Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations

Jie Qiu
Pittsburgh, PA, USA

Colin Cai
University of California, Berkeley, CA, USA

Sahil Bhatia
University of California, Berkeley, CA, USA

Niranjan Hasabnis
Intel Labs, Menlo Park, CA, USA

Sanjit A. Seshia
University of California, Berkeley, CA, USA

Alvin Cheung
University of California, Berkeley, CA, USA

Abstract

Tensor processing infrastructures such as deep learning frameworks and specialized hardware accelerators have revolutionized how computationally intensive code from domains such as deep learning and image processing is executed and optimized. These infrastructures provide powerful and expressive abstractions while ensuring high performance. However, to utilize them, code must be written specifically using the APIs / ISAs of such software frameworks or hardware accelerators. Importantly, given the fast pace of innovation in these domains, code written today quickly becomes legacy as new frameworks and accelerators are developed, and migrating such legacy code manually is a considerable effort.

To enable developers in leveraging such DSLs while preserving their current programming paradigm, we present Tenspiler, a verified-lifting-based compiler that uses program synthesis to translate sequential programs written in general-purpose programming languages (e.g., C++ or Python code that does not leverage any specialized framework or accelerator) into tensor operations. Central to Tenspiler is our carefully crafted yet simple intermediate language, named Tensir, that expresses tensor operations. Tensir enables efficient lifting, verification, and code generation. Unlike classical pattern-matching-based compilers, Tenspiler uses program synthesis to translate input code into Tensir, which is then compiled to the target API / ISA. Currently, Tenspiler already supports **six** DSLs, spanning a broad spectrum of software and hardware environments. Furthermore, we show that new backends can be easily supported by Tenspiler by adding simple pattern-matching rules for Tensir. Using 10 real-world code benchmark suites, our experimental evaluation shows that by translating code to be executed on 6 different software frameworks and hardware devices, Tenspiler offers on average **105**× kernel and **9.65**× end-to-end execution time improvement over the fully-optimized sequential implementation of the same benchmarks.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Program Synthesis, Code Transpilation, Tensor DSLs, Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.32

Related Version *Full Version*: https://arxiv.org/abs/2404.18249 [29]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.10.2.17

© Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung;
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 32; pp. 32:1–32:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work was supported in part by DARPA Contract FA8750-23-C-0080, a Google BAIR Commons project, NSF grants IIS-1955488, IIS-2027575, ARO W911NF2110339, ONR N00014-21-1-2724, and DOE award DE-SC0016260, DE-SC0021982, and the Sloan Foundation.

Acknowledgements We would like to thank Jayaram Bobba and Zhongkai Zhang from Intel's Habana team for inputs on Gaudi architecture, TPC-C programming model, and obtaining high-performance from TPC kernels. We would like to thank Hasan Genc and Sophia Shao for helpful insights into Gemmini code generation.

1 Introduction

We have witnessed an explosion of new computational infrastructures for tensor computation in recent years: from software frameworks such as TensorFlow to specialized hardware accelerators like tensor processing units. Such infrastructures arise due to new application domains such as image processing and training deep learning (DL) models, and they often expose their functionality via various domain-specific languages (DSLs) that range from specialized instruction sets such as vectorized instructions to high-level programming interfaces such as Apple's MLX [27] or TensorFlow's XLA [37].

To leverage the optimization offered by such infrastructures, applications must be written against the provided programming interfaces: developers must first master each DSL's programming model to write new code, and existing applications must be rewritten. This problem is recurring as new DSLs keep appearing targeting different application domains. Manually rewriting existing applications is tedious and increases the likelihood of introducing bugs. The classical way of addressing such issues is to build transpilers [15, 5, 6, 19, 25, 26] that translate code from paradigms developers are familiar with (e.g., C++ code using the STL library) to the one provided by the target DSL (e.g., NumPy API). Nonetheless, building such a transpiler is resource-intensive, error-prone, and each one is specialized to a specific target DSL. For instance, existing compilers such as Dexter [6], STNG [19], and C2TACO [25] target specific DSLs like Halide and TACO, and are not easily extensible to support new operators or backends. While recently developed DL models such as GPT have shown promise in code translation, they do not provide any guarantees on the correctness of output. Moreover, GPT fails to generate even syntactically correct code for DSLs it has not seen in training data, limiting its applicability to new or less popular DSLs.

In this paper, we describe a tensor compiler that addresses these challenges. We introduce Tenspiler– a compiler designed to automate the transpilation of code to *multiple* tensor processing frameworks and hardware accelerators. Tenspiler uses verified lifting [12] (VL), a technique using inductive program synthesis to infer provably equivalent program summaries expressed using a user-defined intermediate representation (IR), and generate executable code from the synthesized summary to the target DSL. In contrast to conventional compilers that rely on pattern-matching to compile code, VL uses a search-based technique for the translation process. The two key steps of VL are:

- **Search Phase:** This stage lifts the input code to an equivalent program written using a user-provided IR, where the IR is used to model the functionality of each operator in the target DSL. Lifting is formulated as a syntax-guided synthesis [8] problem.
- **Verification:** Once lifted, a theorem prover is used to validate if the synthesized summary is functionally equivalent to the input. If so, executable code is produced by calling the user-provided code generator from the summary; otherwise, another summary is generated by the search phase.

The key to making lifting efficient lies in the design of the IR (i.e., how the target DSL is modeled). In prior work [5, 6, 7, 34, 21], each function or instruction exposed by the target DSL is modeled explicitly. While doing so makes the search efficient, such explicit modeling makes the compiler hard to extend to other DSLs. With TENSPILER, we introduce, for the first time, a *single unified* IR, TENSIR, that is designed for tensor operations and can easily generate code to *multiple* tensor processing software frameworks and hardware accelerators. Surprisingly, TENSIR is a small language based on tensor algebra that includes commonly used vector and matrix operations. While other unifying IR exists (e.g., MLIR [23]), they are targeted for classical pattern-matching compilers. As we will discuss in Sec. 5 and Sec. 4.2, TENSIR instead is designed for synthesis-based compilers and thus aims to enable both efficient search and verification.

In summary, this paper makes the following contributions:

1. We describe the design of TENSIR for transpiling code to tensor processing DSLs. TENSIR is simple yet expressive enough to model the functionalities provided by different software frameworks and hardware accelerators, and enables efficient code transpilation using verified lifting, as detailed in Sec. 3.
2. Based on TENSIR, we devise various optimization techniques to make synthesis and verification tractable, and scale to real-world programs in Sec. 5.
3. We implement TENSPILER, a verified lifting-driven transpiler built using TENSIR as the modeling language. We demonstrate the effectiveness of TENSPILER by using it to lift real-world code from 10 different suites to **6** different open-source and commercially-available tensor processing software frameworks and hardware accelerators. We illustrate the ease of constructing such transpilers by building one for MLX, a new tensor processing framework that was released only four months ago, using less than **200** lines of code in Sec. 6.

We have released Tenspiler's code on `https://github.com/tenspiler/tenspiler`.

2 Overview

TENSPILER takes in C/C++ or Python code as input[1] and transpiles it to a functionally equivalent program that leverages different software frameworks and hardware accelerators (details described in Sec. 4.3) for tensor computation. As mentioned, TENSPILER uses verified lifting to first translate the source program into TENSIR. Unlike traditional pattern-matching compilers, TENSPILER formulates code translation as a search for a program expressed in TENSIR that is provably semantic-equivalent to the input. Doing so avoids the need to devise pattern-matching rules and prove their correctness. To make the search scalable, instead of directly searching within the DSL exposed by each target, we designed a high-level IR called TENSIR that abstracts away the low-level implementation details of each DSL operator and captures only their semantics, unifying various DSLs into a common set of tensor operators. TENSPILER uses a program synthesizer (currently Rosette [36], a synthesizer for finite domain theories) to lift the input code to TENSIR during the search phase. The synthesized output is then verified using a theorem prover (currently, an SMT solver, CVC5 [10]) for the unbounded domain. "Unbounded domain" means the verification

[1] TENSPILER currently supports a subset of the C/C++ and Python language (in particular it does not support code that uses pointers or objects, which we have not encountered such use in our benchmarks). It also expects any external libraries used in the input to be functionally modeled, which is how TENSPILER currently supports code that uses the `STL::vector` library.

```
1 inline uint8_t screen_8x8 (uint8_t a, uint8_t b) { return a + b - (a * b) / 255; }
2 vector<vector<int>> screen_blend(vector<vector<int>> b, vector<vector<int>> a) {
3   vector<vector<int>> out; int m = b.size(); int n = b[0].size();
4   for (int row = 0; row < m; row++) {
5     vector<int> r_v;
6     for (int col = 0; col < n; col++)
7       r_v.push_back(screen_8x8(b(col, row), a(col, row)));
8     out.push_back(r_v);}
9   return out;}
```

(a) Original Blend function in C++.

```
1 def t_t(x, y, operation):
2   if len(x) < 1 or len(x) != len(y): return []
3   else: return [operation(x[0], y[0])] + t_t(x[1:], y[1:], operation)
4
5 def t_s(x, a, operation):
6   if len(x) < 1: return []
7   else: return [operation(x[0]), a] + t_s(x[1:], a, operation)
```

(b) Operators in Tensir. We represent `tensor_scalar` as `t_s` and `tensor_tensor` as `t_t`.

```
1 def inner_loop(row, col, b, a, r_v, out):
2   return col >= 0 and col <= len(b[0]) and row >= 0 and row < len(b) and
3     r_v == t_t(t_t(b[row][:col], a[row][:col], +),
4                t_s(t_t(b[row][:col], a[row][:col], *), 255, /), -) and
5     out == t_t(t_t(b[:row], a[:row], +), t_s(t_t(b[:row], a[:row], *), 255, /), -)
6
7 def outer_loop(row, col, b, a, row_vec, out):
8   return row >= 0 and row < len(b) and
9     out == t_t(t_t(b[:row], a[:row], +), t_s(t_t(b[:row], a[:row], *), 255, /), -)
```

(c) Synthesized loop invariants.

```
1 def screen_blend(b, a): return b + a - b * a // 255 # NumPy/TensorFlow/PyTorch/MLX
2 uchar256 Screen8x8(uchar256 a, uchar256 b) { # TPC-C implementation for Gaudi
3   uchar256 c = v_u8_mul_b(a, b) * v_reciprocal_fast_f32(255);
4   uchar256 d = v_u8_add_b(a, v_u8_sub_b(b, c));
5   return d; }
```

(d) Generated executable code for different tensor processing DSL.

