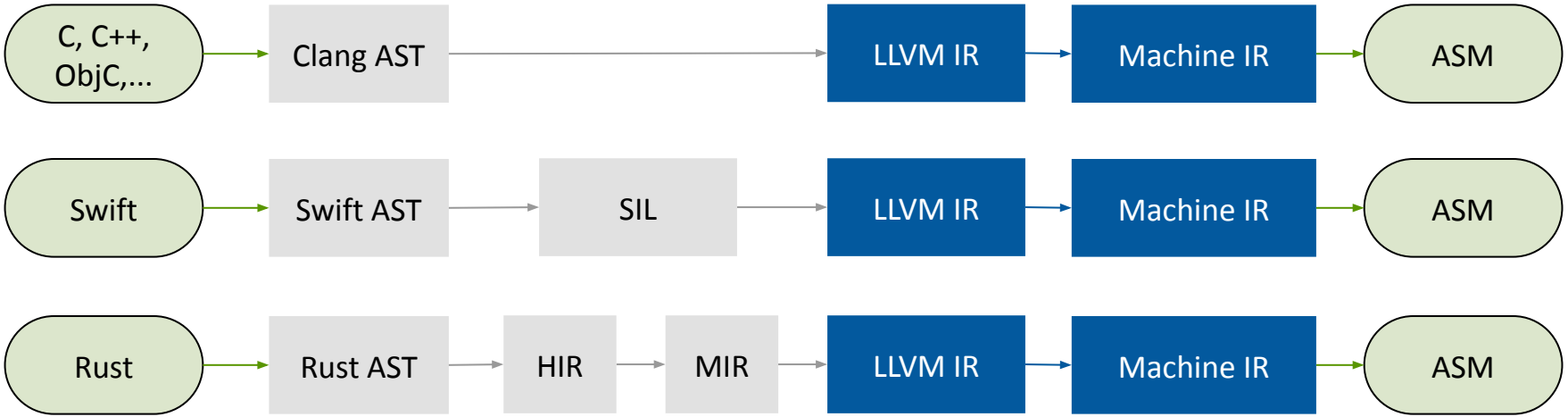
**Berke Ates,** Prof. Dr. Torsten Hoefler, Dr. Tal Ben-Nun, Dr. Alexandru Calotoiu

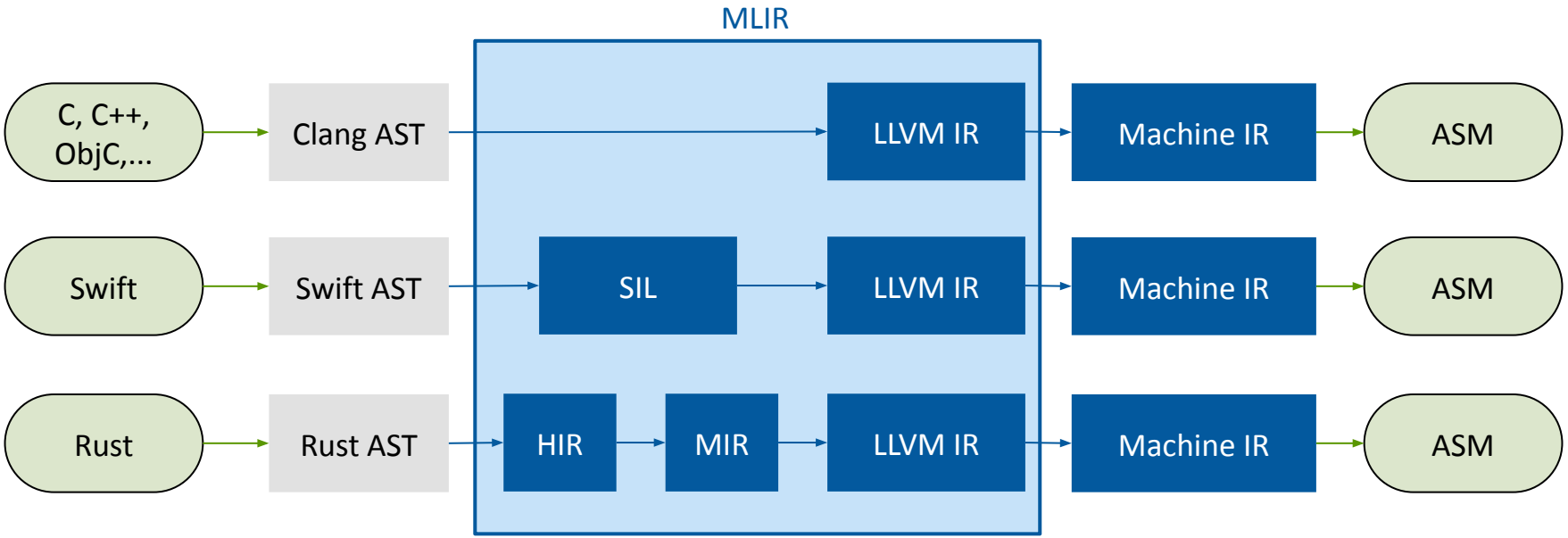
# MLIR-SDFG: A Data-Centric Dialect for MLIR

# Motivation

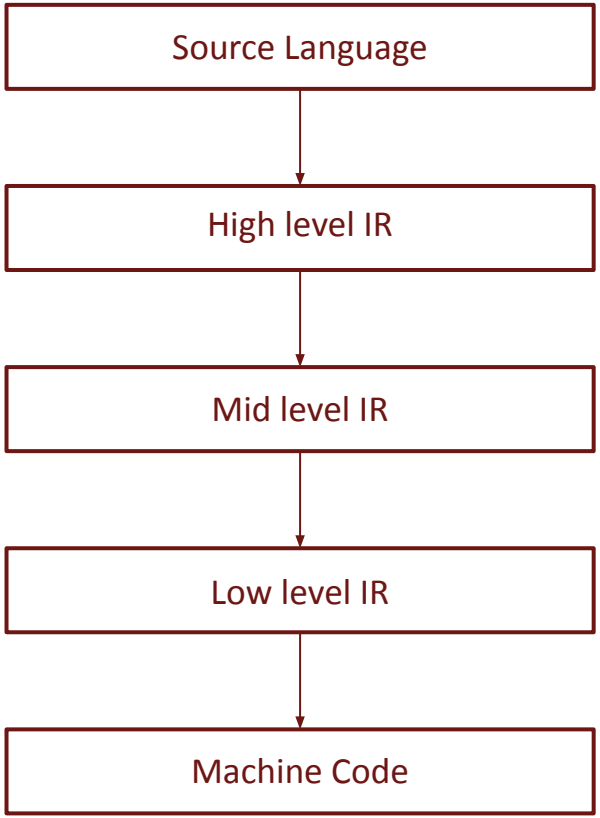
# Motivation


C, C++, ObjC,...
Clang AST
LLVM IR
Machine IR
ASM
Swift
Swift AST
SIL
LLVM IR
Machine IR
ASM
Rust
Rust AST
HIR
MIR
LLVM IR
Machine IR
ASM

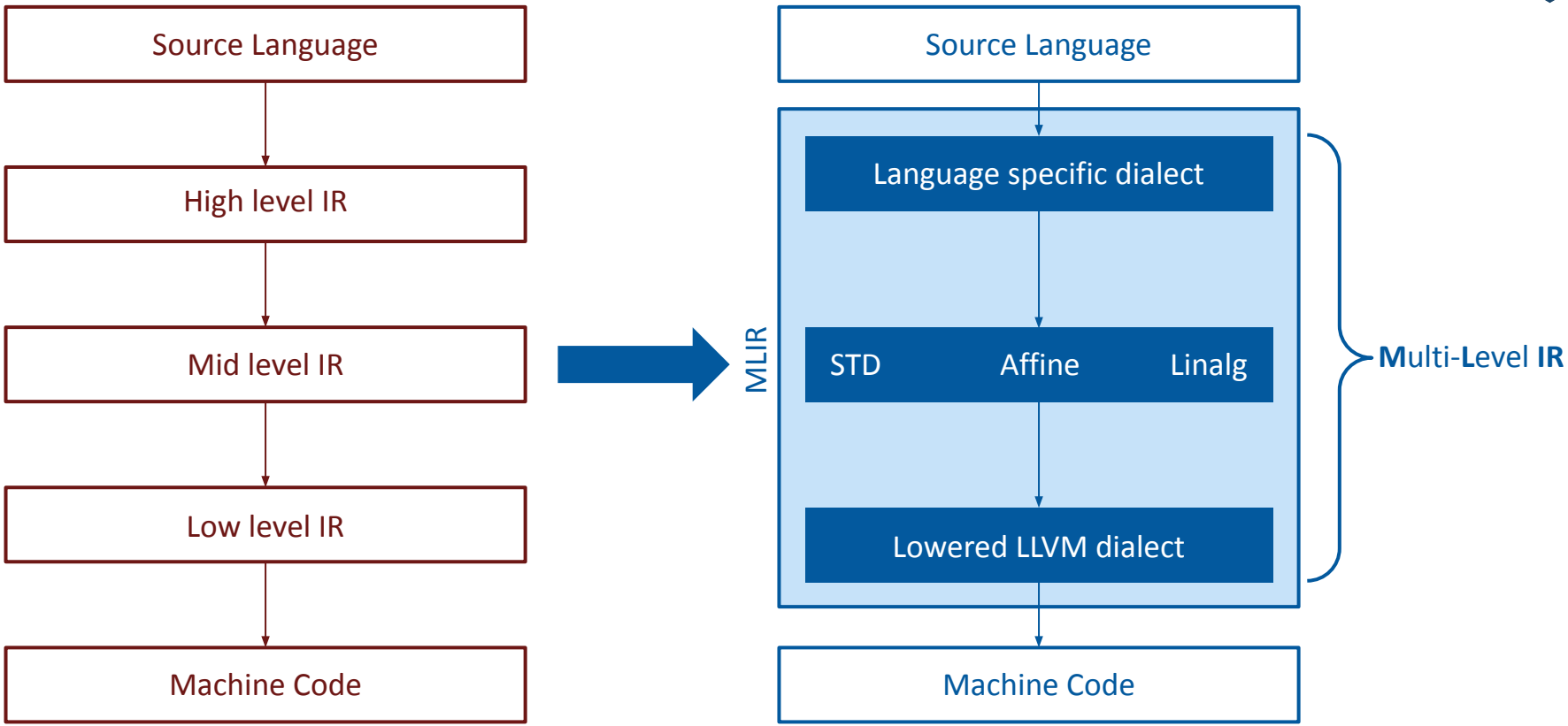
# Motivation


MLIR
C, C++, ObjC,...
Clang AST
LLVM IR
Machine IR
ASM
Swift
Swift AST
SIL
LLVM IR
Machine IR
ASM
Rust
Rust AST
HIR
MIR
LLVM IR
Machine IR
ASM

# Compiling with MLIR



# Compiling with MLIR



# Dialects: Logical groups

- Define a namespace
- List of custom types
- List of operations
- Custom parser, printer and verifier
- List of passes: Analysis, Transformation, Dialect conversion

# Dialects can coexist

```
affine.for %j = 0 to %nk {

    %ij = arith.muli %i, %j : index
    %ij_1 = arith.addi %ij, %1 : index
    %ij_1_64 = arith.index_cast %ij_1 : index to i64
    %ni_64 = arith.index_cast %ni : index to i64

    %rem = llvm.urem %ij_1_64, %ni_64 : i64
    %rem_f = llvm.bitcast %rem : i64 to f64
    %ni_64_f = llvm.bitcast %ni_64 : i64 to f64
    %entry = llvm.fdiv %rem_f, %ni_64_f : f64

    memref.store %entry, %A[%i, %j] : memref<?x?xf64>

}
```

# Dialects can coexist

Affine

```
affine.for %j = 0 to %nk {

    %ij = arith.muli %i, %j : index
    %ij_1 = arith.addi %ij, %1 : index
    %ij_1_64 = arith.index_cast %ij_1 : index to i64
    %ni_64 = arith.index_cast %ni : index to i64

    %rem = llvm.urem %ij_1_64, %ni_64 : i64
    %rem_f = llvm.bitcast %rem : i64 to f64
    %ni_64_f = llvm.bitcast %ni_64 : i64 to f64
    %entry = llvm.fdiv %rem_f, %ni_64_f : f64

    memref.store %entry, %A[%i, %j] : memref<?x?xf64>
```

Affine

```
}
```

# Dialects can coexist

Affine
```
affine.for %j = 0 to %nk {
```

Arith
```
    %ij = arith.muli %i, %j : index
    %ij_1 = arith.addi %ij, %1 : index
    %ij_1_64 = arith.index_cast %ij_1 : index to i64
    %ni_64 = arith.index_cast %ni : index to i64

    %rem = llvm.urem %ij_1_64, %ni_64 : i64
    %rem_f = llvm.bitcast %rem : i64 to f64
    %ni_64_f = llvm.bitcast %ni_64 : i64 to f64
    %entry = llvm.fdiv %rem_f, %ni_64_f : f64

    memref.store %entry, %A[%i, %j] : memref<?x?xf64>
```

Affine
```
}
```

# Dialects can coexist

Affine
```
affine.for %j = 0 to %nk {
```

Arith
```
  %ij = arith.muli %i, %j : index
  %ij_1 = arith.addi %ij, %1 : index
  %ij_1_64 = arith.index_cast %ij_1 : index to i64
  %ni_64 = arith.index_cast %ni : index to i64
```

LLVM
```
  %rem = llvm.urem %ij_1_64, %ni_64 : i64
  %rem_f = llvm.bitcast %rem : i64 to f64
  %ni_64_f = llvm.bitcast %ni_64 : i64 to f64
  %entry = llvm.fdiv %rem_f, %ni_64_f : f64
```

```
  memref.store %entry, %A[%i, %j] : memref<?x?xf64>
```

Affine
```
}
```

# Dialects can coexist

- Code reuse
- Dialects can target specific problems
- Dialects can optimize for specific hardware

| | |
|---|---|
| Affine | affine.for %j = 0 to %nk { |
| Arith | %ij = arith.muli %i, %j : index<br>%ij_1 = arith.addi %ij, %1 : index<br>%ij_1_64 = arith.index_cast %ij_1 : index to i64<br>%ni_64 = arith.index_cast %ni : index to i64 |
| LLVM | %rem = llvm.urem %ij_1_64, %ni_64 : i64<br>%rem_f = llvm.bitcast %rem : i64 to f64<br>%ni_64_f = llvm.bitcast %ni_64 : i64 to f64<br>%entry = llvm.fdiv %rem_f, %ni_64_f : f64 |
| Memref | memref.store %entry, %A[%i, %j] : memref<?x?xf64> |
| Affine | } |

# Operations, Not Instructions

- No fixed instruction set
- Operations are “opaque functions”

```
%res = mydialect.opname {someAttribute = true} (%arg1) : i32
```

# Operations, Not Instructions

- No fixed instruction set
- Operations are “opaque functions”

```
%res = mydialect.opname {someAttribute = true} (%arg1) : i32
```

Name of the result

# Operations, Not Instructions

- No fixed instruction set
- Operations are “opaque functions”

