RISE

A functional pattern-based language in MLIR

Martin Lücke | Michel Steuwer | Aaron Smith

THE UNIVERSITY of EDINBURGH

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

Why RISE?

[HotOS'19]

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
Google Brain

Original authors of TensorFlow

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

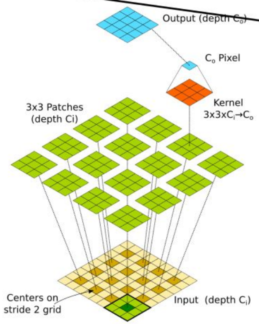


Figure 1. Conv2D operation with 3×3 kernel, stride=2

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 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

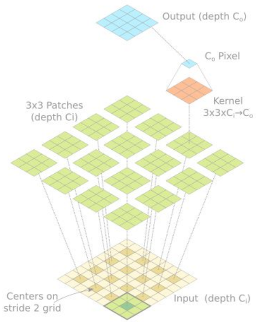


Figure 1. Conv2D operation with 3×3 kernel, stride=2

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

We should aim for more principled higher level intermediate representations

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

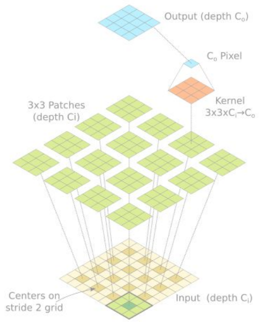


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convo-

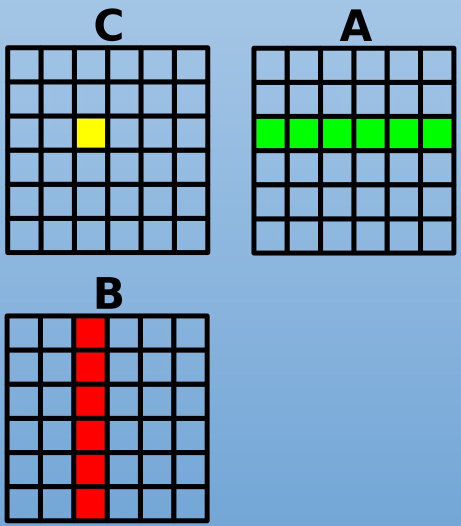
smaller models. We wanted to understand why.

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

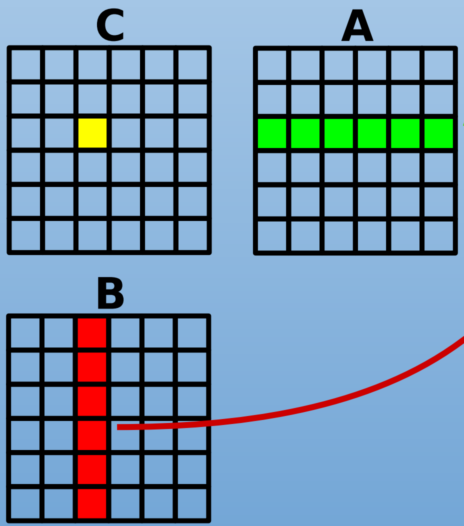
RISE by example: Matrix Multiplication

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

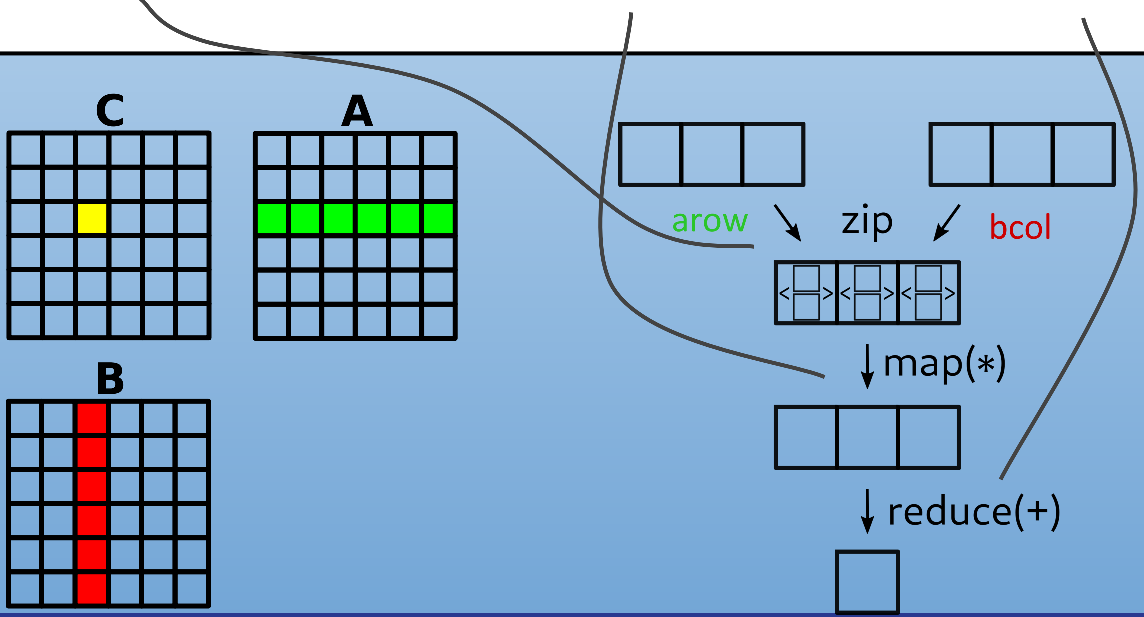


dot product computation:

