MLIR Pattern Matching for Library Acceleration Instruction Rewriting

Vinicius Espindola
v188115@gmail.com
Guido Araújo
guido@unicamp.br

University of Campinas (Unicamp)
Institute of Computing (IC)
Computer Systems Laboratory (LSC)

MLIR Open Meeting • July 6th, 2023

UNICAMP

Agenda

1. Introduction
2. SMR
3. Algorithm
4. Results

Introduction

Introduction - Context

- Restrictive abstraction lowering process
- High-level hardware accelerators
- Raise the abstraction level

Introduction - Existing solutions

- IDL
- KernelFaRer
- MLT
- Difficult to write patterns

Introduction - Goals

- ▷ New rewriting tool
- ▷ Raising - rewrite complex patterns
- ▷ Easy - simple rewrite specification
- ▷ Embeddable - existing compilation flows

SMR

1. Introduction
2. **SMR**
3. Algorithm
4. Results

SMR - Overview

- **What is SMR?**
  Source-based Matching and Rewriting
- Tool for easily rewriting code
- Specify rewrites at source code-level
- SMR matches/replaces at MLIR level
- Outputs optimized MLIR

SMR - Foundation

- Tools for the job
- MLIR
  - High-level IR
  - Multiple frontends
- TWIG
  - Compiler made by Aho
  - Clever ideas to encode patterns as string-based automata

SMR - Usage

- Input:

```
program matrix_multiplication
  integer :: i, j, k
  double precision, dimension(3,3) :: a, b, c

  a = reshape([1.0, 2.0, 3.0, 4.0], [2, 2])
  b = reshape([10.0, 11.0, 12.0, 13.0], [2, 2])
  c = 0.0

  do i = 1, 3
    do j = 1, 3
      do k = 1, 3
        c(i, j) = c(i, j) + a(i, k) * b(k, j)
      end do
    end do
  end do

  print *, 'Result:'
  do i = 1, 3
      print '(3F8.2)', c(i, :)
  end do
end program matrix_multiplication
```

- Rewrites (PAT file):

```
f90 {
  subroutine gemm_double(i, j, k, a, b, c)
    integer :: i, j, k
    double precision, dimension(3,3) :: a, b, c
    do i = 1, 3
      do j = 1, 3
        do k = 1, 3
          c(i, j) = c(i, j) + a(i, k) * b(k, j)
        end do
      end do
    end do
  end subroutine
}={
  subroutine gemm_double(i, j, k, a, b, c)
    integer :: i, j, k
    double precision, dimension(3,3) :: a, b, c
    external :: dgemm
    call dgemm('N', 'N', 3, 3, 3, 1.0D0,
               a, 3, b, 3, 0.0D0, c, 3)
  end subroutine
}
```

SMR - Usage

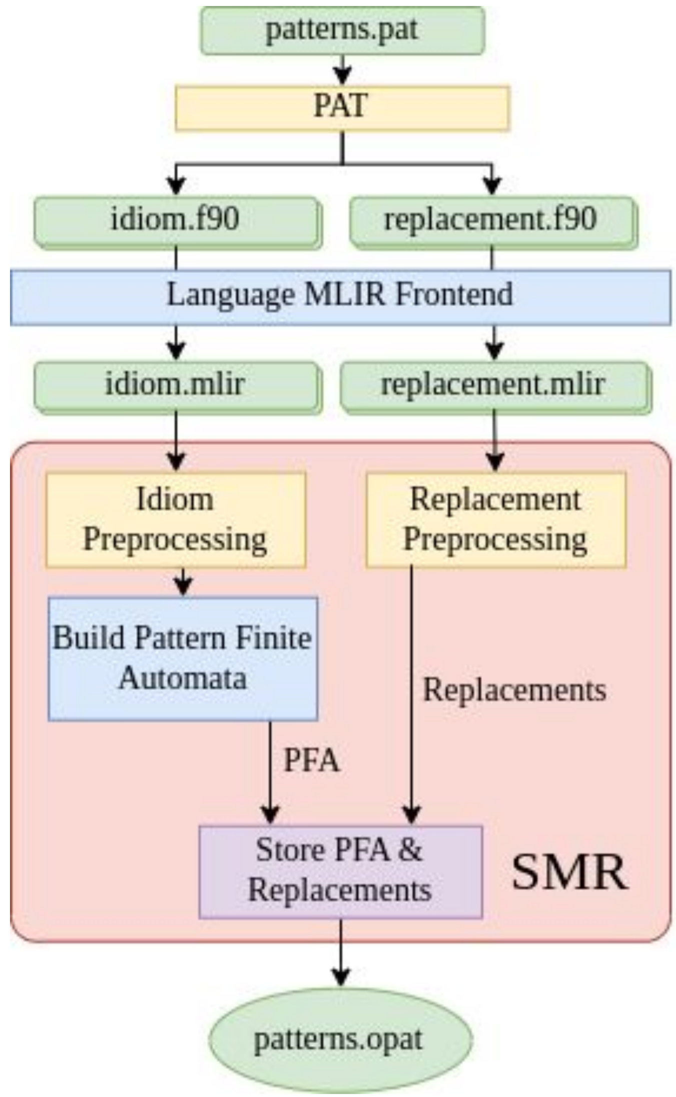
- Serialize PAT file

```
smr rewrites.pat --serialize=./rewrites.opat
```

- Apply rewrites to some input

```
smr input.f90 rewrites.opat -o input-opt.mlir
```

