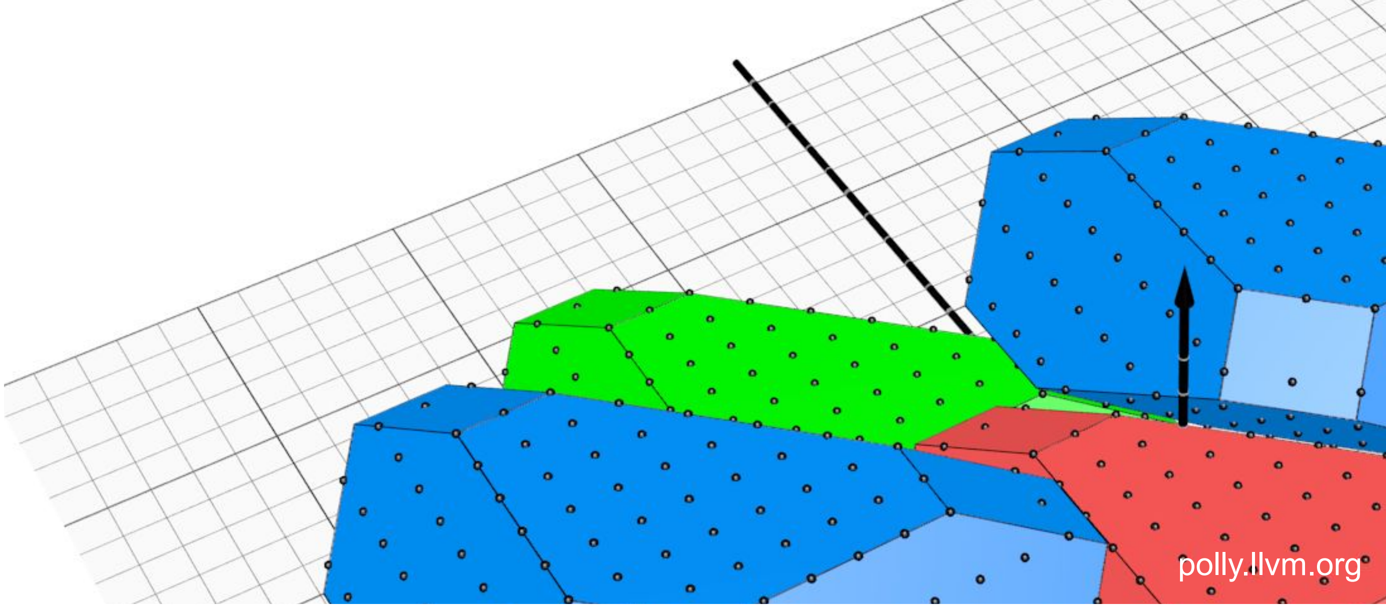


Scalable Polyhedral Compilation in Open-Source and AI Compilers

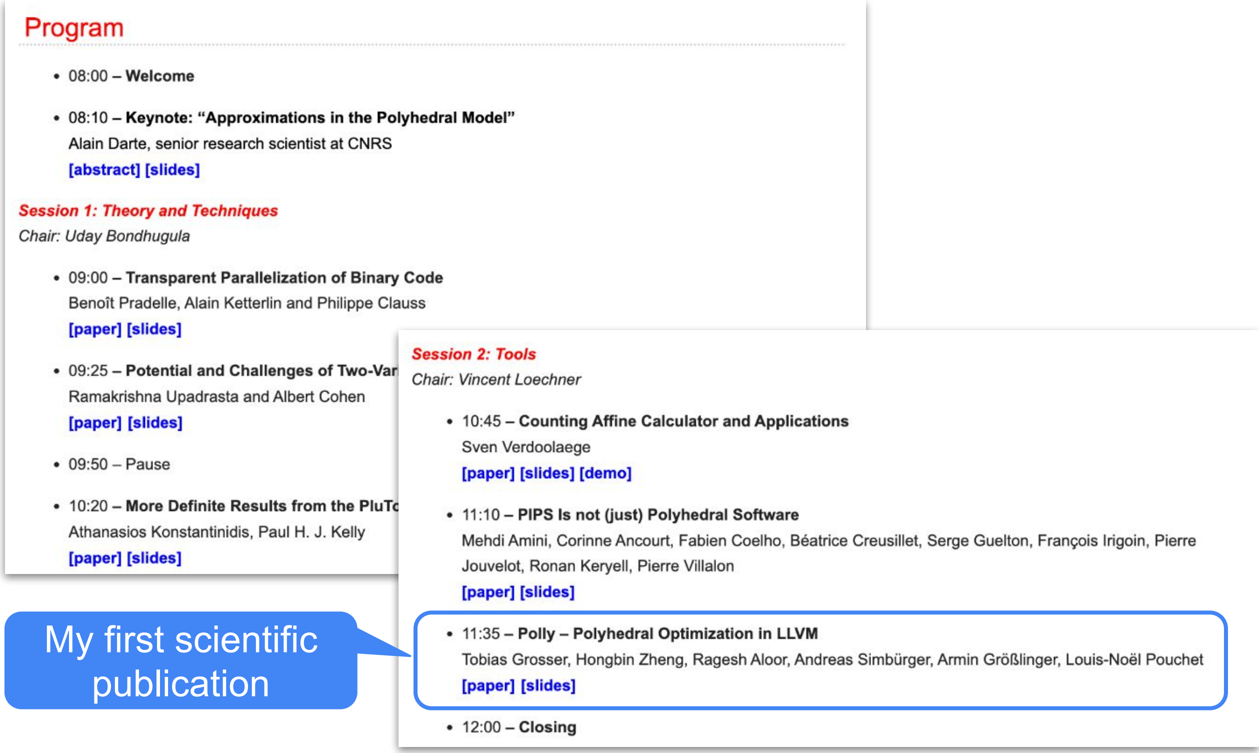
Tobias Grosser

UNIVERSITY OF CAMBRIDGE

IMPACT

First International Workshop on Polyhedral Compilation Techniques

In conjunction with **CGO 2011**



Program

- 08:00 – **Welcome**
- 08:10 – **Keynote: “Approximations in the Polyhedral Model”**
  Alain Darte, senior research scientist at CNRS
  [abstract] [slides]

Session 1: Theory and Techniques
Chair: Uday Bondhugula

- 09:00 – **Transparent Parallelization of Binary Code**
  Benoît Pradelle, Alain Ketterlin and Philippe Clauss
  [paper] [slides]
- 09:25 – **Potential and Challenges of Two-Var**
  Ramakrishna Upadrasta and Albert Cohen
  [paper] [slides]
- 09:50 – Pause
- 10:20 – **More Definite Results from the PluTo**
  Athanasios Konstantinidis, Paul H. J. Kelly
  [paper] [slides]

Session 2: Tools
Chair: Vincent Loechner

- 10:45 – **Counting Affine Calculator and Applications**
  Sven Verdoolaege
  [paper] [slides] [demo]
- 11:10 – **PIPS Is not (just) Polyhedral Software**
  Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, Pierre Villalon
  [paper] [slides]
- 11:35 – **Polly – Polyhedral Optimization in LLVM**
  Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, Louis-Noël Pouchet
  [paper] [slides]
- 12:00 – **Closing**

My first scientific publication

UNIVERSITY OF CAMBRIDGE

2011

1794

ENS

ÉCOLE NORMALE SUPÉRIEURE

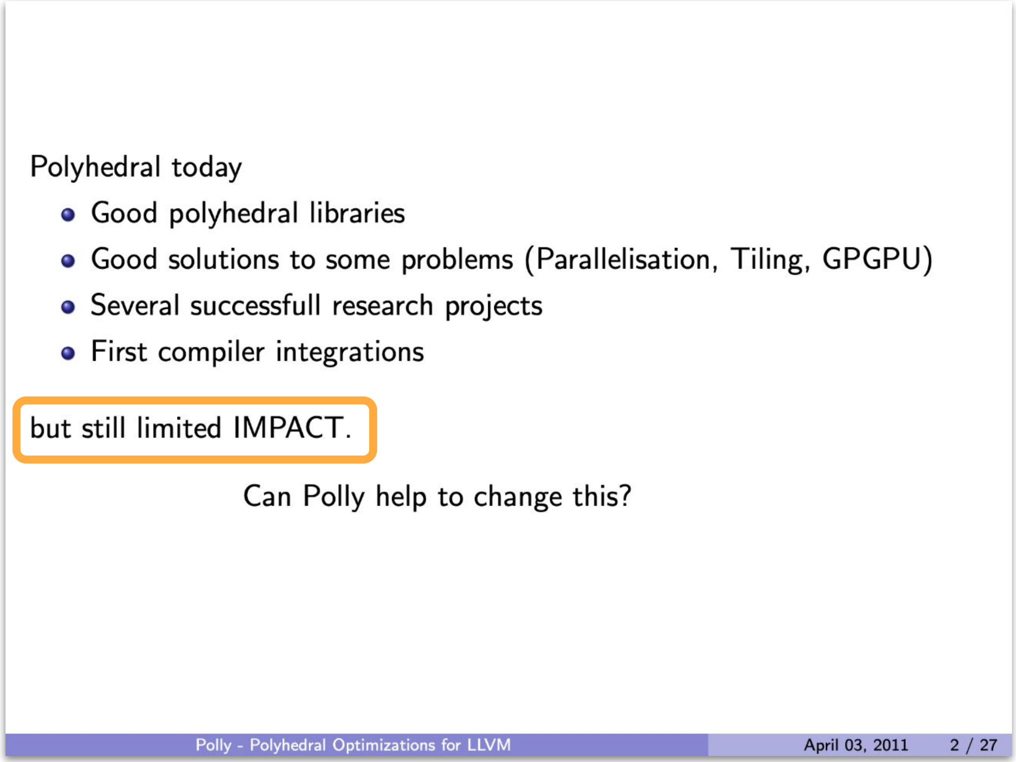
CARP

CARP: Correct and Efficient Accelerator Programming

Google Research

PhD Fellowship

Polly - Polyhedral Optimization in LLVM @ IMPACT 2011



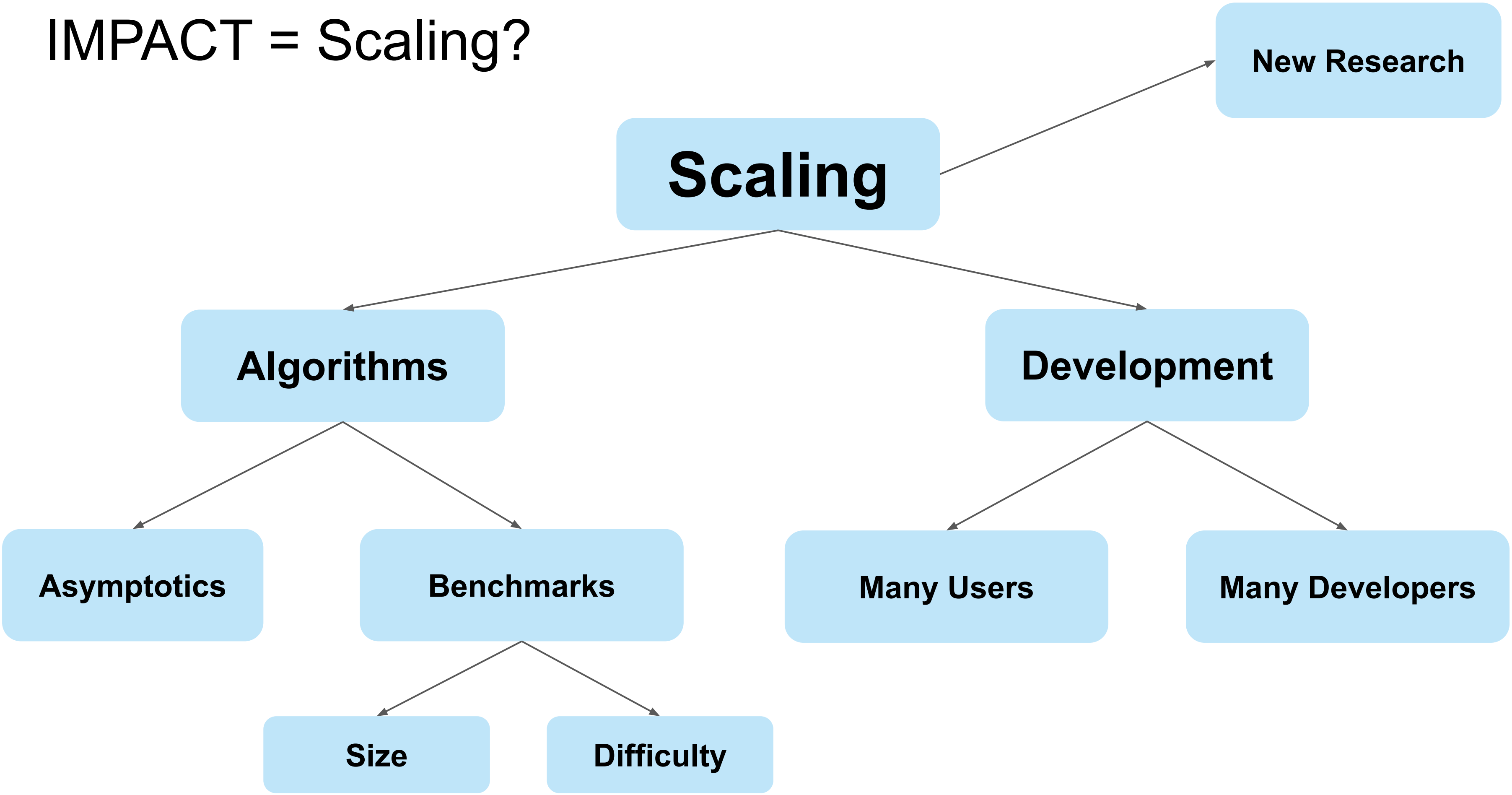
Polyhedral today

- Good polyhedral libraries
- Good solutions to some problems (Parallelisation, Tiling, GPGPU)
- Several successfull research projects
- First compiler integrations

but still limited IMPACT.

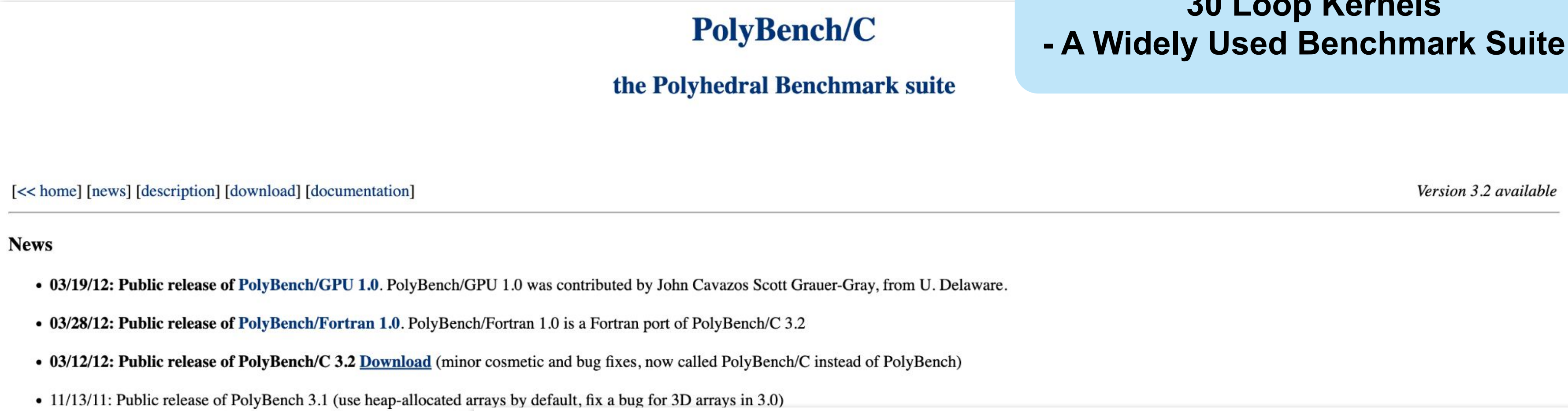
Can Polly help to change this?

IMPACT = Scaling?



Polybench - Likely our most widely used artifact

30 Loop Kernels
- A Widely Used Benchmark Suite -



PolyBench/C

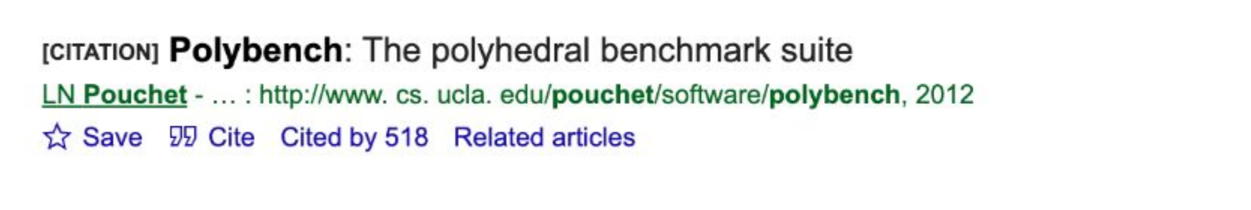
the Polyhedral Benchmark suite

[<< home] [news] [description] [download] [documentation]

Version 3.2 available

News

- **03/19/12: Public release of PolyBench/GPU 1.0**. PolyBench/GPU 1.0 was contributed by John Cavazos Scott Grauer-Gray, from U. Delaware.
- **03/28/12: Public release of PolyBench/Fortran 1.0**. PolyBench/Fortran 1.0 is a Fortran port of PolyBench/C 3.2
- **03/12/12: Public release of PolyBench/C 3.2** Download (minor cosmetic and bug fixes, now called PolyBench/C instead of PolyBench)
- 11/13/11: Public release of PolyBench 3.1 (use heap-allocated arrays by default, fix a bug for 3D arrays in 3.0)
- 10/28/11: Public release of PolyBench 3.0 (support of heap-allo
- 3/16/11: Public release of PolyBench 2.0 (superset of 1.0 + C99
- 4/12/10: Public release of PolyBench 1.0



[CITATION] **Polybench**: The polyhedral benchmark suite

LN **Pouchet** - … : http://www. cs. ucla. edu/**pouchet**/software/**polybench**, 2012

☆ Save 99 Cite Cited by 518 Related articles

Polybench - A Dream Benchmark

Small Kernels
5-50 lines
3-10 loops

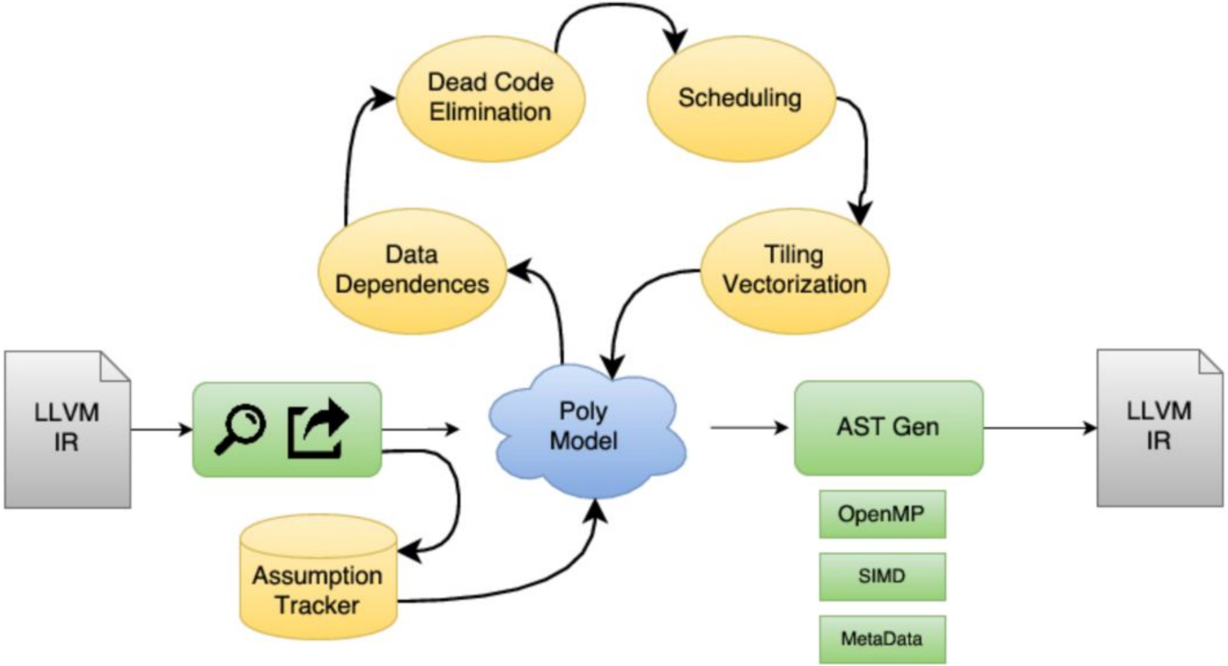
Well-Behaved
No Integer Wrapping
No-Unbounded Loops

