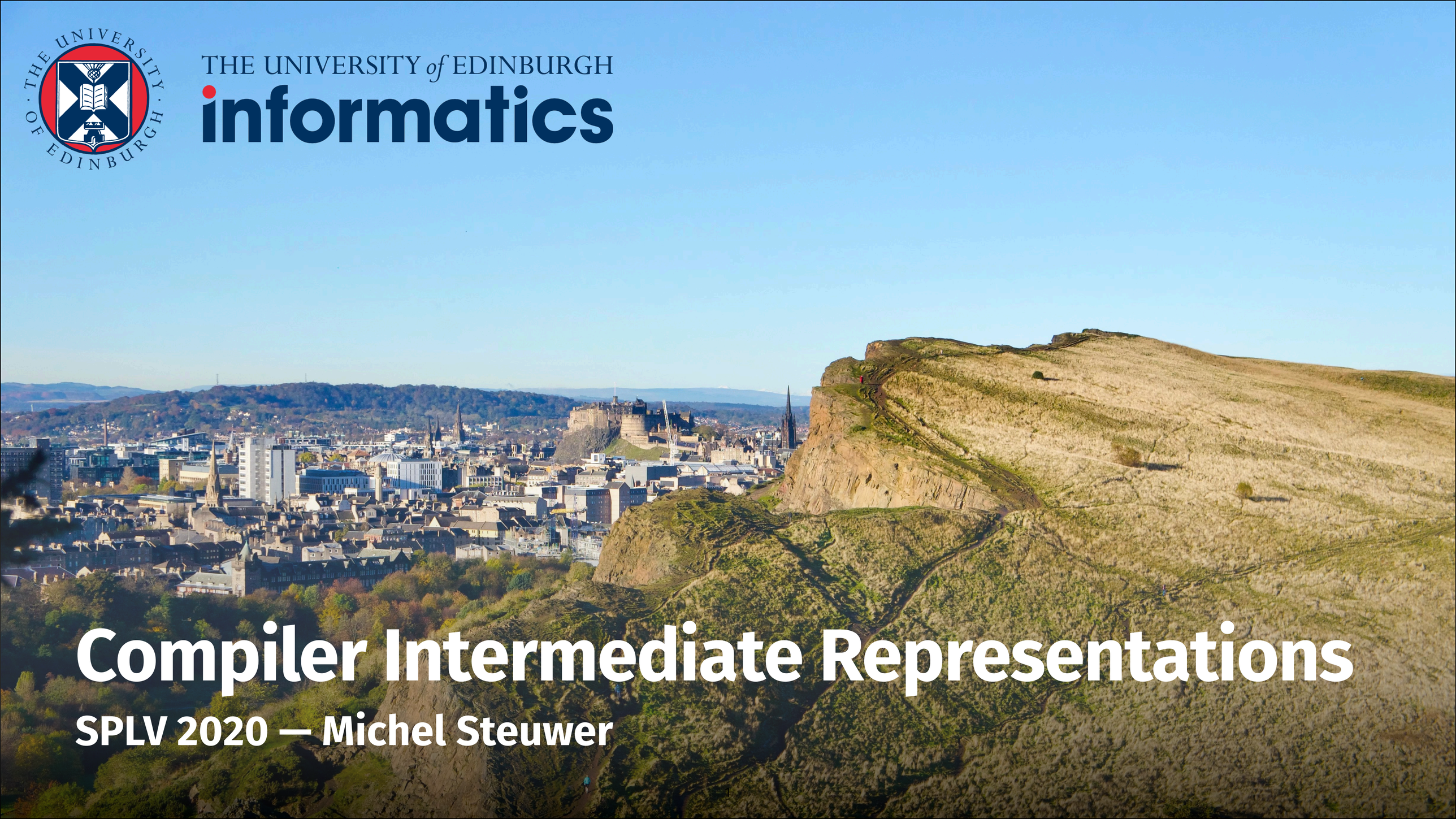


Compiler Intermediate Representations

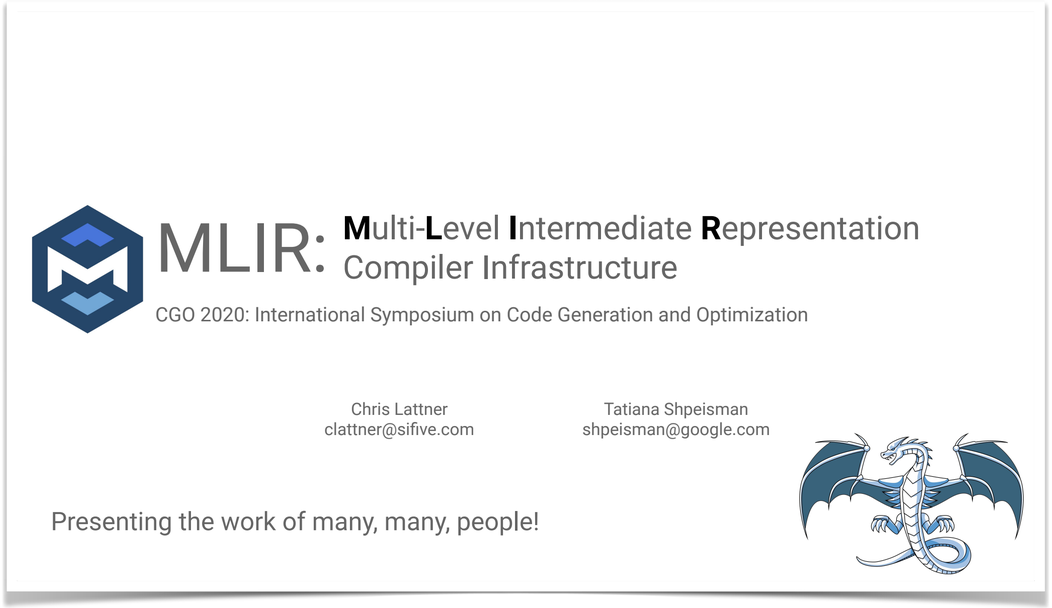
SPLV 2020 — Michel Steuwer

Outline of Lectures over the week

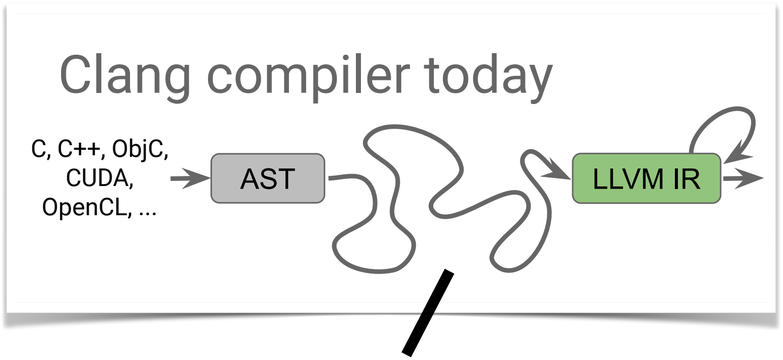
- **Tuesday:** Functional Intermediate Representations
  - Lambda Calculus and the Lambda Cube
  - Implementation Strategies for System F (ADTs across different PLs)
  - Compiler transformations as rewrite rules
- **Wednesday:** Imperative Intermediate Representations
  - Data-flow analysis
  - Control-Flow Graphs
  - Foundations of Single Static Assignment (SSA)