SMR - Serialization

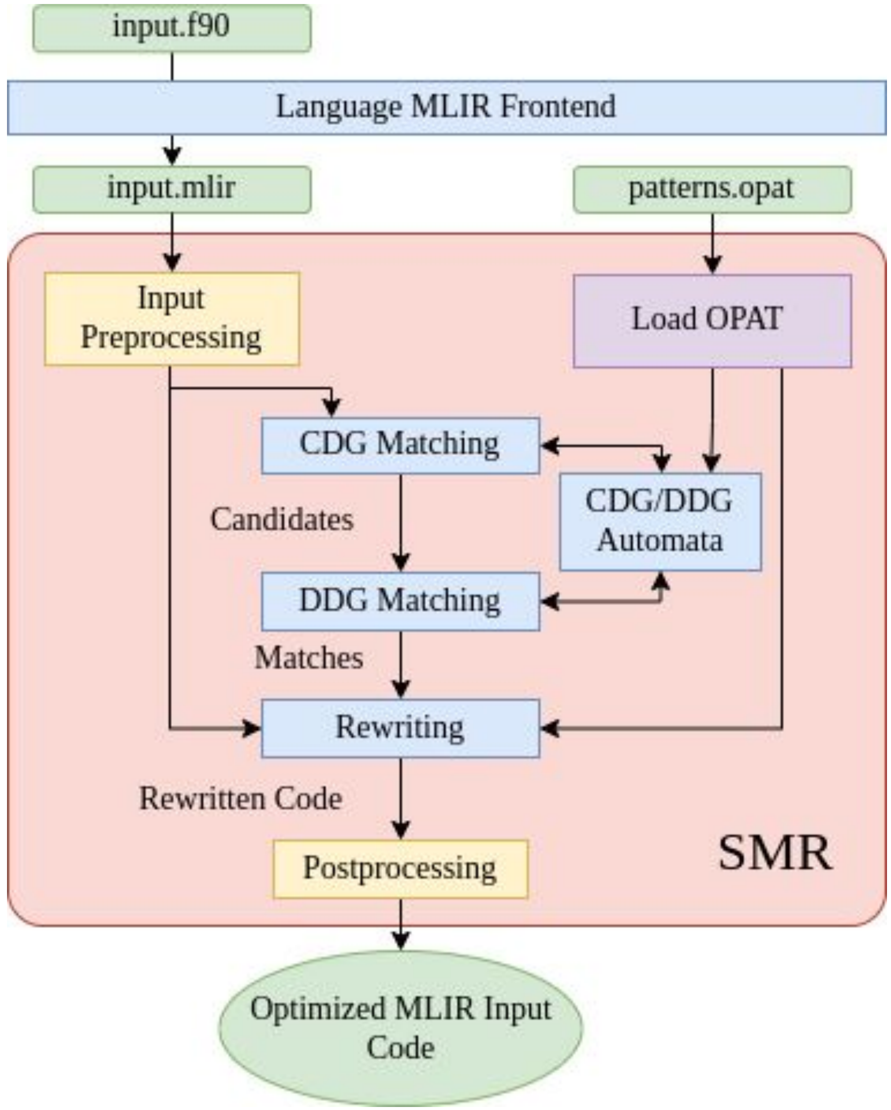



```
smr rewrites.pat --serialize=./rewrites.opat
```

SMR - Serialization

- ▷ Why serialize the PAT file?
- ▷ **Reusability**
- ▷ Compile code and build automata only once
- ▷ Avoid overhead in future reuses
- ▷ OPAT is like a “library of patterns”

SMR - Matching




```
smr input.f90 rewrites.opat -o input-opt.mlir
```

Algorithm

Algorithm - Overview

- Parse PAT file
- Lower source code to MLIR
- Match control structure
  - Control Dependency Graph (CDG)
- Match data flow
  - Data Dependency Graph (DDG)
- **Is a match?** Rewrite.

Algorithm - Input

Input Code:

```
subroutine input (abs, n, array)
    INTEGER, DIMENSION(n) :: array
    INTEGER :: n, v, t

    abs = 0
    DO i = 1, n
        IF (array(i) > 0) THEN
            abs = abs + n
        ELSE IF (array(i) < 0) THEN
            abs = abs - n
        END IF
    END DO

    IF (abs > 0) THEN
        !!!! SNIPPET TO MATCH !!!!
        IF (t == 1) THEN
            v = 1
        ELSE
            v = v - 1
            IF (v == 1) THEN
                t = 0
            END IF
        END IF
        !!!!!!!!!!!!!!!!!!!!!!!!!!
    END IF

end subroutine
```

PAT File:

```
f90 {
  subroutine sum(test, val)
    integer :: val, test
    IF (test == 1) THEN
      val = 1
    ELSE
      val = val - 1
      IF (val == 1) THEN
        test = 0
      END IF
    END IF
  end subroutine
} = {
  subroutine sum (test, val)
    integer :: val, test
    call some_lib(test, val)
  end subroutine
}
```

Algorithm - PAT Language

<lang> {
<pattern_code>
} = {
<replacement_code>
}

<lang> := f
| f90
| c
| cc

f, f90 → flang

c, cc → cil

PAT File:

```
f90 {
  subroutine sum(test, val)
    integer :: val, test
    IF (test == 1) THEN
      val = 1
    ELSE
      val = val - 1
      IF (val == 1) THEN
        test = 0
      END IF
    END IF
  end subroutine
} = {
  subroutine sum (test, val)
    integer :: val, test
    call some_lib(test, val)
  end subroutine
}
```

Algorithm - PAT Parsing

Input Code:

```
subroutine input (abs, n, array)
    INTEGER, DIMENSION(n) :: array
    INTEGER :: n, v, t

    abs = 0
    DO i = 1, n
        IF (array(i) > 0) THEN
            abs = abs + n
        ELSE IF (array(i) < 0) THEN
            abs = abs - n
        END IF
    END DO

    IF (abs > 0) THEN
        !!!! SNIPPET TO MATCH !!!!
        IF (t == 1) THEN
            v = 1
        ELSE
            v = v - 1
            IF (v == 1) THEN
                t = 0
            END IF
        END IF
        !!!!!!!!!!!!!!!!!!!!!!!!!!
    END IF

end subroutine
```

Pattern Code:

```
subroutine sum(test, val)
  integer :: val, test

  IF (test == 1) THEN
    val = 1
  ELSE
    val = val - 1
    IF (val == 1) THEN
      test = 0
    END IF
  END IF

end subroutine
```

Replacement Code:

```
subroutine sum (test, val)
    integer :: val, test

    call some_lib(test, val)

end subroutine
```

Algorithm - Wrapper functions

Pattern:

```
subroutine sum(test, val)
  integer :: val, test

  IF (test == 1) THEN
    val = 1
  ELSE
    val = val - 1
    IF (val == 1) THEN
      test = 0
    END IF
  END IF

end subroutine
```

Replacement:

```
subroutine sum (test, val)
    integer :: val, test

    call some_lib(test, val)

end subroutine
```

- ▷ Functions are not matched
- ▷ Make code valid
- ▷ Map input variables

Algorithm - Compilation

▷ Lower inputs to MLIR

```fortran
subroutine sum(test, val)
  integer :: val, test

  IF (test == 1) THEN
    val = 1
  ELSE
    val = val - 1
    IF (val == 1) THEN
      test = 0
    END IF
  END IF

end subroutine
```