Structured Control
Loops + If-Conditions
(Im)perfectly Nested

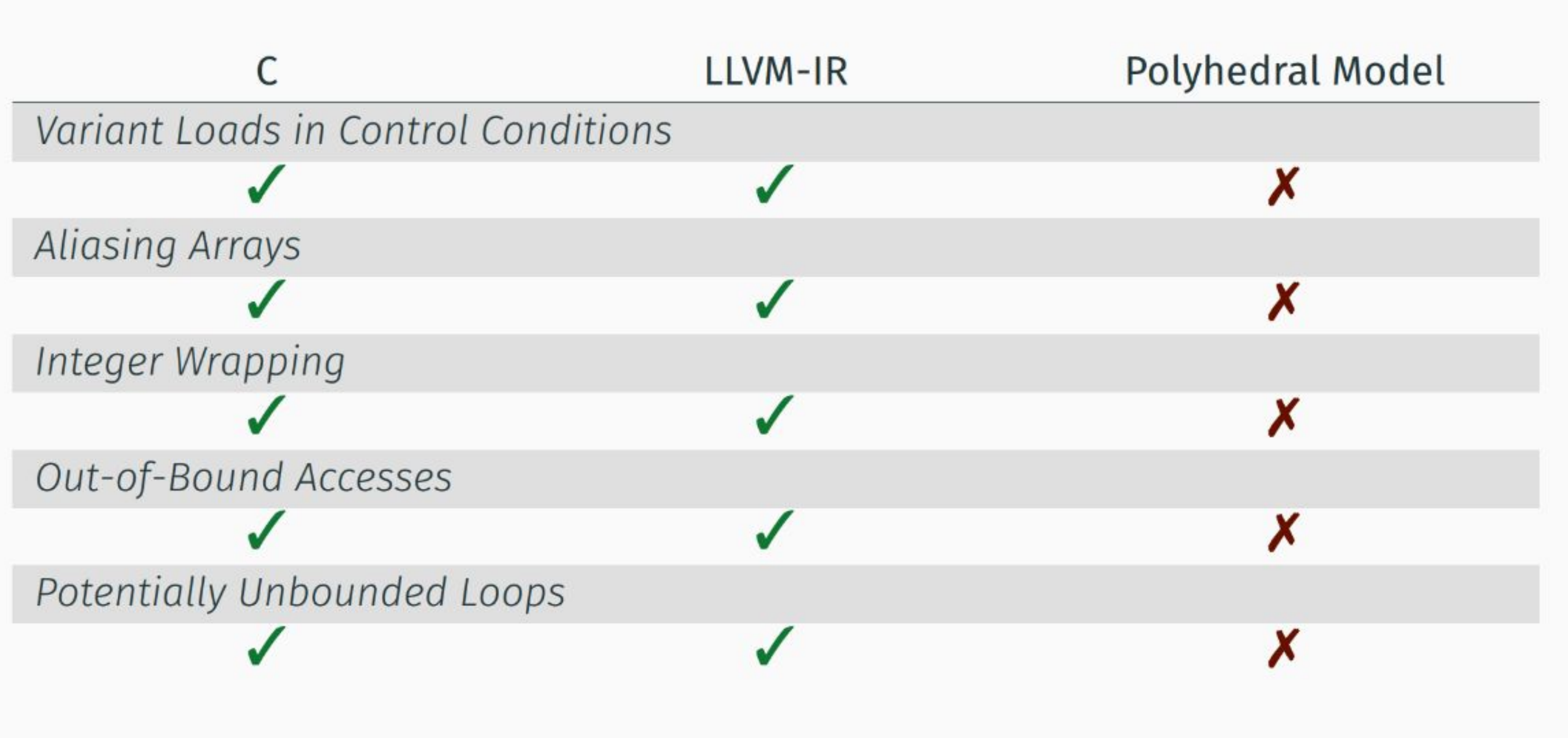
Diverse Kernels

Challenging to Optimize

Polly - A polyhedral Compiler for LLVM-IR



Challenges for Polly



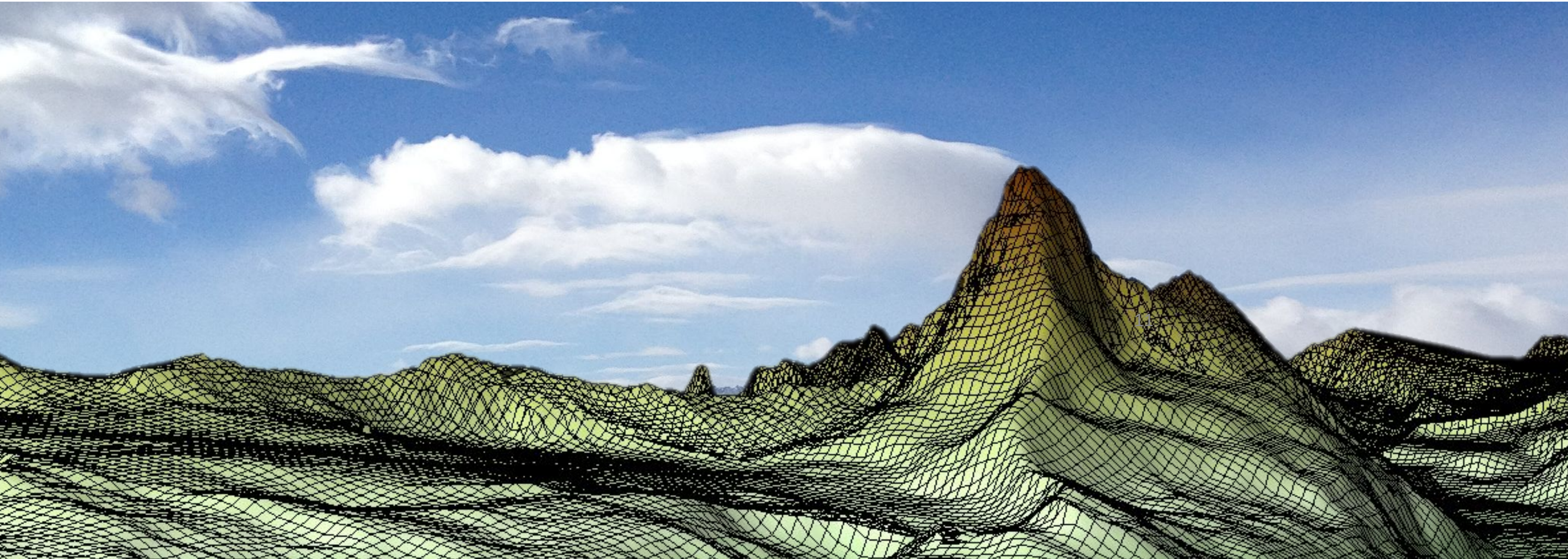
C	LLVM-IR	Polyhedral Model
Variant Loads in Control Conditions		
✓	✓	✗
Aliasing Arrays		
✓	✓	✗
Integer Wrapping		
✓	✓	✗
Out-of-Bound Accesses		
✓	✓	✗
Potentially Unbounded Loops		
✓	✓	✗

The COSMO Atmospheric Model

Used by 7 national weather services (DE, CH, IT, …)

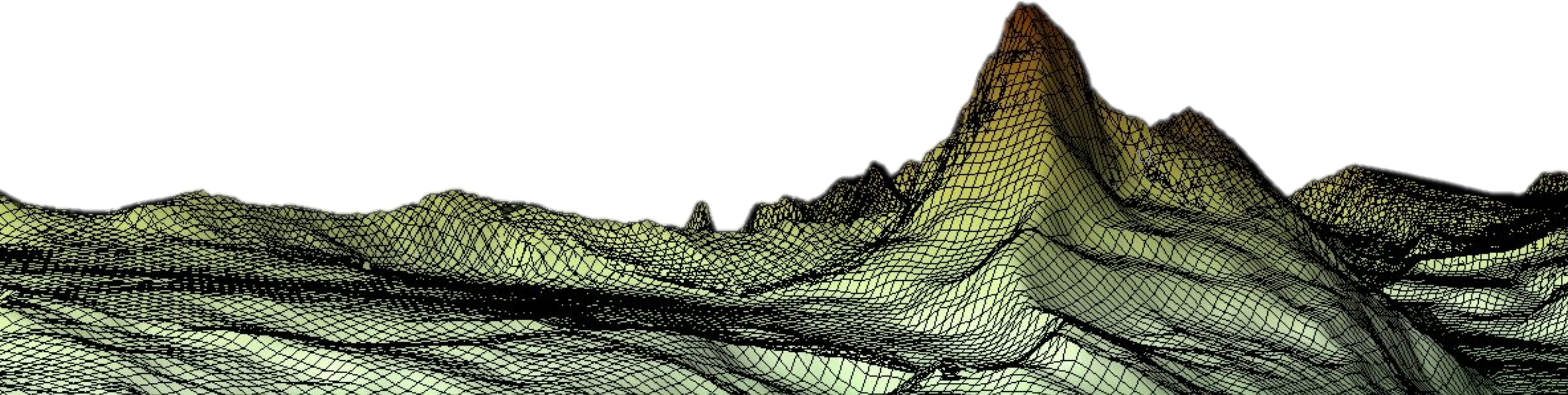
Resolution: 35m

What resolution is needed?



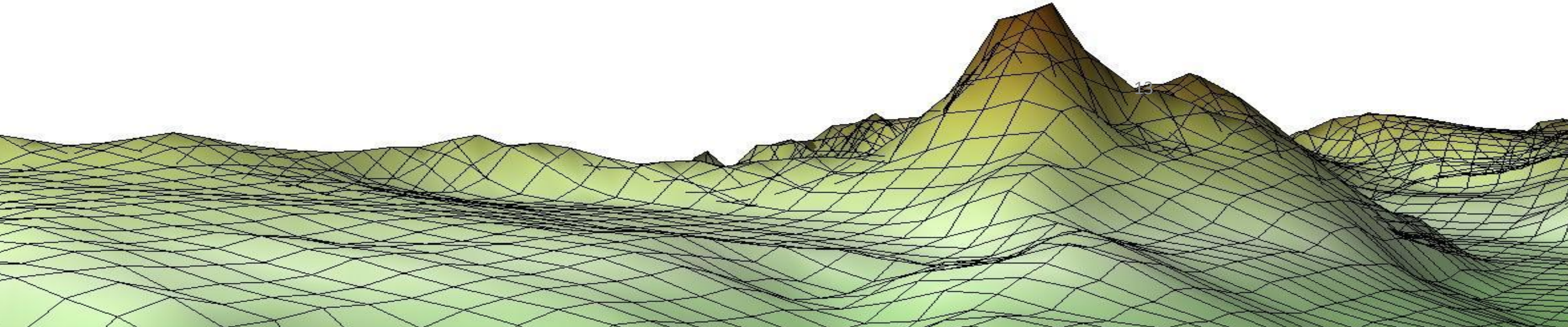
Resolution: 35m

What resolution is needed?



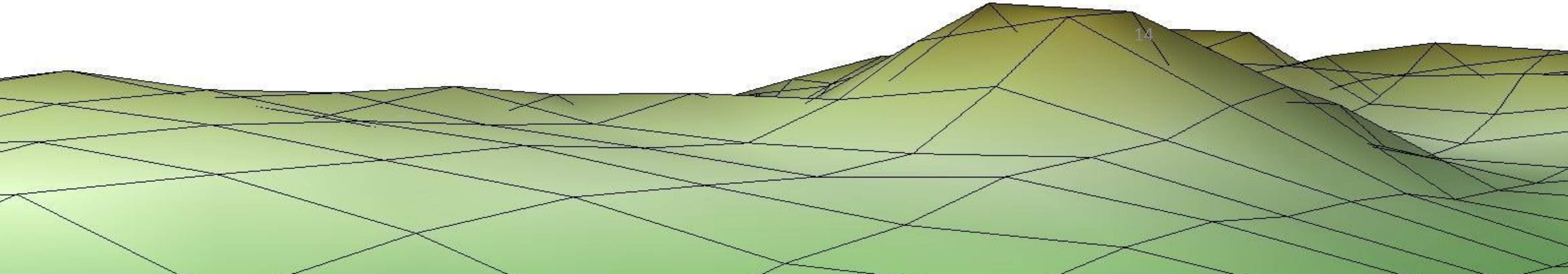
Resolution: 140m

What resolution is needed?



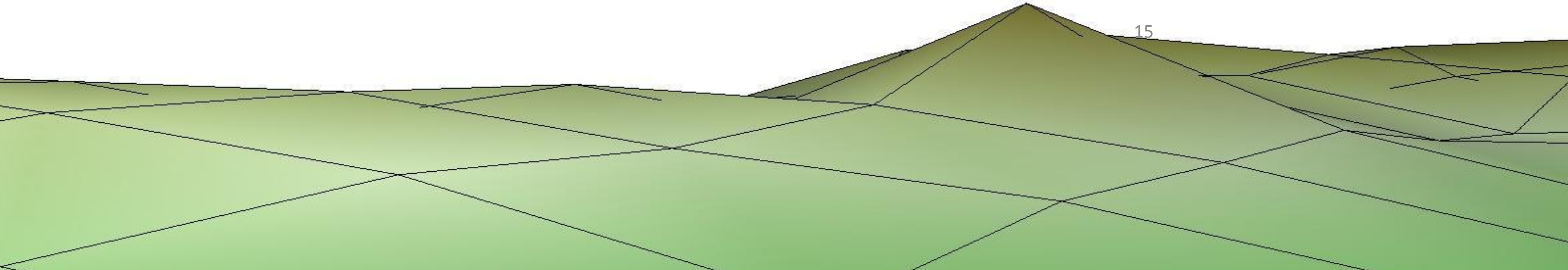
Resolution: 560m

What resolution is needed?



Resolution: 1.1km (Today)

What resolution is needed?



Challenges

Computation

Resolution	1 km^2	
Surface	40,000 km^2 (CH)	-> 500,000,000 km^2
Duration	2-7 days (weather)	-> 100 years
Time-to-Solution	3 months	

Software (COSMO)

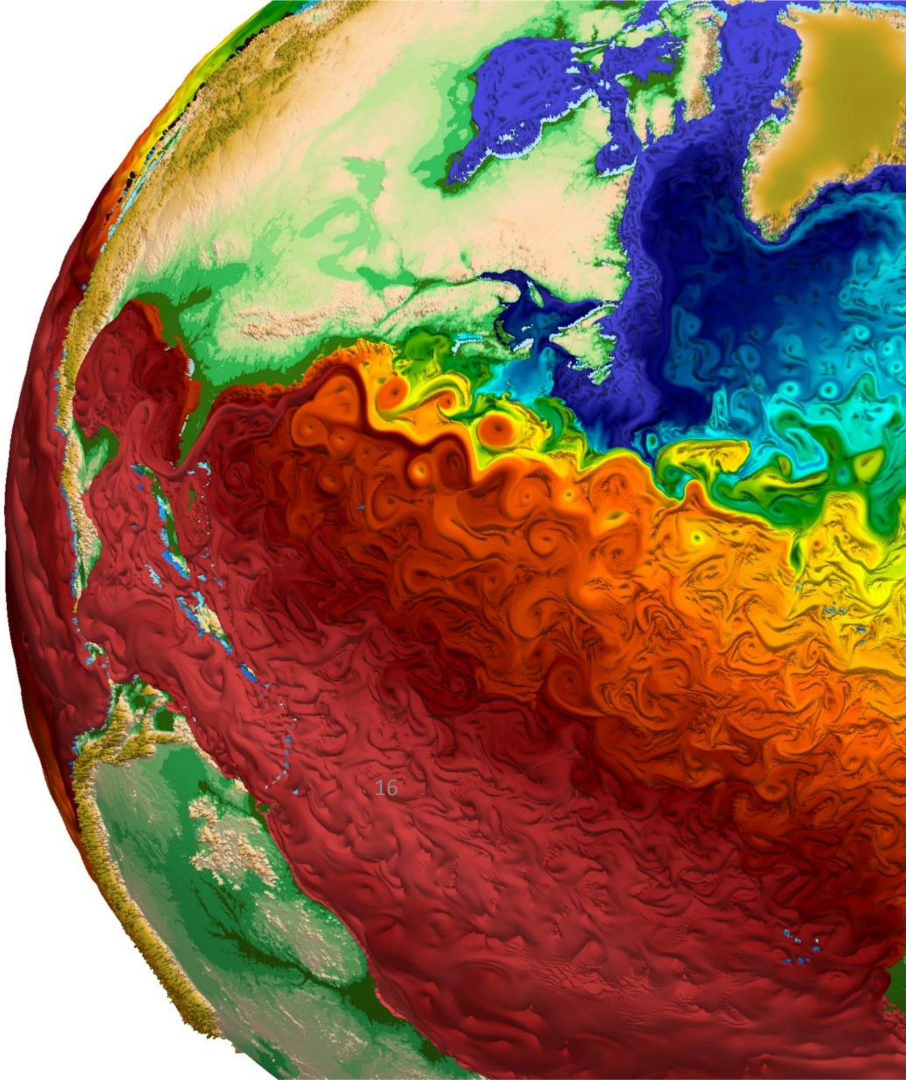
Language	Fortran
Size	300,000 LoC
Loops	thousands
Multi-Domain	Physics, Stencils, General-Purpose, MPI

Hardware

Insufficient memory bandwidth

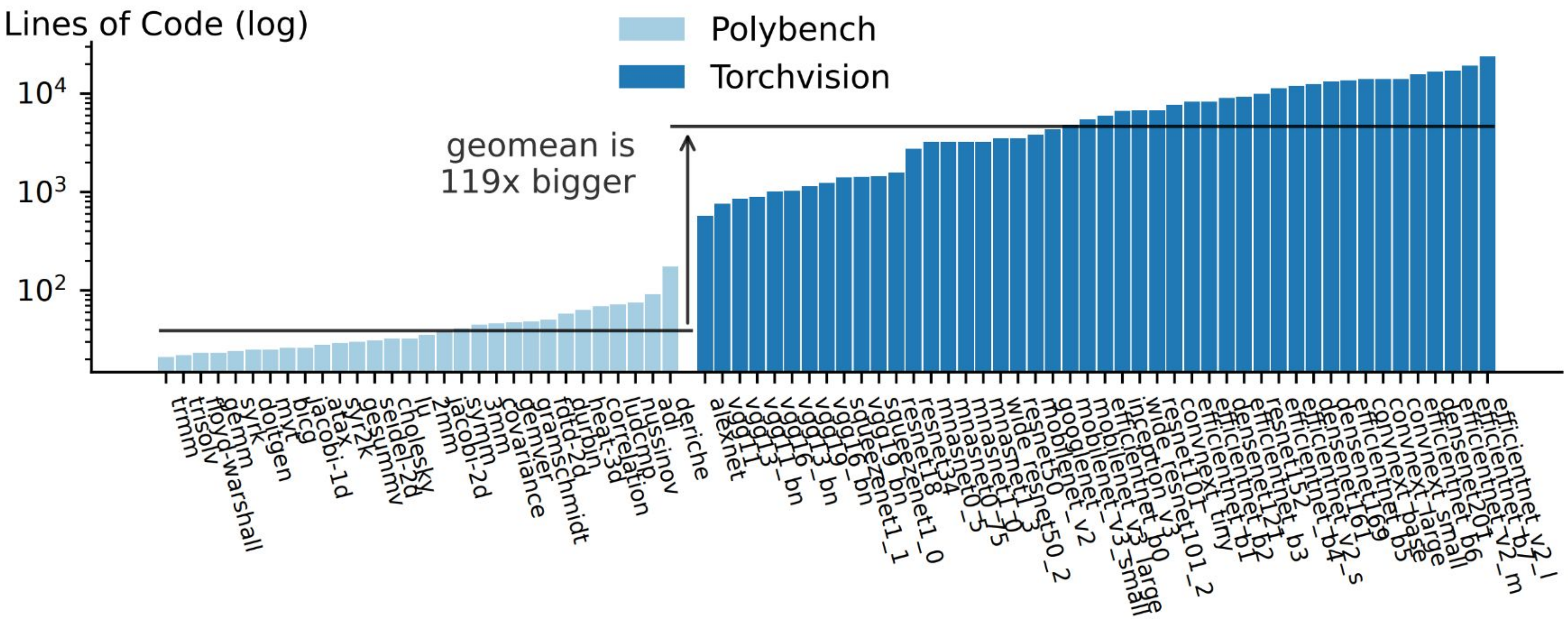
Community

DSL and Non-DSL code, HPC engineer wants control, …



The Size of Deep Neural Networks

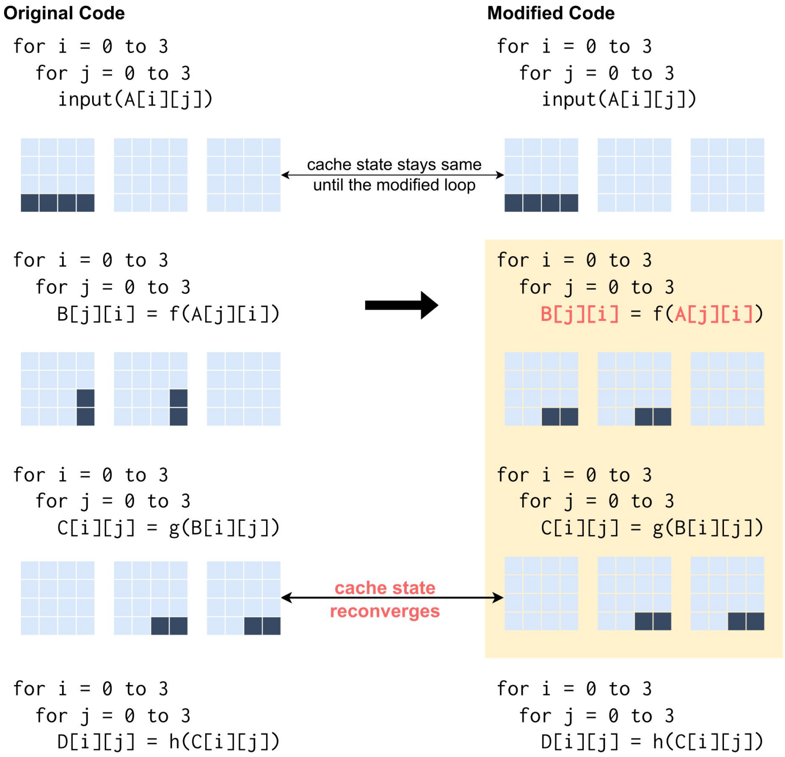
How to scale to such large networks?



mlir-meminfo: A Memory Model for MLIR

Kunwar Grover
Arjun Pitchanathan
Tobias Grosser
… and other contributors

IIIT Hyderabad
University of Edinburgh
University of Edinburgh



Original Code

```
for i = 0 to 3
  for j = 0 to 3
    input(A[i][j])
```

```
for i = 0 to 3
  for j = 0 to 3
    B[j][i] = f(A[j][i])
```

```
for i = 0 to 3
  for j = 0 to 3
    C[i][j] = g(B[i][j])
```

```
for i = 0 to 3
  for j = 0 to 3
    D[i][j] = h(C[i][j])
```

