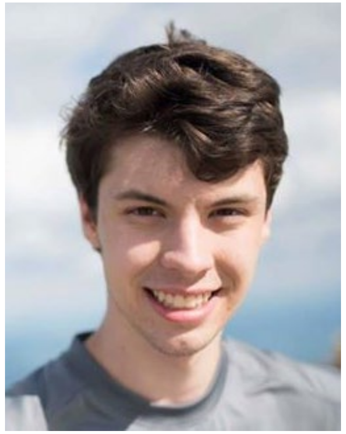
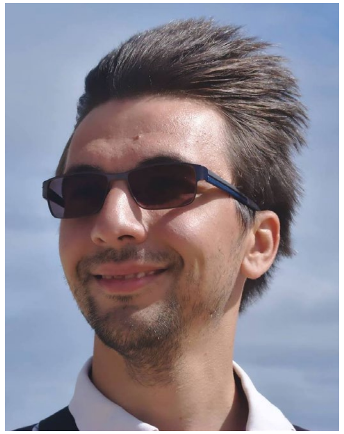
Polygeist: Raising C to Polyhedral MLIR


William S. Moses
wmoses@mit.edu


Lorenzo Chelini
l.chelini@tue.nl

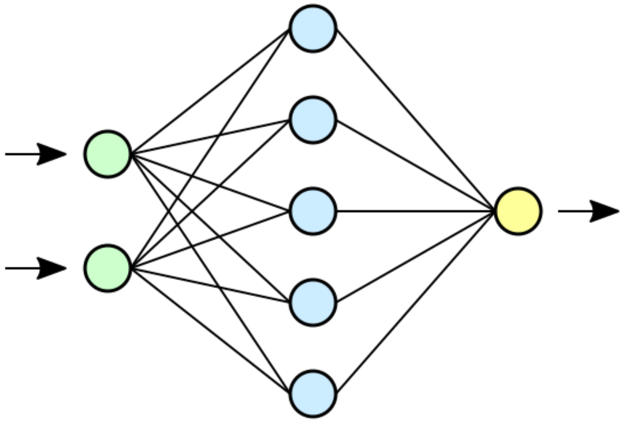

Ruizhe Zhao
rz3515@ic.ac.uk


Alex Zinenko
zinenko@google.com

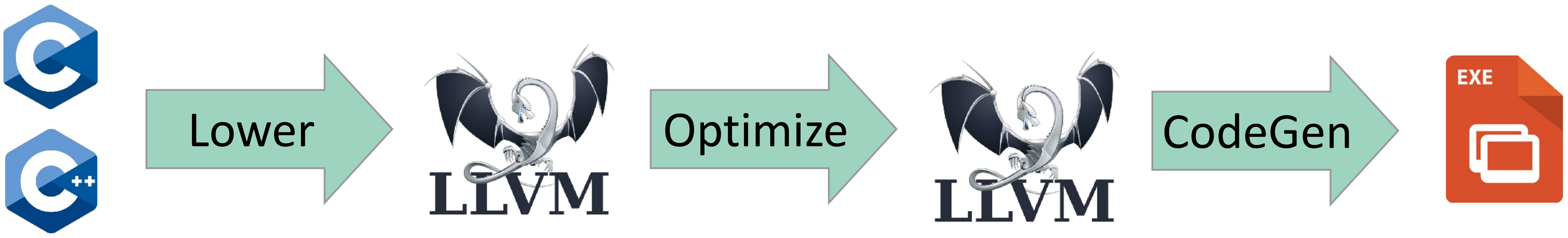
PACT
September 2021

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



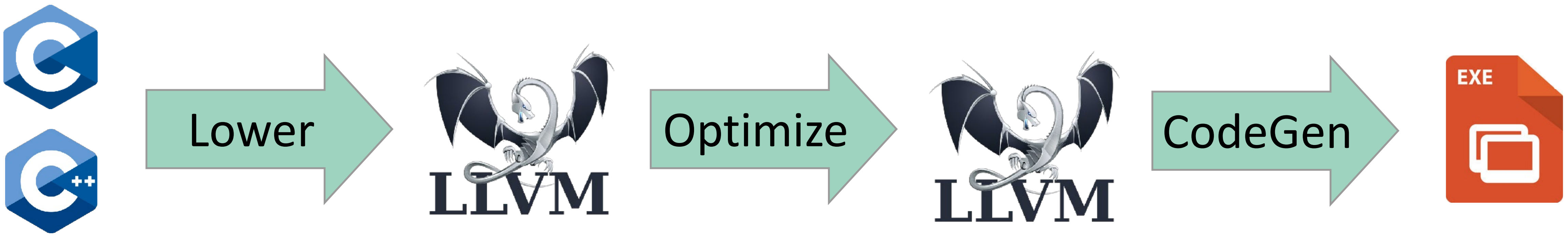
Source-Based
Transformation

Pluto, PPCG

Compiler-Based
Transformation

Polly (LLVM)
Graphite (GCC)

Polyhedral Compilation Today



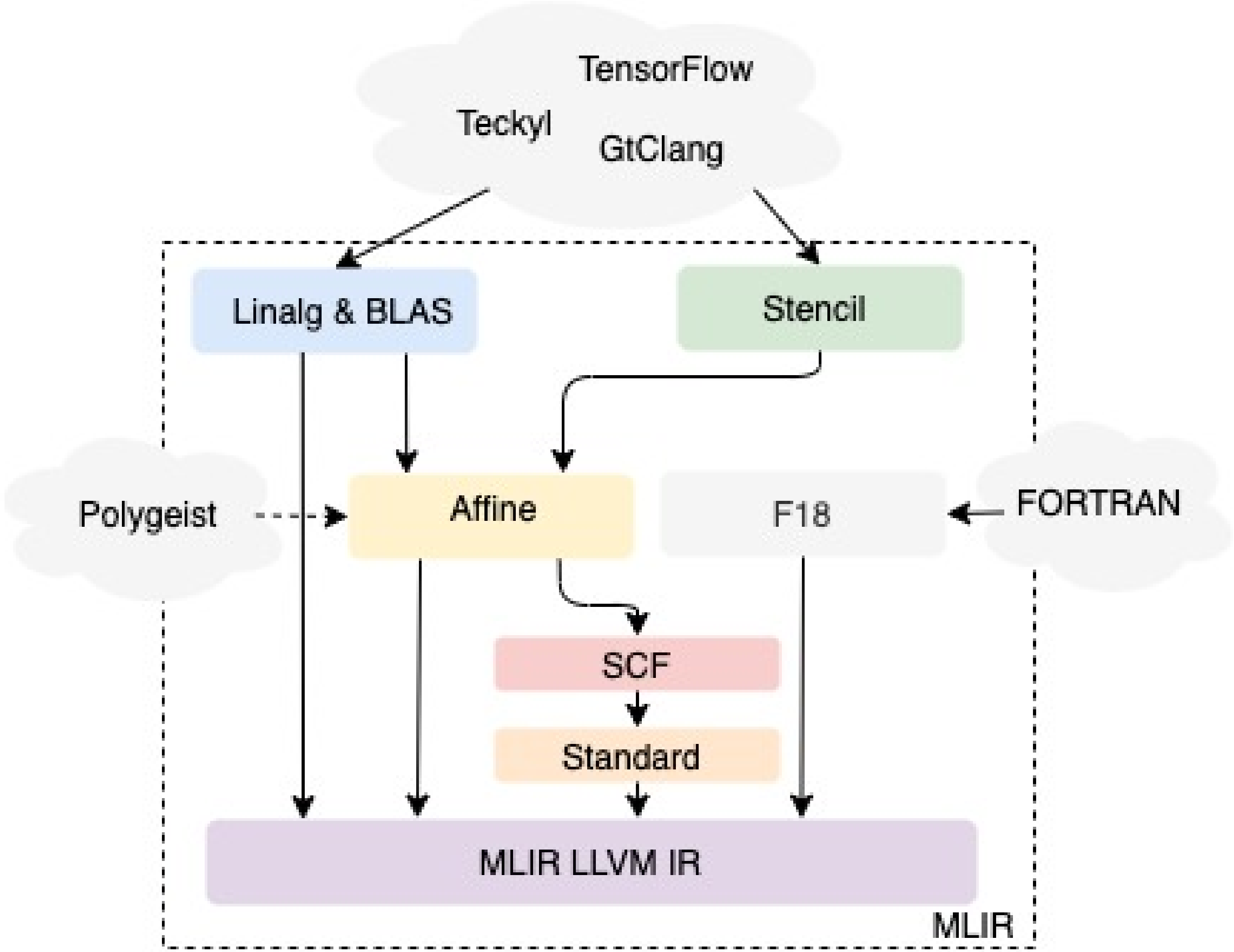
Source-Based Transformation

Pluto, PPCG

Compiler-Based Transformation

Polly (LLVM)
Graphite (GCC)

