

Polygeist: C++ Frontend for MLIR



William S. Moses
wmoses@mit.edu

LLVM HPC @ SC
Nov 2022



William S. Moses
wmoses@mit.edu



Lorenzo Chelini
l.chelini@tue.nl



Ruizhe Zhao
rz3515@ic.ac.uk



Alex Zinenko
zinenko@google.com



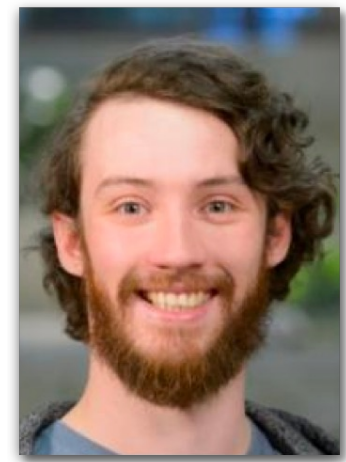
Ivan R. Ivanov
Tokyo Tech



Jens Domke
Riken



Toshio Endo
Tokyo Tech



Johannes Doerfert
ANL

The Current Compilation Pipeline

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



Parse

Clang AST

Lowering



Optimize

CodeGen



```
FunctionDecl set 'void (int *, int)'  
ForStmt  
DeclStmt  
  '-VarDecl used i 'int' cinit  
  '-IntegerLiteral 'int' 0  
BinaryOperator 'bool' '<'  
  |-ImplicitCastExpr 'int' <LValueToRValue>  
  | '-DeclRefExpr 'int' lvalue Var 0x563e22a396b8 'i'  
  'int'  
  '-IntegerLiteral 'int' 10  
UnaryOperator 'int' postfix '++'  
  '-DeclRefExpr 'int' lvalue Var 0x563e22a396b8 'i' 'int'  
...
```

```
define void @_Z3setPii(i32* %0, i32 %1) {  
    br label %4  
  
3: ; preds = %4  
    ret void  
  
4: ; preds = %2, %4  
    %5 = phi i64 [ 0, %2 ], [ %8, %4 ]  
    %6 = shl i64 %5, 1  
    %7 = getelementptr inbounds i32, i32* %0, i64 %6  
    store i32 %1, i32* %7  
    %8 = add i64 %5, 1  
    %9 = icmp eq i64 %8, 10  
    br i1 %9, label %3, label %4  
}
```

Losing High Level Structure

- LLVM, while general enough to represent any program, must represent all parts of a program in a single, low-level IR
 - Loses control flow constructs (if, for, etc)
 - Hides parallelism behind runtime calls (and for GPU in a separate module)
 - High-level semantics & properties cannot be represented and are lost.

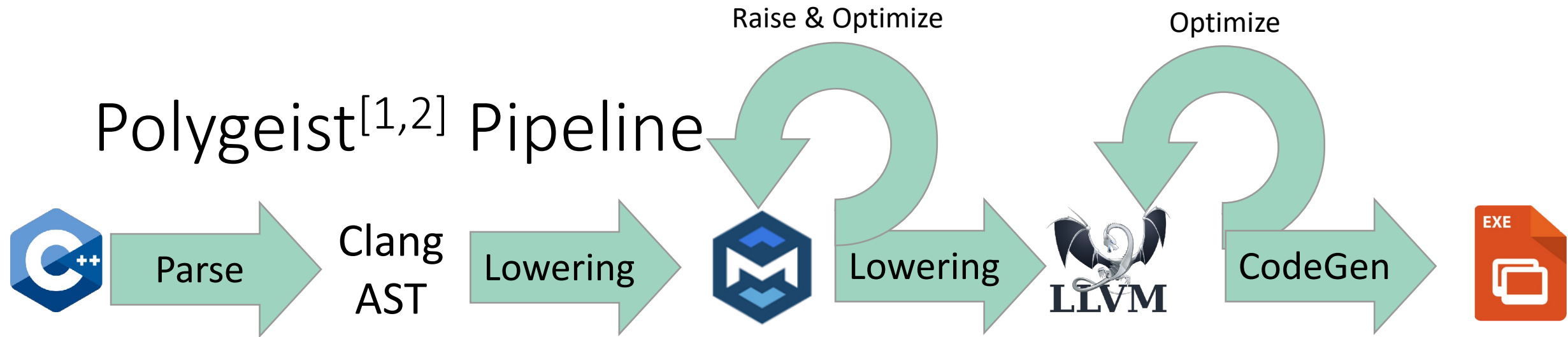
```
void foo(DataStructure& x) {  
    print(size(x));  
    insert(x);  
    print(size(x));  
}
```

```
define void @foo(ptr %x) {  
    %2 = call @size(ptr %x)  
    call @print(i32 %2)  
    call @insert(ptr %x)  
    ; %3 = add i32 %2, 1  
    %3 = call @size(ptr %x)  
    call @print(i32 %3)  
    ret void  
}
```



The MLIR Framework

- MLIR is a recent compiler infrastructure designed for reuse and extensibility
- Rather than providing a predefined set of instructions and types, MLIR operates on collections of dialects that contain sets of interoperable user-defined operations, attributes and types
- Anyone can define their own optimizable dialect/operation, with a large set of existing dialects (structured control flow, affine, GPU, quantum, fully homomorphic encryption, circuits, LLVM, and more!)



- Generic C and C++ frontend that generates "standard" and user-defined MLIR (templates, classes, unions, etc. all supported)
- Raising transformations for raising "standard" MLIR to high-level
- Collection of high-level optimization passes (general mem2reg, parallel optimizations)
- Polyhedral optimization via novel optimizations and integrating prior tools (Pluto, CLooG) into MLIR
- Parallel/GPU optimizations & transformations

[1] Polygeist: Raising C to Polyhedral MLIR; Moses, Chelini, Zhao, and Zinenko. PACT '21.

[2] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs; Moses, Ivanov, Domke, Endo, Doerfert, and Zinenko. PPOPP '23

Polygeist Frontend Example

```
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

```
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

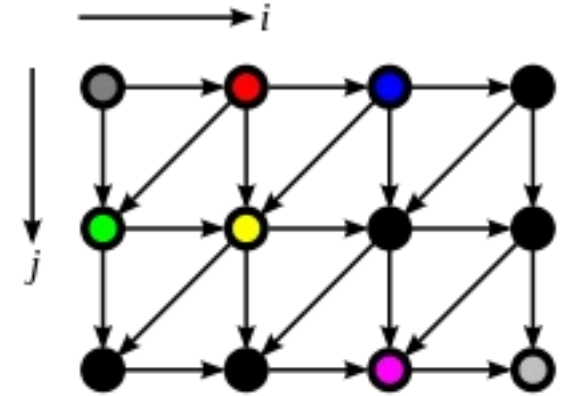
Polygeist Raising

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

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

Two Case Studies

- Demonstrate the benefits of a compiler-based IR with multiple abstraction levels by two case studies in Polygeist/MLIR
 - Polyhedral optimization [1]
 - GPU optimization and transpilation to the CPU [2]

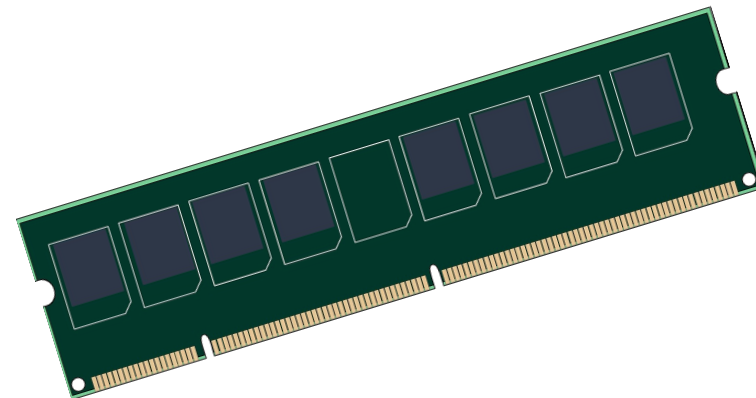
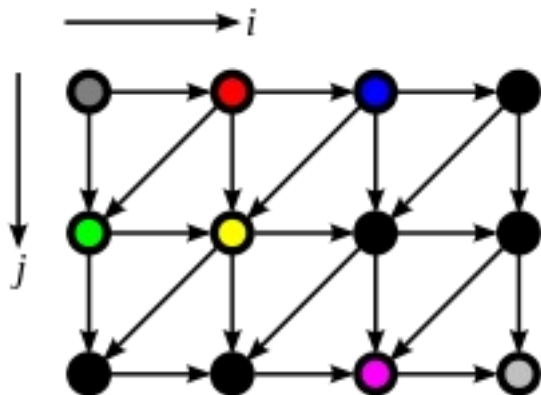


[1] Polygeist: Raising C to Polyhedral MLIR; Moses, Chelini, Zhao, and Zinenko. PACT '21.

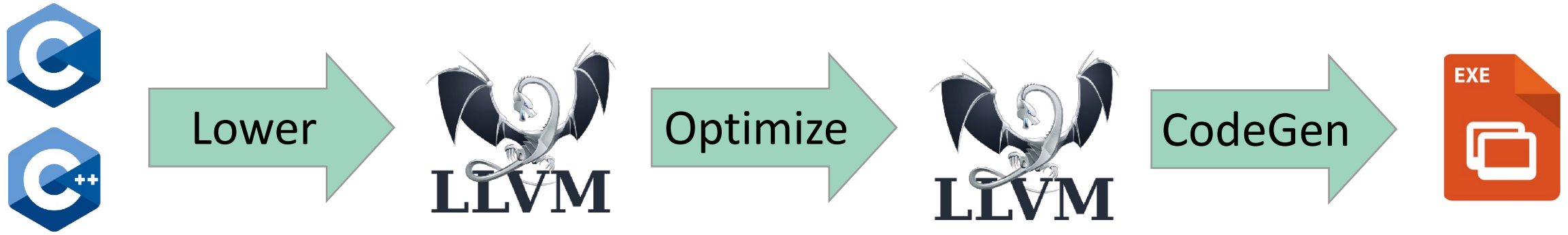
[2] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs; Moses, Ivanov, Domke, Endo, Doerfert, and Zinenko. PPOPP '23

Case Study 1: The Polyhedral Model

- Represent programs as a collection of computations and constraints on a multi-dimensional grid (polyhedron)
- 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.

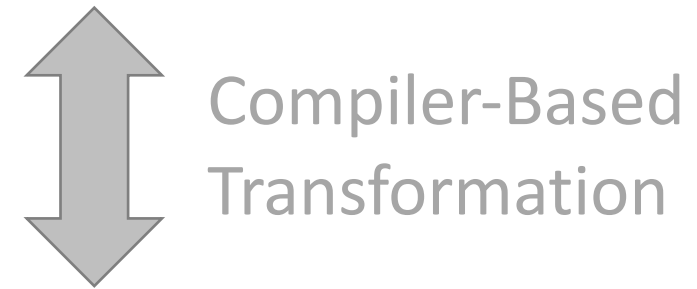


Polyhedral Compilation Today



Pluto, PPCG

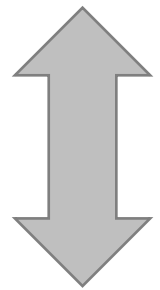
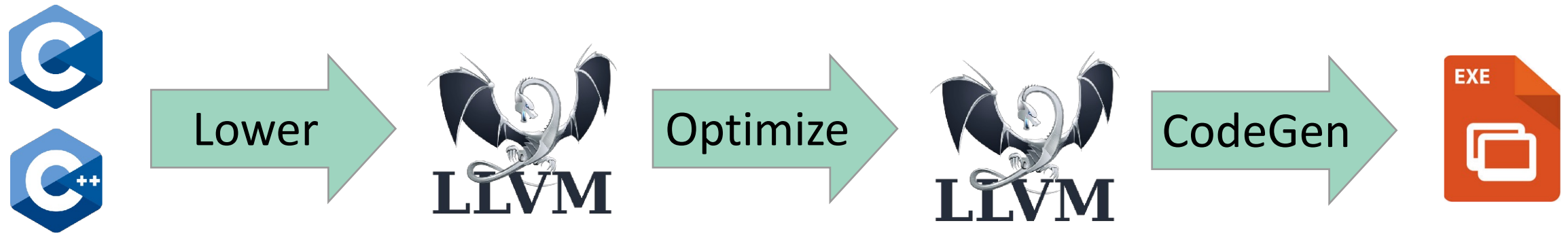
Keeps High Level Info;
Fails On Unoptimized Code



Polly (LLVM), Graphite (GCC)

Runs Optimizations First
Attempts to Recover Structure

Polyhedral Compilation Today



Source-Based
Transformation

Pluto, PPCG

Keeps High Level Info;
Fails On Unoptimized Code



Compiler-Based
Transformation

Polly (LLVM), Graphite (GCC)

Runs Optimizations First
Attempts to Recover Structure

Polygeist + Polyhedral

- Polygeist gets the benefits of both worlds
 - Preserves the high-level (polyhedral/affine) structure of programs AND
 - Optimizes & simplifies programs prior to transformations
 - New polyhedral optimizations that cannot otherwise be easily expressed

```
int mul2(int val) { return 2 * val; }

void set(int *arr, int val) {
    for(int i=0; i<10; i++){
        arr[mul2(i)] = val;
    }
}
```

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

- Serial Programs: 80% speedup over LLVM-based; 8% speedup over source-based
- Parallel Programs: 190% speedup over LLVM-based; 26% speedup over source-based

Case Study 2: GPUs

- Mainstream compilers do not have a high-level representation of parallelism, making optimization difficult or impossible
- This is accentuated for GPU programs where the kernel is kept in a separate module to allow emission of different assembly and synchronization is treated as a complete optimization barrier.

```
__global__ void normalize(int *out, int* in, int n) {  
    int tid = blockIdx.x;  
    if (tid < n)  
        out[tid] = in[tid] / sum(in, n);  
}  
  
void launch(int *out, int* in, int n) {  
    normalize<<<n>>>(d_out, d_in, n);  
}
```