Modified Code

```
for i = 0 to 3
  for j = 0 to 3
    input(A[i][j])
```

```
for i = 0 to 3
  for j = 0 to 3
    B[j][i] = f(A[j][i])
```

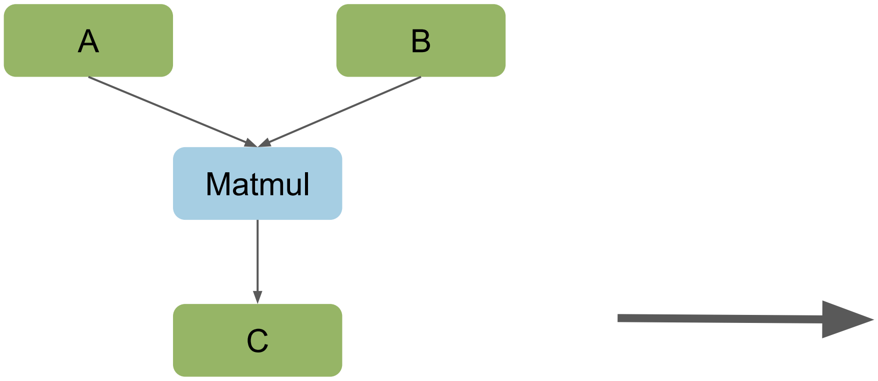
```
for i = 0 to 3
  for j = 0 to 3
    C[i][j] = g(B[i][j])
```

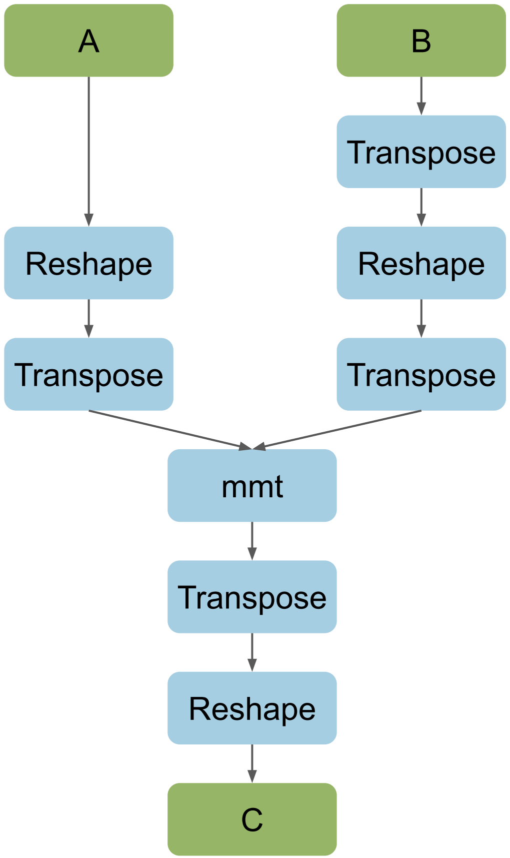
```
for i = 0 to 3
  for j = 0 to 3
    D[i][j] = h(C[i][j])
```

cache state stays same until the modified loop

cache state reconverges

A Transformation on 2D Matrix Multiplication

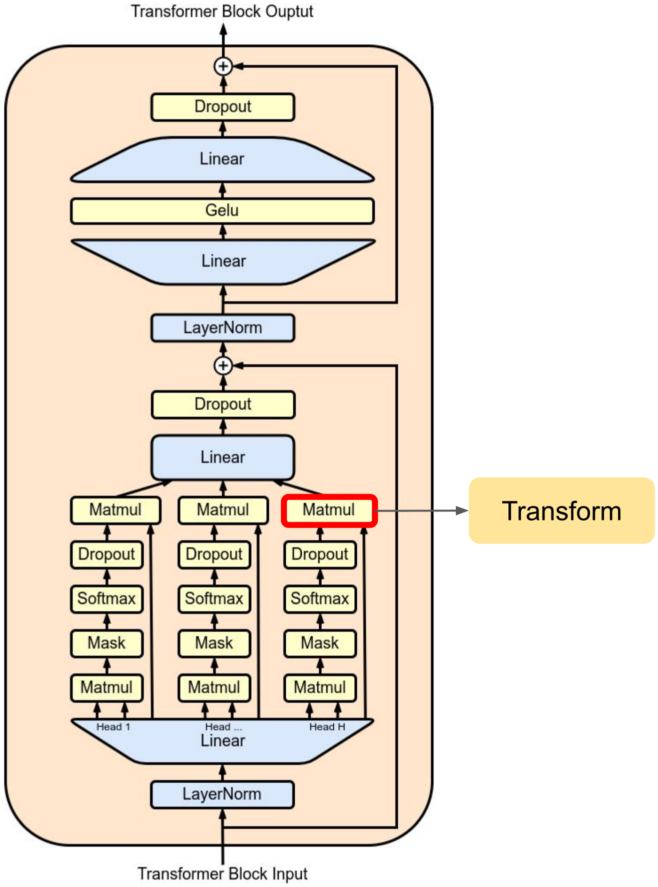




Is the miss rate of the final program intuitive?

How long does it take to profile?

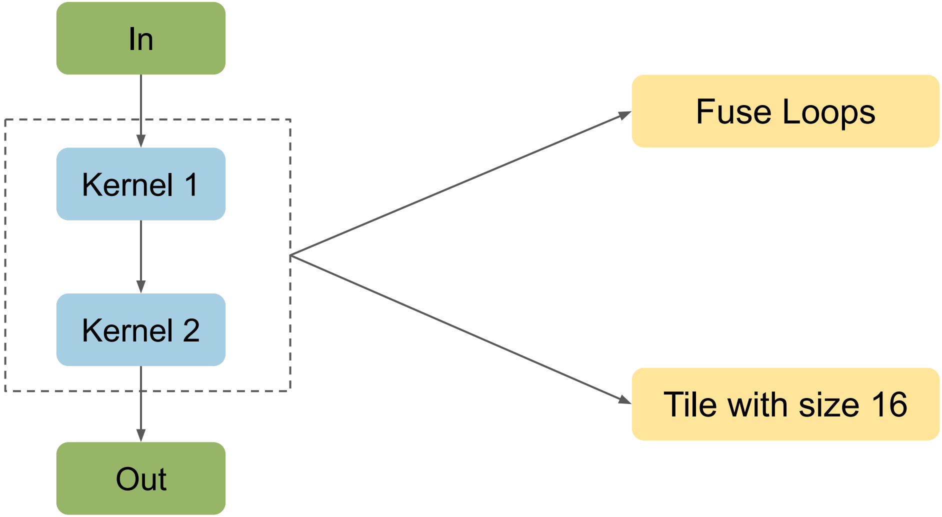
What about bigger programs? : Transformers



Is the miss rate of the final program intuitive?

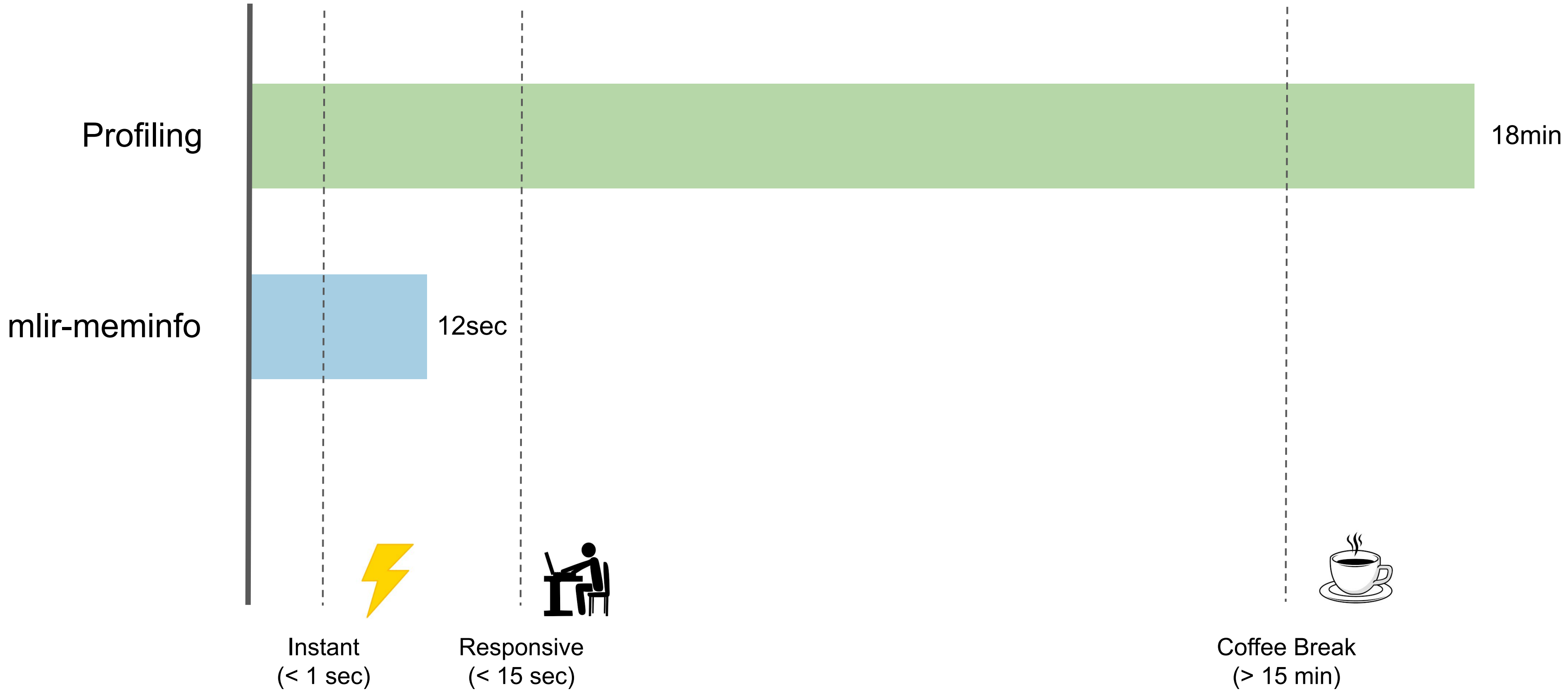
How long does it take to profile?

The Unintuitive Cost of Data-Movement



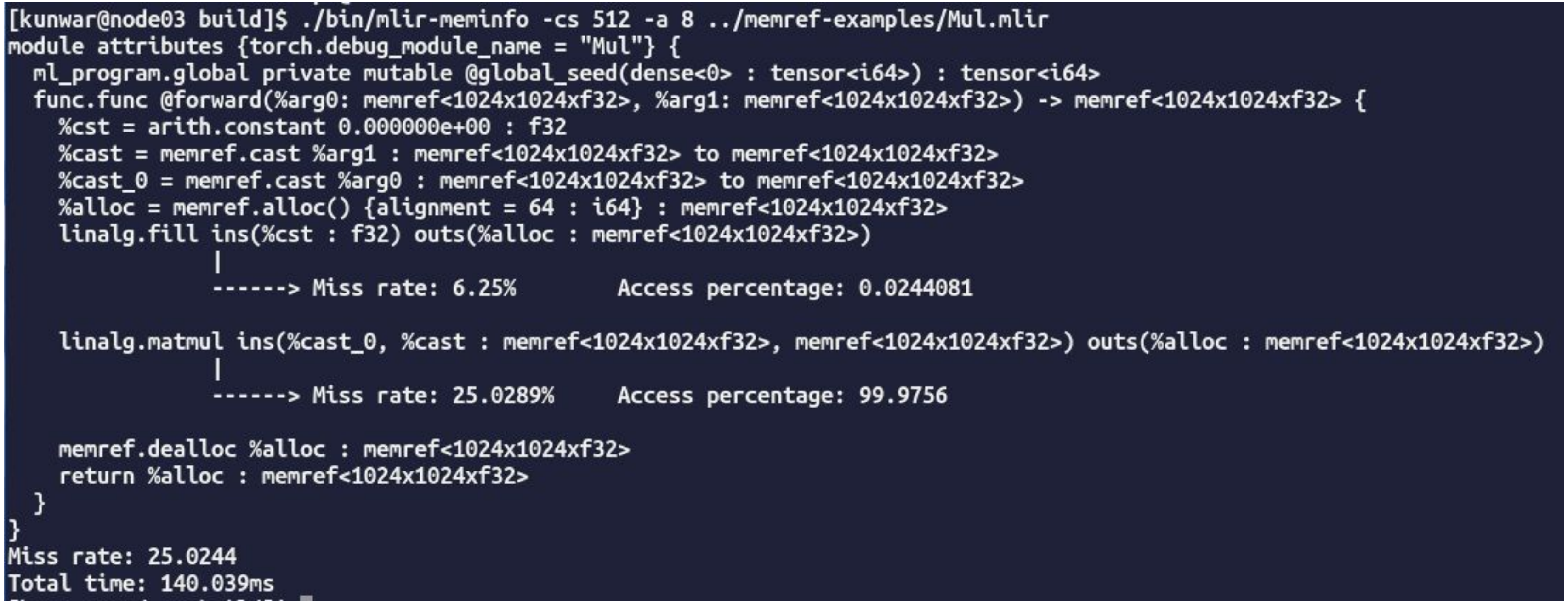
Which has the best cache miss rate?

Time to understand this cache behavior on BERT



mlir-meminfo on Matmul Transform

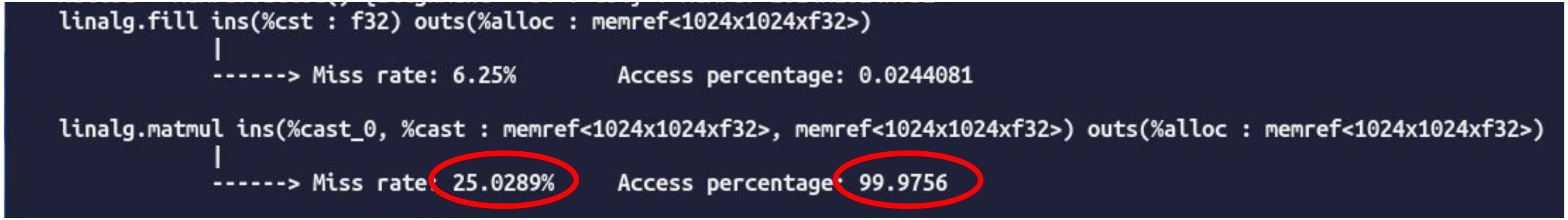
```
mlir-meminfo --cache-lines 512 --associativity 8 Mul.mlir
```



```
[kunwar@node03 build]$ ./bin/mlir-meminfo -cs 512 -a 8 ../memref-examples/Mul.mlir
module attributes {torch.debug_module_name = "Mul"} {
  ml_program.global private mutable @global_seed(dense<0> : tensor<i64>) : tensor<i64>
  func.func @forward(%arg0: memref<1024x1024xf32>, %arg1: memref<1024x1024xf32>) -> memref<1024x1024xf32> {
    %cst = arith.constant 0.000000e+00 : f32
    %cast = memref.cast %arg1 : memref<1024x1024xf32> to memref<1024x1024xf32>
    %cast_0 = memref.cast %arg0 : memref<1024x1024xf32> to memref<1024x1024xf32>
    %alloc = memref.alloc() {alignment = 64 : i64} : memref<1024x1024xf32>
    linalg.fill ins(%cst : f32) outs(%alloc : memref<1024x1024xf32>)
                |
                ------> Miss rate: 6.25%        Access percentage: 0.0244081

    linalg.matmul ins(%cast_0, %cast : memref<1024x1024xf32>, memref<1024x1024xf32>) outs(%alloc : memref<1024x1024xf32>)
                |
                ------> Miss rate: 25.0289%     Access percentage: 99.9756

    memref.dealloc %alloc : memref<1024x1024xf32>
    return %alloc : memref<1024x1024xf32>
  }
}
Miss rate: 25.0244
Total time: 140.039ms
```

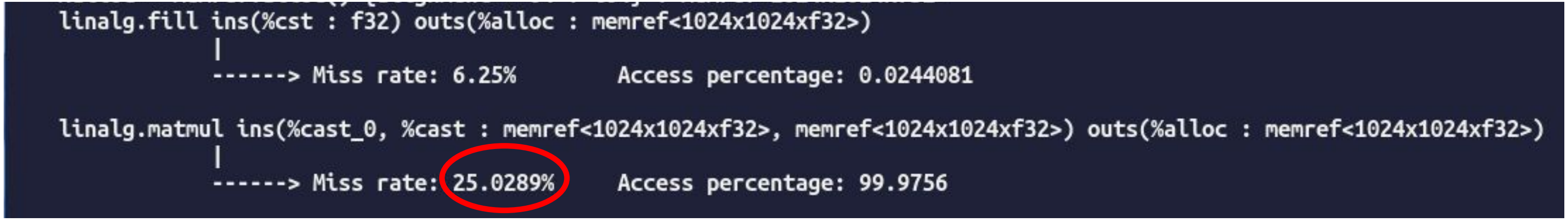
mlir-meminfo on Matmul Transform

```
mlir-meminfo --cache-lines 512 --associativity 8 Mul.mlir
```



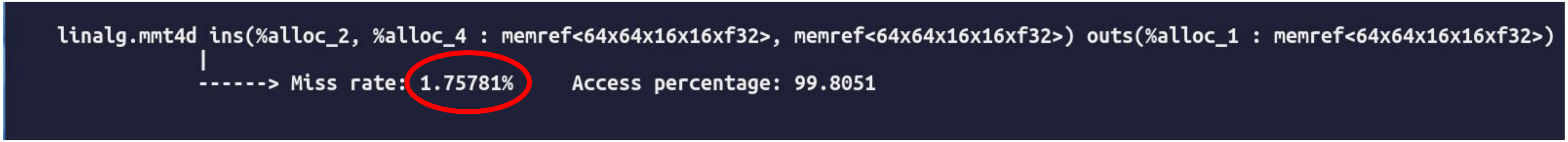
```
linalg.fill ins(%cst : f32) outs(%alloc : memref<1024x1024xf32>)
           |
           ------> Miss rate: 6.25%       Access percentage: 0.0244081

linalg.matmul ins(%cast_0, %cast : memref<1024x1024xf32>, memref<1024x1024xf32>) outs(%alloc : memref<1024x1024xf32>)
           |
           ------> Miss rate: 25.0289%    Access percentage: 99.9756
```

mlir-meminfo on Matmul Transform



```
linalg.fill ins(%cst : f32) outs(%alloc : memref<1024x1024xf32>)
                |
                ------> Miss rate: 6.25%        Access percentage: 0.0244081

linalg.matmul ins(%cast_0, %cast : memref<1024x1024xf32>, memref<1024x1024xf32>) outs(%alloc : memref<1024x1024xf32>)
                |
                ------> Miss rate: 25.0289%     Access percentage: 99.9756
```

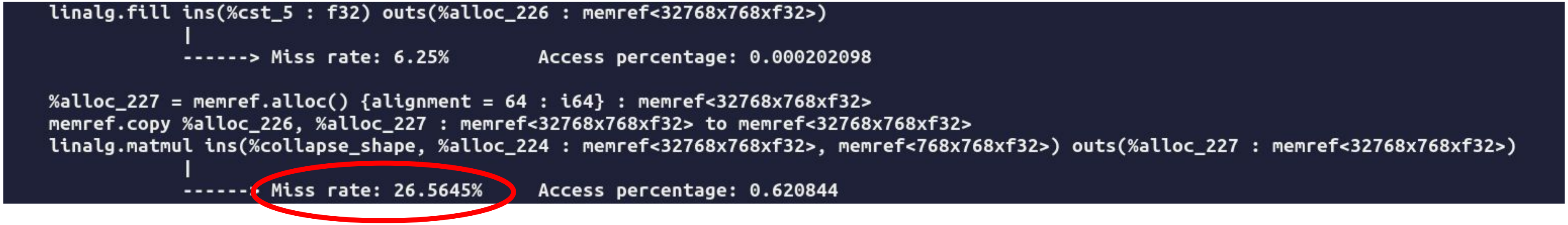
< 150 ms!



```
linalg.mmt4d ins(%alloc_2, %alloc_4 : memref<64x64x16x16xf32>, memref<64x64x16x16xf32>) outs(%alloc_1 : memref<64x64x16x16xf32>)
                |
                ------> Miss rate: 1.75781%     Access percentage: 99.8051
```

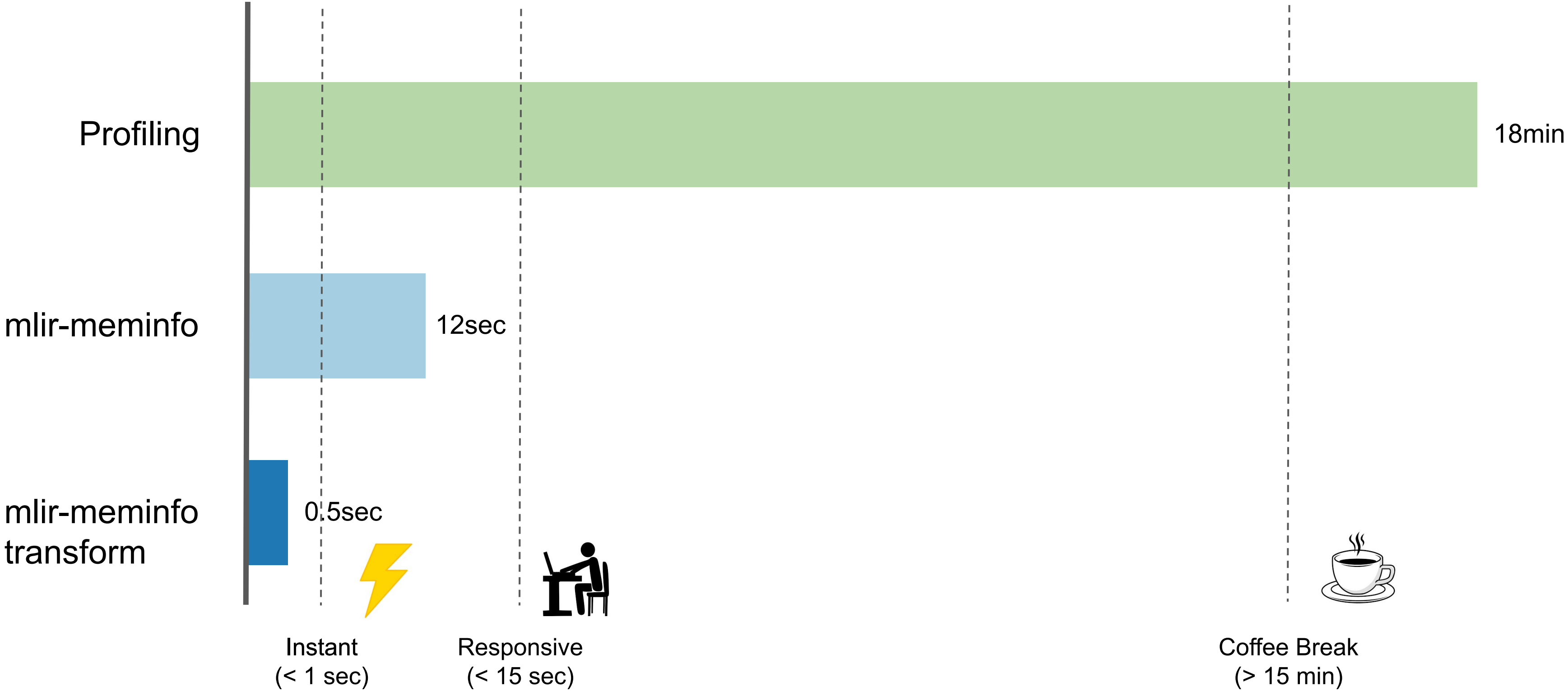
< 150 ms!