$$\sum arow_i * bcol_i$$

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) )) )) ))
```

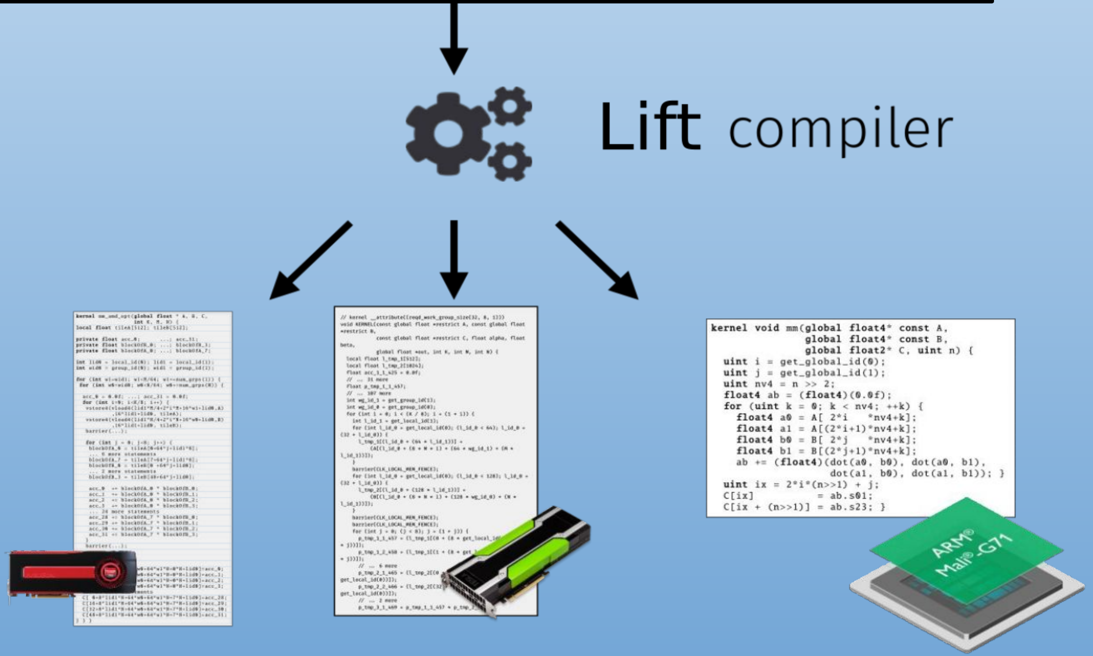


Lessons from Lift

- RISE (https://rise-lang.org/) is a spiritual successor to the Lift project
- Lift has demonstrated:
  - optimizing by rewriting
  - achieving high performance
  - performance portability

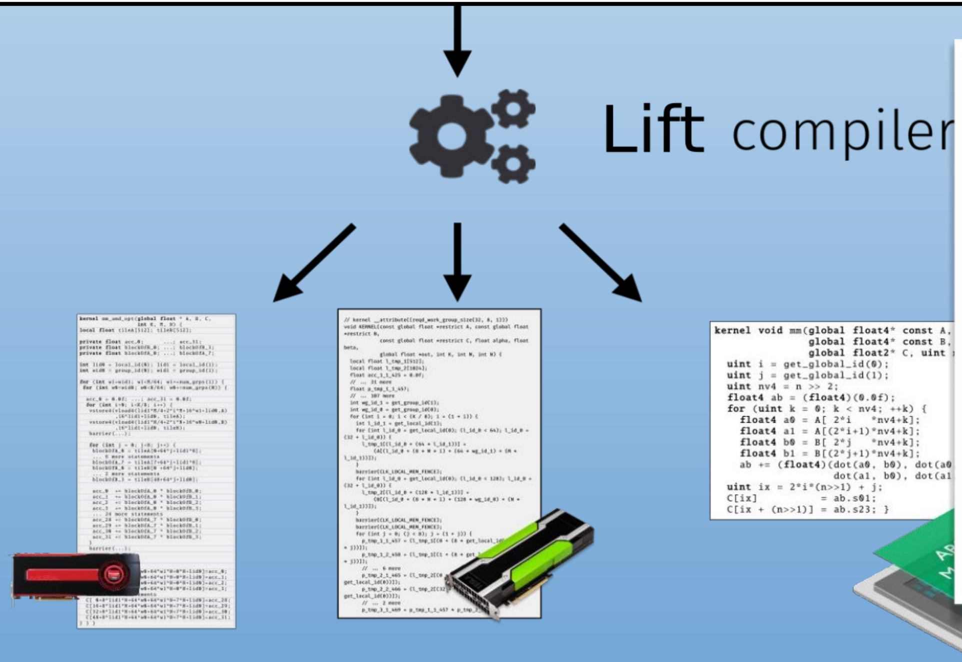
Lift: Performance Portability

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

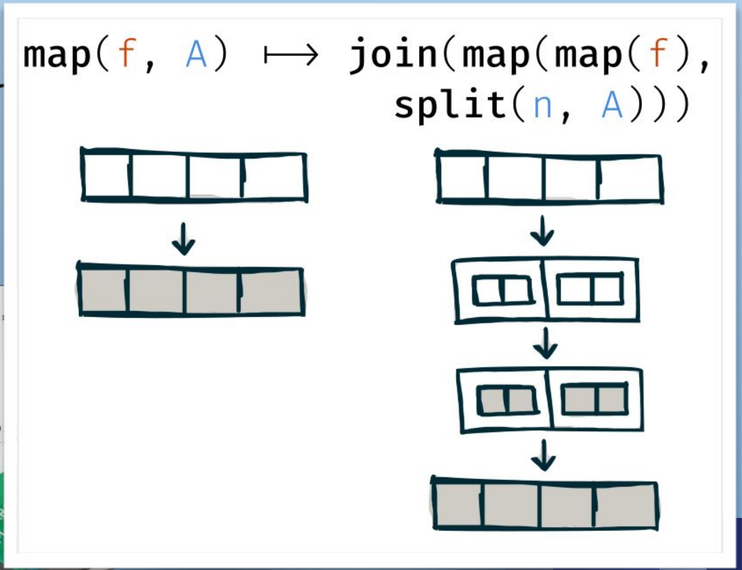


Lift: Optimization by Rewriting

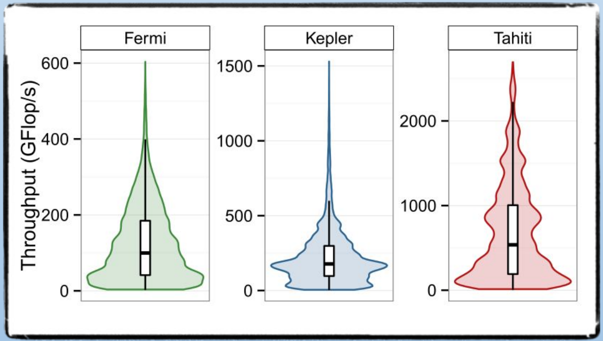
```
fun(A : N.K.float ⇒ fun(B : K.M.float ⇒
  A ▷ map(fun(arow ⇒
    B ▷ transpose ▷ map(fun(bcol ⇒
      zip(arow, bcol) ▷ map(*) ▷ reduce(+, 0) )) )) ))
```



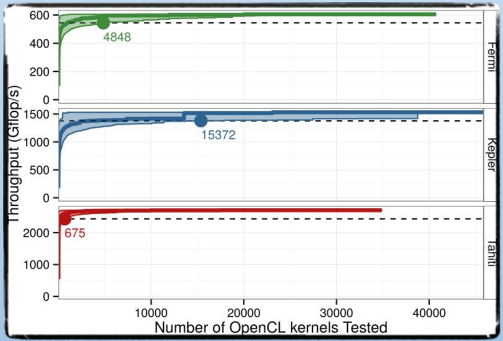
Rewrite Rules



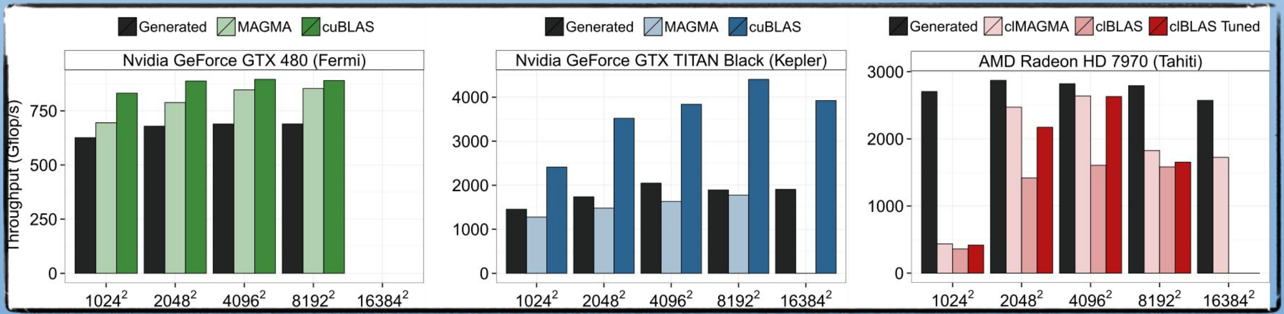
Lift High Performance Results for Matrix Multiplication



Only few generated code with very good performance



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



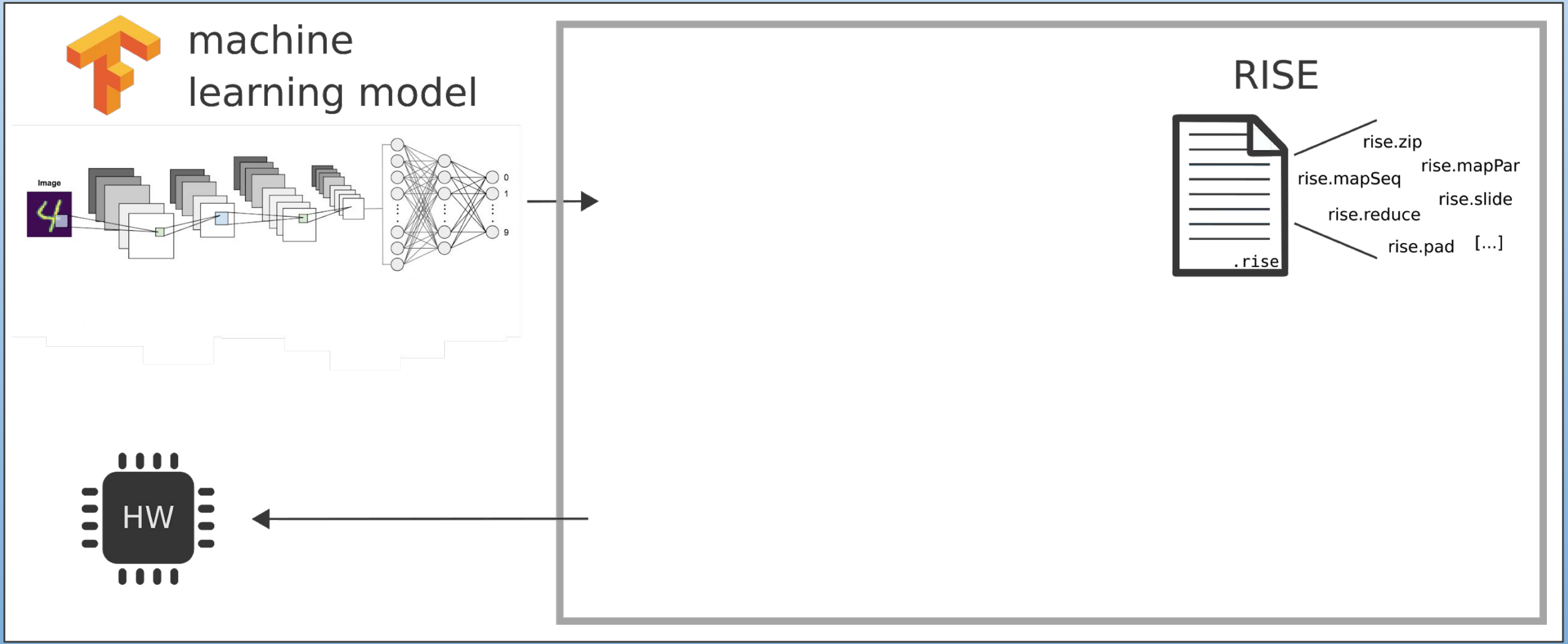
[GPGPU'16]

Performance close or better than hand-tuned library code

Lift - The Caveats

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

How can we achieve end-to-end integration with existing solutions?



MLIR - Multi-Level Intermediate Representation

- Extensible infrastructure to define compiler intermediate representations (dialects)

- 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)

