Data-flow Analyses for Structured Sparsity Propagation

by

Danila Seliayeu

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

Machine learning and high-performance computing (HPC) applications rely on computational graphs that often process tensors containing zeros, frequently in a structured way. An optimizer can exploit zeros that propagate through chained multiplications to enable various optimizations. When an analysis determines that entire sections of a tensor must contain zeros, the implementation can avoid storing those zeros and instead store the locations of non-zero values, reducing memory usage. Traversing tensors using this information can also allow the computation to skip entire sections of the tensor, leading to speedup. Both optimizations apply to linear algebra domains where speed and memory usage matter significantly.

This thesis presents two data-flow analyses that propagate structured sparsity through a computational graph. Although this thesis evaluates the analyses in the context of propagating zeros for tensor operations in machine learning and HPC, both analyses generalize to cover algebraic semirings with an annihilating element.

The first analysis, Sparsity Propagation Analysis (SPA), determines which tensor slices must contain zero values during execution of the computational graph, enabling sparsity optimizations. In the symbolic execution of a single matrix multiplication benchmark, SPA takes only $29\mu s$, compared to up to $0.85s$ for the state-of-the-art SparTA [74]. This thesis implements SPA on top of the Tensor Algebra Compiler (TACO) [37]. In a chained matrix multiplication benchmark, SPA achieves speedups and memory reductions once sparsity reaches 50%, demonstrating its efficacy.

The second analysis, Bandedness Propagation Analysis (BPA), similarly enables sparsity optimization, but propagates banded structured sparsity. BPA scales linearly with the size of the computational graph, and experiments show that its symbolic execution runs at least 1000× faster than both SparTA [74] and StructTensor [25]. This thesis implements BPA in the Multi-Level Intermediate Representation (MLIR) framework. BPA leads to performance and memory improvements for 1024×1024 tensors with upper and lower bandwidths of 64.

Preface

The research presented in this thesis was made possible through a joint research project between my supervisor, José Nelson Amaral, and Fernando Magno Quintão Pereira from the Universidade Federal de Minas Gerais (UFMG) in Brazil, through the "Projeto e Implementação de Otimizações de Código para Modelos de Aprendizagem Profunda" financed by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). This funding enabled me to spend time in residence at the Laboratório de Compiladores at UFMG and also enabled one of my co-authors, Kaio Henrique Andrade Ananias Rodrigues, to spend a year in the Compiler Design and Optimization Laboratory at the University of Alberta. This thesis presents my contributions to this collaborative research and this preface specifies my specific contributions to this work.

Chapter 3 of this thesis was published as K. H. A. Ananias, **D. Seliayeu**, J. N. Amaral, and F. M. Q. Pereira, "Multidirectional propagation of sparsity information across tensor slices," in *CGO* 2026. I was responsible for deriving the proofs of convergence, monotonicity, and soundness, as well as for implementing the forward, backward, and lateral propagation of the analysis in C++. Both myself and K. H. A. Ananias worked together to derive the transfer functions for the analysis. I made contributions to the evaluation and experimentation harness, which were primarily developed by K. H. A. Ananias. K. H. A. Ananias was also responsible for deriving the asymptotic complexity of the analysis and its proofs. J. N. Amaral and F. M. Q.

Pereira performed supervisory roles: they contributed greatly to guiding the direction of the experimentation and to editing the manuscript. F. M. Q. Pereira provided the key research idea behind this project.

Chapter 4 of this thesis has been submitted for publication as **D. Seliayeu**, K. H. A. Ananias, J. N. Amaral, and F. M. Q. Pereira, "Bandedness propagation analysis". I was responsible for deriving the transfer functions, their equations, their proofs of soundness and convergence, and implementing the analysis for MLIR. Alongside K. H. A. Ananias, I worked on implementing the lowerings from our bespoke DIA dialect to standard MLIR dialects. K. H. A. Ananias was responsible for implementing the majority of the evaluation and testing infrastructure, and for implementing the DIA dialect in MLIR. J. N. Amaral and F. M. Q. Pereira performed supervisory roles: they contributed greatly to guiding the direction of the experimentation and to editing the manuscript. F. M. Q. Pereira suggested the idea and suggested interesting research questions for us to explore.

Acknowledgements

I want to thank my supervisor Dr. José Nelson Amaral for inviting me to be a part of CDOL; it is hard to overstate how many opportunities for both personal and professional growth were borne out of working together. I appreciate the advice, criticism, and compassion that he provided me throughout my degree, which have consistently helped steer me in the right direction. I still think about many of these insights, and they continue to fuel my personal growth.

I want to thank my co-supervisor, Fernando Magno Quintão Pereira, for his infectious passion for research and for his seemingly endless supply of interesting ideas, which served to kickstart the work in this thesis. I would also like to thank him for taking me along on hikes and for the hospitality he offered to me in Brazil

I want to thank Kaio, who was a great co-author, collaborator, and friend to work with. I am thankful for the time we spent testing prototypes and having discussions over lunch at the bandejão.

I would also like to thank all the friends I made during my studies for the time we spent together, which provided a relaxing break from thesis work. I want to thank Caio, Quinn, and Nathan for the times we spent running and climbing; I want to thank Dhanraj and Patrick for the entertaining discussions about random topics; I want to thank João and Caio (de café) for making my stay in Minas Gerais so enjoyable, owing to the many excursions and trips; And to everyone else who made the past two years so special: thank you.

Table of Contents

List of Tables

List of Figures

Abbreviations

AOT Ahead-of-Time.

BPA Bandedness Propagation Analysis.

Central Processing Unit CPU.

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico.

CSC Compressed sparse column.

CSR Compressed sparse row.

DIA Diagonal.

Graphics Processing Unit GPU.

HPC High-performance computing.

JIT Just-in-Time.

MNC Matrix Non-Zero Count.

Natural-Language Processing NLP.

Partial-Differential-Equation PDE.

Rectified Linear Activation Unit ReLU.

SPA Sparsity Propagation Analysis.

Tensor AlgebraCompiler TACO.

TeSA Tensor-with-Sparsity-Attribute.

UFMG Universidade Federal de Minas Gerais.

Chapter 1

Introduction

Machine learning and High Performance Computing (HPC) are important domains that rely on linear algebra operations on tensors. Among the many approaches to optimizing the linear algebra operations relevant to these domains, one approach is sparsity optimization, which the analyses in this thesis enable. Model training in machine learning may rely on tensors with either structured or unstructured sparsity: the former causes zeros to appear in the output, following some kind of structured pattern, and the latter causes them to appear at random, with both forms of sparsity introducing zeros to the model's final weights [74]. Alternatively, sparsity can be deliberately introduced at some point in the computation [18]. For example, in transformer-style models, attention scores are obtained by deliberately masking some positions from attending to others, a technique called masked attention.

Sparse data formats, such as compressed sparse row (CSR) and compressed sparse column (CSC), store only the non-zero elements, yielding memory savings for higher sparsity levels. Given that zero is an annihilating element for multiplication, knowing the locations of the non-zero elements ahead of time allows multiplications by zeros to be skipped entirely, leading to speedup when the matrix is sufficiently sparse.

Data flow analysis is a type of static analysis used to derive bounds on program values, making it useful for linear algebra. Though traditionally restricted to bounds

on individual values, data flow analyses that reason about the *structure* of tensors can be highly useful in both machine learning and HPC. In these domains, the structured sparsity of certain tensors, like weights or masks, is known in advance, which provides opportunities for such analyses.

The two data flow analyses introduced in this thesis reason about the structured sparsity of tensor slices and matrix bands. Though evaluated specifically with machine learning and HPC, the theory behind the analyses extends beyond only zero to cover general algebraic semirings with an annihilating element. The analyses show that through the derived transfer functions, the analysis enables propagation of sparsity forward, laterally, and backward in a computational graph.

This thesis introduces two data-flow analyses, Sparsity Propagation Analysis (SPA) and Bandedness Propagation Analysis (BPA). Both analyses target a specific kind of structured sparsity and derive equations that are able to propagate it through a computational graph.

This thesis explores the various kinds of structured sparsity and strives to answer questions pertaining to it. The descriptions of the analyses span two chapters.

The contributions in Chapter 3 (SPA) are:

1. The introduction of transfer functions and theory describing the propagation of sparsity through tensor slices. These functions concisely describe the sparsity propagation of slices through einsum operations and addition. The theory includes proofs of monotonicity, one-pass convergence, soundness, and asymptotic complexity.

2. An implementation of the analysis built on top of the Tensor Algebra Compiler (TACO) [37] and an evaluation framework.

3. Experiments investigating the following research questions: (a) how the analysis

compares to executing the computational graph, (b) the effect of the different kinds of propagation on sparsity estimation, and (c) how sparsity propagation influences memory consumption and runtime.

The contributions in Chapter 4 (BPA) are:

1. The introduction of transfer functions and theory describing the propagation of structured sparsity represented by upper and lower bandwidths of a matrix, generalizing this notion to cover propagation through tensors. The transfer functions describe sparsity propagation by describing how various operations affect the upper and lower bandwidths. The proofs include those of soundness and one-pass convergence.

2. An implementation of the analysis and evaluation harness built using the MLIR framework.

3. Experiments investigating the following research questions: (a) how the analysis behaves asymptotically; how significant the (b) memory savings and (c) runtime optimizations the analysis enables are; (d) how the analysis impacts compilation time; and (e) to what extent the forward and backward passes propagate sparsity.

The rest of this thesis is structured as follows: Chapter 2 covers the background necessary for SPA and BPA. Chapters 3 and 4 introduce the two analyses, SPA and BPA respectively, which are the focus of this thesis. Chapter 5 introduces the related work. Finally, 5 concludes the thesis and discusses potential directions for future work.

Chapter 2

Background

This chapter introduces the necessary background for the rest of the thesis. Section 2.1 introduces the notion of a computational graph, covers the heavily referenced einsum operation, and introduces two sparse tensor data structures. Section 2.2 introduces terminology and concepts necessary for understanding the concept of a data-flow analysis. Section 2.3.1 covers the annihilating element in algebraic semirings and related concepts. Section 2.4 defines the 2D-projection as it is used in this thesis. Finally, Section 2.5 defines the notion of a banded matrix to be used in the Bandedness Propagation Analysis (BPA).

2.1 Computational Graphs and Sparse Tensors

A *computational graph* represents a program as a directed graph where nodes denote operations (kernels) and edges capture the flow of data (tensors) between them [9].

Definition 1 (Computational graph) *Formally, we define a* ***computational graph*** *as a graph $G = (V, E)$, where each node $u \in V$ consumes a set of input tensors $I(u)$ and produces a single output tensor $O(u)$. An edge $v \to u \in E$ denotes a data dependency from node v to node u.*

Since the computational graph G is acyclic, an analysis can process nodes in topo-

logical order.

A *tensor* is a multi-dimensional array of numerical values that generalizes vectors and matrices. The Sparsity Propagation Analysis (SPA) portion of this thesis focuses on two types of kernels: element-wise *addition* and the generalized *einsum* operation, which Definition 2, adapted from Nayak et al. [51], formalizes:

Definition 2 (Einsum) *An* `einsum` *expression over n input tensors $T_1, T_2, \ldots, T_n$ is:*

$$\texttt{einsum}(\alpha_1, \alpha_2, \ldots, \alpha_n \to \gamma;\ T_1, T_2, \ldots, T_n) = T_{out}$$

*where each α_i is a sequence of symbolic index labels for the dimensions of T_i, and γ specifies the dimensions of the output tensor T_{out}. Indices that appear in the inputs but not in γ are **reduction dimensions**; indices that appear in γ are **output dimensions**. The operation contracts tensors by multiplying along shared dimensions and summing along the reduction dimensions.*

Example 3 *Figure 2.1 illustrates a computational graph with three 2×2 tensors A, B, C and two* `einsum` *kernels. The graph first computes $D = \texttt{einsum}(ik, kj \to ij; A, B)$ and then $E = \texttt{einsum}(ij, jk \to ik; C, D)$. Matrix multiplication exemplifies this notation: given $A \in \mathbb{R}^{M \times K}$ and $B \in \mathbb{R}^{K \times N}$,*

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

*Here, k is a **reduction dimension** because it vanishes in the output, while i and j are **output dimensions**, defining the shape of C.*

2.1.1 Sparse Representations

A dense tensor stores all of its elements in contiguous memory. A sparse tensor instead records only nonzero values and their coordinates. Classical formats such

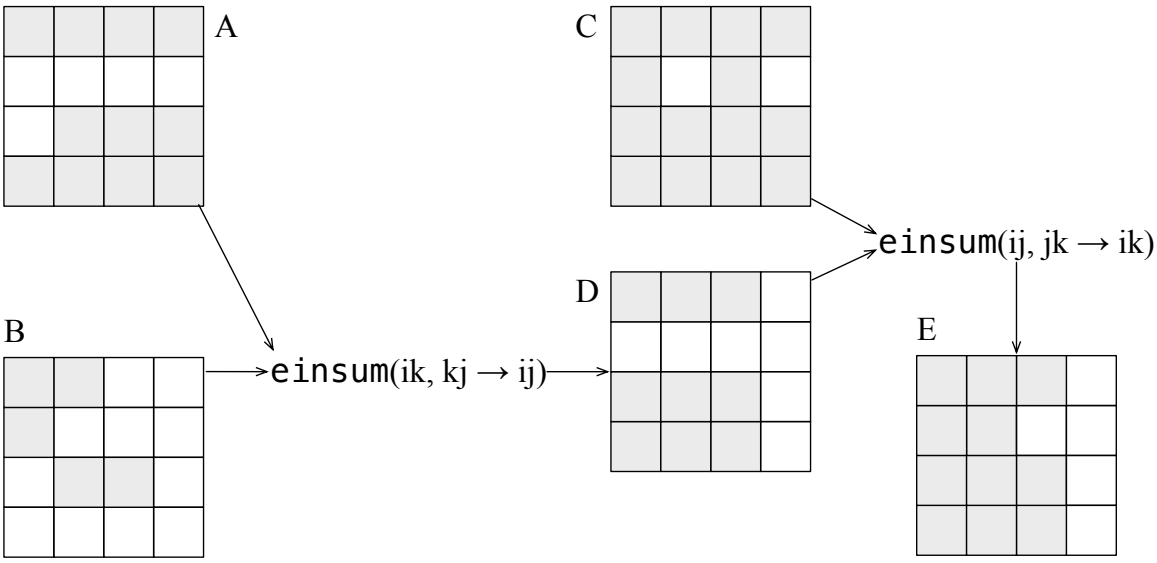


Figure 2.1: A computational graph with 2D-tensors where gray boxes mean non-zero and white boxes mean zero.

as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) handle matrices, while more general data structures extend to higher dimensions.

One such structure is the *fiber tree* [77]. In a fiber tree, each level corresponds to one tensor dimension. A node at depth d represents a **fiber**, i.e., the slice obtained by fixing the first d indices. Edges connect fibers to sub-fibers along the next dimension, and leaves store nonzero values. Analyses such as SPA can improve the utilization of fiber trees, as they enable the elimination of leaves, as Section 3.2.3 will demonstrate.

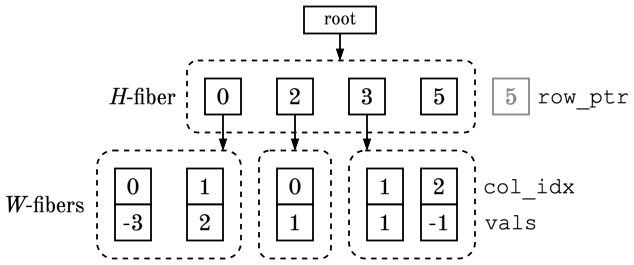


Figure 2.2: The CSR format represented as a fiber tree.

Example 4 *Figure 2.2, adapted from Zhong et al. [77], shows tensor B from Fig-*

ure 2.1 as a fiber tree. The first rank corresponds to rows (H) and the second to columns (W). Here, H is stored as a single uncompressed fiber, while W expands into three fibers holding the nonzero values of B (e.g., $-3, 2, 1, 1, -1$). This hierarchical view naturally generalizes to higher-order tensors.

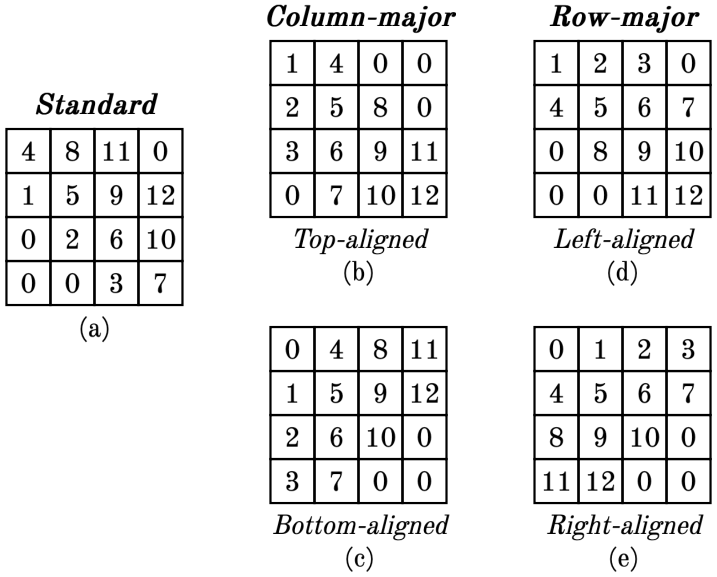


Figure 2.3: An example of four DIA formats.

2.1.2 Diagonal Storage Format

The Diagonal (DIA) storage format [57] stores only the diagonals of a matrix. Figure 2.3 shows various DIA formats representing the original (Standard) matrix. In addition to square matrices, the format also supports non-square banded matrices; each band of an $M \times N$ matrix contains at most size $min(M, N)$ elements. In a wide matrix where $N \gg M$, upper bands will retain their full length. In a tall matrix where $M \gg N$, the same is true of lower bands.

In this format, each diagonal is stored as its own row or its own column, padded to uniform length. Though this means bands occupy the same amount of space, this

still leads to memory savings when there are few non-zero bands. Conversely, a large number of bands can lead to increased storage compared to the dense baseline.

Multiplication of two DIA tensors can result in speedup over standard matrix multiplication because when multiplying two bands b_1 of matrix A and b_2 of matrix B, they output to the diagonal $b_1 + b_2$ of matrix C in the product $A \times B = C$.

2.2 Data-flow Analysis

A static analysis is a program analysis performed without execution of the program. Data-flow analysis is a type of static analysis used to infer the possible states a given value can take on in a program.

Definition 5 (Partial order) *A **partial order** is an ordering of a set where some elements precede others. We say $a \sqsubseteq b$ to mean that a precedes b in the partial order.*

Definition 6 (Join) *In a partial order, for a pair of elements a and b: their **join** or least upper bound ($u = a \vee b$) is defined as 1) an element such that $a \sqsubseteq u$ and $b \sqsubseteq u$; and 2) for any upper bound u' where $a \sqsubseteq u'$ and $b \sqsubseteq u'$ we have that $u \sqsubseteq u'$ (since u is the least such upper bound).*

Definition 7 (Meet) *In a partial order, for a pair of elements a and b: their **meet** or greatest lower bound ($l = a \wedge b$) is defined as 1) an element such that $l \sqsubseteq a$ and $l \sqsubseteq b$; and 2) for any upper bound l' where $l' \sqsubseteq a$ and $l' \sqsubseteq b$ we have that $l' \sqsubseteq l$ (since l is the greatest such lower bound).*

Definition 8 (Lattice) *A **complete lattice** is a partial order where each subset has both a meet and a join. We refer to the join of the entire set as $\top$ and the meet of the entire set as $\bot$.*

Example 9 *If we have a set $S = \{a, b, c, d\}$ where we have $a \sqsubseteq b$, $a \sqsubseteq c$, $b \sqsubseteq d$, and $c \sqsubseteq d$, this is a lattice. Here, $\top = d$ and $\bot = a$.*

Figure 2.4 shows a visual representation of the lattice.

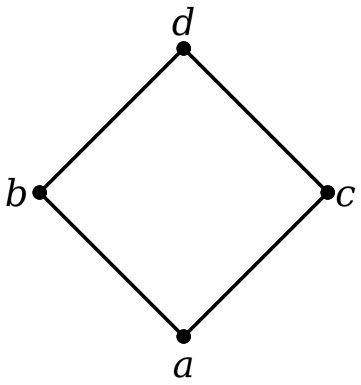


Figure 2.4: This is the Hasse diagram of the lattice in Example 9

Definition 10 (Abstract domain) *A data-flow analysis typically maps nodes in a program representation as a graph to abstract values in the lattice that represent those nodes. In this context, the lattice is the* ***abstract domain****.*

Definition 11 (Transfer functions) *A* ***transfer function*** *indicates how to update an abstract value given some input.*

2.3 Annihilating and Irrelevant Entries

The analyses in this work approximate the set of values that can be *treated as zeros* without changing the final value computed by a computational graph. The analysis proceeds in two passes: a *forward pass*, which interprets the graph's operations abstractly from inputs to outputs, and a *backward pass*, which propagates information in the reverse direction. The forward pass propagates *annihilating elements*, whereas the lateral and backward passes, as defined in Chapters 3 and 4 identify *irrelevant elements*.

2.3.1 Annihilating Elements

A *semiring* is an algebraic structure $(S, \oplus, \otimes, 0, 1)$ consisting of a set S equipped with two binary operations: addition ($\oplus$) and multiplication ($\otimes$), such that $(S, \oplus, 0)$ is a commutative monoid, $(S, \otimes, 1)$ is a monoid, multiplication distributes over addition, 0 is the identity element of $\oplus$ and the *annihilating element* of $\otimes$, e.g.:

Definition 12 (Annihilating Element) *An element $z \in S$ is an* annihilating element *for $\otimes$ if, for all $a \in S$, we have $a \otimes z = z \otimes a = z$*

Example 13 *Many tensor operations have an annihilating element. In matrix multiplication and in the Hadamard product, 0 is an annihilating element, as any multiplication involving 0 yields 0. This notion extends beyond standard floating-point arithmetic to other semirings. For example, in a semiring where $\oplus$ is logical OR and $\otimes$ is logical AND, the value* true *is an annihilating element for $\oplus$, and the value* false *is an annihilating element for $\otimes$, because* $\mathsf{true} \vee x = \mathsf{true}$ *and* $\mathsf{false} \wedge x = \mathsf{false}$*. Similarly, in a semiring where $\otimes$ is defined as the* min *operator over non-negative values, 0 is an annihilating element because* $\min(0, x) = 0$ *for all $x \geq 0$.*

2.3.2 Irrelevant Elements

In many tensor programs, not all tensor elements contribute semantically to a computation. Some elements are guaranteed to be masked by subsequent operations, typically through an operation with an annihilating element. Such entries cannot influence the outcome of the computation. Definition 14 formalizes this notion.

Definition 14 (Irrelevant Entries) *Let e be an expression in a computational graph. A value is irrelevant to e if it may take any concrete value without affecting the observable result of e.*

Example 15 *The* Hadamard product *of two tensors A and B of the same shape, denoted $C = A \odot B$, is defined element-wise as $C_{i,j} = A_{i,j} \cdot B_{i,j}$. If A is upper triangular, then all entries below the main diagonal are zero, i.e., $A[i,j] = 0$ for $i > j$. Consequently, the corresponding entries of C are zero regardless of the values in B. Therefore, the values of B below the diagonal are irrelevant to the computation.*

The two data-flow analyses introduced in this thesis approximate the set of *annihilable tensor elements*. These elements can be treated as zeros (more generally, as annihilating elements) in any evaluation of the graph.

Definition 16 (Annihilable Elements) *An entry of a tensor is* annihilable *if it can be replaced by an annihilating element without affecting the observable result of the computation.*

2.4 Projections

The analysis introduced in Chapter 4 operates on *2D-projections* of tensors. These projections are as defined below.

Definition 17 (2D-Projection) *Let $A \in \mathbb{R}^{d_1 \times \cdots \times d_n}$ be an n-dimensional tensor, and let $i, j \in \{1, \ldots, n\}$ with $i \neq j$. The ij-*projection *A_{ij} is the matrix in $\mathbb{B}^{d_i \times d_j}$ defined as:*

$$A_{ij}(x_i, x_j) = \bigvee_{\substack{x_k \in [0, d_k) \\ k \notin \{i,j\}}} (A(x_1, \ldots, x_n) \neq \star)$$

where $\mathbb{B} = \{false, true\}$ and $\vee$ denotes logical disjunction.