Matmul Transform on BERT

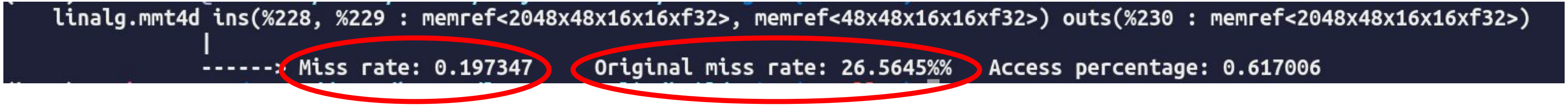


```
    linalg.fill ins(%cst_5 : f32) outs(%alloc_226 : memref<32768x768xf32>)
                |
                ------> Miss rate: 6.25%        Access percentage: 0.000202098

    %alloc_227 = memref.alloc() {alignment = 64 : i64} : memref<32768x768xf32>
    memref.copy %alloc_226, %alloc_227 : memref<32768x768xf32> to memref<32768x768xf32>
    linalg.matmul ins(%collapse_shape, %alloc_224 : memref<32768x768xf32>, memref<768x768xf32>) outs(%alloc_227 : memref<32768x768xf32>)
                |
                ------> Miss rate: 26.5645%     Access percentage: 0.620844
```

Time to understand this cache behavior on BERT



mlir-meminfo Matmul Transform on BERT



```
    linalg.mmt4d ins(%228, %229 : memref<2048x48x16x16xf32>, memref<48x48x16x16xf32>) outs(%230 : memref<2048x48x16x16xf32>)
                 |
                 ------> Miss rate: 0.197347    Original miss rate: 26.5645%%  Access percentage: 0.617006
```

Algorithm

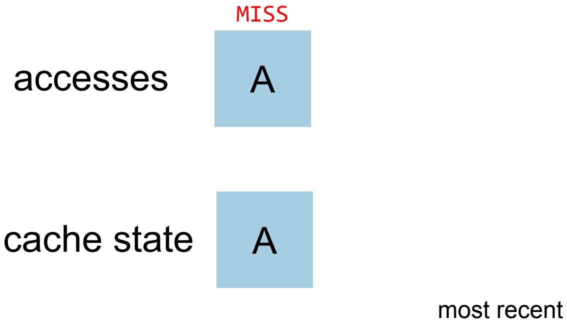
LRU Cache Policy

accesses

cache state

most recent

LRU Cache Policy



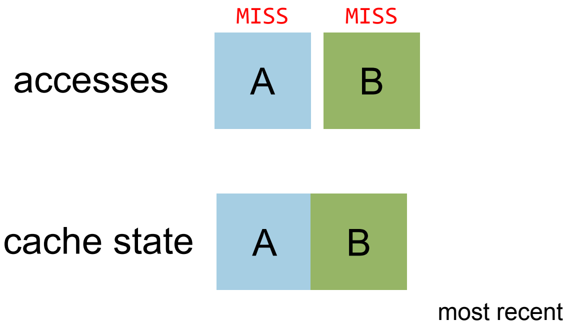
MISS

accesses A

cache state A

most recent

LRU Cache Policy



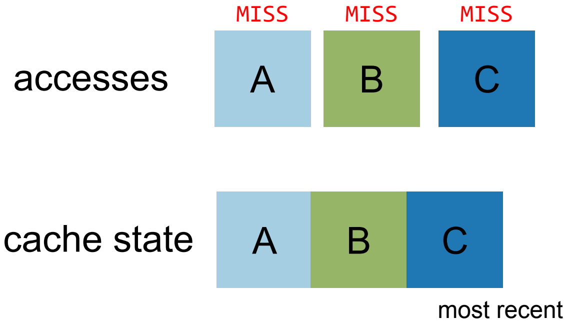
MISS MISS

accesses A B

cache state A B

most recent

LRU Cache Policy



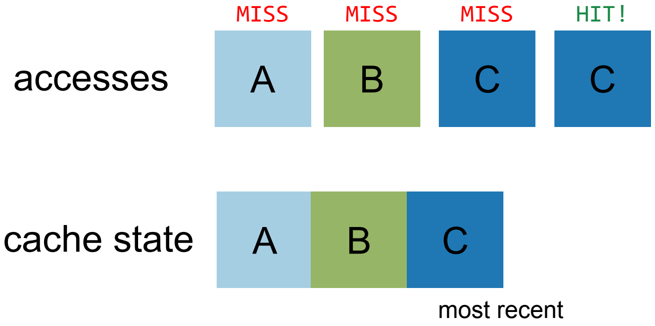
MISS MISS MISS

accesses A B C

cache state A B C

most recent

LRU Cache Policy



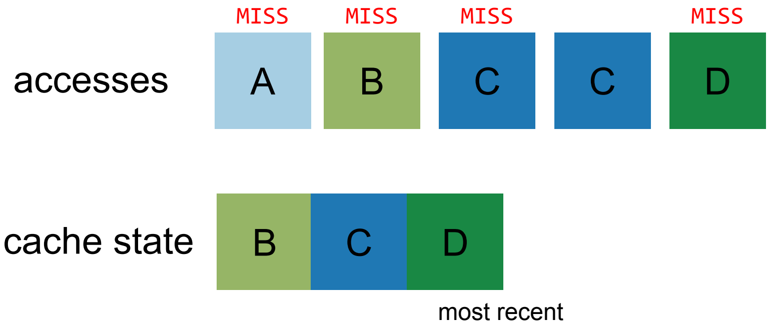
MISS MISS MISS HIT!

accesses A B C C

cache state A B C

most recent

LRU Cache Policy



accesses

MISS A

MISS B

MISS C

C

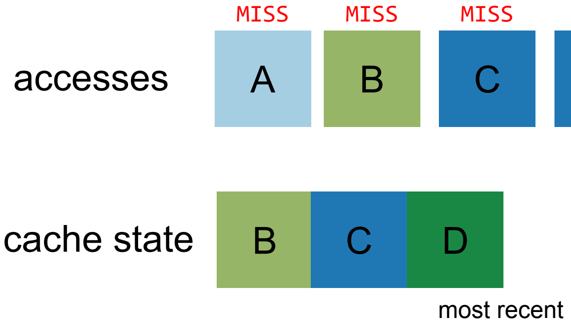
MISS D

cache state

B C D

most recent

LRU Cache Policy



accesses

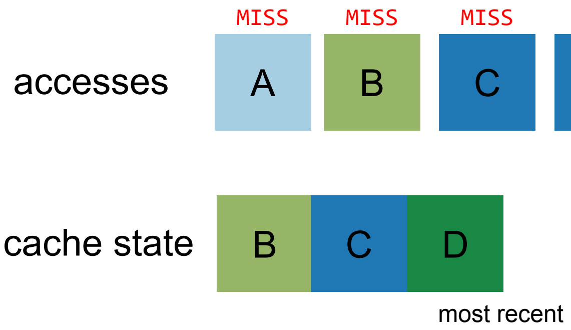
MISS A · MISS B · MISS C · C · MISS D · HIT! D

cache state

B C D

most recent

LRU Cache Policy



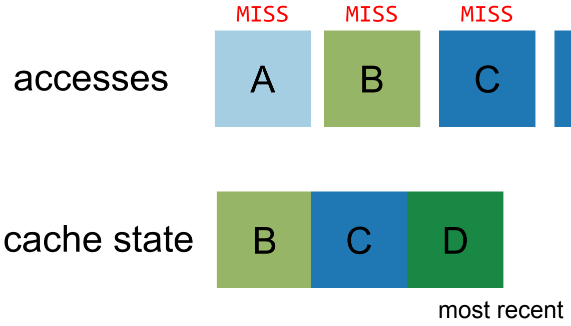
MISS MISS MISS MISS HIT!

accesses A B C C D D D

cache state B C D

most recent

LRU Cache Policy

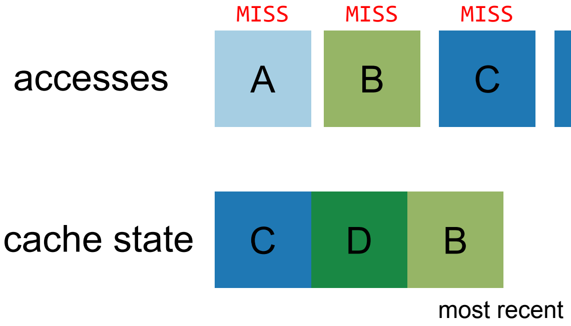


	MISS	MISS	MISS		MISS			HIT!
accesses	A	B	C	C	D	D	D	D

cache state B C D

most recent

LRU Cache Policy



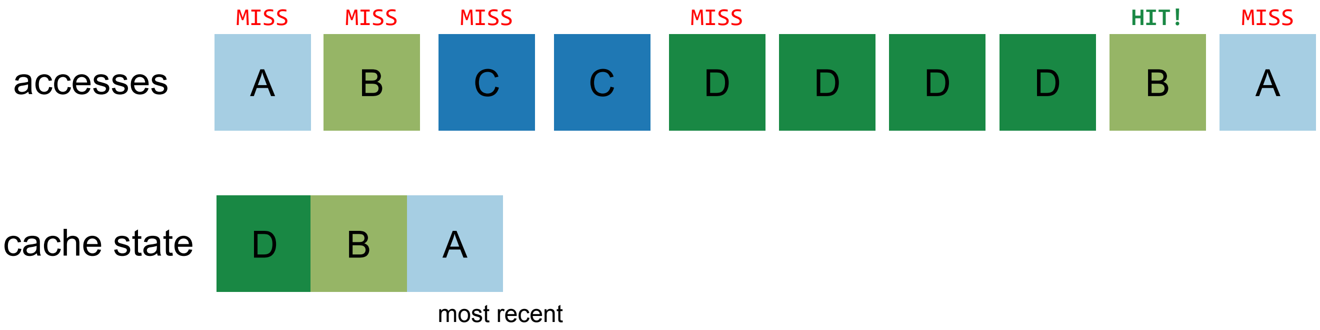
MISS MISS MISS MISS HIT!

accesses A B C C D D D D B

cache state C D B

most recent

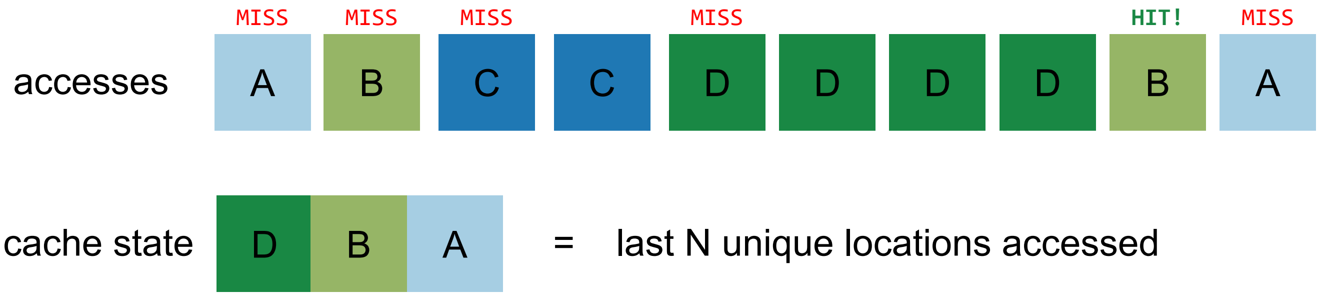
LRU Cache Policy



	MISS	MISS	MISS		MISS				HIT!	MISS
accesses	A	B	C	C	D	D	D	D	B	A

cache state	D	B	A
			most recent

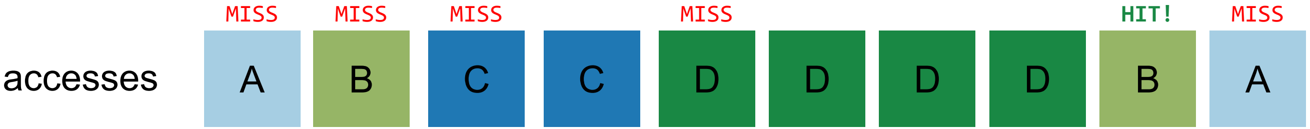
LRU Cache Policy



	MISS	MISS	MISS		MISS				HIT!	MISS
accesses	A	B	C	C	D	D	D	D	B	A

cache state D B A = last N unique locations accessed

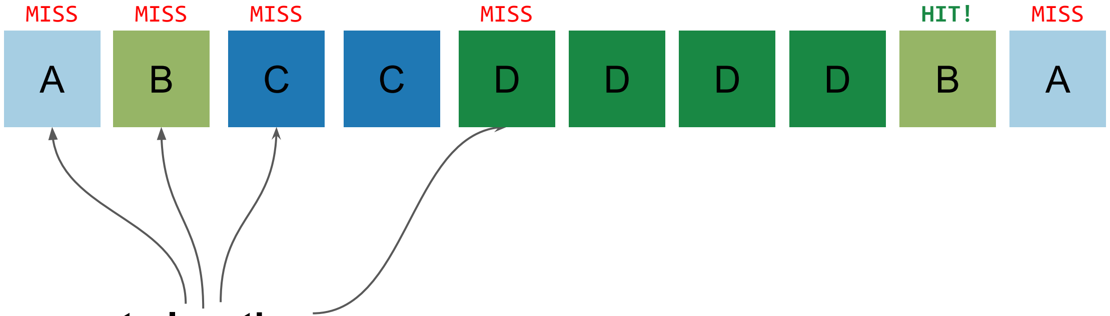
LRU Cache Policy



	MISS	MISS	MISS		MISS				HIT!	MISS
accesses	A	B	C	C	D	D	D	D	B	A

miss if…

LRU Cache Policy

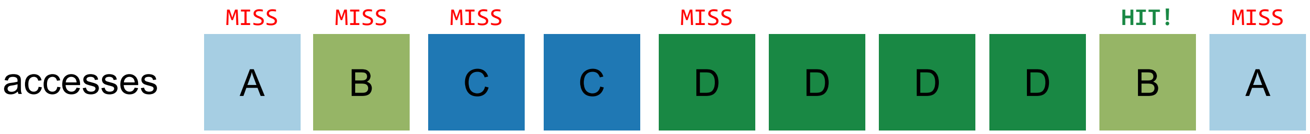


	MISS	MISS	MISS		MISS				HIT!	MISS
accesses	A	B	C	C	D	D	D	D	B	A

miss if…

- **first access to location**

LRU Cache Policy

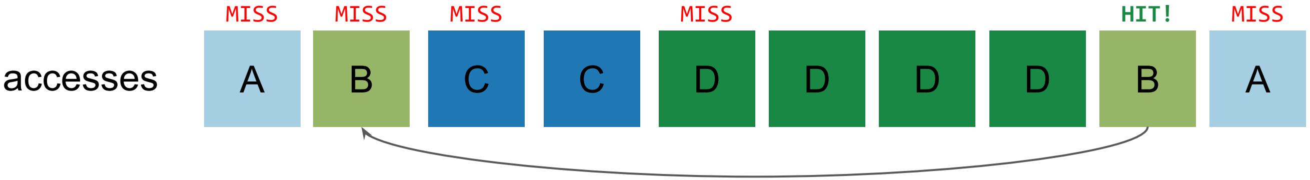


	MISS	MISS	MISS		MISS				HIT!	MISS
accesses	A	B	C	C	D	D	D	D	B	A

miss if…

- first access to location
- **previous access to same location was “long” ago**

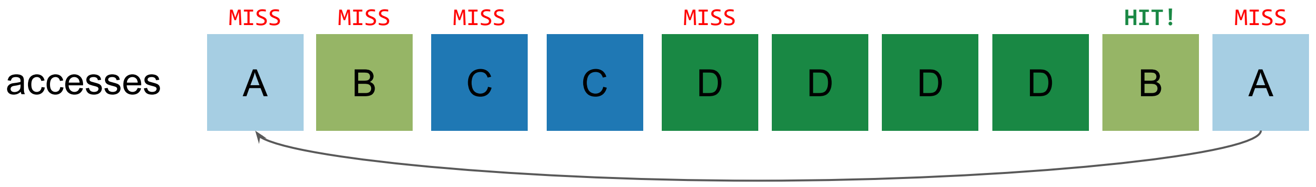
Most recent access: how long ago for miss?



	MISS	MISS	MISS		MISS				HIT!	MISS
accesses	A	B	C	C	D	D	D	D	B	A

2 unique memory accesses ago; hit

Most recent access: how long ago for miss?



	MISS	MISS	MISS		MISS				HIT!	MISS
accesses	A	B	C	C	D	D	D	D	B	A

3 unique memory accesses ago; **miss**

Earlier Cache Modelling Algorithms

$\mathbf{A} =$ access map

$\mathbf{I} =$ iteration domain

$\mathbf{S} =$ schedule map

$$\mathbf{L}_{<} = \{(i_0, \ldots, i_n) \longrightarrow (j_0, \ldots, j_n) : \\ (i_0, \ldots, i_n) \prec (j_0, \ldots, j_n) \wedge \\ (i_0, \ldots, i_n), (j_0, \ldots, j_n) \in \mathbf{S}_{ran}\}$$

$$\mathbf{L}_{\leq} = \{(i_0, \ldots, i_n) \longrightarrow (j_0, \ldots, j_n) : \\ (i_0, \ldots, i_n) \preceq (j_0, \ldots, j_n) \wedge \\ (i_0, \ldots, i_n), (j_0, \ldots, j_n) \in \mathbf{S}_{ran}\}$$

$$\mathbf{F} = (\mathbf{S}^{-1} \circ \mathbf{L}_{\leq} \circ \mathbf{S}) \circ \mathbf{N}^{-1}$$

$$\mathbf{E} = \mathbf{S} \circ \mathbf{A}^{-1} \circ \mathbf{A} \circ \mathbf{S}^{-1}$$

$$\mathbf{B} = \mathbf{S}^{-1} \circ \mathbf{L}_{\leq}^{-1} \circ \mathbf{S}$$

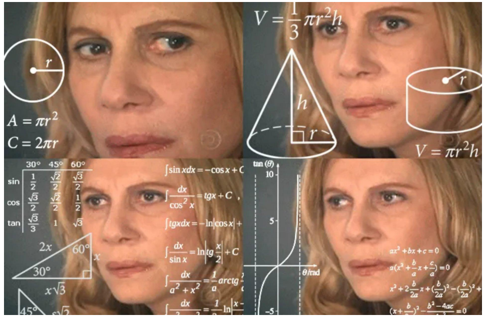
$$\mathbf{N} = \mathbf{S}^{-1} \circ \operatorname{lexmin}(\mathbf{L}_{<} \cap \mathbf{E}) \circ \mathbf{S}$$

$$\mathbf{D} = \{|\mathbf{A} \circ (\mathbf{F} \cap \mathbf{B})|\}$$

$$\mathbf{F} = \mathbf{S}^{-1} \circ \operatorname{lexmin}(\mathbf{S} \circ \mathbf{A}^{-1})$$

(different F)

compulsory misses = $|\mathbf{F}_{dom}|$



Most recent access = dependence analysis

Most recent access = dependence analysis

```
for i = 0 to 499:
     load A[i]
```

Most recent access = dependence analysis

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]
```

Most recent access = dependence analysis

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]
```

Most recent access = dependence analysis

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]
```

Most recent access = dependence analysis

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
```

Most recent access = dependence analysis

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
```

Most recent access = dependence analysis

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

(in bounds)

Most recent access = dependence analysis

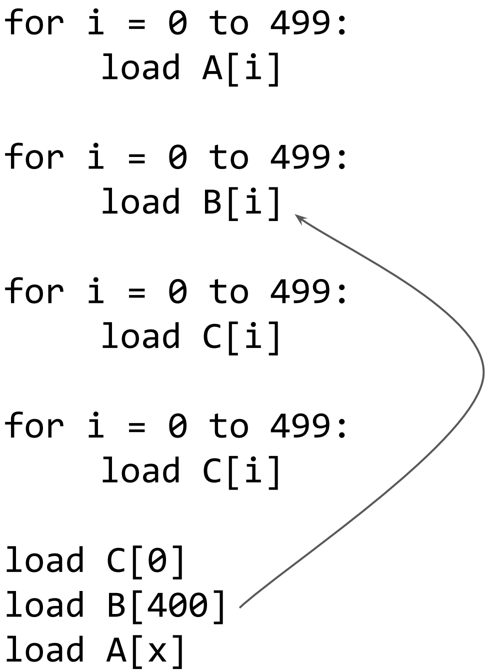
```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

no dependence; first access; **miss**

Most recent access = dependence analysis

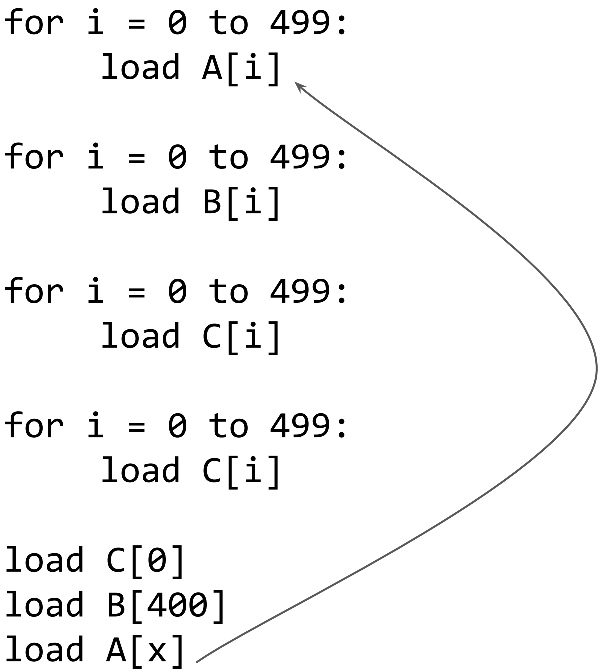
```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Most recent access = dependence analysis



```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Most recent access = dependence analysis



```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

How far is the most recent access?

cache size = 512

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

499 unique memory accesses; **hit**

`C[1..499]`

How far is the most recent access?

cache size = 512

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

500 + 99 unique memory accesses; **miss**
`C[0..499], B[401..499]`

How far is the most recent access?

cache size = 512

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

500 + 500 + ??? ≥ 512; miss

`C[0..499], B[0..499], A[???]`

Loop nests beyond 512 have no relevance

cache size = 512

```
for i = 499 to 0:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

500 + 500 + ??? ≥ 512; miss

`C[0..499], B[0..499], A[???]`

Loop nests beyond 512 have no relevance

cache size = 512

```
for i = 499 to 0:
     load A[i]
     load A[i + 1]
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

500 + 500 + ??? ≥ 512; miss

`C[0..499], B[0..499], A[???]`

Loop nests beyond 512 have no relevance

cache size = 512

```
for i = 499 to 0:
     load B[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

500 + 500 + ??? ≥ 512; miss

`C[0..499], B[0..499],` **`???`**

Loop nests beyond 512 have no relevance

cache size = 512

???

