RISE in MLIR

A functional Pattern-based Dialect

Martin Lücke | Michel Steuwer | Aaron Smith





supported by a Google Faculty Award sponsored by Jacques Pienaar and Albert Cohen

Why RISE?

Machine Learning Systems are Stuck in a Rut

[HotOS'19]

Paul Barham __ Google Brain

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on high-performance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards high-performance general purpose numerical computing systems on modern accelerators.

ACM Reference Format:

Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

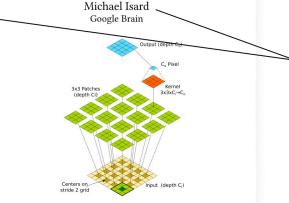


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convolutional neural network (CNN) we were comparing it to, implementations in both TensorFlow[2] and PyTorch[3] were much slower and ran out of memory with much smaller models. We wanted to understand why.

1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper¹, but for our purposes it is sufficient to consider a simplified form of the inner loop, which is Original authors of TensorFlow



Why RISE?

Machine Learning Systems are Stuck in a Rut

Paul Barham Google Brain

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on highperformance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards high-performance general purpose numerical computing systems on modern accelerators.

ACM Reference Format:

Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

Michael Isard Google Brain

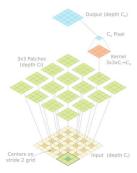


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convolutional neural network (CNN) we were comparing it to, implementations in both TensorFlow[2] and PyTorch[3] were much slower and ran out of memory with much smaller models. We wanted to understand why.

1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper¹, but for our purposes it is sufficient to consider a simplified form of the inner loop, which is

Why RISE?

Machine Learning Systems are Stuck in a Rut

Paul Barham Google Brain

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on highperformance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards.

Michael Isard Google Brain

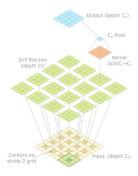


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convo-

We should aim for more principled higher level intermediate representations

Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

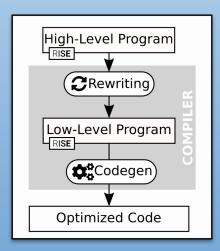
smaller models. We wanted to understand why.

1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper¹, but for our purposes it is sufficient to consider a simplified form of the inner loop, which is

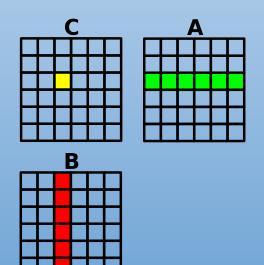
RISE - A functional pattern-based data-parallel language

- RISE (https://rise-lang.org/) is a spiritual successor to the Lift project
- Computations represented using compositions of flexible and generic patterns
- Rewriting system to explore optimization choices

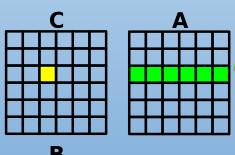


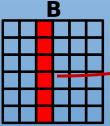
RISE by example: Matrix Multiplication

```
fun(A : N.K.float ⇒ fun(B : K.M.float ⇒
A ▷ map(fun(arow ⇒
B ▷ transpose ▷ map(fun(bcol ⇒
    zip(arow, bcol) ▷ map(*) ▷ reduce(+, 0) )) )))
```



RISE by example: Matrix Multiplication

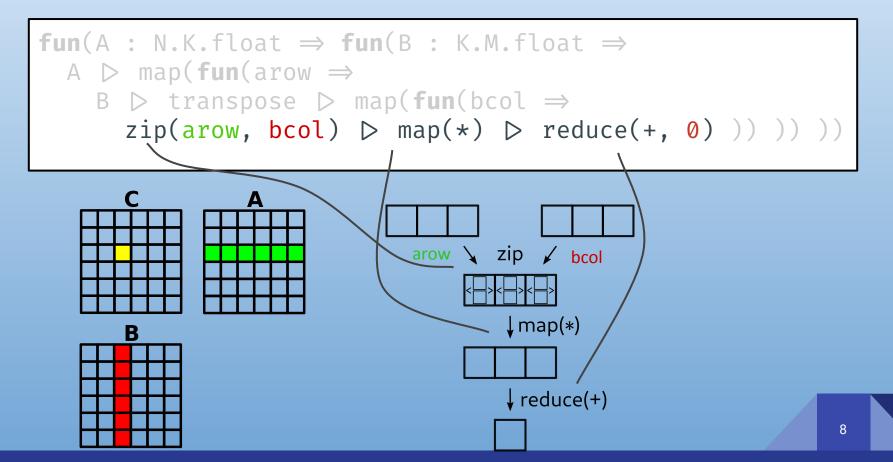




dot product computation:

$$\sum arow_i * bcol_i$$

RISE by example: Matrix Multiplication



Optimization choices for Matrix Multiplication

```
fun(A : N.K.float ⇒ fun(B : K.M.float ⇒
                                A \triangleright map(fun(arow \Rightarrow
                                                                    B \triangleright transpose \triangleright map(fun(bcol \Rightarrow
                                                                                                     zip(arow, bcol) \triangleright map(*) \triangleright reduce(+, 0)))))
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 RISE compiler
                                                                                                                                                     / kernel _attribute(freed,work.group.size(3), 8, 1)))
wid HEMBEL(const global float *restrict A, const global float
restrict B,
const global float *restrict C, float olpha, float
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    kernel void mm(global float4" const A.
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      global float4* const B,
                                                                                                                                                                                                                                                                                                                                                                                                   which produces that we produce t_i and t_i 
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      global float2* (, uint n) {
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           uint i = get_global_id(0);
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           uint j = get_global_id(1);
                                                                                                                                                        for (int wi-wid); wi-8/64; wi--num_grps(1)) {
   for (int wi-wid); wi-8/64; wi--num_grps(0))
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           uint nv4 = n >> 2;
float4 ab = (float4)(0.0f);
                                                                                                                                                      acc. 8 = 0.05; ...; acc. 31 = 0.05; acr. 32 = 0.05; ...; acc. 31 = 0.05; acr. 32 = 0.05; acc. 32 = 0.05; acc. 33 = 0.05; acc. 33 = 0.05; acc. 34 = 0.05; acc. 
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             for (uint k = 0; k < nv4; ++k) {
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     float4 a6 = A[ 2*i *nv4+k];
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       float4 a1 = A[(2*i+1)*nv4+k];
                                                                                                                                                                                                                                                                                                                                                                                                     for (int j = 0; j=0; j=0; )
blockOfA.6 = tlist(0=04*j=1[d]*0];
... 0 mer statement
blockOfA.7 = tlist(7=04*j=1[d]*0];
blockOfA.5 = tlist(0=04*j=1[d]*0];
... 2 mer statement
blockOfA.5 = tlist(0=04*j=1[d0]);
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       float4 b0 = B[ 2*j *nv4+k];
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       float4 b1 = B[(2*j+1)*nv4+k];
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     ab += (float4)(dot(a0, b0), dot(a0, b1),
                                                                                                                                                                                                                                                                                                                                                                                                       }
barrier(CK_LBCM_ABM_FBKCE);
fire (Line_1_Lb_0 = get_Lecol_Lb(0); (L_bd_0 = 128); (L_bd_0 = 1
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         dot(a1, b0), dot(a1, b1)); }
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             uint ix = 2*i*(n>>1) + j;
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           Cfixl
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      = ab.s01:
                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           C[ix + (n>>1)] = ab.s23; }
                                                                                                                                                                                                                                                                                                                                                                                                  p_tmp_2_2_466 = (1_tmp_2[()2])
pet_lecal_3d(0))]);
```

Optimization choices for Matrix Multiplication

(32 + 1_(6,8)) (1_tmp_S((1_(6,8 + (6+1_(6,1)))) + (4((1_(6,8 + (8 + 8 + 1) + (64 + mg_1(6,1) + (8 + (6,1))))) (64 + mg_1(6,1) + (8 + (6,1))) (64 + (6,1)) (64

