Tensor Program Superoptimization through Cost-Guided Symbolic Program Synthesis

Alexander Brauckmann
University of Edinburgh
United Kingdom
alexander.brauckmann@ed.ac.uk

Aarsh Chaube
University of Edinburgh
United Kingdom
achaube@ed.ac.uk

José W. de Souza Magalhães
University of Edinburgh
United Kingdom
jwesley.magalhaes@ed.ac.uk

Elizabeth Polgreen
University of Edinburgh
United Kingdom
elizabeth.polgreen@ed.ac.uk

Michael F. P. O'Boyle
University of Edinburgh
United Kingdom
mob@inf.ed.ac.uk

***Abstract*—Modern tensor compiler frameworks like JAX and PyTorch accelerate numerical programs by compiling or mapping Domain-Specific Language (DSL) code to efficient executables. However, they rely on a fixed set of transformation rules and heuristics, which means they can miss profitable optimization opportunities. This leaves significant optimization potential unused for programs that fall outside these fixed patterns.**

This paper presents STENSO, a tensor DSL program superoptimizer that discovers such missing rewrites. STENSO's core is a symbolic program synthesis based search algorithm that systematically explores the space of equivalent programs. By combining symbolic execution and sketch-based program synthesis, it generates equivalent candidate implementations. To make the search computationally tractable, STENSO further integrates a cost model with a branch-and-bound algorithm scheme. This effectively prunes the search space, arriving at optimal solutions in a reasonable time.

We evaluate STENSO on over 30 benchmarks. The discovered programs achieve geometric mean speedups of 3.8x over NumPy and 1.6x over state-of-the-art compilers like JAX and PyTorch-Inductor. These results underscore the limitations of heuristic-based compilation and demonstrate STENSO's effectiveness in finding such optimizations automatically.

***Index Terms*—Superoptimization, Tensor Programs, Program Synthesis, Compiler Optimization**

I. Introduction

High-performance libraries, frameworks and Domain Specific Languages (DSLs) are a popular way to access the full potential of modern high-performance CPUs, GPUs and other hardware accelerators. This is especially true for numerical computing, where *tensor frameworks* have become popular, because the gap between a naive implementation and an optimized one can be orders of magnitude. Therefore, domain-specific frameworks like NumPy, PyTorch and JAX have become the standard for writing high-performance tensor code.

Each of these frameworks provides a composable, flexible library supporting a wide range of tensor operations. However, this flexibility means that there are many different ways to encode exactly the same computation. While semantically equivalent, each of these encodings may have vastly different performance characteristics when evaluated on different platforms. Ideally, we would have a system that automatically rewrites inefficient tensor programs into better equivalents.

A. Existing Approaches

Rewriting programs to make them more efficient is at the heart of compiler optimization and is extremely well studied.

1) Conventional Compilers: Conventional mainstream compilers, including Tensor DSL frameworks, usually have a fixed set of transformations and heuristics to apply them [1]–[5]. However, these frameworks often miss optimization opportunities, including at the highest level of abstraction, e.g. the tensor DSL level. While existing rules are effective for common cases, they do not cover the space of profitable rewrites and identities. Consequently, user-written programs that do not fall into the space of common cases, may not be optimized. As Psarras et al. demonstrate, many popular Tensor DSL frameworks fail to optimize even common linear algebra patterns [6].

2) Searching Optimization Spaces: Alternative code optimization approaches explore a predefined transformation, rewrite, or schedule space and select the best based on a cost function [7], [8]. While often delivering better performance, they require the creation of a set of transformations or rules before exploration can begin, and there may be profitable rules not considered by the compiler developer.

3) Superoptimization: Instead of applying predefined rules, superoptimization offers a more powerful alternative than the previous two. They search the space of all possible programs to find a program that is both semantically equivalent to the original and has a desired property, such as optimal performance [9]. While this guarantees finding the optimal variant, superoptimization is limited by the combinatorial explosion of the search space, making it computationally infeasible for all but the simplest programs.

B. Our Approach

This paper presents STENSO (Sketch-based TENsor Super Optimizer), a superoptimizer for tensor programs that

979-8-3315-9288-2/26 © 2026 IEEE

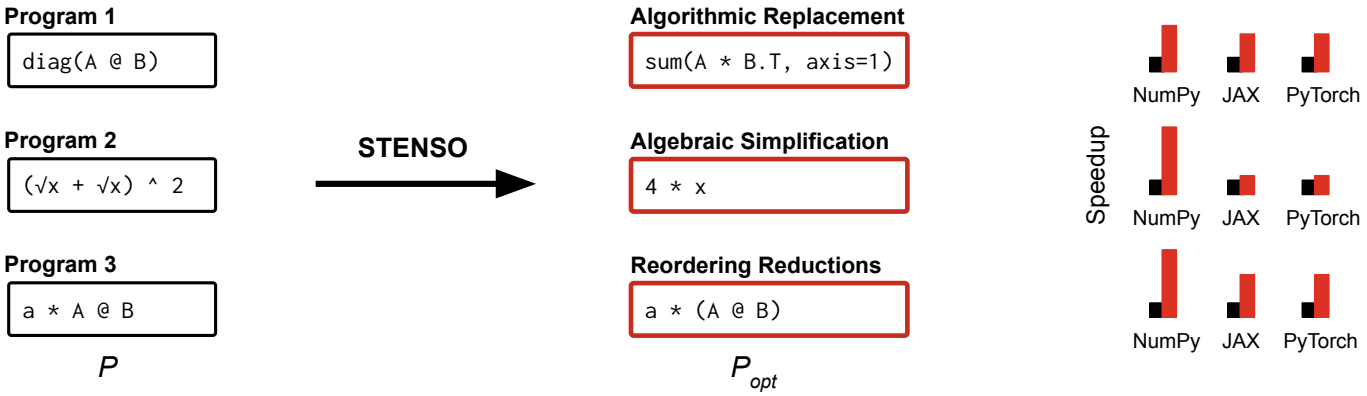

Fig. 1. STENSO enables significant speedups with NumPy, JAX and PyTorch by finding more efficient variants of Tensor DSL programs.

makes the powerful superoptimization technique computationally practical, finding better performing program variants by systematically exploring the space of symbolically equivalent Tensor DSL programs. The core idea behind STENSO is to use symbolic program synthesis to search for computationally cheaper program variants. It uses symbolic execution and sketch-based program synthesis to effectively build a search space of Tensor DSL programs. The search is guided by (1) a simplification objective, which prunes large parts of the search space and guides the synthesis towards tensor program completion, and (2) by pruning the search space further using a cost estimation model, which considers the cost of sketches. These guides effectively manage the combinatorial explosion of the search. Specifically, STENSO uses a branch-and-bound algorithm to prune the space for solutions that are too costly. This combination allows STENSO to systematically explore equivalent programs, while pruning search tree branches that cannot yield a better solution than the optimum already found.

By operating at the high-level abstraction of the tensor DSL, STENSO uncovers powerful, non-obvious rewrites that are missed by conventional tensor frameworks, leading to significant performance improvements.

This paper makes the following contributions:

- The design and implementation of STENSO, a novel superoptimization algorithm that uses symbolic sketching and symbolic algebra solving.
- A new benchmark suite for Tensor DSL program optimization, consisting of 21 real-world and 12 synthetic benchmarks.
- A demonstration that STENSO discovers Tensor DSL rewrites that result in geometric mean speedups of 3.8x for NumPy, 1.6x for JAX and PyTorch.

II. Motivating Examples

Given a Tensor DSL program as input, P, STENSO produces an optimized version of the program as output, denoted P_{opt}, which maintains the same semantics but has reduced computational complexity. To illustrate the potential for such optimizations, Figure 1 shows several examples where high-level optimizations can yield substantial performance gains.

A. Algorithmic Replacements

A potentially very profitable class of tensor DSL level rewrite optimizations is replacing a computationally complex algorithm with a less complex one. For example, a common way of computing the diagonal of a matrix product is $\text{diag}(A \ @ \ B)$. While intuitive, this is inefficient, because the off-diagonal elements are discarded when performing a full product $A \ @ \ B$. A more efficient approach, which can be expressed as $\text{sum}(A * B^T, \text{axis} = 1)$, avoids computing the unused elements by only computing the used diagonal values. This reduces the computational complexity from cubic to quadratic.

