Integrating a Functional Pattern-Based IR into MLIR

Martin Lücke
University of Edinburgh
United Kingdom
martin.luecke@ed.ac.uk

Michel Steuwer
University of Edinburgh
United Kingdom
michel.steuwer@ed.ac.uk

Aaron Smith
University of Edinburgh / Microsoft
USA
aaron.smith@microsoft.com

Abstract

The continued specialization in hardware and software due to the end of Moore's law forces us to question fundamental design choices in compilers, and in particular for domain specific languages. The days where a single universal compiler intermediate representation (IR) was sufficient to perform all important optimizations are over. We need novel IRs and ways for them to interact with one another while leveraging established compiler infrastructures.

In this paper, we present a practical implementation of a functional pattern-based IR in the SSA-based MLIR framework. Our IR captures the program semantics as compositions of common computational patterns enabling rewrite-based optimizations. We discuss the integration with other IRs by demonstrating the compilation of a neural network represented as a TensorFlow graph down to optimized LLVM code via our functional pattern-based IR. Our implementation demonstrates for the first time a practical integration of a functional pattern-based IR with other IRs and it enables the construction of sophisticated code generators for domain specific languages.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Parallel programming languages.*

Keywords: Intermediate Representation, MLIR, Rise

ACM Reference Format:
Martin Lücke, Michel Steuwer, and Aaron Smith. 2021. Integrating a Functional Pattern-Based IR into MLIR. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3446804.3446844

1 Introduction

Software and hardware is becoming increasingly specialized. Only a few years ago the focus was on optimizing single threaded performance represented by SPEC benchmarks for general purpose CPUs. Now interest has shifted away from general purpose compute into application specific domains – none more prominent than deep learning. Deep learning hardware today is represented by CPUs, GPUs, FPGAs and a zoo of specialized hardware devices such as Google's TPU.

Traditional compiler designs evolved around the idea of a *"one size fits all"* single intermediate representation (IR) that unifies the representation of multiple high-level programming languages and allows optimizations to use a single representation for generating optimized code for different hardware targets. The highly successful LLVM compiler infrastructure [16] is built around this idea.

But LLVM IR provides a single level of abstraction that is often too low-level and difficult for optimizations to exploit the rich high-level semantics of domain specific languages. For this reason many higher level IRs have been developed for specific domains such as machine learning (e.g, the graph-based IRs of TensorFlow [4], PyTorch [20], and TVM [9]) or image processing (e.g., Halide [22]), but also more generic higher level IRs such as INSPIRE [15] and Thorin [19].

Lift [26] is an unconventional higher level IR that represents computations as compositions of functional patterns. This design is easily extended to new application domains for example by adding new patterns to support stencil computations [14]. Optimizations are expressed as semantics preserving rewrite rules that are either applied automatically as part of a stochastic search process [25] or their application can be controlled by a developer precisely [13]. Using this approach Lift has demonstrated high-performance on linear algebra and stencil codes used in machine learning and physical simulations for hardware architectures ranging from multi-core CPUs to mobile- and desktop-class GPUs.

Employing a novel IR, such as Lift, in practical end-to-end toolchains is a challenging task. The recently proposed MLIR project [17] seems a promising solution as it aims to provide a common framework to enable the integration of multiple IRs. Individual IRs are implemented as *dialects* following a basic SSA-based design. Each dialect can define a custom type system and operations as well as optimization passes. For interaction among dialects MLIR provides common infrastructure to facilitate the translation from one dialect to another.

But how do we encode a functional pattern-based IR such as Lift in the SSA-based MLIR framework? Thanks to Appel [5] we know for a long time about a direct correspondence between functional programming and SSA, but how does this look like in practice? How does a functional IR integrate with other IRs in MLIR? While our functional IR is convenient for expressing domain specific computations at a high-level, will we pay a performance penalty when compiling to imperative loop-based code?

In this paper, we are answering these questions. We present a practical implementation of the functional pattern-based IR Rise [13] as an MLIR dialect. We discuss its implementation (Section 3) and its integration with other MLIR dialects (Section 4) to build a practical end-to-end code generation solution for machine learning that progressively lowers the representations from a domain-specific TensorFlow graph to our generic high-level functional pattern-based representation before lowering it to a lower level polyhedral loop-based representation and eventually to LLVM IR. Our evaluation (Section 5) shows that our implementation of the functional pattern-based IR introduces negligible compile time overhead and generates code with no runtime overhead.

2 Motivation and Background

2.1 What's Wrong with Existing Compiler IRs?

Specialized domain-specific compilers have become an integral component of achieving high-performance in many crucial application domains such as machine learning. But according to Paul Barham and Michael Isard, two of the original authors of TensorFlow, machine learning systems are stuck in a rut [7]. They argue that while TensorFlow and similar frameworks enabled great advances in machine learning, their current design and implementations focus on a fixed set of monolithic and inflexible kernels (such as matrix-multiplication) that are expressed as fixed nodes in the IR. They continue to say that the *"reliance on high performance but inflexible kernels reinforces the dominant style of programming model"* and argue that *"these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress"* in machine learning.

To overcome these problems, we need new intermediate representations that break up the monolithic and inflexible kernels and represent computations using more flexible and finer grained building blocks. Pattern-based IRs are built around this idea and are an interesting sweet spot between specialized high-level IRs with monolithic domain-specific abstractions and low-level loop-code or three-address style IRs similar to LLVM. In pattern-based IRs, computations are represented at a high level as compositions of generic computational patterns common across many domains. This allows to easily perform algorithmic optimizations at the right abstraction level as demonstrated by Lift [24, 26] that encodes optimizations as rewrites of pattern-based programs.

2.2 RISE: A Functional Pattern-Based IR

Rise [13] is a functional pattern-based IR in the style of Lift. Rise provides a set of data-parallel high-level patterns that are composed to describe computations over higher dimensional arrays (i.e. tensors) such as matrix multiplication. For that, the **`map`** pattern is used twice (in lines 2 and 3) to apply the dot product computation to each combination of a row of matrix `A` and a column of matrix `B`:

```
1 fun( (A: N.K.f32, B: K.M.f32),
2  A |> map(fun(arow,
3   B |> transpose |> map(fun(bcol,
4    zip(arow,bcol) |> map(mult) |> reduce(add,0)))))))
```

Rise uses the lambda calculus to compose patterns. Lambdas (i.e., anonymous functions) are written `fun(x, e)` and applying `x` to the function `f` is written as `x |> f` (alternatively as `f(x)`). The type of parameters of top-level lambda expressions are annotated. In the example matrix `A` is a $N\times K$-matrix of float values with type `N.K.f32` (line 1).

High-level programs are gradually transformed into low-level programs by applying semantics preserving rewrite rules that encode optimization and implementation decisions. For example, fusing **`map`** and **`reduce`** to ensure they are computed in a single loop is expressed with this rule:

```
map(f) |> reduce(⊕,0) ↦ reduceSeq(fun((a,x), a ⊕ f(x)),0)
```

Prior work on Rise [13] has shown that compiler optimizations such as tiling and vectorization are expressible as compositions of rewrites. The closely related Lift project has demonstrated high performance by exploring optimization choices encoded as rewrite rules for tensor-algebra [25], stencil computations and kernel convolutions [14].

2.3 End-to-End Compilers by Integration of IRs

Rise (and Lift) are implemented in the Scala programming language, making the development of end-to-end compiler solutions e.g. for machine learning challenging. Even for IRs implemented in easier to integrate languages such as C++, e.g., INSPIRE [15] or Thorin [19], integration is hard in practice due to the code required to convert between IRs.