Host Code

```
target triple = "x86_64-unknown-linux-gnu"  
  
define void @_Z6launchPiS_i(i32* %out,  
                           i32* %in,  
                           i32 %n) {  
    call i32 @pushCallConfiguration(...)  
    call i32 @cudaLaunch(@_device_stub, ...)  
    ret void  
}
```

Device Code

```
target triple = "nvptx"  
  
define void @_Z9normalize(i32* %out,  
                        i32* %in, i32 %n) {  
    %4 = call i32 @llvm.tid.x()  
    %5 = icmp slt i32 %4, %n  
    br i1 %5, label %6, label %13  
  
6:  
    %8 = getelementptr i32, i32* %in, i32 %4  
    %9 = load i32, i32* %8, align 4  
    %10 = call i32 @_Z3sumPii(i32* %in, i32 %n)  
    %11 = sdiv i32 %9, %10  
    %12 = getelementptr i32, i32* %out, i32 %4  
    store i32 %11, i32* %12, align 4  
    br label %13  
  
13:  
    ret void  
}
```

Preserve the parallel structure

- Maintain GPU parallelism in a form understandable to the compiler
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {  
    int tid = blockIdx.x;  
    if (tid < n)  
        out[tid] = in[tid] / sum(in, n);  
}  
  
void launch(int *out, int* in, int n) {  
    normalize<<<n>>>(d_out, d_in, n);  
}
```



```
func @_Z6launch(%out: memref<?xi32>,  
               %in: memref<?xi32>, %n: i32) {  
    %c1 = constant 1 : index  
    %c0 = constant 0 : index  
  
    parallel (%tid) = (%c0) to (%n) step (%c1) {  
        %2 = load %in[%tid]  
        %sum = call @_Z3sumPii(%in, %n)  
        %4 = divsi %2, %sum : i32  
        store %4, %out[%tid]  
        yield  
    }  
    return  
}
```

Preserve the parallel structure

- Maintain GPU parallelism in a form understandable to the compiler
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {  
    int tid = blockIdx.x;  
    if (tid < n)  
        out[tid] = in[tid] / sum(in, n);  
}  
  
void launch(int *out, int* in, int n) {  
    normalize<<<n>>>(d_out, d_in, n);  
}
```



```
func @_Z6launch(%out: memref<?xi32>,  
               %in: memref<?xi32>, %n: i32) {  
    %c1 = constant 1 : index  
    %c0 = constant 0 : index  
    %sum = call @_Z3sumPii(%in, %n)  
    parallel (%tid) = (%c0) to (%n) step (%c1) {  
        %2 = load %in[%tid]  
  
        %4 = divsi %2, %sum : i32  
        store %4, %out[%tid]  
        yield  
    }  
    return  
}
```

Synchronization via Memory

- Synchronization (`sync_threads`) ensures all threads within a block finish executing `codeA` before executing `codeB`
- The desired synchronization behavior can be reproduced by defining `sync_threads` to have the union of the memory semantics of the code before and after the sync.
- This prevents code motion of instructions which require the synchronization for correctness, but permits other code motion (e.g. index computation).

```
codeA(fib(idx));  
  
sync_threads;  
  
codeB(fib(idx));
```



```
off = fib(idx);  
codeA(off);  
  
sync_threads;  
  
codeB(off);
```

Synchronization via Memory

- High-level synchronization representation enables new optimizations, like sync elimination.
- A synchronize instruction is not needed if the set of read/writes before the sync don't conflict with the read/writes after the sync.

```
__global__ void bpn_layerforward(...) {
    __shared__ float node[HEIGHT];
    __shared__ float weights[HEIGHT][WIDTH];

    if ( tx == 0 )
        node[ty] = input[index_in] ;

    // Unnecessary Barrier #1
    // None of the read/writes below the sync
    // (weights, hidden)
    // intersect with the read/writes above the sync
    // (node, input)
    __syncthreads();

    // Unnecessary Store #1
    weights[ty][tx] = hidden[index];

    __syncthreads();

    // Unnecessary Load #1
    weights[ty][tx] = weights[ty][tx] * node[ty];
    ...
}
```

GPU Transpilation

- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. CPU/OpenMP)
- Most CPU backends do not have an equivalent block synchronization
- Efficiently lower a top-level synchronization by distributing the parallel for loop around the sync, and interchanging control flow

```
parallel_for %i = 0 to N {  
    codeA(%i);  
    sync_threads;  
    codeB(%i);  
}
```

```
parallel_for %i = 0 to N {  
    codeA(%i);  
}  
parallel_for %i = 0 to N {  
    codeB(%i);  
}
```

GPU Synchronization Lowering: Control Flow

- Synchronization within control flow (for, if, while, etc) can be lowered by splitting around the control flow and interchanging the parallelism.

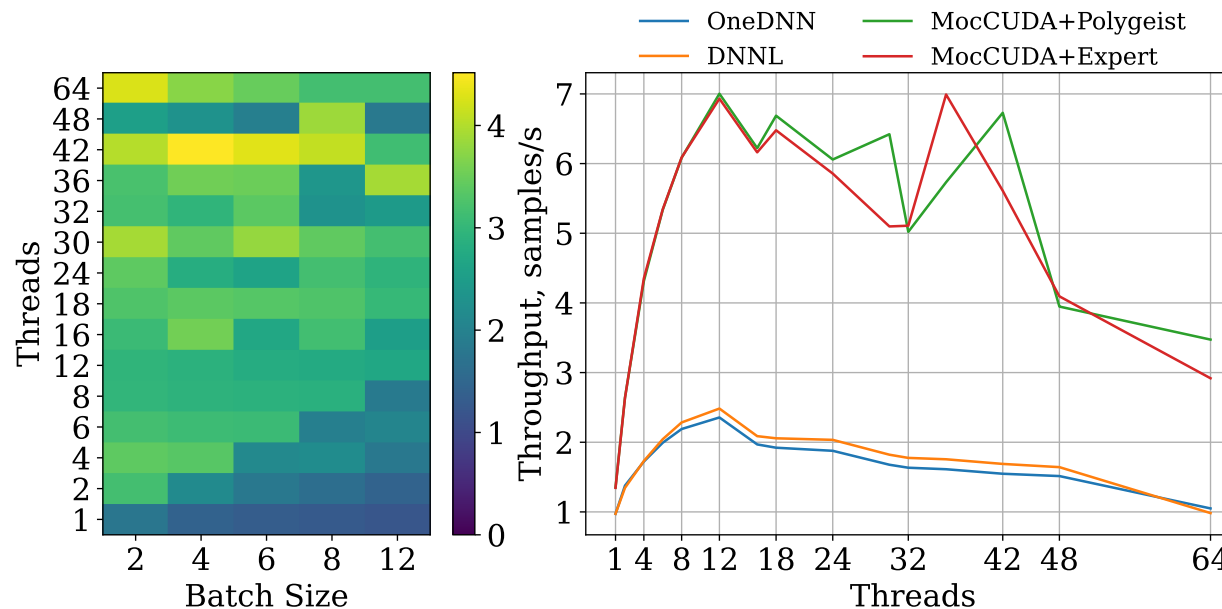
```
parallel_for %i = 0 to N {  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
}
```

```
for %j = ... {  
  parallel_for %i = 0 to N {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
}
```

```
for %j = ... {  
  parallel_for %i = 0 to N {  
    codeB1(%i, %j);  
  }  
  parallel_for %i = 0 to N {  
    codeB2(%i, %j);  
  }  
}
```


GPU Transpilation Performance

- CUDA programs transcompiled by Polygeist not only match the performance of handwritten OpenMP programs, but ***achieve a speedup!***
 - 58% geomean speedup on Rodinia
 - 2.7x geomean speedup on PyTorch versus built-in CPU backend (also using our MocCUDA compatibility layer)



“Case Study 3”: Your Programs!

- There are already several efforts starting using Polygeist/MLIR to leveraging the benefits of optimizable multi-level operations
 - SYCL
 - Circuit Compilation
 - BLAS Kernels
 - Databases
 - ...
- If you're interested in applying such techniques to your programs, please reach out!

Conclusion

- Optimizable, multi-level operations are key to compiler extensibility and therefore performance
- Polygeist/MLIR is a new Clang-based compiler that allows you to leverage this extensibility
 - C/C++ frontend for MLIR
 - Compiler transformations for raising MLIR to a higher-level
 - Collection of high-level optimization passes (general mem2reg, etc)
 - Polyhedral optimization via novel optimizations and integrating prior tools into MLIR
 - Parallel/GPU optimizations & transformations
- Polygeist beats existing polyhedral tools on sequential and parallel code
- Polygeist can optimize and transcompile your GPU/parallel code
- Supports recognizing and lowering to custom ops/dialects
- LLVM incubator project, open sourced on Github, see <https://polygeist.mit.edu> and discuss on Discourse!

Acknowledgements

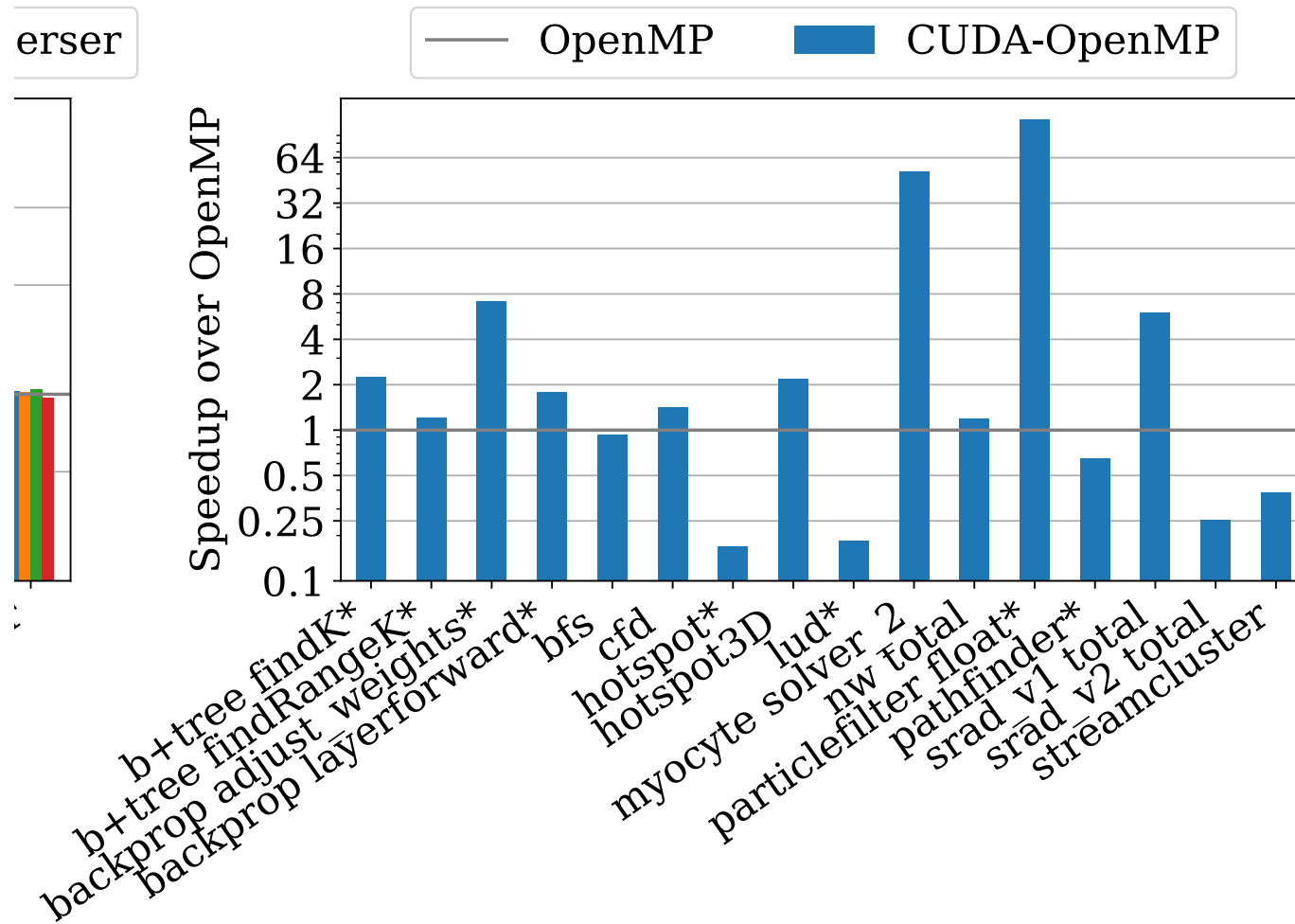
- Thanks to Valentin Churavy, Albert Cohen, Henk Corporaal, Tobias Grosser, and Charles Leiserson for thoughtful discussions on this work.
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship, in part by Los Alamos National Laboratories, and in part by the United States Air Force Research Laboratory. 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.
- 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.
- The work was supported in part by JSPS KAKENHI Grant Number JP19H04119 and in part by the Japan Society for the Promotion of Science KAKENHI Grant Number JP19H04119, and in part by the New Energy and Industrial Technology Development Organization (NEDO).

Conclusion

- Optimizable, multi-level operations are key to compiler extensibility and therefore performance
- Polygeist/MLIR is a new Clang-based compiler that allows you to leverage this extensibility
 - C/C++ frontend for MLIR
 - Compiler transformations for raising MLIR to a higher-level
 - Collection of high-level optimization passes (general mem2reg, etc)
 - Polyhedral optimization via novel optimizations and integrating prior tools into MLIR
 - Parallel/GPU optimizations & transformations
- Polygeist beats existing polyhedral tools on sequential and parallel code
- Polygeist can optimize and transcompile your GPU/parallel code
- Supports recognizing and lowering to custom ops/dialects
- LLVM incubator project, open sourced on Github, see <https://polygeist.mit.edu> and discuss on Discourse!

