Polygeist: Raising C to Polyhedral MLIR

William S. Moses*
MIT CSAIL
Cambridge, MA, USA
wmoses@mit.edu

Lorenzo Chelini*
TU Eindhoven
Eindhoven, The Netherlands
l.chelini@tue.nl

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

Oleksandr Zinenko
Google Inc.
Paris, France
zinenko@google.com

***Abstract*—We present Polygeist, a new compilation flow that connects the MLIR compiler infrastructure to cutting edge polyhedral optimization tools. It consists of a C and C++ frontend capable of converting a broad range of existing codes into MLIR suitable for polyhedral transformation and a bi-directional conversion between MLIR and OpenScop exchange format. The Polygeist/MLIR intermediate representation featuring high-level (affine) loop constructs and n-D arrays embedded into a single static assignment (SSA) substrate enables an unprecedented combination of SSA-based and polyhedral optimizations. We illustrate this by proposing and implementing two extra transformations: statement splitting and reduction parallelization. Our evaluation demonstrates that Polygeist outperforms on average both an LLVM IR-level optimizer (Polly) and a source-to-source state-of-the-art polyhedral compiler (Pluto) when exercised on the Polybench/C benchmark suite in sequential (2.53x vs 1.41x, 2.34x) and parallel mode (9.47x vs 3.26x, 7.54x) thanks to the new representation and transformations.**

I. Introduction

Improving the efficiency of computation has always been one of the prime goals of computing. Program performance can be improved significantly by reaping the benefits of parallelism, temporal and spatial locality, and other performance sources. Relevant program transformations are particularly tedious and challenging when targeting modern multicore CPUs and GPUs with deep memory hierarchies and parallelism, and are often performed automatically by optimizing compilers.

The polyhedral model enables precise analyses and a relatively easy specification of transformations (loop restructuring, automatic parallelization, etc.) that take advantage of hardware performance sources. As a result, there is growing evidence that the polyhedral model is one of the best frameworks for efficient transformation of compute-intensive programs [1], [2], [3], and for programming accelerator architectures [4], [5], [6]. Consequently, the compiler community has focused on building tools that identify and optimize parts of the program that can be represented within the polyhedral model (commonly referred to as static-control parts, or SCoP's). Such tools tend to fall into two categories.

Compiler-based tools like Polly [7] and Graphite [8] detect and transform SCoPs in compiler intermediate representations (IRs). While this offers seamless integration with rest of the compiler, the lack of high-level structure and information hinders the tools' ability to perform analyses and transformations. This structure needs to be recovered from optimized IR, often imperfectly or at a significant cost [9]. Moreover, common compiler optimizations such as LICM may interfere with the process [10]. Finally, low-level IRs often lack constructs for, e.g., parallelism or reductions, produced by the transformation, which makes the flow more complex.

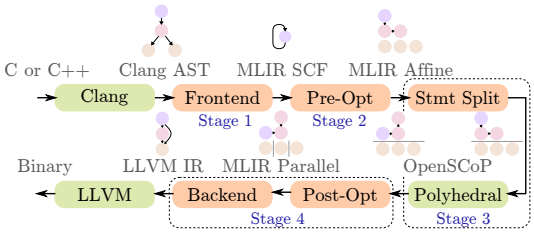

Fig. 1. The Polygeist compilation flow consists of 4 stages. The frontend traverses Clang AST to emit MLIR SCF dialect (Section III-A), which is raised to the Affine dialect and pre-optimized (Section III-B). The IR is then processed by a polyhedral scheduler (Sections III-C,III-D) before post-optimization and parallelization (Section III-E). Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM.

Source-to-source compilers such as Pluto [11], PoCC [12] and PPCG [5] operate directly on C or C++ code. While this can effectively leverage the high-level information from source code, the effectiveness of such tools is often reduced by the lack of enabling optimizations such as those converting hazardous memory loads into single-assignment virtual registers. Furthermore, the transformation results must be expressed in C, which is known to be complex [13], [14] and is also missing constructs for, e.g., reduction loops or register values not backed by memory storage.

This paper proposes and evaluates the benefits of a polyhedral compilation flow, Polygeist (Figure 1), that can leverage both the high-level structure available in source code and the fine-grained control of compiler optimization provided by low-level IRs. It builds on the recent MLIR compiler infrastructure that allows the interplay of multiple abstraction levels within the same representation, during the same transformations [15]. Intermixable MLIR abstractions, or *dialects*, include high-level constructs such as loops, parallel and reduction patterns; low-level representations fully covering LLVM IR [16]; and a polyhedral-inspired representation featuring loops and memory accesses annotated with affine expressions. Moreover, by combining the best of source-level and IR-level tools in an end-to-end polyhedral flow, Polygeist preserves high-level information and leverages them to perform new or improved

* Equal contribution.

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

Fig. 2. Generic MLIR syntax for an operation with two operands, one result, one attribute and a single-block region.

optimizations, such as statement splitting and loop-carried value detection, on a *lower*-level abstraction as well as to influence downstream optimizations.

We make the following contributions:

- a C and C++ frontend for MLIR that preserves high-level loop structure from the original source code;
- an end-to-end flow with raising to and lowering from the polyhedral model, leveraging our abstraction to perform more optimizations than both source- and IR-level tools, including reduction parallelization;
- an exploration of new transformation opportunities created by Polygeist, in particular, statement splitting;
- and an end-to-end comparison between Polygeist and state-of-the-art source- and IR-based tools (Pluto [11] and Polly [14]) along with optimization case studies.

II. The MLIR Framework

A. Overview

MLIR is an optimizing compiler infrastructure inspired by LLVM [16] with a focus on extensibility and modularity [15]. Its main novelty is the IR supporting a fully extensible set of instructions (called *operations*) and types. Practically, MLIR combines SSA with nested regions, allowing one to express a range of concepts as first-class operations including machine instructions such as floating-point addition, structured control flow such as loops, hardware circuitry [17], and large machine learning graphs. Operations define the runtime semantics of a program and process immutable values. Compile-time information about values is expressed in *types*, and information about operations is expressed in *attributes*. Operations can have attached regions, which in turn contain (basic) blocks of further operations. The generic syntax, accepted by all operations, illustrates the structure of MLIR in Figure 2. Additionally, MLIR supports user-defined custom syntax.

Attributes, operations, and types are organized in *dialects*, which can be thought of as modular libraries. MLIR provides a handful of dialects that define common operations such as modules, functions, loops, memory or arithmetic instructions, and ubiquitous types such as integers and floats. We discuss the dialects relevant to Polygeist in the following sections.

B. Affine and MemRef Dialects

The *Affine* dialect [18] aims at representing SCoP's with explicit polyhedral-friendly loop and conditional constructs. The core of its representation is the following classification of value categories:

```
%c0 = constant 0 : index
%0 = memref.dim %A, %c0 : memref<?xf32>
%1 = memref.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>
  }
}
```

Fig. 3. Polynomial multiplication in MLIR using Affine and Standard dialects.

- *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 affine map (see below) application results 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 (`intptr_t` in C).

MLIR provides two attributes relevant for the Affine dialect:

- *Affine maps* are multi-dimensional (quasi-)linear functions that map a list of dimension and symbol arguments to a list of results. For example, $(d_0, d_1, d_2, s_0) \to (d_0 + d_1, s_0 \cdot d_2)$ is a two-dimensional quasi-affine map, which can be expressed in MLIR as `affine_map<(d0,d1,d2)[s0] -> (d0+d1, s0*d2)>`. Dimensions (in parentheses on the left) and symbols (in brackets on the left) are separated to allow quasi-linear expressions: symbols are treated as constants, which can therefore be multiplied with dimensions, whereas a product of two dimensions is invalid.
- *Integer sets* are collections of integer tuples constrained by conjunctions of (quasi-)linear expressions. For example, a "triangular" set $\{(d_0, d_1) : 0 \le d_0 < s_0 \wedge 0 \le d_1 \le d_0\}$ is represented 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 loop-invariant lower and upper bounds expressed as affine maps with a constant step. An `affine.parallel` is a "multifor" loop nest, iterations of which may be executed concurrently. Both kinds of loops support reductions via loop-carried values as well as max(min) expression lower(upper) bounds. An `affine.if` is a conditional construct, with an optional `else` region, and a condition defined as inclusion of the given values into an integer set. Finally, `affine.load` and `affine.store` express memory accesses where the address computation is expressed as an affine map.

Figure 3 illustrates the Affine dialect for a polynomial multiplication, `C[i+j] += A[i] * B[j]`. This simple example highlights the fact that MLIR supports, and encourages, IRs from different dialects to be used together.