B. Algebraic Simplifications

Another class of rewrite optimizations is using simplification rules to simplify an input tensor DSL program. For example, an expression like $(\sqrt{(x)} + \sqrt{(x)})^2$ can be mathematically simplified to $(2 * \sqrt{(x)})^2$, effectively reducing 4 operations to 3 operations. It further simplifies to $4 * x$, a reduction to 1 operation. Such algebraic simplifications reduce the number of required operations, therefore reducing computational complexity.

C. Reordering Reductions

Another powerful class of rewrite optimizations is reordering operations that reduce dimensions. Consider the expression $a * A \ @ \ B$, whereas a is a scalar, A a 2D tensor, and B a vector. A left-to-right evaluation first performs an element-wise multiplication of a with A, before the matrix-vector product. An algebraically equivalent but more efficient variant is to reorder the computation as $a * (A \ @ \ B)$. Because $A \ @ \ B$ results in a vector, the scalar a is multiplied by a vector, instead of a 2D matrix, reducing the number of performed operations.

Evaluating these examples with the tensor DSL frameworks NumPy, JAX and PyTorch reveals that the rewritten forms indeed result in improved performance, suggesting that the rewrites are missing in these tensor DSL frameworks. STENSO aims to automatically discover such rewrites by using program synthesis.

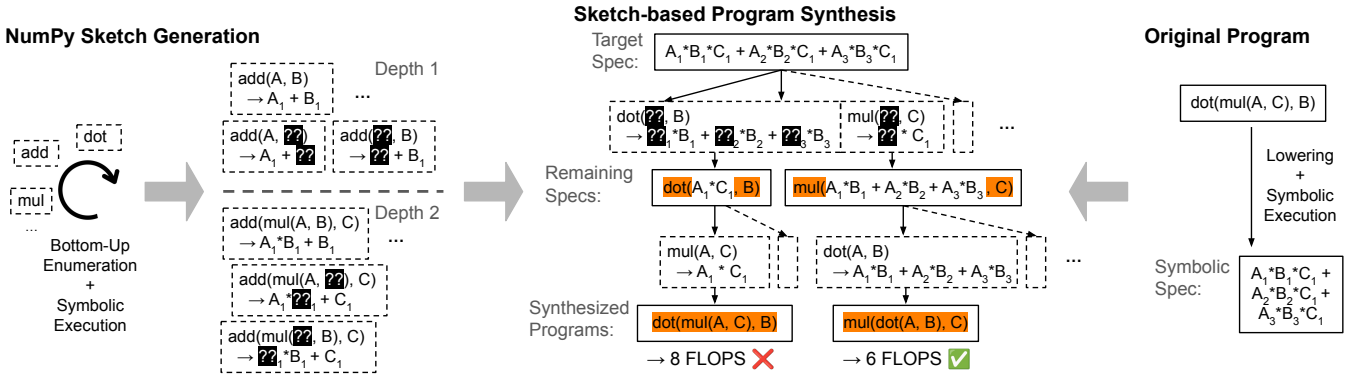

Fig. 2. Overview of the STENSO Synthesis Flow, which generates NumPy DSL sketches using bottom-up enumeration and symbolic execution. STENSO then performs a sketch-based synthesis search to recursively translate the symbolic trace of the original NumPy program into a symbolically equivalent, but computationally more efficient version. Note that spec shows only the leftmost element in the target tensor.

III. Overview

As a tensor program superoptimizer, STENSO takes a tensor program as input and searches for a semantically equivalent, but more computationally efficient, version. It systematically explores the space of possible program implementations using a sketch-based program synthesis approach [10], [11], illustrated in Figure 2. A core contribution of STENSO is its search strategy. While sketch-based synthesis with simple greedy search has been used for code translation [11], this is insufficient superoptimization. STENSO thus integrates a cost model with a branch-and-bound algorithm to make the otherwise exhaustive exploration computationally tractable.

A. Symbolic Execution of the Original Program

The process begins with the *Original Program*, P, in this example `dot(mul(A, C), B)`. This program is first converted into a symbolic mathematical representation through *Symbolic Execution*. The result is a symbolic expression capturing the program semantics, which serves as the *Target Specification*, denoted Φ, for the synthesis stage (In this example, $A_1 \cdot B_1 \cdot C_1 + A_2 \cdot B_2 \cdot C_2 + A_3 \cdot B_3 \cdot C_3$).

B. NumPy Sketch Generation

Concurrently, as shown on the left of Figure 2, STENSO systematically enumerates programs from a given *NumPy Grammar*, consisting of fundamental operations (e.g. `dot`, `add`, `mul`, etc.) to generate sketches. Through a systematic *Bottom-Up Enumeration*, it first generates a set of small program stubs of increasing depth. Each of these stubs is further converted into *sketches* by systematically introducing unknown variables (holes). The stubs and sketches are then symbolically executed to create a library of (sketch, expression) tuples. For example, the simple sketch `add(A, B)` is paired with the symbolic expression A_1+B_1. This library of sketches serves as the building blocks that the main synthesis stage uses to search for NumPy implementations that are symbolically equivalent to the symbolic target expression.

C. Sketch-based Program Synthesis

The core of STENSO is a *Sketch-based Program Synthesis* algorithm, shown in the center, which performs a top-down search to assemble the previously generated sketches in a way that is equivalent to the symbolic target expression. This recursive search is guided by two key metrics: Firstly, we restrict the search space to only consider sketches that *simplify* the specification (Section V-A). Secondly, we guide the search with an estimate of the computational cost of the partial program assembled so far (Section V-B). This integration of cost guidance via a branch-and-bound approach ensures that STENSO produces highly efficient tensor code in a reasonable time.

The search starts with the symbolic execution of the original program as the target specification. It then matches each of the library's sketches to the specification, using a symbolic algebra solver. For instance, consider a target specification $A_1 + B_1$ and a sketch `add(??, B)`. The solver computes what the expression for the unknown `??` would need to be to make the program equivalent to the target specification, which would be A_1. As this specification is simpler than the previous one, this sketch qualifies for further exploration. The process continues recursively – in each step, STENSO searches for a sketch in its library that refines this remaining part of the spec further, eventually finding `add(A, B)`.

In the more complex example in Figure 4, the recursive process eventually synthesized a first program that is symbolically equivalent to the target specification: `dot(mul(A, C), B)`. However, note that other parts of the search tree lead to valid solutions as well, resulting in other candidate programs. As valid candidate programs are found, a cost estimator evaluates their computational efficiency (e.g. in floating-point operations). If, at any point, a branch we are exploring exceeds the best estimated cost we have seen so far, we discard this branch and proceed to the next branch in the search tree. The search continues, finding alternative programs, e.g. `mul(dot(A, B), C)`, which is also equivalent to the target spec but has a lower cost of 6 FLOPS. When the search

Algorithm 1 Core Synthesis Algorithm

```
1: function SYNTHESIZE(P)
2:     cost_min ← ESTIMATE(P)
3:     Φ ← SYMEX(P)
4:     stubs, sketches ← GENSKETCHES(P, G)
5:     score ← GETSIMPSCORE(Φ)
6:     (res, score) ←DFS(Φ, score, 0, stubs, sketches, cost_min)
7:     if score < cost_min then
8:         return res
9:     else
10:        return P
```

tree is fully explored, the program with the minimal cost is returned as the final, optimized output.

In general, sketch-based synthesis can require exploring a large search space. A key contribution of STENSO is to make this search computationally tractable via the use of a cost model and a branch and bound scheme to terminate any search tree paths that are guaranteed not to contain a solution better than the best one found so far.

In the next sections, we will give more detailed descriptions of each of the stages.

IV. Symbolic Execution and Sketch Generation

To synthesize equivalent NumPy programs, STENSO's synthesis algorithm requires several inputs, as shown in Algorithm 1: The symbolic expression of the input program, which serves as the target specification; A set of sketches used as building blocks for the target program; And a cost estimate of the initial program for the branch-and-bound pruning optimization.

