 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. Finally, we need to figure out the type of the result lattice point being intersected. There are two cases when figuring out the type of the result point formed by intersecting two lattice points. They are:

**Case 1:** The region *described by the result* is a producer in both input lattices.

*Rule:* The result should always be a producer.

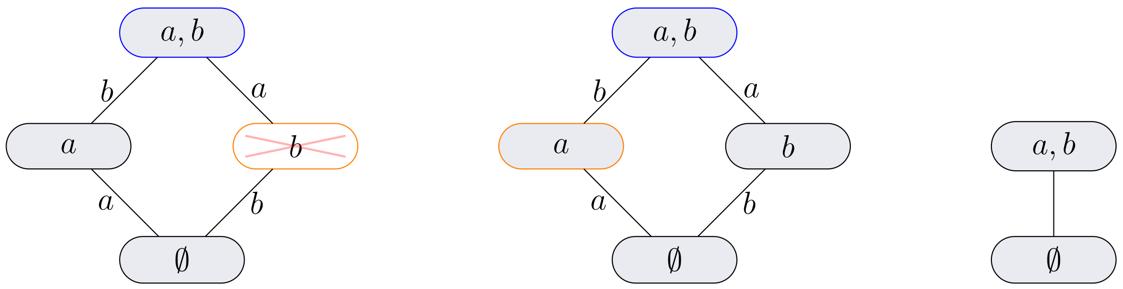
For example if intersecting lattice point $a$ and lattice point $b$ from two lattices, we must check if the region $a \cap b$ is a producer in both lattices. If so, the result point type must be a producer point.

Intuitively, this is because the intersection takes elements both in set $a$ and set

$b$. Therefore, if a subset exists on both sets (indicated by the fact that both sets have a producer) it must also exist in the result.

Note that in this case, the types of the lattice points being intersected do not matter.

An example showing the necessity of this rule is shown in Figure 4-23.



(a) Operand 1. Set: $a$ (b) Operand 2. Set: $a \cup b$ (c) Result lattice. Set: $a \cap b$

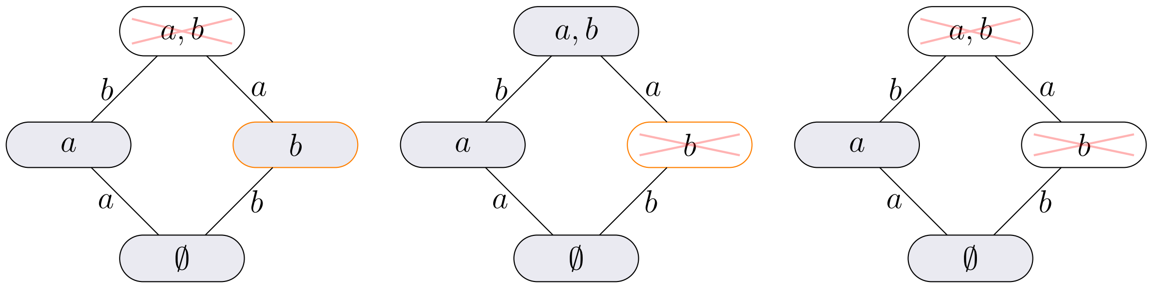
Figure 4-23: Here, we are computing $(a) \cap (a \cup b)$ and the inputs to the intersection rule are shown in (a) and (b). At this point, we do not have the rules required to construct the operand lattices. However, they can be derived by using the intuitive framework described previously and removing the universe point. I use these lattices since they allow a non-trivial example of the rules to be shown. This example shows case 1 for the intersection rule using the lattice points highlighted in orange. When intersecting the lattice points highlighted in orange, we get the region $a, b$. This region is marked as a producer in both of the input lattices as seen by the blue highlighted lattice points. Therefore, it must be a producer in the result lattice shown in (c).

**Case 2:** The *region described by the result* is not a producer in both input lattices. This means that the region is either absent or an omitter.

Similar to case 1, this means that if intersecting the lattice points $a$ and $b$ from two lattices, and we find that $a \cap b$ is not a producer in both lattices (it is either missing or an omitter) then the result type depends on the types of $a$ and $b$ and is given by the rule below.

*Rule:* If both of the lattice points being intersected are producers then the result is a producer. Otherwise, the result should be an omitter.

An example showing this rule is shown in Figure 4-24.



(a) Operand 1. Set: $a \oplus b$ (b) Operand 2. Set: $a$ (c) Result lattice. Set: $a \cap \bar{b}$

Figure 4-24: Example showing case 2 of the intersection rule using the highlighted lattice points. As before, we do not have the rules required to construct the operand lattices. They can be derived by using the intuitive framework described previously and removing the universe point. I use these lattices since they allow a non-trivial example of the rules to be shown. When intersecting the two highlighted points, we get the region defined by $b$. This region is not marked as a producer in both lattices so we apply the rule in case 2 of the intersection rule. This results in an omitter point for $b$ as seen in the result lattice.

**Union Rule**: The union rule combines lattice of its operands to produce a new lattice that describes iteration over the union of the iteration spaces described by the operands. To union two lattices, we first apply the intersection rule to get an intersection lattice then take the union of the lattice points of the intersection lattice and the two input lattices. As with the intersection rule, there are two cases when figuring out the type of the result point formed when intersecting two lattice points under the union rule. They are:

**Case 1:** The *region described by the result* is an omitter in both input lattices.

*Rule:* The result must be an omitter point.

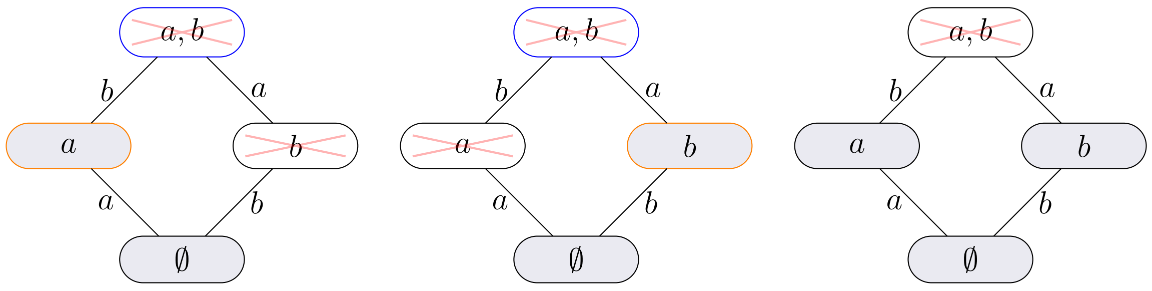
Note that, in this case, the types of the lattice points being unioned do not matter.

For example, if intersecting two lattice points $a$ and $b$, under the union rule we check to see if their intersection $a \cap b$ is present as an omitter in both of the operands to the union rule. If this is the case then we ignore the types of $a$ and $b$ and mark the result as an omitter.

Intuitively, this is because the union takes elements in either set $a$ or set $b$.

However, if an element is omitted from both sets then it cannot appear in the result.

An example showing the necessity of this rule is shown in Figure 4-25.



(a) Operand 1: set $a \cap \bar{b}$ (b) Operand 2: set $\bar{a} \cap b$ (c) Result lattice $a \oplus b$

Figure 4-25: Example showing case 1 of the union rule using the points highlighted in orange. The lattices in the operands are derived using the intuitive framework and removing the universe point as in the prior examples. When intersecting the highlighted points $a$ and $b$, we get the region defined by $a, b$ which is highlighted in blue in the operands. Since this region is an omitter in both of the operands it must be an omitter in the result as seen in the result lattice in (c).

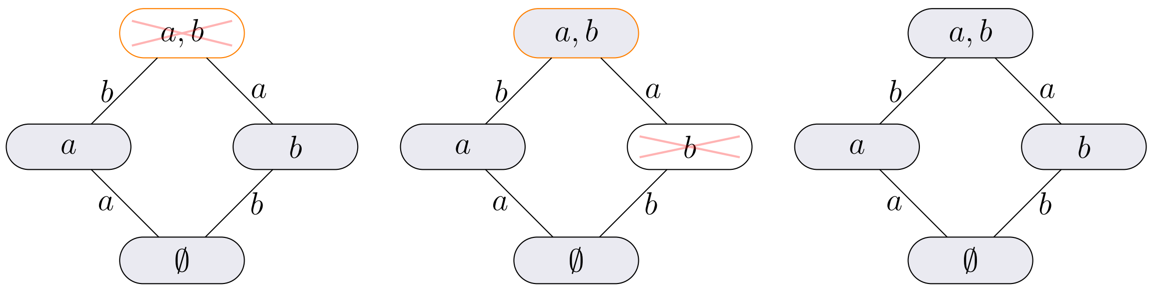
**Case 2:** The region described by the result is not an omitter in both input lattices.

For example, if intersecting two lattice points $a$ and $b$, under the union rule we check to see if their intersection $a \cap b$ is present as an omitter in both of the operands to the union rule. If the lattice point describing $a, b$ is missing or is a producer in either operand then the point type returned when intersecting $a$ and $b$ is given by the rule below:

*Rule:* If both of the input lattice points being intersected under the union rule are omitters, then the result must be an omitter. Otherwise, the result is a producer.

An example showing the necessity of this rule is shown in Figure 4-26.

