A Dataflow Analysis For Symmetry in Tensor Programs

GAURAV ARYA and APURVA GANDHI

1 Introduction

Many tensor and linear algebra workloads in machine learning and general scientific computing—such as attention matrices, covariance computations, or physical simulation kernels—exhibit computation with symmetric structure. But without explicit user intervention, modern compiler stacks may treat such tensors as dense and unstructured, resulting in redundant computation and memory usage. In this project, we implement compiler analyses for automatically detecting and exploiting symmetry in tensor computations. Specifically, we build compiler analysis and rewrite passes in MLIR [2] that:

(1) Detect symmetric outputs from tensor computations.
(2) Propagate symmetry metadata through the IR.
(3) Apply symmetry-exploiting code transformations.

By detecting symmetry at compile time, we can perform optimizations that significantly reduce memory usage and redundant computation of tensor programs.

The main contribution of this project is a lattice-based *dataflow analysis* for propagating tensor symmetry metadata, which leverages the notion of partial symmetry [6] to support tensors of arbitrary rank. Additionally, we contribute optimizations that leverage this metadata to perform symmetry-exploiting code transformations on tensor programs, which achieve performance improvements ranging from 1.1x to 4x on preliminary benchmarks. Our results show that symmetry detection can be effectively performed at the MLIR level and utilized for speedups on on linear algebra workloads.

Overview Example. Figure 1 illustrates a tensor program that benefits substantially from our symmetry-aware compiler passes. The program constructs a partially symmetric tensor by adding an input tensor to a three-axis permutation of itself, and then performs a reduction over one of the symmetric dimensions. In the unoptimized StableHLO IR (Fig. 1b), the compiler materializes the symmetry explicitly, introducing two transposes and two additions. Because the generated tensor is partially symmetric across its first and third axes, our analysis is able to infer this structure, annotate the IR with partial-symmetry metadata, and propagate this information into the subsequent reduction.

Authors' Contact Information: Gaurav Arya, gauravar@cs.cmu.edu; Apurva Gandhi, apurvag@cs.cmu.edu.

In the optimized IR (Fig. 1c), the redundant transpose–addition pair is eliminated entirely, and the reduction is rewritten to operate over a symmetry-equivalent axis. This avoids unnecessary transpositions and enables layout-aware simplifications. This example demonstrates how partial symmetry inference unlocks optimizations that are not visible to standard canonicalization or CSE passes.

2 Partial Symmetry

A tensor T is *symmetric* if its values are invariant to permutations of its indices.

Definition 2.1 (Symmetry). Let T be a tensor of rank r. We say that T is symmetric if

$$T[i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}] = T[i_1, i_2, \ldots, i_r] \tag{1}$$

for all permutations σ of $\{1, 2, \ldots, r\}$.

However, in practical use cases, a tensor is often only *partially* symmetric. For example, a tensor of shape $N \times N \times K$ representing a batched set of symmetric matrices may only be symmetric with respect to its first two indices. To capture such symmetries in a tensor of arbitrary rank, Patel et al. [6] used the notion of *partial symmetry*.

Definition 2.2 (Partial Symmetry). Let T be a tensor of rank r, and $\Pi = S_1 \sqcup S_2 \sqcup \ldots \sqcup S_k$ be a partition of $\{1, 2, \ldots, r\}$. We say that T is partially symmetric with respect to Π if

$$T[i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}] = T[i_1, i_2, \ldots, i_r] \tag{2}$$

for all permutations σ of $\{1, 2, \ldots, r\}$ such that for all $j \in \{1, 2, \ldots, k\}$,

$$i \in S_j \implies \sigma(i) \in S_j. \tag{3}$$

In words, T is partially symmetric with respect to Π if its values are invariant to permutations of its indices that only permute elements within their parts in Π. We can consider some examples:

- A fully symmetric tensor T of rank r is partially symmetric with respect to $\Pi = \{1, 2, \ldots, r\}$.
- Every tensor T of rank r is partially symmetric with respect to $\Pi = \{1\} \sqcup \{2\} \sqcup \ldots \sqcup \{r\}$.
- A tensor T of shape $N \times N \times K$ that satisfies $T[i, j, k] = T[j, i, k]$ for all $i, j \in \{1, 2, \ldots, N\}$ and $k \in \{1, 2, \ldots, K\}$ is partially symmetric with respect to $\Pi = \{1, 2\} \sqcup \{3\}$.

