Controllable Transformations in MLIR

Alex Zinenko
<zinenko@google.com>

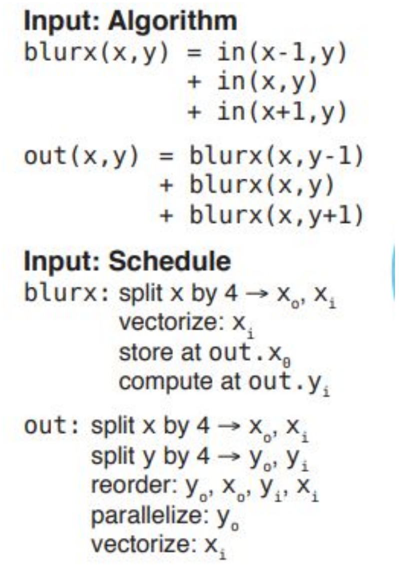
2023 EuroLLVM, Glasgow, May 10-11

Google Research

01

Why?

Scheduling DSLs in the Wild



Input: Algorithm

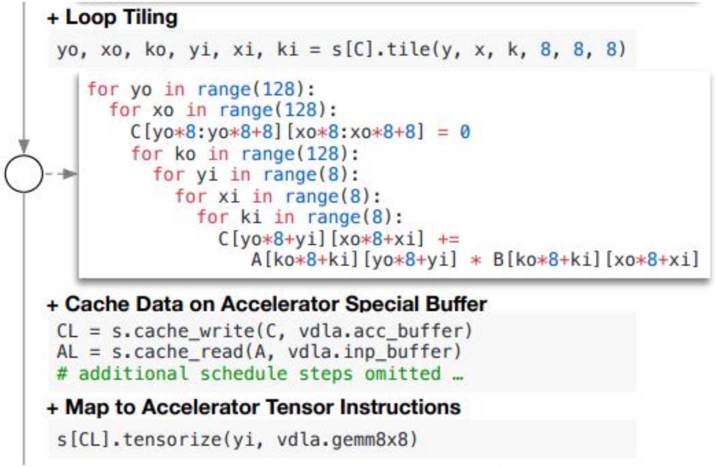
```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)

out(x,y) = blurx(x,y-1)
         + blurx(x,y)
         + blurx(x,y+1)
```

Input: Schedule

blurx: split x by 4 → x_o, x_i
vectorize: x_i
store at out.x_o
compute at out.y_i

out: split x by 4 → x_o, x_i
split y by 4 → y_o, y_i
reorder: y_o, x_o, y_i, x_i
parallelize: y_o
vectorize: x_i

Halide (Ragan-Kelley et.al. 2013)



+ Loop Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

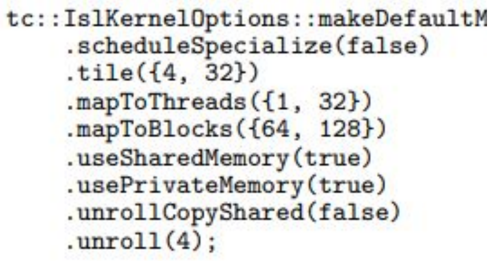
+ Cache Data on Accelerator Special Buffer

```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ Map to Accelerator Tensor Instructions

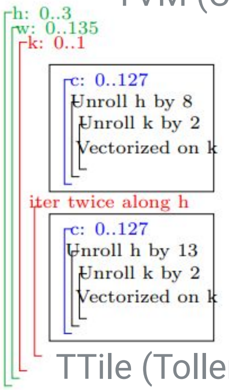
```
s[CL].tensorize(yi, vdla.gemm8x8)
```

TVM (Chen et.al. 2018)

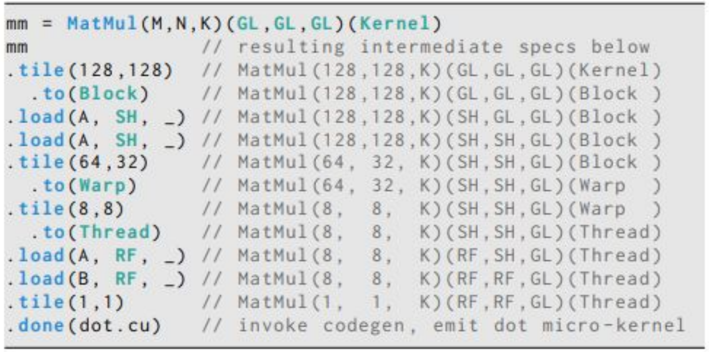


