RISE in MLIR

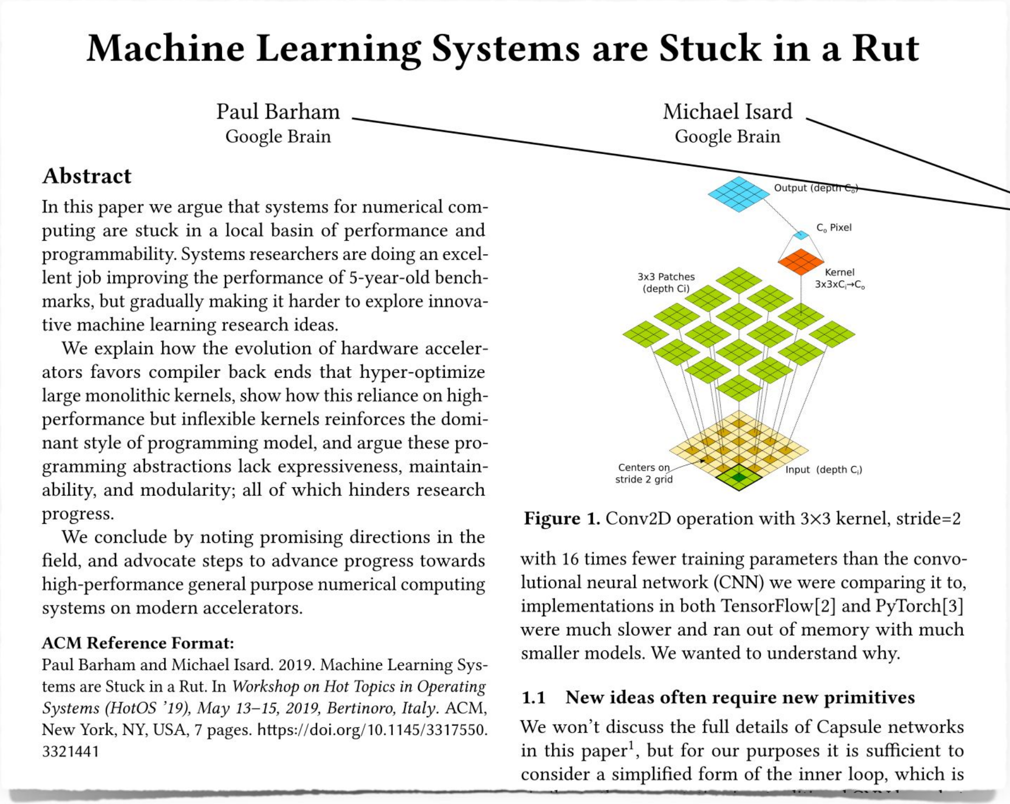
A functional Pattern-based Dialect

Martin Lücke | Michel Steuwer | Aaron Smith

THE UNIVERSITY of EDINBURGH | University of Glasgow

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

Why RISE?



Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
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 paper[1], but for our purposes it is sufficient to consider a simplified form of the inner loop, which is

[HotOS'19]

Original authors of TensorFlow

Why RISE?

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
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

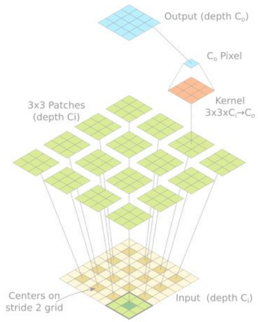


Figure 1. Conv2D operation with 3×3 kernel, stride=2

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 paper[1], 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

Michael Isard
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

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

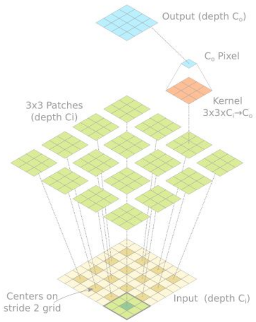


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convo-

smaller models. We wanted to understand why.

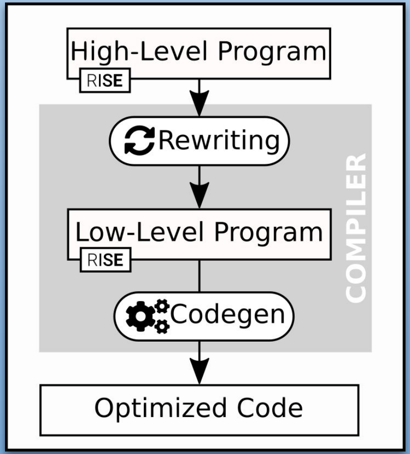
1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper[1], but for our purposes it is sufficient to consider a simplified form of the inner loop, which is

We should aim for more principled higher level intermediate representations

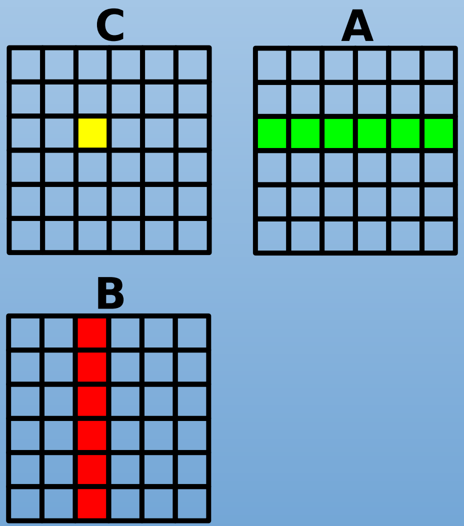
RISE - A functional pattern-based data-parallel language

- 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



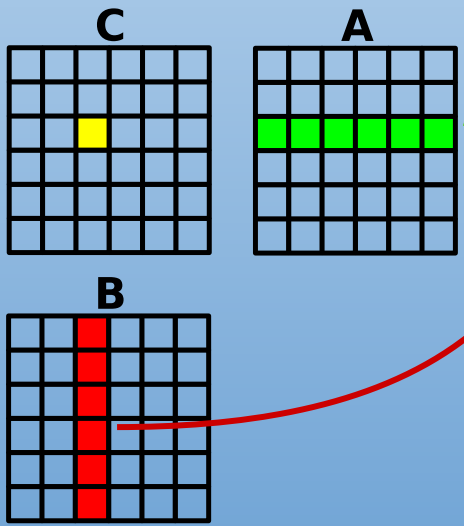
RISE by example: Matrix Multiplication

```
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