Figure 1 End-to-End example of using Tenspiler to transpile code.

is performed for all possible program states, not just a bounded set of states (e.g., all states where integers are represented using 8 bits) that Rosette considers during the synthesis phase. Once verified, Tenspiler then translates the Tensir program to the concrete syntax of the target DSL using simple pattern-matching rules.

We illustrate Tenspiler using the example in Fig. 1a as our `S` (source), where `S` implements blending, a common image processing operation. It lightens the base color by iterating over each pixel, implemented as a nested loop over all the `rows` and `cols` in the image. Our goal is to transpile this code to the target DSLs supported by Tenspiler as shown in Fig. 1d.

Tenspiler first translates the input code to Tensir. To be discussed in Sec. 4, Tensir consists of several operators that model common tensor algebra operations, two of which are shown in Fig. 1b. The `t_t` function performs element-wise operations (one of $+, -, *, /, \%$) on tensors `x` and `y` and is defined recursively on each element. Meanwhile, `t_s` performs element-wise scalar operations on tensor `x` using the scalar value `a` and is similarly defined. Importantly, both operators are purely functional models of the tensor operations that lack implementation details that a specific target might leverage (e.g., tiling, vectorization, etc). The idea is that if `S` can be expressed using only these operators via lifting, then the lifted program can be easily translated to the targeted backends.

In Tenspiler, lifting is formulated as a Syntax-Guided Synthesis (SyGuS) [8] problem, where the goal is to synthesize a semantically equivalent program summary (PS), represented as a sequence of operators from our TensIR, with the input code as the specification. A search space (specified using grammar) describes the set of potential candidate programs for the given specification. An input program `S` is semantically equivalent to the synthesized expression `S'` if for all possible program inputs `i`, `S(i) = S'(i)`.

Tenspiler uses symbolic search to solve the synthesis problem. Symbolic search is typically implemented through enumerative or deductive search, and using constraint-solving approaches which often rely on domain-specific heuristics to scale. As a SyGuS problem, symbolic search is implemented as enumerating different expressions over a user-provided grammar, where the grammar encodes all possible combinations of operators in the target DSL up to a specified depth. However, as the depth increases, the number of choices grows exponentially, making the search intractable. As we will discuss in Sec. 5, TensIR is designed to make synthesis scalable. For `S`, the synthesis phase returns the following solution:

```
def lifted_program(b, a): return t_t(t_t(b, a, +), t_s(t_t(b, a, *), 255, /), -)
```

As Tenspiler's synthesizer currently can only reason about finite domains, all synthesized solutions are checked for full functional equivalence using an automated theorem prover. Since `S` has loops, checking equivalence with the generated program on all inputs requires loop invariants. Such invariants are synthesized during the synthesis phase by constructing a grammar similar to the PS grammar. For instance, for `S`, the synthesis phase yields two loop invariants (one for each loop) alongside a PS. As shown in Fig. 1c, these loop invariants are not arbitrary; within the loop invariants, the output variable, `out`, is expressed as a combination of operators from the TensIR that help prove the synthesized PS. We will leverage this to improve synthesis efficiency, to be explained in Sec. 5

With the synthesized solution expressed in TensIR, the final step is to translate it into the concrete syntax of the target DSL(s). In Tenspiler, this is done via simple pattern-matching rules. In Fig. 1d, we present the translated code for different supported DSLs. As we will discuss in Sec. 4.3, TensIR is designed such that generating executable code is straightforward. In fact, the code generators for the different tensor processing infrastructures supported by Tenspiler are highly similar to each other, as we will discuss in Sec. 6.

3 The TensIR Intermediate Representation

We now discuss our intermediate representation, TensIR, which plays a pivotal role in Tenspiler. While prior lifting-based compilers all use search to compile programs, their IRs are specialized for their use cases. Those IRs consist of operators from the target languages, describing their high-level semantics while avoiding low-level implementation details. For example, Casper [5] was built to translate sequential Java to MapReduce programs. The MapReduce framework consists of several versions of `map` that differ by their input types. However, Casper only defines one operator in its IR that models `map`'s functionality, and decides on the implementation to use during code generation. While doing so makes synthesis tractable, it also makes the compiler inflexible as adding another target (e.g., a hardware accelerator that supports map over tensors) will require modeling its functionality, which may be incompatible with the existing `map` from Casper's IR.

To address this challenge, we designed a novel IR, TensIR, by studying the DSLs provided by various software frameworks and hardware accelerators for tensor computation. TensIR is rooted in tensor algebra and is designed for flexibility, allowing translation to both software (deep learning frameworks, vector processing libraries) and hardware environments (machine

learning accelerators) as to be discussed in Sec. 4.3. This flexibility enables developers to select which target to execute the translated code based on availability and specific performance requirements. Given the dynamic nature of tensor processing infrastructures, TENSIR can be modified easily in terms of both adding support for new tensor operators and new target backends. This is illustrated in Sec. 6, where it only took **200** lines of code for TENSPILER to support Apple's recently introduced MLX framework [27].

Comparison with MLIR. MLIR [23] is a compiler infrastructure that enables the representation and transformation of code at various levels of abstraction. The core idea behind MLIR is to provide a unified IR that can capture the semantics of the program at different levels of detail (dialects), from high-level abstractions down to low-level, target-specific instructions. Developers can use MLIR by progressively lowering the code through different dialects until it reaches a level suitable for the target hardware. While MLIR and its dialects offer a powerful infrastructure for progressively targeting multiple hardware backends, we found that the existing dialects do not fully support all the operators required for our use case. Independently, both the `linalg` and `tensor` dialects do not support all the operators TENSIR supports. For example, the *select* operator, which is crucial for image processing kernels that apply operations conditioned on pixel values, is not supported by any MLIR dialects. Additionally, unifying these dialects can be challenging for developers. Instead of unifying, recent work such as mlirSynth [14] has explored using program synthesis to translate between different MLIR dialects. In contrast, TENSIR is designed to be flexible and easily extensible. Developers can add new operators to TENSIR by simply describing their high-level semantics, and new backend support can be incorporated by defining simple pattern-matching rules. This approach allows developers to extend TENSIR without going into the intricacies of MLIR. Moreover, TENSIR can practically be compiled into different MLIR dialects, providing developers the flexibility to leverage the MLIR infrastructure if desired.

3.1 Language Definition

The operators and grammar of TENSIR are shown in Fig. 2. TENSIR operates on tensors[2] and includes various operations. The core strength of TENSIR lies in `tensorOp`, which forms the backbone of tensor operations, including a diverse range of manipulations on tensors, such as element-wise operations, tensor vector multiplication, and reductions. These are grouped into different categories:

- `tensor_scalar` operations describe element-wise operations involving tensors and scalars, such as scalar multiplication of each element in a matrix.
- `tensor_tensor` operations perform element-wise operations between two tensors, such as element-wise multiplication of two tensors.
- Tensor reshaping such as `transpose`.
- `tensor_vec_prod` operation denotes tensor-vector products, enabling operations like matrix-vector multiplication.[3]
- Tensor reductions such as `reduce_max` and `reduce_sum`, which focus on aggregating tensor values, with the former determining the maximum element and the latter computing the sum across specified dimensions.

---

[2] In the grammar, the *Tensor Literal* refers to 1D or 2D tensors as we did not encounter higher dimensional tensors in our benchmarks.

[3] While we can also define a tensor-tensor product operator, we did not encounter such benchmarks in our evaluation and hence omitted it from TENSIR's grammar.

$$
\begin{aligned}
p \in Op &:= s_{comp} \mid t_{comp} \mid c_{control} \\
t_{comp} \in tensorOp &:= tensor_scalar(t, l, o) \mid tensor_tensor(t, t, o) \mid \\
& transpose(t) \mid tensor_vec_prod(t, t) \mid a \\
s_{comp} \in scalarOp &:= reduce_max(t) \mid reduce_sum(t) \\
o \in op &:= + \mid - \mid / \mid * \mid \% \\
c_{control} \in controlflowOp &:= ite(cond, i, i) \\
cond \in boolExpr &:= i \; rop \; i \\
i \in inp &:= l \mid t \\
rop \in relOp &:= > \mid < \mid == \mid \neg \\
a \in accessOp &:= take(t, l) \mid tail(t, l) \mid slice(t, l, l_1, l_2) \\
l &:= Integer\ Literal \mid size(t, l) \mid s_{comp} \\
t &:= Tensor\ Literal \mid t_{comp}
\end{aligned}
$$

Figure 2 TENSIR grammar.

The recursive nature of TENSIR's grammar allows tensor operations to be composed, facilitating the expression of complex algorithms encountered in source code. TENSIR extends its expressiveness beyond tensor operations by also including a control flow operator (`controlflowOp`). It integrates control flow through the `ite` operator, enabling conditional logic into tensor computation. This operator is crucial for translating real-world loopy programs that contain branches.

TENSIR is notable not only for its diversity of operations but also for the granularity of each operator, which significantly enhances its utility in translating code. The fine-grained nature of operations, from basic element-wise computations to advanced tensor reductions and control flow constructs, allows the grammar to capture the diverse tensor computation present in the input. Moreover, the selected set of operations aligns with the core functionalities supported by most tensor processing infrastructures. This ensures that TENSIR can seamlessly integrate with various frameworks and accelerators, offering flexibility in supporting multiple target DSLs. This comprehensive yet concise grammar serves as a bridge between traditional loop-based programming paradigms and the highly parallelizable world of tensor computation, providing a clear and expressive language for describing mathematical operations on tensors.

Besides tensor operations, TENSIR also supports tensor accessing and manipulation:

- `take(t, n)` extracts `n` elements from the beginning.
- `tail(t, n)` returns all the elements in tensor `t` after the first `n` elements.
- `slice(t, l, s, e)` extracts a contiguous sub-tensor from `t` from indices `s` (inclusive) to `e` (exclusive) along dimension `l`.
- `size(t, l)` returns the size of tensor `t` in dimension `l`.

Such functions are used to express the loop invariants and program summaries in the synthesis phase, as we describe next.

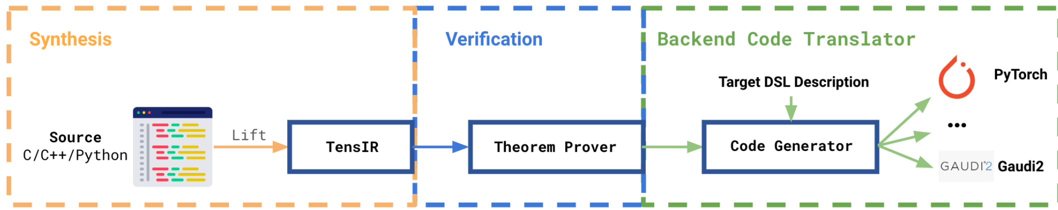


Figure 3 An overview of the TENSPILER Framework.

