RISE in MLIR

A functional Pattern-based Dialect

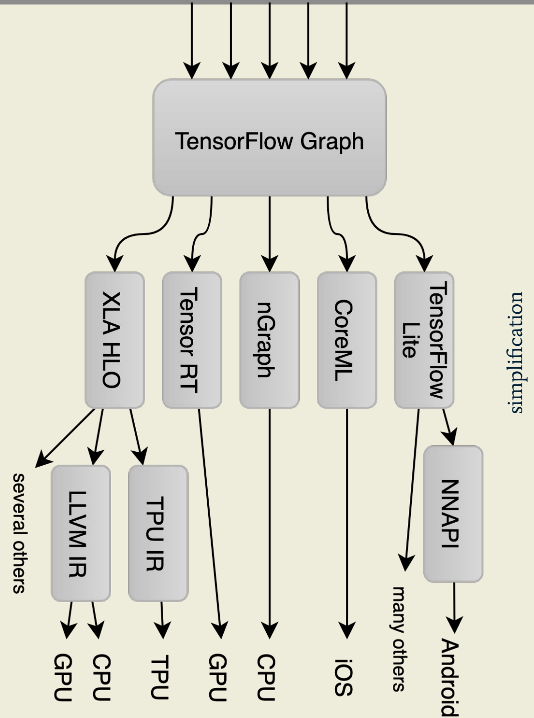
Martin Lücke | Michel Steuwer | Aaron Smith

THE UNIVERSITY of EDINBURGH | University of Glasgow

Machine Learning Systems are stuck in a Rut

Paul Barham and Michael Isard [HotOS2019]

- Currently much focus on optimizing 5 year old ML benchmarks
- New ideas in ML often require new primitives that are hard to compile and optimize for the zoo of modern hardware



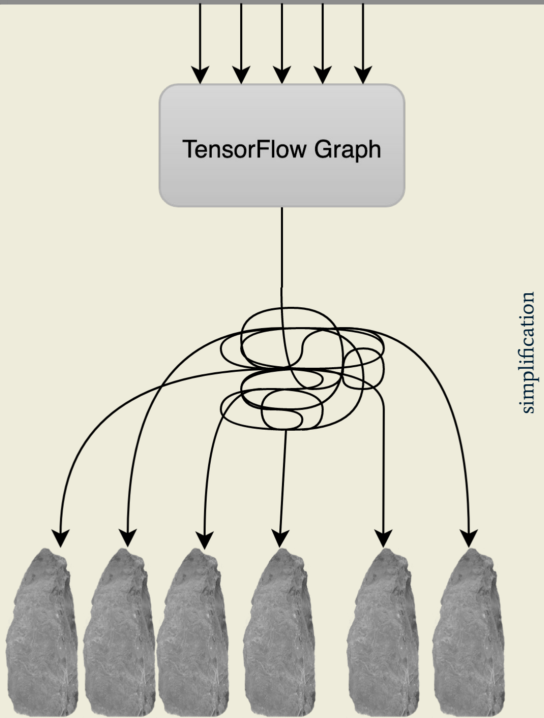
Machine Learning Systems are stuck in a Rut

Paul Barham and Michael Isard [HotOS2019]

- Currently much focus on optimizing 5 year old ML benchmarks
- New ideas in ML often require new primitives that are hard to compile and optimize for the zoo of modern hardware

Key Problem:

- Reliance on high-performance but **inflexible monolithic kernels**
- The resulting lack of expressiveness, maintainability, and modularity hinders research progress



RISE - A functional pattern-based data-parallel language

- To break up monolithic kernels we argue for:

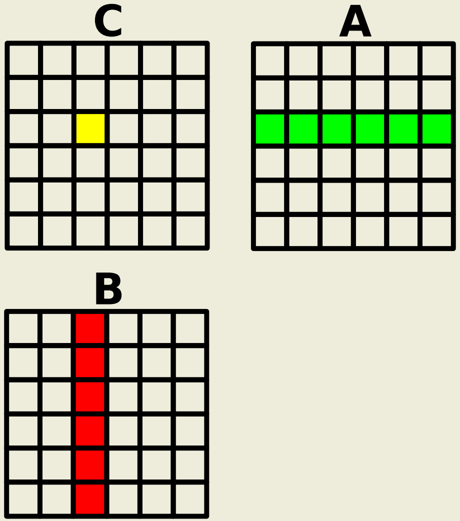
  representing computations using compositions of flexible and generic patterns