We are interested in developing end-to-end compiler solutions for machine learning to allow the development of novel optimizations and analyses by leveraging the best of what is available from other domain experts.

This paper presents a C++ based implementation of Rise in MLIR and shows how to build an integrated compiler solutions using this novel implementation. Figure 1 shows a prototype machine learning compiler that we have built with the technologies described in this paper. Starting from an unmodified TensorFlow machine learning model (top left), e.g., for handwriting detection using the popular MNIST dataset, the model is directly encoded in the existing MLIR XLA dialect (top middle). Computational intensive operations such as matrix multiplication encoded as `xla_hlo.dot` operations are lowered into the MLIR implementation of Rise (top right).

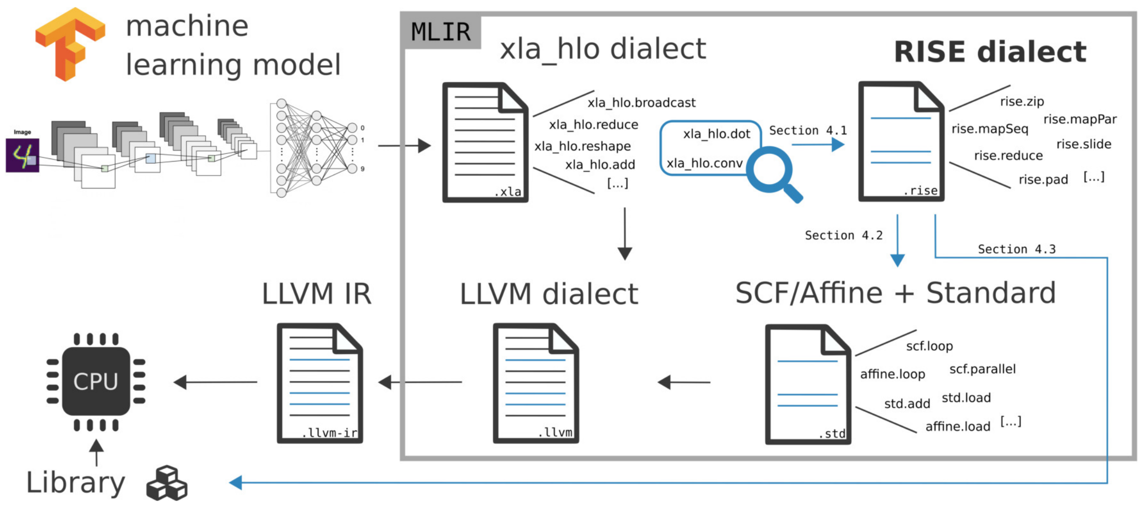


Figure 1. End-to-end code generation using the pattern-based Rɪsᴇ intermediate representation implemented as an ᴍʟɪʀ dialect. A TensorFlow machine learning model (top left) is represented in ᴍʟɪʀ with the XLA HLO dialect. Supported operators are lowered into compositions of computational patterns in Rɪsᴇ (top right) before being lowered to a loop based representation (bottom right) that is compiled to ʟʟᴠᴍ IR (bottom left). Or alternatively, Rɪsᴇ patterns are lowered directly into library code.

Rɪsᴇ expressions are then transformed into lower level IRs such as the polyhedral affine dialect and the standard dialect (bottom right) from which eventually LLMV IR is generated (bottom left). Crucially, we can easily leverage existing optimizations – at the domain-specific level as well as perform polyhedral optimizations and ultimately performing classical compiler optimizations before code generation – while building a foundation to drop in rewrite-based optimizations at the pattern-based level.

The remainder of the paper discusses the technical details of how we achieved this integration by starting with a discussion on how to implement the functional Rɪsᴇ IR in the SSA-based ᴍʟɪʀ framework in the following section.

3 Rɪsᴇ as an ᴍʟɪʀ Dialect

ᴍʟɪʀ [17] is a framework for implementing custom IRs, performing optimizations on them and converting them to other ᴍʟɪʀ IRs. It uses static single assignment (SSA) [10, 23] as the base representation in which each use of an IR value is dominated by its definition. This property guarantees values in the IR are always defined before they are used. SSA-based IRs – including ʟʟᴠᴍ IR – usually use a special ϕ node to model joins in the control flow graph (CFG) for IR values. Instead, ᴍʟɪʀ uses a functional SSA-form inspired by the observations of Appel [5], where terminator nodes pass values into block arguments defined by the successor block.

An IR node in MLIR may contain regions allowing for nested representations. IR nodes in MLIR are called *operations* (rather than *instructions* in ʟʟᴠᴍ IR). Operations are characterized by a type and are allowed to produce multiple results. An ᴍʟɪʀ dialect may define a set of custom operations and types, along with passes to transform the IR.

$$
\begin{array}{llr}
T ::= T \to T \mid D & & \textit{Function and Data Types} \\
D ::= N.D \mid D \times D \mid S & & \textit{Array, Tuple, and Scalar Types} \\
S ::= \texttt{i32} \mid \texttt{f32} \mid \ldots & & \textit{standard Scalar Types} \\
N ::= 0 \mid \ldots \mid N + N \mid N * N \mid \ldots & & \textit{Array Length}
\end{array}
$$

Figure 2. Grammar of Rɪsᴇ types

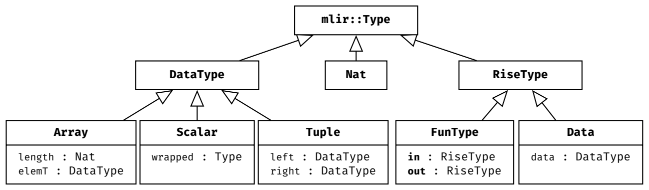
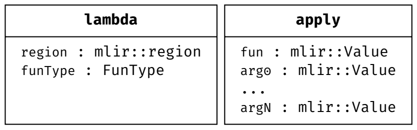


Figure 3. Rɪsᴇ dialect types

3.1 RISE Types

Rɪsᴇ is a functional IR where computations are represented by compositions of pattern applications. Patterns are represented as built-in functions and have a corresponding function type. The grammar of Rɪsᴇ types is shown in Figure 2. Function types are separate from data types only allowing data types to be stored to memory. Rɪsᴇ data types include array, tuple, and scalar types. For arrays, their length is tracked in the type. Different to many tensor representations, higher-dimensional tensors are represented by nesting array types such as the matrix type $N.M.\texttt{f32}$ from the prior example.

The grammar directly corresponds to the class hierarchy shown in Figure 3. The Rɪsᴇ dialect follows the ᴍʟɪʀ notation to prefix types by `!rise`. For example, a function type with one argument and return type is written: `!rise.fun<scalar<i32> -> scalar<i32>>`. Note, `i32` is an existing ᴍʟɪʀ standard type and it is reused by wrapping it in a `!rise.scalar` type.



```
lambda
region : mlir::region
funType : FunType

apply
fun : mlir::Value
arg0 : mlir::Value
...
argN : mlir::Value
```

Figure 4. Implementation of **lambda** and **apply** classes

3.2 RISE Lambda Calculus as MLIR Operations

RISE is a functional IR based on the lambda calculus with two core operations: function abstraction and application. We represent these with the **lambda** and **apply** operations.