A core MLIR type—`memref`, which stands for **mem**ory **ref**erence—and the corresponding *`memref` dialect* are also featured in Figure 3. The `memref` type describes a structured multi-index pointer into memory, e.g., `memref<?xf32>` denotes a 1-d array of floating-point elements; and the *`memref` dialect* provides memory and type manipulation operations, e.g., `memref.dim` retrieves the dimensionality of a `memref` object. `memref` does not allow internal aliasing, i.e., different subscripts always point to different addresses. This effectively defines away the delinearization problem that hinders the application of polyhedral techniques at the LLVM IR level [9]. Throughout this paper, we only consider `memref`s with the default *layout* that corresponds to contiguous row-major storage compatible with C ABI (Application Binary Interface). In practice, `memref`s support arbitrary layouts expressible as affine maps, but these are not necessary in Polygeist context.

C. 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, the `scf.for` loop accepts any integer value as loop bounds, which are not necessarily affine expressions.
- The *Standard* (`std`) dialect contains common operations such as integer and float arithmetic, which is used as a common lowering point from higher-level dialects before fanning out into multiple target dialects and can be seen as a generalization of LLVM IR [16].
- The *LLVM* dialect directly maps from LLVM IR instructions and types to MLIR, primarily to simplify the translation between them.
- The *OpenMP* dialect provides a dialect- and platform-agnostic representation of OpenMP directives such as "parallel" and "workshare loop", which can be used to transform OpenMP constructs or emit LLVM IR that interacts with the OpenMP runtime.
- The *Math* dialect groups together mathematical operations on integer and floating type beyond simple arithmetic, e.g., `math.pow` or `math.sqrt`.

III. An (Affine) MLIR Compilation Pipeline

The Polygeist pipeline consists of 4 components (Figure 1):

1) a frontend that allows entering MLIR at the SCF loops level from C or C++ code (Section III-A);
2) a preprocessing step within MLIR that raises to the Affine dialect (Section III-B);
3) a polyhedral scheduler of the Affine parts of the program *via* a round-trip to and from OpenSCoP (Section III-C) and running Pluto transformations, controlled by the new statement splitting heuristic (Section III-D);
4) a backend that runs postprocessing MLIR optimizations (section III-E) and final lowering to an executable.

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<?xty>`
`ty &`	`ty *`	`memref<1xty>`
`ty **`	`ty **`	`memref<memref<?xty>>`
`ty[N][M]`	`[N x [M x ty]]*`	`memref<NxMxty>`

Fig. 4. Type correspondence between C, LLVM IR and MLIR types.

A. Frontend

Polygeist builds off the Clang compiler to emit MLIR, directly analyzing Clang's AST. Polygeist thus avoids reimplementing parsing and language-level semantic analysis and handles modern C and C++ features. As is typical for compiler frontends, Polygeist creates a recursive symbol table data structure to look up the correct variable for a given scope. Polygeist lazily registers all global variables and functions found in the AST to its symbol table before generating any code. Polygeist then traverses the call graph from a given entry function (`main` by default), creating and defining MLIR functions as necessary.

a) Control Flow & High Level Information: In contrast to traditional compiler pipelines, targeting a branch-based IR, Polygeist leverages the high-level MLIR operations such as `scf.while` (a looping construct) and `scf.if` (a conditional construct) within the SCF dialect to preserve the control flow structure of the source code. C-level `continue` and `break` constructs are handled by introducing signal variables and checking them before each operation that follows original constructs. Furthermore, within a `#pragma scop`, Polygeist assumes that the program is affine and uses an `affine.for` to represent loops directly.

b) Types & Polygeist ABI: While emitting operations, Polygeist must decide how to represent C or C++ types within MLIR. For primitive types such as `int` or `float`, Polygeist emits an MLIR variant of that type with the same width as would be used within LLVM/Clang. This allows Polygeist to keep the same ABI as code compiled by a normal C or C++ compiler when calling a function with only primitive types. On the other hand, for pointer, reference and array types, Polygeist uses `memref` type (Figure 4). This allows Polygeist to preserve more of the structure available within the original program (e.g., multi-dimensional arrays) and enables interaction with MLIR's high-level memory operations.

This diverges from the C ABI for any functions with pointer arguments and wouldn't interface correctly with C functions. Polygeist addresses this by providing an attribute for function arguments and allocations to use a C-compatible pointer type rather than `memref`, applied by default to external functions such as `strcmp` and `scanf`. When calling a pointer-ABI

function with a memref-ABI argument, Polygeist generates wrapper code that recovers the C ABI-compatible pointer from memref and ensures the correct result. Figure 5 shows an example demonstrating how the Polygeist and C ABI may interact for a small program.

When allocating and deallocating memory, this difference in ABI becomes significant. This is because allocating several bytes of an array with malloc then casting to a memref will not result in legal code (as memref itself may not be implemented with a raw pointer). Thus, Polygeist identifies calls to allocation and deallocation functions and replaces them with legal equivalents for memref.

Functions and global variables are emitted using the same name used by the C or C++ ABI. This ensures that all external values are loaded correctly, and multi-versioned functions (such as those generated by C++ templates or overloading) have distinct names and definitions.

c) Instruction Generation: For most instructions, Polygeist directly emits an MLIR operation corresponding to the equivalent C operation (addi for integer add, call for function call, etc.). For some special instructions such as a call to pow, Polygeist chooses to emit a specific MLIR operation in the Math dialect, instead of a call to an external function (defined in libm). This permits such instructions to be better analyzed and optimized within MLIR.

Operations that involve memory or pointer arithmetic require additional handling. MLIR does not have a generic pointer arithmetic instruction; instead, it requires that load and store operations contain all of the indices being looked up. This presents issues for operations that perform pointer arithmetic. To remedy this, we introduce a temporary subindex operation for memref's keeps track of the additional address offsets. A subsequent optimization pass within Polygeist, forwards the offsets in a subindex to any load or store which uses them.

d) Local Variables: Local variables are handled by allocating a memref on stack at the top of a function. 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 Polygeist. 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 memrefs of memref in the Polybench suite.

B. Raising to Affine

The translation from C or C++ to MLIR directly preserves high-level information about loop structure and n-D arrays, but does not generate other Affine operations. Polygeist subsequently raises memory, conditional, and looping operations into their Affine dialect counterparts if it can prove them to be legal affine operations. If the corresponding frontend code was enclosed within #pragma scop, Polygeist assumes it is always legal to raise all operations within that region without additional checks.[1] Any operations which are not proven or assumed to be affine remain untouched. We perform simplifications on affine maps to remove loops with zero or one iteration and drop branches of a conditional with a condition known at compile time.

a) Memory operations and loop bounds: To convert an operation, Polygeist replaces its bound and subscript operands with identity affine maps (affine_map<()[s0]->(s0)>[%bound]). It then folds the operations computing the map operands, e.g., addi, muli, into the map itself. Values that are transitively derived from loop induction variables become map dimensions and other values become symbols. For example, affine_map<()[s0]->(s0)>[%bound] with %bound = addi %N, %i, where %i is an induction variable, is folded into affine_map<(d0)[s0] ->(s0 + d0)>(%i)[%N]. The process terminates when no operations can be folded or when Affine value categorization rules are satisfied.

b) Conditionals: Conditional operations are emitted by the frontend for two input code patterns: if conditions and ternary expressions. The condition is transformed by introducing an integer set and by folding the operands into it similarly to the affine maps, with in addition and operations separating set constraints and not operations inverting them (affine.if only accepts ≥ 0 and $= 0$ constraints). Polygeist processes nested conditionals with C-style short-circuit semantics, in which the subsequent conditions are checked within the body of the preceding conditionals, by hoisting conditions outside the outermost conditional when legal and replacing them with a boolean operation or a select. This is always legal within #pragma scop.

Conditionals emitted for ternary expressions often involve memory loads in their regions, which prevent hoisting due to side effects. We reuse our mem2reg pass to replace those to equivalent earlier loads when possible to enable hoisting. Empirically, this is sufficient to process all ternary expressions in the Polybench/C suite [19]. Otherwise, ternary expressions would need to be packed into a single statement by the downstream polyhedral pass.

C. Connecting MLIR to Polyhedral Tools

Regions of the input program expressed using MLIR Affine dialect are amenable to the polyhedral model. Existing tools, however, cannot directly consume MLIR. We chose to implement a bi-directional conversion to and from OpenScop [20], an exchange format readily consumable by numerous polyhedral tools, including Pluto [11], and further convertible to isl [21] representation. This allows Polygeist to seamlessly connect with tools created in polyhedral compilation research without having to amend those tools to support MLIR.