Example 18 *Figure 2.5 shows an example of 2D projection over two particular dimensions of a 3D tensor. 2D projections are defined over tensors of $n \geq 2$ dimensions. An n-dimensional tensor has $n \times (n-1)/2$ projections.*

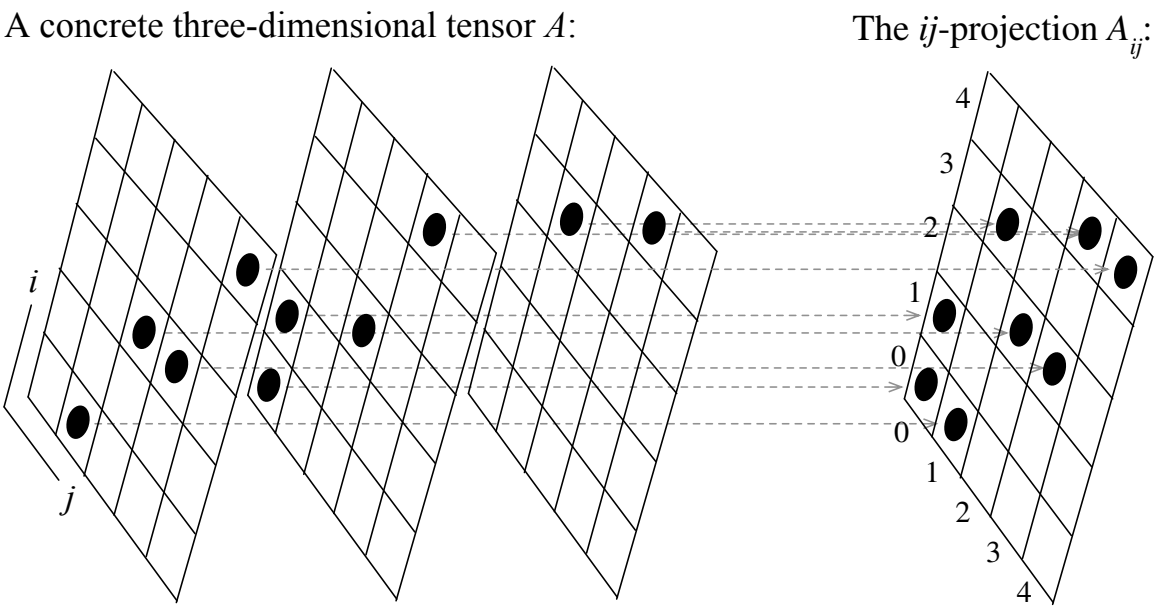


Figure 2.5: The A_{ij} projection of a 3D tensor A.

2.5 Banded Matrices

A *banded matrix* has its nonzero entries confined to a diagonal band around the main diagonal, as per Definition 19. Entries sufficiently far from the diagonal contain zero. Definition 19 modifies this traditional definition to consider annihilable values, as per Definition 16, rather than zeros.

Definition 19 (Banded Matrix) *Let $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, and let $p, q \in \mathbb{N}$. Matrix A is a* (p, q)-banded matrix *if:*

$$a_{ij} = \star \quad \textit{whenever} \quad i - j > p \;\; \textit{or} \;\; j - i > q.$$

The integers p and q from Definition 19 are called the *lower* and *upper extents* of A, respectively. As Example 20 lists, several well-known matrix classes arise as special cases:

Example 20 *Only entries within the band may influence the computation; entries outside being irrelevant; hence:*

- Diagonal matrices*:* $p = q = 0$*.*
- Tridiagonal matrices*:* $p = q = 1$*.*
- Upper triangular matrices*:* $p = 0$*.*
- Lower triangular matrices*:* $q = 0$*.*
- k-diagonal matrices*:* $p + q + 1 = k$*, i.e., exactly* k *diagonals may be nonzero.*

In an $m \times n$ matrix, the maximal possible extents are $m - 1$ and $n - 1$. We denote the maximum extents for any given dimension by $*$.

Chapter 3

Sparsity Propagation Analysis

Tensor compilers [3, 44, 55] translate high-level machine learning programs, often expressed as *computational graphs* [47], into optimized code for central processing units (CPUs), graphics processing units (GPUs), and accelerators. A computational graph is a directed graph in which nodes represent operations (such as matrix multiplication or element-wise addition), and edges represent data dependencies between multi-dimensional arrays known as tensors. With the rapid adoption of machine learning, systems such as XLA [26], TVM [11], ONNX Runtime [48], TensorRT [15], MXNet [49], nGraph [14], XNNC [10], and Glow [56] have become central to deploying models across diverse hardware.

Sparsity as an optimization opportunity. A tensor is sparse when most of its entries are zero. Sparsity arises in many domains: embedding tables in Natural-Language Processing (NLP), Partial-Differential-Equation (PDE) solvers in scientific computing, and activation maps after the Rectified Linear activation Unit (ReLU) in computer vision. Sparse formats store only nonzero entries, enabling (1) elimination of multiplications with zero operands and (2) reduced memory footprint.

There exist previous work, such as SparTA [74] and Mnc [61], that attempt to predict sparsity. SparTA computes the graph with boolean-valued tensors that track zero/nonzero status of each element, incurring element-wise cost. Mnc estimates

lower and upper bounds on the number of zero entries in 2D tensors, but does not generalize efficiently to higher dimensions or arbitrary operations.

Our approach. We propose SPA (Sparsity Propagation Analysis), a static analysis that identifies tensor slices whose elements can be treated as zeros under any evaluation of the graph. This form of *structural sparsity* [68] appears in practice in neural network weights [21] or after transformations such as image clipping [34, 63]. SPA differs substantially from MNC: instead of counting non-zero elements per row or column, SPA associates general tensor slices with zero/non-zero information. As a consequence of this choice of abstract domain, unlikely MNC, which is restricted to matrices and only propagates information forwardly, SPA generalizes to tensors of any number of dimensions, and propagates sparsity information in more directions. Unlike SPARTA, SPA represents sparsity at the slice level, requiring only $\mathcal{O}(m+n)$ bits for an $m \times n$ tensor rather than $\mathcal{O}(m \times n)$. However, as SPARTA does, SPA can also propagate sparsity information in three directions:

- *Forward:* information about inputs A and B constrains the result C in $C = A \times B$;
- *Backward:* information about C refines that of A and B;
- *Lateral:* information about one operand, e.g., A constrains the other, e.g., B.

Together, these propagation modes yield a precise under-approximation of sparsity that is asymptotically more efficient than even a single graph execution.

Contributions. This section of the thesis makes the following contributions:

- **Abstract Domain:** A representation of structural sparsity as dimension-indexed bitmaps proportional to tensor rank, not element count.
- **Transfer Functions:** Equations that propagate sparsity forward, backward, and laterally across the operations that constitute a computational graph.

- **Implementation and Evaluation:** An implementation in the *Tensor Algebra Compiler* (TACO) [37], which can be easily extended to operations fitting into algebraic semirings, such as multiplication and addition.

Experiments in Section 3.2.1 show that the analysis can be up to seven orders of magnitude faster than executing a single iteration of the computational graph. Section 3.2.2 demonstrates that the lateral and backward propagation of sparsity information might yield more than a 20% increase in precision over forward propagation only. Section 3.2.3 shows that SPA enables optimizations that lead to an 80% reduction in memory usage and speedups of more than 50%.

3.1 Static Sparsity Propagation Analysis

This section formalizes the proposed Sparsity Propagation Analysis (SPA). It starts with the abstract domain, then establishes soundness, defines propagation rules and transfer functions, and analyzes convergence and complexity.

3.1.1 Abstract Domain

SPA associates each *tensor slice* (Definition 21) with a boolean value: 0 indicates that all elements of the slice can be treated as zero; 1 indicates that the slice may contain values relevant to the computation.

Definition 21 (Tensor Slice) *Given a tensor T of N dimensions, a slice $T_{:,\ldots,i,\ldots:}$ is the subset of elements sharing the same index i. Thus, a slice has $N-1$ dimensions.*

Example 22 *Figure 3.1 shows four tensors. The sparsity of the single slice of Tensor A is represented by a scalar. Tensor C has three row slices and seven column slices. Tensor E has seven slices, each shown explicitly.*

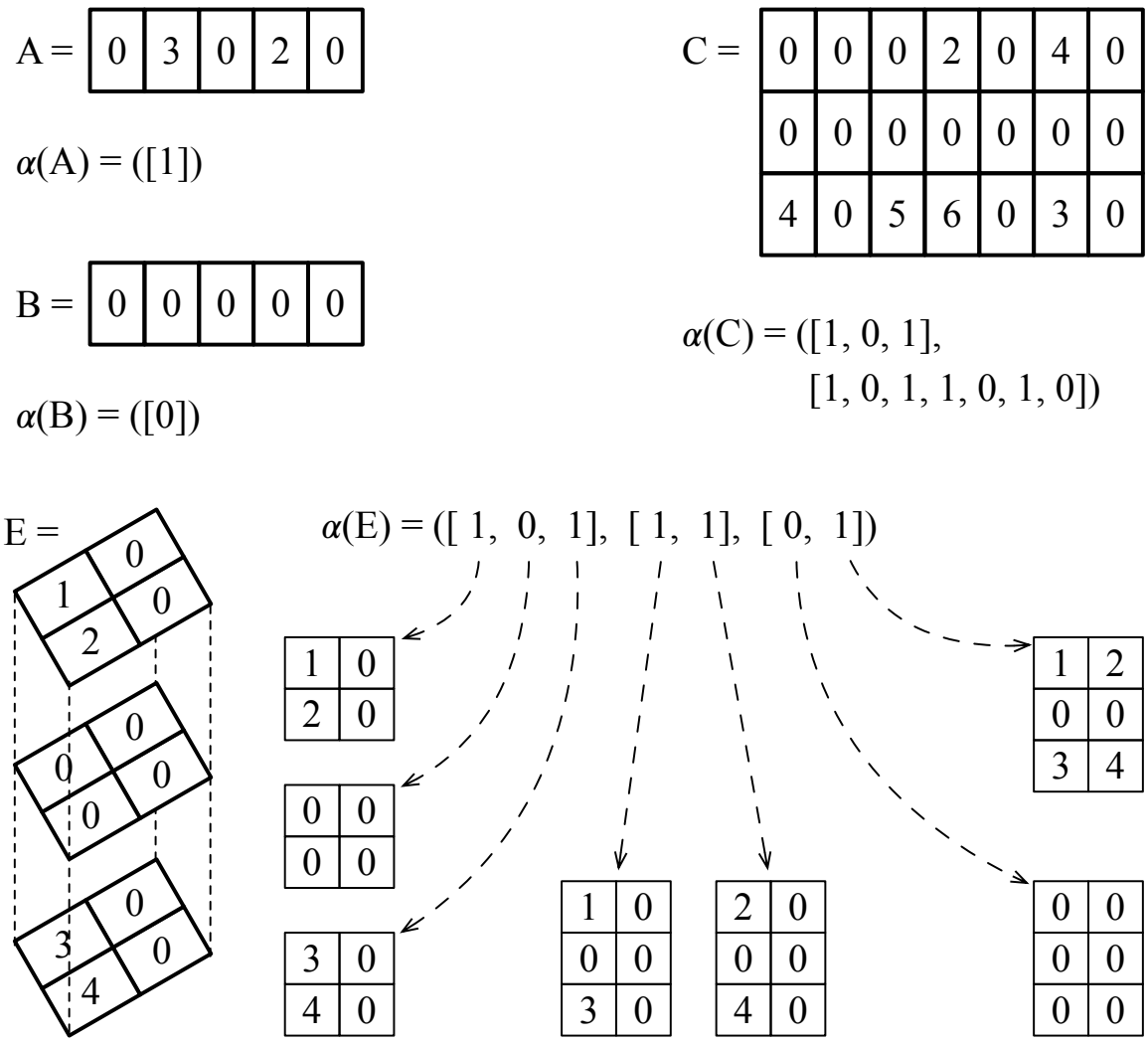


Figure 3.1: Examples of the sparsity abstraction computed by SPA.

Each dimension of a multi-dimensional tensor determines as many slices as there are indices along that dimension. The analysis groups these slices into bitmaps, one per tensor dimension, which we call *sparsity vectors*.

Definition 23 (Sparsity Vectors) *For an N-dimensional tensor T of shape $(M_0, M_1, \ldots, M_{N-1})$ in a computational graph G, SPA computes an abstract state $\alpha(T)$:*

$$\alpha(T) = (S_0, S_1, \ldots, S_{N-1}),$$

where S_i is a bitmap for dimension i. If M_i is the size of dimension i, then $S_i \in$

$\{0,1\}^{M_i}$. *A bit* $S_i[j] = 0$ *means that slice* j *can be treated as zero in any execution of* G.

Example 24 *In Figure 3.1, each tensor is paired with its sparsity vectors. Tensor* A *has* $\alpha(A) = ([1])$*, indicating a potential nonzero element. Tensor* B *has* $\alpha(B) = ([0])$*, meaning it is entirely zero. Tensor* C *has* $\alpha(C) = ([1,0,1],[1,0,1,1,0,1,0])$*, showing which rows and columns may contain nonzeros. For tensor* $E \in \mathbb{R}^{3\times2\times2}$*,* $\alpha(E) = (S_0, S_1, S_2)$*. Figure 3.1 shows that* $S_0[1] = 0$*, meaning that all elements* $E[1,i,j]$ *are zero for any* $0 \leq i < 2$ *and* $0 \leq j < 2$.

Semirings: SPA generalizes over any operations that support *absorbing* (e.g., zero in multiplication) and *identity* (zero in addition) elements. In other words, Theorem 41 holds for any semiring $(\mathcal{D}, \oplus, \otimes, \gamma, \delta)$, where γ is the absorbing element for $\otimes$ and δ is its identity element, and for any computational graph G defined over $\mathcal{D}$. In this section of the thesis, this framework is instantiated with the semiring $(\mathbb{R}, +, \times, 0, 1)$, because addition and multiplication are the fundamental operations in computational graphs. However, the result applies to other semirings, such as $(\mathbb{N}, \max, \min, 0, +\infty)$, $(\mathbb{Z}, \max, \min, -\infty, +\infty)$, or the Boolean semiring $(\{0,1\}, \vee, \wedge, 0, 1)$.

Soundness does not imply that a slice marked zero contains literal zeros at runtime. It only guarantees that any nonzeros in that slice do not affect the graph's output.

Example 25 *Consider* $X = A \times B^T$*, where* A *and* B *are the 1D-tensors in Figure 3.1. SPA determines not only that* $\alpha(B) = ([0])$*, but also that* $\alpha(A) = ([1])$ *because* 0 *is the* ***absorbing element*** *of multiplication: all values in* A *can be treated as zeros.*

3.1.2 Sparsity Propagation

SPA propagates data across operations in three directions relative to the execution flow of the computational graph:

- **Forward:** from inputs to outputs,
- **Backward:** from outputs to inputs,
- **Lateral:** across inputs of the same operator.

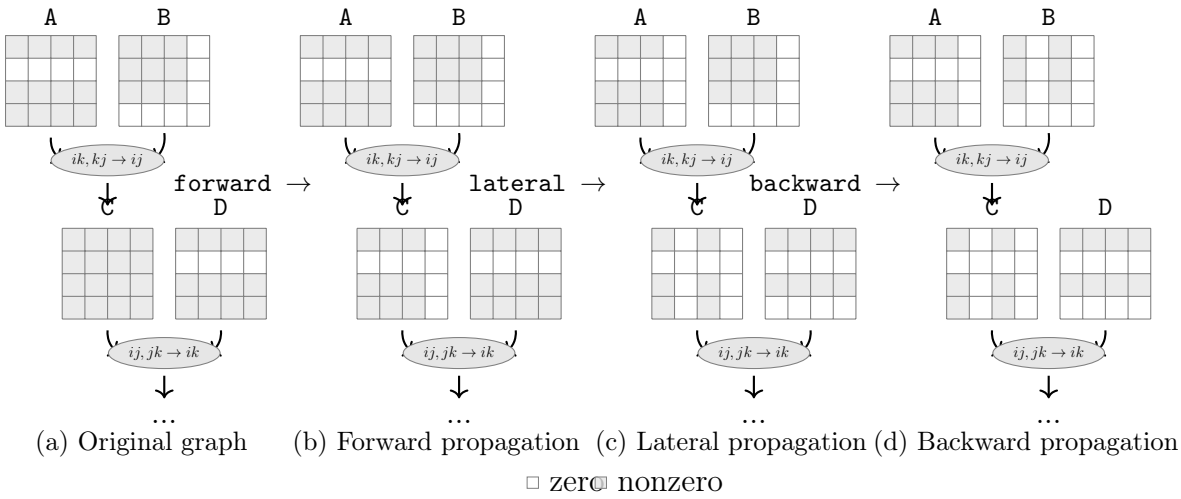
Forward and backward propagation apply to semirings with absorbing and identity elements. Lateral propagation applies only when an absorbing element exists (e.g., multiplication).

Example 26 Forward Propagation: *In Figure 3.2(b),* $C = \texttt{einsum}(ik, kj \to ij; A, B)$. *If row 2 of* A *and column 4 of* B *are zero, then row 2 and column 4 of* C *are also zero by absorption.* Lateral Propagation: *In Figure 3.2(c), if the last row of* B *is zero, then the last column of* A *is irrelevant and can be marked sparse. Similarly, the last column of* C *laterally propagates to the last row of* D. Backward Propagation: *In Figure 3.2(d), column 2 of* C *is zero, so column 2 of* B *can be marked sparse due to backward propagation.*

3.1.3 Transfer Functions

Transfer functions determine how abstract information, represented by the sparsity vectors, propagates across operations. This section formalizes the transfer functions for addition and general einsum operations. However, as discussed in Section 3.1.5, transfer functions can be defined for any combination of operands and types that fit into a semiring. These transfer functions shall use the concepts of *output* and *reduction* dimensions.

Definition 27 (Output and Reduction Dimensions) *For an operation op,* ***output dimensions*** $Od(op)$ *is the set of dimensions that appears in the output;* ***reduction dimensions*** $Rd(op)$ *are the dimensions that appear only in inputs and are contracted.*



(a) Original graph (b) Forward propagation (c) Lateral propagation (d) Backward propagation

Figure 3.2: The three directions along which the sparsity analysis propagates abstract information.

Example 28 *In* $\texttt{einsum}(ik, kj \rightarrow ij)$, $i, j \in Od$, *and* $k \in Rd$. *In element-wise addition, no dimension is reduced, so all dimensions are output dimensions.*

Transfer functions are expressed using bitwise operators $\wedge$ and $\vee$ on sparsity vectors (Definition 29).

Definition 29 (Operations Over Sparsity Vectors) *Given two sparsity vectors of the same size* N, S^X *and* S^Y, $S^Z = S^X \vee S^Y$ *corresponds to a bitwise OR. Specifically,* $S^Z[i]$ *is 1 if either* $S^X[i]$ *or* $S^Y[i]$ *are 1 and 0 otherwise, for* $i \in [0, N)$. *Similarly,* $S^Z = S^X \wedge S^Y$ *corresponds to a bitwise AND and* $S^Z[i]$ *is 1 if both* $S^X[i]$ *and* $S^Y[i]$ *are 1 and 0 otherwise, for* $i \in [0, N)$.

Definition 29 can be generalized for a set of sparsity vectors. The operators $\bigwedge$ and $\bigvee$ denote the application of bitwise AND and OR, respectively, to a set of sparsity vectors of the same size. For example, $S_i^Y = \bigwedge_{X \in \mathbf{X}} S_i^X$ where $\mathbf{X}$ is a set of tensors that share dimension i, implicitly creating the set of sparsity vectors $\{S_i^X | X \in \mathbf{X}\}$.

$S_i^Y[j] = 1$ if $\forall\ X \in \mathbf{X}$ we have $S_i^X[j] = 1$. Similarly, in the case of $S_i^Y = \bigvee_{X \in \mathbf{X}} S_i^X$ we have that $S_i^Y[j] = 1$ if $\exists\ X \in \mathbf{X}$ such that $S_i^X[j] = 1$.

3.1.3.1 Transfer Function for Addition

In an addition such as $T = I_1 + I_2 + \ldots + I_n$, all input tensors share the same shape. The sparsity of inputs determines the sparsity of T.

Forward Propagation For each output dimension i, the forward transfer function computes:

$$S_i^T = \bigvee_{I \in \mathbf{I}} S_i^I \tag{3.1}$$

By Definition 29, the sparsity of dimension i of T at position j is zero, $S_i^T[j] = 0$, if and only if $S_i^I[j] = 0$ for every input tensor $I \in \mathbf{I}$.

Backward Propagation The backward transfer function, for dimension i, from the output tensor O of an addition operation to its input tensors T is:

$$S_i^T = S_i^T \wedge S_i^O \tag{3.2}$$

Equation 3.2 updates the dimension of T that works as an output dimension of addition since it also appears in the output tensor O. If tensors can belong to multiple operators, then sparsity can only be propagated when the semantics of all the operators are preserved. In this case, Equation 3.2 must be adjusted to guarantee semantics-preserving propagation. Equation 3.3 shows the backward transfer function for output tensors that may be used as inputs to multiple operators.

$$S_i^T = S_i^T \wedge \left(\underbrace{\left(\bigvee_{\mathbf{J} \in \mathcal{I}} \bigwedge_{J \in \mathbf{J}} S_i^J \right)}_{\text{C1}} \vee \underbrace{\left(\bigvee_{O \in \mathbf{O}} S_i^O \right)}_{\text{C2}} \right) \tag{3.3}$$

Equation 3.3 augments Equation 3.2 with two components:

- **C1:** This component ranges over a set $\mathcal{I}$, where each element is a set of input tensors. For a given kernel that uses T with i as a reduction dimension, $\mathbf{J} \in \mathcal{I}$ denotes the set of all its inputs. The role of this component is to prevent reduction dimensions in different kernels that also use the tensor T from being marked sparse in a way that would incorrectly make outputs zero. Correctness requires that, for each operation, at least one input tensor has a zero in the corresponding reduction dimension (captured by the inner $\wedge$), and that this condition holds across all relevant operations (captured by the outer $\vee$).

- **C2:** This component ranges over $\mathbf{O}$, the set of every downstream operation that uses dimension i of T. It ensures that output dimensions cannot be marked sparse because doing so could incorrectly introduce sparsity into the outputs produced by those operators.

Example 30 *Figure 3.3 illustrates backward sparsity propagation in an addition. Assume the analysis determines that tensor C is sparse in dimension 0, with slices $S_0^C[1]$ and $S_0^C[2]$ marked as zero. Because C is computed as $A+B$, the analysis can propagate this sparsity information backward to both inputs.*

Propagation to B: *Tensor B does not participate in any einsum operations, so it has no reduction dimensions and $\mathcal{I} = \emptyset$. The only operator that consumes B is the addition producing C, hence $\mathbf{O} = \{C\}$. Applying Equation 3.2 (i.e., Equation 3.3 without component C1), we obtain $S_0^B = S_0^B \wedge S_0^C$. Therefore, the sparsity in dimension 0 of C propagates directly to B.*

Propagation to A: *Tensor A also contributes to C, but it additionally participates in an einsum that produces $T = A^\top$. In this case, dimension 0 of A becomes dimension 1 of T, which is not a reduction dimension, so $\mathcal{I} = \emptyset$. Since dimension 0 of A is used as an output dimension in both the addition (C) and the einsum (T), we have $\mathbf{O} = \{C, T\}$. Suppose slice 2 of dimension 1 in T is entirely zero. The propagation*

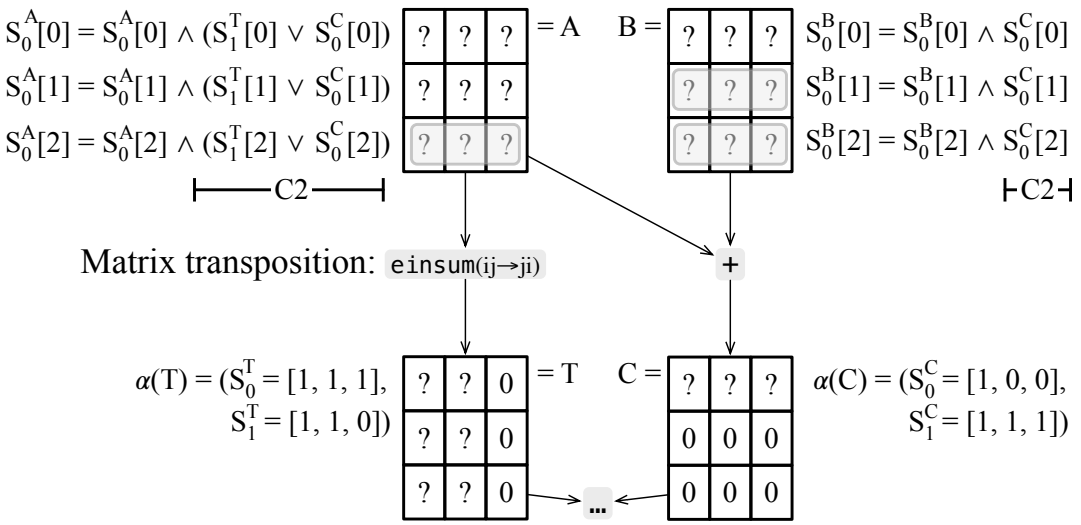


Figure 3.3: The effects of backward propagation through addition and einsum.

then yields:

$$S_0^A = S_0^A \wedge (S_1^T \vee S_0^C)$$
$$= [1, 1, 1] \wedge ([1, 1, 0] \vee [1, 0, 0])$$
$$= [1, 1, 0].$$

Thus, only the last slice of S_0^A is marked sparse, preserving semantic correctness.