Backup Slides

- Text



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	0	## %i >= 0
1	-1	0	1	-1	-1	## -%i+%N-1 >= 0
1	0	1	0	0	0	## %j >= 0
1	0	-1	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	0	-1	0	0	0	0	1	0	0	0
0	0	0	-1	0	0	0	0	0	0	0
0	0	0	0	-1	0	0	0	1	0	0
0	0	0	0	0	-1	0	0	0	0	0

READ/WRITE Accesses

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

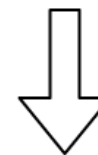
- ✓ The compiler research has recently been enamored by the MLIR framework, whose first-class polyhedral representation may provide benefits on a variety of codes
- ✓ We can fully leverage decades of polyhedral research by connecting MLIR with existing polyhedral tools.
- ✓ Without MLIR-versions of standard 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

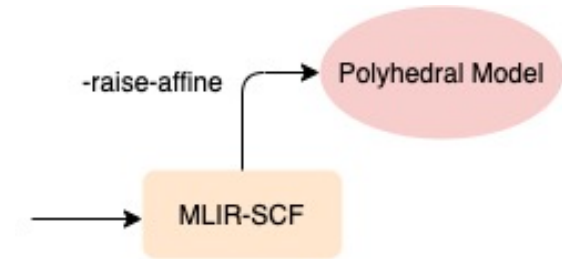
```
func @S0(%A: memref<?xf32>) {  
    %c0 = constant 0 : index  
    %s0 = dim %A, %c0 : index  
  
    %1 = affine.load %A[0]  
    affine.store %1, %A[symbol(%s0) - 1]  
    return  
}
```

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

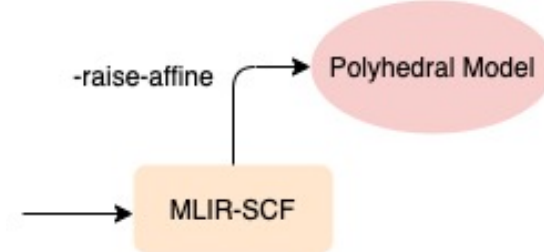
- Represent SCoP with polyhedral-friendly loops and conditions
- Core Affine representation
 - Symbols - parameters
 - Dimensions - symbol extension that accepts induction variables
 - Maps - multi-dimensional function of symbols and dimensions
 - Sets - integer tuples constrained by a conjunction

```
%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  
}
```

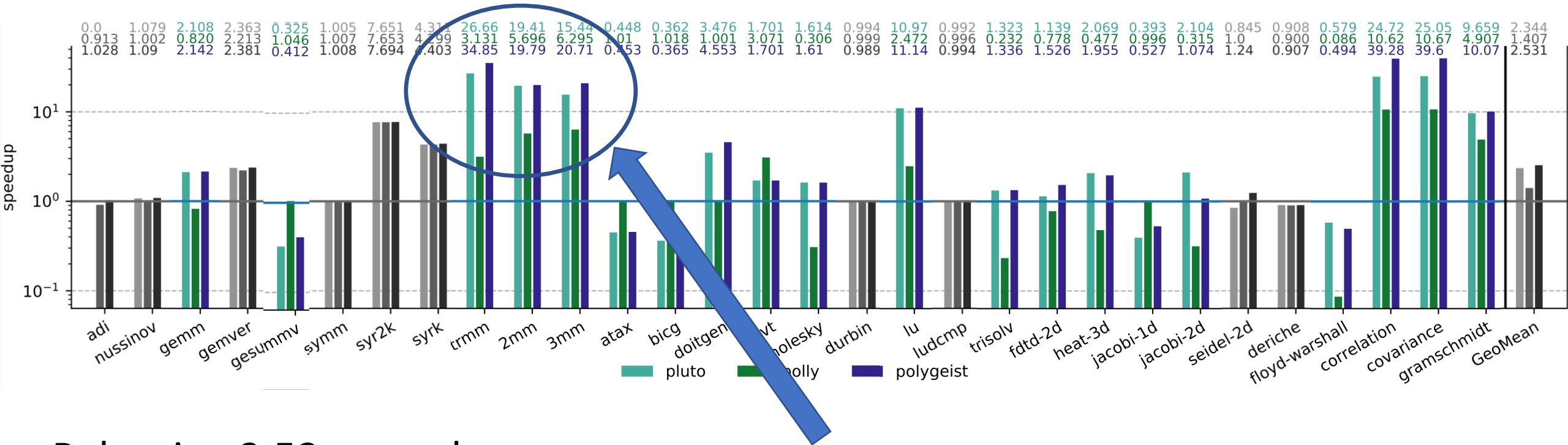
```
func @set(%arg0: memref<?xi32>, %arg1: i32) {  
  affine.for %arg2 = 0 to 10 {  
    affine.store %arg1, %arg0[%arg2 * 2]  
      : memref<?xi32>  
  }  
  return  
}
```



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 ($\sim 10^{11}$) more integer instructions on seidel-2d than Polygeist, which is ~ 59 s 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



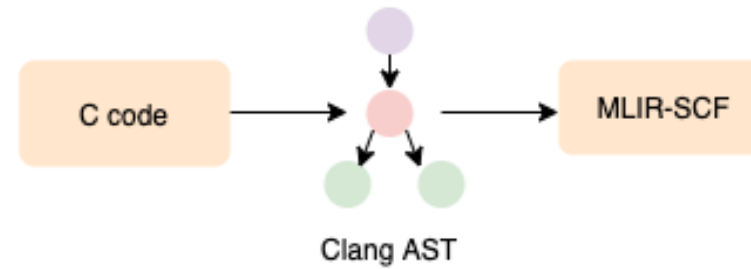
Polygeist: 2.53x speedup
 Pluto: 2.34x speedup
 Polly: 1.41x speedup

Big gaps come from
 different schedules

Parallel Performance Differences

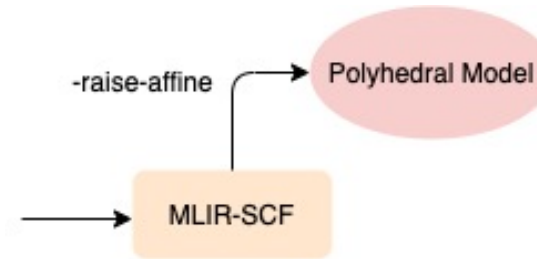
- Same scheduling differences as sequential (Cholesky and LU are better on Pluto/Polygeist than Polly; Gemver and MVT are better on Polly)
- Ludcmp and syr(2)k benefit from SSA optimizations
- Polygeist is only framework that can parallelize deriche (6.9x) and symm (7.7x) by analyzing and removing the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

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.

Polygeist Raising



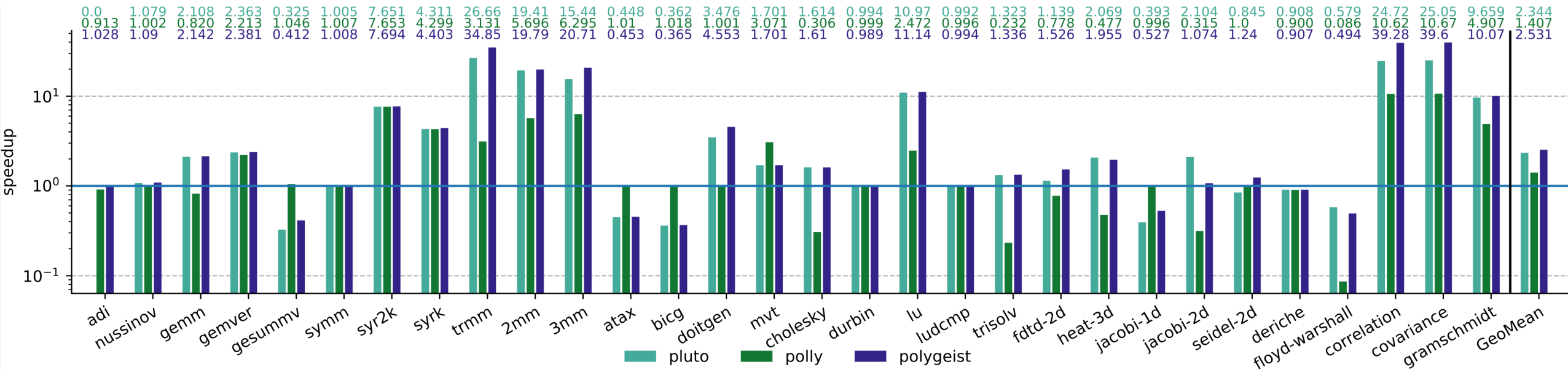
- Local variables eliminated by new MLIR mem2reg pass
- Canonicalizations run to simplify the code, including while => for
- Raising an operation (for, if, load, store) to its affine-variant:
 - Detect if index calculation is a valid affine expression
 - Progressively fold index calculation into a new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

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



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

Sequential Polyhedral Comparison

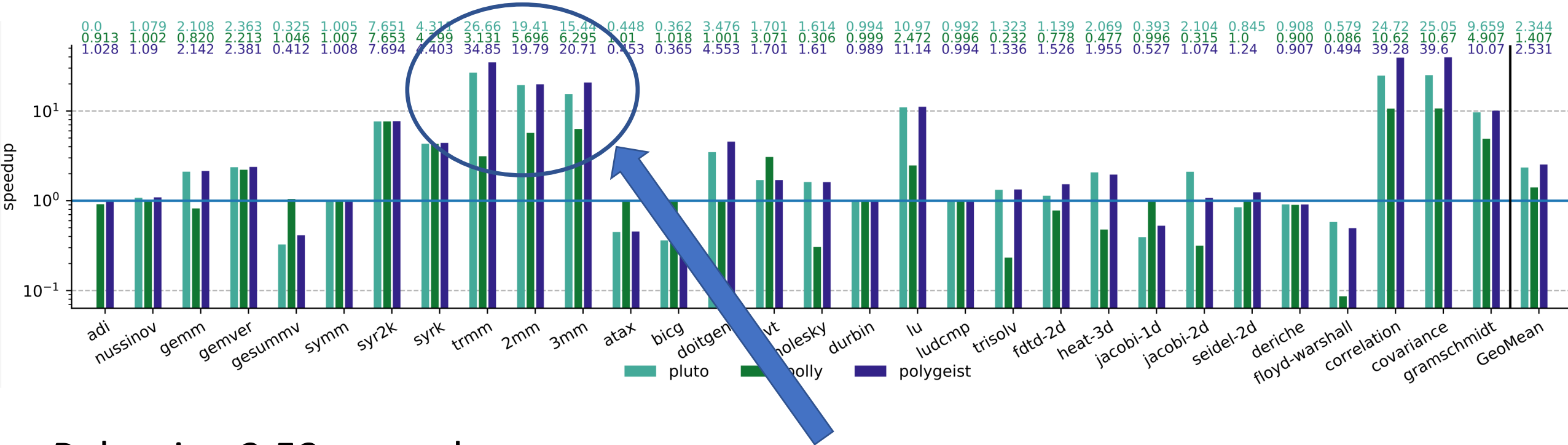


Polygeist: 2.53x speedup

Pluto: 2.34x speedup

Polly: 1.41x speedup

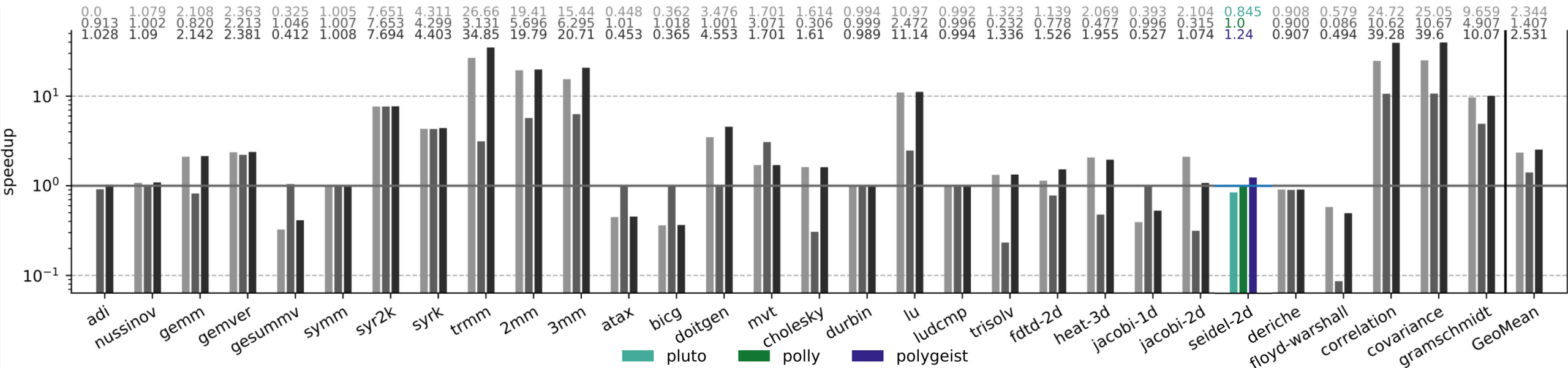
Sequential Polyhedral Comparison



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

Big gaps come from
different schedules

Sequential Polyhedral Comparison



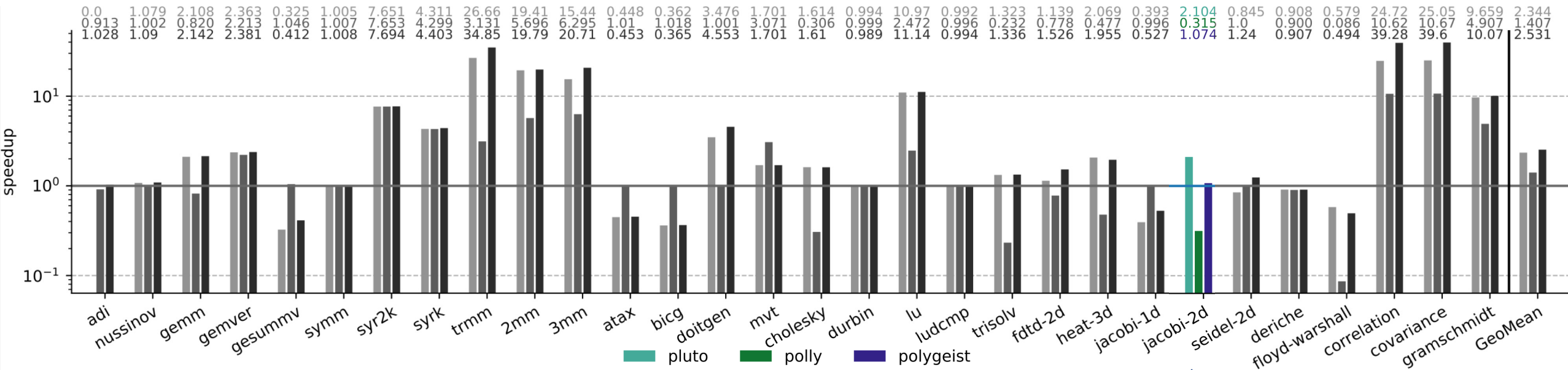
Polygeist: 2.53x speedup

Pluto: 2.34x speedup

Polly: 1.41x speedup

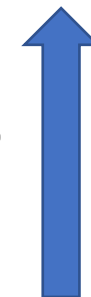
24% Polygeist speedup
comes from better integer
operations