Most polyhedral tools are designed to operate on C or FORTRAN inputs build around *statements*, which do not have a direct equivalent in MLIR. Therefore, we design a mechanism to create statement-like structure from chains of MLIR

[1] All kernels within Polybench are successfully raised to Affine with or without the use of #pragma scop.

```
void setArray(int N, double val, double* array) {...}
int main(int argc, char** argv) {
  ...
  cmp = strcmp(str1, str2)
  ...
  double array[10];
  setArray(10, 42.0, 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 = memref.alloca() : memref<10xf64>
  %arraycst = memref.cast %array : memref<10xf64> to
    memref<?xf64>
  %val = constant 42.0 : f64
  call @setArray(%N, %val, %arraycst) :
           (i32, f64, memref<?xf64>) -> ()
}
```

Fig. 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`), Polygeist falls back to emitting `llvm.ptr` and generates conversion code.

```
affine.for %i = ...
  %0 = affine.load %A[%i]
  affine.store %other, %A[%i] // motion-barrier
  affine.for %j = ... {
    %1 = affine.load %B[%j]
    %10 = mulf %0, %1 : f64    // use-1
    store %10, %res[%i, %j]
  %20 = addf %0, %0 : f64      // use-2

                                 ⇓

%tmp = memref.alloca() : memref<1xf64>
affine.for %i = ...
  %0 = affine.load %A[%i]
  affine.store %0, %tmp[0]     // store to scratchpad
  affine.store %other, %A[%i] // motion-barrier
  affine.for %j = ...
    %1 = affine.load %B[%j]
    %2 = affine.load %tmp[0]   // load back for use-1
    %10 = mulf %2, %1 : f64    // use-1 (%2 instead of %0)
    affine.store %10, %res[%i, %j]
  %19 = affine.load %tmp[0]    // load back for use-2
  %20 = addf %19, %19 : f64    // use-2 (%19 instead of %0)
  // ...
```

Fig. 6. Polygeist breaks region-spanning use-def chains and handles multi-use values by introducing scratchpad storage when operation duplication is illegal. In absence of `motion-barrier` statement, the `%0` load would be duplicated and sunk. Pseudo-MLIR with types and braces omitted for brevity.

operations. We further demonstrate that this gives Polygeist an ability to favorably affect the behavior of the polyhedral scheduler by controlling statement granularity (Section III-D).

a) Simple Statement Formation: Observing that C statements amenable to the polyhedral model are (mostly) variable assignments, we can derive a mechanism to identify statements from chains of MLIR operations. A `store` into memory is the last operation of the statement. The backward slice of this operation, i.e., the operations transitively computing its operands, belong to the statement. The slice extension stops at operations producing a value categorized as affine dimension or symbol, directly usable in affine expressions. Such values are loop induction variables or loop-invariant constants.

Some operations may end up in multiple statements if the value is used more than once. However, we need the mapping between operations and statements to be bidirectional in order to emit MLIR after the scheduler has restructured the program without considering SSA value visibility rules. If an operation with multiple uses is side effect free, Polygeist simply duplicates it. For operations whose duplication is illegal, Polygeist stores their results in stack-allocated `memref`'s and replaces all further uses with memory loads. Figure 6 illustrates the transformation for value `%0` used in operation `%20`. This creates a new statement.

b) Region-Spanning Dependencies: In some 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 cannot be communicated to the polyhedral scheduler. Polygeist resolves this by storing the value in a stack-allocated `memref` in the defining region and loading it back in the user regions. Figure 6 illustrates this transformation for value `%0` used in operation `%10`. Similarly to the basic case, this creates a new statement in the outer loop that can be scheduled independently.

This approach can be seen as a reg2mem conversion, the inverse of mem2reg performed in the frontend. It only applies to a subset of values, and may be undone after polyhedral scheduling has completed. Furthermore, to decrease the number of dependencies and memory footprint, Polygeist performs a simple value analysis and avoids creating stack-allocated buffers if the same value is already available in another memory location and can be read from there.

c) 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 [22]. This process also makes the inter-statement SSA dependencies clear. These dependencies exist between calls that *use* the same SSA value, but there are no values defined by these calls. We 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. Figure 7 demonstrates the resulting IR.

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

```
func @S1(%A: memref<?xf64>, %tmp: memref<1xf64>,
         %i: index)
  %0 = affine.load %A[%i]
  affine.store %0, %tmp[0]   // store to scratchpad

func @S2(%A: memref<?xf64>, %other: f64, %i: index)
  affine.store %other, %A[%i]

func @S3(%B: memref<?x?xf64>, %tmp: memref<1xf64>,
         %res: memref<?x?xf64>, %i: index, %j: index)
  %1 = affine.load %B[%k, %j]
  %2 = affine.load %tmp[0]   // load back for use-1
  %10 = mulf %2, %1 : f64    // use-1
  affine.store %10, %res[%i, %j]

func @S4(%tmp: memref<1xf64>, ...)
  %19 = affine.load %tmp[0]  // load back for use-2
  %20 = addf %19, %19 : f64  // use-2
  // ...

%tmp = memref.alloca() : memref<1xf64>
affine.for %i = ...
  call @S1(%A, %tmp, %i)
  call @S2(%A, %other, %i)
  affine.for %j = ...
    call @S3(%B, %tmp, %res, %i, %j)
  call @S4(%tmp)
```

Fig. 7. Outlining makes polyhedral "statements" visible in code from Fig. 6.

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. Affine constructs in OpenScop are represented as lists of linear equality ($= 0$) or inequality (≥ 0) coefficients, which matches exactly the internal representation in MLIR, making the conversion straightforward.

d) Code Generation Back to MLIR: The Pluto scheduler produces new schedules in OpenScop as a result. Generating loop structure back from affine schedules is a solved, albeit daunting, problem [13], [14]. Polygeist relies on CLooG [13] to generate an initial loop-level AST, which it then converts to Affine dialect loops and conditionals. There is no need to simplify affine expressions at code generation since MLIR accepts them directly and can simplify them at a later stage. Statements are introduced as function calls with rewritten operands and then inlined.

D. Controlling Statement Granularity

Recall that Polygeist reconstructs "statements" from sequences of primitive operations (Section III-C). We initially designed an approach that recovers the statement structure similar to that in the C input, but this is not a requirement. Instead, statements can be formed from any subsets of MLIR operations as long as they can be organized into loops and sorted topologically (i.e., there are no use-def cycles between statements). To expose the dependencies between such statements to the affine scheduler, we reuse the idea of going through scratchpad memory: each statement writes the values required by other statements to dedicated memory locations,

```
for(i=0; i<NI; i++)
  for(j=0; j<NJ; j++)
    for(k=0; k<NK; k++)
S:    A[i][j]+=f(B[k][i],C[k][j]);
```

⇓

```
for(i=0; i<NI; i++)                    double M[NK];
  for(j=0; j<NJ; j++)                  for(k=0; k<NK; k++)
    double M[NK];                  ⇒     for(i=0; i<NI; i++)
    for(k=0; k<NK; k++)                    for(j=0; j<NJ; j++)
S:    M[k]=f(B[k][i],C[k][j]);       S:    M[k]=f(B[k][i],C[k][j]);
T:    A[i][j] += M[k];               T:    A[i][j] += M[k];
```

Fig. 8. Splitting a nested reduction statement (top) into a fully parallel compute statement and a trivial reduction statement (bottom left) makes Pluto generate different schedules (bottom right). Further scratchpad array expansion may enable loop fission and give scheduler even more liberty.

and the following statements read from those. The scratchpads are subject to partial array expansion [23] to minimize their effect on the affine scheduler as single-element scratchpad arrays create artificial scalar dependencies. This change in *statement granularity* gives the affine scheduler unprecedented flexibility allowing it to chose different schedules for different *parts* of the same C statement.

Consider, for example, the statement `S` in Figure 8(top) surrounded by three loops iterating over `i`, `j` and `k`. Such contraction patterns are common in computational programs (this particular example can be found in the `correlation` benchmark with `B`≡`C`, see Section V-E). The loop order that best exploits the locality is (`k`, `i`, `j`), which results in temporal locality for reads from `B` (the value is reused in all iterations of the now-innermost `j` loop) and in spatial locality for reads from `C` (consecutive values are read by consecutive iterations, increasing the likelihood of L1 cache hits). Yet, Pluto never proposes such an order because of a reduction dependency along the `k` dimension due to repeated read/write access to `A[i][j]` as Pluto tends to pick loops with fewer dependencies as outermost. While the dependency itself is inevitable, it can be moved into a separate statement `T` in Figure 8(bottom left). This approach provides scheduler with more freedom of choice for the first statement at a lesser memory cost than expanding the entire `A` array. It also factors out the reduction into a "canonical" statement that is easier to process for the downstream passes, e.g., vectorization.

