mlirSynth: Automatic, Retargetable Program Raising in Multi-Level IR using Program Synthesis

mlirSynth: Automatic, Retargetable Program Raising in Multi-Level IR using Program Synthesis

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

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

Tobias Grosser
University of Edinburgh
United Kingdom
tobias.grosser@ed.ac.uk

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

***Abstract*—MLIR is an emerging compiler infrastructure for modern hardware, but existing programs cannot take advantage of MLIR's high-performance compilation if they are described in lower-level general purpose languages. Consequently, to avoid programs needing to be rewritten manually, this has led to efforts to automatically raise lower-level to higher-level dialects in MLIR. However, current methods rely on manually-defined raising rules, which limit their applicability and make them challenging to maintain as MLIR dialects evolve.**

We present *mlirSynth* – a novel approach which translates programs from lower-level MLIR dialects to high-level ones without manually defined rules. Instead, it uses available dialect definitions to construct a program space and searches it effectively using type constraints and equivalences. We demonstrate its effectiveness by raising C programs to two distinct high-level MLIR dialects, which enables us to use existing high-level dialect specific compilation flows. On Polybench, we show a greater coverage than previous approaches, resulting in geomean speedups of 2.5x (Intel) and 3.4x (AMD) over state-of-the-art compilation flows. mlirSynth also enables retargetability to domain-specific accelerators, resulting in a geomean speedup of 21.6x on a TPU.

I. Introduction

The end of Dennard scaling has led, in recent years, to the development of a diverse range of specialized hardware. Examples include tensor cores (TPU [34] , NVIDIA [39]) and AI specialized accelerators [47]. Such hardware holds the promise of efficient performance, but at the cost of increased programming complexity.

A popular approach to overcoming this challenge is the use of domain-specific programming languages (DSLs), such as Halide [46], TensorFlow [2] and PyTorch [45]. These languages allow programmers to easily specify the essential structure of a problem without concern for low-level details. Crucially, this separation of concerns enables domain-specific compilers [54], [16] to efficiently map programs down to a wide range of idiosyncratic accelerators.

The need for existing code to harness the power of domain specific compilation has been well recognized by the compiler community with the development of MLIR [38]. MLIR is a new extensible representation within LLVM that captures high-level representations of programs. Once programs are expressed in the appropriate MLIR dialect, vendors can develop and exploit an efficient compilation path to their platform.

Need for Raising IRs.. This, however, presents a new challenge: how to translate existing code, currently represented in a low-level intermediate representation (IR), into a higher dialect so as to leverage domain-specific compilation. Furthermore, given the proliferation of MLIR dialects, any compiler technique that attempts to translate from a low to high dialect faces an additional challenge: how to adapt to a world of ever-changing high-level targets.

Previous Approaches.. This lifting of abstraction from a low to a high level is called program lifting [35] or raising [2], [56]. There is a large body of work in this area that has recently received increased interest [41], [9], [44], [8], [22], [18], [33]. In [35], lifting is applied to legacy FORTRAN code to generate high-performance Halide programs. This is achieved by representing both source and target languages in a common internal language and deploying an off-the-shelf program synthesis tool [51], [53], and proving the synthesized target code is equivalent to the original via loop invariants. This was later applied to C++ [6] and expanded into an LLVM-based framework [1]. While powerful, such an approach requires the user to manually define the semantics of all operations in the target language semantics in the common internal language. As the number of target high-level languages diversifies, this is not a scalable approach. There has been adjacent work in replacing code with library calls [23], [40]. Such approaches, however, are also fundamentally non-scalable as they focus on a fixed API rather than the open-ended nature of DSLs and their IRs.

Multi Level Tactics (MLT) [15] more recently, directly addressed this issue by showing raising to a high-level MLIR dialect enabled significant performance improvement. However, their approach requires domain-specific raising rules to be implemented by the compiler writer which are dialect specific. As we show in section VI, they are restricted in the number of programs they can tackle and rules need to be rewritten for each source and target dialect. Ideally, we would like a generic scheme that is robust, raising a large number of programs and is able to target new MLIR dialects without any compiler writer intervention.

Our Approach.. This paper presents mlirSynth, which automatically raises MLIR dialects from low to high-level without any hardwired compiler transformation or raising rules. Instead, mlirSynth automatically uses the available dialect definitions (within MLIR's TableGen [38]) to construct a program space

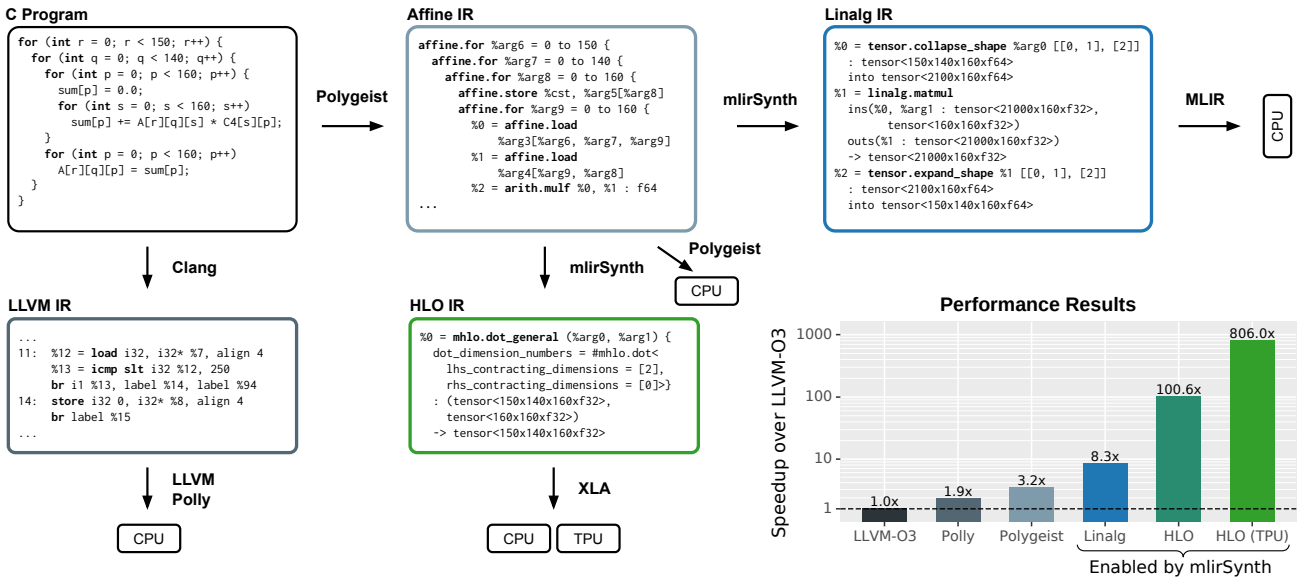

Fig. 1. The `doitgen` computation in different representations and their relative performance on different devices.[2] MlirSynth enables compiling the C program with domain-specific compilers such as MLIR-Linalg and XLA, resulting in significant speedups.

and effectively searches it using candidate equivalences. It is based on bottom-up enumerative program synthesis exploiting type constraints and IO behavioral equivalence to quickly prune the space. Essentially it generates programs in the target dialect, starting from the smallest one first. Then it uses a combination of testing and model checking to identify an equivalent program in the target dialect. A key characteristic of our approach is that it is not tied to one dialect. However, if there is domain specific analysis available, we can use this as a heuristic to speed up the search. Thus we have a domain agnostic raiser that can exploit domain specific analysis where available.