Sequential Polyhedral Comparison

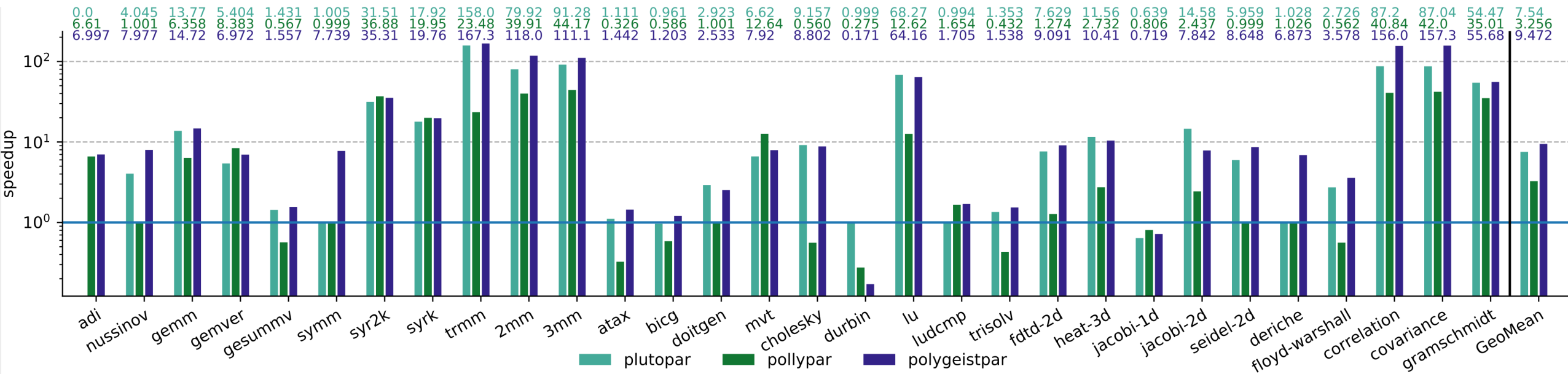


Polygeist: 2.53x speedup
 Pluto: 2.34x speedup
 Polly: 1.41x speedup

Polygeist slower due to
 smaller branch-
 simplification timeout



Parallel Polyhedral Comparison

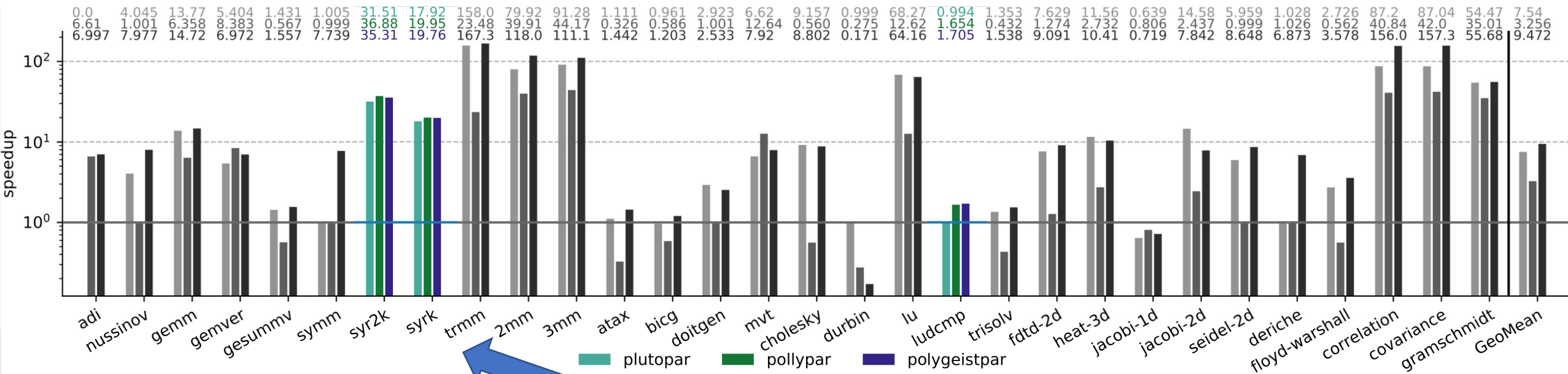


Polygeist: 9.47x speedup

Pluto: 7.54x speedup

Polly: 3.26x speedup

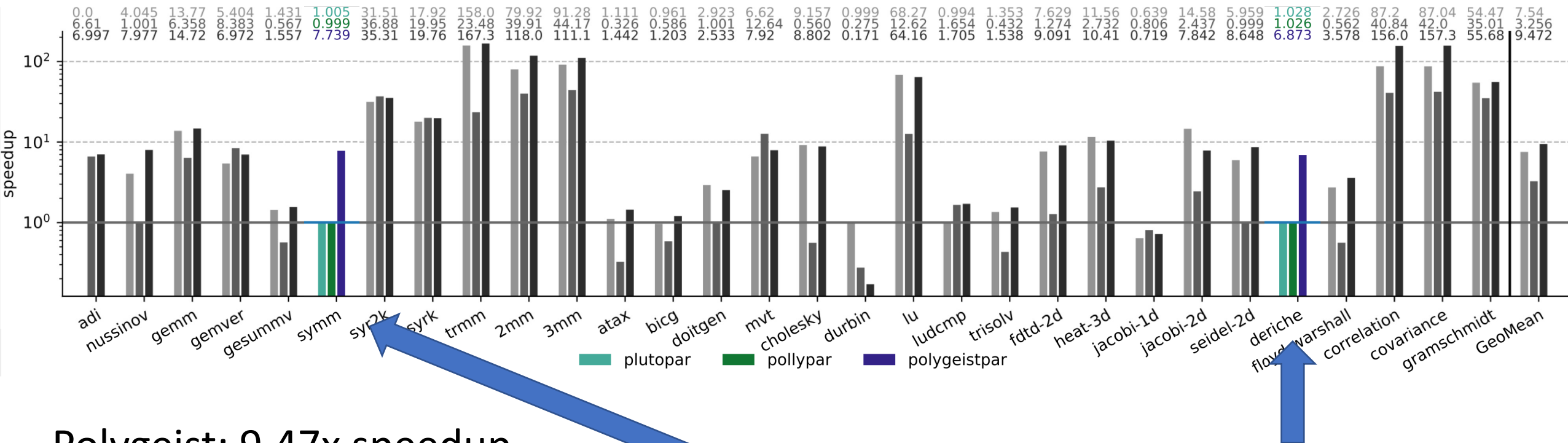
Parallel Polyhedral Comparison



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

Polygeist and Polly benefit
 from SSA optimizations

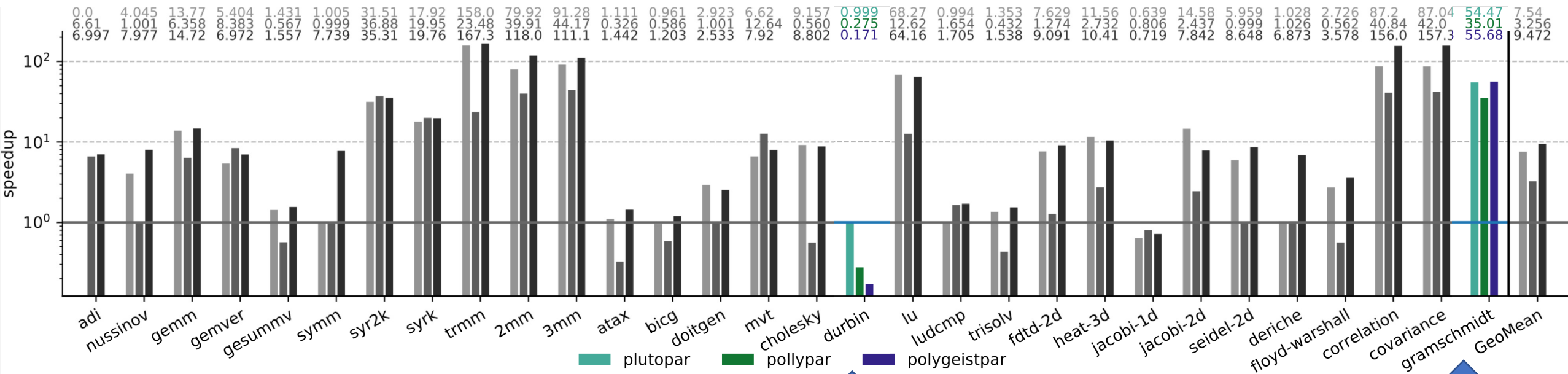
Parallel Polyhedral Comparison



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



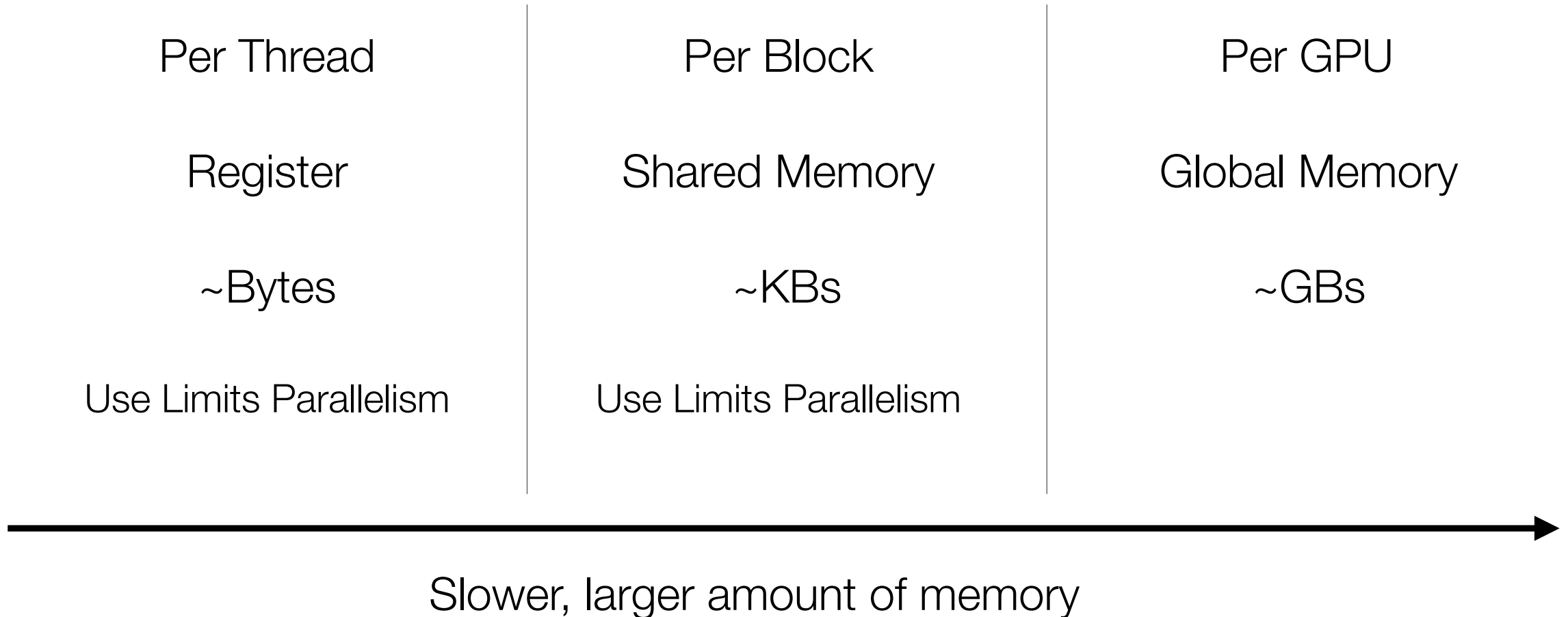
Polygeist: 9.47x speedup

Pluto: 7.54x speedup

Polly: 3.26x speedup

Polygeist can detect parallel reductions

GPU Memory Hierarchy



Case Study 2: GPUs

GPU Code

```
__device__ int sum(int* data, int n);

__global__ void square(int *out, int* in, int n) {
    int tid = blockIdx.x;
    if (tid < n)
        out[tid] = in[tid] / sum(in, n);
}
```

CPU Code

```
void launch(int *h_out, int* h_in, int n) {

    int *d_out, *d_in;
    cudaMalloc(&d_out, n*sizeof(n));
    cudaMalloc(&d_in, n*sizeof(n));

    cudaMemcpy(d_in, h_in, n*sizeof(n), cudaMemcpyHostToDevice);

    square<<<(n+31)/32, 32>>>(d_out, d_in, n);

    cudaMemcpy(h_out, d_out, n*sizeof(n), cudaMemcpyDeviceToHost);
}
```

CPU Memory

GPU Memory

Block

Thread

A first-class representation of parallelism

- Current mainstream compilers do not have a good notion or representation of parallelism
- This is accentuated for GPU programs where the kernel is kept in a separate module to allow emission of different assembly

```
target triple = "x86_64-unknown-linux-gnu"
```

```
define void @_Z6launchPiS_i(i32* %out, i32* %in, i32 %n)
{
    call i32 @__cudaPushCallConfiguration(...)
    call i32 @cudaLaunchKernel(@_device_stub, ...)
    ret void
}
```

```
target triple = "nvptx"
```

```
define void @_Z9normalizePiS_i(i32* %out, i32* %in, i32 %n) {
    %4 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
    %5 = icmp slt i32 %4, %n
    br i1 %5, label %6, label %13

6: ; preds = %3
    %8 = getelementptr inbounds i32, i32* %in, i32 %4
    %9 = load i32, i32* %8, align 4
    %10 = call i32 @_Z3sumPii(i32* %in, i32 %n) #5
    %11 = sdiv i32 %9, %10
    %12 = getelementptr inbounds i32, i32* %out, i32 %4
    store i32 %11, i32* %12, align 4
    br label %13
}
```

```
13:
```

GPU Synchronization Lowering

- Most CPU backends (e.g. Cilk, OpenMP) do not have an equivalent & general synchronization instruction (more akin to a barrier)
- Existing approaches create a heavy-weight state machine of all synchronizations that stores all values

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.

```
parallel_for %i = 0 to N {  
  %off = %i + 1  
  codeA(%off);  
  sync_threads;  
  codeB(%off);  
}
```

```
%offm = alloca N  
parallel_for %i = 0 to N {  
  %off = %i + 1  
  %offm[%i] = %off  
  codeA(%off);  
}  
parallel_for %i = 0 to N {  
  codeB(%off_m[%i]);  
}
```

```
parallel_for %i = 0 to N {  
  %off = %i + 1  
  codeA(%off);  
}  
parallel_for %i = 0 to N {  
  %off = %i + 1  
  codeB(%off);  
}
```

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.
- ***[Question] Is distributing the parallelism around the barrier the best approach?***
- ***[Question] How do we minimize the runtime of preserving registers?***
 - Tradeoff parallel recompute vs preserve
 - Min Cut?

