Polygeist: Affine C in MLIR

William S. Moses
MIT CSAIL
wmoses@mit.edu

Lorenzo Chelini
TU Eindhoven
l.chelini@tue.nl

Ruizhe Zhao
Imperial College London
ruizhe.zhao15@imperial.ac.uk

Oleksandr Zinenko
Google
zinenko@google.com

Abstract

We present Polygeist, a new tool that reroutes polyhedral compilation flows to use the representation available in the recent MLIR compilation infrastructure. It consists of two parts: a C and C++ frontend capable of converting a wide variety of existing codes into MLIR suitable for polyhedral transformation, and a bi-directional conversion between MLIR's polyhedral representation and existing polyhedral exchange formats. We demonstrate Polygeist's flow by converting the entire Polybench/C benchmark suite into MLIR, and by performing an IR-to-IR optimization leveraging an existing polyhedral compiler (Pluto). Our flow produces results within 1.25% of the state-of-the-art Clang compiler, enabling direct comparison of source-to-source and IR-to-binary compilers. We believe Polygeist can improve the interoperation between MLIR and the existing polyhedral tooling, benefiting both the research and the production compiler communities.

1 Introduction

The polyhedral model has remained on the cutting edge of research into compiler optimizations for several decades [15]. It provides deep loop analysis and restructuring capabilities by transforming the input program into a mathematical abstraction based on integer sets and binary relations, reasoning on this abstraction and generating the new, optimized code. The process of transforming the program from the representations commonly used in production compilers such as LLVM intermediate representation (IR) [21] or syntax tree is non-trivial [16, 29], and the inverse process is even more complex [3, 18, 31]. This process, together with high algorithmic complexity of the underlying transformation mechanism, has led to the polyhedral optimization being rather poorly adopted by compilers beyond research.

MLIR is a new compiler infrastructure proposed and developed in the scope of the LLVM project [22]. One of its design goals is to provide a production-grade infrastructure that simplifies the expression of advanced compiler optimization, in particular those that require to cast the input program into an additional, higher-level abstraction. Given the growing evidence that the polyhedral model is one of the best frameworks for efficient transformation of machine learning programs [13, 25, 35] and is particularly well suited for existing and emerging accelerator architectures [14, 33, 37], MLIR has always considered the polyhedral representation as a first-class citizen in its infrastructure [12]. Its approach to polyhedral optimization attempts to address the complexity and comprehensibility issues by designing and implementing all relevant algorithms from scratch, and by using a simplified representation that updates the code after each transformation instead of relying on a monolithic code generation mechanism [10].

The design of MLIR's affine representation [11] makes it challenging to directly apply existing polyhedral tools, which are often based on `isl` [36] or `Polylib` [23] and designed for C source-to-source transformation [7, 37]. Additionally, benchmarks used in the literature on polyhedral compilation are commonly written in C [30] and are not benefiting from the higher level representations available in MLIR [34]. As a result, empirical comparisons can only be performed for the entire end-to-end compilation flows, and it is difficult to identify to which extent each part of the flow (high-level representation, polyhedral transformation, post-polyhedral downstream compiler) affects the final performance.

We address these issues in two ways. First, we create a compilation flow that connects MLIR polyhedral representation to existing tools such as Pluto [7] and CLooG [3]. Our bi-directional conversion between the MLIR Affine dialect [11] representation and the OpenScop format [5] allows developers to build flows that originate in MLIR, use existing polyhedral tools, and return to MLIR for further transformation and executable code generation. Second, we create a `Clang`-based C and C++ MLIR frontend capable of identifying static control parts of the program (SCoP) to produce the Affine dialect along with other "standard" MLIR dialects. This enables MLIR flows to compile common polyhedral benchmarks as well as other code bases written in C or C++ without rewriting them in a different input language.

Bringing both standard tools and benchmarks to MLIR provides researchers with several benefits. Notably, this enables ablation analysis of the benefits of a given MLIR-based polyhedral transformation and the use of existing polyhedral tools to optimize MLIR inputs (rather than C or C++

IMPACT 2021, January 20, 2021, Budapest, Hungary

inputs). While this naturally opens up the door for creating and evaluating new transformations, in this work, we *only* consider the effectiveness of our Polygeist workflow. More specifically, we demonstrate that Polygeist is able to process the benchmarks of interest without significant overhead, and that MLIR transformations compose with existing polyhedral flows.

2 Background: MLIR Framework

2.1 Overview

MLIR is an optimizing compiler infrastructure inspired by LLVM [21] with focus on extensibility and modularity [22]. It is based on the principles of concept parsimony (the IR has as little built-in concepts as possible), effect traceability (the provenance of any IR construct can be traced back to some location in the input program) and transformation progressivity (optimizations are performed in incremental, verifiable steps that maintain the validity of the IR).

Practically, MLIR defines an SSA-based [8] IR and provides algorithms and tools to analyze and transform it. Like many SSA representations, MLIR uses *values* as a unit of data processed by the represented program. MLIR values cannot be redefined. The actions of a program are described using *operations*, which can be seen as a generalization of (machine) instructions or high-abstraction operations such as matmul, that use values and define new values. Operations are the main mechanism for defining the semantics of the program. Values in MLIR have a *type*, which contains the information known about the value at compile time. Similarly, operation *attributes* contain the additional information about the operation known at compile time. Operations are organized into linear sequences of (basic) *blocks* that are executed sequentially. Blocks may accept values as arguments, following the functional SSA form [1] as opposed to the ϕ-node form. Groups of blocks are in turn collected into *regions*. MLIR supports regions with classical control flow graph (CFG) structure where the control can flow from one block to one of its *successors* as well as arbitrary graph regions that can have custom semantics such as TensorFlow graphs. We only consider CFG regions throughout this paper. Regions can be attached to an operation, which defines how the control flows into and from these regions, allowing the IR to be arbitrarily nested at multiple levels. MLIR supports polyhedral analysis by providing attributes for *integer sets* and *affine maps*, described in more detail in Section 2.2. The generic syntax, accepted by all operations, is illustrated in Figure 1. Additionally, MLIR allows attributes, operations, and types to define custom syntax.

The key power of MLIR resides in its extensibility: its only built-in concepts are attributes, types, operations and regions described above. For example, modules and functions *need not* be a first-class concept in MLIR and are instead defined as operations with specific semantics, "symbol name" attributes, and a region to represent their body. In the same spirit, MLIR allows one to define operations ranging from high-level constructs such as "for" loops or functional "map/reduce" to low-level hardware instructions, and even hardware itself [9].

```
%result = "dialect.operation"(%operand, %operand)
          {attribute = #dialect<"value">} ({
^basic_block(%block_argument: !dialect.type):
  "another.operation"() : () -> ()
}) : (!dialect.type) -> !dialect.result_type
```

Figure 1. Generic MLIR syntax for an operation with two operands, one result, one attribute and a single-block region.