}
harrise(CLK_LECKL_MBM_FERCE);
for (ine _ie_0 = ger(lecs_is(0), (_is_0) = r.28); _is_0 = fr.
for + _is_0 = ger(lecs_is(0)); _is_0 = r.
for + _is_0 = fr.
for _is_0 = f

for (lat j = 6; j-8; j++) {
 blockOfs_6 = tiles[0-64*j-lid1*8];
 ... 6 more statements
 blockOfs_7 = tiles[1*64*j-lid1*8];
 blockOfs_6 = tiles[0 +64*j-lid9);

```
fun(A : N.K.float ⇒ fun(B : K.M.float ⇒
     A \triangleright map(fun(arow \Rightarrow
          B \triangleright transpose \triangleright map(fun(bcol \Rightarrow
                zip(arow, bcol) \triangleright map(*) \triangleright reduce(+, 0)))))
                                                                                                                                                                                    Rewrite Rules
                                                                                                                                                       map(f, A) \mapsto join(map(map(f),
                                                                                                 RISE compiler
                                                                                                                                                                                                                split(n, A)))
                        kernel void mm(global float4" const A.
                                                               d Remotilianos grows.

Strict B,

const global float *Pestrict C, Float alpha, float
                                                                                                                                global float4* const B.
                                                                                                                                global float2* C, uint r
                                                                  global float east, let K, let M, let W) (
                                                                                                                    uint i = get_global_id(0);
                                                                                                                    uint j = get_global_id(1);
                                                              float ac(,),(a5 = 4.09)

// ... 31 one;

float p.tag,(1,54);

for #2,64 > pttps=0,1641;

for #2,64 > pttps=0,1640;

for (for 1 = 4, 1 < (K / 8); 1 < 1 + 1)) {

   ist (3,1 > pttps=0,1643,1640);

   ist (3,1 > pttps=0,343);

   ist (3,1 > pttps=0,343);
                        for (int vi-wid); vi-8/64; vi-mum.grps(1)) (
                                                                                                                    uint nv4 = n >> 2;
float4 ab = (float4)(0.0f);
```

for (uint k = 0; k < nv4; ++k) { float4 a6 = A[2*i *nv4+k]; float4 a1 = A[(2*i+1)*nv4+k];

float4 b0 = B[2*j *nv4+k];

float4 b1 = B[(2*j+1)*nv4+k]; ab += (float4)(dot(a0, b0), dot(a0,

= ab.s01: C[ix + (n>>1)] = ab.s23; }

uint ix = 2*i*(n>>1) + j;

Cfixl

dot(a1, b0), dot(a1,

RISE Compilation

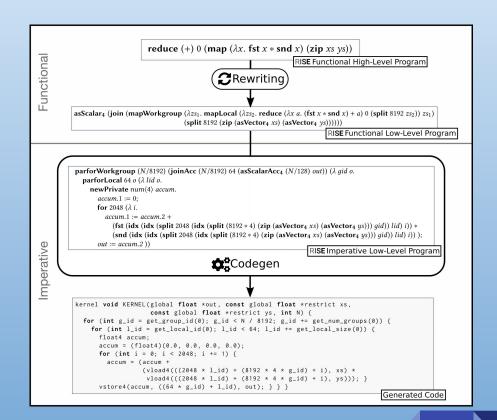
Compilation is divided into two steps:

- 1. Translation from functional to imperative low level program
- 2. Produce generated code

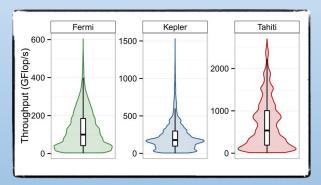
Compilation described formally in:

Strategy Preserving Compilation for Parallel Functional Code by Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach.

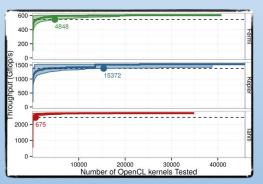
https://arxiv.org/abs/1710.08332



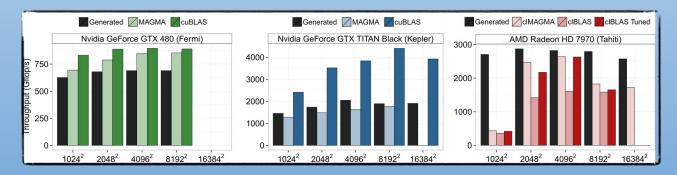
High Performance Results for Matrix Multiplication



Only few generated code with very good performance



Still: One can expect to find a good performing kernel quickly!



[GPGPU'16]

Performance close or better than hand-tuned library code

RISE - The Caveats

- Does everyone have to write functional programs now?
- Academic work written in Scala
- Does not integrate well with existing compiler infrastructures



MLIR - Multi-Level Intermediate Representation

- Extensible infrastructure to define compiler intermediate representations
- Dialects can capture different levels of abstraction:
 - High-level domain specific ----- Hardware specific backend
- Existing dialects available for:
 - TensorFlow / TensorFlow Lite
- Performing polyhedral optimizations

- Targeting GPUs

- LLVM IR

-



MLIR - Martin Lücke Intermediate Representation

- Extensible infrastructure to define compiler intermediate representations
- Dialects can capture different levels of abstraction:
 - High-level domain specific ----- Hardware specific backend
- Existing dialects available for:
 - TensorFlow / TensorFlow Lite
- Performing polyhedral optimizations

- Targeting GPUs

- LLVM IR

- ...



MLIR - Multi-Level Intermediate Representation

- Extensible infrastructure to define compiler intermediate representations
- Dialects can capture different levels of abstraction:
 - High-level domain specific ----- Hardware specific backend
- Existing dialects available for:
 - TensorFlow / TensorFlow Lite
- Performing polyhedral optimizations

- Targeting GPUs

- LLVM IR

- ...

RISE in MLIR = Lambda Calculus + Patterns in MLIR

- RISE in MLIR opens up opportunities to integrate with other MLIR dialects
- We do not have to write programs in RISE directly, but lower from domain-specific dialects to it
- MLIR is written in C++, using an established toolchain -> widely usable
- Natural integration with existing toolchains #include libMLIR

- -> to implement RISE as an MLIR dialect we implement:
 - λ-calculus
 - patterns

RISE dialect by example: Matrix Multiplication

```
1 func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
   %A = rise.in %inA : !rise.arrav<1024. arrav<1024. scalar<f32>>>
   %B = rise.in %inB : !rise.arrav<1024. arrav<1024. scalar<f32>>>
   %f1 = rise.lambda (%arow : !rise.arrav<1024, scalar<f32>>) → !rise.arrav<1024, scalar<f32>> {
     %f2 = rise.lambda (%bcol : !rise.arrav<1024. scalar<f32>>) → !rise.arrav<1024. scalar<f32>> {
       %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
       %zipped = rise.apply %zip, %arow, %bcol
       %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>>) → !rise.scalar<f32> {
         %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
         %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
11
         %fst = rise.applv %fstFun. %tuple
12
         %snd = rise.applv %sndFun. %tuple
13
         %result = rise.embed(%fst, %snd) {
14
          %res = mulf %fst. %snd : f32
15
           return %res : f32
16
         } : !rise.scalar<f32>
17
         rise.return %result : !rise.scalar<f32>
18
19
       %map = rise.mapSeg #rise.nat<1024> #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
20
       %multipliedArray = rise.apply %map, %f, %zipped
21
       %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
22
         %result = rise.embed(%a. %b) {
23
           %res = addf %a. %b : f32
24
           return %res : f32
25
         }: !rise.scalar<f32>
26
         rise.return %result : !rise.scalar<f32>
27
28
       %init = rise.literal #rise.lit<0.0>
29
       %reduce = rise.reduceSeg #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
30
       %result = rise.applv %reduce. %add. %init. %multipliedArrav
       rise.return %result : !rise.scalar<f32>
31
32
     %mapB = rise.mapSeg #rise.nat<1024> #rise.array<1024. scalar<f32>> #rise.array<1024. scalar<f32>>
33
     %result = rise.apply %mapB, %f2, %B
     rise.return %result : !rise.arrav<1024. arrav<1024. scalar<f32>>>
36 }
37 %mapA = rise.mapSeg #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
38 %result = rise.apply %mapA, %f1, %A
39 rise.out %out ← %result
40 return
41 }
```