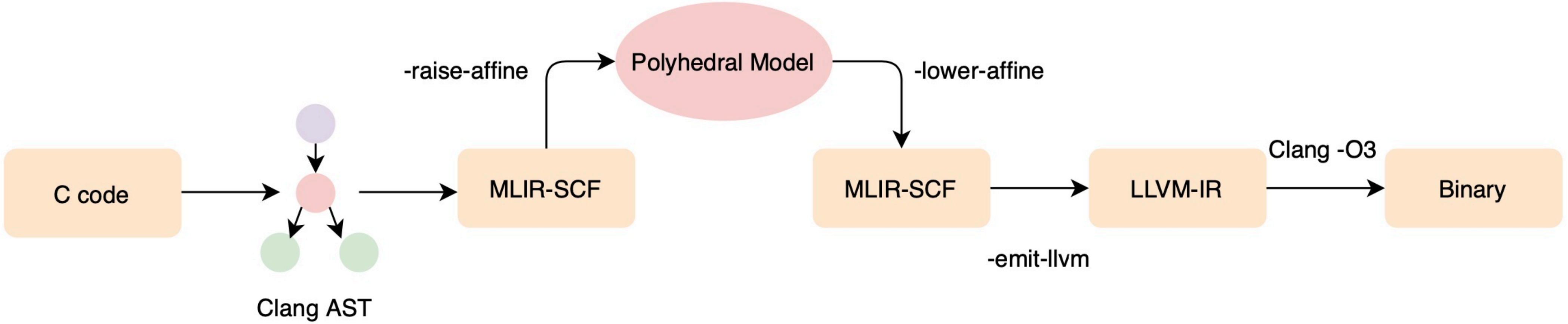
The MLIR Framework



Polygeist

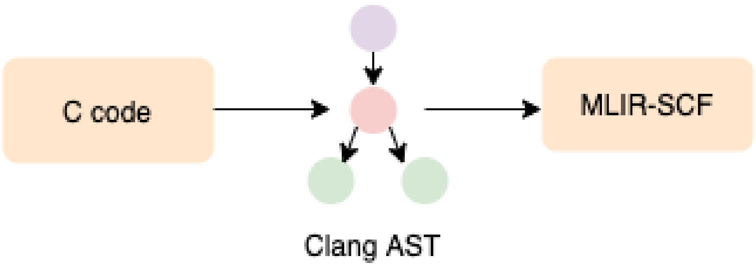
- An MLIR-based end-to-end polyhedral compilation flow
- Keeps the best parts of high-level abstractions and low-level optimizations by directly lowering to and optimizing MLIR.
- Multiple abstraction levels allows new polyhedral optimization opportunities and simplifies the implementation of others
- 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>
  ```

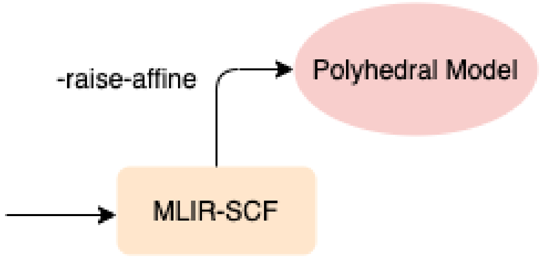
- Allocation and deallocation instructions converted to memref alloc/dealloc.
- Control flow and loops are directly lowered to structured MLIR-equivalent operations.
- Supports advanced C++ features like templates and constructors.

Simplified Frontend Example

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

```
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

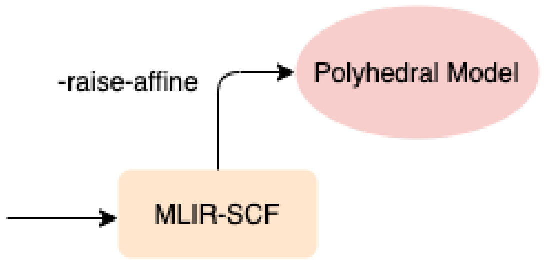


- Local variables eliminated by new MLIR mem2reg pass
- Canonicalizations run to simplify the code, including while => for
- Raising an operation (for, if, load, store) to its affine-variant:
  - Detect if index calculation is a valid affine expression
  - Progressively fold index calculation into a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

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

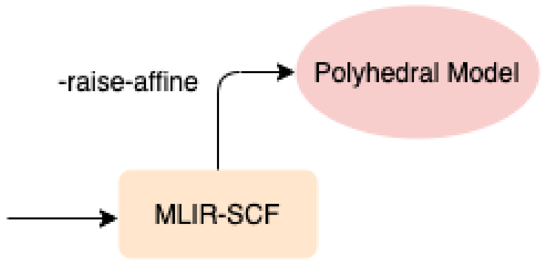
```
affine.store %val, %ptr[2 * %idx] : memref<?xi32>
```

Polygeist Raising




```
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
  }
```

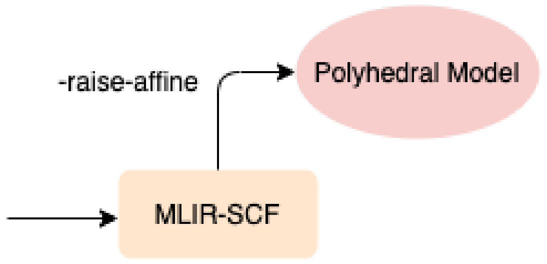
Polygeist Raising




```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index




    %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



      %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

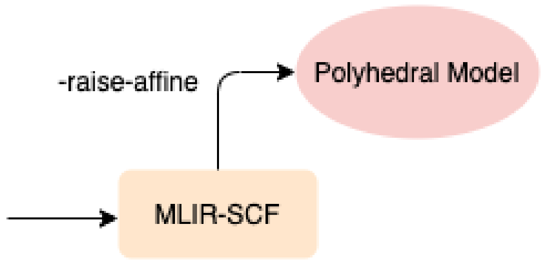
Polygeist Raising




```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index
    %c2 = constant 2 : i32
    %c10 = constant 10 : i32






    scf.for %arg2 = %c0 to %c10 {






      %7 = muli %c2_i32, %arg2 : index


      store %arg1, %arg0[%7] : memref<?xi32>
    }
    return
  }
```

1. Mem2Reg
2. Canonicalize

Polygeist Raising

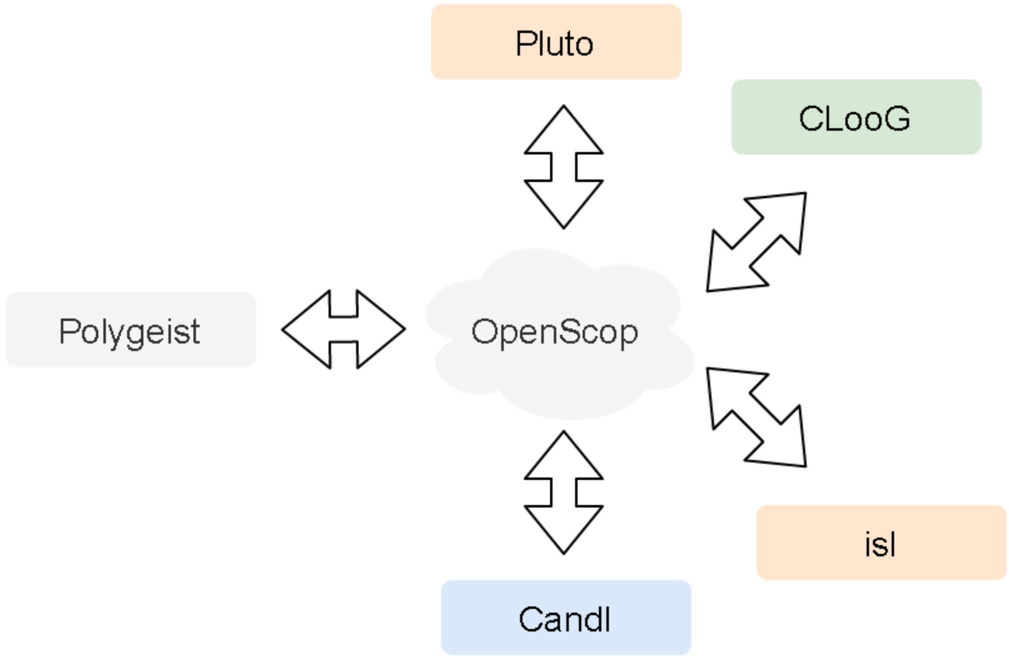