- 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

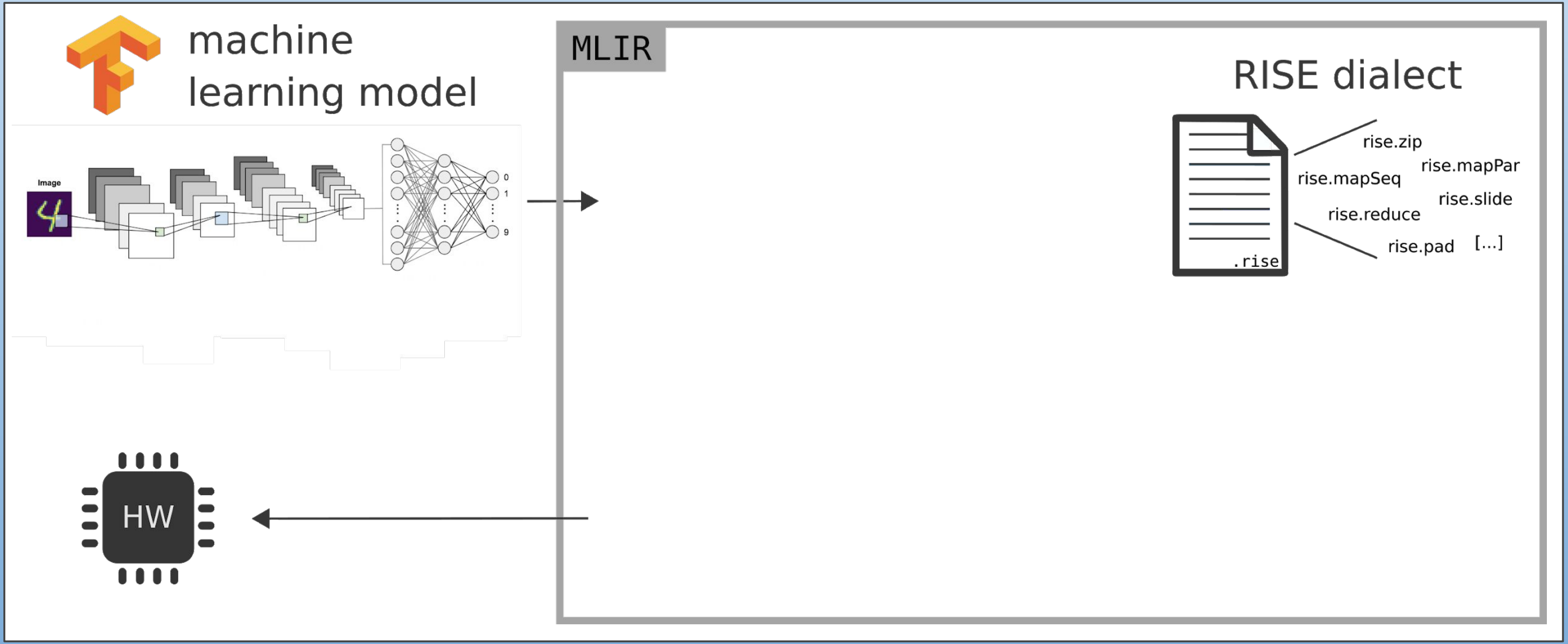
MLIR - Multi-Level Intermediate Representation

- Extensible infrastructure to define compiler intermediate representations (dialects)

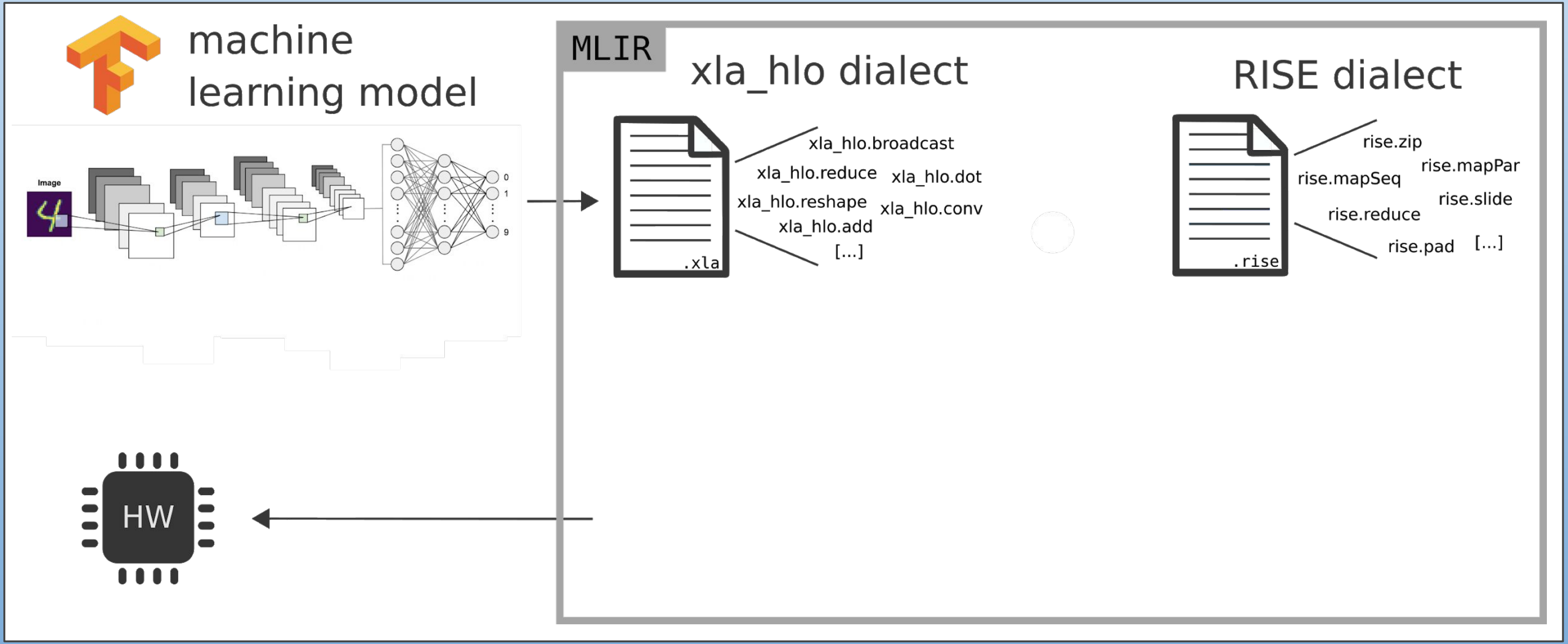
- Dialects can capture different levels of abstraction:
  - High-level domain specific ------------- Hardware specific backend

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

How can we achieve end-to-end integration with existing solutions?



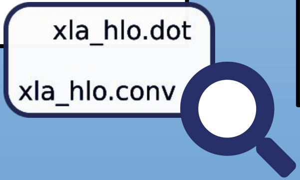
How can we achieve end-to-end integration with existing solutions?



Lowering TensorFlow models to RISE

1. Lower TensorFlow model to XLA_HLO dialect
2. Match for supported operations
3. Replace operations with corresponding RISE operations

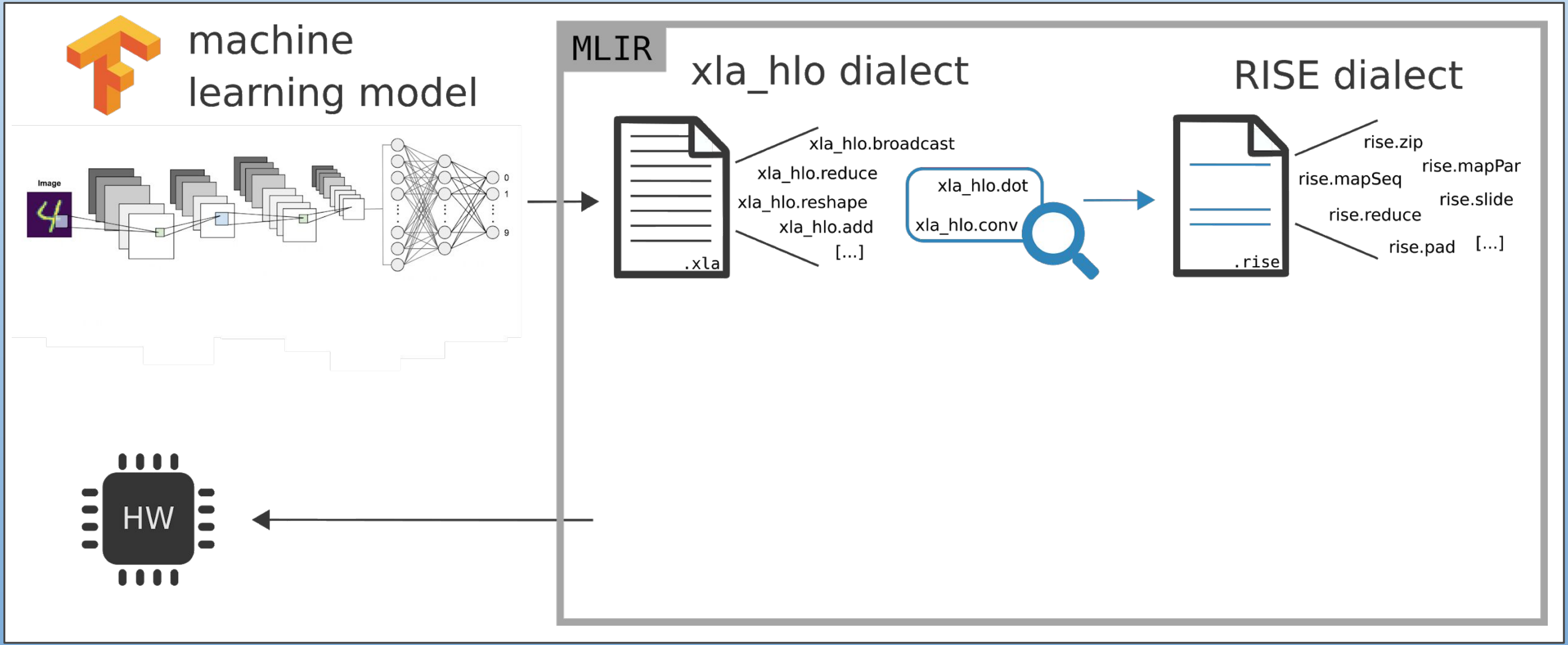
```
func @mnist_predict(%input: tensor<1×28×28×1xf32>) → tensor<1×10×f32> {
    %1 = hlo.reshape (%input) : (tensor<1×28×28×1xf32>) → tensor<1×784xf32>
    %2 = hlo.dot (%1, %kernel) : (tensor<1×784xf32>, tensor<784×128xf32>) → tensor<1×128xf32>
    %3 = hlo.add (%2, %bias) : (tensor<1×128xf32>, tensor<128xf32>) → tensor<1×128xf32>
    [ … ]
    %4 = hlo.dot (%3, %kernel_2) : (tensor<1×128xf32>, tensor<128×10×f32>) → tensor<1×10×f32>
    %5 = hlo.add %4, %bias_2 : (tensor<1×10×f32>, tensor<10×f32>) → tensor<1×10×f32>
    [ … ]
    return %5 : tensor<1×10×f32>
  }
}
```