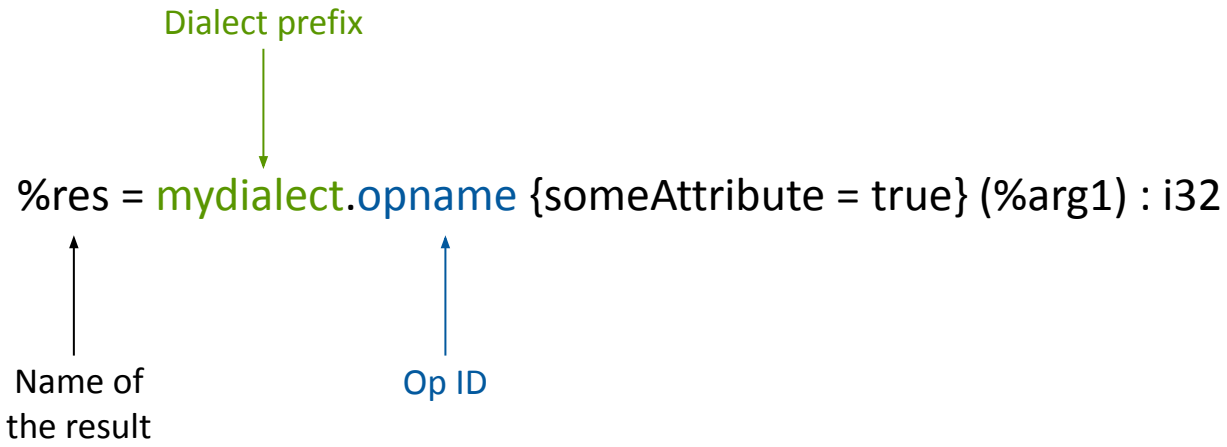
Dialect prefix

```
%res = mydialect.opname {someAttribute = true} (%arg1) : i32
```

Name of the result

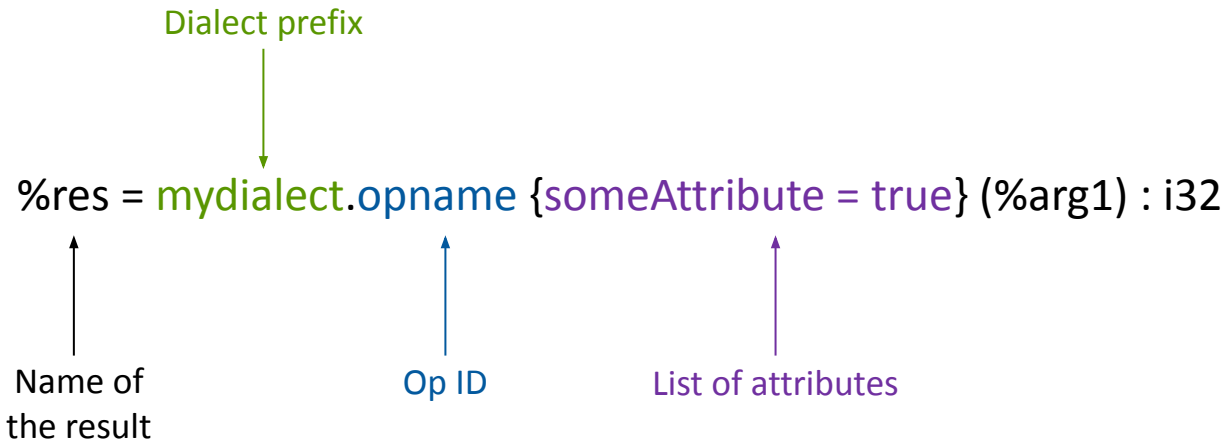
# Operations, Not Instructions

- No fixed instruction set
- Operations are “opaque functions”



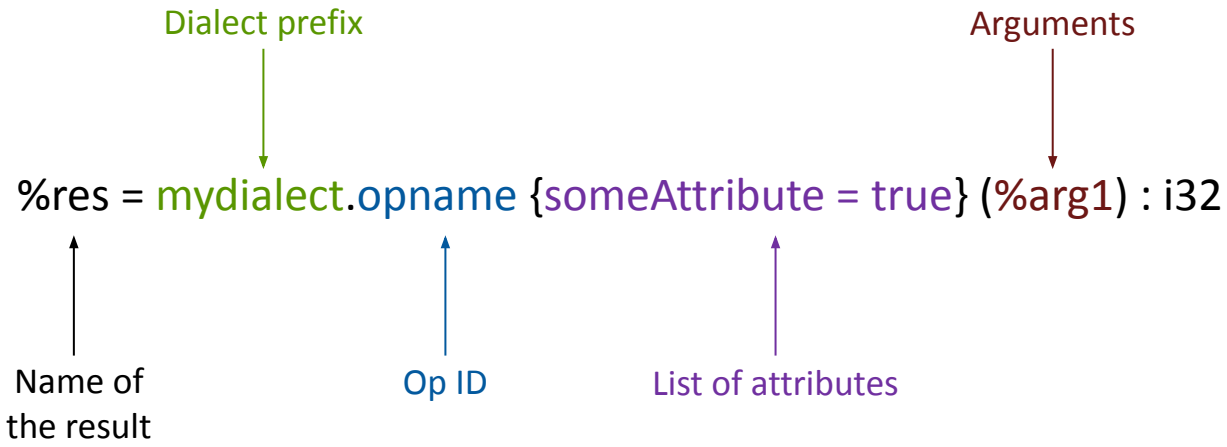
# Operations, Not Instructions

- No fixed instruction set
- Operations are “opaque functions”



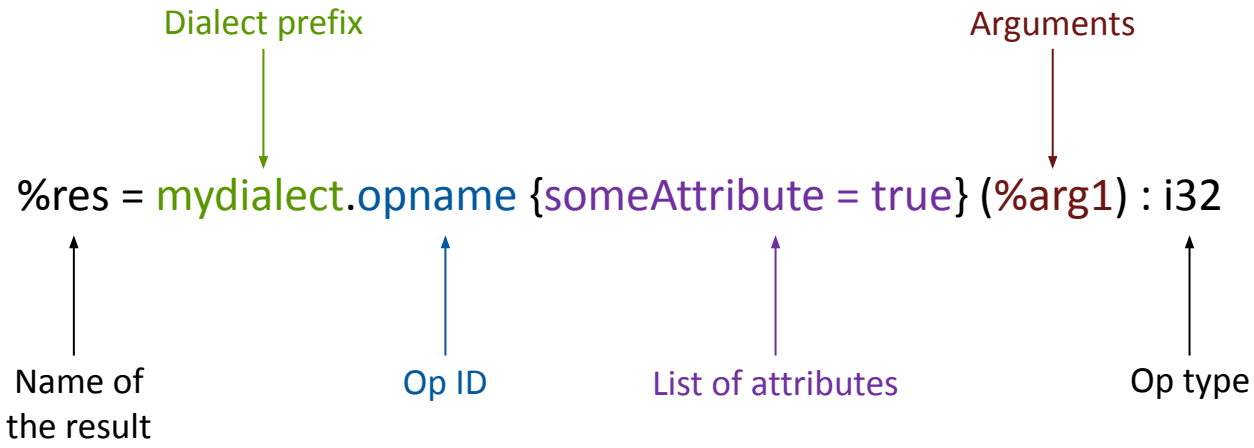
# Operations, Not Instructions

- No fixed instruction set
- Operations are “opaque functions”



# Operations, Not Instructions

- No fixed instruction set
- Operations are “opaque functions”



# Recursive nesting: Operations -> Regions -> Blocks

```
mydialect.whileNZ (%i) ({

        ^block1:
            %res = mydialect.add %a, %b : i32
            br ^block2

        ^block2:
          // Some more code

})
```

# Recursive nesting: Operations -> Regions -> Blocks

Op

```
mydialect.whileNZ (%i) ({

        ^block1:
            %res = mydialect.add %a, %b : i32
            br ^block2

        ^block2:
            // Some more code

})
```

# Recursive nesting: Operations -> Regions -> Blocks

- Operations may contain regions

```
Op mydialect.whileNZ (%i) ({
   Region
         ^block1:
               %res = mydialect.add %a, %b : i32
               br ^block2

         ^block2:
              // Some more code
})
```

# Recursive nesting: Operations -> Regions -> Blocks

- Operations may contain regions
- Regions contain list of blocks

```
Op mydialect.whileNZ (%i) ({
    Region
        Block   ^block1:
                    %res = mydialect.add %a, %b : i32
                    br ^block2

        Block   ^block2:
                    // Some more code
})
```

# Recursive nesting: Operations -> Regions -> Blocks

- Operations may contain regions
- Regions contain list of blocks
- Blocks contain list of operations

```
Op mydialect.whileNZ (%i) ({
    Region
        Block  ^block1:
                 Op %res = mydialect.add %a, %b : i32
                 Op br ^block2

        Block  ^block2:
                 // Some more code
})
```

# Recursive nesting: Operations -> Regions -> Blocks

- Operations may contain regions
- Regions contain list of blocks
- Blocks contain list of operations
- Allows modelling hierarchical structures

```
Op mydialect.whileNZ (%i) ({
    Region
        Block   ^block1:
                  Op %res = mydialect.add %a, %b : i32
                  Op br ^block2

        Block   ^block2:
                  // Some more code
})
```

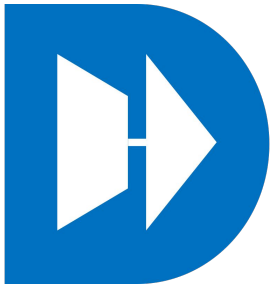
# Recursive nesting: Operations -> Regions -> Blocks

- Operations may contain regions
- Regions contain list of blocks
- Blocks contain list of operations
- Allows modelling hierarch
  structures

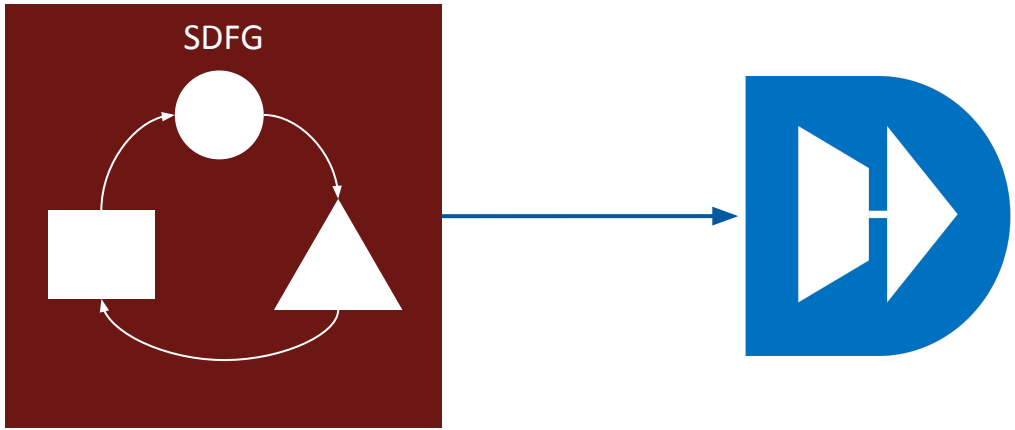
Op mydialect.whileNZ (%i) ({
  Region
    ^block1:
    Block
      Op %res = mydialect.add %a, %b : i32
      Op br ^block2
    2:
    ome more code