A. Program Specification via Symbolic Execution

The primary goal of the synthesis task is to find a program in the NumPy Tensor DSL that is semantically equivalent to an input program. To check for this equivalence without relying on concrete example executions, we first translate the input program into a symbolic representation, which we denote Φ.

Specifically, we lower the NumPy program into a loop-level representation and execute it on SymPy symbols. Instead of concrete numerical values, the input tensors are populated with symbolic variables. For example, input tensors A or B have elements that are symbolic variables $A_{i,j}, B_k$. As the symbolic execution engine processes the operations in the input program, it builds up a single, comprehensive mathematical expression over these input symbols. The result is a symbolic expression, denoted Sym_{in}, capturing the data-flow and computational semantics of the original program, which is invariant to its syntactic form, removing the need for explicit normalization on the syntactic level. For example, a program that computes the element-wise squared difference of two input tensors A and B would produce a symbolic expression algebraically equivalent to $(A - B)^2$. This expression serves as the formal specification that our algorithm will try to synthesize programs for.

```
⟨F⟩ ⊨ np.full(⟨S⟩,⟨F_scalar⟩) | np.triu(⟨F⟩) |
      np.tril(⟨F⟩) | np.sum(⟨F⟩,⟨D⟩) |
      np.transpose(⟨F⟩,⟨D⟩) | np.sqrt(⟨F⟩) |
      np.add(⟨F⟩,⟨F⟩) | np.subtract(⟨F⟩,⟨F⟩) |
      np.multiply(⟨F⟩,⟨F⟩) | np.divide(⟨F⟩,⟨F⟩) |
      np.dot(⟨F⟩,⟨F⟩) |
      np.tensordot(⟨F⟩,⟨F⟩,⟨D⟩,⟨D⟩) |
      np.power(⟨F⟩,⟨F⟩) |
      np.where(⟨B⟩,⟨F⟩,⟨F⟩) | FArg | FCons
⟨B⟩ ⊨ np.full(⟨S⟩,⟨B_scalar⟩) |
      np.triu(⟨B⟩) | np.tril(⟨B⟩) |
      np.less(⟨F⟩,⟨F⟩) | BArg | BCons
⟨ B_scalar⟩ ⊨ BArg_scalar | BCons_scalar
⟨ F_scalar⟩ ⊨ FArg_scalar | FCons_scalar
⟨ D⟩ ⊨ DCons
⟨ S⟩ ⊨ SCons
```

Fig. 3. Grammar of considered NumPy operations. F represents a tensor of floats, B a tensor of bools, F_scalar a float scalar type, B_scalar a boolean scalar type. S is a shape type and D is a dimension type, both referred to as attributes, FArg, BArg, SArg, DArg, FArg_scalar, and BArg_scalar are terminal symbols and represent any input to P of the corresponding type. FCons, BCons, SCons, DCons, FCons_scalar, and BCons_scalar are constants found in the input program and have the corresponding types. Any non-terminal symbol can be replaced by an input to the original function or a constant from the original function that is of the same type.

B. NumPy Sketch Generation

Given the symbolic expression, the next step is to generate a set of NumPy sketches from the target NumPy DSL, defined by the grammar in Figure 3. A *sketch* is a short program from the grammar shown, containing one or more "holes", which are placeholder variables that can be filled in by other expressions, or on the sketch level sketches. This technique effectively structures and reduces the search space of the search algorithm.

The process begins by creating a base set of simple program stubs. These stubs are small, complete programs directly resulting from production rules in the grammar. We generate these stubs using a standard *bottom-up enumerative algorithm* [12], [13]. That is, we initially generate all stubs that are constants or inputs. At each iteration, we generate more program stubs by combining operations with the previously generated stubs.

For each stub, we then generate a corresponding sketch by systematically replacing its variable symbols (i.e., its concrete inputs) with holes. For example, from the stub `np.subtract(A, B)`, we would generate sketches `np.subtract(??, B)` and `np.subtract(A, ??)`, where `??` is a hole to be expanded on.

We limit the depth of the bottom-up search to 2 iterations, which provides a sufficiently expressive set of program sketches for our sketch-based search process to efficiently search. Throughout this enumeration process, we leverage type checking to discard any syntactically or semantically invalid

Algorithm 2 Synthesis via Recursive Sketch Simplification and Branch and Bound

function DFS($\phi, score, level, cost, stubs, sketches, cost_{min}$) ▷ $cost_{min}$ is pass-by-reference.
  ▷ **Base Case**: Find a direct template match.
  $matching \leftarrow \emptyset$
  for $stub$ **in** $stubs$ **do**
    if MATCH($\phi, stub$) **then**
      $matching \leftarrow matching \cup \{stub\}$
  if $matching \neq \emptyset$ **then**
    $best\_match \leftarrow \text{argmin}_{t \in matching}(t.cost)$
    return $(best\_match, best\_match.cost)$
  ▷ **Recursive Case**: Simplify the problem using sketches.
  $best\_program \leftarrow nil$
  $best\_cost \leftarrow \infty$
  $sketches_{explore} \leftarrow$ SOLVE($\phi, sketches$) ▷ Find sketches that could match Φ.
  $sketches_{explore} \leftarrow$ PRUNE($sketches_{explore}, score$) ▷ Prune sketches that do not simplify Φ
  for sk **in** $sketches_{explore}$ **do**
    $cost_{total} \leftarrow cost + sk.cost$, $inputs \leftarrow []$, $success \leftarrow true$
    for $hole$ **in** $sk.holes$ **do** ▷ Recursively synthesize each input (hole) of the sketch.
      if $cost_{total} \geq cost_{min}$ **then** ▷ Branch and bound pruning.
        $success \leftarrow false$, **break**
      $(res, cost_{hole}) \leftarrow$ DFS($hole.spec, sk.score, level + 1, cost_{total}, stubs, sketches, cost_{min}$)
      if $res =$ nil **then** ▷ Abandon sketch if a hole fails to synthesize.
        $success \leftarrow false$, **break**
      $cost_{total} \leftarrow cost_{total} + cost_{hole}$
      $inputs.append(res)$
    if $success$ **then**
      if $cost_{total} < best\_cost$ **then**
        $filled\_sketch \leftarrow sk$ ▷ Fill the sketch with synthesized sub-programs.
        $filled\_sketch.operands \leftarrow inputs$
        $best\_program \leftarrow filled\_sketch$
        $best\_cost \leftarrow total\_cost$
      if $level = 0$ **then** ▷ If a complete program has been assembled...
        if $cost_{total} < cost_{min}$ **then**
          $cost_{min} \leftarrow cost_{total}$ ▷ ...and it's a new best, update the bound.
  return $(best\_program, best\_cost)$

program stubs. The resulting valid stubs are transformed into sketches and provide the raw material for the synthesis algorithm, which will attempt to find sketches that simplify the specification.

V. Top-Down Synthesis with Branch-and-bound

The core of STENSO is the synthesis algorithm described in Algorithm 2, which takes the symbolic specification of the input program, the cost of the input program, and the library of generated sketches to discover a more efficient, equivalent program. The synthesizer employs a top-down, recursive search that systematically explores the space of possible programs. To navigate this vast search space efficiently, the algorithm integrates two key strategies: A simplification objective to guarantee progress and a cost-based branch-and-bound method to prune suboptimal solutions.

A. Recursive Simplification via Sketching

The synthesis process, embodied by the `DFS` function in Algorithm 2, starts with the symbolic expression generated by symbolic execution of the original program as its initial target specification, denoted Φ. The aim is to find a program P_{opt} that behaves the same as Φ, but is computationally more efficient. Formally, $\forall \vec{x}\, P_{opt}(\vec{x}) = \Phi$, where $\vec{x}$ is the list of inputs received by the program, and Φ is the expression obtained by symbolically executing the original program on those inputs.

We assemble P_{opt} iteratively, by nesting sketches. That is, in each recursive step, the algorithm attempts to decompose the current specification by applying sketches from its library.