We demonstrate this by lifting to two dialects, Linalg IR and HLO IR, exploiting polyhedral analysis in the synthesis phase. Furthermore, we show that our approach is able to cover a wider set of programs and generate more efficient implementations than MLT [15], the state-of-the-art scheme.

Of the 14 Polybench that can be expressed in these dialects, we are able to raise 13 compared to MLT's 6. By exploiting a dialect specific compiler, we are able to achieve an average 20.8x (20.9x) speedup on an Intel (AMD) platform relative to LLVM-O3. This compares to the 3.3x (4.2x) of MLT and the 6.4x (6.1x) of Polly [31]. As we can raise to HLO, we can also use the XLA compiler to target Tensor Processing Units (TPUs), achieving a 175.3x average speedup.

Our contributions are:

- mlirSynth, a framework for raising low-level MLIR dialects to higher ones
- A scalable method to synthesize code in multiple MLIR dialects, automatically generating the search space based on the dialect's TableGen definition
- A fast bottom-up enumerative search synthesizer exploiting observational equivalence and polyhedral analysis
- Greater coverage, performance and accuracy compared to state-of-the-art raising approaches

II. Motivating Example

To illustrate the benefits of raising, consider the program in Figure 1 in the box labeled `C Program`. This loop nest implements the `doitgen` computation, and was taken from the Polybench benchmark suite.

LLVM IR. As it is written in C, the loop nest is represented within the LLVM compiler by the standard SSA IR form shown in the box labeled `LLVM IR`. If we apply Polly [31], a polyhedral optimizing compiler to this IR, it is able to automatically generate parallel and cache efficient code. In this case, it is able to achieve a 1.9x speedup over the default -03 pass on an Intel i7 platform as shown in the performance results plot.

Affine. Although Polly delivers a significant speedup, if the LLVM IR could be rewritten in an alternative MLIR dialect, then we can potentially achieve greater performance. This is the motivation behind the Affine dialect in MLIR, which captures high level polyhedral information, such as linear array access, convex iteration space and static control-flow. Polygeist [42] is a tool takes in C code and produces Affine IR for appropriate loop nests. They then apply the Pluto [14] polyhedral cache and parallelism optimizer to this Affine IR, which results in a 3.2x speedup. While the performance achieved is greater than Polly, the Affine dialect also acts as a convenient starting point for lifting to higher level dialects and is the source IR for the MLT compiler [15].

Linalg. Consider the Linalg IR version of the program in the box labeled `Linalg` in Figure 1. It is semantically equivalent to the Affine version, but is in a form that the MLIR compiler can generate more efficient code from. Rather than the Affine IR polyhedral representation of the program, Linalg describes

[2]CPU: Intel I7-8700k (6 cores / 12 threads), TPU: Google TPU v3 (8 cores).

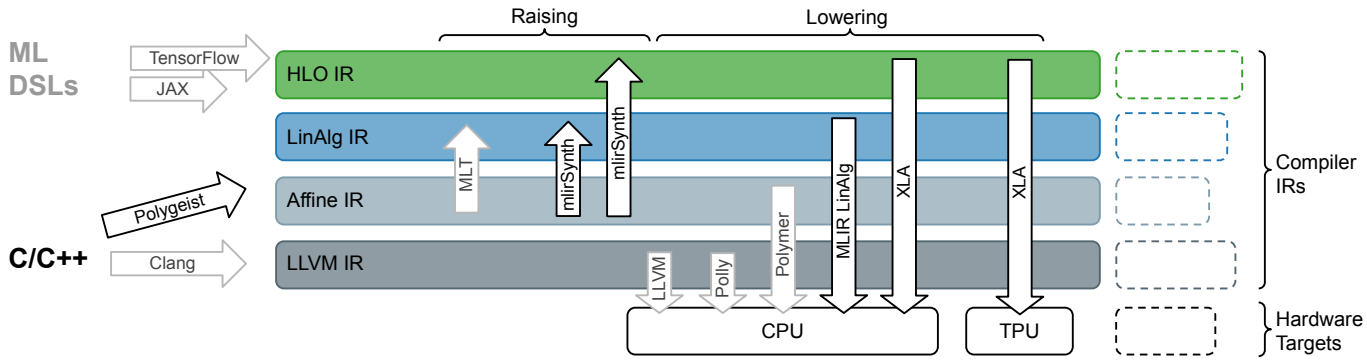

Fig. 2. Compilation flows relevant in this paper, associating programming languages, MLIR dialects, raising methods, compilers and hardware targets. Flows relevant to mlirSynth are highlighted.

the given code as three high level operators: a core matrix multiplication operation, surrounded by two tensor reshaping operations. As matrix multiplication and reshaping are often highly tuned on different hardware targets, it allows the MLIR compiler to exploit kernel libraries and generate a highly efficient implementation. The performance results plot shows that if we were able to lift code to this dialect, we were able to achieve an even greater speedup of 8.3x.

A. Raising IR

While MLT tries to automatically raise the Affine code to the Linalg form, it, unfortunately, fails as its hand-coded matching rules do not consider this IR pattern. If additional patterns were added, then it would achieve a higher speedup.

HLO. Our approach is not limited to one target dialect of MLIR. It is able to raise to the higher level HLO dialect. Figure 1 further shows an implementation of the same computation in HLO (box labeled `HLO`). Rather than the three operations of Linalg, the computation is expressed using a single high-level operation. This allows XLA [48], a compiler for HLO, greater flexibility in its implementation. If we consider the i7 platform, XLA able to achieve a speedup of over 100x relative to LLVM -03, justifying such a dialect. One benefit of this representation is that it can leverage platform specific compilers. If we change the hardware target to a TPU, the XLA compiler delivers a speedup of over 806x.

Summary. Higher-level representations in MLIR allow greater performance than lower-level ones and LLVM IR, as they allow compilation with high-performance domain-specific compilers. To enable such compilers for programs written in lower-level programming languages such as C, a raising technique is required. With the ongoing emergence of new dialects, we need a *flexible* raising technique to automatically leverage high-performance compilation flows.

III. System Overview

In this section, we briefly introduce the notion of dialects within MLIR before describing the mlirSynth design which raises low-level dialects to higher ones.

A. MLIR

MLIR is an infrastructure for developing domain-specific compilers. To aid this, MLIR provides reusable building blocks and shared tools that allow us to define domain-specific languages and their compilation pipelines. The key concept that enables this is a *dialect*.

Dialects define sets of operations, types, and attributes. There are many dialects currently deployed (35 in the MLIR repository) and, crucially for automatic synthesis, each of these is defined by a structured TableGen description which contains the typed operands, attributes, and regions for each operation.

Figure 2 shows a small subset of MLIR's dialects, relevant to this paper and the associated compilation flows. The lowest level IR we consider is LLVM IR, the default format for languages such as C/C++/FORTRAN. From this, the LLVM compiler generates code for all supported platforms. Many higher dialect compilers can progressively lower their dialect to LLVM IR.