FIR →

```mlir
"func"() ( {
^bb0(%arg0: !fir.ref<i32>, %arg1: !fir.ref<i32>):  // no predecessors
  %c1_i32 = "std.constant"() {value = 1 : i32} : () -> i32
  %c0_i32 = "std.constant"() {value = 0 : i32} : () -> i32
  %0 = "fir.load"(%arg0) : (!fir.ref<i32>) -> i32
  %1 = "std.cmpi"(%0, %c1_i32) {predicate = 0 : i64} : (i32, i32) -> i1
  "fir.if"(%1) ( {
    "fir.store"(%c1_i32, %arg1) : (i32, !fir.ref<i32>) -> ()
    "fir.result"() : () -> ()
  }, {
    %2 = "fir.load"(%arg1) : (!fir.ref<i32>) -> i32
    %3 = "std.subi"(%2, %c1_i32) : (i32, i32) -> i32
    "fir.store"(%3, %arg1) : (i32, !fir.ref<i32>) -> ()
    %4 = "std.cmpi"(%3, %c1_i32) {predicate = 0 : i64} : (i32, i32) -> i1
    "fir.if"(%4) ( {
      "fir.store"(%c0_i32, %arg0) : (i32, !fir.ref<i32>) -> ()
      "fir.result"() : () -> ()
    }, {
      "fir.result"() : () -> ()
    }) : (i1) -> ()
    "fir.result"() : () -> ()
  }) : (i1) -> ()
  "std.return"() : () -> ()
}) {sym_name = "_QPsum", type = (!fir.ref<i32>, !fir.ref<i32>) -> ()} : () -> ()
```

Algorithm - Control Dependency Graph

- We know the pattern/input control structure
- Must match control structure
- Represent input and pattern as CDG
- Match input and pattern CDG in automaton

Algorithm - Control Dependency Graph

- Transform input and pattern MLIR into CDG

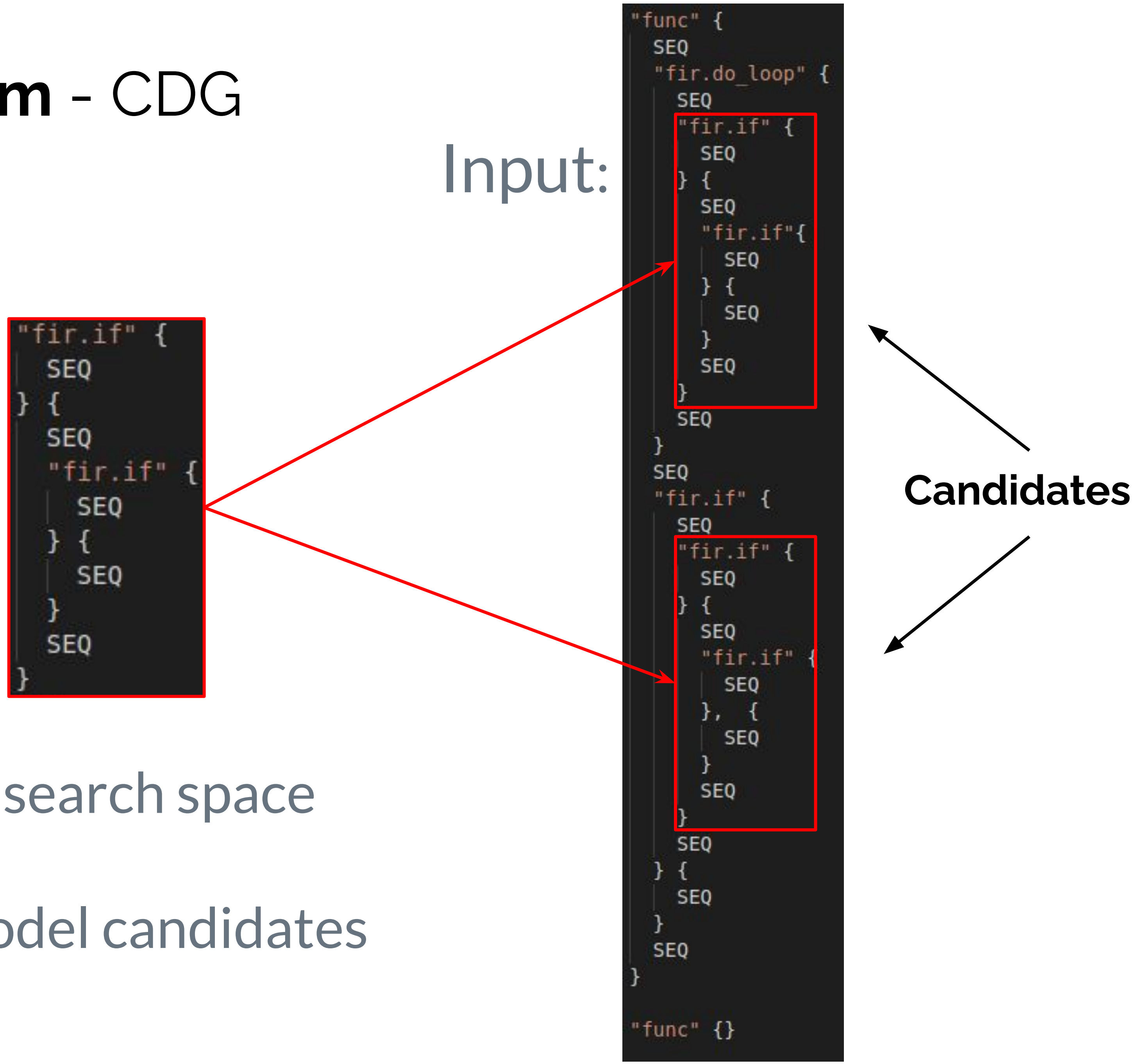
Pattern:

```
"func"() ( {
^bb0(%arg0: !fir.ref<i32>, %arg1: !fir.ref<i32>):  // no predecessors
  %c1_i32 = "std.constant"() {value = 1 : i32} : () -> i32
  %c0_i32 = "std.constant"() {value = 0 : i32} : () -> i32
  %0 = "fir.load"(%arg0) : (!fir.ref<i32>) -> i32
  %1 = "std.cmpi"(%0, %c1_i32) {predicate = 0 : i64} : (i32, i32) -> i1
  "fir.if"(%1) ( {
    "fir.store"(%c1_i32, %arg1) : (i32, !fir.ref<i32>) -> ()
    "fir.result"() : () -> ()
  },  {
    %2 = "fir.load"(%arg1) : (!fir.ref<i32>) -> i32
    %3 = "std.subi"(%2, %c1_i32) : (i32, i32) -> i32
    "fir.store"(%3, %arg1) : (i32, !fir.ref<i32>) -> ()
    %4 = "std.cmpi"(%3, %c1_i32) {predicate = 0 : i64} : (i32, i32) -> i1
    "fir.if"(%4) ( {
      "fir.store"(%c0_i32, %arg0) : (i32, !fir.ref<i32>) -> ()
      "fir.result"() : () -> ()
    },  {
      "fir.result"() : () -> ()
    }) : (i1) -> ()
    "fir.result"() : () -> ()
  }) : (i1) -> ()
  "std.return"() : () -> ()
}) {sym_name = "_QPsum", type = (!fir.ref<i32>, !fir.ref<i32>) -> ()} : () -> ()
```

```
"fir.if" {
  SEQ
} {
  SEQ
  "fir.if" {
    SEQ
  } {
    SEQ
  }
  SEQ
}
```

