Lowering XLA HLO using RISE - A Functional Pattern-based MLIR Dialect

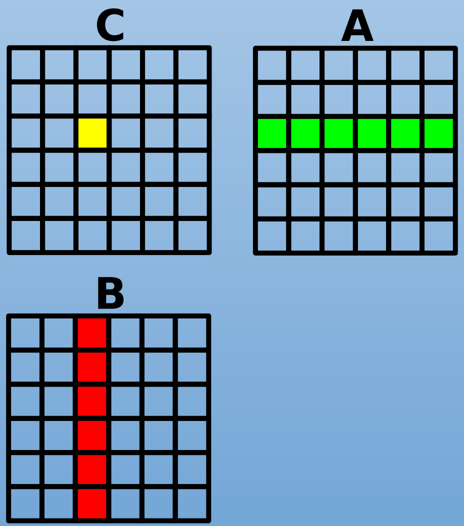
Martin Lücke | Michel Steuwer | Aaron Smith

THE UNIVERSITY *of* EDINBURGH

supported by Google Faculty Research Awards

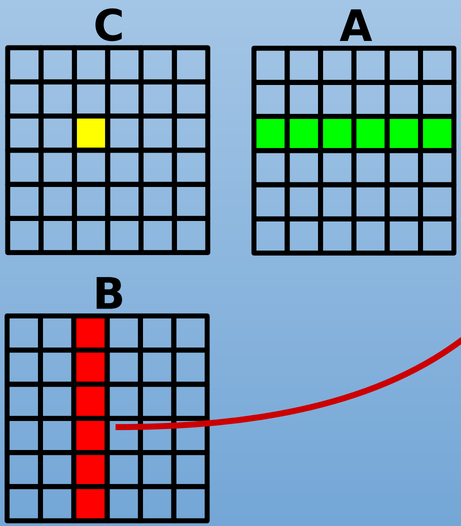
RISE by example: Matrix Multiplication

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

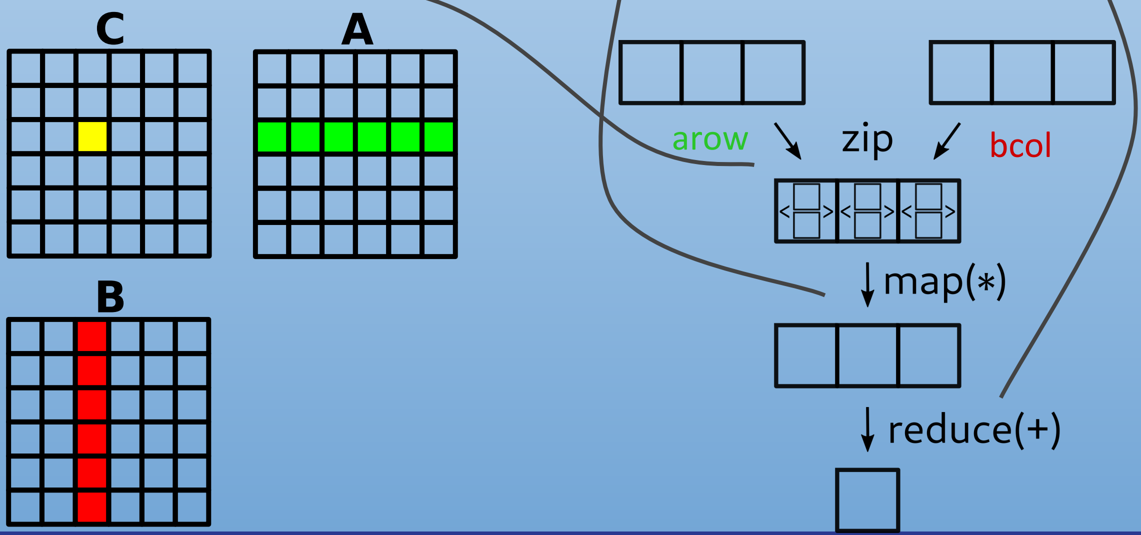


dot product computation:

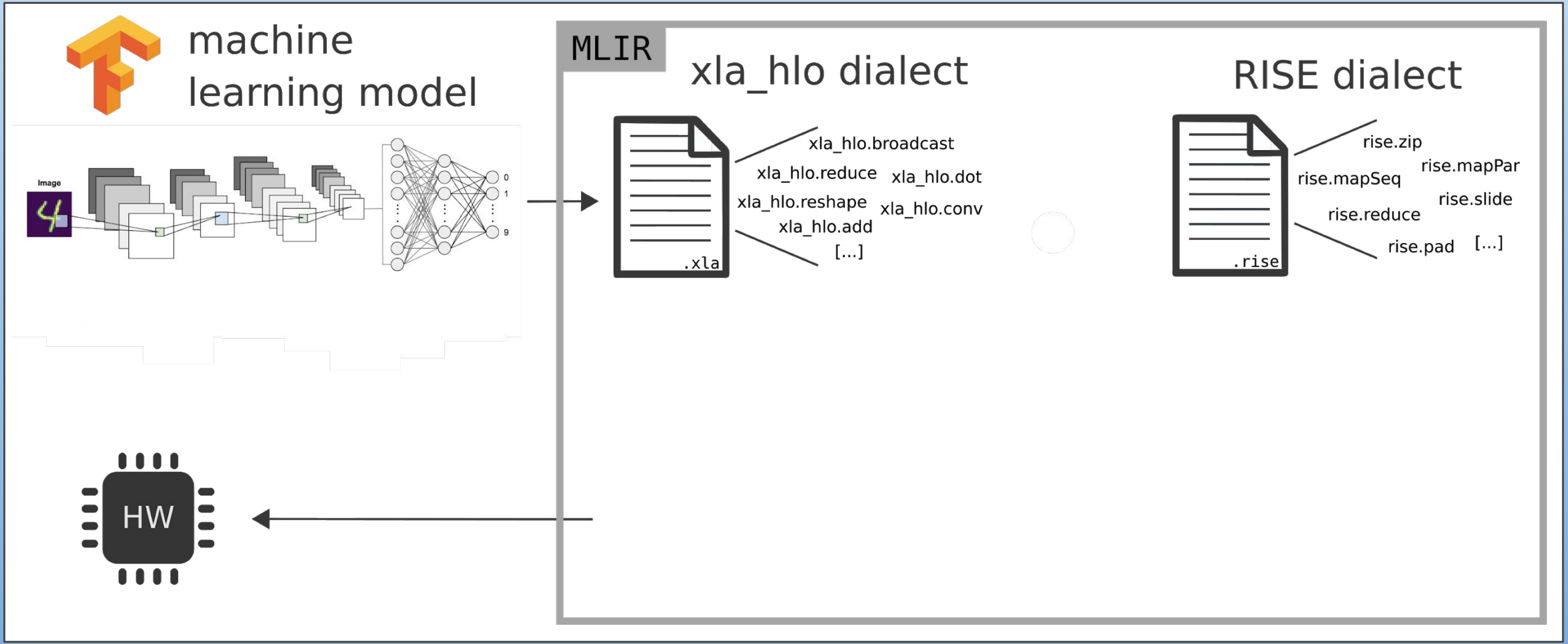
$$\sum arow_i * bcol_i$$

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) )) )) ))
```



Lowering XLA HLO using RISE



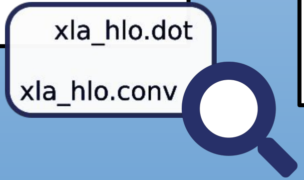
Lowering XLA HLO to RISE

1. Lower TensorFlow model to XLA_HLO dialect

2. Match for supported operations

3. Replace operations with corresponding RISE operations

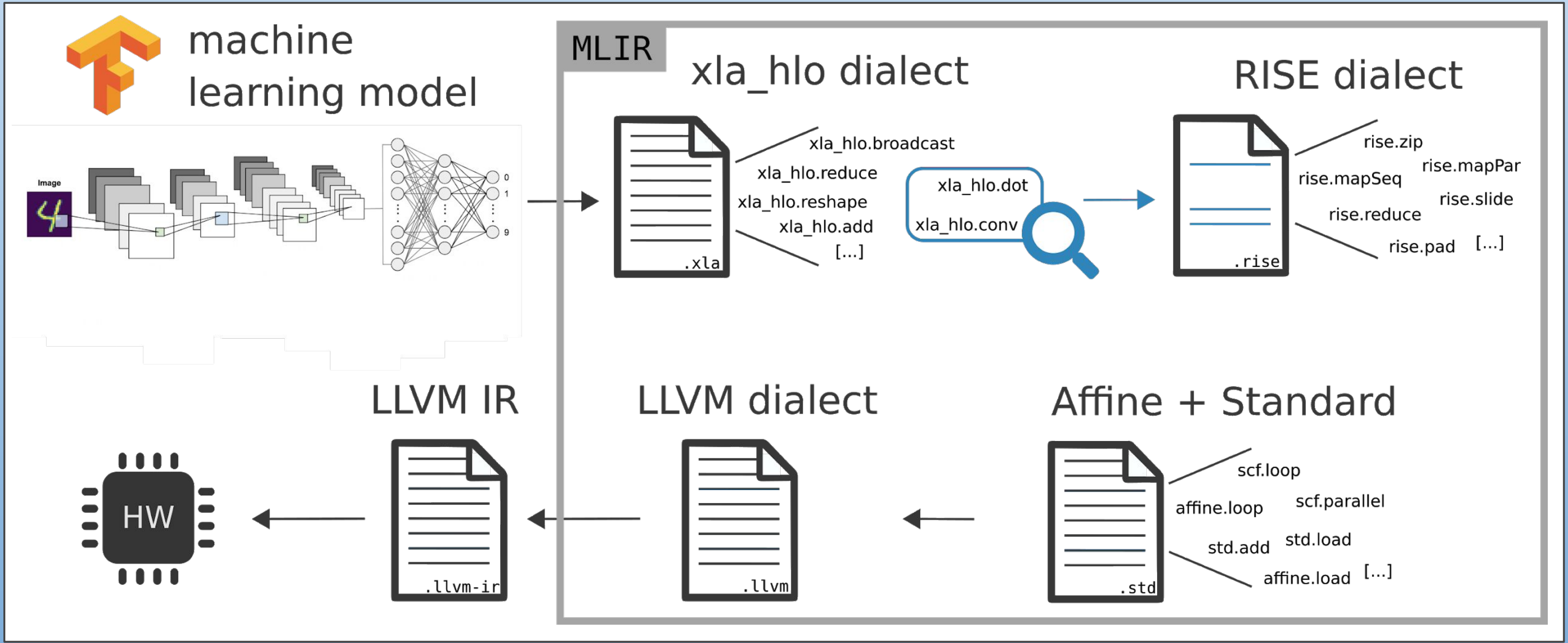
```
func @mnist_predict(%input: tensor<1×28×28×1xf32>) → tensor<1×10×f32> {
    %1 = mhlo.reshape (%input) : (tensor<1×28×28×1xf32>) → tensor<1×784xf32>
    %2 = mhlo.dot (%1, %kernel) : (tensor<1×784xf32>, tensor<784×128xf32>) → tensor<1×128xf32>
    %3 = mhlo.add (%2, %bias) : (tensor<1×128xf32>, tensor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