4 Transpiling Code Using Tenspiler

As shown in Fig. 3, TENSPILER is designed to translate a program in high-level languages, source (S), into another program that leverages different tensor processing infrastructures. TENSPILER currently support a vector processing library (NumPy) for CPU execution, DL frameworks (PyTorch, TensorFlow, MLX) for GPU execution, and ISAs for specialized hardware accelerators (Gaudi, Gemmini). TENSPILER is a verified-lifting-based compiler, leveraging search to find a program within the target domain. Instead of relying on traditional pattern-matching rules, TENSPILER translates source programs with a 3-phase workflow:

1. synthesis,
2. verification, and
3. backend code generation.

TENSPILER uses a single IR, TENSIR, to facilitate all 3 phases of its workflow. TENSIR is designed to include tensor processing operators common to all target backends. As shown in the figure, the synthesis phase takes in S and generates a program summary expressed using TENSIR. Then, in the verification phase, TENSPILER verifies the generated summaries to ensure their semantic equivalence with S. Finally, in the code generation phase, the TENSIR program is translated to the concrete syntax of the target DSL(s).

4.1 Synthesis

The objective of this phase is to search for a program expressed using the operators in TENSIR, and to ensure that the generated program is semantically equivalent to S. We formulate the search as a SyGuS problem [8] characterized by three parameters:

1. the **specification** describing the property the synthesized TENSIR expression should satisfy,
2. the **search space** that describes the space of possible solutions,
3. the **search algorithm** which searches for the candidate programs.

For TENSPILER, the specification is to find a functionally equivalent program to S. Various methods exist to express this specification, such as using input-output examples, bounded-model checking, and verification conditions (VC) [18].

In TENSPILER, we use VCs as the specification for the synthesis phase as it provides full guarantees (i.e., for all program states up to a bound, e.g., states where all integers are encoded using 8 bits) on the equivalence of S and the translated program. VCs are logical expressions encoding the correctness properties of S.

Specifically, given a program P with $vars$ representing all the variables appearing in P, and pre, $post$, inv representing the pre-conditions, the post-condition, and the invariant(s), respectively, the VCs for a program with loops consist of the following clauses:

1. **Initial Condition**: $\forall\ vars.\ pre(vars) \rightarrow inv(vars)$: loop invariants must hold before the loop begins its execution.
2. **Loop Preservation**: $\forall\ vars, vars'.\ inv(vars)\ \wedge\ P(vars, vars') \rightarrow inv(vars')$: if the invariant holds before a loop iteration, they should continue to hold after that iteration.
3. **Post-Condition**: $\forall\ vars.\ inv(vars) \rightarrow post(vars)$: invariants should hold once the loop has completed its execution.

There exist standard techniques for generating VCs from a given source program [11]. In Tenspiler, the PS and invariants in the VCs are generated as placeholders as `S` is analyzed, with their bodies to be synthesized during the synthesis phase.

Next, we define the search space for synthesis. This space outlines the potential solutions for both the PS and invariant(s), describing the solutions that could potentially satisfy the VC. Expressed as a context-free grammar (CFG), the search space imposes syntactic constraints on the structure of the outputs. In Tenspiler, the goal is not to find any PS or invariants but ones that represent the output variables in `S` as some sequence of operators from Tensir, expressed mathematically as:

$$\forall\ o \in outputVars.\ o = p, \text{where } p \in Op \text{ as defined in Fig. 2.} \quad (1)$$

This states that all return variables in `S` should be expressed as a program from Tensir. With the specification in the form of VCs, p expressed using Tensir, and the search space for the PS and invariants, the synthesis problem can be formally defined as:

$$\exists\ inv_0,\ inv_1,\ \ldots,\ PS \in G.\ \forall \sigma.\ VC(S,\ inv_0,\ inv_1,\ ...,\ PS, \sigma) \quad (2)$$

The goal of synthesis is then to find expressions from the search space G for PS and invs such that, for all program states σ, they satisfy the VC.

For Tenspiler, we use an off-the-shelf symbolic search engine, Rosette [36], which uses constraint solving to address the synthesis problem. In a constraint-solving-based approach, the specification ϕ (i.e., VC) and the search space G are encoded as a single formula, and an SMT solver is then utilized to find a model that satisfies the formula. As a constraint-based solver, increasing the number of constraints makes the problem more challenging. Given that ϕ is fixed for a particular benchmark, the design of the G becomes crucial. In Sec. 5, we discuss how the design of Tensir helps keep the grammar size reasonable and scales the synthesis process.

4.2 Verification

During the synthesis phase, as the PS and loop invariant(s) are validated against the VC only for a bounded set of program states,[4] it is essential to check their validity for all program states. Tenspiler uses an SMT solver to do so by negating the VC in program verification i.e., checking if $\neg VC(S,\ inv_1,\ inv_2,\ ...,\ PS,\ \sigma)$ is satisfiable for some σ. The placeholders in the VC are substituted with the synthesized bodies of PS and invs. If the solver cannot find any such σ, then the generated PS and invs are correct for all possible program states, thus proving PS and invs hold for all program states. If a σ is found, then Tenspiler will iterate back to the synthesis phase in search of another candidate expression.

Besides using SMT solvers for Eq. (2), Tenspiler also leverages SMT solvers' support of algebraic data types (ADT) to allow users to define common data structures such as lists and tuples. Internally, Tenspiler models tensors using the list data structure defined using

[4] We are unaware of any SyGuS solvers that can validate against an unbounded set of program states efficiently, including state of the art solvers such as Z3 and CVC5.

```
(assert (forall ((data (Tensor Int)) (a Int) (idx Int))
(=> (>= index 0) ∧ (<= index len(data))
(= t_s(data[:idx],a,*) (+ [data[0]*a] t_s(data[1:idx], a, *))))))
```

Figure 4 Example of an inductive axiom for the `tensor_scalar` operator in TENSIR described using SMT-LIB. "+" corresponds to the concat operator.

ADTs. We use ADT's accessor and constructor functions to retrieve and create new tensors. All the tensor accessing functions like `slice`, `take` are modeled as recursive functions over the list data structure. Currently, while image processing kernels use integers and deep learning kernels operate over floats, we verify all the benchmarks using the theory of integers and reals, due to poor solver support for reasoning about floats.

Since the verification of loop invariants is undecidable in general, we define additional *axioms* for the operators in TENSIR to aid verification. These axioms describe the behavior of functions that cannot be automatically deduced by the solver. Identifying the axioms requires an understanding of the program's semantics and the properties that need verification. Such axioms describe simple attributes such as distributivity, associativity, and commutativity of the tensor operators. In Fig. 4, we show an inductive axiom for the `tensor_scalar` operator which states that the result for a given index is determined by the product of the first element of the tensor and an integer, plus the result for the remaining sub-tensor up to that index. As shown, having tensors as first-class objects in TENSIR greatly simplifies the task of defining these properties. Instead of defining these properties using low-level SMT-LIB list data structures, TENSIR enables users to define them at the tensor level, abstracting away the low-level solver-related details. This high-level representation greatly simplifies the task of defining these properties and makes the axioms more readable and maintainable.

4.3 Code Generation

After successfully verifying the synthesized TENSIR program, the final stage in TENSPILER's workflow is to translate the TENSIR program into the concrete syntax of a target DSL. TENSIR makes this easy as it inherently represents tensor operations supported by all the target DSLs, and code can be generated using simple syntax-driven rules that map TENSIR operators to their DSL-specific counterparts.

To translate the TENSIR expression into an executable DSL program, our code generation step recursively processes each part of the TENSIR expression. Fig. 5 illustrates a portion of the code generation function for PyTorch. The function maps TENSIR variables to their names (line 3), literals to their values (line 5), and function calls to their PyTorch equivalents based on function signatures (lines 6-15).

Consider the running example in Fig. 1a, where the synthesized TENSIR solution is `t_t(t_t(b, a, +), t_s(t_t(b, a, *), 255, /), -)`. This expression represents a `t_t` function call with the `-` operator, which maps to `torch.subtract` as shown in line 13. Next, the `codegen` function is called recursively on the two arguments, `t_t(b, a, +)` and `t_s(t_t(b, a, *), 255, /)`. This results in the final translated PyTorch expression as `torch.subtract(torch.add(b, a), torch.divide(torch.add(b, a), 255))`.

To extend support for a new backend, one simply needs to replace the DSL operator names in lines 11, 15, 17, and 19. For example, in MLX's codegen, `torch.add` on line 11 would be replaced by `mlx.core.add`.

This direct and syntactic translation simplifies the integration of new tensor-based target DSL into TENSPILER, as one would only need to add simple translation rules in the code generation process. For instance, we add support for MLX by changing only 65 lines of code to an existing 200-line template, as its API closely follows that of NumPy.

```
def codegen(expr: Expr):
  if isinstance(expr, Var):
      return expr.name()
  elif isinstance(expr, Lit):
      return expr.val()
  elif isinstance(expr, Call):
    f_name, args = expr.name(), expr.arguments()
    if f_name in {"t_t", "t_s"}:
      op = args[-1]
      if op == "+":
          return f"torch.add({codegen(args[0])},{codegen(args[1])})"
          #corresponding MLX return statement
          #return f"mlx.core.add({codegen(args[0])},{codegen(args[1])})"
      elif op == "-":
          return f"torch.subtract({codegen(args[0])},{codegen(args[1])})"
      elif op == "*":
          return f"torch.multiply({codegen(args[0])},{codegen(args[1])})"
      elif op == "/":
          return f"torch.divide({codegen(args[0])},{codegen(args[1])})"
      ...
```

Figure 5 Code generation for the element-wise add operator to different targets.

As a part of this work, we have implemented support for **six** different target DSLs in our code generator: **NumPy**, **TensorFlow**, **PyTorch**, **MLX** (an ML framework for Apple silicon), **TPC-C** (C-based programming language for Intel's Gaudi processor), and **Gemmini** (an open-source neural network accelerator generator).[5]

5 Synthesis Optimizations

A naive approach to constructing the grammar for search space is to enumerate all possible combinations of Tensir expressions up to a fixed depth. For Tensir as defined in Fig. 2, if we focus solely on the compute operators, a depth-4 grammar (i.e., sequence of 4 operators) results in a search space of ∼200k expressions, since it grows exponentially with the depth and the number of operations. In Fig. 6, we show a small part of the depth-4 grammar. We have devised several optimizations to reduce the search space and make the search tractable.