```
func @set(%arg0: memref<?xi32>, %arg1: i32) {








    affine.for %arg2 = 0 to 10 {









      affine.store %arg1, %arg0 [2 * %arg2] :
                                  memref<?xi32>
    }
    return
  }
```

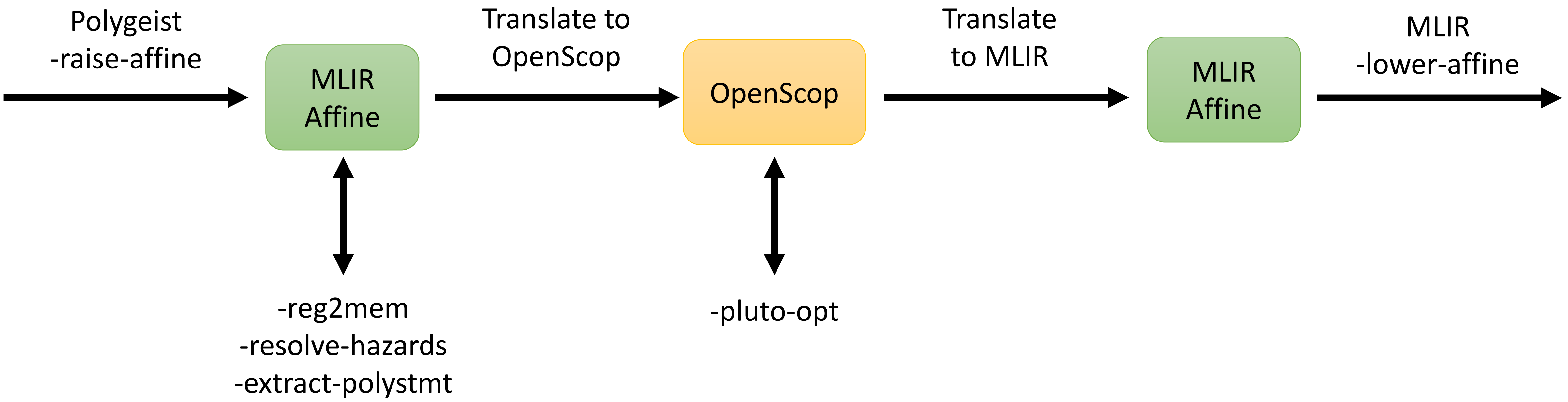
1. Mem2Reg
2. Canonicalize
3. Raise to Affine

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

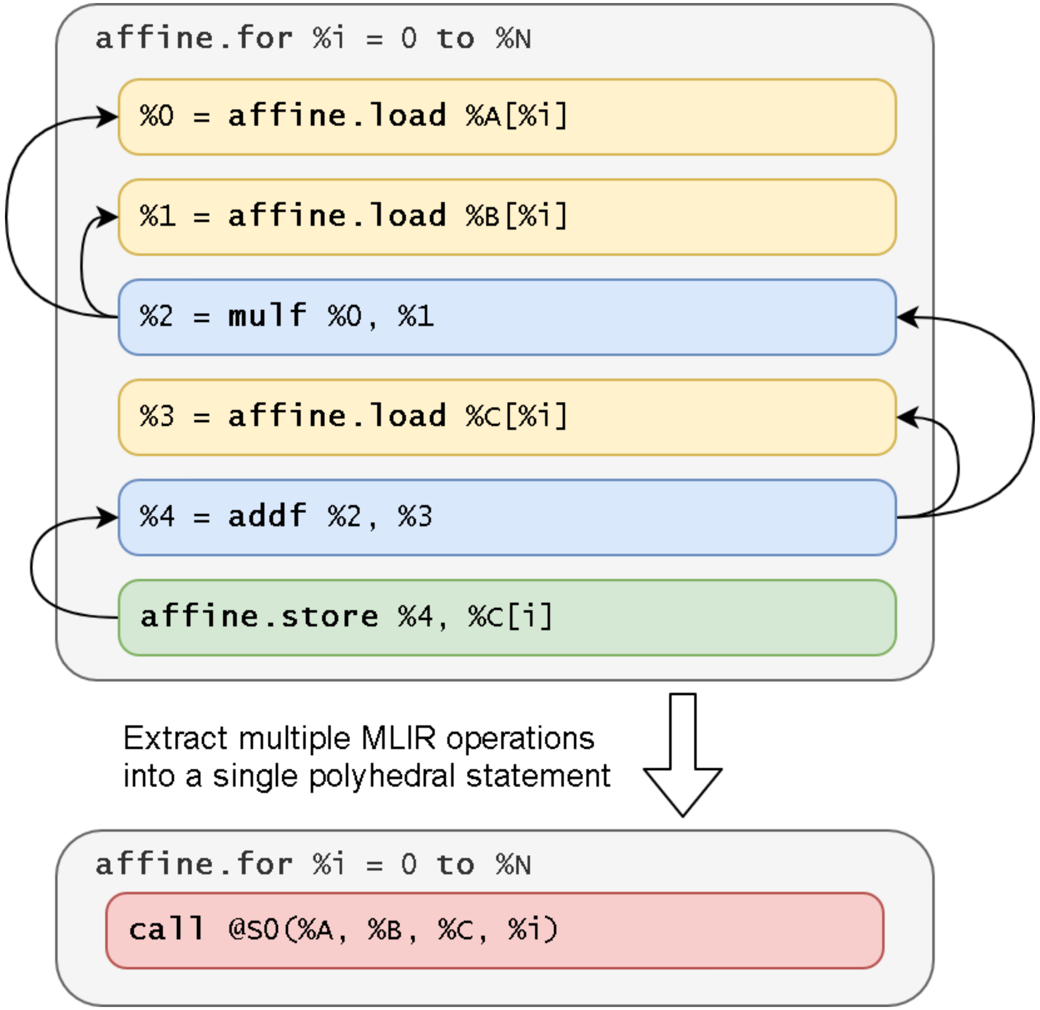


Polyhedral Statement

- **OpenScop** expects **C-like** statements:

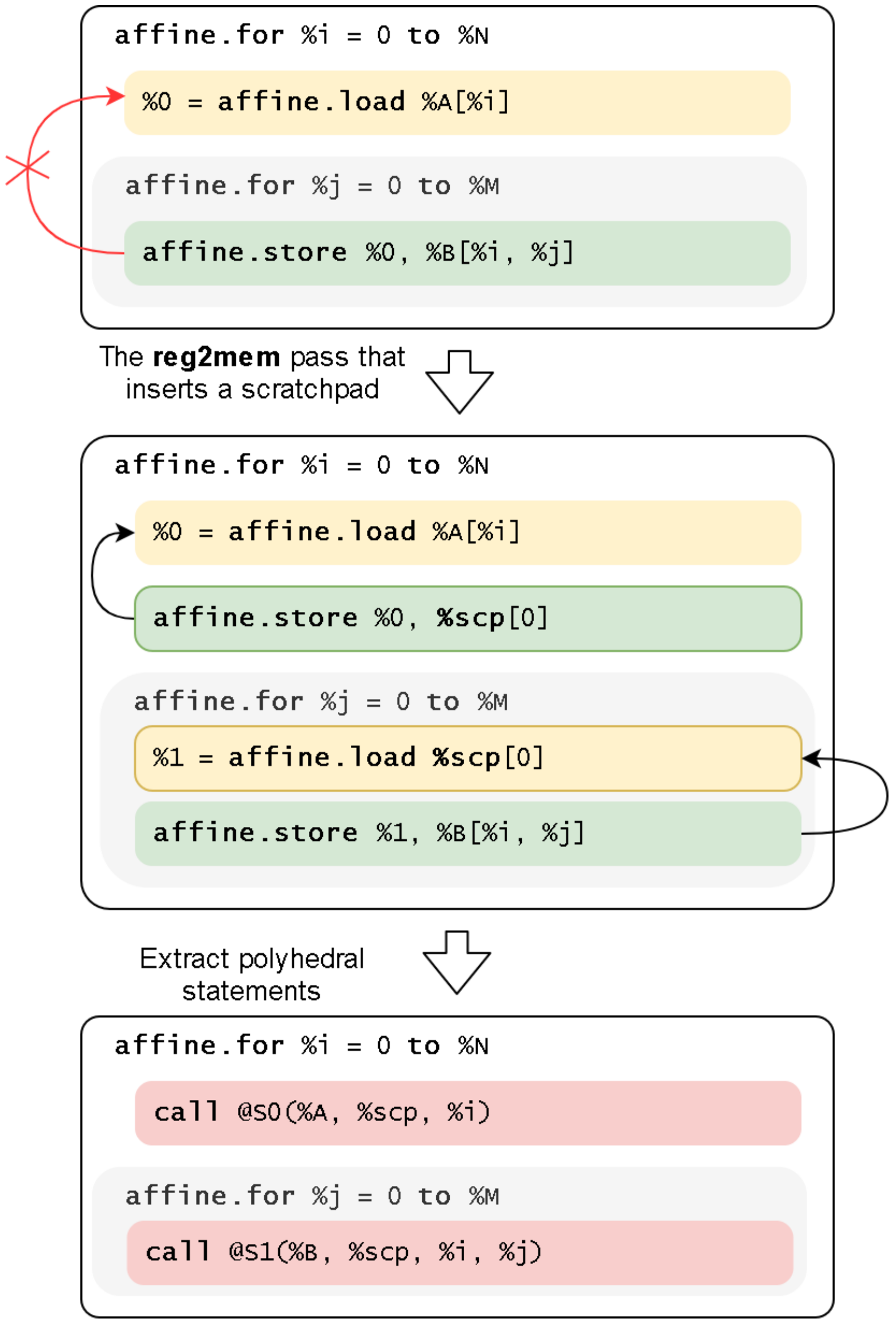
```
C[i][j] += A[i][k] * B[k][j]
```

- 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



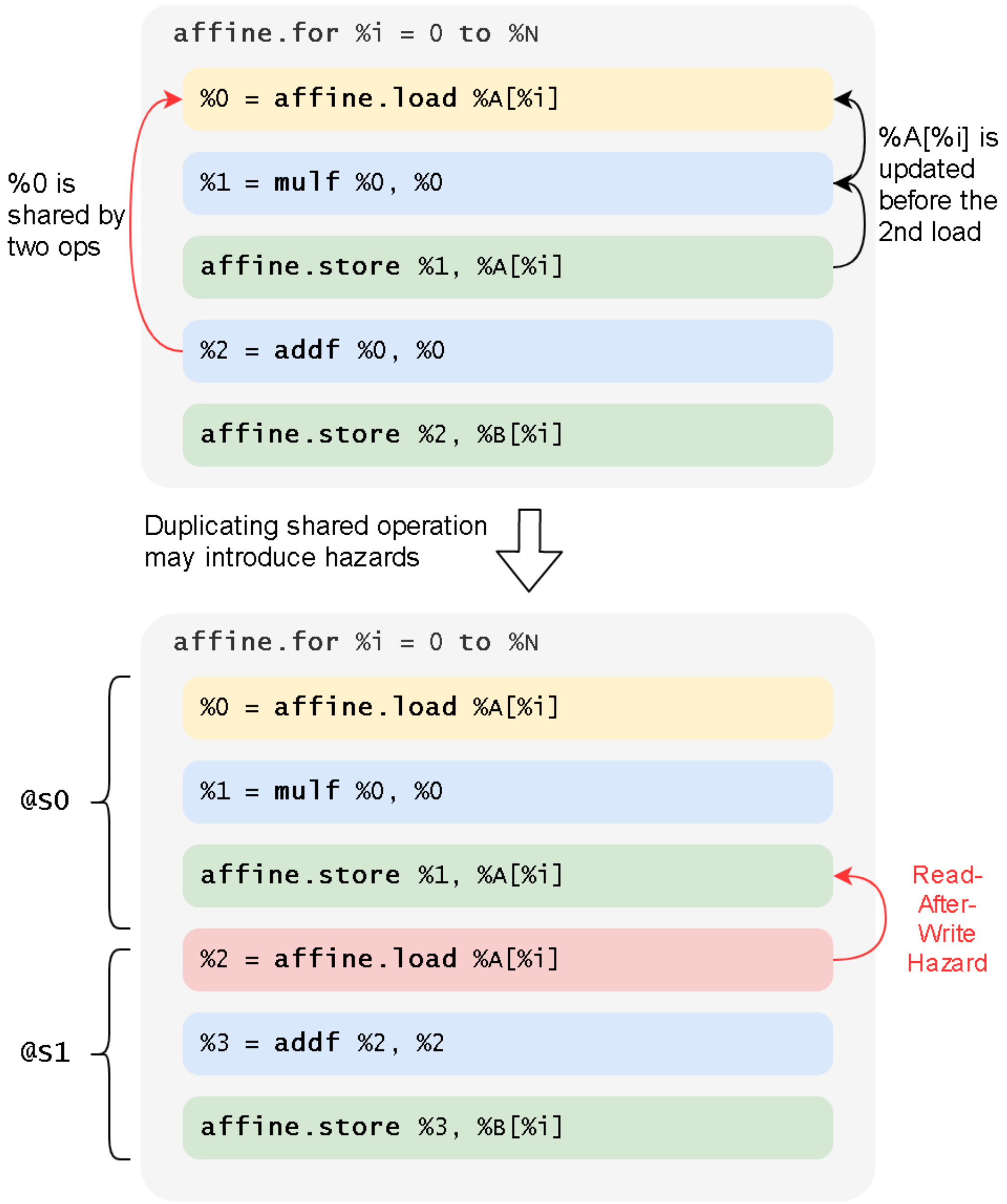
Region-Spanning Problem