- RISE is a spiritual successor to the Lift project
- Pattern-based style efficiently represents complex multidimensional computations of various domains
- Optimization choices encoded as rewrite rules are explored automatically for competitive performance

We believe this style will allows us to express optimizations on a higher level
than what is currently available in the existing ML frameworks

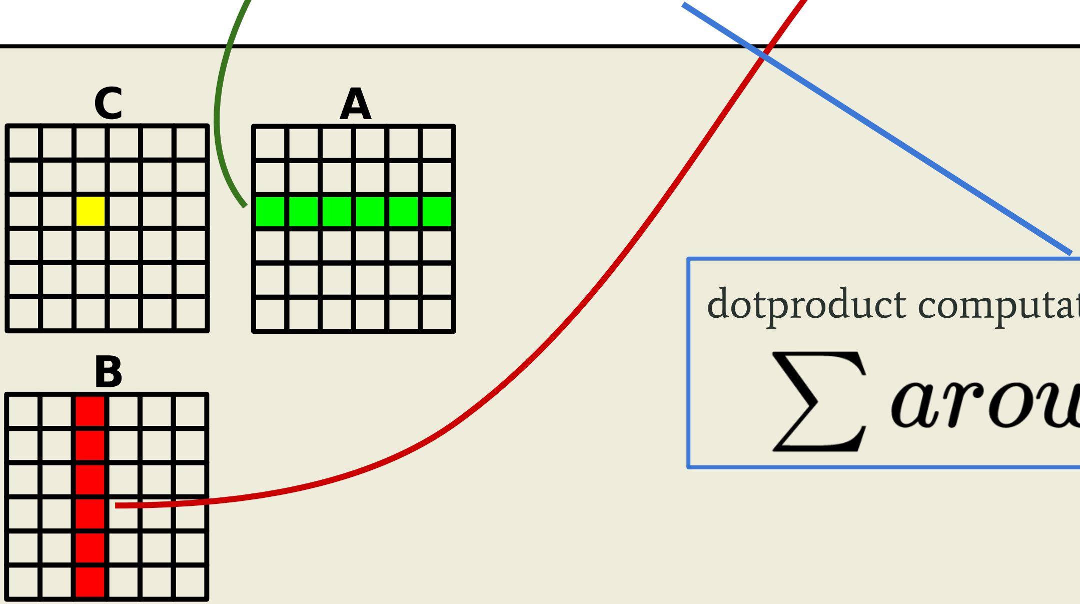
RISE by *example*: **Matrix Multiplication**

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

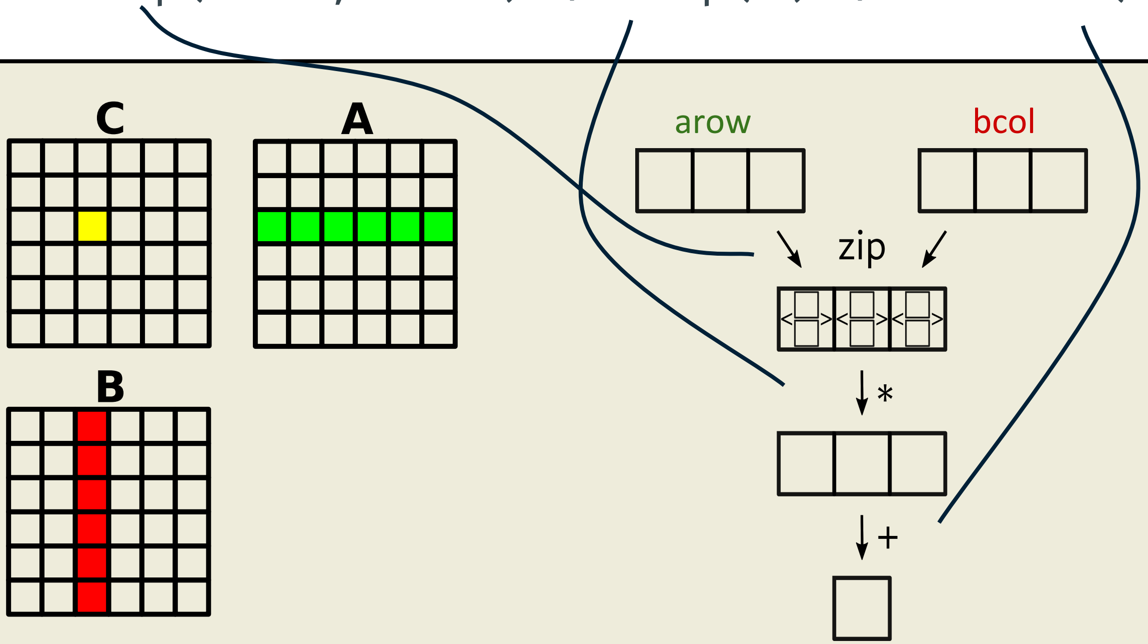


dotproduct computation:

$$\sum arow_i * bcol_i$$

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) )) )) ))
```



RISE - The Caveats

Does everyone have to write functional programs now?

Academic work written in Scala

Does not integrate well with existing ML compiler infrastructures

MLIR - Multi-Level Intermediate Representation

- Extensible infrastructure to define compiler intermediate representations

- *Dialects* can capture different levels of abstraction:
    - High-level domain specific
    - Hardware specific backend

- Existing dialects available for:
    - TensorFlow / TensorFlow Lite
    - Performing polyhedral optimizations
    - Targeting GPUs
    - Wrapping 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
    - Targeting GPUs
    - Performing polyhedral optimizations
    - Wrapping LLVM IR

MLIR - Multi-Level Intermediate Representation

- Extensible infrastructure to define compiler intermediate representations

- *Dialects* can capture different levels of abstraction:
  - High-level domain specific
  - Hardware specific backend

- Existing dialects available for:
  - TensorFlow / TensorFlow Lite
  - Performing polyhedral optimizations
  - Targeting GPUs
  - Wrapping LLVM IR

RISE in MLIR - Lambda Calculus 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 → usable for industry
- Natural integration with machine learning toolchains

→ to implement RISE we implement **λ-calculus** as an MLIR dialect

RISE dialect by example: Matrix multiplication

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) ->
              (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
              !rise.fun<data<array<3, int>> -> data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
              !rise.fun<data<array<3, int>> -> data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> -> data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```

Types of the RISE dialect

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) →
                (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
               !rise.fun<data<array<3, int>> → data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
               !rise.fun<data<array<3, int>> → data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t)  !rise.fun<data<tuple<int, int>> → data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```

DataTypes

FunctionTypes

Types of the RISE dialect

DataTypes

```
func @mm(%A: !rise.data<array<4, array<4, int>>>,
         %B: !rise.data<array<4, array<4, int>>>) →
            (!rise.data<array<4, array<4, int>>>) {
```

- Rise DataTypes: array types, tuple types, scalar types
- Nested array types represent higher dimensional data

FunctionTypes

```
%f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> → data<int>>
```

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

Patterns in the RISE dialect

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) →
              (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> → data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```

Patterns in the RISE dialect

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) →
              (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> → data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```

Patterns in the RISE dialect: `zip`

```
%zip = rise.zip #rise.nat<3> #rise.int #rise.int
```

Patterns in the RISE dialect: `zip`

```
%zip = rise.zip #rise.nat<3> #rise.int #rise.int

// type: () → !rise.fun<data<array<3, int>> →
                          fun<data<array<3, int>> →
                             data<array<3, tuple<int, int>>>>>

// type: () →
```

- One `MLIR` operation per RISE pattern
- Operations customized with attributes specifying information for the type
- Patterns encoded as operations have a RISE function type

Function application in the RISE dialect

```
%zip = rise.zip #rise.nat<3> #rise.int #rise.int
```

Function application in the RISE dialect

```
%f1 = rise.lambda (%arow) : !rise.fun<data<array<3, int>> →
                                                  data<array<3, int>>> {
    %f2 = rise.lambda (%bcol) : !rise.fun<data<array<3, int>> →
                                                  data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol
```

- `rise.apply` models function applications
- SSA value with RISE `function type` and arguments to the function are passed to `rise.apply`
- Partial function application naturally supported (i.e. not specifying all function arguments at once)

Function application in the RISE dialect

```
%f1 = rise.lambda (%arow) : !rise.fun<data<array<3, int>> →
                                      data<array<3, int>>> {
   %f2 = rise.lambda (%bcol) : !rise.fun<data<array<3, int>> →
                                          data<array<3, int>>> {
         %zip = rise.zip #rise.nat<3> #rise.int #rise.int
         %zipped = rise.apply %zip, %arow, %bcol