```
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) )) )) ))
```

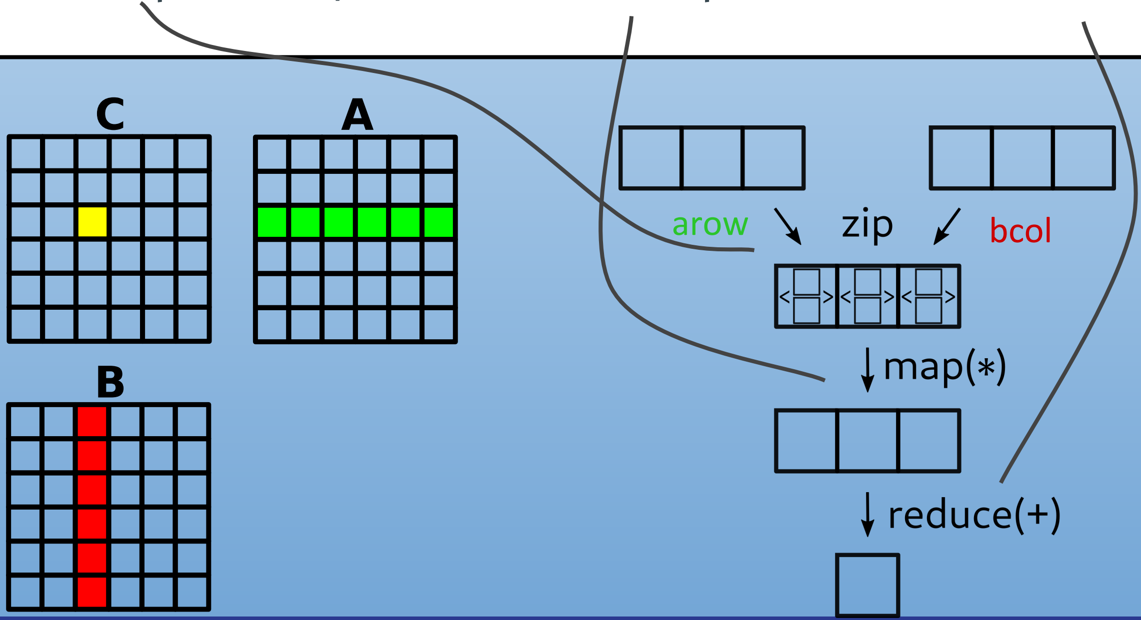


dot product computation:

$$\sum arow_i * bcol_i$$

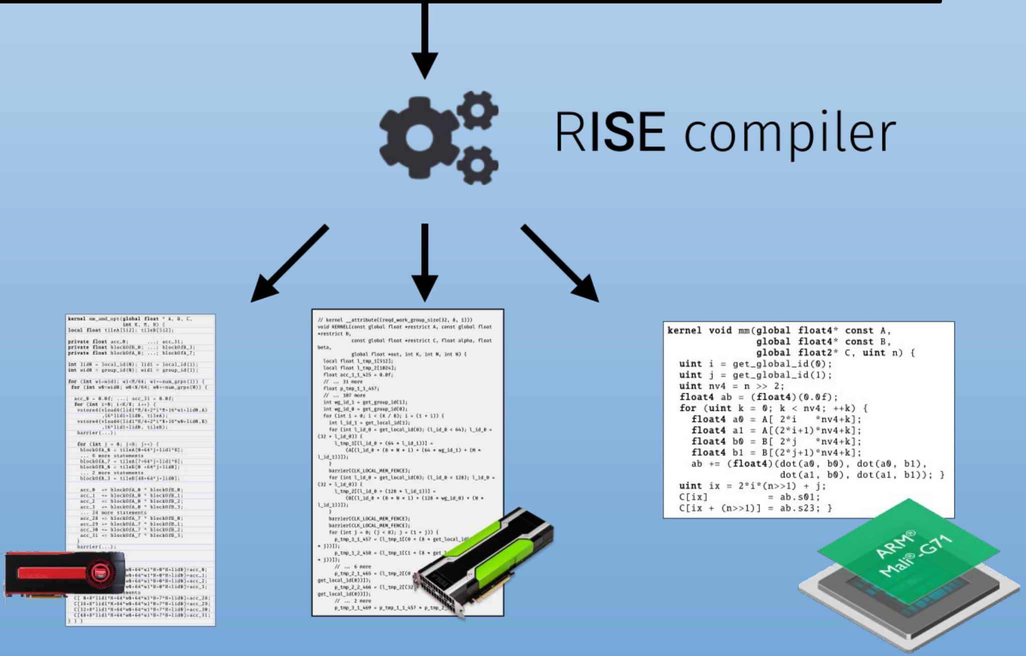
RISE by example: Matrix Multiplication

```
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) )) )) ))
```



Optimization choices for Matrix Multiplication

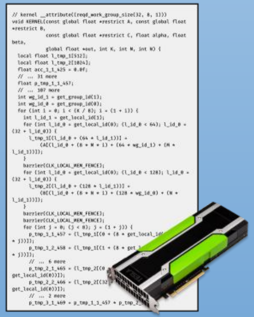
```
fun(A : N.K.float ⇒ fun(B : K.M.float ⇒
  A ▷ map(fun(arow ⇒
    B ▷ transpose ▷ map(fun(bcol ⇒
      zip(arow, bcol) ▷ map(*) ▷ reduce(+, 0) )) )) ))
```



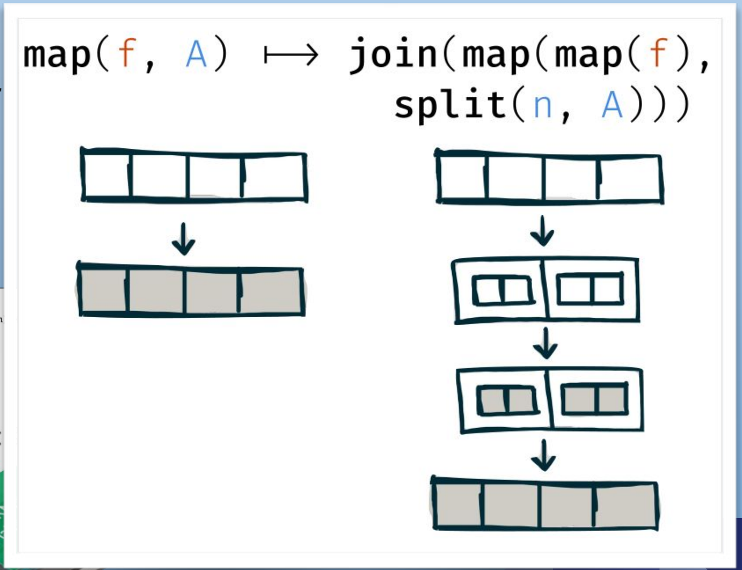
Optimization choices for Matrix Multiplication