Different compilation flows exploit high level dialect information to generate efficient implementations. For instance, Polygeist leverages the polyhedral representation available in Affine IR [42]. While some programming languages / DSLs such as TensorFlow can benefit from the XLA compiler due to its representation as HLO, this is not available to languages such as C. If we can raise code using MLT or mlirSynth to higher MLIR levels, then we can leverage the pre-existing compilation flows for performance. While a lowering path is always provided, raising is considerably more challenging.

B. mlirSynth design

mlirSynth operates in a three stage process that takes in a user program in the Affine IR dialect plus a description of the target dialect and outputs a raised program in the new dialect, as shown in Figure 3. The central idea is that we apply classic program synthesis techniques to lift a dialect to a higher one and then lower both the original and raised program down to the same representation and check they are equivalent. We use smallest-program first enumeration, discarding candidate programs based on type information and guided by heuristics where available. The approach is comprised of 3 stages:

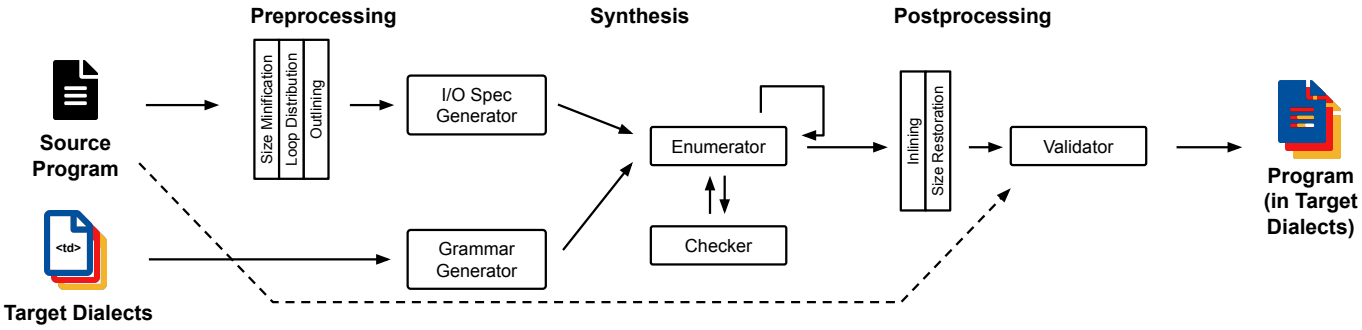

Fig. 3. Design of mlirSynth, showing its processing pipeline and component interaction.

Pre-processing. Initially, we apply pre-processing steps to simplify the program before enumerative synthesis is applied. Specifically, we use polyhedral analysis to distribute the original loop structure into smaller ones that can be synthesized independently and reduce the size of any array/matrix-like data structures in the code.

Enumerative synthesis. The key lifting is done by a classic enumerative synthesis technique. We generate a grammar (box `Grammar Generator`) for the synthesis process from the TableGen for the target dialect (`Target Dialects`), automatically generate an input-output specification (`IO Spec Generator`) for the target program, and then use a bottom-up synthesis process (`Enumerator`) to search the space of possible programs. This space is large, so we guide the enumerative search with a number of heuristics, based on polyhedral analysis and observational equivalence. Lifted candidates that satisfy the input-output specification (`Checker`) can be proved to be equivalent to the original code using bounded model checking (`Validator`).

Post-processing. Once we have a successful candidate, it is inlined back into the program and all data structure sizes are restored. It is now in the appropriate high level dialect and can be mapped to hardware by an appropriate compiler toolchain.

The following section gives details of the enumerative synthesis stage.

IV. Synthesis

We base our synthesis procedure on classic enumerative synthesis algorithms from the literature [12]. The core synthesis loop, shown in Algorithm 1, is two phases: enumerating candidate programs and then checking the candidate against a specification. We thus need two inputs: a grammar that defines the space of possible programs to enumerate and a specification for the synthesized program. In the following section, we describe how we automatically generate grammars and specifications for our synthesis algorithm, the exact details of the enumeration process and the heuristics we use.

A. Grammar generation

In MLIR, dialects are defined in the declarative TableGen language. This language provides a structured way to define new dialects, including specifying the number and types of operands, attributes and regions that are required per operation. Given a target MLIR dialect, we use a custom TableGen extraction tool to generate a simple recursive context-free grammar for our enumerative synthesis process. Specifically, we generate an initial grammar G that contains a single non-terminal for each type in the MLIR dialect, and a production rule for each operator.

This grammar does not precisely capture the syntactic restrictions of the MLIR dialect (since MLIR dialects are in general not context-free languages), but we are able to use inbuilt MLIR syntactic checks to discard any invalid programs later on in the enumerative loop. The grammar is, however, specific enough to rule out the majority of poorly typed programs. This automatic grammar generation simplifies retargeting our technique to new dialects.

B. Specification generation

Given a grammar G and a reference function f that we wish to lift, which takes input x, our goal is to synthesize a function f', such that $\forall x. f(x) = f'(x)$ and f' and f' is in $\mathcal{L}(G)$, the language defined by the grammar.

Checking a function satisfies the full equivalence specification above, for arbitrarily large input data structures, is in general undecidable and a significant challenge to state-of-the-art verification tools. Consequently, we apply a multi-staged synthesis process using two approximations of the specification. In the first specification, $\phi_{\delta-eq}$, we minify the input data structures, reducing x, a potentially very large data structure, to x_{min}, a small, bounded size data structure. We then assert that the relative error between the outputs of the functions is smaller than a small δ. This δ accounts for the deviations introduced by compiler optimizations due to non-associativity of floating-point arithmetic. The second specification checks the observational equivalence of the functions, i.e., it checks for equivalent behavior on a finite set of n inputs. We denote this ϕ_{obs_n} when it checks behavior on n inputs and n is small (i.e., less than 10) and ϕ_{obs_N} when it checks behavior on N inputs, where N is large (i.e., ≥ 10). Formally, the specifications hold

Algorithm 1 Core synthesis algorithm

```
function SYNTHESIZE(f,G)
    C ← initCandidates(f)
    I_n ← genRandomInputs(f, n)
    operations ← pickOperations(f, G)
    while true do
        f' ← enumerate(C, I_n, operations, f)
        I_N ← genRandomInputs(f, N)
        if specCheck(I_N, f, f') then
            return f'
        else
            I_n ← genRandomInputs(f, n)
```

for a given candidate f' under the following conditions:

$$\phi_{\delta-eq} \Leftrightarrow \forall x_{min}.abs(f(x_{min}) - f'(x_{min}))/f(x_{min}) < \delta$$

$$\phi_{obs_n} \Leftrightarrow \forall i \in I_n.f(i) = f'(i)$$

where I_n is a small finite set of n inputs

$$\phi_{obs_N} \Leftrightarrow \forall i \in I_N.f(i) = f'(i)$$

where I_N is a large finite set of N inputs

We automatically generate ϕ_{obs_n} and ϕ_{obs_N} by randomly generating sets of inputs. We then check for observational equivalence by compiling and executing both f and f' on the inputs, shown in Algorithm 3. The data for ϕ_{obs_n} is initially sampled from the range $[-10, 10]$ to work around the effect of numerical instabilities and to avoid cases where it is likely that the synthesized code will crash.