Algorithm - CDG

Pattern:



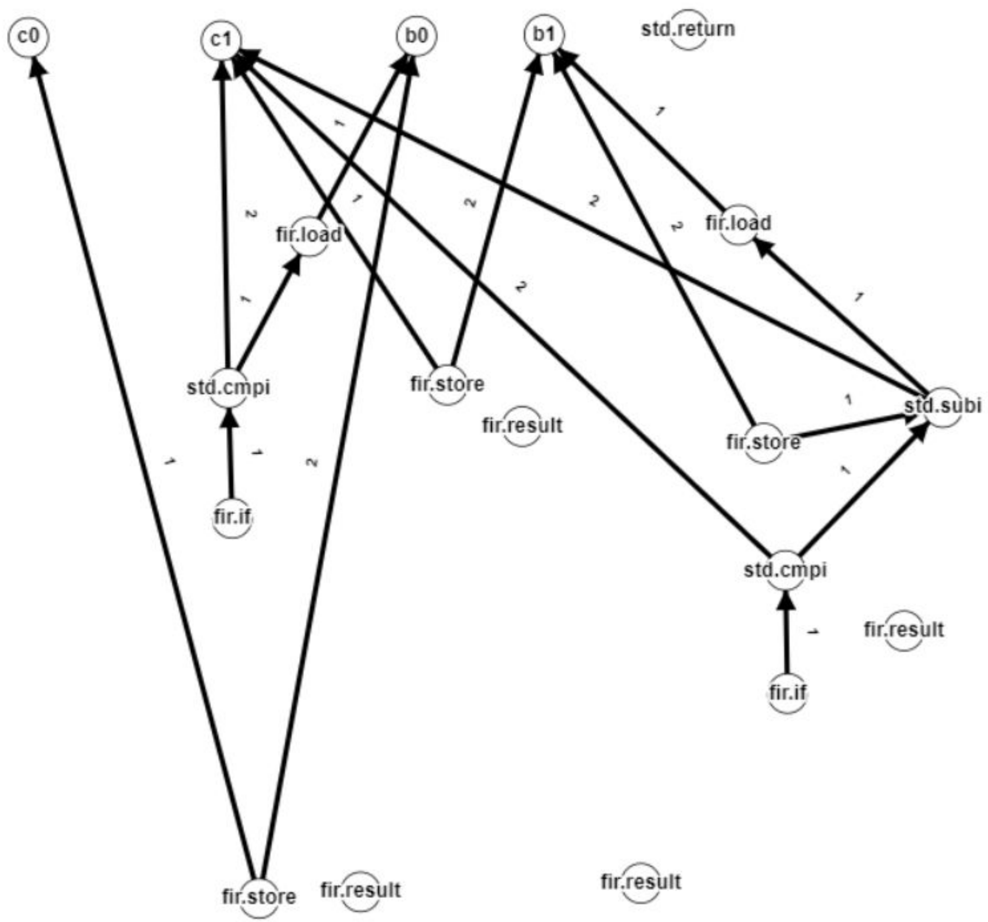
- ▷ Reduce search space
- ▷ Only model candidates

Algorithm - Data Dependency Graph

- CDG matched, but it's not enough.
- Same control structure =/= Same computation
- Must match data flow within each region
- **Enter the Data Dependency Graph (DDG)**