RISE dialect by example: Matrix Multiplication

```
1 func @mm(%out:memref<1024x1024xf32>, %inA:memref<1024x1024xf32>, %inB:memref<1024xf32>)
   %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
   %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
   %f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) \rightarrow !rise.array<1024, scalar<f32>> {
    %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) \rightarrow !rise.array<1024, scalar<f32>> {
       %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
       %zipped = rise.apply %zip, %arow, %hcol
       %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>>) → !rise.scalar<f32>
         %fstFun = rise.fst #rise.scalar<t32> #rise.scalar<t32>
10
         %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
11
        %fst = rise.apply %fstFun, %tuple
12
         %snd = rise.apply %sndFun, %tuple
         %result = rise.embed(%fst, %snd) {
13
          %res = mulf %fst, %snd : f32
14
15
          return %res : f32
16
         }: !rise.scalar<f32>
         rise.return %result : !rise.scalar<f32>
17
18
       %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
19
       %multipliedArray = rise.apply %map, %f, %zipped
20
21
       %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
22
         %result = rise.embed(%a, %b) {
23
          %res = addf %a, %b : f32
24
          return %res : f32
         } : !rise.scalar<f32>
25
26
         rise.return %result : !rise.scalar<f32>
27
28
       %init = rise.literal #rise.lit<0.0>
       %reduce = rise.reduceSeq #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
       %result = rise.apply %reduce, %add, %init, %multipliedArray
30
31
       rise.return %result : !rise.scalar<f32>
32
33
     %mapB = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
34
     %result = rise.apply %mapB, %f2, %B
     rise.return %result : !rise.array<1024, array<1024, scalar<f32>>>
35
36
38 %result = rise.apply %mapA, %f1, %A
39 rise.out %out ← %result
40 return
41 }
```

Data Types

Function Types

Types

Data Types

```
2 %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
```

- Rise data types: array types, tuple types, scalar types
- Nested array types represent higher dimensional data
- Scalar wraps an arbitrary scalar MLIR type

Function Types

```
%f = rise.lambda (%t : !rise.tuple<scalar<f32>, scalar<f32>>) \rightarrow scalar<f32> // %f : !rise.fun<tuple<scalar<f32>, scalar<f32>> \rightarrow scalar<f32>>
```

- Rise function types: types of lambda expressions and functional patterns
- Type system prevents mixing of function and data types

Patterns

```
1 func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
   %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
   %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
   %f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
     %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
        %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
        %zipped = rise.apply %zip, %arow, %bcol
        %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>>) → !rise.scalar<f32> {
8
         %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
10
          %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
11
         %fst = rise.apply %fstFun, %tuple
12
         %snd = rise.apply %sndFun, %tuple
13
         %result = rise.embed(%fst, %snd) {
14
           %res = mulf %fst, %snd : f32
15
           return %res : f32
         } : !rise.scalar<f32>
16
17
          rise.return %result : !rise.scalar<f32>
18
        %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
19
        %multipliedArray = rise.apply %map, %f, %zipped
20
        %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
21
22
         %result = rise.embed(%a, %b) {
23
           %res = addf %a, %b : f32
24
           return %res : f32
          } : !rise.scalar<f32>
25
26
          rise.return %result : !rise.scalar<f32>
27
28
        %init = rise.literal #rise.lit<0.0>
       %reduce = rise.reduceSeq #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
29
30
        %result = rise.apply %reduce, %add, %init, %multipliedArray
31
        rise.return %result : !rise.scalar<f32>
32
33
     %mapB = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
     %result = rise.apply %mapB, %f2, %B
34
35
     rise.return %result : !rise.array<1024, array<1024, scalar<f32>>>
36 }
37 %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
38 %result = rise.apply %mapA, %f1, %A
39 rise.out %out ← %result
40 return
41 }
```

Patterns

```
1 func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
   %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
   %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
   %f1 = rise.lambda (%arow : !rise.arrav<1024. scalar<f32>>) → !rise.arrav<1024. scalar<f32>> {
     %f2 = rise.lambda (%bcol : !rise.arrav<1024. scalar<f32>>) → !rise.arrav<1024. scalar<f32>> {
6
       %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
7
       %zipped = rise.apply %zip, %arow, %bcol
        %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>. scalar<f32>>) → !rise.scalar<f32> {
8
         %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
10
         %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
11
         %fst = rise.applv %fstFun. %tuple
12
         %snd = rise.apply %sndFun, %tuple
         %result = rise.embed(%fst. %snd) {
13
         %res = mulf %fst, %snd : f32
14
           return %res : f32
15
16
         }: !rise.scalar<f32>
         rise.return %result : !rise.scalar<f32>
17
18
        %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
19
       %multipliedArray = rise.apply %map, %f, %zipped
20
21
       %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
22
         %result = rise.embed(%a, %b) {
           %res = addf %a, %b : f32
23
           return %res : f32
24
         } : !rise.scalar<f32>
25
         rise.return %result : !rise.scalar<f32>
26
27
28
       %init = rise.literal #rise.lit<0.0>
       %reduce = rise.reduceSeg #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
29
       %result = rise.apply %reduce, %add, %init, %multipliedArray
30
31
        rise.return %result : !rise.scalar<f32>
32
     %mapB = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
33
     %result = rise.apply %mapB, %f2, %B
34
     rise.return %result : !rise.arrav<1024. arrav<1024. scalar<f32>>>
35
36
37 %mapA = rise.mapSeg #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
38 %result = rise.apply %mapA, %f1, %A
39 rise.out %out ← %result
40 return
41 }
```

Patterns: zip

```
6 %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
```

Patterns: zip

- Each RISE pattern is implemented as an MLIR operation
- Operations customized with attributes specifying information for the type
- Patterns encoded as operations have a RISE function type

Function application

Function application

rise.apply models function application:
 expects an SSA value with a RISE function type (%zip)
 and arguments to the function (%arow, %bcol)

Function application

rise.apply models function application:
 expects an SSA value with a RISE function type (%zip)
 and arguments to the function (%arow, %bcol)

```
1 func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
   %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
3 %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
4 %f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
      %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
       %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
       %zipped = rise.apply %zip, %arow, %bcol
       %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>>) → !rise.scalar<f32> {
          %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
          %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
         %fst = rise.apply %fstFun, %tuple
12
         %snd = rise.apply %sndFun, %tuple
13
         %result = rise.embed(%fst, %snd) {
14
           %res = mulf %fst, %snd : f32
15
           return %res : f32
16
         } : !rise.scalar<f32>
17
          rise.return %result : !rise.scalar<f32>
18
        %map = rise.mapSeg #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
19
        %multipliedArray = rise.apply %map, %f, %zipped
20
21
       %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
22
         %result = rise.embed(%a, %b) {
23
           %res = addf %a, %b : f32
24
           return %res : f32
         } : !rise.scalar<f32>
         rise.return %result : !rise.scalar<f32>
26
27
28
        %init = rise.literal #rise.lit<0.0>
       %reduce = rise.reduceSeg #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
       %result = rise.apply %reduce, %add, %init, %multipliedArray
31
        rise.return %result : !rise.scalar<f32>
32
33
      %mapB = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
      %result = rise.apply %mapB, %f2, %B
      rise.return %result : !rise.arrav<1024. arrav<1024. scalar<f32>>>
35
36 }
37 %mapA = rise.mapSeg #rise.nat<1024> #rise.arrav<1024, scalar<f32>> #rise.arrav<1024, scalar<f32>>
38 %result = rise.apply %mapA, %f1, %A
39 rise.out %out ← %result
40 return
41 }
```

```
1 func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
2 %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
3 %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
  %f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
     %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
       %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
       %zipped = rise.apply %zip, %arow, %bcol
       %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>>) → !rise.scalar<f32> {
         %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
          %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
11
         %fst = rise.apply %fstFun, %tuple
12
          %snd = rise.apply %sndFun, %tuple
13
          %result = rise.embed(%fst, %snd) {
14
           %res = mulf %fst, %snd : f32
15
           return %res : f32
16
         } : !rise.scalar<f32>
17
          rise.return %result : !rise.scalar<f32>
18
19
        %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
        %multipliedArray = rise.apply %map, %f, %zipped
20
       %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) \rightarrow !rise.scalar<f32>> {
21
22
          %result = rise.embed(%a, %b) {
23
           %res = addf %a, %b : f32
24
           return %res : f32
25
          } : !rise.scalar<f32>
26
          rise.return %result : !rise.scalar<f32>
27
28
       %init = rise.titerat #rise.tit<0.0>
       %reduce = rise.reduceSeq #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
30
       %result = rise.apply %reduce, %add, %init, %multipliedArray
31
       rise.return %result : !rise.scalar<f32>
32
33
     %mapB = rise.mapSeg #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
     %result = rise.apply %mapB, %f2, %B
35
     rise.return %result : !rise.array<1024, array<1024, scalar<f32>>>
36 }
37 %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
38 %result = rise.apply %mapA, %f1, %A
39 rise.out %out ← %result
40 return
41 }
```

```
21 %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32> {
22     %result = rise.embed(%a, %b) {
23         %res = addf %a, %b : f32
24         return %res
25         } : !rise.scalar<f32>
26         rise.return %result : !rise.scalar<f32>
27     }
```