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

500 + 500 + ??? ≥ 512; miss

`C[0..499], B[0..499], ???`

Asymptotic improvement on practical programs

```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

checking all pairs of statements
for dependences: **$O(n^2)$**

Asymptotic improvement on practical programs

???

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

checking a few loop nests
for each statement: **O(n)**

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

this always misses

Incremental recomputation

cache size = 512

```
for i = 499 to 0:
     load A[i]      ←———  change this

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]      ←———  this always misses
```

Incremental recomputation

cache size = 512

```
for i = 499 to 0:
     load A[i]          ← change this
     load A[i + 1]
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]          ← this always misses
```

Incremental recomputation

cache size = 512

```
for i = 499 to 0:
     load B[i]    ←   change this

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]   ←   this always misses
```

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

this always misses

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

```
for i = 0 to 499:
     load B[i]
```

500

```
for i = 0 to 499:
     load C[i]
```

1000

```
for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

```
for i = 0 to 499:
     load B[i]
```

500

```
for i = 0 to 499:
     load C[i]
```

1000 ≥ 512

```
for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]
```

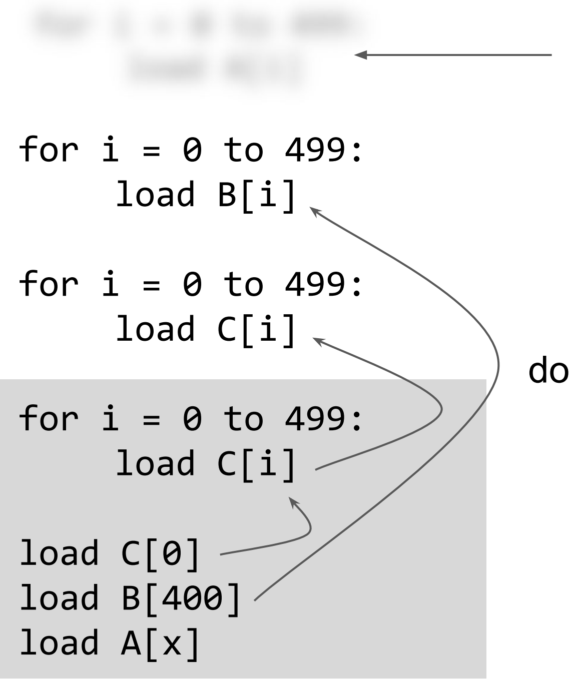
1000 ≥ 512

```
for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

same cache performance!

Incremental recomputation

cache size = 512



```
for i = 0 to 499:
     load A[i]

for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]

for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

change this

don’t involve the changed region

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]
```

```
for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Involves changed region,
but sure miss anyway

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

← change this

```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]
```

possibly changed cache performance

1000 ≥ 512

```
for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Incremental recomputation

cache size = 512

```
for i = 0 to 499:
     load A[i]
```

change this

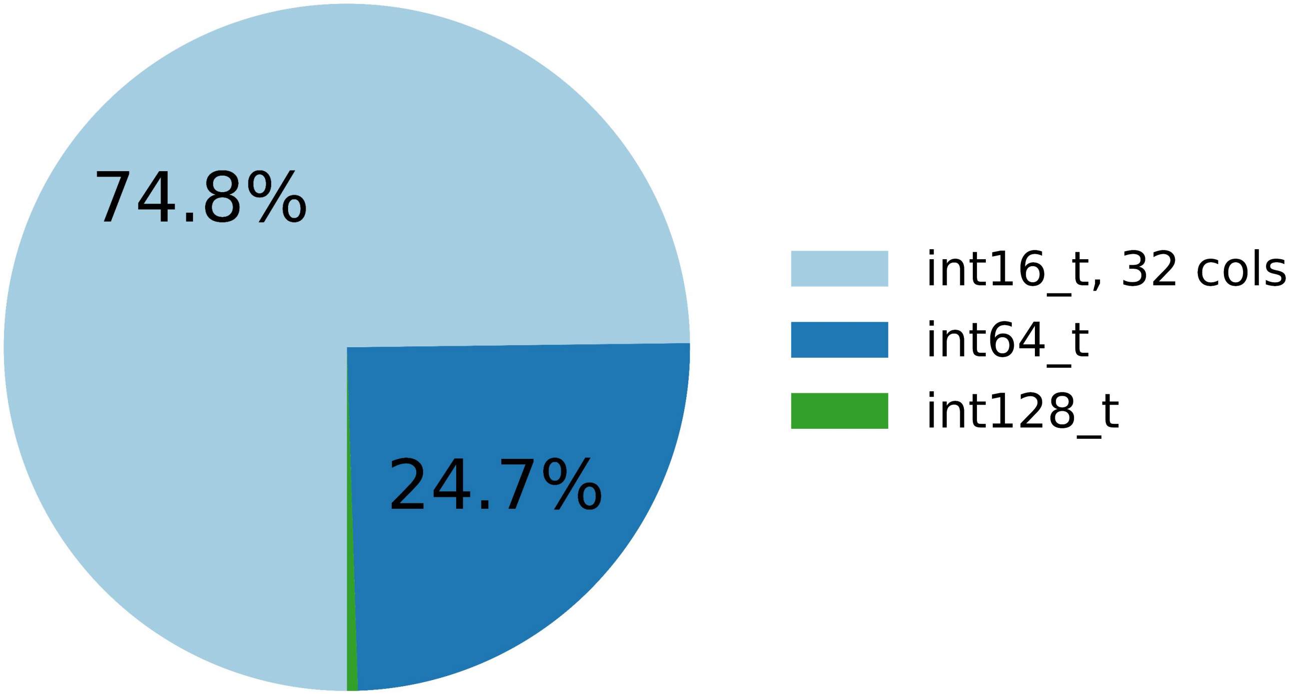
```
for i = 0 to 499:
     load B[i]

for i = 0 to 499:
     load C[i]
```

constant amount of recomputation

```
for i = 0 to 499:
     load C[i]

load C[0]
load B[400]
load A[x]
```

Fast Polyhedral Library

Applicability of Transprecision

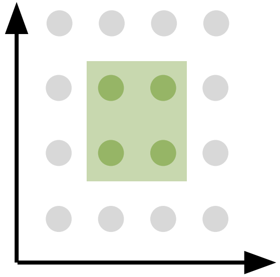


Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, Tobias Grosser. *FPL: Fast Presburger Arithmetic through Transprecision. OOPSLA 2021.*

Presburger Arithmetic in MLIR

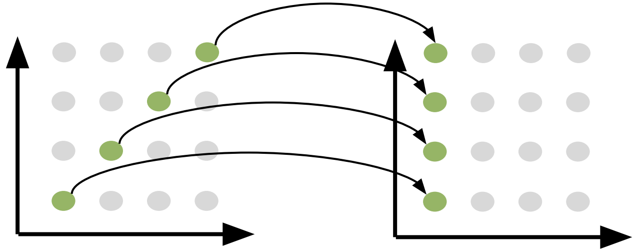
`FlatAffineConstraints`

Integer Polyhedron



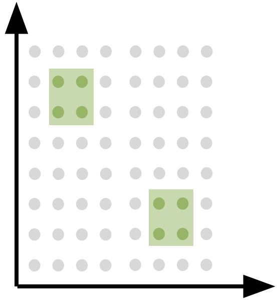
`FlatAffineRelation`

Affine Relations



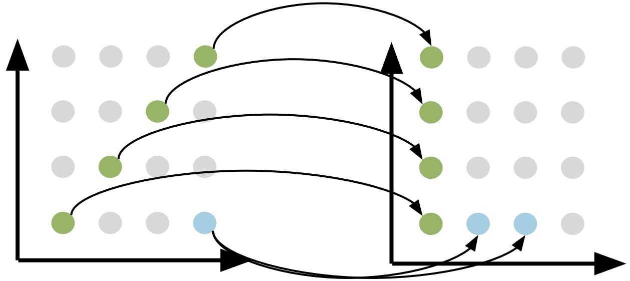
`PresburgerSet`

Union of Integer Polyhedra



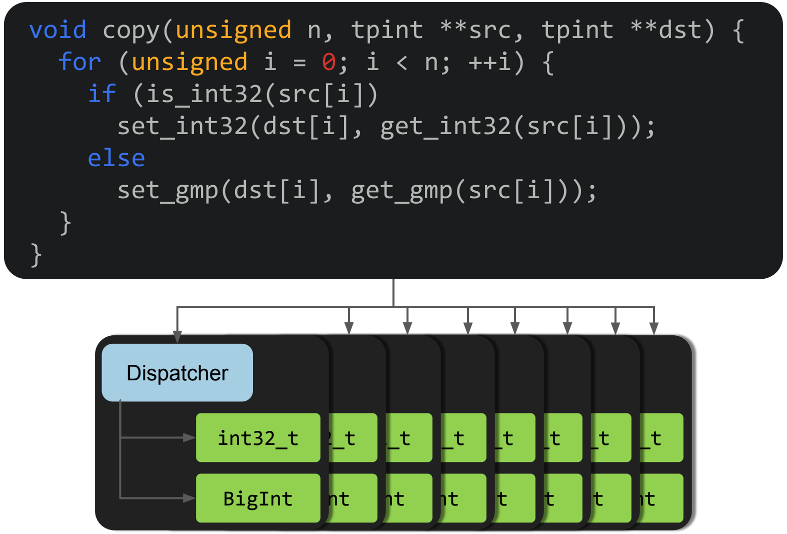
`PresburgerRelation`

Union of Affine Relations



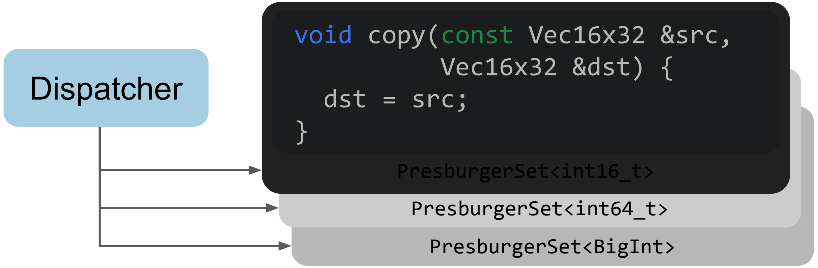
Transprecision Computing

isl optimization: element-TP



```
void copy(unsigned n, tpint **src, tpint **dst) {
  for (unsigned i = 0; i < n; ++i) {
    if (is_int32(src[i])
      set_int32(dst[i], get_int32(src[i]));
    else
      set_gmp(dst[i], get_gmp(src[i]));
  }
}
```

FPL: library-level transprecision

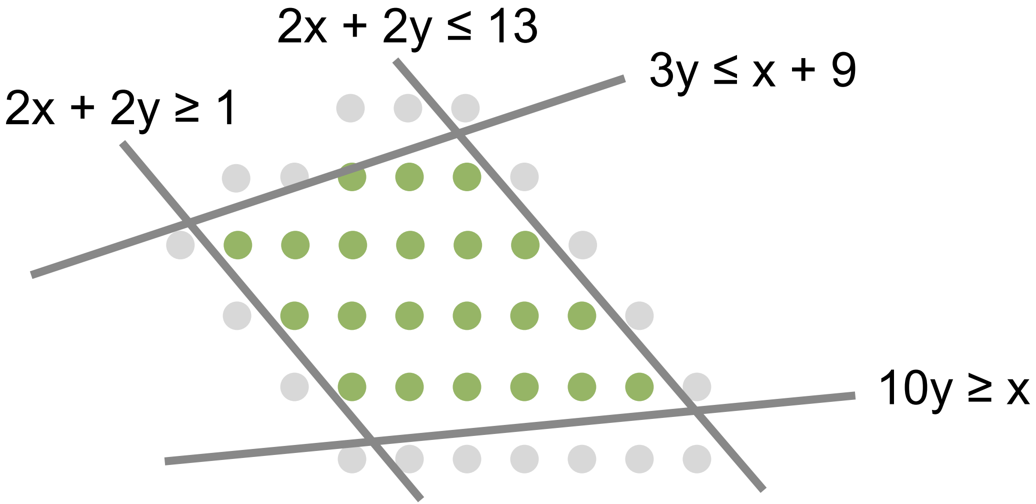