For speed, the core enumerative algorithm, Algorithm 2 checks candidates against ϕ_{obs_n}. Initially, I_n contains only a single input example, although this input is a high-dimensional tensor or matrix. The outer loop checks candidates against ϕ_{obs_N}. We use CBMC [36], a bounded model checker, to check $\phi_{\delta-eq}$ post-synthesis.

Algorithm 2 Enumeration

```
function ENUMERATE(C, I_n, operations, f)
    while true do
        for op in operations do
            ops ← filterTypes(C, op)
            attr ← genAttrs(op)
            regs ← genRegions(op)
            for f'in cartesianProduct(ops, attr, regs) do
                if not staticCheck(f') then
                    continue
                if observationallyUnique(C, f') then
                    C ← C ∪ f'
                if specCheck(I_n, f, f') then
                    return f'
```

C. Bottom-Up Enumeration

The core enumeration is a bottom-up synthesis algorithm inspired by [7]. The enumeration combines previous candidates with each other to generate more complex ones until a candidate matching the specification is found. An example of this is shown in Fig. 4.

Algorithm 3 Specification checking

```
function SPECCHECK(I, f,f' )
    for i in I do
        if f(i) ≠ f'(i) then
            return false
    return true
```

Initialisation. We start by creating a candidate set C of valid (i.e., well-formed) candidates that each produce a computationally distinct value from the other candidates. We initialize this set with candidates that return the arguments of the reference function f, as shown in the left-most `Candidate Set` box in Figure 4, as well as simple constants in the shape and data type of the arguments and results of the function. We will use this active candidate set as the base of our enumeration.

Enumeration. The synthesis loop enumerates through the set of operations in the grammar. For each operation, we first identify sets of possible operands, attributes and regions. We do this according to the operation signature in the grammar.

We populate the set of possible operands with all expressions in the candidate set of the correct type, highlighted yellow in each iteration in Fig 4. For operators with y operands of type τ, we add each active candidate of type τ to the set of possible operands y-times to allow operators to have two or more identical operands.

For the attributes, which need to be known statically, we generate a large number of them, depending on their type, once the operator is selected. For regions, which contain groups of operations with arguments, we generate simple ones that perform binary mathematical operations on the function arguments (specifically, the operations addition, subtraction, multiplication and division). Region generation is not shown in Fig. 4, but these are generated in each iteration after the operator is selected.

We generate a set of all possible candidates by taking the Cartesian product of sets of operands, attributes and regions.

Candidate Checking. Each candidate in the set is validated using a series of static checks, ordered by their complexity, such that the cheapest checks are performed first, and the expensive checks are performed last. We use MLIR's type and shape inference system and built-in verification method chain to perform these checks.

Equivalence pruning and validating the candidate. If the static checks succeeded so far, we use MLIR's execution engine to just-in-time compile the candidate. We then check ϕ_{obs_n} by executing the candidate program f' on the set of inputs and comparing the output value with the output value produced by the reference function. If ϕ_{obs_n} is satisfied, we send the candidate to the outer loop check.

We also check if the candidate is observationally unique on the inputs used by $specObs$, that is, there does not exist a

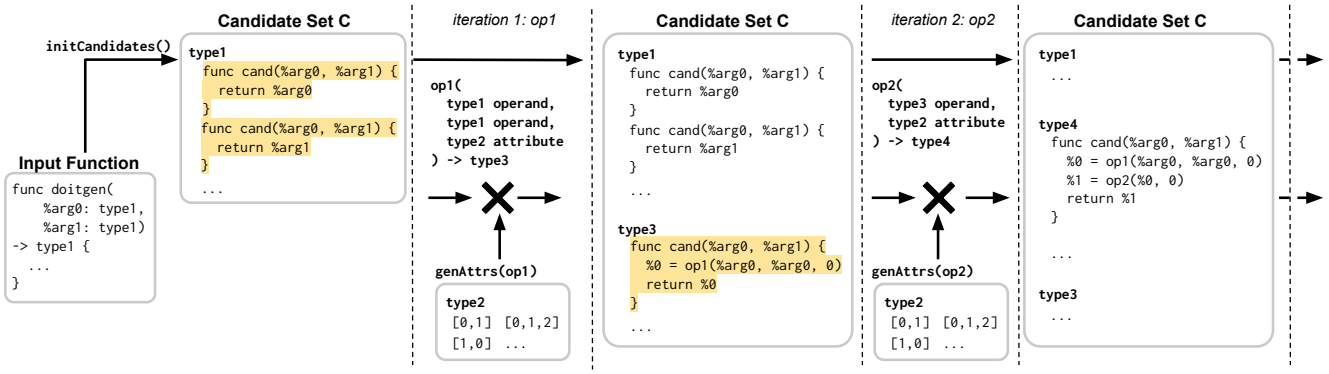

Fig. 4. Synthesis example showing the enumeration of the candidate set at each iteration. In each iteration, an operator is selected, and the active set (highlighted in yellow) is chosen based on the types of the operands, and a set of attributes and regions is generated. We enumerate through the cartesian product until a correct candidate is found. If no correct candidate is found, any observationally unique candidates are added to the candidate set and we start a new iteration with a new operator.

candidate in the candidate set C that behaves the same as f' on all inputs. In other words, if f' is observationally unique, the following formula is valid $\nexists f_c \in C. \forall i \in I_n f'(i) = f_c(i)$. If this is the case, we add f' to the candidate set C. If not, we discard the candidate.

This central enumeration process is repeated for each operation, until either a program matching the specification ϕ_{obs_n} is found, or a timeout expires. Once a candidate satisfies ϕ_{obs_n}, it is then checked against ϕ_{obs_N}. If a candidate fails this check, we can restart the synthesis loop with a new set of random inputs for ϕ_{obs_n}.

D. Heuristics

Given the search space, heuristics are essential for selecting a set of operations to enumerate. We implement two heuristics: polyhedral model-based and dialect-based heuristics. These heuristics alter the behavior of the function `pickOperations`. Polyhedral model-based heuristics perform a value-based dependence analysis on the reference function to identify reduction dependencies, which are visible as cycles in the polyhedral dependence graph. If such reduction dependencies exist, this heuristic selects reduction operations in the target dialect. For a more detailed discussion of this see [24], [57]

Dialect-based heuristics look at the grammar for the target dialect and the source dialect, and, if an operator is present in the source function, written in the source dialect, it is prioritized in the target dialect. For example, if an add operation exists in the function, this heuristic selects any arithmetic operations that perform an addition in the target dialect. We evaluate the impact of heuristics in Section VI-E.

E. Translation Validation

We use CBMC [36] to perform a post-synthesis check for the specification $\phi_{\delta-eq}$. CBMC uses symbolic execution to generate a logical formula that is satisfiable if and only if the two functions are not equivalent (or, in path-based mode, multiple logical formulas representing different paths through the program). If the functions are not equivalent, CBMC will generate a *counterexample*, in the form of a set of inputs for which the outputs of the two functions are not identical. If this were to happen, we could repeat the synthesis process, using the counterexample input generated as one of the input examples in specification ϕ_{obs_n}. If CBMC exceeds a timeout of 1 hour, we substitute this check with extensive testing of ϕ_{obs_N}.