  - LLVM IR
- **Thursday:** Domain-Specific Intermediate Representations
  - MLIR — a compiler infrastructure for building domain-specific intermediate representations
  - Dataflow graphs — TensorFlow
  - Pattern-based (and functional) — RISE

MLIR: Multi-Level Intermediate Representation

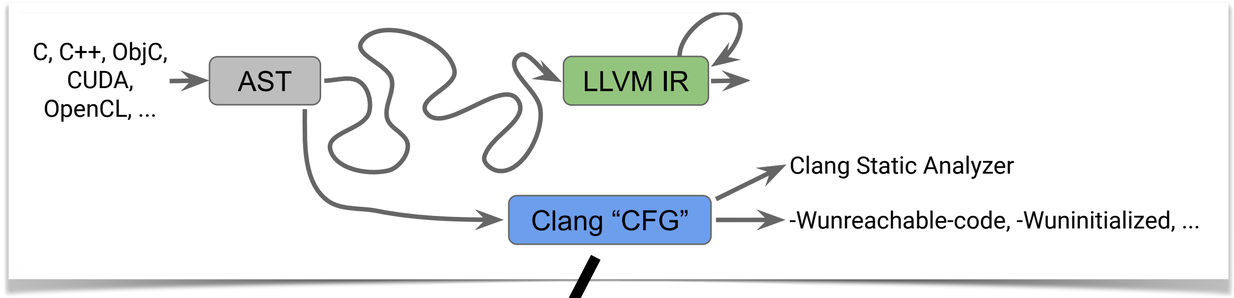
A *compiler infrastructure* to define your own intermediate representation to have it interact and integrate with other intermediate representations



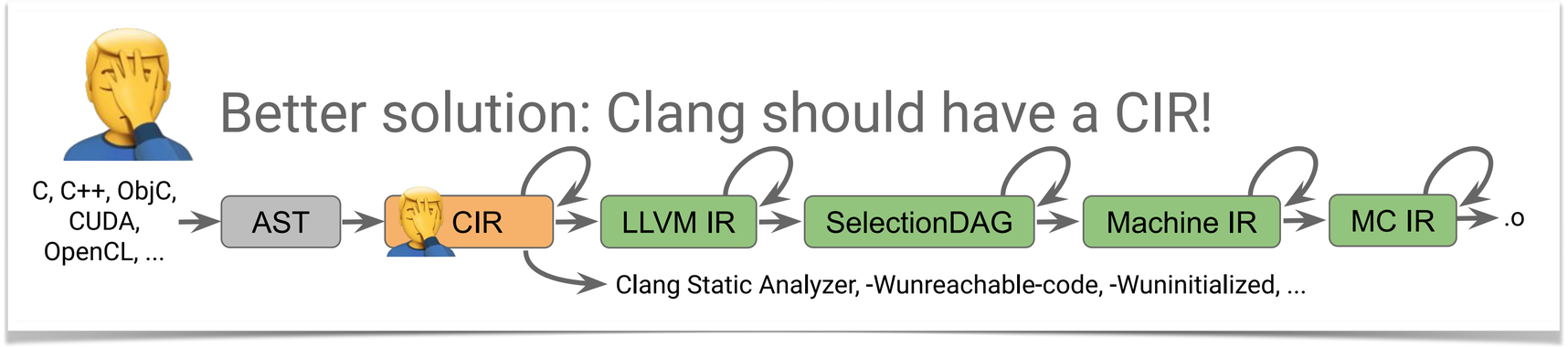
What is wrong with existing compilers? — Clang