Attributes, operations and types that are expected to be used together are organized in *dialects*, which can be thought of as modular dynamic libraries. MLIR provides a handful of dialects that define common operations such as modules, functions, loops, memory or arithmetic instructions as well as ubiquitous types such as integers, floats, and tuples.

2.2 Affine Dialect

The *Affine* dialect is one of the first dialects created in the MLIR project [11]. It is intended for representing SCoP's with explicit polyhedral-friendly loop and conditional constructs. The core of its representation is the following classification of value categories:

- *Symbols*—integer values that are known to be loop-invariant but unknown at compile time, also referred to as program *parameters* in polyhedral literature, typically array dimensions or function arguments. In MLIR, symbols are values defined in the top-level region of an operation with "affine scope" semantics, e.g. functions; or array dimensions, constants and results of affine map application regardless of their definition point.
- *Dimensions*—are an extension of symbols that also accepts induction variables of affine loops.
- *Non-affine*—any other values.

Symbols and dimensions have `index` type, which is a platform-specific integer that fits a pointer (i.e., `intptr_t` in C).

Affine maps are multi-dimensional (quasi-)linear functions of a list of dimension and symbol arguments. For example, $(d_0, d_1, d_2, s_0) \rightarrow (d_0 + d_1, s_0 \cdot d_2)$ is a two-dimensional quasi-affine map from three dimensions and one symbol. The same map in MLIR syntax is `affine_map<(d0, d1, d2)[s0] -> (d0 + d1, s0 * d2)>`. The affine map construct does *not* require its arguments to have the symbol and dimension category, only *some* operations in the affine dialect do. Instead, the separation between dimensions and symbols allows for quasi-linear expressions: symbols are treated as constants and can therefore be multiplied with dimensions whereas a product of two dimensions is invalid.

Integer sets are collections of integer tuples that are constrained by a conjunction of (quasi-)linear expressions. For

```
%c0 = constant 0 : index
%0 = dim %A, %c0 : memref<?xf32>
%1 = dim %B, %c0 : memref<?xf32>
affine.for %i = 0 to affine_map<()[s0] -> (s0)>()[%0] {
  affine.for %j = 0 to affine_map<()[s0] -> (s0)>()[%1] {
    %2 = affine.load %A[%i] : memref<?xf32>
    %3 = affine.load %B[%j] : memref<?xf32>
    %4 = mulf %2, %3 : f32
    %5 = affine.load %C[%i + %j] : memref<?xf32>
    %6 = addf %4, %5 : f32
    affine.store %6, %C[%i + %j] : memref<?xf32>
  }
}
```

Figure 2. Polynomial multiplication in MLIR using Affine and Standard dialects.

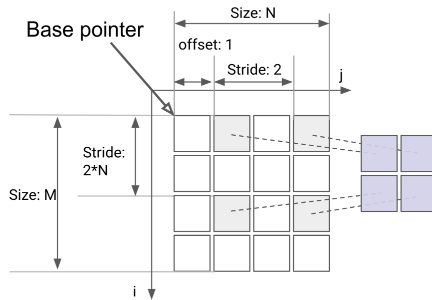

Figure 3. MLIR supports multi-dimensional memory references indexed with affine maps.

example, a "triangular" set $\{(d_0, d_1) : 0 \le d_0 < s_0 \wedge 0 \le d_1 \le d_0\}$ can be expressed as `affine_set<(d0, d1)[s0] : (d0 >= 0, s0 - d0 - 1 >= 0, d1 >= 0, d0 - d1 >= 0)>`.

The Affine dialect makes use of the concepts above to define a set of operations. An `affine.for` is a "for" loop with lower and upper bounds expressed as affine maps of symbol and dimension values, and a constant step. The bounds are computed when the loop is about to be executed and are loop-invariant. If the affine maps are multidimensional, a `max` (`min`) of the results defines the lower (upper) bound. The region is a single block that corresponds to the body of the loop and takes the induction variable as loop argument. An `affine.parallel` is a "multifor" loop nest, iterations of which can be executed concurrently. An `affine.if` is a conditional construct, with an optional `else` region, and a condition defined as inclusion of the given dimension and symbol values into an integer set. Finally, `affine.load` and `affine.store` are used to express memory accesses where the address computation is expressed as an affine map of dimensions and symbols.

Figure 2 illustrates the Affine dialect by using it to define a polynomial multiplication, `C[i+j] += A[i] * B[j]`. Operations not prefixed with `affine.` are defined in the Standard dialect and correspond to common instructions. Even such a simple example highlights the fact that MLIR supports, and encourages, IRs from different dialects to be used together.

2.3 Memory References

Figure 2 also makes use of a core MLIR type—`memref`, which stands for **`mem`**ory **`ref`**erence. It is a structured multi-index pointer into memory that does not allow internal aliasing, i.e., different indices always point to different addresses. This effectively defines away the delinearization problem that hinders the application of polyhedral techniques at the LLVM IR level [17]. By default, `memrefs` are expected to have a *strided* format similar to the one used for tensors in machine learning framework. A strided `memref` is described by its *rank*, *offset* from the base pointer, a list of *sizes* and a list of *strides*. The latter indicates the number of elements one needs to skip to obtain the next element along a dimension. Strides can thus express various layouts. For example, an $M \times N$ `memref` with strides $N, 1$ is row-major, with strides $1, M$ is column-major, while the strides $2N$ and 2 combined with offset = 1 define a layout accessing odd columns in even lines as illustrated in Figure 3. In general, the list of indices is transformed into the linear address as $A = \text{base}+\text{offset}+\sum_i \text{stride}_i \cdot \text{index}_i$. One can observe that, for a fixed rank, this can be expressed using an affine map by treating offset and strides as symbols: `affine_map<(i0,i1)[A,off,s0,s1] -> (A + off + s0*i0 + s1*i1)>`. This is intentional and makes `memrefs` compatible with affine transformations.

2.4 Other Relevant Core Dialects

MLIR provides several dozen dialects. Out of those, only a handful are relevant for our discussion. The *Structured Control Flow* (`scf`) dialect defines the control flow operations such as loops and conditionals that are not constrained by affine categorization rules. For example, a `scf.for` loop accepts any integer value as lower bound, upper bound or step and does not support any affine maps. The *Standard* (`std`) dialect contains common operations such as integer and float arithmetic, (non-affine) memory accesses or branching control flow. It is used as a common lowering point from higher-level dialects before fanning out into multiple target dialects and can be seen as an extreme generalization of LLVM IR [21]. At the same time, the *LLVM* dialect provides a direct mapping of LLVM IR instructions and types into MLIR. It is primarily used to simplify the translation process between the two representations if further processing by LLVM is necessary. Finally, the *OpenMP* dialect provides a dialect- and platform-agnostic representation of OpenMP directives such as "parallel" and "workshare loop". It can be used to emit LLVM IR interacting with an OpenMP runtime.