5.1 Restricting Operators

First, we generate the grammar based on types, i.e., we only include the operators whose output types match with the expected return type. In the case of `S` in Fig. 1a, since the return type is `vector`⟨`vector`⟨`int`⟩⟩, all reduction operations are excluded. In Fig. 6, the operators in v_4 and l_4 will be removed (shown in red). These correspond to operators returning 1-D vectors and integers respectively.

5.2 Restricting Program States

We further optimize the search space by restricting the set of program states in Eq. (2). Instead of satisfying the VC for all σ, we find PS and invs that satisfy a bounded set. Bounded synthesis is crucial because most SyGuS solvers have limited support for recursive function definitions and require SMT solvers for validation. However, SMT solvers lack inherent support for reasoning about Tensir operators that are not covered by the standard theories defined in SMT-LIB [9] and require additional axioms to be defined. We instead

[5] We provide further details of these DSLs in Appendix A in the extended version of this paper[29].

$$
\begin{aligned}
out &:= m_4 \mid v_4 \mid l_4 \\
m_4 &:= tensor_scalar(m_3,\ l_4,\ o) \mid tensor_tensor(m_3,\ m_3,\ o) \mid \\
&\quad\ transpose(m_3) \mid ite(cond_4,\ m_3,\ m_3) \mid m_3 \\
v_4 &:= tensor_scalar(v_3,\ l_4,\ o) \mid tensor_tensor(v_3,\ v_3,\ o) \mid \\
&\quad\ tensor_vec_prod(m_3, v_3) \mid ite(cond_4,\ v_3,\ v_3) \mid v_3 \\
l_4 &:= reduce_max(l_3) \mid reduce_sum(l_3) \mid l_3 \\
cond_4 &:= l_3\ rop\ l_3 \\
\dots \\
l_1 &:= 255 \mid size(t_1) \\
t_1 &:= a\langle a_1,\ a_2 \dots a_n\rangle \mid b\langle b_1,\ b_2 \dots b_n\rangle\ \ a\langle a_1,\ a_2\ \rangle \mid b\langle b_1,\ b_2\rangle \\
rop &:= > \mid < \mid == \mid \neg \\
o &:= + \mid - \mid / \mid *
\end{aligned}
$$

Figure 6 A depth 4 general synthesis grammar for the source in Fig. 1a.

integrate bounded synthesis by restricting the maximum unrollings of recursive operators, thereby eliminating the need for additional axioms. Specifically, we restrict the program states by limiting the lengths of the data structures and the sizes of the data types. For instance, in TENSPILER, we constrain all 1D tensors to length 2 and the integers to 6 bits or less for the first rounds of synthesis. In Fig. 6, all the tensor literals in t_1 are changed from an unbounded length "n" (shown in orange) to length 2 (shown in blue). If the synthesized choices fail to verify, we then increase the bounds in subsequent rounds. Note that since the synthesized solutions only work for a restricted set of program states, we invoke the theorem prover for subsequent verification to check if PS and invs are valid for all states.

5.3 Leveraging Expression Trees

Despite the above two optimizations, the synthesis search space remains large. For example, in the context of `S` in Fig. 1a for which we need to synthesize two invs and one PS, a depth-4 grammar, after removing the reduction operations, still presents around 100k potential solutions just for PS. TENSIR plays a significant role in the further pruning of this search space. The design of TENSIR operators effectively bridges the gap between high-level tensor operations and the loop-based paradigm commonly used for computing on tensors. This property of TENSIR allows us to leverage the expression-tree-based filtering approach, which we describe next, to efficiently prune the synthesis search space.

Our approach starts with the static analysis of `S` to identify the computations performed; the static analysis pass emits an expression tree that represents the computation. For example, the pre-order traversal of the expression tree for `S` from Fig. 1a is: `(- (+ b a) (/ (* b a) 255))`. In Fig. 6, this results in the pruning of `tensor_scalar` and `ite(cond5, m4, m4)` at the top-level (shown in teal) and similarly operators at other depths (m_4, m_3) are filtered. The generated expression tree is then transformed into an abstract expression tree, where variables and constants are replaced with placeholders, resulting in a synthesis template.

The abstract expression tree for `S` is then: `(- (+ var var) (/ (* var var) lit))`. This abstract expression tree guides Tenspiler in identifying the sequence of TensIR operators. In this example, Tenspiler deduces the sequence of operators from the tree as: `t_t(t_t(var, var, +), t_s(t_t(var, var, *), lit, /), -)`, where `var` and `lit` are variables and literals to be synthesized, respectively.

Our expression trees are amenable to vectorized operations, which simultaneously perform the same computation on multiple data elements. Specifically, each level of the tree corresponds to an operation with the branches indicating data flow. In the example expression shown above, we see element-wise subtraction, addition, scalar division, and element-wise multiplication orchestrated such that it aligns with the vectorized execution of the original computation.

This approach is not confined to specific operators but is adaptable to a range of operations in TensIR. It can identify constructs like if-else blocks, where *ite* arguments are determined using the same expression tree strategy, allowing the synthesis process to determine the optimal sequence of operators within the constructs. This flexibility extends to reduction operators and other complex operations, aiding in the synthesis of efficient operational sequences.

5.4 Constraining Variables

The final optimization is to pinpoint specific variables (`var`s such as `a,b` in Fig. 1a) and literals (`lit`s such as `255` in Fig. 1a) to be used in the grammar. Specifically, we constrain the variables to the set of live variables and also constrain constants to the set of constants that have appeared in the program. This strategy simplifies the computational task, avoiding the complexity of synthesizing a complete depth-4 operator sequence. By leveraging our expression tree-based approach, the search space reduces to 64 expressions, and the synthesizer promptly yields the correct solution within 76 secs: `t_t(t_t(b, a, +), t_s(t_t(b, a, *), 255, /), -)`.

5.5 Overall Synthesis Algorithm

The algorithm described in Fig. 7 summarizes the synthesis phase in Tenspiler. This phase is used to search for the bodies of PS and invariants which satisfy the VC. The synthesis is an iterative process conducted over multiple rounds, assuming a filtered search space leveraging type-based and expression tree optimizations described earlier. We start with the tensor bound size set to 2 which corresponds to restricting the program states optimization. In each round, we invoke Rosette's search algorithm (line 5) to generate candidates for PS and invs. Upon obtaining a solution, the candidate undergoes validation against the VC for all program states, as the synthesis phase only checks within a constrained set of program states. We invoke a verifier (CVC5) (line 8) to perform this check. If the verifier yields "UNSAT," the generated candidates are correct. Conversely, if "SAT" is returned, indicating incorrect candidates (line 11), the VC is augmented with blocking constraints. These constraints state that the generated PS or invs in next round should differ from those in the previous rounds. This iterative process continues for a specified number of rounds (`max_rounds`) before incrementing the tensor bound sizes. In cases where Rosette's search algorithm does not produce a solution initially (line 13), indicating an overly restrictive grammar, the initial grammar is expanded to include additional options for both PS and invariants, such as choices for loop bounds, indexing, and operator sequences. We keep a separate timer (not shown in Fig. 7) that maintains a maximum time bound for the entire synthesis process.

```
def synthesis_algorithm(spec, tensor_size_bound, holing_grammar, search_algorithm,
    verifier, max_rounds, timeout):
  r = 0 # rounds within one list bound
  #bounded synthesis optimization
  while r < max_rounds:
    ps_inv = search_algorithm(spec, holing_grammar) #rosette
    if ps_inv is not None:
      ps_r, inv_r = ps_inv
      if verifier(specification, ps_r, inv_r) == "UNSAT": return ps_r, inv_r
      else:
        spec = spec and (ps != ps_r) and (inv != inv_r) #add blocking constraint
        r += 1
    else:
      expand_holing_grammar(holing_grammar)
  # Increment tensor size bound
  return synthesis_algorithm(spec, tensor_size_bound + 1, holing_grammar,
      search_algorithm, verifier, max_rounds, timeout)
```

Figure 7 TENSPILER synthesis algorithm.

6 Experiments

We evaluate TENSPILER's effectiveness in converting code into various tensor processing infrastructures using 10 loop-based real-world benchmark sets: **blend**, **Llama** [24], **blas**, **darknet**, **dsp**, **dspstone**, **makespeare**, **mathfu**, **simpl_array**, and **utdsp**. The **blend** benchmarks focus on image processing kernels, the **Llama** benchmarks contain traditional deep learning applications, and the rest 8 are all sourced from various existing software libraries, such as the BLAS linear algebra library and the TI signal processing library, and are used recently to evaluate C to TACO translations [25]. This combination of benchmarks allows for a comprehensive assessment of TENSPILER's effectiveness and adaptability across diverse domains and programming paradigms.

1. Used in prior work [6], the **blend** benchmarks consist of 12 functions dedicated to point-wise image blending operations – a fundamental aspect of image processing known for diverse visual effects. These functions span 180 lines of C++ code, and 10 are characterized by doubly-nested loops, which are common in image processing algorithms.
2. **Llama** benchmarks are derived from the C++ based inference code of Llama2 [24], an open-source LLM from Meta. We labeled the portion of code to be lifted from the source code without doing any extensive syntax or code logic edits. These benchmarks include 11 functions capturing essential operations like computing activations, attention mechanisms, and layer norms. They total around 106 lines of code, with 2 functions incorporating doubly-nested loops.
3. **blas**: 2 benchmarks from the BLAS [13] linear algebra library.
4. **darknet**: 10 neural network functions sourced from the Darknet [2] deep learning framework.
5. **dsp**: 12 signal processing functions from the TI library [4].
6. **dspstone**: Kernels from the DSPStone suites [38] that target digital signal architectures.
7. **makespeare**: Programs originally from Rosin [31] that manipulate integer arrays.
8. **mathfu**: 11 mathematical functions extracted from the Mathfu library [3].
9. **simpl_array**: 5 functions for computations on integer arrays originally from prior work [35].
10. **utdsp**: Kernels from the UTDSP suite [33] that targets digital signal architectures.

6.1 Evaluation Setup

The synthesis and verification phases for all benchmarks are executed on a MacBook Pro 2 GHz Quad-Core Intel Core i5 Processor with a timeout of 1 hour. After lifting the code to TENSIR, the code generator, as explained in Sec. 4.3, generates executable code for each target backend. In the next section, we first describe the datasets used for evaluating the performance of lifted benchmarks. Then, we describe each target backend used for executing these benchmarks.