Implementing this transformation at the C level would require manipulating C AST and reasoning about C (or even C++) semantics. This is typically out of reach for source-to-source polyhedral optimizers such as Pluto that treat statements as black boxes. While it is possible to implement this transformation at the LLVM IR level, e.g., in Polly, where statements are also reconstructed and injection of temporary allocations is easy, the heuristic driving the transformation is based on the loop structure and multi-dimensional access patterns which are difficult to recover at such a low level [9].

The space of potential splittings is huge—each MLIR operation can potentially become a statement. Therefore, we devise a heuristic to address the contraction cases similar to Figure 8. Reduction statement splitting applies to statements:

- surrounded by at least 3 loops;
- with LHS≠RHS, and using all loops but the innermost;
- with two or more different access patterns on the RHS.

This covers statements that could have locality improved by a different loop order and with low risk of undesired fission. This heuristic merely serves as an illustration of the kind of new transformations Polygeist can enable.

E. Post-Transformations and Backend

Polygeist allows one to operate on both quasi-syntactic and SSA level, enabling analyses and optimizations that are extremely difficult, if not impossible, to perform at either level in isolation. In addition to statement splitting, we propose two techniques that demonstrate the potential of Polygeist.

a) Transforming Loops with Carried Values (Reductions): Polygeist leverages MLIR's first-class support for loop-carried values to detect, express and transform reduction-like loops. This support does not require source code annotations, unlike source-level tools [24] that use annotations to enable detection, nor complex modifications for parallel code emission, unlike Polly [25], which suffers from LLVM missing first-class parallel constructs. We do not modify the polyhedral scheduler either, relying on post-processing for reduction parallelization, including outermost parallel reduction loops.

The overall approach follows the definition proposed in [26] with adaptations to MLIR's region-based IR, and is illustrated in Figure 9. Polygeist identifies memory locations modified on each iteration, i.e. `load`/`store` pairs with loop-invariant subscripts and no interleaving aliasing `stores`, by scanning the single-block body of the loop. These are transformed into *loop-carried values* or secondary induction variables, with the `load`/`store` pair lifted out of the loop and repurposed for reading the initial and storing the final value. Loop-carried values may be updated by a chain of side effect-free operations in the loop body. If this chain is known to be associative and commutative, the loop is a *reduction*. Loop-carried values are detected even in absence of reduction-compatible operations. Loops with such values contribute to mem2reg, decreasing memory footprint, but are not subject to parallelization.

b) Late Parallelization: Rather than relying on the dependence distance information obtained by the affine scheduler, Polygeist performs a separate polyhedral analysis to detect loop parallelism in the generated code. The analysis itself is a classical polyhedral dependence analysis [27], [28] implemented on top of MLIR region structure. Performing it after SSA-based optimizations, in particular mem2reg and reduction detection, allows parallelizing more loops. In particular, reduction loops and loops with variables whose value is only relevant within a single iteration similar to live-range reordering [29] but without expensive additional polyhedral analyses (live-range of an SSA value defined in a loop never extends beyond the loop).

IV. Evaluation

Our evaluation has two goals. 1) We want to demonstrate that the code produced by Polygeist without additional optimization does not have any inexplicable performance differences than a state-of-the-art compiler like Clang. 2) We explore how Polygeist's internal representation can support a mix of affine and SSA-based transformation in the same compilation flow, and evaluate the potential benefits compared to existing source and compiler-based polyhedral tools.

```
affine.for %i = ... {
  // Reduction into r1[0]
  %1 = affine.load %r1[0]
  %5 = addi %1, %2
  affine.store %5, %r1[0]
  // Loop-dependent load
  %10 = affine.load %r2[%i]
  %15 = addi %10, %2
  // Inteleaving store
  %20 = affine.load %r2[0]
  affine.store %21, %r2[0]
  %25 = addi %20, %2
  // May have side effects
  %30 = affine.load %r3[0]
  call @f(%30, %2)
}
```

⇒

```
%init = affine.load %r1[0]
%red = affine.for %i = ...
 iter_args(%arg = %init) {
  // Reduction accumulation
  %5 = addi %arg, %2
  // Loop-dependent load
  %10 = affine.load %r2[%i]
  %15 = addi %10, %2
  // Inteleaving store
  %20 = affine.load %r2[0]
  affine.store %21, %r2[0]
  %25 = addi %20, %2
  // May have side effects
  %30 = affine.load %r3[0]
  call @f(%30, %2)
  // Yield accumulated
  affine.yield %5
}
affine.store %red, %r1[0]
```

Fig. 9. Polygeist detects memory locations accessed in all loop iterations, e.g. reduction accumulators such as `%r1[0]` and transforms them to loop-carried values (secondary induction variables), except when computed with side-effects, interleaved stores or by non-associative/commutative operations.

A. Experimental Setup

We ran our experiments on an AWS `c5.metal` instance with hyper-threading and Turbo Boost disabled. The system is Ubuntu 20.04 running on a dual-socket Intel Xeon Platinum 8275CL CPU at 3.0 GHz with 24 cores each, with 0.75, 35, 35.75 MB L1, L2, L3 cache per socket, respectively, and 256 GB RAM. We ran all 30 benchmarks from PolyBench [19], using the "EXTRALARGE" dataset. Pluto is unable to extract SCoP from the `adi` benchmark. We ran a total of 5 trials for each benchmark, taking the execution time reported by PolyBench; the median result is taken unless stated otherwise. Every measurement or result reported in the following sections refers to double-precision data. All experiments were run on cores 1-8, which ensured that all threads were on the same socket and did not potentially conflict with processes scheduled on core 0.

In all cases, we use two-stage compilation: (i) using `clang` at `-O3` excluding unrolling and vectorization; or Polygeist to emit LLVM IR from C; (ii) using `clang` at `-O3` to emit the final binary. As several optimizations are not idempotent, a second round of optimization can potentially significantly boost (and rarely, hinder) performance. This is why we chose to only perform vectorization and unrolling at the last optimization stage. Since Polygeist applies some optimizations at the MLIR level (e.g., mem2reg), we compare against the two-stage compilation pipeline as a more fair baseline (Clang). We also evaluate a single-stage compilation to assess the effect of the two-stage flow (ClangSing).

B. Baseline Performance

Polygeist must generate code with runtime *as close as possible* to that of existing compilation flows to establish a solid baseline. In other words, Polygeist should *not introduce overhead nor speedup* unless explicitly instructed otherwise, to allow for measuring the effects of additional optimizations. We evaluate this by comparing the runtime of programs produced by Polygeist with those produced by Clang at the same commit (Apr 2021)[2]. Figure 10 summarizes the results with the following flows:

- CLANG: A compilation of the program using Clang, when running two stages of optimization;
- CLANGSING: A compilation of the program using Clang, when running one stage of optimization;
- MLIR-CLANG: A compilation flow using the Polygeist frontend and preprocessing optimizations within MLIR, but not running polyhedral scheduling nor postprocessing.

C. Compilation Flows

We compare Polygeist with a source-level and an IR-level optimizer (Pluto and Polly) in the following configurations:

- PLUTO: Pluto compiler auto-transformation [11] using `polycc`[3] with `-noparallel` and `-tile` flags;
- PLUTOPAR: Same as above but with `-parallel` flag;
- POLLY: Polly [7] LLVM passes with affine scheduling and tiling, and no pattern-based optimizations [30];
- POLLYPAR: Same as above with auto-parallelization;
- POLYGEIST: Our flow with Pluto and extra transforms;
- POLYGEISTPAR: Same as above but with `-parallel` Pluto schedule, Polygeist parallelization and reductions.

Running between source and LLVM IR levels, we expect Polygeist to benefit from both worlds, thus getting code that is on par or better than competitors. When using Pluto, both standalone and within Polygeist, we disable the emission of vectorization hints and loop unrolling to make sure both transformations are fully controlled by the LLVM optimizer, which also runs in Polly flows. We run Polly in the latest stage of Clang compilation, using `-mllvm -polly` and additional flags to enable affine scheduling, tiling and parallelization as required. Polly is taken at the same LLVM commit as Clang. We disable pattern-based optimizations [30] that are not available elsewhere. Figures 11 and 12 summarize the results for sequential and parallel flows, respectively.

V. PERFORMANCE ANALYSIS

A. Benchmarking

The transformation of reduction loops, in particular parallelization, may result in a different order of partial result accumulation. This is not allowed under IEEE 754 semantics, but is supported by compilers with `-ffast-math` option.