Lowering XLA HLO using RISE



Lowering RISE to Affine

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

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

Matrix Multiplication in RISE

1. Lowering functional to imperative representation

```
func @mm_codegen(%outC, %inA, %inB){
 %A = codegen.cast(%inA)
 %B = codegen.cast(%inB)
 %C = codegen.cast(%outC)
 %B_t = codegen.transpose(%B)
 affine.for %i = 0 to 2048 {
  %A@i = codegen.idx(%A, %i)
  %C@i = codegen.idx(%C, %i)
  affine.for %j = 0 to 2048 {
   %B_t@j = codegen.idx(%B_t, %j)
   %c = codegen.idx(%C@i, %j)
   %arow&bcol = codegen.zip(%A@i,%B_t@j)
   %init = rise.embed() {
    %cst_0 = constant 0.0 : f32
    return(%cst_0) : (f32) -> ()
   }
   codegen.assign(%init, %c)
   affine.for %k = 0 to 2048 {
    %a&b = codegen.idx(%arow&bcol, %k)
    %a = codegen.fst(%a&b)
    %b = codegen.snd(%a&b)
    %result = rise.embed(%a, %b, %c) {
     %0 = mulf %a, %b : f32
     %1 = addf %0, %c : f32
     return(%1) : (f32) -> ()
    }
    codegen.assign(%result, %c)
   }
  }
 }
 return
}
```

Matrix Multiplication in imperative RISE

?

```
%init = constar
affine.for %i =
  affine.for %j
    affine.stor
    affine.for
      %a = affi
      %b = affi
      %c = affi
      %0 = mulf
      %1 = addf
      affine.st
    }
  }
}
return
```

Lowering RISE to Affine

```
 #scalar<f32>
f32>> {
#scalar<f32>
%bcol
scalar<f32>{
2>
2>
) {
scalar<f32>
Lambda,
alar<f32>>
alar<f32>>
lar<f32>>
```

1. Lowering functional to imperative representation

```
func @mm_codegen(%outC, %inA, %inB){
 %A = codegen.cast(%inA)
 %B = codegen.cast(%inB)
 %C = codegen.cast(%outC)
 %B_t = codegen.transpose(%B)
 affine.for %i = 0 to 2048 {
  %A@i = codegen.idx(%A, %i)
  %C@i = codegen.idx(%C, %i)
  affine.for %j = 0 to 2048 {
   %B_t@j = codegen.idx(%B_t, %j)
   %c = codegen.idx(%C@i, %j)
   %arow&bcol = codegen.zip(%A@i,%B_t@j)
   %init = rise.embed() {
     %cst_0 = constant 0.0 : f32
     return(%cst_0) : (f32) -> ()
   }
   codegen.assign(%init, %c)
   affine.for %k = 0 to 2048 {
     %a&b = codegen.idx(%arow&bcol, %k)
     %a = codegen.fst(%a&b)
     %b = codegen.snd(%a&b)
     %result = rise.embed(%a, %b, %c) {
       %0 = mulf %a, %b : f32
       %1 = addf %0, %c : f32
       return(%1) : (f32) -> ()
     }
     codegen.assign(%result, %c)
   }
  }
 }
 return
```