The SySTeC system contributed by Patel et al. [6] worked at the level of *individual kernels*, analyzing scalar arithmetic expressions and loops. To enable SySTeC's optimizations, the

1: **function** MAIN(X) ▷ $1024 \times 3 \times 1024$ *tensor*
2: $A \leftarrow X + \text{Transpose}(X,\ (2, 1, 0))$
3: $B \leftarrow A + \text{Transpose}(A,\ (2, 1, 0))$
4: $R \leftarrow \text{Reduce}(B,\ \text{axis} = 0,\ \text{op} = \text{add})$
5: **return** R

(a) Pseudocode

```
func.func private @main_1(%arg0: tensor<1024x3x1024xf32>)
    -> (tensor<3x1024xf32> {jax.result_info = "result"}) {
  %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
  %0 = stablehlo.transpose %arg0, dims = [2, 1, 0]
       : (tensor<1024x3x1024xf32>) -> tensor<1024x3x1024xf32>
  %1 = stablehlo.add %arg0, %0
       : tensor<1024x3x1024xf32>
  %2 = stablehlo.transpose %1, dims = [2, 1, 0]
       : (tensor<1024x3x1024xf32>) -> tensor<1024x3x1024xf32>
  %3 = stablehlo.add %1, %2
       : tensor<1024x3x1024xf32>
  %4 = stablehlo.reduce(%3 init: %cst)
        applies stablehlo.add across dimensions = [0]
        : (tensor<1024x3x1024xf32>, tensor<f32>) -> tensor<3x1024xf32>
  return %4 : tensor<3x1024xf32>
}
```

(b) StableHLO IR before symmetry-aware optimization. The compiler materializes two transposes and two additions.

```
func.func private @main_1(%arg0: tensor<1024x3x1024xf32>)
    -> (tensor<3x1024xf32> {jax.result_info = "result"}) {
  %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
  %0 = stablehlo.transpose %arg0, dims = [2, 1, 0]
    : (tensor<1024x3x1024xf32>) -> tensor<1024x3x1024xf32>
  %1 = stablehlo.add %arg0, %0
    {enzymexla.partial_symmetry = [#enzymexla.partial_symmetry<<[0, 2]>>]}
    : tensor<1024x3x1024xf32>
  %2 = stablehlo.add %1, %1
    {enzymexla.partial_symmetry = [#enzymexla.partial_symmetry<<[0, 2]>>]}
    : tensor<1024x3x1024xf32>
  %3 = stablehlo.reduce(%2 init: %cst) applies stablehlo.add across dimensions = [0]
    {enzymexla.partial_symmetry = [#enzymexla.partial_symmetry<<[0, 1]>>]}
    : (tensor<1024x3x1024xf32>, tensor<f32>) -> tensor<3x1024xf32>
  return %3 : tensor<3x1024xf32>
}
```

(c) Optimized StableHLO IR. Partial symmetry detection eliminates the second transpose–addition pair and rewrites the reduction to operate over an equivalent symmetric axis.

Fig. 1. A motivating example showing how our symmetry-aware analysis detects and propagates partial symmetry, enabling elimination of redundant transposes and rewriting reductions over symmetry-equivalent axes.

user is expected to annotate the input tensors with partial symmetry information. In contrast, in this project we work with the StableHLO dialect of MLIR, in which operations are surfaced at the tensor level. Rather than requiring user annotation of each symmetric tensor, we wish to both *generate* and *propagate* symmetry metadata through the IR. This motivates the development of a dataflow analysis based on partial symmetry.

3 Dataflow Analysis for Partial Symmetry

We use a lattice-based dataflow analysis. The analysis annotates each tensor in the program with a partition Π, where the annotation is sound if the tensor is partially symmetric with respect to Π in all possible executions of the program.

3.1 Partial Symmetry Lattice

We first define a lattice of possible partial symmetries for a tensor of rank r. The points of the lattice are the partitions Π of $\{1, 2, \ldots, r\}$. The *top element* of the lattice is the partition $\Pi = \{1, 2, \ldots, r\}$, which corresponds to a fully symmetric tensor. The *meet* $\Pi_1 \wedge \Pi_2$ of two partitions Π_1 and Π_2 is defined as follows: two indices i and j are placed in the same part of $\Pi_1 \wedge \Pi_2$ if and only if they are in the same part of both Π_1 and Π_2. This definition of the meet ensures that annotated partial symmetries are present in *all* possible executions of the program.