We found that Polybench allocation function hinders Clang/LLVM alias analysis, negatively affecting performance in, e.g., `adi`. Therefore, we modified all benchmarks to use `malloc` that is known to produce non-aliasing pointers.

B. Baseline Comparison

We did not observe a significant difference between the runtimes of CLANG and CLANGSING configurations, with a geometric mean of 0.43% symmetric difference[4] across benchmarks. Therefore, we only consider CLANG as baseline throughout the remainder of this paper. We did not observe a significant difference between the runtimes of CLANG and MLIR-CLANG configurations either, with a geometric mean of 0.24% symmetric difference.

We found a variation in runtimes of short-running benchmarks, in particular `jacobi-1d`. This can be attributed to the interaction with the data initialization and benchmarking code, and with other OS processes. Excluding the benchmarks running in under 0.05s (`jacobi-1d`, `gesummv`, `atax`, `bicg`) from the analysis, we obtain 0.32% and 0.17% geomean symmetric differences respectively for the two comparisons above. These results suggest that our flow has no unexplained (dis)advantages over the baseline.

C. Performance Differences in Sequential Code

Overall, Polygeist leads to larger speedups, with $2.53\times$ geometric mean, than both Pluto ($2.34\times$) and Polly ($1.41\times$), although improvements are not systematic. Some difference between Polygeist and Polly is due to the employed polyhedral schedulers, e.g., in `lu` and `mvt`. Polygeist produces code faster tha both Pluto and Polly in `2mm`, `3mm` and others thanks to statement splitting, see Section V-E.

Given identical statements and schedules, codegen-level optimization accounts for other performance difference. `seidel-2d` is the clearest example: Pluto executes $2.7{\cdot}10^{11}$ more integer instructions than Polygeist. Assuming these to be index/address computations, a mix of `add` (throughput $1/2$ or $1/4$) and `imul/shl` (thoughput 1), we can expect a $\approx$ 59s difference at 3GHz, consistent with experimental observations. Polygeist optimizes away a part of those in its post-optimization phase and emits homogeneous address computation from `memref` with proper machine size type, enabling more aggressive bound analysis and simplification in the downstream compiler. Conversely, `jacobi-2d` has poorer performance because Polygeist gives up on simplifying CLooG code, with up to 75 statement copies in 40 branches, for compiler performance reasons, as opposed to Clang that takes up to 5s to process it but results in better vectorization. Further work is necessary to address this issue by emitting vector instructions directly from Polygeist.

D. Performance Differences In Parallel Code

Similarly to sequential code, some performance differences are due to different schedulers. For example, in `cholesky` and `lu`, both Pluto and Polygeist outperform Polly, and the remaining gap can be attributed to codegen-level differences. Conversely, in `gemver` and `mvt` Polly has a benefit over both

[2]LLVM commit `20d5c42e0ef5d252b434bcb610b04f1cb79fe771`

[3]Pluto commit `dae26e77b94b2624a540c08ec7128f20cd7b7985`

[4]Symmetric difference is computed as $2 \cdot |a-b|/(a+b)$.

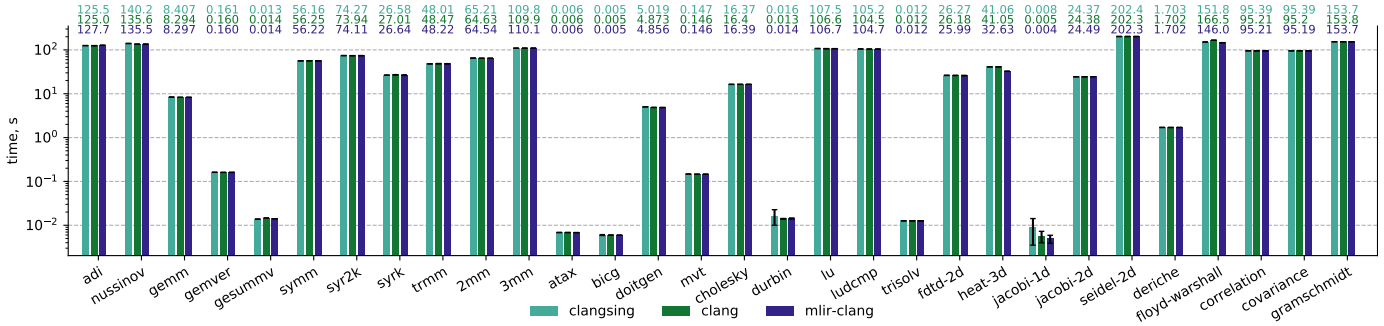

Fig. 10. Mean and 95% confidence intervals (log scale) of program run time across 5 runs of Polybench in CLANG, CLANGSING and MLIR-CLANG configurations, lower is better. The run times of code produced by Polygeist without optimization is comparable to that of Clang. No significant variation is observed between single and double optimization. Short-running `jacobi-1d` shows high intra-group variation.

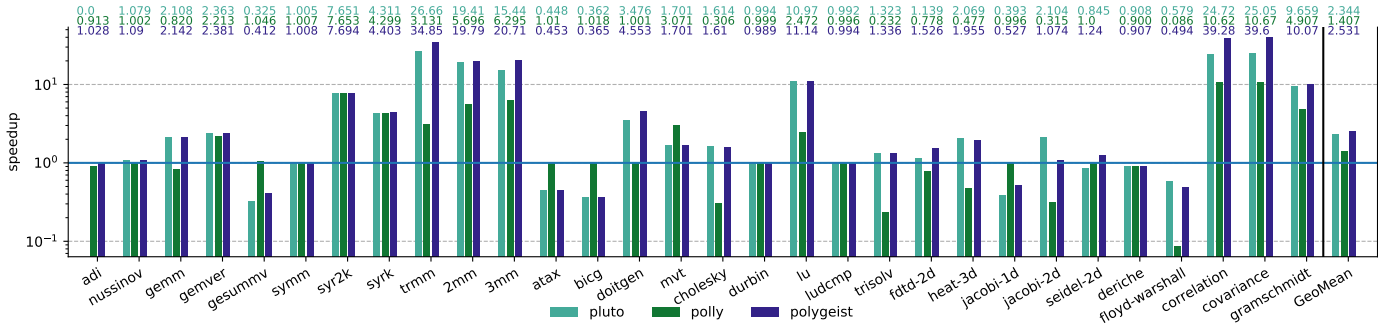

Fig. 11. Median speedup over CLANG for sequential configurations (log scale), higher is better. Polygeist outperforms (2.53× geomean speedup) both Pluto (2.34×) and Polly (1.41×) on average. Pluto can't process `adi`, which is therefore excluded from summary statistics.

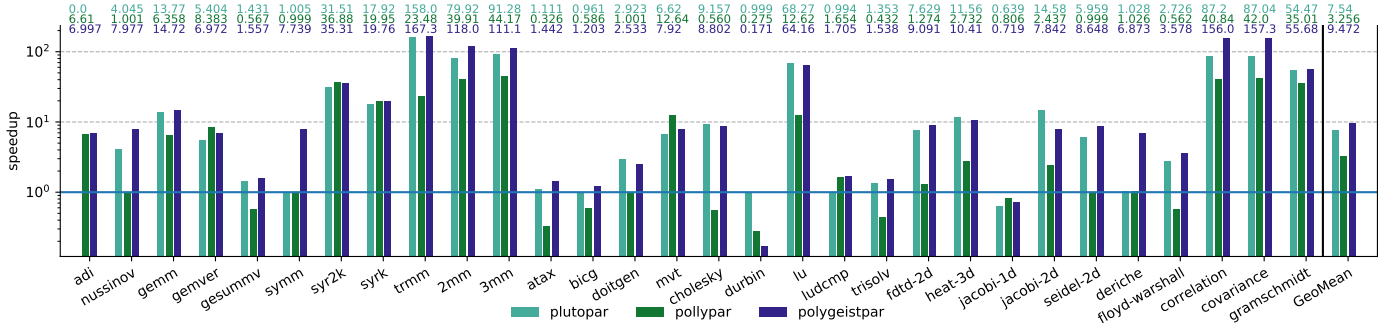

Fig. 12. Median speedup over CLANG for parallel configurations (log scale), higher is better. Polygeist outperforms (9.47× geomean speedup) both Pluto (7.54×) and Polly (3.26×) on average. Pluto can't process `adi`, which is therefore excluded from summary statistics.

Pluto and Polygeist. On `ludcmp` and `syr(2)k`, SSA-level optimizations let Polygeist produce code which is faster than Pluto and at least as fast as Polly. These results demonstrate that Polygeist indeed leverages the benefits of both the affine and SSA-based optimizations.