Huge abstraction gap
between C++,... and LLVM IR

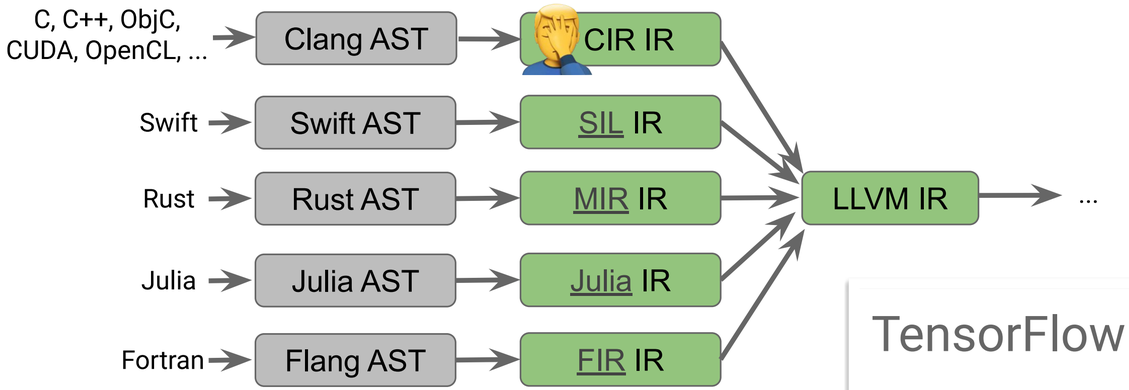


Build specialised data structures
for C/C++-specific static analysers ...

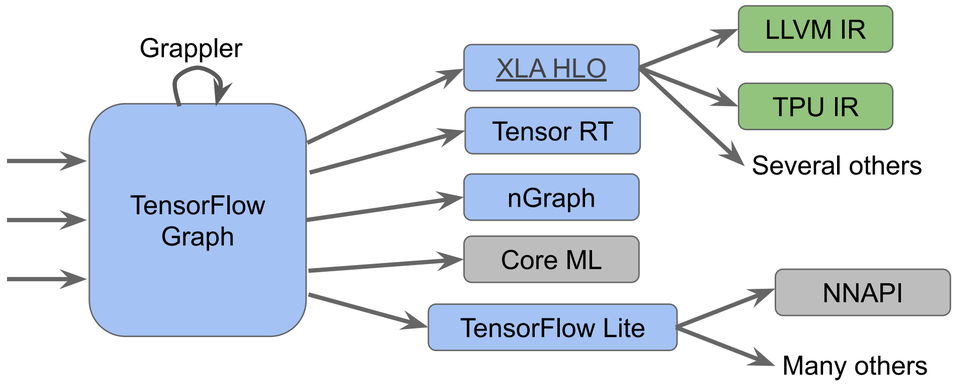


High level IRs are not just a good idea for Clang ...

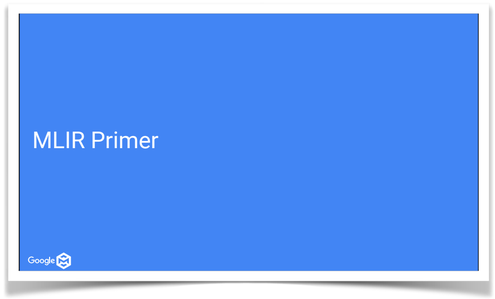
Modern languages pervasively invest in high level IRs