- rise.lambda has an MLIR region of exactly one block
- Arbitrary number of arguments and a result
- rise.lambda associates the region with a RISE function type

- rise.lambda has an MLIR region of exactly one block
- Arbitrary number of arguments and a result
- rise.lambda associates the region with a RISE function type

Embedding of other dialects

```
21 %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32> {
22     %result = rise.embed(%a, %b) {
23         %res = addf %a, %b : f32
24         return %res
25     } : !rise.scalar<f32>
26         rise.return %result : !rise.scalar<f32>
27     }
```

- rise.embed unwraps the operands of type !rise.scalar<t> and
 exposes the values of the wrapped types t inside the region
- The region may contain operations from arbitrary MLIR dialects
- rise.embed returns the result value of !rise.scalar type

Embedding of other dialects

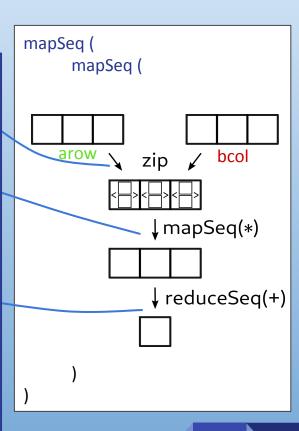
- rise.embed unwraps the operands of type !rise.scalar<t> and
 exposes the values of the wrapped types t inside the region
- The region may contain operations from arbitrary MLIR dialects
- rise.embed returns the result value of !rise.scalar type

RISE dialect by example: Matrix Multiplication

```
1 func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
   %A = rise.in %inA : !rise.array<1024. array<1024. scalar<f32>>>
   %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
   %f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
     %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
       %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
        %zipped = rise.apply %zip, %arow, %bcol
       %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>>) → !rise.scalar<f32> {
         %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
          %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
         %fst = rise.apply %fstFun, %tuple
11
12
         %snd = rise.apply %sndFun, %tuple
13
         %result = rise.embed(%fst, %snd) {
14
           %res = mulf %fst, %snd : f32
15
           return %res : f32
16
         } : !rise.scalar<f32>
          rise.return %result : !rise.scalar<f32>
17
18
        %map = rise.mapSeg #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
19
        %multipliedArray = rise.apply %map, %f, %zipped
20
        %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
21
22
         %result = rise.embed(%a. %b) {
23
           %res = addf %a, %b : f32
24
           return %res : f32
25
         } : !rise.scalar<f32>
          rise.return %result : !rise.scalar<f32>
26
27
28
        %init = rise.literal #rise.lit<0.0>
       %reduce = rise.reduceSeq #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
29
       %result = rise.apply %reduce, %add, %init, %multipliedArray
31
        rise.return %result : !rise.scalar<f32>
32
33
     %mapB = rise.mapSeg #rise.nat<1024> #rise.arrav<1024. scalar<f32>> #rise.arrav<1024. scalar<f32>>
     %result = rise.apply %mapB, %f2, %B
     rise.return %result : !rise.arrav<1024. arrav<1024. scalar<f32>>>
35
36 }
37 %mapA = rise.mapSeg #rise.nat<1024> #rise.array<1024. scalar<f32>> #rise.array<1024. scalar<f32>>
38 %result = rise.apply %mapA, %f1, %A
39 rise.out %out ← %result
40 return
41 }
```

RISE dialect by example: Matrix Multiplication

```
1 func @mm(%out:memref<1024×1024xf32>. %inA:memref<1024×1024xf32>. %inB:memref<1024×1024xf32>) {
   %A = rise.in %inA : !rise.array<1024. array<1024. scalar<f32>>>
   %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
   %f1 = rise.lambda (%arow : !rise.arrav<1024. scalar<f32>>) → !rise.arrav<1024. scalar<f32>>
     %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>>
       %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
6
       %zipped = rise.apply %zip, %arow, %bcol
       %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>) → !rise.scalar<f32>
8
          %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
          %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
11
         %fst = rise.applv %fstFun. %tuple
12
         %snd = rise.apply %sndFun, %tuple
13
          %result = rise.embed(%fst, %snd) {
14
           %res = mulf %fst, %snd : f32
15
           return %res : f32
16
         } : !rise.scalar<f32>
          rise.return %result : !rise.scalar<f32>
17
18
       %map = rise.mapSeg #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
19
20
       %multipliedArray = rise.apply %map, %f, %zipped
21
       %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
22
         %result = rise.embed(%a. %b) {
23
           %res = addf %a, %b : f32
24
           return %res : f32
25
          } : !rise.scalar<f32>
26
          rise.return %result : !rise.scalar<f32>
27
28
       %init = rise.literal #rise.lit<0.0>
       %reduce = rise.reduceSeg #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
29
       %result = rise.apply %reduce, %add, %init, %multipliedArray
30
31
       rise.return %result : !rise.scalar<f32>
32
     %mapB = rise.mapSeg #rise.nat<1024> #rise.arrav<1024, scalar<f32>> #rise.arrav<1024, scalar<f32>>
33
     %result = rise.apply %mapB, %f2, %B
     rise.return %result : !rise.array<1024, array<1024, scalar<f32>>>
35
36
   %mapA = rise.mapSeg #rise.nat<1024> #rise.arrav<1024, scalar<f32>> #rise.arrav<1024, scalar<f32>>
   %result = rise.apply %mapA, %f1, %A
   rise.out %out ← %result
40 return
41 }
```



RISE dialect - modelling λ -calculus + patterns

-> direct correspondence of MLIR dialect with original functional representation

Lowering of the RIS**E** dialect

Lowering of patterns in not context free, e.g. zip influences the code generation of the next pattern

Lowering is divided into two steps:

- 1. Lowering functional to imperative
- 2. Lowering to final target code

Lowering described formally in:

Strategy Preserving Compilation for Parallel Functional Code by Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach https://arxiv.org/abs/1710.08332

```
Functional
           asScalar<sub>4</sub> (join (mapWorkgroup (\lambda zs_1, mapLocal (\lambda zs_2, reduce (\lambda x a. (fst x* snd x) + a) 0 (split 8192 zs_2)) zs_1)
                                             (split 8192 (zip (asVector<sub>4</sub> xs) (asVector<sub>4</sub> ys))))))
                                                                                           RISE Functional Low-Level Program
            parforWorkgroup (N/8192) (joinAcc (N/8192) 64 (asScalarAcc<sub>4</sub> (N/128) out)) (\lambda qid o.
             parforLocal 64 o (\lambda lid o.
                newPrivate num(4) accum.
                  accum.1 := 0:
                  for 2048 (λ i.
                    accum.1 := accum.2 +
                       (fst (idx (split 2048 (idx (split (8192 * 4) (zip (asVector<sub>4</sub> xs) (asVector<sub>4</sub> ys))) qid)) lid) i)) *
                       (snd (idx (split 2048 (idx (split (8192 * 4) (zip (asVector<sub>4</sub> xs) (asVector<sub>4</sub> ys))) gid)) lid) i)) );
                  out := accum.2)
Imperative
                                                                                    RISE Imperative Low-Level Program
                                                      Codegen
           kernel void KERNEL(global float *out. const global float *restrict xs.
                                  const global float *restrict ys, int N) {
              for (int g_id = get_group_id(0); g_id < N / 8192; g_id += get_num_groups(0)) {</pre>
                for (int l_id = get_local_id(0); l_id < 64; l_id += get_local_size(0)) {</pre>
                  float4 accum:
                   accum = (float4)(0.0, 0.0, 0.0, 0.0);
                   for (int i = 0; i < 2048; i += 1) {
                                (vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), xs) *
                                 vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), ys))); }
                   vstore4(accum, ((64 * g_id) + l_id), out); } }
                                                                                                           Generated Code
```