TABLE I
DIALECT COVERAGE ACROSS POLYBENCH

Category	Benchmark	Category	Benchmark
datamining	correlation	blas	syrk
	covariance		syr2k
kernels	atax		trmm
	2mm		gemver
	3mm		symm
	doitgen		gesummv
	mvt		gemm
	bicg		

V. EXPERIMENTAL SETUP

This section briefly describes the platforms, benchmarks and various compilation flows used to compare against mlirSynth.

A. Platforms

All experiments were performed on two multi-core hardware platforms an Intel i7-8700k and an AMD Ryzen 9 3900X with multi-threading enabled. We also evaluated on 1 domain specific accelerator, the Google TPU v3.8.

B. Benchmarks

We evaluated all techniques on those benchmarks from Polybench that can be represented in either Linalg and/or HLO. These are shown in Table I. We used Polybench 4.2.1-beta in the large data size and float configuration.

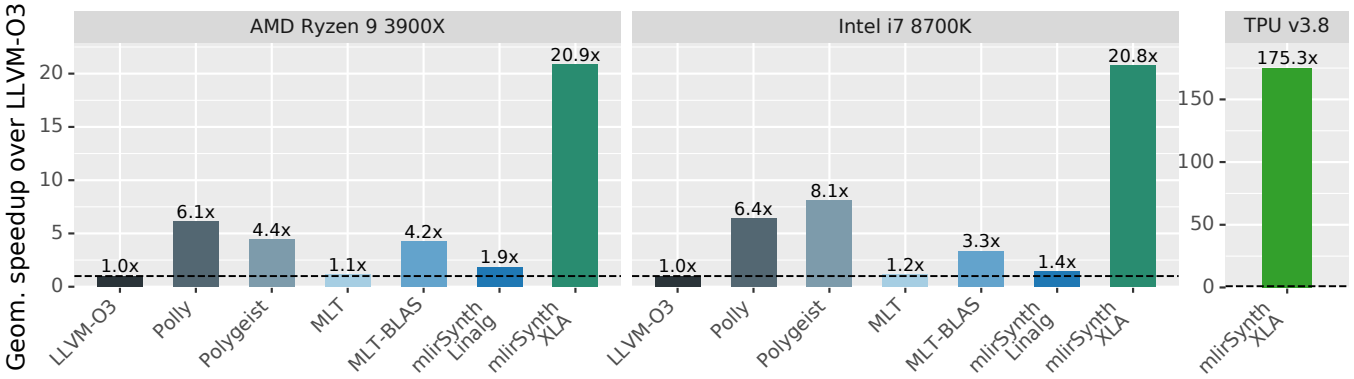

Fig. 5. Geo-mean speedup of each compilation approach on 3 different hardware platforms: Intel CPU, AMD CPU, and TPU.

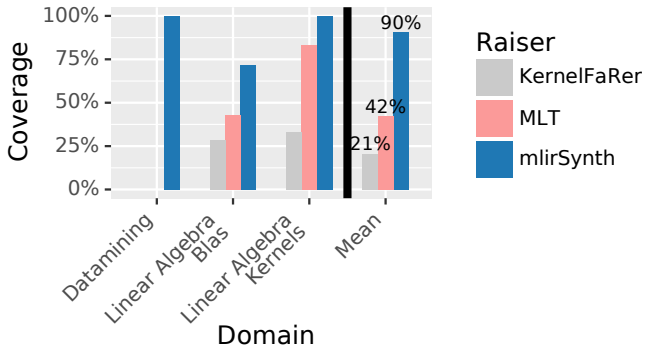

Fig. 6. Comparison of coverage across Polybench benchmarks using different raising techniques.

C. Compilation flows

We evaluated a number of different compilation flows:
LLVM-O3: General-purpose compiler, used as baseline [37]
Polly: Polyhedral compiler, optimizes LLVM IR for caches and parallelism [30]
Polygeist: Polyhedral compiler, takes C to Affine IR, then uses Pluto for cache- and parallelism optimization [42]
MLT: Raises Affine to Linalg IR before invoking a tuned MLIR Linalg compilation flow [15]
MLT-BLAS: As MLT, replacing named Linalg operations with BLAS calls after raising [15]
mlirSynth-Linalg: Our approach, raises Affine to Linalg IR (on tensors) and invokes latest untuned MLIR Linalg compiler
mlirSynth-XLA: Our approach, raises Affine to HLO IR and then invokes the XLA compiler targeting CPUs
mlirSynth-XLA (TPU): As mlirSynth-XLA, targeting TPU

D. Methodology

All experiments were run 10 times with median end-to-end execution time reported. To ensure there was no caching, we ensured a cold start for each experiment, spawning a new process for each run. To further evaluate our raising ability we compare against MLT [15] and KernelFaRer [23], which are state-of-the-art methods for raising to high-level operations from lower-level code.

mlirSynth-Linalg uses the latest default MLIR lowering compiler without any tuning or replacement of named linalg operations with BLAS calls. MLT, however, uses a legacy version of MLIR tuned for performance (e.g. optimized tile sizes). To provide a fair comparison, we retain MLT's use of this tuned legacy compiler. mlirSynth-Linalg currently targets the exact same subset of Linalg as MLT to allow side-by-side comparison. This, however, restricts the number of programs that can be raised.

While mlirSynth could raise programs to combinations of target dialects, we are limiting the experiments to individual ones to allow a more tractable search. We plan to explore combinations of target dialects in future work.

VI. Evaluation

This section first summarises the performance achieved by each approach before examining coverage. This is followed by a detailed performance comparison and an analysis of mlirSynths compilation time. It concludes with an validity evaluation.

A. Overall Summary

Figure 5 shows the average speedups of the various compilation flows on three platforms across the Polybench benchmarks. Speedups are relative to LLVM-O3. MLT lifting to Linalg achieves a 1.1x speedup on the AMD platform, and 1.2x on Intel. Replacing named operations with BLAS routines however achieves 3.3x improvement on the Intel platform, rising to 4.2x on AMD. Although significant, MLT-BLAS's performance is limited by the number of kernels it can raise. In fact, Polly is able to achieve greater improvement: 6.4x and 6.1x speedup on each platform as it can optimize more kernels. Polygeist is able to exploit the Affine IR representation, achieving 8.1x and 4.4x speedups. mlirSynth-Linalg is able to synthesize a larger number of kernels than MLT, which gives a performance improvement across both CPU platforms. However, MLT-BLAS uses a tuned MLIR legacy compiler and substituted BLAS routines gives increased performance. When lifting to HLO and invoking the XLA compiler, mlirSynth-XLA achieves a geometric mean speedup of 20.8x on the Intel platform, rising to a geometric mean of 20.9x on the AMD. This significant increase is because XLA makes use of vendor-optimized kernel