Everything is an operation

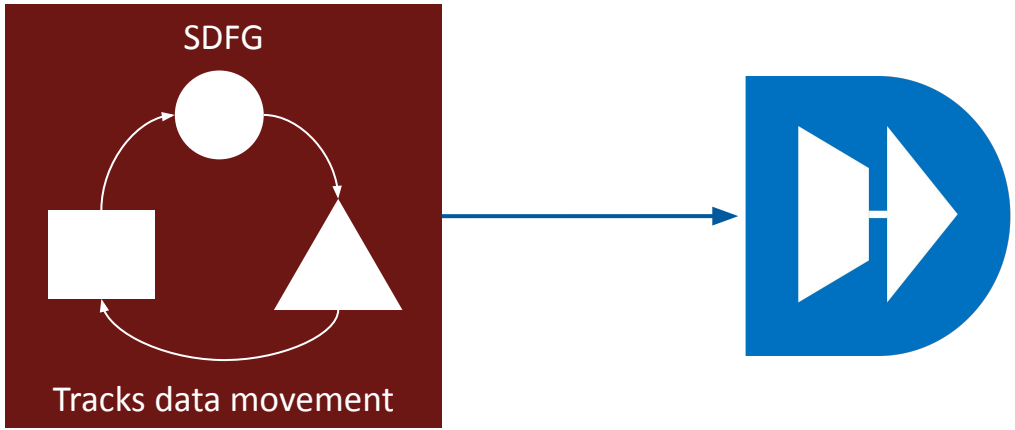
# DaCe: Data-Centric



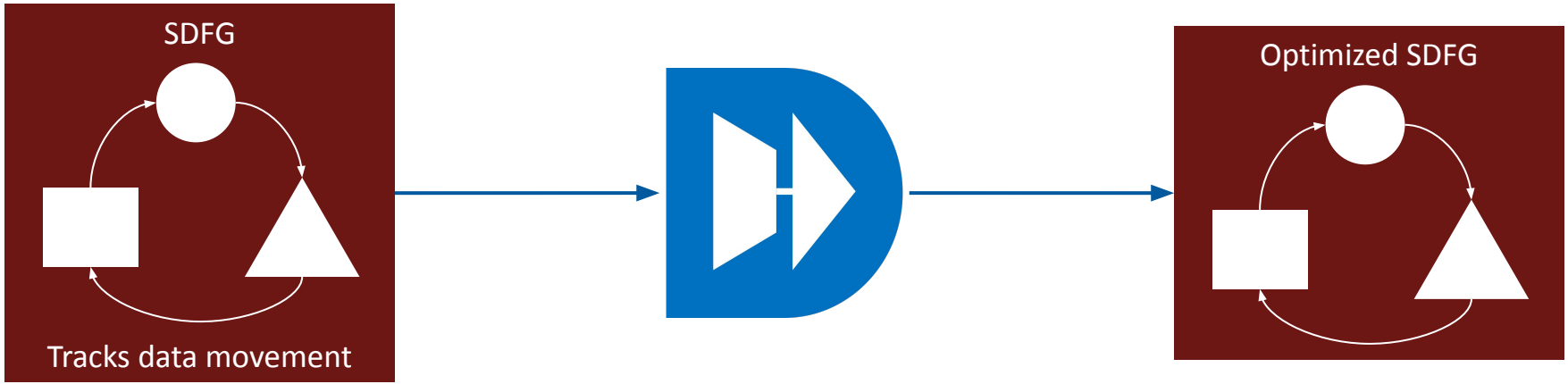
# DaCe: Data-Centric



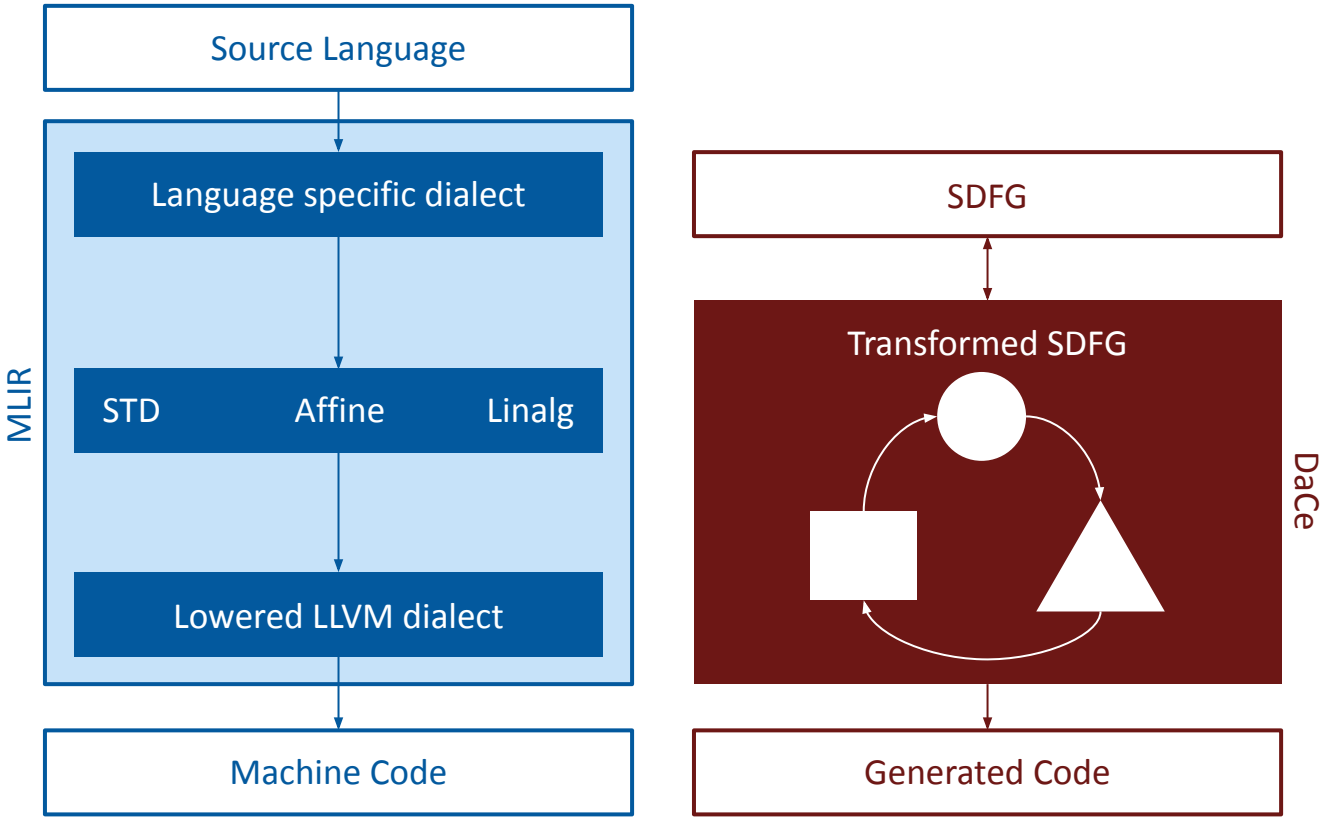
# DaCe: Data-Centric



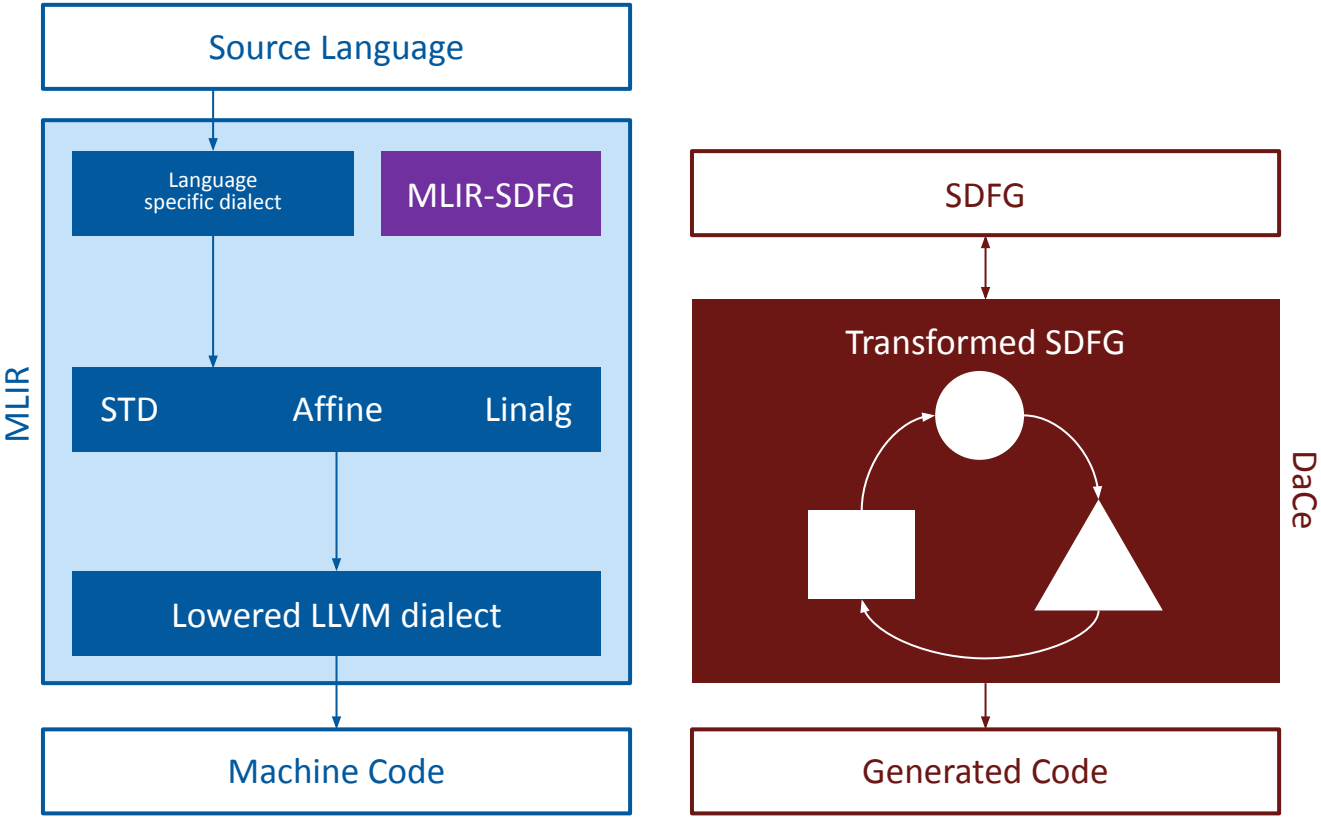
# DaCe: Data-Centric



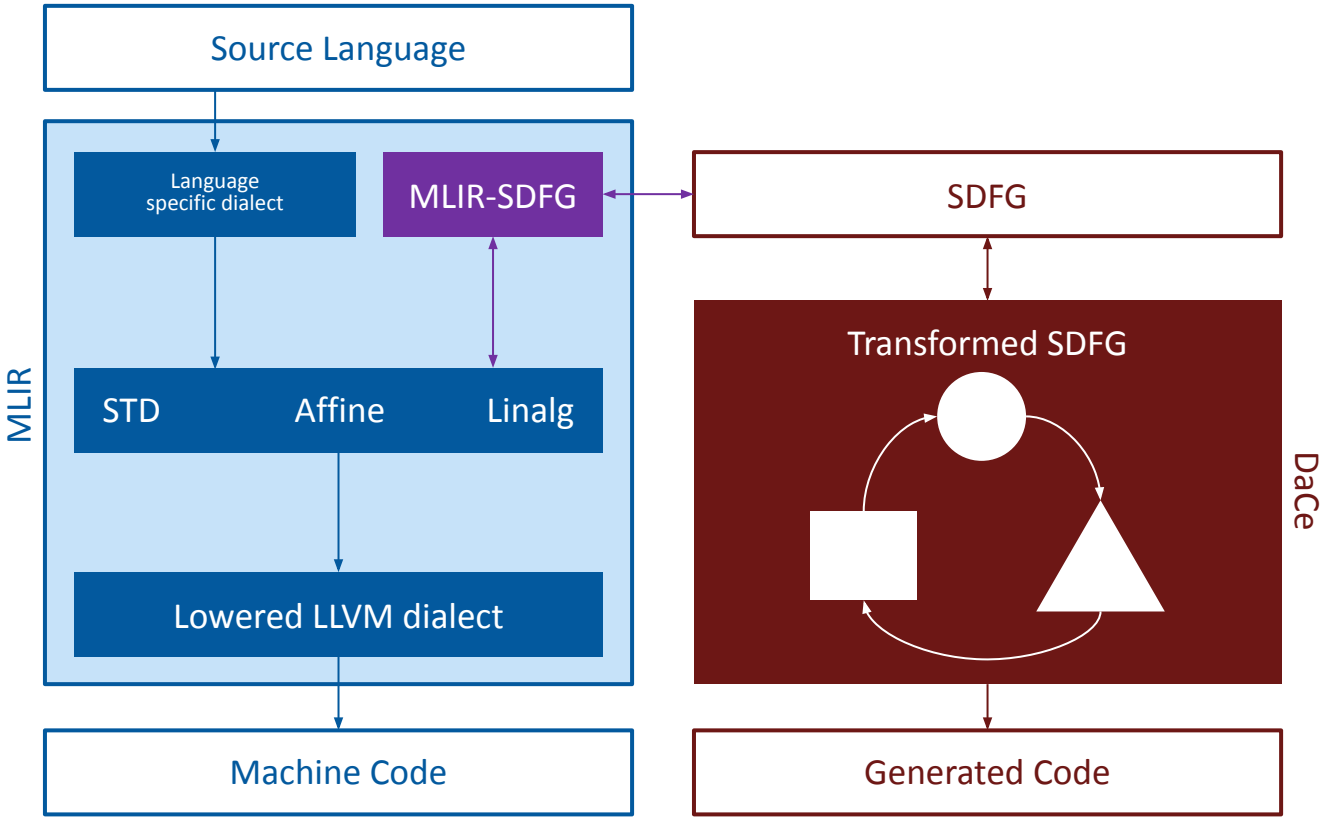
# MLIR-SDFG: Between MLIR and DaCe



# MLIR-SDFG: Between MLIR and DaCe

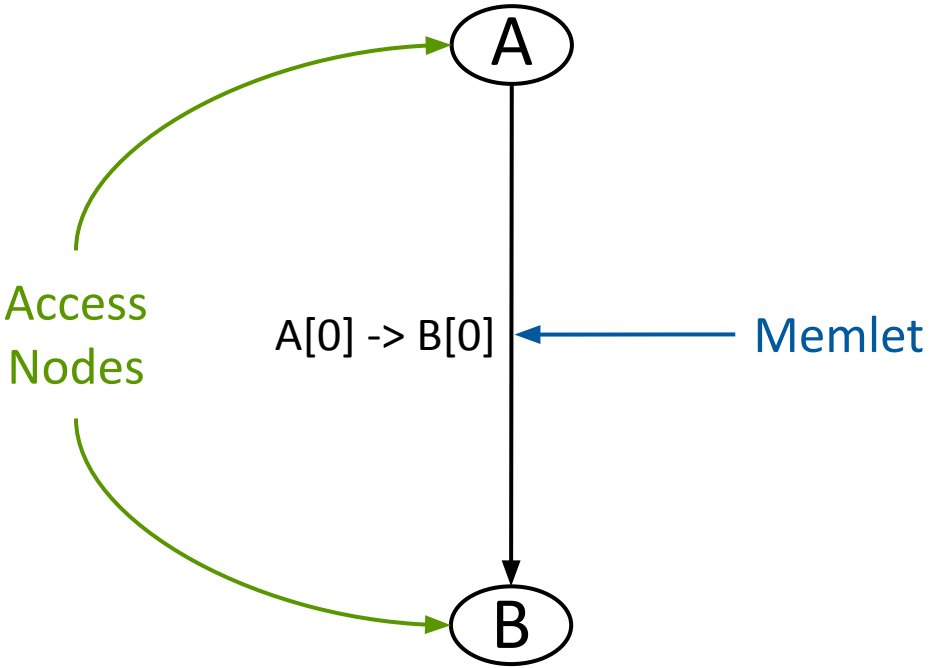


# MLIR-SDFG: Between MLIR and DaCe



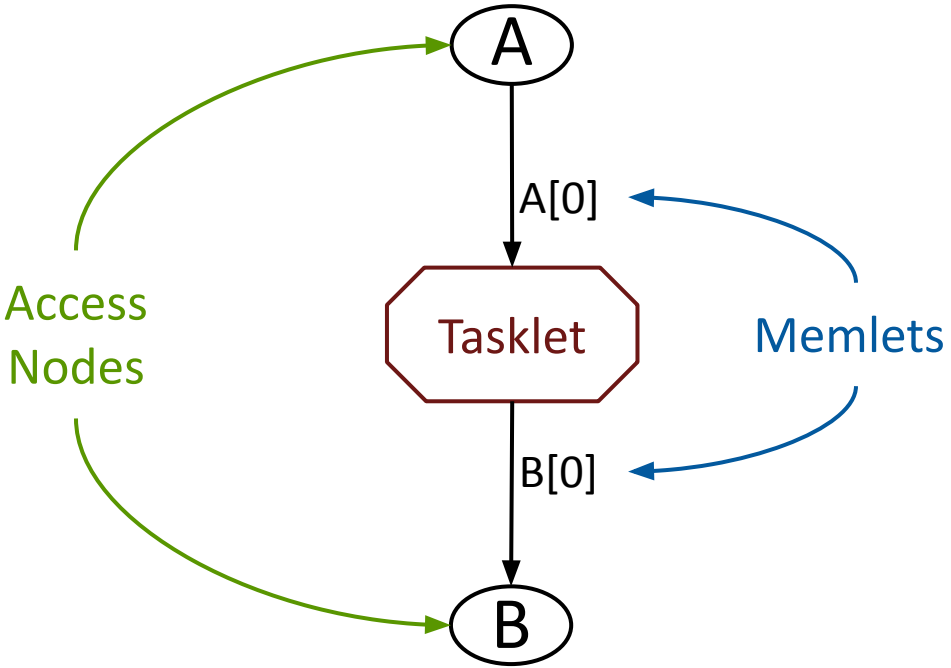
# SDFG: Stateful DataFlow multiGraph


```
%A = sdfg.alloc() : !sdfg.array<2xi32>
%B = sdfg.alloc() : !sdfg.array<2xi32>

%a = sdfg.load %A[0]

sdfg.store %c, %B[0]
```