TensorFlow is basically a huge compiler ecosystem



A brief MLIR Tutorial

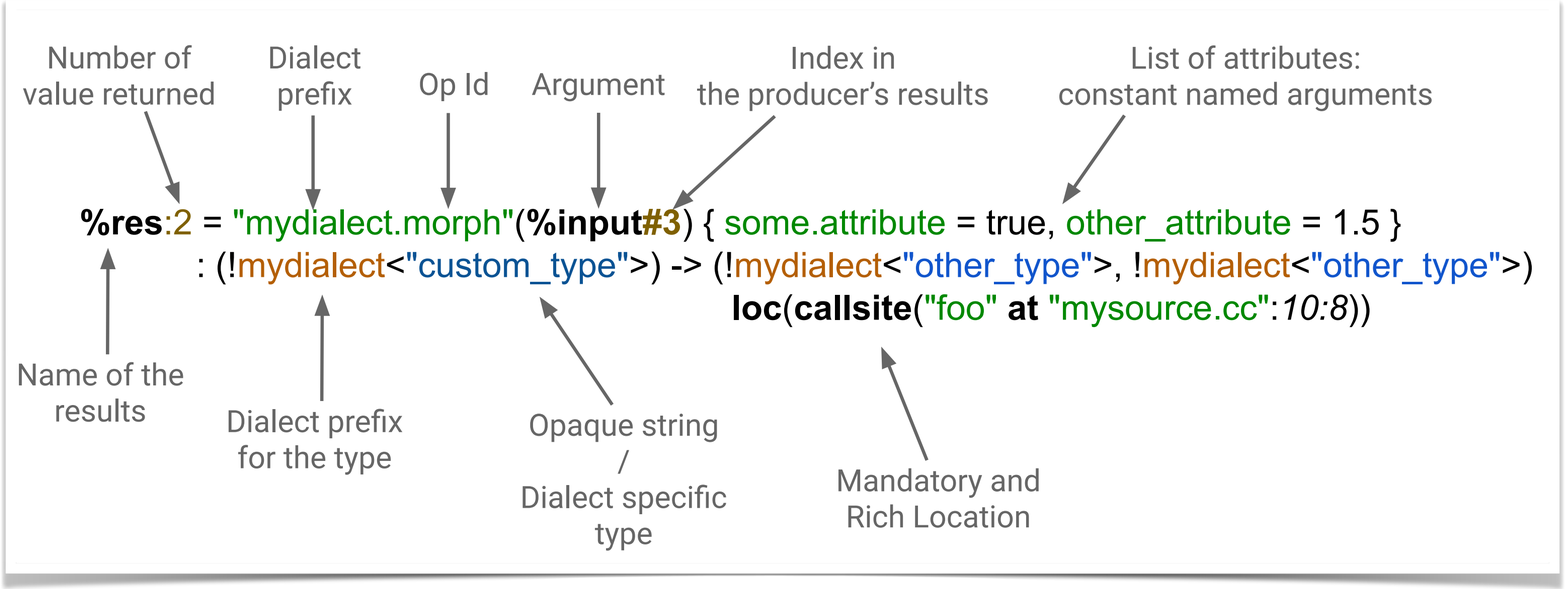


MLIR Tutorial; Jacques Pienaar, Sana Damani @ MLIR4HPC 2019

A MLIR *Dialect* defines a custom IR and comprises:

- Operations
- Types
- Passes

Operations are “opaque functions” to MLIR



Regions allow nested operations

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.        Region
  ^block(%argument: !d.type):                                  Block
    // Ops have function types (expressing mapping).
    %value = "nested.operation"() ({
      // Ops can contain nested regions.                       Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:                                                Block
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()
  })
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

Example:

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (8*d0 - 4 >= 0, -8*d0 + 7 >= 0) (%k) {
        // Dead code, because no multiple of 8 lies between 4 and 7.
        "foo"(%k) : (index) -> ()
      }
    }
  }
  return
}
```

Extra semantics constraints in this dialect: the if condition is an affine relationship on the enclosing loop indices.

Block arguments instead of ϕ nodes

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.        Region
  ^block(%argument: !d.type):                                   Block
    // Ops have function types (expressing mapping).
    %value = "nested.operation"() ({
      // Ops can contain nested regions.                        Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:                                                 Block
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()
  })
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

Instead of using ϕ nodes, MLIR uses a functional form of SSA [2] where terminators pass values into *block arguments* defined by the successor block. Each block has a (potentially empty) list of typed block arguments, which are regular values and obey SSA. The semantics of terminator Ops defines what values the arguments of the block will take after the control is transferred. For the first (entry) block of the region, the values are defined by the semantics of the enclosing Op. For example, `affine.for` uses the entry block argument `%arg4` as loop induction variable.

[Chris Lattner et. all, MLIR: A Compiler Infrastructure for the End of Moore's Law]

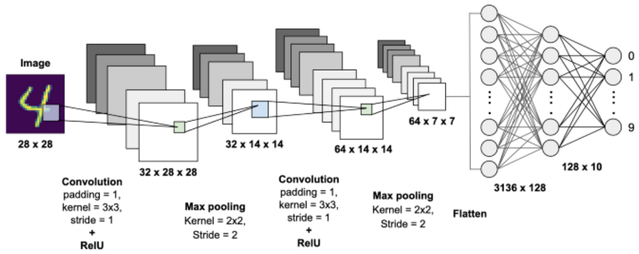
Example: Matrix Multiplication in the Affine MLIR Dialect