- 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

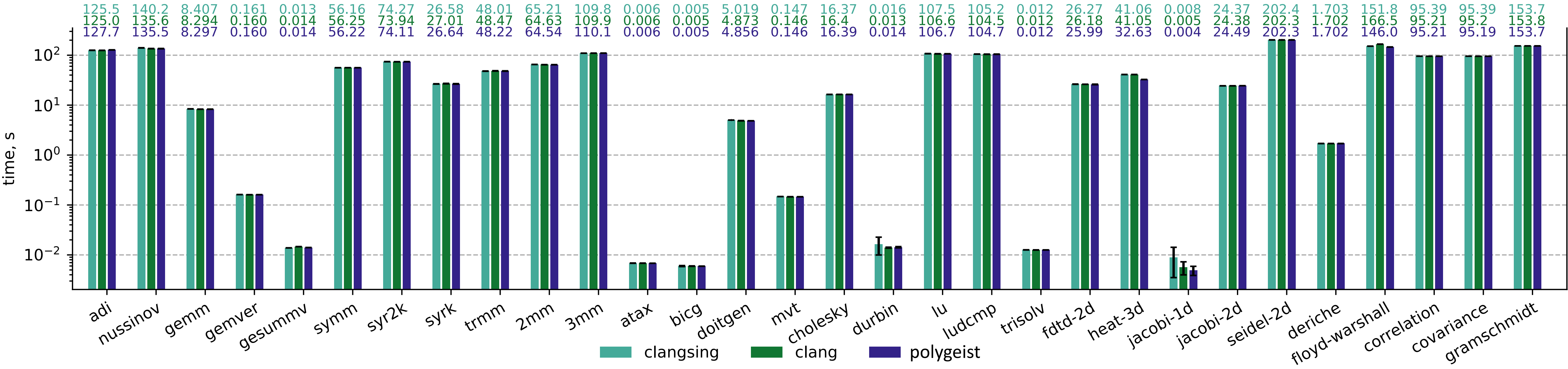
- Two stores **share** the same load
- 1st store **overwrite** the address
- Detected by **value analysis**
- **Solution**: insert scratchpads



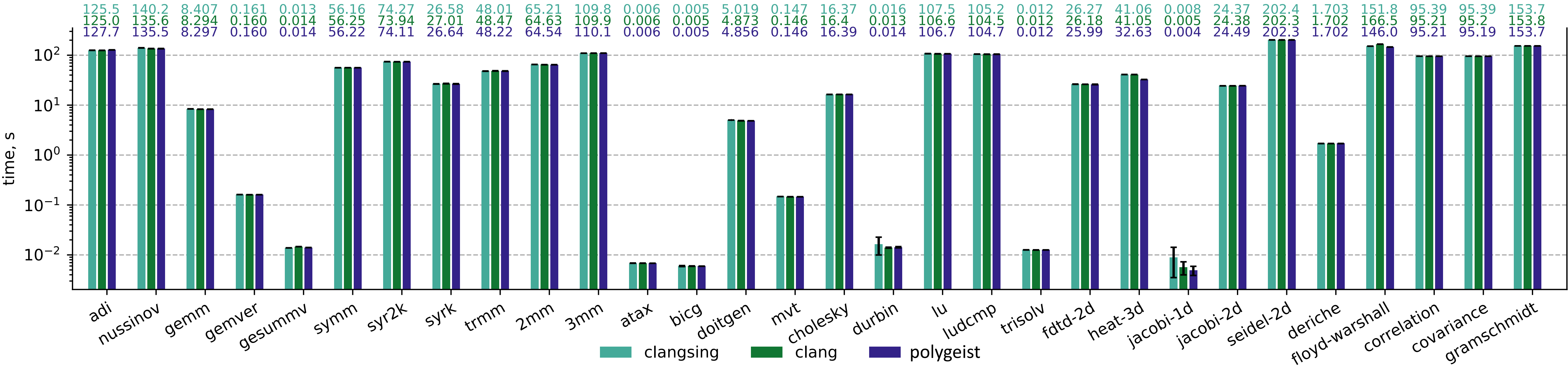
Evaluation

- 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

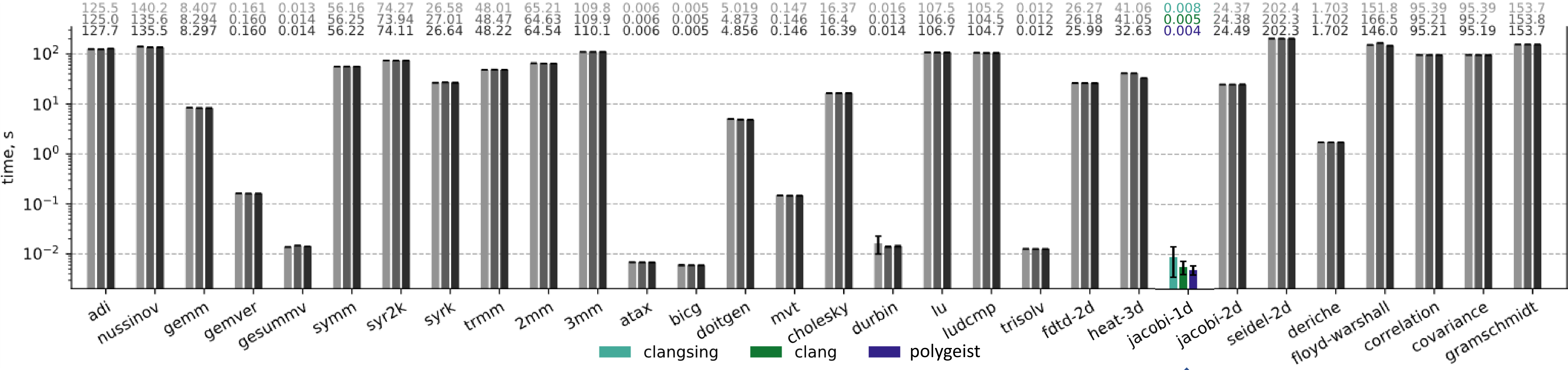


Serial Non-Polyhedral Comparison



Frontend within 0.32% of "standard" frontend

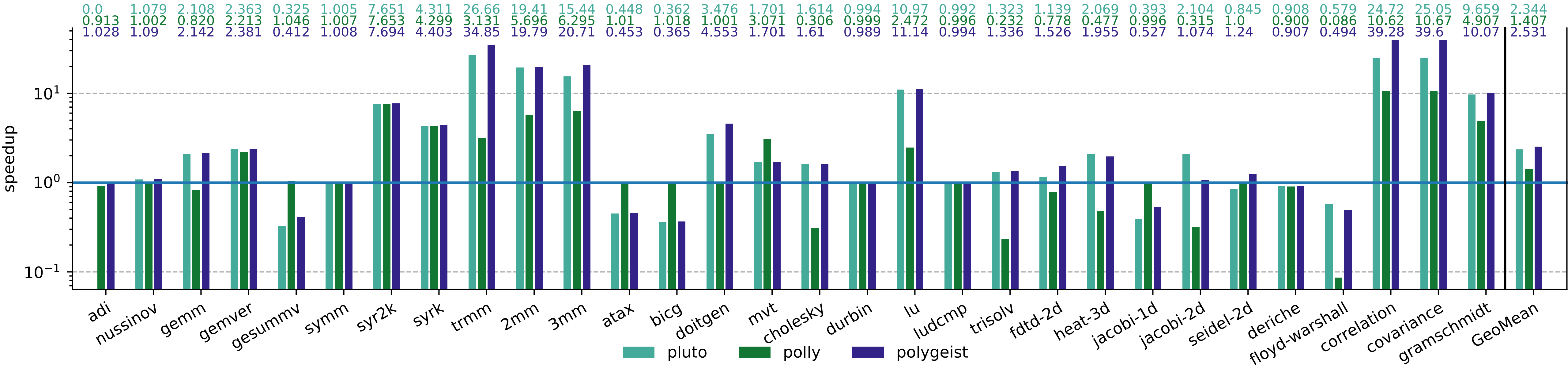
Serial Non-Polyhedral Comparison



Frontend within 0.32% of "standard" frontend

Remaining gap attributed to small tests where minor assembly differences matter

Sequential Polyhedral Comparison