Lowering of the RIS**E** dialect

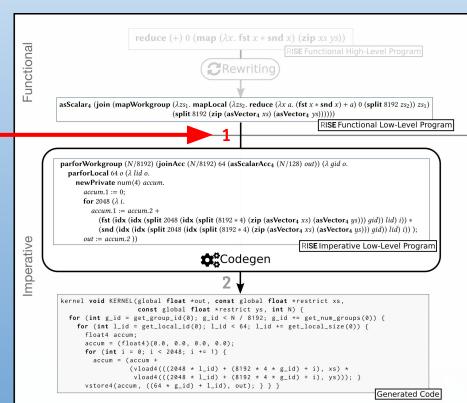
Lowering of patterns in not context free, e.g. zip influences the code generation of the next pattern

Lowering is divided into two steps:

- 1. Lowering functional to imperative
- 2. Lowering to final target code

Lowering described formally in:

Strategy Preserving Compilation for Parallel Functional Code by Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach https://arxiv.org/abs/1710.08332



```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.arrav<1024. scalar<f32>>>
     %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
8
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
       %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
         return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

lowering

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
    %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) \rightarrow f32
11
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
13
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
       %6 = mulf %4. %5 : f32
14
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
    return
19 }
                                                         Loop + RISE Intermediate
```

lowering

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.arrav<1024. scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
8
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
     rise.codegen.assign(%cst0. %1): (f32. f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) <math>\rightarrow f32
11
       \%3 = rise.codegen.idx(\%out, \%c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
13
       %6 = mulf %4. %5 : f32
14
15
       %7 = addf %6. %3 : f32
       rise.codegen.assign(\%7, \%3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                          Loop + RISE Intermediate
```

Start translation from rise.out

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.arrav<1024. scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product. %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
     %c1 = constant 1 : index
    %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) <math>\rightarrow f32
11
       \%3 = rise.codegen.idx(\%out, \%c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
13
       %6 = mulf %4. %5 : f32
14
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                          Loop + RISE Intermediate
```

- Start translation from rise, out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.arrav<1024. scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product. %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
     rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
10
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) <math>\rightarrow f32
11
       \%3 = rise.codegen.idx(\%out, \%c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
13
       \%5 = rise.codegen.snd(\%2) : (f32) \rightarrow f32
14
       %6 = mulf %4. %5 : f32
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

```
\begin{array}{lll} out :=_{\delta_2} \ \mathbf{reduceSeq}(N,\delta_1,\delta_2,F,I,E) & = & C(|I|)_{\delta_2}(\lambda \ init. \\ & C(|E|)_{N.\delta_1}(\lambda \ x. \\ & out :=_{\delta_1} \ init; \\ & \mathbf{for} \ N \ (\lambda i. \ out :=_{\delta_2} \ F(x[i],out));)) \end{array}
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

reduceSeq

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.arrav<1024. scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
     rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
10
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) <math>\rightarrow f32
11
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
13
       \%5 = rise.codegen.snd(\%2) : (f32) \rightarrow f32
14
       %6 = mulf %4. %5 : f32
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                          Loop + RISE Intermediate
```

```
\begin{array}{lll} out :=_{\delta_2} \ \mathbf{reduceSeq}(N,\delta_1,\delta_2,F,I,E) & = & C(I)_{\delta_2}(\lambda \ init. \\ & C(E)_{N.\delta_1}(\lambda \ x. \\ & out :=_{\delta_1} \ init; \\ \hline \mathbf{for} \ N(\lambda i. \ out :=_{\delta_2} \ F(x[i],out));)) \end{array}
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

reduceSeq

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product. %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
10
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) <math>\rightarrow f32
11
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
13
       %6 = mulf %4. %5 : f32
14
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

```
out :=_{\delta_2} \mathbf{reduceSeq}(N, \delta_1, \delta_2, F, \underline{I}, E) = \underbrace{C(\underline{I})_{\delta_2}(\lambda \ init.)}_{C(\underline{E}|_{N.\delta_1}(\lambda \ x.)} \underbrace{c(\underline{E}|_{N.\delta_1}(\lambda \ x.)}_{out :=_{\delta_1} \ init;} \underbrace{for \ N \ (\lambda i. \ out :=_{\delta_2} F(x[i], out));))}
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

literal

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
     %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
10
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) <math>\rightarrow f32
11
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
13
       \%5 = rise.codegen.snd(\%2) : (f32) \rightarrow f32
14
       %6 = mulf %4. %5 : f32
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

```
\begin{array}{lll} out :=_{\delta_2} \ \mathbf{reduceSeq}(N,\delta_1,\delta_2,F,I,\overline{E}) & = & C(I)_{\delta_2}(\lambda \ init. \\ \hline & C(E)_{N.\delta_1}(\lambda \ x.) \\ & out :=_{\delta_1} \ init; \\ & \mathbf{for} \ N \ (\lambda i. \ out :=_{\delta_2} \ F(x[i], out));)) \end{array}
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

zip

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.arrav<1024. scalar<f32>>>
     %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product. %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
     %c1 = constant 1 : index
    %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
10
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) <math>\rightarrow f32
11
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
13
       %6 = mulf %4, %5 : f32
14
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

```
\begin{array}{lll} out :=_{\delta_2} \ \mathbf{reduceSeq}(N,\delta_1,\delta_2,F,I,\overline{E}) & = & C(I)_{\delta_2}(\lambda \ init. \\ \hline & C(E)_{N.\delta_1}(\lambda \ x.) \\ & out :=_{\delta_1} \ init; \\ & \mathbf{for} \ N \ (\lambda i. \ out :=_{\delta_2} \ F(x[i], out));)) \end{array}
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

zip

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.arrav<1024. scalar<f32>>>
     %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product. %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
10
       %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) \rightarrow f32
11
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
13
       %6 = mulf %4, %5 : f32
14
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

```
out :=_{\delta_2} \mathbf{reduceSeq}(N, \delta_1, \delta_2, F, I, \underline{E}) = C(I)_{\delta_2}(\lambda \ init. \\ \underline{C(E)_{N.\delta_1}(\lambda \ x.}) \\ out :=_{\delta_1} init; \\ \mathbf{for} \ N(\lambda i. \ out :=_{\delta_2} F(\underline{x[i]}, out));))
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

zip

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
                        %cst0 = constant 0.000000e+00 : f32
                        %c0 = constant 0 : index
                        %c1024 = constant 1024 : index
                        %c1 = constant 1 : index
                        %0 = rise.codegen.zip(%a. %b)
                             : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
                        %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
                        rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
                        loop.for %i = %c0 to %c1024 step %c1 {
                          \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) \rightarrow f32
lambda{ ... ]
                          %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
                          \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
                   13
                          \%5 = rise.codegen.snd(\%2) : (f32) \rightarrow f32
                   14
                          %6 = mulf %4. %5 : f32
                          %7 = addf %6. %3 : f32
                          rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
                   17
                   18
                        return
                   19 }
                                                                            Loop + RISE Intermediate
```

```
out :=_{\delta_2} \mathbf{reduceSeq}(N, \delta_1, \delta_2, F, I, E)
                                                                           C(I)_{\delta_2}(\lambda init.
                                                                           C(E)_{N,\delta_1}(\lambda x.
                                                                                out :=\delta, init;
                                                                               for N(\lambda i. out :=_{\delta_2} F(x[i], out));))
```

- Start translation from rise, out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
9
      %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst, %snd, %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
         return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
    %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) \rightarrow f32
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
13
       \%5 = rise.codegen.snd(\%2) : (f32) \rightarrow f32
14
       %6 = mulf %4. %5 : f32
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

lambda{ ...]

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
    %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
       %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst. %snd. %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
        return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
    rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) \rightarrow f32
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
       %4 = rise.codegen.fst(%2) : (f32) \rightarrow f32
13
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
       %6 = mulf %4. %5 : f32
14
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

lambda{ ...]