```
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 compiler



Rewrite Rules



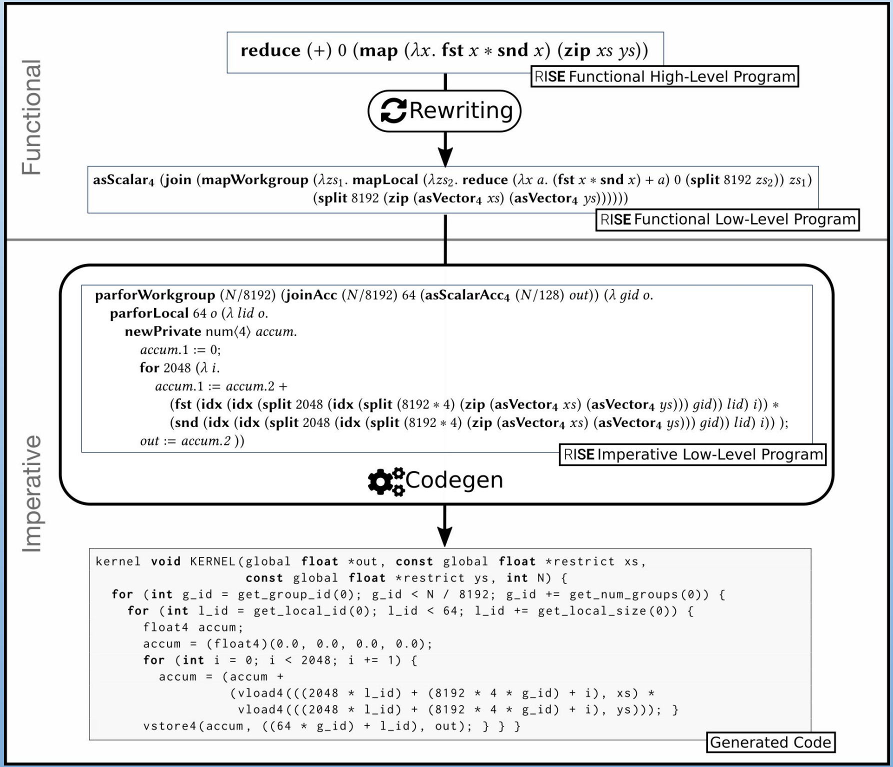
RISE Compilation

Compilation is divided into two steps:

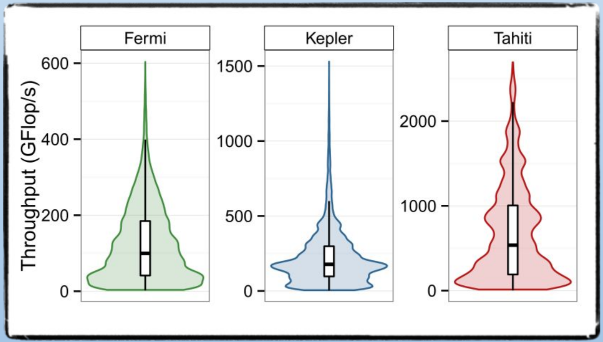
1. Translation from functional to imperative low level program
2. Produce generated code

Compilation described formally in:

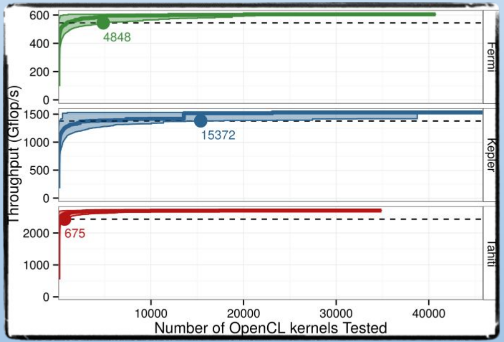
Strategy Preserving Compilation for Parallel Functional Code *by Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach.*
https://arxiv.org/abs/1710.08332



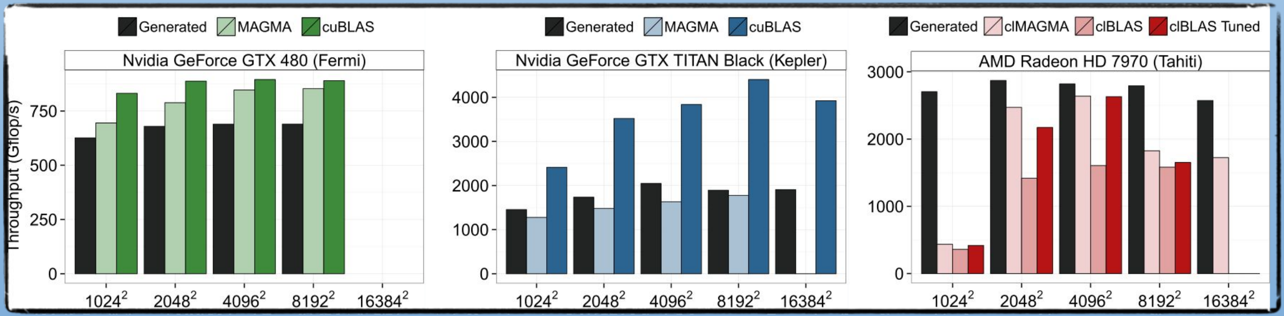
High Performance Results for Matrix Multiplication



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
    - Targeting GPUs
    - ...
    - Performing polyhedral optimizations
    - 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
  - Targeting GPUs
  - ...
  - Performing polyhedral optimizations
  - 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

```
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
        %snd = rise.apply %sndFun, %tuple
        %result = rise.embed(%fst, %snd) {
          %res = mulf %fst, %snd : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
      %multipliedArray = rise.apply %map, %f, %zipped
      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
        %result = rise.embed(%a, %b) {
          %res = addf %a, %b : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %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
      rise.return %result : !rise.scalar<f32>
    }
    %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.array<1024, array<1024, scalar<f32>>>
  }
  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
  %result = rise.apply %mapA, %f1, %A
  rise.out %out ← %result
  return
}
```