Figure 4 shows their implementations as subclasses of the `mlir::Op` superclass. The **lambda** operation together with its associated `FunType` models a lambda expression by wrapping an MLIR region containing a single block whose arguments become the arguments of the lambda expression. The **apply** operation models a function application (or function call). The function can be any value with a `FunType`, such as **lambda**s or patterns. The argument must have a matching RISE type.

The listing below shows how **lambda** and **apply** are used to represent a call to the identity function: `y |> fun(x => x)`

```
%id  = rise.lambda (%x)    : !rise.fun<scalar<i32> ->
                                       scalar<i32>> {
    rise.return %x         : !rise.scalar<i32>
}
%res = rise.apply %id, %y : !rise.scalar<i32>
```

3.3 RISE Patterns as MLIR Operations

Computations in RISE are represented by computational patterns such as **map**, **reduce** and **zip**. Each pattern is represented by an operation and implemented as a subclass of the `mlir::Op` class. Implementation sketches are shown in Figure 5.

Each pattern in RISE has a function type. For example, **map**:

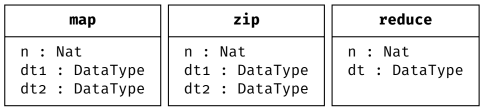
```
map : (n: Nat) ↦ (dt1: DataType) ↦ (dt2: DataType) ↦
        (dt1 → dt2) → n.dt1 → n.dt2
```

The ↦ arrow represents a special function type for representing polymorphism, i.e. for introducing type variables. The MLIR RISE dialect does not support polymorphic function types and instead provides the type arguments when a pattern is created and stores them in the pattern classes. This design results in patterns having monomorphic function types that are straightforward to handle. Applying **map** to a function `%f` and an array of 1024 `int`s is represented as:

```
%m   = rise.map #nat<1024> #scalar<i32> #scalar<i32>
   : !rise.fun< fun< scalar<i32> -> scalar<i32> > ->
          fun< array<1024, scalar<i32>>            ->
               array<1024, scalar<i32>> > >
%res = rise.apply %m, %f, %array
   : !rise.array<1024, scalar<i32>>
```

3.4 Matrix Multiplication in the RISE MLIR Dialect

Figure 6 shows an example matrix multiplication in the RISE dialect. This corresponds to the functional RISE expression seen earlier after the **map**-**reduce** fusion rule has been applied. Some type annotations have been removed for readability.

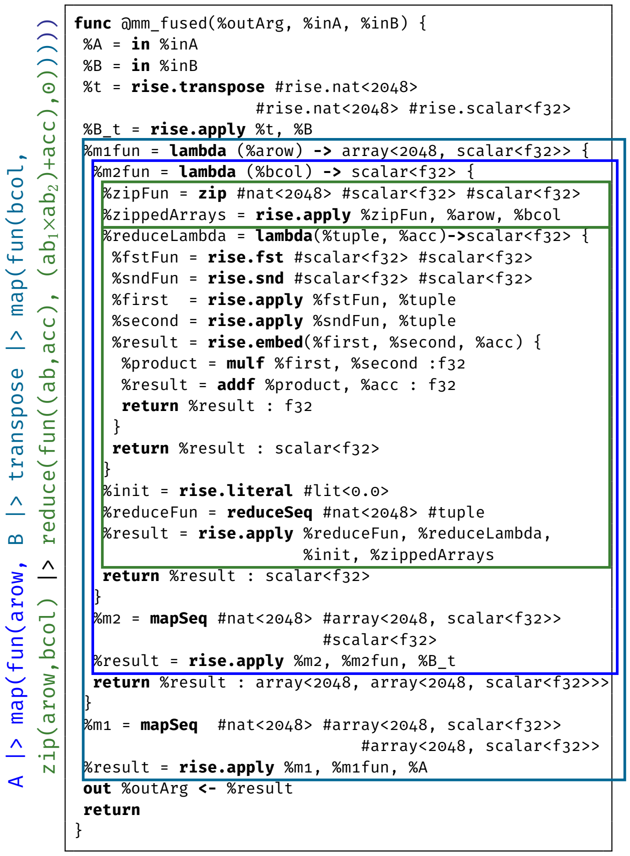


```
map
n : Nat
dt1 : DataType
dt2 : DataType

zip
n : Nat
dt1 : DataType
dt2 : DataType

reduce
n : Nat
dt : DataType
```

Figure 5. Implementation sketches of pattern classes



```
A |> map(fun(arow, B |> transpose |> map(fun(bcol,
  zip(arow,bcol) |> reduce(fun((ab,acc), (ab₁×ab₂)+acc),0)))))
```

```
func @mm_fused(%outArg, %inA, %inB) {
 %A = in %inA
 %B = in %inB
 %t = rise.transpose #rise.nat<2048>
                     #rise.nat<2048> #rise.scalar<f32>
 %B_t = rise.apply %t, %B
 %m1fun = lambda (%arow) -> array<2048, scalar<f32>> {
  %m2fun = lambda (%bcol) -> scalar<f32> {
   %zipFun = zip #nat<2048> #scalar<f32> #scalar<f32>
   %zippedArrays = rise.apply %zipFun, %arow, %bcol
   %reduceLambda = 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 = reduceSeq #nat<2048> #tuple
   %result = rise.apply %reduceFun, %reduceLambda,
                        %init, %zippedArrays
   return %result : scalar<f32>
  }
  %m2 = mapSeq #nat<2048> #array<2048, scalar<f32>>
                          #scalar<f32>
  %result = rise.apply %m2, %m2fun, %B_t
  return %result : array<2048, array<2048, scalar<f32>>>
 }
 %m1 = mapSeq  #nat<2048> #array<2048, scalar<f32>>
                              #array<2048, scalar<f32>>
 %result = rise.apply %m1, %m1fun, %A
 out %outArg <- %result
 return
}
```

Figure 6. 2048x2048 matrix multiplication in the RISE dialect

The dot product computation is defined in the innermost (green) boxes with the **zip** and **reduceSeq** patterns. This computation is nested inside two **lambda**s (`%m1fun` and `%m2fun`) that are used as arguments when calling the **map** patterns and applying them to matrix `%A` and the transposed matrix `%B_t`.

The multiplication and addition performed on the scalar values is represented using the standard MLIR operations **mulf** and **addf**. These are nested inside of an **rise.embed** operation that makes the named MLIR values with RISE `ScalarType`s (here: `%first`, `%second`, and `%acc`) available inside the nested block with their types unwrapped (here with type `f32`). The **rise.in** and **rise.out** operations allow the integration with external values that have non RISE types.

3.5 Building RISE IR in C++

When using our RISE dialect as a tool it is tedious to construct the RISE IR from scratch supplying type information for all operations explicitly. To simplify building RISE expressions we have developed an easy to use C++ API using

```
makeRiseProgram(C, A, B)([&](Value A, Value B) {
  return mapSeq([&](Value arow) {
    return mapSeq([&](Value bcol) {
      return reduceSeq([&](Value tuple, Value accum){
        return embed({fst(tuple), snd(tuple), accum},
              [&](Value a, Value b, Value accum) {
                return accum + (a * b);  });
      },literal(scalarF32(), "0.0"),zip(arow, bcol));
    }, transpose(B));
  }, A);  });
```

Listing 1. C++ API to build a composition of Rise patterns modelling matrix multiplication as shown in Figure 6

MLIR builders. Listing 1 shows how this API is used for building the Rise IR code shown in Figure 6. The **makeRiseProgram** function accepts the output value (C) and the input values (A, B) for the computation as arguments and handles generation of the **rise.in** and **rise.out** operations. Each highlighted C++ function builds the corresponding operations without the need to specify types explicitly as they are inferred. We use C++ lambda expressions to build the **rise.lambda** operations. The **rise.apply** operations are inserted automatically.