This decomposition is not a simple pattern match. Instead, for each sketch containing a hole (e.g., `sketch(??, arg_1, ...)`), a symbolic algebra solver is invoked. The solver's task is to determine if there exists a symbolic ex-

pression for the hole ?? that would make the sketch's output semantically equivalent to the current target specification. That is, we look for a sketch such that:

$$\exists \texttt{expr}\ \texttt{sketch}(\texttt{expr}, \texttt{arg\_1}, \ldots) = \Phi$$

If a solution exists, this expression for the hole becomes the new, simpler sub-problem, the "hole specification", and is used as the specification on the next recursive call. Note that a sketch with multiple holes will have a hole specification for each hole. In algorithm 2, the call `SOLVE` returns the list of all sketches from this library that can be made equivalent to the current specification, along with their corresponding hole specifications.

Crucially, the search exclusively considers sketch applications that *monotonically simplify* the specification along a search path. In algorithm 2, the call `PRUNE` returns all sketches whose hole specification(s) are simpler than the current specification.

We estimate how complex a specification is by considering the number of unique program inputs it contains. That is, given a specification Φ, we calculate the specification complexity as $|var(\Phi)|density(\Phi)$, where $|var(spec)|$ is the number of unique program inputs in Φ. $density(\Phi)$ represents the sparsity of the computation, defined as the ratio of non-zero elements to total elements in the symbolic tensor. Operations that mask outputs, such as np.where, reduce the density. By incorporating this metric, the simplification objective supports such masking operations.

A sketch is deemed to simplify the specification ϕ if the average specification complexity of all its hole specifications is less than the specification complexity of ϕ. This simplification objective is crucial, as it ensures that each recursive step makes progress toward a solution.

For instance, if the target specification is algebraically equivalent to $A_1 \cdot B_1 \cdot C_1$, and the synthesizer would test the sketch `mul(??, C)`. The solver would deduce that for this to be a valid decomposition, the remaining specification for the hole must be $A_1 \cdot B_1$. This creates a new recursive call to the `DFS` function with $A_1 \cdot B_1$ as its target. The recursion terminates in the base case, when a remaining specification is simple enough to be matched by a program stub (a hole-free sketch) from the library (lines 3-8 of Algorithm 2).

B. Cost-Guided Pruning with Branch-and-Bound

STENSO's goal is not just to find *any* equivalent program, but the *optimal* one within its search space. This requires a complete search that explores all valid program derivations in the search space. However, an exhaustive exploration of this potentially vastly large search space is computationally infeasible.

To make this search tractable, the algorithm uses a classic *branch-and-bound* pruning strategy [14]. It maintains the cost of the best complete program found so far, denoted as $cost_{min}$. During the recursive search, as a candidate program is constructed from sketches, its estimated execution cost ($cost_{total}$) is accumulated by adding the costs of the used sketches. At each step, before exploring a new branch, the algorithm checks if the current accumulated cost, $cost_{total}$, already exceeds $cost_{min}$. If so, that entire search path is abandoned (pruned). This is because any complete program derived from this path is guaranteed to be more expensive than the best solution already found (line 16). If the search successfully constructs a new complete program whose total cost is lower than $cost_{min}$, the algorithm updates $cost_{min}$ with this new, lower cost and saves the program as the current best candidate, $prog_{min}$ (lines 26-27).

The effectiveness of this pruning depends on an accurate cost estimator. STENSO supports the FLOPS cost estimation model from the JAX tensor DSL framework, as well as a cost model built from performance measurements of each individual sketch, obtained by running it on random inputs of representative sizes. While the FLOPS estimator provides a theoretical cost estimation, the latter provides a more realistic estimate of a program's performance on a given platform, leading to more effective and reliable pruning.

VI. Experimental Setup

This section details the setup for our experimental evaluation. We first describe the benchmarks used, followed by the tensor frameworks we evaluate on, and finally, the hardware and software specifications.

A. Benchmarks

We evaluate our approach on a collection of benchmarks consisting of numeric programs. These are drawn from two sources: Real-world code extracted from public GitHub repositories, as detailed in Table I, and a set of synthetically generated expressions, shown in Table II. For the benchmarks sourced from GitHub, we preserved the tensor characteristics of their inputs, such as their ranks and data types, to ensure they are representative of practical use cases. The combined suite covers a diverse set of domains and varies in complexity, containing expressions with up to six operations.

B. Tensor DSL Frameworks

Our performance evaluation includes three prominent tensor DSL frameworks. NumPy as a representative of a framework with an eager execution model. Further, we evaluate on JAX and PyTorch-Inductor, which are compilers that optimize across operations.

- **NumPy**: A fundamental library for numerical computing in Python. It operates through an eager execution model, where operations are processed statement-by-statement, invoking highly optimized native routines [15]. Consequently, NumPy performs no global analysis or program rewriting.
- **JAX**: A high-performance machine learning framework that uses the Accelerated Linear Algebra (XLA) compiler [3]. Unlike NumPy, JAX captures the computation graph and lowers it to an intermediate representation. XLA then applies a series of compiler passes that utilize pattern matching to perform rule-based rewrites and operator fusion.

TABLE I
GITHUB BENCHMARKS USED TO EVALUATE STENSO.

Benchmark	Computational Pattern	Application Domain	Original Implementation	GitHub
diag_dot	Calculates Gaussian variance reduction.	Astrophysics	`np.diag(np.dot(A, B))`	
elem_square	Calculates differences for L2 norm.	AI/ML	`np.power(A, 2)`	
log_exp_1	Adds two Gaussian probability densities.	AI/ML	`np.exp(np.log(A + B))`	
log_exp_2	Builds up a constraint Gaussian.	Statistical Computing	`np.exp(np.log(A) - np.log(B))`	
mat_vec_prod	Computes total profit for items.	Optimization Algorithms	`np.sum(A * x, axis=1)`	
dot_trans	Calculates rotation matrix for alignment.	Biomechanics	`np.dot(A.T, x.T)`	
scalar_sum	Calculates a weighted statistical moment.	Environmental Science	`np.sum(A * x, axis=0)`	
vec_lerp	Creates a color gradient from distance.	Computer Graphics	`np.stack([(x*a + (1-a)*y) for a in A])`	
euclidian_dist	Calculates Euclidean distance of matrix.	Scientific Computing	`np.sum(np.power(A, 2), axis=-1)`	
common_factor	Combines vectors for smoothing.	Augmented Reality	`A * B + C * B`	
inner_prod	Calculates weighted average ion charge.	Physics	`np.sum(a, b)`	
scale_dot	Computes matrix product with scaling.	Benchmarking	`np.dot(a * A, B)`	
reshape_dot	Kernel of a scientific simulation.	Benchmarking	`np.reshape(np.dot(np.reshape(A, (r,q,1,p)), B), (r,q,p))`	
dot_trans_2	Double transpose of a matrix.	Physics Simulation	`np.transpose(np.transpose(A))`	
power_neg	Element-wise inverse of a matrix.	AI/ML	`np.power(A, -1)`	
sum_sum	Sums a matrix over two axes.	AI/ML	`np.sum(np.sum(A, axis=0), axis=0)`	
sum_stack	Stacks and sums multiple matrices.	Computational Biology	`np.sum(np.stack([A, B, C], axis=0), axis=0)`	
sum_diag_dot	Calculates trace of a dot product.	Audio Processing	`np.sum(np.diag(np.dot(A, B)))`	
max_stack	Stacks and finds element-wise max.	Computational Biology	`np.max(np.stack([A, B], axis=0), axis=0)`	
trace_dot	Calculates trace of a matrix product.	Computer Graphics	`np.trace(A @ B.T)`	
reorder_dot	Computes the quadratic form $x^T Ax$.	Network Simulation	`x.T @ A @ x`	

TABLE II
SYNTHETIC BENCHMARKS USED TO EVALUATE STENSO.