In our implementation, we represent points of the lattice by associating each dimension with an ID. Each ID identifies a group of symmetric dimensions, so that the underlying partition Π can be recovered by grouping the dimensions by their ID.

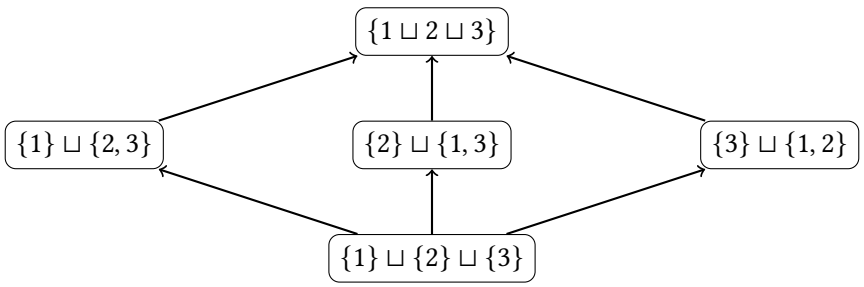

Fig. 2. The partial symmetry lattice for tensors of rank 3.

3.2 Transfer Function

At the start of the dataflow analysis, we initialize the state of function block arguments to the bottom element $\Pi = \{1\} \sqcup \{2\} \sqcup \ldots \sqcup \{r\}$, which corresponds to a tensor that

is not symmetric. An exception to this rule is if the input IR contains an explicit user-provided annotation of partial symmetry of an input, in which case that annotation is used to initialize the state. We now discuss the transfer function for a number of StableHLO operations.

Constants. StableHLO features constant operations, which define ranked tensors from constant literals. To check partially symmetry of a constant tensor T of rank r, we iterate over all $\binom{r}{2}$ pairs of dimensions. For each pair of dimensions (n, m), we check whether

$$T[i_1, i_2, \ldots, i_n, \ldots, i_m, \ldots, i_r] = T[i_1, i_2, \ldots, i_m, \ldots, i_n, \ldots, i_r] \tag{4}$$

for all $i_1, i_2, \ldots, i_r \in \{1, 2, \ldots, N\}$. If so, we merge the set of symmetric dimensions associated with n with that associated with m. After iterating over all pairs of dimensions, we return the final partition of dimensions.

Elementwise Operations. StableHLO features a large number of elementwise operations, which apply a scalar function to either a single input tensor or two input tensors of identical shape. If the operation is unary (e.g. elementwise sin, cos, exp, log), we simply propagate the partial symmetry of the input tensor to the output tensor, i.e., the transfer function is the identity. If the operation is binary (e.g. addition, subtraction, multiplication, division), let the input tensors be T_{LHS} and T_{RHS}, with annotations Π_{LHS} and Π_{RHS}, respectively, and the output tensor be T, with annotation Π. The transfer function for this operation has two cases.

(1) *No aliasing detected between T_{LHS} and T_{RHS}.* If no aliasing is detected, we simply apply the meet of the lattice:

$$\Pi = \Pi_{\text{LHS}} \wedge \Pi_{\text{RHS}}. \tag{5}$$

(2) *Aliasing detected between T_{LHS} and T_{RHS}.* While the previous analysis is always conservative, if we are able to detect aliasing between T_{LHS} and T_{RHS}, then we have an opportunity to perform a more refined analysis. In particular, our analysis attempts to detect that T_{RHS} is a transposition of T_{LHS} by checking the defining operations of each tensor. If this detection is successful, we check the transposition permutation σ to see if it is equivalent to a swap of a pair of dimensions (n, m). If the transposition is of this