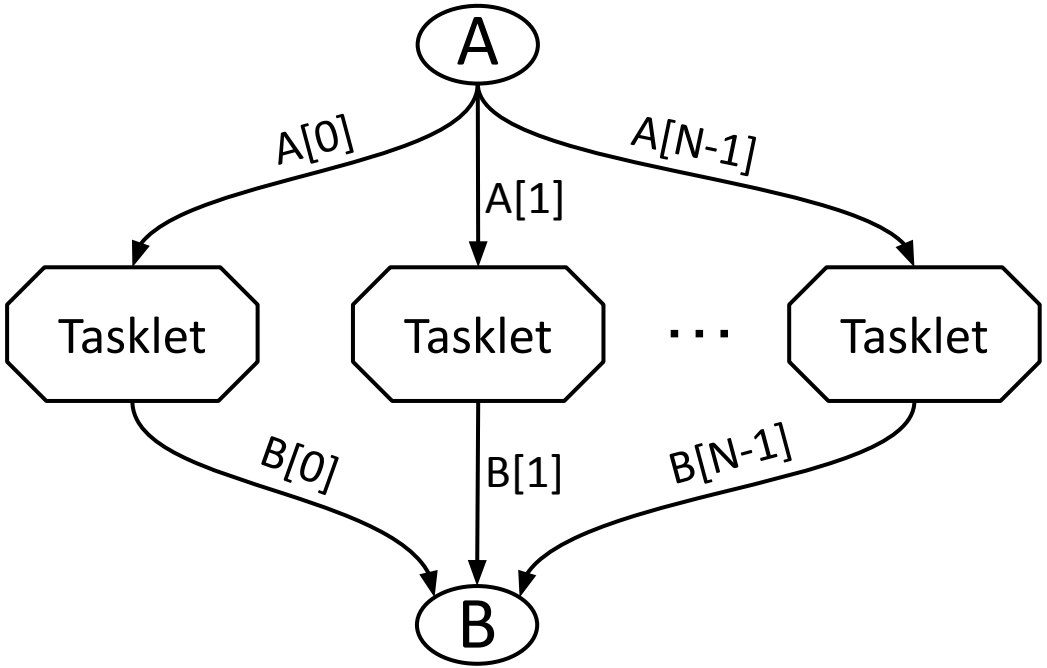


# SDFG: Stateful DataFlow multiGraph

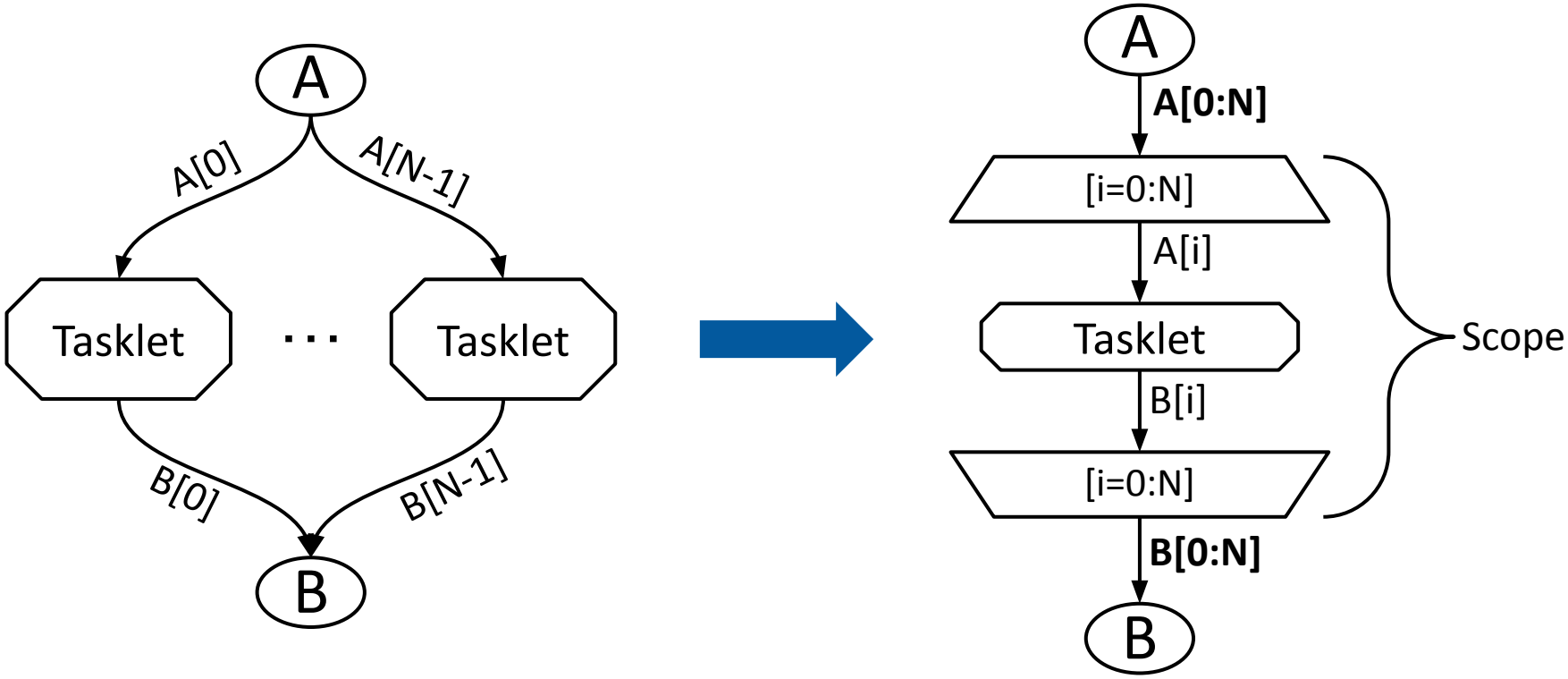

```
%A = sdfg.alloc() : !sdfg.array<2xi32>
%B = sdfg.alloc() : !sdfg.array<2xi32>

%a = sdfg.load %A[0]

%c = sdfg.tasklet @name(%a: i32) -> i32 {
  %r = // Some computation
  sdfg.return %r
}

sdfg.store %c, %B[0]
```



# SDFG: Stateful DataFlow multiGraph

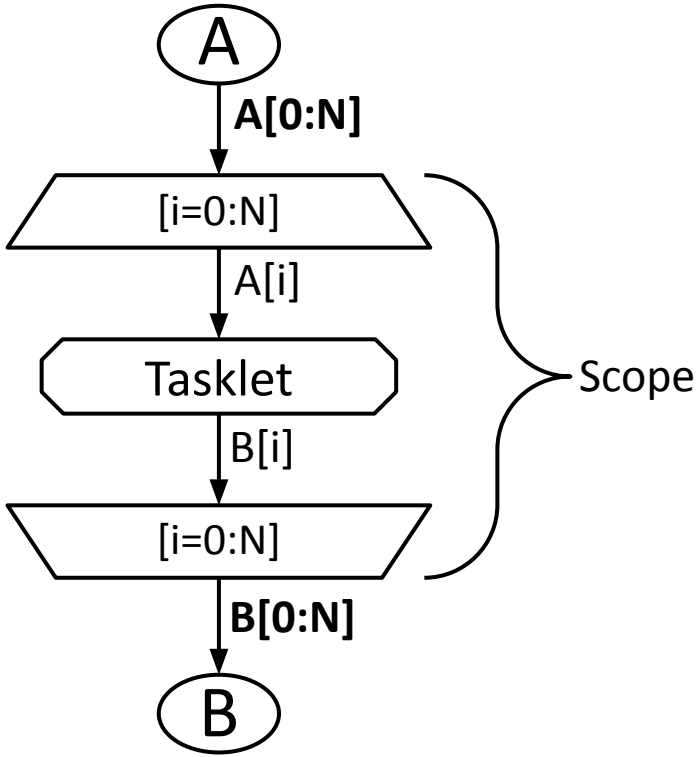



# SDFG: Stateful DataFlow multiGraph

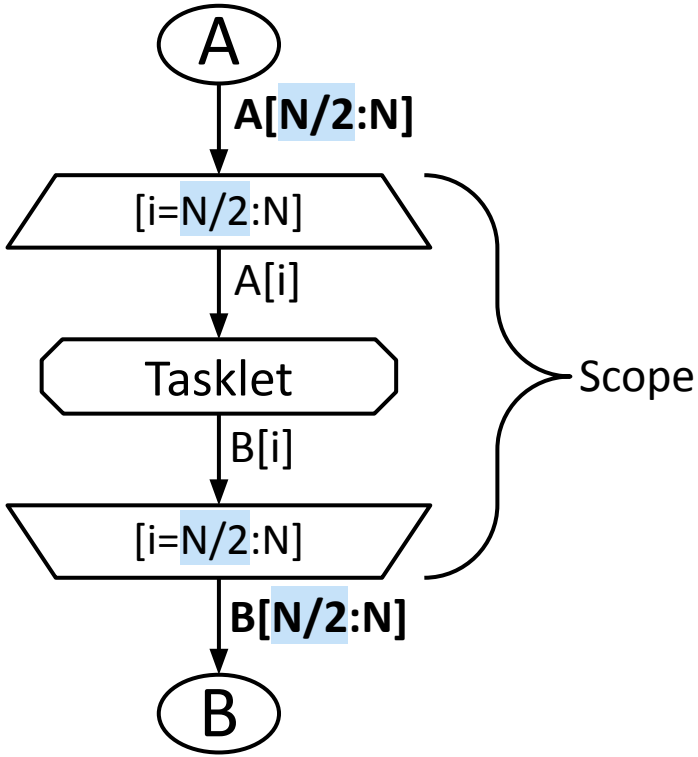
# SDFG: Stateful DataFlow multiGraph


```
%A = sdfg.alloc() : !sdfg.array<sym("N")xi32>
%B = sdfg.alloc() : !sdfg.array<sym("N")xi32>

sdfg.map (%i) = (0) to (sym("N")) step (1) {
  %a = sdfg.load %A[%i]

  %c = sdfg.tasklet @name(%a: i32) -> i32 {
    %r = // Some computation
    sdfg.return %r
  }

  sdfg.store %c, %B[%i]
}
```

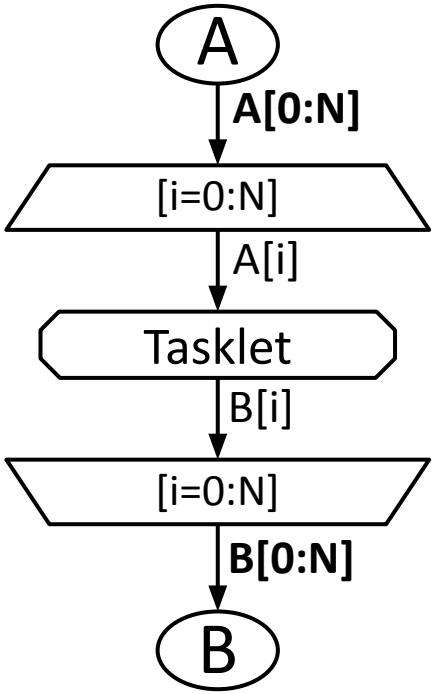


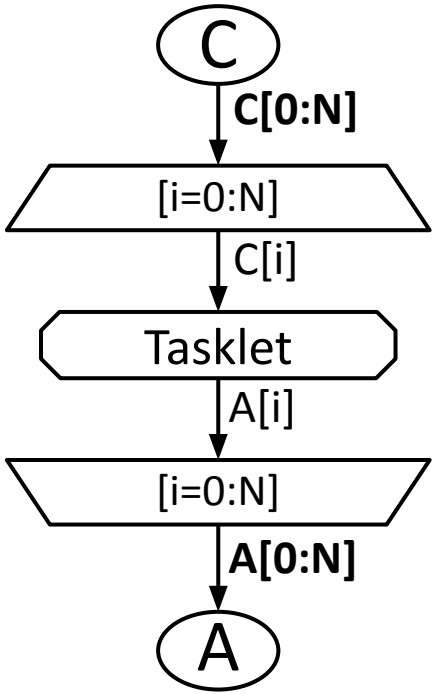
# SDFG: Stateful DataFlow multiGraph


```
%A = sdfg.alloc() : !sdfg.array<sym("N")xi32>
%B = sdfg.alloc() : !sdfg.array<sym("N")xi32>

sdfg.map (%i) = (sym("N/2")) to (sym("N")) step (1) {
  %a = sdfg.load %A[%i]

  %c = sdfg.tasklet @name(%a: i32) -> i32 {
    %r = // Some computation
    sdfg.return %r
  }

  sdfg.store %c, %B[%i]
}
```