3 An (Affine) C or C++ Frontend for MLIR

Figure 4 shows an overview of Polygeist. Starting from a code fragment expressed using C or C++, Polygeist traverses the Clang AST and for each visited node emits the corresponding MLIR SCF or Standard dialect construct. In contrast to LLVM-based tools like Polly [16], Polygeist takes advantage

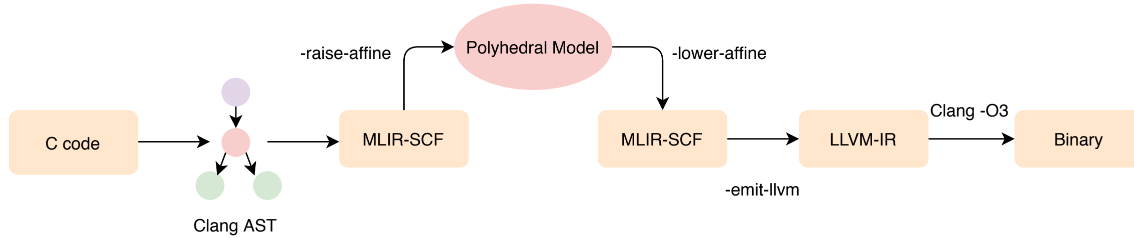

Figure 4. Polygeist flow. MLIR's SCF constructs are generated by traversing the Clang AST after the emission Polygeist raising passes enables emission of affine constructs.

of MLIR's ability to express control flow constructs such as loops directly, eliminating the need to discover loops in CFG.[1] At SCF level, Polygeist exposes a raising pass which allows lifting Standard load, store, as well as SCF "for" loops and "if" conditions to the Affine dialect. At the Affine level, code is optimized and lowered back to SCF, which in turn, gets lowered to LLVM IR for code generation. Finally, `Clang` takes LLVM IR and emits binary code, to be executed on the target platform.

3.1 Converting `Clang` AST to MLIR

Polygeist leverages Clang infrastructure to perform syntactic and semantic analysis of the input code. Provided with C or C++ files and the name of the entry function, Polygeist lazily emits IR for each function transitively called from the entry point. Polygeist achieves this by traversing the AST and converting each node ("if", "for", a binary operator, etc.) to an equivalent construct in MLIR's SCF or Standard dialect. This approach enables handling multi-versioned functions and allows users to, e.g., only produce IR for specific functions.

C or C++ types in the AST are first lowered to LLVM, as is done in Clang's normal compilation process, and then converted to an equivalent MLIR type in the Standard dialect (see Table 1). Doing so allows Polygeist to generate code with a compatible Application Binary Interface (ABI) as existing compilers without re-implementing significant pieces of infrastructure. Standard library calls such as `pow` are emitted as operations of the Standard dialect or declared as external functions.

3.2 Memory References

Most of MLIR's high-level constructs and transformations involve operations on `memref` (see Section 2.3), that contain standard integer or float types. C or C++ has no language construct that represents the equivalent of a `memref` (structured tensor object), nor does MLIR have a pointer type usable in high-level constructs. To best fit these incompatible abstractions we extend MLIR to permit a `memref` to contain other `memrefs`, and use 1-dimensional `memrefs` of unknown size to represent pointers. This allows allocations of values to be represented by a `memref` to the corresponding type, which can be then optimized by MLIR.

[1]Polly is still useful to discover loop constructs in code that was not originally written as explicit C or C++ loops.

C type	LLVM IR type	MLIR type
`int`	`i32 (on machine X)`	`i32 (on machine X)`
`intNN_t`	`iNN`	`iNN`
`uintNN_t`	`iNN`	`uiNN`
`float`	`float`	`f32`
`double`	`double`	`f64`
`ty *`	`ty *`	memref<? x ty>
`ty **`	`ty **`	memref<memref<? x ty>>
`ty[N][M]`	`ty[N][M]`	memref<N x M x ty>

Table 1. Type correspondence between C, LLVM IR and MLIR Standard types.

Extra care needs to be taken in the emission of `memref` operations to successfully represent all of the desired behavior of pointers. As a consequence we create an internal state within the MLIR generation process (`ValueWithOffsets`). This state contains a value representing a `memref` and a current list of indices looking into that `memref`. This allows pointer operations to index into a `memref` without necessarily creating a load or store, thus generating code that better represents the intent of the program.

Finally, for code that uses or creates a `memref` (such as in allocation functions), simply allocating a number of bytes of an array with `malloc` then casting to a `memref` will not result in legal code (as `memref`'s underlying implementation or ABI may not be a raw pointer). As a consequence, Polygeist replaces calls to allocation and dellocation functions with legal equivalents for `memref`. With rare exceptions, other functions that previously had a pointer argument are now declared to have a `memref` argument. So as long as all code with such an argument is generated by Polygeist, the ABI remains consistent and the calls legal. But, there exist certain functions (such as `main` or `strcmp`) for which it is not desirable to modify the ABI to accept a `memref`. These functions will have pointers from the LLVM dialect as arguments with their uses modified with an appropriate conversion (or emit a compile-time error if not possible). Figure 5 shows an example demonstrating Polygeist ABI.

```
void setArray(int N, double val, double* array) {...}
int main(int argc, char** argv) {
  ...
  cmp = strcmp(str1, str2)
  ...
  double array[10];
  set_array(10, array)
}

func @setArray(%N: i32, %val: f64
               %array: memref<?xf64>) {
  %0 = index_cast %N : i32 to index
  affine.for %i = 0 to %0 {
    affine.store %val, %array[%i] : memref<?xf64>
  }
  return
}

func @main(%argc: i32,
           %argv: !llvm.ptr<ptr<i8>>) -> i32 {
  ...
  %cmp = llvm.call @strcmp(%str1, %str2) :
           (!llvm.ptr<i8>, !llvm.ptr<i8>) -> !llvm.i32
  ...
  %array = alloca() : memref<10xf64>
  %arraycst = memref_cast %array : memref<10xf64> to
    memref<?xf64>
  call @setArray(%N, %val, %arraycst) :
           (i32, f64, memref<?xf64>) -> ()
}
```

Figure 5. Example demonstrating Polygeist ABI. For functions expected to be compiled with Polygeist such as `setArray`, pointer arguments are replaced with `memref`'s. For functions that require external calling conventions (such as `main`/`strcmp`), we fall back to using `llvm.ptr` and generating conversion code where appropriate.

3.3 Local Variables

Local variables are handled by allocating a `memref` at the top of a function, and loading or storing to said memref when it is used. This permits the desired semantics of C or C++ to be implemented with relative ease. However, as many local variables and arguments contain `memref` types, this immediately results in a `memref` of a `memref` — a hindrance for most MLIR optimizations as it is illegal outside of our MLIR patch to have a `memref` of a `memref`.

As a remedy, we implement a heavyweight memory-to-register (mem2reg) transformation pass that eliminates unnecessary loads, stores and allocations within MLIR constructs. Empirically this eliminates all `memref`s of `memref` in the Polybench suite.

