Tensorize: Fast Synthesis of Tensor Programs from Legacy Code using Symbolic Tracing, Sketching and Solving

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

Luc Jaulmes
University of Edinburgh
United Kingdom
ljaulmes@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

Tensor domain specific languages (DSLs) achieve substantial performance due to high-level compiler optimization and hardware acceleration. However, to achieve such performance for existing applications requires the programmer to manual rewrite their legacy code in evolving Tensor DSLs. Prior efforts to automate this translation face significant scalability issues which greatly reduces their applicability to real-world code.

This paper presents *Tensorize*, a novel MLIR-based compiler approach to automatically lift legacy code to high level Tensor DSLs using *program synthesis*. Tensorize uses a *symbolic trace* of the legacy program as a specification and automatically selects *sketches* from the target Tensor DSLs to drive the program synthesis. It uses an algebraic solver to rapidly *simplify* the specification, resulting in a fast, automatic approach that is correct by design. We evaluate Tensorize on several legacy code benchmarks and compare against state-of-the-art techniques. Tensorize is able to lift more code than prior schemes, is an order of magnitude faster in synthesis time, and guarantees correctness by construction.

CCS Concepts: • **Software and its engineering** → **Retargetable compilers**; **Source code generation**.

Keywords: Program Synthesis, Tensor Compilers, Lifting

ACM Reference Format:
Alexander Brauckmann, Luc Jaulmes, José W. de Souza Magalhães, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2025. 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 (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708956

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708956

1 Introduction

High-level, domain specific languages (DSLs) provide easy access to high performance. In the particular domain of tensor computation, highly engineered compiler tool-chains are able to efficiently map programs to heterogeneous hardware.

While compilers can deliver high performance for languages and frameworks such as NumPy [28], PyTorch [41], TensorFlow [1], JAX [18], and MLX [27], they are less successful with lower-level programs written in C and Python. To access greater performance, programmers must manually rewrite their existing code in evolving high-level Tensor DSLs. Manual rewriting or porting is, however, an error-prone and time-consuming activity that prevents the large body of pre-existing legacy code benefiting from emerging specialized accelerators.

Given this barrier, there have been a number of attempts at automatically translating, or *lifting* legacy code to higher-level DSLs. Unfortunately, such approaches are either highly restricted, do not guarantee correctness, or cannot scale to complex tensor programs. Furthermore, they are limited in the source and target programming languages they consider.

1.1 Existing Schemes

Current approaches can be broadly classified into pattern-matching based compilation, bottom-up enumerative program synthesis, and invariant-based verified lifting.

Pattern Matching. MultiLevelTactics [21] is a compiler-based scheme that raises operations written in a lower-level language, here the affine MLIR dialect, into a higher one, Linalg IR. It is well defined, has the advantage of being fast and shows significant speedup on certain benchmarks. This approach, however, requires domain-specific matching rules,

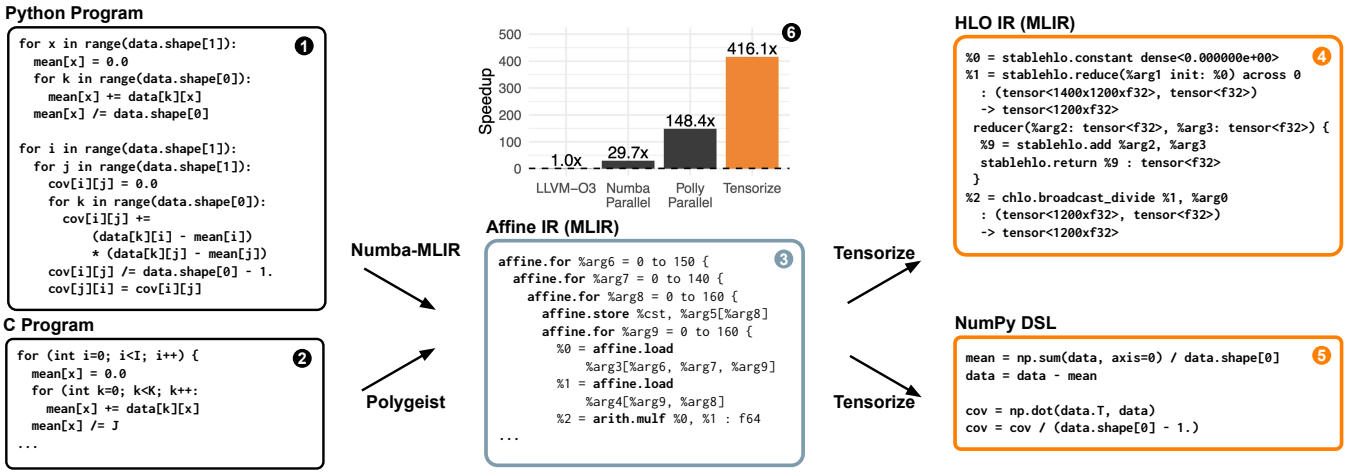

Figure 1. Tensorize lifts C and Python code to Tensor DSLs, such as NumPy or MLIR HLO, enabling significant speedups.

which are also dialect specific. These have to be manually added for each new code pattern and dialect and, as we show in section 8, is both restricted and not a scalable solution.

Enumerative Program Synthesis. There have been several attempts at an alternative approach which searches the target-program grammar to find programs whose behavior matches the low-level program. They employ bottom-up enumeration to generate high-level programs of increasing length [36, 48]. MlirSynth [20] uses this approach within MLIR [35] to lift programs represented in a low-level affine IR dialect to the higher-level HLO IR. These approaches suffer from the fundamental problem of bottom-up enumerative search, namely search time growing exponentially with target program length. Furthermore, input-output examples are used as specifications, which is both costly to evaluate and *cannot guarantee* correctness over all valid inputs. It is therefore not scalable and does not ensure correctness.

Verified Lifting. This is another approach applying program synthesis techniques to lifting, which proves that source and target programs are equivalent by synthesizing an inductive loop invariant as well as the target program. Early work successfully lifted Fortran code to the Halide DSL [31]. Later work considered other domains [5] and reformulated the approach within a new LLVM framework [15]. One drawback is that the user must define all semantics of source and target language within a specialized language, making porting to new targets non-scalable. The most recent work Tenspiler [43] targets tensor DSLs. However, as we show in section 8.3 it also suffers from exponential scalability. In an attempt to mitigate this, Tenspiler relies upon very strong, hand-coded, heuristics to reduce the search space. However, this comes at the cost of decreased generality, significantly reducing the number of programs it can successfully lift to tensor code.

1.2 Our Approach

Tensorize exploits the power of existing MLIR infrastructure to lift currently supported low-level languages into any supported tensor target, without additional user effort. It is highly scalable, utilizing a novel synthesis algorithm powered by an algebraic solver. It is guaranteed correct by construction, using symbolic equivalence to derive target programs. It is robust, does not require hand-crafted search heuristics, and is able to lift more programs than all existing approaches.

Programs, once compiled to MLIR, are symbolically executed to generate an expression that forms our synthesis specification. This novel use of symbolic expression as specification allows the source and target programs to be described in the same representation, enabling algebraic equational reasoning and robust normalization of different syntactic forms of the same computation.

We automatically derive symbolic sketches from the target language/dialect. These are outlines of partial programs that we iteratively refine to a complete program. Given these symbolic sketches and the symbolic expression as specification, we use a solver to determine which of the sketches simplifies the specification the most and choose it as the specification for the next synthesis step. This proceeds until a complete equivalent program is found. Any complete program found is correct by construction.

This recursive sketch and solve approach finds solutions an order of magnitude faster than enumerative synthesis schemes, as it decomposes the synthesis problem into smaller subproblems. Despite the space of possible Tensor DSL programs increasing exponentially with the length of the program, Tensorize, in practice, solves the benchmarks with a runtime that is approximately linear with respect to the number of operations in the target programs, a significant achievement.

