Polygeist: Raising C to Polyhedral MLIR

William S. Moses
wmoses@mit.edu

Lorenzo Chelini
l.chelini@tue.nl

Ruizhe Zhao
rz3515@ic.ac.uk

Alex Zinenko
zinenko@google.com

PACT
September 2021
The Polyhedral Model

- Makes it easy to analyze and specify program transformations best exploit the available hardware
  - Loop restructuring for spatial/temporal locality, automatic parallelization, etc.
- One of the best frameworks for optimizing compute-intensive programs like machine learning kernels or scientific simulations as well as for programming accelerators.
Polyhedral Compilation Today

Source-Based Transformation

Pluto, PPCG

Compiler-Based Transformation

Polly (LLVM)
Graphite (GCC)

Lower
Optimize
CodeGen

C
C++

LLVM

EXE
Polyhedral Compilation Today

C  
Lower  
LLVM  
Optimize  
LLVM  
CodeGen  
EXE

Source-Based Transformation
Pluto, PPCG

Compiler-Based Transformation
Polly (LLVM)
Graphite (GCC)
The MLIR Framework
Polygeist

• An MLIR-based end-to-end polyhedral compilation flow
• Keeps the best parts of high-level abstractions and low-level optimizations by directly lowering to and optimizing MLIR.
• Multiple abstraction levels allows new polyhedral optimization opportunities and simplifies the implementation of others
• State-of-the-art performance when compared with existing source and compiler-based tools.
The Polygeist Compilation Flow

- Generic C or C++ frontend that generates "standard" MLIR
- Raising transformations for transforming "standard" MLIR to polyhedral MLIR (Affine)
- Embedding of existing polyhedral tools (Pluto, CLooG) into MLIR
- Novel transformations (statement splitting, reduction detection) that rely on high-level compiler representation
- End-to-end evaluation of standard polyhedral benchmarks (Polybench)
Polygeist Frontend

- Ingests Clang AST to simplify parsing, semantic analysis, linkage, etc.
- Each C/C++ type is defined to have a corresponding MLIR. Pointers, arrays, and some structs can use MLIR’s structured pointer or memref type, preserving sizes and multi-dimensional indexing.
  - `int[12][30]` => `memref<12x30xi32>`
  - `float*` => `memref<?xf32>`
- Allocation and deallocation instructions converted to memref alloc/dealloc.
- Control flow and loops are directly lowered to structured MLIR-equivalent operations.
- Supports advanced C++ features like templates and constructors.
Simplified Frontend Example

```c
void set(int *arr, int val) {
    for(int i=0; i<10; i++){
        arr[2*i] = val;
    }
}
```

```assembly
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index
    %0 = alloca() : memref<1xmemref<?xi32>>
    store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
    %1 = alloca() : memref<1xi32>
    store %arg1, %1[%c0] : memref<1xi32>
    %c0_i32 = constant 0 : i32
    %c2_i32 = constant 2 : i32
    %c10_i32 = constant 10 : i32
    %2 = index_cast %c10_i32 : i32 to index
    scf.for %arg2 = %c0_i32 to %2 {
        %3 = index_cast %arg2 : index to i32
        %4 = alloca() : memref<1xi32>
        store %3, %4[%c0] : memref<1xi32>
        %5 = load %0[%c0] : memref<1xmemref<?xi32>>
        %6 = load %4[%c0] : memref<1xi32>
        %7 = muli %c2_i32, %6 : i32
        %8 = index_cast %7 : i32 to index
        %9 = load %1[%c0] : memref<1xi32>
        store %9, %5[%8] : memref<?xi32>
    }
    return
}
```
Polygeist Raising