3.4 Generating Affine Code

When within a SCoP — defined as the code between `#pragma scop` and `#pragma endscop` — Polygeist will explicitly emit an `affine.for` for loops rather than `scf.for`. Bounds for the loop are set to the value of the corresponding expression, relying on the identity affine map (`affine_map<()[s0]->(s0)>[%bound]`) for both. The bound arguments are not necessarily *symbols* as per affine categorization (see Section 2.2), they can be, e.g., results of integer arithmetic operations. However, the expressions that produce these values are guaranteed to affine by the semantics of `#pragma scop` and can be raised to affine expressions as described below. The Affine dialect does not support loops with negative steps, so Polygeist rewrites such loops to have a positive step.

Polygeist makes the IR valid by running an "affine fixup" pass that folds standard scalar operations (add, sub, mul) that produce the loop bounds into the affine maps present in the loops. For example, `affine_map<()[s0]->(s0)>[%bound]` with `%bound = addi %N, %i` is folded into `affine_map<()[s0, s1]->(s0 + s1)>[%N, %i]`. Polygeist also promotes any "*symbol*" representing an induction variable to its proper description as a *parameter* (becoming `affine_map<(d0)[s0]->(s0 + d0)>(%i)[%N]`). Since the original bound expression is guaranteed to be affine, this canonicalization process will fold all operations into the affine map until all symbols and parameters are valid.

If statements. We introduce an additional `scf.if` transformation that ensures all "if" statements are transformed into their affine counterparts. This is done by descending into the condition and ensuring it is composed of and, add, sub, and mul operations on legal affine arguments. Conjunctions are then separated into separate conditional expressions, which are then converted to their equivalent "canonical affine comparison" (being either equal to zero or greater than or equal to zero).

It is legal to have a short-circuiting boolean operator within `#pragma scop`. This will result in an `scf.if` with the first expression of the condition and the remaining expressions evaluated in the body of `scf.if`. The result of this `scf.if` representing the boolean operator can then be used as the conditional for a C or C++ "if" statement. A value generated from an `scf.if` is certainly non-affine, preventing the transformation of the C or C++ if statement into an `affine.if`. To remedy this we introduce an optimization that identifies the legality and utility of moving the instructions within an `scf.if` outside, replacing the "if" with either a boolean operation or a select. Upon simplification, this will result in a valid affine condition for all short-circuiting operations within a `#pragma scop`.

Not all `scf.if`s generated within an `affine.for` are transformed into an `affine.if`. This is because polyhedral programs (at least in Polybench) use C ternary operators within `#pragma scop`. The default semantics of C ternaries is to lazily evaluate the true and false operands only if required by the condition. This may require special handling by polyhedral tools (Section 4.1). To ease the burden on polyhedral tools, we create and run an additional mem2reg pass that replaces loads to equivalent earlier loads when possible. For the operations inside Polybench, this is sufficient to remove remaining `scf.if`s, but is not necessarily sufficient in

general. Finally, we also introduce simplifications that fold `affine.apply` into `affine.if`.

Other operations. After the "affine fixup" pass and the mem2reg described, Polygeist runs a transformation pass that will attempt to raise all `load`s, `store`s, and `if`s to their affine counterparts within affine loops. This is done by checking if a construct is within an affine loop and if its arguments can be transformed by the "affine fixup" procedure to satisfy the categorization requirements.

4 Connecting MLIR to Polyhedral Tools

The compilation flow described above allows one to obtain a representation of the input as MLIR Affine dialect, which is suitable for transformation *within* MLIR, but cannot be consumed by existing *external* polyhedral tools such as Pluto. To establish the connection, Polygeist transforms MLIR Affine operations into an existing polyhedral exchange format, runs the polyhedral tools, and regenerates MLIR. We choose OpenScop [5] as the main export format since it (or its predecessor — ScopLib) is supported by a variety of existing polyhedral tools, namely CLooG [3], Pluto [7] and isl [36]. A major challenge of this process stems from OpenScop's design being oriented towards C or Fortran statements, which does not match exactly the structure of MLIR. Therefore, we propose a mechanism for deriving polyhedral "statements" from MLIR that are usable in external tools, and suitable for MLIR code generation after polyhedral transformation.

4.1 Statement Formation

Polyhedral tools expect statements to have read and/or write specific memory accesses, be enclosed within affine loops, and be "instantiated" by loop induction variables. A polyhedral statement is noted as `S0(i0,i1)`, in which the statement `S0` is instantiated at loop induction variables `i0` and `i1`. Each statement has a body that represents exactly one statement in a C-like language. For example, the expression `C[i][j] += A[i][k] * B[k][j]` would be a valid body for the statement `S1(i,j,k)`. This evidences the representational gap between MLIR and polyhedral tools: MLIR operations — the closest construct to a polyhedral statement — can only define values but not update them. Therefore a polyhedral statement should be formed from several MLIR operations. Our objective is to find a mechanism for aggregating MLIR operations into statements, which can precisely capture the behavior of the original program, be friendly to polyhedral transformation, and permit regeneration into MLIR.

To match C-like statement structures, which typically write into a single memory address, we create one statement per `affine.store` operation. We then traverse the SSA use-def chains upwards and aggregate the operations we visit into the statement, until an `affine.load`, loop induction variable, or affine symbol is reached. This allows our

```
func @S1(%i: index, %j: index, %alpha: f32,
         %C: memref<?x?xf32>) {
  %0 = affine.load %C [ %i, %j ] : memref<?x?xf32>
  %1 = mulf %0, %alpha : f32
  affine.store %1, %C[%i, %j] : memref<?x?xf32>
  return
}
func @S2(%i: index, %j: index, %k: index, %beta: f32,
         %A: memref<?x?xf32>, %B: memref<?x?xf32>,
         %C: memref<?x?xf32>) {
  %0 = affine.load %C[%i, %j] : memref<?x?xf32>
  %1 = affine.load %A[%i, %k] : memref<?x?xf32>
  %2 = affine.load %B[%k, %j] : memref<?x?xf32>
  %3 = mulf %1, %2 : f32
  %4 = mulf %beta, %3 : f32
  %5 = addf %0, %4 : f32
  affine.store %C[%i, %j] : memref<?x?xf32>
  return
}
func @gemm(%alpha: f32, %beta: f32, %A: memref<?x?xf32>,
           %B: memref<?x?xf32>, %C: memref<?x?xf32>) {
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %0 = dim %A, %c0 : memref<?x?xf32>
  %1 = dim %B, %c1 : memref<?x?xf32>
  %2 = dim %A, %c1 : memref<?x?xf32>
  affine.for %i = 0 to %0 {
    affine.for %j = 0 to %1 {
      call @S1(%i, %j, %alpha, %C)
      affine.for %k = 0 to %2 {
        call @S2(%i, %j, %k, %beta, %A, %B, %C)
      }
    }
  }
  return
}
```

Figure 6. GEMM kernel in MLIR after outlining that makes polyhedral statement visible as functions.

statements to resemble those obtained from C input, close to what the existing tools usually process.