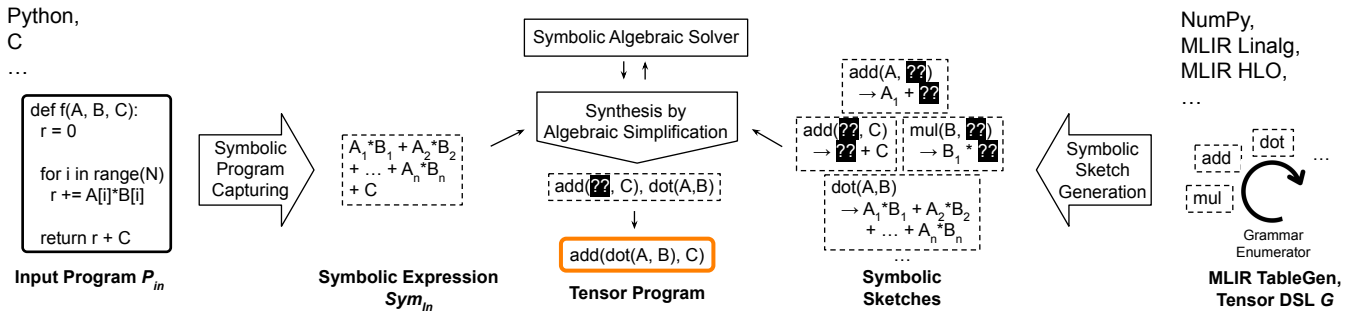

Figure 2. Overview of the TENSORIZE Synthesis Flow.

1.3 Contributions

This paper makes the following contributions:

- A novel scalable synthesis algorithm that uses an algebraic solver to lift programs from symbolic traces.
- A synthesizing, correct by construction MLIR-based lifter targeting IRs and Tensor DSLs.[1]
- A large scale systematic evaluation of state of the art tensor program lifting.
- Faster synthesis times and greater coverage than existing techniques.
- 4.1× to 4102× geo-mean speedups on a range of platforms.

2 Motivating Example

TENSORIZE is implemented as an MLIR tool, which enables it to support various legacy programming languages as input. As an example, consider PolyBench's Python and C implementations [3, 42] of the covariance computation in ❶ and ❷ of Figure 1. While they are straightforward, they are not optimized for performance. To improve efficiency, developers can apply parallelization techniques – either by manually annotating parallelizable loops using frameworks like Numba [33] or through automatic parallelizing compilers like Polly [26].

As shown in ❻ of Figure 1, this results in speedups of 29× (Numba-Parallel) and 148× (Polly-Parallel) over the LLVM -O3 sequential baseline (lowered to LLVM through Clang and Numba) on an AMD Ryzen 9 7950X.

2.1 TENSORIZE: Lifting for Performance

TENSORIZE takes a different approach. Rather than parallelizing and lowering the program to hardware, it first lifts the code to a high-level Tensor DSL. This is achieved in a fully automatic way, by compiling it to MLIR's affine IR using the Polygeist (C) or Numba-MLIR (Python) frontends. The affine IR code ❸ is then *lifted* to an equivalent Tensor DSL program, as shown in ❹ and ❺ of Figure 1. We then leverage the power of Tensor DSL compilers, in this case NumPy, which results in a speedup of 436×.

[1] TENSORIZE source code: https://doi.org/10.5281/zenodo.14095398.

3 Overview

TENSORIZE lifts low-level source programs to high-level Tensor programs based on two key insights. Firstly, we use symbolic execution to generate both the source specification and the potential target program in the same symbolic representation. Secondly, we use program sketches to iteratively *simplify* our synthesis specification. Crucially, we use a solver operating on a common symbolic representation to determine whether a sketch does *simplify* the specification. The approach of TENSORIZE, as shown in Figure 2, consists of three main stages:

Capturing the Symbolic Program. The source program, P_{in} is compiled into MLIR using Polygeist [39] (C) or Numba-MLIR [30] (Python). These frontends generate representations in the affine IR dialect, which models affine static control-flow programs, where loop bounds and conditions are affine relations of the inputs. Only programs that are representable in this dialect are candidates for lifting to a higher-level Tensor DSL. The source program is symbolically executed to obtain an expression, Sym_{in} (Section 4). For example, the Python program P_{in} on the left of Figure 2, which calculates a vector dot product and an addition, is translated into affine IR and symbolically executed to give the symbolic expression: $Sym_{in} = A_1B_1 + A_2B_2 + \cdots + A_nB_n + C$.

This symbolic expression is the source specification for the synthesis algorithm.

Generating Symbolic Tensor DSL Sketches. We generate sketches that are automatically derived from the target Tensor DSL grammar (Section 5). Sketches are short programs with symbolic placeholder variables ?? in place of certain inputs. We symbolically execute each sketch on a combination of symbolic placeholder variables and program inputs to generate a set of expressions referred to as *symbolic sketches*. The placeholder variables are used as extension points for later program synthesis. Having the sketches in the same representation as the target program enables step-wise simplification of the synthesis specification. The generation of sketches is a one-time effort for each target DSL of interest.

$$\begin{pmatrix} cov_{0,0} & \cdots \\ cov_{1,0} & \cdots \\ \vdots & \ddots \\ cov_{m-1,0} & \cdots \end{pmatrix} = \begin{pmatrix} \frac{(A_{0,0}-B_0)(A_{0,0}-B_0)+(A_{1,0}-B_0)(A_{1,0}-B_0)+\cdots+(A_{n-1,0}-B_0)(A_{n-1,0}-B_0)}{C} & \cdots \\ \frac{(A_{0,1}-B_1)(A_{0,0}-B_0)+(A_{1,1}-B_1)(A_{1,0}-B_0)+\cdots+(A_{n-1,1}-B_1)(A_{n-1,0}-B_0)}{C} & \cdots \\ \vdots & \ddots \\ \frac{(A_{0,m-1}-B_{m-1})(A_{0,0}-B_0)+(A_{1,m-1}-B_{m-1})(A_{1,0}-B_0)+\cdots+(A_{n-1,m-1}-B_{m-1})(A_{n-1,0}-B_0)}{C} & \cdots \end{pmatrix}$$

Figure 3. Capturing the symbolic representation of the 2nd part of the covariance computation from Figure 1 box 1 by executing it on symbolic tensors A, B which correspond to `data` and `mean` respectively.

On the right-hand-side of Figure 2, short programs from the target dialects are symbolically executed with placeholder variables to give symbolic sketches. As an example, `add(A,B)` is a program that adds two tensors, $A + B$. If we replace `B` with a symbolic variable or hole ??, then we have a program sketch with a symbolic expression $A +$??, where ?? can be a concrete expression such as $C * D$ or another sketch such as $C *$??.

The symbolic language for sketches is the same as that used for source specification, allowing algebraic simplification during the synthesis.

Synthesis Driven by Algebraic Simplification. The core of Tensorize's synthesis method is a novel top-down synthesis algorithm (Section 6), which uses algebraic solving to determine which sketches are useful. By solving symbolic algebra problems, we incrementally refine the specification by substituting in the symbolic sketch, and then recurse on emerging symbolic hole expressions. This process continues until a complete program is constructed that is equivalent to the specification.

In Figure 2, each of the 4 potential symbolic sketches is matched to the symbolic expression and a symbolic solver is invoked to determine the sketch that most reduces the source expression. This is applied incrementally, first producing `add(??,C)`, then `dot(A,B)`. In combination they result in the Tensor DSL program `add(dot(A,B),C)`.

4 Symbolic Program Capturing

Translating the input program P_{in} into a symbolic representation Sym_{in} enables the incremental synthesis of the equivalent Tensor program.

After initialization, we commence symbolic execution acting on symbolic inputs. We currently restrict attention to static control-flow programs representable in affine IR. Execution begins with the set of initial symbolic inputs and evaluates the program statements using the symbolic variables. At the end of execution, instead of a concrete output value, the result, Sym_{in}, will be a symbolic expression over the input symbols.

As an example consider, the second part of the program in Figure 1, box ❶. The corresponding symbolic execution traces are shown in Figure 3, using symbolic tensors A and B as inputs to `data` and `mean`. On symbolic execution of the triple loop nest, we obtain for instance for cov_{00} the symbolic, algebraic expression $((A_{0,0}-B_0)(A_{0,0}-B_0)+(A_{1,0}-B_0)(A_{1,0}-B_0)+\cdots+(A_{m-1,0}-B_0)(A_{m-1,0}-B_0))/C$ and for $cov_{m-1,m-1}$ the symbolic expression $((A_{0,m-1}-B_m)(A_{0,m-1}-B_m)+(A_{1,m-1}-B_m)(A_{1,m-1}-B_m)+\cdots+(A_{n-1,m-1}-B_{m-1})(A_{n-1,m-1}-B_{m-1}))/C$.