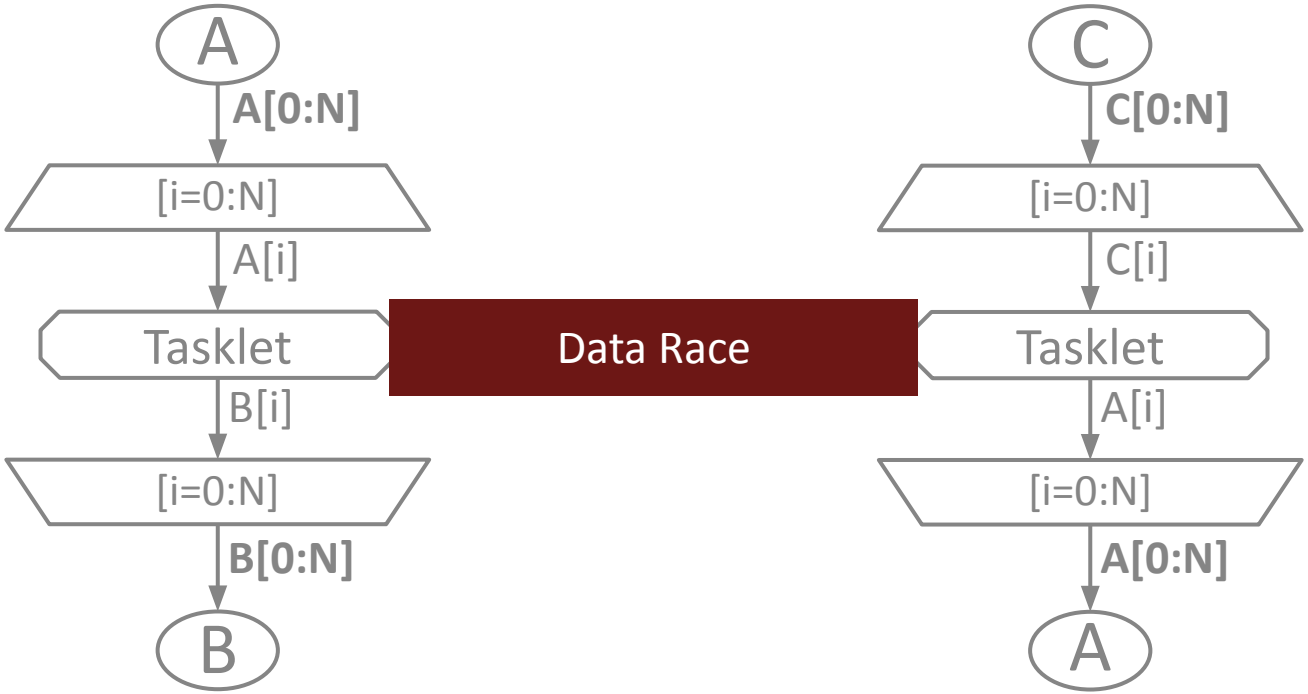


# SDFG: Stateful DataFlow multiGraph

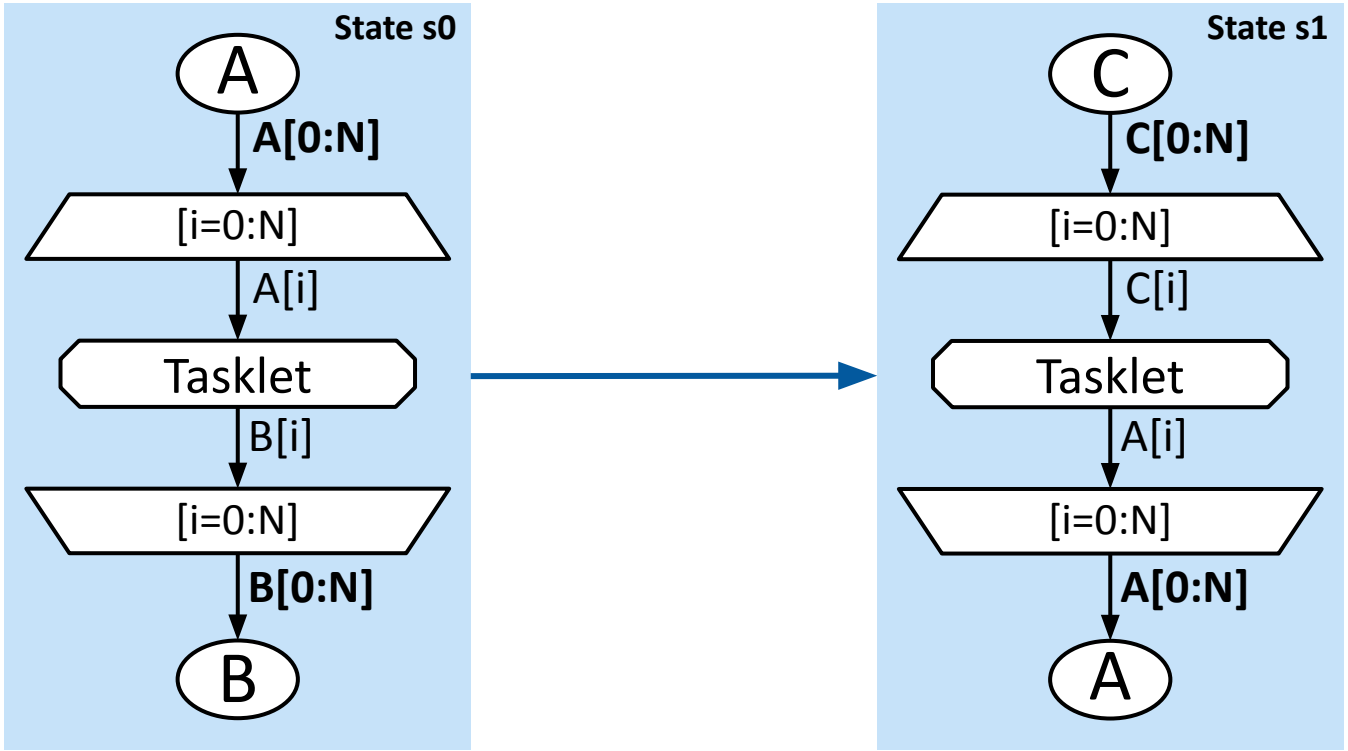



# SDFG: Stateful DataFlow multiGraph

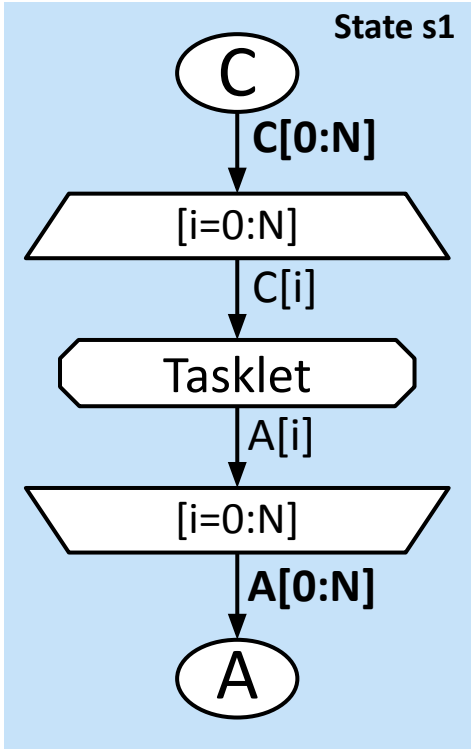



# SDFG: Stateful DataFlow multiGraph

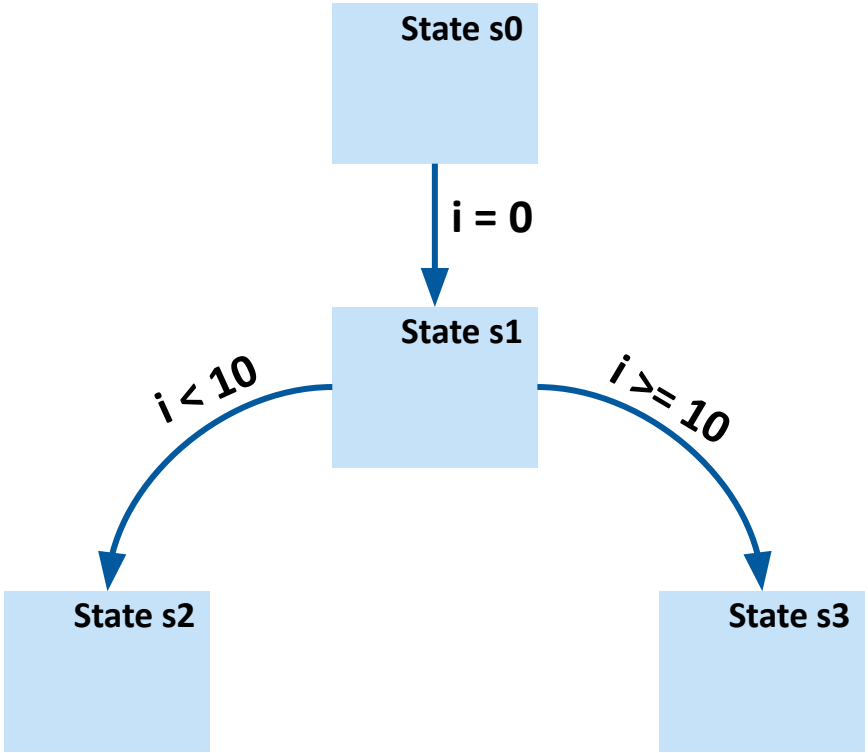
# SDFG: Stateful DataFlow multiGraph


```
%C = sdfg.alloc() : !sdfg.array<sym("N")xi32>
%A = sdfg.alloc() : !sdfg.array<sym("N")xi32>

sdfg.state @s1 {
  sdfg.map (%i) = (0) to (sym("N")) step (1) {
    %c = sdfg.load %C[%i]

    %res = sdfg.tasklet @name(%c: i32) -> i32 {
      %r = // Some computation
      sdfg.return %r
    }

    sdfg.store %res, %A[%i]
  }
}
```



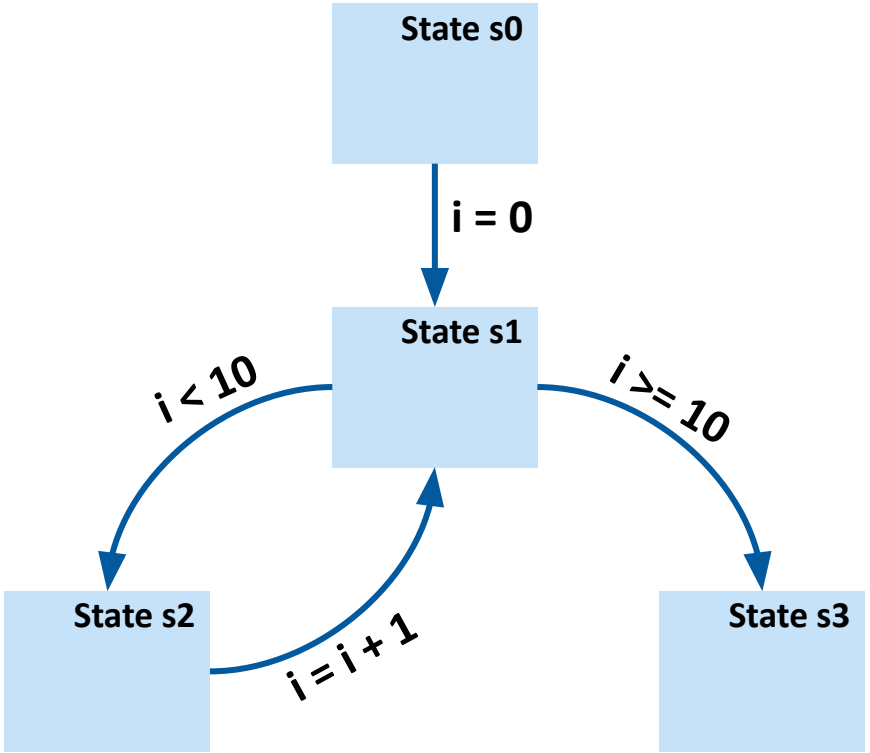
# SDFG: Stateful DataFlow multiGraph


```
sdfg.sdfg{entry=@s0} {
  sdfg.state @s0 {...}
  sdfg.state @s1 {...}
  sdfg.state @s2 {...}
  sdfg.state @s3 {...}

  sdfg.edge{assign=["i: 0"]} @s0 -> @s1
  sdfg.edge{condition="i < 10"} @s1 -> @s2
  sdfg.edge{condition="i >= 10"} @s1 -> @s3

}
```

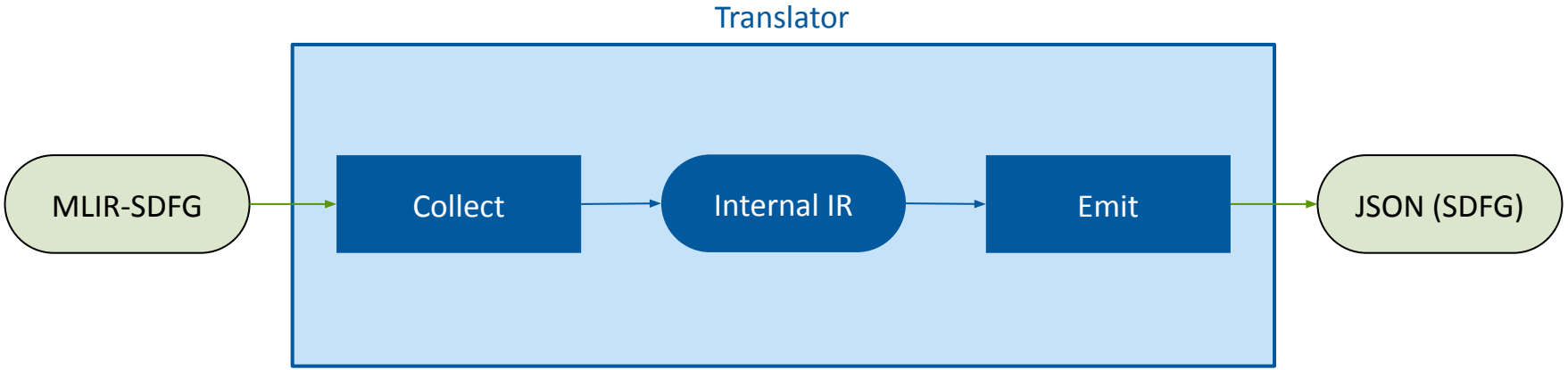


# SDFG: Stateful DataFlow multiGraph


```
sdfg.sdfg{entry=@s0} {
  sdfg.state @s0 {...}
  sdfg.state @s1 {...}
  sdfg.state @s2 {...}
  sdfg.state @s3 {...}

  sdfg.edge{assign=["i: 0"]} @s0 -> @s1
  sdfg.edge{condition="i < 10"} @s1 -> @s2
  sdfg.edge{condition="i >= 10"} @s1 -> @s3
  sdfg.edge{assign=["i: i + 1"]} @s2 -> @s1
}
```



# MLIR-SDFG -> SDFG Translator

# Lifting to Python


```
sdfg.tasklet @add(%a, %b) -> f64 {
        %r = arith.addi %a, %b
        sdfg.return %r
}

sdfg.tasklet @get_0() -> f64 {
        %r = arith.constant 0.0
        sdfg.return %r
}
```

# Lifting to Python


```
sdfg.tasklet @add(%a, %b) -> f64 {
        %r = arith.addi %a, %b
        sdfg.return %r
}
```

```
__out = (a + b)
```

```
sdfg.tasklet @get_0() -> f64 {
        %r = arith.constant 0.0
        sdfg.return %r
}
```

```
__out = dace.float64(0.0)
```

# Optimizing Passes


```
sdfg.sdfg @name(%arg0: i32, %arg1: i32) {
  …
  %c = sdfg.tasklet @add(%arg0, %arg1) -> (i32) {...}
  …
}
```

```
sdfg.sdfg @name(%arg0: i32, %arg1: i32) {
  %0 = sdfg.alloc {name = "_arg0"}() : !sdfg.array<i32>
  %1 = sdfg.alloc {name = "_arg1"}() : !sdfg.array<i32>
  …
  %a = sdfg.load %0[] : !sdfg.array<i32> -> i32
  %b = sdfg.load %1[] : !sdfg.array<i32> -> i32
  %c = sdfg.tasklet @add(%a, %b) -> (i32) {...}
  …
}
```

# Optimizing Passes


```
sdfg.sdfg @name(%arg0: i32, %arg1: i32) {
  …
  %c = sdfg.tasklet @add(%arg0, %arg1) -> (i32) {...}
  …
}
```

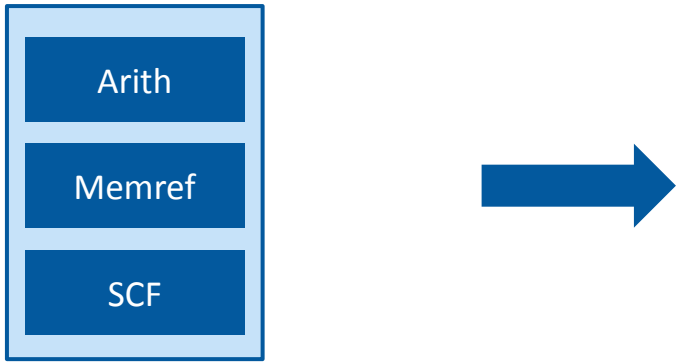
```
sdfg.sdfg @name(%arg0: i32, %arg1: i32) {
  %0 = sdfg.alloc {name = "_arg0"}() : !sdfg.array<i32>
  %1 = sdfg.alloc {name = "_arg1"}() : !sdfg.array<i32>
  …
  %a = sdfg.load %0[] : !sdfg.array<i32> -> i32
  %b = sdfg.load %1[] : !sdfg.array<i32> -> i32
  %c = sdfg.tasklet @add(%a, %b) -> (i32) {...}
  …
}
```

```
sdfg.state @state_1 {
  …
  %8 = sdfg.load %5[] : !sdfg.array<index> -> index
}
```

```
sdfg.state @state_1 {
  …
  // Removed dead load operation
}
```