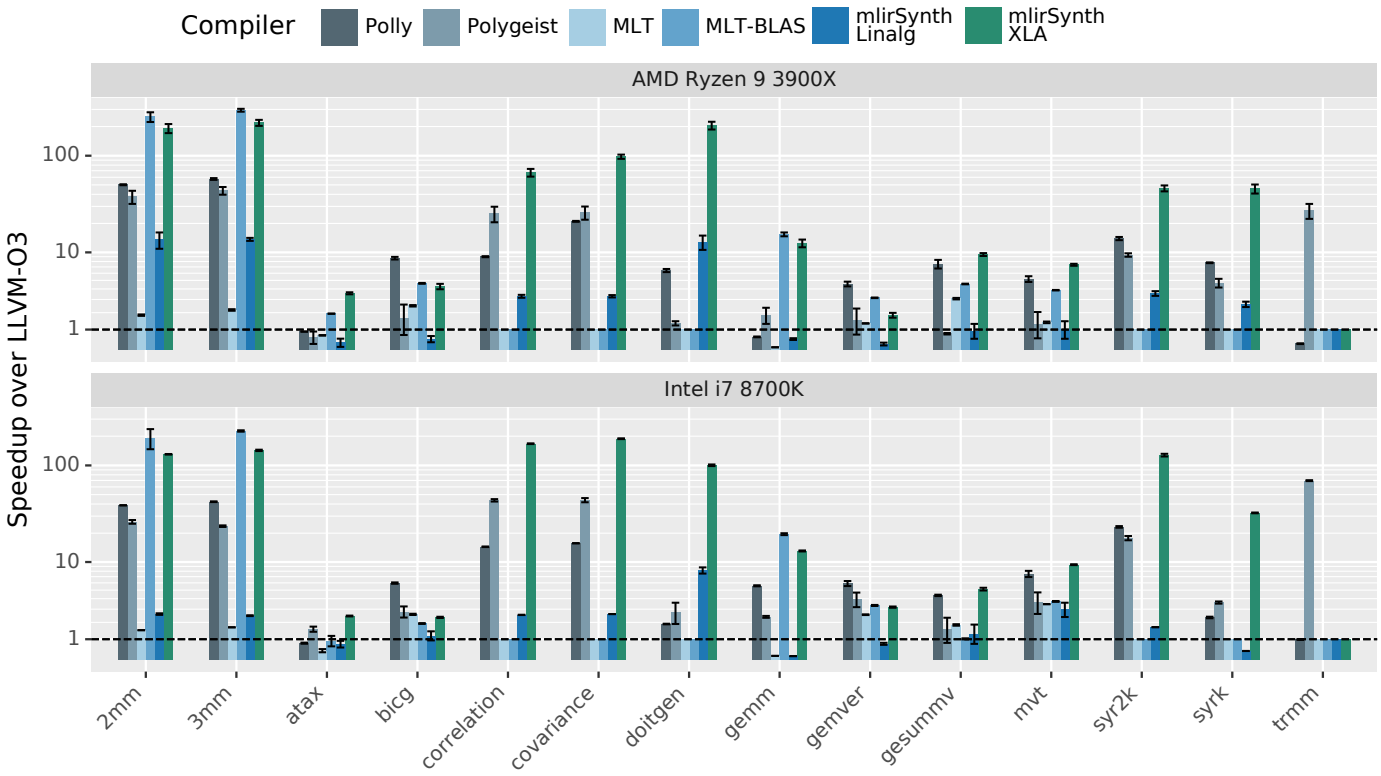

Fig. 7. Detailed speedups on the CPU platforms. Bars show median, lines the standard deviation of 10 runs.

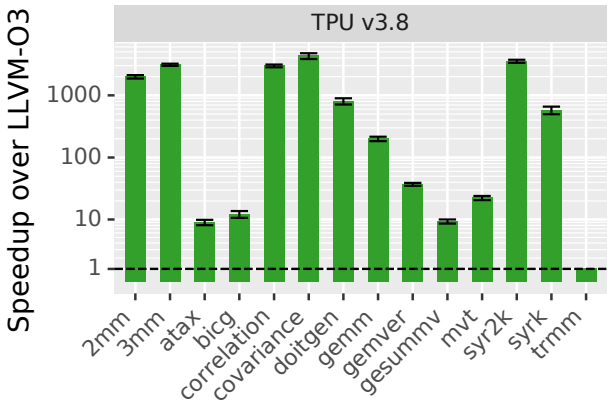

Fig. 8. Detailed speedups on TPU over the Intel i7 8700K. Bars show median, lines the standard deviation of 10 runs.

libraries. When targeting the TPU, mlirSynth is able to achieve over 175x improvement, a significant result.

B. Coverage

The performance achieved by raising critically depends on the number of raised programs. Figure 6 shows the percentage of programs raised by different approaches for each of the Polybench categories shown in Table I. KernelFarer [23] is a robust GEMM detector and is able to detect two routines in the BLAS and kernel category, but no others due to its hardwired pattern matching rules. MLT performs better capturing most of the kernels (5 out of 6) and some of the BLAS (3 out of 7) categories. It is unable to capture any of the data mining kernels due to its restricted Linalg coverage. When raising to HLO, mlirSynth is able to raise all candidates except for `trmm`. Its computation cannot be represented in HLO operations and therefore, results in raising to fail.

C. Detailed CPU performance results

Figure 7 shows a more detailed performance evaluation of different compilation paths relative to LLVM-O3. We show Polly and Polygeist as the polyhedral compilers, MLT, MLT-BLAS, finally mlirSynth raising Linalg and XLA operations.

MLT-BLAS is able to improve on Polly on the Intel platform in 3 cases where large matrix multiplications dominate execution time, *2mm, 3mmm, gemm*. On the AMD platform, in addition, it performs well on `atax`. Overall however it performs less well than Polly as it is unable to raise all the benchmarks to Linalg.

As mlirSynth-Linalg is able to synthesize a larger number of kernels than MLT it achieves performance improvement in seven of the benchmarks. In 2 cases, `bicg` and `gemver` MLT performs better due to its tuned legacy MLIR compiler.

Comparing mlirSynth-XLA to MLT-BLAS, we see MLT-BLAS is able to achieve comparable performance to mlirSynth-XLA on matrix-matrix multiply like kernels (`2mm`, `3mm`, `gemm`). This is because XLA uses similar BLAS kernel libraries like MLT, particularly on the Intel platform. On other programs such as `mvt`, mlirSynth-XLA is superior as it synthesizes a more efficient, but computationally equivalent program.

It is clear that raising beyond Linalg to HLO enables significant performance improvement due to the superior XLA

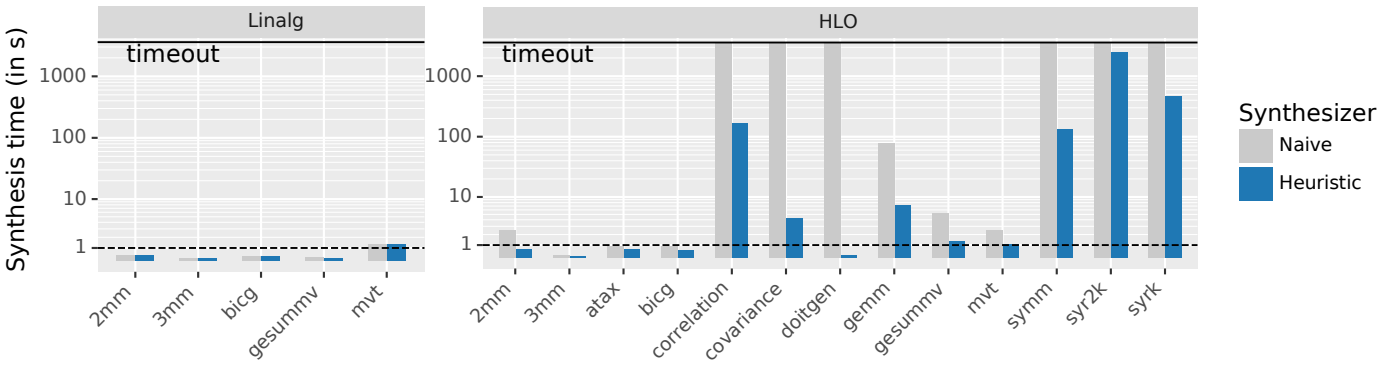

Fig. 9. Synthesis times in seconds of different benchmarks in the Linalg and HLO dialects.