4 Integration with other MLIR Dialects

The MLIR infrastructure allows us to easily integrate Rise with other dialects. In this section, we are going to discuss the integration that allow the building of a full machine learning compiler by compiling a TensorFlow machine learning model via our functional pattern-based dialect into low-level loop-based code. First, We will discuss how to lower domain specific dialects into Rise. Then, we will discuss how a program in the Rise dialect is lowered into a loop-based representation. As an alternative we discuss the lowering from Rise directly to optimized library implementations.

4.1 Lowering Domain Specific Dialects to RISE

Rise is an attractive target for many high-level domain specific dialects. The high-level Rise patterns are easy to compose to flexibly express a wide variety of computations. Conveniently, providing lower level details such as memory allocation of temporaries or committing to an inherent sequential or parallel implementation early is not necessary.

Let us consider machine learning as a popular domain: The MLIR XLA_HLO dialect encodes TensorFlow graphs representing the computation of a neural network. Operations in this graph represent computations processing tensors, such as the `xla_hlo.dot` operation that represents a tensor dot product. Depending on the tensors' dimensionality this represents a vector dot product, a matrix multiplication, or their higher dimensional equivalent.

To lower XLA operations to Rise, the MLIR infrastructure allows for defining declarative rewrites that automatically match specific operations and sequences of operations as a starting point to modifying the IR. For lowering the `xla_hlo.dot` operation we provide an MLIR rewrite that checks for the types of the tensors to determine the equivalent Rise expression that is emitted to replace the operation. For the two dimensional matrix multiplication we use the C++ builder API to generate the Rise IR (Listing 1). Currently, our implementation supports lowering `xla_hlo.dot` and convolutions which are two of the most computational intensive operations in machine learning. Our C++ builder API and the MLIR declarative rewriting makes it straightforward to target Rise from other domain specific MLIR dialects.

4.2 Lowering RISE to Loop-Based Dialects

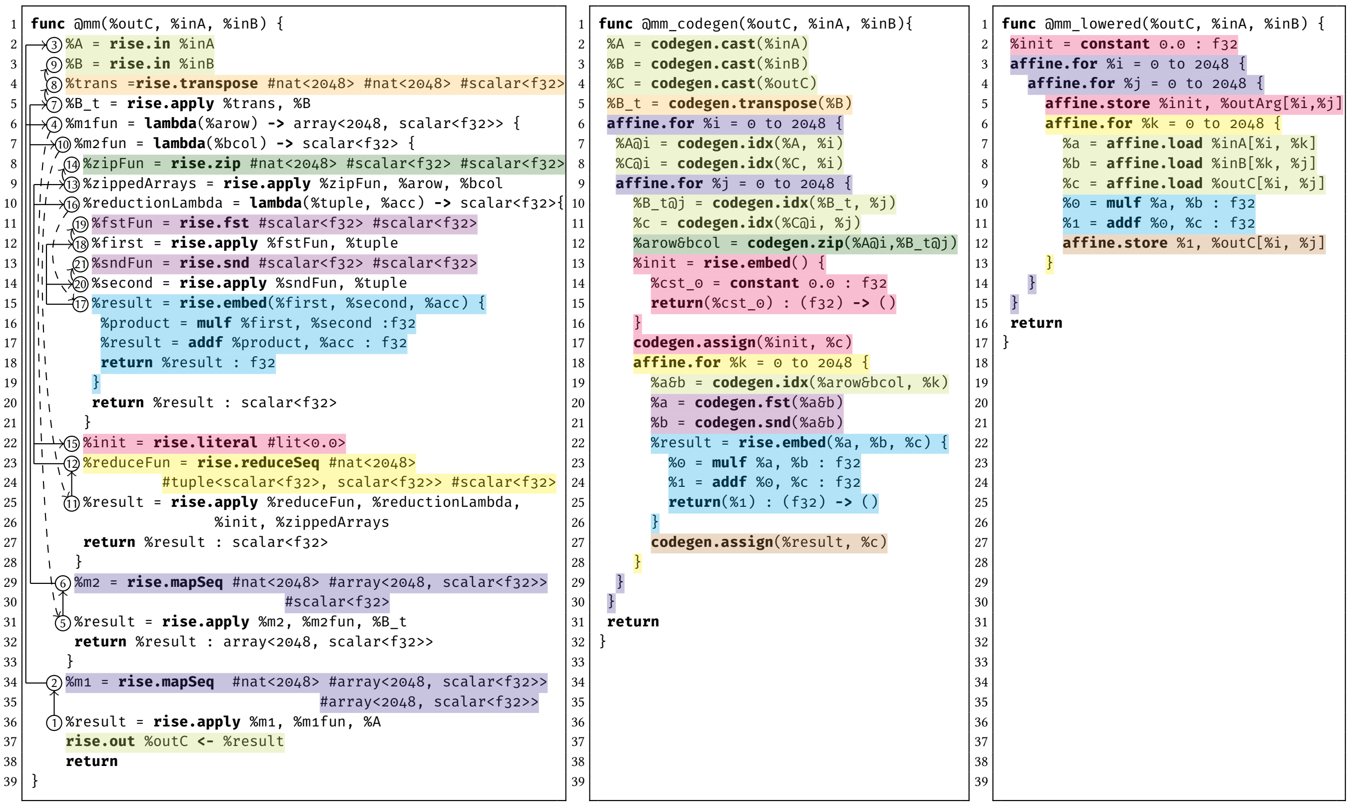
The functional patterns in Rise provide a convenient abstraction to express computations at a high level. Before execution these functional patterns have to be lowered into imperative code and eventual LLVM instructions that can easily be compiled to executable code. For this process we gradually lower the MLIR Rise dialect into a loop-based MLIR dialect from which LLVM IR is generated. We adopt a formal compilation method for translating a functional pattern-based IR to imperative code originally presented in [6]. This lowering process is complete and capable of lowering all possible Rise expressions into loop code. It is split in two phases:

1. **Functional → Intermediate IR**:
   The functional Rise IR is lowered into an intermediate imperative representation without the functional lambda calculus representations and the functional patterns. This intermediate representation still uses Rise specific types and has not yet resolved indices for accessing memory.
2. **Intermediate IR → Target Representation**: The indexing into multi-dimensional arrays is resolved.

Splitting the lowering into these two phases greatly improves the composability of the implementation. While the process is fully generic, we explain it by example to be more approachable without requiring a background in functional programming. Figure 7 shows the lowering of matrix multiplication in the Rise dialect (7a) to a combination of the loop-based Affine dialect and the standard MLIR dialect (7c) via the intermediate imperative representation (7b). The colors show which parts of the left-hand side are translated into which parts in the middle and on the right-hand side.