```
// C += A * B.
func @matmul(%A: memref<2048x2048xf64>, %B: memref<2048x2048xf64>, %C: memref<2048x2048xf64>) {
  affine.for %arg3 = 0 to 2048 {
    affine.for %arg4 = 0 to 2048 {
      affine.for %arg5 = 0 to 2048 {
        %a = affine.load %A[%arg3, %arg5] : memref<2048x2048xf64>
        %b = affine.load %B[%arg5, %arg4] : memref<2048x2048xf64>
        %ci = affine.load %C[%arg3, %arg4] : memref<2048x2048xf64>
        %p = mulf %a, %b : f64
        %co = addf %ci, %p : f64
        affine.store %co, %C[%arg3, %arg4] : memref<2048x2048xf64>
      }
    }
  }
  return
}
```

[High Performance Code Generation in MLIR: An Early Case Study with GEMM - Uday Bondhugula]

Example: TensorFlow in MLIR

MNIST

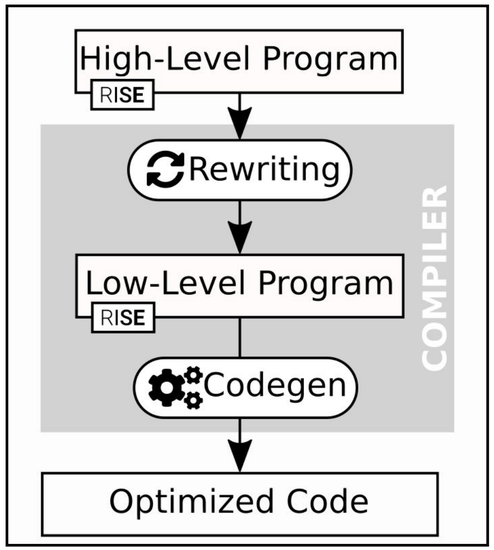


Representation via the XLA HLO Dialect:
https://github.com/tensorflow/mlir-hlo

```
module attributes {tf.versions = {bad_consumers = [], min_consumer = 12 : i32, producer = 175 : i32}} {
  flow.variable @"__iree_flow___sm_node15__model.layer-2.kernel" dense<0.5> : tensor<784x128xf32>
  flow.variable @"__iree_flow___sm_node16__model.layer-2.bias" dense<0.1> : tensor<128xf32>
  flow.variable @"__iree_flow___sm_node21__model.layer-3.kernel" dense<0.5> : tensor<128x10xf32>
  flow.variable @"__iree_flow___sm_node22__model.layer-3.bias" dense<0.1> : tensor<10xf32>
  // CHECK-LABEL: EXEC @predict
  func @predict(%arg0: tensor<1x28x28x1xf32>) -> tensor<1x10xf32> attributes {iree.module.export, iree.reflection = {abi = "sip", abiv = 1 : i32, sip =
"I8!S5!k0_0R3!_0"}, tf._input_shapes = ["tfshape$dim { size: 1 } dim { size: 28 } dim { size: 28 } dim { size: 1 }", "tfshape$unknown_rank: true",
"tfshape$unknown_rank: true", "tfshape$unknown_rank: true", "tfshape$unknown_rank: true", "tfshape$unknown_rank: true"], tf.signature.is_stateful} {
    %0 = flow.variable.address @"__iree_flow___sm_node15__model.layer-2.kernel" : !iree.ptr<tensor<784x128xf32>>
    %1 = flow.variable.address @"__iree_flow___sm_node16__model.layer-2.bias" : !iree.ptr<tensor<128xf32>>
    %2 = flow.variable.address @"__iree_flow___sm_node21__model.layer-3.kernel" : !iree.ptr<tensor<128x10xf32>>
    %3 = flow.variable.address @"__iree_flow___sm_node22__model.layer-3.bias" : !iree.ptr<tensor<10xf32>>
    %4 = mhlo.constant dense<0xFF800000> : tensor<f32>
    %5 = mhlo.constant dense<0.000000e+00> : tensor<f32>
    %6 = flow.variable.load.indirect %3 : !iree.ptr<tensor<10xf32>> -> tensor<10xf32>
    %7 = flow.variable.load.indirect %2 : !iree.ptr<tensor<128x10xf32>> -> tensor<128x10xf32>
    %8 = flow.variable.load.indirect %1 : !iree.ptr<tensor<128xf32>> -> tensor<128xf32>
    %9 = flow.variable.load.indirect %0 : !iree.ptr<tensor<784x128xf32>> -> tensor<784x128xf32>
    %10 = "mhlo.reshape"(%arg0) : (tensor<1x28x28x1xf32>) -> tensor<1x784xf32>
    %11 = "mhlo.dot"(%10, %9) : (tensor<1x784xf32>, tensor<784x128xf32>) -> tensor<1x128xf32>
    %12 = chlo.broadcast_add %11, %8 {broadcast_dimensions = dense<1> : tensor<1xi64>} : (tensor<1x128xf32>, tensor<128xf32>) -> tensor<1x128xf32>
    %13 = chlo.broadcast_maximum %12, %5 {broadcast_dimensions = dense<[]> : tensor<0xi64>} : (tensor<1x128xf32>, tensor<f32>) -> tensor<1x128xf32>
    %14 = "mhlo.dot"(%13, %7) : (tensor<1x128xf32>, tensor<128x10xf32>) -> tensor<1x10xf32>
    %15 = chlo.broadcast_add %14, %6 {broadcast_dimensions = dense<1> : tensor<1xi64>} : (tensor<1x10xf32>, tensor<10xf32>) -> tensor<1x10xf32>
    %16 = "mhlo.reduce"(%15, %4) ( {
    ^bb0(%arg1: tensor<f32>, %arg2: tensor<f32>):  // no predecessors
      %23 = mhlo.maximum %arg1, %arg2 : tensor<f32>
      "mhlo.return"(%23) : (tensor<f32>) -> ()
    }) {dimensions = dense<1> : tensor<1xi64>} : (tensor<1x10xf32>, tensor<f32>) -> tensor<1xf32>
    %17 = "mhlo.broadcast_in_dim"(%16) {broadcast_dimensions = dense<0> : tensor<1xi64>} : (tensor<1xf32>) -> tensor<1x10xf32>
    %18 = mhlo.subtract %15, %17 : tensor<1x10xf32>
    %19 = "mhlo.exponential"(%18) : (tensor<1x10xf32>) -> tensor<1x10xf32>
    %20 = "mhlo.reduce"(%19, %5) ( {
    ^bb0(%arg1: tensor<f32>, %arg2: tensor<f32>):  // no predecessors
      %23 = mhlo.add %arg1, %arg2 : tensor<f32>
      "mhlo.return"(%23) : (tensor<f32>) -> ()
    }) {dimensions = dense<1> : tensor<1xi64>} : (tensor<1x10xf32>, tensor<f32>) -> tensor<1xf32>
    %21 = "mhlo.broadcast_in_dim"(%20) {broadcast_dimensions = dense<0> : tensor<1xi64>} : (tensor<1xf32>) -> tensor<1x10xf32>
    %22 = mhlo.divide %19, %21 : tensor<1x10xf32>
    return %22 : tensor<1x10xf32>
  }
}
```

How can we represent lambda calculus in MLIR?

- We will look at the functional array programming language RISE (rise-lang.org)
- This is one of my research projects that I am working on with many students and collaborators



Matrix Multiplication in RISE

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

Data-Parallel Patterns

RISE in MLIR = λ-calculus + Patterns

- For representing λ-calculus we need:
  - FunctionTypes

