RISE
A functional pattern-based language in MLIR

Martin Lücke | Michel Steuwer | Aaron Smith

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

Machine Learning Systems are Stuck in a Rut

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:

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, but for our purposes it is sufficient to consider a simplified form of the inner loop, which is
Why RISE?

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

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

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on 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:

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

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

1.1 New ideas often require new primitives
We won’t discuss the full details of Capsule networks in this paper¹, but for our purposes it is sufficient to consider a simplified form of the inner loop, which is
We should aim for more principled higher level intermediate representations
**RISE by example: Matrix Multiplication**

```plaintext
fun(A : N.K.float ℚ) ⇒ fun(B : K.M.float ℚ) ⇒
  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: \[
\sum arow_i \ast bcol_i
\]
**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/](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) ))))))
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

Rewrite Rules

map(f, A) ⇔ join(map(map(f), split(n, A)))
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!

Performance close or better than hand-tuned library code

[GPGPU’16]
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  - Performing polyhedral optimizations
  - Targeting GPUs  - 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   - Performing polyhedral optimizations
  - Targeting GPUs               - 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

```text
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>
  %4 = hlo.dot (%3, %kernel_2) : (tensor<1x128xf32>, tensor<128x10xf32>) -> tensor<1x10xf32>
  %5 = hlo.add %4, %bias_2 : (tensor<1x10xf32>, tensor<10xf32>) -> tensor<1x10xf32>
  return %5 : tensor<1x10xf32>
}
```
How can we achieve end-to-end integration with existing solutions?
%A \map (\text{fun}(\text{arow} \Rightarrow \\
\text{B} \map (\text{fun}(\text{bcol} \Rightarrow \\
\text{zip(aro, bcol) \map(*)} \map(\text{reduce(+, 0)}))))}

\begin{verbatim}
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, array<1024, scalar<f32>>) -> !rise.array<1024, scalar<f32>> {
    %f2 = rise.lambda (%bcol: !rise.array<1024, 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.scalar<f32>
      rise.return %result : !rise.scalar<f32>
    }
  }
  %map = rise.mapSeq #rise.nat<1024> #rise.array<1024, 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.scalar<f32>
  rise.return %result : !rise.scalar<f32>
}
\end{verbatim}
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
        } : !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
      } : !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, %f1, %B
  rise.out %out ← %result
  return
}
```
**Types**

**Data Types**

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

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

**Function Types**

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

```
8  %f = rise.lambda (%t : !rise.tuple<scalar<f32>, scalar<f32>>) → scalar<f32>  
// %f : !rise.fun<tuple<scalar<f32>, scalar<f32>> → scalar<f32>>
```
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>> {
    %res = mulf %arow, %arow : f32
    return %res : f32
  }
  %mapA = rise.mapSeq %f1, %A
  %result = rise.apply %mapA, %f1, %A
  rise.out %out -- %result
  return
}

%zip = rise.zip %rise.nat<1024> %rise.scalar<f32> %rise.scalar<f32>
%mapB = rise.mapSeq %zip, %B, %out, %out
%result = rise.apply %mapB, %f2, %B
rise.return %result : !rise.array<1024, scalar<f32>>
Patterns: zip

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

Each RISE pattern is implemented as an MLIR operation.

Operations customized with attributes specifying information for the type.

Patterns encoded as operations have a RISE function type.
Function application

4
5
6
7
%zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
Function application

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

- `rise.apply` models function application:
  - expects an SSA value with a \texttt{RISE function type} (%zip)
  - and arguments to the function (%arow, %bcol)
Function application

4  %f1 = rise.lambda (%arow : !rise.array<1024, scalar<f32>>) →
   !rise.array<1024, scalar<f32>> {
5  %f2 = rise.lambda (%bcol : !rise.array<1024, scalar<f32>>) →
   !rise.scalar<f32> {
6   %zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
7   %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