3.1.3.2 Transfer Functions for Einsum

The general form of an `einsum` expression, $\texttt{einsum}(\alpha_1, \alpha_2, \ldots, \alpha_n \to \gamma;\ T_1, T_2, \ldots, T_n) = T_{\text{out}}$, is given in Definition 2. In this notation, labels that appear in γ are *output dimensions*, while labels present in some α_i but absent from γ are *reduction dimensions*, as specified in Definition 27.

Forward Propagation For each output dimension i, the forward transfer function updates the abstract state of the output tensor by intersecting the states of all input tensors that contribute dimension i:

$$S_i^T = \bigwedge_{I \in \mathbf{I}} S_i^I \tag{3.4}$$

Here, $\mathbf{I} = \{I_1, \ldots, I_n\}$ is the set of input tensors.

Lateral Propagation Lateral transfer applies only to *reduction dimensions*, and allows sparsity in one input to imply sparsity in another. For the reduction dimensions i, the simplified backward transfer function computes:

$$S_i^T = S_i^T \wedge S_i^I$$

for input tensor T where I is another output tensor with the same reduction dimensions. This transfer function updates the output dimensions of tensors T with those of tensors I.

Combining the simple equation above with the correctness-ensuring expressions ($\left(\bigvee_{O \in \mathbf{O}} S_i^O\right)$ and $\left(\bigvee_{\mathbf{J} \in \mathcal{I}} \bigwedge_{J \in \mathbf{J}} S_i^J\right)$) and simplifying gives us the full equation for lateral propagation:

$$S_i^T = S_i^T \wedge \left(\left(\bigvee_{\mathbf{J} \in \mathcal{I}} \bigwedge_{J \in \mathbf{J}} S_i^J \right) \vee \left(\bigvee_{O \in \mathbf{O}} S_i^O \right) \right) \tag{3.5}$$

Where:

- $\mathcal{I}$ is the collection of sets of tensors that share i as a reduction dimension in a single einsum.
- $\mathbf{O}$ is the set of outputs of einsums where i appears as a reduction dimension.

Backward Propagation The simplified equation for the backwards transfer function is $S_i^T = S_i^T \wedge S_i^O$, which is the same as that for the add expression. After combining the above equation with the two correctness-ensuring expressions (components C1 and C2 in Equation 3.3), we get the same equation as Equation 3.5, except that i is an output dimension. Thus, for each output dimension i in an input tensor

T, if i also appears in the output of an einsum, then backward transfer propagates sparsity from the output back to the inputs. This expression mirrors lateral transfer but is applied to output dimensions instead of reduction dimensions. Example 31 will clarify this difference.

Example 31 *Consider applying Equation 3.5 to tensor C in Figure 3.2. The application is in backward order, starting with the einsum that takes C and D as inputs. The reduction dimensions are dimension 1 of C and dimension 0 of D, both denoted by index j.*

For S^C the equation becomes:

$$S_1^C = S_1^C \wedge (S_1^C \wedge S_0^D)$$
$$= [1,1,1,0] \wedge ([1,1,1,0] \wedge [1,0,1,1])$$
$$= [1,0,1,0].$$

Only j is updated because it is the sole reduction dimension. $\mathcal{I}$ contains the single set $\mathbf{J} = \{C, D\}$, yielding the term $(S_1^C \wedge S_0^D)$, because C is input to a single operation. Dimension 1 of C does not appear as an output dimension in any operator, so $\mathbf{O} = \emptyset$. The same reasoning applies to D, A, and B, producing the sparsity annotations shown in Figure 3.2.

The transfer functions in this section are monotonic. They operate over a boolean lattice with $0 > 1$, so once the analysis marks a slice as zero, no transfer function can later mark it as nonzero. Theorem 32 formalizes this property, followed by its proof.

Theorem 32 *[Monotonicity] The transfer functions discussed in Sections 3.1.3.1 and 3.1.3.2 are monotonic.*

Proof. Consider the update of dimension i of tensor T as $i^{T'} = i^T \wedge x$, where x is a sparsity vector of the same domain. By definition of $\wedge$, the result is zero wherever

either operand is zero, and one only where both operands are one. Thus $i^{T'} \leq i^T$, preserving monotonicity. Given that all transfer functions can be expressed in this form, the entire analysis is monotonic. Combined with the finite height of the lattice, convergence is guaranteed. ■

3.1.4 Propagation Order

The computational graph is *acyclic*, which means its nodes admit a topological ordering that the analysis exploits for fast convergence. Let $[u_0, u_1, \ldots, u_{n-1}]$ be a topologically sorted sequence of nodes in a computational graph G. Then:

1. **Forward Propagation:** A single pass in the order $u_0 \to \ldots \to u_{n-1}$ reaches a fixed point.

2. **Lateral Propagation:** A single pass in the reverse order $u_{n-1} \to \ldots \to u_0$ reaches a fixed point.

3. **Backward Propagation:** A single pass in the same reverse order also reaches a fixed point.

These results follow intuitively from the absence of cycles: information always flows along edges consistent with data dependencies. Lemmas 33, 34, and 35 establish them formally. More importantly, when the analysis applies these phases in the specific sequence *forward, then lateral, then backward*, propagation converges to the global fixed point after just one iteration. Theorem 36 formalizes this result.

Lemma 33 *A second pass of forward propagation does not change the analysis result.*

Proof. Forward propagation applied in topological order processes each node exactly once. Because the graph is acyclic, repeating the pass cannot introduce new information: every output has already incorporated all inputs. Even if followed by

lateral or backward propagation, a subsequent forward pass is redundant, as all zero information would have already propagated through the equations in Section 3.1.3.

For the sake of contradiction, suppose that applying forward propagation to the graph in this state leads to changes in the sparsity of the outputs. This must mean that Equations 3.2, 3.3, or 3.5 update the sparsity vector of an input tensor T at some dimension d such that S_d^T could propagate and increase sparsity of the output S_d^O. However, this is impossible: suppose S_d^O was 1 at some indices $i \in [0, n)$. Then, Equations 3.2, 3.3, and 3.5 would evaluate to $S_d^T[i] = S_d^T[i] \wedge 1 = S_d^T[i]$, meaning that they cannot change sparsity of S_d^T at indices i . If $S_d^T[i]$ was already 0, then the forward analysis would have already propagated sparsity to S_d^O by assumption, leading to a contradiction. ■

Lemma 34 *A second pass of lateral propagation after forward propagation does not change the result.*

Proof. Suppose a slice $S_i^T[j]$ changed from one to zero after a second application of lateral propagation. This would require that the bounding expression $\left(\bigvee_{\mathbf{J}\in\mathcal{I}} \bigwedge_{J\in\mathbf{J}} S_i^J\right) \vee \left(\bigvee_{O\in\mathbf{O}} S_i^O\right)$ had decreased between applications. However, this expression does not change across iterations, so the assumption leads to a contradiction. ■

Lemma 35 *A second pass of backward propagation after lateral propagation does not change the result.*

Proof. The argument mirrors that of Lemma 34: the bounding expression in backward propagation does not change. ■

Theorem 36 (One-Pass Convergence) *For any acyclic computational graph G, the sparsity analysis reaches its least fixed point after a single global iteration consisting of forward, lateral, and backward propagation (in that order).*

Proof. Follows from Lemmas 33, 34, and 35. ■

Theorem 36 relies on a special treatment of tensors that feed into multiple operations. For such tensors, lateral and backward propagation apply only when the effect can be inferred from *all* of their outputs (e.g., case C2 in Equations 3.3 and 3.5). This restriction follows directly from the $\vee$ operator in those equations. Example 37 illustrates the idea.

Example 37 *Figure 3.4 shows the combined effects of forward, lateral, and backward propagation on a subgraph with seven tensors, four of which are inputs (A, B, D, and X). Although the analysis applies to tensors of arbitrary rank, this example uses 2D tensors (matrices) for clarity. Initially, only the last row of D is known to be zero. The scenario raises two reasoning exercises:*

1. *Lateral propagation marks the last column of C as zero. Does this change enable additional forward propagation (e.g., on tensor E)?*

2. *Backward propagation marks the last column of A as zero. Does this update trigger further forward or lateral propagation?*

The answer to (1) is no: columns of C, unlike its rows, do not contribute to the forward computation of E. The answer to (2) is also no: lateral propagation does not apply to addition, and forward propagation could eliminate the last column of C, but that effect has already taken place. Finally, backward propagation does not modify B because doing so would require information from Y, which contains no known zeros.

3.1.5 Soundness

An abstract state is *sound* if a slice marked as zero contributes nothing to any execution of the computational graph. Theorem 41 formalizes this property.

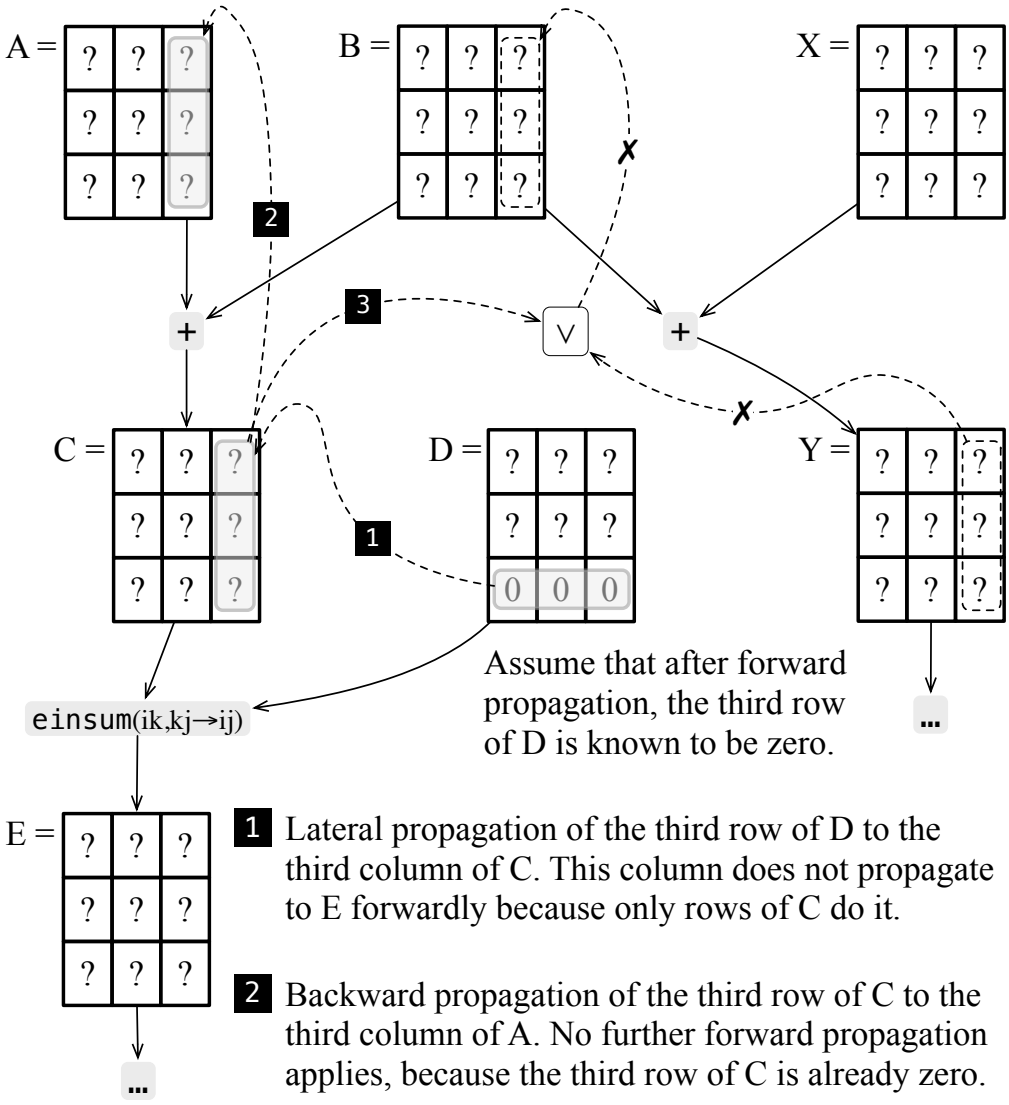


Figure 3.4: The effects of forward, lateral, and backward propagation, assuming that only row 2 of input tensor D was initially known to be zero. Solid lines indicate data movement to and from operations, and dashed lines indicate sparsity propagation.

Soundness requires that slices marked as zero in the abstract domain never affect the outcome of the concrete computation. Forward propagation provides a sound under-approximation of zero elements (Lemma 38), while lateral and backward propagation may mark slices as zero even if their concrete values are nonzero. However, in these cases, the marked values are always multiplied by zero or otherwise nullified, preserving semantics (Lemmas 39 and 40).

Lemma 38 (Forward Propagation Soundness) *Let k be a kernel with inputs $T_1, \ldots, T_m$ and output T_o. If $\alpha(T_1), \ldots, \alpha(T_m)$ are sound, then forward propagation produces a sound $\alpha(T_o)$.*

Proof. The lemma states that for every dimension i and index j of T_o, if the analysis concludes $S_i[j] = 0$ in $\alpha(T_o)$, then the corresponding slice in T_o is entirely zero. The proof is by induction on the structure of the computational graph. We show that for every kernel in the graph, if the abstract states of the input tensors are sound, the abstract state of the output tensor computed by the forward transfer function is also sound. The proof is broken down by kernel type.

Case 1: The Addition Operation. Forward propagation only propagates zeros if all the dependencies of the output tensor are zeros (due to the bitwise OR of slice states). This is sound, as zero is the identity of addition.

Case 2: The General `einsum` Operation. Forward propagation only propagates zeros if one operand of the slide is zero (bitwise AND), which is sound, as zero is the nullifying element of multiplication. ■

Lemma 39 (Lateral Propagation Soundness) *If lateral propagation assigns zero to a slice in $\alpha(T)$, then that slice does not contribute to the result of kernels where T participates.*

Proof. Equation 3.5 zeroes S_i^T only if every use of that slice involves multiplication with another slice already marked zero in the corresponding dimension of i. ■

Lemma 40 (Backward Propagation Soundness) *If backward propagation assigns zero to a slice in $\alpha(T)$, then that slice always produces zero-valued slices in downstream outputs.*

Proof. The reasoning parallels Lemma 39: a slice inferred as zero can only generate outputs already proven to be zero. ■

Theorem 41 (Soundness of SPA) *Let G be a computational graph and T a tensor in G with abstract state $\alpha(T) = (S_0, \ldots, S_{N-1})$. If $S_i[j] = 0$, then every element in slice $S_i[j]$ does not affect the result of evaluating G.*

Proof. Follows from Lemmas 38, 39, and 40. ■

3.1.6 Asymptotic Complexity

Theorems 42 and 44 state the asymptotic complexity of SPA . Complexity is measured in terms of the number of bitwise operations performed by the analysis. These theorems refer to a computational graph as defined in Definition 1.

Theorem 42 (Complexity of Forward Propagation) *For a node u, forward sparsity propagation to its output tensor $O(u)$ requires $\mathcal{O}(K \times |I(u)| \times O(u)_{rank})$ bitwise operations, where* rank *denotes the number of dimensions of the tensor.*

Proof. Each dimension of $O(u)$ must be updated by combining the corresponding dimensions of all input tensors. This requires $|I(u)|$ sparsity vector operations per dimension, and $O(u)_{rank}$ dimensions in total. Finally, each sparsity vector is at most size K. ■

Definition 43 (Graph Maxima) *Let K be the size of the biggest dimension of any tensor in $I(u)$. Let $I_{max} = \max_{v \in V} |I(v)|$ be the maximum number of input tensors to any node. For a tensor t, let $C(t) \subseteq V$ be the set of consumer nodes of t, i.e., all $v \in V$ such that $t \in I(v)$. Define $C_{max}(u) = \max_{t \in I(u)} |C(t)|$ as the maximum number of consumers of any input tensor to u. Finally, extend Definition 27 to define $Rd_{max}(u) = \max_{t \in I(u)} |Rd(t)|$ and $Od_{max}(u) = \max_{t \in I(u)} |Od(t)|$, the maximum numbers of reduction and output dimensions, respectively, among the inputs to u.*

Theorem 44 (Complexity of Lateral Propagation) *In the worst case, lateral sparsity propagation requires $\mathcal{O}(K \times |I(u)| \times Rd_{max}(u) \times C_{max}(u) \times I_{max})$ logical operations.*

Proof. For each input $t \in I(u)$ and each reduction dimension $d \in Rd(t)$, propagation must iterate over all consumers $v \in C(t)$, performing $|I(v)|$ sparsity vector operations per consumer. By definition, $|Rd(t)| \leq Rd_{\max}(u)$, $|C(t)| \leq C_{\max}(u)$, and $|I(v)| \leq I_{\max}$. Each sparsity vector is at most K in size. Multiplying these factors yields the claimed bound. ■

Backward propagation for `einsum` nodes has the same asymptotic complexity as lateral propagation, except that the bound depends on $Od_{max}(u)$ rather than $Rd_{max}(u)$ because backward propagation applies to output dimensions. We therefore omit it as a separate theorem.

3.2 Evaluating the Impact of Sparsity Propagation

This section evaluates SPA along three research questions:

- **RQ1:** How does the analysis time compare to the cost of executing the computational graph?

- **RQ2:** What is the effect of forward, backward, and lateral propagation on sparsity estimation?
- **RQ3:** How does sparsity propagation influence memory consumption and runtime?

Evaluation Setup. SPA could be integrated into different compilers that support sparse formats. Our prototype implementation extends the *Tensor Algebra Compiler* (TACO) [37], which provides the fiber trees that Section 2.1.1 mentions. In this representation, each dimension can independently use a sparse or dense layout, enabling any slice identified as zero by SPA to be stored sparsely. Some limitations of TACO affect this evaluation: the compiler does not generate correct code for certain sparse formats [45]. These issues are not related to the implementation of SPA. Nevertheless, all results reported in this thesis have been validated to ensure correctness.

Hardware: An AMD Ryzen 7 3700X (8 cores, 16 threads) with 32GB of RAM under Ubuntu 22.04 LTS.

Benchmarks. This study evaluates two benchmark suites. The first is an adaptation of the BERT model [16], focusing on the kernels affected by sparsity propagation (addition and matrix multiplication). To obtain measurable runtimes, all tensors were instantiated as 2048×2048 matrices.

The second suite consists of the twelve einsum expressions seen in Table 3.1. These graphs come from the *Einsum Benchmark Collection* [7]. This collection allows us to evaluate runtime, analysis time, and sparsity propagation on more complex computational graphs. These expressions involve tensors with up to thirty dimensions and are compiled in TACO using contraction trees, where each internal node reduces to a binary einsum operation. Contraction paths are generated with `opt_einsum` [59] under the `auto` heuristic. Because the original collection does not provide computational graphs, each einsum expression was transformed into a binary tree–shaped

graph, where every node corresponds to an einsum with exactly two input tensors and a set of contraction dimensions. Such trees, commonly known as contraction trees, are constructed according to a chosen contraction path, which defines the order and dimensions over which tensors should contract [32].

Table 3.1: The list of einsum benchmarks used in the experiments. Rows show number of tensors, nodes, maximum rank (number of dimensions), and size of the dimension of the tensor with the largest dimension.

ID	Benchmark Name	Tensors	Nodes	Max. Rank	Max. Dim.
1	*lm_batch_likelihood_brackets_3_16d*	75	37	6	1196
2	*lm_batch_likelihood_sentence_3_12d*	75	37	5	1100
3	*str_nw_mera_open_26*	51	25	9	79
4	*lm_batch_likelihood_sentence_4_8d*	167	83	7	1900
5	*str_nw_ftps_open_30*	59	29	30	14
6	*str_matrix_chain_multiplication_100*	199	99	2	511
7	*str_nw_ftps_open_28*	55	27	28	16
8	*lm_batch_likelihood_sentence_4_4d*	167	83	9	1900
9	*str_mps_varying_inner_product_200*	399	199	3	127
10	*str_nw_mera_closed_120*	239	119	8	81
11	*gm_queen5_5_3.wcsp*	319	159	18	30
12	*str_matrix_chain_multiplication_1000*	1999	999	2	128

Data Initialization: We initialize tensors with random floating-point numbers. For a given tensor, if a slice has any zeros, we set it to sparse. Otherwise, the dimension remains dense. Then we load the tensors and execute the computation. We control initial sparsity by randomly choosing some amount of slices in a dimension (e.g. 30% of the rows in a matrix) and setting them to zero.

3.2.1 RQ1 – Relative Runtime of the Analysis

Tensor compilers often rely on profiling runs to infer sparsity before specializing code [1, 66]. In contrast, SPA performs sparsity propagation statically, prior to execution. Its overhead is incurred only once: either at compile time, when tensors with known zeros are available (e.g., network weights), or right before execution in a just-in-time setting, once the inputs are known. This section evaluates that overhead.

Discussion: The results in Figure 3.5 indicate that SPA is up to seven orders of magnitude faster than both compilation and execution of computational graphs. This efficiency stems from two factors: (i) the compact abstract domain based on bit vectors and (ii) guaranteed one-pass convergence of propagation (Theorem 36). The analysis reduces each propagation step to a handful of bitwise conjunctions and disjunctions, regardless of tensor size. For instance, in benchmark 6, the computational graph performs a series of matrix multiplications such as $C = A \times B$. If dimensions are $|A| = m \times n$ and $|B| = n \times k$, then the number of operations that a sparse kernel performs in the worst case scales cubically ($m \times n \times k$). On the other hand, as proven by Theorems 42 and 44, SPA scales linearly, e.g., $O(m + n + k)$, for a given graph.

Empirical Asymptotic Analysis: Figure 3.6 compares the analysis time of SPA against SparTA. We use an optimized C++ version of SparTA's Tensor-with-Sparsity-Attribute (TeSA) abstraction[1]. As Section 3.1.6 indicates, the growth of SPA's runtime is very slow relative to SparTA. SparTA's runtime grows from $0.007s$ to $0.85s$ as tensor size increases from 256 to 2048, while SPA's runtime remains close to $29\mu s$. SPA's linear complexity means that it has negligible overhead even for large tensors, making it suitable for JIT compilers.

[1]This code is a C++ implementation optimized with bitsets, which was adapted from the original Python code at https://github.com/microsoft/nni/tree/sparta_artifact to allow for a fair comparison.