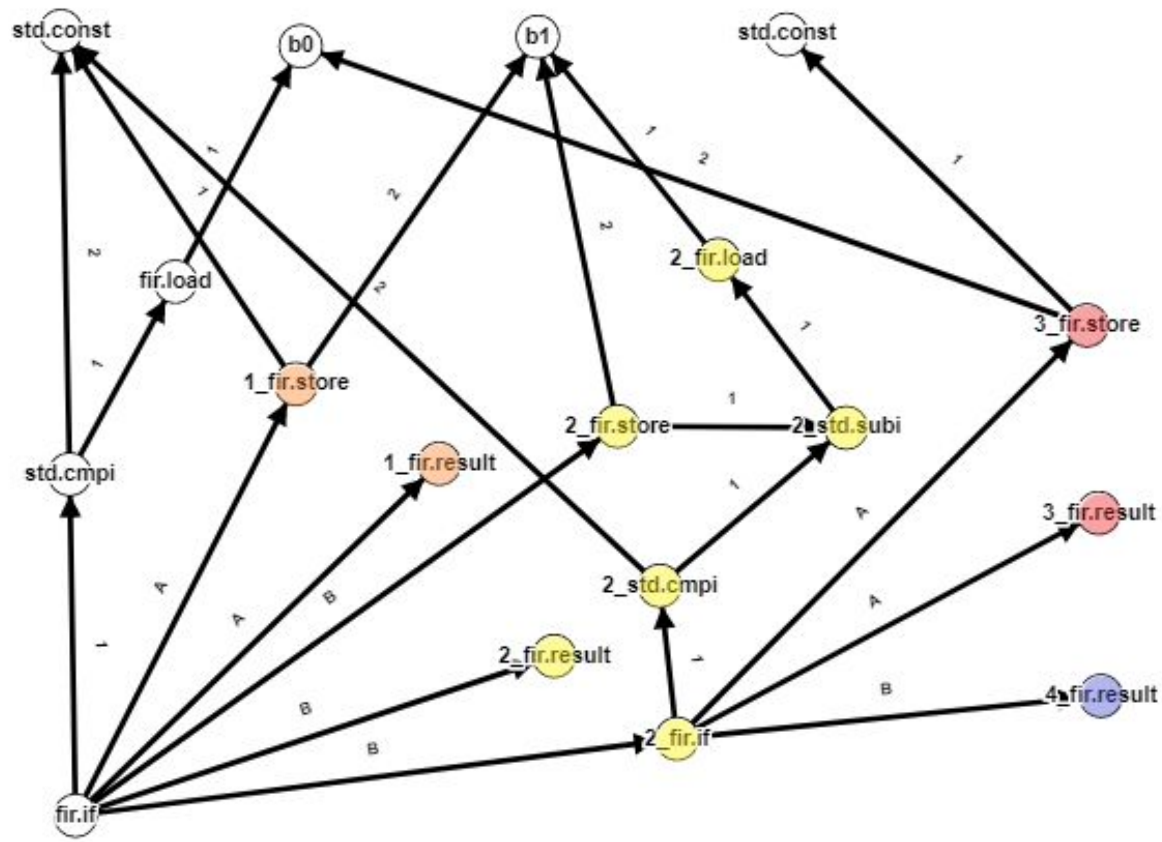
Algorithm - Data Dependency Graph

- Use-def chain graph



Algorithm - Data Dependency Graph

- Color regions and add region edges



Algorithm - Dialect-wise configuration

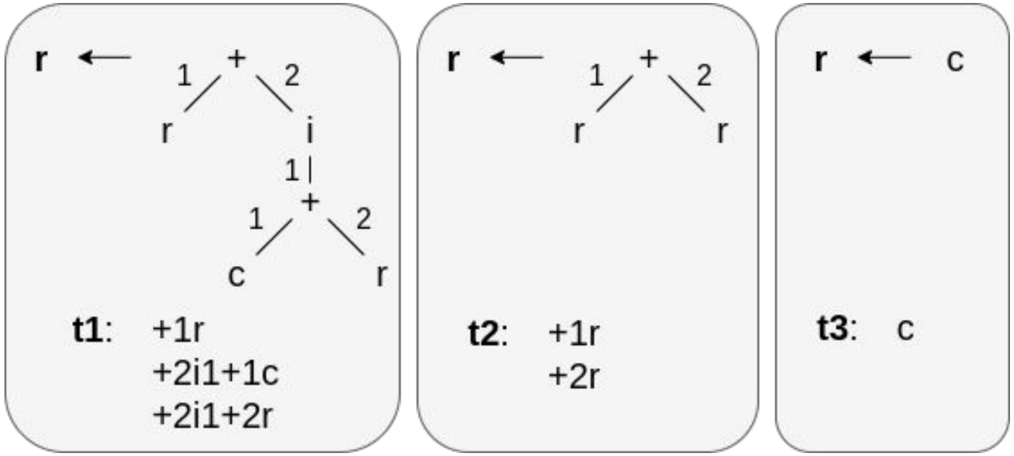
- Each dialect has its own configuration
- What has to be matched might change
- Dialect-wise configuration

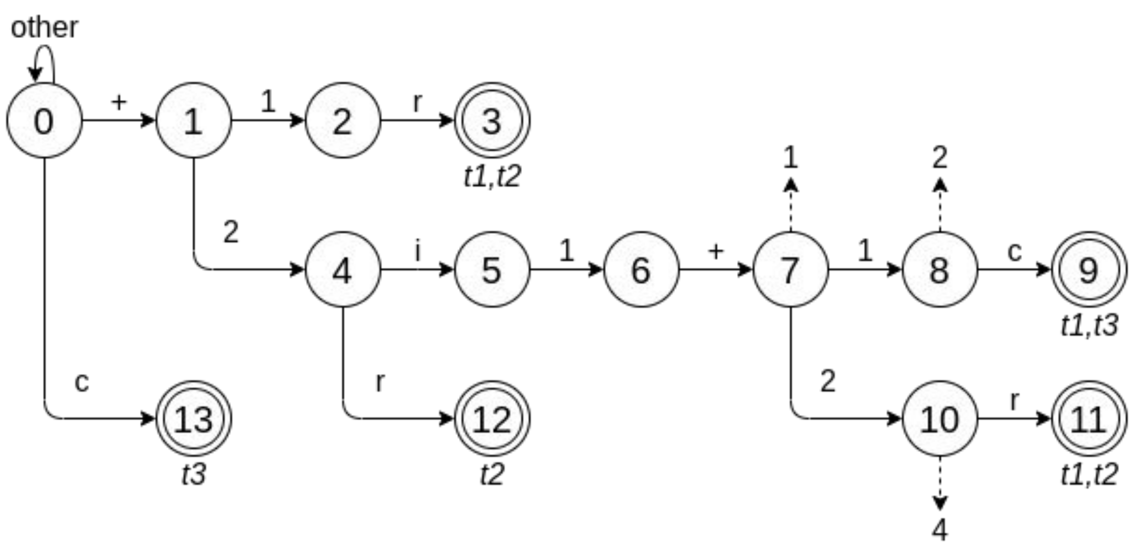
```
fir:
  cmpf:
    must-match-attr: predicate
  if:
    must-match-attr: predicate
```

Algorithm - DDG Automaton

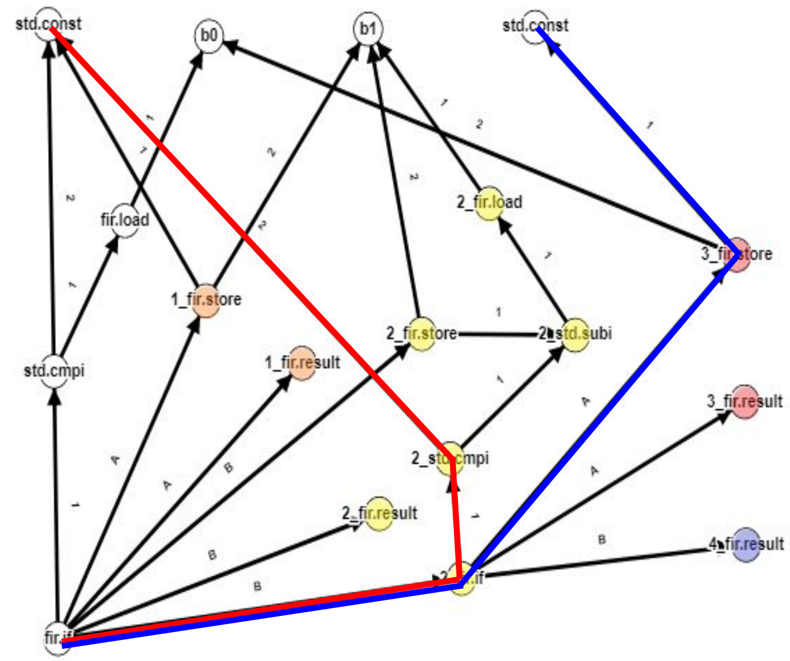
- ▷ Two rooted DAGs: input and pattern
- ▷ How to match rooted DAGs?
- ▷ Convert rooted DAGs to set of strings
- ▷ Match set of strings in automaton

Algorithm - TWIG Inspiration





Algorithm - DDG Automaton



- ▷ Paths from root to leafs
- ▷ Convert paths to strings:

[fir.if, B, 2_fir.if, 1, 2_std.cmpi, 2, std.const]