# Polygeist

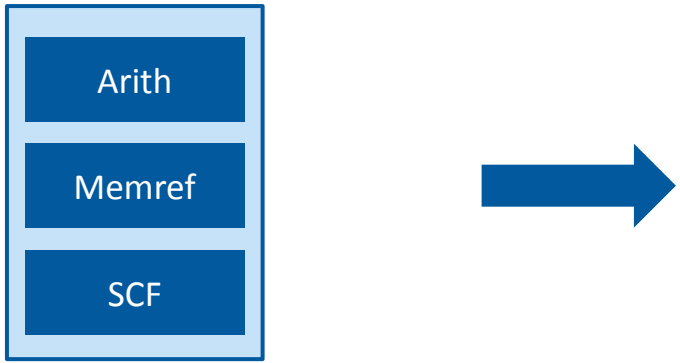

# Converter




MLIR-SDFG

# Converter



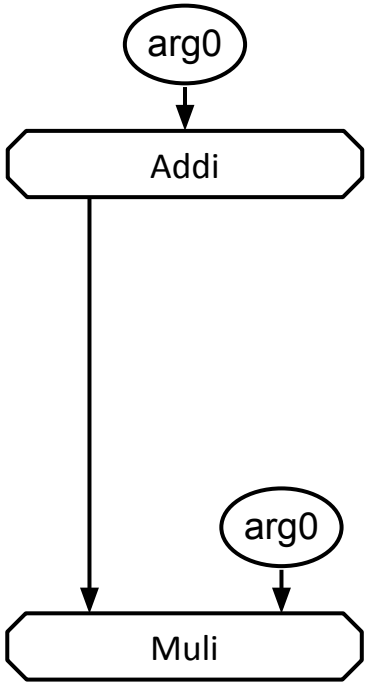

MLIR-SDFG

# Converter: Funcs


```
func @binops(%arg0: i32) {
  %c0 = arith.addi %arg0, %arg0 : i32
  %c1 = arith.muli %c0, %arg0 : i32
  return
}
```

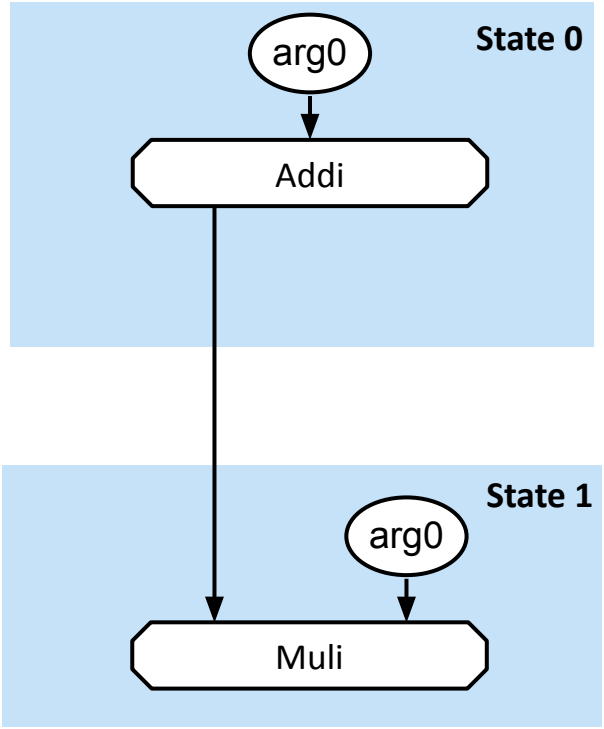
```
sdfg.sdfg @binops(%arg0: i32) {

  %c0 = arith.addi %arg0, %arg0 : i32

  %c1 = arith.muli %c0, %arg0 : i32

}
```

# Converter: Arith Operations

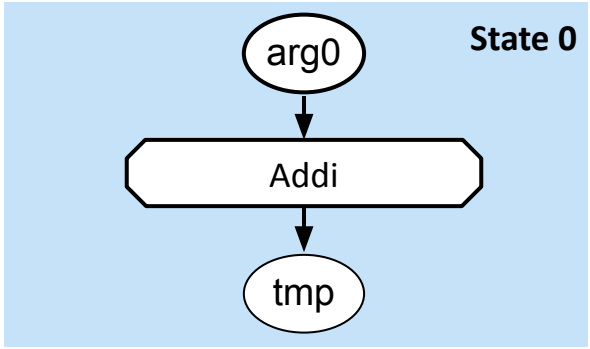

```
sdfg.sdfg @binops(%arg0: i32) {

  %0 = sdfg.tasklet @task_0(%arg0: i32) -> i32 {
    %c0 = arith.addi %arg0, %arg0 : i32
    sdfg.return %c0 : i32
  }

  %1 = sdfg.tasklet @task_1(%0: i32, %arg0: i32) -> i32 {
    %c1 = arith.muli %0, %arg0 : i32
    sdfg.return %c1 : i32
  }

}
```

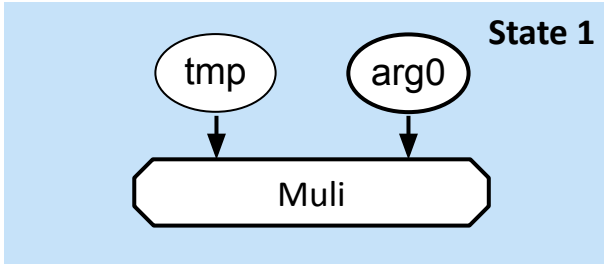
# Converter: Arith Operations


```
sdfg.sdfg @binops(%arg0: i32) {

  sdfg.state @state_0 {
    %0 = sdfg.tasklet @task_0(%arg0: i32) -> i32 {
      %c0 = arith.addi %arg0, %arg0 : i32
      sdfg.return %c0 : i32
    }
  }

  sdfg.state @state_1 {
    %1 = sdfg.tasklet @task_1(%0: i32, %arg0: i32) -> i32 {
      %c1 = arith.muli %0, %arg0 : i32
      sdfg.return %c1 : i32
    }
  }


}
```

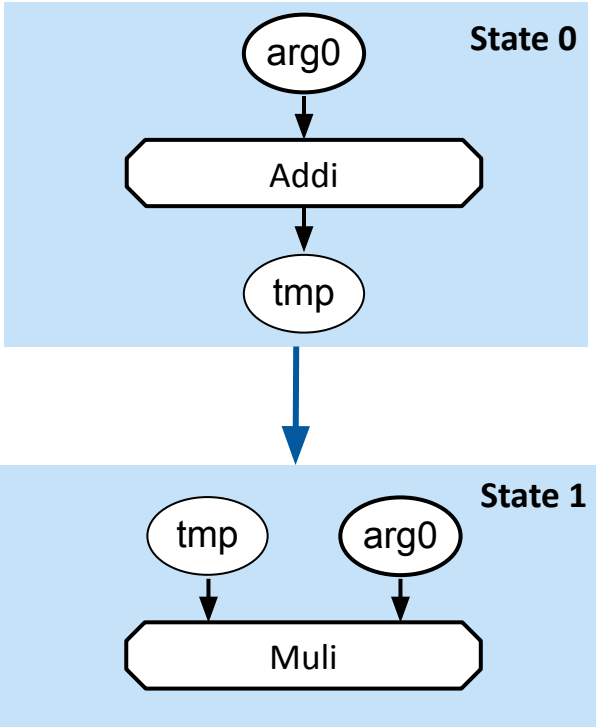
# Converter: Arith Operations


```
sdfg.sdfg @binops(%arg0: i32) {
 %tmp = sdfg.alloc {transient}() : !sdfg.array<i32>

 sdfg.state @state_0 {
  %0 = sdfg.tasklet @task_0(%arg0: i32) -> i32 {
   %c0 = arith.addi %arg0, %arg0 : i32
   sdfg.return %c0 : i32
  }
 }

 sdfg.state @state_1 {
  %1 = sdfg.tasklet @task_1(%0: i32, %arg0: i32) -> i32 {
   %c1 = arith.muli %0, %arg0 : i32
   sdfg.return %c1 : i32
  }
 }


}
```

# Converter: Arith Operations


```
sdfg.sdfg @binops(%arg0: i32) {
  %tmp = sdfg.alloc {transient}() : !sdfg.array<i32>

  sdfg.state @state_0 {
    %0 = sdfg.tasklet @task_0(%arg0: i32) -> i32 {
      %c0 = arith.addi %arg0, %arg0 : i32
      sdfg.return %c0 : i32
    }
  }

  sdfg.state @state_1 {
    %1 = sdfg.tasklet @task_1(%0: i32, %arg0: i32) -> i32 {
      %c1 = arith.muli %0, %arg0 : i32
      sdfg.return %c1 : i32
    }
  }

  sdfg.edge @state_0 -> @state_1
}
```

# Converter: Memref Operations


```
%a = memref.load %A[%0] : memref<900xi32>
```

```
%a = sdfg.load %A[%0] : !sdfg.array<900xi32> -> i32
```

```
memref.store %a, %A[%0] : memref<900xi32>
```

```
sdfg.store %a, %A[%0] : i32 -> !sdfg.array<900xi32>
```

# Converter: Memref Operations


```
%a = memref.load %A[%0] : memref<900xi32>
```

```
%a = sdfg.load %A[%0] : !sdfg.array<900xi32> -> i32
```

```
memref.store %a, %A[%0] : memref<900xi32>
```

```
sdfg.store %a, %A[%0] : i32 -> !sdfg.array<900xi32>
```

# Converter: Memref Operations


```
%a = memref.load %A[%0] : memref<900xi32>
```

```
memref.store %a, %A[%0] : memref<900xi32>
```

```
%a = sdfg.load %A[%0] : !sdfg.array<900xi32> -> i32
```

```
sdfg.store %a, %A[%0] : i32 -> !sdfg.array<900xi32>
```

```
memref<?x900xi32>
```

# Converter: Memref Operations


%a = memref.load %A[%0] : memref<900xi32>

%a = sdfg.load %A[%0] : !sdfg.array<900xi32> -> i32

memref.store %a, %A[%0] : memref<900xi32>

sdfg.store %a, %A[%0] : i32 -> !sdfg.array<900xi32>

memref<?x900xi32>

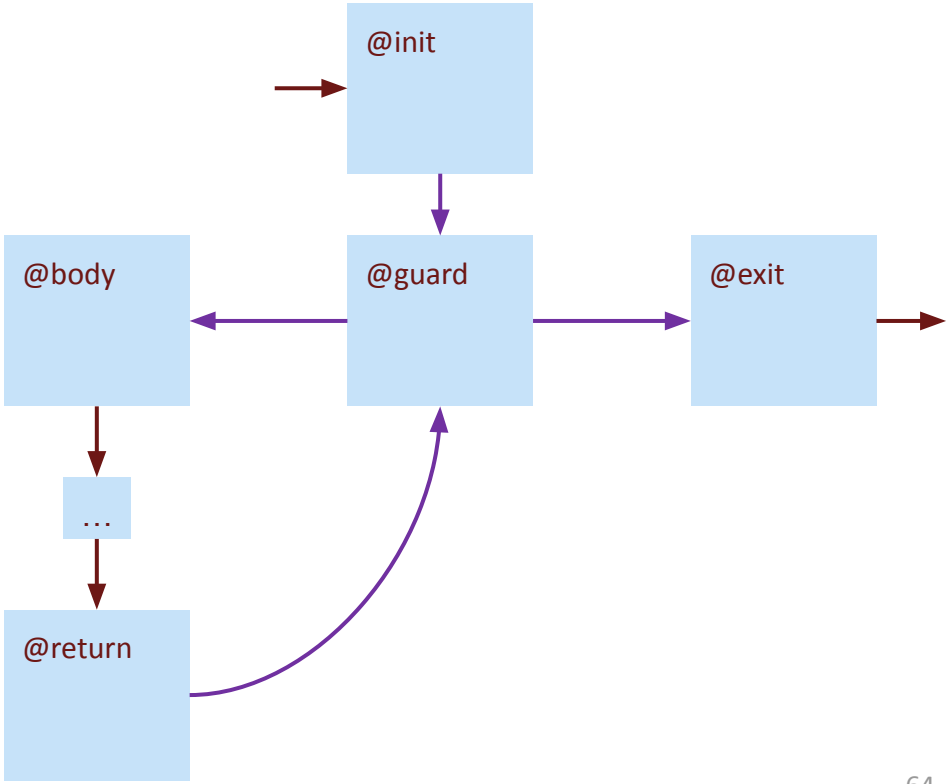
!sdfg.array<sym("N")x900xi32>

# Converter: SCF For Loops


```
scf.for %arg11 = %c0 to %c1 step %c2 {
  // Loop Body
}
```

# Converter: SCF For Loops


```
sdfg.state @init {}
sdfg.state @guard {}
sdfg.state @body {}

// Loop Body

sdfg.state @return {}
sdfg.state @exit {}

sdfg.edge{assign=["i: 0"]} @init -> @guard
sdfg.edge{condition="i < 10"} @guard -> @body
sdfg.edge{condition="i >= 10"} @guard -> @exit
sdfg.edge{assign=["i: i + 1"]} @return -> @guard
```



# Experimental Setup

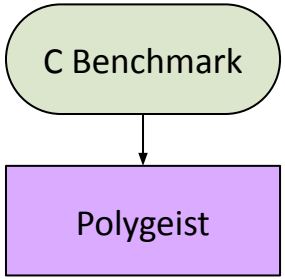
- PolyBench's 2mm, adi, gemver, heat-3d, trmm

# Experimental Setup

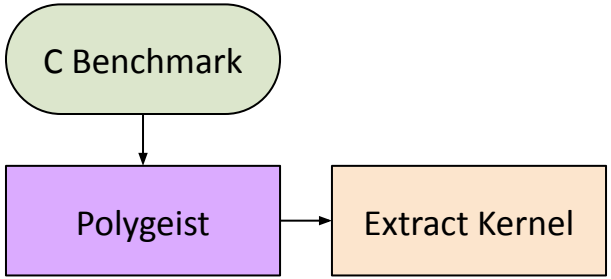
- PolyBench's 2mm, adi, gemver, heat-3d, trmm
- CSCS Ault server
- Build compilation pipeline for MLIR-SDFG
- Compare with clang, gcc, polly, pluto
- Single- & Multi-Threaded
- Run 100 times on the large dataset
- 10 Runs warm-up

| | |
|---|---|
| Architecture | x86_64 |
| CPUs | 72 |
| Cores per socket | 18 |
| Base frequency | 3.00 GHz |
| L1I cache size | 32kB |
| L1D cache size | 32kB |
| LLVM/MLIR version | 14.0.0git |
| Clang version | 10.0.1 |
| DaCe version | 0.11.4 |