Benchmark	Original Implementation
synth_1	`(A * B) + 3 * (A * B)`
synth_2	`A + B - A - A + B * B - B`
synth_3	`(A + B) / np.sqrt(A + B)`
synth_4	`A + A + B - A - A - B * B`
synth_5	`np.power(np.sqrt(a), 4) + 2 * B`
synth_6	`np.power(np.sqrt(A) + np.sqrt(A), 2)`
synth_7	`np.power(A, 6) / np.power(A, 4)`
synth_8	`A * B + A * B`
synth_9	`np.sum(np.sum(A * x, axis=0))`
synth_10	`np.stack([x * 2 for x in A], axis=0)`
synth_11	`A * A * A * A * A`
synth_12	`A + A + A + A + A`

- **PyTorch**: We evaluate PyTorch-Inductor, PyTorch 2's compiler backend [4]. Like in JAX, the Python program is captured into a graph representation. Inductor then applies passes of graph transformations, including pattern matching and fusion, before compiling to parallel C++ or Triton [16] code for accelerated execution.

C. Cost Model for Branch-and-Bound Pruning

We use the measurement-based cost model, as it more accurately captures hardware-specific performance than the FLOPS model. For instance, it distinguishes between the costs of FLOP-equivalent operations like np.power(A, 2) and A*A, enabling more effective pruning. During a one-time offline phase, we benchmark every generated sketch on the target hardware using representative tensor shapes and store these measurements in a lookup table. During the synthesis search, we do not re-measure performance; instead, the estimated cost ($cost_{total}$) of a partial program is calculated by summing the pre-computed costs of its constituent sketches from this table, providing an estimate of a program's performance on a given platform.

D. Implementation, Hardware, and Software

STENSO is implemented in C++ and Python and builds on several core software libraries. We use JAX [3] and MLIR-HLO [17] to compile NumPy programs into a scalar-level MLIR representation and further use the MLIR compiler infrastructure [2] for program manipulation. The SymPy library [18] is used for symbolic execution, manipulation and solving.

Synthesis experiments were performed on a machine equipped with an AMD Ryzen 9 7950X CPU (32 threads) and 64 GB memory. Performance benchmarks were run on three separate systems: The described AMD system, one with an Intel Core i7-8700K CPU (12 threads) with 32 GB memory and another with an Apple M3 Pro CPU (12 cores) with 18 GB memory. The performance evaluations were ran using the latest software libraries at the time of submission, specifically NumPy v2.2.6, JAX v0.7.1, and PyTorch v2.8.

VII. EVALUATION

In this section, we present the evaluation of STENSO. We start with presenting the overall speedups that the programs

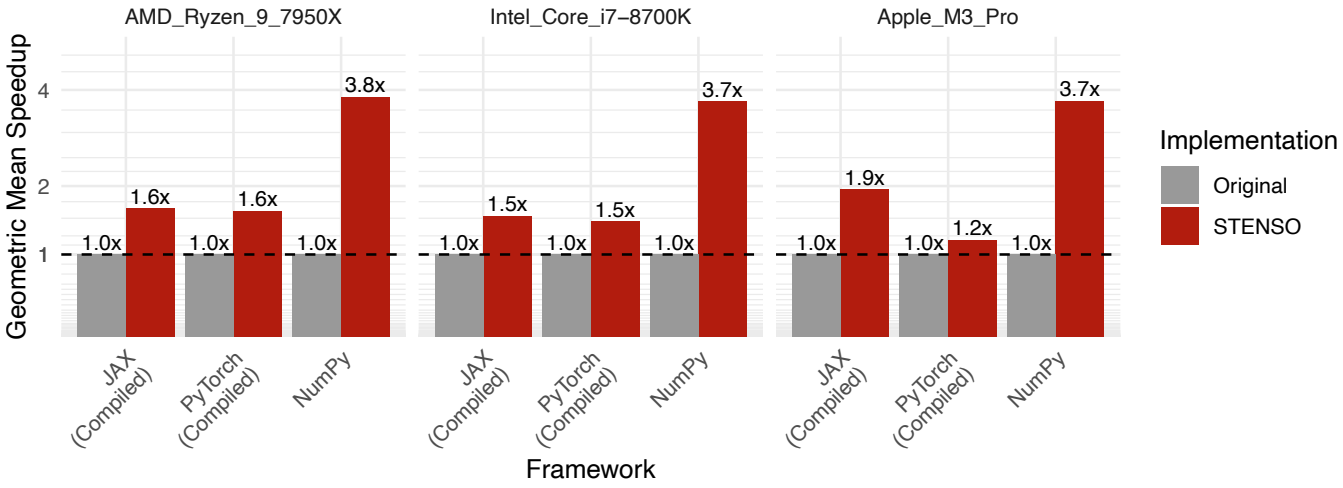

Fig. 4. Geometric mean speedups of programs optimized by STENSO over original implementations on using different Tensor DSL frameworks.

discovered by STENSO achieve, then evaluate the performance of STENSO's synthesis algorithm. We then analyze the discovered programs by grouping them into transformation classes and analyzing the performance impact per class. Finally, we express some of the most impactful discovered rewrites as transformation rules.

A. Speedups of Synthesized Programs

We run STENSO on all of the benchmarks presented in section VI-A and measure the performance of the original and STENSO-optimized implementations. We run these measurements on three platforms - AMD, Intel and Apple based.

Figure 4 shows the geometric mean performance improvements from programs optimized by STENSO compared to their original implementations. The results show that STENSO achieves a consistent speedup across all tested configurations. The most significant performance gains are observed in the NumPy framework, reaching speedups of 3.8x on the AMD CPU, with similar substantial gains of 3.7x and 3.7x on the Intel and Apple platforms, respectively. Compiled frameworks also show notable improvements. For the JAX compiler, the speedup ranges from 1.5× to 1.9×, while for PyTorch, it ranges from 1.2× to 1.6×.

These results show the limitations of several modern Tensor compiler frameworks and demonstrate STENSO's effectiveness in finding powerful optimizations.

B. Synthesis Time

Having demonstrated the runtime performance that STENSO achieves, we will now analyze how quickly STENSO can optimize programs. Figure 5 shows the time required for the search algorithm to terminate, practically speaking, how much time it takes to explore its search space completely. To further evaluate the effectiveness of the branch and bound objective, we run the synthesizer in two configurations. First, only with the simplification objective. Second, with the branch and bound pruning objective, based on the runtime cost model.

The simplification-only synthesizer, while fast on many simpler benchmarks, struggles with more complex programs. It exceeds the synthesis time of the branch-and-bound enabled synthesizer in 1/3 of the benchmarks and fails to synthesize approx. 1/4 of the benchmarks, exceeding the timeout threshold of 10 minutes. Using the simplification objective in combination with branch-and-bound pruning turns out to be highly effective. It makes search more tractable, as results show significantly shorter synthesis times, successfully synthesizing all benchmarks, including the 1/4 of benchmarks that cause the unbounded search to time out. With this optimization, STENSO synthesizes nearly every benchmark in well under 200 seconds. The sole exceptions are diag_dot and vec_lerp, which complete in just over 200 seconds. When we compare the programs found by the simplification-only synthesizer with the branch-and-bound enabled one, we find that the programs are equivalent, showing that solution quality doesn't degrade with the branch-and-bound optimization.

The results show that the branch and bound optimization is critical for making the synthesis algorithm computationally tractable for more complex benchmarks, ensuring it can find solutions within a reasonable time budget, without degrading solution quality.

We further compare STENSO against a baseline approach representative of prior work on tensor program superoptimization: a bottom-up enumerator that is similar to the one used in TASO [19]. As shown in Figure 5, the bottom-up enumerator failed to scale beyond small kernels due to the exponential search complexity. STENSO 's symbolic cost-guided branch-and-bound search consistently synthesized kernels faster, where these baseline failed, while guaranteeing correctness by construction.