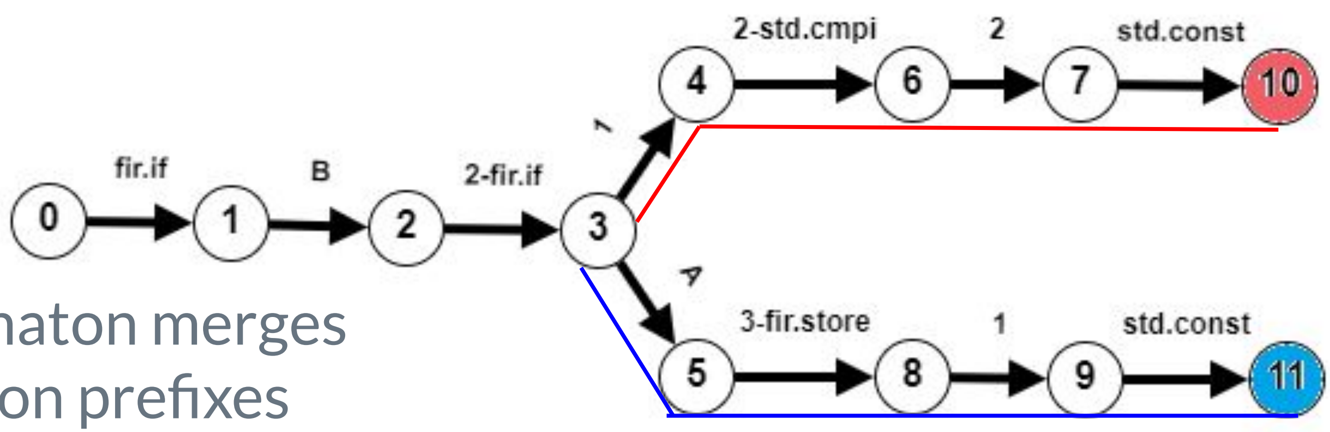
[fir.if, B, 2_fir.if, A, 3_fir.store, 1, std.const]

Algorithm - DDG Automaton

- Each pattern is a set of strings
- Build automaton for all set of strings

10 - [fir.if, B, 2-fir.if, 1, 2-std.cmpi, 2, std.const]

11 - [fir.if, B, 2-fir.if, A, 3-fir.store, 1, std.const]

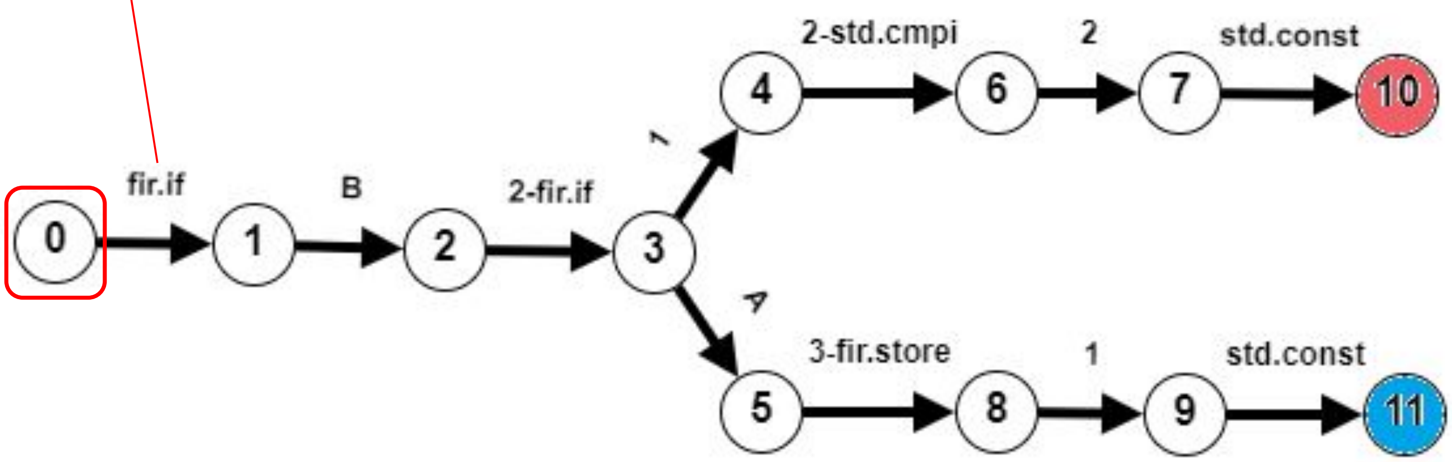


- Automaton merges common prefixes

Algorithm - DDG Automaton

- Feed input code strings to automaton

[fir.if, B, 2-fir.if, A, 3-fir.store, 1, fir.load, ...]



Algorithm - Recap

- Process and compile input and PAT
- Filter input with CDG matching
- Apply DDG matching on filtered input
- **DDG matched?** Apply rewrite

Results

1. Introduction
2. SMR
3. Algorithm
4. **Results**

Results

- Is SMR:
  - Capable of raising?
  - Simple?
  - Scalable?
  - Flexible?

Methodology - Usability

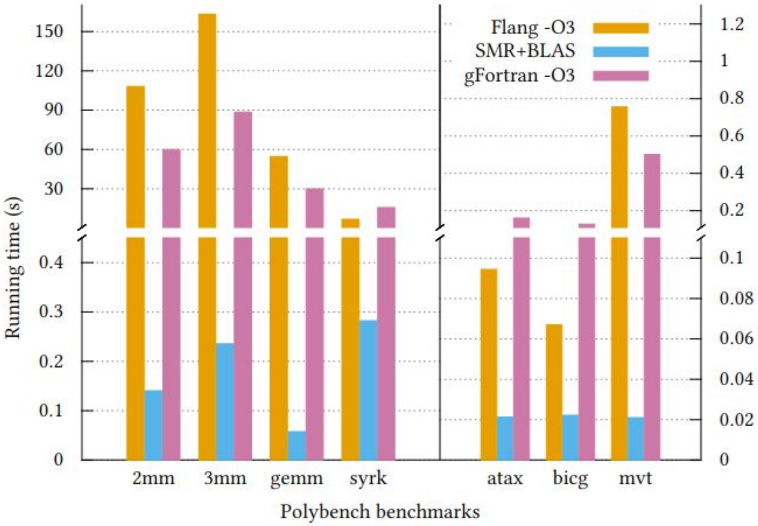
```
f90 {
subroutine p3mm_double(a, b, e, ni, nj, nk)
  double precision, dimension(nj, nk) :: b
  double precision, dimension(nj, ni) :: e
  double precision, dimension(nk, ni) :: a
  integer :: ni, nj, nk

  do i = 1, ni
    do j = 1, nj
      e(j,i) = 0.0
      do k = 1, nk
        e(j,i) = e(j,i) + a(k,i) * b(j,k)
      end do
    end do
  end do
end subroutine
}={
subroutine p3mm_double(a, b, e, ni, nj, nk)
  double precision, dimension(nj, nk) :: b
  double precision, dimension(nj, ni) :: e
  double precision, dimension(nk, ni) :: a
  integer :: ni, nj, nk

  external :: dgemm

  call dgemm('N', 'N', nj, ni, nk, 1.0D0,
             b, nk, a, nj, 0.0D0, e, nj)
end subroutine
}
```