```
void copy(const Vec16x32 &src,
               Vec16x32 &dst) {
  dst = src;
}
```

isl	FPL
Full Presburger Arithmetic	Full Presburger Arithmetic
Scalar Arithmetic	Exploits SIMD Parallelism
Standalone C Library	Modern C++ Library, Integrated into MLIR

Currently Upstreaming

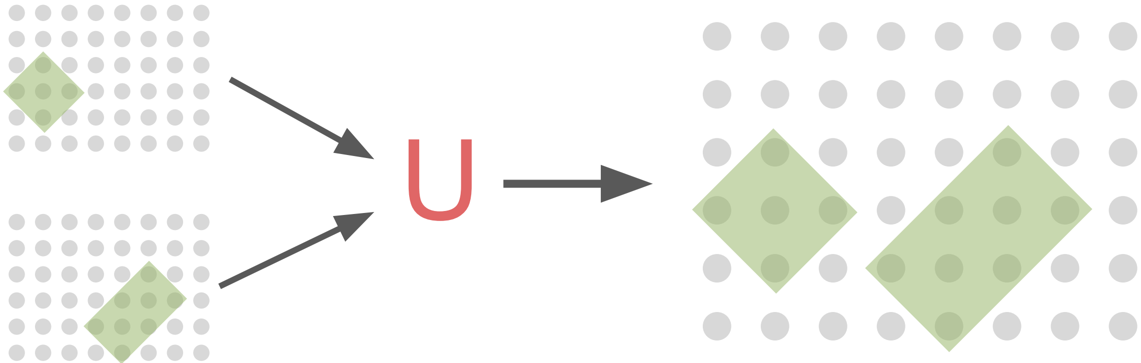
The internal representation



Internal Representation

Integer Polyhedron
2x + 2y ≤ 13
-x + 3y ≤ 9
2x + 2y ≥ 1
-x + 10y ≥ 0

Presburger Sets: Unions of Integer Polyhedra



Presburger Set

Integer Polyhedron

x - y ≥ 0

x - y ≤ 2

x + y ≥ 2

x + y ≤ 4

Integer Polyhedron

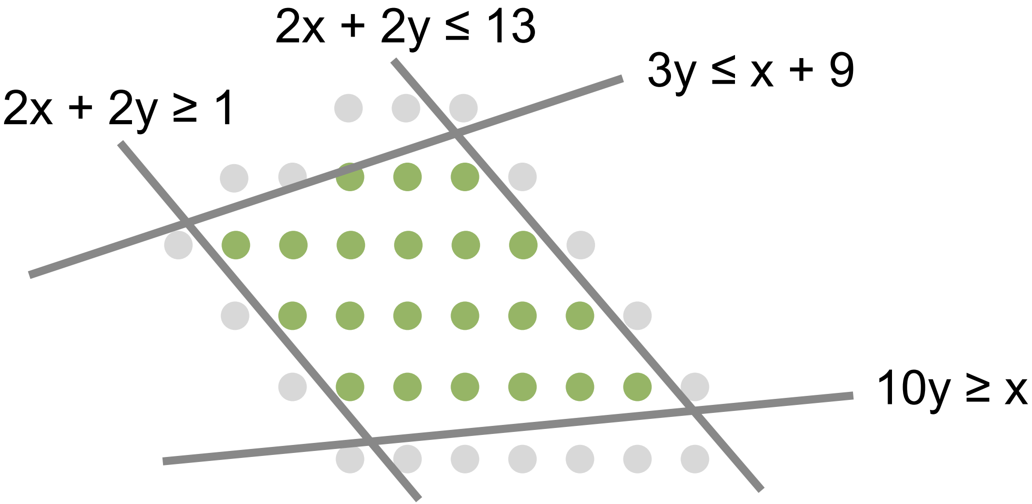
x - y ≥ -4

x - y ≤ -2

x + y ≥ 4

x + y ≤ 8

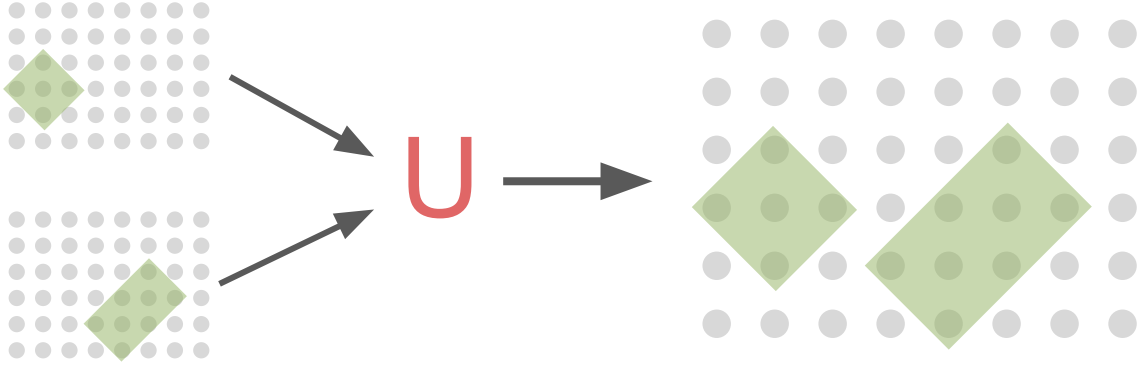
The internal representation



Internal Representation

Integer Polyhedron
2x + 2y ≤ 13
-x + 3y ≤ 9
2x + 2y ≥ 1
-x + 10y ≥ 0

Presburger Sets: Unions of Integer Polyhedra



Presburger Set

Integer Polyhedron

$x - y \geq 0$

$x - y \leq 2$

$x + y \geq 2$

$x + y \leq 4$

Integer Polyhedron

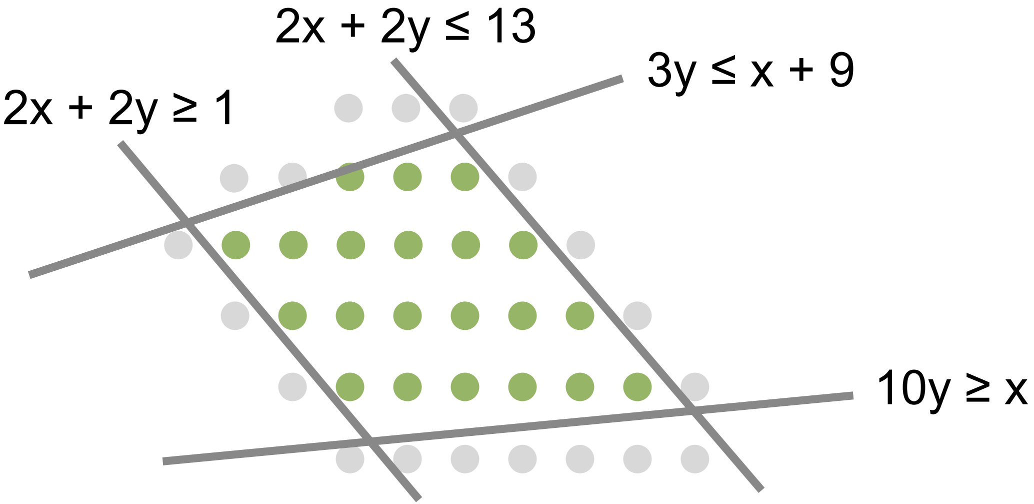
$x - y \geq -4$

$x - y \leq -2$

$x + y \geq 4$

$x + y \leq 8$

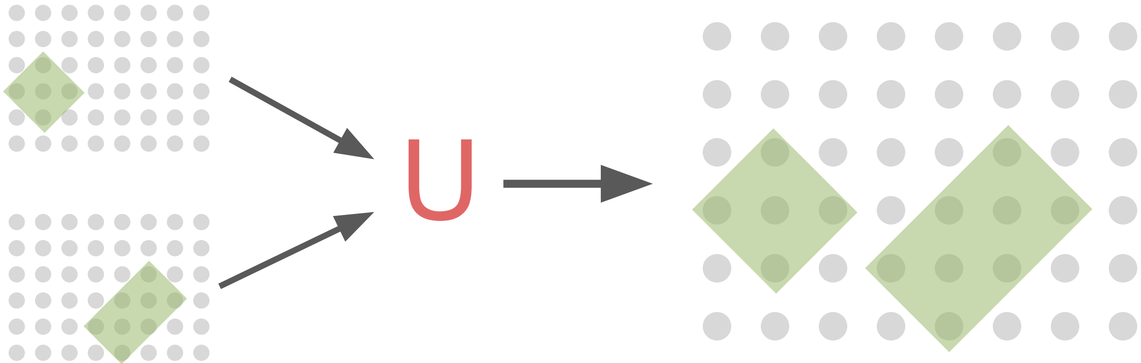
The internal representation



Internal Representation

Integer Polyhedron
$2x + 2y \leq 13$
$-x + 3y \leq 9$
$2x + 2y \geq 1$
$-x + 10y \geq 0$

Presburger Sets: Unions of Integer Polyhedra



Presburger Set

Integer Polyhedron

$x - y \geq 0$

$x - y \leq 2$

$x + y \geq 2$

$x + y \leq 4$

Integer Polyhedron

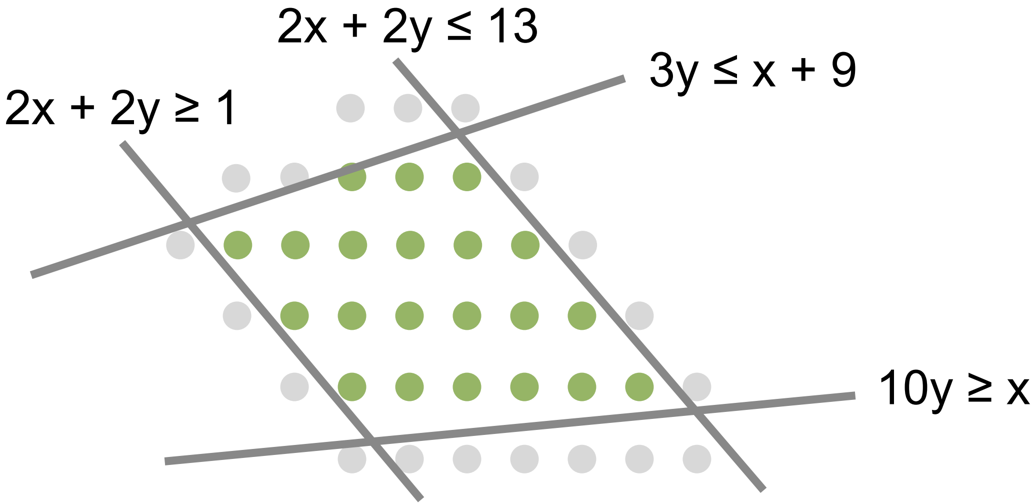
$x - y \geq -4$

$x - y \leq -2$

$x + y \geq 4$

$x + y \leq 8$

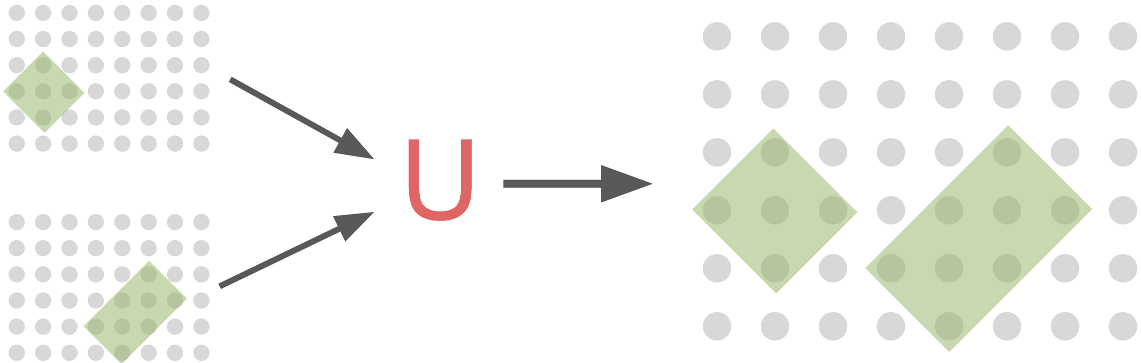
The internal representation



Internal Representation

Integer Polyhedron
2x + 2y ≤ 13
-x + 3y ≤ 9
2x + 2y ≥ 1
-x + 10y ≥ 0

Presburger Sets: Unions of Integer Polyhedra



Presburger Set

Integer Polyhedron

$x - y \geq 0$

$x - y \leq 2$

$x + y \geq 2$

$x + y \leq 4$

Integer Polyhedron

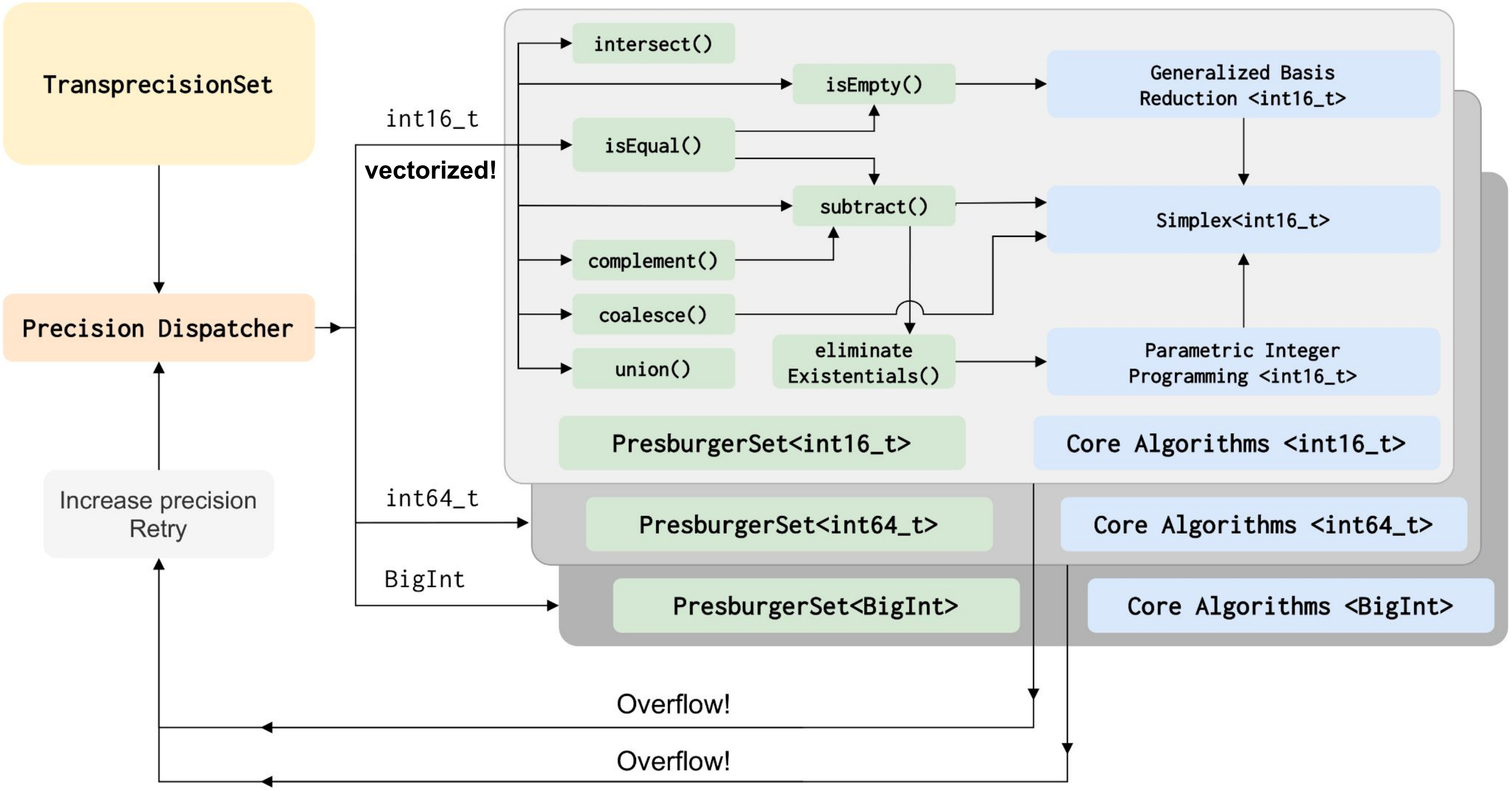
$x - y \geq -4$

$x - y \leq -2$

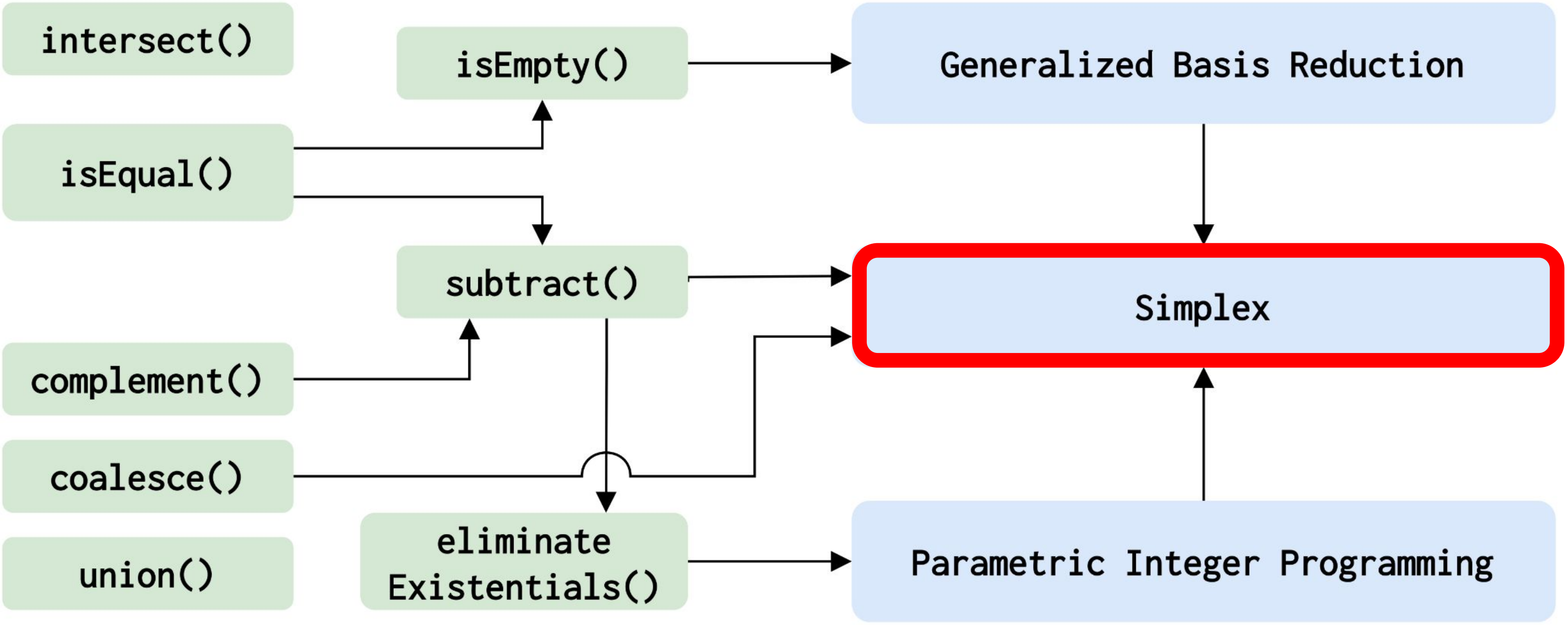
$x + y \geq 4$

$x + y \leq 8$

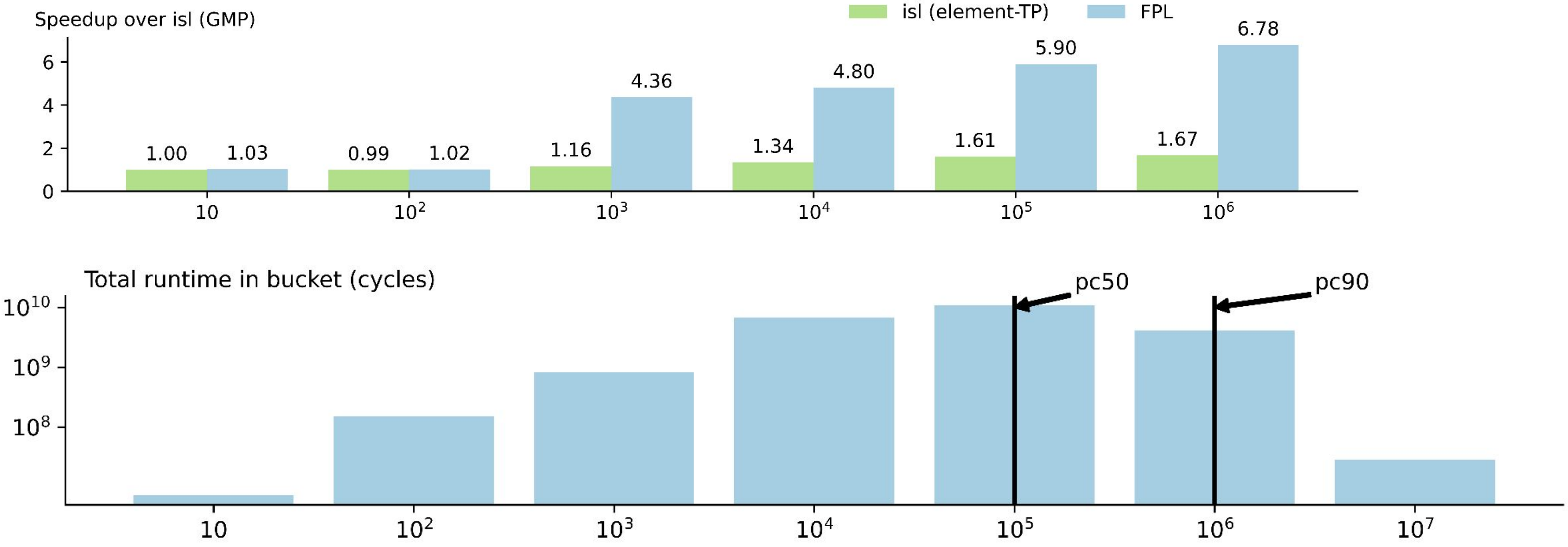
A Transprecision Presburger Library



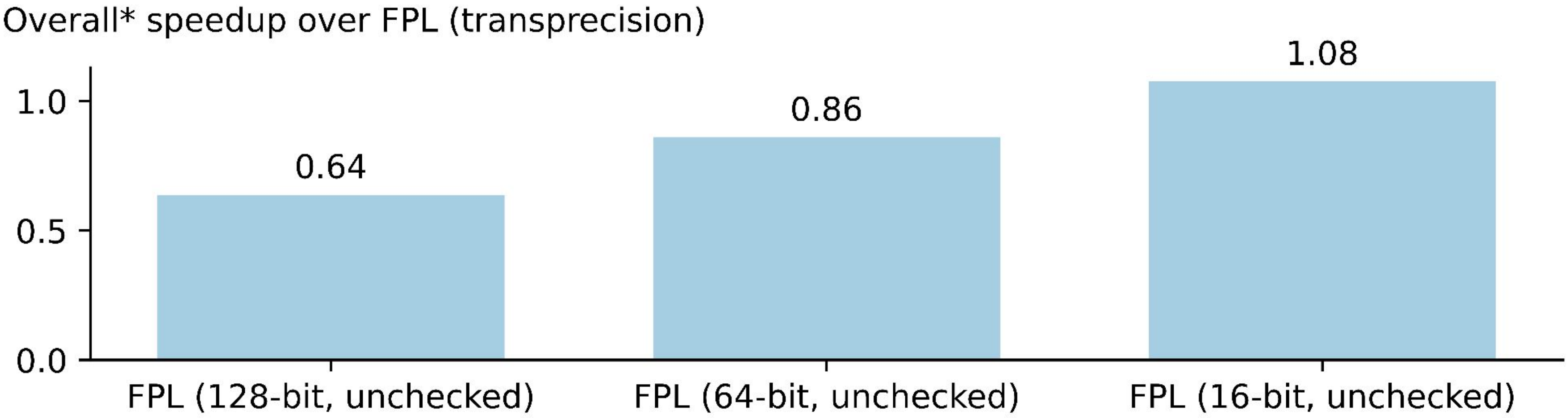
Algorithmic Design



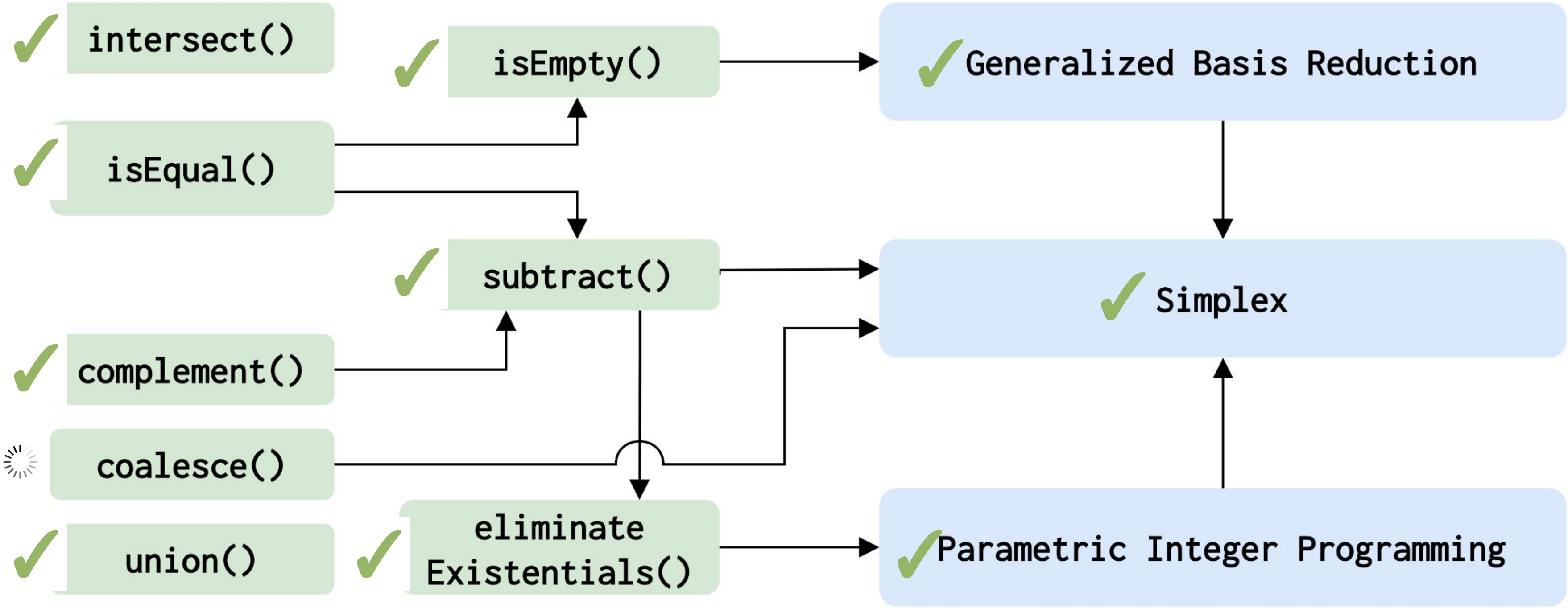
Our speedup comes from the long-running cases



Library-level transprecision has low overhead

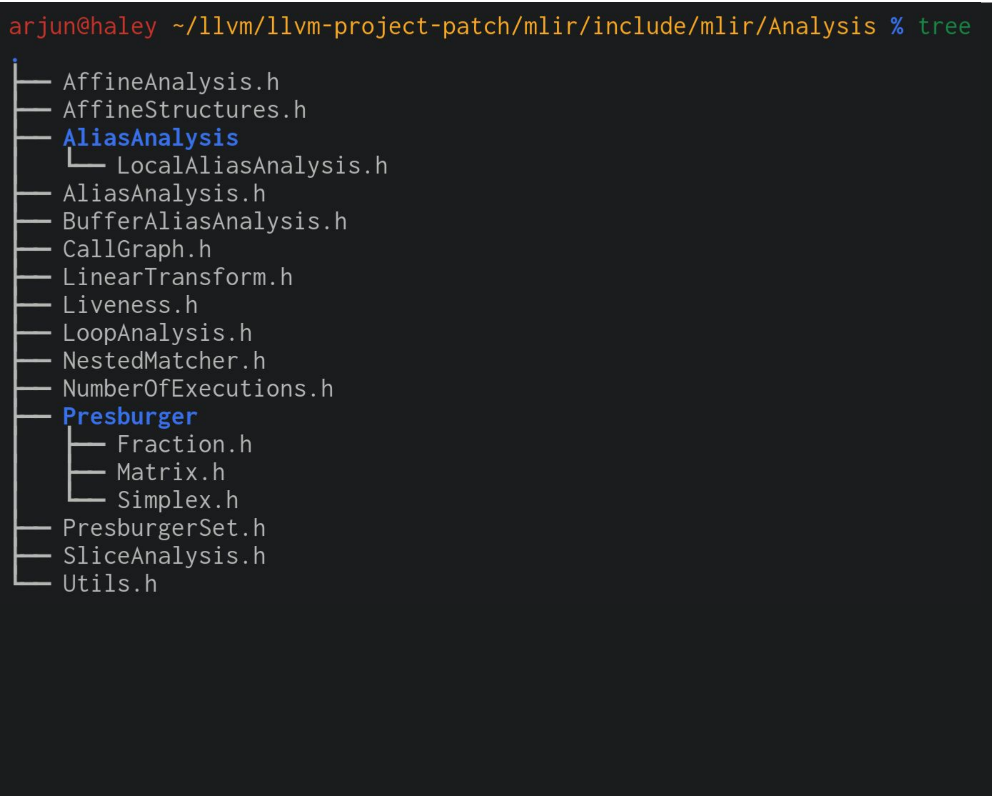


Now in MLIR — used in Affine Loop Fusion, CIRCT …

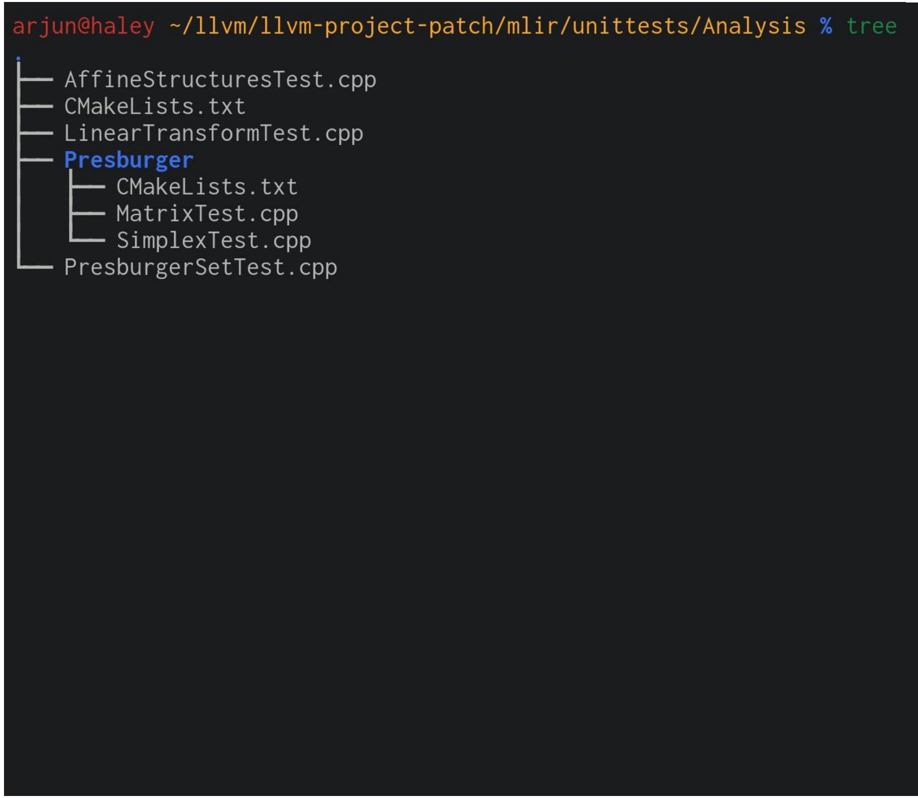


Currently uses unchecked 64-bit arithmetic; patch for introducing fast arbitrary precision is under review.

Now in MLIR



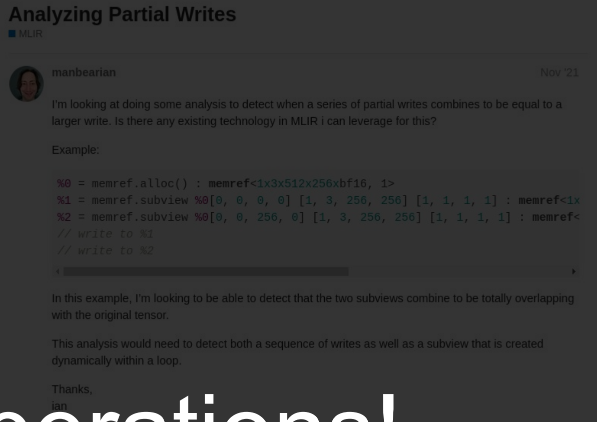
```
arjun@haley ~/llvm/llvm-project-patch/mlir/include/mlir/Analysis % tree
.
├── AffineAnalysis.h
├── AffineStructures.h
├── AliasAnalysis
│   └── LocalAliasAnalysis.h
├── AliasAnalysis.h
├── BufferAliasAnalysis.h
├── CallGraph.h
├── LinearTransform.h
├── Liveness.h
├── LoopAnalysis.h
├── NestedMatcher.h
├── NumberOfExecutions.h
├── Presburger
│   ├── Fraction.h
│   ├── Matrix.h
│   └── Simplex.h
├── PresburgerSet.h
├── SliceAnalysis.h
└── Utils.h
```