```
!rise.fun<tuple<scalar<f32>, scalar<f32>> → scalar<f32>>
```

  - DataTypes

```
!rise.array<1024, array<1024, scalar<f32>>>
```

  - Abstraction

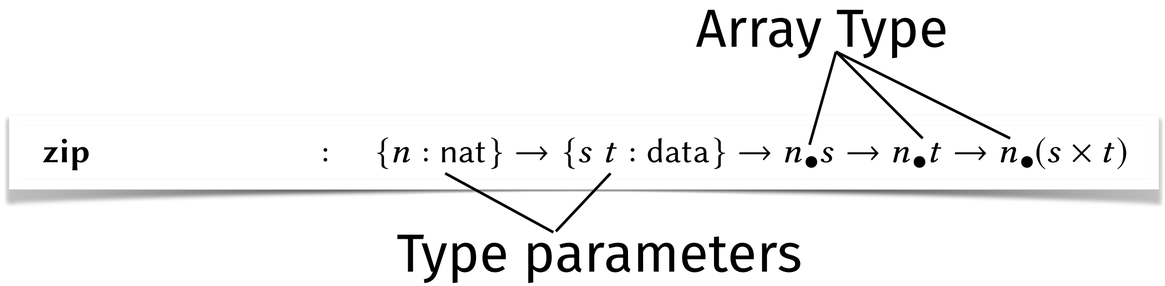
```
%add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32> {
  %result = rise.embed(%a, %b) {
    %res = addf %a, %b : f32
    return %res
  } : !rise.scalar<f32>
  rise.return %result : !rise.scalar<f32>
}
```

Associates a region with a RISE function type.

  - Application

```
%zipped = rise.apply %zip, %arow, %bcol
```

Representing patterns



In MLIR:

Passing types to type parameters

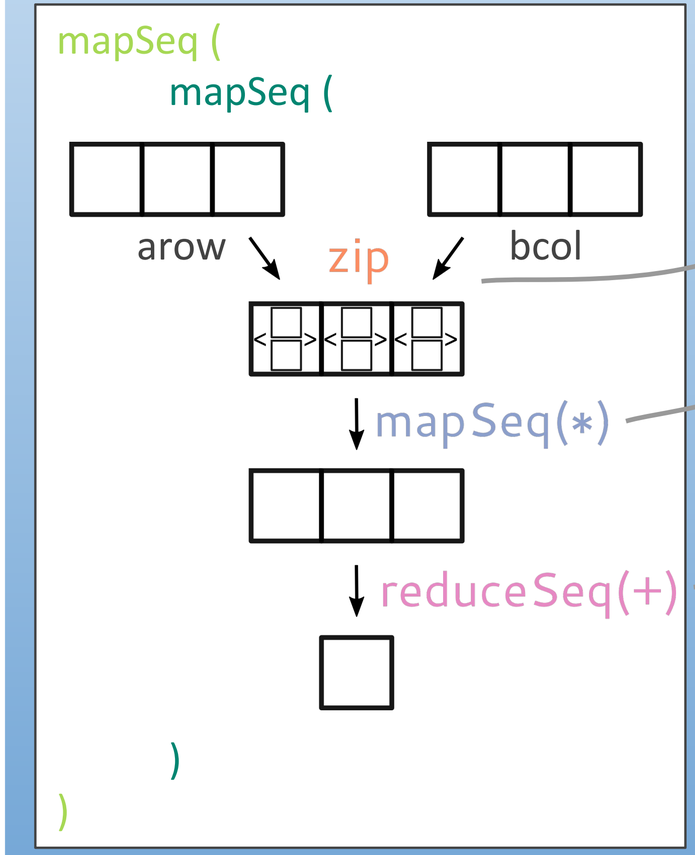
```
%zip = rise.zip #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>