These construction rules are sufficient when all operator operands are present in the iteration algebra. However, in the sparse vector-sparse vector element-wise multiplication discussed in Section 4.1 and shown in Figure 4-5, the entire iteration algebra can be accurately specified with just $a$. Although the iteration space and



(a) Operand 1: symmetric difference set (b) Operand 2: set: $a$ (c) Result lattice $a \cup b$

Figure 4-26: Example showing case 2 of the union rule using the highlighted lattice points. The lattices in the operands are derived using the intuitive framework and removing the universe point as in the prior examples. When unioning the highlighted points $(a, b)$ and $(a, b)$, we get the region defined by $(a, b)$. This region is not marked as an omitter in both operands so we apply the rule in case 2 of the union rule. This results in a producer point in the result lattice for the region $(a, b)$

is fully specified by the algebra $a$, the computation partially happens over $a$ and $b$ and so they must be coiterated. Since the construction algorithm only recurses on the iteration algebra, there is no way for it to get information about $b$. This means that TACO will only make a lattice point for $a$ during construction since there is no notion of the iteration space containing the tensor $b$ in the algebra. Intuitively this can be described as coiterating over a set of operands, and if one of the operands is a set complement then you want to iterate over everything else instead. But you don't have information about everything else in that iteration algebra node, so you need an environment that contains every tensor so that you know about them. As a result, before running the construction algorithm the augmentation rule is applied to add in elements that are in the operator's operands but not present in the iteration algebra.

**Augmentation Rule**: The augmentation rule handles case where operands appear in an operator's scalar definition but not in the iteration algebra. These set expressions are valid but cause issues for the lattice construction algorithm since it only recurses on iteration algebra for simplicity. To augment an algebra with information about missing operands, I note two set relations. Given a set $\mathbb{S}$, a set $\mathbb{T}$ and the empty set $\emptyset$ the following relations hold:

$$\mathbb{S} \cup \emptyset = \mathbb{S}$$

$$\mathbb{T} \cap \overline{\mathbb{T}} = \emptyset$$

Therefore, we can combine these relations to see that

$$\mathbb{S} \cup (\mathbb{T} \cap \overline{\mathbb{T}}) = \mathbb{S} \tag{4.1}$$

The above gives us a way to augment an arbitrary set expression $\mathbb{S}$ with information about another set $\mathbb{T}$ without changing the semantics of $\mathbb{S}$. This relation forms the foundation for the augmentation rule and it can now be stated.

*Rule*: If operands exist in the operator's scalar definition but not in the iteration algebra, the iteration algebra is augmented using Equation (4.1) where $\mathbb{S}$ is iteration algebra tagged with the operator and $\mathbb{T}$ is an operand to the operator that is not in the algebra. The result is updated to be the new tagged algebra of the operator. This rule is then repeated once for each operand in the operator's definition that is missing from the iteration algebra. If $\mathbb{T}$ is a set expression instead of a single tensor, De Morgan's laws are applied to $\mathbb{T}$ and $\overline{\mathbb{T}}$ to simplify the expression.

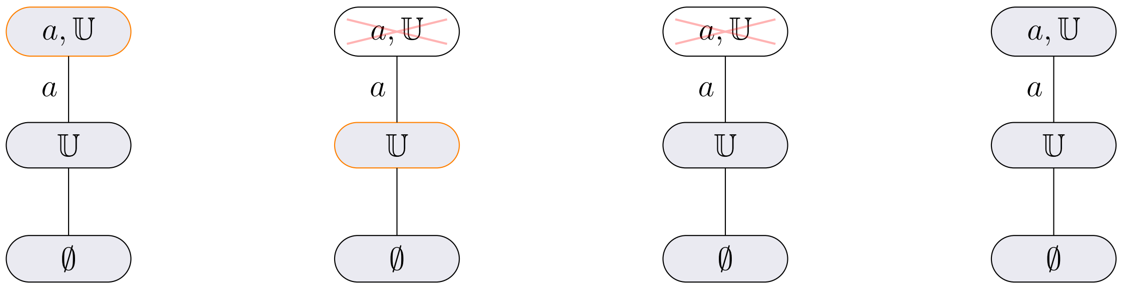
I note that the trick above only works due to the fact that simplifications to the lattice are done after the lattice has been fully constructed. This would not work if simplifying was done during construction since the augmented expression $\mathbb{T} \cap \overline{\mathbb{T}}$ would return a lattice with no lattice points.

So for the algebra $a$ with operands $a$ and $b$, the augmentation rule will update the tagged operator algebra to include information about $b$ by applying Equation (4.1) as shown below:

$$\mathbb{S}_{new} = a \cup (b \cap \overline{b})$$

It will then set $\mathbb{S}_{new}$ as the new tagged algebra. If there were more operands in the operator, the augmentation rule would be applied once for each of operands missing from the algebra.

As a final note about lattice construction, I add that there are cases where duplicate lattice points defining the same region appear with conflicting point types. These are resolved each time the intersection and union rules are applied as a final step. When applying the intersection rule, omitters always win the conflict and when applying the union rule, producers always win the conflict. Example input lattices that conflict for both intersections and unions are shown in Figure 4-27.



(a) Input lattice. Set: $a \cup \overline{a}$

(b) Lattice for the Set $\overline{a}$

(c) Intersection of lattice (a) with lattice (b)

(d) Union of lattice (a) with lattice (b)

Figure 4-27: An example showing lattices that cause duplicate points for intersections and unions along with the result for both operations. When combining the two input lattices (a) and (b) above, the highlighted points always combine to result in a producer $(a, \mathbb{U})$ while the points that are not highlighted always combine to give an omitter $(a, \mathbb{U})$. Intersections always resolve duplicates by preferring omitters so the region $(a, \mathbb{U})$ is an omitter in the result in (c) while the opposite is true for unions as seen in (d).

### 4.2.4 Special Region Definitions

The introduction of generic operators also allows for special computations to be performed at each lattice point. This is another piece of information that can be tagged to the operator. This is done by specifying the input tensors that should have a defined non-fill value and the computation to perform when all of those input tensors have a non-fill value defined at a given coordinate. This can be conceptualized as attaching a computation to each lattice point. By default, TACO will replace exhausted

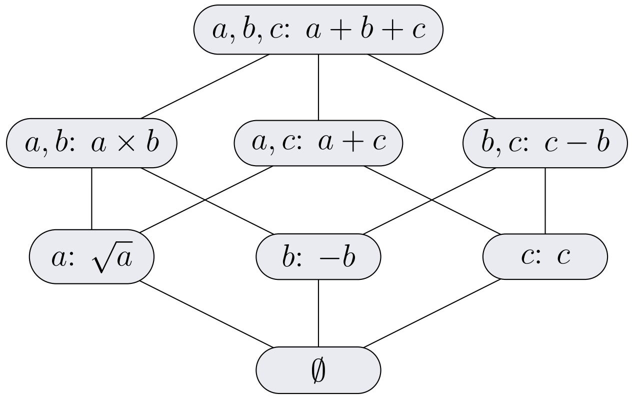


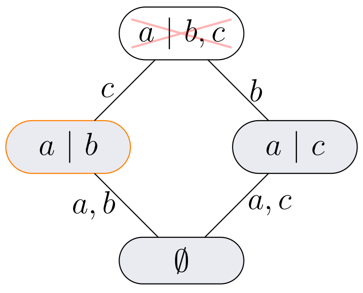
Figure 4-28: The above shows how TACO handles computations in special regions. Assume the fill values of all of the tensors are 0. Regions $(a, b, c)$, $(a, c)$ and $c$ highlight the default behaviour of TACO as the exhausted tensors are replaced with 0 and the expression is simplified. All the other points in the lattice perform a different operation specified in the operator interface.

tensors in the default operator with their fill values unless a special computation has been defined for a given lattice point. In that case, TACO will perform the special computation at that lattice point instead. Figure 4-28 shows an example of special operator where the default operator is a ternary addition $a + b + c$.

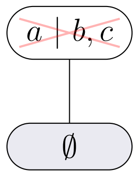
### 4.2.5 Case Lattices and Loop Lattices

Optimizations can be applied to the lattice after it is constructed in order to prune points. However, we can no longer unconditionally apply optimizations in lattice construction since we have to account for omitter points and points where some special computation must be performed. If one of these points is removed during construction, the code generator downstream will lose the information telling it to skip computing in some regions or to compute a special function in some region. As a result, these special points must be kept when applying optimizations during construction and leads to the distinction between loop and case lattices.

A case lattice is simply an iteration lattice after construction with no points removed. It is called a case lattice since it will have all the lattice points where a computation has to be performed under the emitted if cases along with all the lattice points where computation should be skipped in the if cases. A loop lattice is a case



(a) Case lattice



(b) Loop lattice

Figure 4-29: This example shows a case and a loop lattice for $(a \cap b) \cup (a \cap c) - (a \cap b \cap c)$. The highlighted lattice point denotes a special region. The code generator only needs to generate a loop over $a$, so when generating loops it is fine to use the lattice in (b). However, if attempting to generate cases within the loops using the lattice (b), the code generator lacks information saying it needs to perform a different computation in the highlight lattice point. It also has no way of knowing that it should not compute when $a$ and $c$ have defined values but not when all of $a$, $b$ and $c$ are defined. The case lattice in (a) contains all of this information and therefore this is the lattice that will be given to the code generator as input.

lattice formed after pruning unneeded lattice points. It dictates what loops should be emitted but it lacks all special points where different computations should be done or computations should be omitted.

The lattice construction algorithm always returns a case lattice. The code generator simplifies the case lattice to a loop lattice later when generating loops and uses the case lattice to generate the conditionals inside of those loops. These two uses of lattice already existed in TACO, but the addition of new point types required an explicit distinction between the two uses of the iteration lattices.

Figure 4-29 illustrates a case lattice for the expression $(a \cap b) \cup (a \cap c) - (a \cap b \cap c)$ along with the loop lattice. This example assumes $b$ and $c$ support locate while $a$ only supports iterate.

This concludes the discussion about iteration spaces and iteration lattices. In Chapter 5, I describe how one can derive iteration algebras from reasoning about the scalar properties of operators along with their input values to allow the compiler to automatically generate the correct lattices. Additionally, Chapter 6 describes how omitter points and special regions are handled when lowering the iteration lattices.