This symbolic expression is the specification of the original program and any lifted program must be algebraically equivalent to it.

5 Symbolic Sketch Generation

Our core algorithm builds on synthesis by sketching [51]. In conventional sketching synthesis, the user provides a sketch, i.e., a program with holes, and the solver searches for expressions to fill the holes such that the program satisfies the provided specification. In our approach, we automatically generate the sketches, and iteratively search for sketches that incrementally simplify the symbolic specification.

5.1 Target Grammar

We use MLIR's TableGen dialect definitions to automatically construct a target grammar that over-approximates the language of valid expressions [20]. This grammar is then enumerated to generate potential target sketches.

The grammar is context-free, and so production rules can be applied to non-terminal symbols, regardless of context. As MLIR dialects are not normally context-free, this grammar over-approximates the space of semantically correct programs. Hence, some generated programs are not valid, and we use MLIR's static checks to filter these out. As an example while `reduce(A) across 5`, is a syntactically correct program, it is semantically incorrect if `A` is a 2D tensor.

The semantics are captured by using existing MLIR compiler passes, that lower the sketches into an abstraction compatible with SymPy's operational semantics, which enables symbolic execution.

Given that NumPy also is an important Tensor DSL, but is currently not available as an MLIR dialect, we have provided a NumPy target grammar as well and use the lowerings from NumPy to MLIR from the JAX compiler [18].

Algorithm 1 Sketch Generation using Bottom-Up Enumeration of Grammar

```
1:  function GenSketches(f, operations, depth = 2)
2:      S ← constants and program inputs
3:      for d = 1 to depth do
4:          for op in operations do
5:              opnds ← filterTypes(S, op)
6:              attrs ← genAttrs(op)
7:              for s in cartesianProduct(opnds, attrs) do
8:                  if not staticCheck(s) then
9:                      continue
10:                 S ← S ∪ s
11:     for s ∈ S do
12:         for x_i ∈ occurrences of x_1, ... x_n in s do
13:             S ← s[x_i/v_1]
14:             for x_j ∈ occurrences of x_1, ... x_n in s do
15:                 if x_i ≠ x_j then
16:                     S ← s[x_i/v_1, x_j/v_2]
17:     Sk ← ∅
18:     for s ∈ S do
19:         Sk ← (s, SymEx(s))
20:     return Sk
```

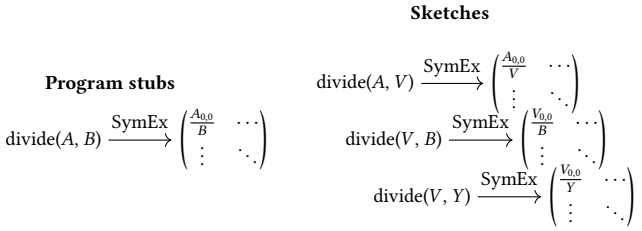

Figure 4. Example generation of symbolic program stubs and sketches for two Tensor DSL operations.

5.2 Sketch Generation

We begin by generating a set of program stubs. A program stub is a short but complete program derived from the grammar containing only terminal symbols. Sketches are created by systematically replacing terminal symbols with placeholder variables.

The sketch generation, detailed in Algorithm 1, starts by generating all program stubs with a parse tree of depth at most 3, i.e., that can be enumerated with no more than 2 iterations of the algorithm's outer loop. Note that this does not limit us to synthesizing only programs of depth 3, because our synthesis algorithm combines multiple sketches together. This enumeration algorithm starts with all program stubs that are constants or inputs. At each iteration, it combines operations with previously generated stubs to generate more complex ones. We then use MLIR's static checks to discard any program stubs that are invalid Tensor programs.

We then use our set of program stubs to generate sketches that contain holes that can be expanded.

Definition 5.1 (Tensor DSL Sketches)**.** We define a sketch to be the result of taking a program stub s and replacing zero or more concrete terminals in the stub with fresh symbols that can be replaced with other program stubs.

As an example consider Figure 4. Here the terminal symbols tensor A and scalar B are replaced with symbolic variables V and Y which act as placeholders to be replaced by other stubs. For instance V could be replaced with $add(x, z)$ and Y replaced with $power(w, 2)$ where x, y, w are scalars or any other program stub. Executing sketches results in symbolic expressions that additionally contain the new placeholder variables, here V and Y.

The symbols that appear in the sketch can be considered as "holes" that can be later expanded. Thus, a sketch is an expression

$$s[x_i/v_i, x_j/v_j, \ldots],$$

where:

- s is a valid expression in the grammar G,
- $\{x_i, x_j, \ldots\}$ is a subset of the program terminal variables, i.e., $x_i, x_j, \ldots \in \{x_1, \ldots, x_n\}$,
- $\{v_i, v_j, \ldots\}$ is a set of fresh symbols (referred to as *placeholder variables*), and
- $e[x/v]$ denotes the result of replacing all occurrences of x in expression e with v

Each sketch is also accompanied by a symbolic sketch that results from symbolically executing the sketch.

For each program stub, we generate all possible sketches that can be generated by replacing the inputs with placeholder variables. We use the symbolic sketches to guide our synthesis but we keep the mapping to the corresponding sketches in the Tensor DSL, in order to reconstruct the Tensor DSL program at the end of the synthesis process.

6 Synthesis by Symbolic Simplification

Instead of searching for a program in the target grammar G that is equivalent to our full input specification Sym_{in}, we incrementally search *simplifying sketches*. A simplifying sketch is one where substituting an expression in place of the placeholder variables results in a simpler expression than the input specification expression.

Our method is similar to classic top-down search methods synthesis [8], but our search is *guided* by selecting sketches that reduce the complexity of the specification.

Example of Sketch Simplification. Figure 5 shows an example of the synthesis algorithm. The input specification is the full symbolic expression of the input program (shown in detail in Figure 3). For simplicity, we show only the left-most element $(A_{0,0}-B_0)(A_{0,0}-B_0)+(A_{1,0}-B_0)(A_{1,0}-B_0))/C$. The synthesis algorithm begins by selecting the sketch `div(X,C)`, which simplifies the expression by removing the divisor `C`,

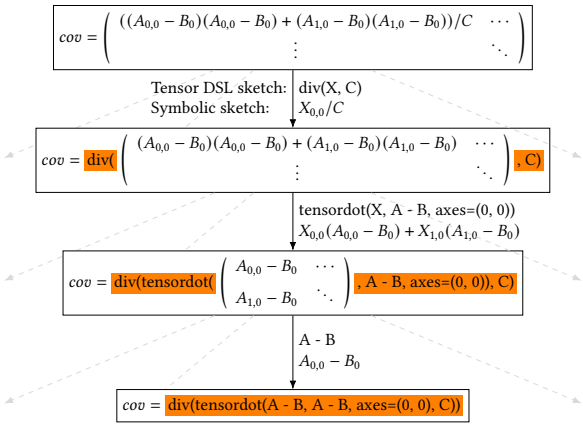

Figure 5. Example expansion tree of the synthesis algorithm. Using the sketches on the edges, the symbolic expressions are incrementally transformed into a Tensor program. Recursive expansion proceeds until symbolic expressions are eliminated.

resulting in $(A_{0,0} - B_0)(A_{0,0} - B_0) + (A_{1,0} - B_0)(A_{1,0} - B_0)$. This simplification produces a partial Tensor DSL program and a reduced specification.

n The algorithm then recurses using the simplified specification, progressively translating the symbolic specification to a Tensor program, until the specification matches a leaf sketch. In this example, the sketch `tensordot(X, A-B, axes=(0,0))` is selected next, as it further simplifies the expression to $A_{0,0} - B_0$. The simplified expression finally matches the symbolic expression of the leaf sketch `A-B`.

The resulting Tensor DSL program is `div(tensordot(A-B, A-B, axes=(0,0),C))`, which is symbolically equivalent to the input program.