//type: (λ, data<array<3, int>>, data<array<3, int>>) →
                                    data<array<3,tuple<int,int>>>
```

- rise.apply models function applications
- SSA value with RISE function type and arguments to the function are passed to rise.apply
- Partial function application naturally supported (i.e. not specifying all function arguments at once)

Function abstraction in the RISE dialect

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) →
              (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> → data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```

Function abstraction in the RISE dialect

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) →
              (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
              !rise.fun<data<array<3, int>> → data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> → data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```

Function abstraction in the RISE dialect

```
%f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> → data<int>> {
        %fst = rise.fst #rise.int #rise.int
        %snd = rise.snd #rise.int #rise.int
        %t1  = rise.apply %fst, %t
        %t2  = rise.apply %snd, %t
        %mul = rise.mult #rise.int
        %res = rise.apply %mul, %t1, %t2
        rise.return %res
}
```

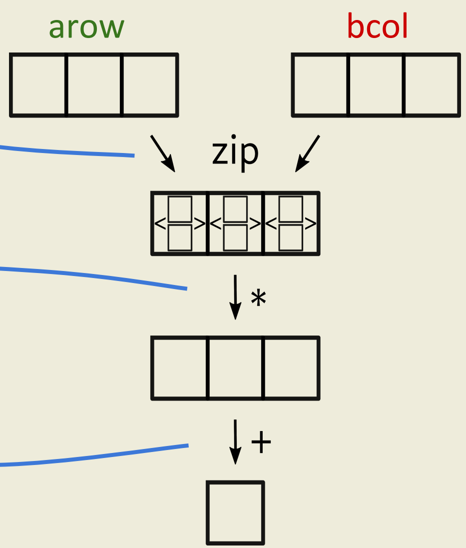
- `rise.lambda` wraps an MLIR region of exactly one block
- Arbitrary number of arguments and one result
- `rise.lambda` associates the region with a RISE `function type`

RISE dialect by example: Matrix multiply

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) ->
              (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
              !rise.fun<data<array<3, int>> -> data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
              !rise.fun<data<array<3, int>> -> data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> -> data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```

RISE dialect by example: Matrix multiply

```
func @mm(%A: !rise.data<array<3, array<3, int>>>, %B: !rise.data<array<3, array<3, int>>>) →
                (!rise.data<array<3, array<3, int>>>) {

  %f1 = rise.lambda (%arow) :
                !rise.fun<data<array<3, int>> → data<array<3, int>>> {
      %f2 = rise.lambda (%bcol) :
                !rise.fun<data<array<3, int>> → data<array<3, int>>> {
          %zip = rise.zip #rise.nat<3> #rise.int #rise.int
          %zipped = rise.apply %zip, %arow, %bcol

          %f = rise.lambda (%t) : !rise.fun<data<tuple<int, int>> → data<int>> {
              %fst = rise.fst #rise.int #rise.int
              %snd = rise.snd #rise.int #rise.int
              %t1  = rise.apply %fst, %t
              %t2  = rise.apply %snd, %t
              %mul = rise.mult #rise.int
              %res = rise.apply %mul, %t1, %t2
              rise.return %res
          }
          %map = rise.map #rise.nat<3> #rise.tuple<int, int> #rise.int
          %mapped = rise.apply %map, %f, %zipped

          %add = rise.add #rise.int
          %init = rise.literal #rise.lit<int<0>>
          %reduce = rise.reduce #rise.nat<3> #rise.int #rise.int
          %res = rise.apply %reduce, %add, %init, %mapped
          rise.return %res
      }

      %map = rise.map #rise.nat<3> #rise.array<3, int>  #rise.array<3, int>
      %res = rise.apply %map, %f2, %B
      rise.return %res
  }

  %map = rise.map #rise.nat<3> #rise.array<3,int> #rise.array<3,int>
  %res = rise.apply %map, %f1, %A
  return %res
}
```



RISE in MLIR next steps

- Explore interaction and integration with other MLIR dialects:
  - Lowering RISE to LLVM IR or other dialects (e.g. polyhedral)
  - Lowering from the TensorFlow dialect to RISE
  - Raise RISE to TensorFlow or other domain-specific dialects?
- Express optimisations as rewrite rules over the RISE dialect

RISE in MLIR

A functional Pattern-based Dialect

Fork me on GitHub

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