RISE dialect by example: Matrix Multiplication

```
func @mm(%out:memref<1024x1024xf32>, %inA:memref<1024x1024xf32>, %inB:memref<1024x1024xf32>) {
  %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
        %snd = rise.apply %sndFun, %tuple
        %result = rise.embed(%fst, %snd) {
          %res = mulf %fst, %snd : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
      %multipliedArray = rise.apply %map, %f, %zipped
      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
        %result = rise.embed(%a, %b) {
          %res = addf %a, %b : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %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
      rise.return %result : !rise.scalar<f32>
    }
    %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.array<1024, array<1024, scalar<f32>>>
  }
  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
  %result = rise.apply %mapA, %f1, %A
  rise.out %out ← %result
  return
}
```

Data Types

Function Types

Types

Data Types

```
%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>>) → scalar<f32>
// %f : !rise.fun<tuple<scalar<f32>, scalar<f32>> → scalar<f32>>
```

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

Patterns

```
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
       %snd = rise.apply %sndFun, %tuple
       %result = rise.embed(%fst, %snd) {
         %res = mulf %fst, %snd : f32
         return %res : f32
       } : !rise.scalar<f32>
       rise.return %result : !rise.scalar<f32>
     }
     %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
     %multipliedArray = rise.apply %map, %f, %zipped
     %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
       %result = rise.embed(%a, %b) {
         %res = addf %a, %b : f32
         return %res : f32
       } : !rise.scalar<f32>
       rise.return %result : !rise.scalar<f32>
     }
     %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
     rise.return %result : !rise.scalar<f32>
   }
   %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.array<1024, array<1024, scalar<f32>>>
 }
 %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
 %result = rise.apply %mapA, %f1, %A
 rise.out %out ← %result
 return
}
```

Patterns

```
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
        %snd = rise.apply %sndFun, %tuple
        %result = rise.embed(%fst, %snd) {
          %res = mulf %fst, %snd : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
      %multipliedArray = rise.apply %map, %f, %zipped
      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
        %result = rise.embed(%a, %b) {
          %res = addf %a, %b : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %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
      rise.return %result : !rise.scalar<f32>
    }
    %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.array<1024, array<1024, scalar<f32>>>
  }
  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
  %result = rise.apply %mapA, %f1, %A
  rise.out %out ← %result
  return
}
```

Patterns: zip

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

Patterns: zip

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

// type: () → !rise.fun<array<1024, scalar<f32>> →
                          fun<array<1024, scalar<f32>> →
                              array<1024, tuple<scalar<f32>, scalar<f32>>> >>

// type: () →
```

- 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

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

Function application

```
%f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) →
                                               array<1024, scalar<f32>> {
   %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) →
                                               array<1024, scalar<f32>> {
         %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
         %zipped = rise.apply %zip, %arow, %bcol
```

- `rise.apply` models function application:

  expects an SSA value with a RISE `function type` (`%zip`)

  and arguments to the function (%arow, %bcol)

Function application

```
%f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) →
                                           array<1024, scalar<f32>> {
   %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) →
                                           array<1024, scalar<f32>> {
        %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
        %zipped = rise.apply %zip, %arow, %bcol

//type: (%zip.type, array<1024, scalar<f32>>, array<1024, scalar<f32>>) →
                                           array<1024,tuple<scalar<f32>,scalar<f32>>>
```

- `rise.apply` models function application:

  expects an SSA value with a RISE `function type` (`%zip`)

  and arguments to the function (%arow, %bcol)

Function abstraction, aka λ-expressions

```
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
        %snd = rise.apply %sndFun, %tuple
        %result = rise.embed(%fst, %snd) {
          %res = mulf %fst, %snd : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
      %multipliedArray = rise.apply %map, %f, %zipped
      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
        %result = rise.embed(%a, %b) {
          %res = addf %a, %b : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %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
      rise.return %result : !rise.scalar<f32>
    }
    %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.array<1024, array<1024, scalar<f32>>>
  }
  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
  %result = rise.apply %mapA, %f1, %A
  rise.out %out ← %result
  return
}
```

Function abstraction, aka λ-expressions

```
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
        %snd = rise.apply %sndFun, %tuple
        %result = rise.embed(%fst, %snd) {
          %res = mulf %fst, %snd : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
      %multipliedArray = rise.apply %map, %f, %zipped
      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
        %result = rise.embed(%a, %b) {
          %res = addf %a, %b : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %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
      rise.return %result : !rise.scalar<f32>
    }
    %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.array<1024, array<1024, scalar<f32>>>
  }
  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
  %result = rise.apply %mapA, %f1, %A
  rise.out %out ← %result
  return
}
```

Function abstraction, aka λ-expressions

```
%add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32> {

  %result = rise.embed(%a, %b) {

    %res = addf %a, %b : f32
    return %res
  } : !rise.scalar<f32>
  rise.return %result : !rise.scalar<f32>
}
```

- `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`

Function abstraction, aka λ-expressions

```
%add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) -> !rise.scalar<f32> {

  %result = rise.embed(%a, %b) {

    %res = addf %a, %b : f32
    return %res
  } : !rise.scalar<f32>
  rise.return %result : !rise.scalar<f32>
} // type: !rise.fun<scalar<f32> -> fun<scalar<f32> -> scalar<f32>>>
```

- `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

```
%add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32> {

  %result = rise.embed(%a, %b) {

    %res = addf %a, %b : f32
    return %res
  } : !rise.scalar<f32>
  rise.return %result : !rise.scalar<f32>
}
```

- `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

```
%add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32> {
  // %a, %b : !rise.scalar<f32>
  %result = rise.embed(%a, %b) {
    // %a, %b : f32
    %res = addf %a, %b : f32
    return %res
  } : !rise.scalar<f32>
  rise.return %result : !rise.scalar<f32>
}
```

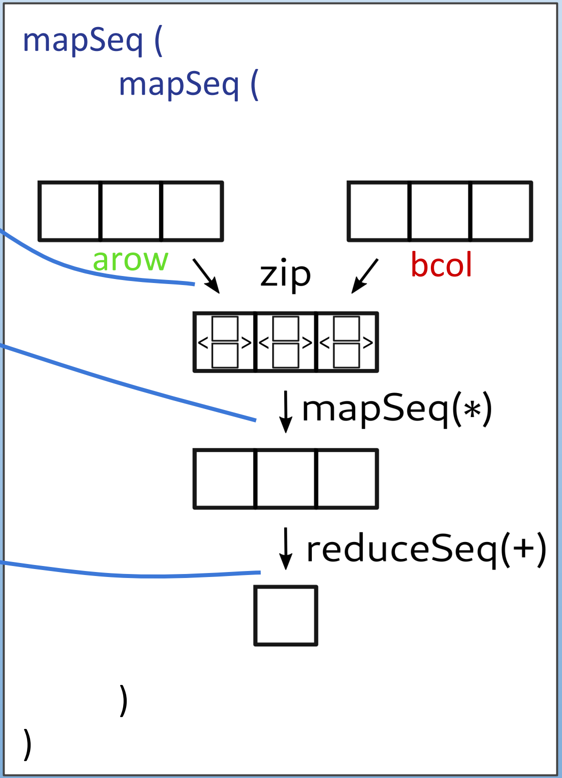
- `rise.embed` unwraps the operands of type `!rise.scalar<t>` and exposes the values of the wrapped types `t` inside the region
- The region may contain operations from arbitrary MLIR dialects
- `rise.embed` returns the result value of `!rise.scalar` type

RISE dialect by example: Matrix Multiplication