```
%1 ▷ map(fun(arow ⇒
  %kernel ▷ transpose ▷ map(fun(bcol ⇒
    zip(arow, bcol) ▷ map(*) ▷ reduce(+, 0) )) ))
```

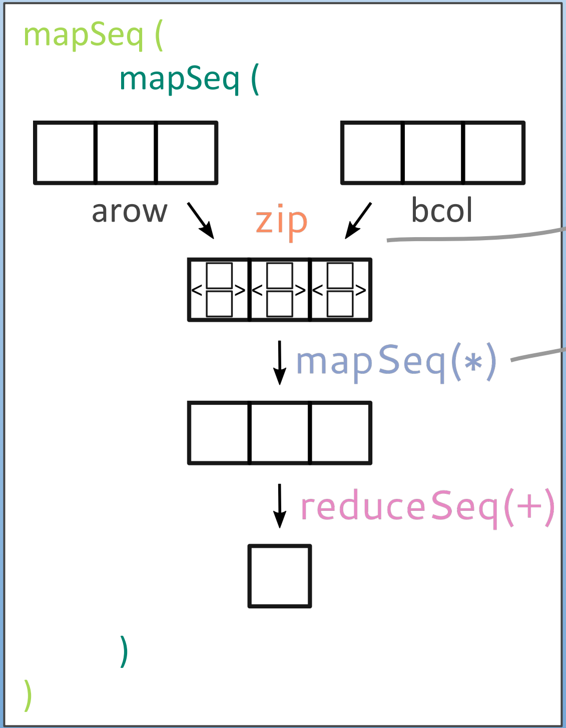
```
%3 ▷ map(fun(arow ⇒
  %kernel_2 ▷ transpose ▷ map(fun(bcol ⇒
    zip(arow, bcol) ▷ map(*) ▷ reduce(+, 0) )) ))
```

How can we achieve end-to-end integration with existing solutions?



RISE dialect by example: Matrix Multiplication