form, *and* the elementwise operation f is commutative, we have that

$$
\begin{aligned}
&T[i_1, i_2, \ldots, i_n, \ldots, i_m, \ldots, i_r] \\
&= f(T_{\mathrm{LHS}}[i_1, i_2, \ldots, i_n, \ldots, i_m, \ldots, i_r], T_{\mathrm{RHS}}[i_1, i_2, \ldots, i_n, \ldots, i_m, \ldots, i_r]) && (6)\\
&= f(T_{\mathrm{LHS}}[i_1, i_2, \ldots, i_n, \ldots, i_m, \ldots, i_r], T_{\mathrm{LHS}}[i_1, i_2, \ldots, i_m, \ldots, i_n, \ldots, i_r]) && (7)\\
&= f(T_{\mathrm{LHS}}[i_1, i_2, \ldots, i_m, \ldots, i_n, \ldots, i_r], T_{\mathrm{LHS}}[i_1, i_2, \ldots, i_n, \ldots, i_m, \ldots, i_r]) && (8)\\
&= f(T_{\mathrm{LHS}}[i_1, i_2, \ldots, i_m, \ldots, i_n, \ldots, i_r], T_{\mathrm{RHS}}[i_1, i_2, \ldots, i_m, \ldots, i_n, \ldots, i_r]) && (9)\\
&= T[i_1, i_2, \ldots, i_m, \ldots, i_n, \ldots, i_r]. && (10)
\end{aligned}
$$

Thus, if this pattern is detected, our analysis is able to soundly generate *additional* symmetry by merging the dimension sets for n and m in $\Pi_{\mathrm{LHS}} \wedge \Pi_{\mathrm{RHS}}$ to form Π. This logic allows us, in a general way, to generate symmetry from operations such as $A + A^T$.

Broadcasting Operations. StableHLO offers a "broadcast in-dimension" operation, which is a unary instruction that creates additional dimensions over which an existing input tensor is broadcasted. This operation can be handled by preserving the symmetry between existing dimensions, while marking all broadcasted dimensions as symmetric with one another (in fact, the tensor is constant in each of these dimensions, but our current lattice is not refined enough to capture this additional structure, as it only considers symmetry). For example, if a tensor T of rank r with annotation Π_{IN} is broadcasted to create additional dimensions $r+1, r+2, \ldots, r+k$, then the annotation of the output is

$$\Pi = \Pi_{\mathrm{IN}} \sqcup \{r+1, r+2, \ldots, r+k\}. \tag{11}$$

Tensor Contraction. The most complicated case for our analysis is that of tensor contraction. StableHLO offers a "dot general" operation, which is a generalization of matrix multiplication for more flexible tensor contractions. The operation accepts arrays of batch dimensions *lhsBatch* and *rhsBatch* for the LHS and RHS tensors T_{LHS} and T_{RHS}, as well as array of contracting dimensions *lhsContract* and *rhsContract*. All dimensions that are non-batching and non-contracting in each tensor are designated as free dimensions, and the mapping from the free dimensions of the output back to the inputs T_{LHS} and T_{RHS} can be constructed via arrays *lhsFree* and *rhsFree*. The result of a dot general operation is a tensor of rank $b + r_1 + r_2$, where b is the number of batching dimensions, r_1 is the number of free dimensions taken from the LHS tensor, and r_2 is the number of free dimensions taken from the RHS tensor.

Just as for elementwise operations, we split the transfer function for dot general operations into two cases.

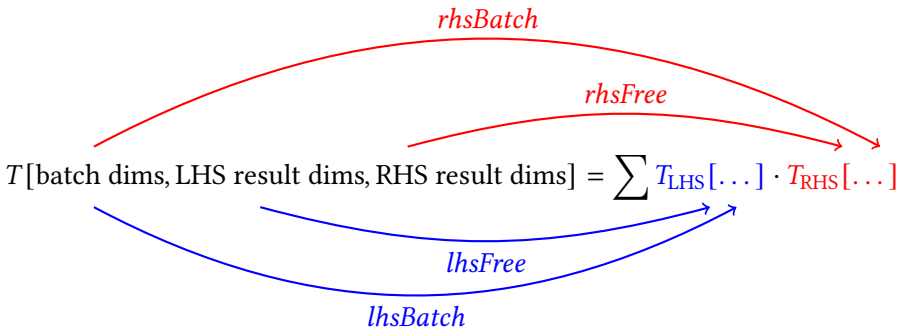

Fig. 3. The mapping of dimensions for a StableHLO dot general operation.

(1) *No aliasing detected between* T_{LHS} *and* T_{RHS}. We place batch dimensions (i, j) in the same part of Π if $(lhsBatch[i], lhsBatch[j])$ are in the same part of Π_{LHS} and $(rhsBatch[i], rhsBatch[j])$ are in the same part of Π_{RHS}. We place free dimensions from the LHS (i, j) in the same part of Π if $(lhsFree[i], lhsFree[j])$ are in the same part of Π_{LHS}, and we place free dimensions from the RHS (i, j) in the same part of Π if $(rhsFree[i], rhsFree[j])$ are in the same part of Π_{RHS}.

