Bandedness Propagation Analysis of Tensor Compilers

Danila Seliayeu*
University of Alberta
Edmonton, Alberta, Canada
seliayeu@ualberta.ca

Kaio Henrique Ananias*
UFMG
Belo Horizonte, Minas Gerais, Brazil
kaiohenrique@dcc.ufmg.br

José Nelson Amaral
University of Alberta
Edmonton, Alberta, Canada
jamaral@ualberta.ca

Fernando M. Quintão Pereira
UFMG
Belo Horizonte, Minas Gerais, Brazil
fernando@dcc.ufmg.br

Abstract

In computations involving attention mechanisms, temporal dependencies, and truncated backpropagation, tensor slices may contain nonzero elements only within a bounded distance from the main diagonal. Operations on such tensors admit efficient implementations via specialized kernels; however, no static analysis currently infers these bands. This paper introduces *Bandedness Propagation Analysis* (BPA) to address this limitation. BPA decomposes multidimensional tensors into collections of 2D banded projections, each abstracting the non-null bands along two dimensions of the original tensor. We define an abstract domain and multidirectional transfer functions that propagate band information across tensor operations, including contractions, transpositions, and element-wise operators, over arbitrary semirings. We implement BPA in MLIR and establish its soundness. Our evaluation shows that the analysis runs in a fraction of the time required to compile or execute a computational graph. Furthermore, depending on the benchmark and the level of sparsity, BPA enables the generation of kernels with asymptotically better memory usage and execution time.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: Tensor compiler, static analysis, banded matrix

1 Introduction

A matrix A is banded if $A[i, j] = 0$ whenever $|i - j| > k$ for some small band radius k [12]. 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) [13, 16, 17, 30], convolutional layers (which can be seen as banded Toeplitz operators [5]), and sparse connectivity patterns designed to reduce complexity [11] (See Sec. 2 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, 9, 14, 16, 19, 20, 25, 26, 35, 39, 41, 42]. And there exist static analyses that propagate sparsity information through computational graphs [2, 38, 45]. However, they are oblivious to the structure of non-diagonal sparsity. The work of Zheng et al. [45] 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 Ananias et al. [2] 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 Section 5 explains, this analysis misses diagonal information: a single nonzero diagonal element taints every slice. In this paper, we propose a new analysis that succeeds where Ananias et al.'s fails, while being asymptotically more efficient than Zheng et al.'s.

Contributions of this Work. This paper 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 2.10 on Page 3) and associates each 2D substructure with an interval of *non-null bands* projected along those dimensions. Figure 1 illustrates this idea.

As Figure 1 shows, each banded matrix overapproximates the non-null bands of a tensor along two dimensions. Section 3 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. The paper defines these transfer functions for common tensor operations, including `einsum` contractions (which subsume matrix multiplication [35]), transpositions, additions, and clamping.

*Both authors contributed equally to this work.

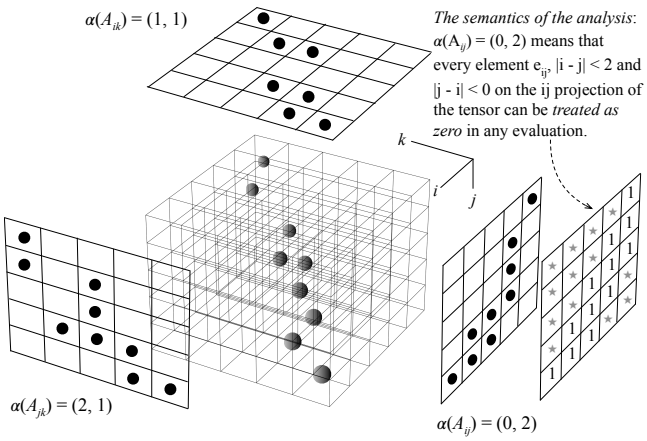

Figure 1. A decomposition of a 3D tensor into three 2D projections with the respective abstract states. The abstract information α associated with an n-dimensional tensor A is a table with $n \times (n-1)/2$ entries. Each entry is a pair (l_{ij}, u_{ij}) over-approximating the lower extent of A_{ij} (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).

As discussed in Section 2.1, 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 [27], defining transfer functions for operations in the `linalg` dialect, including multiplication, addition, transposition, and batch matrix multiplication. Section 4.1 shows that BPA is asymptotically faster than previous sparsity inference analyses, such as those proposed by Ghorbani et al. [15] and Zheng et al. [45], although it computes less detailed information. Section 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 4.3 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 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 4.5 demonstrates empirically that backward propagation of abstract states can improve the precision of forward propagation by up to 33% when analyzing masked attention kernels.

2 Preliminary Definitions

Already in 1990, Saad observed that "The matrices that arise in many applications often consist of a few diagonals" [36]. The occurrence of such data structures has since been studied extensively [12]. Examples include:

- **Linear Correlations:** The more linearly correlated two variables are, the more likely their representation in correlation space is to be concentrated around the main diagonal. For a visual example, see Figure 59.1 of Bird's book [7, Ch. 59].
- **Attention Mechanisms:** Attention scores frequently form lower-triangular or banded matrices due to causal masking or sliding-window restrictions [10, Fig. 1].
- **Temporal Dependencies:** Sequence models and recurrent computations induce triangular or banded structures because dependencies are constrained by causal ordering or finite temporal histories [40].
- **Convolutions:** Temporal and spatial convolutions induce banded Toeplitz-like structures when represented as matrices, since each output depends only on a local neighborhood of inputs [3, 6, 23, 37].
- **Truncated Backpropagation:** Backpropagation over time induces triangular dependency structures across time steps, which become banded when gradients are truncated to finite temporal windows [32, 44].

In fact, as Example 2.1 illustrates, banded matrices such as those considered in this paper are recurring motivating examples in academic work on sparse tensor computations.

Example 2.1. Several recent code-generation techniques for sparse tensors use banded matrices as motivating examples. Instances include Figure 7(a) in Henry et al. [19]'s work; Figure 1(a) in Hsu et al. [20]'s; Figures 2 and 3 in Li et al. [30]'s; Figure 1(d) in Huang et al. [21]'s; Figure 1 in Ahrens et al. [1]'s; Figure 2 in Gupta et al. [16]'s; and Figure 1 in Ghorbani et al. [14]'s. The analysis proposed in this paper is able to represent all of these structures.