4.2.1 Phase 1: Functional → Intermediate IR.

In this first lowering phase we eliminate all operations modelling the functional lambda calculus and patterns from the program. Patterns such as **mapSeq** and **reduceSeq** are transformed into loops, while patterns such as **zip** and **fst** are rearranged to model the multidimensional indexing of memory. For each pattern there exist translation rules that explain how the pattern is lowered. These rules are detailed in [6]. Lambda expressions and function application nodes that make the control flow in Rise explicit are removed as the control flow is now expressed as an imperative program performing a sequence of loop-based computations.



```
1  func @mm(%outC, %inA, %inB) {
2  ③ %A = rise.in %inA
3  ⑨ %B = rise.in %inB
4  ⑧ %trans =rise.transpose #nat<2048> #nat<2048> #scalar<f32>
5  ⑦ %B_t = rise.apply %trans, %B
6  ④ %m1fun = lambda(%arow) -> array<2048, scalar<f32>> {
7   ⑩ %m2fun = lambda(%bcol) -> scalar<f32> {
8    ⑭ %zipFun = rise.zip #nat<2048> #scalar<f32> #scalar<f32>
9    ⑬ %zippedArrays = rise.apply %zipFun, %arow, %bcol
10   ⑯ %reductionLambda = lambda(%tuple, %acc) -> scalar<f32>{
11    ⑲ %fstFun = rise.fst #scalar<f32> #scalar<f32>
12    ⑱ %first = rise.apply %fstFun, %tuple
13    ㉑ %sndFun = rise.snd #scalar<f32> #scalar<f32>
14    ⑳ %second = rise.apply %sndFun, %tuple
15    ⑰ %result = rise.embed(%first, %second, %acc) {
16          %product = mulf %first, %second :f32
17          %result = addf %product, %acc : f32
18          return %result : f32
19        }
20        return %result : scalar<f32>
21      }
22   ⑮ %init = rise.literal #lit<0.0>
23   ⑫ %reduceFun = rise.reduceSeq #nat<2048>
24                #tuple<scalar<f32>, scalar<f32>> #scalar<f32>
25   ⑪ %result = rise.apply %reduceFun, %reductionLambda,
26                    %init, %zippedArrays
27      return %result : scalar<f32>
28    }
29  ⑥ %m2 = rise.mapSeq #nat<2048> #array<2048, scalar<f32>>
30                                   #scalar<f32>
31  ⑤ %result = rise.apply %m2, %m2fun, %B_t
32    return %result : array<2048, scalar<f32>>
33   }
34 ② %m1 = rise.mapSeq  #nat<2048> #array<2048, scalar<f32>>
35                                      #array<2048, scalar<f32>>
36 ① %result = rise.apply %m1, %m1fun, %A
37    rise.out %outC <- %result
38    return
39 }
```

(a) Functional Rise IR

```
1  func @mm_codegen(%outC, %inA, %inB){
2   %A = codegen.cast(%inA)
3   %B = codegen.cast(%inB)
4   %C = codegen.cast(%outC)
5   %B_t = codegen.transpose(%B)
6   affine.for %i = 0 to 2048 {
7    %A@i = codegen.idx(%A, %i)
8    %C@i = codegen.idx(%C, %i)
9    affine.for %j = 0 to 2048 {
10     %B_t@j = codegen.idx(%B_t, %j)
11     %c = codegen.idx(%C@i, %j)
12     %arow&bcol = codegen.zip(%A@i,%B_t@j)
13     %init = rise.embed() {
14       %cst_0 = constant 0.0 : f32
15       return(%cst_0) : (f32) -> ()
16     }
17     codegen.assign(%init, %c)
18     affine.for %k = 0 to 2048 {
19       %a&b = codegen.idx(%arow&bcol, %k)
20       %a = codegen.fst(%a&b)
21       %b = codegen.snd(%a&b)
22       %result = rise.embed(%a, %b, %c) {
23         %0 = mulf %a, %b : f32
24         %1 = addf %0, %c : f32
25         return(%1) : (f32) -> ()
26       }
27       codegen.assign(%result, %c)
28     }
29    }
30   }
31   return
32 }
```

(b) Imperative Rise IR + Affine IR

```
1  func @mm_lowered(%outC, %inA, %inB) {
2   %init = constant 0.0 : f32
3   affine.for %i = 0 to 2048 {
4     affine.for %j = 0 to 2048 {
5       affine.store %init, %outArg[%i,%j]
6       affine.for %k = 0 to 2048 {
7         %a = affine.load %inA[%i, %k]
8         %b = affine.load %inB[%k, %j]
9         %c = affine.load %outC[%i, %j]
10        %0 = mulf %a, %b : f32
11        %1 = addf %0, %c : f32
12        affine.store %1, %outC[%i, %j]
13      }
14    }
15  }
16  return
17 }
```

(c) Affine IR

Figure 7. Lowering of matrix multiplication from Rise (left) via the intermediate imperative representation (middle) to the affine loop-based dialect (right). Colors indicate which part from the left is lowered into which part in the middle and right.

For multi-dimensional memory accesses of Rise values at this intermediate stage, we introduce a number of intermediate operations that are not exposed to the user:

- **rise.codegen.idx**
- **rise.codegen.fst**
- **rise.codegen.zip**
- **rise.codegen.slide**
- **rise.codegen.assign**
- **rise.codegen.snd**
- **rise.codegen.transpose**
- **rise.codegen.pad**

Lowering by example: Matrix Multiplication. We explain the lowering process by example following the matrix multiplication. Figure 7 shows on the left the matrix multiplication represented in the Rise dialect. To lower this to the intermediate imperative representation in the middle, we start at the end of the Rise program with the **rise.out** operation that specifies the value representing the computed result. From here the lowering process chases all referred SSA values (as indicated by arrows in Figure 7 (a)) and visits the defining operation in the highlighted order form ① to ㉑. As Rise programs are compositions of pattern and function applications, the referenced operations are mostly **rise.apply** nodes whose lowering depends on the specific pattern or lambda which is applied, as well as it's arguments.

For lowering the entire matrix multiplication example we first lower the ① **rise.apply** node in line 36 that was referred to by **rise.out** operation and represents the call to the outermost **rise.mapSeq** ② in the functional expression.

Lowering mapSeq. Listing 2 shows the pseudo code for lowering an application of the **mapSeq** pattern. These steps are performed (The numbering matches the line numbers in Listing 2):

1 First, the input array is lowered. Here this is ③ %A represented as a **rise.in** operation that is translated into the **codegen.cast** operation in line 2 Figure 7 (b).

```
  // op = rise.apply(mapSeq n s t, fun, input)
  void lowerAndStore(ApplyOp op, Value out) {
1  Value in = lower(op.input);
2  generate_affine_for(0, op.n, inc, [&](index i){
3   Value inAtI = idx_gen(in, i);
4   Value outAtI = idx_gen(out, i);
5   lowerAndStore(rise.apply(op.fun,inAtI),outAtI);});}
```

Listing 2. Pseudo code for translating the **mapSeq** pattern

2 A for loop is generated by building an **affine.for** operation (line 6 Figure 7 (b)) with the `generate_affine_for` helper function using the array length captured in the Rise type as the iteration bound of the loop.

3&4 The loop index is used to generate **codegen.idx** operations (lines 7 and 8 Figure 7 (b)) representing indexing into the already translated input and output values.

5 Finally, a temporary **rise.apply** is created representing the application of the indexed input value (`%A@i`) to the lambda (④ `%m1fun`). Then the lowering of this temporary **apply** (described next) is invoked using the indexed output value (`%C@i`) as the output location to write to.

Lowering lambda. To lower the application of a **lambda** operation we first substitute the block argument of the lambda expression with the values that are applied to it. In the example, we substitute `%arow`, the parameter of the **lambda** (line 6 Figure 7 (a)) with `%A@i` the argument created in the prior step. After the substitution, we start the lowering from the **return** operation traversing the nested block backwards.