Definition 6.1 (Simplifying Sketches)**.** We have a specification of the form $\exists f.\forall \vec{x}.f(\vec{x}) = Spec$, where $Spec$ is an expression that reasons about the variables $\vec{x}$. $Spec$ is derived from symbolic execution of the source program. A symbolic sketch is an expression s that contains the input variables $\vec{x}$ and one or more placeholder variables $\vec{v} = \{v_1, \ldots v_n\}$. A simplifying sketch is a sketch such that:

$$\exists e_1, \ldots e_n,\ s[v_1/e_1, \ldots v_n/e_n] = Spec, \text{ and}$$

$$\frac{1}{n}\sum_{i=1}^{n} |var(e_i)|density(e_i) < |var(Spec)|density(Spec)$$

where:

- $e_1 \ldots e_n$ are expressions that can be used to replace placeholder variables in s to make s equivalent to $Spec$,
- $s[v_1/e_1, \ldots v_n/e_n]$ indicates the result of substituting v_1 for e_1, v_2 for e_2 etc., in the sketch s,
- $|var(e)|$, and $|var(Spec)|$ are the number of unique program inputs in each expression, and

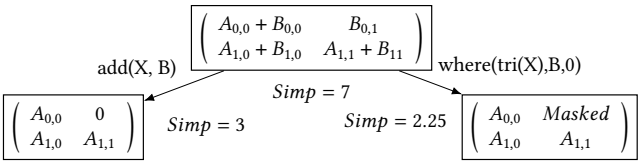

Figure 6. Example of simplifying sketches and their simplification scores.

- $density(e)$ and $density(Spec)$ are the average density values for all tensors/matrices in each expression, i.e., the proportion of elements in all tensor/matrices that are not masked. This is relevant for Tensor DSL operations that apply irregular updates to the specification.

We refer to $Simp(s) = \frac{1}{n}\sum_{i=1}^{n} |var(e)|density(e)$ as the simplifying score for a sketch s. We refer to $e_1 \ldots e_n$ as the simplifying expressions that correspond to the simplifying sketch s. Note that the simplifying expressions will never contain any placeholder variables.

Example of Simplification Score. Consider the example in Figure 6 of a tensor with symbolic expressions as specification. The initial specification on the top has a score of 7, as it has $|\{A_{0,0}, A_{1,0}, A_{1,1}, B_{0,0}, B_{0,1}, B_{1,0}, B_{1,1}\}| = 7$ unique symbols in a full tensor. Applying a addition sketch would result in a simplification score of $|\{A_{0,0}, A_{1,0}, A_{1,1}\}| = 3$, therefore qualifies as a simplifying sketch. The masking sketch however masks one element in the resulting tensor, therefore has a reduced density of $\frac{3}{4}$, resulting in a better score of $|\{A_{0,0}, A_{1,0}, A_{1,1}\}| * \frac{3}{4} = 2.25$.

6.1 Synthesis Algorithm

Our synthesis algorithm iteratively searches for simplifying sketches, and updates the specification using these sketches, until a complete program is found. The core algorithm is shown in Algorithm 2.

Let us consider first the case where each sketch contains only one symbolic variable to be substituted. Let $Spec^i$ denote the specification at iteration i, s^i denote the simplifying sketch found in iteration i, e^i denote the corresponding simplifying expression, and v^i denote the placeholder variable in s^i. We initialize the loop with $Spec = Sym_{in}$, and a set of sketches Sk, generated as per the description in Section 5.

In each iteration, we iterate through all sketches in Sk and check whether they are simplifying sketches. If they are simplifying sketches, we rank them according to their simplifying score. We take the sketch with the lowest value for $Simp(s)$ to be s^i, and its corresponding expression e to be e^i. We then recursively call the synthesis process with an updated $Spec = e^i$, to search for a sketch that simplifies e^i. If the sketch found is complete, i.e., contains no placeholder

Algorithm 2 Core Synthesis Algorithm

```
1: function SYNTHESIZE(f, operations)
2:     spec ← SYMEX(P_in)
3:     sketches ← GENSKETCHES(P_in, G)
4:     prog, history ← SYMSKETCH(spec, sketches, 0, {})
5:     return BUILDAST(prog, history)
```

variables, we return the expression:

$$s^0\left[v^0 \Big/ s^1\left[v^1 / s^2\left[v^2 / s^3[v^3 / \ldots s^i]\right]\right]\right]$$

This is the expression obtained by recursively substituting the symbolic variables with the sketches obtained over all iterations 0 to i, assuming that a complete program was found at iteration i. If no complete program (without holes) is found before we hit a pre-defined depth bound, we backtrack and explore the sketch with the next-best simplifying score.

Sketches with Multiple Holes. The algorithm can be simply extended to sketches with multiple holes, by calculating the *Simp* score across all holes in the sketch, and iteratively handling the holes one by one when we update the specification (cf. Algorithm 3).

6.2 Symbolic Solving

Our approach is enabled by a symbolic solver that can find the expressions $e_1 \ldots e_n$ in the equation $s[v_1/e_1, \ldots v_n/e_n] = Spec$. We first apply a static check that discards sketches that do not contain any overlapping variables and therefore cannot possibly simplify the specification. We then select sketches to consider using simple pattern matching. If no sketch is found at this point, the above query is solved using SymPy [38], a symbolic solver for systems of linear and polynomial equations. SymPy treats all numbers as reals, and so our results are guaranteed to be equivalent based on this assumption but may not preserve IEEE floating point equivalence. Through combining static checks and pattern matching with symbolic equation solving, we are able to optimize the runtime of the synthesis process.

Example of Solver Usage. Consider the specification expression $2A + 2B$ and a sketch $2v$. To find the expression to be in the placeholder of this sketch, we create the symbolic equation: $2A + 2B = 2v$. Solving for v gives: $v = A + B$.

In this case, a symbolic solver is required, because a syntatic pattern matcher would've failed on the syntactic differences.

6.3 Correctness Guarantees

TENSORIZE is sound, i.e., any result returned by the synthesis algorithm is guaranteed to be correct, provided we assume that 1) the symbolic execution accurately and fully captures the behavior of the input program and the sketches; and 2) the symbolic solver called at line 5 and line 10 of Algorithm 3 is sound and never returns an incorrect result.

Algorithm 3 Synthesis Driven by Simplifying Sketches

```
1: function SYMSKETCH(Spec, sketches, depth, history)
2:     if depth > limit then
3:         return nil
4:     for (s, n) in sketches do
5:         if isComplete(s) ∧ s ⊨ Spec then
6:             history ← {history, n};
7:             return (s, history)
8:     Q ← ∅                                   ▷ Queue of sketches
9:     for (s, n) in sketches do
10:        if ∃e_1, ... e_n.s[v_1/e_1, ... v_n/e_n] ⊨ Spec then
11:            Simp ← (1/n) Σ_{i=1}^{n} |var(e_i)| * sparsity(e_i)
12:            if Simp < |var(Spec)| * sparsity(Spec) then
13:                q ← (Simp, (s, n), [v_1 ... v_n], [e_1 ... e_n])
14:                Q ← Q ∪ q
15:    while Q ≠ ∅ do
16:        (Simp, (s, n), vars, exprs) ← Q.pop()
17:        history ← {history, n}, next_h ← {}, prog ← s
18:        Failed ← false
19:        for i ∈ {1 ... n} do
20:            (newExpr, h) ← SYMSKETCH(exprs[i], sketches, depth + 1, history)
21:            if newExpr ⊨ nil then
22:                Failed ← true
23:                history.pop()
24:                break
25:            else
26:                prog ← prog[vars[i]/newExpr]
27:                next_h ← {next_h, h}
28:        if not Failed then
29:            history ← {history, next_h}
30:            return (prog, history)
31:    return nil
```

Assumption 1 holds provided the input program does not contain data-dependent control flow (an exception are selection statements), and the input data structures are large enough to fully capture the behavior of the program. Additionally, note that symbolic execution effectively captures runtime program state, including aliasing and loop-carried dependencies. Assumption 2 holds for real arithmetic, but does not hold for IEEE floating-point semantics, which are also not preserved by aggressive compiler optimization.

7 Experimental Setup

This sections describes the experimental methodology used to evaluate TENSORIZE.

7.1 Benchmarks

We selected the union of all benchmarks used in the most recent closest related works on Tensor DSL lifting [20, 36, 43] with no cherry-picking. The total of 99 C benchmarks can be

grouped into 11 categories as shown in Table 1, all of which can be expressed in Tensor DSLs. They vary significantly in complexity, such as the number of combined operations or the loop depth, thus provide a comprehensive benchmark of the accuracy and scalability of Tensorize and other lifting methods.