GPU Synchronization Lowering: Control Flow

- Synchronization within control flow (for, if, while, etc) can be lowered by splitting around the control flow and interchanging the parallelism.

```
parallel_for %i = 0 to N {  
  codeA(%i);  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
  codeC(%i);  
}
```

```
parallel_for %i = 0 to N {  
  codeA(%i);  
  sync_threads;  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
  sync_threads;  
  codeC(%i);  
}
```

```
parallel_for %i = 0 to N {  
  codeA(%i);  
}  
parallel_for %i = 0 to N {  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
}  
parallel_for %i = 0 to N {  
  codeC(%i);  
}
```

GPU Synchronization Lowering: Control Flow

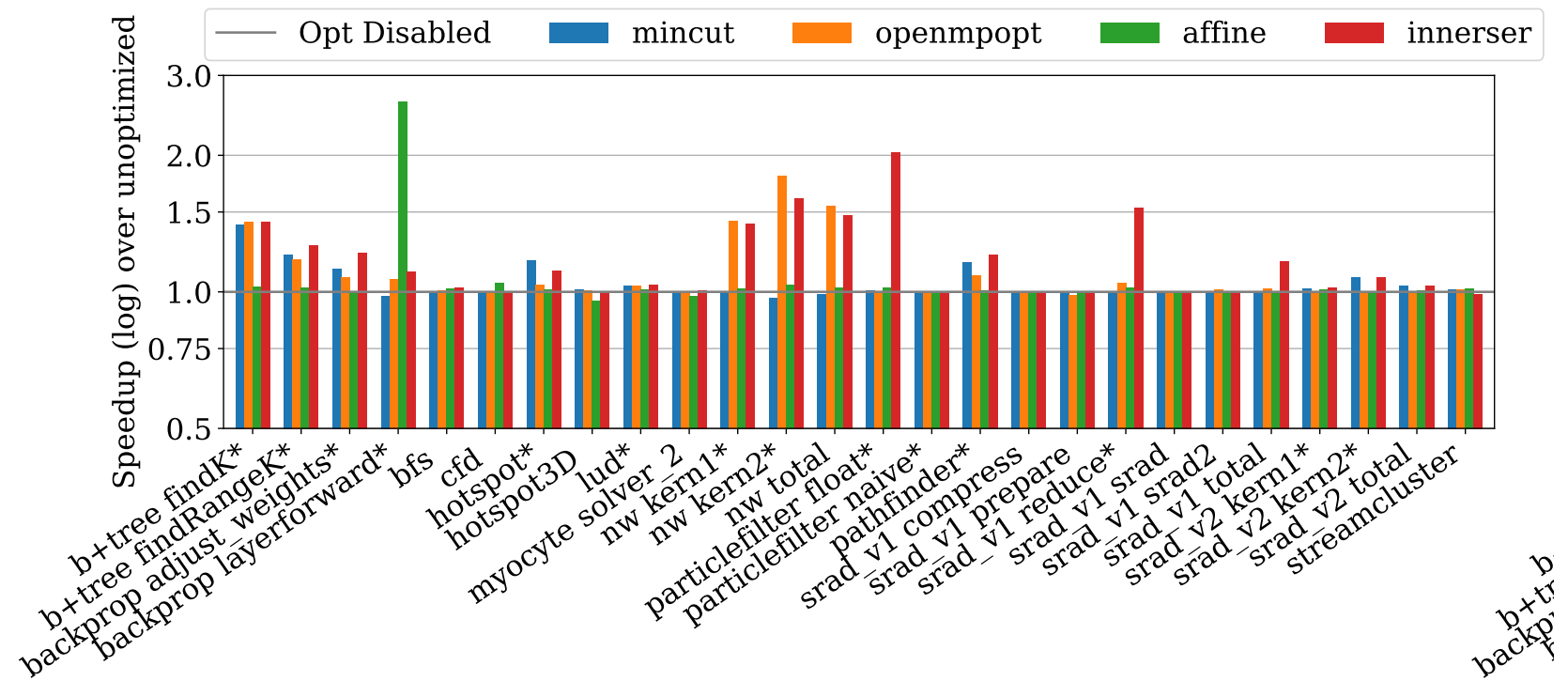
- Synchronization within control flow (for, if, while, etc) can be lowered by splitting around the control flow and interchanging the parallelism.

```
parallel_for %i = 0 to N {  
    codeA(%i);  
}  
parallel_for %i = 0 to N {  
    for %j = ... {  
        codeB1(%i, %j);  
        sync_threads;  
        codeB2(%i, %j);  
    }  
}  
parallel_for %i = 0 to N {  
    codeC(%i);  
}
```

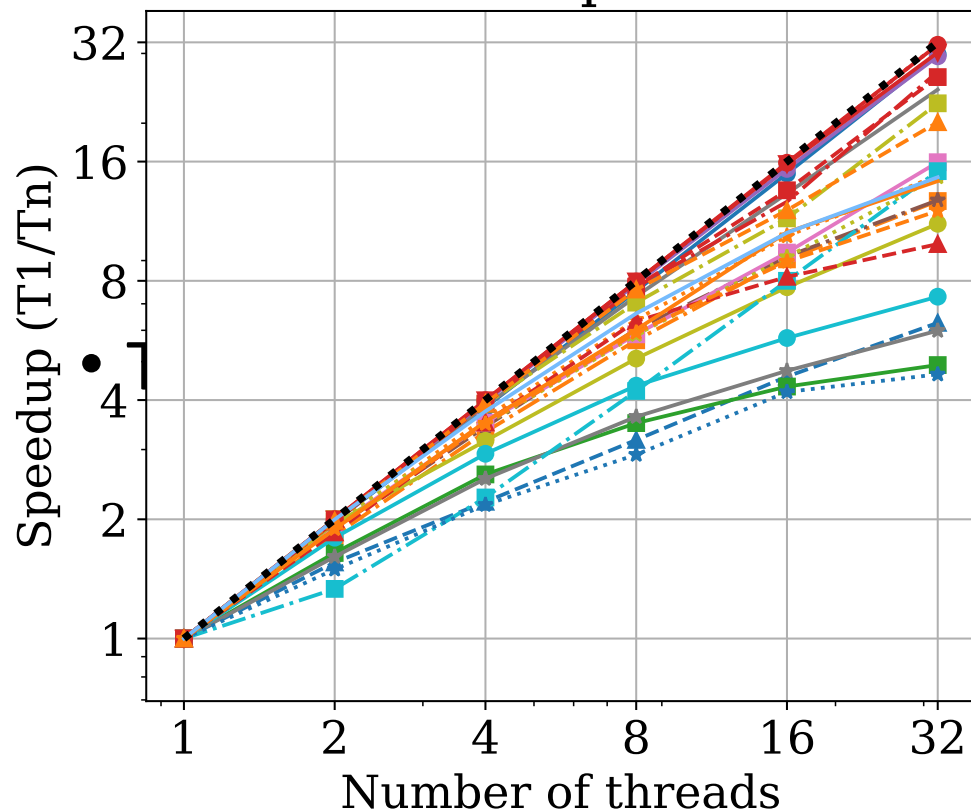
```
parallel_for %i = 0 to N {  
    codeA(%i);  
}  
for %j = ... {  
    parallel_for %i = 0 to N {  
        codeB1(%i, %j);  
        sync_threads;  
        codeB2(%i, %j);  
    }  
}  
parallel_for %i = 0 to N {  
    codeC(%i);  
}
```

```
parallel_for %i = 0 to N {  
    codeA(%i);  
}  
for %j = ... {  
    parallel_for %i = 0 to N {  
        codeB1(%i, %j);  
    }  
    parallel_for %i = 0 to N {  
        codeB2(%i, %j);  
    }  
}  
parallel_for %i = 0 to N {  
    codeC(%i);  
}
```

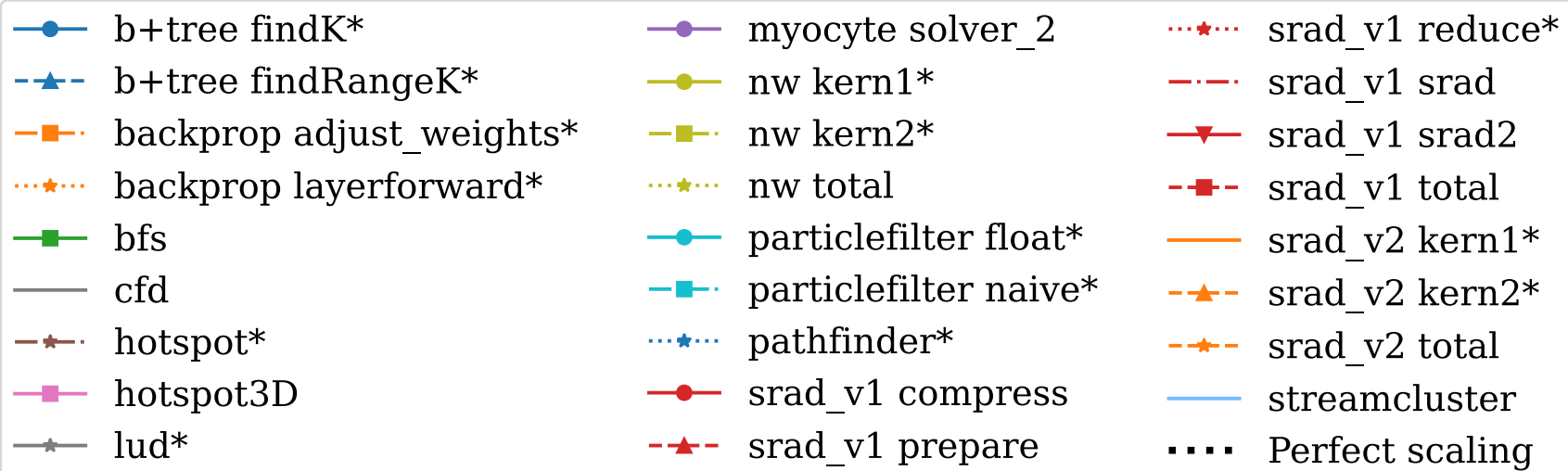
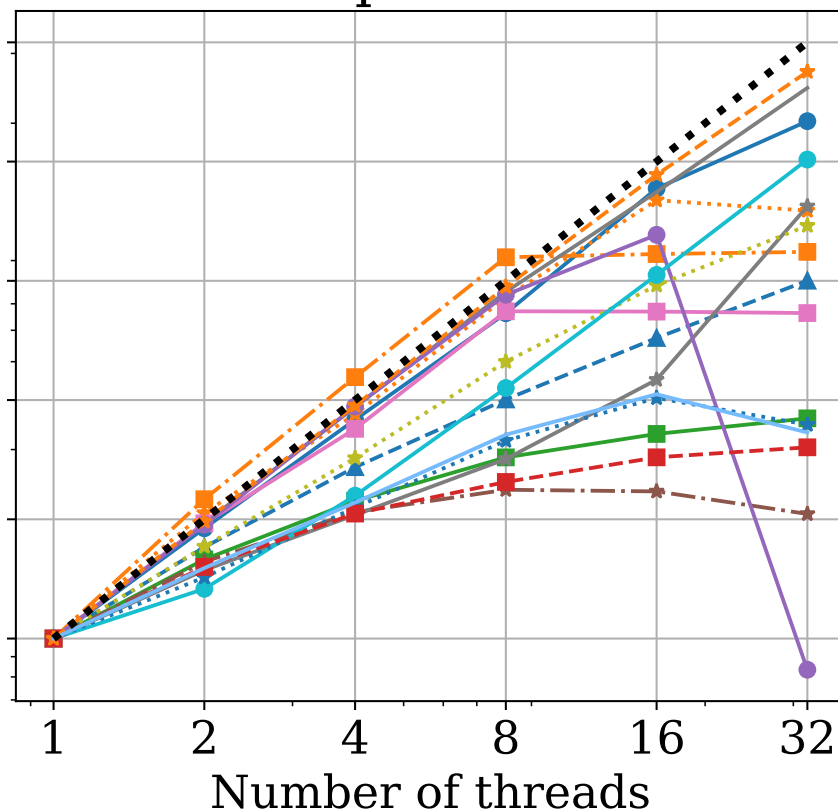
- Text