6.1.1 Datasets for Evaluation

To mimic real-world settings, we evaluate the translated code on actual datasets instead of generating random inputs.

For the **blend** image processing benchmarks, **blas**, **darknet**, **dsp**, **dspstone**, **makespeare**, **mathfu**, **simpl_array**, and **utdsp**, we source images from ImageNet [17], a large-scale image dataset. We process these images as grayscale to ensure pixel values fall within the appropriate range. For benchmarks with 1-D tensor inputs, we flatten the images before feeding them as inputs and pass them in as they are for 2-D tensor inputs. For the blending layers in the blend benchmarks, we generate random pixel values from 0 to 255. We randomly select a set of 10,000 images from the dataset for evaluation.

For the **Llama** benchmarks, we evaluate the synthesized code using weights from Vicuna [16], an open-source LLM with similar model size as Llama2.[6] Some kernels operate over model weights, for which we directly use the weight matrices from Vicuna. For kernels operating over inputs, we simulate embeddings by creating random 32-bit float vectors within the range [0, 1). The evaluation primarily uses the 33B-parameter Vicuna model; however, for evaluating the MLX framework, the 7B-parameter version is used due to memory limitations.

6.1.2 Target Software Frameworks and Hardware Accelerators

The core objective of TENSPILER is to translate sequential programs to a spectrum of diverse target DSLs, which can then be executed on conventional CPUs, GPUs, or specialized hardware accelerators. Although finding the optimal target DSL for the given input program would be an interesting feature in TENSPILER, currently TENSPILER is designed to provide users with the flexibility of choosing their preferred environment.

For our experimental evaluation, we choose 6 different target DSLs as we mentioned in Sec. 4.3: **NumPy**, **TensorFlow**, **PyTorch**, **MLX**, **TPC-C**, and **Gemmini**. We believe that our comprehensive selection of DSLs is necessary to test the robustness of TENSPILER.

The TENSIR design greatly simplified this process as NumPy, TensorFlow, PyTorch, and MLX have similar APIs, and each of these DSLs uses only 200 lines of code for generating executable code.

To establish a baseline for execution performance, we compile C++ code for all the benchmarks using `gcc-8.3.0` with `-O3` flag and then run them on an Intel Xeon 8380 CPU. Given that each DSL is tailored to enhance performance on specific hardware, we conduct evaluation across five distinct platforms: the Intel Xeon 8380 CPU, Nvidia V100 GPU, Apple M1 Pro, Intel Gaudi 2 processor, and the Gemmini accelerator.[7] In all, we utilized 7 different

---

[6] We did not use Llama2 model weights as they are not publicly available.

[7] Due to lack of physical device, Gemmini evaluations are done on a simulator with limited computing power and no file system support. Thus, it is compared with smaller random inputs.

DSL-hardware device combinations for our experiments: (1) NumPy-CPU, (2) TensorFlow-V100, (3) PyTorch-V100, (4) MLX-Apple M1 (5) TPC-C-Gaudi, (6) PyTorch-Gaudi, and (7) Gemmini. This comprehensive mapping of each backend to its corresponding device enables us to accurately assess the benefits TENSPILER provided through lifting.[8]

6.2 Synthesis Timings

In this section, we present the time TENSPILER takes to synthesize and verify solutions for each of our benchmarks. During the synthesis phase, we apply all the optimizations mentioned in Sec. 5 with a 1 hour timeout. TENSPILER synthesizes the correct translations for all the benchmarks under **15 mins**.

Fig. 8 illustrates the synthesis performance for our 10 benchmark suites. Fig. 8a illustrates the synthesis performance for the **blend** benchmarks. All but three benchmarks are synthesized in one synthesis (and verification) iteration, with an average synthesis time of 40.7 seconds and a median synthesis time of 2.357 seconds. Single-loop benchmarks synthesize with an average of 2.17 seconds and a median of 1.92 seconds. Double-loop benchmarks, on the other hand, have an average of 128.91 seconds and a median of 22.74 seconds for synthesis.

The three benchmarks that take more than one round of synthesis are `softmax1`, `transformer1`, and `transformer2` from the **LLama** suite. `softmax1` fails to synthesize within one round because the initial grammar is overly restrictive for its loop invariant. `transformer1` and `transformer2` involve complex indexing constraints for their invariants, initially leading to spurious solutions with tensor size limit of 2. `transformer2` finds the correct solution after 6 tries, while `transformer1` exhausts the maximum number of tries (10) with tensor size 2. We then increase the tensor size limit to 3 for `transformer1` and a correct solution is generated within 3 rounds.

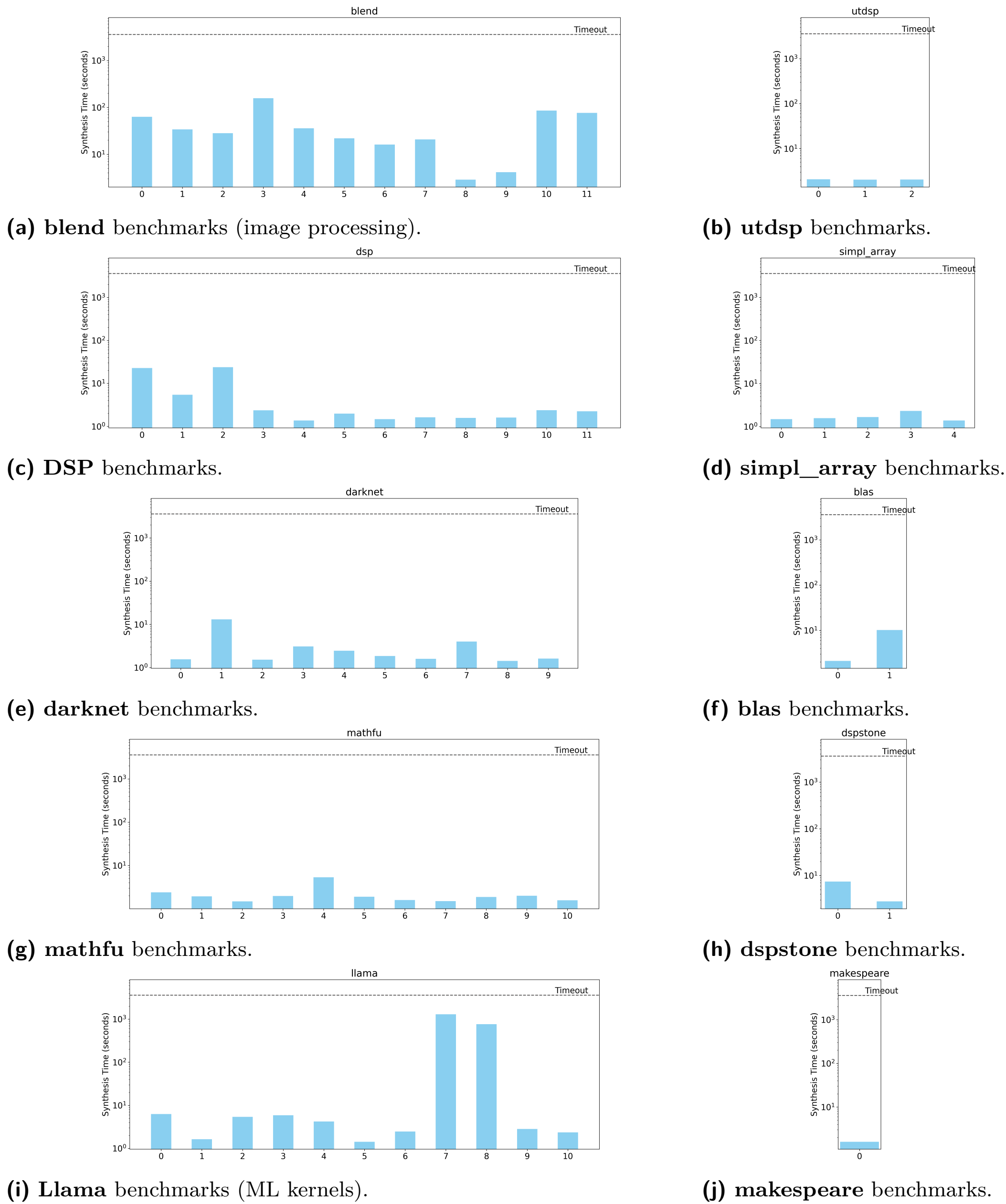
6.2.1 Analysis

We observe that synthesis difficulty is correlated to both the number of loops and the complexity of the TENSIR solution. For example, the `dot` benchmark in the blas benchmark set has a single loop. Its TENSIR solution is `reduce_sum(t_t(a, b, *))`, which has two operators and only two arguments, `a` and `b`, to be synthesized. This benchmark synthesizes in around two seconds. On the other hand, the `transformer_part1` benchmark from the **LLama** benchmark set has a doubly-nested loop and a complex solution with six operators. All arguments to these operators must be synthesized, with some requiring 3 expressions such as `head * head_size + head_size` and `sqrt(head_size * 1)`. This benchmark takes around 1300 seconds to synthesize.

In addition to synthesis challenges, we recognize that the tree approach may restrict the ability of TENSPILER to synthesize different solutions. However, through manual verification, we confirm that TENSPILER consistently generates optimal solutions across our benchmarks. We evaluate the solutions using expression length as the cost function. Generally, shorter expressions mean fewer function calls and thus lower execution cost. To illustrate, we use the `linear_dodge` example from the **blend** benchmarks for which the synthesized solution is as follows:

```
def linear_dodge(a, b): t_t(a, b, +)
```

[8] The exact configurations of all the hardware devices and their software environments are described in Appendix C in the extended version of this paper[29]



(a) blend benchmarks (image processing).

(b) utdsp benchmarks.

(c) DSP benchmarks.

(d) simpl_array benchmarks.

(e) darknet benchmarks.

(f) blas benchmarks.

(g) mathfu benchmarks.

(h) dspstone benchmarks.

(i) Llama benchmarks (ML kernels).

(j) makespeare benchmarks.

Figure 8 Synthesis timings for all the benchmark suites. Benchmark name legend in Appendix B in the extended version of this paper[29].

After relaxing the constraint on the tree structure, we were able to synthesize a different solution as shown below:

```
def linear_dodge(a, b): t_t(a, t_s(b, -1, *), -)
```

The latter solution is longer in length, and involves **2** tensor operations – a `tensor_tensor` operation and a `tensor_scalar` operation – as opposed to 1 `tensor_tensor` operation synthesized using the tree approach. Therefore, it is less optimized. In addition, the tree approach also speeds up the synthesis process as shorter expressions are easier to synthesize.