# Chapter 5

# Properties

In this chapter, I introduce the concept of operator properties to the TACO system of Kjolstad et al. [21]. I describe how these properties, along with the fill value of the input tensors to the operators can be used to reason about the iteration space of the operator. I show that these properties allow the compiler to infer the iteration space algebra and hence the iteration pattern needed to compute the operator. I also discuss some limitations of the properties and consider iteration spaces that they cannot describe. I demonstrate that in these cases, a user can simply supply the compiler with the desired algebra that describes the space they would like to iterate.

## 5.1 Operator Properties

This work introduces six operator properties to the Tensor Algebra Compiler. Properties are another piece of information that can be tagged with an operator along with the iteration algebra and the special region definitions. This list is in no way exhaustive; it may be beneficial to add more operators in the future. Each operator can be tagged with a list of properties which can be any one of the following properties:

1. **Associative** - This property is only applied to binary operators. In an expression containing two or more occurrences of a binary operator, this property tells the compiler that the order in which it performs the computations does

not matter as long as the sequence remains the same. This tells the compiler that it can parallelize reductions over the binary operator.

2. **Commutative** - This means that the order in which the operands appear in the operator does not change the result. An operator with this property allows TACO to rearrange forall loops in concrete index notation, and therefore the reorder command in the TACO scheduling language [30] requires that the operands affected by changing the loop ordering are used by operators with the commutative property.

3. **Idempotent** - This is only defined for binary operators. An idempotent operator is one where the operator can be applied multiple times without changing the result beyond the first application. The idempotent law for an operator $\bullet$ states that $\forall x,\ x \bullet x = x$. This property helps the compiler realize when it does not need to iterate over the fill values when performing reductions. It also helps the compiler infer the iteration algebra for a given operator. Examples of idempotent operators are max and min.

4. **Annihilator** - The annihilator property specifies a value $a$ for an operator $\bullet$ such that $\forall x,\ x \bullet a = a$. This property is used to short circuit during reductions and to infer what the iteration algebra should be for a given operator based on the fill value of its operands. An example of an operator which benefits from this is multiplication, which has an annihilator of 0.

5. **Identity** - The identity property specifies a value $i$ for an operator $\bullet$ such that $\forall x,\ x \bullet i = x$. This property helps the compiler realize when it can skip reducing all of the fill values of the input tensors since, if the fill value is the identity value, explicitly reapplying it does not change the result. It also helps with inferring what the iteration algebra of an operator should be based on its input tensors.

6. **Distribution** - This is a special property that specifies a relationship between two operators $\bullet$ and $\odot$. This tells the compiler that $\bullet$ distributes over $\odot$ and is

essential for performing the workspace transformation [20] for an arbitrary pair of operators. In this work, the operators $\bullet$ and $\odot$ must be binary operators. So the distributive property tells the system that:

$$a \bullet (b \odot c) = (a \bullet b) \odot (a \bullet c)$$

For example, $\times$ distributes over $+$, $+$ distributes over max and min and $\wedge$ distributes over $\vee$. I show this rule concretely for the $(\max, +)$ pair of operators below. In this case, the max assumes the usual role of addition while $+$ assumes the usual role of multiply.

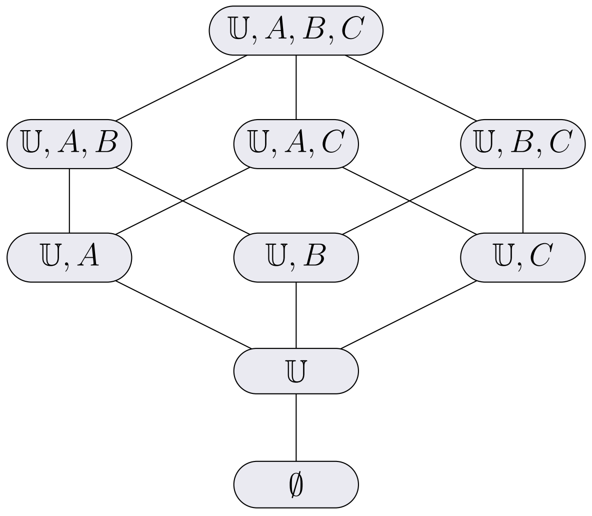
$$a + \max(b, c) = \max(a + b, a + c)$$

As mentioned, it is possible to use some of these scalar properties in order to infer what the iteration algebra given the operands and the properties. The rest of this chapter details the method for inferring iteration algebra from properties and operand fill values then discusses iteration spaces that the properties above are not sufficient to describe.

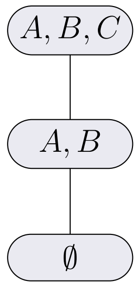
## 5.2 Deriving Algebras from Properties

Of the six operators introduced, TACO can leverage four of those in order to automatically infer an iteration algebra for a given operator. TACO attempts to generate the fewest number of lattice points needed to correctly evaluate an expression since this generally reduces the overall code size since there fewer loops and conditionals that have to be generated. The properties are checked in the order below, since properties earlier in the list will generate algebras resulting in weakly fewer lattice points that those later in the list. This derivation is only attempted if the user did not specify an iteration algebra explicitly. In that case, this step is skipped and the user's algebra is used to lower the operator.

Each operator starts with a default algebra which is simply the universe $\mathbb{U}$. This implies that the compiler needs to iterate over every point in the dense space. I did not include an explicit representation of the universe in the set language so the algebra is initialized to the complement of the empty set in the implementation. In the examples below, I use the notation $T_f$ to refer to a tensor $T$ with a fill value of $f$.



(a) Default lattice where all points are visited

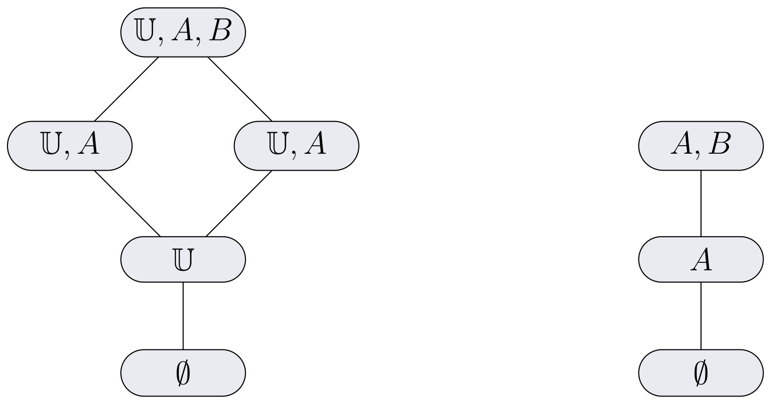


(b) Lattice generated for $A \cap B$.

Figure 5-1: Example lattice with and without generating an algebra for an operator with a commutative annihilator. The augmentation rule is crucial in both cases to inform the compiler of all the tensors forming the iteration space.

1. **Commutative Annihilator:** The system gathers all tensors with fill values equal to the annihilator and intersects them. This is because any time one of those input tensors is missing a coordinate, we know that its fill value will annihilate the expression meaning there is no need for it to be explicitly evaluated. In the case where there is only one tensor with its fill value equal to the annihilator, the system generates an algebra only for that tensor.

   For example, I consider the ternary max operator $\max(A_\infty, B_\infty, C_0)$ with $\infty$ specified as an annihilator. Since max is commutative, the compiler will check the fill values of all of the tensors and since only $A$ and $B$ have fill values of $\infty$, the compiler will generate an algebra for $A \cap B$ and override the default algebra of $\mathbb{U}$. This is because outside of the intersection of $A$ and $B$ at least one of the inputs to max will be $\infty$ meaning the result will be $\infty$. Figure 5-1a illustrates the lattice that would be generated if the default algebra was used instead of the generated algebra while Figure 5-1b shows the lattice produced for the generated algebra.



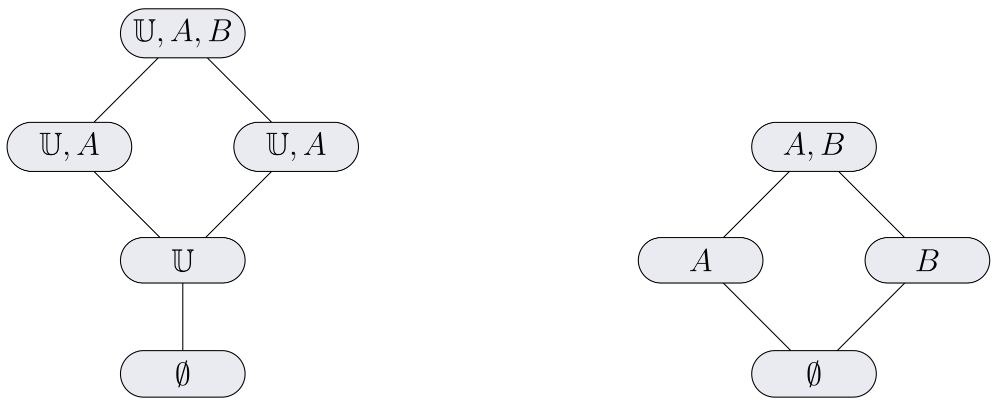
(a) Default lattice where all points are visited

(b) Lattice generated for $A$.

Figure 5-2: Example lattice with and without generating an algebra for an operator with a non-commutative annihilator. This optimization allows the compiler to skip considering points where $A$ is not defined. However, it must still co-iterate with $B$ to actually perform division where the values of $B$ are defined.

2. **Non-commutative Annihilator:** In this case, the user must specify which arguments in the operator can annihilate the expression. This is done by passing in a vector of argument positions that can annihilate along with the annihilator value. The system uses this value and the vector of argument positions to perform the same check as above with the exception that it only considers the tensors that the user specified.