7.2 Lifting Methods

To critically evaluate Tensorize, we selected representative alternative lifting approaches and compared against direct LLVM compilation and Polly parallelizing compilation. Each selected lifting method represents the state-of-the-art in pattern matching, bottom-up enumerative synthesis and verified lifting.

- **LLVM-O3:** General-purpose lowering compiler, at highest optimizaiton level [34].
- **Polly:** Polyhedral cache and parallelism optimizing compiler [26]. Reported best of `-polly` and `-polly-parallel`.
- **MultiLevel Tactics:** Pattern matching MLIR lifter [21].
- **MlirSynth:** State-of-the-art bottom-up synthesizer for MLIR dialects [20].
- **Tenspiler:** State-of-the-art verified lifting synthesizer for Tensor DSLs [43].
- **Tensorize:** Our symbolic simplification synthesis approach.

7.3 Tensor DSL Compilers

Once the code is lifted, there are a number of different Tensor compilers available that can target hardware. We use 4 different compilers and evaluate their performance.

- **NumPy:** Runs individual operations as kernel calls. [28].
- **JAX:** Captures computation graph and optimizes it using the Accelerated Linear Algebra (XLA) compiler. [18].
- **PyTorch:** Runs individual operations similar to NumPy, with additional support for GPU execution. [41].
- **PyTorch Compiled:** Captures computation graph and optimizes it with the PyTorch-Inductor compiler [9].

Table 1. Benchmarks used to evaluate lifting methods.

Suite	Workload	Benchmarks
blas [16]	Linear Algebra	3
blend [6]	Image Processing	12
darknet [45]	Machine Learning	14
dsp [29]	Image Processing	15
dspstone [58]	Signal Processing	5
llama [10]	Machine Learning	11
makespeare [46]	Linear Algebra	1
mathfu [11]	Math	12
polybench [42]	Data Mining, Lin. Alg.	15
simpl_array [50]	Array Programming	5
utdsp [47]	Signal Processing	6
TOTAL		**99**

7.4 Methodology

Each method is evaluated for its ability to find an equivalent high-level Tensor DSL program for the C program input, within a 60 second timeout. Tenspiler synthesizes programs in the NumPy DSL; MultiLevel Tactics generates MLIR-Linalg; while MlirSynth generates MLIR-HLO. Tensorize can synthesize both MLIR-HLO and NumPy. These Tensor DSLs are at the same level of abstraction and have a comparable set of operations but to ensure fair comparison between lifters we evaluate all lifted code with the 4 tensor DSL compilers described above. The performance of each approach depends solely on the number of programs lifted, not the Tensor DSL targeted.

7.5 Systems and Software Libraries

All runtime measurements were performed on two systems. The first is an AMD system, with a Ryzen 9 7950X CPU with 32 threads and a Nvidia GTX 1080TI GPU. This system has 32GB of memory, clocked at 6,000 MT/s. The second system has an Intel Core i7-7800K CPU with 12 threads and 16GB of memory, clocked at 4,300 MT/s. All lifting experiments were run on the AMD platform. Both systems run Ubuntu 22.04. For compiler and libraries, we use Clang 18, GCC 11.4, Numba 0.60.0, NumPy 2.0.1, PyTorch 2.4.0 and JAX 0.4.31.

8 Evaluation

We first evaluate the impact of lifting on runtime performance before analyzing the success rate of each lifter.

8.1 Speedups on Benchmarks

The end goal of Tensor lifting is to improve programs runtime performance. Figure 7 shows the geometric mean speedup for the various approaches, on 3 different platforms. For each lifting method, we compile any resulting high level code with the 4 Tensor DSL compilers, described in section 7.3, and present results from the best performing of the 4 compilers to ensure a fair evaluation. A detailed breakdown of backend performance can be found in section 8.5

As an initial, Polly, a parallelizing compiler is able to achieve approx. a 2× speedup on the CPU platforms, outperforming MultiLevelTactics on both platforms and Tenspiler on the Intel platform.

As we move to the AMD CPU and the NVIDIA GPU, the three synthesizers, Tensorize, Tenspiler and MlirSynth are able to achieve increasing levels of performance. This is due to the amount of hardware parallelism available which the Tensor DSL compilers are able to exploit.

Tensorize consistently provides the highest speedups, achieving an overall geometric mean speedup of 4.1× on Intel, 11.7× on AMD, and 4102× on NVIDIA. This equates to between a 1.7× and 4.3× improvement over the next best

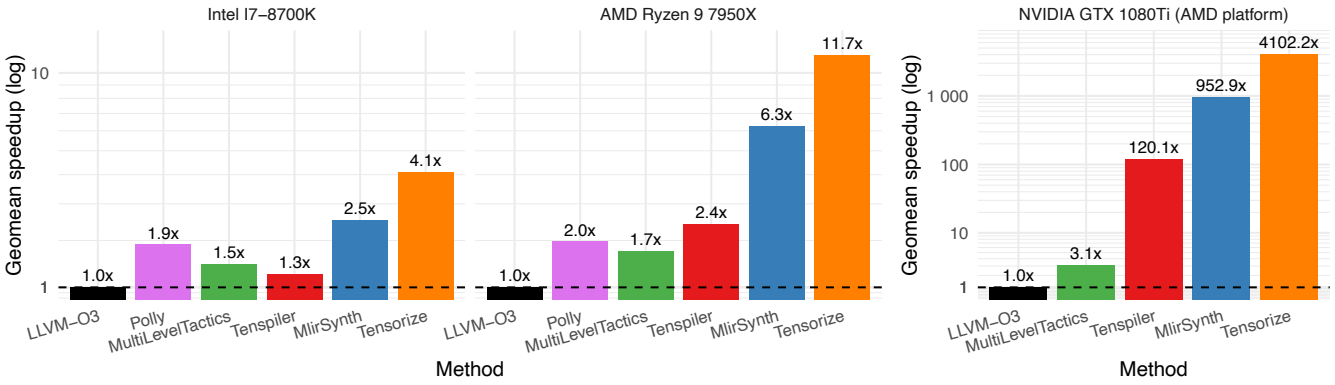

Figure 7. Speedups of lifting methods and TENSORIZE over LLVM-O3 on different compute devices. For each lifting method we show the performance achieved when using the best backend Tensor DSL compiler available.

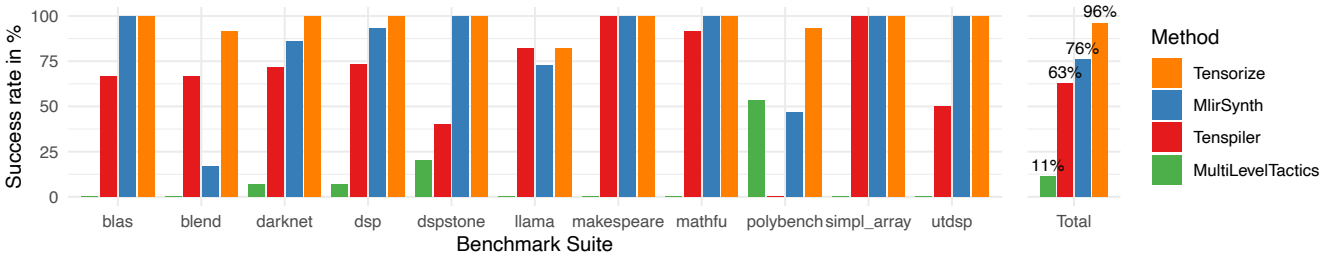

Figure 8. Success rates of lifting methods and TENSORIZE on 11 benchmark suites within a 60 second timeout.

performing scheme MlirSynth. The relatively poorer performance of the other lifting schemes, is due to their failure to lift enough programs which is detailed in the next section.

8.2 Synthesizer Performance

The different performance of lifting-based methods is due to varying success rates on the benchmarks, as shown in Figure 8. MultiLevelTactics achieves the lowest success rate, lifting just 11% of the benchmarks. This low success rate is due to incompleteness of pattern-matching rules – a common disadvantage of rule-based methods. Further, they miss cases, where operations are need to be composed, i.e. where one operation enables another operation. In contrast, the synthesis-based methods are more robust, as they work on the semantics of a program, resulting in significantly higher success rates.