TABLE II
SYNTHESIS STATISTICS OF MLIRSYNTH ON HLO IN HEURISTIC MODE. STATIC CHECKS INCLUDE FILTERING BASED TYPE CORRECTNESS AND ADDITIONAL CHECKS VIA DIALECTS VERIFICATION INTERFACE. THE FINAL COLUMN SHOWS THE LEVEL OF POST-SYNTHESIS GUARANTEES (SECTION VI-F).

Benchmark	Enumerated	Static filtered	Evaluated	Equiv filtered	Ops (max)	Synth time	Formal Guarantee
2mm	49067	46504	1043	709	7 (3)	0.65s	ϕ_{int-eq}
3mm	2484	2409	3	0	3 (1)	0.14s	ϕ_{int-eq}
atax	18960	17042	1166	763	8 (3)	0.62s	$\phi_{\delta-eq}$
bicg	18961	17046	1173	771	8 (3)	0.59s	$\phi_{\delta-eq}$
correlation	1420241	1173035	188577	159679	22 (3)	174.11s	ϕ_{obs_N}
covariance	382674	374083	5799	2049	6 (3)	4.21s	$\phi_{\delta-eq}$
doitgen	9972	9879	71	18	1 (1)	0.16s	ϕ_{int-eq}
gemm	607638	572798	13695	6745	4 (3)	7.26s	ϕ_{int-eq}
gesummv	29221	24566	3919	2333	9 (3)	1.37s	$\phi_{\delta-eq}$
mvt	27977	24460	2855	1631	4 (2)	1.09s	ϕ_{obs_N}
symm	5353361	4943595	309752	163310	4 (4)	134.85s	ϕ_{int-eq}
syr2k	20820281	18547932	1467901	1022725	12 (5)	2438.69s	ϕ_{int-eq}
syrk	3532229	2954620	433798	297594	7 (5)	467.79s	ϕ_{int-eq}

compilation flow. On both the CPU platforms, in all cases, it outperforms the corresponding Linalg implementations. While it is less performant than polyhedral compilers for small problems, on average it significantly outperforms them.

D. TPU results

If we consider the domain-specific TPU accelerator, then HLO enables even greater performance improvement across the benchmarks, as shown in Figure 8. The XLA compiler is able to achieve over 3000x speedups in some cases, performing particularly well on programs containing large matrix operations.

E. Synthesis time

While mlirSynth is able to raise programs to higher-level dialects, it requires search-based synthesis which could be expensive. Figure 9 shows the synthesis time for each benchmark for each dialect and compares it against a naive synthesizer that does not restrict the types and the set of operations.

In 7 out of 13 cases, we are able to raise programs in less than 2 seconds. In the 4 more complicated examples, synthesis time increases to over 2 minutes. The impact of type information and candidate pruning is significant. Compared to the naive algorithm, we are able to reduce synthesis time by an order of magnitude.

Table II provides a more detailed breakdown of the synthesis process for HLO. The synthesis time is correlated to the largest synthesis subproblem size, whereas the largest solved one was 5 high-level operations. While the number of candidates considered varies from 720 to c. 20 million, over 90% of these can be discarded based on type and shape filtering. The remaining candidates are then evaluated, with those that are found to be equivalent to existing candidates eliminated from further consideration. The number discarded this way varies and increases in impact as the number of enumerated candidates increases: from 0% for 3mm up to 60% for symm.

F. Validity

As described in section IV-E, we employ model checking [36] to determine if our raised program is equivalent to the original [17]. In our post-synthesis checks, 4 solutions are proven to be equivalent or δ-equivalent (satisfying $\phi_{\delta-eq}$). CBMC failed to find any errors within 1 hour when checking $\phi_{\delta-eq}$ for 8 of the remaining HLO solutions, but we were able to prove that these are equivalent when using integers and discounting floating-point arithmetic. That is, the specification ϕ_{int-eq} was shown to hold, where $\phi_{int-eq} \Leftrightarrow \forall x \in X_{int}.f_{int}(x) = f'_{int}(x)$ and X_{int} is the set of all possible integers and f_{int} and f'_{int} use integer arithmetic throughout. We are unable to verify the remaining two solutions (correlation and

mvt) due to bugs in CBMC but all solutions passed extensive testing, i.e., ϕ_{obs_N} holds. We identify 2 issues:

1) All queries use small input data structures so there is a risk that bigger floating-point errors will accumulate on larger data structures. There are also some solutions where CBMC times out and we default to using integer arithmetic, so there is a small risk that an error trace for floating point might exist. These risks are mitigated by testing performed on large numbers of input examples with variable data structure sizes.

2) The specification $\phi_{\delta-eq}$ permits a relative error of $\delta = 10^{-5}$. δ is chosen to account for subtle differences introduced by the MLIR and XLA compilers due to the order in which floating-point operations are performed but this results in a precise verification tool like CBMC reporting that the synthesized functions are not equivalent. For practical purposes, we can thus consider the solutions that are proven to satisfy $\phi_{\delta-eq}$ to be equivalent.

G. Discussion

Enumerative search is able to raise programs to two MLIR dialects and leverage the pre-existing compiler flows to deliver portable, high-performance code. In particular, it enables access to accelerators such as TPU from low-level languages. While IO testing in practice is sufficient for correct lifting, verification is needed to guarantee correctness. Existing raising schemes such as MLT provide no such guarantee. If the compiler writer inadvertently inserts an incorrect rule, it is not checked. In fact, we discovered that MLT incorrectly identified an in-place matrix update as a functional operation and actually gave incorrect results in one case. Synthesis times for bottom-up enumeration scales with program complexity. For more complex dialects, smarter sketch-based or probabilistic schemes will be useful and the subject of future work.

VII. Related Work

Pattern matching raising. The raising of LLVM IR to a higher level MLIR has been investigated in MLT [15]. It develops a language that describes Affine IR [42] patterns and their corresponding replacement in Linalg IR. While flexible, it requires the writing of matching code for each pattern of interest. Furthermore, the replacement, or builder, code has to be rewritten for every target and is not scalable with IR evolution. A similar approach was investigated in [28] where an external constraint language is used to pattern match LLVM IR. Unlike MLT it replaces matches with calls to external APIs and again has to be rewritten for changing targets.

API replacement. Replacing matched code/IR to a fixed API call [19] is a limited form of raising. KernelFarer [23] works at the program level and restricts its attention to just GEMM API targets, but is more robust than IDL matching significantly more user code. This robustness is extended further in [55], [40] which uses behavioral equivalence to match code. Such approaches, however, are intrinsically limited as they focus on fixed APIs rather than the open-ended nature of DSLs and their IRs.