```
%A ▷ map(fun(arow ⇒
  %B ▷ transpose ▷ map(fun(bcol ⇒
    zip(arow, bcol) ▷ map(*) ▷ reduce(+, 0) )) ))
```




```
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.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.scalar<f32>
    %result = rise.apply %mapB, %f2, %B
    rise.return %result : !rise.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.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.scalar<f32>
    %result = rise.apply %mapB, %f2, %B
    rise.return %result : !rise.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

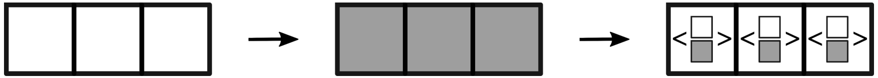
```
func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
  %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
  %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
  %f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) → !rise.array<1024, scalar<f32>> {
    %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) → !rise.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.scalar<f32>
    %result = rise.apply %mapB, %f2, %B
    rise.return %result : !rise.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>>) →
                                      !rise.array<1024, scalar<f32>> {
    %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) →
                                      !rise.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

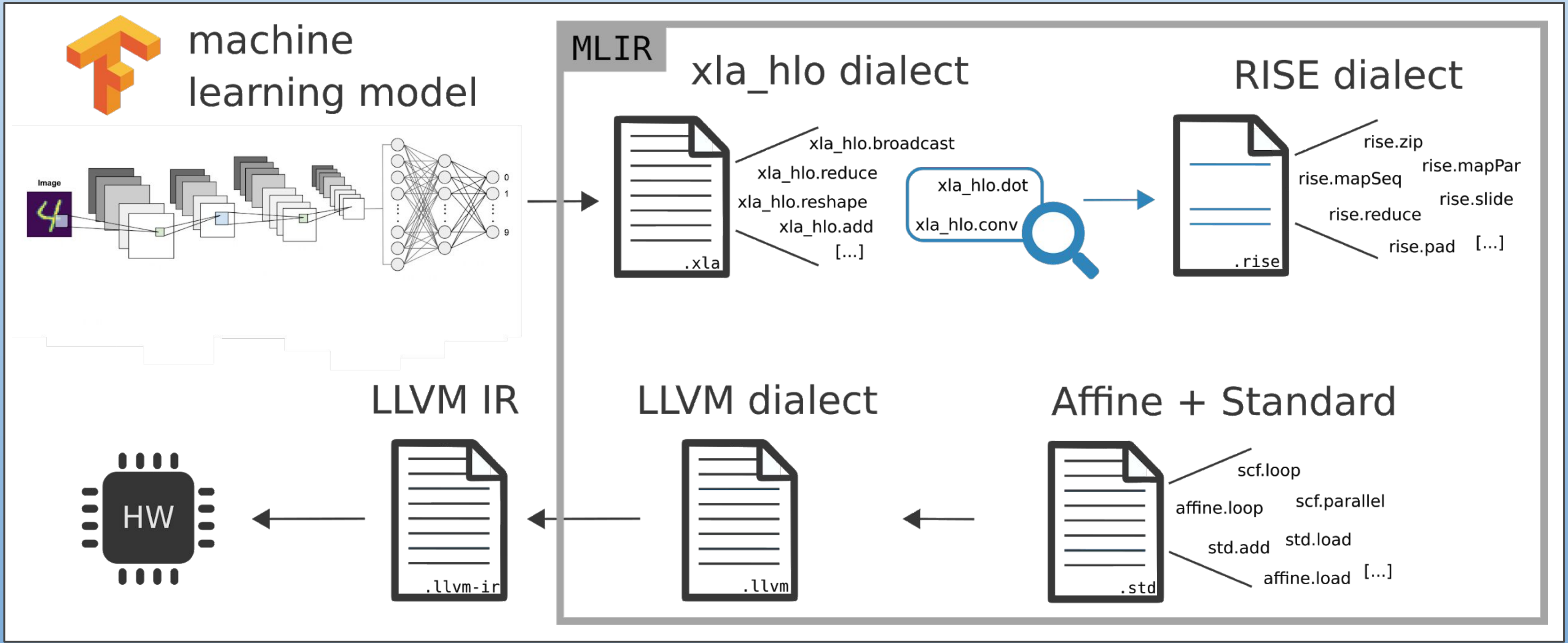
```
%f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) →
                                        !rise.array<1024, scalar<f32>> {
   %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) →
                                        !rise.scalar<f32> {
        %zip = rise.zip #rise.nat<1024>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

How can we achieve end-to-end integration with existing solutions?