(2) *Aliasing detected between* T_{LHS} *and* T_{RHS}. If aliasing is detected, we have an opportunity to detect additional symmetries *between* the free dimensions from the LHS and the free dimensions from the RHS. Suppose that we detect that T_{RHS} is a transposition of T_{LHS} with transpose permutation σ. Additionally, suppose that it holds that $\sigma(rhsBatch[i])$ and $lhsBatch[i]$ are in the same part of Π_{LHS} for each batch dimension i, and that $\sigma(rhsContract[i])$ and $lhsContract[i]$ are in the same part of Π_{LHS} for each contracting dimension i. Then, we can conclude that a free dimension from the LHS i is symmetric to a free dimension from the RHS j if $\sigma(rhsFree[i]) = lhsFree[j]$. This logic allows us, in a general way, to generate symmetry from operations such as AA^T.

4 Symmetry-Exploiting Optimizations

Using the information from our symmetry-detection dataflow analysis, we have implemented two initial optimization passes, each of which works on tensors of arbitrary rank.

Transpose Elimination. Our first symmetry-exploiting optimization eliminates transpose operations that can be determined to be redundant. In particular, consider a transpose operation on a tensor T of rank r, with transpose permutation σ. If the tensor T is partially symmetric with respect to Π, and furthermore it holds that $\sigma(i) = j \implies i \in \Pi_j$, then we can conclude that

$$T[i_{\sigma(1)}, i_{\sigma(2)}, \ldots, i_{\sigma(r)}] = T[i_1, i_2, \ldots, i_r]. \tag{12}$$

In words, if each dimension is permuted to another dimension in the same symmetric dimension set, then the transpose is equivalent to a no-op. Our rewrite pass detects this and elides the transpose operation, replacing all uses of the transposed tensor with the original tensor.

Reduction Axis Rotation. Our second symmetry-exploiting optimization optimizes the axes used in a reduction to maximize memory locality, by permuting the reduction dimensions to prioritize reducing over dimensions with lower stride lengths. In particular, given a reduction operation, our optimization checks if there exists an axis m that is

(1) not reduced over;
(2) partially symmetric to an axis n that is reduced over;
(3) and satisfies $m > n$.

If such an axis exists, our optimization pass modifies the reduction axes such that we reduce over m instead of n. Assuming that the tensors are stored in row-major order (as is the case for the XLA backend), this leads to an improvement in memory locality and consequent performance improvements.

5 Benchmarks

We evaluate our symmetry detection and optimization framework on a suite of microbenchmarks designed to stress-test different aspects of partial symmetry exploitation. All experiments are conducted on a CPU backend using JAX [1] with the Enzyme-JAX[1] [4, 5, 7] custom compiler infrastructure to connect JAX to MLIR [3]. Each benchmark is run for 100 iterations, with timing measurements taken after JIT compilation and warm-up.

5.1 Implementation Overview

Our dataflow analysis is written using the MLIR dataflow framework, where we implement a `PartialSymmetryLattice` class that implements the MLIR dataflow module's `AbstractSparseLattice` interface and a `PartialSymmetryAnalysis` class that implements the MLIR dataflow module's `SparseForwardDataflowAnalysis` interface. Using these types, we implement a pass `partial-symmetry-annotate` that annotates the IR with partial symmetry information. Finally, our optimization passes are implemented as MLIR pattern rewrites, which detect the operation of interest, and, if a partial symmetry annotation is present in the IR, performs the appropriate optimization.

[1] https://github.com/EnzymeAD/Enzyme-JAX.

5.2 Experimental Setup

Our benchmarks are implemented in JAX and compiled through the StableHLO dialect of MLIR. We compare two compilation pipelines:

(1) **Baseline**: Standard compilation with inlining, canonicalization, and CSE passes.
(2) **Optimized**: Baseline passes plus our partial symmetry analysis (`partial-symmetry-annotate`) and symmetry-exploiting transformations (`transpose-partial-symmetry-simplify` and `reduce-partial-symmetry-rotate-axes`).