```
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
        %snd = rise.apply %sndFun, %tuple
        %result = rise.embed(%fst, %snd) {
          %res = mulf %fst, %snd : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
      %multipliedArray = rise.apply %map, %f, %zipped
      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
        %result = rise.embed(%a, %b) {
          %res = addf %a, %b : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %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
      rise.return %result : !rise.scalar<f32>
    }
    %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.array<1024, array<1024, scalar<f32>>>
  }
  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
  %result = rise.apply %mapA, %f1, %A
  rise.out %out ← %result
  return
}
```

RISE dialect by example: Matrix Multiplication

```
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
        %snd = rise.apply %sndFun, %tuple
        %result = rise.embed(%fst, %snd) {
          %res = mulf %fst, %snd : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
      %multipliedArray = rise.apply %map, %f, %zipped
      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32>> {
        %result = rise.embed(%a, %b) {
          %res = addf %a, %b : f32
          return %res : f32
        } : !rise.scalar<f32>
        rise.return %result : !rise.scalar<f32>
      }
      %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
      rise.return %result : !rise.scalar<f32>
    }
    %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.array<1024, array<1024, scalar<f32>>>
  }
  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
  %result = rise.apply %mapA, %f1, %A
  rise.out %out ← %result
  return
}
```



RISE dialect - modelling λ-calculus + patterns

Core λ-calculus: `rise.lambda`, `rise.apply`, *`rise.let`*,
`!rise.array`, `!rise.tuple`, `!rise.fun`

Patterns: `rise.zip`, `rise.mapSeq`, `rise.reduceSeq`, *`rise.mapPar`*, *`rise.reducePar`*,
`rise.literal` `rise.tuple`, `rise.fst`, `rise.snd`,
`rise.split`, *`rise.join`*, *`rise.transpose`*, *...*

Interoperability: `rise.embed`, `rise.in`, `rise.out`

-> direct correspondence of MLIR dialect with original functional representation

`italic → not implemented yet`

Lowering of the RISE dialect

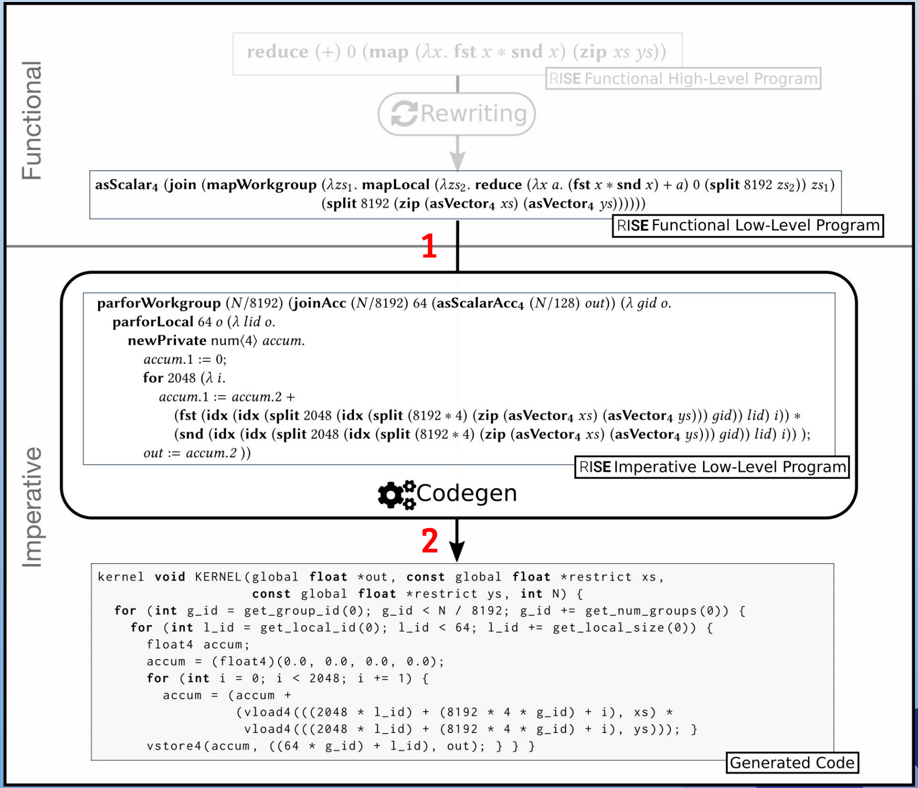
Lowering of patterns in not context free, e.g. zip influences the code generation of the next pattern

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



Lowering of the RISE dialect

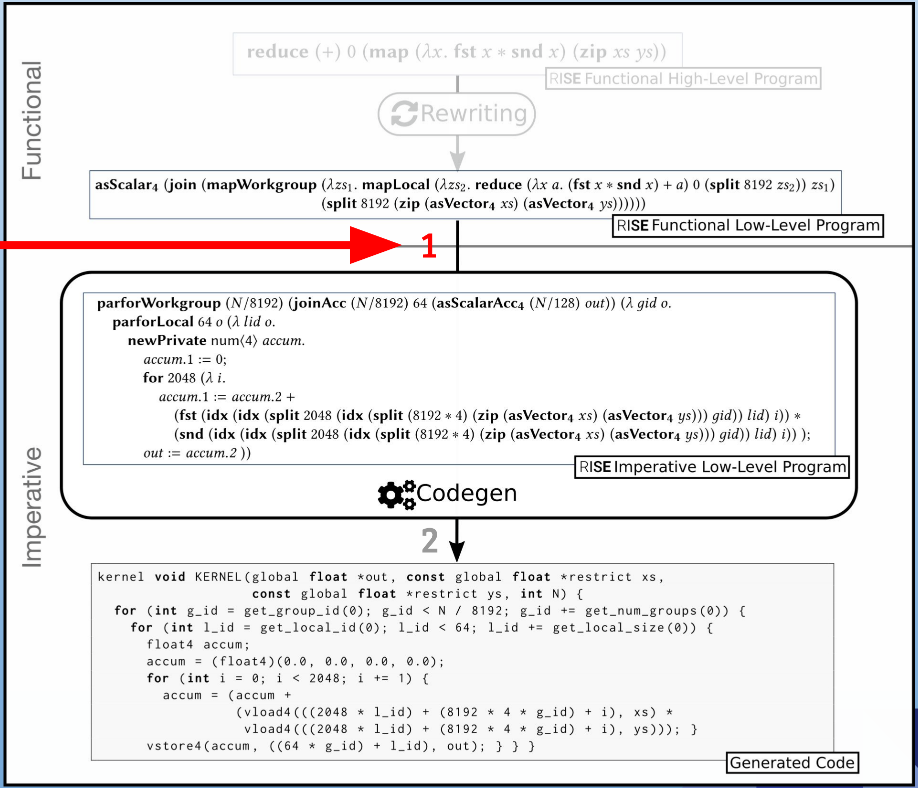
Lowering of patterns in not context free, e.g. zip influences the code generation of the next pattern

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



Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering →

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

- Start translation from `rise.out`

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering →

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering → reduceSeq

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

$$
\begin{aligned}
out :=_{\delta_2} \textbf{reduceSeq}(N, \delta_1, \delta_2, F, I, E) \quad = \quad & C(\!|I|\!)_{\delta_2}(\lambda\ init. \\
& C(\!|E|\!)_{N.\delta_1}(\lambda\ x. \\
& \quad out :=_{\delta_1} init; \\
& \quad \textbf{for}\ N\ (\lambda i.\ out :=_{\delta_2} F(x[i], out));))
\end{aligned}
$$

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering

reduceSeq

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering → literal

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

$$out :=_{\delta_2} \textbf{reduceSeq}(N, \delta_1, \delta_2, F, \boxed{I}, E) \quad = \quad \begin{array}{l} \boxed{C(\!|I|\!)_{\delta_2}(\lambda\, init.} \\ C(\!|E|\!)_{N.\delta_1}(\lambda\, x. \\ \quad \boxed{out :=_{\delta_1} init;} \\ \quad \textbf{for}\ N\ (\lambda i.\ out :=_{\delta_2} F(x[i], out));)) \end{array}$$

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering → zip

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
      #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering → zip

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering → zip

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

$\xrightarrow[\texttt{lambda\{ ... \}}]{\texttt{lowering}}$

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

$$
\begin{aligned}
out :=_{\delta_2} \textbf{reduceSeq}(N, \delta_1, \delta_2, \boxed{F}, I, E) \quad = \quad & C(\!|I|\!)_{\delta_2}(\lambda\ init. \\
& C(\!|E|\!)_{N.\delta_1}(\lambda\ x. \\
& \quad out :=_{\delta_1} init; \\
& \quad \textbf{for}\ N\ \boxed{(\lambda i.\ out :=_{\delta_2} F(x[i], out))};))
\end{aligned}
$$

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering

lambda{ … }

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering

lambda{ … }

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

$$
\begin{aligned}
out :=_{\delta_2} \textbf{reduceSeq}(N, \delta_1, \delta_2, \boxed{F}, I, E) \quad = \quad & C(\!|I|\!)_{\delta_2}(\lambda\, init. \\
& C(\!|E|\!)_{N.\delta_1}(\lambda\, x. \\
& \quad out :=_{\delta_1} init; \\
& \quad \textbf{for } N \boxed{(\lambda i.\ out :=_{\delta_2} F(x[i], out))};))
\end{aligned}
$$

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering

embed{ … }

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

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

IR trimmed for brevity

Lowering to rise.codegen: dot