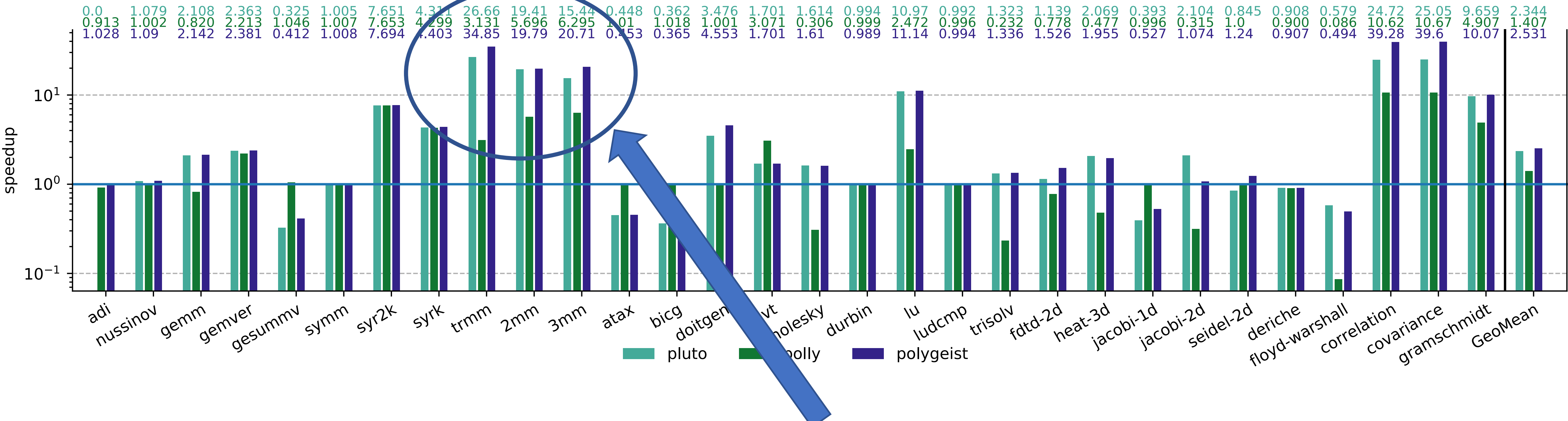


Polygeist: 2.53x speedup

Pluto: 2.34x speedup

Polly: 1.41x speedup

Sequential Polyhedral Comparison



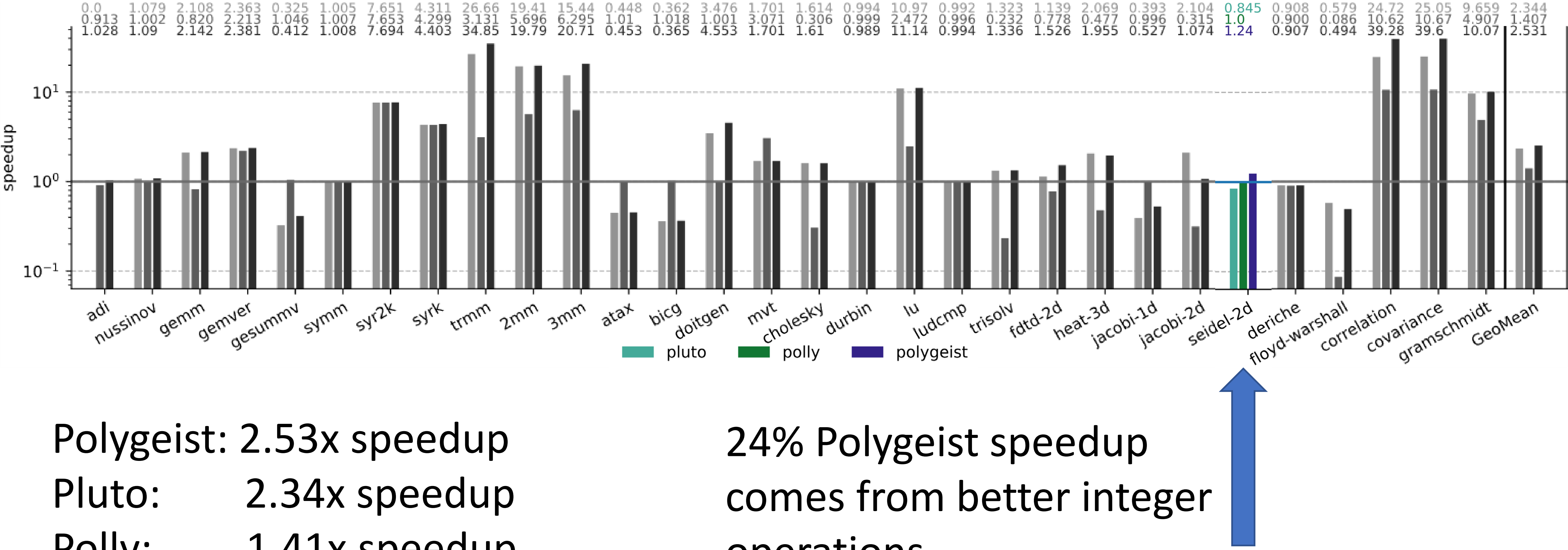
Polygeist: 2.53x speedup

Pluto: 2.34x speedup

Polly: 1.41x speedup

Big gaps come from different schedules

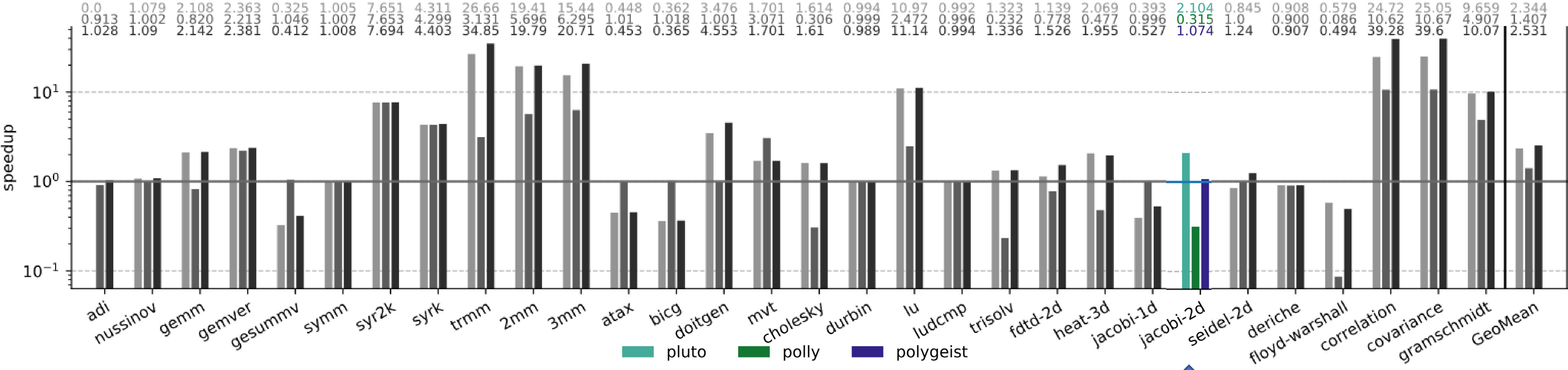
Sequential Polyhedral Comparison



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

24% Polygeist speedup comes from better integer operations

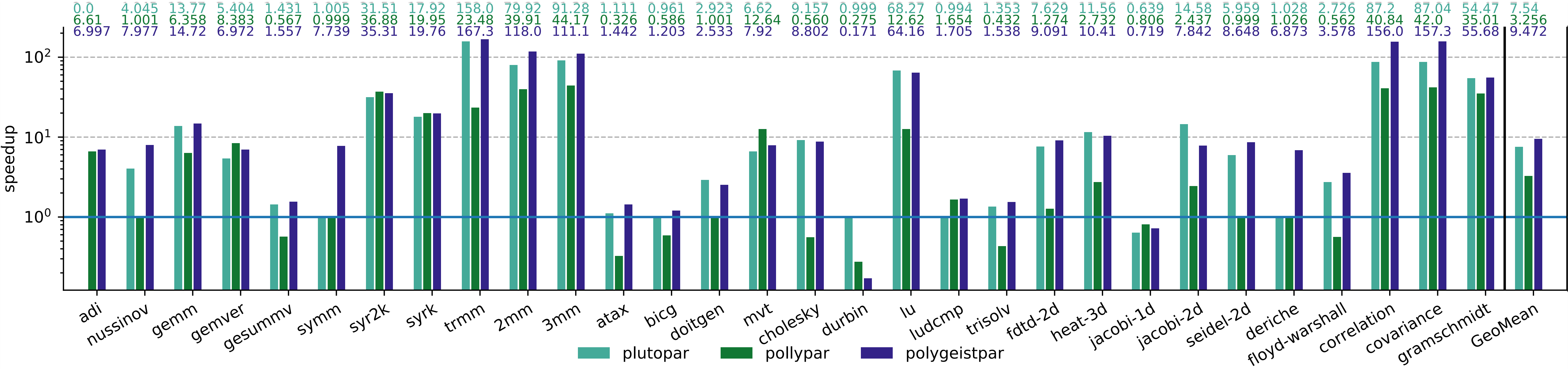
Sequential Polyhedral Comparison



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

Polygeist slower due to smaller branch-simplification timeout

Parallel Polyhedral Comparison