Lowering of the RISE dialect

```
func @mm(%outArg, %inA, %inB) {
 %A = in %inA
 %B = in %inB
 %m1fun = lambda(%arow) -> array<2048, scalar<f32>> {
  %m2fun = lambda(%bcol) -> scalar<f32> {
   %zipFun = zip #nat<2048> #scalar<f32> #scalar<f32>
   %zippedArrays = apply %zipFun, %arow, %bcol
   %reductionLambda = lambda(%tuple, %acc) -> scalar<f32>{
    %fstFun = fst #scalar<f32> #scalar<f32>
    %sndFun = snd #scalar<f32> #scalar<f32>
    %first = apply %fstFun, %tuple
    %second = apply %sndFun, %tuple
    %result = embed(%first, %second, %acc) {
     %product = mulf %first, %second :f32
     %result = addf %product, %acc : f32
     return %result : f32
    }
    return %result : scalar<f32>
   }
   %init = literal #lit<0.0>
   %reduceFun = reduceSeq #nat<2048>
            #tuple<scalar<f32>, scalar<f32>> #scalar<f32>
   %result = apply %reduceFun, %reductionLambda,
                  %init, %zippedArrays
   return %result : scalar<f32>
  }
  %m2 = mapSeq #nat<2048> #array<2048, scalar<f32>>
                           #scalar<f32>
  %result = apply %m2, %m2fun, %B
  return %result : array<2048, scalar<f32>>
 }
 %m1 = mapSeq  #nat<2048> #array<2048, scalar<f32>>
                               #array<2048, scalar<f32>>
 %result = apply %m1, %m1fun, %A
 out %outArg <- %result
 return
}
```

Matrix Multiplication in RISE

?

```
func @mm(%outArg, %inA, %inB) {
 %init = constant 0.000000e+00 : f32
 affine.for %i = 0 to 2048 {
   affine.for %j = 0 to 2048 {
     affine.store %init, %outArg[%i, %j]
     affine.for %k = 0 to 2048 {
       %a = affine.load %inA[%i, %k]
       %b = affine.load %inB[%k, %j]
       %c = affine.load %outArg[%i, %j]
       %1 = mulf %a, %b : f32
       %2 = addf %1, %c : f32
       affine.store %2, %outArg[%i, %j]
     }
   }
 }
 return
}
```

Matrix Multiplication in Affine + Standard

Lowering of the RISE dialect

```
func @mm(%outArg, %inA, %inB) {
 %A = in %inA
 %B = in %inB
 %m1fun = lambda(%arow) -> array<2048, scalar<f32>> {
  %m2fun = lambda(%bcol) -> scalar<f32> {
   %zipFun = zip #nat<2048> #scalar<f32> #scalar<f32>
   %zippedArrays = apply %zipFun, %arow, %bcol
   %reductionLambda = lambda(%tuple, %acc) -> scalar<f32>{
    %fstFun = fst #scalar<f32> #scalar<f32>
    %sndFun = snd #scalar<f32> #scalar<f32>
    %first = apply %fstFun, %tuple
    %second = apply %sndFun, %tuple
    %result = embed(%first, %second, %acc) {
     %product = mulf %first, %second :f32
     %result = addf %product, %acc : f32
     return %result : f32
    }
    return %result : scalar<f32>
   }
   %init = literal #lit<0.0>
   %reduceFun = reduceSeq #nat<2048>
            #tuple<scalar<f32>, scalar<f32>> #scalar<f32>
   %result = apply %reduceFun, %reductionLambda,
          %init, %zippedArrays
   return %result : scalar<f32>
  }
  %m2 = mapSeq #nat<2048> #array<2048, scalar<f32>>
             #scalar<f32>
  %result = apply %m2, %m2fun, %B
  return %result : array<2048, scalar<f32>>
 }
 %m1 = mapSeq  #nat<2048> #array<2048, scalar<f32>>
                #array<2048, scalar<f32>>
 %result = apply %m1, %m1fun, %A
 out %outArg <- %result
 return
}
```

Matrix Multiplication in RISE

1. Lowering functional to imperative representation

```
func @mm_codegen(%outArg, %inA, %inB){
 %A = codegen.cast(%inA)
 %B = codegen.cast(%inB)
 %out = codegen.cast(%outArg)
 affine.for %i = 0 to 2048 {
  %3 = codegen.idx(%A, %i)
  %4 = codegen.idx(%out, %i)
  affine.for %j = 0 to 2048 {
   %5 = codegen.idx(%B, %j)
   %6 = codegen.idx(%4, %j)
   %7 = codegen.zip(%3, %5)
   %init = embed() {
    %cst = constant 0.0 : f32
    return(%cst) : (f32) -> ()
   }
   codegen.assign(%init, %6)
   affine.for %k = 0 to 2048 {
    %9 = codegen.idx(%7, %k)
    %10 = codegen.fst(%9)
    %11 = codegen.snd(%9)
    %12 = embed(%10, %11, %6) {
     %13 = mulf %arg6, %arg7 : f32
     %14 = addf %13, %arg8 : f32
     return(%14) : (f32) -> ()
    }
    codegen.assign(%12, %6)
   }
  }
 }
 return
}
```

Matrix Multiplication in imperative RISE

?

```
func @mm(%outA
 %init = const
 affine.for %i
  affine.for %
   affine.st
   affine.fo
    %a = af
    %b = af
    %c = af
    %1 = mu
    %2 = ad
    affine.
   }
  }
 }
 return
}
```

Lowering of the RISE dialect

```
r<f32>> {

alar<f32>
ol
> scalar<f32>{

#scalar<f32>
bda,

r<f32>>

r<f32>>
calar<f32>>
```

1. Lowering functional to imperative representation

```
func @mm_codegen(%outArg, %inA, %inB){
 %A = codegen.cast(%inA)
 %B = codegen.cast(%inB)
 %out = codegen.cast(%outArg)
 affine.for %i = 0 to 2048 {
  %3 = codegen.idx(%A, %i)
  %4 = codegen.idx(%out, %i)
  affine.for %j = 0 to 2048 {
    %5 = codegen.idx(%B, %j)
    %6 = codegen.idx(%4, %j)
    %7 = codegen.zip(%3, %5)
    %init = embed() {
      %cst = constant 0.0 : f32
      return(%cst) : (f32) -> ()
    }
    codegen.assign(%init, %6)
    affine.for %k = 0 to 2048 {
      %9 = codegen.idx(%7, %k)
      %10 = codegen.fst(%9)
      %11 = codegen.snd(%9)
      %12 = embed(%10, %11, %6) {
        %13 = mulf %arg6, %arg7 : f32
        %14 = addf %13, %arg8 : f32
        return(%14) : (f32) -> ()
      }
      codegen.assign(%12, %6)
    }
  }
 }
 return
}
```

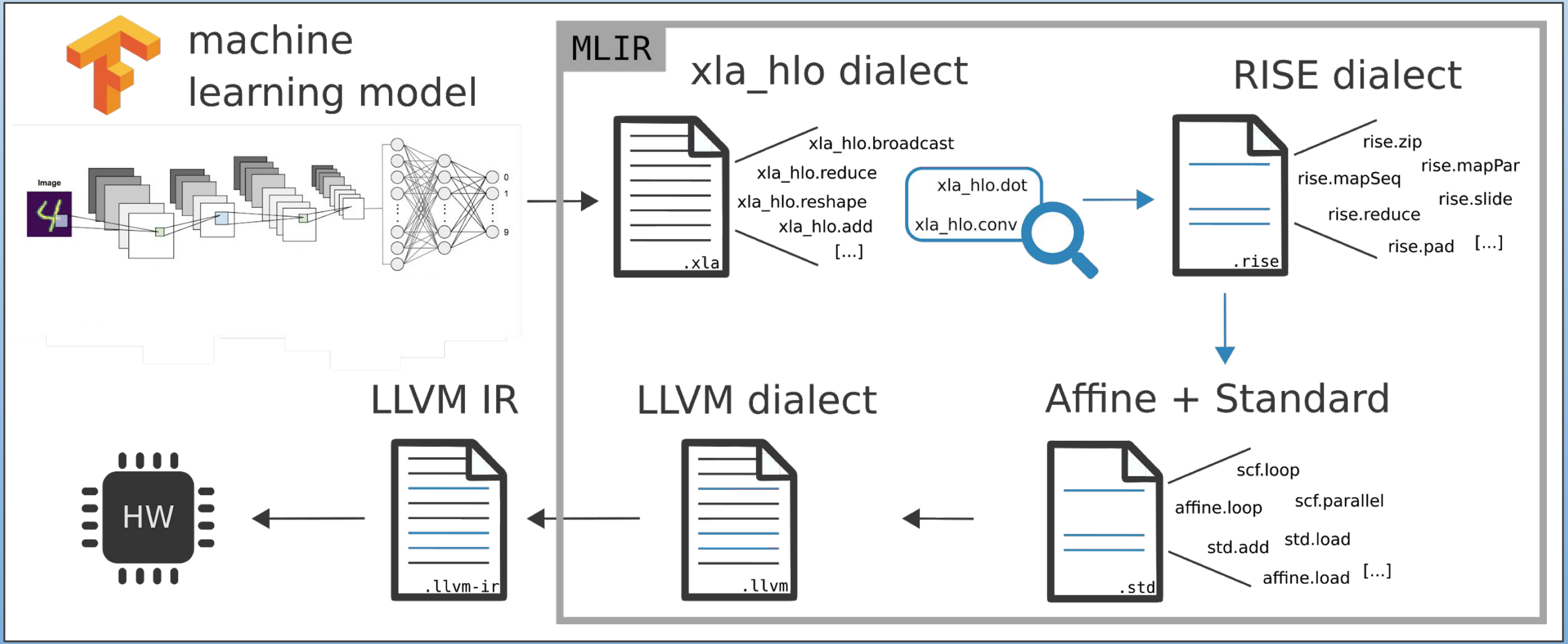
Matrix Multiplication in imperative RISE