2.1 Annihilating and Irrelevant Entries

Bandedness analysis approximates 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 backward pass identifies *irrelevant elements*.

2.1.1 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 2.2 (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 2.3. 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 $\text{true} \vee x = \text{true}$ and $\text{false} \wedge x = \text{false}$. 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 $x \geq 0$.

2.1.2 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 2.4 formalizes this notion.

Definition 2.4 (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 2.5. The *Hadamard product* of two tensors A and B of the same shape, denoted $C = A \odot B$, is defined elementwise 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 bandedness propagation analysis introduced in this paper approximates 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 2.6 (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.

As shown in Theorem 2.7, 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.

Theorem 2.7. *Bandedness Propagation Analysis is sound.*

Proof: Available in the supplementary material. □

2.2 Banded Matrices

A *banded matrix* has its nonzero entries confined to a diagonal band around the main diagonal, as per Definition 2.8. Entries sufficiently far from the diagonal contain zero. Definition 2.8 modifies this traditional definition to consider annihilable values, as per Definition 2.6, rather than zeros.

Definition 2.8 (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:

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

The integers p and q from Definition 2.8 are called the *lower* and *upper extents* of A, respectively. As Example 2.9 lists, several well-known matrix classes arise as special cases:

Example 2.9. 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 $*$.

2.3 Projections

A static analysis computes an *abstract environment*, i.e., a function $\alpha : \mathcal{E} \to \mathcal{D}$, where $\mathcal{E}$ is a set of *program entities* and $\mathcal{D}$ is an *abstract domain* (to be introduced in Section 2.4). In the static analysis proposed in this paper, the program entities are *2D-projections* of tensors, as defined below.

Definition 2.10 (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 $\mathbb{B} = \{\text{false}, \text{true}\}$ and $\vee$ denotes logical disjunction.

Example 2.11. Figure 2 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.

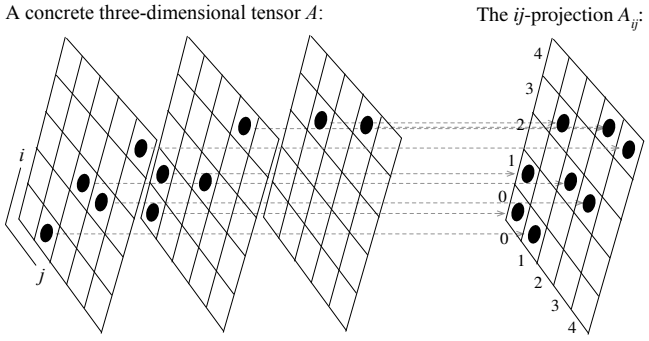

Figure 2. The A_{ij} projection of a 3D tensor A.

2.4 Abstract Domain

Bandedness analysis maps 2D-projections to pairs (p, q) that represent the bands defined in Section 2.2, 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 2.12 shows.

Example 2.12. Continuing with Example 2.11, Figure 3 shows the abstract representation of the bands along the ij projection of the three-dimesional tensor A in Figure 2. 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.

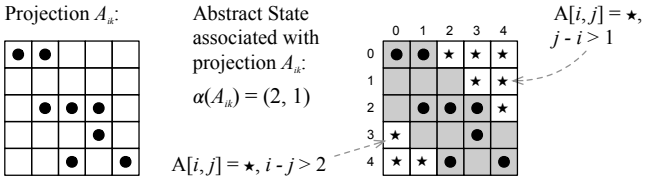

Figure 3. The abstract state that represents the projection A_{ij} seen in Figure 2.

2.5 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 operations, such as elementwise clamping (e.g., ReLU), tensor addition, and `einsum` expressions, which are pervasive in computational graphs. Example 2.13 shows the effect of einsum contractions.

Example 2.13 (Einstein summation)**.** Figure 4 depicts the operation $\texttt{einsum}(xik,\ kj \rightarrow xij)$. 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.

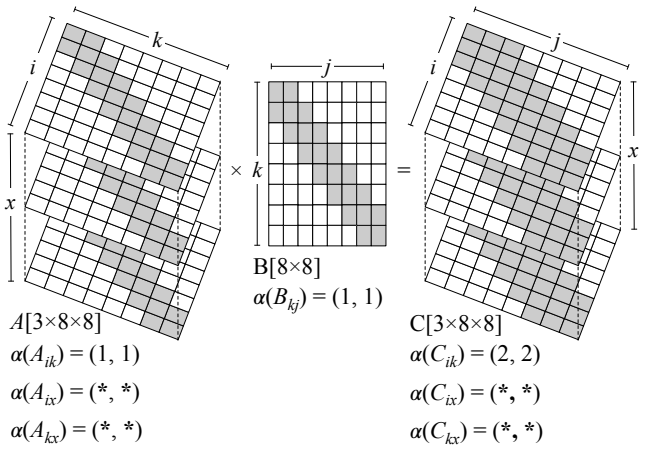

Figure 4. Tensor contraction $\texttt{einsum}(xik,\ kj \rightarrow xij)$. We use * as a generic placeholder for the maximum bandwidth.

Example 2.13 shows the *concrete* effect of an einsum contraction. This paper is interested in the *abstract* effect the contraction (and other operations) has on the banded sparsity pattern of matrices. Example 2.14 shows this abstract effect, explaining how bandedness analysis over-approximates the bands of tensors produced by `einsum` operations.

Example 2.14 (Band propagation)**.** A tensor A, illustrated in Figure 4. is banded along the (i, k) dimensions with symmetric half-bands b, i.e.,

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

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

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

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

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

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 3 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 2.14 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 3.3 discusses the propagation of abstract information in the inverse direction.

2.6 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, 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, such as Ananias et al. [2]'s. Nor does it depend on the number of elements in a tensor, in contrast to the work of Zheng et al. [45]. Section 4 demonstrates empirically that this design choice yields a significantly faster implementation.