   To illustrate an example, I consider the division operator $\text{div}(A_0, B_4)$. The user can pass in a property to the operator as follows `Annihilator(0, {0})`. This tells the compiler that the $0^{th}$ operand annihilates the expression when it has a value of 0. Since division is not commutative and the annihilator has a position value, the compiler will only check the fill value of $A$ and will realize that is is 0. This will cause it to generate an iteration algebra that only iterates over $A$ and ignores everywhere where $A$ lacks a coordinate since it knows that the result of the expression will be 0. Therefore, the default algebra of the operator, $\mathbb{U}$ will be overridden to contain the algebra of $A$. Figure 5-2a illustrates the lattice that would be generated without using the properties while Figure 5-2b illustrates the lattice generated when the algebra is inferred to be $A$ from the properties. In the cases where $A$ has a coordinate but $B$ does not, the compiler will perform the division using the fill value of $B$.



(a) Default lattice where all points are visited

(b) Lattice generated for $A \cup B$.

Figure 5-3: Example lattice with and without generating an algebra for an idempotent operator. This optimization allows the compiler to skip considering the points where both $A$ and $B$ lack coordinates.

3. **Idempotent Operator:** This is similar to the commutative identity case. If we fall through to this case, the system checks if all of the input tensors have the same fill value. If this is true and the operator is idempotent then the system generates an algebra to iterate over the union of the tensors. This is because of the idempotent law above; if the two fill values are the same then applying the operator to them will not change the result.

   Here, I consider the min operator $\min(A_x, B_x)$. I do not set the fill values to any specific value for this example since for idempotent operators, the only requirement is that the fill values are equal. A user can specify that an operator is idempotent by specifying the `Idempotent` property in the operator's property list. Since $A$ and $B$ have the same fill value in this example, the compiler can infer that the result of the operator when both tensors lack a coordinate is $x$. This means that the compiler will replace the default iteration algebra with the algebra $A \cup B$ since the code generator only needs to consider coordinates where either $A$ or $B$ is defined. Figure 5-3a illustrates the lattice without the properties while Figure 5-3b illustrates the lattice generated for computing the minimum of these two tensors.

4. **Commutative Identity:** Once we fall through to this case, the best the system can do is generate a union algebra. If all of the input tensors have a fill value that is the identity of the operator then TACO can emit an algebra to iterate over the union of the sparse spaces to avoid computing on the entire dimension. TACO can also emit a union algebra if only one of the input tensors has a fill value that is not the operator identity. This case is examined in the example below.

   An example of an operator where this is useful is the max operator. Suppose the operator is called as $\max(A_{-\infty}, B_{42})$ and $-\infty$ is specified as the identity of the operator. The identity of max is $-\infty$ since $\max(-\infty, x) = x$. In this example, TACO realizes that since only one operand has a fill value that is not the identity, it can infer the result of the computation when both tensors lack a defined coordinate. More concretely, since $A$'s fill value is $-\infty$ and is the identity of the operator, TACO knows the result has to be $B$'s fill value which is 42. This means that the compiler can emit a union algebra since when both $A$ and $B$ lack a coordinate, it knows the result of the computation so it does not need to consider this space. The iteration lattices are the same as the ones shown in Figure 5-3a for the default algebra and Figure 5-3b for the union algebra inferred from the operator's properties and fill values.

5. **Non-commutative identity:** Again, the user must specify which arguments to the operator can actually be identity elements using the same interface as the non-commutative annihilator case. We can only generate a union algebra here if all of the tensors are specified as possible identity elements and all of the

fill values are equal to the identity element. If this does not hold then we need to generate an algebra for the entire space.

However, the code generator can avoid using tensors whose fill values are the identities when performing computation since it will know that the identity element will not change the result. Currently, this is only possible for binary operators since the generator can simply return one side of the operator when the other is the identity element.

In general, the result tensor can have any fill value. Therefore, these algebras can only be generated if either the result tensor's fill value is left undefined (it is illegal to do this for any other tensor), allowing TACO to constant to replace the tensors in the expression with their fill values and constant propagate to set the fill value of the result. Once this is done, TACO can leverage the operator properties.

## 5.3 Limitations of Properties

As seen from above, properties cannot generate algebras for every iteration space. For example, we cannot infer a symmetric set difference or infer that we need to iterate over a tensor $A$ without its intersection with other tensors. In general, it is unclear how to extract the notation of complements from properties so any iteration space requiring a complement cannot currently be inferred. It is for this reason that the algebra was exposed directly to the user.

# Chapter 6

# Code Generation

In this chapter I explain how the representations described earlier are used to extend the sparse tensor algebra compiler code generator described by Kjolstad et al. [22, 21] to generate code for any iteration space as well as for any fill value. I will begin by describing how an iteration algebra is extracted from a general expression before it is fed as input to the lattice construction algorithm. I will then describe places where the code generator needs to change to account for the extensions made to the iteration lattices described in Chapter 4. Finally, I will describe how the code generator can use fill values and the properties of operators to realize certain optimization when performing generic contractions and relate these optimizations to the BFS and Bellman-Ford algorithms described in Chapter 2.

## 6.1 Extracting Algebra from Index Notation

The first change to the code generation algorithm is that instead of just lowering Index Expressions, we must first generate iteration lattices from explicit iteration algebra so that the correct loops and cases can be constructed. Prior to this work, it was possible to simply recurse on the index expressions since the iteration lattices were inferred directly from the operator. To convert index expressions to algebra, we recurse down each node in the expression tree and query these nodes for the algebra that should be used. We combine the algebra of different nodes by combining the iteration algebras returned by their operands. This is shown schematically in figure Figure 6-1.

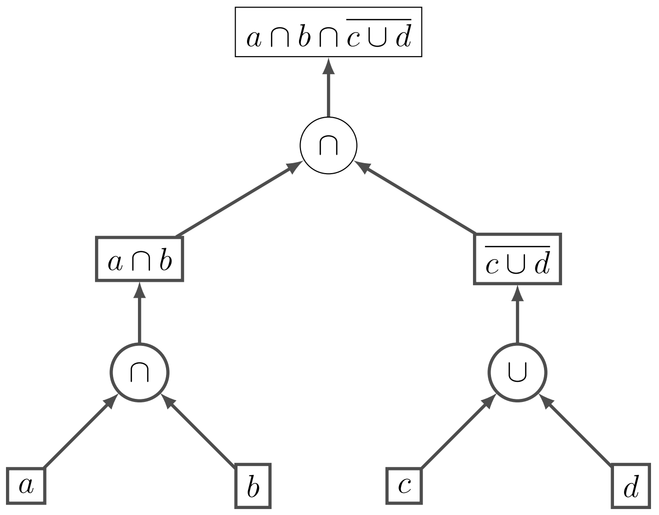


Figure 6-1: The above shows a schematic of how the algebra is extracted from an index expression. Assume an expression in the form foo$(a \times b, c + d)$ where foo has the algebra $l \cap \overline{r}$. The above depicts how the algebra is derived by combining nodes bottom up in the expression tree. Complements are applied to each operand which are surrounded by boxes and operands are either unioned or intersected and combined at the circles above.

## 6.2 Lowering Iteration Lattices

The construction algorithm describe in Chapter 4 is used to generate iteration lattices from the extracted iteration algebra. I will describe the changes made to the lowering mechanism in TACO to account new concepts introduced in the iteration lattices and the minimal changes made to case and loop generation.

### 6.2.1 Explicit and Implicit Fill Values

The code generator interprets complements to mean that it should skip computation everywhere a tensor has a non-fill value. Since complements are expressed by omitter points, this means that the code generator needs to check for two conditions when deciding if to omit computing using certain tensors. These are:

1. The sparse tensor data structure does not have a coordinate for some point in the iteration space. This means that the structure has an implicit value at that coordinate which is equal to the fill.

2. The value at some coordinate is equal to the fill value. In cases where a coordinate is defined and equal to the fill value, this is called an explicit fill.

When deciding if to omit computing using certain tensors, both conditions need to be satisfied for sparse tensors in order to omit the region. However, to omit a dense tensor region, only condition two needs to be satisfied.

The explicit fill value checks allow the code generator to ensure that the semantics of the lattice for sparse tensors matches the semantics for dense tensors. Without the explicit checks omitter points including a dense space would remove the entire iteration space. However for sparse spaces, it would only remove the coordinates that have been defined in tensor's data structure. This would break the core principle of the compiler generating the same results regardless of the format used.

The code generator will only perform checks for explicit fill values when there is an omitter point present in the lattice or there are special regions of computation defined in the lattice.

### 6.2.2 Loop Generation

The way the code generator generates loops from the merge lattice changes minimally from the original work. One difference is that the part of the code generator that lowers loops is always passed a case lattice. The code generator must now remove lattice points that have the same iterators itself in order to get a lattice describing the loops that need to be lowered. Loops are lowered for producer points as well as omitter points. Figure 4-29 demonstrates the application of the subset optimization to a case lattice and shows the resulting loop lattice which lets the code generator know that only one loop is needed over the tensor $a$.

### 6.2.3 Case Generation

Inside the loop body, the code generator emits cases from the full case lattice. If the input lattice only consists of producer points without any special region definitions, the code generator uses the original method of generating cases within the loop body [22, 21]. However, if this is not the case the code generator emits a boolean flag indicating whether a tensor value is non-zero. It emits this check for each tensor whose value array can be read at the current level. This flag is true if the tensor coordinate is defined and the value at that coordinate is non-zero. Since coordinates are always defined for dense structures the code generator does not perform the coordinate check for dense tensors.

There are two ways in which one could perform the case checks with the boolean flag. The first is that the method used by the current work which is to check if increasingly larger subsets are defined. In this case, compute statements would be excluded for omitter points meaning omitter points would simply have empty conditionals. The other option is to only emit checks for producer points. In this case, the code generator sometimes needs to include extra checks to ensure that it doesn't accidentally compute in a region that should be omitted. I implement the latter strategy since it seemed more readable but I show the two possible strategies in Figure 6-2 for performing a symmetric difference. These strategies may exhibit different performance based on the input data so it may be best to leave the decision about what case generation format to use for the scheduling language.