Matrix Multiplication in RISE

Matrix Multiplication in Affine + Standard

1 func @mm(%outArg, %inA, %inB) {  
2   %A = in %inA  
3   %B = in %inB  
4   %mfun = lambda(%row) -> array<2048, scalar<f32>> {  
5     %n2fun = lambda(%col) -> scalar<f32> {  
6       %zipFun = zip %mat<2048> %scalar<f32> %scalar<f32>  
7       %zipedArrays = apply %zipFun, %row, %col  
8       %reductionLambda = lambda(%tuple, %acc) -> scalar<f32>{  
9         %fstFun = fst %scalar<f32> %scalar<f32>  
10        %sndFun = snd %scalar<f32> %scalar<f32>  
11        %first = apply %fstFun, %tuple  
12        %second = apply %sndFun, %tuple  
13        %result = embed(%first, %second, %acc) {  
14          %product = mul %first, %second : f32  
15          %result = addf %product, %acc : f32  
16        return %result : f32  
17      }  
18      return %result : scalar<f32>  
19    }  
20    %init = literal #lit<0.0>  
21    %reduceFun = reduceSeq %nat<2048>  
22    %tuple<scalar<f32>>, scalar<f32>>> %scalar<f32>  
23    %result = apply %reduceFun, %reductionLambda,  
24    %init, %zipedArrays  
25    return %result : scalar<f32>  
26  }  
27  %m2 = mapSeq %nat<2048> %array<2048, scalar<f32>>  
28  %scalar<f32>  
29  %result = apply %m2, %mfun, %B  
30  return %result : array<2048, scalar<f32>>  
31 }  
32 %m1 = mapSeq %nat<2048> %array<2048, scalar<f32>>  
33 %array<2048, scalar<f32>>  
34 %result = apply %m1, %m1fun, %A  
35 out %outArg <- %result  
36 return  
37 }

1 func @mm(%outArg, %inA, %inB) {  
2   %init = constant 0.0000000e+00 : f32  
3   affine.for %x1 = 0 to 2048 {  
4     affine.for %k = 0 to 2048 {  
5       affine.store %init, %outArg[%x1, %k]  
6       affine.for %k = 0 to 2048 {  
7         %a = affine.load %inA[%x1, %k]  
8         %b = affine.load %inB[%k, %k]  
9         %c = affine.load %outArg[%x1, %k]  
10        %x1 = mulf %a, %b : f32  
11        %x2 = addf %a, %c : f32  
12        affine.store %x2, %outArg[%x1, %k]  
13      }  
14    }  
15    return  
16  }
Lowering of the RISE dialect

1. Lowering functional to imperative representation

Matrix Multiplication in RISE

Matrix Multiplication in imperative RISE
Lowering of the RISE dialect

1. Lowering functional to imperative representation

Matrix Multiplication in imperative RISE

2. Lowering imperative to target representation

Matrix Multiplication in Affine+Standard
RISE end-to-end integration

The diagram illustrates the flow from a machine learning model through various intermediate representations (IRs) and dialects to hardware (HW).

1. **Machine Learning Model**: This is represented by TensorFlow, which generates an MLIR file.
2. **MLIR**: The MLIR file contains operations like `xla_hlo.broadcaster`, `xla_hlo.reduce`, and `xla_hlo.add`. These operations are then converted into RISE dialect.
3. **RISE Dialect**: The RISE dialect includes operations such as `rise.zip`, `rise.mapSeq`, and `rise.pad`.
4. **Affine + Standard Dialect**: This dialect contains operations like `sct.loop`, `affine.loop`, and `affine.load`.
5. **LLVM**: The final step is to convert the IR to an LLVM file, preparing it for hardware execution.

The diagram shows the transformation process from high-level operations to low-level instructions, suitable for hardware execution.
RISE end-to-end integration
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.return %result: !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.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

Lowering RISE to library calls
func @mm(%out:memref<1024x1024xf32>, %A:memref<1024x1024xf32>, %B:memref<1024x1024xf32>) {
    %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<1024x1024xf32>, i32, memref<1024x1024xf32>, i32, f32, memref<1024x1024xf32>, 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 0.0 0.1 0.0 0.0]
```
We are Open Source!

https://rise-lang.org/mlir

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

Martin Lücke | Michel Steuwer | Aaron Smith