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                                                                                         %cst0 = constant 0.000000e+00 : f32
                                                %b:memref<1024xf32>) {
                                                                                                         %c0 = constant 0 : index
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>
                                                                                                         %c1024 = constant 1024 : index
     %rhs = rise.in %b : !rise.array<1024, scalar<f32>>
                                                                                                         %c1 = constant 1 : index
                                                                                                         %0 = rise.codegen.zip(%a. %b)
     %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
                                                                                                              : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
          #rise.scalar<f32>
                                                                                                         %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     %zipped = rise.apply %zip, %lhs, %rhs
                                                                                                         rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
                                                                                 lowering
                                                                                                         loop.for %i = %c0 to %c1024 step %c1 {
       %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
                                                                                                           \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) \rightarrow f32
       %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
                                                                                 embed{ ...
                                                                                                    11
                                                                                                           %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
10
       %fst = rise.applv %fstFun. %tuple
                                                                                                           \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
11
       %snd = rise.apply %sndFun, %tuple
                                                                                                    13
                                                                                                           %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
12
       %result = rise.embed(%fst. %snd. %acc) {
                                                                                                          \%6 = \text{mulf } \%4. \%5 : f32
                                                                                                    14
13
         %product = mulf %fst, %snd : f32 )
                                                                                                          1\%7 = addf \%6. \%3 : f32
14
         %res = addf %product, %acc : f32
                                                                                                           rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
15
          return res : f32
                                                                                                    17
16
       }: !rise.scalar<f32>
                                                                                                    18
                                                                                                         return
17
       rise.return %result : !rise.scalar<f32>
                                                                                                    19 }
18
                                                                                                                                                            Loop + RISE Intermediate
19
     %init = rise.literal #rise.lit<0.0>
     %reduce = rise.reduceSeg #rise.nat<1024>
          #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
     rise.out %out ← %result
                                                                                          out :=_{\delta_2} \mathbf{reduceSeq}(N, \delta_1, \delta_2, \overline{F}, I, E)
                                                                                                                                      C(I)_{\delta_2}(\lambda init.
23
    return
                                                                                                                                      C(E)_{N,\delta_1}(\lambda x.
24 }
```

- Start translation from rise, out
- Traverse program back to front and lower patterns as specified by their arguments

RISE

out := δ , init;

for $N(\lambda i. out :=_{\delta_2} F(x[i], out));))$

```
1 func @dot(%out:memref<1xf32>, %a:memref<1024xf32>,
                                           %b:memref<1024xf32>) {
     %lhs = rise.in %a : !rise.array<1024, scalar<f32>>>
     %rhs = rise.in %b : !rise.arrav<1024. scalar<f32>>
     %zip = rise.zip #rise.nat<1024> #rise.scalar<f32>
         #rise.scalar<f32>
     %zipped = rise.apply %zip, %lhs, %rhs
     %f = rise.lambda (%tuple, %acc) → !rise.scalar<f32> {
      %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
       %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
10
      %fst = rise.applv %fstFun. %tuple
11
      %snd = rise.apply %sndFun, %tuple
12
      %result = rise.embed(%fst, %snd, %acc) {
13
        %product = mulf %fst, %snd : f32
14
        %res = addf %product, %acc : f32
15
         return res : f32
16
      }: !rise.scalar<f32>
17
      rise.return %result : !rise.scalar<f32>
18
19
    %init = rise.literal #rise.lit<0.0>
    %reduce = rise.reduceSeg #rise.nat<1024>
         #rise.tuple<scalar<f32>. scalar<f32>> #rise.scalar<f32>
    %result = rise.apply %reduce, %f, %init, %zipped
     rise.out %out ← %result
23
    return
24 }
                                                                RISE
```

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) \rightarrow f32
11
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) <math>\rightarrow f32
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
       %6 = mulf %4. %5 : f32
14
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                         Loop + RISE Intermediate
```

```
\begin{array}{lll} out :=_{\delta_2} \ \mathbf{reduceSeq}(N,\delta_1,\delta_2,F,I,E) & = & C(|I|)_{\delta_2}(\lambda \ init. \\ & C(|E|)_{N.\delta_1}(\lambda \ x. \\ & out :=_{\delta_1} \ init; \\ & \mathbf{for} \ N \ (\lambda i. \ out :=_{\delta_2} \ F(x[i],out));)) \end{array}
```

- Start translation from rise.out
- Traverse program back to front and lower patterns as specified by their arguments

lowering

Lowering of the RIS**E** dialect

Lowering of patterns in not context free, e.g. zip influences the code generation of the next pattern

Lowering is divided into two steps:

- 1. Lowering functional to intermediate imperative
- 2. Lowering to final target code

Lowering described formally in:

Strategy Preserving Compilation for Parallel Functional Code by Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach https://arxiv.org/abs/1710.08332

```
Functional
                                                         Rewriting
           asScalar<sub>4</sub> (join (mapWorkgroup (\lambda zs_1, mapLocal (\lambda zs_2, reduce (\lambda x a. (fst x* snd x) + a) 0 (split 8192 zs_2)) zs_1)
                                             (split 8192 (zip (asVector<sub>4</sub> xs) (asVector<sub>4</sub> ys))))))
                                                                                          RISE Functional Low-Level Program
            parforWorkgroup (N/8192) (joinAcc (N/8192) 64 (asScalarAcc<sub>4</sub> (N/128) out)) (\lambda qid o.
             parforLocal 64 o (\lambda lid o.
                newPrivate num(4) accum.
                  accum.1 := 0:
                  for 2048 (λ i.
                    accum.1 := accum.2 +
                      (fst (idx (split 2048 (idx (split (8192 * 4) (zip (asVector<sub>4</sub> xs) (asVector<sub>4</sub> ys))) qid)) lid) i)) *
                      (snd (idx (split 2048 (idx (split (8192 * 4) (zip (asVector<sub>4</sub> xs) (asVector<sub>4</sub> ys))) gid)) lid) i)) );
                  out := accum.2)
perative
                                                                                    RISE Imperative Low-Level Program
                                                     Codegen
느
           kernel void KERNEL(global float *out. const global float *restrict xs.
                                  const global float *restrict ys, int N) {
              for (int g_id = get_group_id(0); g_id < N / 8192; g_id += get_num_groups(0)) {</pre>
                for (int l_id = get_local_id(0); l_id < 64; l_id += get_local_size(0)) {</pre>
                  float4 accum:
                  accum = (float4)(0.0, 0.0, 0.0, 0.0);
                  for (int i = 0; i < 2048; i += 1) {
                                (vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), xs) *
                                 vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), ys))); }
                  vstore4(accum, ((64 * g_id) + l_id), out); } }
                                                                                                          Generated Code
```

```
// map(map(dot))
                          lowering
1 func @mm(%out, %A, %B) {
    %cst0 = constant 0.000000e+00 : f32
    %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
    loop.for %i = %c0 to %c1024 step %c1 {
6
      %0 = rise.codegen.idx(%A. %i)
      %1 = rise.codegen.idx(%out. %i)
8
      loop.for %i = %c0 to %c1024 step %c1 {
        %2 = rise.codegen.idx(%B, %i)
         %3 = rise.codegen.idx(%1, %j)
10
11
         %4 = rise.codegen.zip(%0. %2)
12
        %5 = rise.codegen.idx(%3, %c0)
13
         rise.codegen.assign(%cst0, %5)
14
         loop.for %k = %c0 to %c1024 step %c1 {
15
          \%6 = rise.codegen.idx(\%4. \%k)
16
          %7 = rise.codegen.idx(%3. %c0)
17
          %8 = rise.codegen.fst(%6)
18
          %9 = rise.codegen.snd(%6)
19
          %10 = mulf %8, %9 : f32
20
          %11 = addf %10. %7 : f32
21
           rise.codegen.assign(%11, %7)
22
23
24
    return
26 }
                        Loop + RISE Intermediate
```

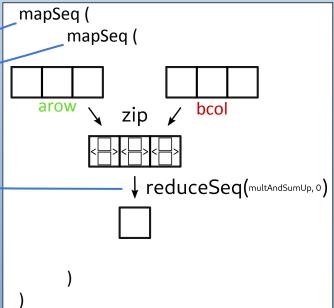
func @mm(%out, %A, %B) {

```
1 func @dot(%out:memref<1xf32>,%a:memref<1024xf32>,%b:memref<1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     %0 = rise.codegen.zip(%a. %b)
          : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
     %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
     rise.codegen.assign(%cst0, %1) : (f32, f32) \rightarrow ()
     loop.for %i = %c0 to %c1024 step %c1 {
10
       \%2 = rise.codegen.idx(\%0, \%i) : (memref<1024xf32>, index) \rightarrow f32
       %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) \rightarrow f32
11
12
       \%4 = rise.codegen.fst(\%2) : (f32) \rightarrow f32
       %5 = rise.codegen.snd(%2) : (f32) \rightarrow f32
       %6 = mulf %4. %5 : f32
15
       %7 = addf %6. %3 : f32
       rise.codegen.assign(%7, %3) : (f32, f32) \rightarrow ()
17
18
     return
19 }
                                                        Loop + RISE Intermediate
```