PAT for Polybench's 3mm kernel

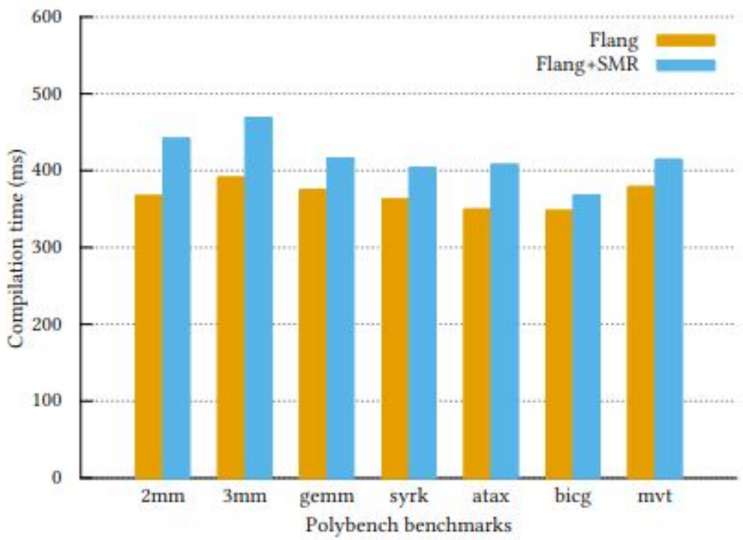
```
f90 {
subroutine atax_double(a, x, y, tmp, nx, ny)
  double precision, dimension(ny, nx) :: a
  double precision, dimension(ny) :: x
  double precision, dimension(ny) :: y
  double precision, dimension(nx) :: tmp
  integer :: nx, ny

  do i = 1, nx
    tmp(i) = 0.0D0
    do j = 1, ny
      tmp(i) = tmp(i) + (a(j, i) * x(j))
    end do
    do j = 1, ny
      y(j) = y(j) + a(j, i) * tmp(i)
    end do
  end do
end subroutine
}={
subroutine atax_double(a, x, y, tmp, nx, ny)
  double precision, dimension(ny, nx) :: a
  double precision, dimension(ny) :: x
  double precision, dimension(ny) :: y
  double precision, dimension(nx) :: tmp
  integer :: nx, ny

  external :: dgemv

  call dgemv('T', nx, ny, 1.0D0, a,
             ny, x, 1, 0.0D0, tmp, 1)
  call dgemv('N', ny, nx, 1.0D0, a,
             ny, tmp, 1, 0.0D0, y, 1)
end subroutine
}
```

PAT for Polybench's atax kernel

Results - Usability



Polybench running time after blas replacement



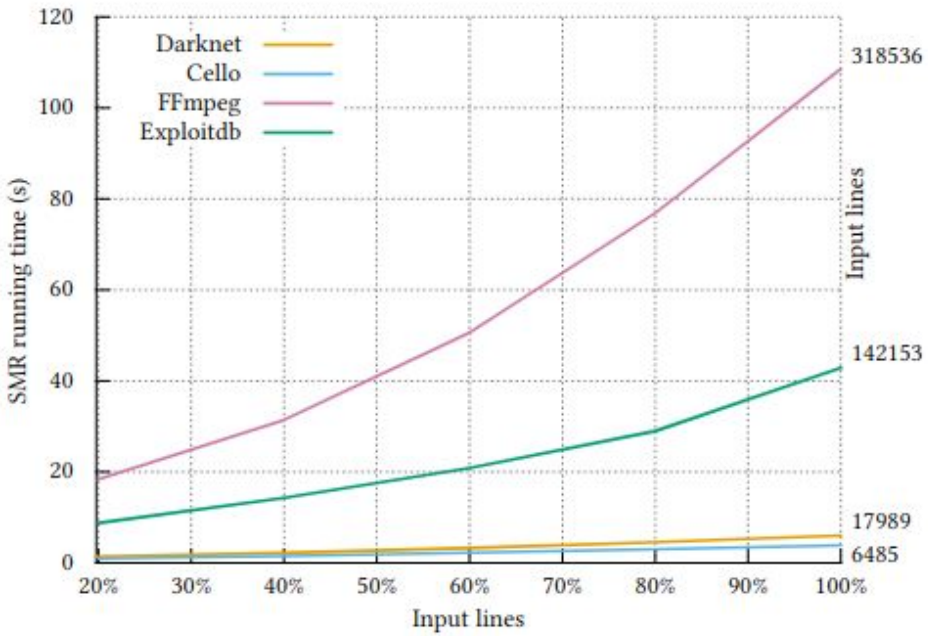
FIR compilation time with/without SMR+BLAS

Results - Dialects Flexibility

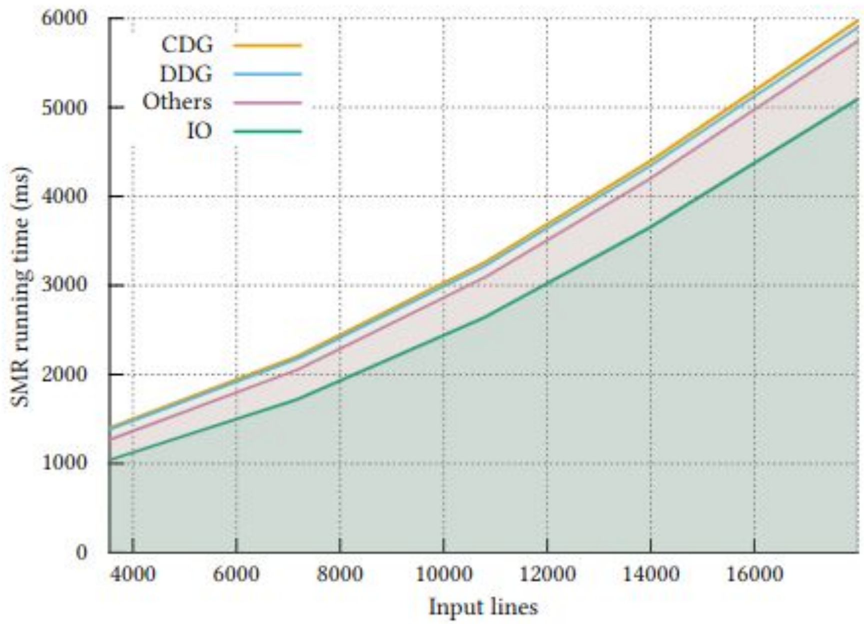
Idiom	Darknet [40]	Cello [26]	Exploitdb [46]	Ffmpeg [16]	Hpgmg [1]	Nekrs [17]	**Total**
saxpy	1						1
scopy	1						1
sdot	1			1			2
sgemm	4						4
scall	2						2
ddot		1			1	2	4
dgemm			1			3	4
dgemmv						1	1
dscal						3	3
Total	9	1	1	1	1	9	22