3 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 the extended material.

3.1 Lattice

The abstract domain introduced in Section 2.4 induces a natural concretization function γ that maps abstract values, representing the band extents (p, q) (Definition 2.5), 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 3.1. Figure 5 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 $a_0 \longrightarrow a_1$, 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.

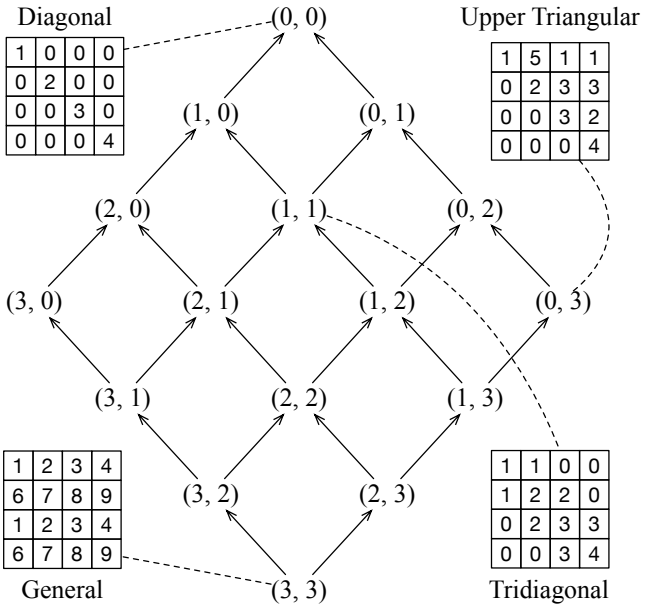

Figure 5. Partial order induced by the least-upper bound of algebraic properties.

3.2 Forward Transfer Functions

Figure 6 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 5, 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 2.1) before the analysis starts.

3.2.1 Unary Transfer Functions. Rules `csum`, `padx`, `zpew`, and `trsp` in Figure 6 show the abstract interpretation of unary operations. The latter refers to transposition, e.g., `einsum`$(ij \longrightarrow ji)$ 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 7 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.

[2]Figure 6 adds names to the indices of einsum operations to be able to refer to specific projections. Thus, we use B_{ij} = `einsum`$(\mathtt{A}_{ji})$ to represent the standard `einsum`$(ij \longrightarrow ji)$ operation.

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

$$\frac{B_{ij} = \text{trim}(A_{ij}, t, b, l, r) \in G \quad \alpha(A_{ij}) = (p, q)}{\alpha(B_{ij}) = (\max(0, \min(*, p+l-t)), \max(0, \min(*, q+t-l)))} \quad \texttt{[trim]}$$

$$\frac{B_{ij} = \text{pad}(A_{ij}, t, b, l, r) \in G \quad \alpha(A_{ij}) = (p, q)}{\alpha(B_{ij}) = (\max(0, p+t-l), \max(0, q+l-t))} \quad \texttt{[padx]}$$

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

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

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

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

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

Figure 6. Forward transfer functions associated with unary and binary tensor operations.

An inverse rule applies for trimming operations that remove elements from matrices (see Figure 7).

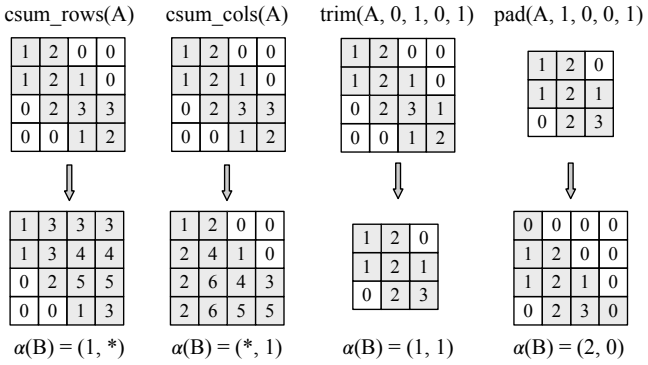

Figure 7. Unary operations and the abstract states they generate, assuming $\alpha(A) = (1, 1)$.

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.

Example 3.2. 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.: $\alpha(\text{clamp}(A)) = \alpha(A)$.

3.2.2 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. Boolean and follows Rule hdmd. If no rule applies, the transfer function conservatively returns the bottom element $(*, *)$, representing an unknown structure.

Together, the rules in Figure 6 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 3.3 illustrates how a more complex operation is handled.

Example 3.3. 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.

3.3 Backward Transfer Functions

Binary operations that form a semiring (see Section 2.1) 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 3.4 illustrates this process.

Example 3.4. 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:

$$\alpha(A) = \alpha(A) \vee (\alpha(B_1) \wedge \cdots \wedge \alpha(B_k)). \quad (1)$$

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 3.5 illustrates this property.

Example 3.5. Figure 8 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 $(*, *)$.

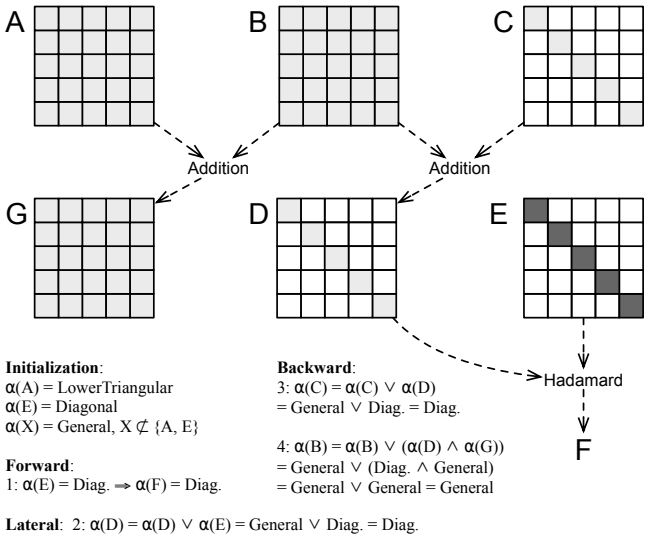

Figure 8. Example illustrating the role of the meet operation in sound backward propagation.