• Local variables eliminated by new MLIR mem2reg pass
• Canonicalizations run to simplify the code, including while => for
• Raising an operation (for, if, load, store) to its affine-variant:
  • Detect if index calculation is a valid affine expression
  • Progressively fold index calculation into a new replacement affine operation

```
%off = muli %c2, %idx : index
store %val, %ptr[%off] : memref<?xi32>
```

```
affine.store %val, %ptr[2 * %idx] : memref<?xi32>
```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index
    %0 = alloca() : memref<1xmemref<?xi32>>
    store %arg0, %0[0] : memref<1xmemref<?xi32>>
    %1 = alloca() : memref<1xi32>
    store %arg1, %1[0] : memref<1xi32>
    %c0_i32 = constant 0 : i32
    %c10_i32 = constant 10 : i32
    %2 = index_cast %c10_i32 : i32 to index
    scf.for %arg2 = %c0_i32 to %2 {
        %3 = index_cast %arg2 : index to i32
        %4 = alloca() : memref<1xi32>
        store %3, %4[0] : memref<1xi32>
        %5 = load %0[0] : memref<1xmemref<?xi32>>
        %c2_i32 = constant 2 : i32
        %6 = load %4[0] : memref<1xi32>
        %7 = mul %c2_i32, %6 : i32
        %8 = index_cast %7 : i32 to index
        %9 = load %1[0] : memref<1xi32>
        store %9, %5[0] : memref<?xi32>
    }
    return
}
Polygeist Raising

```swift
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index
    %0 = alloca() : memref<1xmemref<?xi32>>
    store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
    %1 = alloca() : memref<1xi32>
    store %arg1, %1[%c0] : memref<1xi32>
    %c0_i32 = constant 0 : i32
    %c10_i32 = constant 10 : i32
    %2 = index_cast %c10_i32 : i32 to index
    scf.for %arg2 = %c0_i32 to %2 {
        %3 = index_cast %arg2 : index to i32
        %4 = alloca() : memref<1xi32>
        store %3, %4[%c0] : memref<1xi32>
        %5 = load %0[%c0] : memref<1xmemref<?xi32>>
        %c2_i32 = constant 2 : i32
        %7 = muli %c2_i32, %3 : i32
        %8 = index_cast %7 : i32 to index
        store %arg1, %arg0[%8] : memref<?xi32>
    }
    return
}
```

1. Mem2Reg
# Polygeist Raising

```swift
func @set(%arg0: memref<?xi32>, %arg1: i32) {
  %c0 = constant 0 : index
  %0 = alloca() : memref<1xmemref<?xi32>>
  store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
  %1 = alloca() : memref<1xi32>
  store %arg1, %1[%c0] : memref<1xi32>
  %c0_i32 = constant 0 : i32
  %c10_i32 = constant 10 : i32
  %2 = index_cast %c10_i32 : i32 to index
  scf.for %arg2 = %c0_i32 to %2 {
    %3 = index_cast %arg2 : index to i32
    %4 = alloca() : memref<1xi32>
    store %3, %4[%c0] : memref<1xi32>
    %5 = load %0[%c0] : memref<1xmemref<?xi32>>
    %c2_i32 = constant 2 : i32
    %6 = load %4[%c0] : memref<1xi32>
    %7 = muli %c2_i32, %6 : i32
    %8 = index_cast %7 : i32 to index
    %9 = load %1[%c0] : memref<1xi32>
    store %9, %5[%8] : memref<?xi32>
  }
  return
}
```

1. **Mem2Reg**
2. **Canonicalize**
Polygeist Raising

```swift
func @set(%arg0: memref<?xi32>, %arg1: i32) {

    %c0 = constant 0 : index
    %0 = alloca() : memref<1xmemref<?xi32>>
    store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
    %1 = alloca() : memref<1xi32>
    store %arg1, %1[%c0] : memref<1xi32>
    %c0_i32 = constant 0 : i32
    %c10_i32 = constant 10 : i32
    %2 = index_cast %c10_i32 : i32 to index
    scf.for %arg2 = %c0_i32 to %2 {
        %3 = index_cast %arg2 : index to i32
        %4 = alloca() : memref<1xi32>
        store %3, %4[%c0] : memref<1xi32>
        %5 = load %0[%c0] : memref<1xmemref<?xi32>>
        %c2_i32 = constant 2 : i32
        %6 = load %4[%c0] : memref<1xi32>
        %7 = muli %c2_i32, %6 : i32
        %8 = index_cast %7 : i32 to index
        %9 = load %1[%c0] : memref<1xi32>
        store %9, %5[%8] : memref<?xi32>
    }
    return
}
```

1. Mem2Reg
2. Canonicalize
3. Raise to Affine
Connecting MLIR to Polyhedral Tools

• Polygeist produces MLIR **Affine**
• MLIR **Affine** <-> polyhedral model
• Existing tools don't take **Affine**
• **OpenScop** is a target representation
• How to translate **Affine** <-> **OpenScop**?
Polyhedral Optimization Pipeline

- Polygeist
  - raise-affine

- MLIR Affine
  - reg2mem
  - resolve-hazards
  - extract-polystmt

- OpenScop
  - pluto-opt

- MLIR Affine
  - MLIR
  - lower-affine
Polyhedral Statement

- **OpenScop** expects **C-like** statements:
  
  \[ C[i][j] += A[i][k] \times B[k][j] \]