In order to figure out if extra checks are necessary, the code generator queries the lattice to figure out if any other lattice points are omitters and are subsets of the current lattice point. Specifically, the code generator checks if there are any parent lattice points that are omitters and contain the same tensors as the current lattice point. If this is true, the code generator needs to emit a check to ensure that it is not in one of those subsets. To do so, it collects all tensors present in the omitter points in a set and removes all tensors from that set that are part of the current lattice point. Figure 6-3 demonstrates an example of this check.

```
bool a_is_nonzero = a_has_coord && a_has_value;
bool b_is_nonzero = b_has_coord && b_has_value;

if (a_is_nonzero && b_is_nonzero) {
} else if (a_is_nonzero) {
    c = a;
} else if (b_is_nonzero) {
    c = b;
}
```

(a) Old Cases: Here a case is generated for each lattice point and computation omitted for omitter points

```
bool a_is_nonzero = a_has_coord && a_has_value;
bool b_is_nonzero = b_has_coord && b_has_value;

if (a_is_nonzero && !b_is_nonzero) {
    c = a;
} else if (b_is_nonzero && !a_is_nonzero) {
    c = b;
}
```

(b) New Cases: A case is only generated for producer points and extra checks are added when necessary to ensure we are not in some other region.

Figure 6-2: Two potential ways for generating the cases for the symmetric difference lattice such as the one shown in Figure 4-16b

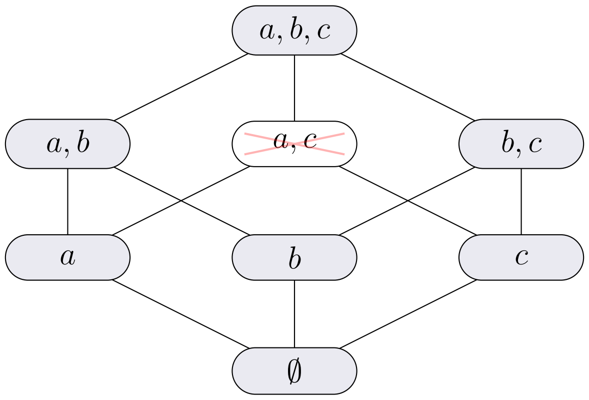


Figure 6-3: This demonstrates how the code generator handles lowering cases. When lowering the merge point for $a$, the code generator sees that $(a, c)$ is an omitter and realizes it must emit a check to ensure that $c$ is zero before computing within $a$. However, when emitting a check for $b$, the code generator knows that it has emitted checks for all of the subsets within $b$ so will avoid added extra checks in the if statement.

### 6.2.4 Special Regions

To emit special compute statements described in Section 4.2.4, the code generator leverages the fact that TACO zeros tensors that are exhausted. That is, when we are lowering a lattice point that does not contain an iterator for some tensor expression, it is removed from the index expression. In this work, 'zeroing' means replacing the tensor with its fill value. However, the code generator also keeps track of the indices of the tensors that are still defined for a given operator. Since the user gives a list of the argument indices that must be defined in order to emit some special code, the code generator queries the operator for this list. If the code generator's current list of defined arguments matches one of the lists the user gave as input, the code generator substitutes the users computation in the lowered code.

## 6.3 Reduction Optimizations

Currently, the code generator can leverage three of the operator properties when emitting reductions. It does this by first attempting to infer the fill value of the tensor expressions during lowering and comparing that fill value against properties. This means that it replaces all the tensors in the expression tree with their fill values and attempts to simplify this to a single Literal. For now the code generator uses properties to attempt to simplify the expression with the tensors replaced by fill values. This can be generalized by allowing the TACO system to constant fold through the low-level IR so that any expression can be simplified regardless of its operator properties. The properties used to simplify reductions are useful since without them, the system would be forced to assume it needed to iterate over the entire dimension even in the sparse case instead of simply reducing over the defined values. The three properties I currently use in the code generator are as follows:

1. **Identity:** If the fill value of the tensor expression being reduced is equal to the identity of the operator then for sparse tensors this tells TACO it can skip iterating over the entire dimension since the explicit fill values will not change the result.

   The code generator implements this by checking if there is a reduction being performed inside of an Assignment Node. If this is the case, it infers the fill value of the tensor expression being reduced after removing all tensors that are not indexed by the forall loop variable that is currently being lowered. This is so that the computation controlled by that loop can be captured by the code generator. Once the generator infers the fill value for the input expression with unneeded tensors removed, it checks if the fill value of that expression is the identity of the reduction operator. If this is the case, then the code generator leaves the lattice returned by the construction algorithm as is. However, in the event that the fill of the expression is not the operator identity, the code generator unions a dimension iterator forcing the fill values of the sparse tensors to be considered.

An example of this is the well-known SpMV kernel which can be expressed in TACO as $y(i) = A(i,j) \times v(j)$. Since there is an implicit reduction using addition being performed, the code generator will check the fill value of $A$ against the addition identity when lowering the $i$ loop. When lowering the $j$ loop, it will check the fill value of the expression $A(i,j) \times v(j)$ against the identity of the addition operator.

I note that this optimization alone allows TACO to realize when the identity of the addition is equal to the annihilator of the multiplication operator as is the case in a semi-ring. In the SpMV example for instance, if the fill of $A$ is 0, TACO keeps the $i$ loop sparse since 0 is the identity of addition. Additionally, since $0 \times v(j) = 0$ (because 0 annihilates multiply), the inferred fill value of the expression in the reduction loop $j$ is 0. Again, since 0 is the identity of the addition operator, the $j$ loop also remains sparse due to the relationship between the annihilator and the identity of the operators being used in the expression.

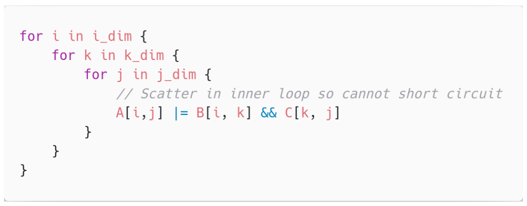
2. **Idempotent:** If reducing over an idempotent operator, the code generator can simply initialize the result variable to be the fill value of the reduction expression and iterate over the sparse space. This is because the fill value of the expression is some constant $c$ and applying idempotent operator repeatedly to a constant does not change its value.

   For example, suppose in the SpMV expression above the reduction was over a max instead of addition. Since $\max(0,0) = 0$ then there is no need to explicitly apply the max operator to all the fill values. As a result, the code generator only needs to initialize the reduction variable to 0 then perform the reduction as normal.

3. **Annihilator:** If the reduction operator has an annihilator, then the code generator can insert code to short circuit only if the reduction is being performed into a scalar. That is, we cannot short circuit scatter reductions since it is possible that value updates would be missed by erroneously exiting early.

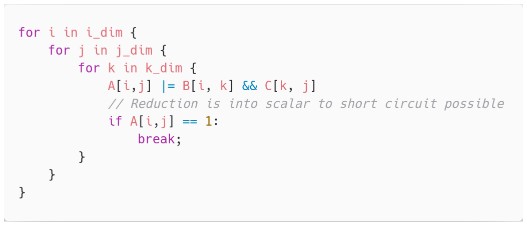
   To check if the reduction is into a scalar, the code generator simply checks the ordering of the forall loops in concrete index notation and sets a flag indicating if the reduction is a scatter. Reductions into scalars will always maintain a loop ordering with all of the loops over free variables coming before all of the loops over reduction variables. If any loop over a reduction variable appears before a free variable loop, then a scatter is being performed. Figure 6-4 demonstrates loop orderings where short circuiting can and cannot be performed.

   If the code generator detects that reduction is being done into a scalar, it will emit a condition in the reduction loop that checks if the value of the reduction variable is the annihilator of the reduction operator. If this is true, if will break out of the loop to early exit.



```
for i in i_dim {
    for k in k_dim {
        for j in j_dim {
            // Scatter in inner loop so cannot short circuit
            A[i,j] |= B[i, k] && C[k, j]
        }
    }
}
```

(a) A loop ordering where short circuiting is not possible



```
for i in i_dim {
    for j in j_dim {
        for k in k_dim {
            A[i,j] |= B[i, k] && C[k, j]
            // Reduction is into scalar to short circuit possible
            if A[i,j] == 1:
                break;
        }
    }
}
```

(b) A loop ordering where short circuiting is possible

Figure 6-4: Loop orderings for the expression $A(i, j) = B(i, k) \wedge C(k, j)$ where I assume the reduction is being performed over the logical or operator. In the first case, the code generator cannot short circuit because the reduction variable $k$ appears before the free variable $j$. However, in the second case short circuiting is possible since the loops appear with the free variables loops followed by the reduction variable loops.

# Chapter 7

# Operator API

I expose the iteration algebra explicitly in the TACO system. This is needed since the iteration algebra was decoupled from scalar computation performed inside the loop in this work and because the properties of the operators were not sufficient to describe any iteration space. I also introduce a new class called the Properties class which is a super class for all of the properties described in Chapter 5. Lastly, I expose a class needed to define new operators in taco as a new class called TensorOp. This class takes in a scalar implementation for a function and optionally an iteration algebra, properties of the operator and scalar implementations for special regions described in Chapter 4. This new class is an extension of the IndexNotation class in TACO.

## 7.1 Iteration Algebra Class

The declarations below are the constructors that a user would use to create an IterationAlgebra expression.

```
Region(IndexExpr expr);
Complement(IterationAlgebra alg);
Intersect(IterationAlgebra left, IterationAlgebra right);
Union(IterationAlgebra left, IterationAlgebra right);
IterationAlgebra(IndexExpr expr);
```

These classes only have one method called accept which allows algorithms to be expressed over these data structures using the visitor pattern. I now show how a user could define the symmetric difference set algebra using the classes above.

```
Tensor<double> a("A", {NUM_I}, {Compressed});
Tensor<double> b("x", {NUM_I}, {Compressed});

IndexVar i, j;
IterationAlgebra tensorUnion = Union(a(i), b(i));
IterationAlgebra noIntersect = Complement(Intersect(a(i), b(i));
IterationAlgebra symDiff = Intersect(tensorUnion, noIntersect);
...
```

The constructors are simple and self explanatory but I explain them briefly below:

`Region(IndexExpr expr)`: This instantiates a region in the iteration space that one can intersect, union and complement. These regions can be arbitrary tensor expressions and their iteration algebras can be extracted when combining iteration spaces.