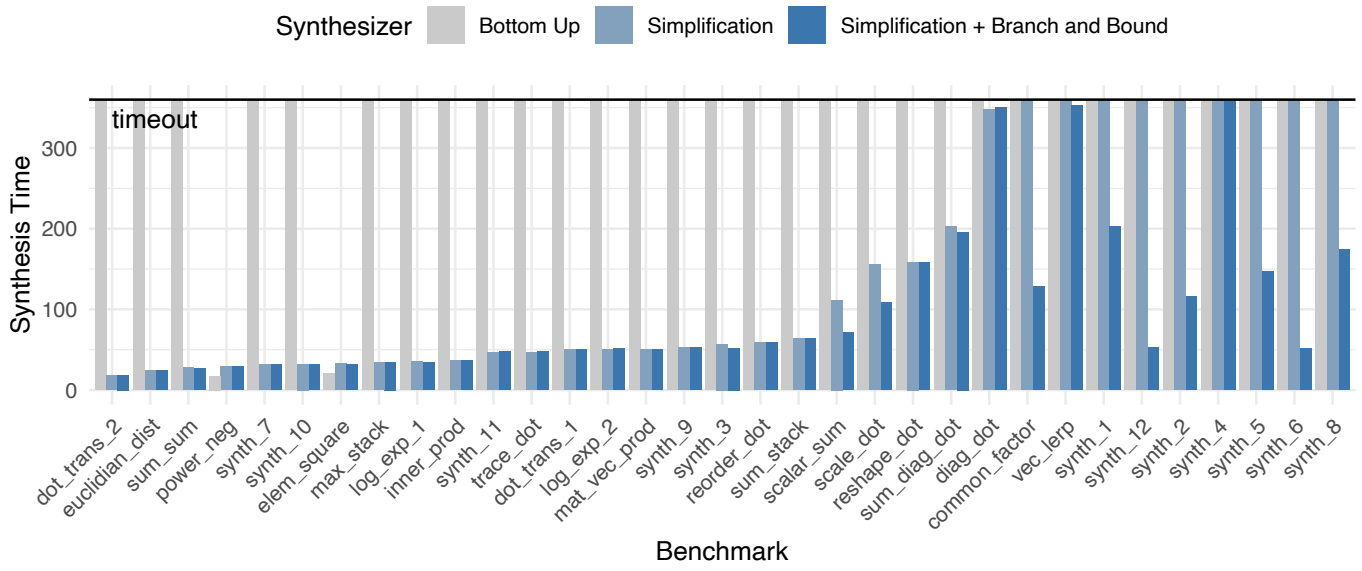

Fig. 5. Synthesis times of STENSO variants and a bottom-up synthesizer as baseline.

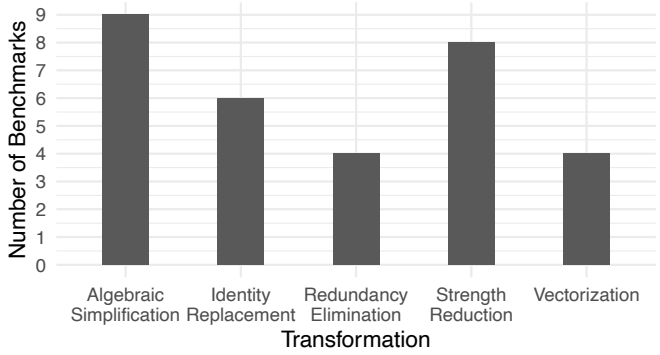

Fig. 6. Number of benchmarks per transformation class.

C. Analysis of Optimized Programs

To better understand the sources of performance improvements, we manually analyze the optimized programs and group them into transformation classes.

In total, we identify five distinct transformation classes, as shown in Figure 6: *Algebraic Simplification*, *Identity Replacement*, *Redundancy Elimination*, *Strength Reduction*, and *Vectorization*. The plot shows the distribution of the benchmarks across these categories. The most common classes are Algebraic Simplification and Strength Reduction, comprising nine and eight benchmarks, respectively.

An analysis of the geometric mean speedup per class reveals significant performance patterns, as shown in Figure 7 for the AMD platform. The Vectorization class yields the most significant improvements, achieving a geometric mean speedup of 10.7x on NumPy, 4.4x on PyTorch, and 2.9x on JAX. This highlights the performance potential of vectorizing iteration-based NumPy programs, as well as STENSO's ability to replace such inefficient code with more effective vectorized equivalents. The Identity Replacement class also delivers substantial gains, with a 6.1x speedup on NumPy, 3.5x for JAX, and 2.1x for PyTorch. This optimization works by substituting combinations of expensive operations with mathematically equivalent but cheaper alternatives. This difference stems from the fact that JAX (via XLA) and PyTorch (via Inductor) already employ sophisticated compiler passes, including their own algebraic rewrites and operator fusion optimizations. These existing strategies significantly optimize the baseline performance and partially overlap with the efficiency gains STENSO provides, narrowing the gap between the baseline and the superoptimized program. However, because these compilers rely on fixed optimization rules, STENSO is still able to identify high-level efficiencies that fall outside their predefined patterns. Thus, even on compiled frameworks, STENSO finds valuable optimizations, such as the 4.4x speedup for PyTorch in the Vectorization class. While geometric means provide a high-level summary, the detailed benchmark results in Figure 8 show the performance that STENSO can achieve on individual benchmarks. For instance, in the Vectorization category, the vec_lerp benchmark achieves a remarkable 16.4x speedup on NumPy. Similarly, log_exp (Identity Replacement) and reshape_dot (Redundancy Elimination) show speedups of 23.6x and 6.1x, respectively. These results show that STENSO can discover optimizations beyond marginal improvements.

D. Recognizing Rewrite Rules

Having classified the synthesized optimized programs, we further analyze them to define rewrite rules that can optimize the input programs. Indeed, we can express some of the profitable rewrite rules discovered by STENSO. Below are several examples:

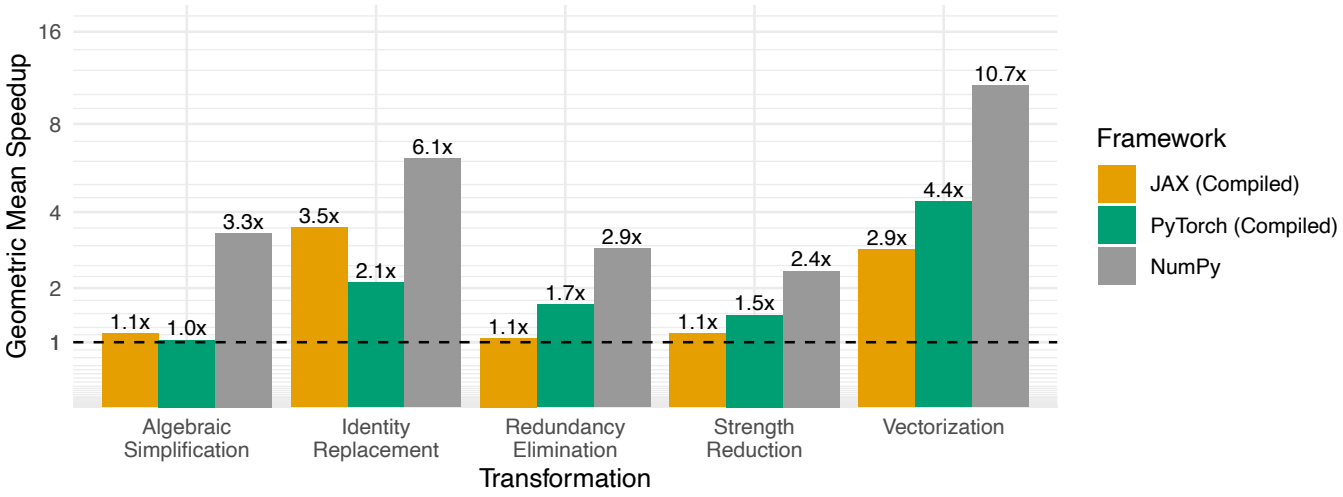

Fig. 7. Geometric mean speedups of programs optimized by STENSO over original implementations by transformation class on AMD platform.