- **MLIR Affine** is at a lower level.
- To match C-like statements:
  - **Extract** 1 MLIR memory write
  - **Traverse** SSA use-def chains
  - **Gather** until loads or symbols

```
affine.for %i = 0 to %N

%0 = affine.load %A[%i]
%1 = affine.load %B[%i]
%2 = mulf %0, %1
%3 = affine.load %C[%i]
%4 = addf %2, %3
affine.store %4, %C[%i]
```

```
affine.for %i = 0 to %N

call @S0(%A, %B, %C, %i)
```
Region-Spanning Problem

- A use-def chain spans across loops
- Statement nesting is ambiguous
- MLIR reconstruction is difficult
- Reg2mem pass: insert a scratchpad for each region-spanning use-def chain
Avoid RAW Hazard

- Two stores share the same load
- 1st store overwrite the address
- Detected by value analysis
- Solution: insert scratchpads
Evaluation

• Compare Polygeist frontend with Clang to establish a fair baseline

• Compare Polygeist polyhedral optimization with Pluto (source-based) and Polly (compiler-based)

• Novel optimizations
Serial Non-Polyhedral Comparison
Frontend within 0.32% of “standard” frontend
Serial Non-Polyhedral Comparison

Frontend within 0.32% of “standard” frontend

Remaining gap attributed to small tests where minor assembly differences matter
Polygeist: 2.53x speedup
Pluto: 2.34x speedup
Polly: 1.41x speedup
Polygeist: 2.53x speedup
Pluto: 2.34x speedup
Polly: 1.41x speedup

Big gaps come from different schedules
Sequential Polyhedral Comparison

Polygeist: 2.53x speedup
Pluto: 2.34x speedup
Polly: 1.41x speedup

24% Polygeist speedup comes from better integer operations
Sequential Polyhedral Comparison

Polygeist: 2.53x speedup
Pluto: 2.34x speedup
Polly: 1.41x speedup

Polygeist slower due to smaller branch-simplification timeout
Parallel Polyhedral Comparison

Polygeist: 9.47x speedup
Pluto: 7.54x speedup
Polly: 3.26x speedup
Parallel Polyhedral Comparison

Polygeist: 9.47x speedup
Pluto: 7.54x speedup
Polly: 3.26x speedup

Polygeist and Polly benefit from SSA optimizations
Parallel Polyhedral Comparison

Polygeist: 9.47x speedup
Pluto: 7.54x speedup
Polly: 3.26x speedup

Polygeist can remove loop-carried dependency and parallelize when others cannot
Polygeist: 9.47x speedup
Pluto: 7.54x speedup
Polly: 3.26x speedup

Polygeist can detect parallel reductions
Parallel Reduction Detection (durbin)
Statement Splitting

• We don’t need to reconstruct the original C statements from Affine
• We can split statements by inserting scratchpads.

\[
\begin{align*}
\text{for}(i=0; i<\text{NI}; i++) & \quad \text{double } M[\text{NK}]; \\
\text{for}(j=0; j<\text{NJ}; j++) & \quad \text{for}(k=0; k<NK; k++) \\
& \quad \text{for}(k=0; k<\text{NK}; k++) \\
S: & \quad A[i][j] += f(B[k][i], C[k][j]); \\
\text{↓} & \\
\text{for}(i=0; i<\text{NI}; i++) & \quad \text{double } M[\text{NK}]; \\
\text{for}(j=0; j<\text{NJ}; j++) & \quad \text{for}(k=0; k<NK; k++) \\
\quad & \quad \text{for}(k=0; k<\text{NK}; k++) \\
S: & \quad M[k] = f(B[k][i], C[k][j]); \\
T: & \quad A[i][j] += M[k];
\end{align*}
\]
Statement Splitting

(a) sequential

(b) parallel

Lower is better
Conclusion

• Polygeist is an end-to-end polyhedral compiler that combines the best parts of source-level and compiler-level polyhedral tools
  • C/C++ frontend for MLIR
  • Transformations for raising to Affine
  • Integrating existing polyhedral tools with MLIR

• Polygeist provides an easy platform to introduce novel polyhedral optimizations (statement splitting, reduction detection) that are difficult to perform on existing representations

• Polygeist outperforms existing Polyhedral optimizers for both sequential and parallel code generation

• Open Sourced on Github, see https://polygeist.mit.edu
Acknowledgements

• Thanks to Valentin Churavy, Albert Cohen, Henk Corporaal, Tobias Grosser, and Charles Leiserson for thoughtful discussions on this work.