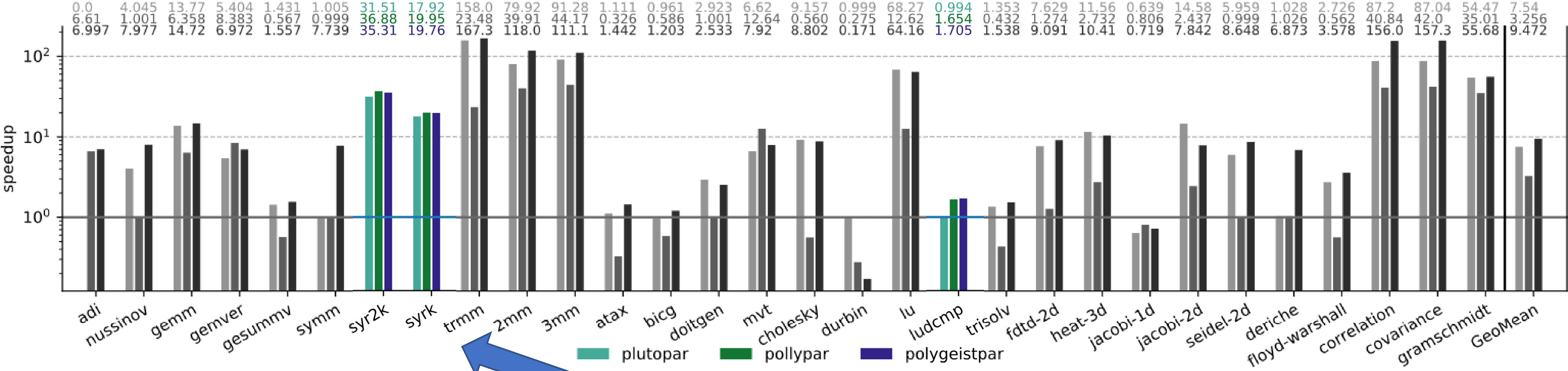


Polygeist: 9.47x speedup

Pluto: 7.54x speedup

Polly: 3.26x speedup

Parallel Polyhedral Comparison



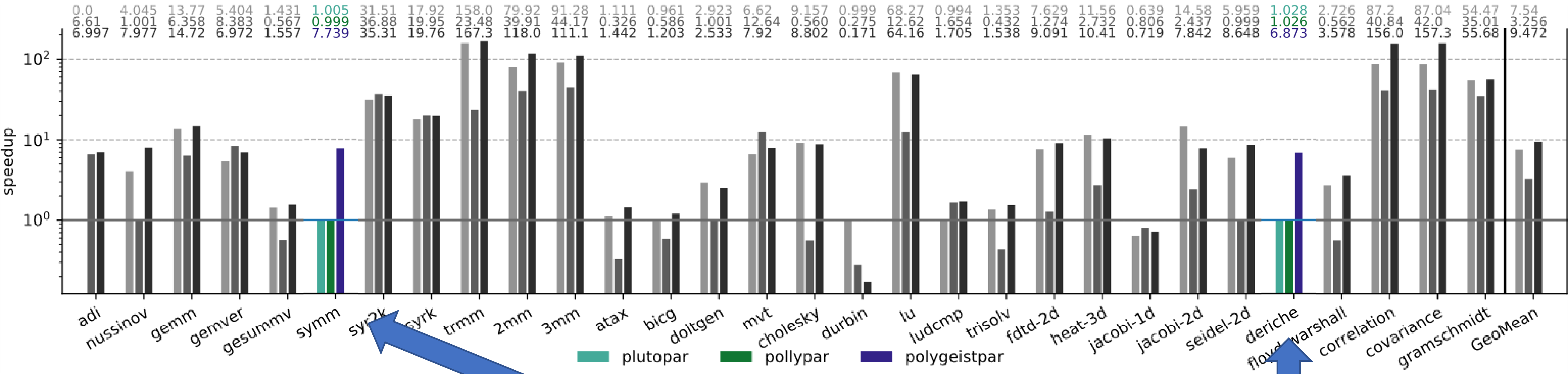
Polygeist: 9.47x speedup

Pluto: 7.54x speedup

Polly: 3.26x speedup

Polygeist and Polly benefit from SSA optimizations

Parallel Polyhedral Comparison



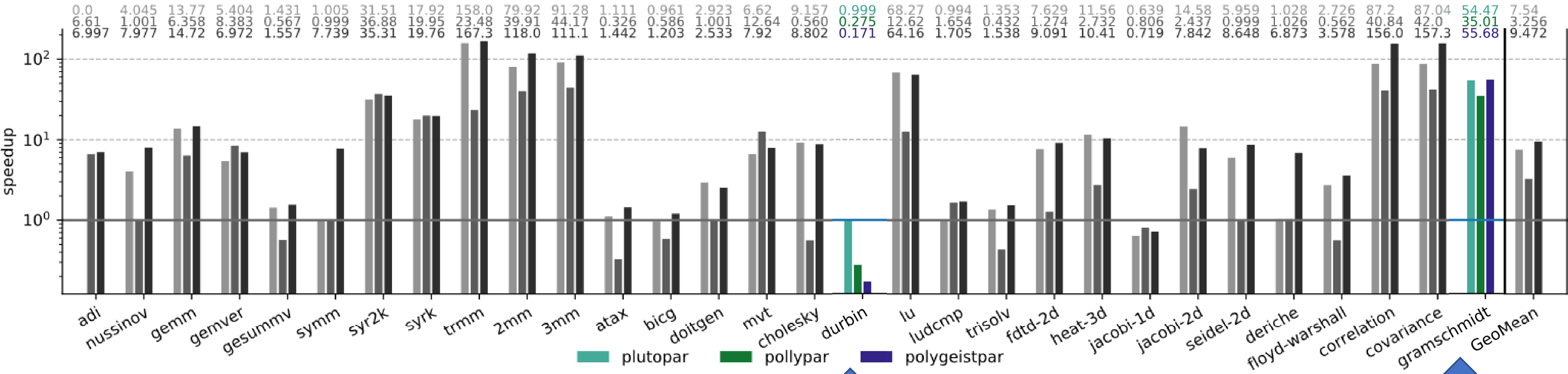
Polygeist: 9.47x speedup

Pluto: 7.54x speedup

Polly: 3.26x speedup

Polygeist can remove loop-carried dependency and parallelize when others cannot

Parallel Polyhedral Comparison



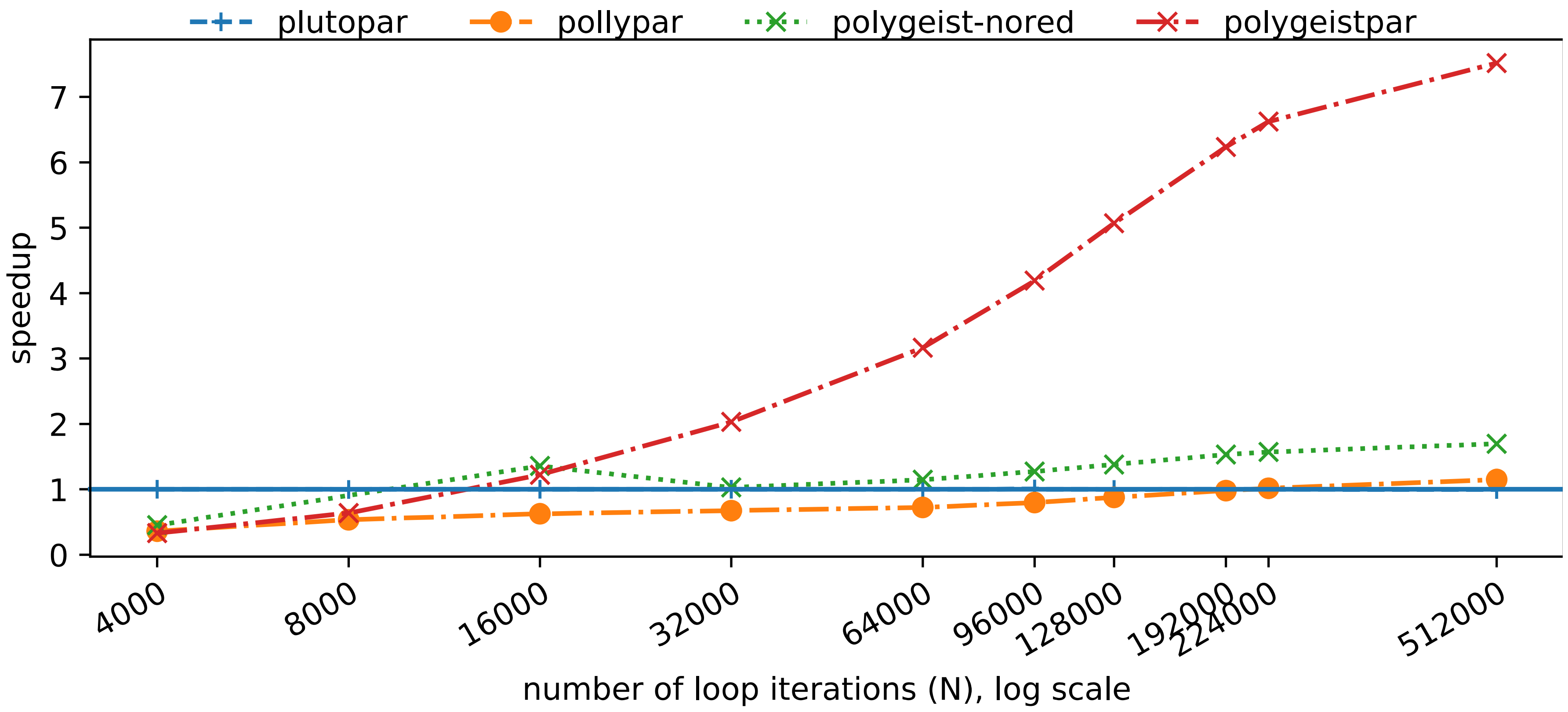
Polygeist: 9.47x speedup

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**.