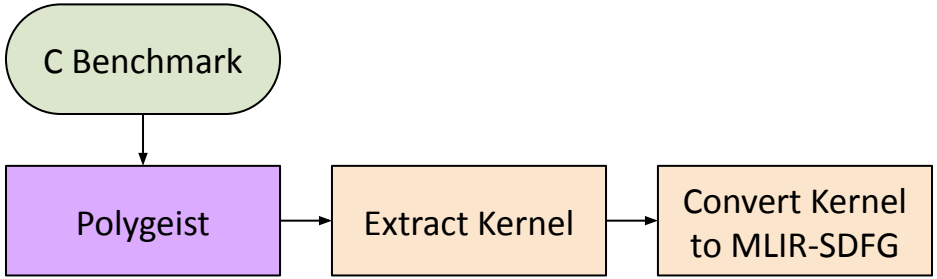
# Compilation Pipeline



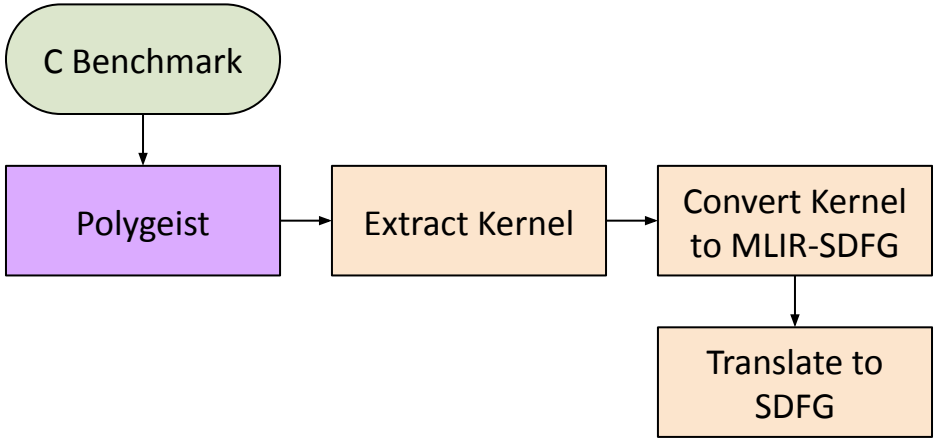
# Compilation Pipeline



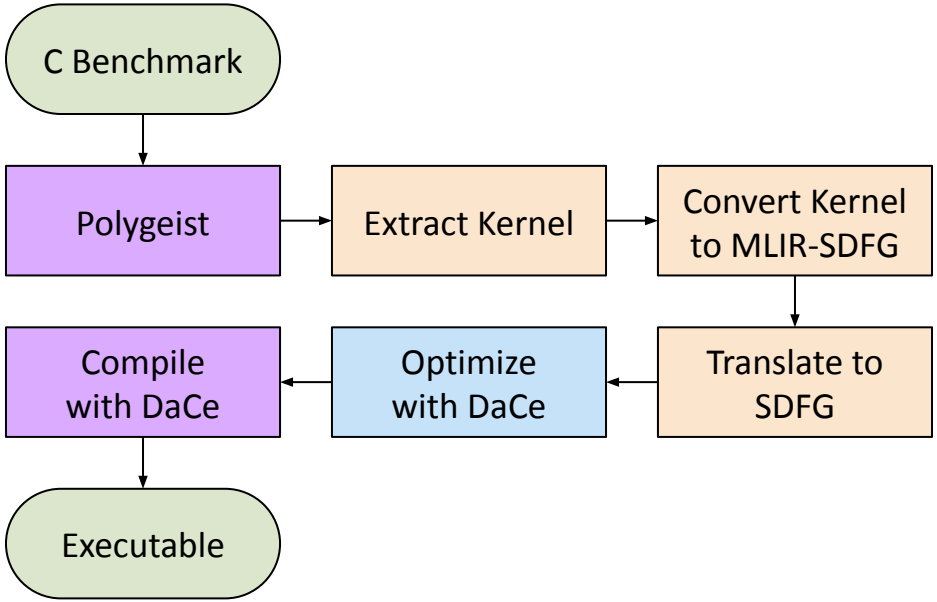
# Compilation Pipeline



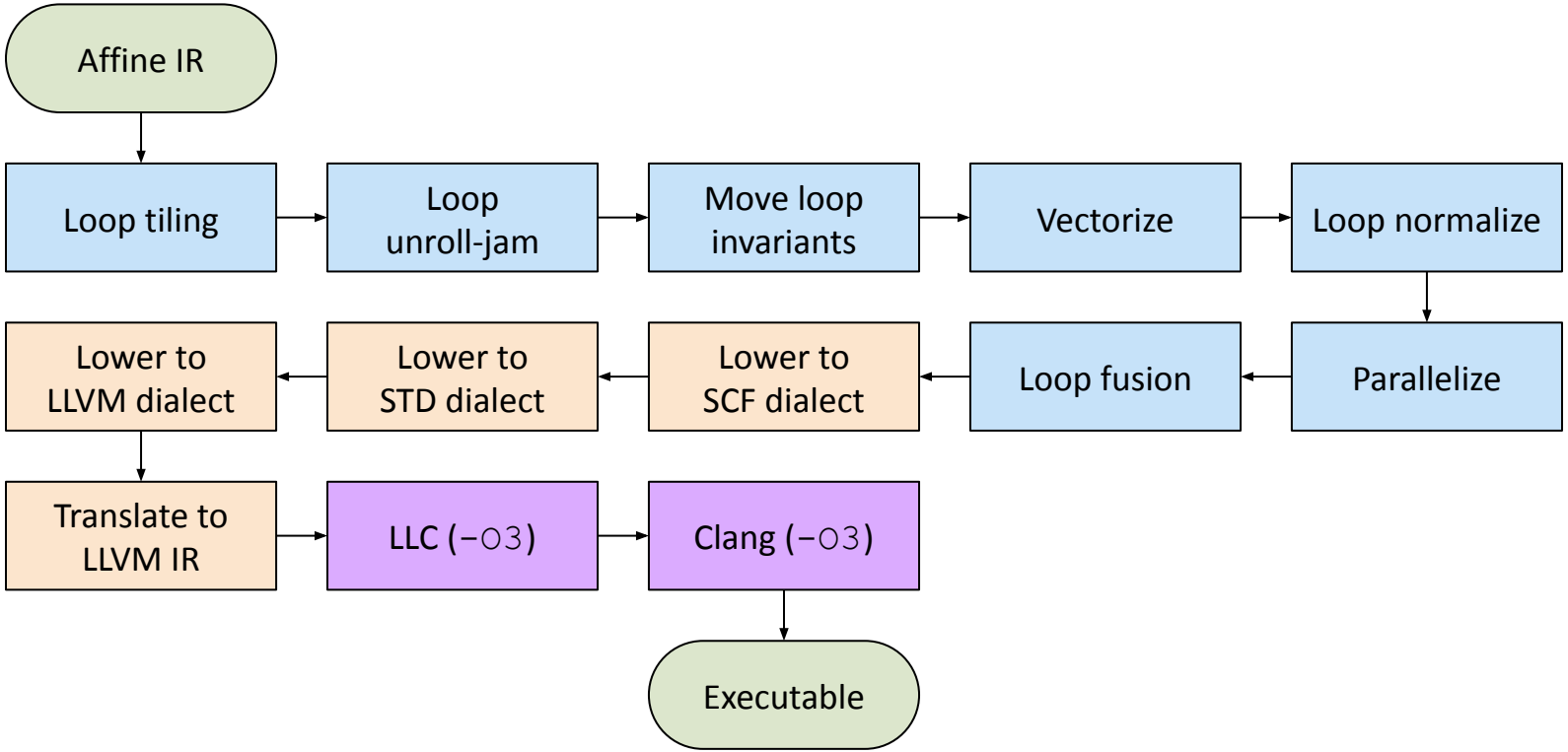
# Compilation Pipeline



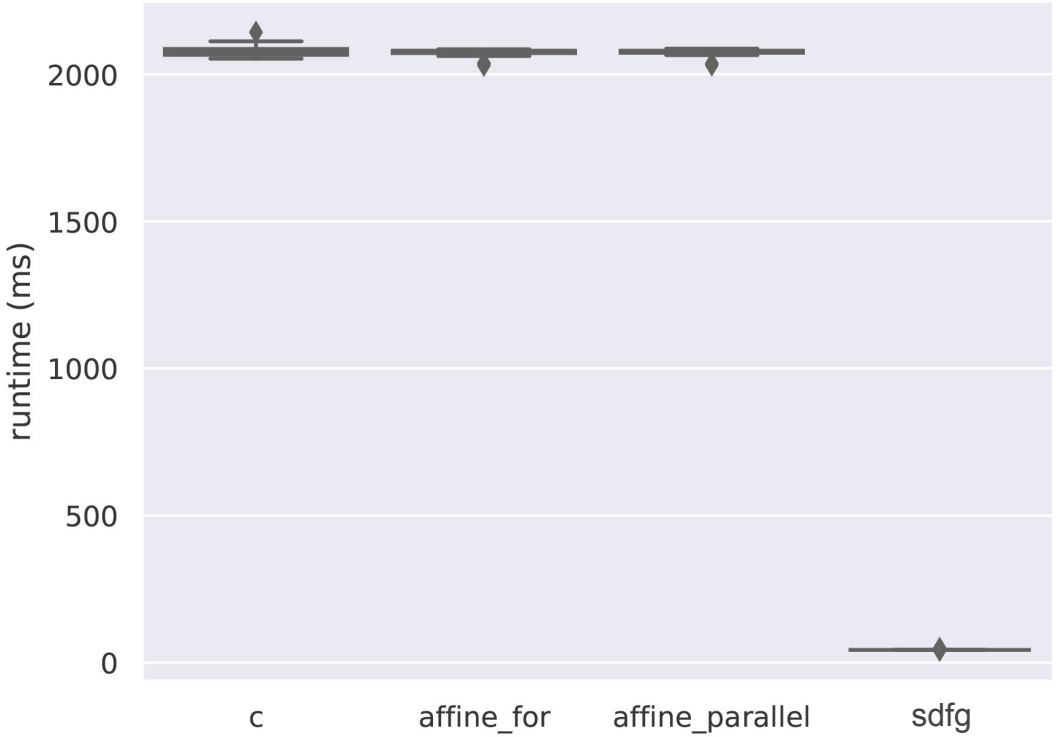
# Compilation Pipeline



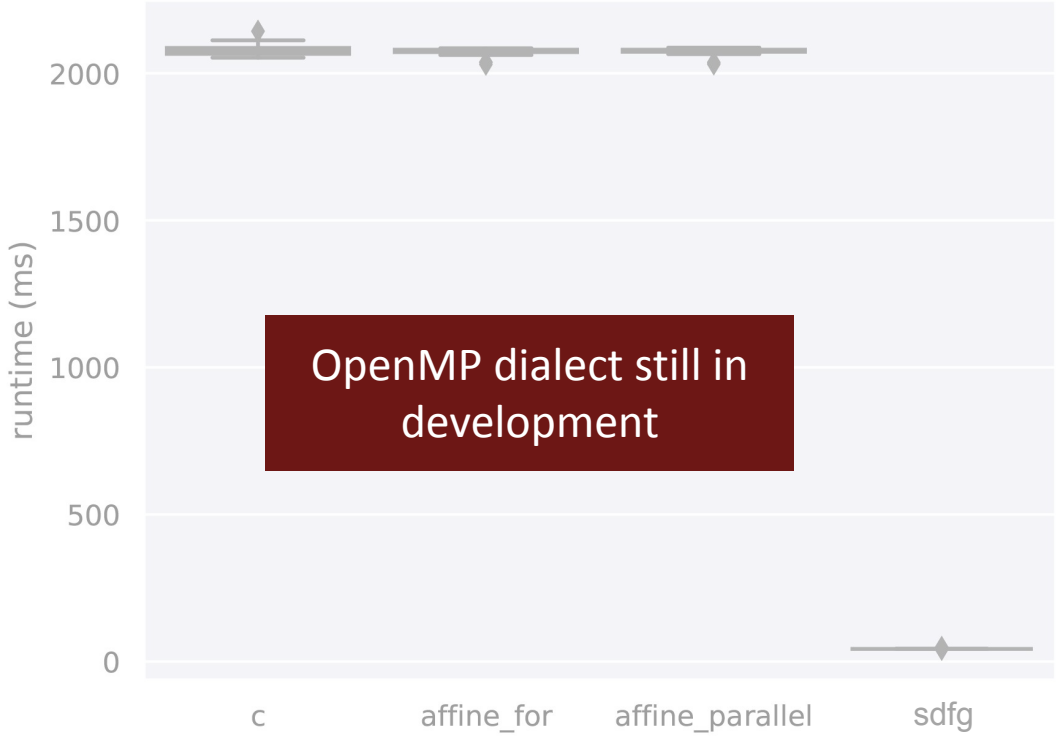
# Affine pipeline



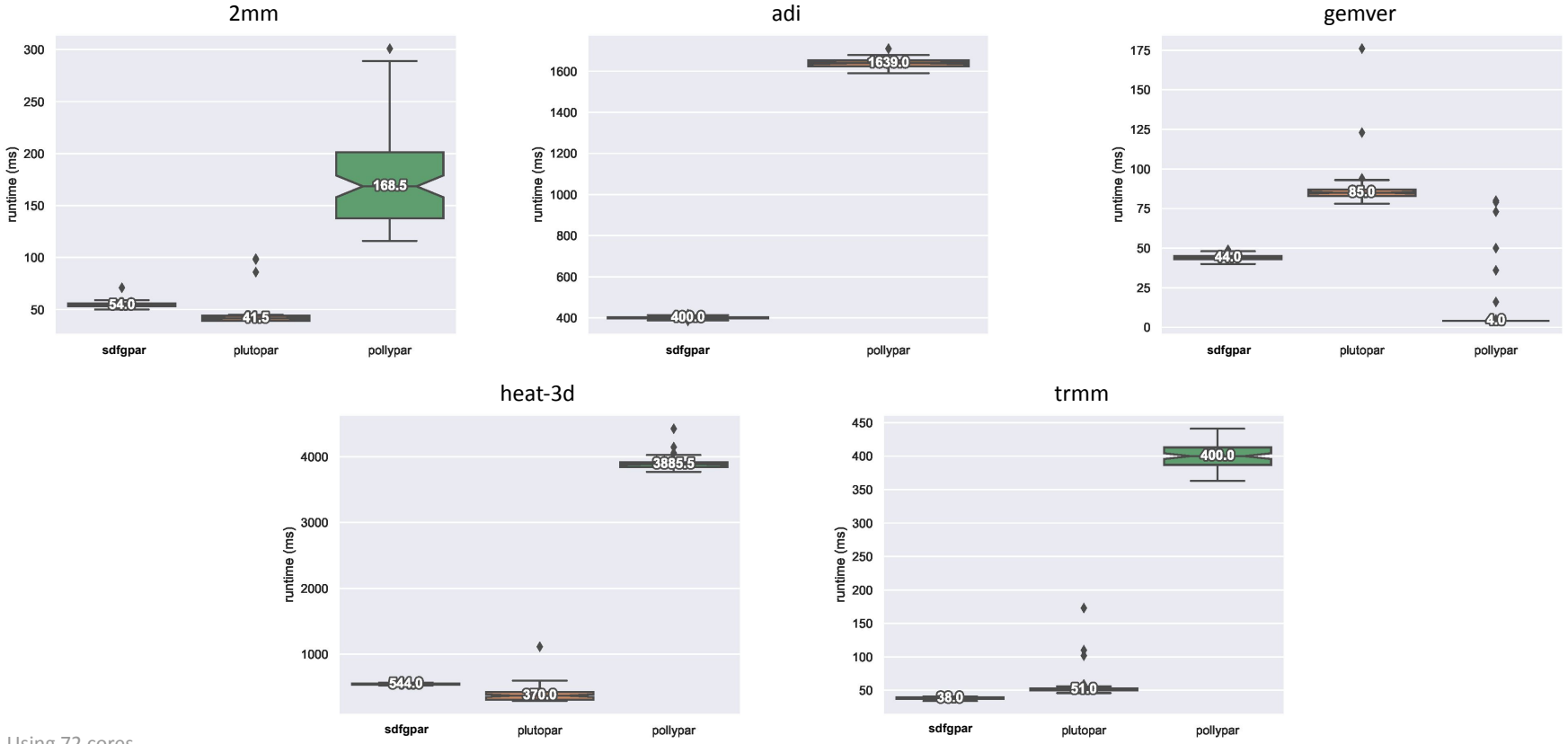
# 2MM Affine vs. MLIR-SDFG



# 2MM Affine vs. MLIR-SDFG

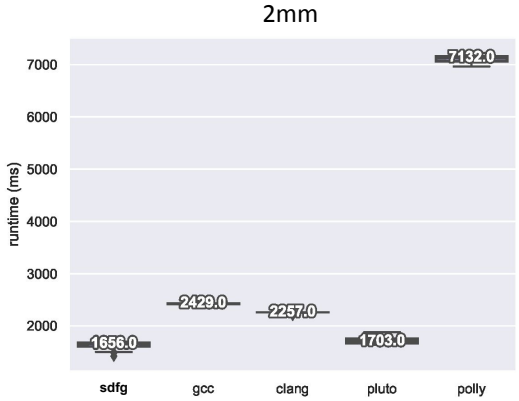


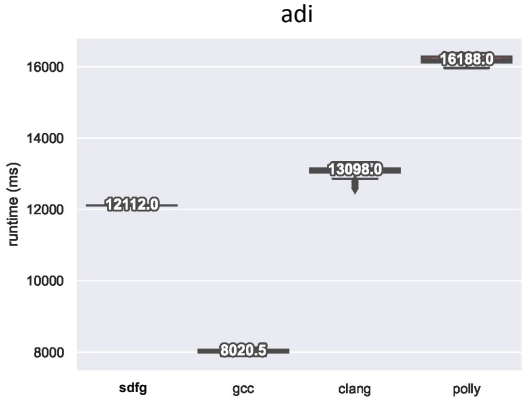
# Parallel Benchmarks

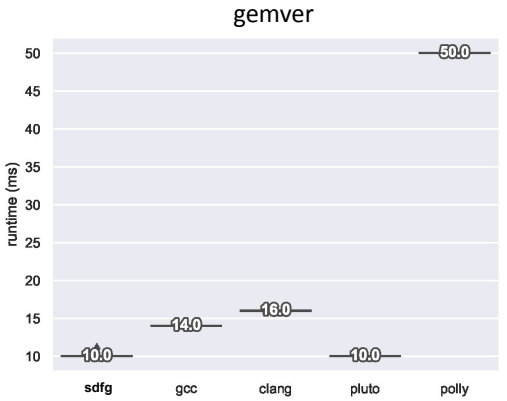


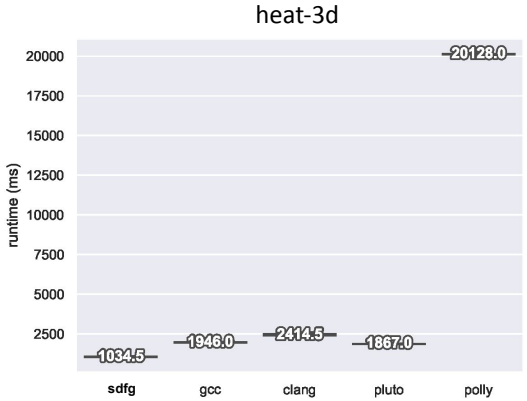

Using 72 cores

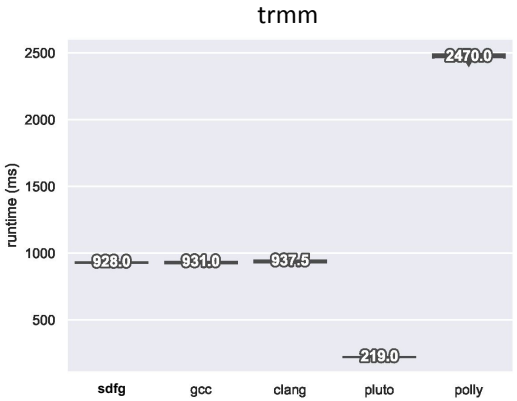
# Sequential Benchmarks



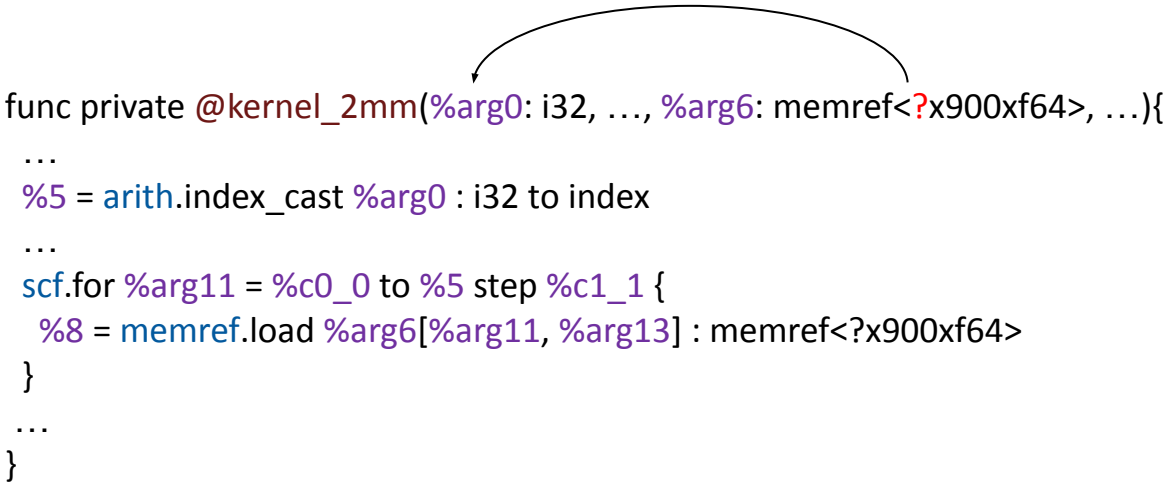



# Open MLIR Challenges

```
func private @kernel_2mm(%arg0: i32, …, %arg6: memref<?x900xf64>, …){
  …
  %5 = arith.index_cast %arg0 : i32 to index
  …
  scf.for %arg11 = %c0_0 to %5 step %c1_1 {
    %8 = memref.load %arg6[%arg11, %arg13] : memref<?x900xf64>
  }
 …
}
```

# Open MLIR Challenges



# Open MLIR Challenges

func private @kernel_2mm(%arg0: i32, …, %arg6: memref<?x900xf64>, …)

!sdfg.array<sym("s_0")x900xf64>

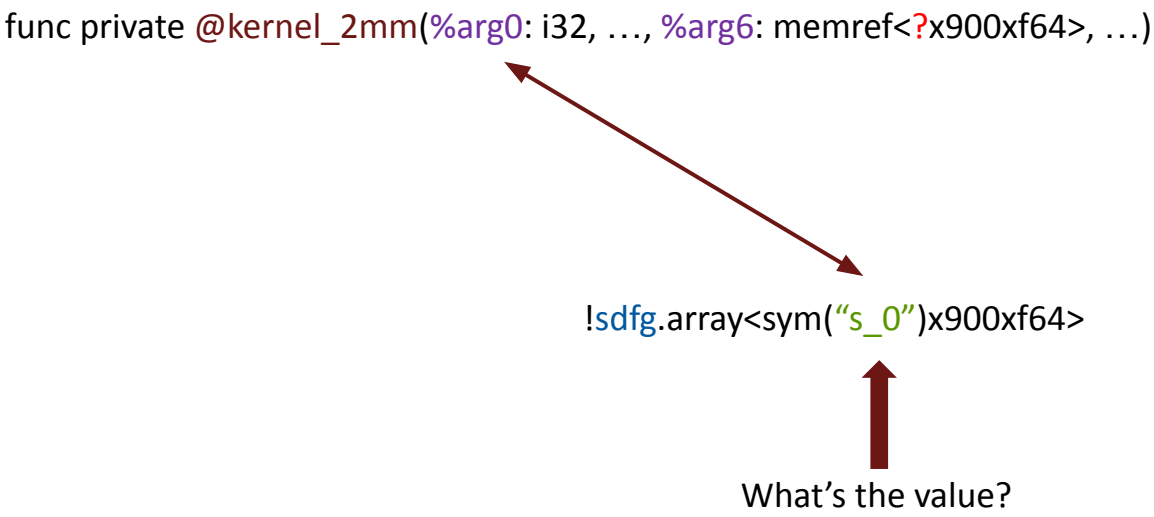