• William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship, in part by Los Alamos National Laboratories, and in part by the United States Air Force Research Laboratory.

• Lorenzo Chelini is partially supported by the European Commission Horizon 2020

• Ruizhe Zhao is sponsored by UKRI and Corerain Technologies Ltd. The support of the UK EPSRC is also gratefully acknowledged.
Conclusion

- Polygeist is an end-to-end polyhedral compiler that combines the best parts of source-level and compiler-level polyhedral tools
  - C/C++ frontend for MLIR
  - Transformations for raising to Affine
  - Integrating existing polyhedral tools with MLIR
- Polygeist provides an easy platform to introduce novel polyhedral optimizations (statement splitting, reduction detection) that are difficult to perform on existing representations
- Polygeist outperforms existing Polyhedral optimizers for both sequential and parallel code generation
- Open Sourced on Github, see https://polygeist.mit.edu
Backup slides
Future Work

• Exploration of Statement Splitting beyond a simple heuristic
• GPU optimization and GPU <-> CPU
• Embedded DSL / C-style semantics for directly generating MLIR Ops
• Upstreaming LLVM Incubator Project
Frontend Performance Differences

- 8% performance boost on Floyd-Warshall occurs if Polygeist generates a single MLIR module for both benchmarking and timing code by default
- MLIR doesn't properly generate LLVM datalayout, preventing vectorization for MLIR-generated code (patched in our lowering)
- Different choice of allocation function can make a 30% impact on some tests (adi)
- LLVM strength-reduction is fragile and sometimes misses reversed loop induction variable (remaining gap in adi)
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    affine.for %arg2 = 0 to 10 {
        affine.store %arg1, %arg0[%arg2 * 2] : memref<?xi32>
    }
    return
}
Conclusion

• Polygeist providing tools to fairly compare MLIR-based polyhedral representations with prior art in Polyhedral representations
  • C/C++ frontend for (Affine) MLIR
  • Integration of existing polyhedral tools for transforming MLIR (via OpenScop)
  • End-to-end comparison using existing Polyhedral benchmarks (Polybench)
• Polygeist enables future research on polyhedral MLIR transformations
• MLIR-based frontend differs from Clang by 1.25%
• @Ruizhe, add a good polymer conclusion
Translate to OpenScop

• First pre-process MLIR Affine code by previous passes
• For each extracted polyhedral statement:
  • Domain: get constraints from affine.for/if
  • Initial Schedule: derive from region nesting and operation order
  • Access: extract from affine load/stores
• Store symbols in OpenScop extensions
Translate to OpenScop

affine.for %i = 0 to %N

affine.for %j = 0 to %N

call @S0(%A, %i, %j )

func @S0(%A: memref<%xf32>, %i: index, %j: index) {
  %o = affine.load %A[%i, %j]
  %l = mulf %o, %o
  affine.store %l, %A[%i, %j]
  return
}
Regenerate MLIR Code

• Obtain a CLooG AST from an optimized OpenScop representation
• Regenerate MLIR code by traversing AST
• OpenScop symbols will be translated to MLIR values or operations based on a maintained symbol table.
Motivation

- The compiler research has recently been enamored by the MLIR framework, whose first-class polyhedral representation may provide benefits on a variety of codes.
- We can fully leverage decades of polyhedral research by connecting MLIR with existing polyhedral tools.
- Without MLIR-versions of standard polyhedral benchmarks, one cannot perform a fair assessment.
- Goal of this work is to provide a fair baseline for subsequent work AND explore the potential of polyhedral optimizations that require both high level and low level information.
Outlining

• Outline statements into functions
• Function interfaces:
  • Memory to access
  • Lifted stack allocated symbols

```rust
func @S0(%A: memref<!xf32>) {%
  %c0 = constant 0 : index
  %s0 = dim %A, %c0 : index

  %l = affine.load %A[0]
  affine.store %l, %A[symbol(%s0) - 1]
  return
}
```

Lift local symbols to the function interface

```rust
func @S0(%A: memref<!xf32>, %s0: index ) {%
  %0 = affine.load %A[0]
  affine.store %0, %A[%s0 - 1]
  return
}
```
Polygeist Raising

• Select statements must be represented by a C ternary operator
  • C ternaries have lazy-evaluation semantics which are replicated in the generated MLIR
  • Mem2Reg and code motion attempt to remove unnecessary loads within if's to generate a valid select.