Theorem 3.6 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 3.6. *Let G be an acyclic computational graph. A single forward pass followed by a single backward pass computes a fixed point of the analysis.*

Special Cases. Given an operation $B = f(\ldots, A, \ldots)$, Equation 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 1. This loss of precision is necessary for soundness, as every non-zero entry of A may influence the result.

Example 3.7. Figure 9 illustrates the effect of transposition on backward propagation. In this case, Equation 1 swaps the band extents of B when computing the abstract state of A.

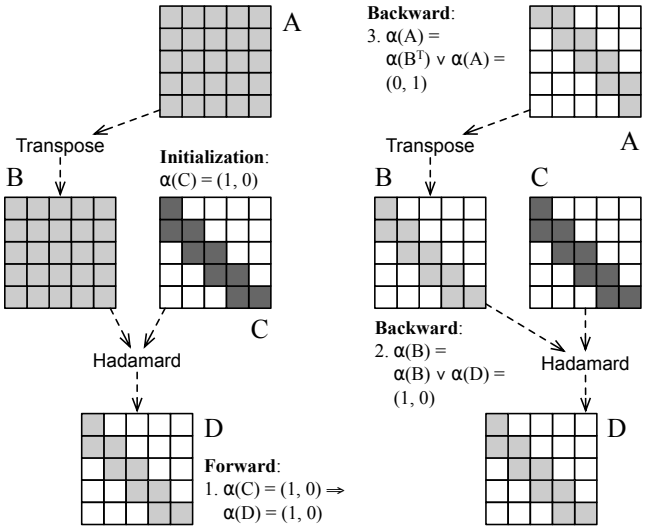

Figure 9. (Left) Effects of forward propagation. (Right) Effects of backward propagation.

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. The reported time and memory values were measured using the Linux `time` command.

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 [22], 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 by Ananias et al. [2], 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. [28]. Matrices are 1024×1024.

Configurations: This paper 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 [36] (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 paper 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.

Notice that several systems already provide abstractions and code generation support for sparse computations, including SparseTIR [43], TACO [24], and DASTAC/StructTensor [15]. In principle, these tools could serve as baselines for comparing kernel execution time. However, extending this comparison to entire computational graphs is challenging. We have, in fact, evaluated StructTensor for this purpose. Although StructTensor can represent banded matrices through its unique-set and redundancy-map abstractions, it generates a single fused kernel for the entire computational graph. In our benchmarks, the resulting binaries were several orders of magnitude slower than those produced by our MLIR-oriented baselines. Consequently, while we use StructTensor as a baseline for the running time of our analysis (Section 4.1), we do not include it, or similar kernel-oriented code generators, in the end-to-end performance comparison of computational graphs (Section 4.3).

Correctness: In every experiment reported in this section, we verified that the code produced by each generator, with or without optimizations, produces identical output when given the same input.

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 [15]: the static analysis used in the implementation of the DASTAC compiler [14].

SparTA [45]: 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 [2], because it is unable to infer any sparsity for the benchmarks used in this paper.

Discussion: Figure 10 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; however, it propagates more information, e.g., it can track sparse subregions within tensors.

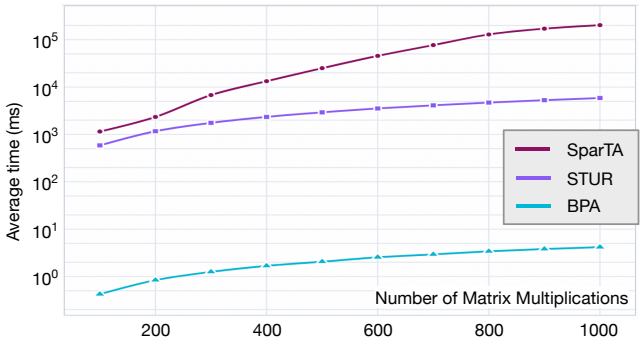

Figure 10. Logarithmic plot comparing the symbolic execution cost of different analyses. STUR refers to StructTensor.

SparTA and BPA both scale linearly with the number of tensors in the graph, but BPA has a 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 10, SparTA's runtime grows quadratically with the tensor dimensions, while BPA's remains constant.

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.

Discussion: Figure 11 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 $(p+q) \geq N-1$.

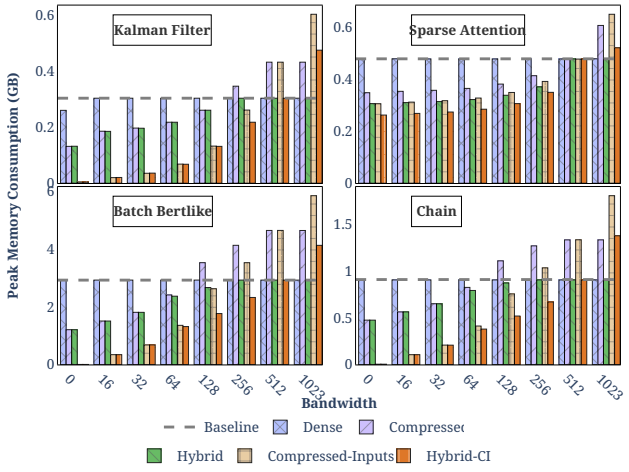

Figure 11. Peak memory consumption of the four computational graphs implemented using the bandedness analysis across different formats and bandwidth values (p = q).

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. In this work, 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 4.1, 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.