```
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>
    %fst = rise.apply %fstFun, %tuple
    %snd = rise.apply %sndFun, %tuple
    %result = rise.embed(%fst, %snd, %acc) {
      %product = mulf %fst, %snd : f32
      %res = addf %product, %acc : f32
      return res : f32
    } : !rise.scalar<f32>
    rise.return %result : !rise.scalar<f32>
  }
  %init = rise.literal #rise.lit<0.0>
  %reduce = rise.reduceSeq #rise.nat<1024>
       #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
  %result = rise.apply %reduce, %f, %init, %zipped
  rise.out %out ← %result
  return
}
```

RISE

lowering →

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

Loop + RISE Intermediate

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

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

IR trimmed for brevity

Lowering of the RISE 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

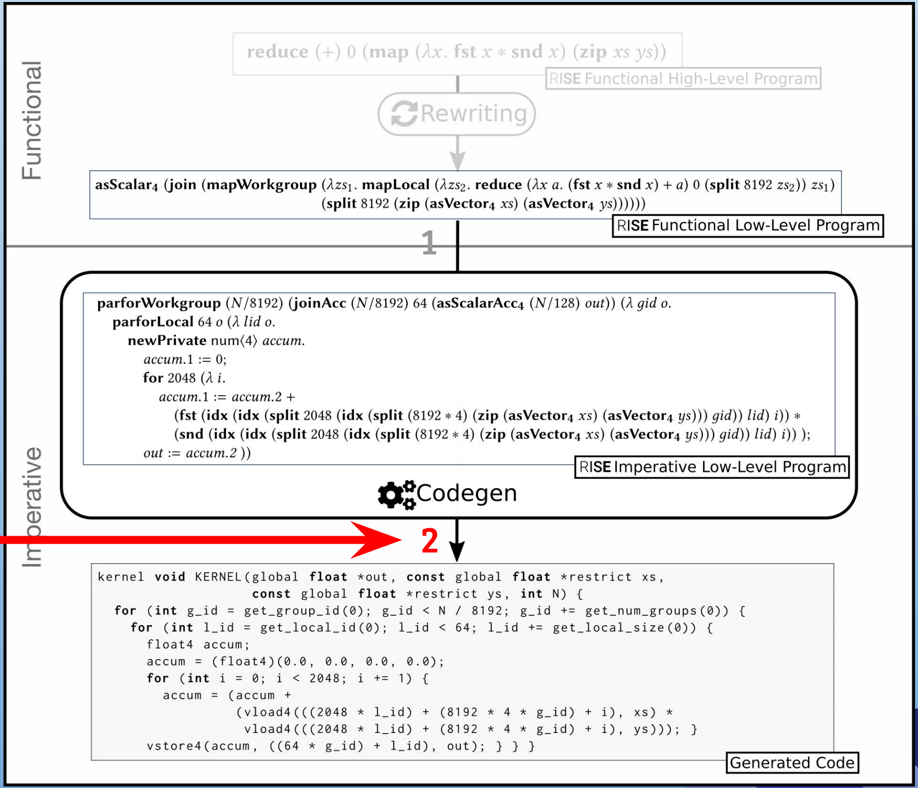
2. Lowering to final target code

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



Lowering rise.codegen to loop: Matrix Multiplication