`Complement(IterationAlgebra alg)`: Instantiates a class that represents the complement of the input iteration algebra. That is, represents complement(alg).

`Intersect(IterationAlgebra left, IterationAlgebra right)`: Instantiates a class that represents the intersection of the left and right algebras.

`Union(IterationAlgebra left, IterationAlgebra right)`: Instantiates a class that represents the union of the left and right algebras.

`IterationAlgebra(IndexExpr expr)`: Instantiates an IterationAlgebra region from an IndexExpression.

## 7.2 Properties

In addition to the algebra, the six properties explained in Chapter 5 were also introduced as new classes. Their class constructors are shown and explained below.

`Commutative(std::vector<int>& positions):` Instantiates a properties class that implies that the associated operator is commutative. The positions list is optional and omitting it implies that the operator is fully commutative (any of the elements can be swapped). However, if only some subset of the arguments to the operator can be swapped while maintaining the correctness of the operator, then their argument positions can be specified.

`Associative():` Instantiates a properties class that implies that the associated operator is associative. The operator tagged with this property must be a binary operator.

`Idempotent():` Instantiates a properties class that implies that the associated operator is idempotent. The operator tagged with this property must be a binary operator.

`Annihilator(Literal annihilator, std::vector<int>& positions):` Instantiates a properties class that implies that associated operator has annihilator given by annihilator. The positions list can be left empty implying that any input can annihilate the operator. However, a list of positions in the operator's arguments can be specified to let the compiler know that only the inputs at these specific positions can annihilate an expression.

`Identity(Literal identity, std::vector<int>& positions):` Instantiates a properties class that implies that associated operator has identity given by identity. The positions list can be left empty implying that any input can annihilate the operator. However, a list of positions in the operator's arguments can be specified to let the compiler know that only the inputs at these specific positions can annihilate an expression.

`Distribute(TensorOp op1, TensorOp op2):` Instantiates a properties class that informs the code generator that op1 distributes over op2.

## 7.3 TensorOp Class

Before I describe the TensorOp class, I first introduce two typedefs to simplify the operator definition. I introduce an opImpl which is a functor that takes in some scalar expressions in the TACO IR and returns a single expression representing the computation that should occur using the input. Similarly, algebraImpl is also a functor that takes in some IndexExprs and returns an iteration algebra describing how the regions described by those IndexExprs should be iterated. These typedefs are shown below:

```
typedef std::function<ir::Expr(const std::vector<ir::Expr>&)> opImpl;
typedef std::function<IterationAlgebra(const std::vector<IndexExpr>&)> algebraImpl;
```

Now I introduce the constructor of a TensorOp. The only required input to construct the operator is the lowererFunc.

```
TensorOp(std::string name, opImpl lowererFunc, algebraImpl algebraFunc,
         std::vector<Property> properties,
         std::map<std::vector<int>, opImpl> specialDefinitions = {});
```

The arguments of the TensorOp constructor are as follows:

1. **name:** This is the name of the operator that will be used when debugging. If no name is given, the compiler generates a random name for the operator.

2. **lowererFunc:** This is a functor that the lowerer calls to get the TACO imperative IR expressing the computation this operator performs. The functor takes in a vector of exprs that it combines based on the body in the IR. Part of the reason a functor is used is because it makes the issue of matching formals to actuals trivial. Additionally, a user has more flexibility with a functor as they can analyze the arguments themselves and potentially return optimized IRs when certain arguments are certain values.

3. **algebraFunc:** This is a functor used to create an iteration algebra from a vector of IndexExpr. If this is functor specified, TACO will not attempt to infer the algebra from the operator properties and will trust what the user gave. If this is not specified, and no properties are given, TACO will attempt to iterate over the entire space.

4. **properties:** A vector of properties which can be any of the six properties shown above. It is illegal to have duplicate properties in this vector.

5. **specialDefinitions:** A map containing a vector of indices that must be defined and the corresponding computation to perform when these indices are defined. By default, when a tensor is exhausted or not involved in an expression TACO replaces it with the tensor's fill value. However, if this region map specifies that some special operation needs to be performed when a tensor is defined then the IR given by the user is used instead.

Once a TensorOp has been instantiated, there is one method that the user can call on the TensorOp which is shown below:

```
TensorOp operator()(IndexExprs&&... exprs);
```

The call operator is overloaded to take in an arbitrary number of index expressions. The order of the expressions passed in here is the same as the order of the expressions passed to the algebra and lowerer functions.

## 7.4 API Example

Appendix A shows a comprehensive example using operators on a series of graph algorithms. However, I include a simple example here to show how one can compute the symmetric difference between two tensors. This has a relatively complex algebra and illustates the use of both the algebra and the operator properties.

```
// Define a functor to return the algebra for XOR.
struct XorAlgebraGenerator {
  IterationAlgebra operator()(const std::vector<IndexExpr>& regions) {
    // First, remove the intersection
    IterationAlgebra noIntersect = Complement(Intersect(regions[0], regions[1]));
    // Intersect the set without the intersection with the union of the sets to
    // remove the background.
    return Intersect(noIntersect, Union(regions[0], regions[1]));
  }
};

// Define a functor that actually performs XOR.
struct XorImpl {
  ir::Expr operator()(const std::vector<ir::Expr> &v) {
    return ir::Xor::make(v[0], v[1]);
  }
};

// Default fill value is 0.
Tensor<double> a("a", {NUM_I}, {Compressed});
Tensor<double> b("b", {NUM_I}, {Compressed});
Tensor<double> c("c", {NUM_I}, {Compressed});

...

IndexVar i;
Op xorOp(XorAlgebraGenerator(), XorImpl(), {Associative(), Commutative(), Identity(0)});
c(i) = xorOp(a(i), b(i));
```

Figure 7-1 demonstrates the code generated for the example in this section. The algebra causes the compiler to skip computing when in the intersection of the two tensors. This means that in the sparse case, the compiler never actually emits an xor operation since the loop structure can describe the operator.

```
int32_t ic = 0;
int32_t ia = a1_pos[0];
int32_t pa1_end = a1_pos[1];
int32_t ib = b1_pos[0];
int32_t pb1_end = b1_pos[1];

while (ia < pa1_end && ib < pb1_end) {
  int32_t ia0 = a1_crd[ia];
  int32_t ib0 = b1_crd[ib];
  int32_t i = TACO_MIN(ia0,ib0);
  bool a_isNonZero = ia0 == i && a_vals[ia] != 0;
  bool b_isNonZero = ib0 == i && b_vals[ib] != 0;
  if (a_isNonZero && !b_isNonZero) {
    c_vals[ic] = a_vals[ia];
    ic++;
  }
  else if (b_isNonZero && !a_isNonZero) {
    c_vals[ic] = b_vals[ib];
    ic++;
  }
  ia += (int32_t)(ia0 == i);
  ib += (int32_t)(ib0 == i);
}
while (ia < pa1_end) {
  c_vals[ic] = a_vals[ia];
  ic++;
  ia++;
}
while (ib < pb1_end) {
  c_vals[ic] = b_vals[ib];
  ic++;
  ib++;
}
```

Figure 7-1: Code for computing symmetric difference with sparse tensors.

# Chapter 8

# Evaluation

I carry out experiments to compare the performance of the code generated by this technique with other libraries for Breadth-First search. I chose to focus on this kernel since it is prevalent in the performance engineering literature. However, the generality of this work extends to more applications such as the Viterbi algorithm or general sparse array operations similar to numpy. This work also allows graph algorithms other than BFS to be expressible in TACO since custom operators, properties and general fill values allow users to define arbitrary semi-rings. I demonstrate that this work has the potential to deliver good performance on these graph algorithms as TACO's scheduling language matures to support parallel sparse assembly.

## 8.1 Methodology

The work in this thesis is implemented as an extension to the Tensor Algebra Compiler (TACO). To evaluate the performance of this work on CPUs, I compare against the optimized versions of the graph algorithms in the GAP Benchmark Suite [6], GraphIt (commit 7973721) and LAGraph 0.0.1 with SuiteSparse:GraphBLAS 3.2.2 as the back-end.

I used the five graphs from the GAP Benchmark Suite as inputs to test BFS on each library. The graphs used are listed in table 8.1 and represents a diverse range of properties. The kron and uniform graphs are synthetic graphs while the twitter, road and web graphs are from real world data. The twitter, kron and web graphs are scale-free graphs with a low diameter. The road graph is smaller than the others but has a high diameter which can cause issues for some implementations. The road graph is also weighted so the weights were removed to evaluate the performance of BFS since they are unneeded. The uniform graph represents the worst case with respect to locality since every vertex has an equal probability of being a neighbor of any other vertex.

| Name | Description | # Vertices (M) | # Edges (M) | Degree | Degree Distribution |
|---|---|---|---|---|---|
| Twitter | Twitter Follow links | 61.6 | 1.468.4 | 23.8 | power |
| Web | Web Crawl of the .sk Domain | 50.6 | 1949.4 | 38.5 | power |
| Road | Distances of roads in the USA | 23.9 | 58.3 | 2.4 | bounded |
| Kron | Synthetic Kronecker Graph | 134.2 | 2111.6 | 16 | power |
| Urand | Synthetic Uniform Random Graph | 134.2 | 2147.4 | 16 | normal |

Table 8.1: Description of Graphs used in Evaluation

I use the timing method specified in the GAP Benchmark Suite for timing to time CPU codes. That is, I do not include the time to load the graphs into shared memory. For all graphs, I load a CSR and CSC version in memory to allow for direction optimization in each kernel without including the transpose time in the algorithm run time. Since BFS has a single source, I repeat the algorithm 64 times with random vertices and report the average time across those 64 runs in order to account for the variance of starting at different nodes. The random number generator used to pick start vertices has a fixed seed so the same sequence of start vertices is picked for every framework tested. For LAGraph, the source vertices were generated using the GAP suite and stored in an Matrix Market file that the framework could read. Table 8.2 below summarizes the trials and sources for BFS.

| Algorithm | Trials | Sources |
|---|---|---|
| BFS | 64 | 64 unique sources |

Table 8.2: Summary of timing method for each algorithm evaluated.