Polygeist is the only flow that obtains speedup on `deriche` (6.9×) and `symm` (7.7×). Examining the output code, we observe that only Polygeist manages to parallelize these two benchmarks. Considering the input code in Figure 13, one can observe that the `i` loop reuses the `ym1` variable, which is interpreted as parallelism-preventing loop-carried dependency by polyhedral schedulers. Polygeist performs its own parallelism analysis after promoting `ym1` to an SSA register (carried by the `j` loop) whose use-def range does not prevent parallelization.

Similarly, the Polygeist parallelizer identifies two benchmarks with parallel reduction loops that are not contained in other parallel loops: `gramschmidt` and `durbin`. `gramschmidt` benefits from a 56× speedup with Polygeist, compared to 34× with Polly and 54× with Pluto. `durbin` sees a 6× slowdown since the new parallel loop has relatively

```
for (i=0; i<_PB_W; i++){
  ym1 = SCALAR_VAL(0.0);
  // ...
  for (j=0; j<_PB_H; j++){
    ym1 = y1[i][j];
  /*...*/ }
}
```

```
%z = constant 0.0 : f64
affine.parallel %i = ... {
  affine.for %j = ...
  iter_args(%ym1=%z)->f64 {
    %0=affine.load %y1[%i,%j]
    // ...
    affine.yield %0
}}
```

Fig. 13. Excerpt from the `deriche` benchmark. The outer loop reuses `ym1` which makes it appear non-parallel to affine schedulers (left). Polygeist detects parallelism thanks to its mem2reg optimization, reduction-like loop-carried `%ym1` value detection and late parallelization (right).

few iterations and is nested inside a sequential loop, leading to synchronization costs that outweigh the parallelism benefit. Section V-F explores the `durbin` benchmark in more detail. Polybench is a collection of codes (mostly) known to be parallel and, as such, has little need for reduction parallelization on CPU where one degree of parallelism is sufficient. When targeting inherently target architectures as GPUs, however, exploiting reduction parallelism could be vital for achieving peak performance [31], [24].

E. Case Study: Statement Splitting

We identified 5 benchmarks where the statement splitting heuristic applied: `2mm`, `3mm`, `correlation`, `covariance` and `trmm`. To assess the effect of the transformation, we executed these benchmarks with statement splitting disabled, suffixed with `-nosplit` in Figure 14. In sequential versions, `2mm` is 4.1% slower (3.13s vs 3.26s), but the other benchmarks see speedups of 25%, 50%, 51% and 27%, respectively. For parallel versions, the speedups are of 36%, 20%, 44%, 40% and -9% respectively.

Examination of polyhedral scheduler outputs demonstrates that it indeed produced the desired schedules. For example, in the `correlation` benchmark which had the statement `A[i][j] += B[k][i]*B[k][j]` Polygeist was able to find the (k, i, j) loop order after splitting. Using hardware performance counters on sequential code we confirm that the overall cache miss ratio has indeed decreased by $75\%, 50\%, 20\%, 27\%$, and -26%, respectively. However, the memory traffic estimated by the number of bus cycles has *increased* by 9% for `2mm`, and decreased by $18\%, 32\%, 32\%$, and 21% for the other benchmarks. This metric strongly correlates with the observed performance difference in the same run ($r = 0.99, p = 3 \cdot 10^{-11}$). This behavior is likely due to the scheduler producing a different fusion structure, e.g., not fusing outermost loops in `2mm`, which also affects locality. Similar results can be observed for parallel code. Further research is necessary to exploit the statement splitting opportunities, created by Polygeist, and interplay with fusion.

F. Case Study: Reduction Parallelization in `durbin`

In this benchmark, Polygeist uses its reduction optimization to create a parallel loop that other tools cannot. For the relatively small input run by default, $N = 4000$ iterations inside another sequential loop with N iterations, the overall

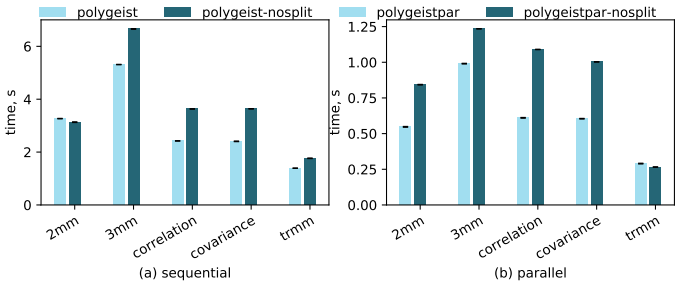

Fig. 14. Mean and 95% confidence intervals of run time across 5 runs of Polybench where statement splitting is applicable (Section III-D), lower is better. It results in faster run time (geomean $1.28\times$ sequential, $1.39\times$ parallel speedup) except for sequential `2mm` (-4%) and parallel `trmm` (-9%).

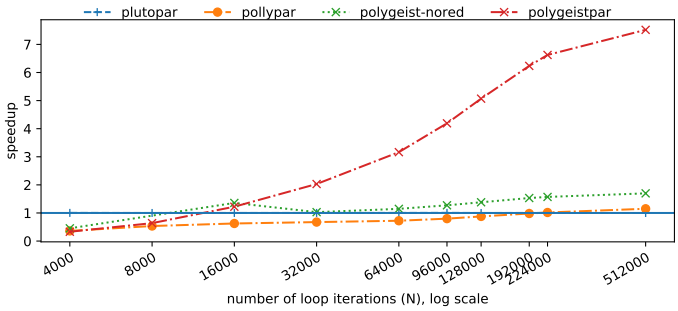

Fig. 15. Reduction parallelization allows PolygeistPar to produce larger speedups in `durbin` and at smaller sizes than PollyPar, PolygeistPar without reduction support. PlutoPar fails to parallelize, hence no speedup.

performance decreases. We hypothesize that the cost of creating parallel threads and synchronizing them outweighs the benefit of the additional parallelism and test our hypothesis by increasing N. Considering the results in Figure 15, one observes that Polygeist starts yielding speedups (> 1) for $N \geq 16000$ whereas Polly only does so at $N \geq 224000$, and to a much lesser extent: $6.62\times$ vs $1.01\times$. Without reduction parallelization, Polygeist follows the same trajectory as Polly. Pluto fails to parallelize any innermost loop and shows no speedup. This evidences in favor of our hypothesis and highlights the importance of being able to parallelize reductions.

VI. Related Work

a) MLIR Frontends: Since the adoption of MLIR under the LLVM umbrella, several frontends have been created for generating MLIR from domain-specific languages. Teckyl [2] connects the productivity-oriented Tensor Comprehensions [1] notation to MLIR's Linalg dialect. Flang—the LLVM's Fortran frontend—models Fortran-specific constructs using the FIR dialect [32]. COMET, a domain-specific compiler for chemistry, introduces an MLIR-targeting domain-specific frontend from a tensor-based language [33]. NPComp aims at providing the necessary infrastructure to compile numerical Python and PyTorch programs taking advantage of the MLIR infrastructure [34]. PET-to-MLIR converts a subset of polyhedral C code to MLIR's Affine dialect by parsing `pet`'s internal represen-

tation. In addition to currently not handling specific constructs (`if`s, 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 [35]. In contrast, Polygeist generates MLIR from non-polyhedral code (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 [17]. Stripe uses MLIR Affine dialect as a substrate for loop transformations in machine learning models, including tiling and vectorization, and accepts a custom DSL as input [36].

b) Compilers Leveraging Multiple Representations: The SUIF compiler infrastructure pioneered a combined internal representation that supports higher-level transformations, including loop optimization and parallelization [37] and, in particular, reduction parallelization [38]. Polygeist leverages MLIR abstractions unavailable in SUIF: regular and affine `for` loops, OpenMP reduction constructs, etc. It also benefits from the SSA+regions form, which is only available as external extension in SUIF [39], for IR simplification. PIPS supports loop transformations and inter-procedural optimization when targeting OpenMP [40], [41]. Polygeist differs from both by emitting machine code rather than source code, which allows it to emit parallel runtime and other directives that have no representation in the source language such as C.

c) Combining "Classical" and Polyhedral Flows: Few papers have focused on combining "classical", mostly AST-level, and polyhedral transformations. PolyAST pioneered the approach by combining an affine scheduler with AST-level heuristics for fusion and tiling [42], although similar results were demonstrated with only polyhedral transformations [43]. An analogous approach was experimented in CUDA-CHiLL [44]. Arguably, many automated polyhedral flows perform loop fusion and/or tiling as a separate step that can be assimilated to classical transformations. Pluto [11] uses several "syntactic" postprocessing passes to exploit spatial locality and parallelism in stencils [45]. Several tools have been proposed to drive polyhedral loop transformations with scripts using classical loop transformations such as fusion and permutation as operations, including URUK [46], CHiLL [47] and Clay [48]. Polygeist differs from all of these because it preserves the results of such transformations in its IR *along with* polyhedral constructs and enables interaction between different levels of abstraction.