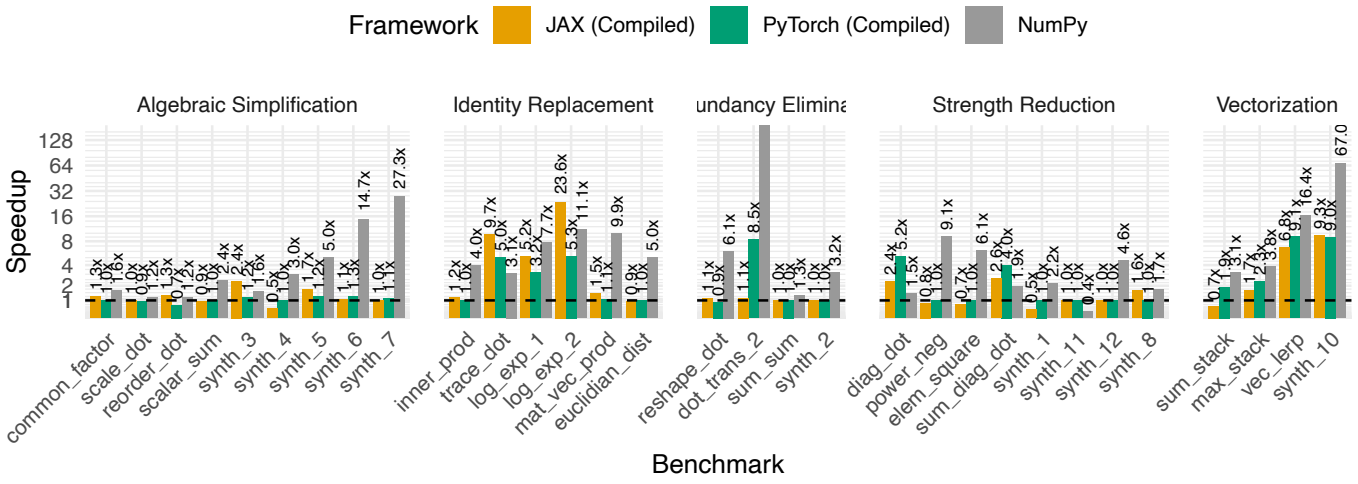

Fig. 8. Detailed speedups of programs optimized by STENSO over original implementations by transformation class.

- *Diagonal Identity Replacement*: This rule avoids a costly full matrix multiplication ($A \cdot B$) by using a mathematical identity to compute the diagonal more efficiently, saving significant compute and memory.

$$\text{diag}(X @ Y) \implies \text{sum}(X \odot Y^T, \text{axis=1})$$

- *Vectorization*: This transformation replaces an inefficient Python loop with a single, highly optimized broadcasted tensor operation, avoiding Python interpreter overhead in eager execution modes and long traces in compiled frameworks. $\odot$ represents any broadcastable, element-wise binary operation (e.g., addition, multiplication, etc.).

$$\text{stack}([(c \odot x) \text{ for } x \text{ in } X]) \implies c \odot X$$

- *Algebraic Simplification*: This rule uses a linear algebra simplification, which reduces the number of floating-point operations.

$$(X)/\sqrt{X} \implies \sqrt{X}$$

These are just some examples of rewrite patterns in programs discovered by STENSO. These rewrites could be added to compilers, enabling significant performance speedups.

E. Limitations and Scalability

While STENSO successfully optimizes complex kernels, its synthesis is computationally demanding, with times reaching approx. 200 seconds for the largest benchmarks. Such high cost is typical for superoptimization, which exhaustively searches for global optima, whereas rule-based compilers settle with local ones. However, this can be seen as a one-time overhead; the resulting optimized kernels are correct-by-construction and can be cached and reused indefinitely, effectively amortizing the synthesis cost over the synthesizer lifetime.

The scalability of STENSO is fundamentally bounded by the search space, which is defined by the grammar (Figure 3) and the sketch enumeration depth. Increasing the depth causes an exponential explosion in the number of sketches, however enables less exploration steps. We found that an enumeration depth of $d = 2$ is the optimal value in this trade-off.

Finally, although the current evaluation focuses on CPUs, STENSO 's architecture is hardware-agnostic. Because the cost model is built on profiling of sketches (Section VI-C), adapting the system to optimize for GPUs or TPUs requires re-profiling the sketch library on the new target hardware.

VIII. Related Work

Recent work has explored tensor program optimization from several angles. Methods using three different paradigms have been explored: Stochastic search, program synthesis, and equality saturation using e-graphs.

a) General Program Superoptimization: Early work in superoptimization for general-purpose IRs relied on stochastic search. STOKE [20], for instance, uses Markov chain Monte Carlo (MCMC) sampling to explore a space of instruction rewrites, often discovering sequences that outperform standard compilers but without offering completeness guarantees. Other search techniques focus on systematically pruning the vast optimization space, for example, Telamon uses branch-and-bound for GPU kernels [21].

A significant advancement came with *program synthesis* methods. Souper [22] pioneered the use of SMT-based synthesis to automatically discover and verify peephole optimizations for LLVM IR. Building on this, recent systems like Hydra [23] generalize concrete rewrites into polymorphic rules, and Minotaur [24] synthesizes optimal, verified SIMD code sequences. The practical applications of program synthesis have also been demonstrated in other domains, such as for generating highly optimized sorting kernels [25]. Program synthesis has also been used extensively for the translation of legacy code into high-level DSLs [11], [26]–[30]. None of these techniques considers searching for best-performing program variants in the target DSL.

Equality saturation offers a more systematic approach, using e-graphs to compactly represent a vast space of program equivalences [31], [32]. Rather than depending on a fixed set of manually written rules, systems like Ruler [33] automate the discovery of compact, expert-quality rewrite rules that can then be used by e-graph–based optimizers.

b) Superoptimization in the Tensor Domain: These foundational techniques have been adapted to the domain of tensor programs. E-graph-based optimizers for tensor graphs, such as TENSAT [34], apply a set of rewrite rules to find efficient program variants. While equality saturation effectively optimizes programs by applying a large set of equivalences, it is fundamentally limited by the completeness of its given rewrite rules. In contrast, STENSO discovers programs from first principles without given rules. Consequently, STENSO is complementary to equality saturation: the novel transformations it discovers can be extracted and added as new rules to e-graph-based systems to enhance their optimization power.

Synthesis-based approaches for tensors, in contrast, often focus on generating a library of rewrites. TASO [19] generates graph substitutions by bottom-up enumerating small neural network subgraphs, while PET [35] extends this to partially equivalent transformations that are later corrected. More recently, Mirage [36] extended this idea to optimize across different representation levels. These approaches produce a static library of transformations to be applied in a separate optimization stage.

STENSO effectively bridges these paradigms. In contrast to bottom-up methods like TASO, which generate small graph substitutions, STENSO scalably and exhaustively explores the implementation space of a given grammar for a given symbolic expression (cf. Figure 5). Furthermore, while e-graph systems are fundamentally limited by defined rule sets, STENSO is not constrained in this way and can discover novel transformation rules. This makes STENSO a complementary approach, as the transformations it discovers can be incorporated into the rule sets of conventional compilers and e-graph-based optimizers to enhance their capabilities.

IX. Conclusion

We presented STENSO, a tensor program superoptimization approach that uses symbolic sketching and cost-driven pruning to discover efficient tensor programs missed by conventional compilation methods. We showed that STENSO discovered more efficient programs, achieving significant geometric mean speedups over a variety of benchmarks, uncovering impactful missing rewrites in state-of-the-art tensor frameworks. Future work will develop hardware-aware cost models and explore integrating the synthesizer into compiler backends to automatically discover rewrites.

Data-Availability Statement

The source code, benchmarks, evaluation, and automated scripting are publicly available on Zenodo, allowing experimentation and reproducibility [37].

Appendix

This artifact provides the complete source code, evaluation benchmarks, and analysis scripts for STENSO, a sketch-based program synthesizer designed for superoptimizing Tensor DSL programs. The artifact includes an automated Docker-based pipeline to:

- Build and run the STENSO synthesizer and a bottom-up synthesizer as baseline.
- Execute synthesis experiments on provided and custom benchmarks.
- Measure key metrics presented in the paper: Synthesis times and speedup of superoptimized programs.
- Reproduce all quantitative results and plots (Figures 4, 5, 6, 7, and 8) presented in the paper.