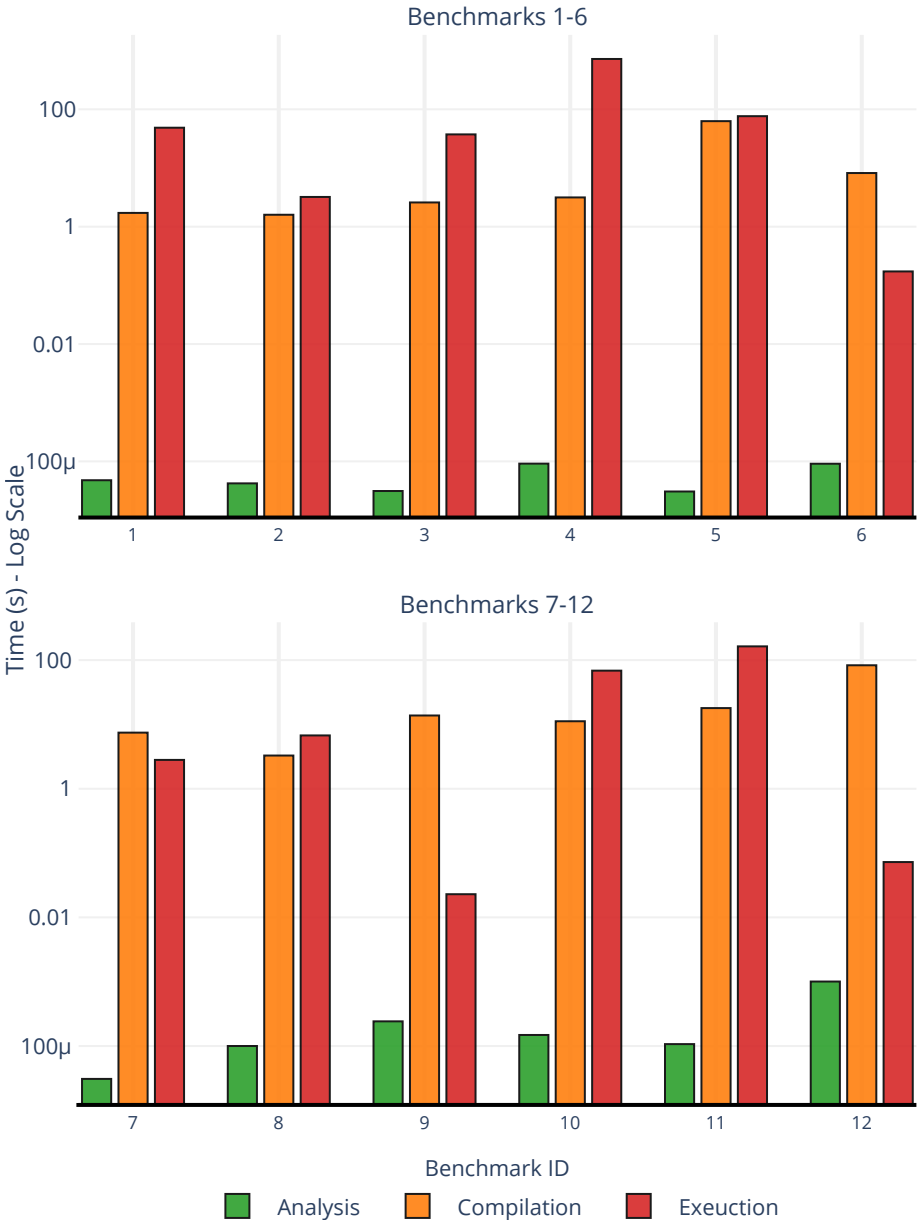


Figure 3.5: The runtime of SPA compared to the time to compile or run computational graphs (Log-scale). Each point on the X axis is a computational graph from Table 3.1. The lower the bar, the better.

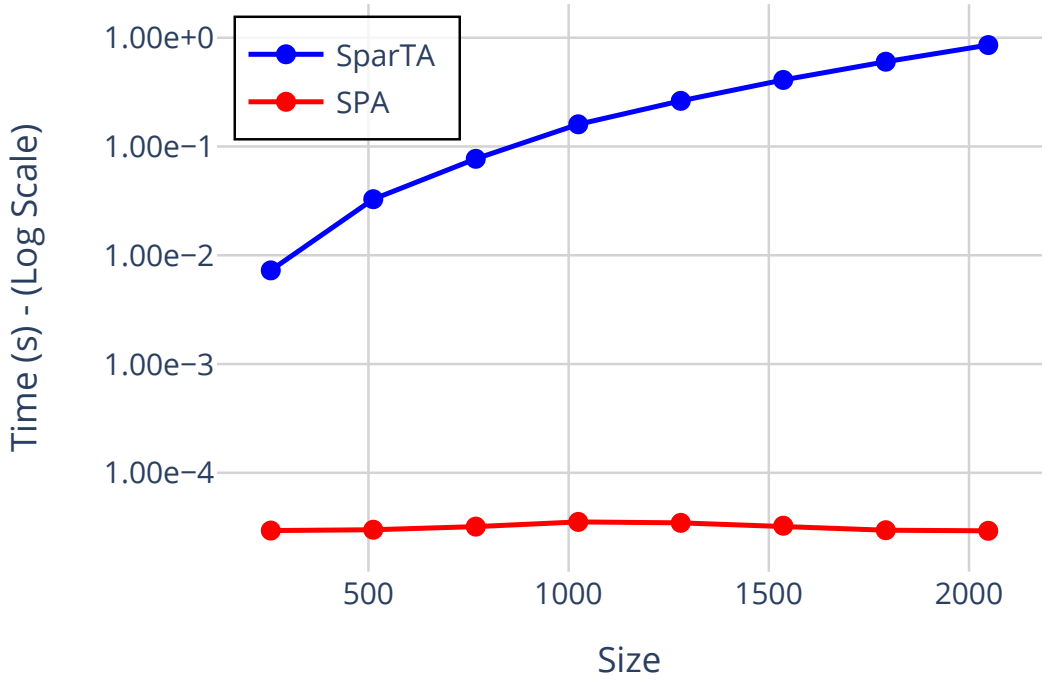


Figure 3.6: Execution time of SPA and of SparTA's TeSA abstraction on one matrix multiplication of two square matrices of the listed size.

3.2.2 RQ2 – Sparsity Propagation

This study ablates the contributions of forward, lateral, and backward propagation. For each einsum expression, one tensor is randomly assigned slice sparsity ranging from 30% to 90%. The results are similar for different amounts of sparsity; thus, this study focused on sparsity equal to 50%. Forward propagation is always included, as it is necessary for enabling lateral and backward propagation.

Discussion: The baseline ("No Prop") in the graph of Figure 3.7 does not propagate information; hence, reflecting only sparsity in the input tensors. Individually, forward propagation has the strongest effect because it carries sparsity along the reduction dimensions of downstream operands. Lateral propagation builds on this by exploiting structural sparsity between operands, often yielding additional gains. Backward propagation generally has a limited effect after forward propagation because most zeros already originate in the inputs.

Most benchmarks benefit from the different propagation modes, but there are exceptions. Benchmark 11 (Table 3.1), for instance, contains only six (out of 159) operators with reduction dimensions. This lack of reduction dimensions limits the potential of lateral propagation. Benchmark 9, whose contraction tree degenerates into a linear chain, also shows minimal benefit from propagation in backward or lateral directions due to rare common output–input dimension sharing.

Combinations that include all three directions (FLB) consistently discover the highest sparsity ratios. For example, in benchmark 4, the sparsity ratio grows from 27% to 84% after propagation. Notice that a profiling execution alone would only expose the sparsity captured by forward propagation (red bars in Figure 3.7); the additional sparsity discovered by lateral and backward propagation (green bars) is unique to SPA.

Due to the way tensors in our benchmarks are initialized, we did not observe

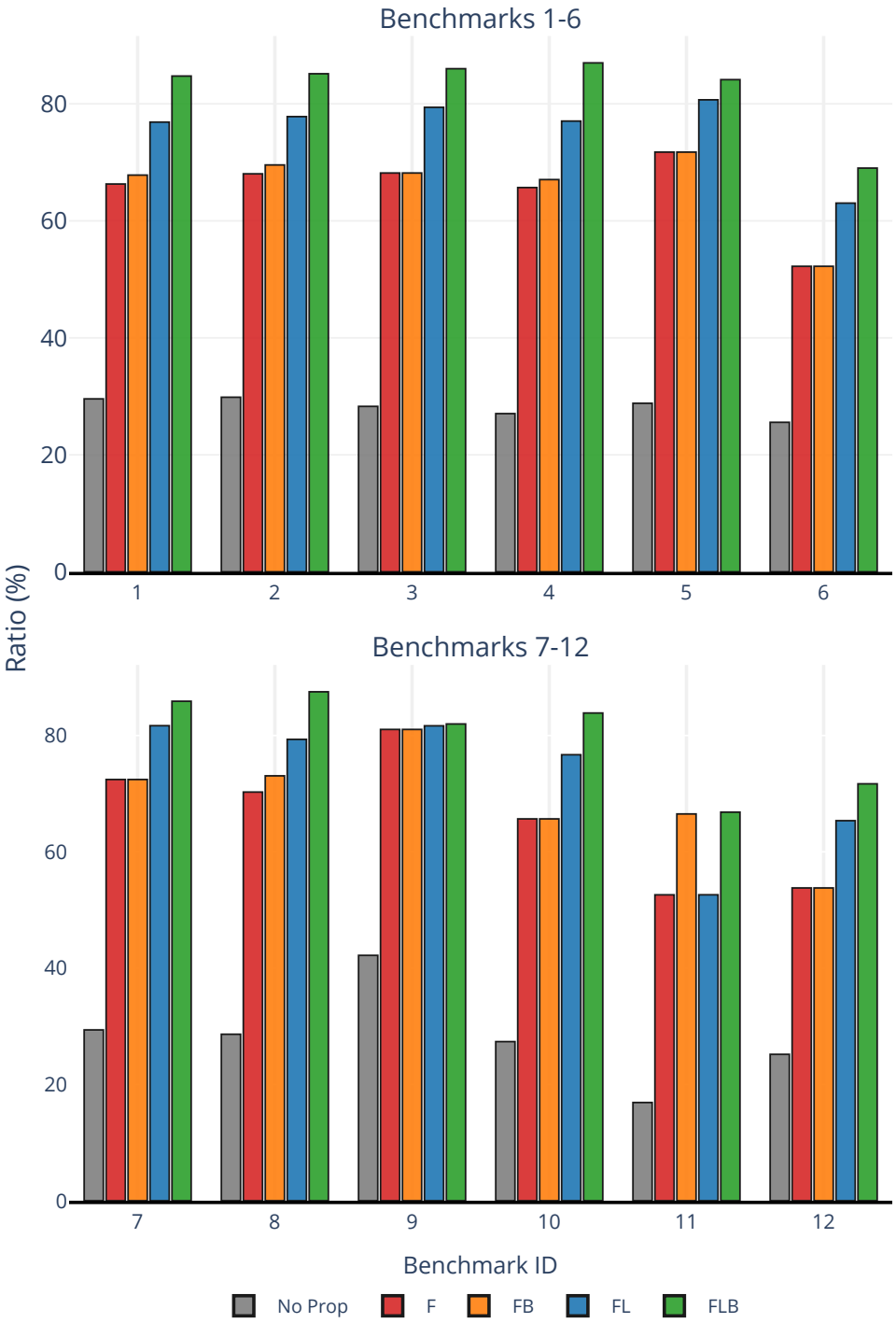


Figure 3.7: Effect of forward (F), lateral (L), and backward (B) propagation on sparsity ratios of einsum benchmarks with 50% input sparsity. The y-axis is average sparsity across all tensors in the benchmark. The higher the bars, the better, as more sparsity is discovered in the computational graph.

cases where SparTA achieved higher precision than SPA. In practice, such cases are expected to be rare. For unstructured sparsity to propagate forward in a 2D matrix multiplication $C = A \times B$, all non-zero elements in a row of A must multiply with zeros in a column of B: an unlikely event when sparsity is around 50%. With higher sparsity levels and a normal distribution of zeros, this alignment becomes more probable; however, higher sparsity also increases the likelihood that entire slices (rows or columns) become zero. In this case, the slice-based abstraction captures sparsity effectively, narrowing the precision gap between SPA and SparTA.

3.2.3 RQ3 – Performance Improvement

Propagated sparsity reduces both memory footprint and runtime. Sparse formats become more compact than dense ones when sparsity is high, while eliminating operations on zero values lowers computational cost. Fiber-tree formats amplify these benefits because propagation can prune entire fibers from the storage structure (see Example 4). For example, increasing the number of zeros in the tensors seen in Figure 4 removes some leaves from the *vals* fiber, as compressed formats only store nonzero elements. This section analyzes the benefits of SPA on the execution of different computational graphs, looking at the BERT-like computational graph and benchmark 12 (the biggest of the Einsum Benchmarks) for runtime, and the entire Einsum Benchmark suite for memory reduction.

Discussion – Running Time: The BERT-like graph contains eight matrix multiplications and two additions. Additions offer limited propagation opportunities because no lateral transfer applies (Section 3.1.3.1). Even so, Figure 3.8 shows consistent runtime reductions from sparsity propagation. At 50% sparsity, the sparse+propagation version is 1.34× faster than the dense baseline and 1.11× faster than the sparse baseline. At 90% sparsity, the speedups reach 1.58× relative to

dense and 1.17× relative to sparse. Notice that even modest gains at lower sparsity come with negligible overhead, due to the compact bitmap representation and fast convergence of the analysis. Even without propagation, the sparse format outperforms the dense format because the sparsity aligns well with matrix multiplications. Entire rows are made sparse in the associated fiber tree, enabling speedups even at 10% sparsity.

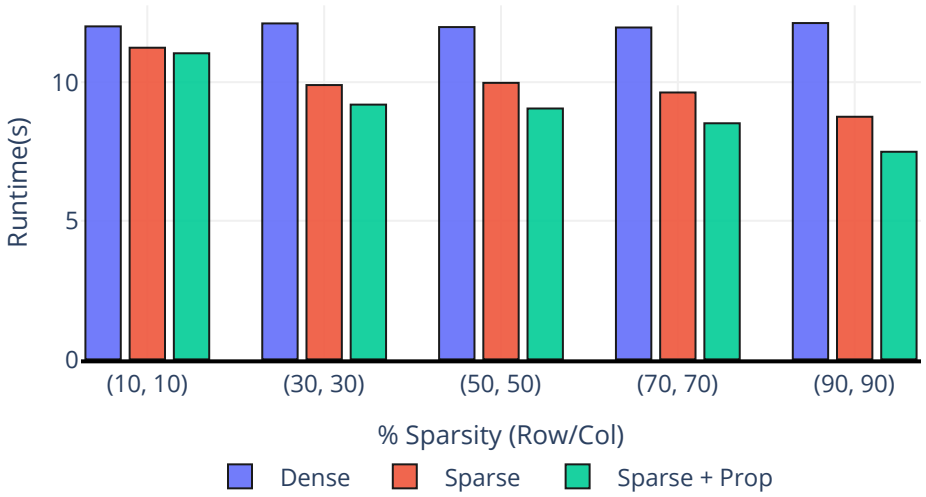


Figure 3.8: Runtime of the BERT-like graph under different sparsity ratios. Configurations compare dense, sparse, and sparse with propagation (SPA). The lower the bars, the better.

Figure 3.9 reports the execution time of different versions of the largest computational graph in our collection, Benchmark 12, which contains 1,999 tensors and 999 kernels. Unlike the results in Figure 3.8, sparsity improves performance only when at least 50% of the slices are sparse. Moreover, in this setting, speedups appear only when SPA leverages both lateral and backward propagation of information. With around 10% sparse slices, SPA has no opportunities for optimization and actually slows down execution by about 60%. In contrast, when 90% of slices are sparse, SPA achieves up to a 2.15× speedup over the dense representation.

Discussion – Memory Usage: Memory savings follow the same trend. Fig-

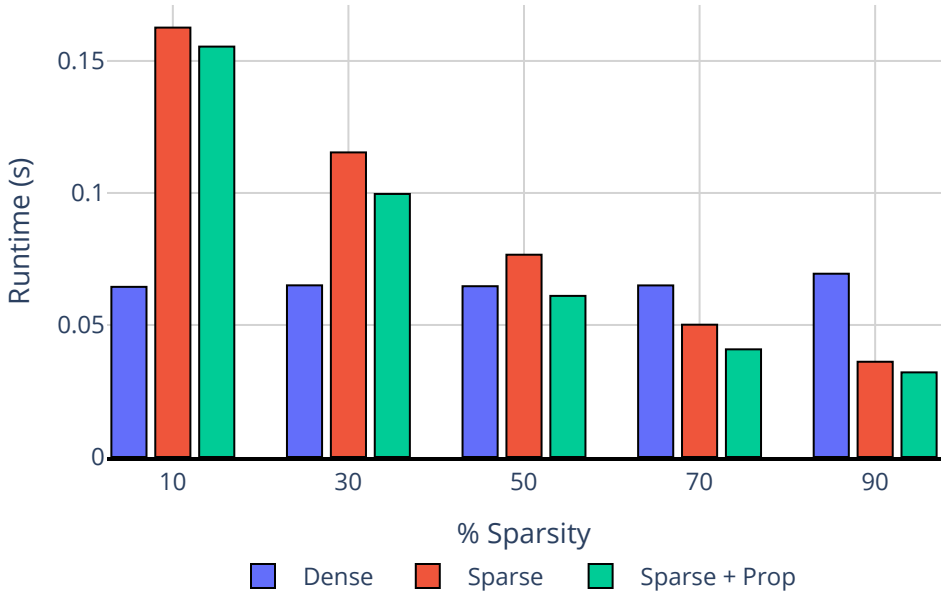


Figure 3.9: Execution time comparison of benchmark 12 using dense, sparse, and sparse with propagation under different sparsity ratios (30-90%). The lower the bars, the better.

ure 3.10 shows tensor sizes normalized to dense storage. Propagation consistently reduces storage requirements beyond what sparse formats alone achieve by exposing additional zero slices eligible for compression. At high sparsity levels, the reduction can exceed 80%.

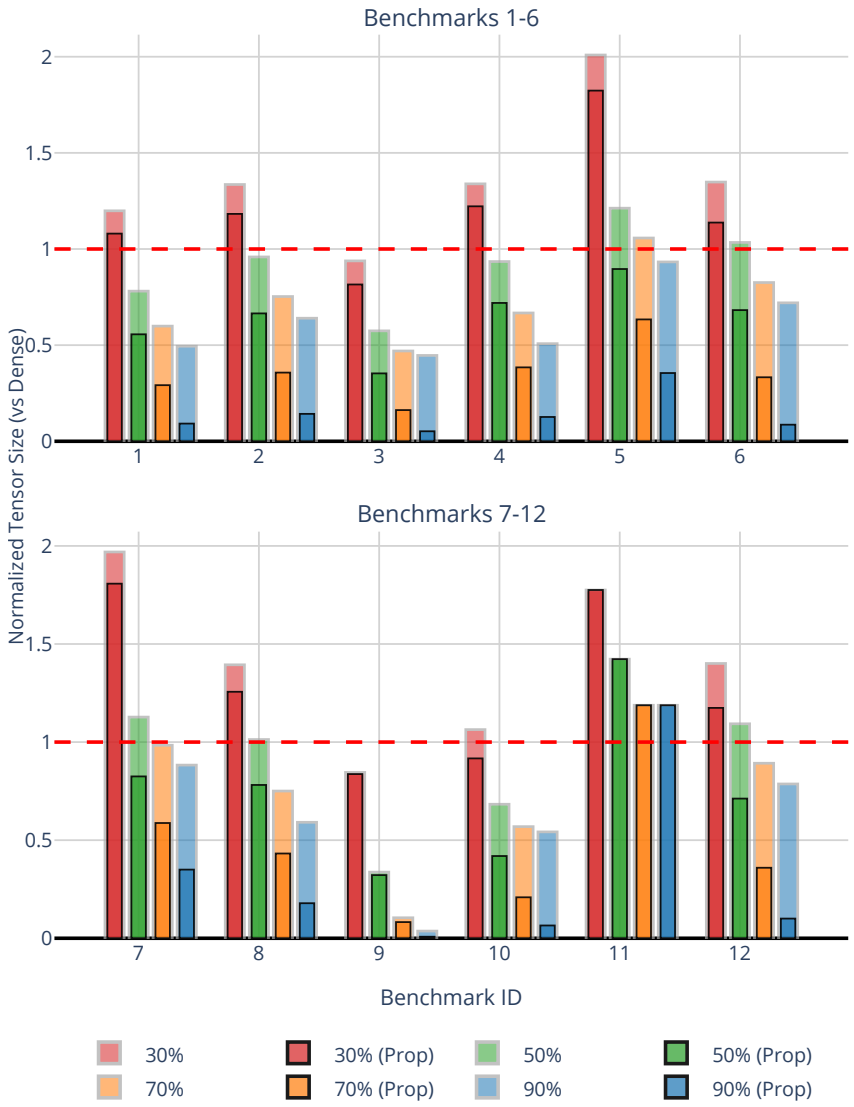


Figure 3.10: Normalized tensor sizes under different sparsity ratios (30–90%). Solid colors: sparse+propagation. Light colors: sparse without propagation. Dashed line: dense baseline. The lower the bars, the better, as less space is required to store the tensors.

Chapter 4

Bandedness Propagation Analysis

A matrix A is banded if $A[i, j] = 0$ whenever $|i - j| > k$ for some small band radius k [22]. When such banded properties appear as slices of higher-dimensional tensors, they capture localized interactions between nearby indices rather than strictly independent ones. Bandedness is relevant in computational graphs: in neural networks, and especially in transformer models, banded substructures naturally arise in mechanisms like local attention (where each token attends only to a neighborhood) [23, 27, 28, 43], convolutional layers (which can be seen as banded Toeplitz operators [5]), and sparse connectivity patterns designed to reduce complexity [19] (See Sec. 4.1 for more examples). Exploiting bandedness allows compilers and runtime systems to reduce both memory footprint and computational cost asymptotically.

There are compiler optimizations designed for computations involving sparse matrices [4, 13, 24, 27, 29, 31, 38, 39, 54, 62, 70, 71]. And there exist static analyses that propagate sparsity information through computational graphs [61, 74]. However, they are oblivious to the structure of non-diagonal sparsity. The work of Zheng et al. [74] propagates sparsity at the level of individual tensor elements, which is asymptotically equivalent to executing the computational graph with a value profiler. The analysis proposed by in Chapter 3 is asymptotically cheaper; however, it associates sparsity with *tensor slices*. In their formulation, a tensor slice is a subset of elements

in which all indices but one are fixed. In the 2D case, these slices correspond to the rows and columns of a matrix. Ironically, as the related work explains, this analysis misses diagonal information: a single nonzero diagonal element taints every slice. In this part of the thesis, we propose a new analysis that succeeds where SPA fails, while being asymptotically more efficient than Zheng et al.'s.

Contributions of this section. This part of the thesis introduces a *Bandedness Propagation Analysis* to infer banded-diagonal invariants across the operations of a computational graph. The analysis decomposes each tensor into a set of *2D Projections* (Definition 17 on Page 11) and associates each 2D substructure with an interval of *non-null bands* projected along those dimensions. Figure 4.1 illustrates this idea.

As Figure 4.1 shows, each banded matrix overapproximates the non-null bands of a tensor along two dimensions. Section 4.2 defines an abstract domain to represent these non-null bands and introduces sound transfer functions that propagate this information throughout the computational graph. Given a generic tensor operation of the form $A = f(B, C)$, information flows in two directions: forward, when the band information of B and C refines that of A; and backward, when the information of A refines that of B and/or C. This part of the thesis defines these transfer functions for common tensor operations, including `einsum` contractions (which subsume matrix multiplication [54]), transpositions, additions, and clamping. As discussed in Section 2.3, these operations can be defined over arbitrary semirings, including the standard semiring of floating-point numbers with addition and multiplication.