d) Additional (Post-)Polyhedral Transformations: Support for handling reduction loops was proposed in Polly [25], but the code generation is not implemented. At the syntactic level, reduction support was added to PET via manual annotation with PENCIL directives [24]. R-Stream reportedly uses a variant of statement splitting to affect scheduler's behavior and optimize memory consumption [49]. POLYSIMD uses variable renaming around PPCG polyhedral flow to improve vectorization [50]. Polygeist automates these leveraging both SSA and polyhedral information.

e) Integration of Polyhedral Optimizers into Compilers: Polyhedral optimization passes are available in production (GCC [8], LLVM [7], IBM XL [51]) and research (R-Stream [49], ROSE [52]) compilers. In most cases, the polyhedral abstraction must be extracted from a lower-level representation before being transformed and lowered in a dedicated code generation step [13], [14]. This extraction process is not guaranteed and may fail to recover high-level information available at the source level [9]. Furthermore, common compiler optimizations such as LICM are known to interfere with it [10]. Polygeist maintains a sufficient amount of high-level information, in particular loop and n-D array structure, to circumvent these problems by design.

Source-to-source polyhedral compilers such as Pluto [11] and PPCG [5] operate on a C or C++ level. They lack interaction with other compiler optimizations and a global vision of the code, which prevents, e.g., constant propagation and inlining that could improve the results of polyhedral optimization. Being positioned between the AST and LLVM IR levels, Polygeist enables the interaction between higher- and lower-level abstractions that is otherwise reduced to compiler pragmas, i.e. mere optimization hints. Furthermore, Polygeist can rely on MLIR's progressive raising [53] to target abstractions higher level than C code with less effort than polyhedral frameworks [54].

VII. DISCUSSION

A. Limitations

a) Frontend: While Polygeist could technically accept any valid C or C++ thanks to building off Clang, it has the following limitations. Only `struct`s with values of the same type or are used within specific functions (such as `FILE` within `fprintf`) are supported due to the lack of a struct-type in high-level MLIR dialects. All functions that allocate memory must be compiled with Polygeist and not a C++ compiler to ensure that a `memref` is emitted rather than a pointer.

b) Optimizer: The limitations of the optimizer are inherited from those of the tools involved. In particular, the MLIR affine value categorization results in all-or-nothing modeling, degrading any loop to non-affine if it contains even one non-affine access or a negative step. Running Polygeist's backend on code not generated by Polygeist's frontend, which reverses loops with negative steps, is limited to loops with positive indices. 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.

c) Experiments: While our experiments clearly demonstrate the benefits of the techniques implemented in Polygeist— statement splitting and late (reduction) parallelization — non-negligible effects are due to scheduler difference: Pluto in Polygeist and `isl` in Polly. The version of Polly using Pluto[5] is not compatible with modern LLVM necessary to leverage MLIR. Connecting `isl` scheduler to Polygeist may have yielded results closer to Polly, but still not comparable

[5]http://pluto-compiler.sourceforge.net/#libpluto

more directly because of the interplay between SCoP detection, statement formation and affine scheduling.

B. Opportunities and Future Work

Connecting MLIR to existing polyhedral flows opens numerous avenues for compiler optimization research, connecting polyhedral 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. MLIR exposes similar information at a sufficiently high level to make it usable in affine transformations.

By mixing abstractions in a single module, MLIR provides finer-grain control over the entire transformation process. An extension of Polygeist can, e.g., ensure loop vectorization by directly emitting 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. Conversely, polyhedral analyses can guarantee downstream passes that, e.g., address computation never produces out-of-bounds accesses and other information.

Future work is necessary on controlling statement granularity made possible by Polygeist. Beyond affecting affine schedules, this technique enables easy rematerialization and local transposition buffers, crucial on GPUs [55], as well as software pipelining; all without having to produce C source which is known to be complex [56]. 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.

C. Alternatives

Instead of allowing polyhedral tools to parse and generate MLIR, one could emit C (or C++) code from MLIR[6] and use C-based polyhedral tools on the 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 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. `isl` uses schedule trees [57] 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 [14].

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

VIII. 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 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 optimization opportunities enabled by Polygeist considering two complementary transformations: statement splitting and reduction parallelization. In both cases, Polygeist achieves better performance than state-of-the-art polyhedral compiler and source-to-source optimizer.

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, and in part by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator 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.

Appendix

In this artifact appendix, we describe how to build Polygeist and evaluate its performance (as well as baseline compilers) on the Polybench benchmark suite. We provide two mechanisms for artifact evaluation: a Docker container[7], and a command-by-command description of the installation process, along with

[7]Script available here: https://github.com/wsmoses/Polygeist-Script/blob/main/Dockerfile

comments regarding how this may need to be modified to run on a system with hardware or software configuration that is distinct from what we used. As expected, the command description mirrors much of the content of the docker file. While a docker file is certainly more convenient and a good way of getting the compiler set up, similar changes to expectations of how many cores the system has in the evaluation will be required even with Docker.

To compile Polygeist, one must first compile several of its dependencies. We ran our experiments on an AWS c5.metal instance based on Ubuntu 20.04. We've tailored our build instructions to such a system. While many of the instructions are general and independent of machine, or OS, some steps may not be (and we describe what locations they may occur below).

```
$ sudo apt update
$ sudo apt install apt-utils
$ sudo apt install tzdata build-essential \
  libtool autoconf pkg-config flex bison \
  libgmp-dev clang-9 libclang-9-dev texinfo \
  cmake ninja-build git texlive-full numactl
# Change default compilers to make Pluto happy
$ sudo update-alternatives --install \
  /usr/bin/llvm-config llvm-config \
  /usr/bin/llvm-config-9 100
$ sudo update-alternatives --install \
  /usr/bin/FileCheck FileCheck-9 \
  /usr/bin/FileCheck 100
$ sudo update-alternatives --install \
  /usr/bin/clang clang \
  /usr/bin/clang-9 100
$ sudo update-alternatives --install \
  /usr/bin/clang++ clang++ \
  /usr/bin/clang++-9 100
```

To begin, let us download a utility repository, which will contain several scripts and other files useful for compilation and benchmarking:

```
$ cd
$ git clone \
  https://github.com/wsmoses/Polygeist-Script\
  scripts
```

One can now compile and build Pluto as shown below:

```
$ cd
$ git clone \
  https://github.com/bondhugula/pluto
$ cd pluto/
$ git checkout e5a039096547e0a3d34686295c
$ git submodule init
$ git submodule update
$ ./autogen.sh
$ ./configure
$ make -j`nproc`
```

Next one can build LLVM, MLIR, and the frontend by performing the following:

```
$ cd
$ git clone -b main-042621 --single-branch \
   https://github.com/wsmoses/Polygeist \
   mlir-clang
$ cd mlir-clang/
$ mkdir build
$ cd build/
$ cmake -G Ninja ../llvm \
 -DLLVM_ENABLE_PROJECTS="mlir;
                         polly;clang;openmp" \
 -DLLVM_BUILD_EXAMPLES=ON \
 -DLLVM_TARGETS_TO_BUILD="host" \
 -DCMAKE_BUILD_TYPE=Release \
 -DLLVM_ENABLE_ASSERTIONS=ON
$ ninja
```

From here, we need to modify `omp.h` by copying the version from the `scripts` repository and replacing the version we just built.[8]

```
$ cd
$ export OMP_FILE=`find \
  $HOME/mlir-clang/build -iname omp.h`
$ cp $HOME/scripts/omp.h $OMP_FILE
```

Let us now build the MLIR polyhedral analyses, along with the specific version of LLVM it requires. We shall begin by downloading the requisite code and building its dependencies.

```
$ cd
$ git clone --recursive \
  https://github.com/kumasento/polymer -b pact
$ cd polymer/
$ cd llvm/
$ mkdir build
$ cd build/
$ cmake ../llvm \
  -DLLVM_ENABLE_PROJECTS="llvm;clang;mlir" \
  -DLLVM_TARGETS_TO_BUILD="host" \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_INSTALL_UTILS=ON \
  -G Ninja
$ ninja -j`nproc`
$ ninja check-mlir
```