Some operations may end up in multiple statements if the value is reused. For side effect-free operations, it is safe to just have a copy in each statement. Polygeist performs additional analysis for values produced by operations with side effects. In particular, if a value produced by an `affine.load` is still used after the memory location it was loaded from is overwritten, it is illegal to copy the `load`. In such cases, Polygeist immediately stores the value into a stack-allocated dedicated scratchpad `memref` and loads it from there instead.

In many cases, a statement may consist of MLIR operations across different (nested) loops, e.g., a load from memory into an SSA register happens in an outer loop while it is used in inner loops. The location of such a statement in the loop hierarchy is unclear. More importantly, it may not be possible to generate it back after the polyhedral scheduler reconstructs the loop hierarchy entirely since the scheduler is not aware of a statement potentially spanning multiple loops. We address this **region-spanning problem** by implementing a register-to-memory (`reg2mem`) pass that detects any def-use

pair crossing a region boundary, such as loops and/or conditionals, and uses stack-allocated single-element scratchpad `memrefs` to hold the value. The allocation operation will be immediately followed by a store of the desired value, ensuring that all uses of the scratchpad are valid. `reg2mem` effectively creates a new polyhedral statement in the outer loop and makes all use-def chains local to a region, e.g., a loop body. Thus all statements are given a definite position in the loop hierarchy, and may be connected through data dependencies produced by the respective `affine.load/store`. In addition to the basic `reg2mem` algorithm, we perform a simplified value analysis to conservatively avoid creating scratchpad for values that will be stored to an existing memory buffer, which can be loaded from in the any region that needs the value. This helps decrease the number of dependencies in the input as well as the memory footprint.

There is a trade-off between sinking load operations into inner loops by applying the inverse of Loop-Invariant Code Motion (LICM) in standard MLIR transformations, reducing the frequency of `mem2reg` in the Polygeist frontend, and using `reg2mem` in the Polygeist polyhedral flow. We argue that it is necessary to do `reg2mem` systematically, mainly because `affine.loads` cannot always be sunk into the inner loop: potentially, there can be `affine.stores` following them and write to the same address, and sinking them will violate the write-after-read dependencies. And not doing `mem2reg` in the frontend will produce write-once variables that can consequently complicate polyhedral analysis.

4.2 SCoP Formation

To define a SCoP, we outline individual statements into functions so that they can be represented as opaque calls with known memory footprints, similarly to Pencil [2]. This process also makes the inter-statement SSA dependencies clear. These dependencies exists between calls that *use* the same SSA value since all use-def chains (except for induction variables that are processed separately) had been encapsulated into statements. We also lift all local stack allocations and place them at the entry block of the surrounding function in order to keep them visible after loop restructuring.

The remaining components of the polyhedral representation are derived as follows. The domain of the statement is defined to be the iteration space of its enclosing loops, constrained by their respective lower and upper bounds, and intersected with any “if” conditions. This process leverages the fact that MLIR expresses bounds and conditions directly as affine constructs. The access relations for each statement are obtained as unions of affine maps of the `affine.load` (read) and `affine.store` (must-write) operations, with RHS of the relation annotated by an “array” that corresponds to the SSA value of the accessed `memref`. Initial schedules are assigned using the $(2d+1)$ formalism, with odd dimensions representing the lexical order of loops in the input program and even dimensions being equal to loop induction variables.

CPU	Clock rate	OS	RAM (GB)	L1/L2/L3 (MB)
Intel Xeon Platinum 8275CL	3.0 GHz	Ubuntu 20.04	189	1.5, 48, 71.5

Table 2. Hardware setup.

This format is required by the OpenScop specification, and we only use it when exporting MLIR to OpenScop. Affine constructs in OpenScop are represented as lists of affine function coefficients interpreted as either equalities ($= 0$) or inequalities (≥ 0). Internally, MLIR affine constructs use a similar representation so the bi-directional conversion is straightforward.

4.3 Code Generation Back to MLIR

The OpenScop representation can be directly consumed by various tools, including the Pluto optimizer [7], producing a new OpenScop program as a result. Converting this representation back to a form with loops and conditionals is a challenging problem, so we rely on CLooG [3] to generate the initial loop-level AST. Polygeist then traverses the AST and creates the affine constructs that correspond to loops and conditionals. The conversion process is simplified mainly by MLIR using affine expressions directly in loop constructs, so only the general control flow structure needs to be generated.

Individual statements are introduced back as calls to the previously outlined functions that correspond to statements, sparing the need to clone SSA subgraphs at this point. These calls will be later inlined by Polygeist. We use an in-memory symbol table of MLIR values in the original code, which will be alive before and after Pluto optimization. All the symbols appeared in the OpenScop representation, and its CLooG AST will be mapped to an MLIR value.

5 Evaluation

Our goal is twofold: First, we want to demonstrate that the code generated by Polygeist is on-par with the code generated by a state-of-the-art compiler like Clang (Section 5.2). Second, we wish to demonstrate the feasibility of utilizing existing polyhedral flows, especially research compilers, to process MLIR (Section 5.3). We are *not* interested in using MLIR to produce better-optimized code than polyhedral flows.

5.1 Experimental Setup

We ran our experiments on an AWS `c5.metal` instance with hyper-threading and Turbo Boost disabled (see Table 2). We ran all 30 benchmarks from PolyBench [30], using the “EXTRALARGE” dataset. For each benchmark, we ran a total of 5 trials, taking the geometric mean and standard deviation of the execution time reported by PolyBench. Every measurement or result reported in the following sections refers to double-precision data processed in a single thread.

Benchmark	Clang	±	σ	Polygeist	±	σ	%-diff
2mm	63.191	±	0.139	62.117	±	0.169	1.73%
3mm	106.955	±	0.261	104.705	±	0.087	2.15%
adi	111.024	±	0.215	121.765	±	0.203	-8.82%
atax	0.007	±	<0.001	0.007	±	<0.001	EXCL
bicg	0.012	±	<0.001	0.006	±	<0.001	EXCL
cholesky	15.591	±	0.017	15.578	±	0.007	0.09%
correlation	95.262	±	0.020	94.764	±	0.065	0.52%
covariance	95.283	±	0.016	94.769	±	0.068	0.54%
deriche	1.686	±	0.003	1.669	±	0.001	1.03%
doitgen	5.123	±	0.012	4.920	±	0.005	4.12%
durbin	0.017	±	0.005	0.015	±	<0.001	EXCL
fdtd-2d	25.803	±	0.039	25.879	±	0.060	-0.30%
floyd-wars.	146.009	±	0.061	146.015	±	0.052	0.00%
gemm	8.467	±	0.013	8.626	±	0.011	-1.83%
gemver	0.160	±	<0.001	0.158	±	<0.001	1.27%
gesummv	0.024	±	<0.001	0.014	±	<0.001	EXCL
gramschmidt	152.444	±	0.086	152.254	±	0.067	-0.12%
heat-3d	33.022	±	0.127	32.964	±	0.028	0.17%
jacobi-1d	0.005	±	0.002	0.006	±	0.002	EXCL
jacobi-2d	24.149	±	0.050	24.952	±	0.019	-3.22%
lu	101.382	±	0.386	101.495	±	0.388	-0.11%
ludcmp	99.538	±	0.546	99.155	±	0.671	0.39%
mvt	0.146	±	<0.001	0.144	±	<0.001	1.27%
nussinov	133.654	±	0.288	133.811	±	0.094	-0.12%
seidel-2d	202.318	±	0.015	202.289	±	0.001	0.01%
symm	55.253	±	0.071	54.214	±	0.030	1.92%
syr2k	70.523	±	0.200	70.359	±	0.053	0.23%
syrk	25.982	±	0.265	25.993	±	0.177	-0.04%
trisolv	0.012	±	<0.001	0.012	±	<0.001	EXCL
trmm	47.946	±	0.211	47.941	±	0.369	0.01%

Table 3. Geometric mean and standard deviation execution time of programs produced by Polygeist and Clang on Polybench EXTRALARGE double-precision single-thread. The rightmost column shows percent difference between Clang and Polygeist runtimes, with EXCL showing where a benchmark ran in below 0.05s.