```
func @mm(%out, %A, %B) {
  // map(map(dot))
}
```

lowering

```
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 {
    %0 = rise.codegen.idx(%A, %i)
    %1 = rise.codegen.idx(%out, %i)
    loop.for %j = %c0 to %c1024 step %c1 {
      %2 = rise.codegen.idx(%B, %j)
      %3 = rise.codegen.idx(%1, %j)
      %4 = rise.codegen.zip(%0, %2)
      %5 = rise.codegen.idx(%3, %c0)
      rise.codegen.assign(%cst0, %5)
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k)
        %7 = rise.codegen.idx(%3, %c0)
        %8 = rise.codegen.fst(%6)
        %9 = rise.codegen.snd(%6)
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7)
      }
    }
  }
  return
}
```

Loop + RISE Intermediate

```
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>) → memref<1024xf32>
  %1 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
  rise.codegen.assign(%cst0, %1) : (f32, f32) → ()
  loop.for %i = %c0 to %c1024 step %c1 {
    %2 = rise.codegen.idx(%0, %i) : (memref<1024xf32>, index) → f32
    %3 = rise.codegen.idx(%out, %c0) : (memref<1xf32>, index) → f32
    %4 = rise.codegen.fst(%2) : (f32) → f32
    %5 = rise.codegen.snd(%2) : (f32) → f32
    %6 = mulf %4, %5 : f32
    %7 = addf %6, %3 : f32
    rise.codegen.assign(%7, %3) : (f32, f32) → ()
  }
  return
}
```

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

IR trimmed for brevity

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

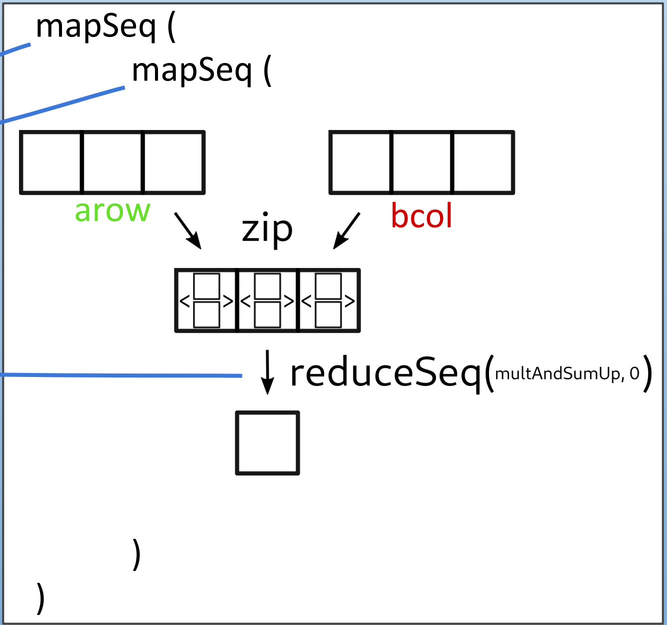
Loop + RISE Intermediate

IR trimmed for brevity

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

Loop + RISE Intermediate



IR trimmed for brevity

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

Loop + RISE Intermediate

```
? = ?
```

- Start codegen from `rise.codegen.assign`

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

Loop + RISE Intermediate

```
? = ? + ?
```

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

```
Loop + RISE Intermediate
```

```
? = ? * ? + ?
```

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

Loop + RISE Intermediate

Path:

```
fst
```

```
? = ? * ? + ?
```

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

Loop + RISE Intermediate

Path:

```
idx(%k)
fst
```

```
? = ?[?] * ? + ?
```

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

Loop + RISE Intermediate

Path:

```
idx(%i)
idx(%k)
```

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

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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) → memref<1024xf32>
    %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<1024×1024xf32>, index) → memref<1024xf32>
      %3 = rise.codegen.idx(%1, %j) : (memref<1024xf32>, index) → memref<1024xf32>
      %4 = rise.codegen.zip(%0, %2) : (memref<1024xf32>, memref<1024xf32>) → memref<1024xf32>
      %5 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
      rise.codegen.assign(%cst0, %5) : (f32, f32) → ()
      loop.for %k = %c0 to %c1024 step %c1 {
        %6 = rise.codegen.idx(%4, %k) : (memref<1024xf32>, index) → f32
        %7 = rise.codegen.idx(%3, %c0) : (memref<1024xf32>, index) → f32
        %8 = rise.codegen.fst(%6) : (f32) → f32
        %9 = rise.codegen.snd(%6) : (f32) → f32
        %10 = mulf %8, %9 : f32
        %11 = addf %10, %7 : f32
        rise.codegen.assign(%11, %7) : (f32, f32) → ()
      }
    }
  }
  return
}
```

generate()

%new8 = load %A[%k, %i]

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

Lowering rise.codegen to loop: Matrix Multiplication

```
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>
      loop.for %k = %c0 to %c1024 step %c1 {
        %0 = load %A[%k, %i] : memref<1024×1024xf32>
        %1 = load %B[%k, %j] : memref<1024×1024xf32>
        %2 = load %out[%j, %i] : memref<1024×1024xf32>

        %3 = mulf %0, %1 : f32
        %4 = addf %3, %2 : f32
        store %4, %out[%j, %i] : memref<1024×1024xf32>
      }
    }
  }
  return
}
```