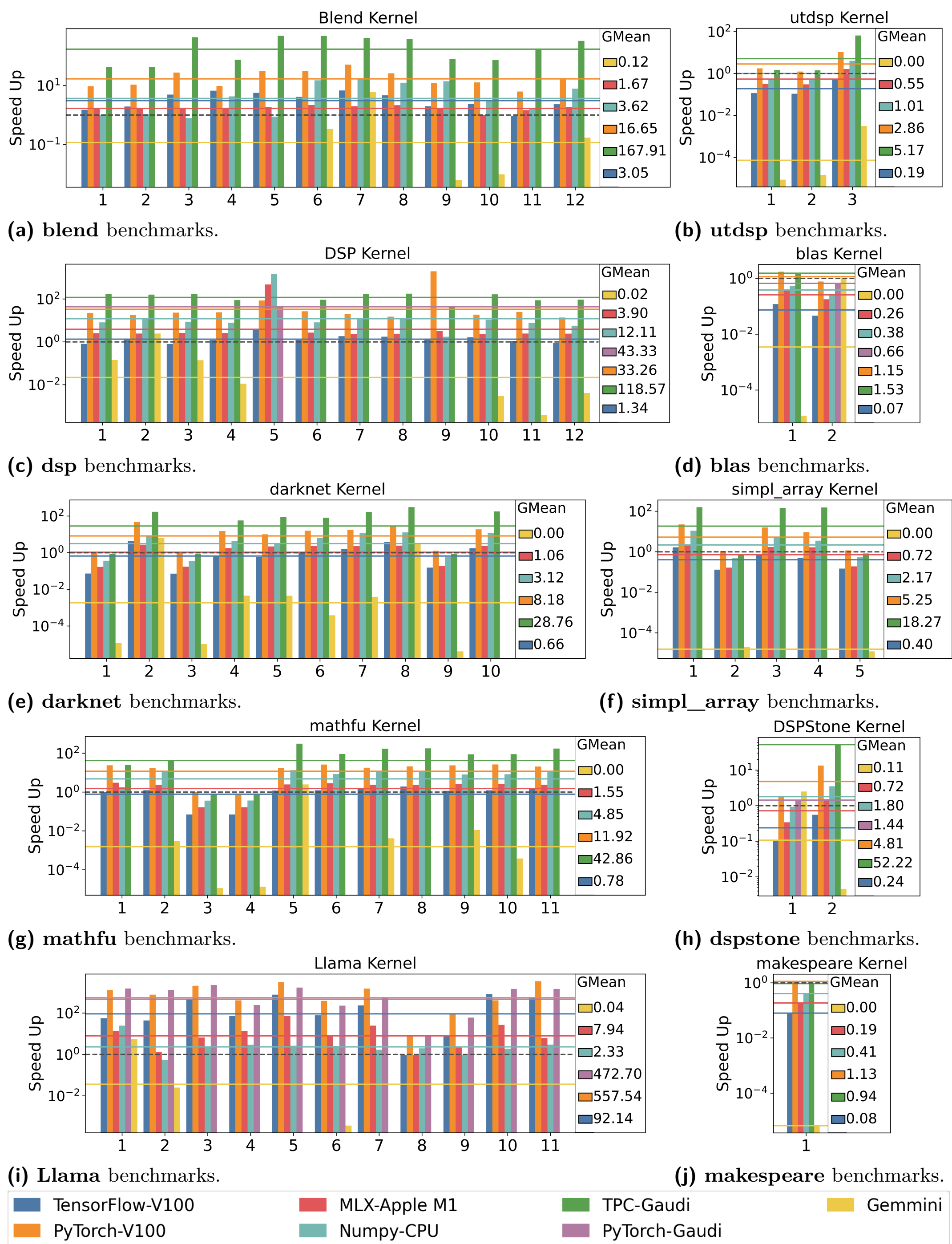


Figure 9 Kernel speedup over baseline. Benchmark name legend in Appendix B in the extended version of this paper[29].

6.3 Performance Timings

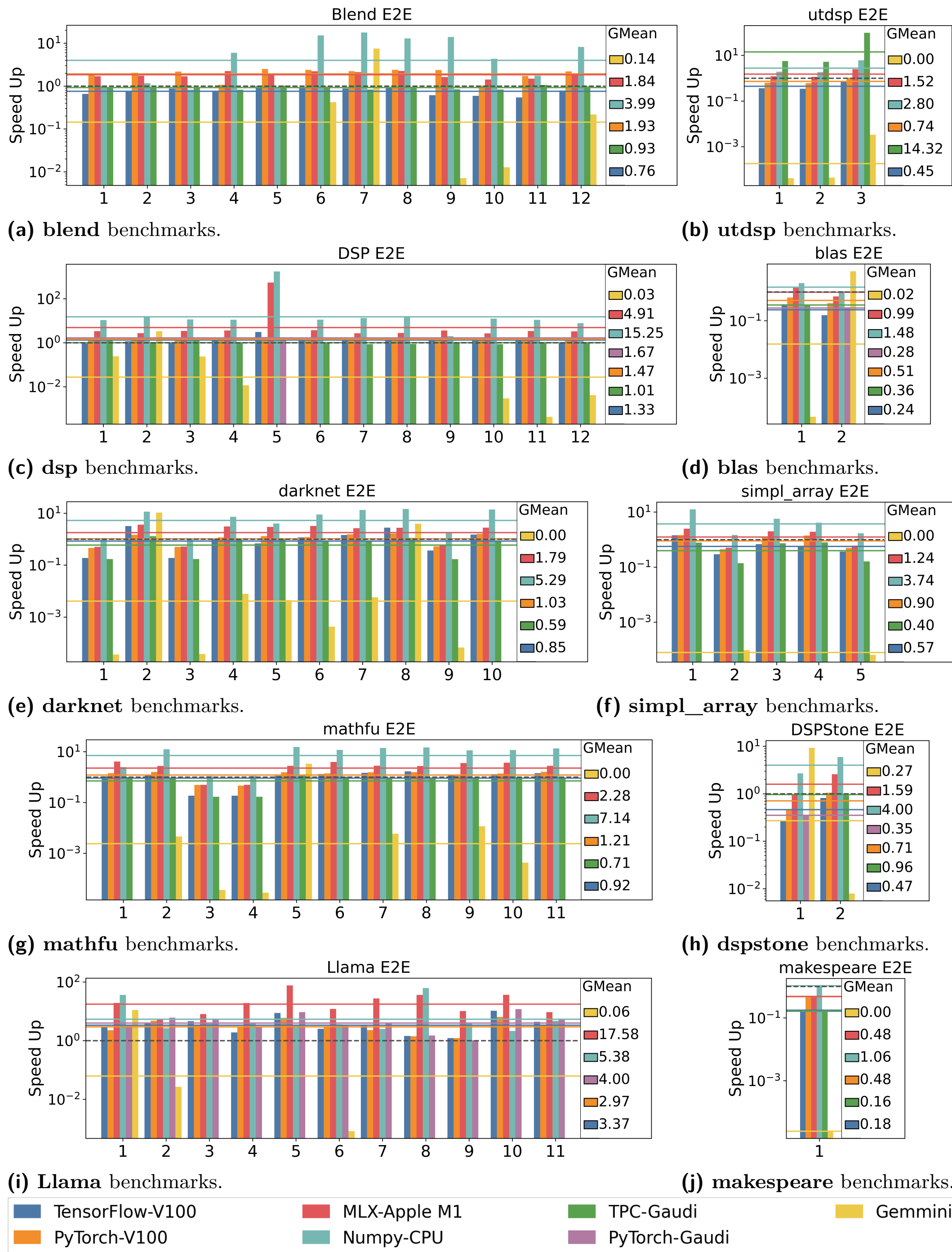
In this section, we evaluate the performance of the original input code – sequential C++ programs compiled with `gcc -O3` – by comparing them with their translated versions executed across different target backends as detailed in Sections 6.1.2 and 6.1.1.

Kernel Performance. Kernel performance focuses on computation time excluding data transfer overhead. We see significant improvements as illustrated in Figures 9, with an average speedup of **105.1**× across all benchmarks. Notably, the Gaudi 2 processor demonstrates an exceptional speedup of **241**×, highlighting the advantages of migrating legacy code to newer hardware platforms. For other backends such TensorFlow, PyTorch, MLX and NumPy we see speedups of **46**×, **244**×, **10.5**× and **26.12**× respectively.

However, compatibility issues can emerge with certain backends. For example, the Gemmini accelerator does not support certain operations in our TensIR like `tensor_tensor` element-wise multiplication, `slice`, and `tail`. To address this, we only translate supported TensIR operations from the synthesized PS, and default to running the unsupported operations using sequential C on CPUs. Out of 69 benchmarks, 41 are translatable to Gemmini 's instruction set architecture (ISA), yet only 10 can be fully expressed using Gemmini instructions alone. Challenges are notable in benchmarks like `screen_blend` (Fig. 9a), where element-wise vector multiplication must fallback to execution on a less powerful CPU. Furthermore, most Gemmini's instructions require square matrices inputs. This means that we need to pad vector inputs to square matrices before being able to utilize Gemmini's instructions, effectively squaring the data volume to be processed. This results in varied performance as shown in Fig. 9i and Fig. 9e.

End-to-end Performance. While frameworks and accelerators deliver substantial kernel performance enhancements, a comprehensive assessment must account for end-to-end benchmark times, encompassing initial setup and data movement between the host (CPU) and the accelerator device. Our focus here is on data transfer (TensorFlow, PyTorch, and Gaudi processor) and memory management (C++). As illustrated in Fig. 10, we again observe an overall speedup, averaging **9.7**×. In particular, CPU libraries like NumPy and MLX show more improvements with the notable advantage of avoiding transferring data to specialized hardware. These benchmarks, involving the processing of 1D or 2D character vectors, benefit largely from C++'s efficient handling of contiguous data structures. Meanwhile, Gaudi 2 drivers encounter performance bottlenecks due to the overheads associated with hardware initialization and frequent small data transfers. This significant upfront cost, especially pronounced in small-scale data operations, leads to a much less announced speedup. We believe such a phenomenon is uncommon in real-world use cases such as training deep learning models, due to techniques like batch processing or pipelining to minimize data transfers or to overlap computations with communications, thereby reducing or hiding transfer overhead and enhancing overall efficiency.