We can now build the MLIR polyhedral analyses and export the corresponding build artifacts.

```
$ cd ~/polymer
$ mkdir build
$ cd build
$ export BUILD=$PWD/../llvm/build
$ cmake .. \
  -DCMAKE_BUILD_TYPE=DEBUG \
  -DMLIR_DIR=$BUILD/lib/cmake/mlir \
  -DLLVM_DIR=$BUILD/lib/cmake/llvm \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DLLVM_EXTERNAL_LIT=$BUILD/bin/llvm-lit \
  -G Ninja
$ ninja -j`nproc`
$ export LD_LIBRARY_PATH= \
    `pwd`/pluto/lib:$LD_LIBRARY_PATH
$ ninja check-polymer
```

Finally, we are ready to begin benchmarking. We begin by running a script that disables turbo boost & hyperthreading and remaining nonessential services on the machine. The script is

[8]We modify omp.h to prevent a compilation error for Pluto parallel. The generated code does not include stdint.h, thus getting the error: unknown type name 'intptr_t'

specific to both the number of cores on the AWS instance (all cores except the non hyperthreaded cores on the first socket were disabled), as well as the image used (all nonessential services still present on the image were disabled) and thus may require modification if intending to be used on a different machine.

```
$ cd ~/scripts/
$ sudo bash ./hyper.sh
```

We can now run the benchmarking script. The script itself has assumptions about cores and layout (setting `taskset -c 1-8 numactl -i all` for example). If using a different machine, these settings may need to be tweaked as appropriate.

```
cd ~/scripts/
$ cd polybench-c-4.2.1-beta/
$ ./run.sh
# Output comes through stdout
```

The output of this script will contain the runtime of each trial, describing what compilation setting was used, as well as which benchmark was run.

REFERENCES

[1] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "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)*, vol. 16, no. 4, pp. 1–26, 2019.

[2] A. Drebes. (2020) Teckyl: An MLIR frontend for tensor operations. [Online]. Available: https://github.com/andidr/teckyl

[3] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 429–443, 2015.

[4] S. Verdoolaege, M. Kudlur, H. Kamepalli, and R. Schreiber, "Generating SIMD instructions for cerebras cs-1 using polyhedral compilation techniques," in *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[5] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[6] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, and N. Vasilache, "TC-CIM: Empowering tensor comprehensions for computing-in-memory," in *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[7] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[8] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Polyhedral analyses and optimizations for gcc," in *Proceedings of the 2006 GCC Developers Summit*, 2006, p. 2006.

[9] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, "Optimistic delinearization of parametrically sized arrays," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 351–360.

[10] M. Kruse and T. Grosser, "DeLICM: scalar dependence removal at zero memory cost," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 241–253.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, 2008.

[12] PoCC, "The polyhedral compiler collection," 2020, Online; accessed on December 2020. [Online]. Available: https://sourceforge.net/projects/pocc/

[13] C. Bastoul, "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, 2004, pp. 7–16.

[14] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral AST generation is more than scanning polyhedra," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 4, pp. 1–50, 2015.

[15] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[17] CIRCT Developers. (2020) CIRCT charter. [Online]. Available: https://github.com/llvm/circt/blob/master/docs/Charter.md

[18] MLIR Developers. (2020) MLIR affine dialect. [Online]. Available: https://mlir.llvm.org/docs/Dialects/Affine/

[19] L.-N. Pouchet and T. Yuki. Polybench/c 4.2.1. [Online]. Available: https://sourceforge.net/projects/polybench/files/

[20] C. Bastoul, "Openscop: A specification and a library for data exchange in polyhedral compilation tools," Paris-Sud University, Tech. Rep., 2011.

[21] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*. Springer, 2010, pp. 299–302.

[22] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema *et al.*, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 138–149.

[23] P. Feautrier, "Array expansion," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: Association for Computing Machinery, 1988, p. 99–111.

[24] C. Reddy, M. Kruse, and A. Cohen, "Reduction drawing: Language constructs and polyhedral compilation for reductions on gpu," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 87–97. [Online]. Available: https://doi.org/10.1145/2967938.2967950

[25] J. Doerfert, K. Streit, S. Hack, and Z. Benaissa, "Polly's polyhedral scheduling in the presence of reductions," *CoRR*, vol. abs/1505.07716, 2015. [Online]. Available: http://arxiv.org/abs/1505.07716

[26] P. Jouvelot and B. Dehbonei, "A unified semantic approach for the vectorization and parallelization of generalized reductions," in *Proceedings of the 3rd International Conference on Supercomputing*, ser. ICS '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 186–194. [Online]. Available: https://doi.org/10.1145/318789.318810

[27] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[28] C. Eisenbeis and J.-C. Sogno, "A general algorithm for data dependence analysis," in *Proceedings of the 6th International Conference on Supercomputing*, ser. ICS '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 292–302. [Online]. Available: https://doi.org/10.1145/143369.143422

[29] S. Verdoolaege and A. Cohen, "Live-range reordering," in *International Workshop on Polyhedral Compilation Techniques*, 2016.

[30] R. Gareev, T. Grosser, and M. Kruse, "High-performance generalized tensor operations: A compiler-oriented approach," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3235029

[31] R. W. Larsen and T. Henriksen, "Strategies for regular segmented reductions on GPU," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, 2017, pp. 42–52.

[32] E. Schweitz, "An MLIR dialect for high-level optimization of Fortran," in *2019 LLVM Developers Meeting*, 2019.

[33] E. Mutlu, R. Tian, B. Ren, S. Krishnamoorthy, R. Gioiosa, J. Pienaar, and G. Kestor, "Comet: A domain-specific compilation of high-performance computational chemistry," in *The 33rd Workshop on Languages and Compilers for Parallel Computing*, 2020.

[34] npcomp developers. (2020) MLIR npcomp. [Online]. Available: https://github.com/llvm/mlir-npcomp

[35] K. Komisarczyk, L. Chelini, K. Vadivel, R. Jordans, and H. Corporaal, "PET-to-MLIR: A polyhedral front-end for mlir," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2020, pp. 551–556.

[36] T. Zerrell and J. Bruestle, "Stripe: Tensor compilation via the nested polyhedral model," *CoRR*, vol. abs/1903.06498, 2019. [Online]. Available: http://arxiv.org/abs/1903.06498

[37] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam *et al.*, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM Sigplan Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[38] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[39] G. Holloway, "The machine-suif static single assignment library," *Division of Engineering and Applied Sciences, Harvard University*, 2002.

[40] M. Amini, C. Ancourt, F. Coelho, F. Irigoin, P. Jouvelot, R. Keryell, P. Villalon, B. Creusillet, and S. Guelton, "Pips is not (just) polyhedral software," in *International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Chamonix, France*, 2011.

[41] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, P. Villalon *et al.*, "Par4all: From convex array regions to heterogeneous computing," in *2nd International Workshop on Polyhedral Compilation Techniques, IMPACT (Jan 2012)*, 2012.

[42] J. Shirako, L.-N. Pouchet, and V. Sarkar, "Oil and water can mix: An integration of polyhedral and ast-based transformations," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 287–298.

[43] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen, "Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling," in *Proceedings of the 27th International Conference on Compiler Construction*, 2018, pp. 3–13.

[44] H. Zhang, A. Venkat, P. Basu, and M. Hall, "Combining polyhedral and ast transformations in chill," in *Proceedings of the Sixth International Workshop on Polyhedral Compilation Techniques, IMPACT*, vol. 16, 2016.

[45] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen, "Unified Polyhedral Modeling of Temporal and Spatial Locality," Inria Paris, Research Report RR-9110, Nov. 2017. [Online]. Available: https://hal.inria.fr/hal-01628798

[46] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, 2006.

[47] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Citeseer, Tech. Rep., 2008.

[48] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, "Opening polyhedral compiler's black box," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 128–138.

[49] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, *R-Stream Compiler*. Boston, MA: Springer US, 2011, pp. 1756–1765. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_515

[50] P. Chatarasi, J. Shirako, A. Cohen, and V. Sarkar, "A unified approach to variable renaming for enhanced vectorization," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2018, pp. 1–20.

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

[52] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel processing letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[53] L. Chelini, A. Drebes, O. Zinenko, A. Cohen, N. Vasilache, T. Grosser, and H. Corporaal, "Progressive raising in multi-level ir," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 15–26.

[54] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal, "Declarative loop tactics for domain-specific optimization," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3372266

[55] I.-J. Sung, J. Gómez-Luna, J. M. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place transposition of rectangular matrices on accelerators," *SIGPLAN Notices*, vol. 49, no. 8, p. 207–218, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2692916.2555266

[56] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[57] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule trees," in *International Workshop on Polyhedral Compilation Techniques*, 2014.