Tenspiler, a verified lifting based method, successfully lifts 63% of benchmarks, but it misses significant ones. In particular, it fails on tensor contractions, as found in dspstone and polybench, for which it does not have appropriate hand-coded heuristics. These are needed to reduce its exponentially growing search space to a tractable size.

MlirSynth, a method based on bottom-up enumerative synthesis, is able to lift 76% of benchmarks. However, it

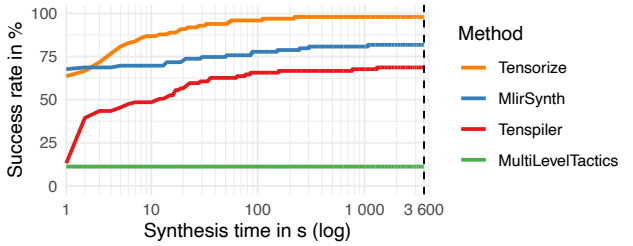

Figure 9. Number of successfully lifted benchmarks over a 1 hour time frame.

struggles to scale to the more complex benchmarks found in blend and polybench, as we show in the next section.

TENSORIZE achieves the highest success rate, lifting 96% of benchmarks. It misses `dissolve_blend_8` and `gesummv` because the algorithm looses time in backtracking. If the timeout was increased slightly, it would successfully lift those, increasing the success rate to 98% It fails on two other benchmarks `transformer_part_1` and `transformer_part_2` because of currently unsupported program structures.

Increasing Timeout. To explore the methods further improvement potentials, we increase the 60 second synthesis

Table 2. Statistics of Tensorize's synthesis process, summarized for each benchmark suite. Exploration steps refers to number of `symSketch` function calls in Algorithm 3.

	Exploration Steps			Exploration Backtracks			Sketches Pattern Matched			Sketches Algeb. Solved			Synthesis Time (s)		
Suite	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
blas	1	3	2.0	0	0	0.0	1	4	3.0	0	25	13.0	0.02	2.08	1.28
blend	1	6	2.8	0	73	6.1	1	20	6.9	0	4239	511.7	0.0	112.2	19.7
darknet	1	3	1.4	0	1	0.1	1	4	1.6	0	192	40.9	0.01	8.88	2.01
dsp	1	1	1.0	0	0	0.0	1	1	1.0	0	0	0.0	0.00	0.07	0.02
dspstone	1	1	1.0	0	0	0.0	1	1	1.0	0	0	0.0	0.03	0.13	0.08
llama	1	5	1.9	0	0	0.0	1	8	2.5	0	82	16.8	0.01	4.68	1.29
makespeare	1	1	1.0	0	0	0.0	1	1	1.0	0	0	0.0	0.02	0.02	0.02
mathfu	1	3	1.2	0	0	0.0	1	4	1.3	0	162	14.7	0.01	2.99	0.28
polybench	1	287	26.9	0	96	8.4	1	38	6.9	0	5094	525.0	0.02	220.78	29.13
simpl_array	1	2	1.4	0	0	0.0	1	4	2.2	0	14	5.4	0.02	3.96	1.59
utdsp	1	1	1.0	0	0	0.0	1	1	1.0	0	0	0.0	0.00	0.09	0.04

timeout to 60 minutes and show the success rates in Figure 9. Very few additional benchmarks are lifted by the other methods, never approaching Tensorize's success rate.

8.3 Scalability Analysis of Synthesizers

To evaluate the scalability of the different synthesizers, we analyze their synthesis times based on the complexity of the source and target programs as shown in Figure 10.

Loop Depth. The maximum loop depth of the source input program is a key complexity factor for programs containing tensor contractions which operate on multi-dimensional tensors. As seen in Figure 10, Tenspiler is unable to scalably synthesize when input programs contain loop nests greater than 2, a significant limitation. Hand-crafted heuristics begin to fail when source programs become more complex. In contrast, MlirSynth and Tensorize scale well with loop depth of the source program.

Tensor Program Length. The length of a Tensor program, defined as the number of operations, is a critical target program metric, as longer programs are potentially more profitable to lift. The plot shows that MlirSynth and Tenspiler are able to scale to DSL programs up to lengths 4 and 5 respectively, with decreasing success rates as complexity increases. MlirSynth's bottom-up enumeration scales exponentially with target program length, which is reflected 0 in the experimental data. Tensorize scales well even on long programs, which is enabled by its incremental divide-and-conquer style synthesis technique.

Overall and in contrast to other methods, Tensorize scales with source and target complexity making it the overall most scalable lifting method.

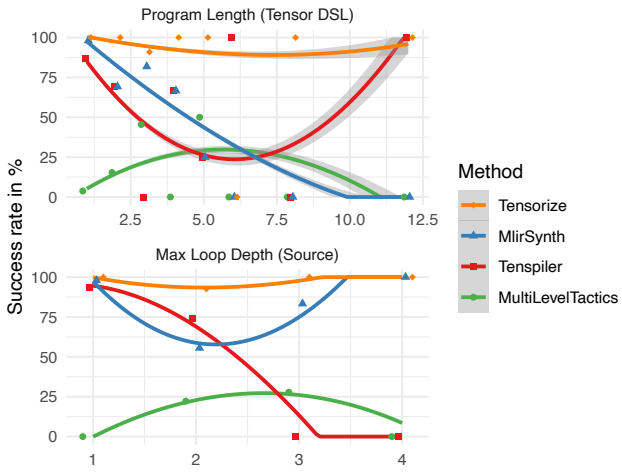

Figure 10. Scalability of Tensorize, Tenspiler, and MlirSynth in different complexity metrics with smoothed trend lines fitted to the data points.

8.4 Tensorize Synthesis Algorithm

Table 2 shows statistics of Tensorize's synthesis algorithm. The algorithm works optimally for 96 out of 99 benchmarks, directly leading to the solution without requiring backtracking. In the remaining 3 benchmarks, it initially explored a local optimum before successfully finding the global one. This leads to synthesis times that are in practice linear with the program length. Most simplification is done through the algebraic symbolic solver, especially in the cases of more complex benchmarks. Only a minority of simplification is done by pattern matching. An exception are the benchmarks

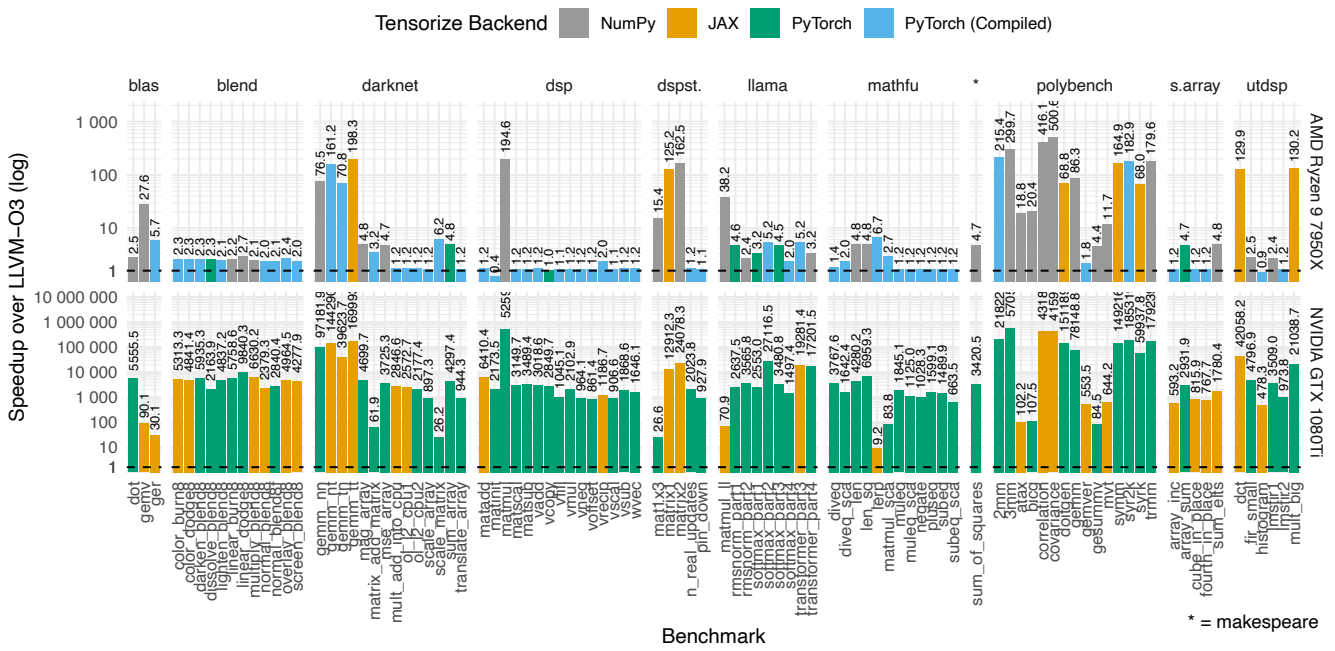

Figure 11. Detailed speedups of fastest TENSORIZE backend over LLVM-O3 on different compute devices.

suites dsp, dspstone and utdsp that contain simple benchmarks, that can are pattern matched in only one exploration step, resulting in synthesis times of under one second.