A. Artifact Check-List (Meta-Information)

- **Algorithm:** Sketch-based program synthesis for Tensor DSL superoptimization.
- **Compilation:** Docker (version `28.3.3`) is used for isolated build/run environment.
- **Dataset:** Benchmarks from open-source software repositories, as described in the paper, as well as synthetic benchmarks.
- **Run-time environment:** Ubuntu 22.04 (in Docker container).
- **Hardware:** Multicore CPU with **32 threads** (e.g., AMD 7950X or equivalent) and **64GB RAM**.
- **Metrics:** Synthesis Time (seconds), Speedup (relative to baseline).
- **Output:** Synthesized program files, CSV metrics, PDF plots (Figures 4–8).
- **How much disk space required (approximately)?:** 30 GB.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** Approx. 24 hours (on 32-thread CPU).
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Current Version:** https://doi.org/10.5281/zenodo.17638076

B. Description

1) How Delivered: The artifact is delivered as a compressed `.tar.gz` archive via Zenodo for persistent storage and is configured to run using a self-contained Docker setup, which simplifies installation and dependency management.

2) Hardware Dependencies: Experiments were run on an AMD 7950X multi-core CPU (32 threads), 64 GB @ 6000 MT/s memory. Similar hardware is required for results similar to the plots.

3) Software Dependencies: Docker is the only required software on the host system. STENSO builds several libraries, such as MLIR, JAX, and SymPy, which are automatically built and installed.

4) Data Sets: Benchmarks are stored as python files in the `stenso/eval/benchmarks` directory.

C. Installation

Create a new directory, download the file from Zenodo, and untar.

```
tar -xzf artifact.tar.gz
```

D. Experiment Workflow

We provide a fully-automated workflow that automatically sets up the docker environment, runs all experiments, and plots the results. Plots and synthesized programs are stored in the `out` directory.

Change into the unzipped `cgo26_stenso` directory. The automated workflow can be invoked with the following command.

```
./run_all.sh
```

This will first build the necessary Docker environment, then run several synthesis experiments for STENSO and a bottom-up synthesizer as baseline. Finally, it executes performance measurements on the synthesized programs and generates the resulting evaluation plots (Figures 4 through 8) and synthesized program files.

E. Evaluation and Expected Result

After running the `run_all.sh` script, results are stored in the host's local `out` directory, which is mounted into the container.

```
ls out
  benchmarks_synthesized/
  fig4.pdf fig5.pdf fig6.pdf fig7.pdf fig8.pdf
```

Synthesized programs are stored in the `benchmarks_synthesized` directory. The PDF should approximately match Figures 4-8, assuming experiments are ran on hardware comparable to the one described. Besides the currently evaluated platform, we plot performance numbers for the three platforms evaluated in this paper for comparison. Specifically, these three platforms are: AMD 7950X, Intel I7-8700K, and Apple M3 Pro.

F. Reusability and Experiment Customization

STENSO is reusable as individual executable Python program, allowing experimentation with custom Tensor DSL programs, and integration in custom compilation flows.

The entry point for running STENSO individually, outside of the evaluation flow is the script `stenso/main.py`. This script provides several configuration options for users to adapt the synthesizer to their needs. Below is an excerpt, summarizing the main options:

```
python stenso/main.py --help
  --program          Source program in Python.
  --synth_out        Output file containing
                     the synthesized program.
  --cost_estimator   Cost estimator to use.
                     Supported: flops, measured
  ...
```

The following shows two example use cases for customization, along with corresponding STENSO invocations.

a) Synthesizing Custom Programs: To synthesize custom programs, STENSO can be invoked directly with the desired source program. Source files must be in Python and run on NumPy arrays.

Assuming the source file is `original.py`, the command below superoptimizes the program, and upon completion saves it as `optimized.py`.

```
python stenso/main.py --program original.py \
                      --synth_out optimized.py
```

b) Configuring the Cost Estimator: STENSO's search algorithm is guided by a cost estimator, which can be configured using the `--cost_estimator` flag. The `flops` estimator corresponds to the FLOP count. The `measuring` option can be used to perform actual runtime measurements for a more accurate cost model, which is useful when optimizing for hardware-specific performance.

To run synthesis using the `measuring` cost estimator on a custom program, omitting the output file to print the result to stdout:

```
python stenso/main.py --program input.py \
                      --synth_out optimized.py \
                      --cost_estimator measured
```

REFERENCES

[1] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[2] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[3] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/jax-ml/jax

[4] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 929–947.

[5] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[6] C. Psarras, H. Barthels, and P. Bientinesi, "The linear algebra mapping problem. current state of linear algebra languages and libraries," *ACM Transactions on Mathematical Software (TOMS)*, vol. 48, no. 3, pp. 1–30, 2022.

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[8] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[9] H. Massalin, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, no. 5, pp. 122–126, 1987.

[10] A. Solar-Lezama, *Program synthesis by sketching*. University of California, Berkeley, 2008.

[11] A. Brauckmann, L. Jaulmes, J. W. de Souza Magalhães, E. Polgreen, and M. F. O'Boyle, "Tensorize: Fast synthesis of tensor programs from legacy code using symbolic tracing, sketching and solving," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 15–30.

[12] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 1–8.

[13] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *International conference on computer aided verification*. Springer, 2013, pp. 934–950.

[14] A. H. Land and A. G. Doig, "An automatic method for solving discrete programming problems," in *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, 2009, pp. 105–132.

[15] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with numpy," *nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[16] P. Tillet, H.-T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[17] "Mlir-hlo," https://github.com/openxla/xla/tree/main/xla/mlir_hlo, 2025, accessed: 2025-09-11.

[18] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh *et al.*, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, 2017.

[19] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 47–62.

[20] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2013.

[21] U. Beaugnon, A. Pouille, M. Pouzet, J. Pienaar, and A. Cohen, "Optimization space pruning without regrets," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 34–44.

[22] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *arXiv preprint arXiv:1711.04422*, 2017.

[23] M. Mukherjee and J. Regehr, "Hydra: Generalizing peephole optimizations with program synthesis," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 725–753, 2024.

[24] Z. Liu, S. Mada, and J. Regehr, "Minotaur: A simd-oriented synthesizing superoptimizer," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1561–1585, 2024.

[25] M. Ullrich and S. Hack, "Synthesis of sorting kernels," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, 2025, pp. 1–14.

[26] J. W. d. S. Magalhães, J. Woodruff, E. Polgreen, and M. F. O'Boyle, "C2taco: Lifting tensor code to taco," in *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2023, pp. 42–56.

[27] Y. Nishida, S. Bhatia, S. Laddad, H. Genc, Y. S. Shao, and A. Cheung, "Code transpilation for hardware accelerators," *arXiv preprint arXiv:2308.06410*, 2023.

[28] A. Brauckmann, E. Polgreen, T. Grosser, and M. F. O'Boyle, "mlirsynth: Automatic, retargetable program raising in multi-level ir using program synthesis," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2023, pp. 39–50.

[29] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama, "Verified lifting of stencil computations," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 711–726, 2016.

[30] Y. Li, J. W. d. S. Magalhães, A. Brauckmann, M. F. O'Boyle, and E. Polgreen, "Guided tensor lifting," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1984–2006, 2025.

[31] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 264–276.

[32] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[33] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, "Rewrite rule inference using equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–28, 2021.

[34] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality Saturation for Tensor Graph Superoptimization," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 255–268, Mar. 2021.

[35] H. Wang, J. Zhai, M. Gao, Z. Ma, S. Tang, L. Zheng, Y. Li, K. Rong, Y. Chen, and Z. Jia, "PET: Optimizing tensor programs with partially equivalent transformations and automated corrections," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 37–54.

[36] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, M. K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia, "Mirage: A {Multi-Level} superoptimizer for tensor programs," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025, pp. 21–38.

[37] A. Brauckmann, "Stenso: Tensor program superoptimization through cost-guided symbolic program synthesis (artifact)," Nov. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17638077