Summary of Results. We implemented BPA in MLIR [40], defining transfer functions for operations in the `linalg` dialect, including multiplication, addition, transposition, and batch matrix multiplication. Section 4.4.1 shows that BPA is asymptotically faster than previous sparsity inference analyses, such as those proposed by

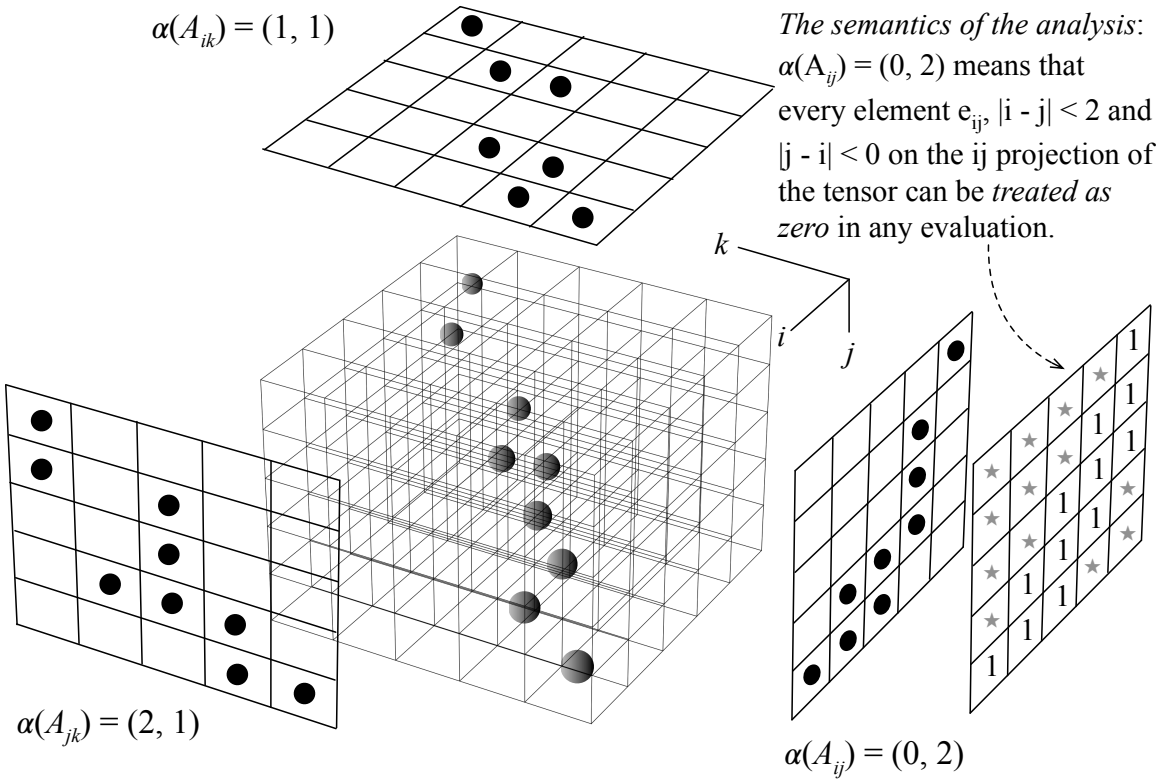


Figure 4.1: A decomposition of a 3D tensor into three 2D projections with the respective abstract states. The abstract information α associated with an n-dimensional tensor A is a table with $n \times (n-1)/2$ entries. Each entry is a pair (l_{ij}, u_{ij}) overapproximating the lower extent of A_{ij} (the largest number of subdiagonals that may contain relevant values) and the upper extent of A_{ij} (the largest number of superdiagonals that may contain relevant values).

Ghorbani et al. [25] and Zheng et al. [74], although it computes less detailed information. Section 4.4.2 presents examples in which BPA-enabled optimizations reduce the memory consumption of computational graphs, such as sparse attention, by up to 50%. Section 4.4.3 provides evidence that kernels generated with the support of BPA can be asymptotically faster, for example, by reducing matrix multiplications to vector products. Section 4.4.4 shows that BPA incurs only a small compilation overhead and that its running time remains constant regardless of the size of the input tensors. Finally, Section 4.4.5 demonstrates empirically that backward propagation of abstract states can improve the precision of forward propagation by up to 33%

when analyzing masked attention kernels.

4.1 Preliminary Definitions

Banded structures emerge in computational graphs that process data with *local*, *monotone*, or *time-ordered* dependencies. The occurrence of such data has been extensively studied by Garriga et al. [22]. In multidimensional tensors, these patterns often arise in 2D projections obtained by fixing a subset of indices. Examples include:

Attention Mechanisms: Attention scores frequently form lower-triangular or banded matrices due to causal masking or sliding-window restrictions [18, Fig.1].

Temporal Dependencies: Sequence models and recurrent computations induce triangular or banded structures because dependencies are constrained by causal ordering or finite histories [67].

Convolutions: Temporal and spatial convolutions induce banded Toeplitz-like structures when lowered to matrix representations, as each output depends only on a local neighborhood of inputs [2, 6, 36, 58].

Truncated Backpropagation: Backpropagation through time produces triangular dependency structures across time steps, which become banded when gradients are truncated to finite temporal windows [50, 73].

The remainder of this section introduces the core concepts used to statically approximate the occurrence of such banded structures in multidimensional tensors.

4.1.1 Annihilating and Irrelevant Entries

As shown in Theorem 60, the analysis soundly approximates the set of annihilable elements. Soundness holds because every element identified as annihilating *must* be

so, and every element identified as irrelevant *may* be treated as annihilating without influencing results. Henceforth, elements that are marked as zero at the bootstrapping of the analysis and elements that are deemed to be annihilating or irrelevant are represented as $\star$ in examples and definitions.

4.1.2 Abstract Domain

Bandedness analysis maps 2D-projections to pairs (p, q) that represent the bands defined in Section 2.5, e.g.:

- $p \in \mathbb{N}$ denotes the lower extent of a matrix t. If $\alpha(t) = (p, \_)$, then $t[i, j] = \star$ for all $i > j + p$.
- $q \in \mathbb{N}$ denotes the upper extent of a matrix t. If $\alpha(t) = (\_, q)$, then $t[i, j] = \star$ for all $j > i + q$.

Each abstract value (p, q) denotes the set of all matrices satisfying the corresponding extent and symmetry constraints, as Example 45 shows.

Example 45 *Continuing with Example 18, Figure 4.2 shows the abstract representation of the bands along the ij projection of the three-dimensional tensor A in Figure 2.5. The abstract representation of the projection A_{ij} is the pair $(1, 2)$. The bandedness propagation analysis associated with A involves three such pairs, one for each possible projection.*

4.1.3 Bandedness Analysis in One Example

The proposed *bandedness analysis* maps each sub-matrix of a tensor to an over-approximation of the band of relevant elements along the corresponding pair of dimensions. The analysis characterizes how bands propagate through common algebraic

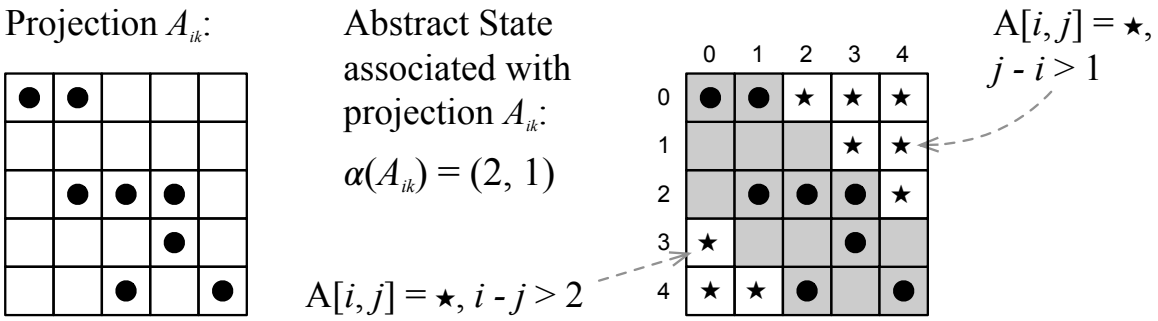


Figure 4.2: The abstract state that represents the projection A_{ij} seen in Figure 2.5.

operations, such as elementwise clamping (e.g., ReLU), tensor addition, and `einsum` expressions, which are pervasive in computational graphs. Example 46 shows the effect of einsum contractions.

Example 46 (Einstein summation) *Figure 4.3 depicts the operation* `einsum`$(xik,\ kj \to xij)$*. This expression denotes a tensor contraction (in this case a batch matrix-matrix product) written as an Einstein summation. Let* $A \in \mathbb{R}^{X \times I \times K}$ *and* $B \in \mathbb{R}^{K \times J}$*. The resulting tensor* $C \in \mathbb{R}^{X \times I \times J}$ *is defined elementwise by the formula below:*

$$C_{xij} = \sum_{k=1}^{K} A_{xik} \cdot B_{kj}$$

The index k *appears in both operands but not in the output because it is summed over. Indices* x, i, j *appear in the output because they are preserved. Operationally, this notation represents a family of matrix multiplications: for each fixed* x*, the slice* $C_{x,:,:}$ *is obtained by multiplying the matrix* $A_{x,:,:}$ *by* B*.*

Example 46 shows the *concrete* effect of an einsum contraction. This part of the thesis is interested in the *abstract* effect the contraction (and other operations) has on the banded sparsity pattern of matrices. Example 47 shows this abstract effect, explaining how bandedness analysis over-approximates the bands of tensors produced by `einsum` operations.

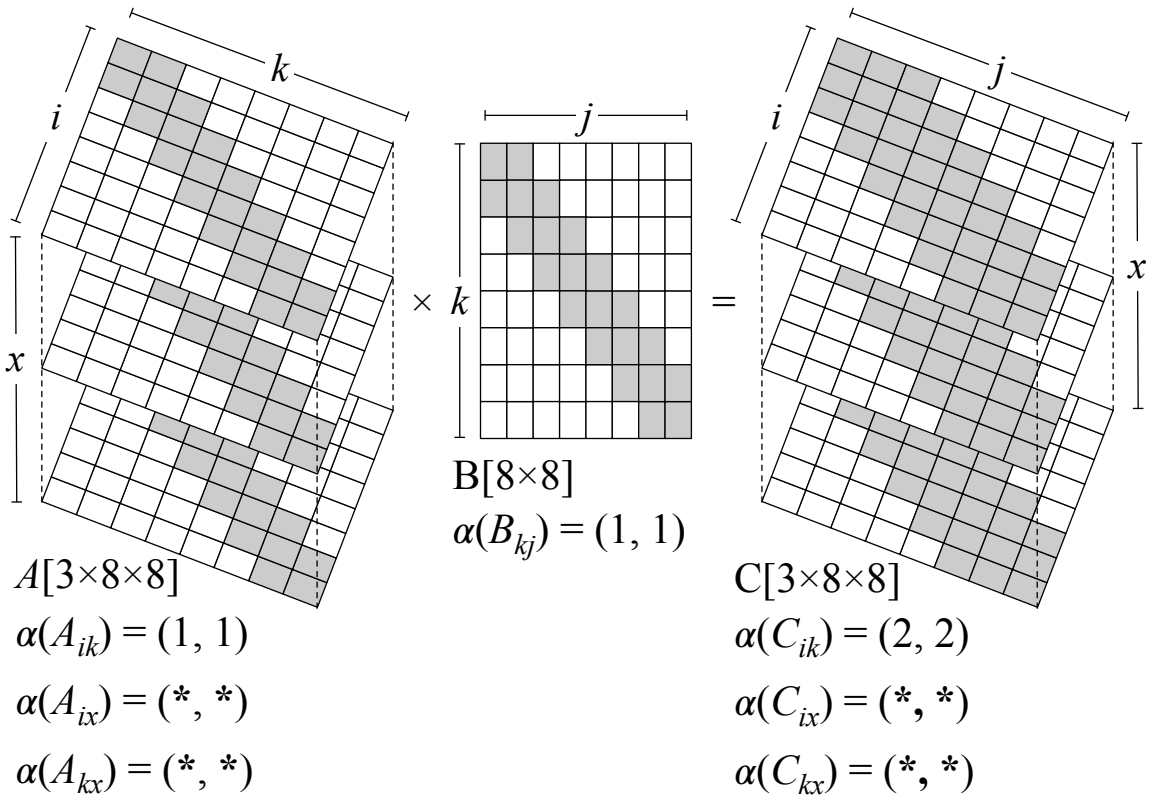


Figure 4.3: Tensor contraction $\texttt{einsum}(xik,\ kj \to xij)$. We use * as a generic placeholder for the maximum bandwidth.

Example 47 (Band propagation) *A tensor A, illustrated in Figure 4.3. is banded along the (i, k) dimensions with symmetric half-bands b, i.e.,*

$$A[x, i, k] = 0 \quad \textit{whenever} \quad |i - k| > b,$$

and that B is banded along (k, j) with the same half-bands:

$$B[k, j] = 0 \quad \textit{whenever} \quad |k - j| > b.$$

A product term $A[x, i, k] \times B[k, j]$ is zero whenever one of the operands is zero. Hence, a contribution to $C[x, i, j]$ exists only for indices k such that

$$|i - k| \leq b \quad \textit{and} \quad |k - j| \leq b.$$

By the triangle inequality, these conditions imply

$$|i - j| \leq |i - k| + |k - j| \leq 2b.$$

Therefore, the resulting tensor C is banded along the (i, j) dimensions with half-extent at most $2b$. The bandedness analysis that Section 4.2 formalizes is able to conclude that the 2D projection $\alpha(C_{ij}) = (2b, 2b)$, provided that $\alpha(A_{ik}) = (b, b)$ and $\alpha(B_{kj}) = (b, b)$.

Example 47 illustrates the forward propagation of abstract information. In this case, we have C computed as a function of tensors A and B. Knowledge of annihilating elements present in A and B lets the analysis infer such elements in C. Section 4.2.3 discusses the propagation of abstract information in the inverse direction.

4.1.4 On the Choice of Granularity

The bandedness propagation analysis tracks band information per 2D projection. A projection A_{ij} aggregates all tensor entries obtained by fixing the indices i and j while varying the remaining modes. In Figure 4.3, A_{ij} summarizes the relevant elements across $A[i, j, 1]$, $A[i, j, 2]$, and $A[i, j, 3]$.

In principle, BPA could be made more precise by tracking information at a finer granularity, e.g., $\alpha(A[i, k, 0]) = (1, 2)$ and $\alpha(A[i, k, 1]) = (2, 1)$, instead of the coarser abstraction $\alpha(A_{ij}) = (2, 2)$. We deliberately avoid this refinement to keep the analysis simple and efficient. Consequently, the amount of information associated with each tensor grows quadratically with the number of its dimensions, but remains independent of the *sizes* of these dimensions, in contrast to previous work. Nor does it depend on the number of elements in a tensor, in contrast to the work of Zheng et al. [74]. Section 4.4 demonstrates empirically that this design choice yields a significantly faster implementation.

4.2 Static Analysis

This section outlines the formal components of the proposed analysis, including its lattice and transfer functions. Soundness and convergence guarantees are discussed in Section 4.3.

4.2.1 Lattice

The abstract domain introduced in Section 4.1.2 induces a natural concretization function γ that maps abstract values, representing the band extents (p, q), to sets of concrete matrices. The partial order $\sqsubseteq$ on abstract values is defined by set inclusion over their concretizations:

$$(p_0, q_0) \sqsubseteq (p_1, q_1) \quad \text{iff} \quad \gamma(p_0, q_0) \supseteq \gamma(p_1, q_1).$$

This relation holds if and only if $p_0 \geq p_1 \ \wedge \ q_0 \geq q_1$. Therefore, the larger the extents of irrelevant values (the null bands), the higher up in the lattice the matrix is. The highest element in this lattice is the set of diagonal matrices, defined by $\top = (0, 0)$. The lowest element, in turn, is the general matrix, whose abstract state is $\bot = (*, *)$, where $*$ is the largest index value in the matrix. The least upper bound (join) of two abstract values is given by:

$$(p_0, q_0) \ \vee \ (p_1, q_1) = (\min(p_0, p_1), \min(q_0, q_1)).$$

Example 48 *Figure 4.4 shows the lattice induced by the set of* 4×4 *matrices. The figure represents the lattice as a Hasse Diagram, along with some abstract values and the instances of tensors that belong to those sets. The arrows show the partial order that determines the lattice structure, meaning that if* $a_0 \to a_1$*, then the relationship* $a_1 > a_0$ *holds between abstract states* a_0 *and* a_1[1]*.*

[1]This lattice is equivalent to several well-known structures. For instance, it is isomorphic to the divisor lattice of numbers of the form $m^{N-1}n^{N-1}$, where m and n are distinct primes, or to the lattice of rectangles anchored at the origin in the grid $[0, N-1] \times [0, N-1]$, ordered by inclusion.

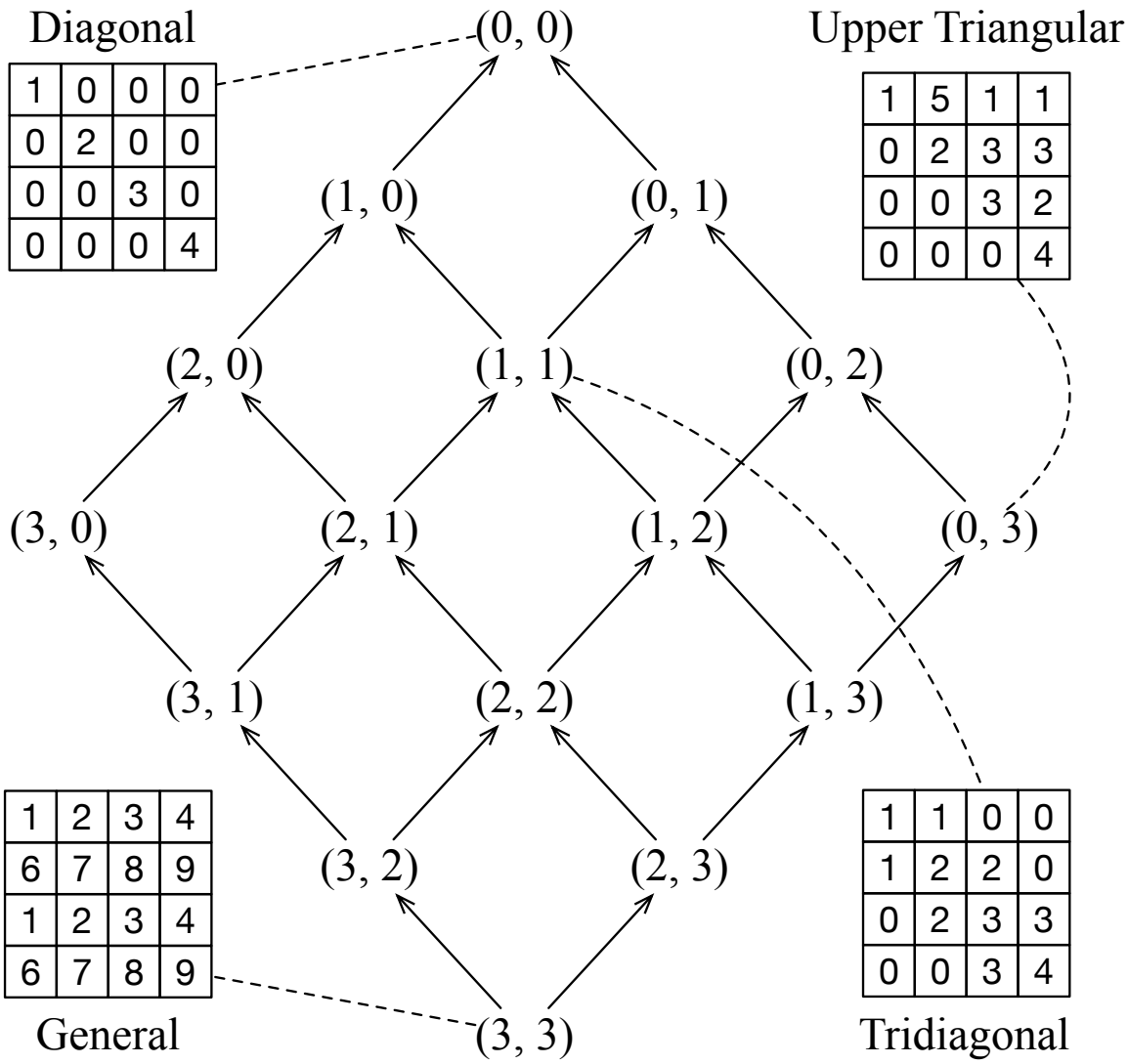


Figure 4.4: Partial order induced by the least-upper bound of algebraic properties.

4.2.2 Forward Transfer Functions

Figure 4.5 shows the transfer functions that propagate information about elements that *must* be *annihilating*, like zeros in the $(\mathbb{R}, +, \times)$ semiring, for instance. Abstract interpretation follows any topological ordering of the computational graph. Each operation *op* is associated with a transfer function $f_{op} : \mathcal{P}^k \to \mathcal{P}$, where $\mathcal{P}$ is the lattice of bands shown in Figure 4.4, and k is the arity of the operation.

The abstract states of the input operands are initialized outside the analysis. This process is called *bootstrapping*, and happens, for instance, via user annotations. This initialization determines which elements of a tensor are known to be *zeros* (or, more generally, annihilating elements, as explained in Section 2.3) before the analysis starts.

4.2.2.1 Unary Transfer Functions

Rules `csum`, `padx`, `zpew`, and `trsp` in Figure 4.5 show the abstract interpretation of unary operations. The latter refers to transposition, e.g., $\texttt{einsum}(ij \to ji)$ is equivalent to $B = A^{T}$[2]. Rule `csum` refers to cumulative reductions, e.g., prefix sums involving sum, logical or, product, etc. We show reduction along rows, but reductions along columns apply too, in this case yielding $(*, q)$, as Figure 4.6 shows. Rule `padx` refers to any padding operation that extends a 2D matrix with annihilating elements along its top (t), bottom (b), left (l), and right (r) sides. Similar rule applies for trimming operations that remove elements from matrices (see Figure 4.6).

Rule `zpew` refers to any operation $B = g(A)$, where g is a zero-preserving element-wise unary operation (e.g., abs, neg, relu, floor, round, sqrt, etc.). They preserve banded structure, e.g., $\alpha(B) = \alpha(A)$. Preservation of zeros excludes operations like exp, sigmoid, and scalar additions.

[2]Figure 4.5 adds names to the indices of einsum operations to be able to refer to specific projections. Thus, we use $B_{ij} = \texttt{einsum}(\texttt{A}_{\texttt{ji}})$ to represent the standard $\texttt{einsum}(ij \to ji)$ operation.

$$\frac{B_{ij} = \text{csum\_rows}(A_{ij}) \in G \quad \alpha(A_{ij}) = (p, q)}{\alpha(B_{ij}) = (p, *)} \quad \texttt{[csum]}$$

$$\frac{B_{ij} = \text{pad}(A_{ij}, t, b, l, r) \in G \quad \alpha(A_{ij}) = (p, q)}{\alpha(B_{ij}) = (p, q)} \quad \texttt{[padx]}$$

$$\frac{B_{ij} = \text{zero\_preserving\_elementwise}(A_{ij}) \in G \qquad \alpha(A_{ij}) = (p, q)}{\alpha(B_{ij}) = (p, q)} \quad \texttt{[zpew]}$$

$$\frac{B_{ij} = \text{einsum}(A_{ji}) \in G \qquad \alpha(A_{ji}) = (p, q)}{\alpha(B_{ij}) = (q, p)} \quad \texttt{[trsp]}$$

$$\frac{C_{ij} = \text{einsum}(A_{ik}, B_{kj}) \in G \quad \alpha(A_{ik}) = (p_A, q_A) \quad \alpha(B_{kj}) = (p_B, q_B)}{\alpha(C_{ij}) = (\max(p_A + p_B, *), \max(q_A + q_B, *))} \quad \texttt{[mmul]}$$

$$\frac{C_{ij} = \text{add}(A_{ij}, B_{ij}) \in G \quad \alpha(A_{ij}) = (p_A, q_A) \quad \alpha(B_{ij}) = (p_B, q_B)}{\alpha(C_{ij}) = (\max(p_A, p_B), \max(q_A, q_B))} \quad \texttt{[plus]}$$

$$\frac{C_{ij} = \text{hadamard}(A_{ij}, B_{ij}) \in G \quad \alpha(A_{ij}) = (p_A, q_A) \quad \alpha(B_{ij}) = (p_B, q_B)}{\alpha(C_{ij}) = (\min(p_A, p_B), \min(q_A, q_B))} \quad \texttt{[hdmd]}$$

Figure 4.5: Forward transfer functions associated with unary and binary tensor operations.

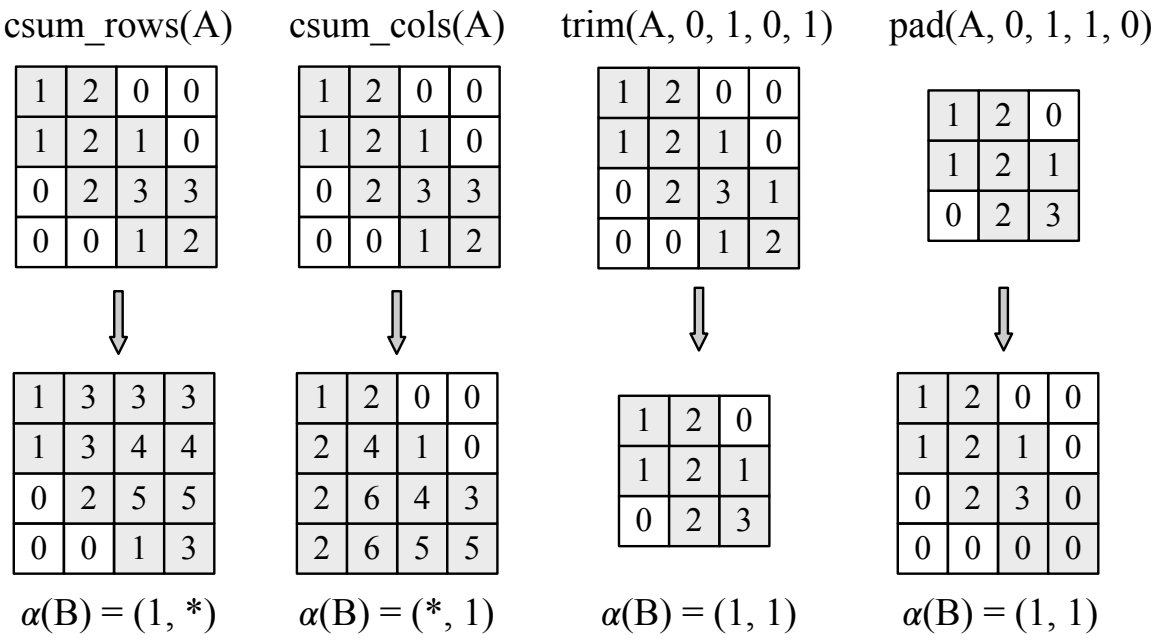


Figure 4.6: Unary operations and the abstract states they generate, assuming $\alpha(A) = (1, 1)$.