4.3 RQ3: Execution Time Savings

Bandedness Propagation Analysis (BPA) enables compiler optimizations that 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 12 illustrates the log-scaled execution times of the four computational graphs under the same experimental configurations described in Section 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

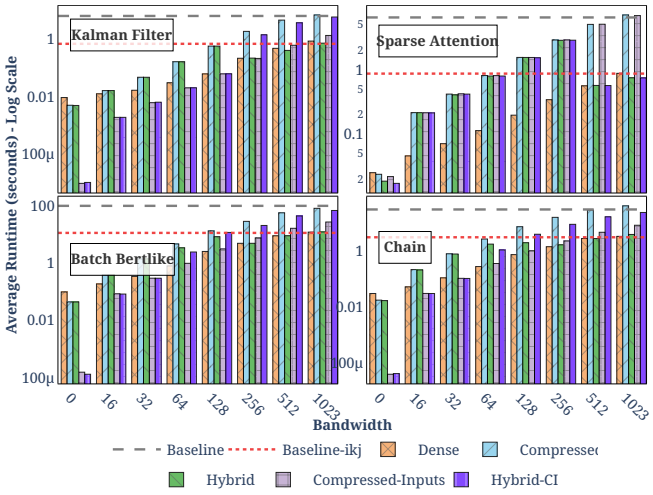

Figure 12. Log-scaled execution time comparison of the bandedness-aware compiler using DIA and dense formats against naive MLIR code (**Baseline**) and MLIR configured with an *i-k-j* matrix-multiplication loop ordering (**Baseline-ikj**). Each value on the x-axis represents a different bandwidth parameter (p = q) for the input tensors of the graph.

reduces the asymptotic time complexity of a naive matrix multiplication from $O(N^3)$ to $O(w^2N)$, 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** configuration 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 4.2) 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 RQ4: Compilation Overhead

Section 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 13 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 products.

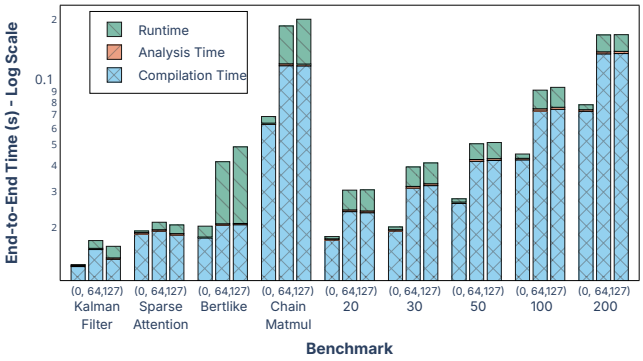

Figure 13. The execution time of BPA compared against the standard compilation and execution times of different computational graphs, evaluated across initial bandwidth values of 0, 64, and 127 (from left to right).

Figure 13 evaluates these workloads using 128×128 tensors, and Figure 14 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 14 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.

4.5 RQ5: Impact of Multi-Propagation

The bandedness propagation analysis transmits abstract state information along two complementary directions: forward (Sec. 3.2) and backward (Sec. 3.3). While forward propagation alone is sufficient for many computational graphs, backward propagation becomes particularly valuable when operations with annihilating elements are present (Sec. 2.1.1)—masked

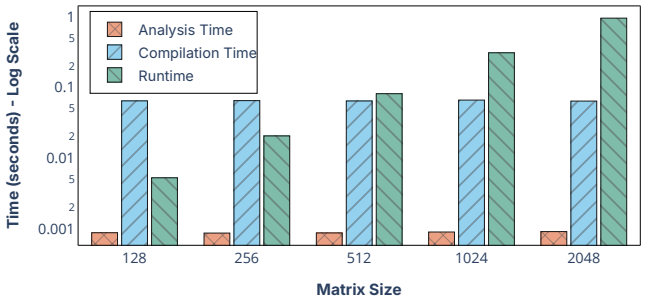

Figure 14. The BPA execution time compared to the compilation, and runtime of the Chain benchmark using different matrix sizes at zero bandwidth.

attention being a representative case. This section empirically evaluates the incremental benefit of backward propagation in this context.

Results and Discussion: Figure 15 illustrates the sparsity ratios inferred during the analysis of masked attention under varying initial non-null bandwidths. The sparsity ratio is defined as the number of zero-valued elements divided by the total number of elements. Bars labeled "*Initial*" denote the sparsity level prior to any propagation, determined solely by the initial annotations of null bands.

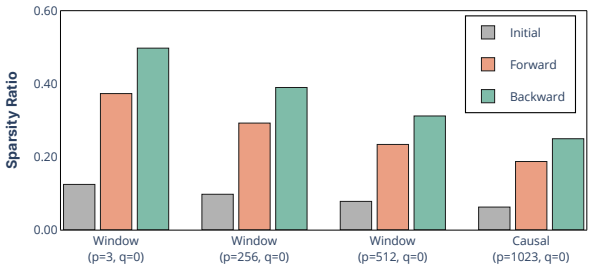

Figure 15. Sparsity ratio (zero elements / total elements) of tensors in masked attention under different masks, before and after each propagation phase.

As shown, forward propagation more than doubles the extent of bands identified as null. This outcome is expected, as the computational graph comprises three input tensors and four intermediate tensors that are amenable to forward analysis. Backward propagation, while yielding a comparatively smaller absolute increase, still contributes meaningfully: in several configurations, it more than triples the null-band extent beyond the forward-only result.

The practical impact of this additional sparsity is further demonstrated in Figure 16. For the attention benchmark, code generated using bidirectional propagation is approximately 15 times faster than forward-only generation in the best case scenario (p=3, q=0) and 2 times faster in the worst case (p=1023, q=0). This difference shows the value of backward propagation for performance-critical applications.

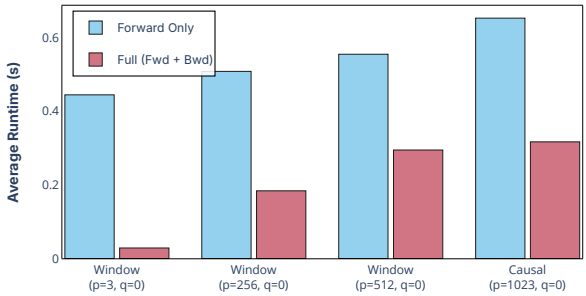

Figure 16. Comparison between the execution time of code generated with forward-only versus forward+backward propagation.

5 Related Work

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. [31], 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. [38]'s MNC, which estimates zeros in 2D matrices. Some analyses are designed without guarantees of soundness, such as the heuristic proposed by Yan et al. [41], 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 BPA, generalize to multidimensional tensors and to different kinds of tensor operations. Figure 17 illustrates their differences. In the rest of this section, we explain these differences, emphasizing that they all operate on *acyclic* computational graphs. This property implies that they are non-iterative: due to acyclicity, a fixed point is guaranteed to be reached after a single round of abstract interpretation (or three rounds, in the case of multidirectional analyses).