CUDA-OpenMP

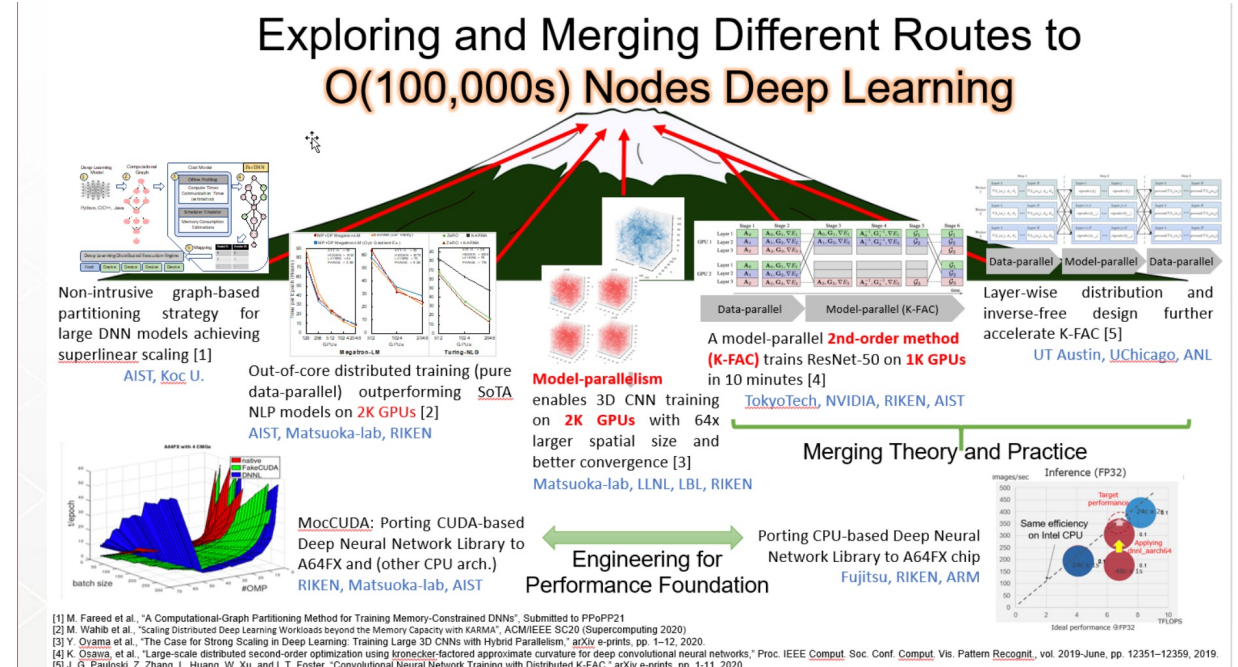


OpenMP



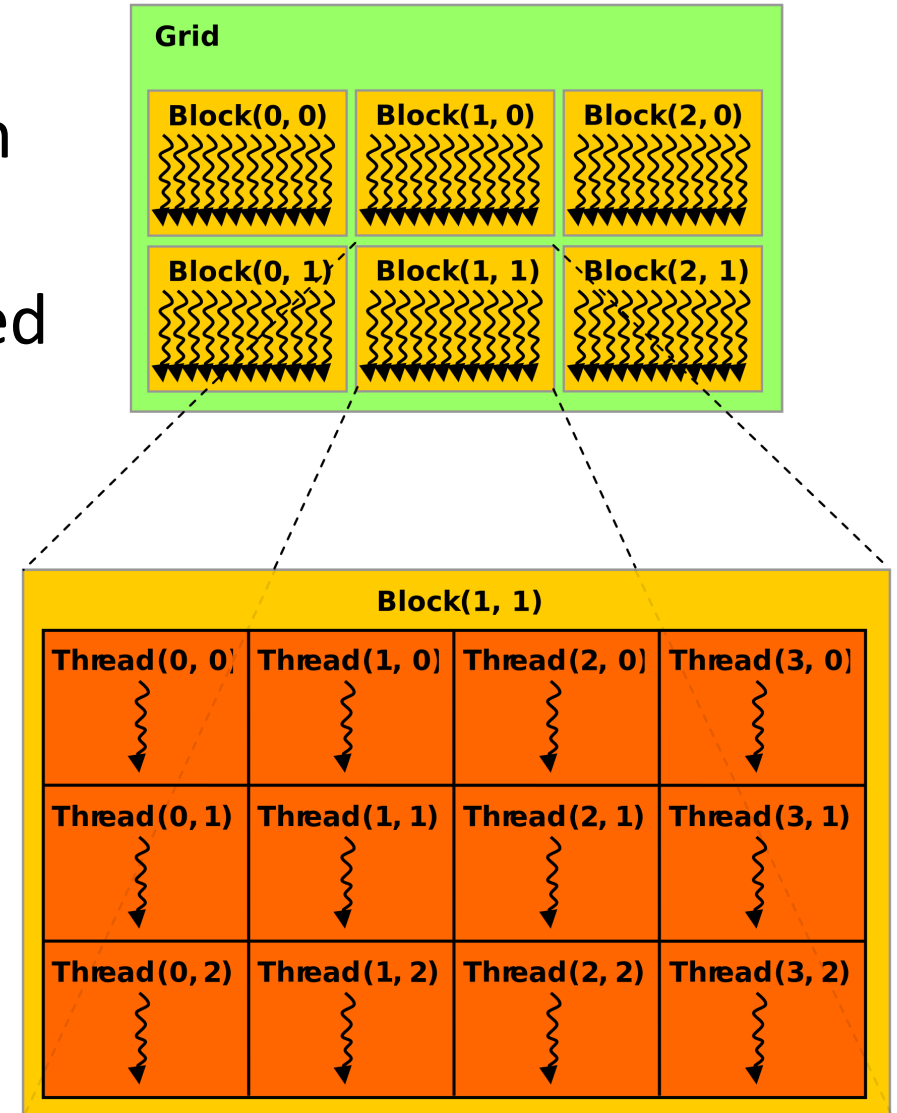
Open Research Directions

- How can we optimize GPU programs?
- Can we convert GPU to CPU (and vice versa)?
 - Working with Riken/Tokyo Tech to port GPU to Fugaku supercomputer
- What advantages can we gain from compiler representations?

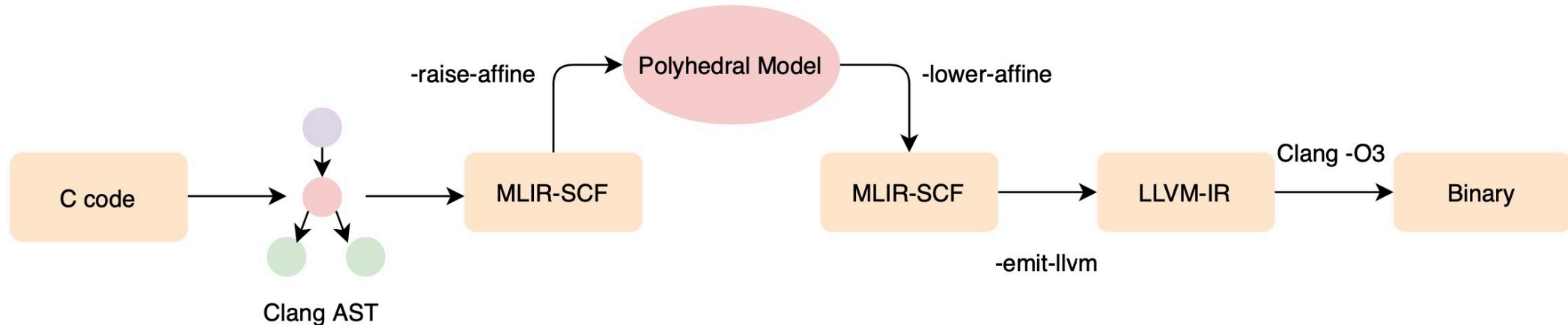


Introduction GPU Programming

- GPU threads are like CPU threads in which they can run in parallel.
- A group of threads (up to 32) are combined in a block
- Threads can share data and/or sync within a block but not between blocks
- All threads in a block are guaranteed to execute at the same time (and may run in lockstep)
- Blocks are not



The Polygeist Compilation Flow



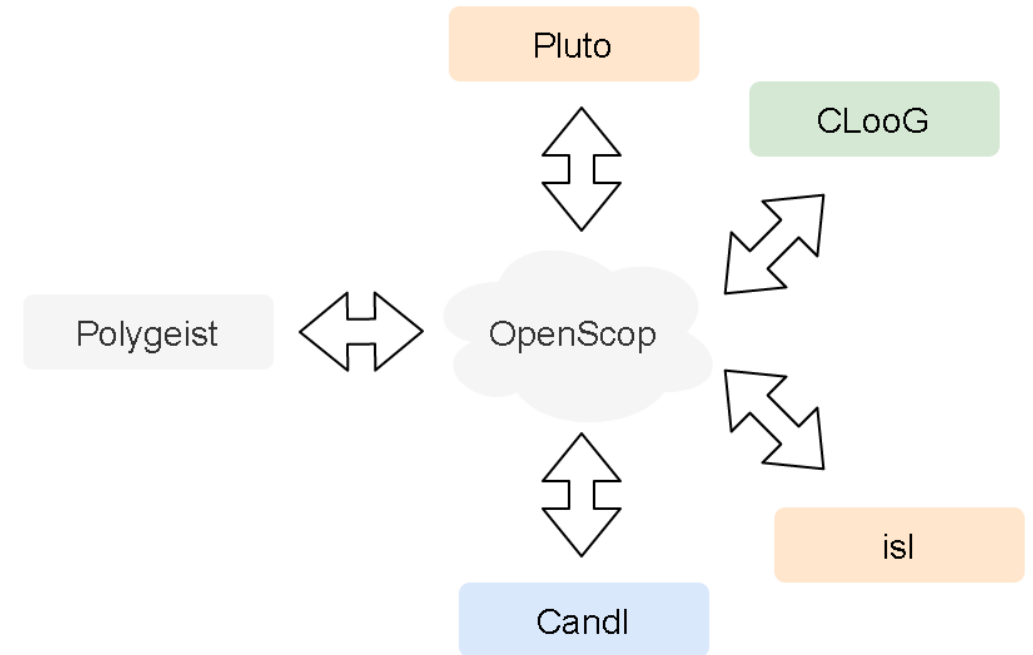
- Generic C or C++ frontend that generates "standard" MLIR
- Raising transformations for transforming "standard" MLIR to polyhedral MLIR (Affine)
- Embedding of existing polyhedral tools (Pluto, CLoog) into MLIR
- Novel transformations (statement splitting, reduction detection) that rely on high-level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

Polygeist on Polyhedral

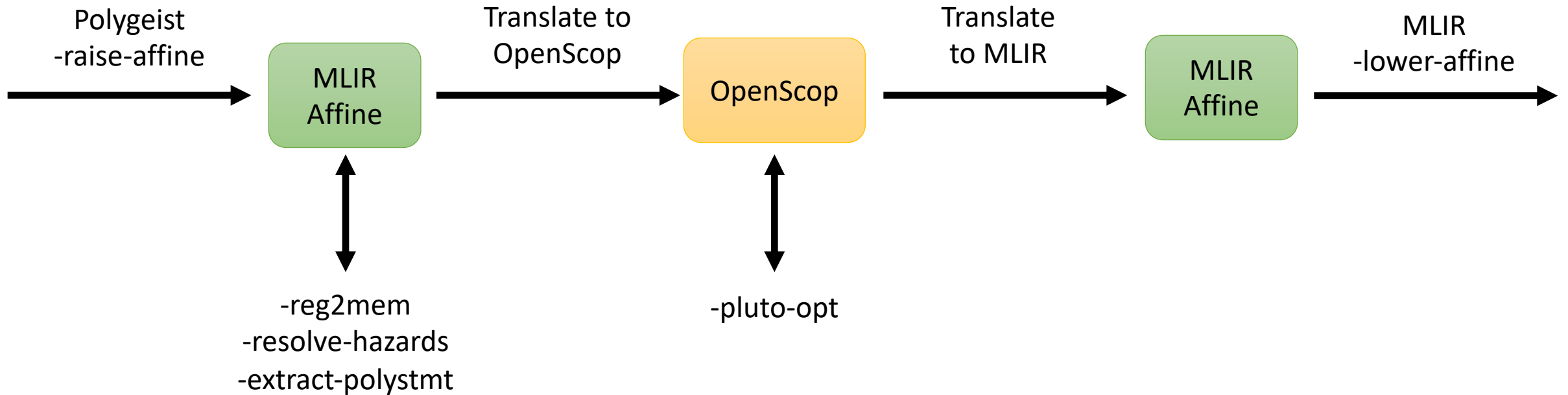
- An MLIR-based end-to-end polyhedral compilation flow
- Keeps the best parts of high-level abstractions and low-level optimizations by directly lowering to and optimizing MLIR.
- Multiple abstraction levels allows new polyhedral optimization opportunities and simplifies the implementation of others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

Connecting MLIR to Polyhedral Tools

- Polygeist produces MLIR **Affine**
- MLIR **Affine** $\leftrightarrow$ polyhedral model
- Existing tools don't take **Affine**
- **OpenScop** is a target representation
- How to translate **Affine** $\leftrightarrow$ **OpenScop**?