Example 49 *The clamp operator* $\text{clamp}(x) = \max(0, x)$*, which implements the Rectifier Linear Unit (ReLU), is a case of Rule* `zpew`*: it is an element-wise unary operation that preserves all zero entries. As a result, the algebraic structure induced by zeros is preserved, e.g.:* $\alpha(\text{clamp}(A)) = \alpha(A)$.

4.2.2.2 Binary Transfer Functions

Rules `mmul` (matrix multiplication), `plus` (matrix addition), and `hdmd` (Hadamard product) refer to binary operations. Other additive element-wise binary operators (subtraction, or, xor, etc.) follow the transfer function of Rule `plus`. Other multiplicative element-wise binary operators (such as boolean and, modulo and division) follow Rule `hdmd`. If no rule applies, the transfer function conservatively returns the top element $(*, *)$, representing an unknown structure.

Together, the rules in Figure 4.5 can be understood as a set of *abstract bytecodes*, which allow BPA to analyze complex operations as decompositions of simpler ones.

This view enables the inference of banded structures over operations involving multi-dimensional tensors and permutations of indices. As an example, BPA treats an operation such as $\texttt{einsum}(ki, kj \rightarrow ji)$ in the same way as $(A^T B)^T$. Example 50 illustrates how a more complex operation is handled.

Example 50 *Consider* $C_{abcef} = \texttt{einsum}(A_{abcd}, B_{def})$*. This operation performs a tensor contraction over the shared index* d *and an outer product over the remaining unique indices* a*,* b*,* c*,* e*, and* f *to produce a five-dimensional output tensor. BPA represents this operation as a product* $C_{ce} = \texttt{einsum}(A_{cd}, B_{de})$*, and treats the other combinations of indices as identities. Thus, if the projections over dimensions* cd *and* de *are banded, the* ce *projection will also be banded, whereas the other projections are propagated as-is.*

4.2.3 Backward Transfer Functions

Binary operations that form a semiring (see Section 2.3) enables the backward inference of elements that *may* be irrelevant to computations. In this setting, interactions with annihilating values restrict which elements of the operands can influence that result. Example 51 illustrates this process.

Example 51 *Consider the operation* $C = A + B$*. Assume that it has been determined that* $\alpha(C) = (1, 1)$*, then no entry of* A *or* B *outside the tridiagonal band can influence the computation. Consequently, these entries are irrelevant and can be treated as zeros for this operation. In a computational graph, an entry is deemed to be zero if it can be regarded as zero for all the operations for which it is an operand.*

More generally, suppose the computational graph contains k operations that consume a tensor A to produce tensors $B_1, \ldots, B_k$. Backward propagation updates $\alpha(A)$ as follows:

$$\alpha(A) = \alpha(A) \vee (\alpha(B_1) \wedge \cdots \wedge \alpha(B_k)). \tag{4.1}$$

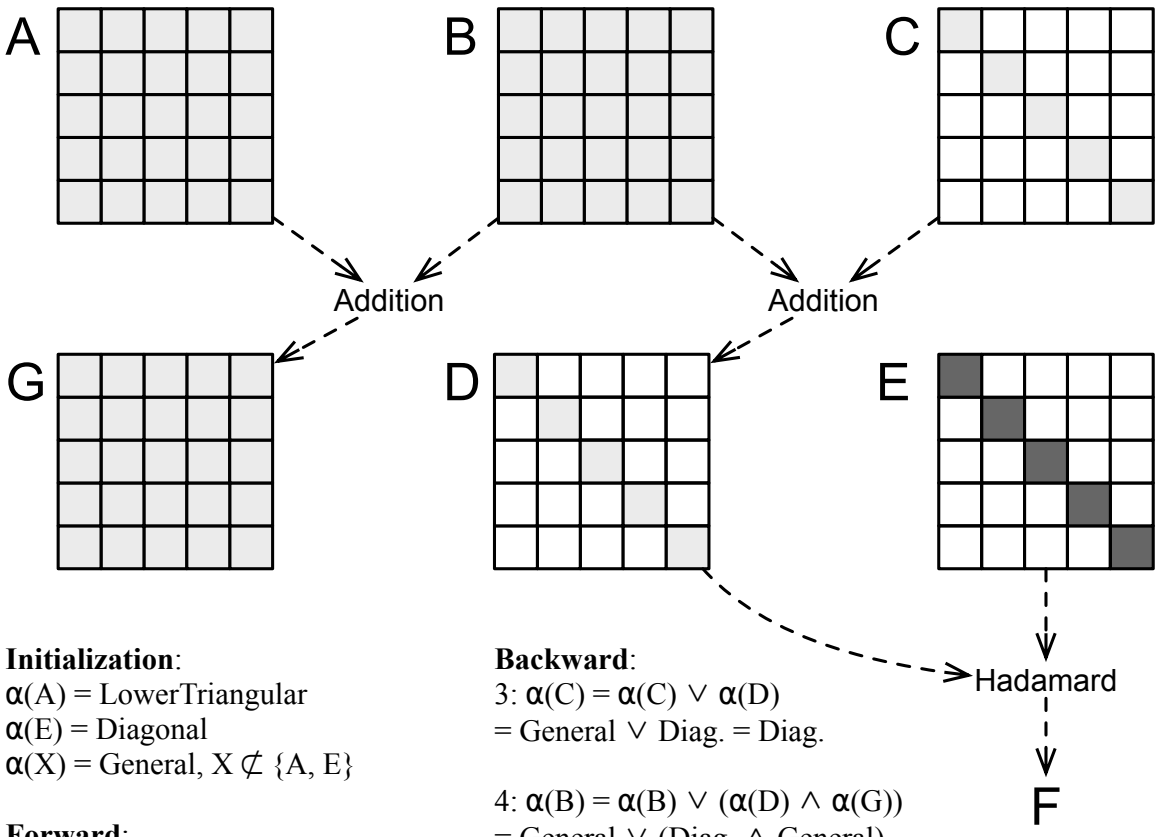


Figure 4.7: Example illustrating the role of the meet operation in sound backward propagation.

The meet operation ensures soundness when a tensor participates in multiple operations: an entry of A is marked as irrelevant only if it is irrelevant for *all* of its uses. Example 52 illustrates this property.

Example 52 *Figure 4.7 shows how information flows backward from tensors D and G to tensors A, B, and C. Backward propagation from D refines the abstract state of C, but cannot refine the state of B, because B also receives backward information from G, whose abstract state is $(*, *)$.*

Theorem 63 shows that the bandedness propagation analysis reaches a fixed point

on any acyclic computational graph G after a single forward pass followed by a single backward pass. The forward pass follows a topological ordering of the nodes in G, while the backward pass follows the corresponding reverse ordering.

Theorem 63: Let G be an acyclic computational graph. A single forward pass followed by a single backward pass computes a fixed point of the analysis.

Proof. Available in the supplementary material. ■

Special Cases. Given an operation $B = f(\ldots, A, \ldots)$, Equation 4.1 uses $\alpha(B)$ to refine $\alpha(A)$. There are two important cases in which $\alpha(B)$ must be adjusted:

Transposition: Transposition permutes tensor dimensions by swapping rows and columns. Thus, if $B = A^T$ and $\alpha(B) = (p, q)$, then backward propagation uses (q, p) when updating $\alpha(A)$.

Multiplication: For matrix multiplication, e.g., $B = A \times X$, backward propagation must conservatively assume that all entries of A may contribute to B. Thus, we use $\alpha(B) = (*, *)$ in Equation 4.1. This loss of precision is necessary for soundness, as every non-zero entry of A may influence the result.

Example 53 *Figure 4.8 illustrates the effect of transposition on backward propagation. In this case, Equation 4.1 swaps the band extents of B when computing the abstract state of A.*

4.3 Correctness

The goals of this section are two fold: first, in Section 4.3.1 we demonstrate that the bandedness propagation analysis formalized in Section 4.2 produces a sound approximation of relevant values; second, in Section 4.3.2 we demonstrate that the analysis

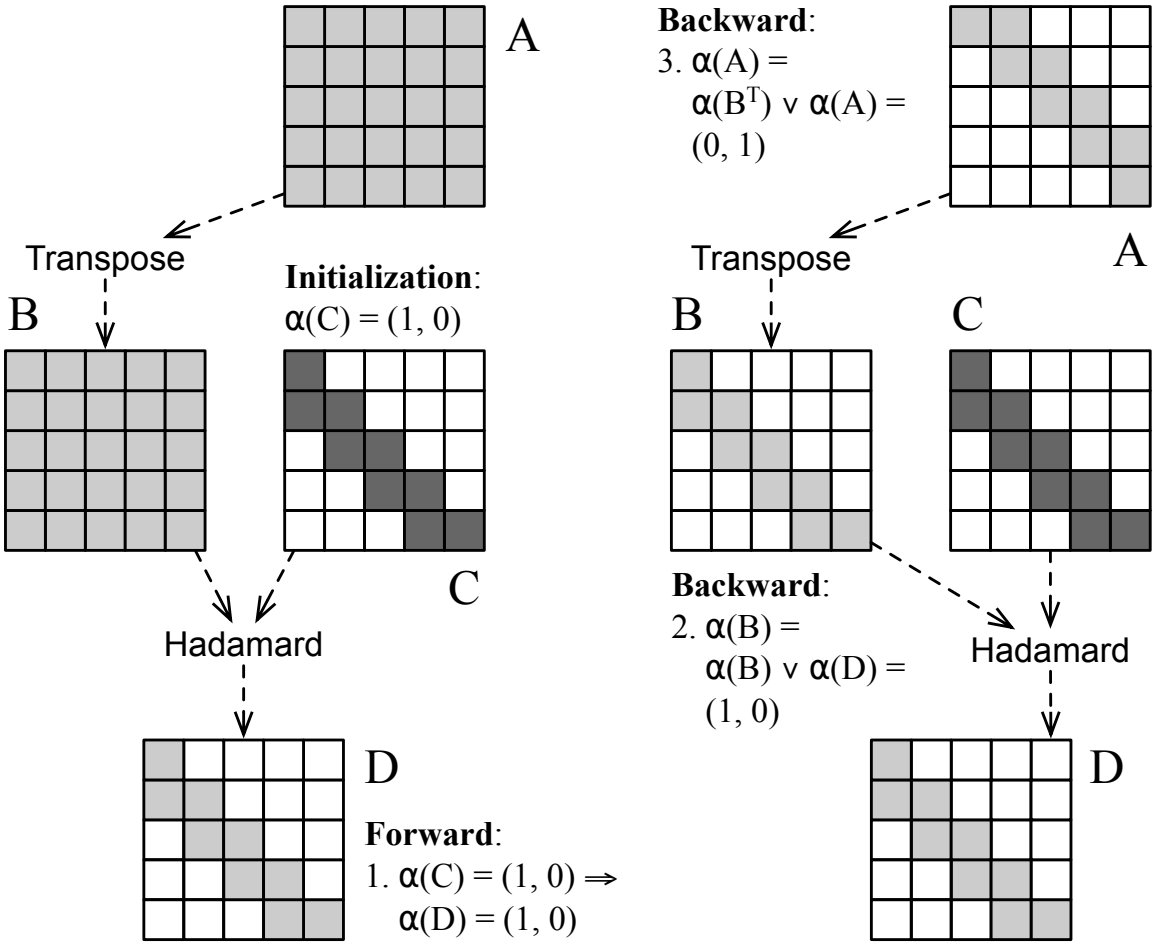


Figure 4.8: (Left) Effects of forward propagation. (Right) Effects of backward propagation.

always terminates. Termination is ensured after a forward plus a backward pass on an acyclic computational graph. If the graph contains cycles, the analysis still terminate; however, multiple passes might be necessary.

4.3.1 Soundness

We define soundness in terms of the semantic irrelevance introduced in Definition 16. Intuitively, the analysis is sound if every tensor entry marked as irrelevant ($\star$) can be replaced by an arbitrary value (the annihilating element of the underlying semiring, for example) without affecting the observable result of the computational graph. Definition 54 formalizes this notion of soundness.

Definition 54 (Soundness of Bandedness Propagation Analysis) *Let G be a computational graph, and let α be the abstract environment computed by the analysis. For each tensor A, let $\mathcal{I}_\alpha(A)$ be the set of indices marked as irrelevant by $\alpha(A)$. The abstraction α is* sound *if, for every valuation $[\![G]\!]$ of G, replacing the entries of any tensor $A \in G$ at indices in $\mathcal{I}_\alpha(A)$ by arbitrary values does not change the observable result of G. Formally, for any tensor A' such that*

$$A[i] = A'[i] \text{ for all } i \notin \mathcal{I}_\alpha(A),$$

we have:

$$[\![G]\!](A_1, \ldots, A, \ldots, A_n) = [\![G]\!](A_1, \ldots, A', \ldots, A_n).$$

4.3.1.1 Soundness of Forward Propagation

Definition 55 (Forward Pass) *The forward pass over a computational graph G is a topological traversal in which, for each operation, its forward transfer function is applied only after the abstract states α of all its inputs have been computed.*

Theorem 56 (Soundness of Forward Analysis) *Let G be a computational graph. If the initial abstract states α of the input tensors are sound, then the forward pass computes an abstract environment α that is sound for G.*

Proof. The proof proceeds by induction on any forward pass over G. In the base case, we have only input tensors, which have no predecessors in G. In this case, α is sound by assumption, as the abstract states of inputs are provided by the user and satisfy Definition 54.

For the inductive step, consider an operation whose inputs have already been assigned sound abstract states. We show that the abstract state computed for its output is also sound. The argument proceeds by case analysis on the forward transfer functions defined in Sections 4.2.2.1 and 4.2.2.2.

For instance, consider the transpose operation. Its transfer function is:

$$\alpha(A) = (p, q) \Rightarrow \alpha(A^T) = (q, p).$$

By definition of $\alpha(A)$, entries of A outside the band defined by (p, q) are annihilable (Def. 16). The transpose operation is a bijection on indices, mapping each entry $A[i, j]$ to $A^T[j, i]$. Hence, an entry of A^T is irrelevant if and only if the corresponding entry of A is irrelevant. Therefore, swapping the extents preserves the set of irrelevant entries, and the transfer function is sound.

The remaining cases follow by similar arguments, relying on the semantics of each operation and the definition of irrelevant entries. We omit them for brevity.

Thus, by induction, every tensor in G is assigned an abstract state α that is sound for G. ■

4.3.1.2 Soundness of Backward Propagation

Definition 57 (Backward Pass) *The backward pass over a computational graph G is a reverse topological traversal in which, for each operation, its backward transfer*

function is applied only after the abstract states α of all its outputs have been computed.

Lemma 58 (Soundness of Backward Transfer Function) *Let A be a tensor used by a set of operations with outputs $C_1, \ldots, C_m$ and $D_1, \ldots, D_n$, as in Equation 4.1. If the abstract states $\alpha(C_i)$ and $\alpha(D_j)$ are sound, then the update in Equation 4.1 produces an abstract state $\alpha(A)$ that is also sound.*

Proof. By assumption, each $\alpha(C_i)$ and $\alpha(D_j)$ is sound, i.e., entries marked as irrelevant in these tensors do not affect the observable result of G. Consider an entry of A that lies outside the band defined by

$$\bigwedge_{i=1}^{m} \alpha(C_i) \ \wedge \ \bigwedge_{j=1}^{n} \alpha(D_j).$$

By construction of the backward transfer function, such an entry cannot influence any of the outputs C_i or D_j. Hence, this entry can be replaced by an arbitrary value without affecting the observable result of any operation that consumes A. Taking the meet ($\wedge$) ensures that only entries that are irrelevant for *all* uses of A are marked as irrelevant. Joining ($\vee$) this result with the existing state $\alpha(A)$ preserves any previously known irrelevance information. Therefore, the updated abstract state $\alpha(A)$ is sound.

■

Theorem 59 (Soundness of Backward Analysis) *Let G be a computational graph. If the abstract states α of the output tensors are sound, then the backward pass computes an abstract environment α that is sound for G.*

Proof. The proof proceeds by induction on any backward pass over G. In the base case, we have output tensors, whose abstract states are sound as a consequence of forward propagation (Theorem 56).

For the inductive step, we consider Equation 4.1. By Lemma 58, applying the backward transfer function produces a sound abstract state for each of its operands. For the transpose operation, if $\alpha(B) = (p, q)$ with $B = A^T$, then the backward transfer function sets $\alpha(A) = (q, p)$. Since transposition is a bijection on indices, an entry of A is irrelevant if and only if the corresponding entry of B is irrelevant. Thus, the transfer function preserves irrelevance and is sound. For matrix multiplication, we notice that we replace $\alpha(B)$ with $(*, *)$, which is always sound, as it is the bottom of the lattice. ■

Theorem 60 (Soundness) *Bandedness Propagation Analysis is sound.*

Proof. The result follows from Theorems 56 and 59. ■

4.3.2 Convergence

Intuitively, a propagation pass is *idempotent* if applying it more than once has no further effect after the first application. In other words, once the abstract environment α has stabilized, reapplying the same pass does not refine or change any abstract state. This property is essential to establish convergence, as it ensures that repeated applications of the pass do not lead to additional updates. In what follows, we assume that the results apply over an *acyclic* computational graph.

Lemma 61 (Backward Pass Idempotence) *After one forward pass and one backward pass on an acyclic computational graph, the backward pass is idempotent.*

Proof. By Equation 4.1, the backward update for a tensor A computes the meet ($\wedge$) over the abstract states of all operations that consume A. Let the combined constraint be:

$$S = \bigwedge_{i=1}^{m} \alpha(C_i) \ \wedge \ \bigwedge_{j=1}^{n} \alpha(D_j)$$

After one backward pass, the abstract state of A is:

$$\alpha(A) = \alpha(A) \vee S.$$

A subsequent backward pass recomputes the same meet S:

$$\alpha(A) = (\alpha(A) \vee S) \vee S.$$

By associativity and idempotence of $\vee$, this simplifies to

$$\alpha(A) \vee S.$$

Thus, the abstract state of A remains unchanged under repeated backward updates. ■

Lemma 62 (Forward Pass Idempotence) *Let G be a computational graph. After one forward pass followed by one backward pass, the forward pass is idempotent.*

Proof. Since each forward update has the form $\alpha(X) \leftarrow \alpha(X) \vee f(\cdot)$, and $f(\cdot) \sqsubseteq \alpha(X)$ after the backward pass, the join does not change $\alpha(X)$. Hence, no abstract state is modified during a subsequent forward pass. ■

Theorem 63 (Global Convergence) *Let G be an acyclic computational graph. A single forward pass followed by a single backward pass computes a fixed point of the analysis, i.e., the resulting abstract environment α is unchanged by any subsequent forward or backward passes.*

Proof. Let α be the abstract environment obtained after one forward pass followed by one backward pass. By Lemma 61, reapplying the backward pass does not change α. By Lemma 62, reapplying the forward pass does not change α either. ■

4.4 Evaluation

This section evaluates the impact of the bandedness propagation analysis by addressing the following research questions:

RQ1: What is the asymptotic complexity of the bandedness propagation algorithm?

RQ2: What are the potential memory savings enabled by optimizations based on bandedness propagation?

RQ3: What are the potential time savings enabled by optimizations based on bandedness propagation?

RQ4: How does the analysis runtime impact the compilation pipeline?

RQ5: How much sparsity can be propagated by the forward and backward passes?

Software and Hardware: The bandedness propagation analysis was implemented in MLIR (LLVM 21.0). All experiments were conducted on an AMD Ryzen Threadripper 7970X (32 cores, 64 threads) with 128 GiB of RAM, running Ubuntu 24.04.4 LTS. All experiments were collected using the average of 10 executions plus one warmup execution.

Benchmarks: To evaluate the proposed research questions, we use four benchmarks that typically show banded patterns:

Kalman: The prediction covariance step of the Kalman Filter algorithm [35], e.g.: $P_{k|k-1} = FP_{k-1|k-1}F^T + Q$,å where the matrices P, Q and F are 1024×1024.

Sparse Attention: An implementation of masked attention, where addition with $-\infty$ is replaced by a Hadamard product of a boolean tensor, and softmax operates within bandwidth boundaries. This is identical to traditional softmax on a fully dense matrix.

Batch Bertlike: a batched implementation of the BERT-like model used in Chapter 3 but written in MLIR's linalg dialect. Tensors are $4 \times 1024 \times 1024$.

Chain: A kernel formed by a chain of matrix multiplications, following the benchmarking methodology used by Lenadora et al. [41]. Matrices are 1024×1024.

Configurations: This thesis uses the bandedness propagation analysis to implement five different code generators:

Dense: Stores all tensor elements (including zeroes) in a standard row-major format, but uses the static bandedness analysis to skip empty diagonals during computation and eliminate redundant graph operations (like transposing a diagonal matrix).

Compressed : Stores only the diagonals of matrices using a diagonal compressed format [57] (with padding) and produces all intermediate tensors in this format; however, it expects the initial input tensors to be provided in a dense format.

Compressed-Input: Process and produce all intermediate tensors in the compressed diagonal format, but strictly requires the initial input tensors to be compressed in the DIA format as well.

Hybrid: Employs a static heuristic based on output bandwidth to dynamically choose the optimal storage format for each intermediate tensor, using the compressed DIA format at low bandwidths to save space and switching to the dense format at high bandwidths to avoid diagonal padding overhead.

Hybrid-Compressed-Input (Hybrid-CI): Combines the hybrid heuristic with the requirement of compressed inputs; it starts with DIA-formatted inputs and dynamically switches intermediate and output tensor formats between dense and DIA based on the predicted storage overhead.

Baselines: This part of the thesis uses two baselines:

Baseline: A standard implementation of the computational graphs in MLIR's linalg dialect. MLIR's code generator produces loops in the common IJK order when implementing matrix multiplication.

Baseline-ikj: The same implementation, however, with the IKJ loop ordering, which we invert via a non-standard MLIR pass, to improve locality.

4.4.1 RQ1: Asymptotic Behavior

The asymptotic behavior of an algorithm describes how its time or space complexity scales as the size of the input grows. We claim that the bandedness propagation analysis (BPA) exhibits better asymptotic behavior than other static analyses capable of subsuming its results. To support this claim, we compare BPA against the following techniques:

StructTensor [25]: the static analysis used in the implementation of the DASTAC compiler [24].

SparTA [74]: a static analysis that propagates sparsity information per individual tensor elements.

Section 5 provides additional details about these analyses. We do not compare against SPA, because it is unable to infer any sparsity for the benchmarks used in this part of the thesis.

Discussion: Figure 4.9 compares the asymptotic behavior of the different static analyses. Because the figure uses a logarithmic scale, the poorer asymptotic complexity of StructTensor is readily apparent. StructTensor scales poorly because it requires constructing and simplifying symbolic expressions.