# Open MLIR Challenges

func private @kernel_2mm(%arg0: i32, …, %arg6: memref<?x900xf64>, …)

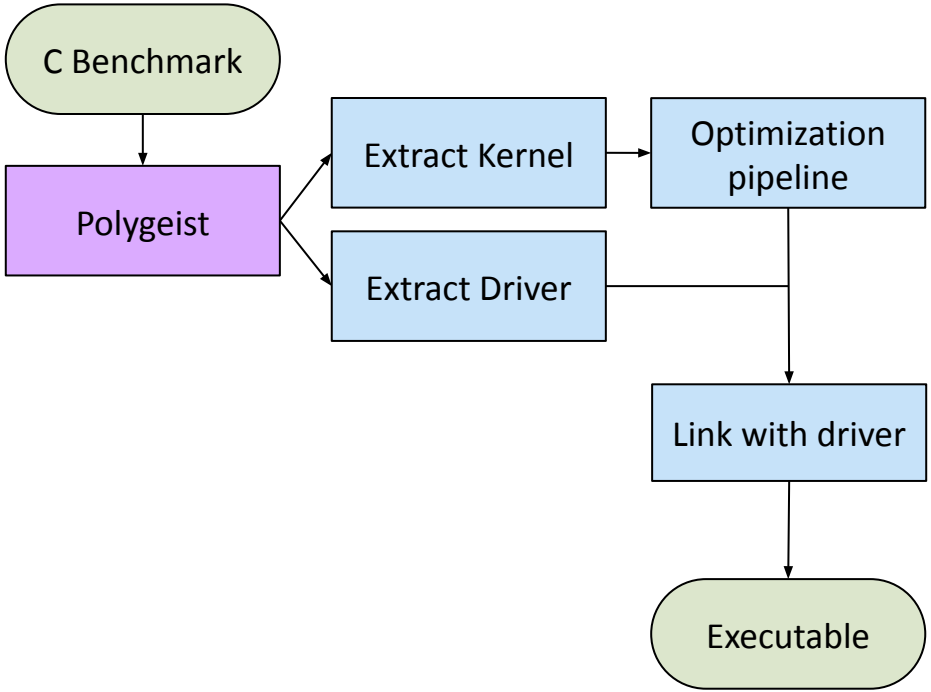
!sdfg.array<sym("s_0")x900xf64>

What's the value?

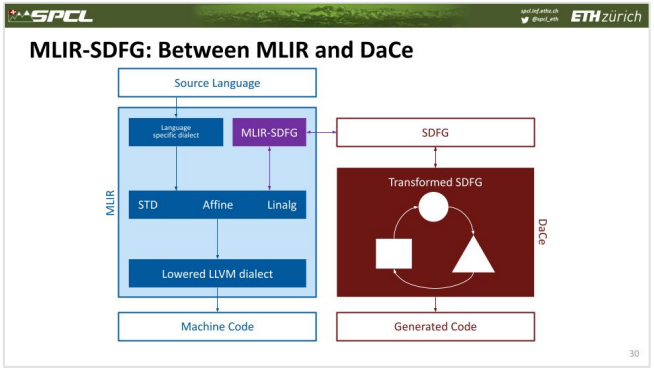
# Open MLIR Challenges

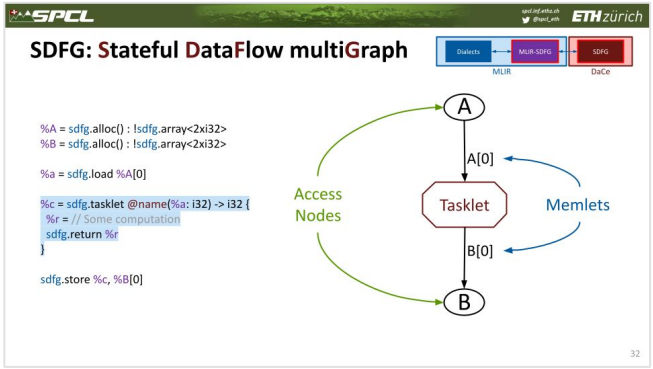


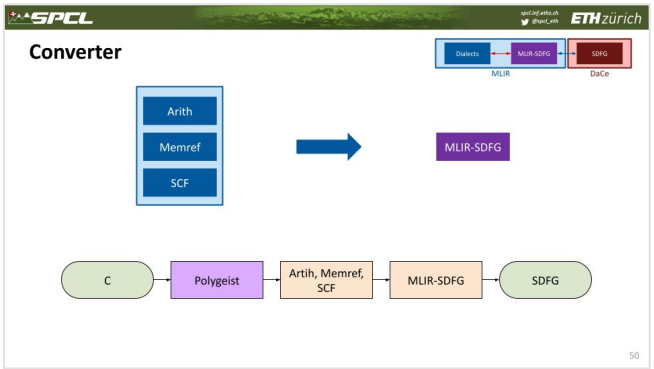
# Future Work

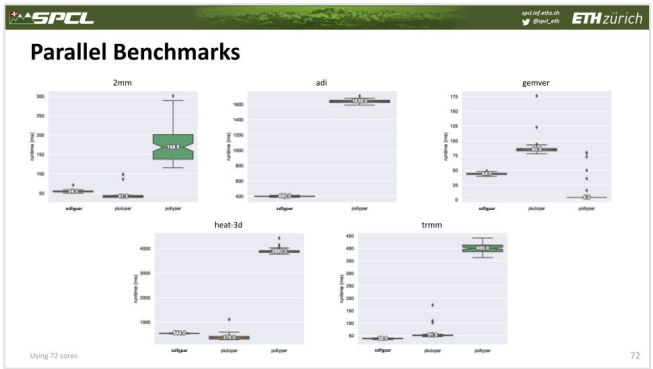

C Benchmark
Polygeist
Extract Kernel
Optimization pipeline
Extract Driver
Link with driver
Executable

# Summary


MLIR-SDFG: Between MLIR and DaCe
Source Language
Language specific dialect
MLIR-SDFG
STD
Affine
Linalg
Lowered LLVM dialect
MLIR
Machine Code
SDFG
Transformed SDFG
DaCe
Generated Code
30


SDFG: Stateful DataFlow multiGraph
Dialects
MLIR-SDFG
SDFG
MLIR
DaCe
%A = sdfg.alloc() : !sdfg.array<2xi32>
%B = sdfg.alloc() : !sdfg.array<2xi32>
%a = sdfg.load %A[0]
%c = sdfg.tasklet @name(%a: i32) -> i32 {
%r = // Some computation
sdfg.return %r
}
sdfg.store %c, %B[0]
Access Nodes
A
A[0]
Tasklet
Memlets
B[0]
B
32


Converter
Dialects
MLIR-SDFG
SDFG
MLIR
DaCe
Arith
Memref
SCF
MLIR-SDFG
C
Polygeist
Artih, Memref, SCF
MLIR-SDFG
SDFG
50


Parallel Benchmarks
2mm
adi
gemver
heat-3d
trmm
Using 72 cores
72

**Berke Ates,** Prof. Dr. Torsten Hoefler, Dr. Tal Ben-Nun, Dr. Alexandru Calotoiu

# MLIR-SDFG: A Data-Centric Dialect for MLIR