- Matrix multiply builds on dot -> similar inner structure
- Only differences:
 - different input to rise.codegen.zip
 - accumulating indexed with j and i

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024×f32>, index) <math>\rightarrow memref<1024×f32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) → memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
         %2 = rise.codegen.idx(%B, %j) : (memref<1024×1024×f32>, index) <math>\rightarrow memref<1024×f32>
10
         \%3 = rise.codegen.idx(\%1, \%1) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
         \%4 = rise.codegen.zip(\%0, \%2) : (memref<1024xf32>, memref<1024xf32>) <math>\rightarrow memref<1024xf32>
11
          %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
12
          rise.codegen.assign(%cst0, %5) : (f32, f32) \rightarrow ()
13
          loop.for %k = %c0 to %c1024 step %c1 {
14
15
            \%6 = rise.codegen.idx(\%4, \%k) : (memref<1024xf32>, index) \rightarrow f32
            \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
16
            \%8 = rise.codegen.fst(\%6) : (f32) \rightarrow f32
17
            \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
18
            %10 = mulf %8, %9 : f32
19
20
            %11 = addf %10, %7 : f32
21
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
22
23
24
     return
26 }
                                                                                    Loop + RISE Intermediate
```

```
mapSeq (
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
                                                                                                                              mapSeq (
     %c0 = constant 0 : index
     %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) → memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
                                                                                                                         arow
                                                                                                                                         zip
         %2 = rise.codegen.idx(%B, %j) : (memref<1024×1024×f32>, index) <math>\rightarrow memref<1024×f32>
10
         \%3 = rise.codegen.idx(\%1, \%j) : (memref<1024xf32>, index) <math>\rightarrow memref<1024xf32>
         \%4 = rise.codegen.zip(\%0, \%2) : (memref<1024xf32>, memref<1024xf32>) <math>\rightarrow memref<1024xf32>
11
         \%5 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
12
         rise.codegen.assign(%cst0, %5): (f32, f32) \rightarrow ()
13
14
         loop.for %k = %c0 to %c1024 step %c1 { -
15
            \%6 = rise.codegen.idx(\%4, \%k) : (memref<1024xf32>, index) \rightarrow f33
            \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) \rightarrow f32
16
            \%8 = rise.codegen.fst(\%6) : (f32) \rightarrow f32
17
            \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
18
            %10 = mulf %8, %9 : f32
19
20
            %11 = addf %10, %7 : f32
21
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
22
23
24
     return
26 }
                                                                                   Loop + RISE Intermediate
```



```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
     %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         \%3 = rise.codegen.idx(\%1.\%i) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
         %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0, %5) : (f32, f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
16
           \%6 = rise.codegen.idx(\%4, \%k) : (memref<1024xf32>, index) \rightarrow f32
17
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
18
           \%8 = rise.codegen.fst(\%6) : (f32) \rightarrow f32
19
           \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
20
           %10 = mulf %8. %9 : f32
21
           %11 = addf %10. %7 : f32
22
            rise.codegen.assign(%11. \%7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
                                                                                                       Loop + RISE Intermediate
```

Start codegen from rise.codegen.assign

? = ?

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         \%3 = rise.codegen.idx(\%1.\%i) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
         %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0. %5): (f32. f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
16
           \%6 = rise.codegen.idx(\%4, \%k) : (memref<1024xf32>, index) \rightarrow f32
17
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
18
           \%8 = rise.codegen.fst(\%6) : (f32) \rightarrow f32
19
           \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
20
           %10 = mulf %8, %9 : f32
        \rightarrow %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
```

Loop + RISE Intermediate

? = ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         \%3 = rise.codegen.idx(\%1.\%i) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
         %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0. %5): (f32. f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
16
           \%6 = rise.codegen.idx(\%4, \%k) : (memref<1024xf32>, index) \rightarrow f32
17
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
18
           \%8 = rise.codegen.fst(\%6) : (f32) \rightarrow f32
19
           \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
20
        >> %10 = mulf %8, %9 : f32
        \rightarrow %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
                                                                                                        Loop + RISE Intermediate
```

? = ? * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
         \%4 = rise.codegen.zip(\%0, \%2) : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0, %5) : (f32, f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
           \%6 = rise.codegen.idx(\%4, \%k) : (memref<1024xf32>, index) \rightarrow f32
16
17
            \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
         \gg %8 = rise.codegen.fst(\frac{\%6}{}) : (f32) \rightarrow f32
18
            \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
19
        %10 = mulf %8, %9 : f32
20
        %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
                                                                                                        Loop + RISE Intermediate
```

Path:

fst

? = ? * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references
- build up path

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
         %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0. %5): (f32. f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
        \rightarrow %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) \rightarrow f32
16
17
            \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
         \gg %8 = rise.codegen.fst(%6) : (f32) \rightarrow f32
18
19
            \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
        %10 = mulf %8, %9 : f32
20
        \rightarrow %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
                                                                                                        Loop + RISE Intermediate
```

Path:

idx(%k) fst

? = ?[?] * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references
- build up path

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
     %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
       \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
       loop.for %j = %c0 to %c1024 step %c1 {
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
       \gg%4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0, %5): (f32, f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
        \gg %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) \rightarrow f32
16
17
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
         \gg %8 = rise.codegen.fst(%6) : (f32) \rightarrow f32
18
19
            \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
        %10 = mulf %8, %9 : f32
20
        \rightarrow %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
                                                                                                        Loop + RISE Intermediate
```

Path:

```
idx(%k)

fst

codegen(lhs)
```

? = ?[?] * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references
- build up path

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.0000000e+00 : f32
    %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
     \_%0 = rise.codegen.idx(%A, %i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
8
       loop.for %j = %c0 to %c1024 step %c1 {
9
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
       \%4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0. %5): (f32. f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
        \gg %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) \rightarrow f32
16
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) \rightarrow f32
17
         \gg %8 = rise.codegen.fst(%6) : (f32) \rightarrow f32
18
19
           \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
20
        %10 = mulf %8, %9 : f32
        \rightarrow %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
                                                                                                       Loop + RISE Intermediate
```

Path:

idx(%i) idx(%k)

? = ?[?,?] * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references
- build up path

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
    %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
     \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
8
       loop.for %j = %c0 to %c1024 step %c1 {
9
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
       \%4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0. %5): (f32. f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
        \gg %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) \rightarrow f32
16
17
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
         \gg %8 = rise.codegen.fst(%6) : (f32) \rightarrow f32
18
19
           \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
20
        %10 = mulf %8, %9 : f32
        \rightarrow %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
24
25
     return
27 }
                                                                                                       Loop + RISE Intermediate
```

Path:

idx(%i) idx(%k)

? = A[?,?] * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references
- build up path

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
    %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
     \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
8
       loop.for %j = %c0 to %c1024 step %c1 {
9
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
       \%4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0. %5): (f32. f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
        \gg %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) \rightarrow f32
16
17
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) <math>\rightarrow f32
         \gg %8 = rise.codegen.fst(%6) : (f32) \rightarrow f32
18
19
           \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
20
        %10 = mulf %8, %9 : f32
        \rightarrow %11 = addf %10, %7 : f32
21
22
            rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
25
     return
27 }
                                                                                                       Loop + RISE Intermediate
```

Path:

idx(%i)
idx(%k)

? = A[k,i] * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references
- build up path
- reverse path

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
    %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
     loop.for %i = %c0 to %c1024 step %c1 {
     \%0 = rise.codegen.idx(\%A, \%i) : (memref<1024×1024xf32>, index) \rightarrow memref<1024xf32>
       %1 = rise.codegen.idx(%out, %i) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
8
       loop.for %j = %c0 to %c1024 step %c1 {
9
10
         \%2 = rise.codegen.idx(\%B, \%j) : (memref<1024×1024xf32>, index) <math>\rightarrow memref<1024xf32>
         %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) \rightarrow memref<1024xf32>
11
       \%4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) \rightarrow memref<1024xf32>
12
13
         %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) \rightarrow f32
         rise.codegen.assign(%cst0. %5): (f32. f32) \rightarrow ()
14
15
         loop.for %k = %c0 to %c1024 step %c1 {
        \sim %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) \rightarrow f32
16
           \%7 = rise.codegen.idx(\%3, \%c0) : (memref<1024xf32>, index) \rightarrow f32
17
         \gg %8 = rise.codegen.fst(%6) : (f32) \rightarrow f32
18
           \%9 = rise.codegen.snd(\%6) : (f32) \rightarrow f32
19
                                                           generate() __ %new8 = load %A[%k, %i]
20
        %10 = mulf %8, %9 : f32
        > %11 = addf %10. %7 : f32
21
22
           rise.codegen.assign(%11, %7) : (f32, f32) \rightarrow ()
23
25
     return
27 }
                                                                                                     Loop + RISE Intermediate
```

Path:

idx(%i) idx(%k)

? = A[k,i] * ? + ?