Compare Against Pattern Matching-Based Compilers. As outlined in Sec. 1, Tenspiler is designed to address the limitations inherent in traditional transpilers that rely on pattern matching to compile. Such compilers are resource-intensive to develop and prone to errors. To the best of our knowledge, no existing compiler matches the breadth of DSL support offered by Tenspiler. However, specialized compilers, such as Numba [28], have been introduced for accelerators like GPUs. Numba leverages LLVM IR to generate GPU-accelerated code from Python code, making it a suitable candidate for comparison.



(a) **blend** benchmarks.
(b) **utdsp** benchmarks.
(c) **dsp** benchmarks.
(d) **blas** benchmarks.
(e) **darknet** benchmarks.
(f) **simpl_array** benchmarks.
(g) **mathfu** benchmarks.
(h) **dspstone** benchmarks.
(i) **Llama** benchmarks.
(j) **makespeare** benchmarks.

Figure 10 E2E speedup over baseline. Benchmark name legend in Appendix B in the extended version of this paper[29].

```
vector<float> matmul(vector<vector<float>> weight, vector<float> input) {
    vector<float> output;
    int m = weight.size();
    int n = input.size();
    for (int row = 0; row < m; row++) {
        float curr = 0;
        for (int col = 0; col < n; col++) {
            curr += weight[row][col] * input[col];}
        output.push_back(curr);}
    return output;}
```

(a) Original `matmul` function in C++.

```
@cuda.jit()
def matmul (weight, input, res):
    m = len(weight)
    n = len(input)
    for i in range(m):
        curr = 0
        for j in range(n):
            curr += weight[i][j] * input[j]
        res[i] = curr
```

(b) Numba kernel annotated version of `matmul`.

Figure 11 Manually rewritten Numba example.

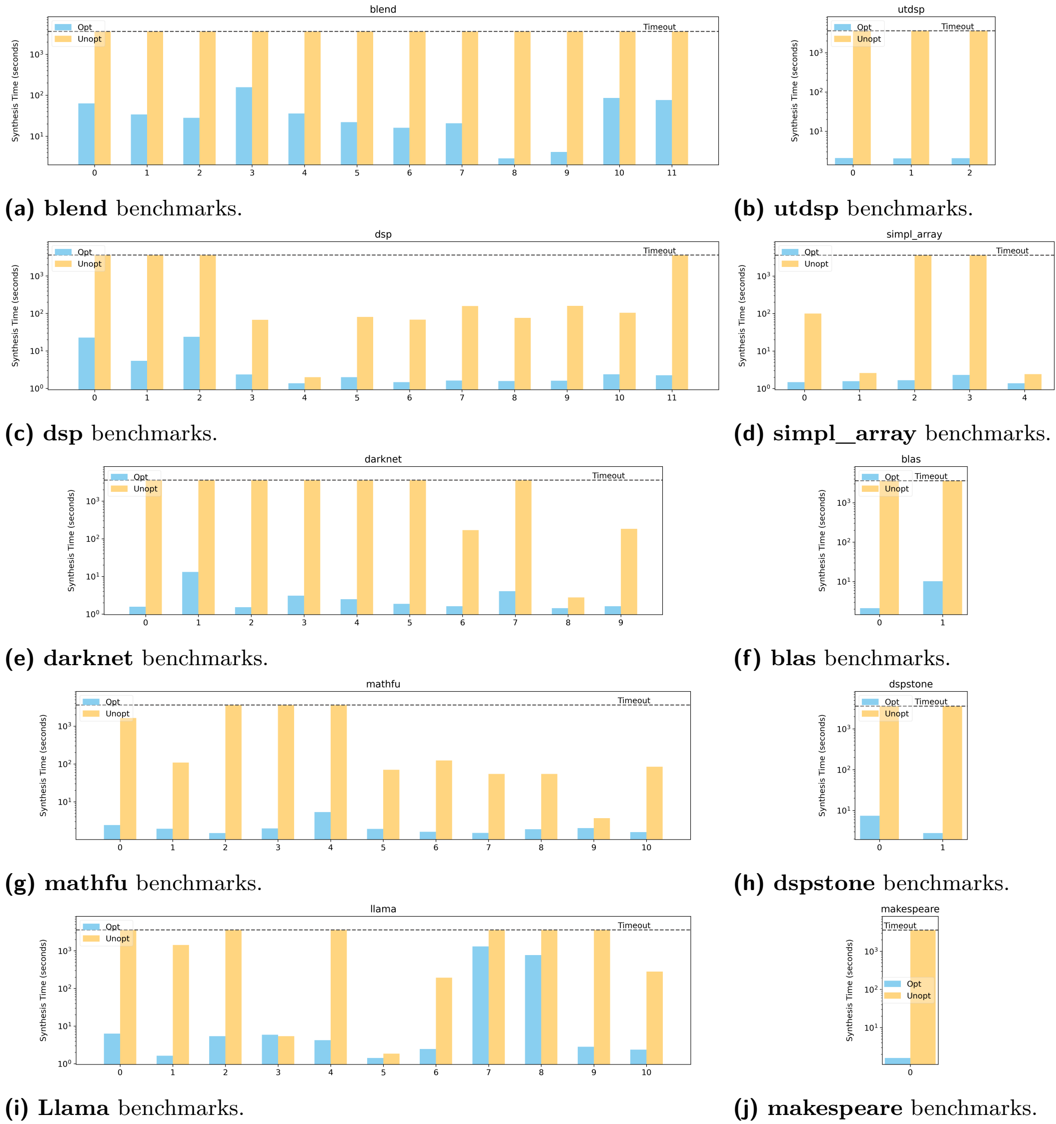
For benchmarking purposes, we utilize the same datasets, test cases, and setup described previously in Sec. 6. Benchmarks are rewritten in Python and adapted to conform to CUDA kernel requirements by removing return statements, as shown in Fig. 11. Additionally, relevant data are cast to NumPy arrays as Numba focuses on optimizing code written against NumPy's API. These syntactic requirements represent a limitation of Numba's approach. In contrast, Tenspiler operates directly on the original benchmark implementations.

Experimental results demonstrate that GPU-based PyTorch and TensorFlow code generated by Tenspiler performs, on average, 1.87× faster than code annotated with Numba. Remarkably, while Numba benefits from years of development by expert engineers, Tenspiler achieves superior performance with only **200** additional lines of code dedicated to code generation. A closer examination of the compiled PTX assembly code for the `matmul` benchmark, which shows a 2.6× speedup, reveals that the Numba-generated code lacks the use of advanced instructions and techniques such as fused multiply-add (FMA), tiled-based computation models, or shared memory,[9] which are crucial for peak performance. These techniques are standards in PyTorch and TensorFlow with optimized kernels. In contrast, Numba requires extensive manual tuning to implement, evident in the more complex and faster `matmul` example in its documentation.[10] Tenspiler, by automatically recognizing and translating matrix multiplication operations to leverage the pre-optimized kernels, avoids the complexities of manual code optimization while achieving high performance.

6.4 Ablation Study

In our ablation study, we evaluate using our benchmark suites the effectiveness of the optimizations (described in Sec. 5) in making synthesis scale.

[9] See Appendix D in the extended version of this paper[29] for the PTX code.
[10] For the detailed example of an optimized `matmul` function with shared memory for Numba, see `https://numba.readthedocs.io/en/stable/cuda/examples.html#id30`



(a) blend benchmarks.

(b) utdsp benchmarks.

(c) dsp benchmarks.

(d) simpl_array benchmarks.

(e) darknet benchmarks.

(f) blas benchmarks.

(g) mathfu benchmarks.

(h) dspstone benchmarks.

(i) Llama benchmarks.

(j) makespeare benchmarks.

Figure 12 Synthesis timings for all the benchmarks with and without TENSPILER's tree-based optimization. Benchmark name legend in Appendix B in the extended version of this paper[29].

Bounded Synthesis. In this experiment, we keep the type-based filtering and tree approach while removing the incremental bounded synthesis optimizations. We start with a static tensor bound of 4 instead of the incremental approach. With this, **6** of the 12 **blend** benchmarks time out. In addition, benchmarks involving 2D tensors that do not time out see an average of **36.75**× slowdown.

Tree Approach. For this experiment, we include type-based filtering and remove the expression tree approach for grammar filtering. We assume a fixed depth for the grammar, i.e., including all operators up to the specified depth, and increase it upon synthesis failure (starting at depth 1). Without static analysis, no assumptions are made about the operators, slice indices, variables, or constants, necessitating their synthesis. Unlike the tree approach with a fixed number of placeholders, this approach exhibits scalability issues as the number of grammar choices increases exponentially with depth. Therefore, only benchmarks with depth 1 and 2 expressions could be successfully synthesized.

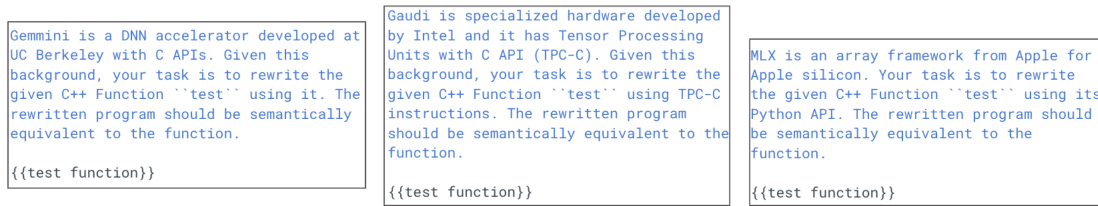


Figure 13 Prompts for LLM.

As illustrated in Fig. 12, without the tree based optimization, **42** out of the total **69** benchmarks timed out. In particular, all benchmarks of the **blend**, **blas**, **dspstone**, **makespeare**, and **utdsp** suites timed out. For the benchmarks that succeed, the synthesis phase slows down by an average **101.55**× compared to the tree-based grammar filtering approach due to the need to synthesize additional expressions in both PS and invs.

6.5 Comparison with LLMs

LLMs have shown promising results in various programming-related tasks, such as code generation, translation, and testing. However, these models suffer from a lack of formal verification of the translated code and face challenges in adapting to new DSLs or backends that are not well-represented in their training corpus.

To test the capabilities of LLMs in generating code for new or low-resource DSLs, we prompt a state-of-the-art proprietary LLM Claude Opus [1] (our evaluation using other LLMs such as GPT4 shows similar results). We selected three backends for this experiment: MLX, a completely new DSL, Gaudi, and Gemmini, which are not well represented in the training corpus of these models. We prompted Claude Opus to generate code for these DSLs for the `linear_dodge` benchmark from the **blend** suite. The prompt instructions are shown in Fig. 13. In Fig. 14 we show the code generated by the LLM for the three prompts.