Matching with CIL and CBLAS idioms

Results - Input Scalability

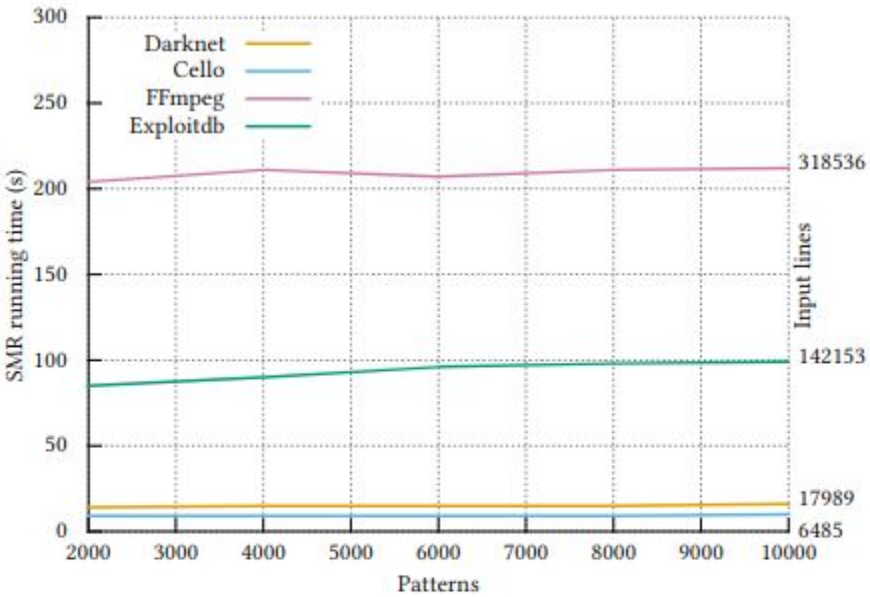


4 input programs against 95 patterns

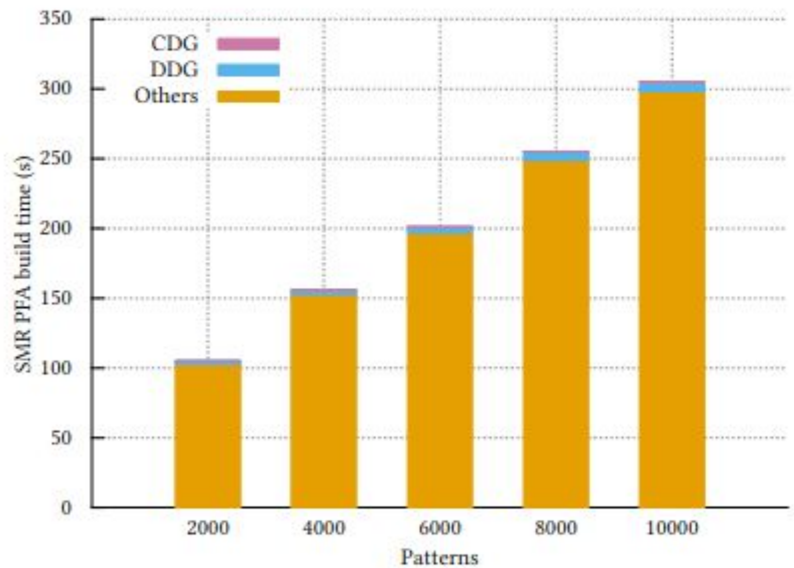


Darknet breakout

Results - Pattern Scalability

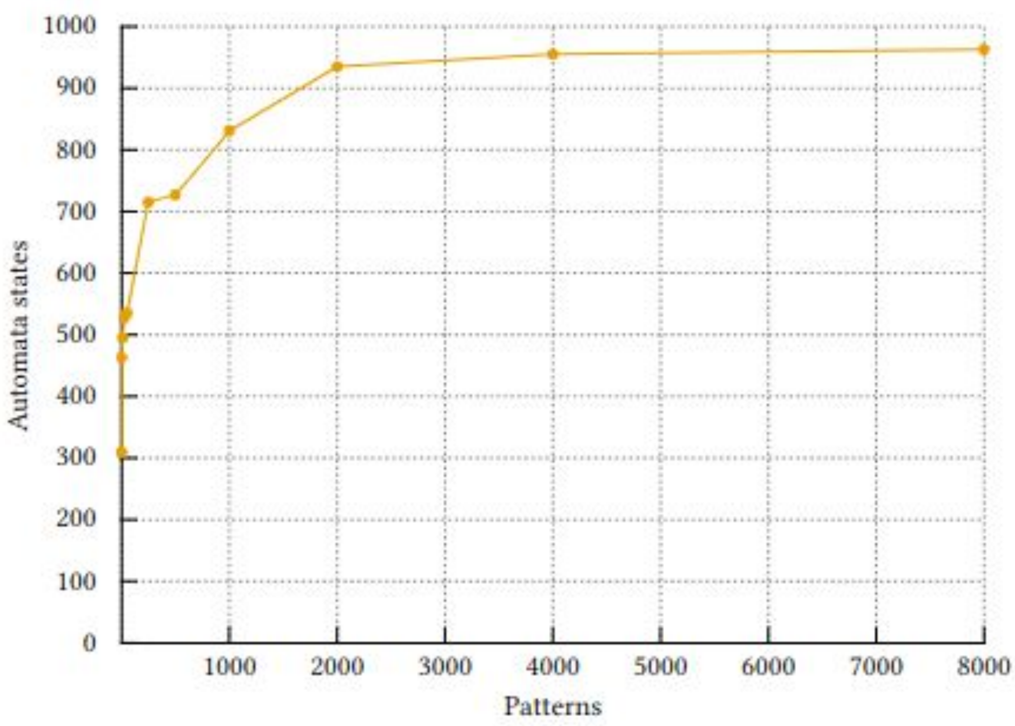


SMR running time vs number of patterns



PFA build time

Results - Pattern Scalability



SMR's automaton prefix merging

Results - Limitations

- Restrictions on Patterns
- Sensibility to front ends and dialects
- Limited pattern generality

Thank you!

- Paper: https://dl.acm.org/doi/full/10.1145/3571283
- Repo: https://gitlab.com/parlab/smr

UNICAMP