// type: () → !rise.fun<array<1024, scalar<f32>> →
                        fun<array<1024, scalar<f32>> →
                            array<1024, tuple<scalar<f32>, scalar<f32>>>>
```

```
%zipped = rise.apply %zip, %arow, %bcol
```

Passing values to value parameters

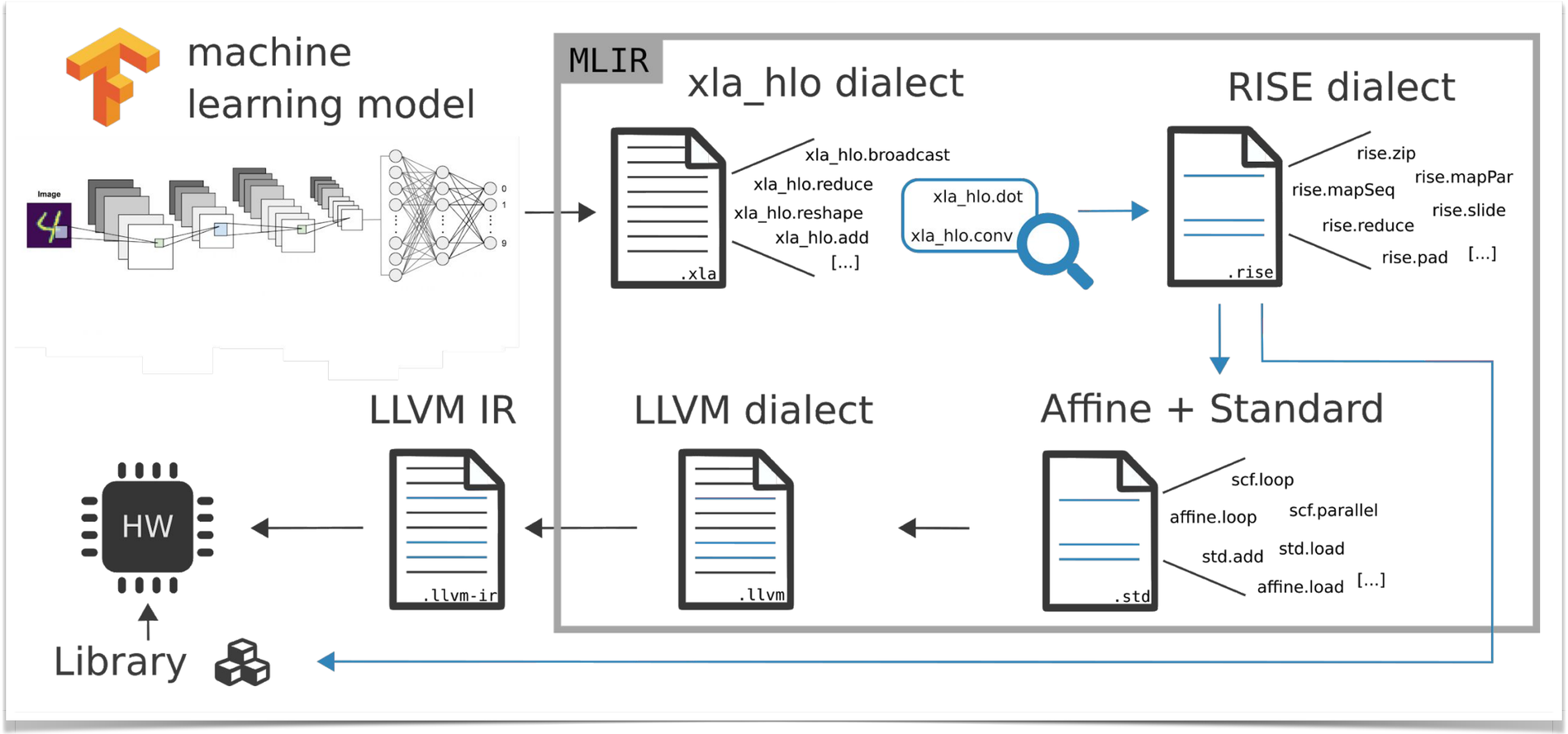
Matrix Multiplication in RISE in MLIR

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