Analyses that infer irrelevant elements. The most general sparsity analysis we are aware of is SpArTA [45]. As Figure 17 shows, it represents sparsity information *per tensor element*. Thus, computing abstract states is equivalent to executing the computational graph over boolean tensors. Although the public implementation of SpArTA handles only 2D matrices[3], its presentation assumes multidimensionality.

One drawback of SpArTA is its high asymptotic cost, which motivated Ananias et al. [2] to develop *Sparsity Propagation Analysis*. As Figure 17 shows, this analysis associates each *tensor slice* with a boolean value indicating whether

[3]In https://github.com/microsoft/nni/tree/sparta_artifact on April'26.

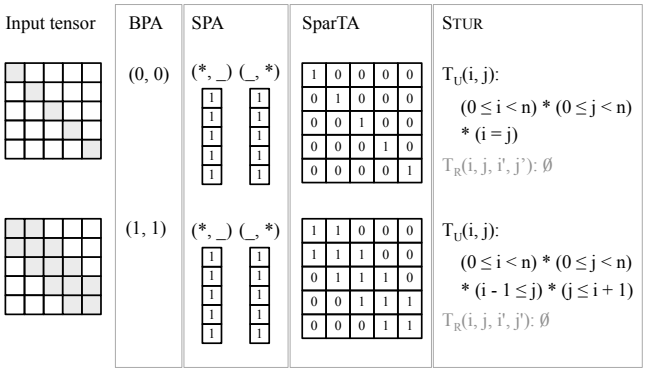

Figure 17. Examples of abstract states used by sparsity propagation analyses. These analyses operate on acyclic computational graphs involving tensors of arbitrary dimensionality. StructTensor supports only forward propagation, whereas the other analyses also support backward propagation.

that slice contains only irrelevant elements or not. A slice, in this context, is a subset of tensor elements in which all but one index are fixed. In the example shown in Figure 17, slices correspond to the one-dimensional rows and columns of a 2D matrix. Due to this choice of representation, Spa is unable to track irrelevant bands, as Figure 17 shows: it infers a diagonal matrix as fully dense.

The analyses of Ananias et al. [2] and Zheng et al. [45] propagate the *irrelevant entries* from Definition 2.4. Thus, in addition to forward propagation, they support lateral and backward inference of irrelevant elements. If this notion is restricted to elements that *must* be zero, rather than elements that *can* be treated as zero, then only forward propagation is possible. This is the approach adopted by Ghorbani et al. [15] in the StructTensor compiler.

Analysis that infer null elements. There exist static analyses that infer elements that must be zero in any execution of a program [15, 29, 31]. However, only StructTensor, from Ghorbani et al. [15], generalizes to tensors. StructTensor represents tensor regions as sums of products of inequalities, as illustrated in Figure 17[4]. 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.

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

6 Conclusion

This paper 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.

We identify two limitations in the current formulation of Bandedness Propagation Analysis that present opportunities for future work. First, the analysis does not support skewed bands. Such structures occur in practice; for example, average-pooling matrices exhibit integer-sloped bands dictated by the pooling factor. Second, our abstract domain uses static constants to define band boundaries, whereas bounds could be specified by symbolic expressions. Much like the extension of interval-based range analysis [8, 34] to symbolic ranges [33], Bandedness Propagation Analysis could be similarly extended to handle symbolic bounds.

Data Availability Statement

An artifact to reproduce the experiments in this paper is available in Zenodo [18]. An up-to-date version of the implementation is available at https://github.com/seliayeu/algebraic-structure-analysis/.

References

[1] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and Structured Tensor Programming with Control Flow. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 117 (April 2025), 31 pages. doi:10.1145/3720473

[2] Kaio Henrique Andrade Ananias, Danila Seliayeu, José Nelson Amaral, and Fernando Magno Quintao Pereira. 2026. Multidirectional Propagation of Sparsity Information across Tensor Slices. In *CGO*. IEEE, New York, USA, 644–656.

[3] Peter Arbenz. 1991. Computing Eigenvalues of Banded Symmetric Toeplitz Matrices. *SIAM J. Sci. Stat. Comput.* 12, 4 (July 1991), 743–754.

[4] Toufik Baroudi, Rachid Seghir, and Vincent Loechner. 2017. Optimization of Triangular and Banded Matrix Operations Using 2d-Packed Layouts. *ACM Trans. Archit. Code Optim.* 14, 4, Article 55 (Dec. 2017), 19 pages. doi:10.1145/3162016

[5] Bernhard Beckermann, Daniel Kressner, and Heather Wilber. 2025. Compression properties for large Toeplitz-like matrices. *Numer. Algorithms* 100, 4 (July 2025), 2041–2067. doi:10.1007/s11075-025-02185-8

[6] Dario Bini and Milvio Capovani. 1983. Spectral and computational properties of band symmetric Toeplitz matrices. *Linear Algebra Appl.* 52 (1983), 99–126.

[7] John Bird. 2010. Higher engineering mathematics. *Elsevier* (2010).

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (Los Angeles, California). Association for Computing Machinery, New York, NY, USA, 238–252.

doi:10.1145/512950.512973

[9] Huimin Cui, Qing Yi, Jingling Xue, and Xiaobing Feng. 2013. Layout-oblivious compiler optimization for matrix computations. *ACM Trans. Archit. Code Optim.* 9, 4, Article 35 (Jan. 2013), 20 pages. doi:10.1145/2400682.2400694

[10] Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2024. Flex Attention: A Programming Model for Generating Optimized Attention Kernels. arXiv:2412.05496 [cs.LG] https://arxiv.org/abs/2412.05496

[11] Harini Eavani, Theodore D Satterthwaite, Roman Filipovych, Raquel E Gur, Ruben C Gur, and Christos Davatzikos. 2015. Identifying sparse connectivity patterns in the brain using resting-state fMRI. *Neuroimage* 105 (2015), 286–299.