2. Lowering imperative to target representation

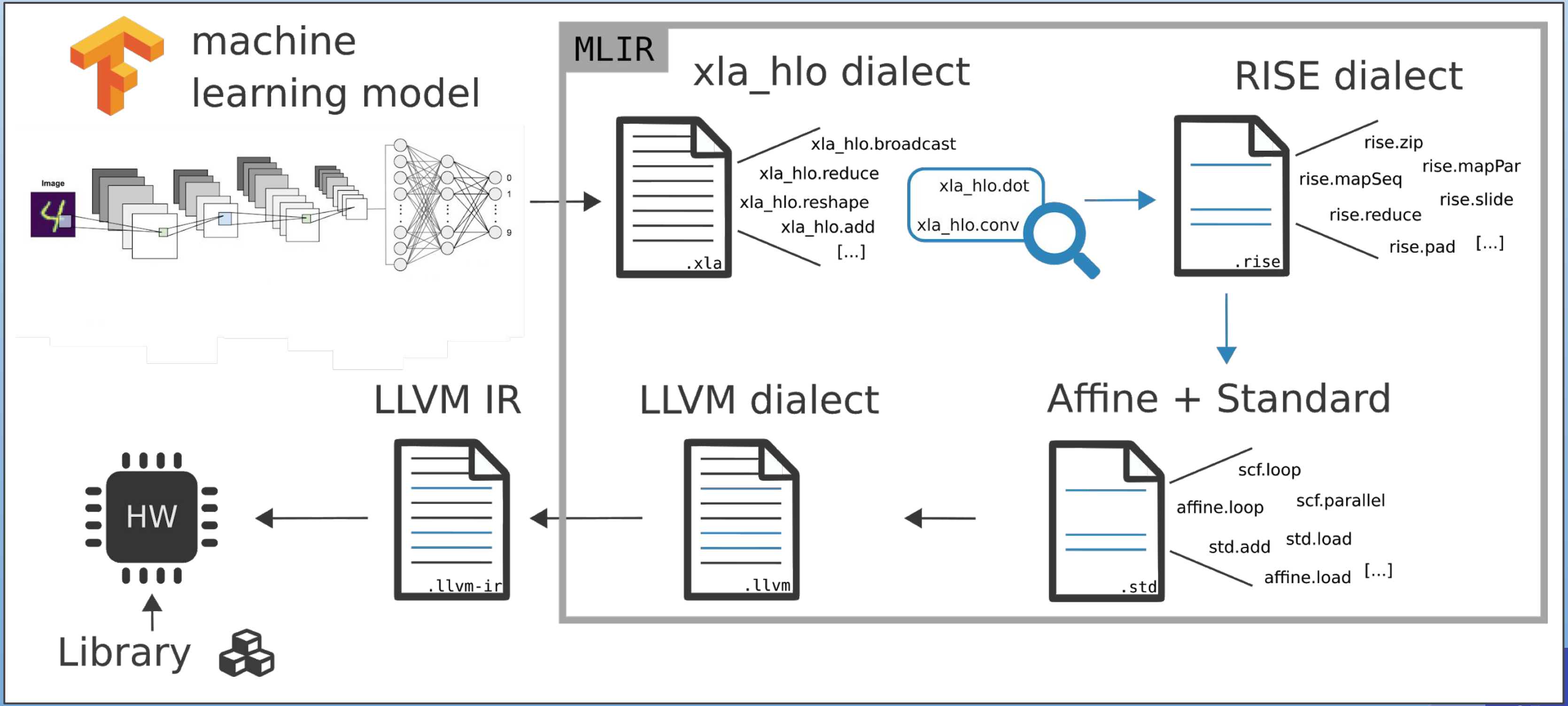
```
func @mm(%outArg, %inA, %inB) {
 %init = constant 0.000000e+00 : f32
 affine.for %i = 0 to 2048 {
  affine.for %j = 0 to 2048 {
   affine.store %init, %outArg[%i, %j]
   affine.for %k = 0 to 2048 {
    %a = affine.load %inA[%i, %k]
    %b = affine.load %inB[%k, %j]
    %c = affine.load %outArg[%i, %j]
    %1 = mulf %a, %b : f32
    %2 = addf %1, %c : f32
    affine.store %2, %outArg[%i, %j]
   }
  }
 }
 return
}
```