```
arjun@haley ~/llvm/llvm-project-patch/mlir/unittests/Analysis % tree
.
├── AffineStructuresTest.cpp
├── CMakeLists.txt
├── LinearTransformTest.cpp
├── Presburger
│   ├── CMakeLists.txt
│   ├── MatrixTest.cpp
│   └── SimplexTest.cpp
└── PresburgerSetTest.cpp
```

Subview fusion through equality checks

```
%0 = memref.alloc() : memref<3x512xbf16, 1>
%1 = memref.subview %0[0, 0]   [3, 256] [1, 1] : ...
%2 = memref.subview %0[0, 256] [3, 256] [1, 1] : ...
// write to %1
// write to %2
```



Analyzing Partial Writes

manbearian Nov '21

I'm looking at doing some analysis to detect when a series of partial writes combines to be equal to a larger write. Is there any existing technology in MLIR i can leverage for this?

Example:

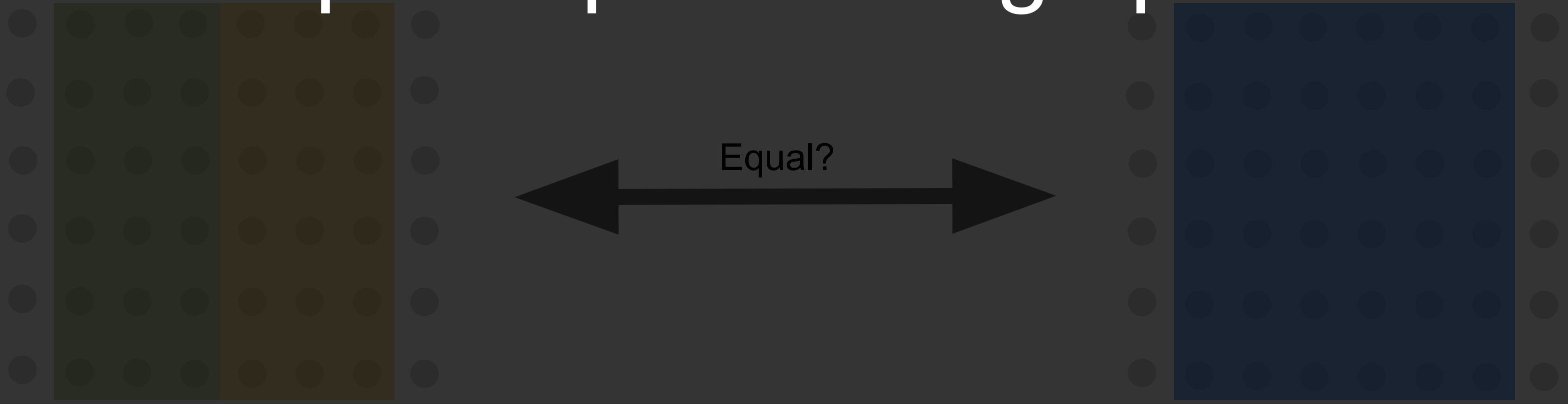
```
%0 = memref.alloc() : memref<1x3x512x256xbf16, 1>
%1 = memref.subview %0[0, 0, 0, 0] [1, 3, 256, 256] [1, 1, 1, 1] : memref<1x
%2 = memref.subview %0[0, 0, 256, 0] [1, 3, 256, 256] [1, 1, 1, 1] : memref<
// write to %1
// write to %2
```

In this example, I'm looking to be able to detect that the two subviews combine to be totally overlapping with the original tensor.

This analysis would need to detect both a sequence of writes as well as a subview that is created dynamically within a loop.

Thanks,
ian

Stop reimplementing operations!



Using FPL in MLIR

Loop Fusion: Checking for inter-loop dependencies

```
affine.for i = 0 to 4 {
     %c = affine.load %C[%i]     : memref<4xf32>
S0:  affine.store %c, %B[%i]     : memref<4xf32>
}

affine.for j = 0 to 4 {
S1:  %c = affine.load %B[3 - %j] : memref<4xf32>
     affine.store %c, %B[%j]     : memref<4xf32>
}
```

Loop Fusion: Checking for inter-loop dependencies

```
affine.for i = 0 to 4 {
     %c = affine.load %C[%i]      : memref<4xf32>
S0:  affine.store %c, %B[%i]      : memref<4xf32>
}

affine.for j = 0 to 4 {
S1:  %c = affine.load %B[3 - %j] : memref<4xf32>
     affine.store %c, %B[%j]      : memref<4xf32>
}
```

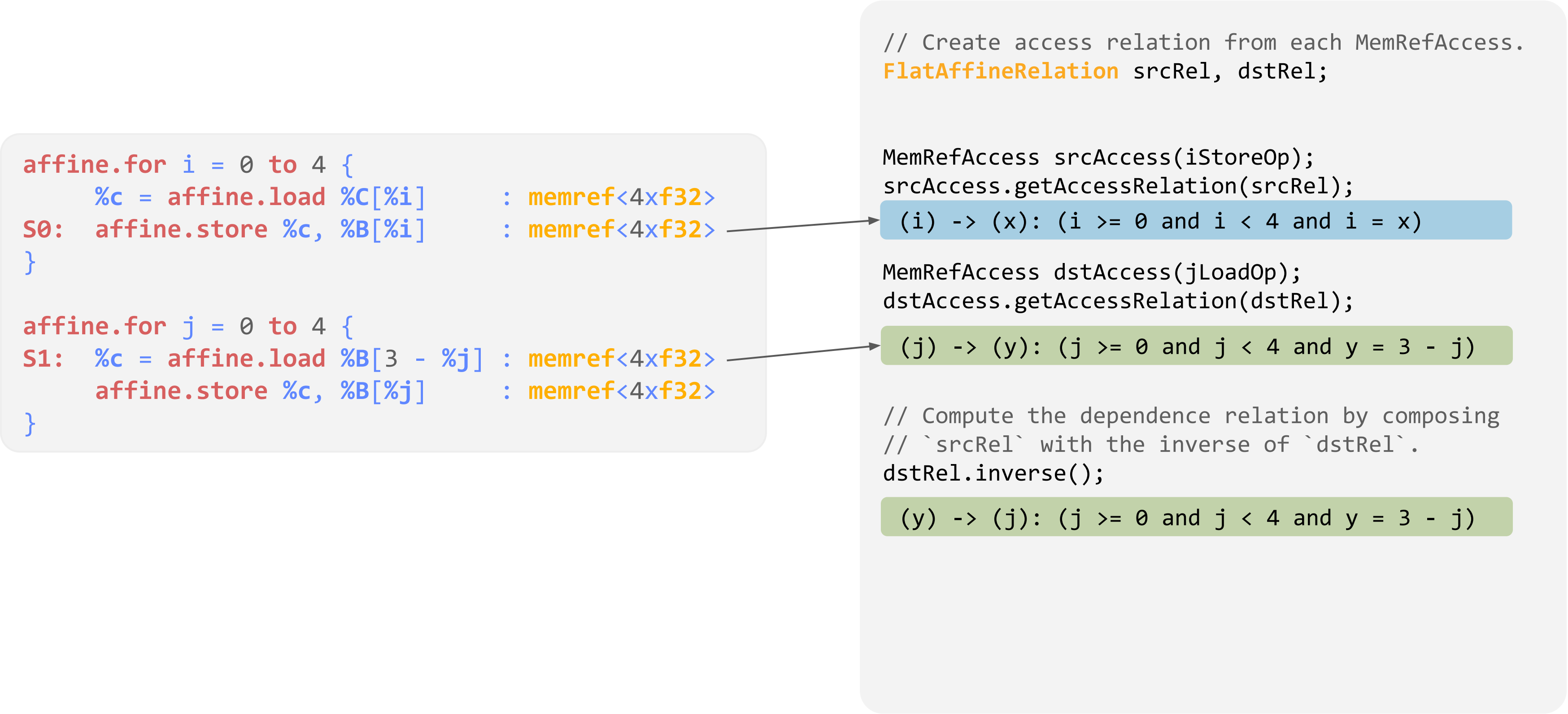
```
// Create access relation from each MemRefAccess.
FlatAffineRelation srcRel, dstRel;


MemRefAccess srcAccess(iStoreOp);
srcAccess.getAccessRelation(srcRel);
 (i) -> (x): (i >= 0 and i < 4 and i = x)

MemRefAccess dstAccess(jLoadOp);
dstAccess.getAccessRelation(dstRel);

 (j) -> (y): (j >= 0 and j < 4 and y = 3 - j)
```

Loop Fusion: Checking for inter-loop dependencies

```
affine.for i = 0 to 4 {
     %c = affine.load %C[%i]      : memref<4xf32>
S0:  affine.store %c, %B[%i]      : memref<4xf32>
}

affine.for j = 0 to 4 {
S1:  %c = affine.load %B[3 - %j] : memref<4xf32>
     affine.store %c, %B[%j]      : memref<4xf32>
}
```

```
// Create access relation from each MemRefAccess.
FlatAffineRelation srcRel, dstRel;


MemRefAccess srcAccess(iStoreOp);
srcAccess.getAccessRelation(srcRel);
 (i) -> (x): (i >= 0 and i < 4 and i = x)

MemRefAccess dstAccess(jLoadOp);
dstAccess.getAccessRelation(dstRel);

 (j) -> (y): (j >= 0 and j < 4 and y = 3 - j)
```

Loop Fusion: Checking for inter-loop dependencies



```
affine.for i = 0 to 4 {
     %c = affine.load %C[%i]      : memref<4xf32>
S0:  affine.store %c, %B[%i]      : memref<4xf32>
}

affine.for j = 0 to 4 {
S1:  %c = affine.load %B[3 - %j] : memref<4xf32>
     affine.store %c, %B[%j]      : memref<4xf32>
}
```

```
// Create access relation from each MemRefAccess.
FlatAffineRelation srcRel, dstRel;


MemRefAccess srcAccess(iStoreOp);
srcAccess.getAccessRelation(srcRel);
 (i) -> (x): (i >= 0 and i < 4 and i = x)

MemRefAccess dstAccess(jLoadOp);
dstAccess.getAccessRelation(dstRel);

 (j) -> (y): (j >= 0 and j < 4 and y = 3 - j)

// Compute the dependence relation by composing
// `srcRel` with the inverse of `dstRel`.
dstRel.inverse();

 (y) -> (j): (j >= 0 and j < 4 and y = 3 - j)
```

Loop Fusion: Checking for inter-loop dependencies

```
affine.for i = 0 to 4 {
     %c = affine.load %C[%i]     : memref<4xf32>
S0:  affine.store %c, %B[%i]     : memref<4xf32>
}

affine.for j = 0 to 4 {
S1:  %c = affine.load %B[3 - %j] : memref<4xf32>
     affine.store %c, %B[%j]     : memref<4xf32>
}
```

```
// Create access relation from each MemRefAccess.
FlatAffineRelation srcRel, dstRel;


MemRefAccess srcAccess(iStoreOp);
srcAccess.getAccessRelation(srcRel);
 (i) -> (x): (i >= 0 and i < 4 and i = x)

MemRefAccess dstAccess(jLoadOp);
dstAccess.getAccessRelation(dstRel);

 (j) -> (y): (j >= 0 and j < 4 and y = 3 - j)


// Compute the dependence relation by composing
// `srcRel` with the inverse of `dstRel`.
dstRel.inverse();
 (y) -> (j): (j >= 0 and j < 4 and y = 3 - j)
dstRel.compose(srcRel);
 (i) ->(j): (i >= 0 and i < 4 and
 j >= 0 and j < 4 and i = 3 - j)
```

Loop Fusion: Checking for inter-loop dependencies

```
affine.for i = 0 to 4 {
     %c = affine.load %C[%i]       : memref<4xf32>
S0:  affine.store %c, %B[%i]       : memref<4xf32>
}

affine.for j = 0 to 4 {
S1:  %c = affine.load %B[3 - %j] : memref<4xf32>
     affine.store %c, %B[%j]       : memref<4xf32>
}
```

```
// Create access relation from each MemRefAccess.
FlatAffineRelation srcRel, dstRel;


MemRefAccess srcAccess(iStoreOp);
srcAccess.getAccessRelation(srcRel);
  (i) -> (x): (i >= 0 and i < 4 and i = x)

MemRefAccess dstAccess(jLoadOp);
dstAccess.getAccessRelation(dstRel);

  (j) -> (y): (j >= 0 and j < 4 and y = 3 - j)