8.5 Impact of Different Backends

One of the main benefits of lifting legacy code, is that it allows the use of a wide range of Tensor compilers that are optimized for target hardware.

Figure 11 shows the speedups achieved by the best performing Tensor DSL backend for each benchmark on the AMD / NVIDIA platform. The detailed results show that lifting is profitable in 97 out of the 99 benchmarks, with the only exceptions being computationally very simple benchmarks. For benchmarks that contain reductions and tensor contractions, the NumPy and JAX backends give the highest speedups. This is because these backends map to pre-optimized kernel libraries. JAX is further able to optimize across operations, e.g. by fusion, enhancing performance further in several cases. PyTorch Compiled performs best in benchmarks that are less computationally demanding and involve writing results back to externally allocated memory (pass-by-reference), which JAX lacks because its tensors are immutable. The bottom of Figure 11 shows the GPU speedups achieved relative to a LLVM-O3 CPU baseline on the AMD platform, whereas only the computation, not the memory transfer is timed. The results demonstrate substantial speedups in the majority of benchmarks when executed on GPU. The evaluated JAX's and PyTorch's GPU backends use highly parallel execution and pre-optimized kernel libraries, which are very profitable to use, especially for computationally intensive tasks such as tensor contractions.

9 Related Work

Program Synthesis. Program synthesis is the task of synthesizing programs that satisfy a given specification. A common way of tackling this is enumerative search, guided by syntactic templates [7, 14, 25, 40, 53].

A popular way of reducing the search space is to use sketches provided by the user, guessing complete programs and checking them against the specification [52]. Our approach generates sketches, automatically checks iteratively whether partial programs might be correct, and never has to check a complete program against the full specification.

Dillig et al use a symbolic solver to check whether there exists a completion to a partial program that would satisfy the specification, and use that information to guide the search [22, 24]. CEGIS-T [2] synthesizes program sketches and then uses a symbolic solver to fill holes in the sketch, but these holes are limited to constants, and so, unlike our technique, sketches cannot be nested. DeepCoder [13] uses a neural network to predict program sketches, which are then filled in with enumerative synthesis, but their approach, in contrast to our symbolic reasoning, requires a significant amount of training data.

Enumerative Synthesis in the Tensor Domain. Synthesizing tensor programs is considerably more challenging than the list or string processing problems classically studied

in program synthesis. Recent work has explored bottom-up synthesis which finds programs that are equivalent on concrete input/output (IO) examples using enumerative search. TF-Coder [48] raises IO examples to the TensorFlow DSL, ranking candidates with neural networks. MlirSynth [20] raises programs from lower-level MLIR dialects to higher-level ones, using an algorithm similar to TF-Coder and heuristics derived from static program analysis. C2TACO [36] similarly raises programs from C to the TACO DSL [32]. In contrast, Tensorize synthesizes a *symbolically* equivalent program in the tensor language. This equivalence holds for all possible inputs, not just specific examples, providing a robust and sound approach to program synthesis.

Verified Lifting. The use of program synthesis to derive programs from a specification has widely been explored before [23, 49]. The approach of using a low-level program as the specification and targeting a high-level DSL was realized by Helium, a tool transforming loop implementations of stencils into the Halide DSL [37, 44]. This concept was further applied to different domains, e.g. Casper, which lifts sequential Java implementations to MapReduce paradigms that can be efficiently executed using Hadoop [4, 5].

This approach has been applied to the tensor domain where Tenspiler [43], synthesizes a loop invariant and translates it to a Tensor DSL program. To mitigate exponentially scaling synthesis times, Tenspiler uses strong heuristics that limit its generalization. In contrast, our approach scales over complex programs by solving the synthesis problem through incremental simplifications.

Compiler Optimization with E-graphs. E-graphs have proved effective in representing large rewrite spaces efficiently. Even so the size and number of terms grows exponentially with program depth [54, 57].

Tensorize achieves scalability by using a symbolically guided search. Further, e-graphs require the source and target to be in the same language, and our problem necessarily requires them to be in different languages, so to use e-graphs we would need to translate both into a common representation. Additionally, in contrast to e-graphs, Tensorize does not need a fixed set of user-provided rewrite rules, as it relies on symbolic equivalences.

Compiler Optimization for Tensor Programs. Polyhedral analysis has been widely applied to optimize tensor programs, targeting data access locality for more efficient use of cache hierarchies, and targeting for parallelism [17, 26]. This has also been applied to GPU code generation [12, 55, 56]. A recent work, Polygeist enables polyhedral optimizations for C code through the MLIR compiler [35], making them available to further use cases.

While polyhedral optimizations enable schedule optimizations on the loop statement level, Tensorize enables a larger class of optimizations by raising to high-level DSLs, such as high-level algebraic rewriting, fusion, and usage of vendor-optimized kernel implementations.

10 Conclusion

This paper presented Tensorize, a program synthesizer that automatically lifts C and Python code into equivalent Tensor DSL counterparts, using a novel program synthesis method based on symbolic traces, sketches and simplification.

We show that Tensorize's symbolic synthesis method scales well, in practice linearly for most benchmarks, with the complexity of synthesized programs, outperforming all previous techniques.

Future work will investigate data-dependent control-flow and explore the use of machine learning for sketch generation and selection in order to further reduce synthesis time.

Data-Availability Statement

The source code, benchmarks, evaluation, and automated scripting are publicly available on Zenodo, allowing experimentation and reproducibility [19].

A Artifact Appendix

A.1 Abstract

This artifact contains the source code of Tensorize, evaluation benchmarks, and the scripts used to generate the plots shown in Figures 7-10. We provide an automated pipeline to build Tensorize, run synthesis experiments, and plot results.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** A sketch-based program synthesis for lifting code to Tensor DSLs.
- **Compilation:** Docker is used to build and run the artifact. In our tests, we used Docker version `26.1.3`.
- **Data set:** Benchmarks from related work and open-source software repositories are used: PolyBench, blas, blend, darknet, dsp, dspstone, llama, mathfu, makespeare, simple_array, utdsp.
- **Run-time environment:** Ubuntu 22.04 as OS.
- **Hardware:** Modern Multi-core CPU (In our experiments, an AMD 7950X with 32 threads) and at least 32 GB memory.
- **Metrics:** Synthesis time, Speedup over baseline.
- **Output:** Synthesized program files, CSV for metrics, PDF for plots.
- **How much disk space required (approximately)?:** 30 GB.
- **How much time is needed to prepare workflow (approximately)?:** 5 minutes.
- **How much time is needed to complete experiments (approximately)?:** approx. 2 hours on a CPU with 32 threads.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Data licenses (if publicly available)?:** Please refer to licenses of individual benchmark suites.
- **Development repository:** https://github.com/alexanderb14/tensorize

- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.14095398

A.3 Description

A.3.1 How Delivered. The artifact is provided as a zip archive on Zenodo for persistent storage and versioning. A Docker-based setup allows straightforward installation and execution, which is described in A.4 and A.5.

A.3.2 Hardware Dependencies. Experiments were conducted on an AMD 7950X multi-core CPU (32 threads), 64 GB @ 6000 MT/s memory. Similar hardware is recommended to obtain results similar to the plots.