In this example, we encounter the ⑤ **rise.apply** operation in line 31 Figure 7 (a) that represents another application of the **mapSeq** pattern ⑥. To lower this operation we just recursively invoke the `lowerAndStore` function as discussed before that visits nodes ⑦ – ⑩. The only noticeable difference is that we pass `%C@i` as the output value to write to, ultimately resulting in a multi-dimensional indexing expression that is resolved in the second phase of our lowering. Inside of the second **lambda** we encounter ⑪ the application of the **reduceSeq** pattern ⑫ in line 25 Figure 7 (a) for which we describe the lowering next.

Lowering reduceSeq. Listing 3 shows the pseudo code for lowering application of the **reduceSeq** pattern in which the following steps are processed line-by-line:

1 First the input array is lowered. In the example, this is ⑬ the result of the application of **zip** ⑭ with arguments `%arow` and `%bcol` (line 9 Figure 7 (a)). By applying the lambdas to the indexed matrices these have been substituted by `%A@i` and `%B_t@j`. We emit a **codegen.zip** with these arguments in line 12 Figure 7 (b).

2 Next, the initialization of the reduction is lowered. In the example, this is ⑮ a **rise.literal** operation with the float value `0.0` specified as an attribute. Literals are simply lowered to the corresponding **constant** operation of the standard MLIR dialect (line 14 Figure 7 (b)). The type of the **constant** operation is the MLIR standard `f32` type. To make this accessible as a Rise type we generate an enclosing **rise.embed** operation (line 13 Figure 7 (b)).

3 To initialize the reduction accumulator we generate a **rise.codegen.assign** operation representing the assignment of the initialization value to the **out**put value that we use as storage for the accumulator value. In the example, the output value at this stage in the lowering is `%c` (line 11, Figure 7 (b)) that represents an individual element of the output matrix C.

4 Then the reduction loop is generated (line 18 Figure 7 (b)) using the input array length as the upper bound.

5 The loop index is then used to generate **codegen.idx** operation (line 19 Figure 7 (b)) indexing into the already translated input that in the example represents the zipped row and column (`%arow&bcol`).

6 Finally, a temporary **rise.apply** is created representing the application of the indexed input value and the accumulator to the reduction lambda ⑯. Then the lowering of this temporary **apply** is invoked using the accumulator as the output location to write to.

```
  // op = rise.apply(reduceSeq n s t, fun, init, input)
  void lowerAndStore(ApplyOp op, Value out) {
1  Value in = lower(op.input);
2  Value init = lower(op.init);
3  assign_gen(init, out);
4  generate_affine_for(0, op.n, inc, [&](index i) {
5   Value inAtI = idx_gen(in, i);
6   lowerAndStore(rise.apply(op.fun,inAtI,out),out);
7  }); }
```

Listing 3. Pseudo code for translating the **reduceSeq** pattern

Lowering embed. Nested inside the `%reductionLambda` is an **embed** operation ⑰ that is lowered next. The lowering is performed in two steps:

1 **embed**'s arguments (line 15 Figure 7 (a)) are lowered:
- The accumulator `%acc` is already lowered at this point.
- `%first` and `%second` are applications (⑱ and ⑳) of the **rise.fst** ⑲ and **rise.snd** ㉑ operations. Their lowering produces the intermediate **rise.codegen.fst** and **rise.codegen.snd** operations (lines 20 & 21 Figure 7 (b)).

2 **embed** remains unchanged and a **codegen.assign** is generated (line 27 Figure 7 (b)) representing writing the computed result to the indicated output.

Now all **rise.apply** operations with patterns such as **mapSeq** and **reduceSeq**, as well as all **lambda** and **embed** operations have been lowered. Only `rise.codegen` operations remain as can be seen in Figure 7 (b). These are resolved to indices next.

4.2.2 Phase 2: Intermediate IR → Target IR.

In this second phase the multi-dimensional indexing is resolved, generating standard MLIR `load` and `store` operations.

Figure 9 visualizes the process of generating the load of a value of matrix B (line 8 Figure 7 (c)). Starting from the `%b` argument of the **rise.embed** operation in line 22 Figure 7 (b) we chase the def-use chains of the SSA-values and construct an *access path* shown on the right of Figure 9.

For every operation encountered on the def-use chain traversal we modify the access path as indicated in Figure 8. Figure 8a shows the path changes when resolving a load and Figure 8b for a store. There are a few Rise patterns (such

```
resolveIndex(cast(%val, type), path)                {  generateLoad(%val, path);                   }
resolveIndex(embed(%args, region), path)            {  resolveIndex(%args, path); inline(region);  }
resolveIndex(idx(%array,%iv), path)                 {  resolveIndex(%array, %iv :: path);          }
resolveIndex(zip(%lhs, %rhs), %iv::(fst|snd)::path) {  resolveIndex((%lhs|%rhs), %iv :: path);     }
resolveIndex(fst(%tuple), path)                     {  resolveIndex(%tuple, fst :: path);          }
resolveIndex(snd(%tuple), path)                     {  resolveIndex(%tuple, snd :: path);          }
resolveIndex(split(%array, n), i :: j :: path)      {  resolveIndex(%array, i * n + j :: path);    }
resolveIndex(join(%array, n), i :: path)            {  resolveIndex(%array, i/n :: i%n :: path);   }
resolveIndex(transpose(%array), i :: j :: path)     {  resolveIndex(%array, j :: i :: path);       }
resolveIndex(slide(%array, stride), i :: j :: path) {  resolveIndex(%array, i * stride + j :: path); }
resolveIndex(pad(%array, n, l, r), i :: path)       {  resolveIndex(%array,
                                                        ((i < l) ? 0 : (i < l + n) ? index : n − 1) :: path);  }
```

(a) Resolve load indexing of `rise.codegen` operations

```
resolveStoreIndex(assign(%val, %storeTo))            {  resolveIndex(%val, {});
                                                        resolveStoreIndex(%storeTo, {});                 }
resolveStoreIndex(cast(%val, type), path)            {  generateStore(%val, path);                       }
resolveStoreIndex(idx(%array, %iv), path)            {  resolveStoreIndex(%array, %iv :: path);          }
resolveStoreIndex(split(%array, n), i :: path)       {  resolveStoreIndex(%array, i/n :: i%n ::path);    }
resolveStoreIndex(join(%array, m), i :: j :: path)   {  resolveStoreIndex(%array, i * m + j :: path);    }
resolveStoreIndex(transpose(%array), i :: j ::path)  {  resolveStoreIndex(%array, j :: i :: path);       }
```

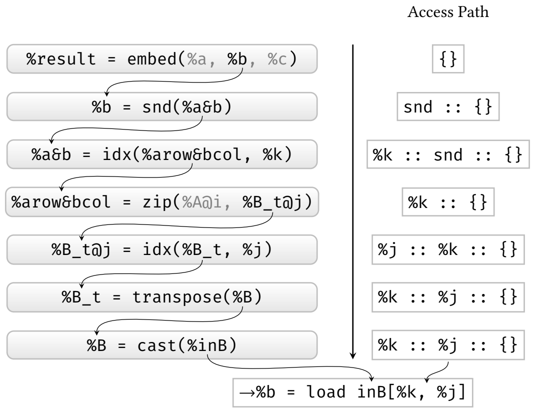
(b) Resolve store indexing of `rise.codegen` operations

Figure 8. Resolving indices and generating `load` and `store` operations by consuming the access path.

as **zip**) that only influence the reading and, therefore, only appear when generating loads.