Polyhedral Optimization Pipeline

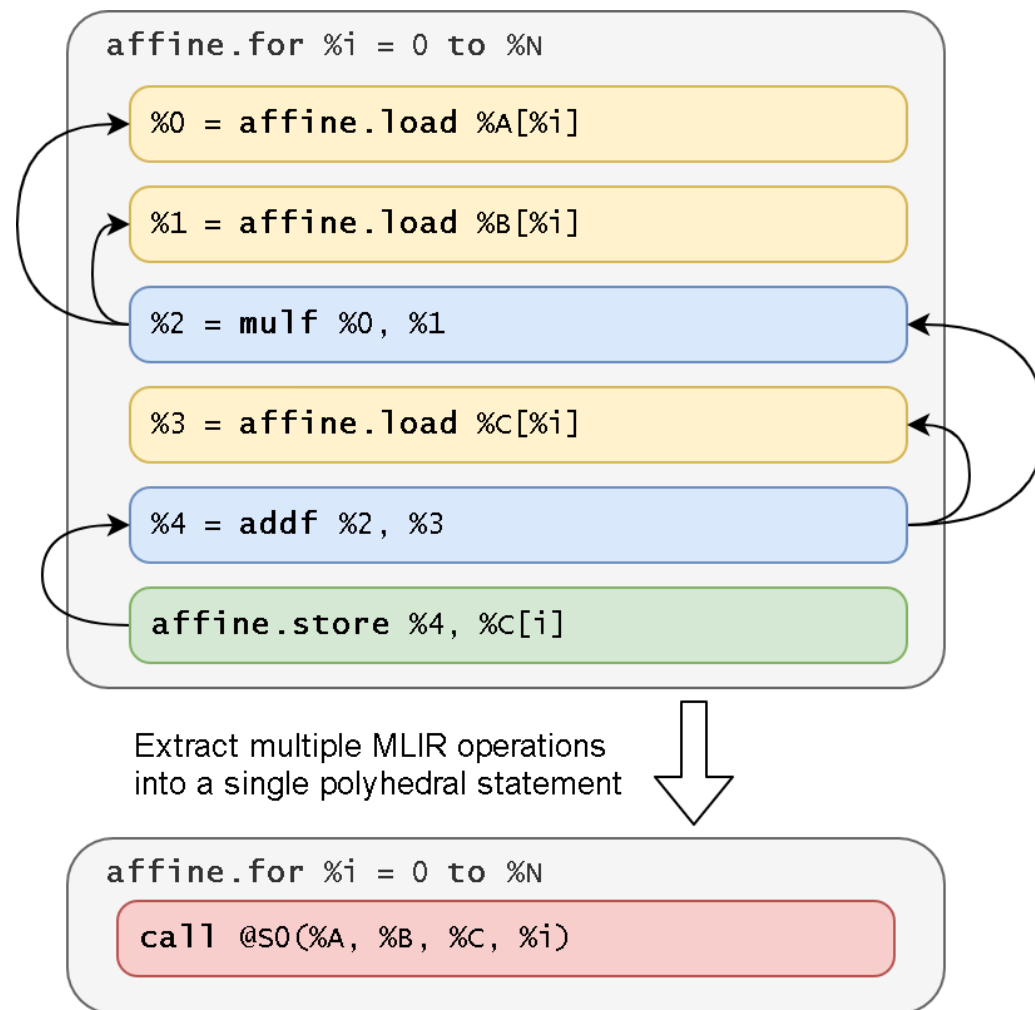


Polyhedral Statement

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

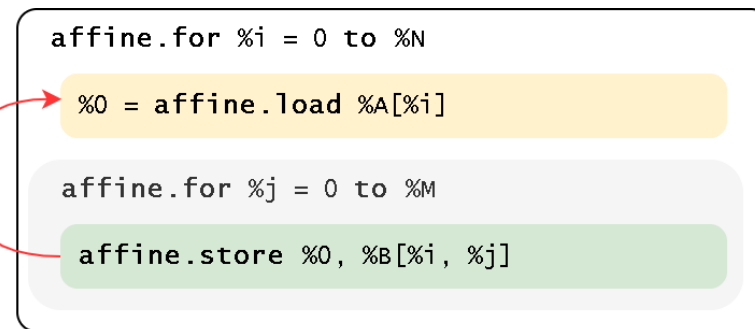
```
C[i][j] += A[i][k] * B[k][j]
```

- 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

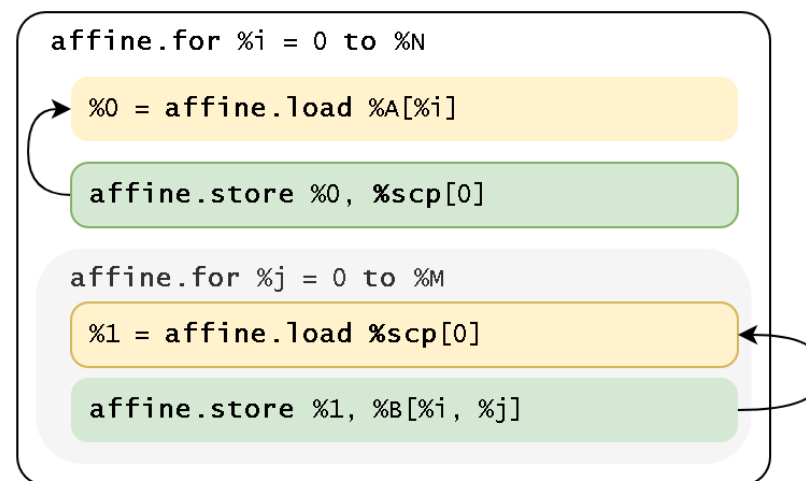


Region-Spanning Problem

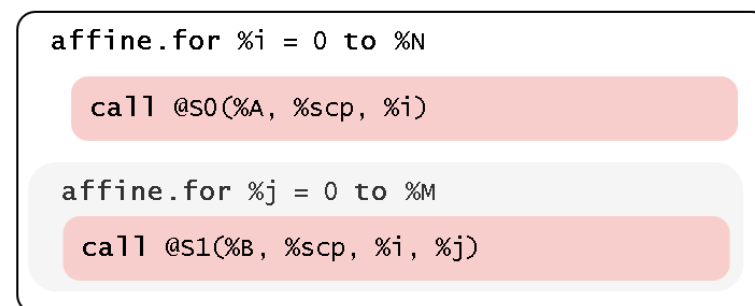
- A use-def chain spans across loops
- Statement nesting is **ambiguous**
- MLIR reconstruction is **difficult**
- Reg2mem pass: insert a **scratchpad** for each region-spanning use-def chain



The **reg2mem** pass that
inserts a scratchpad

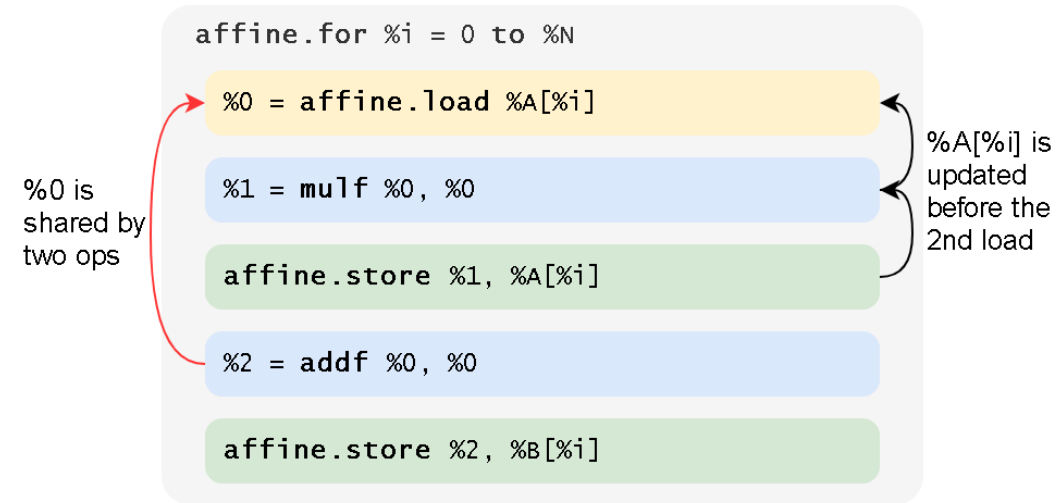


Extract polyhedral
statements

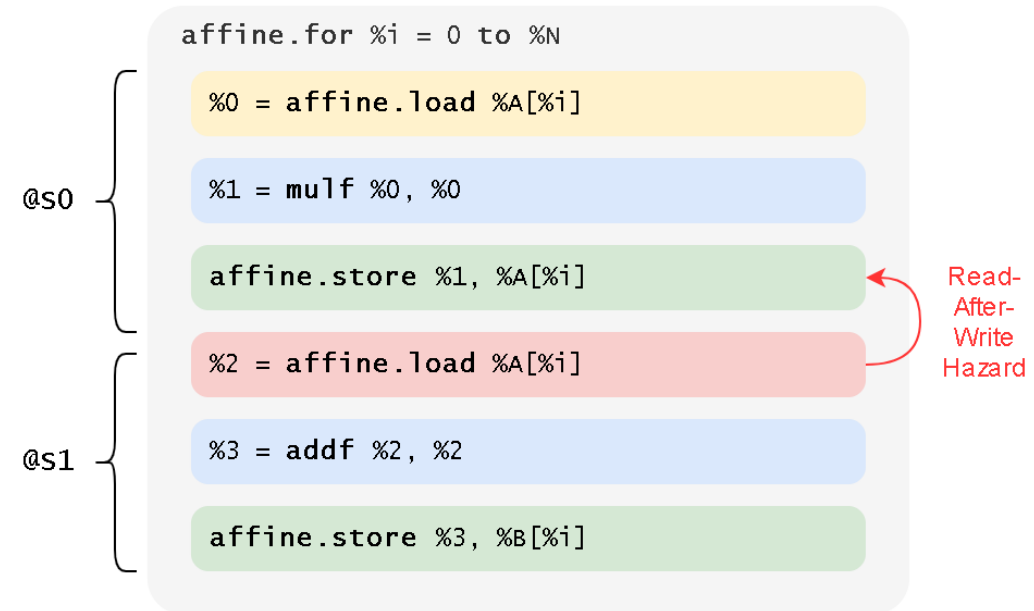


Avoid RAW Hazard

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



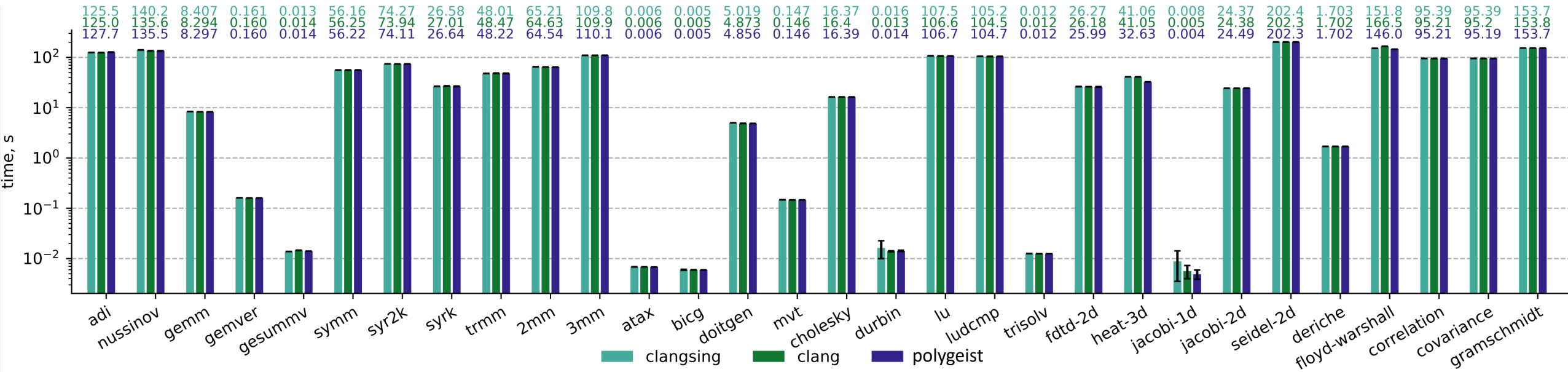
Duplicating shared operation
may introduce hazards



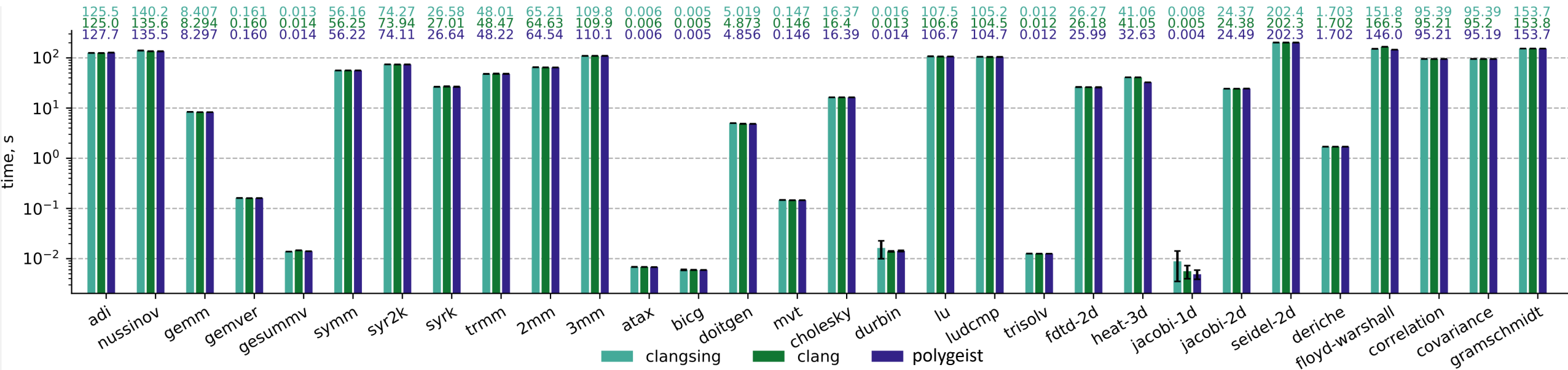
Evaluation

- 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

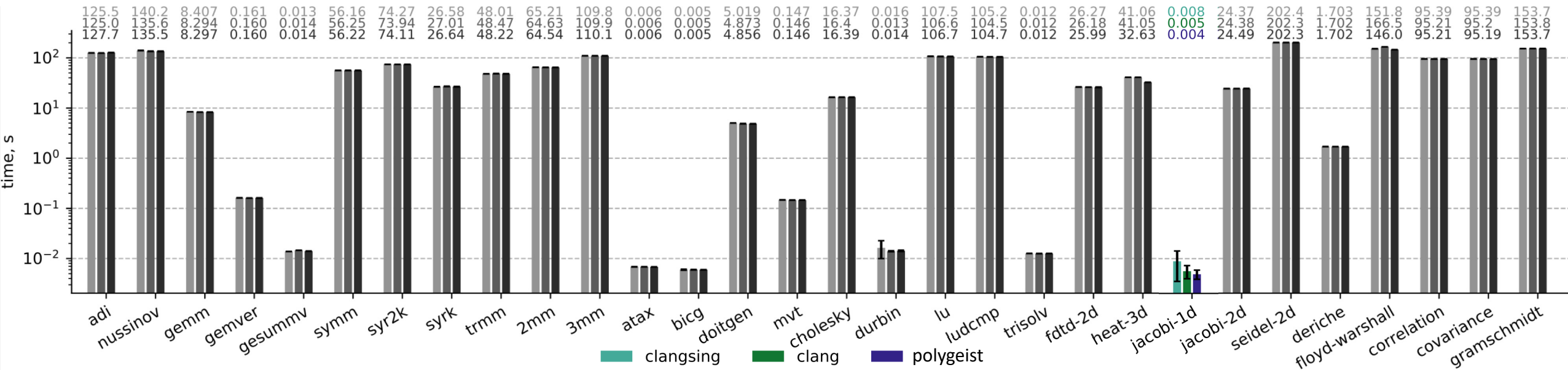


Serial Non-Polyhedral Comparison



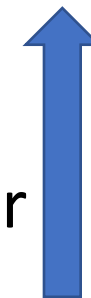
Frontend within 0.32% of
“standard” frontend