Raising with synthesis. Using program synthesis to generate programs from a specification is a long-studied area [25], [50]. Using a low-level program as the specification and a high level-one as the target was tacked in [35]. Here appropriate stencil like loops in FORTRAN are lifted to their equivalent in Halide [46]. This has been extended to a more generic LLVM framework [3] based on a common IR. While this has the potential to allow lifting to multiple targets [5], [4], it requires the compiler writer to provide a compiler and decompiler from each potential source and target into the IR which is not scalable. MLIR-Fuzz [27] offers a fuzzer `mlir-enumerate` which enumerates type-correct MLIR programs bottom-up for any dialect by translating MLIR's TableGen files into the dialect definition language IRDL [26].

Example driven synthesis. The use of input/output examples to synthesize high-level code has been explored in a number of projects [32], [58], [21], [20]. It has been used to generate pytorch or tensor-flow code from tensor inputs [49], [43]. TF-coder [49] uses type- constraints and equivalences to efficiently apply enumerative program search while [43] uses a DeepCoder [11] style predictive model to guide code generation. AutoPandas [13] uses a more powerful graph neural network based model to guide the generation of Panda code. As there is no ground-truth program to lift, just examples, such schemes cannot be directly used for IR raising. Furthermore, both source format and target output are hardwired for each domain.

Polyhedral compilation. The use of polyhedral analysis to drive program optimization [14] has been extensively explored in the compilation community [31]. It has been used for driving systolic code generation [29], memory hierarchy optimization, parallelization and GPU code generation [10] and forms the core for many modern tensor algebra compilers. Polly [30] is able to generate efficient cache optimized and parallel code directly from LLVM IR.

Low-level loop and memory reference representation in LLVM IR can make analysis difficult. This has motivated LLVM IR extensions to facilitate parallelization [52] and motivated Polygeist, a C to Affine IR compiler [42] that uses Pluto [14] to generate cache optimized and parallel code. All of these approaches use polyhedral analysis to lower code, rather than mlirSynth which uses it to raise dialect levels.

VIII. Conclusion and Outlook

This paper presents a bottom-up, enumerative synthesis approach to raising dialect levels within MLIR. The retargetable approach is applied to Affine IR, raising it to Linalg and HLO. It is applied to PolyBench and when the raised code IR is compiled to three platforms, it outperforms existing compilation flows. Future work will raise to multiple target dialects, which needs a faster and more scalable synthesis algorithm. We plan to improve the synthesis search by re-using previous program space explorations, and the full integration of model checking into the synthesis process. We will also evaluate raising to new and emerging dialects of MLIR and apply to larger benchmark suites.

References

[1] Metalift. https://metalift.pages.dev/. Accessed: 2023-04-13.

[2] Martín Abadi. Tensorflow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 1–1, 2016.

[3] Maaz Bin Safeer Ahmad and Alvin Cheung. Leveraging parallel data processing frameworks with verified lifting. *arXiv preprint arXiv:1611.07623*, 2016.

[4] Maaz Bin Safeer Ahmad and Alvin Cheung. Optimizing data-intensive applications automatically by leveraging parallel data processing frameworks. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1675–1678, 2017.

[5] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1205–1220, 2018.

[6] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics (TOG)*, 38(6):1–13, 2019.

[7] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 934–950, 2013.

[8] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[9] Kevin Angstadt, Jean-Baptiste Jeannin, and Westley Weimer. Accelerating legacy string kernels via bounded automata learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–249, 2020.

[10] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[11] M Balog, AL Gaunt, M Brockschmidt, S Nowozin, and D Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR 2017)*. OpenReview. net, 2017.

[12] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[13] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[14] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[15] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level IR. In *CGO*, pages 15–26. IEEE, 2021.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*, pages 579–594, 2018.

[17] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[18] Bruce Collie, Philip Ginsbach, and Michael FP O'Boyle. Type-directed program synthesis and constraint generation for library portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 55–67. IEEE, 2019.

[19] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael FP O'Boyle. M3: Semantic api migrations. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 90–102, 2020.

[20] Bruce Collie and Michael FP O'Boyle. Program lifting using gray-box behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 60–74. IEEE, 2021.

[21] Bruce Collie, Jackson Woodruff, and Michael FP O'Boyle. Modeling black-box components with probabilistic synthesis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 1–14, 2020.

[22] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–671, 2020.

[23] Joao PL De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. Kernelfarer: replacing native-code idioms with high-performance library calls. *ACM Transactions On Architecture And Code Optimization (TACO)*, 18(3):1–22, 2021.

[24] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. *arXiv preprint arXiv:1505.07716*, 2015.

[25] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. Gradual synthesis for static parallelization of single-pass array-processing programs. *ACM SIGPLAN Notices*, 52(6):572–585, 2017.

[26] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. Irdl: an ir definition language for ssa compilers. pages 199–212, 06 2022.

[27] Mathieu Fehr, John Regehr, and Tobias Grosser. Fuzzing tools for mlir. https://github.com/opencompl/mlir-fuzz, 2022. Accessed: 2022-10-22.

[28] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael FP O'Boyle. Automatic matching of legacy code to heterogeneous apis: An idiomatic approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–153, 2018.

[29] Thomas Gross and Monica S Lam. Compilation for a high-performance systolic array. *ACM SIGPLAN Notices*, 21(7):27–38, 1986.

[30] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

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

[32] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[33] Niranjan Hasabnis and R Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 311–324, 2016.

[34] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[35] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *ACM SIGPLAN Notices*, 51(6):711–726, 2016.

[36] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014.

[37] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[38] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[39] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[40] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael FP O'Boyle. Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 85–97, 2023.

[41] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 391–402, 2015.

[42] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising c to polyhedral mlir. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59. IEEE, 2021.

[43] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. Predictive synthesis of api-centric code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 40–49, 2022.

[44] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR, 2019.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[47] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. In *2020 IEEE high performance extreme computing conference (HPEC)*, pages 1–12. IEEE, 2020.

[48] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[49] Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2):1–36, 2022.

[50] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 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 15*, pages 395–414. Springer, 2014.

[51] Armando Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems: 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings 7*, pages 4–13. Springer, 2009.

[52] Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky, et al. Llvm compiler implementation for explicit parallelization and simd vectorization. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–11, 2017.

[53] 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*, pages 135–152, 2013.

[54] Yuanbo Wen, Qi Guo, Zidong Du, Jianxing Xu, Zhenxing Zhang, Xing Hu, Wei Li, Rui Zhang, Chao Wang, Xuehai Zhou, et al. Enabling one-size-fits-all compilation optimization for inference across machine learning computers. *IEEE Transactions on Computers*, 71(9):2313–2326, 2021.

[55] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael FP O'Boyle. Bind the gap: Compiling real software to hardware fft accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 687–702, 2022.

[56] S Bharadwaj Yadavalli and Aaron Smith. Raising binaries to llvm ir with mctoll (wip paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 213–218, 2019.

[57] Cambridge Yang, Eric Atkinson, and Michael Carbin. Simplifying dependent reductions in the polyhedral model. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–33, 2021.

[58] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. *Advances in neural information processing systems*, 31, 2018.