A.3.3 Software Dependencies. Docker is the only required software on the host system. Tensorize mainly builds on MLIR, JAX, and SymPy, which are automatically installed. For a non-Docker setup, which is useful for development purposes, use the following commands to first build the dependencies and then build the Tensorize project.

```
./build_tools/build_dependencies.sh
./build_tools/build.sh
```

A.3.4 Data Sets. Benchmarks are stored as MLIR files in the `benchmark` directory, categorized by benchmark suite.

A.4 Installation

Download the file from Zenodo using a web browser, then unzip it.

A.5 Experiment Workflow

a) Fully Automated. The following command automatically sets up the docker environment, runs the synthesis experiments, and plots the results. Plots and synthesized programs are stored in the `out` directory.

```
./run_all.sh
```

b) Manual. Alternatively, for a step-by-step setup, start with building the docker image:

```
docker build -t tensorize-artifact -f Dockerfile .
```

Then, create and mount an out directory on the host, and start an interactive terminal session. We recommend creating and mounting the `out` directory on the host OS, so it can access and open the plots using host system software.

```
mkdir out
docker run -v $(pwd)/out:/root/out \
           -it tensorize-artifact
```

Inside the container, start the synthesis experiments with:

```
python benchmark/run.py
```

Finally, results can be plotted with the below command, which will produce PDF files in the `out` directory.

```
Rscript benchmark/plot.R
```

A.6 Evaluation and Expected Result

The resulting plots in the `out` directory should match Figures 7-10, assuming experiments are run on hardware similar to ours.

```
ls out
  figure_7.pdf   figure_8.pdf  figure_9.pdf
  figure_10.pdf  stats.csv     synth
```

A.7 Reusability and Experiment Customization

Tensorize is reusable in various aspects. Users can synthesize custom programs beyond the evaluated benchmarks, target alternative domain-specific languages (DSLs) other than the evaluated NumPy DSL, and integrate Tensorize as a component within their MLIR-based compilation flows.

The entry point for running Tensorize individually, outside of the evaluation flow, is the script `tensorize/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 tensorize/main.py --help
  --program   Specifies the MLIR file of the source
              program to synthesize.
  --synth_out Specifies the output file for the
              synthesized program.
  --target    Defines the target DSL.
              (default: numpy)
  ...
```

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

Synthesizing Custom Programs. To synthesize custom programs outside the evaluated benchmarks, invoke Tensorize directly with the desired source program. Source files can originate from other MLIR-based tools or be modified versions of the benchmarks available in the `benchmark` directory.

Assuming the source file is `original.mlir`, the command below synthesizes an equivalent NumPy DSL program, which is then saved as `synth.py`.

```
python tensorize/main.py --program original.mlir
                         --synth_out synth.py
```

Synthesizing Programs in a Different Target DSL. To target a DSL other than NumPy, use the `-target` parameter. Besides the evaluated NumPy, we also tested the MLIR HLO dialect, which is the input language for XLA, the backend compiler used in TensorFlow and JAX. HLO integrates seamlessly with the MLIR compiler infrastructure, making it a good fit for building end-to-end MLIR-based compilation flows.

For example, synthesizing the file `original.mlir` into the MLIR HLO dialect and saving the synthesized program as `synth.mlir` can be achieved with the following command.

```
python tensorize/main.py --program original.mlir
                         --synth_out synth.mlir
                         --target hlo
```

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283. https://www.tensorflow.org/

[2] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 270–288. https://doi.org/10.1007/978-3-319-96145-3_15

[3] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. *PolyBench/Python*. https://doi.org/10.5281/zenodo.4471345

[4] Maaz Bin Safeer Ahmad and Alvin Cheung. 2017. Optimizing Data-Intensive Applications Automatically By Leveraging Parallel Data Processing Frameworks. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1675–1678. https://doi.org/10.1145/3035918.3056440

[5] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1205–1220. https://doi.org/10.1145/3183713.3196891

[6] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Trans. Graph.* 38, 6, Article 204 (Nov. 2019), 13 pages. https://doi.org/10.1145/3355089.3356549

[7] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*. Springer Berlin Heidelberg, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[8] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[9] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. https://doi.org/10.1145/3620665.3640366

[10] Llama Cpp Python Authors. 2023. *llama-cpp-python*. https://github.com/abetlen/llama-cpp-python

[11] Mathfu Authors. 2015. *Mathfu*. https://github.com/google/mathfu

[12] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

[13] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *ICLR (Poster)*. OpenReview.net. https://openreview.net/pdf?id=ByldLrqlx

[14] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 227 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428295

[15] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 38:1–38:30. https://doi.org/10.4230/LIPIcs.ECOOP.2023.38

[16] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[17] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[18] 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/google/jax

[19] Alexander Brauckmann. 2024. Tensorize: Fast Synthesis of Tensor Programs from Legacy Code using Symbolic Tracing, Sketching and Solving (Artifact). https://doi.org/10.5281/zenodo.14095398

[20] Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael F. P. O'Boyle. 2023. mlirSynth: Automatic, Retargetable Program Raising in Multi-Level IR Using Program Synthesis. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 39–50. https://doi.org/10.1109/PACT58117.2023.00012

[21] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive Raising in Multi-level IR. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 15–26. https://doi.org/10.1109/CGO51591.2021.9370332

[22] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 587–610. https://doi.org/10.1007/978-3-030-53291-8_30

[23] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual synthesis for static parallelization of single-pass array-processing programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 572–585. https://doi.org/10.1145/3062341.3062382

[24] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. https:

//doi.org/10.1145/3192366.3192382

[25] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[26] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.

[27] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. 2023. *MLX: Efficient and flexible machine learning on Apple silicon*. https://github.com/ml-explore

[28] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[29] Texas Instrument. 2015. *Texas Instrument Digital Signal Processing (DSP) Library for MSP430 Microcontrollers*. https://www.ti.com/tool/MSP-DSPLIB

[30] Alexander Kalistratov Ivan Butygin, Diptorup Deb. 2023. *numba-mlir: MLIR-based numba backend*. https://github.com/numba/numba-mlir

[31] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 711–726. https://doi.org/10.1145/2908080.2908117

[32] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 943–948. https://doi.org/10.1109/ASE.2017.8115709

[33] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[34] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. https://doi.org/10.48550/arXiv.2002.11054 arXiv:2002.11054 [cs]

[36] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2023. C2TACO: Lifting Tensor Code to TACO. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2023)*. Association for Computing Machinery, 42–56. https://doi.org/10.1145/3624007.3624053

[37] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code. *SIGPLAN Not.* 50, 6 (jun 2015), 391–402. https://doi.org/10.1145/2813885.2737974

[38] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103. https://doi.org/10.7717/peerj-cs.103

[39] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[40] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA. https://dl.acm.org/doi/10.5555/3454287.3455008

[42] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. (2012). https://www.cs.colostate.edu/~pouchet/software/polybench

[43] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2024. Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:28. https://doi.org/10.4230/LIPIcs.ECOOP.2024.32

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[45] Joseph Chet Redmon. 2014. *Darknet*. https://github.com/pjreddie/darknet

[46] Christopher D. Rosin. 2019. Stepping stones to inductive synthesis of low-level looping programs. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 292, 9 pages. https://doi.org/10.1609/aaai.v33i01.33012362

[47] Mazen AR Saghir. 1998. *Application-specific instruction-set architectures for embedded DSP applications*. Citeseer.

[48] Kensen Shi, David Bieber, and Rishabh Singh. 2022. TF-Coder: Program Synthesis for Tensor Manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 2 (2022), 1–36. https://doi.org/10.1145/3517034

[49] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 2014. Modular Synthesis of Sketches Using Models. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*. 395–414. https://doi.org/10.1007/978-3-642-54013-4_22

[50] Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 364–381. https://doi.org/10.1007/978-3-319-66706-5_18

[51] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems*, Zhenjiang Hu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

[52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[53] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

[54] Jonathan Van Der Cruysse and Christophe Dubach. 2024. Latent Idiom Recognition for a Minimalist Functional Array Language Using Equality Saturation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 270–282. https://doi.org/10.1109/CGO57630.2024.10444879

[55] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[56] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[57] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf

[58] Vojin Zivojnovic. 1994. DSPstone: A DSP-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994* (1994), 715–720.

Received 2024-09-12; accepted 2024-11-04