All CPU experiments are run on a dual-socket, 12-core with 24 threads, 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache per socket and 128 GB of main memory, running Ubuntu 18.04.3 LTS. On CPU, I compile codes using g++ 7.5.0 with the compiler flags `-O3`, `-DNDEBUG`, `-march=native`, `-mtune=native`, `-ffast-math` and `-fopenmp`. I used `std::chrono::high_resolution_clock` to record execution times on CPU. Correctness was verified by comparing against the reference version of the algorithm in the GAP Benchmark Suite.

### 8.1.1 Discussion of performance

In this section, I first compare the performance of TACO with and without the optimizations enabled by this work. I omit the road graph from this comparison since it has very long run times in both versions of TACO and it almost exclusively uses the SpVSpM kernel which does not take advantage of the masking and short circuiting capabilities introduced in this work. Figure 8-1 is a graph showing the performance of TACO on all of the graphs except road with and without the features of this work. This work is 5.08 times faster in the geomean compared to the prior version of TACO due to the ability to mask and short circuit.

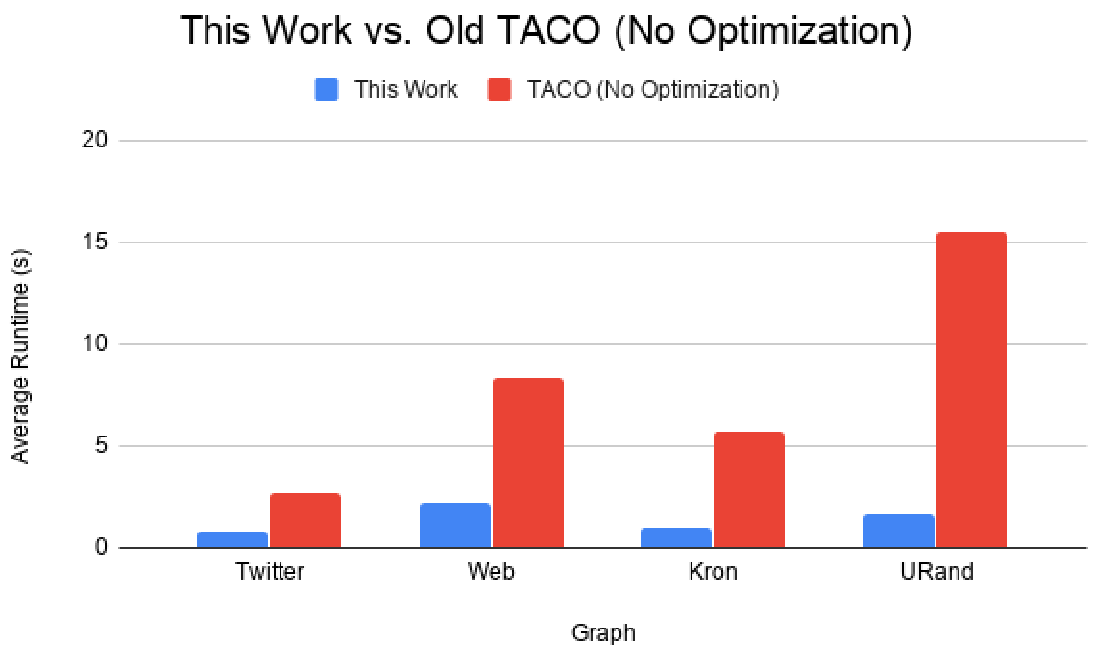


Figure 8-1: Comparison of TACO on BFS with optimizations enabled by using the features in this work (blue) and without (red)

| Graph | Runtime (s) (lower is better) | | | |
|---|---|---|---|---|
| | GAP | GraphIt | LAGraph | TACO |
| Twitter | **0.260** | 0.263 | 1.153 | 0.831 |
| Web | **0.382** | 0.486 | 1.325 | 2.212 |
| Road | 0.328 | **0.287** | 6.145 | 67.79 |
| Kron | **0.393** | 0.404 | OOM | 0.996 |
| URand | **0.637** | 1.038 | OOM | 1.653 |

Table 8.3: Average time in seconds for running BFS on each framework. The times are averaged after running 64 trials from the same start vertices across frameworks. OOM means the framework ran out of memory.

| Graph | Edge throughput (MTEPS) (higher is better) | | | |
|---|---|---|---|---|
| | GAP | GraphIt | LAGraph | TACO |
| Twitter | **5648** | 5583 | 1274 | 1767 |
| Web | **5103** | 4011 | 1471 | 881.3 |
| Road | 152.6 | **203** | 9.487 | 0.860 |
| Kron | **5373** | 5227 | OOM | 2120 |
| URand | **3371** | 2069 | OOM | 1299 |

Table 8.4: Million traversed edges per second (MTEPS) for each framework (Higher is better). OOM means the framework ran out of memory.

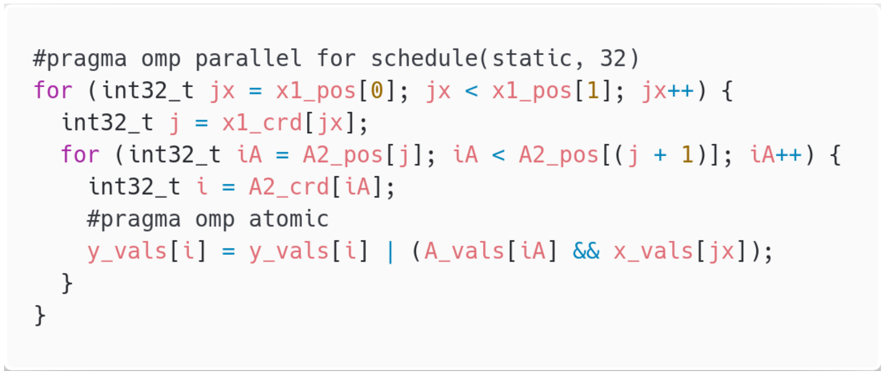
Table 8.3 and Table 8.4 shows the performance results obtained for Breadth-First search against state-of-the-art work. In terms of run time, TACO is slower on all of the graph algorithms with the GAP reference being the fastest. TACO was only able to beat LAGraph, which is a library for performing graph algorithms using linear algebra on the twitter graph. However, LAGraph failed to execute on the large graphs. The disparity in the run times can be explained by a few reasons:

1. when performing the top down step TACO must use a dense result whereas the other frameworks can maintain sparsity.

2. Before every top down step, the frontier must be converted into a sparse representation since both kernels give a dense output. This is done completely serially and takes one third to half of the total time to perform the top-down step.

3. The code generated by TACO has to use an atomic operation inside of the reduction loop in SpVSpM which induces another serial bottleneck as shown in

Figure 8-2.



```
#pragma omp parallel for schedule(static, 32)
for (int32_t jx = x1_pos[0]; jx < x1_pos[1]; jx++) {
  int32_t j = x1_crd[jx];
  for (int32_t iA = A2_pos[j]; iA < A2_pos[(j + 1)]; iA++) {
    int32_t i = A2_crd[iA];
    #pragma omp atomic
    y_vals[i] = y_vals[i] | (A_vals[iA] && x_vals[jx]);
  }
}
```

Figure 8-2: Code for the sparse vector sparse matrix kernel into a dense result. The sparse vector is $x$, the sparse matrix is $A$ and the result is $y$.

The addition of sparse parallel assembly to TACO would help to solve these problems by removing both serial bottlenecks in the current kernel. Firstly, one would only pay the sparse to dense conversion once when initially switching from bottom up traversal to top down traversal instead of at the beginning of each top down traversal. This would be greatly beneficial since this is currently done completely serially. Additionally, a more general way to perform scatter reductions into a sparse result could potentially be free of locks as seen in the works of Azad and Buluc [3] which would further reduce the amount of serialization.

The results support the claim that the top down kernel is the much slower than bottom up. The road graph has a very small average number of neighbors so the frontier rarely becomes large enough for the direction to switch to pull during BFS execution. This explains why it exhibits such a long run time in comparison to all the other frameworks.

Finally, even if the SpVSpM kernel is rectified, the pull direction in the GAP suite is up to 3x faster than TACO on the same BFS step. The code format looks extremely similar where both iterate over the graphs and employ masking and short circuiting. However, GAP uses special data structures for BFS since it knows it is operating on booleans. Namely, GAP packs the bits manually in `uint64_ts` and performs bit manipulations to get and set bits in the frontier. Additionally, it leverages the fact that the graph is a boolean with no explicit zeros stored in the matrix so that it does not need to store a value indicating that an edge is present. The presence of an edge is implicitly inferred by the presence of a coordinate. Both of these features reduce the memory traffic which is very beneficial to graph algorithms. These could potentially be supported in a TACO and exposed via the scheduling language, where a user can instruct TACO to pack larger integer types into smaller ones in general. Additionally, a scheduling or format primitive which instructs TACO that there are no explicit zeros in the tensor could enable it to omit storing the values of a boolean tensor. I leave implementing these features along with parallel sparse assembly as future work.

# Chapter 9

# Related Work

Many researchers have worked on compilers to generate code for computing on sparse tensors. Recently, researchers have explored performing graph algorithms using sparse linear algebra computations over general semi-rings. I discuss prior work from both categories in this section since this work blends the two fields.