Matrix Multiplication in Affine+Standard

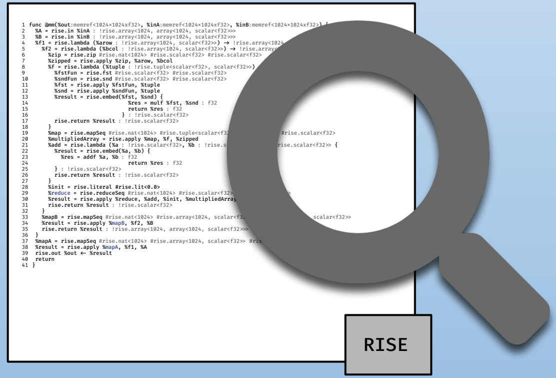
RISE end-to-end integration



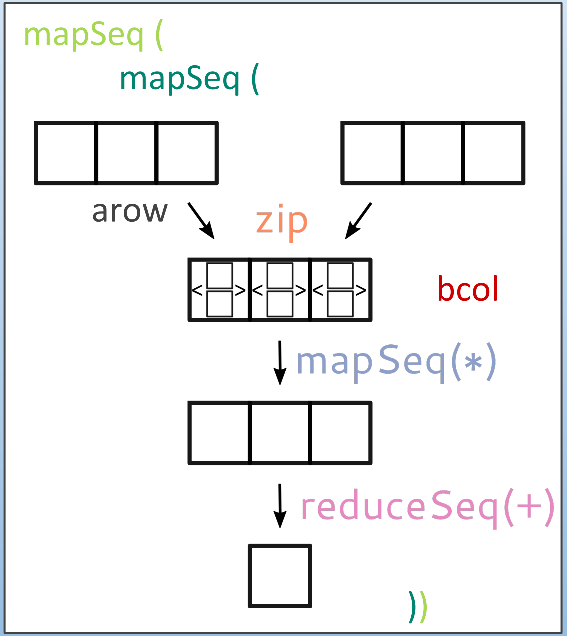
RISE end-to-end integration



Lowering RISE to library calls



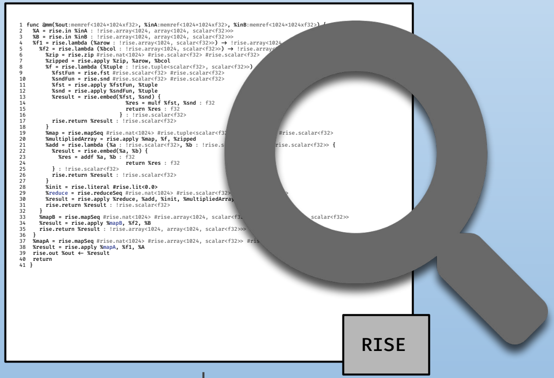

```
match_for(
```



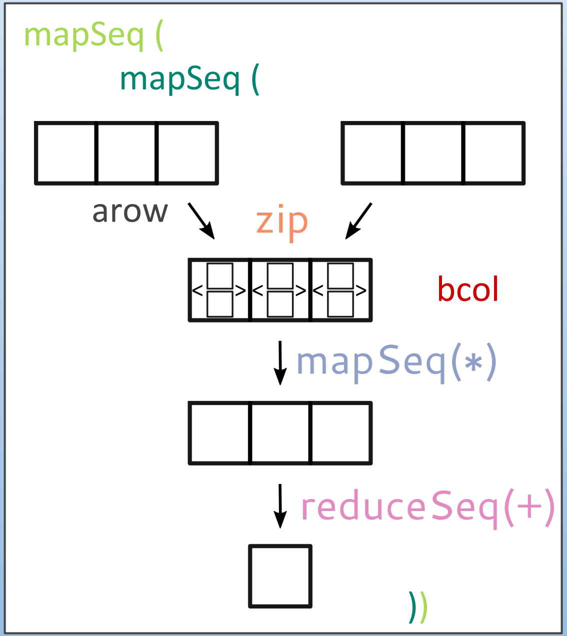

```
)
```

```
matchSuccess
```

Lowering RISE to library calls



match_for(

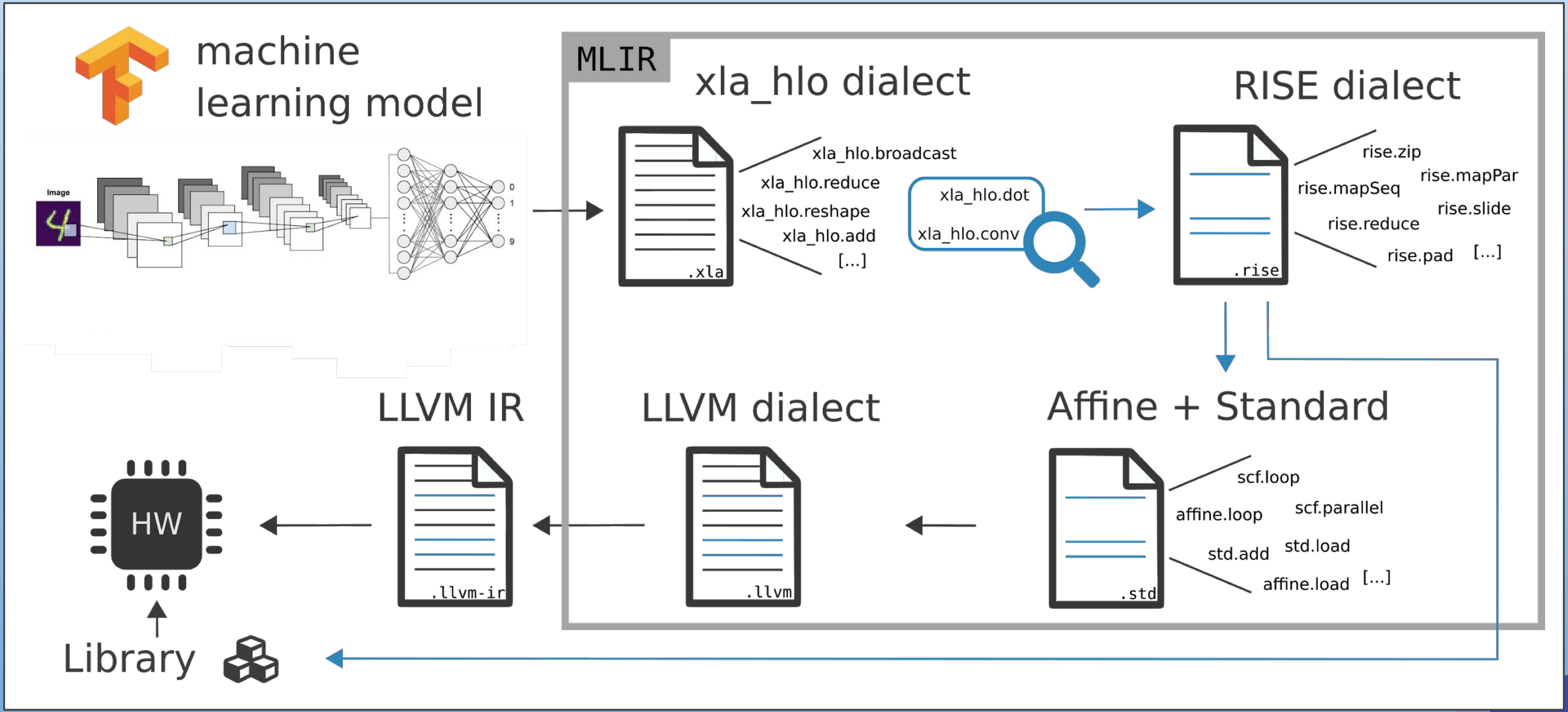


)

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
}
```

RISE end-to-end integration



RISE end-to-end demonstration

```
➜ tools git:(master) head mnist.mlir -n 4
func @mnist_predict(%input: tensor<1x28x28x1xf32>) -> tensor<1x10xf32> {
    %1 = hlo.reshape (%input) : (tensor<1x28x28x1xf32>) -> tensor<1x784xf32>
    %2 = hlo.dot (%1, %kernel) : (tensor<1x784xf32>, tensor<784x128xf32>) -> tensor<1x128xf32>
    %3 = hlo.add (%2, %bias) : (tensor<1x128xf32>, tensor<128xf32>) -> tensor<1x128xf32>
➜ tools git:(master) run-mlir.sh -target-backends=llvm-ir mnist.mlir -input-value="1x28x28x1xf32"
Lowering XLA_HLO -> RISE
Lowering RISE -> Affine
Lowering Affine -> LLVM-IR
Compiling for target backend 'llvm-ir*'
Evaluating all functions in module for driver 'llvm'
Creating driver and device for 'llvm'
EXECUTE @mnist_predict

result[0]: Buffer<float32[1x10]> 1x10xf32=[0.0 0.0 0.9 0.0 0.0 0.0 0.0 0.1 0.0 0.0]
```

Fork me on GitHub

RISE

A functional Pattern-based language in MLIR

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