[12] Gemma C. Garriga, Esa Junttila, and Heikki Mannila. 2008. Banded structure in binary matrices. In *KDD* (Las Vegas, Nevada, USA). Association for Computing Machinery, New York, NY, USA, 292–300. doi:10.1145/1401890.1401929

[13] Zhengyang Geng, Meng-Hao Guo, Hongxu Chen, Xia Li, Ke Wei, and Zhouchen Lin. 2021. Is Attention Better Than Matrix Decomposition? arXiv:2109.04553 [cs.CV] https://arxiv.org/abs/2109.04553

[14] Mahdi Ghorbani, Emilien Bauer, Tobias Grosser, and Amir Shaikhha. 2025. Compressed and Parallelized Structured Tensor Algebra. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 141 (April 2025), 29 pages. doi:10.1145/3720506

[15] Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling Structured Tensor Algebra. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 229:204–229:233. doi:10.1145/3622804

[16] Ahan Gupta, Yueming Yuan, Devansh Jain, Yuhao Ge, David Aponte, Yanqi Zhou, and Charith Mendis. 2025. SPLAT: A Framework for Optimised GPU Code-Generation for SParse reguLar ATtention. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 138 (April 2025), 29 pages. doi:10.1145/3720503

[17] Sukhyun Han, Seongwook Kim, Gwangeun Byeon, Jihun Yoon, and Seokin Hong. 2025. Zebra: Leveraging diagonal attention pattern for vision transformer accelerator. In *DATE*. IEEE, New York, USA, 1–7.

[18] Kaio Henrique. 2026. *BPA Artifact.* doi:10.5281/zenodo.21892149

[19] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (Oct. 2021), 29 pages. doi:10.1145/3485505

[20] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *ASPLOS* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. doi:10.1145/3582016.3582051

[21] Shiyuan Huang, Fangxin Liu, Tian Li, Zongwu Wang, Ning Yang, Haomin Li, and Li Jiang. 2024. STCO: Enhancing Training Efficiency via Structured Sparse Tensor Compilation Optimization. *ACM Transactions on Design Automation of Electronic Systems* 30, 1 (2024), 1–22.

[22] Rudolph Emil Kalman. 1960. A new approach to linear filtering and prediction problems. *J. Basic Eng.* 82, 1 (1960), 35–45.

[23] Ismail Khalfaoui-Hassani. 2024. Dilated Convolution with Learnable Spacings. arXiv:2408.06383 [cs.LG] https://arxiv.org/abs/2408.06383

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[25] Sureyya Emre Kurt, Saurabh Raje, Aravind Sukumaran-Rajam, and P. Sadayappan. 2022. Sparsity-Aware Tensor Decomposition. In *IPDPS*. IEEE, USA, 952–962. doi:10.1109/IPDPS53621.2022.00097

[26] Rubens Lacouture, Nathan Zhang, Ritvik Sharma, Marco Siracusa, Fredrik Kjolstad, Kunle Olukotun, and Olivia Hsu. 2026. FuseFlow: A Fusion-Centric Compilation Framework for Sparse Deep Learning on Streaming Dataflow. In *ASPLOS* (USA). Association for Computing Machinery, New York, NY, USA, 798–820. doi:10.1145/3779212.3790165

[27] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *CGO*. IEEE Press, New York, US, 2–14. doi:10.1109/CGO51591.2021.9370308

[28] Damitha Lenadora, Vimarsh Sathia, Gerasimos Gerogiannis, Serif Yesil, Josep Torrellas, and Charith Mendis. 2026. GRANII: Selection and Ordering of Primitives in GRAph Neural Networks using Input Inspection. In *CGO*. IEEE, New York, 14–27. doi:10.1109/CGO68049.2026.11395219

[29] Guilherme Vieira Leobas and Fernando Magno Quintão Pereira. 2020. Semiring optimizations: dynamic elision of expressions with identity and absorbing elements. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 131 (Nov. 2020), 28 pages. doi:10.1145/3428199

[30] Huize Li, Zhaoying Li, Zhenyu Bai, and Tulika Mitra. 2024. Asadi: Accelerating sparse attention using diagonal-based in-situ computing. In *HPCA*. IEEE, New York, USA, 774–787.

[31] Francisco Lopez, Lars Karlsson, and Paolo Bientinesi. 2026. Compilation of Generalized Matrix Chains with Symbolic Sizes. In *CGO*. IEEE, New York, 466–478. doi:10.1109/CGO68049.2026.11395236

[32] Maxim Naumov. 2017. Parallel Complexity of Forward and Backward Propagation. arXiv:1712.06577 https://arxiv.org/abs/1712.06577

[33] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2014. Validation of memory accesses through symbolic analyses. In *OOPSLA* (Portland, Oregon, USA). Association for Computing Machinery, New York, NY, USA, 791–809. doi:10.1145/2660193.2660205

[34] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *CGO*. IEEE Computer Society, USA, 1–11. doi:10.1109/CGO.2013.6494996

[35] Saurabh Raje, Yufan Xu, Atanas Rountev, Edward F. Valeev, and P. Sadayappan. 2024. CoNST: Code Generator for Sparse Tensor Networks. *ACM Trans. Archit. Code Optim.* 21, 4, Article 82 (Nov. 2024), 24 pages. doi:10.1145/3689342

[36] Youcef Saad. 1990. *SPARSKIT: A basic tool kit for sparse matrix computations.* Technical Report. NASA Ames Research Center.

[37] S. Serra. 1999. Spectral and Computational Analysis of Block Toeplitz Matrices Having Nonnegative Definite Matrix-Valued Generating Functions. *BIT* 39, 1 (March 1999), 152–175. doi:10.1023/A:1022329526925

[38] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD* (Amsterdam, Netherlands). Association for Computing Machinery, New York, NY, USA, 1607–1623. doi:10.1145/3299869.3319854

[39] Daniele G. Spampinato and Markus Püschel. 2016. A Basic Linear Algebra Compiler for Structured Matrices. In *CGO*. Association for Computing Machinery, New York, NY, USA, 117–127. doi:10.1145/2854038.2854060

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS* (Long Beach, California, USA). Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[41] Bobby Yan, Alexander J Root, Trevor Gale, David Broman, and Fredrik Kjolstad. 2026. Fast Autoscheduling for Sparse ML Frameworks. In *CGO*. IEEE, New York, 28–43.