In Figure 9 when encountering the **snd** operation we add **snd** to the path. For `idx` we add the index to the path. For **zip** we look at the path to decide if we follow the def-use of the first or second argument: we observe **snd** on the path and visit the second argument. After adding another index onto the path, **transpose** simply flips the order of the first two indices on the path. When we hit `cast` we are leaving the Rise dialect and have reached the end of our index computation. We generate the `load` operation shown in line 8 Figure 7 (c) using the remaining information on the access path.

We perform a similar process for generating `store` operations using the recursive functions described in Figure 8b. `store` operations are generated when encountering an `assign` operation such as in line 27 Figure 7 (b) that generates the `store` operation in line 12 Figure 7 (c).



```
%result = embed(%a, %b, %c)          {}
%b = snd(%a&b)                       snd :: {}
%a&b = idx(%arow&bcol, %k)           %k :: snd :: {}
%arow&bcol = zip(%A@i, %B_t@j)       %k :: {}
%B_t@j = idx(%B_t, %j)               %j :: %k :: {}
%B_t = transpose(%B)                 %k :: %j :: {}
%B = cast(%inB)                      %k :: %j :: {}
                 →%b = load inB[%k, %j]
```

Figure 9. Resolving indices for reading from matrix B

The generated `load` and `store` operations have replaced all remaining Rise operations leaving us with the final imperative target representation as shown in Figure 7 (c).

4.3 Lowering RISE to Library Code

One of the strengths of the pattern-based Rise IR is the ability to easily detect larger computations represented as compositions of patterns - such as matrix multiplication - without the need to perform analysis e.g. of index arithmetic. For many common computations there exists high performance library code that we want to directly leverage.

As an alternative to the lowering process discussed before, we might also start from our Rise mlir dialect by detecting a composition of patterns that corresponds to a computation provided by a high-performance library. Similar to the process of lowering the `xla_hlo.dot` operation into Rise, we search for a matching composition of patterns and replace them with a mlir `std.call` operation invoking the library interface directly. As a demonstration we implemented a lowering of the matrix multiplication directly to the BLAS MKL library targeting Intel CPUs.

4.4 Summary

In this section, we have discussed the integration of Rise with other mlir dialects. We have discussed how to build a practical machine learning compiler by lowering a TensorFlow graph represented in the XLA dialect to Rise and then lowering further either to loop-based representations, such as the affine dialect, or directly to library calls. Next, we are experimentally evaluating what runtime and compile time overheads we are introducing by compiling via Rise. Furthermore, we are exploring the potential of performing optimizations at the pattern-based level.

5 Evaluation

5.1 Experimental Setup

We performed an experimental evaluation on an Intel Haswell quad core i7-4790K@4.0 GHz with 64KiB L1 and 256 KiB L2 cache per core as well as 8MiB shared L3 cache. The processor supports AVX-2 (256-bit) vector operations and has a turbo boost of 4.4 GHz. For all experiments the frequency governor was set to "performance". Experiments were run 100 times and we report the median run times.

5.2 Overhead of Rise

Runtime Overhead. Figure 10 shows the runtime of matrix multiplication in milliseconds compiled via the Rise MLIR dialect compared to equivalently optimized versions without using Rise. We show two different sizes: a 1024x1024 matrix and a slightly more unusual size of 1x784 × 784x128, which is a taken from the handwritten neural network application. The SCF version is a text book version with three nested loops represented in the Structured Control Flow (SCF) MLIR dialect that is lowered to LLVM IR before `-O3` optimizations are applied. This baseline shows a negligible runtime difference compared to the RISE naive version from Figure 6 lowered to the SCF dialect and then to LLVM IR as before.

Similarly we observe no runtime overhead for the RISE opt version compared to the Affine version. This Rise version is lowered to the affine dialect as describe in Section 4. Then both versions are optimized with polyhedral loop tiling and vectorization passes as described in [8] before lowering to LLVM and applying `-O3` optimizations.

Compile Time Overhead. Figure 11 shows compilation times of MLIR passes when compiling matrix multiplication from Rise via affine to the LLVM dialect. We report the compilation time breakdown for the median total time from 10 runs. The passes are shown in order of their execution, with all verification passes summed up at the bottom. A moderate 10% of the MLIR compilation time is spent in the lowering pass from Rise to affine described in Section 4.

These results show that no runtime overhead and only a moderate compile time overhead is introduced by the functional pattern-based representation and that the lowering process described in section 4 produces efficient code. Furthermore, these results demonstrate the strength of integration within MLIR: starting from a functional representation we easily take advantage of the polyhedral optimizations resulting in a significant performance boost. Next we explore performance gains when optimizing at the functional level.

5.3 Optimizing Separate Convolutions via Rewrites

One advantage of Rise's pattern-based representation is that optimizations are easily expressed as rewrite rules [13]. This is specifically true for algorithmic optimizations that are hard to perform at a lower loop-based level.

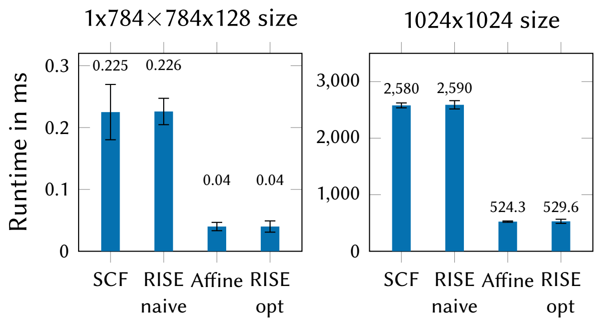


Figure 10. Single precision matrix multiplication runtime comparison. Rise introduces no overhead compared to equivalently optimized versions without using Rise.

```
            ... Compilation time report ...
===-----------------------------------------------===
--- Wall Time (s) ---   --- Name ---
   0.0002 ( 10.4%)   ConvertRiseToAffine
   0.0002 ( 10.0%)   AffineLoopTiling
   0.0003 ( 13.8%)   AffineScalarReplacement
   0.0000 (  2.4%)   AffineVectorize
   0.0001 (  7.0%)   ConvertLinalgToLoops
   0.0001 (  3.2%)   ConvertAffineToStandard
   0.0000 (  2.3%)   ConvertSCFToStandard
   0.0004 ( 21.0%)   ConvertStandardToLLVM
   0.0006 ( 29,9%)   VerificationPasses
  -------------------------
   0.0019 (100.0%)   Total
```

Figure 11. Breakdown of pass compilation time. Lowering of Rise only consumes about 10% of the overall time.

To demonstrate this we consider a 2D convolution which is an operator often used in machine learning workloads such as CNNs. A 2D convolution computes a weighted sum of a 2D neighbourhood of values of an input matrix producing an output matrix of the same size. This computational pattern is elegantly expressible in a pattern-based IR using the three patterns `pad`, `slide`, and **`map`** as discussed in [14]. Listing 4 shows the Rise C++ builder producing the Rise MLIR code for representing a 2D convolution: the `slide2D` pattern in line 10 creates the 2D neighbourhood that are processed by the `map2D` pattern in line 3. Each value in the neighbourhood is multiplied with a weight and then summed up. This computation is expressed similar to the dot product with the `zip2D` and **`reduceSeq`** patterns in line 9 and 4.