Benchmark	Pluto	±	σ	Polygeist	±	σ	%-diff
2mm	4.471	±	0.017	4.258	±	0.018	4.765%
3mm	8.757	±	0.026	7.731	±	0.003	11.724%
adi	Pluto fails to compile						
atax	0.011	±	0.002	0.011	±	0.001	EXCL
bicg	0.010	±	0.001	0.006	±	<0.001	EXCL
cholesky	10.775	±	0.056	11.285	±	0.097	-4.731%
correlation	4.153	±	0.019	4.228	±	0.003	-1.822%
covariance	4.111	±	0.018	4.253	±	0.004	-3.452%
deriche	1.771	±	0.001	1.762	±	0.001	0.571%
doitgen	1.869	±	0.021	1.417	±	0.010	24.192%
durbin	0.017	±	0.005	0.015	±	<0.001	EXCL
fdtd-2d	21.717	±	0.205	15.930	±	0.073	26.627%
floyd-w.	380.402	±	0.694	345.460	±	0.569	9.186%
gemm	4.591	±	0.036	5.110	±	0.006	-11.306%
gemver	0.099	±	0.001	0.097	±	<0.001	2.356%
gesummv	0.035	±	0.001	0.014	±	<0.001	EXCL
gramschmid	14.647	±	0.172	14.730	±	0.171	-0.569%
heat-3d	28.723	±	0.037	29.752	±	0.033	-3.581%
jacobi-1d	0.008	±	0.003	0.010	±	0.002	EXCL
jacobi-2d	17.322	±	0.077	22.616	±	0.217	-30.561%
lu	10.667	±	0.034	10.274	±	0.049	3.689%
ludcmp	98.916	±	0.249	98.803	±	0.716	0.114%
mvt	0.084	±	0.001	0.083	±	<0.001	0.196%
nussinov	124.424	±	0.122	124.062	±	0.147	0.291%
seidel-2d	237.186	±	0.028	164.344	±	0.003	30.711%
symm	53.952	±	0.037	53.921	±	0.086	0.058%
syr2k	9.946	±	0.008	10.006	±	0.008	-0.605%
syrk	5.374	±	0.005	5.328	±	0.003	0.855%
trisolv	0.024	±	<0.001	0.024	±	<0.001	EXCL
trmm	2.079	±	0.025	2.215	±	0.001	-6.550%

Table 4. Geometric mean and standard deviation execution time of programs produced by Polygeist and Pluto on Polybench EXTRALARGE double-precision single-thread. The rightmost column shows the percent difference between Polygeist and Pluto runtimes with same exclusion rule as Table 3. Pluto cannot compile the `adi` benchmark.

5.2 Frontend

Polygeist intends to provide a fair comparison baseline and therefore should produce code with runtime *as close as possible* to that of existing compilation flows. In other words, Polygeist should *not introduce overhead nor speedup* unless explicitly instructed otherwise. We evaluate this by comparing the runtime of programs produced by Polygeist with those produced by Clang at the same commit (Dec 2020)[2]. We run Polygeist flow to produce LLVM IR, which is then compiled to a binary using Clang with `-O3`. We also run Clang with the same flags on the input C code to produce baseline binaries. Table 3 refers to the former as Polygeist and to the latter as Clang.

To evaluate the similarity of Polygeist and Clang, we compute the percent difference of all benchmarks with a runtime greater than 0.05. The mean absolute-value percent difference is only 1.25%, indicating that Polygeist indeed closely matches the performance of Clang.

5.3 External Polyhedral Flow

To evaluate the polyhedral flow in Polygeist, we compare the Polybench MLIR code it produces with the Polybench C programs directly optimized by the Pluto command-line program, namely `polycc`[3]. We ensure that internally Polygeist uses the same configurations as the default for `polycc`, which in general applies polyhedral loop transformations including tiling, fusion, interchanging, etc. Specifically, the Pluto transformation function we use is `pluto_auto_transform`. Here, since we are more interested in single-core performance, we turn off the parallel and vectorization. The polyhedral-optimized MLIR code will be emitted to LLVM IR, and then compiled by Clang using `-O3`. Similarly, we first compile the

[2]LLVM commit `f019362329734ddc7d17fc76bcb7f2a4b3ea50a7`.

[3]Pluto commit `5b13ddccdaa2c125657e9333668fcedab9487271`

polycc optimized C programs by Clang -O3 into LLVM IR, which is further compiled into an executable using Clang -O3 as well. Table 4 summarizes the results. By taking the mean absolute-value of the performance differences of all the benchmarks that have greater than 0.05 sec runtime, we find Polygeist has 7.76% percentage different in runtime compared with Pluto. We will discuss the performance difference in Section 6.2 We do not compare Polygeist with Polly for now since Polly uses a modified version of the Pluto algorithm, making it difficult for an apple-to-apple comparison between them.

6 Discussion

6.1 Benchmarking

The only benchmark with a nontrivial performance difference between Polygeist and Clang is the adi test. This gap exists for two reasons: differences in allocation, and loop reversal. Specifically, compiling Polybench with Clang results in the use of a custom allocator, whereas using Polygeist, this results in a memref allocation which is lowered to a malloc. This difference in allocation function accounts for 48% of the gap. The remaining gap exists because LLVM can strength reduce a specific load for the IR generated by Clang but not MLIR. LLVM cannot recognize this property in a reversed loop and consequently cannot perform the optimization. A future version of LLVM should permit this optimization.

Throughout benchmarking, we also found various behaviors of note. Polygeist supports the ability to compile two source files directly by producing a single MLIR module with all of the necessary functions. This is distinct from Clang, which will produce two LLVM modules that are eventually linked together. For our current benchmarks we strive to emulate the behavior of Clang by compiling the test file with Polygeist (e.g., nussinov.c) and linking it with timing utility code (polybench.c) using Clang. An earlier version of our pipeline, however, compiled both the timing utility code and benchmark together with Polygeist directly. For almost all benchmarks, this did not make a difference. But for the floyd-warshall test, we saw an 8% reduction in performance by using a single module. An investigation into this found that interprocedural constant propagation between the utility and benchmark code allowed for significant optimization of floyd-warshall.