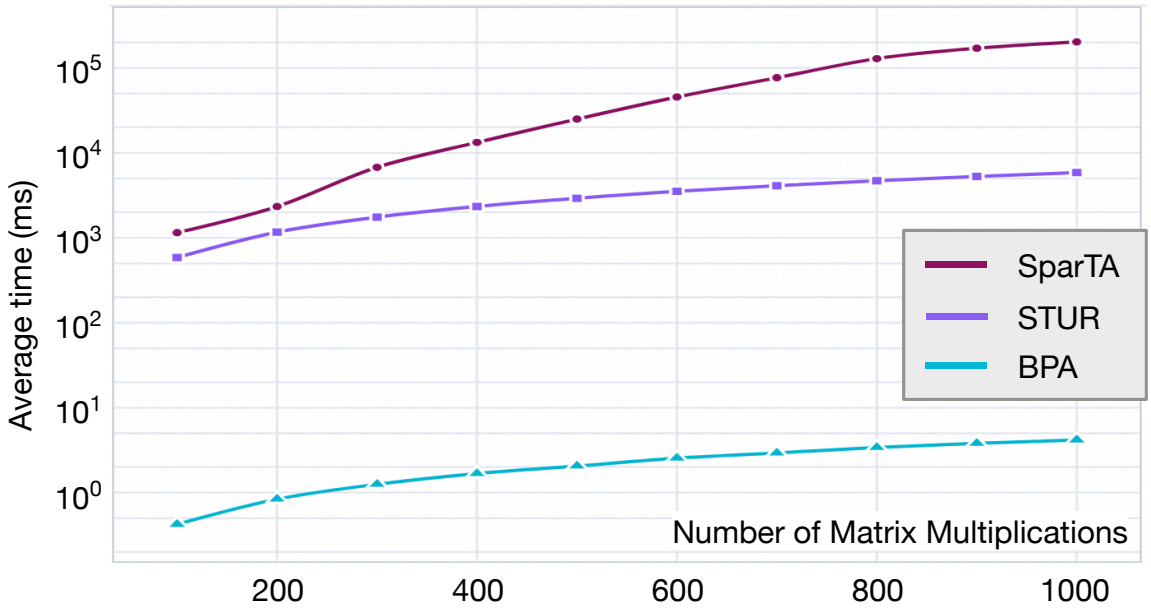


Figure 4.9: Logarithmic plot comparing the symbolic execution cost of different analyses. STUR refers to StructTensor.

SparTA and BPA both scale linearly with the number of tensors in the graph, but BPA has a substantially smaller constant factor. Furthermore, BPA's runtime depends only on the number of tensor dimensions, whereas SparTA propagates information for individual tensor elements, effectively executing the computational graph over boolean values. Consequently, for the matrix-multiplication chains used in Figure 4.9, SparTA's runtime grows quadratically with the tensor dimensions, while BPA's remains constant.

4.4.2 RQ2: Memory Consumption

Bandedness Propagation Analysis enables more compact tensor storage. This section evaluates these memory gains. Peak memory usage is reported in gigabytes, measured via the Linux `/usr/bin/time` utility. We omit results for the MLIR **Baseline-ikj**, noting that loop interchange has no impact on memory footprints.

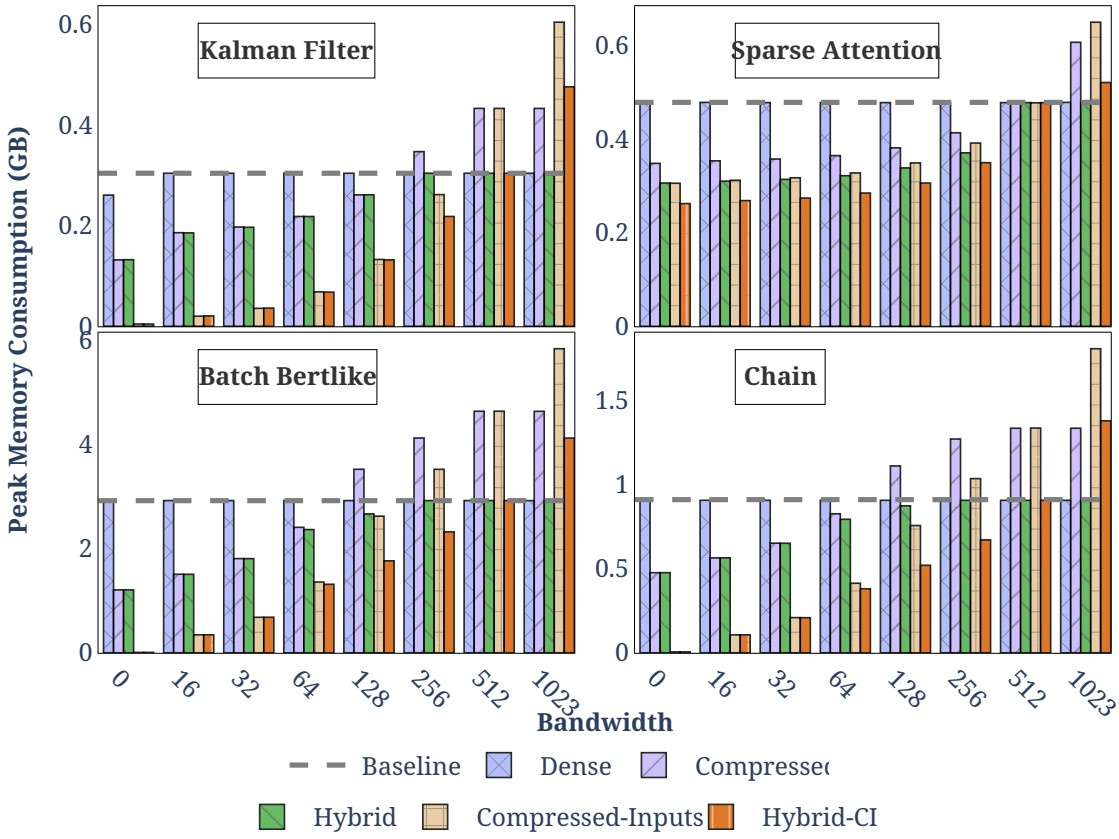


Figure 4.10: Peak memory consumption of the four computational graphs implemented using the bandedness analysis across different formats and bandwidth values.

Discussion: Figure 4.10 illustrates the memory required by the different code generators across the four benchmarks as the input tensor bandwidth varies from 0 (a diagonal tensor) to 1023 (a fully dense tensor). We observe storage savings when the compressed diagonal (**Compressed** and **Compressed-Inputs**) code generators are used. To preserve cache locality, the DIA format pads all diagonals to match the length of the main diagonal, enabling linear memory access. However, this padding introduces a storage overhead of $\frac{p^2+p}{2} + \frac{q^2+q}{2}$ entries, where p and q represent the lower and upper bandwidths, respectively. For an $N \times N$ matrix, this padding overhead causes the DIA representation to exceed the size of a standard dense matrix whenever $(p+q) \geq N-1$.

The **Dense** and **Baseline** code generators employ the same storage scheme. However, the code generated for the **Dense** configuration leverages BPA to skip empty diagonals, computing a diagonal-by-diagonal product. Even though it allocates space for all tensor elements (including empty diagonals), the **Dense** approach can still outperform the **Baseline** in memory consumption. This occurs because BPA eliminates redundant graph operations; for example, it removes a matrix transposition when the bandwidth is zero, as shown in the Kalman Filter plot (row=1, col=1).

The **hybrid** approaches allow the compiler to dynamically choose the storage format of an intermediate or output tensor based on the properties inferred by BPA. For instance, a dense 1024×1024 banded matrix with $(p, q) = (256, 256)$ contains approximately 56% redundant zero entries. In contrast, storing this matrix in the DIA format incurs only a 12.5% overhead due to diagonal padding—yielding a 44% reduction in wasted space. Consequently, the **hybrid** configuration (operating on dense inputs) improves upon the **Dense** baseline at lower bandwidths (0–128) by lowering intermediates to the compressed DIA layout while padding overhead remains minimal. At medium to high bandwidths (256–1023), the selection heuristic switches to a dense format for intermediate allocations, avoiding the explosive padding overhead brought on by a high number of active diagonals. Here, the storage choices for the **hybrid** and **hybrid-dia-inputs** configurations are determined ahead-of-time (AOT) using the results of BPA on user-annotated inputs. Nonetheless, as demonstrated in Section 4.4.1, BPA executes rapidly enough that these structural optimization decisions could be deferred to a just-in-time (JIT) compilation model based on runtime-inferred shapes and bandwidths.

4.4.3 RQ3: Execution Time Savings

Bandedness Propagation Analysis (BPA) enables compiler optimizations that significantly reduce program execution times. Performance gains stem either from specialized code generation that operates exclusively on non-null bands or from improved memory locality that enhances cache utilization. To evaluate this impact, this section compares our various BPA-driven code generators against two standard MLIR baselines. Execution time is reported in seconds and encompasses end-to-end benchmark execution, including data loading, input parsing, and result serialization.

Discussion: Figure 4.11 illustrates the log-scaled execution times of the four computational graphs under the same experimental configurations described in Section 4.4.2. The extent of the non-null bands varies from 0 (diagonal tensors) to 1023 (fully dense tensors). The optimized **Baseline-ikj** baseline outperforms the standard **Baseline** configuration, not only due to superior cache locality but also because its memory access pattern enables more extensive vectorization by the compiler. Nevertheless, all BPA-aware kernels consistently outperform both baselines when operating on tensors with large null bands. The underlying reasons for this enhanced performance vary across configurations, and with the exception of the **Dense** and **hybrid** variants, this competitive advantage diminishes as tensor density increases.

The **Dense** and **Compressed** kernels both restrict their operations exclusively to non-null bands. The **Dense** kernel reduces the asymptotic time complexity of a naive matrix multiplication from $\mathcal{O}(N^3)$ to $\mathcal{O}(w^2N)$, where $w = (p + q + 1)$ represents the total bandwidth, and p and q denote the lower and upper non-null extents, respectively. Concurrently, the **Compressed** kernel leverages a custom diagonal-by-diagonal matrix multiplication routine that writes the computational results directly into a compressed storage scheme during execution. The **Compressed-Inputs** con-

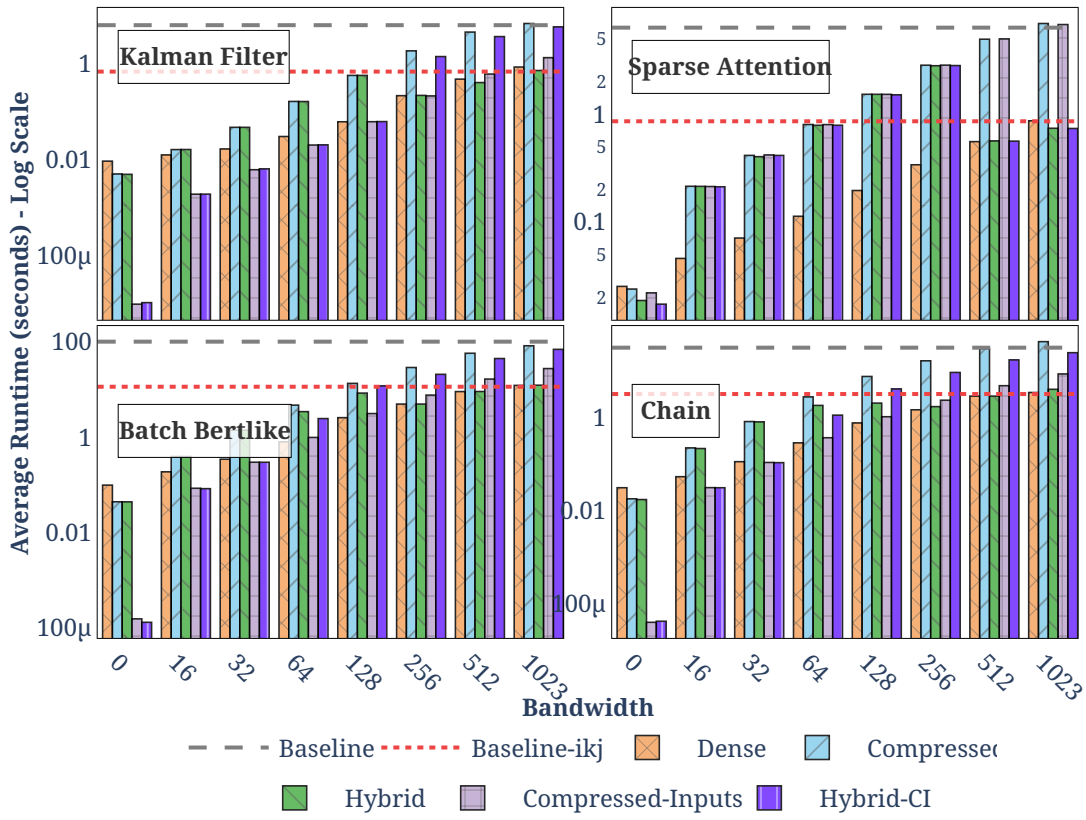


Figure 4.11: Log-scaled execution time comparison of the bandedness-aware compiler using DIA and dense formats against naive MLIR code (**Baseline**) and MLIR configured with an i-k-j matrix-multiplication loop ordering (**Baseline-ikj**). Each value on the x-axis represents a different bandwidth parameter for the input tensors of the graph.

figuration amplifies these performance gains further because both the intermediate tensors and the initial input tensors utilize a compact, low-footprint representation.

Hybrid is beneficial in most scenarios, often matching the best configurations. The decision of which format **hybrid** uses is based on memory consumption (as seen in Section 4.4.2) rather than execution speed. Consequently, at very high tensor densities, regressions can be observed. Also, the **hybrid-dia-inputs** kernel can exhibit performance regressions relative to the optimized **Baseline-ikj** baseline in dense settings because the input tensors are diagonalized. These performance penalties would be mitigated by tuning the code generator's optimization heuristic to use execution time as its primary objective function instead of storage space.

4.4.4 RQ4: Compilation Overhead

Section 4.4.1 evaluated the asymptotic behavior of BPA. This section enriches that study with absolute runtime numbers to show that BPA accounts for a small portion of the time spent to compile a typical computational graph. Furthermore, it is constant per computational graph, regardless of the size of the tensors, in contrast to the time to run that computational graph, which increases as the input tensors grow.

Discussion: Figure 4.12 compares the execution time of the static analysis (orange) against the standard compilation time (blue) and the graph execution time (green) using computational graphs configured with different initial bandwidth values to generate **Dense** kernels. Note that the reported compilation time strictly excludes the analysis overhead. The first four benchmarks plotted on the x-axis correspond to the workloads evaluated in the previous experiments. The remaining entries represent synthetically generated random computational graphs, with operator counts scaling from 20 to 200. These synthetic operators are drawn from a uniform random distribution of matrix multiplications, additions, subtractions, transpositions, and Hadamard

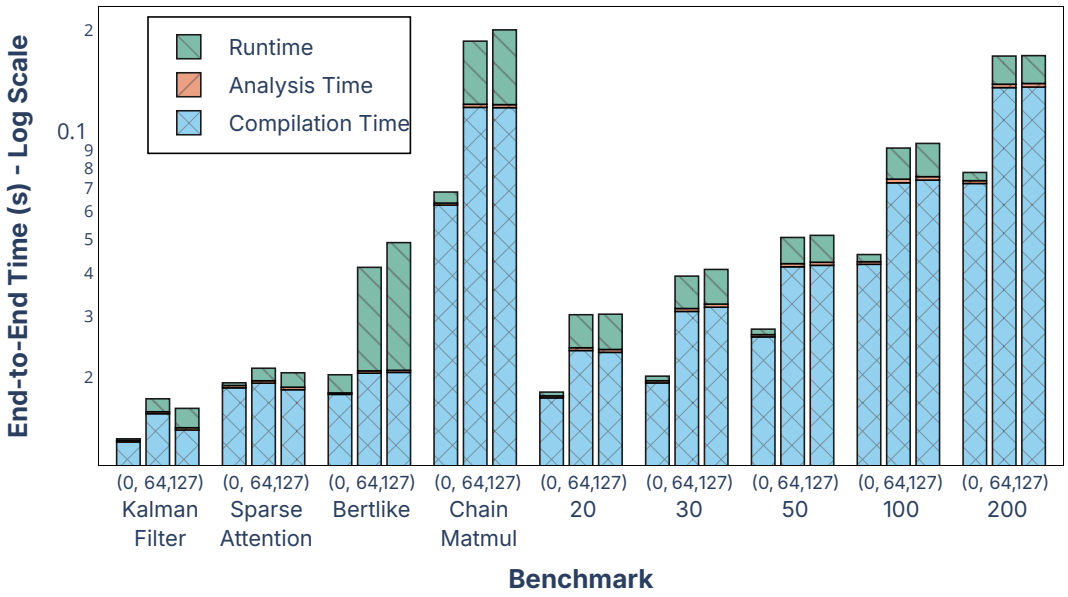


Figure 4.12: The execution time of BPA compared against the standard compilation and execution times of different computational graphs, evaluated across initial bandwidth values of 0, 64, and 127 (from left to right).

products.

Figure 4.12 evaluates these workloads using 128×128 tensors, and Figure 4.13 uses tensors of different sizes. Each benchmark reports three distinct bars corresponding to initial input bandwidth values of 0, 64, and 127 (ordered from left to right). A bandwidth of 0 leads to faster compilation and execution times, as the backend code generator simplifies the general matrix operations into optimized, one-dimensional vector computations along the active diagonals. Figure 4.13 shows that even increasing the size of the matrices (up to 2048), and using the smallest bandwidth value, the analysis still remains constant time, and is much faster than evaluating the computational graph.

4.4.5 RQ5: Impact of Multi-Propagation

The bandedness propagation analysis transfers abstract states along two directions: forward (Sec. 4.2.2) and backward (Sec. 4.2.3). Backward propagation provides addi-

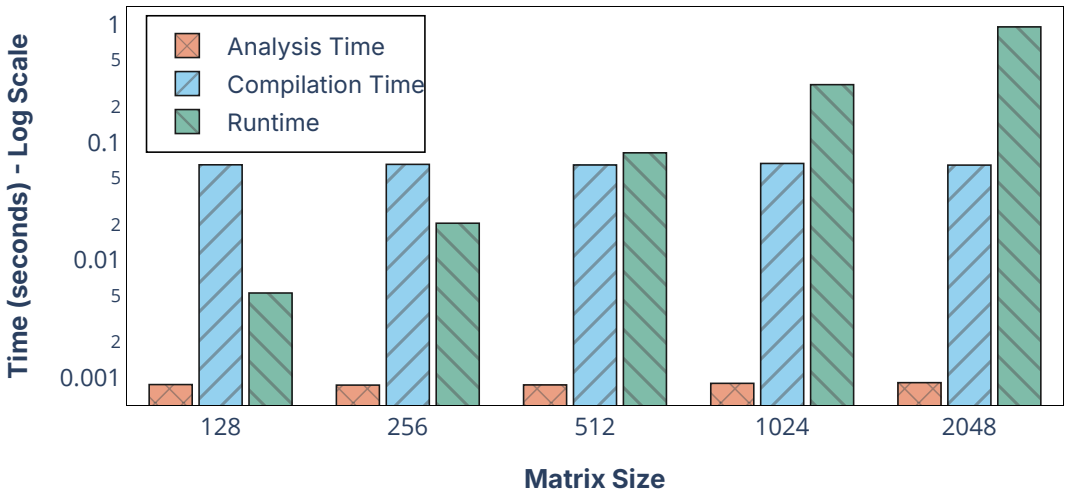


Figure 4.13: The BPA execution time compared to the compilation, and runtime of the Chain benchmark using different matrix sizes at zero bandwidth.

tional information beyond forward propagation whenever computational graphs contain operations with annihilating elements (Sec. 2.3.1). Masked attention is one such example. This section evaluates the impact of backward propagation in this setting.

Discussion: Figure 4.14 compares the sparsity inferred during the analysis of masked attention under different initial non-null bandwidths. The sparsity ratio is the ratio between the number of zero elements and the total number of elements. Bars labeled "*Initial*" show the sparsity before the analysis is executed. In this case, only the initial annotations determine null bands. LBW and UBW denote the lower and upper bandwidths of the input tensors, respectively.

In this benchmark, forward propagation more than doubles the extent of bands known to be null. This result is expected: the computational graph contains three input tensors, and four intermediate tensors analyzable via forward propagation. Backward propagation has a smaller, yet still significant, effect. It further increases the overall sparsity, in some cases more than tripling the extent of the null bands.

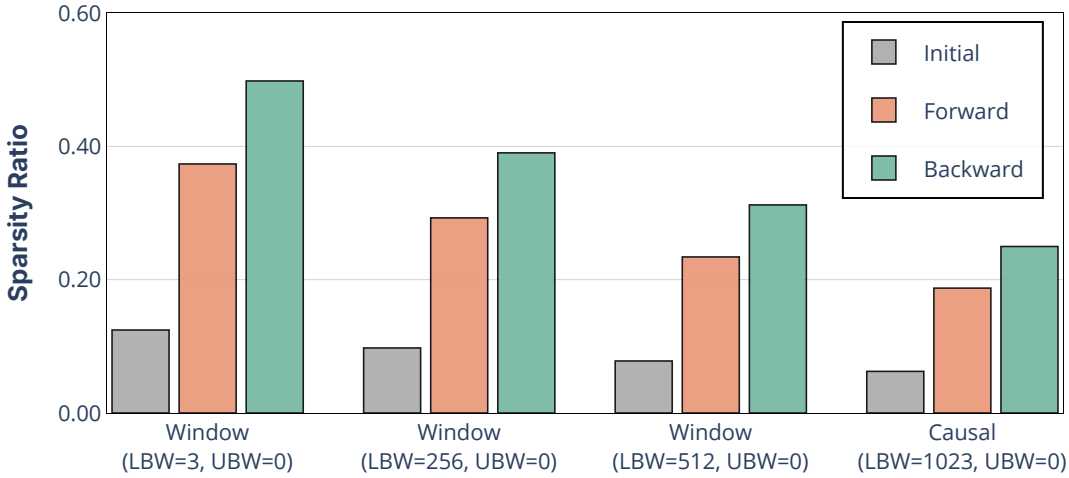


Figure 4.14: Sparsity ratio (irrelevant/total elements) of tensors in masked attention under different masks, before and after each step of the propagation analysis.

4.5 Conclusion

This part of the thesis introduced Bandedness Propagation Analysis (BPA), a static analysis that infers banded-diagonal invariants in computational graphs. BPA represents multidimensional tensors through collections of 2D projections and propagates band information using forward and backward transfer functions over common tensor operations, including contractions, transpositions, element-wise operators, and reductions. We implemented BPA in MLIR and showed that the analysis introduces only a small compilation overhead but enables optimizations that reduce memory consumption and execution time of computational graphs. An interesting direction for future work is extending BPA to support a broader range of tensor operators by systematically decomposing complex operations into simpler algebraic equations that can be analyzed using the transfer functions introduced in this part of the thesis. Such an approach could expand the applicability of the analysis while preserving its simplicity and efficiency.

Chapter 5

Related Work

The exploitation of sparsity in tensors has received significant attention in recent years [17, 20, 37, 60, 69]. While some state-of-the-art systems such as StarDust [30] and SysTeC [53], both released in 2025, rely on users to identify sparse inputs and choose formats based on operand and kernel types, there are many static analyses that infer sparsity in computational graphs. Some of these analyses are restricted to particular operations, such as Lopez et al. [46], which propagates symmetry and triangularity over chains of matrix multiplications. Other techniques are restricted to two-dimensional tensors, such as the pioneering work of Sommer et al. [61]'s Matrix Non-Zero Count (MNC), which estimates zeros in 2D matrices. Some analyses are designed without guarantees of soundness, such as the heuristic proposed by Yan et al. [70], which associates each dimension of a tensor with one of two abstract states: Sparse or Dense, and assume that operations such as addition of sparse data preserves sparsity. Nevertheless, there are a number of static analyses that, like SPA and BPA, generalize to multidimensional tensors and to different kinds of tensor operations. Figure 5.1 illustrates their differences.