\[
\text{prefixMax}[i] = \begin{cases} 
\text{prefixMax}[i-1] & (\text{prefixMax}[i-1] \geq \text{data}[i]) \\
\text{data}[i] & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
%0 &= \text{index\_cast} \ %\text{arg2} : i32 \ \text{to} \ \text{index} \\
%1 &= \text{subi} \ %0, \ %c1 : \text{index} \\
%2 &= \text{load} \ %\text{arg0}[%1] : \text{memref}\langle?xi32\rangle \\
%3 &= \text{load} \ %\text{arg1}[%0] : \text{memref}\langle?xi32\rangle \\
%4 &= \text{cmpi} \ "\text{sgt}" , \ %2, \ %3 : i32 \\
%5 &= \text{scf\_if} \ %4 \ \Rightarrow \ (i32) \{ \\
& \ \ \ %6 = \text{load} \ %\text{arg0}[%1] : \text{memref}\langle?xi32\rangle \\
& \ \ \ \text{scf\_yield} \ %6 : i32 \\
\} \ \text{else} \{ \\
& \ %6 = \text{load} \ %\text{arg1}[%0] : \text{memref}\langle?xi32\rangle \\
& \ \text{scf\_yield} \ %6 : i32 \\
\} \\
\text{store} \ %5, \ %\text{arg0}[%0] : \text{memref}\langle?xi32\rangle
\end{align*}
\]
The Affine dialect

- Represent SCoP with polyhedral-friendly loops and conditions
- Core Affine representation
  - Symbols - parameters
  - Dimensions - symbol extension that accepts induction variables
  - Maps - multi-dimensional function of symbols and dimensions
  - Sets - integer tuples constrained by a conjunction

```c
%c0 = constant 0 : index
%c0 = 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>
  }
}
```
Polygeist Raising

```mlir
func @set(%arg0: memref<?xi32>, %arg1: i32) {
  %c0 = constant 0 : index
  %0 = alloca() : memref<1xmemref<?xi32>>
  store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
  %1 = alloca() : memref<1xi32>
  store %arg1, %1[%c0] : memref<1xi32>
  %c0_i32 = constant 0 : i32
  %c10_i32 = constant 10 : i32
  %2 = index_cast %c10_i32 : i32 to index
  scf.for %arg2 = %c0_i32 to %2 {
    %3 = index_cast %arg2 : index to i32
    %4 = alloca() : memref<1xi32>
    store %3, %4[%c0] : memref<1xi32>
    %5 = load %0[%c0] : memref<1xmemref<?xi32>>
    %c2_i32 = constant 2 : i32
    %6 = load %4[%c0] : memref<1xi32>
    %7 = muli %c2_i32, %6 : i32
    %8 = index_cast %7 : i32 to index
    %9 = load %1[%c0] : memref<1xi32>
    store %9, %5[%8] : memref<?xi32>
  }
  return
}
```

```mlir
func @set(%arg0: memref<?xi32>, %arg1: i32) {
  affine.for %arg2 = 0 to 10 {
    affine.store %arg1, %arg0[%arg2 * 2] : memref<?xi32>
  }
  return
}
```
Polyhedral Performance Differences

• Polly differs from other two as it uses a different scheduler
• Even when using the same scheduler, Polygeist can select a different statement set and thus schedule coming from partially optimized SSA rather than the original C.
• Pluto executes significantly more (~10^11) more integer instructions on seidel-2d than Polygeist, which is ~59s at 3GHz, accounting for the gap. Can be caused by different integer optimization and the use of a proper machine type/bound simplification.
• For jacobi-2d, Polygeist performs worse, stopping earlier when simplifying (75 statement copies in 40 branches), whereas Clang by default takes longer to process this but has better end vectorization.
Sequential Polyhedral Comparison

Polygeist: 2.53x speedup
Pluto: 2.34x speedup
Polly: 1.41x speedup

Big gaps come from different schedules
Parallel Performance Differences

• Same scheduling differences as sequential (Cholesky and LU are better on Pluto/Polygeist than Polly; Gemver and MVT are better on Polly)
• Ludcmp and syr(2)k benefit from SSA optimizations
• Polygeist is only framework that can parallelize deriche (6.9x) and symm (7.7x) by analyzing and removing the loop-carried dependency
• Polygeist identifies a parallel reduction within gram schmidt (56x Polygeist, 54x Pluto, 34x Polly) and durbin (6x slowdown as few iterations)