Another crucial component of generating accurate code was ensuring that the code emitted by Polygeist had the same LLVM DataLayout and Target as that emitted by Clang natively. Polygeist directly parses these and marks the MLIR appropriately when it lowers Clang AST. This is not sufficient, however, as MLIR currently will not propagate the DataLayout to the eventual LLVM. This is problematic as it will result in LLVM not performing vectorization in the same way. We modified MLIR to ensure this information is propagated successfully throughout all stages of the pipeline.

6.2 Performance Differences in Transformed Code

Three factors contribute to the performance gaps in Table 4. As discussed in Section 6.1, code emitted by Polygeist and Clang use different allocation functions. This can result in up to 40% difference, largely stemming from Clang being able to assume malloc results do not alias and propagate that. We fix this by also using malloc in Polybench in Table 4. Even with the same schedule, differences in code generation may lead to performance differences. More specifically, the exact shape of domain relations fed to CLooG (e.g., the inclusion of constraints on parameters into the *domain*) significantly changes the final AST. This can be addressed through a finer-grain control over the AST generation process [18]. Furthermore, MLIR's index type converts to the proper machine index type, which, in conjunction with automatic simplification of affine forms in MLIR, enables a more aggressive bound analysis in the downstream compiler.

Consider, for example, seidel-2d. Polygeist produces 30% faster code. Analyzing the execution with perf, we observe that both Pluto and Polygeist issue $143 \cdot 10^9$ FP instructions, but Pluto issues $585 \cdot 10^9$ total instructions as opposed to $254 \cdot 10^9$ by Polygeist. These instructions are related to control flow and address computations. Assuming a mix of add (throughput 1/3) and imul (throughput 2), the extra $331 \cdot 10^9$ instructions can comfortably explain the performance difference of 73s when running at 3GHz ($219 \cdot 10^9$ extra cycles). This can be attributed to the memref representation that emits homogeneous, LLVM-friendly address computations.

6.3 Limitations

Frontend. While Polygeist could technically accept any valid C or C++ thanks to building off Clang, it has the following limitations. Only structs with values of the same type are supported due to the lack of a struct-type in high-level MLIR dialects. Loops with break or continue statements are not supported because MLIR is missing a construct to represent them. Moreover, adding frontend support for those would still not allow Polygeist to raise these constructs to Affine as they do not fit the polyhedral representation. Finally, we require all functions that allocate memory to be compiled with Polygeist and not a C++ compiler. This ensures a memref is emitted rather than a pointer.

Backend. The limitations in the affine backend are inherited from those of the tools involved. In particular, the value categorization of MLIR's Affine dialect results in all-or-nothing modeling, degrading any loop to non-affine if it contains even one non-affine access. Running Polygeist's backend on code not generated by Polygeist's frontend is limited to loops with positive indices. MLIR's Affine dialect does not support loops with negative steps. But, since Polygeist's frontend rewrites loops with negative steps to have a positive step, this is not a problem for codes using Polygeist's frontend. Finally, MLIR does not yet provide extensive support

for non-convex sets (typically expressed as unions). Work is ongoing within MLIR to address such issues. Affine scheduling in Polygeist inherits limitations of Pluto. In particular, Polygeist has no first-class support for reduction loops and does not model live-range reordering [38].

6.4 Opportunities

Connecting MLIR to existing polyhedral flows opens numerous avenues for compiler optimization research, following the original goal of MLIR to connect Affine and conventional SSA-based compiler transformations. This gives polyhedral schedulers access to important analyses such as aliasing and useful information such as precise data layout and target machine description. Arguably, this information is already leveraged by Polly, but the representational mismatch between LLVM IR and affine loops makes it difficult to exploit them efficiently. This abstraction gap requires complex analyses in the polyhedral optimizer such as scalar dependence removal [20] or array delinearization [17]. MLIR exposes similar information at a sufficiently high level to make it usable in affine transformations.

By allowing different abstractions to mix in a single module, MLIR provides finer-grain control over the entire transformation process. A polyhedral transformation flow built on MLIR can, e.g., ensure that the loop is vectorized by directly emitting the corresponding vector instructions instead of relying on pragmas, which are often merely a recommendation for the compiler. The flow can also control lower-level mechanisms like prefetching or emit specialized hardware instructions. Polyhedral analyses can guarantee downstream passes that address computation never produces out-of-bounds accesses and other information.

Finally, since Polygeist fully controls the definition of *statement*, it becomes possible to vary statement granularity in a polyhedral flow. This allows, on one hand, to have a combined polyhedral/syntactic flows that can easily introduce and control rematerialization or temporary buffers, split long statements, and organize software pipelining; all without having to produce C source which is known to be complex [41]. On the other hand, this may have an effect on the compilation time as the number of statements is an important factor in the complexity bound of the dependence analysis and scheduling algorithms.

6.5 Alternatives

Instead of allowing polyhedral tools to parse and generate MLIR, one could emit C (or C++) code from MLIR [4] and use C-based polyhedral tools on C source, but this approach decreases the expressiveness of the flow. Some MLIR constructs, such as parallel reduction loops, can be directly expressed in the polyhedral model whereas they would require a non-trivial and non-guaranteed raising step in C. Some other constructs, such as prevectorized affine memory operations, cannot be expressed in C at all. Polygeist also enables transparent handling of such constructs in MLIR-to-MLIR flows, but we leave the details of such handling for future work.

The Polygeist flow can be similarly connected to other polyhedral formats, in particular `isl`. We choose OpenScop for this work because it is supported by a wider variety of tools, including `isl` that can construct its representation of affine relations from ScopLib, a predecessor of OpenScop. `isl` also uses schedule trees [40] to represent the initial and transformed program schedule. Schedule trees are sufficiently close to the nested-operation IR model making the conversion straightforward: "for" loops correspond to band nodes (one loop per band dimension), "if" conditionals correspond to filter nodes, function-level constants can be included into the context node. The tree structure remains the same as that of MLIR regions. The inverse conversion can be obtained using `isl`'s AST generation facility [18].

[4]https://github.com/marbre/mlir-emitc

7 Related Work

Polyhedral extractors. The polyhedral model has been on the cutting edge of compiler research for several decades, resulting in the creation of many tools [15]. Polly [16] and Graphite [29] enable polyhedral optimizations in LLVM and GCC, respectively, by raising from the low-level IR to higher and richer polyhedral representations. Other proprietary compilers, such as IBM XL [6] and R-Stream [24], use polyhedral techniques and thus rely on extractor tool, but being proprietary few documentation is available. However, extracting the polyhedral model from low-level IR is not be the best approach for source-to-source optimizers such as Pluto [7] and PoCC [28], since it is difficult or even impossible to relate low-level code to input program. Source-level parsers such as Clan [4] and PET [39] aim at providing a convenient way to extract polyhedral representation directly from the source code. Polygeist falls into this category and aims to enable MLIR to leverage the decades of research in the polyhedral model by lifting C code to the Affine dialect. Besides, we sometimes need polyhedral optimization in MLIR before LLVM IR is produced to adopt MLIR-specific passes, e.g., GPU mapping, which further justifies the necessity of Polygeist even Polly exists.