We measure both the reduction in transpose operations in the generated IR and the end-to-end execution time. The reduction in transposes serves as a proxy for redundant computation elimination, while execution time captures the actual performance impact.

5.3 Benchmark Suite

Our benchmark suite consists of six microbenchmarks that capture common structures in scientific computing and machine learning workloads. Each benchmark is designed to stress a different aspect of partial symmetry detection, symmetry propagation, or symmetry-aware optimization.

Single Symmetric Operation. This benchmark computes `a = x.T + x; return a + a.T` on a 2048×2048 matrix. It tests the ability of the analysis to detect symmetry generation from a single transpose–addition pattern and eliminate the resulting redundant transpose in the final `a.T`.

Chained Symmetric Operations. This benchmark performs ten iterations of the symmetric update `a = a.T + a`, starting from an initial symmetric matrix. It tests whether symmetry information can be propagated across multiple iterations, allowing the compiler to eliminate chains of redundant transposes as symmetry is maintained over time.

Interleaved Symmetric Operations. This benchmark interleaves symmetric and non-symmetric computation through ten iterations of `a = a.T * 1.99 + a * 0.01`. It evaluates whether the analysis remains robust when symmetry is partially preserved but matrix additions and transposes are interleaved with scaling by broadcasted scalars.

Dot Product CSE.. This benchmark evaluates how partial symmetry detection interacts with MLIR's CSE pass. After computing `a = x.T + x`, the three expressions `dot(a, a)`, `dot(a, a.T)`, and `dot(a.T, a)` are mathematically identical when a is symmetric. Symmetry detection therefore allows our transformation to replace all transposes with identity mappings, exposing the three dot products as structurally equivalent. MLIR's

CSE pass can then collapse these into a single dot product, demonstrating how symmetry inference *enables* downstream optimizations that would otherwise be impossible.

```
def dot_cse(x):
    a = x.T + x
    return (jnp.dot(a, a) +
            jnp.dot(a, a.T) +
            jnp.dot(a.T, a))
```

Reduce with Partial Symmetry. This benchmark operates on a $(32, 32, 32, 32)$ tensor and intentionally injects partial symmetry across three of its four axes using a symmetry-preserving transpose pattern. The reduction is performed repeatedly over a symmetric axis, and the tensor is updated in each iteration while maintaining the same symmetry structure. This benchmark evaluates whether the analysis can (i) detect partial symmetry in higher-rank tensors, (ii) propagate this symmetry across iterative updates, and (iii) enable layout-aware optimizations such as axis reordering to improve memory locality during the reduction.

```
def reduce_partial_symmetry(x):
    # Inject partial symmetry: axes (0, 1, 2) become symmetry-equivalent.
    a = x.transpose((3, 1, 2, 0)) + x

    result = jnp.zeros(a.shape[1:])

    for _ in range(20):
        red = jnp.sum(a, axis=0)   # reduction along a symmetric axis
        result = result + red
        a = a + 1                  # preserves partial symmetry

    return result
```

Symmetric Kalman Filter. This benchmark implements a simplified Kalman filter iteration in which all matrices (state covariance, process noise, measurement noise) are symmetric. Each update consists of multiple matrix multiplications and explicit symmetrizations that maintain positive semidefiniteness. Because symmetry is preserved across 100 prediction and update steps, this benchmark evaluates the ability of the analysis to track symmetry through a realistic scientific computing pipeline with dense linear algebra and repeated symmetry restoration.

```python
def kalman_step(P, F, Q, H, R):
    # P, Q, R are symmetric matrices
    P_pred = F @ P @ F.T + Q
    S = H @ P_pred @ H.T + R
    K = P_pred @ H.T @ jnp.linalg.inv(S)
    P_upd = (jnp.eye(P.shape[0]) - K @ H) @ P_pred
    P_upd = 0.5 * (P_upd + P_upd.T)   # explicit symmetrization
    return P_upd
```

5.4 Results and Analysis

Table 1 summarizes our experimental results.

Table 1. Benchmark results showing transpose elimination and performance improvements.