```
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]);
```

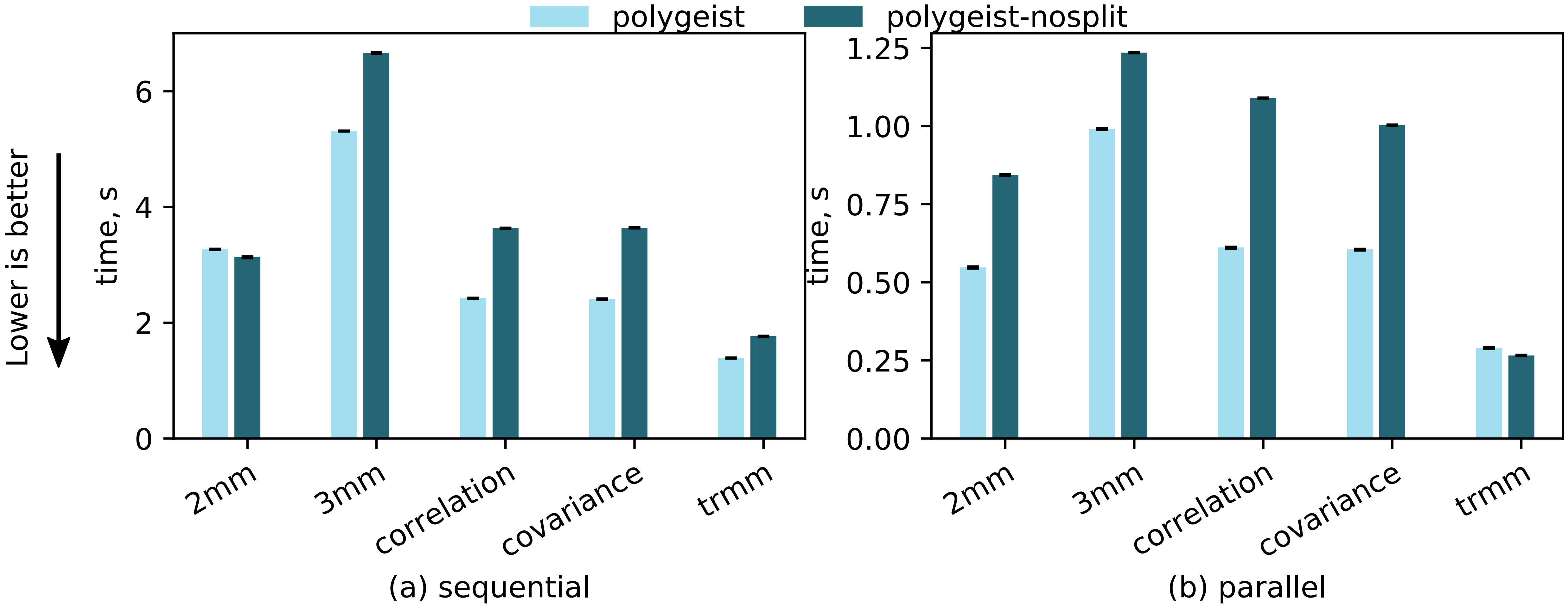
$\Downarrow$

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

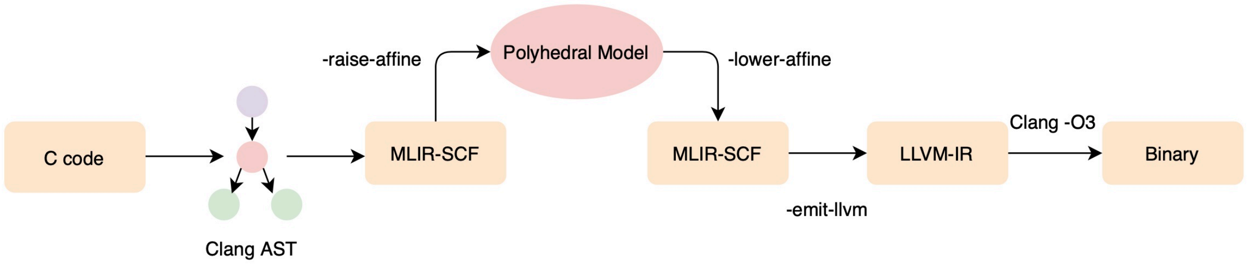
$\Rightarrow$

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

Statement Splitting



Conclusion

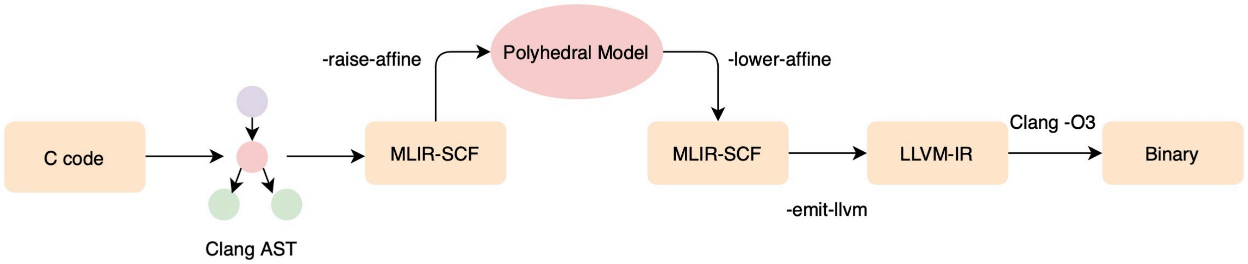


- Polygeist is an end-to-end polyhedral compiler that combines the best parts of source-level and compiler-level polyhedral tools
  - C/C++ frontend for MLIR
  - Transformations for raising to Affine
  - Integrating existing polyhedral tools with MLIR
- Polygeist provides an easy platform to introduce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

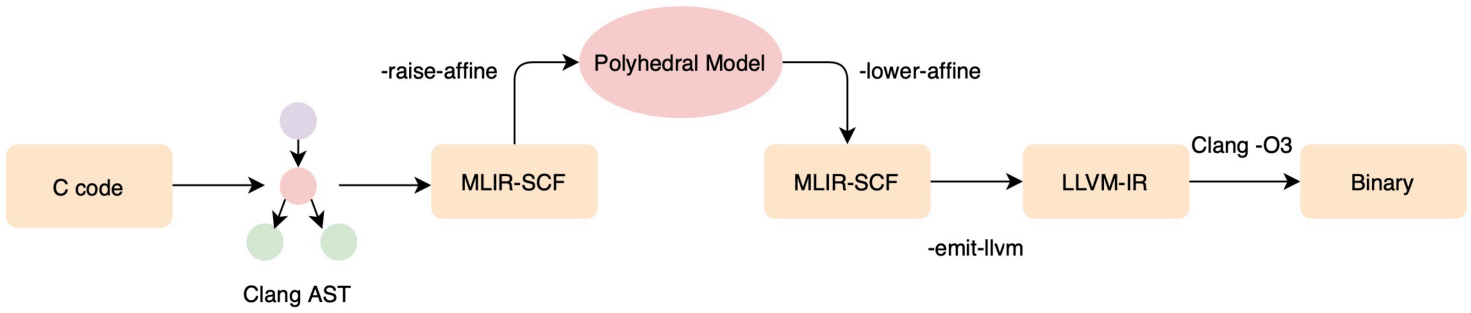
Conclusion



- 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)

Backup Slides