```
1 func @mm(%out:memref<1024×1024xf32>, %inA:memref<1024×1024xf32>, %inB:memref<1024×1024xf32>) {
2   %A = rise.in %inA : !rise.array<1024, array<1024, scalar<f32>>>
3   %B = rise.in %inB : !rise.array<1024, array<1024, scalar<f32>>>
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
8       %f = rise.lambda (%tuple : !rise.tuple<scalar<f32>, scalar<f32>>) → !rise.scalar<f32> {
9         %fstFun = rise.fst #rise.scalar<f32> #rise.scalar<f32>
10        %sndFun = rise.snd #rise.scalar<f32> #rise.scalar<f32>
11        %fst = rise.apply %fstFun, %tuple
12        %snd = rise.apply %sndFun, %tuple
13        %result = rise.embed(%fst, %snd) {
14           %res = mulf %fst, %snd : f32
15           return %res : f32
16         } : !rise.scalar<f32>
17        rise.return %result : !rise.scalar<f32>
18      }
19      %map = rise.mapSeq #rise.nat<1024> #rise.tuple<scalar<f32>, scalar<f32>> #rise.scalar<f32>
20      %multipliedArray = rise.apply %map, %f, %zipped
21      %add = rise.lambda (%a : !rise.scalar<f32>, %b : !rise.scalar<f32>) → !rise.scalar<f32> {
22        %result = rise.embed(%a, %b) {
23          %res = addf %a, %b : f32
24          return %res : f32
25        } : !rise.scalar<f32>
26        rise.return %result : !rise.scalar<f32>
27      }
28      %init = rise.literal #rise.lit<0.0>
29      %reduce = rise.reduceSeq #rise.nat<1024> #rise.scalar<f32> #rise.scalar<f32>
30      %result = rise.apply %reduce, %add, %init, %multipliedArray
31      rise.return %result : !rise.scalar<f32>
32    }
33    %mapB = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.scalar<f32>
34    %result = rise.apply %mapB, %f2, %B
35    rise.return %result : !rise.array<1024, scalar<f32>>
36  }
37  %mapA = rise.mapSeq #rise.nat<1024> #rise.array<1024, scalar<f32>> #rise.array<1024, scalar<f32>>
38  %result = rise.apply %mapA, %f1, %A
39  rise.out %out ← %result
40  return
41 }
```

Compiling TensorFlow to LLVM IR via RISE



XLA HLO → RISE

```
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>
  }
}
```

xla_hlo.dot

xla_hlo.conv

```
%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) )) ))
```

RISE → Affine + Standard 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
}
```

???

```
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
}
```

Compiling RISE to low-level code

Strategy Preserving Compilation for Parallel Functional Code

ROBERT ATKEY, University of Strathclyde
MICHEL STEUWER, University of Edinburgh
SAM LINDLEY, University of Edinburgh
CHRISTOPHE DUBACH, University of Edinburgh

Graphics Processing Units (GPUs) and other parallel devices are widely available and have the potential for accelerating a wide class of algorithms. However, expert programming skills are required to achieving maximum performance. These devices expose low-level hardware details through imperative programming interfaces where programmers explicity encode device-specific optimisation strategies. This inevitably results in non-performance-portable programs delivering suboptimal performance on other devices.

Functional programming models have recently seen a renaissance in the systems community as they offer solutions for tackling the performance portability challenge. Recent work has shown how to automatically choose high-performance parallelisation strategies for a wide range of hardware architectures encoded in a functional representation. However, the translation of such functional representations to the imperative program expected by the hardware interface is typically performed ad hoc with no correctness guarantees and no guarantees to preserve the intended parallelisation strategy.

In this paper, we present a formalised *strategy-preserving* translation from high-level functional code to low-level data race free parallel imperative code. This translation is formulated and proved correct within a language we call Data Parallel Idealised Algol (DPIA), a dialect of Reynolds' Idealised Algol. Performance results on GPUs and a multicore CPU show that the formalised translation process generates low-level code with performance on a par with code generated from ad hoc approaches.

CCS Concepts: • **Software and its engineering** → *Parallel programming languages*; *Functional languages*; *Imperative languages*; *Multiparadigm languages*;

Functional

reduce (+) 0 (**map** (λx. **fst** x ∗ **snd** x) (**zip** xs ys))

↓

asScalar$_4$ (**join** (**mapWorkgroup** (λzs_1. **mapLocal** (λzs_2. **reduce** (λx a. (**fst** x ∗ **snd** x) + a) 0 (**split** 8192 zs_2)) zs_1) (**split** 8192 (**zip** (**asVector**$_4$ xs) (**asVector**$_4$ ys))))))

↓

Imperative

```
parforWorkgroup (N/8192) (joinAcc (N/8192) 64 (asScalarAcc4 (N/128) out)) (λ gid o.
  parforLocal 64 o (λ lid o.
    newPrivate num⟨4⟩ accum.
      accum.1 := 0;
      for 2048 (λ i.
        accum.1 := accum.2 +
          (fst (idx (idx (split 2048 (idx (split (8192 ∗ 4) (zip (asVector4 xs) (asVector4 ys))) gid)) lid) i)) ∗
          (snd (idx (idx (split 2048 (idx (split (8192 ∗ 4) (zip (asVector4 xs) (asVector4 ys))) gid)) lid) i)) );
      out := accum.2 ))
```

↓

```
kernel void KERNEL(global float *out, const global float *restrict xs,
                   const global float *restrict ys, int N) {
  for (int g_id = get_group_id(0); g_id < N / 8192; g_id += get_num_groups(0)) {
    for (int l_id = get_local_id(0); l_id < 64; l_id += get_local_size(0)) {
      float4 accum;
      accum = (float4)(0.0, 0.0, 0.0, 0.0);
      for (int i = 0; i < 2048; i += 1) {
        accum = (accum +
                 (vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), xs) *
                  vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), ys))); }
      vstore4(accum, ((64 * g_id) + l_id), out); } } }
```

RISE → Affine + Standard Dialect

STEP I

```
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
}
```

Lowering functional to imperative

```
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
}
```

RISE → Affine + Standard Dialect

STEP II

```
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
}
```

Resolve index computations

```
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
}
```

It actually works ...