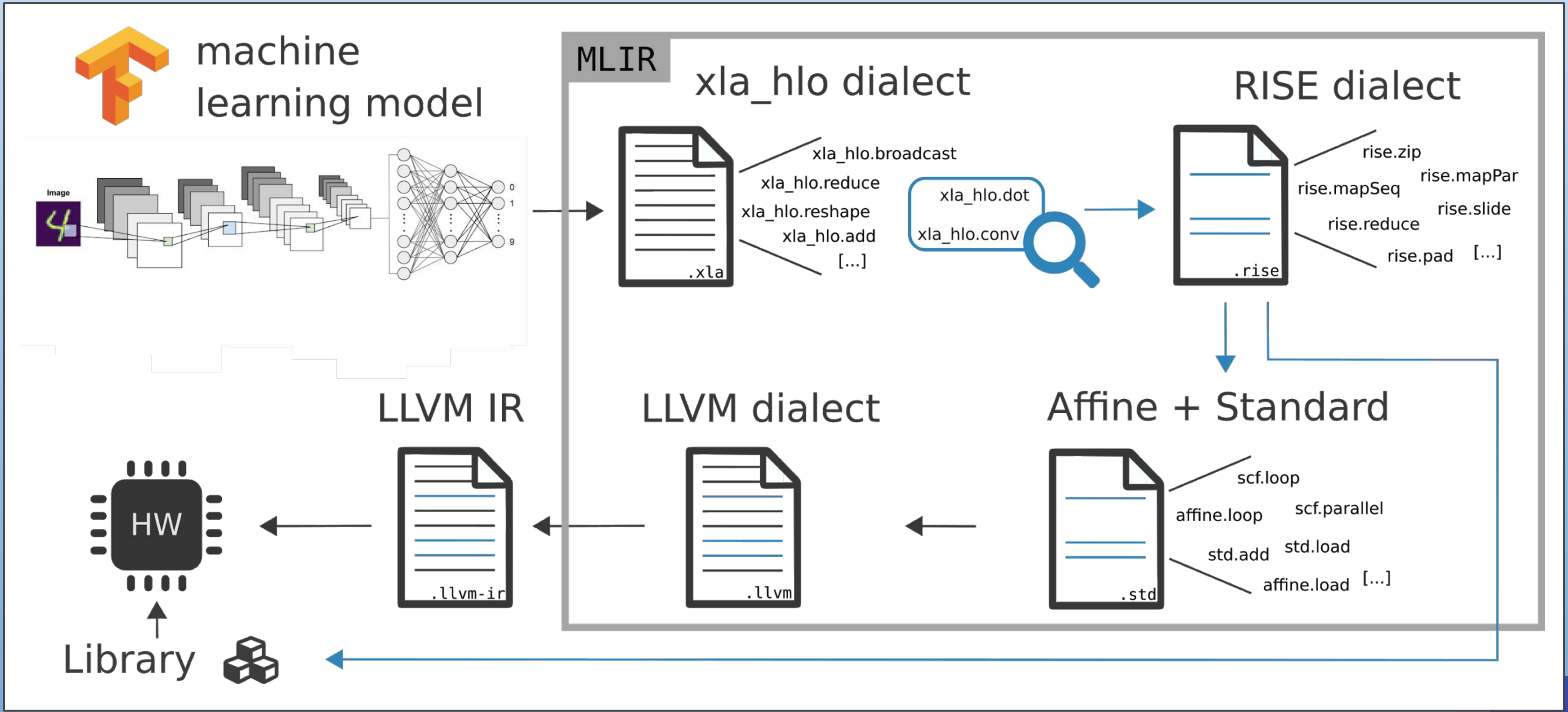
Matrix Multiplication in imperative RISE

2. Lowering imperative to target representation

```
%init = constant 0.0 : 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 %outC[%i, %j]
      %0 = mulf %a, %b : f32
      %1 = addf %0, %c : f32
      affine.store %1, %outC[%i, %j]
    }
  }
}
return
```

Matrix Multiplication in Affine+Standard

Lowering XLA HLO using RISE



Today we have

1. High-level Functional Pattern-based Representation RISE **in MLIR**
2. End-to-end lowering: XLA HLO → RISE → LLVM-IR

Next steps

- Optimizing RISE via rewriting: based on our promising prior academic work on Lift

  → Implement composable rewrite system in MLIR based on our ICFP 2020 paper

Fork me on GitHub

RISE

A Functional Pattern-based MLIR 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