Upon analysis, we found that all three generated programs were incorrect. The Gemmini-generated code in Fig. 14a partially uses the correct APIs (`mvin`, `mvouts`), but the computation with `config_ex` is incorrect. For the Gaudi-generated code in Fig. 14b, the model hallucinates the TPC-C library, which does not exist in the actual Gaudi programming model. The MLX-generated code in Fig. 14c has the correct call to the library function add, but the imports are incorrect, making the code non-functional. In addition to the generated code being incorrect, it is challenging to verify these outputs formally as syntactically translating the generated code to SMT-LIB is not trivial. The experiment highlights two significant challenges in generating verified code using LLMs mentioned earlier. In contrast, Tenspiler, which uses a verified lifting-based approach, can easily handle these challenges. LLMs cannot be directly prompted to generate code in new DSLs. LLMs could potentially be fine-tuned or prompted with few-shot learning to generate code in an IR, which can then be utilized within the Tenspiler's framework for verification; however, we leave this as future work.

6.6 Extension to Higher-Dimensional Tensors

Our benchmarks only involve 1D and 2D tensors, as most operations are performed on images (the **blend** benchmarks) and weight matrices (the **LLama** benchmarks). In this section, we demonstrate that Tenspiler can be extended to support higher-dimensional tensors with the generalizability of TensIR and the synthesis optimizations discussed in Sec. 5. Specifically, we extend TensIR to accommodate 3D tensors and all corresponding element-wise operations.

```
#include <gemmini.h>

void linearDodge8_gemmini(uint8_t* base, uint8_t*
active, uint8_t* out, int height, int width) {
  gemmini_config_ex(BANK_SIZE, BANK_NUM, ACCEL_NUM);
  gemmini_flush(0);
  elem_t A[height * width];
  elem_t B[height * width];
  elem_t C[height * width];
  for (int i = 0; i < height * width; i++) {
      A[i] = base[i];
      B[i] = active[i];
  }
  gemmini_config_st(A, height * width);
  gemmini_config_st(B, height * width);
  gemmini_config_ld(C, height * width);

  gemmini_mvin(A, 0);
  gemmini_mvin(B, 1);
  gemmini_config_ex(ELEM_ADD, 0, 0, 0);
  gemmini_mvout(C, 0);

  for (int i = 0; i < height * width; i++) {
    out[i] = (uint8_t)C[i];
  }
  gemmini_fence();
}
```

(a) LLM generated Gemmini Code.

```
#include <tpc.h>

void linearDodge8_tpc(Buffer<uint8_t,2> base, Buffer<uint8_t,2>
active, Buffer<uint8_t,2> out) {
  int height = out.height();
  int width = out.width();
  tpc::nd_range<2> nd_range({height, width}, {1, 1});
  tpc::parallel_for(nd_range, [&](tpc::nd_item<2> item) {
    int row = item.get_global_id(0);
    int col = item.get_global_id(1);
    tpc::vector<uint8_t, 16> base_vec = tpc::load<uint8_t,
16>(base.data() + (row * width + col));
    tpc::vector<uint8_t, 16> active_vec = tpc::load<uint8_t,
16>(active.data() + (row * width + col));
    tpc::vector<uint8_t, 16> result_vec = base_vec + active_vec;
    tpc::store<uint8_t, 16>(out.data() + (row * width + col),
result_vec);
});
}
```

(b) LLM generated Gaudi Code.

```
import mlx

def linear_dodge_8(base, active):
  out = mlx.add(base, active)
  return out.astype(mlx.uint8)
```

(c) LLM generated MLX Code.

Figure 14 LLM generated code for the prompt.

Additionally, we adapt the operator restriction optimization (introduced in Sec. 5.1) to apply to 3D tensors. When the source program returns a 3D tensor, our grammar is restricted to include only element-wise 3D tensor operations. We also retain the program state restriction optimization technique from Sec. 5.2. Furthermore, we extend our support to leverage expression trees performed on individual elements in tensors, as detailed in Sec. 5.3, to guide the search for vectorized operations within 3D tensor spaces.

We evaluate TENSPILER's synthesis optimizations on artificial benchmarks involving 3D tensors. We create these benchmarks by combining random element-wise operations. The maximum depth of these benchmarks is chosen to be 5 to match that of all our existing real-world benchmarks, as described in Sec. 6. Results in Fig. 15 show that the synthesis time grows linearly with the depths of the benchmarks. The depth-1 benchmark synthesizes the fastest in 6 seconds, while the depth-5 benchmark takes the longest, in 184 seconds.

The sharp increase in timing for depth 5 expressions in Fig. 15 is due to the number of expressions we are synthesizing and their complexity. A benchmark with 3 loops involves synthesizing 3 invariants and 1 post-condition, each with expression sizes up to depth 5. Despite these challenges, we easily extend TENSPILER's optimizations and synthesize these benchmarks well within the 1 hour timeout. As future work, to further scale TENSPILER's synthesis algorithm for handling more complex benchmarks, we could explore strategies such as guiding the search process using machine learning techniques, implementing bottom-up synthesis starting with inner loops first, performing bounded synthesis with unrolled loops, and combining these approaches with TENSPILER's current synthesis optimizations.

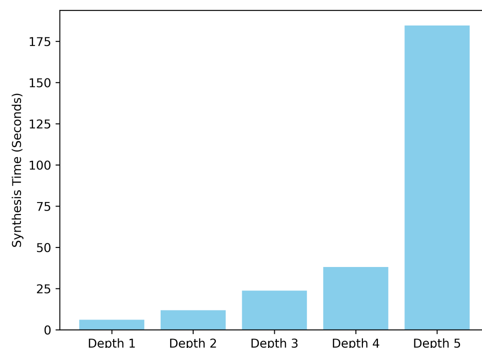


Figure 15 Synthesis timings for artificial 3D tensor benchmarks.

7 Related Work

Verified Lifting. Verified lifting uses program synthesis to translate code instead of designing traditional pattern-matching compilers, and has been used across application domains [15, 5, 6, 19, 21]. Adapting these prior compilers for translation to tensor operations is nontrivial. Tenspiler introduces a novel tensor algebra-based to make synthesis efficient, and supports a diverse set of backends.

Code Translators. Tenspiler differs from other code translation approaches. While symbolic methods like pattern-matching compilers [30] face challenges with the error-prone nature of their rules, Tenspiler uses a search-based approach to avoid these complexities. Neural techniques [32, 26], treat translation as a machine translation task but struggle to ensure correctness. In contrast, Tenspiler uses a theorem prover to guarantee semantic equivalence between the translated and source code. More recently, despite the success of LLMs in programming tasks, they are unable to translate code to unfamiliar frameworks or custom hardware ISAs. Tenspiler's approach of searching in an Tensir and using simple rules for translation makes it easy to support new backends.

Intermediate Representations. LLVM [22], MLIR [23] and TACO's IR [20] are examples of IRs that generate code to multiple backends. LLVM in addition can generate optimized code for various hardware targets. MLIR introduces "dialects," allowing specific optimizations for different domains or hardware targets. Despite their versatility, LLVM and MLIR were originally designed for traditional pattern-matching compilers, posing challenges for search-based compilers due to their extensive set of operators. In contrast, Tensir is designed for expressing tensor operations to be used in search-based compilers. As discussed, Tensir enables efficient lifting, verification, and code generation.

8 Conclusions

We presented our experience in building Tenspiler, a compiler that leverages verified lifting to transpile code to leverage tensor processing infrastructures. At the core of Tenspiler is Tensir which concisely captures various tensor computations. Tenspiler efficiently translates all 69 real-world benchmarks and can generate code to be executed on 6 different software and hardware backends. The generated code achieves an average speedup of **105**× for kernel and **9.65**× for end-to-end execution compared to the input.

References

1. Claude Model. https://www.anthropic.com/news/claude-3-family. [Online].
2. Darknet. http://pjreddie.com/darknet/. [Online].
3. Mathfu. https://github.com/google/mathfu. [Online].
4. Texas Instrument Digital Signal Processing (DSP) Library for MSP430 Microcontrollers. https://www.ti.com/tool/MSP-DSPLIB. [Online].
5. Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1205–1220. ACM, 2018.
6. Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Trans. Graph.*, 38(6), November 2019. doi:10.1145/3355089.3356549.
7. Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 1004–1016, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507714.
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. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013. doi:10.1109/FMCAD.2013.6679385.
9. SMT-LIB Authors. SMT-LIB Standard. https://smtlib.cs.uiowa.edu/. [Online].
10. Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.
11. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1108792.1108813.
12. Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2023.38.
13. L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
14. Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael FP O'Boyle. mlir-synth: Automatic, retargetable program raising in multi-level ir using program synthesis. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 39–50. IEEE, 2023.
15. Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462180.

16 Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL: `https://lmsys.org/blog/2023-03-30-vicuna/`.

17 Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. `doi:10.1109/CVPR.2009.5206848`.

18 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

19 Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *SIGPLAN Not.*, 51(6):711–726, June 2016. `doi:10.1145/2980983.2908117`.

20 Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. `doi:10.1145/3133901`.

21 Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: synthesizing crdts with verified lifting. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1349–1377, 2022.

22 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

23 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020. `arXiv:2002.11054`.

24 llamacpp. `https://github.com/leloykun/llama2.cpp/`, 2024. Accessed: 2024-01-19.

25 José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. C2taco: Lifting tensor code to taco. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2023, pages 42–56, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3624007.3624053`.

26 Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. `doi:10.1145/3527315`.

27 Apple mlx. `https://ml-explore.github.io/mlx/`, 2024.

28 Numba. `https://numba.readthedocs.io/en/stable/cuda/overview.html`, 2024.

29 Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Tenspiler: A verified lifting-based compiler for tensor operations, 2024. `arXiv:2404.18249`.

30 Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 909–927, New York, NY, USA, 2014. ACM.

31 Christopher D Rosin. Stepping stones to inductive synthesis of low-level looping programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33(01), pages 2362–2370, 2019.

32 Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: `https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html`.

33 Mazen AR Saghir. *Application-specific instruction-set architectures for embedded DSP applications.* Citeseer, 1998.

34 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 15–28. ACM, 2016.

35 Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*, pages 364–381. Springer, 2017.

36 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509586.

37 Tensorflow xla. https://www.tensorflow.org/xla/architecture, 2024.

38 Vojin Zivojnovic. Dspstone: A dsp-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994*, pages 715–720, 1994.