[42] Wangdong Yang, Kenli Li, Yan Liu, Lin Shi, and Lanjun Wan. 2014. Optimization of quasi-diagonal matrix–vector multiplication on GPU. *IJHPCA* 28, 2 (2014), 183–195.

[43] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *ASPLOS* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 660–678. doi:10.1145/3582016.3582047

[44] Liang Zhao, Siyu Liao, Yanzhi Wang, Zhe Li, Jian Tang, and Bo Yuan. 2017. Theoretical properties for neural networks with weight matrices of low displacement rank. In *ICML* (Sydney, NSW, Australia). Microtome Publishing, Brookline, USA, 4082–4090.

[45] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA:Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *OSDI*. USENIX Association, Berkeley, USA, 213–232.

A Artifact

This section provides the tools and instructions necessary to reproduce all the results presented in Section 4 via a Zenodo artifact [18].

A.1 Artifact Check-list (Meta-information)

- **Algorithm:** The artifact builds and evaluate the implementation of the Bandedness Propagation Analysis (BPA) used to propagate bandwidth information across computational graphs.
- **Program:** The analysis and the code generator is implemented on top of MLIR [27] compiler infrastructure.
- **Compilation:** The project was implemented using MLIR with commit hash `5e78d5e262c0`, **C++** 17, and Clang 23. The following lowering pipeline converts the upstream and custom dialects to LLVM dialect.

```
--banded-analysis, --banded-rewrite,
--empty-tensor-to-alloc-tensor,
--one-shot-bufferize=bufferize-function-
    boundaries,
--convert-linalg-to-loops, --convert-scf-to-cf,
--convert-math-to-llvm, --convert-math-to-libm,
--expand-strided-metadata, --lower-affine,
--convert-arith-to-llvm, --convert-func-to-llvm,
     --convert-cf-to-llvm, --finalize-memref-to
    -llvm,
--reconcile-unrealized-casts
```

  The LLVM dialect is then translated into LLVM IR using the `mlir-translate` tool, which is then finally compiled with `llc -O3`.
- **Binary:** The artifact will automatically compile the analysis and reproduce all the data and figures,
- **Run-time environment:** The run-time requires a Docker installation with necessary privileges to build and run containers.
- **Hardware:** 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
- **Metrics:** Execution time, compilation time, analysis time, percent sparsity, and memory consumption.
- **Output:** The artifact will call a set of Python scripts to run the BPA binary, compile the data into CSV files, and plot the results. All six figures from Section 4 will be automatically stored in a folder called *results* in the current directory.
- **Experiments:** Each experiment presented in Section 4.
- **How much disk space required (approximately)?:** 10 GiB.
- **How much time is needed to prepare workflow (approximately)?:** No preparation is needed in addition to installing Docker. However, the underlying image used in the container builds the specified MLIR version, which can take a few hours depending on the machine.
- **How much time is needed to complete experiments (approximately)?:** It takes approximately 8 hours to complete the experiments on the hardware described in Section 4 once the Docker image is built.
- **Publicly available?:** BPA is available at https://github.com/seliayeu/algebraic-structure-analysis and the artifact is available at https://zenodo.org/records/21892150.
- **Workflow framework used?:** Docker
- **Archived (provide DOI)?:** https://zenodo.org/records/21892150

A.2 Description

A.2.1 How delivered. Delivered via artifact https://zenodo.org/records/21892150 and source code on https://github.com/seliayeu/algebraic-structure-analysis.

A.2.2 Hardware dependencies. The experiments require 10 GiB of disk space and up to 8 GiB of memory. We tested the artifact on three different hardware configurations:

- AMD Ryzen Threadripper 7970X 32-Cores with 128 GiB of RAM
- Mabook Air M4 2025 with 16 GiB of RAM
- AMD Ryzen 7 3700X (8 cores, 16 threads) with 32 GiB of RAM

A.2.3 Software dependencies. All tests were validated using `Ubuntu 24.04` inside *Docker*, version `29.2.1`, build `a5c7197`.

A.3 Installation

1. Download the artifact at https://zenodo.org/records/21892150.
2. Build the Docker image:

```
$ docker build -f Dockerfile.artifact \
-t bpa-artifact --target runtime .
```

3. Run the experiments:

```
$ docker run -d -v \
$(pwd)/results:/app/bpa/results --name \
bpa-experiments bpa-artifact:latest
```

The `-d` flag will run the container in detached mode. It is possible to inspect the progress and state of the container by running the following command:

```
$ docker logs bpa-experiments
```

A.4 Experiment Workflow

By default, the Docker container will run a Python script called `artifact.py` to collect data and generate all figures from Section 4 using the average of 10 executions per experiment. The artifact is

divided into six figures, which, if not specified, are built in ascending order.

The artifact script has two main phases:

1. Call the respective binary for the experiment. Store metrics in a CSV file inside the figure folder under the `results` directory.
2. Once the data is collected, a PNG file is saved in the figure folder (e.g. `results/Figure 13/compilation_time.png`).

A.5 Evaluation and Expected Result

Once finished, a directory with the csv files and figures from Section 4 is automatically copied from the Docker container to the current directory on the host. Inside the `results` folder, a directory for each figure with its `.csv`, `.png` and an iterative `.html` plot files inside will be available.

A.6 Experiment Customization

We added additional flags for debugging purposes in the artifact. You can specify a list of figures to generate and the number of executions (default is 10) to run each experiment. The `-figures` flag is used to feed a comma-separated list of figures (from 10 to 16), and the `-runs` controls the number of executions.

The following example generates only Figure 11 using the average of 10 executions.

```
docker run -d \
  -v $(pwd)/results:/app/bpa/results \
  --rm \
  --name bpa-experiments \
  bpa-artifact \
  --figures 11 \
  --repeat 10
```