Convolution Separability. A well known optimization for 2D convolutions is to perform two 1D convolutions instead. This is based on the observation that sometimes the 2D weight matrix is decomposable, as for the Sobel filter:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

```
makeRiseProgram(out, image, weights)(
[&](Value image, Value weights) {
  return mapSeq2D([&](Value slidingWindow) {
    return reduceSeq([&](Value tuple, Value acc) {
      return embed({fst(tuple), snd(tuple), acc},
        [&](Value fst, Value snd, Value acc) {
          return acc + fst * snd; });
      }, literal(scalarF32Type(), "0.0"),
            join(zip2D(slidingWindow, weights)));
    }, slide2D(3, 1, pad2D(1, 1, image)));    });
```

Listing 4. C++ builders to generate a composition of Rise patterns modelling a 2D convolution

As this optimization is not universally applicable it is not implemented in compilers, but it is recognised as beneficial in the image processing community and applied manually.

But when building specialized compiler solutions operating on domain specific and high-level representations we are interested in enabling the application this optimization.

By exploiting the semantics of the high-level patterns of Rise we can define a sequence of program transformations as semantic preserving rewrite rules, as shown in [13], that separate the convolution computation. The crucial rewrite step describes how the weighted sum, expressed as a dot product of the 2D weights (`w2d`) and the neighborhood, is transformed into two weighted sums of the horizontal (`wH`) and vertical (`wV`) weights:

```
rule separateConv(w2d, wV, wH) =
    dot(join(w2d), join(nbh))
  ↦ nbh |> transpose |> map(dot(wV)) |> dot(wH)
```

Listing 5 shows the resulting Rise expression for the separated convolution using the C++ builder API showing the separate computations of the vertical convolution (lines 3–11) and the horizontal convolution (lines 13–21).

```
 1  makeRiseProgram(out, image, weightsH, weightsV)(
 2  [&](Value image, Value weightsH, Value weightsV) {
 3   Value vertical = mapSeq([&](Value arr) {
 4    return mapSeq([&](Value nbh) {
 5     return reduceSeq([&](Value t, Value acc) {
 6      return embed(scalarF32(),{fst(t), snd(t),acc},
 7       [&](Value a, Value b, Value acc) {
 8        return acc + a * b; });
 9     },literal(scalarF32(), "0.0"),zip(nbh, weightsV));
10    }, transpose(arr));
11   }, slide(3, 1, pad2D(1, 1, image)));
12
13   return mapSeq([&](Value arr) {
14    return mapSeq([&](Value nbh) {
15     return reduceSeq([&](Value t, Value acc) {
16      return embed(scalarF32(),{fst(t), snd(t),acc},
17       [&](Value a, Value b, Value acc) {
18        return acc + a * b; });
19     }, literal(scalarF32(), "0.0"),zip(nbh,weightsH));
20    }, slide(3, 1, arr));
21   }, vertical);                                    });
```

Listing 5. C++ builders to generate a composition of Rise patterns modelling a spatially separated 2D convolution

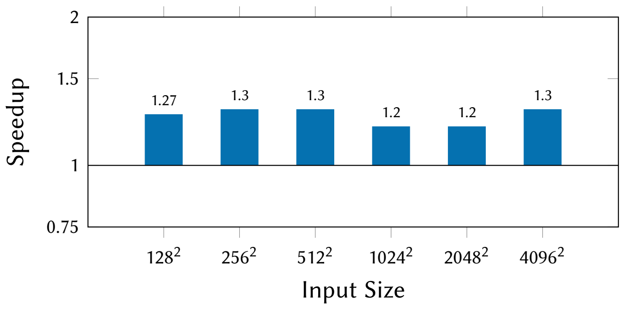


Figure 12. Runtime speedup of 2D spatially separated convolution over a naive 2D convolution

Figure 12 shows the performance gain of the separated over the non separated convolution for different input sizes. We can observe a 30% performance gain of this optimization showing one possible simple performance gain by encoding and applying a domain-specific optimization as a sequence of rewrite rules transforming the Rise IR. We are keen to explore more opportunities such as this and contribute to improve the support for implementing such compositions of rewrites in the mlir framework.

6 Conclusion

This work presented Rise – a functional pattern-based intermediate representation and its implementation in the mlir framework. We demonstrated that integration with other intermediate representations is feasible and facilitated by the ability to easily define custom types, operators and passes using mlir. Our principled IR design based on lambda calculus, rewrite rules and MLIR provides an attractive solution for building machine learning compilers. For the first time, we provide a pattern-based IR as a ready-to-use tool. Our implementation is open source and an artifact accompanying this paper is publicly available.

mlir [17] is still a young project but has nevertheless gained significant traction in the community. An ecosystem of many dialects is developing, for example, for representing TensorFlow graphs (XLA HLO [11]), structured control flow (SCF [3]), polyhedral loop nests (Affine [2]) and even circuits (CIRCT [1]). MLIR has many applications in the machine learning domain such as its integration in the TensorFlow compilation chain [21] or compiling neural networks represented in ONNX [18]. An early performance study suggests promising performance using polyhedral optimizations [8] and early work on a stencil specific dialect has been used to accelerate weather and climate simulations [12].

Acknowledgments

We would like to thank the entire Rise team for their work. This work is supported by a Google Faculty award championed by Jacques Pienaar and Albert Cohen.

References

[1] 2020. *Circuit IR Compilers and Tools (CIRCT)*. https://github.com/llvm/circt

[2] 2020. *MLIR Affine dialect online documentation*. https://mlir.llvm.org/docs/Dialects/Affine/

[3] 2020. *MLIR SCF dialect online documentation*. https://mlir.llvm.org/docs/Dialects/SCFDialect/

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://doi.org/arXiv:1603.04467

[5] Andrew W. Appel. 1998. SSA is Functional Programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20. https://doi.org/10.1145/278283.278285

[6] Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. 2017. Strategy Preserving Compilation for Parallel Functional Code. *CoRR* abs/1710.08332 (2017).

[7] Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *HotOS*. ACM, 177–183. https://doi.org/10.1145/3317550.3321441

[8] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *CoRR* abs/2003.00532 (2020).

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. USENIX Association, 578–594.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems* 13 (1991), 451–490. https://doi.org/10.1145/115372.115320

[11] Google. 2020. *XLA HLO*. https://github.com/tensorflow/mlir-hlo

[12] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2020. Domain-Specific Multi-Level IR Rewriting for GPU. *CoRR* abs/2005.13014 (2020).

[13] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. https://doi.org/10.1145/3408974

[14] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *CGO*. ACM, 100–112. https://doi.org/10.1145/3168824

[15] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: The insieme parallel intermediate representation. In *PACT*. IEEE Computer Society, 7–17. https://doi.org/10.1109/PACT.2013.6618799

[16] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[17] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*.

[18] Tung D. Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E. Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. 2020. Compiling ONNX Neural Network Models Using MLIR. *CoRR* abs/2008.08272 (2020).

[19] Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. In *CGO*. IEEE Computer Society, 202–212. https://doi.org/10.1109/CGO.2015.7054200

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc.

[21] Jacques Pienaar. 2020. MLIR in TensorFlow Ecosystem. Compilers For Machine Learning (C4ML) 2020, San Diego, CA, USA.

[22] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

[23] Fabrice Rastello et al. 2016. *SSA-Based Compiler Design* (1st ed.). Springer Publishing Company, Incorporated. http://ssabook.gforge.inria.fr/latest/book.pdf

[24] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *ICFP*. ACM, 205–217. https://doi.org/10.1145/2858949.2784754

[25] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *CASES*. ACM, 15:1–15:10. https://doi.org/10.1145/2968455.2968521

[26] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. ACM, 74–85. https://doi.org/10.1109/CGO.2017.7863730