```
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<?x?xf32>, %i: index,
         %j: index) {
  %0 = affine.load %A[%i, %j]
  %1 = mulf %0, %0
  affine.store %1, %A[%i, %j]
  return
}
```

Domain

```
# e/i | %i   %j  | %N  | 1
   1     1    0     0    0     ## %i >= 0
   1    -1    0     1   -1     ## -%i+%N-1 >= 0
   1     0    1     0    0     ## %j >= 0
   1     0   -1     1   -1     ## -%j+%N-1 >= 0
```

Scattering

```
# e/i| s1   s2   s3   s4   s5   |%i   %j  | %N  | 1
  0   -1    0    0    0    0    0    0    0    0
  0    0   -1    0    0    0    1    0    0    0
  0    0    0   -1    0    0    0    0    0    0
  0    0    0    0   -1    0    0    1    0    0
  0    0    0    0    0   -1    0    0    0    0
```

READ/WRITE Accesses

```
# e/i| Arr  [1]  [2]  |%i   %j  | %N  | 1
  0   -1    0    0    0    0    0    0     ## %A
  0    0   -1    0    1    0    0    0     ## %i
  0    0    0   -1    0    1    0    0     ## %j
```

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

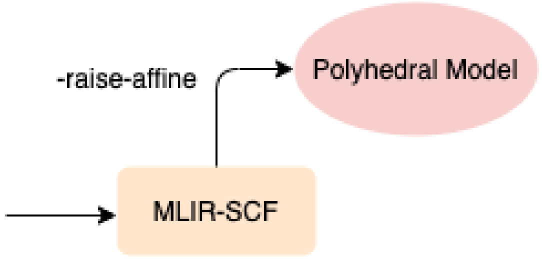
- Outline statements into functions
- Function interfaces:
  - Memory to access
  - Lifted stack allocated symbols

```
func @S0(%A: memref<?xf32>) {
  %c0 = constant 0 : index
  %s0 = dim %A, %c0 : index
  %1 = affine.load %A[0]
  affine.store %1, %A[symbol(%s0) - 1]
  return
}
```

Lift local symbols to the function interface

```
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.

```
prefixMax[i] = (prefixMax[i-1] >= data[i])
                ? prefixMax[i-1] : data[i];
```

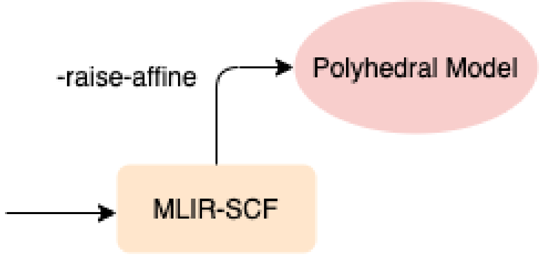
```
%0 = index_cast %arg2 : i32 to index
%1 = subi %0, %c1 : index
%2 = load %arg0[%1] : memref<?xi32>
%3 = load %arg1[%0] : memref<?xi32>
%4 = cmpi "sgt", %2, %3 : i32
%5 = scf.if %4 -> (i32) {
  %6 = load %arg0[%1] : memref<?xi32>
  scf.yield %6 : i32
} else {
  %6 = load %arg1[%0] : memref<?xi32>
  scf.yield %6 : i32
}
store %5, %arg0[%0] : memref<?xi32>
```

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

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

Polygeist Raising



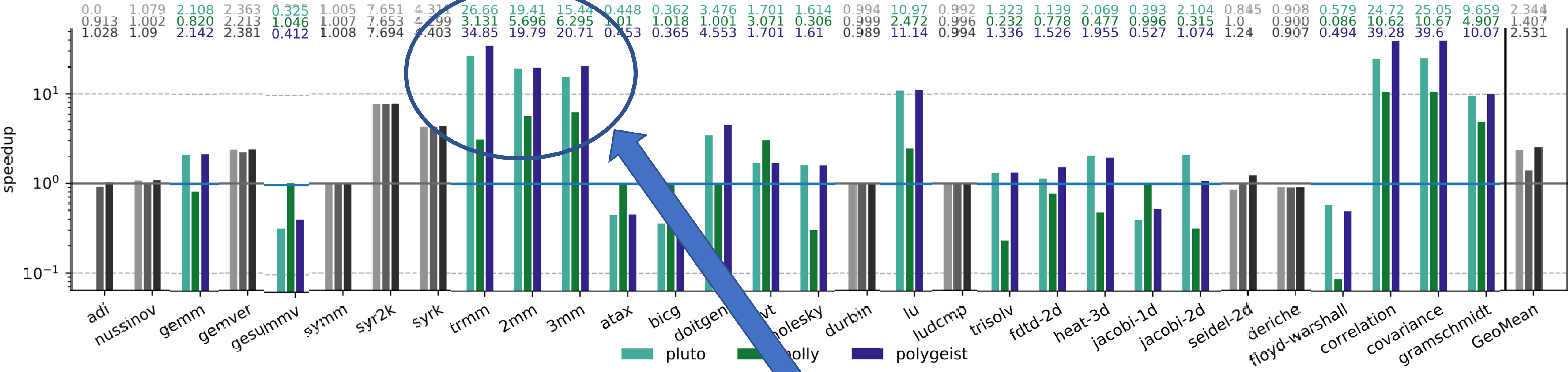

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    %c0 = constant 0 : index
    %0 = alloca() : memref<1xmemref<?xi32>>
    store %arg0, %0[%c0] : memref<1xmemref<?xi32>>
    %1 = alloca() : memref<1xi32>
    store %arg1, %1[%c0] : memref<1xi32>
    %c0_i32 = constant 0 : i32
    %c10_i32 = constant 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
  }
```

```
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 gramschmidt (56x Polygeist, 54x Pluto, 34x Polly) and durbin (6x slowdown as few iterations)