**Sparse Compilers:** Prior work on sparse linear algebra compilers forms the foundation of this approach. Works by Bik and Wijshoff [7, 9] used a technique called guard encapsulation to transform code for computing on dense linear algebra into the analog sparse linear algebra code. Kotlyar et al. [23] soon followed with a relational algebra based framework that compiled efficient sparse matrix code from dense loops and a specification of the representation of the sparse matrix. Pugh and Shpeisman introduced SIPR [28] which was a new intermediate representation that separated the maintenance of the complicated sparse data structures from the actual matrix operations performed. Their new representation allowed for cost-analysis to predict program efficiency allowing them to compile dense code into the equivalent efficient sparse code. There has been recent work to use a sparse polyhedral framework [33, 32] to emit efficient code for computing on sparse data structures. Generally, these strategies inspects and reorders the data at run-time and executes the original computation on the newly transformed data-structures. Most recently, the Tensor Algebra Compiler (TACO) [12, 20, 21] emits efficient code that computes on sparse and dense tensors. It uses a description of the format each tensor is stored in along with the computation to be performed to generate efficient tensor algebra code. TACO is great for tensor algebra operations falling in the usual $(\times, +)$ semi-ring but it does not generalize to arbitrary semi-rings and does not allow computation outside of semi-rings. I build on TACO in this work allowing generic operations and introducing properties that TACO can explicitly use to perform its transformations. This generality allows TACO to be used to compile codes for a wide range of linear algebra based graph algorithms.

**Linear Algebra Graph Algorithms:** Pregel [24] introduced the vertex-centric framework for thinking about graph algorithms. It relies on message passing to send data among the vertices in the graph and updates vertex state based on the messages received. The edge-centric model for graph computations was introduced by PowerGraph [16] to address performance and scalability issues with vertex-centrix model for scale-free graphs. The edge-centric model is conceptually split into three phases - Gather, Apply, Scatter (GAS) where information is first gathered about adjacent vertices and edges to apply an update to the vertices which is then scattered to update data on adjacent edges. Lastly, linear algebra graph frameworks have been proposed. This began with work by Combinatorial BLAS [11] which exposes a small set of linear algebra primitives that target graph applications. One issue is that users often are unaware of how to express certain graph algorithms in linear algebra terms. GraphMat [34] attempts to bridge the gap between the vertex centric model and the linear algebra based frameworks by translating vertex centric graph algorithms to their linear algebra equivalents in the back end. This provides the productivity of the familiar vertex centric models without sacrificing performance. GraphBLAS [10] is an open source effort to define building blocks for graph algorithms in the language of linear algebra. SuiteSparse [15] was the first fully GraphBLAS compliant library and has a multi-threaded CPU version. GraphBLAST [39] is a GPU linear algebra based graph system inspired by GraphBLAS. It generalizes the concepts of semirings to allow users to express a wide range of graph algorithms using highly optimized kernels for operations on sparse matrices and vectors.

The current work on sparse compilers is very promising but it is restricted to one semi-ring. In this work, I lift this restriction to allow the sparse compiler approach taken by TACO to work on arbitrary semi-rings by allowing the compiler to exploit the properties of the operators in the expression. This work also allows operations outside of semi-rings to be defined and generalizes the concept of iteration spaces introduced in TACO enabling new optimizations to be defined that are beneficial to running graph algorithms.

# Chapter 10

# Conclusion and Future Work

This thesis presents a new framework for thinking about optimizations and operators on sparse tensors in terms of the operator properties and generalizes the concept of iteration spaces by introducing complements. I extended the iteration lattices from the previous work and I show how properties can allow a sparse tensor compiler to construct lattices and reason about the loop domains for reductions. I also introduce the concept of a general fill value to the compiler so that fill values no longer have to be just zero. The framework presented here can recreate general optimizations on graph algorithms in linear algebra but does not currently support sparse parallel assembly of tensor results which limits its performance on the graph algorithms tested. I believe that this work is a stepping stone to unlocking a sparse equivalent of numpy since it allows a wide range of operators to be supported efficiently and allows optimizations to be uncovered by simply reasoning about the properties of the operators used in some computation.

Future work includes implementing support for custom tensor types in the compiler to complement the custom operators presented here. This would give users more flexibility when describing algorithms. One instance is that with the sssp problem, it is possible to derive both path lengths and the actual source to target paths using one semiring instead of two [19]. One could also add support for sparse slicing to the compiler to match the kind of numpy-style slicing as well as generalize the notion of indices in index notation so that indices can be anything. This would allow TACO to be used in data science frameworks such as pandas as a backend and allow those users to benefit from the efficient code that it generates. Lastly, one could allow TACO operators to be implemented in C/C++ code instead of in the TACO IR. This would make it easier for users already familiar with C to introduce new operators to the compiler allowing it to be easier to leverage the operator abstractions expressed in this thesis. On the scheduling side, it could be useful to include a scheduling primitive to pack smaller types such as `int8_t`s into larger types such as `int64_t`s. This could be beneficial for a sparse compiler since algorithms are usually memory bound and this could help the compiler emit wider sizes reads and writes so more data is moved with each instruction. Lastly, parallel sparse assembly would be very beneficial since it would give users much more flexibility when using TACO's scheduling language.

# Appendix A

# Operator Implementations

## A.1 BFS Pull Implementation

```
struct BfsLower {
  ir::Expr operator()(const std::vector<ir::Expr> &v) {
    return v[0];
  }
};

struct BfsMaskAlg {
  IterationAlgebra operator()(const std::vector<IndexExpr>& regions) {
    return Intersect(regions[0], Complement(regions[1]));
  }
};

Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> f("f", {NUM_J}, {Dense});
Tensor<double> mask("mask", {NUM_I}, {Dense});
Tensor<double> v("v", {NUM_I}, {Dense});
...
```

```
A.pack();
f.pack();
mask.pack();
IndexVar i, j;
// Instantiate Operator
Op bfsMaskOp("bfsMask", BfsLower(), BfsMaskAlg());
IndexExpr bfsStep = bfsMaskOp(taco::And(A(i,j),f(j)), mask(i));
v(i) = taco::Reduction(taco::Or(), j, bfsStep);

IndexStmt stmt = v.getAssignment().concretize();

stmt = stmt.split(i, i0, i1, 16)
           .reorder({i0, i1, j})
           .parallelize(i0, ParallelUnit::CPUThread, OutputRaceStrategy::NoRaces);

v.compile(stmt);
v.assemble();
v.compute();
```

## A.2 BFS Pull Implementation

```
Tensor<double> A("A", {NUM_I, NUM_J}, CSC);
Tensor<double> f("f", {NUM_J}, {Compressed});
Tensor<double> mask("mask", {NUM_I}, {Dense});
Tensor<double> v("v", {NUM_I}, {Dense});
...

A.pack();
f.pack();
mask.pack();
IndexVar i, j;

v(i) = Reduction(taco::Or(), j, taco::And(A(i, j), f(j)));
IndexStmt stmt = v.getAssignment().concretize();

stmt = stmt.reorder(i,j)
           .parallelize(j, taco::ParallelUnit::CPUThread,
                        taco::OutputRaceStrategy::Atomics);

v.compile(stmt);
v.assemble();
v.compute();
```

# 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.

[2] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, 2015.

[3] Ariful Azad and Aydin Buluc. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm, 2016.

[4] S. Beamer, K. Asanovic, and D. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *2015 IEEE International Symposium on Workload Characterization*, pages 56–65, 2015.

[5] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[6] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2015.

[7] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, 1996.

[8] 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.

[9] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation Techniques for Sparse Matrix Computations. In *International Conference on Supercomputing*, pages 416–424. ACM, jul 1993.

[10] 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.

[11] Aydin Buluç and John Gilbert. The combinatorial blas: Design, implementation, and applications. *IJHPCA*, 25:496–509, 12 2011.

[12] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA):123:1–123:30, October 2018.

[13] 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.

[14] Timothy A. Davis. Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[15] Timothy A. Davis. Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019.

[16] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[17] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson,

Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[18] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José E. Moreira, John D. Owens, Carl Yang, Marcin Zalewski, and Timothy G. Mattson. Mathematical foundations of the graphblas. *CoRR*, abs/1606.05790, 2016.

[19] Jeremy Kepner and John R. Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[20] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. pages 180–192, 2019.

[21] 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.

[22] Fredrik Berg Kjolstad. *Sparse Tensor Algebra Compilation.* PhD thesis, Massachusetts Institute of Technology, 2020.

[23] 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.

[24] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[25] 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.

[26] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), July 2016.

[27] Travis E. Oliphant. *A guide to NumPy.* Trelgol Publishing USA, 2006.

[28] William Pugh and Tatiana Shpeisman. SIPR: A new framework for generating efficient code for sparse matrix computations. In *International Workshop on*

*Languages and Compilers for Parallel Computing*, pages 213–229. Springer, aug 1998.

[29] 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 Trans. Graph.*, 31(4), July 2012.

[30] Ryan Senanayake, Fredrik Kjolstad, Changwan Hong, Shoaib Kamil, and Saman Amarasinghe. A unified iteration space transformation framework for sparse and dense tensor algebra, 2019.

[31] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 135–146. ACM, 2013.

[32] Michelle Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. volume 7760, 09 2012.

[33] 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.

[34] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Satya Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. 03 2015.

[35] E. Theodosis and P. Maragos. Analysis of the viterbi algorithm using tropical algebra and geometry. In *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5, 2018.

[36] Scott Thibault, Lenore Mullin, and Matt Insall. Generating indexing functions of regularly sparse arrays for array compilers. 11 1996.

[37] R.J. Valenza. *Linear Algebra: An Introduction to Abstract Mathematics*. Undergraduate Texts in Mathematics. Springer New York, 2012.

[38] Yangzihao Wang, Yuechao Pan, Andrew A. Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *CoRR*, abs/1701.01170, 2017.

[39] Carl Yang, Aydin Buluc, and John D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu, 2019.

[40] 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.

[41] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.