Loop

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>
         %1 = affine.load %arg2[%k, %j] : memref<1024×1024xf32>
         %2 = affine.load %arg0[%j, %i] : memref<1024×1024xf32>
         %3 = mulf %0, %1 : f32
         %4 = addf %3, %2 : f32
         affine.store %4, %arg0[%j, %i] : memref<1024×1024xf32>
       }
     }
   }
   return
}
```

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

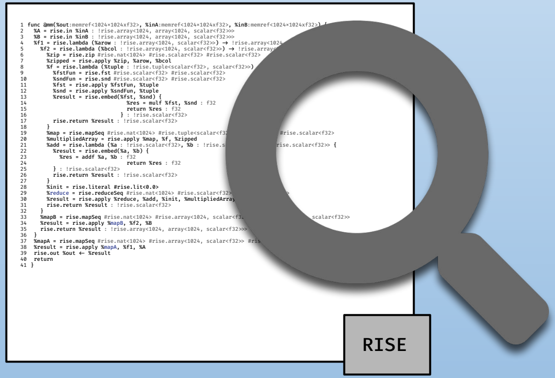
```
func @mm(%out:memref<1024×1024xf32>,%A:memref<1024×1024xf32>,
    %B:memref<1024×1024xf32>) {
  %cst0 = constant 0.000000e+00 : f32
  %c0 = constant 0 : index


  affine.for %i = 0 to 1024 {
    affine.for %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>
        %1 = affine.load %arg2[%k, %j] : memref<1024×1024xf32>
        %2 = affine.load %arg0[%j, %i] : memref<1024×1024xf32>
        %3 = mulf %0, %1 : f32
        %4 = addf %3, %2 : f32
        affine.store %4, %arg0[%j, %i] : memref<1024×1024xf32>
      }
    }
  }
  return
}
```

Affine

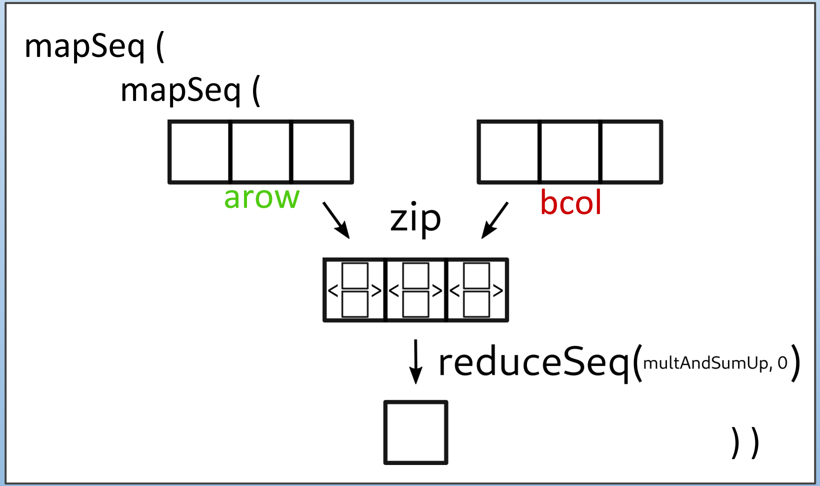
```
func @mm(%out: memref<1024×1024xf32>, %A: memref<1024×1024xf32>,
         %B: memref<1024×1024xf32>) {
  %cst0 = constant 0.000000e+00 : f32
  %c0 = constant 0 : index
  %c1024 = constant 1024 : index
  %c1 = constant 1 : index
  loop.for %i = %c0 to %c1024 step %c1 {
    loop.for %j = %c0 to %c1024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

Loop

Lowering RISE to library calls

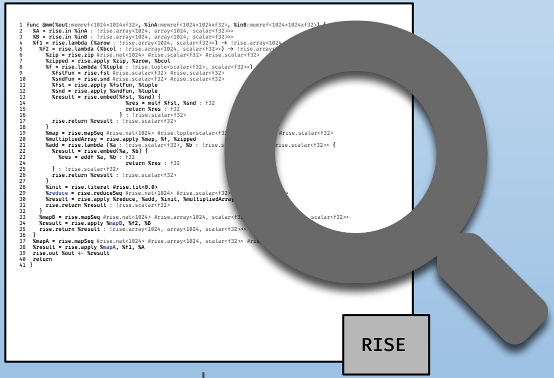


match_for(



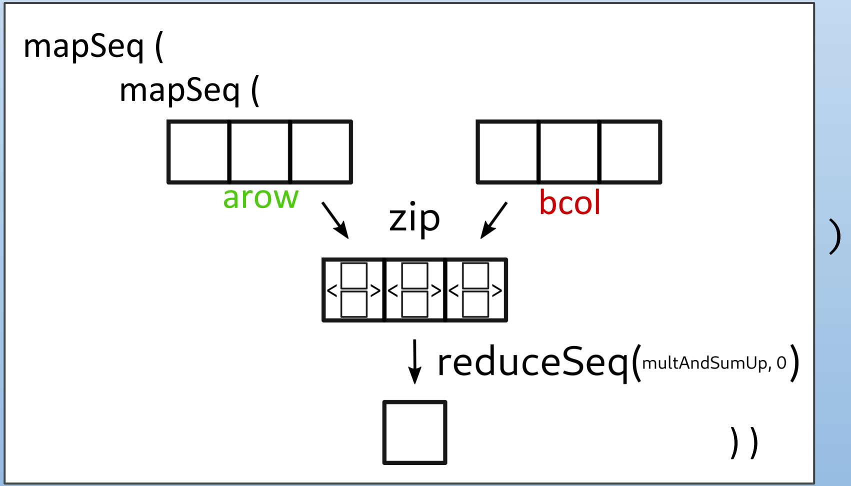
)

Lowering RISE to library calls



match_for(

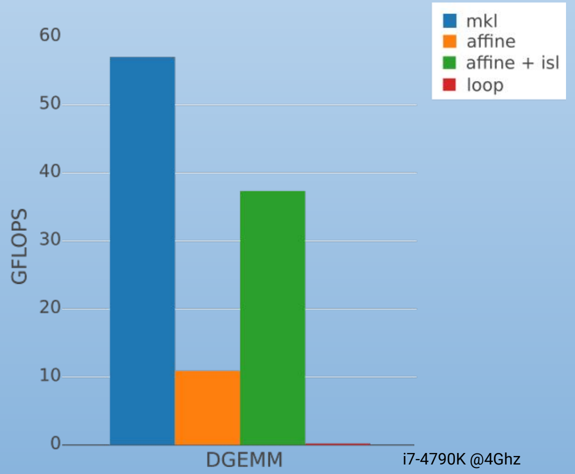
matchSuccess




```
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, i32, i32, i32, i32, i32, f32, memref<1024×1024xf32>, i32, memref<1024×1024xf32>, i32, f32, memref<1024×1024xf32>, i32) → ()
  return
}
```

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



i7-4790K @4Ghz

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)

Fork me on GitHub

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

THE UNIVERSITY of EDINBURGH | University of Glasgow