There are a number of static analyses that infer elements that must be zero in any execution of a program [25, 42, 46]. However, only StructTensor, from Ghorbani et al. [25], generalizes to tensors. StructTensor represents tensor regions as sums of

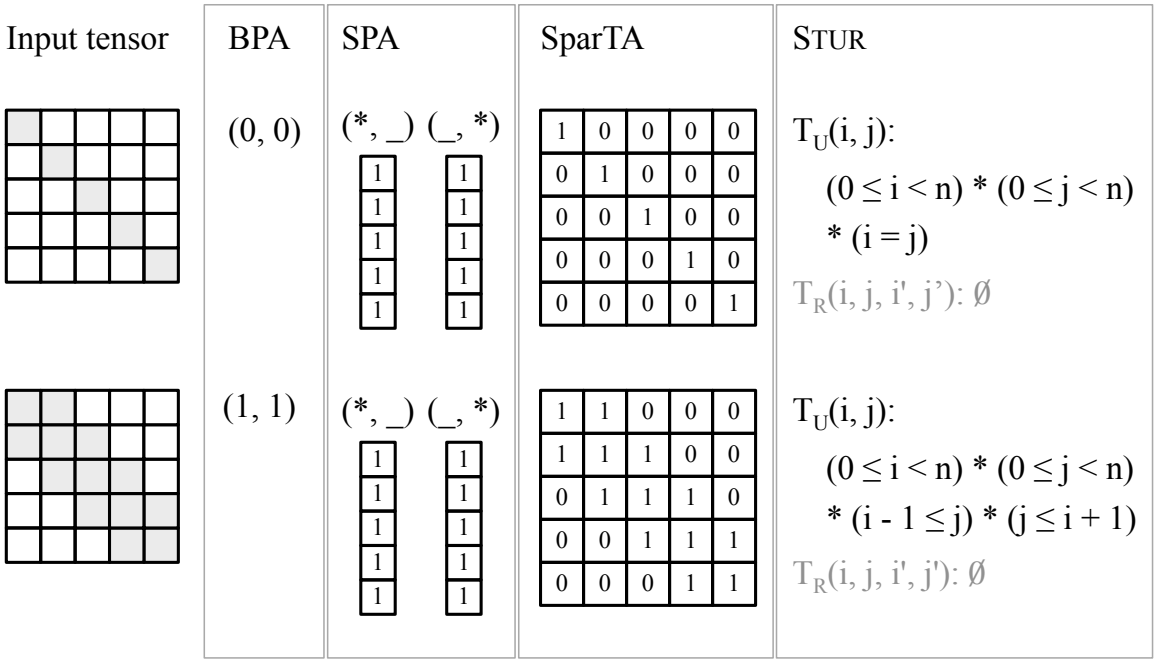


Figure 5.1: Examples of abstract states used by sparsity propagation analyses. These analyses operate on acyclic computational graphs involving tensors of arbitrary dimensionality. StructTensor (labeled STUR) supports only forward propagation, whereas the other analyses also support backward propagation.

products of inequalities, as illustrated in Figure 5.1[1] Non-null regions are annotated by the user and propagated via symbolic rules. By restricting regions to sums of products, the symbolic interpreter can compose regions via union, intersection, and projection without resorting to the expensive integer-linear programming techniques used in the polyhedral model. However, this design has a cost: the representation of non-null elements may contain overlapping regions, which can grow large depending on the bootstrapping annotations. Furthermore, even without overlap, sums of products can grow quickly. For instance, a matrix in which only the last band is null would require one inequality per band, except for the last.

SparTA [74], a deep learning accelerator framework built around the *Tensor-with-*

[1]StructTensor also propagates redundancy due to symmetricity. This kind of propagation is unrelated to this thesis; hence, we do not discuss it.

Sparsity-Attribute (TeSA) abstraction, tracks sparsity at the granularity of individual tensor elements, associating each with a boolean. In order to propagate sparsity, they perform algebraic operations with the TeSA tensors, which enable forward, backward, and even lateral propagation, The TeSA abstraction was first introduced by Zheng *et al.* [76] and later rediscovered by Huang *et al.* [33], who also extended it with propagation rules for operations such as pooling. Although neither Zheng et al. [76] nor Huang et al. [33] state a formal soundness property, we believe that their guarantees align with those in this paper: elements marked "zero" do not contribute to the evaluation of the computational graph; hence, can be treated as actual zeros.

Different implementations of SparTA have been developed for ahead-of-time compilation [20, 33, 75]. However, because the abstraction is defined at element-level granularity, the approach scales poorly in just-in-time scenarios. As shown in Section 3.2.1 and Section 4.4.1, the asymptotic complexities of both SPA and BPA are orders of magnitude faster than SparTA in evaluating single operations and entire computational graphs (see Figure 3.6 and Figure 4.9).

MNC adopts a coarser abstract domain that estimates lower and upper bounds on the number of zeros per slice of a tensor. This design is well-suited for applications such as query optimization in databases [52], and has been demonstrated to support specialization of tensors in compilers [12]. However, the analysis supports only forward propagation and has been evaluated only for matrices (2D tensors). Its complexity lies between element-wise approaches such as SparTA and the slice- and band-level approaches covered in this thesis.

An intermediate abstraction between SparTA and MNC appears in FlexAttention [18], a compiler designed for efficient implementation of attention mechanisms. Here, tensors are divided into blocks, each labeled as *fully masked* (all zeros), *partial*, or *full* (all nonzeros). This abstraction has the asymptotic complexity of SparTA

but reduces constant factors by grouping elements, while regaining some precision through row-wise nonzero counts inspired by MNC. Dong *et al.* [18] show that block masks are effective for self-attention, especially under structured patterns such as causal or sliding-window masks. However, the technique is narrowly specialized: it applies only to the intermediate score matrix of attention kernels, supports only forward propagation, and generalizes poorly beyond attention workloads.

As discussed throughout the thesis, SPA and BPA apply to any semiring. The idea that semiring-based operations enable compiler optimizations is not new. Leobas et al. [42] showed how to guard chains of operations that can collapse due to the presence of absorbing elements. Subsequently, Stephenson et al. [64, 65] demonstrated how such optimizations accelerate video games running on graphics processing units, and You et al. [72] extended these ideas to domains such as the log semiring, i.e., $(\mathbb{R} \cup \{-\infty\}, \mathsf{log\text{-}add}, \mathsf{log\text{-}mul}, -\infty, 0)$. However, none of this prior work developed a static analysis for computational graphs. Leobas [42] or You [72], for instance, relied on standard dataflow analyses that associate scalar values (rather than tensor slices) with zero or nonzero. As a result, backward and lateral propagation were not meaningful in their setting.

Chapter 6

Conclusion

This thesis introduces two novel data-flow analyses, Sparsity Propagation Analysis (SPA) and Bandedness Propagation Analysis (BPA), for propagating sparsity through computational graphs in order to enable sparsity optimizations.

Chapter 3 shows the efficacy of SPA. The analysis operates over a compact domain, enabling a highly efficient implementation that outperforms similar approaches in symbolic execution time. SPA is shown to reduce both runtime and memory in a variety of domains, with the Einsum benchmark suite [8] showing its generality and the BERT-like benchmark showing its practical efficacy.

Chapter 4 shows the efficacy of BPA in the same ways. Its domain enables a highly efficient implementation, beating state-of-the-art in symbolic execution time. BPA's practical applicability is shown through its ability to optimize domain-specific benchmarks like Sparse Attention and the Kalman Filter.

There are several directions for future work. The first is with regard to the application of these analyses. Though both analyses are implemented as ahead-of-time (AOT) compiler passes, their efficient symbolic execution and lightweight abstract domains make them strong candidates for passes in just-in-time (JIT) compilation as well. The second concerns scope. SPA and BPA work under the assumption that the underlying computational graphs have no control flow. This assumption was made

largely to keep the scope of the works tightly focused, and through relaxing this assumption and the one-pass convergence assumption, these analyses could be extended to cover a broader domain.

BPA specifically has one additional interesting direction. Currently, the analysis operates on 2D projections of N-D tensors. However, another approach is to instead have the analysis operate on matrix-sized slices: for a tensor $A \in \mathbb{R}^{m \times n \times p}$, instead of assigning a lattice value to the 2D projection $A[:, :, k]$, each slice would have its own, allowing $A[:, :, 0]$ to differ from $A[:, :, 1]$. This would lead to finer granularity of information to exploit for further optimization, but at the cost of a more complex abstraction.

Bibliography

[1] J. Ansel et al., "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *ASPLOS*, La Jolla, CA, USA: ACM, 2024, pp. 929–947, ISBN: 9798400703850. DOI: 10.1145/3620665.3640366. [Online]. Available: https://doi.org/10.1145/3620665.3640366. 35

[2] P. Arbenz, "Computing eigenvalues of banded symmetric toeplitz matrices," *SIAM J. Sci. Stat. Comput.*, vol. 12, no. 4, pp. 743–754, Jul. 1991, ISSN: 0196-5204. 47

[3] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018. 14

[4] T. Baroudi, R. Seghir, and V. Loechner, "Optimization of triangular and banded matrix operations using 2d-packed layouts," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, Dec. 2017, ISSN: 1544-3566. DOI: 10.1145/3162016. [Online]. Available: https://doi.org/10.1145/3162016. 44

[5] B. Beckermann, D. Kressner, and H. Wilber, "Compression properties for large toeplitz-like matrices," *Numer. Algorithms*, vol. 100, no. 4, pp. 2041–2067, Jul. 2025, ISSN: 1017-1398. DOI: 10.1007/s11075-025-02185-8. [Online]. Available: https://doi.org/10.1007/s11075-025-02185-8. 44

[6] D. Bini and M. Capovani, "Spectral and computational properties of band symmetric toeplitz matrices," *Linear Algebra and its Applications*, vol. 52, pp. 99–126, 1983. 47

[7] M. Blacher et al., "Einsum benchmark: Enabling the development of next-generation tensor execution engines," *Advances in Neural Information Processing Systems*, vol. 37, pp. 98 033–98 048, 2024. 33

[8] M. Blacher et al., "Einsum benchmark: Enabling the development of next-generation tensor execution engines," *Advances in Neural Information Processing Systems*, vol. 37, pp. 98 033–98 048, 2024. 82

[9] M. Blondel and V. Roulet, *The elements of differentiable programming*, 2025. arXiv: 2403.14606 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2403.14606. 4

[10] M. Canesche, V. M. do Rosario, E. Borin, and F. M. Quintão Pereira, "Fusion of operators of computational graphs via greedy clustering: The xnnc experience," in *Compiler Construction*, ser. CC '25, Las Vegas, NV, USA: ACM, 2025, pp. 117–127, ISBN: 9798400714078. DOI: 10.1145/3708493.3712689. [Online]. Available: https://doi.org/10.1145/3708493.3712689. 14

[11] T. Chen et al., "Tvm: An automated end-to-end optimizing compiler for deep learning," in *OSDI*, Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594, ISBN: 9781931971478. 14

[12] Z. Chen, B. Han, C. Xu, W. Qian, and A. Zhou, "Redundancy elimination in distributed matrix computation," in *SIGMOD*, Philadelphia, PA, USA: ACM, 2022, pp. 573–586, ISBN: 9781450392495. DOI: 10.1145/3514221.3517877. [Online]. Available: https://doi.org/10.1145/3514221.3517877. 80

[13] H. Cui, Q. Yi, J. Xue, and X. Feng, "Layout-oblivious compiler optimization for matrix computations," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, Jan. 2013, ISSN: 1544-3566. DOI: 10.1145/2400682.2400694. [Online]. Available: https://doi.org/10.1145/2400682.2400694. 44

[14] S. Cyphers et al., *Intel nGraph: An intermediate representation, compiler, and executor for deep learning*, 2018. 14

[15] P. Davoodi, C. Gwon, G. Lai, and T. Morris, *Tensorrt inference with tensorflow*, 2019. 14

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *BERT: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: 1810.04805 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1810.04805. 33

[17] A. Dias, K. Sundararajah, C. Saumya, and M. Kulkarni, "SparseLNR: Accelerating sparse tensor computations using loop nest restructuring," in *ICS*, Virtual Event: ACM, 2022, ISBN: 9781450392815. DOI: 10.1145/3524059.3532386. [Online]. Available: https://doi.org/10.1145/3524059.3532386. 78

[18] J. Dong, B. Feng, D. Guessous, Y. Liang, and H. He, *Flex attention: A programming model for generating optimized attention kernels*, 2024. arXiv: 2412.05496 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2412.05496. 1, 47, 80, 81

[19] H. Eavani, T. D. Satterthwaite, R. Filipovych, R. E. Gur, R. C. Gur, and C. Davatzikos, "Identifying sparse connectivity patterns in the brain using resting-state fmri," *Neuroimage*, vol. 105, pp. 286–299, 2015. 44

[20] R. Fan et al., "SpInfer: Leveraging low-level sparsity for efficient large language model inference on gpus," in *EuroSys*, Rotterdam, Netherlands: ACM, 2025, pp. 243–260, ISBN: 9798400711961. DOI: 10.1145/3689031.3717481. [Online]. Available: https://doi.org/10.1145/3689031.3717481. 78, 80

[21] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *arXiv preprint arXiv:1902.09574*, 2019. 15

[22] G. C. Garriga, E. Junttila, and H. Mannila, "Banded structure in binary matrices," in *KDD*, Las Vegas, Nevada, USA: Association for Computing Machinery, 2008, pp. 292–300, ISBN: 9781605581934. DOI: 10.1145/1401890.1401929. [Online]. Available: https://doi.org/10.1145/1401890.1401929. 44, 47

[23] Z. Geng, M.-H. Guo, H. Chen, X. Li, K. Wei, and Z. Lin, *Is attention better than matrix decomposition?* 2021. arXiv: 2109.04553 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2109.04553. 44

[24] M. Ghorbani, E. Bauer, T. Grosser, and A. Shaikhha, "Compressed and parallelized structured tensor algebra," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA1, Apr. 2025. DOI: 10.1145/3720506. [Online]. Available: https://doi.org/10.1145/3720506. 44, 68

[25] M. Ghorbani, M. Huot, S. Hashemian, and A. Shaikhha, "Compiling structured tensor algebra," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, 229:204–229:233, Oct. 2023. DOI: 10.1145/3622804. iii, 46, 68, 78

[26] Google, *Xla overview*, https://openxla.org/xla, 2024. 14

[27] A. Gupta et al., "Splat: A framework for optimised gpu code-generation for sparse regular attention," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA1, Apr. 2025. DOI: 10.1145/3720503. [Online]. Available: https://doi.org/10.1145/3720503. 44

[28] S. Han, S. Kim, G. Byeon, J. Yoon, and S. Hong, "Zebra: Leveraging diagonal attention pattern for vision transformer accelerator," in *DATE*, New York, USA: IEEE, 2025, pp. 1–7. 44

[29] R. Henry et al., "Compilation of sparse array programming models," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. DOI: 10.1145/3485505. [Online]. Available: https://doi.org/10.1145/3485505. 44

[30] O. Hsu, A. Rucker, T. Zhao, V. Desai, K. Olukotun, and F. Kjolstad, "Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture," in *CGO*, Las Vegas, NV, USA: ACM, 2025, pp. 628–643, ISBN: 9798400712753. DOI: 10.1145/3696443.3708918. [Online]. Available: https://doi.org/10.1145/3696443.3708918. 78

[31] O. Hsu et al., "The sparse abstract machine," in *ASPLOS*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 710–726, ISBN: 9781450399180. DOI: 10.1145/3582016.3582051. [Online]. Available: https://doi.org/10.1145/3582016.3582051. 44

[32] C. Huang et al., "Efficient parallelization of tensor network contraction for simulating quantum computation," *Nature Computational Science*, vol. 1, no. 9, pp. 578–587, 2021. 34

[33] S. Huang, F. Liu, T. Yang, Z. Wang, N. Yang, and L. Jiang, "SpMMPlu-Pro: An enhanced compiler plug-in for efficient spmm and sparsity propagation algorithm," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 44, no. 2, pp. 669–683, Feb. 2025, ISSN: 0278-0070. DOI: 10.1109/TCAD.2024.3446718. [Online]. Available: https://doi.org/10.1109/TCAD.2024.3446718. 80

[34] V. Jampani, M. Kiefel, and P. V. Gehler, "Learning sparse high dimensional filters: Image filtering, dense CRFs and bilateral neural networks," in *CVPR*, New York, USA: IEEE, 2016, pp. 4452–4461. 15

[35] R. E. Kalman, "A new approach to linear filtering and prediction problems," *J. Basic Eng.*, vol. 82, no. 1, pp. 35–45, 1960. 66

[36] I. Khalfaoui-Hassani, *Dilated convolution with learnable spacings*, 2024. arXiv: 2408.06383 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2408.06383. 47

[37] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. DOI: 10.1145/3133901. [Online]. Available: https://doi.org/10.1145/3133901. ii, 2, 16, 33, 78

[38] S. E. Kurt, S. Raje, A. Sukumaran-Rajam, and P. Sadayappan, "Sparsity-aware tensor decomposition," in *IPDPS*, USA: IEEE, 2022, pp. 952–962. DOI: 10.1109/IPDPS53621.2022.00097. 44

[39] R. Lacouture et al., "Fuseflow: A fusion-centric compilation framework for sparse deep learning on streaming dataflow," in *ASPLOS*, USA: Association for Computing Machinery, 2026, pp. 798–820, ISBN: 9798400723599. DOI: 10.1145/3779212.3790165. [Online]. Available: https://doi.org/10.1145/3779212.3790165. 44

[40] C. Lattner et al., "Mlir: Scaling compiler infrastructure for domain specific computation," in *CGO*, New York, US: IEEE Press, 2021, pp. 2–14, ISBN: 9781728186139. DOI: 10.1109/CGO51591.2021.9370308. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370308. 45

[41] D. Lenadora, V. Sathia, G. Gerogiannis, S. Yesil, J. Torrellas, and C. Mendis, "Granii: Selection and ordering of primitives in graph neural networks using input inspection," in *CGO*, New York: IEEE, 2026, pp. 14–27. DOI: 10.1109/CGO68049.2026.11395219. 67

[42] G. V. Leobas and F. M. Q. Pereira, "Semiring optimizations: Dynamic elision of expressions with identity and absorbing elements," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428199. [Online]. Available: https://doi.org/10.1145/3428199. 78, 81

[43] H. Li, Z. Li, Z. Bai, and T. Mitra, "Asadi: Accelerating sparse attention using diagonal-based in-situ computing," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, New York, USA: IEEE, 2024, pp. 774–787. 44

[44] M. Li et al., "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020. 14

[45] G. U. Llauset, *Can every index be sparse?* https://github.com/tensor-compiler/taco/issues/537, tensor-compiler/taco repository, 2023. Accessed: Sep. 2, 2025. 33

[46] F. Lopez, L. Karlsson, and P. Bientinesi, "Compilation of generalized matrix chains with symbolic sizes," in *CGO*, New York: IEEE, 2026, pp. 466–478. DOI: 10.1109/CGO68049.2026.11395236. 78

[47] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 9, no. 4, e1305, 2019. 14

[48] Microsoft, *Onnx runtime*, https://onnxruntime.ai/, 2024. 14

[49] A. MxNet, *Apache MxNet: A flexible and efficient library for deep learning*, https://mxnet.apache.org/, 2024. 14

[50] M. Naumov, *Parallel complexity of forward and backward propagation*, 2017. arXiv: 1712.06577 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1712.06577. 47

[51] N. Nayak, X. Wu, T. O. Odemuyiwa, M. Pellauer, J. S. Emer, and C. W. Fletcher, "FuseMax: Leveraging extended einsums to optimize attention accelerator design," in *MICRO*, IEEE, 2024, pp. 1458–1473. 5

[52] D. Olteanu, N. Vortmeier, and u. Zivanović, "Givens QR decomposition over relational databases," in *SIGMOD*, Philadelphia, PA, USA: ACM, 2022, pp. 1948–1961, ISBN: 9781450392495. DOI: 10.1145/3514221.3526144. [Online]. Available: https://doi.org/10.1145/3514221.3526144. 80

[53] R. Patel, W. Ahrens, and S. Amarasinghe, "Systec: A symmetric sparse tensor compiler," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 47–62, ISBN: 9798400712753. DOI: 10.1145/3696443.3708919. [Online]. Available: https://doi.org/10.1145/3696443.3708919. 78

[54] S. Raje, Y. Xu, A. Rountev, E. F. Valeev, and P. Sadayappan, "Const: Code generator for sparse tensor networks," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 4, Nov. 2024, ISSN: 1544-3566. DOI: 10.1145/3689342. [Online]. Available: https://doi.org/10.1145/3689342. 44, 45

[55] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *HPEC*, Waltham, MA, USA: IEEE, 2019, pp. 1–9. DOI: 10.48550/arXiv.1908.11348. 14

[56] N. Rotem et al., *Glow: Graph lowering compiler techniques for neural networks*, 2018. 14

[57] Y. Saad, "Sparskit: A basic tool kit for sparse matrix computations," NASA Ames Research Center, Tech. Rep., 1990. 7, 67

[58] S. Serra, "Spectral and computational analysis of block toeplitz matrices having nonnegative definite matrix-valued generating functions," *BIT*, vol. 39, no. 1, pp. 152–175, Mar. 1999, ISSN: 0006-3835. DOI: 10.1023/A:1022329526925. [Online]. Available: https://doi.org/10.1023/A:1022329526925. 47

[59] D. G. Smith and J. Gray, *Opt_einsum-a python package for optimizing contraction order for einsum-like expressions. j. open source softw. 3, 26 (2018), 753*, 2018. 33

[60] E. Solomonik and T. Hoefler, *Sparse tensor algebra as a parallel programming model*, Nov. 2015. arXiv: 1512.00066 [cs]. 78

[61] J. Sommer, M. Boehm, A. V. Evfimievski, B. Reinwald, and P. J. Haas, "MNC: Structure-exploiting sparsity estimation for matrix expressions," in *SIGMOD*, Amsterdam, Netherlands: ACM, 2019, pp. 1607–1623, ISBN: 9781450356435. DOI: 10.1145/3299869.3319854. [Online]. Available: https://doi.org/10.1145/3299869.3319854. 14, 44, 78

[62] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler for structured matrices," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*, New York, NY, USA: Association for Computing Machinery, 2016, pp. 117–127. DOI: 10.1145/2854038.2854060. 44

[63] S. Srinivas, A. Subramanya, and R. V. Babu, *Training sparse neural networks*, 2016. arXiv: 1611.06694 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1611.06694. 15

[64] M. Stephenson and R. Rangan, "PGZ: Automatic zero-value code specialization," in *CC*, Virtual, Republic of Korea: ACM, 2021, pp. 36–46, ISBN: 9781450383257. DOI: 10.1145/3446804.3446845. [Online]. Available: https://doi.org/10.1145/3446804.3446845. 81

[65] M. W. Stephenson and R. Rangan, *AZP: Automatic specialization for zero values in gaming applications*, 2020. arXiv: 2011.10550 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2011.10550. 81

[66] R. Tian, L. Guo, J. Li, B. Ren, and G. Kestor, "A high performance sparse tensor algebra compiler in mlir," in *LLVM-HPC*, 2021, pp. 27–38. DOI: 10.1109/LLVMHPC54804.2021.00009. 35

[67] A. Vaswani et al., "Attention is all you need," in *NIPS*, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010, ISBN: 9781510860964. 47

[68] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016. 15

[69] K. W. Wu, "Accelerating sparse graph neural networks with tensor core optimization," *arXiv preprint arXiv:2412.12218*, 2024. 78

[70] B. Yan, A. J. Root, T. Gale, D. Broman, and F. Kjolstad, "Fast autoscheduling for sparse ml frameworks," in *CGO*, New York: IEEE, 2026, pp. 28–43. 44, 78

[71] W. Yang, K. Li, Y. Liu, L. Shi, and L. Wan, "Optimization of quasi-diagonal matrix–vector multiplication on gpu," *The international journal of high performance computing applications*, vol. 28, no. 2, pp. 183–195, 2014. 44

[72] X. You, H. Yang, K. Lei, Z. Luan, and D. Qian, "TrivialSpy: Identifying software triviality via fine-grained and dataflow-based value profiling," in *SC*, Denver, CO, USA: ACM, 2023, ISBN: 9798400701092. DOI: 10.1145/3581784.3607052. [Online]. Available: https://doi.org/10.1145/3581784.3607052. 81

[73] L. Zhao, S. Liao, Y. Wang, Z. Li, J. Tang, and B. Yuan, "Theoretical properties for neural networks with weight matrices of low displacement rank," in *ICML*, Sydney, NSW, Australia: Microtome Publishing, 2017, pp. 4082–4090. 47

[74] N. Zheng et al., "SparTA: Deep-learning model sparsity via tensor-with-sparsity-attribute," in *OSDI*, Berkeley, USA: USENIX Association, 2022, pp. 213–232. ii, iii, 1, 14, 44–46, 51, 68, 79

[75] S. Zheng et al., "Chimera: An analytical optimizing framework for effective compute-intensive operators fusion," in *HPCA*, New York, US: IEEE, 2023, pp. 1113–1126. DOI: 10.1109/HPCA56546.2023.10071018. [Online]. Available: https://doi.org/10.1109/HPCA56546.2023.10071018. 80

[76] Z. Zheng et al., "Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures," in *ASPLOS*, Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 359–373, ISBN: 9781450392051. DOI: 10.1145/3503222.3507723. [Online]. Available: https://doi.org/10.1145/3503222.3507723. 80

[77] K. Zhong et al., "FEASTA: A flexible and efficient accelerator for sparse tensor algebra in machine learning," in *ASPLOS*, La Jolla, CA, USA: ACM, 2024, pp. 349–366, ISBN: 9798400703867. DOI: 10.1145/3620666.3651336. [Online]. Available: https://doi.org/10.1145/3620666.3651336. 6