// Compute the dependence relation by composing
// `srcRel` with the inverse of `dstRel`.
dstRel.inverse();
  (y) -> (j): (j >= 0 and j < 4 and y = 3 - j)
dstRel.compose(srcRel);
  (i) ->(j): (i >= 0 and i < 4 and
  j >= 0 and j < 4 and i = 3 - j)
 bool hasDependency = dstRel.isIntegerEmpty();
```

Scaling the Community

Top 1.3% of LLVM Contributors (2,929 overall)

32015	Chris Lattner
9508	Craig Topper
8456	Simon Pilgrim
7376	Rafael Espindola
6108	Ted Kremenek
5572	Sanjay Patel
5441	Daniel Dunbar
5371	Evan Cheng
5362	Douglas Gregor
5311	Dan Gohman
4859	Matt Arsenault
4735	Benjamin Kramer
4591	Rui Ueyama
4521	Richard Smith
3939	Chandler Carruth
3704	Reid Spencer
3672	Bill Wendling
3511	Eric Christopher
3150	Reid Kleckner
3104	David Blaikie
3095	Nico Weber
3009	Fariborz Jahanian
3006	Fangrui Song
2924	Greg Clayton
2886	NAKAMURA Takumi
2793	Eli Friedman
2776	Devang Patel
2523	Jim Grosbach
2452	Argyrios Kyrtzidis
2448	Jakob Stoklund Olesen
2381	Pavel Labath
2309	***Tobias Grosser***
2269	Owen Anderson
2219	Anders Carlsson
2202	Eric Fiselier
2170	Johnny Chen
2148	Lang Hames
2143	Zachary Turner

Building a Career on Open-Source Research

32. Most Contributions

https://grosser.science/FPL

Fast Polyhedral Core Library

HiPEAC Technology Transfer Award 2022

- 52 patches committed to ***MLIR***
- Arjun and Kunwar have *commit rights*
- Weekly public video calls

Objectives

Open Source

Open Development

Active Community

Arjun
IIIT Hyderabad

Christian
ETH Zurich

Michel
ETH Zurich

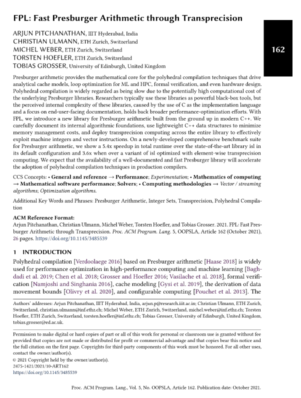
Kunwar
IIIT Hyderabad

Distributed Developer Team

Proposal

Acceptance

Open Development



FPL: Fast Presburger Arithmetic through Transprecision

162

ARJUN PITCHANATHAN, IIIT Hyderabad, India
CHRISTIAN ULMANN, ETH Zurich, Switzerland
MICHEL WEBER, ETH Zurich, Switzerland
TORSTEN HOEFLER, ETH Zurich, Switzerland
TOBIAS GROSSER, University of Edinburgh, United Kingdom

Presburger arithmetic provides the mathematical core for the polyhedral compilation techniques that drive analytical cache models, loop optimization for ML and HPC, formal verification, and even hardware design. Polyhedral compilation is widely regarded as being slow due to the potentially high computational cost of the underlying Presburger libraries. Researchers typically use these libraries as powerful black-box tools, but the perceived internal complexity of these libraries, caused by the use of C as the implementation language and a focus on end-user-facing documentation, holds back broader performance-optimization efforts. With FPL, we introduce a new library for Presburger arithmetic built from the ground up in modern C++. We carefully document its internal algorithmic foundations, use lightweight C++ data structures to minimize memory management costs, and deploy transprecision computing across the entire library to effectively exploit machine integers and vector instructions. On a newly-developed comprehensive benchmark suite for Presburger arithmetic, we show a 5.4x speedup in total runtime over the state-of-the-art library isl in its default configuration and 3.6x when over a variant of isl optimized with element-wise transprecision computing. We expect that the availability of a well-documented and fast Presburger library will accelerate the adoption of polyhedral compilation techniques in production compilers.

CCS Concepts: • **General and reference → Performance**; *Experimentation*; • **Mathematics of computing → Mathematical software performance**; **Solvers**; • **Computing methodologies →** *Vector / streaming algorithms*; *Optimization algorithms*.

Additional Key Words and Phrases: Presburger Arithmetic, Integer Sets, Transprecision, Polyhedral Compilation

ACM Reference Format:
Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. 2021. FPL: Fast Presburger Arithmetic through Transprecision. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 162 (October 2021), 26 pages. https://doi.org/10.1145/3485539

1 INTRODUCTION

Polyhedral compilation [Verdoolaege 2016] based on Presburger arithmetic [Haase 2018] is widely used for performance optimization in high-performance computing and machine learning [Baghdadi et al. 2019; Chen et al. 2018; Grosser and Hoefler 2016; Vasilache et al. 2018], formal verification [Namjoshi and Singhania 2016], cache modeling [Gysi et al. 2019], the derivation of data movement bounds [Olivry et al. 2020], and configurable computing [Pouchet et al. 2013]. The

Authors' addresses: Arjun Pitchanathan, IIIT Hyderabad, India, arjun.p@research.iiit.ac.in; Christian Ulmann, ETH Zurich, Switzerland, christian.ulmann@inf.ethz.ch; Michel Weber, ETH Zurich, Switzerland, michel.weber@inf.ethz.ch; Torsten Hoefler, ETH Zurich, Switzerland, torsten.hoefler@inf.ethz.ch; Tobias Grosser, University of Edinburgh, United Kingdom, tobias.grosser@ed.ac.uk.

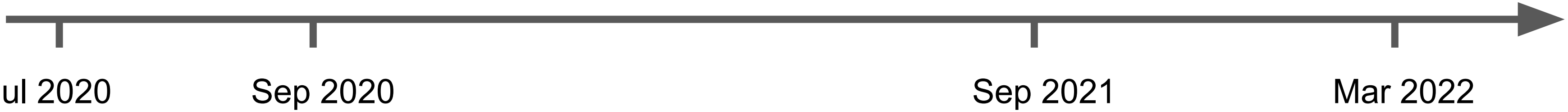
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/10-ART162
https://doi.org/10.1145/3485539

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 162. Publication date: October 2021.

Distinguished Paper @ OOPSLA 21

Presentation @
LLVM Developer Meeting 2021

External Users: 唐适之 @ Tsinghua, Intel, PolyMage, …



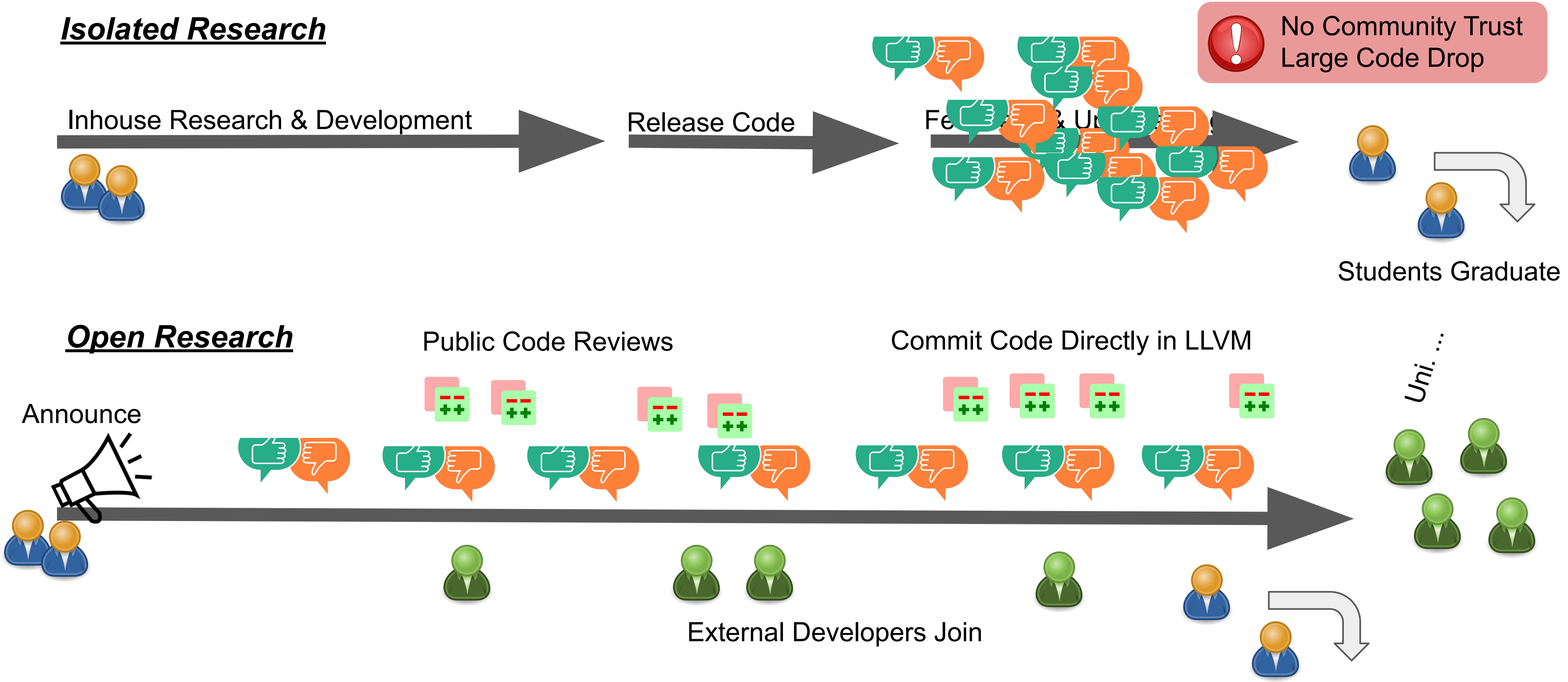
Jul 2020

Sep 2020

Sep 2021

Mar 2022

Building Open Communities



Isolated Research

Inhouse Research & Development

Release Code

Fe & U

No Community Trust
Large Code Drop

Students Graduate

Open Research

Announce

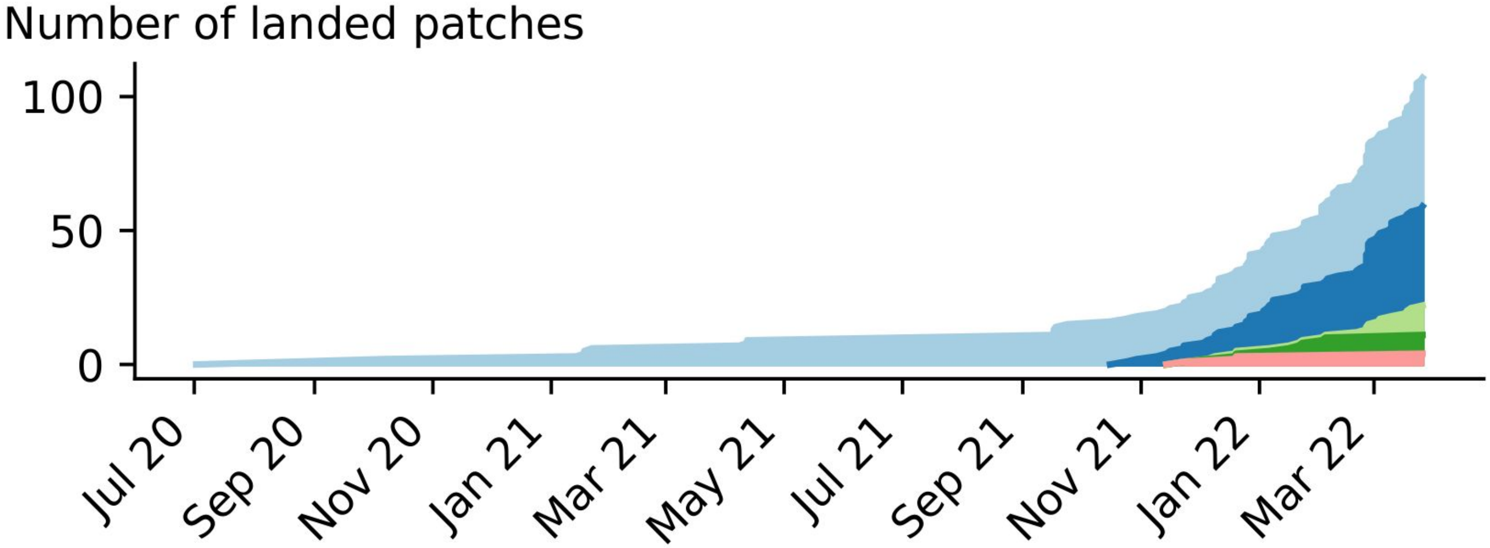
Public Code Reviews

Commit Code Directly in LLVM

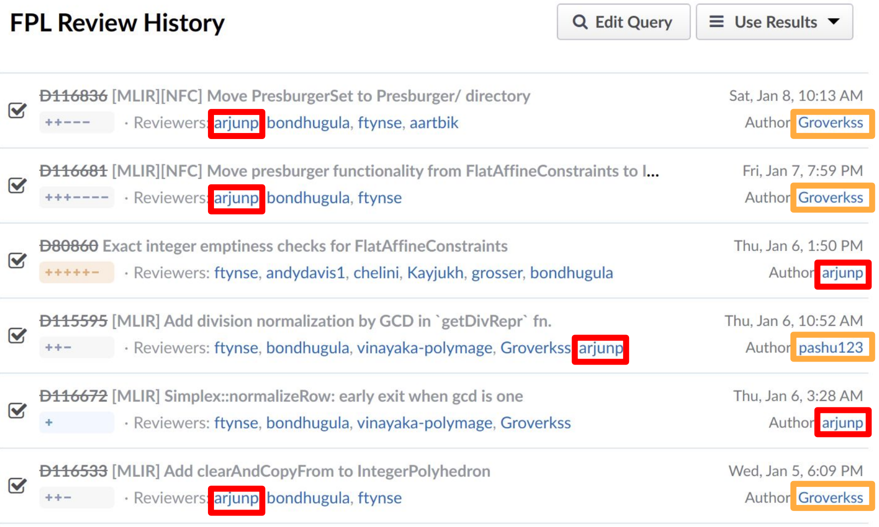
Uni. ...

External Developers Join

FPL’s growing upstream development community



13 Public Code Reviews in 13 Days ~ 1 discussion / day



FPL Review History

☑ ~~D116836~~ [MLIR][NFC] Move PresburgerSet to Presburger/ directory — Sat, Jan 8, 10:13 AM
++--- · Reviewers: arjunp, bondhugula, ftynse, aartbik — Author: Groverkss

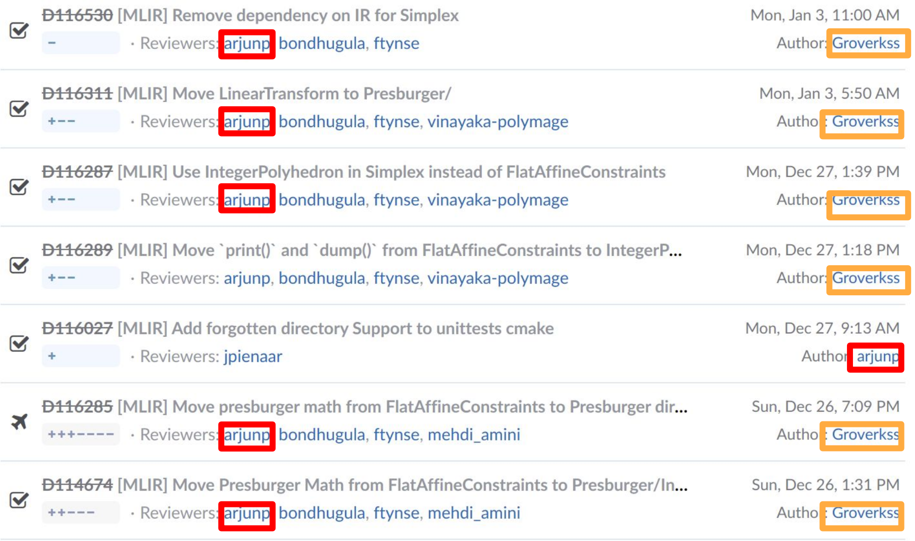
☑ ~~D116681~~ [MLIR][NFC] Move presburger functionality from FlatAffineConstraints to I... — Fri, Jan 7, 7:59 PM
+++---- · Reviewers: arjunp, bondhugula, ftynse — Author: Groverkss

☑ ~~D80860~~ Exact integer emptiness checks for FlatAffineConstraints — Thu, Jan 6, 1:50 PM
+++++- · Reviewers: ftynse, andydavis1, chelini, Kayjukh, grosser, bondhugula — Author: arjunp

☑ ~~D115595~~ [MLIR] Add division normalization by GCD in `getDivRepr` fn. — Thu, Jan 6, 10:52 AM
++- · Reviewers: ftynse, bondhugula, vinayaka-polymage, Groverkss, arjunp — Author: pashu123

☑ ~~D116672~~ [MLIR] Simplex::normalizeRow: early exit when gcd is one — Thu, Jan 6, 3:28 AM
+ · Reviewers: ftynse, bondhugula, vinayaka-polymage, Groverkss — Author: arjunp

☑ ~~D116533~~ [MLIR] Add clearAndCopyFrom to IntegerPolyhedron — Wed, Jan 5, 6:09 PM
++- · Reviewers: arjunp, bondhugula, ftynse — Author: Groverkss



☑ ~~D116530~~ [MLIR] Remove dependency on IR for Simplex — Mon, Jan 3, 11:00 AM
- · Reviewers: arjunp, bondhugula, ftynse — Author: Groverkss

☑ ~~D116311~~ [MLIR] Move LinearTransform to Presburger/ — Mon, Jan 3, 5:50 AM
+-- · Reviewers: arjunp, bondhugula, ftynse, vinayaka-polymage — Author: Groverkss

☑ ~~D116287~~ [MLIR] Use IntegerPolyhedron in Simplex instead of FlatAffineConstraints — Mon, Dec 27, 1:39 PM
+-- · Reviewers: arjunp, bondhugula, ftynse, vinayaka-polymage — Author: Groverkss

☑ ~~D116289~~ [MLIR] Move `print()` and `dump()` from FlatAffineConstraints to IntegerP... — Mon, Dec 27, 1:18 PM
+-- · Reviewers: arjunp, bondhugula, ftynse, vinayaka-polymage — Author: Groverkss

☑ ~~D116027~~ [MLIR] Add forgotten directory Support to unittests cmake — Mon, Dec 27, 9:13 AM
+ · Reviewers: jpienaar — Author: arjunp

✈ ~~D116285~~ [MLIR] Move presburger math from FlatAffineConstraints to Presburger dir... — Sun, Dec 26, 7:09 PM
+++---- · Reviewers: arjunp, bondhugula, ftynse, mehdi_amini — Author: Groverkss

☑ ~~D114674~~ [MLIR] Move Presburger Math from FlatAffineConstraints to Presburger/In... — Sun, Dec 26, 1:31 PM
++--- · Reviewers: arjunp, bondhugula, ftynse, mehdi_amini — Author: Groverkss

🟥 Future Edinburgh Student

🟧 External Contributor

Source: https://reviews.llvm.org/differential/query/cP8Iij2thxgW/

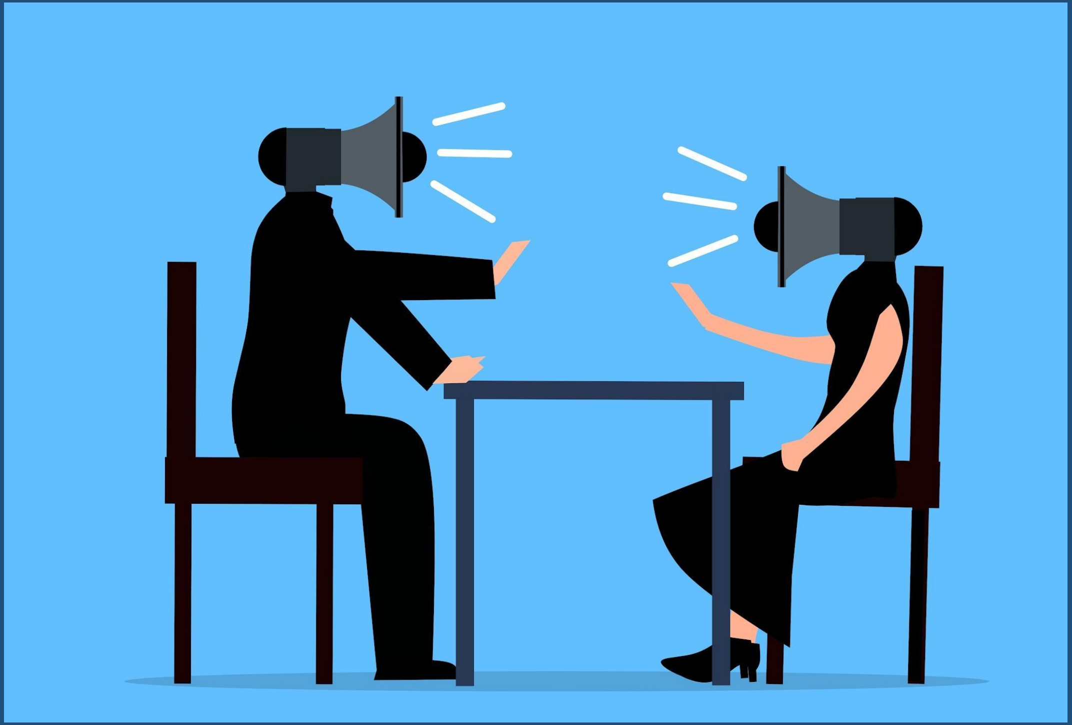
FPL - A Growing Developer Community

Arjun Pitchanthan

Kunwar Grover

Abhinav Menon

Bharathi Ramana Joshi



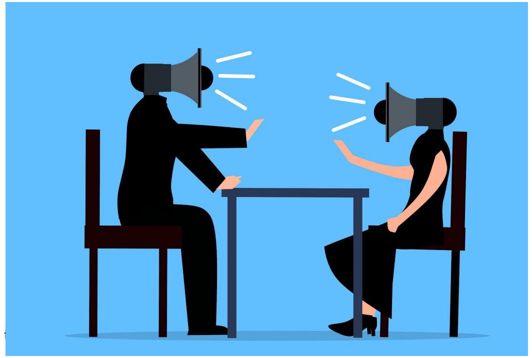
Discussion, where are we heading?

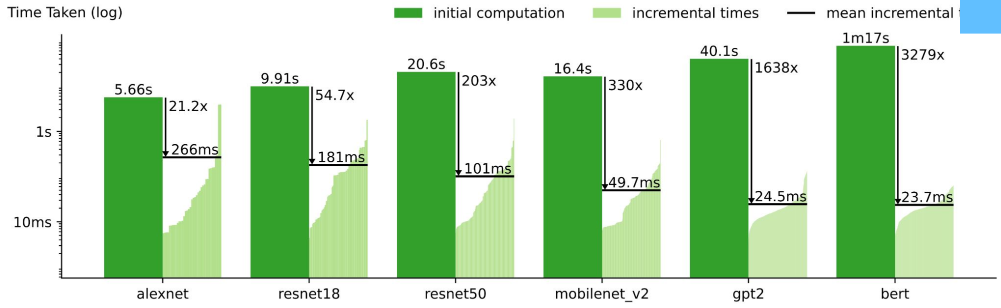
Polyhedral Algorithms are too expensive to Scale?

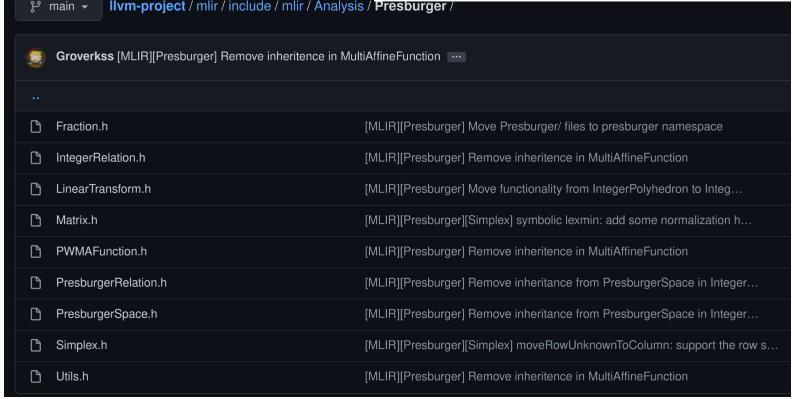
Is Polyhedral Compilation More Than Polyhedral Loop Scheduling?

How Can We Mix Polyhedral Compilation and Real-World Compilers?

Conclusion







llvm / llvm-project Public

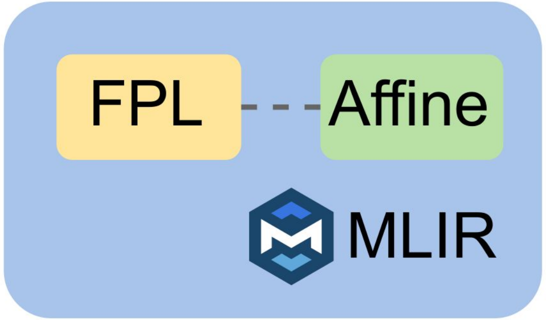
Code Issues 5k+ Pull requests Action

main llvm-project / mlir / include / mlir / Analysis / Presburger /

Groverkss [MLIR][Presburger] Remove inheritence in MultiAffineFunction ···

..

Fraction.h	[MLIR][Presburger] Move Presburger/ files to presburger namespace
IntegerRelation.h	[MLIR][Presburger] Remove inheritence in MultiAffineFunction
LinearTransform.h	[MLIR][Presburger] Move functionality from IntegerPolyhedron to Integ…
Matrix.h	[MLIR][Presburger][Simplex] symbolic lexmin: add some normalization h…
PWMAFunction.h	[MLIR][Presburger] Remove inheritence in MultiAffineFunction
PresburgerRelation.h	[MLIR][Presburger] Remove inheritance from PresburgerSpace in Integer…
PresburgerSpace.h	[MLIR][Presburger] Remove inheritance from PresburgerSpace in Integer…
Simplex.h	[MLIR][Presburger][Simplex] moveRowUnknownToColumn: support the row s…
Utils.h	[MLIR][Presburger] Remove inheritence in MultiAffineFunction



FPL

Affine

MLIR