- Start codegen from rise.codegen.assign
- Traverse program back to front following references
- build up path
- reverse path and generate new operations

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>, %B: memref<1024×1024xf32>) {
    %cst0 = constant 0.000000e+00 : f32
    %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
    loop.for %i = %c0 to %c1024 step %c1 {
8
9
      loop.for %j = %c0 to %c1024 step %c1 {
10
11
12
13
14
        store %cst0, %out[%j, %i] : memref<1024×1024xf32>
15
        loop.for %k = %c0 to %c1024 step %c1 {
          %0 = load %A[%k, %i] : memref<1024×1024xf32>
16
          %1 = load %B[%k, %j] : memref<1024×1024xf32>
17
          %2 = load %out[%j, %i] : memref<1024×1024xf32>
18
19
20
          %3 = mulf %0, %1 : f32
          %4 = addf %3, %2 : f32
21
22
          store %4. %out[%i, %i] : memref<1024×1024xf32>
23
24
25
    return
27 }
                                                                                                               Loop
```

67

Lowering RISE to affine

```
func @mm(%out:memref<1024×1024xf32>,%A:memref<1024×1024xf32>,
        %B:memref<1024×1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     affine.for %i = 0 to 1024 {
       affine.for %j = 0 to 1024 {
         affine.store %cst0, %out[%j, %i] : memref<1024×1024xf32>
         affine.for %k = 0 to 1024 {
           %0 = affine.load %arg1[%k, %i] : memref<1024×1024xf32>
12
           %1 = affine.load %arg2[%k, %j] : memref<1024×1024xf32>
13
           %2 = affine.load %arg0[%j, %i] : memref<1024×1024xf32>
14
           %3 = mulf %0, %1 : f32
15
           %4 = addf %3, %2 : f32
16
           affine.store %4, %arg0[%j, %i] : memref<1024×1024xf32>
17
18
19
20
     return
21
                                                               Affine
```

- Similar to the loop lowering presented before
- Lowering target specified using Attributes
 - rise.reduceSeq {to = "affine"}
- We can always use affine loops with our patterns
- Enables usage of polyhedral optimizations

Lowering RISE to affine vs. loop

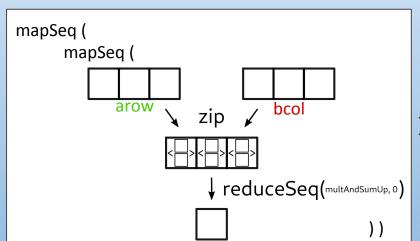
```
func @mm(%out:memref<1024×1024xf32>,%A:memref<1024×1024xf32>,
        %B:memref<1024×1024xf32>) {
     %cst0 = constant 0.000000e+00 : f32
     %c0 = constant 0 : index
     affine.for %i = 0 to 1024 {
       affine.for %i = 0 to 1024 {
         affine.store %cst0. %out[%i, %i] : memref<1024×1024xf32>
10
         affine.for %k = 0 to 1024 {
11
           %0 = affine.load %arg1[%k, %i] : memref<1024×1024xf32>
12
           %1 = affine.load %arg2[%k, %j] : memref<1024×1024xf32>
13
           %2 = affine.load %arg0[%j, %i] : memref<1024×1024xf32>
14
           %3 = mulf %0, %1 : f32
15
           %4 = addf %3, %2 : f32
16
           affine.store %4, %arg0[%j, %i] : memref<1024×1024xf32>
17
18
19
20
     return
21
                                                               Affine
```

```
1 func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>,
         %B: memref<1024×1024xf32>) {
    %cst0 = constant 0.000000e+00 : f32
    %c0 = constant 0 : index
    %c1024 = constant 1024 : index
    %c1 = constant 1 : index
    loop.for %i = %c0 to %c1024 step %c1 {
      loop.for %j = %c0 to %c1024 step %c1 {
        store %cst0, %out[%j, %i] : memref<1024×1024xf32>
10
        loop.for %k = %c0 to %c1024 step %c1 {
11
          %0 = load %A[%k, %i] : memref<1024×1024xf32>
12
          %1 = load %B[%k, %j] : memref<1024×1024xf32>
          %2 = load %out[%j, %i] : memref<1024×1024xf32>
13
14
          %3 = mulf %0. %1 : f32
15
          %4 = addf %3. %2 : f32
16
          store %4, %out[%j, %i] : memref<1024×1024xf32>
17
18
19
20
    return
21 }
                                                                 Loop
```

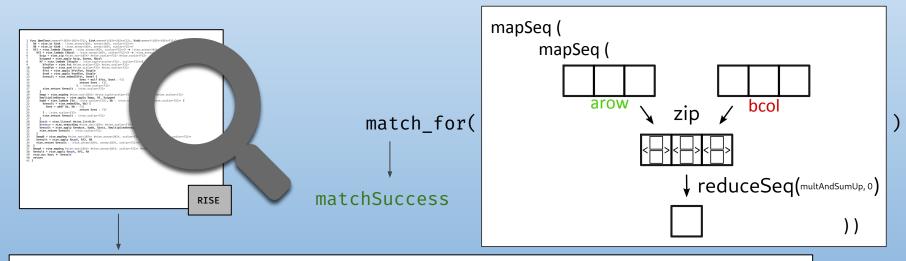
Lowering RISE to library calls



match_for(



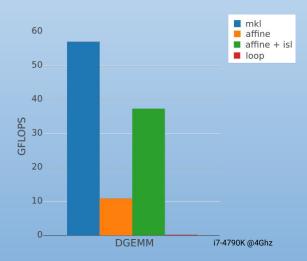
Lowering RISE to library calls



```
func @mm(%out:memref<1024×1024xf32>,%A:memref<1024×1024xf32>,%B:memref<1024×1024xf32>) {
      %CblasRowMajor = constant 101 : i32
      %CblasNoTrans = constant 111 : i32
      M = constant 1024 : i32
      %N = constant 1024 : i32
      %K = constant 1024 : i32
      %LDA = constant 1024 : i32
      %LDB = constant 1024 : i32
      %LDC = constant 1024 : i32
      %alpha = constant 1.0 : f32
      %beta = constant 1.0 : f32
      call @cblas_sgemm_wrapper(%CblasRowMajor, %CblasNoTrans, %CblasNoTrans, %M, %N, %K, %alpha, %A, %LDA, %B, %LDB, %beta, %out, %LDC):
           (i32, \bar{1}32, \bar{1}32, i32, i32, i32, i32, f32, memref<1024×1024xf32>, i32, memref<1024×1024xf32>, i32, f32, memref<1024×1024xf32>, i32) <math>\rightarrow ()
13
      return
14
```

State of our implementation

- Lambda calculus foundations implemented
- Basic set of patterns implemented
- Basic lowering system implemented
- Lowering to Library Calls quite ad-hoc
- Preliminary experiments shows performance as expected



RISE in MLIR: Summary

Today

- Lambda-Calculus + Composable Patterns in MLIR
- Ability to embed arbitrary code from other dialects
- Different lowering approaches

Future directions:

- Explore other lowering approaches (e.g. GPU, OpenMP dialects)
- Introduce symbolic sizes via dependent types
- Rewriting enables composable and reusable high-level transformations
 New language for controlling rewriting (ELEVATE https://elevate-lang.org)

COM The OF CHAPTER

RISE in MLIR

A functional Pattern-based Dialect

We are Open Source!

https://rise-lang.org/mlir

https://github.com/rise-lang/mlir

Martin Lücke | Michel Steuwer | Aaron Smith