Serial Non-Polyhedral Comparison

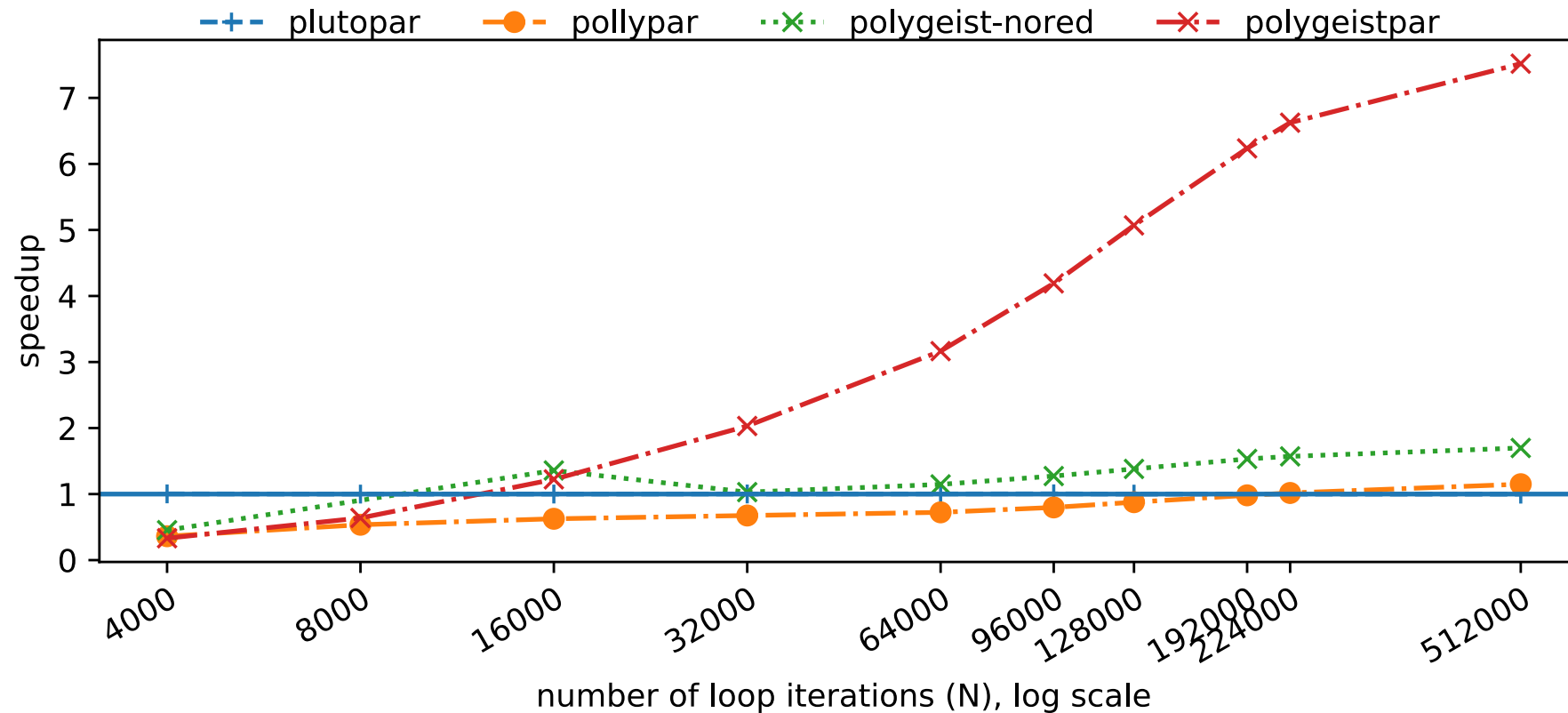


Frontend within 0.32% of
“standard” frontend

Remaining gap attributed
to small tests where minor
assembly differences
matter



Parallel Reduction Detection (durbin)



Statement Splitting

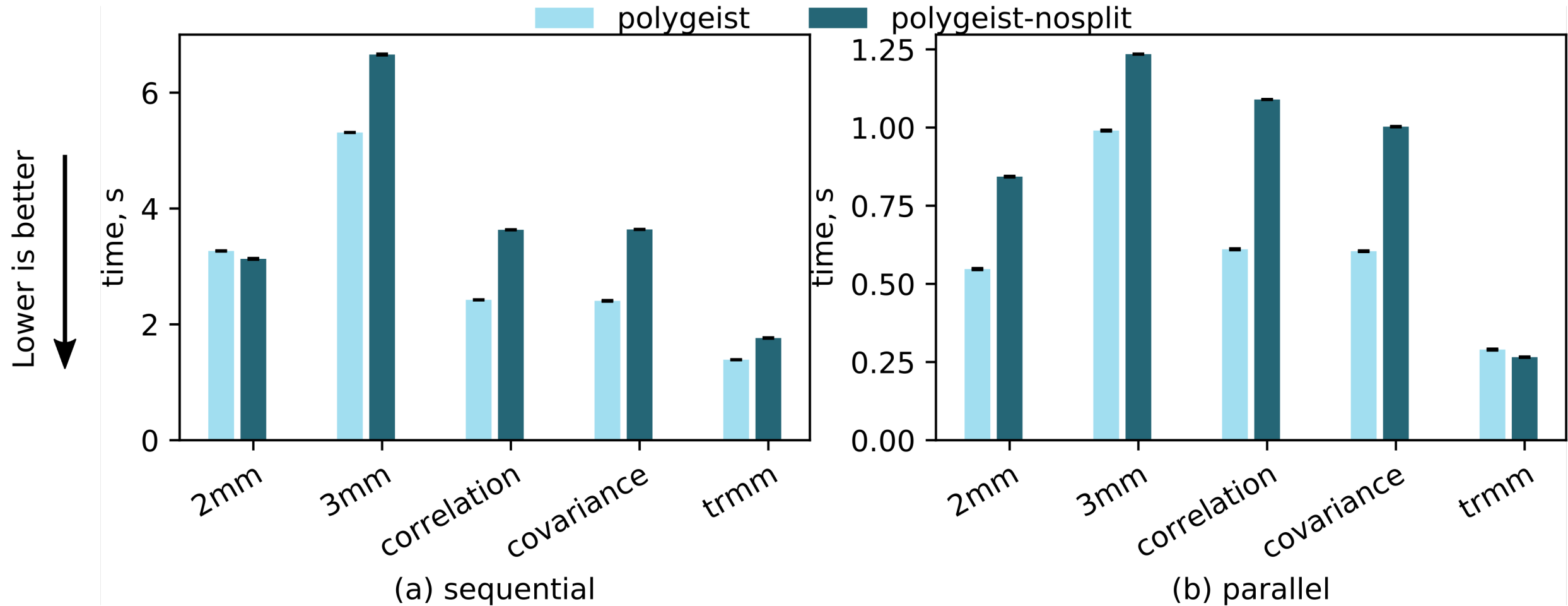
- We don't **need** to reconstruct the **original** C statements from Affine
- We can **split** statements by inserting **scratchpads**.

```
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]);
```

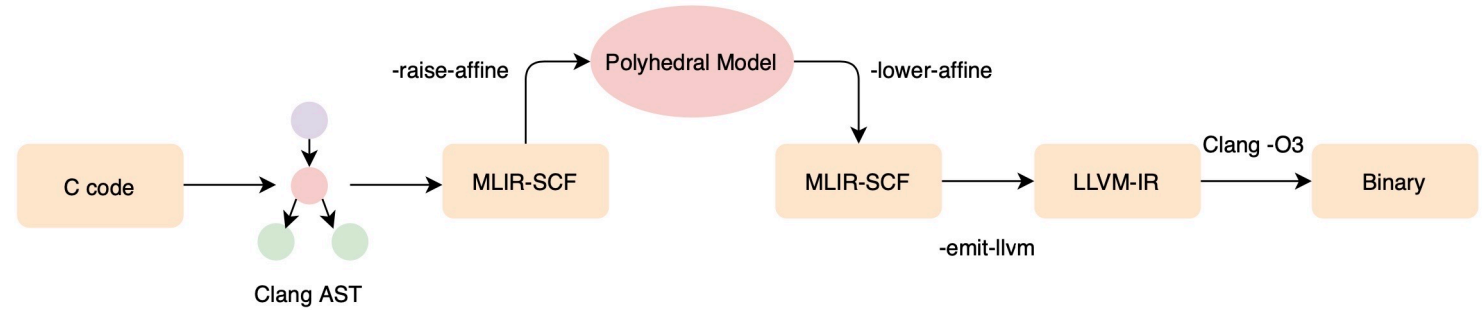
⇓

<pre>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];</pre>	⇒	<pre>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];</pre>
--	---	--

Statement Splitting



Future Work



- Exploration of Statement Splitting beyond a simple heuristic
- GPU optimization and GPU $\leftrightarrow$ 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)

“Case Study 3”: Your Programs!

- There are already several efforts starting using Polygeist/MLIR to leveraging the benefits of optimizable multi-level operations
 - SYCL
 - Circuit Compilation
 - BLAS Kernels
 - Databases
 - ...
- If you're interested in applying such techniques to your programs, please reach out!

GPU Synchronization Lowering

- Most CPU backends (e.g. Cilk, OpenMP) do not have an equivalent & general synchronization instruction (more akin to a barrier)
- Existing approaches create a heavy-weight state machine of all synchronizations that stores all values

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.

```
parallel_for %i = 0 to N {  
  %off = %i + 1  
  codeA(%off);  
  sync_threads;  
  codeB(%off);  
}
```

```
%offm = alloca N  
parallel_for %i = 0 to N {  
  %off = %i + 1  
  %offm[%i] = %off  
  codeA(%off);  
}  
parallel_for %i = 0 to N {  
  codeB(%off_m[%i]);  
}
```

```
parallel_for %i = 0 to N {  
  %off = %i + 1  
  codeA(%off);  
}  
parallel_for %i = 0 to N {  
  %off = %i + 1  
  codeB(%off);  
}
```

GPU Synchronization Lowering: Registers

- Registers defined before the synchronization and used after the synchronization must be preserved through an allocation.
- If the memory semantics allow us to more efficiently recompute the value, it doesn't need to be stored.
- ***[Question] Is distributing the parallelism around the barrier the best approach?***
- ***[Question] How do we minimize the runtime of preserving registers?***
 - Tradeoff parallel recompute vs preserve
 - Min Cut?

GPU Synchronization Lowering: Control Flow

- Synchronization within control flow (for, if, while, etc) can be lowered by splitting around the control flow and interchanging the parallelism.

```
parallel_for %i = 0 to N {  
  codeA(%i);  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
  codeC(%i);  
}
```

```
parallel_for %i = 0 to N {  
  codeA(%i);  
  sync_threads;  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
  sync_threads;  
  codeC(%i);  
}
```

```
parallel_for %i = 0 to N {  
  codeA(%i);  
}  
parallel_for %i = 0 to N {  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
}  
parallel_for %i = 0 to N {  
  codeC(%i);  
}
```

GPU Synchronization Lowering: Control Flow

- Synchronization within control flow (for, if, while, etc) can be lowered by splitting around the control flow and interchanging the parallelism.

```
parallel_for %i = 0 to N {  
    codeA(%i);  
}  
parallel_for %i = 0 to N {  
    for %j = ... {  
        codeB1(%i, %j);  
        sync_threads;  
        codeB2(%i, %j);  
    }  
}  
parallel_for %i = 0 to N {  
    codeC(%i);  
}
```

```
parallel_for %i = 0 to N {  
    codeA(%i);  
}  
for %j = ... {  
    parallel_for %i = 0 to N {  
        codeB1(%i, %j);  
        sync_threads;  
        codeB2(%i, %j);  
    }  
}  
parallel_for %i = 0 to N {  
    codeC(%i);  
}
```

```
parallel_for %i = 0 to N {  
    codeA(%i);  
}  
for %j = ... {  
    parallel_for %i = 0 to N {  
        codeB1(%i, %j);  
    }  
    parallel_for %i = 0 to N {  
        codeB2(%i, %j);  
    }  
}  
parallel_for %i = 0 to N {  
    codeC(%i);  
}
```