Benchmark	Input Shape	Transposes	Baseline	Optimized	Speedup
Single op	2048×2048	$2 \longrightarrow 1$	5.93 ms	5.81 ms	1.02×
Chained (10×)	2048×2048	$11 \longrightarrow 1$	17.81 ms	3.76 ms	4.73×
Interleaved (10×)	2048×2048	$11 \longrightarrow 1$	13.57 ms	4.11 ms	3.30×
Dot CSE	1024×1024	$2 \longrightarrow 1$	12.88 ms	4.58 ms	2.81×
Reduce partial	$(32, 32, 32, 32)$	$1 \longrightarrow 1$	3.74 ms	3.05 ms	1.23×
Kalman filter	128×128	$902 \longrightarrow 401$	71.44 ms	61.30 ms	1.17×

Our results demonstrate that symmetry-aware optimization achieves speedups ranging from 1.02× to 4.73× across the benchmark suite. The largest improvement occurs in the chained symmetric operations benchmark, where eliminating 10 of 11 transpose operations yields a 4.73× speedup. This illustrates how compounding symmetry detection across iterations enables substantial redundant computation elimination.

The interleaved symmetric operations benchmark achieves a 3.30× speedup, demonstrating that the analysis remains effective even when symmetric operations are intermixed with scalar arithmetic. The dot product CSE benchmark achieves a 2.81× improvement by recognizing that

$$\text{dot}(a, a^T) = \text{dot}(a^T, a) = \text{dot}(a, a)$$

when a is symmetric, enabling aggressive common subexpression elimination.

The partial symmetry benchmark shows a 1.23× improvement by identifying symmetry in higher-rank tensors and performing the reduction axis rotation optimization. The Kalman filter benchmark eliminates 501 of 902 transpose operations (a 55% reduction),

producing a 1.17× speedup. While smaller, this demonstrates that symmetry exploitation remains beneficial in complex workloads where other operations dominate execution time.

Overall, the reduction in transpose operations correlates strongly with performance improvements in the simpler benchmarks, confirming that removing redundant symmetry-related computation directly reduces runtime. The single-operation benchmark shows only a modest 1.02× speedup.

6 Surprises and Lessons Learned

In the early stages of the project, we developed a pass to detect symmetry generation in simple 2D tensor (matrix) programs. Since we are building on top of the actively maintained Enzyme-JAX repository, one surprise was discovering that another contributor had independently implemented a very similar pass around the same time.

This prompted us to coordinate more closely with the Enzyme-AD [4, 5, 7] team and to clearly define our contribution. Together, we identified a natural next step: extending the analysis to a more general N-dimensional partial symmetry pass: an area of broad interest within their team as well. This experience reinforced the value of staying in sync with the community and discussing planned features early. We now attend the EnzymeAD team's weekly meetings at MIT and maintain active communication on Slack to remain aligned, receive feedback, and avoid duplicated work.

7 Related Work

Our system builds on the MLIR system [2] and the StableHLO dialect of OpenXLA. Regarding symmetry exploiting optimizations, a relevant work is SySTeC [6]. However, as explained in more detail in Section 2, SySTeC and our work have different design goals: SySTeC is designed to optimize symmetric kernel code as the scalar/loop level, while our goal in this project was to implement a dataflow analysis at the level of StableHLO.

8 Conclusion and Future Work

In this project, we successfully implemented a novel dataflow analysis for representing, detecting, and generating partial symmetry in MLIR (StableHLO) tensor programs. On our targeted microbenchmarks, this analysis enables performance improvements of up to 4× on CPU backends. Our work provides a foundation for several promising future directions, including extending the analysis with additional symmetry-detection and propagation rules, integrating new optimization passes that more aggressively exploit discovered symmetries, and benchmarking the approach on other hardware targets such as GPUs or TPUs. Another important next step is to evaluate the pass on additional real-world machine learning and

scientific computing workloads beyond the benchmarks we performed, to identify further domains and models that benefit most from symmetry-aware optimization.

9 Distribution of Total Credit

We would like to split the credit evenly.

References

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/jax-ml/jax

[2] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[3] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[4] William S. Moses and Valentin Churavy. 2020. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1046, 14 pages.

[5] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. doi:10.1145/3458817.3476165

[6] Radha Patel, Willow Ahrens, and Saman Amarasinghe. 2025. SySTeC: A Symmetric Sparse Tensor Compiler. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3696443.3708919

[7] Mai Jacob Peng, William S. Moses, Oleksandr Zinenko, and Christophe Dubach. 2025. Sound and Modular Activity Analysis for Automatic Differentiation in MLIR. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 347 (Oct. 2025), 28 pages. doi:10.1145/3763125