MLIR Frontends. Since the adoption of MLIR under the LLVM umbrella, several frontends have been created for generating MLIR from domain-specific languages. Teckyl [13] brings Tensor Comprehensions [35], a productivity-orientated language to express computation between tensors, to MLIR's Linalg dialect. Flang — the LLVM's Fortran frontend — enables models Fortran specific constructs (i.e., dispatch table) using the FIR dialect [32]. COMET, a domain-specific compiler, for chemistry compilation enters the MLIR lowering

pipeline using a domain-specific frontend from a tensor-based language [26]. NPComp aims at providing the necessary infrastructure to compile numerical Python programs taking advantage of the MLIR infrastructure. Work is on progress to provide a PyTorch frontend [27]. PET-to-MLIR converts a subset of polyhedral C code to MLIR's Affine dialect by parsing PET's internal representation. In addition to currently not handling specific constructs (ifs, symbolic bounds, and external function calls), parsing PET's representation limits the frontend's usability as it cannot interface with non-polyhedral code such as initialization, verification, or printing routines [19]. In contrast, Polygeist generates MLIR from non-polyhedral code as well (though not necessarily in the Affine dialect). CIRCT is a new project under the LLVM umbrella that aims to apply MLIR development methodology to the electronic design automation industry [9].

8 Conclusion

We present Polygeist, a compilation workflow for importing existing C or C++ code into MLIR and allows polyhedral tools, such as Pluto, to optimize MLIR programs. This enables MLIR to benefit from decades of research in the polyhedral compilation. We demonstrate that the code generated by Polygeist has comparable performance with Clang, enabling unbiased comparisons between transformations built for MLIR and existing polyhedral frameworks. Finally, we demonstrate the utility of our tool to perform such integration by compiling the Polybench benchmark suite into MLIR and importing Pluto transformations to run on MLIR programs, which may already lead to some performance improvements over the existing flows thanks to better integration with the LLVM compiler infrastructure.

Acknowledgements

Thanks to Valentin Churavy and Charles Leiserson of MIT for thoughtful discussions about transformations within MLIR. We are also grateful for the numerous discussions with Tobias Grosser from the University of Edinburgh. As well as, with Albert Cohen of Google and Henk Corporaal at TU Eindhoven.

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323, in part by Los Alamos National Laboratories grant 531711, in part by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. Lorenzo Chelini is partially supported by the European Commission Horizon 2020 programme through the NeMeCo grant agreement, id. 676240. Ruizhe Zhao is sponsored by UKRI (award ref 2021246) and Corerain Technologies Ltd. The support of the UK EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/L00058X/1) is also gratefully acknowledged.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] Andrew W Appel. 1998. SSA is functional programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20.

[2] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. 2015. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 138–149.

[3] Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 7–16.

[4] Cédric Bastoul. 2008. Clan-a polyhedral representation extractor for high level programs.

[5] Cédric Bastoul. 2011. *Openscop: A specification and a library for data exchange in polyhedral compilation tools.* Technical Report. Paris-Sud University.

[6] U. Bondhugula, S. Dash, O. Gunluk, and L. Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 343–352.

[7] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *ACM SIGPLAN Notices* 43, 6 (2008), 101–113.

[8] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[9] CIRCT Developers. 2020. *CIRCT Charter.* https://github.com/llvm/circt/blob/master/docs/Charter.md

[10] MLIR Developers. 2020. *MLIR: A Case for a Simplified Polyhedral Form.* https://mlir.llvm.org/docs/Rationale/RationaleSimplifiedPolyhedralForm/

[11] MLIR Developers. 2020. *MLIR Affine dialect.* https://mlir.llvm.org/docs/Dialects/Affine/

[12] MLIR Developers. 2020. *MLIR Rationale.* https://mlir.llvm.org/docs/Rationale/Rationale/

[13] Andi Drebes. 2020. *Teckyl: An MLIR frontend for Tensor Operations.* https://github.com/andidr/teckyl

[14] Andi Drebes, Lorenzo Chelini, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, Kanishkan Vadivel, and Nicolas Vasilache. 2020. TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory. In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*.

[15] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. *Encyclopedia of parallel computing* 3 (2011), 1581–1591.

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

[17] Tobias Grosser, Jagannathan Ramanujam, Louis-Noel Pouchet, Ponnuswamy Sadayappan, and Sebastian Pop. 2015. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th*

ACM on International Conference on Supercomputing. 351–360.

[18] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 4 (2015), 1–50.

[19] Konrad Komisarczyk, Lorenzo Chelini, Kanishkan Vadivel, Roel Jordans, and Henk Corporaal. 2020. PET-to-MLIR: A polyhedral front-end for MLIR. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 551–556.

[20] Michael Kruse and Tobias Grosser. 2018. DeLICM: scalar dependence removal at zero memory cost. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 241–253.

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

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

[23] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra.

[24] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. *R-Stream Compiler*. Springer US, Boston, MA, 1756–1765. https://doi.org/10.1007/978-0-387-09766-4_515

[25] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 429–443.

[26] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. COMET: A Domain-Specific Compilation of High-Performance Computational Chemistry. In *The 33rd Workshop on Languages and Compilers for Parallel Computing*.

[27] npcomp developers. 2020. *MLIR npcomp*. https://github.com/llvm/mlir-npcomp

[28] PoCC. 2020. The Polyhedral Compiler Collection. https://sourceforge.net/projects/pocc/ Online; accessed on December 2020.

[29] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. 2006. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*. 2006.

[30] Louis-Noël Pouchet and Tomofumi Yuki. [n.d.]. *PolyBench/C 4.2.1*. https://sourceforge.net/projects/polybench/files/

[31] Harenome Razanajato, Vincent Loechner, and Cédric Bastoul. 2017. Splitting Polyhedra to Generate More Efficient Code. In *International Workshop on Polyhedral Compilation Techniques (IMPACT), 2017.*

[32] Eric Schweitz. 2019. An MLIR dialect for high-level optimization of Fortran. In *2019 LLVM Developers Meeting*.

[33] Verdoolaege Sven, Manjunath Kudlur, Harinath Kamepalli, and Rob Schreiber. 2020. Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques. In *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*.

[34] Nicolas Vasilache, Oleksandr Zinenko, and Albert Cohen. 2020. *Linalg Dialect Rationale: The Case For Compiler-Friendly Custom Operations*. https://mlir.llvm.org/docs/Rationale/RationaleLinalgDialect/

[35] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–26.

[36] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302.

[37] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[38] Sven Verdoolaege and Albert Cohen. 2016. Live-range reordering. In *International Workshop on Polyhedral Compilation Techniques*.

[39] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, Vol. 141.

[40] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques*.

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.