```
tc::IslKernelOptions::makeDefaultM
    .scheduleSpecialize(false)
    .tile({4, 32})
    .mapToThreads({1, 32})
    .mapToBlocks({64, 128})
    .useSharedMemory(true)
    .usePrivateMemory(true)
    .unrollCopyShared(false)
    .unroll(4);
```

TC (Vasilache et.al. 2018)



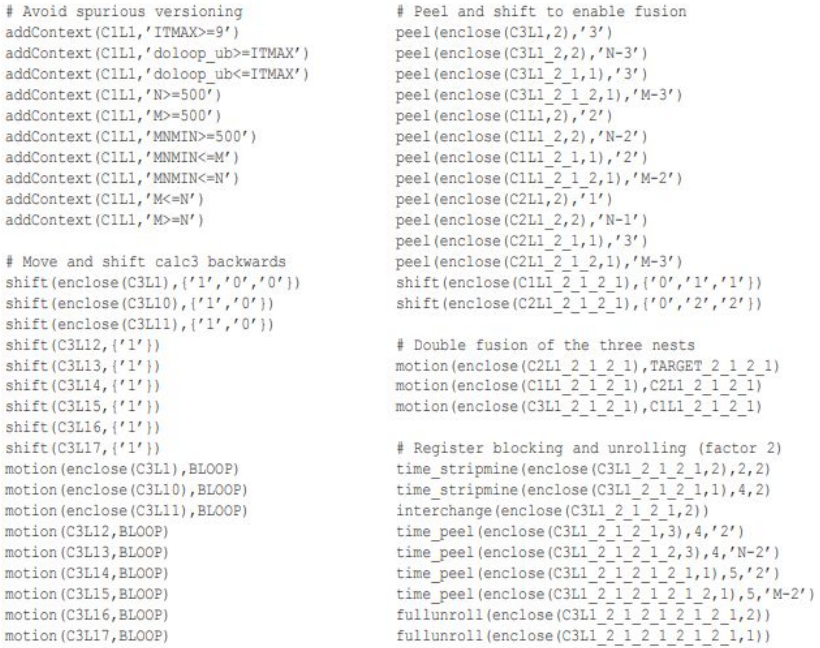
TTile (Tollenaere et.al. 2021)



```
mm = MatMul(M,N,K)(GL,GL,GL)(Kernel)
mm              // resulting intermediate specs below
.tile(128,128)  // MatMul(128,128,K)(GL,GL,GL)(Kernel)
  .to(Block)    // MatMul(128,128,K)(GL,GL,GL)(Block )
.load(A, SH, _) // MatMul(128,128,K)(SH,GL,GL)(Block )
.load(A, SH, _) // MatMul(128,128,K)(SH,SH,GL)(Block )
.tile(64,32)    // MatMul(64, 32, K)(SH,SH,GL)(Block )
  .to(Warp)     // MatMul(64, 32, K)(SH,SH,GL)(Warp  )
.tile(8,8)      // MatMul(8,  8,  K)(SH,SH,GL)(Warp  )
  .to(Thread)   // MatMul(8,  8,  K)(SH,SH,GL)(Thread)
.load(A, RF, _) // MatMul(8,  8,  K)(RF,SH,GL)(Thread)
.load(B, RF, _) // MatMul(8,  8,  K)(RF,RF,GL)(Thread)
.tile(1,1)      // MatMul(1,  1,  K)(RF,RF,GL)(Thread)
.done(dot.cu)   // invoke codegen, emit dot micro-kernel
```

Fireiron (Hagedorn et.al. 2020)

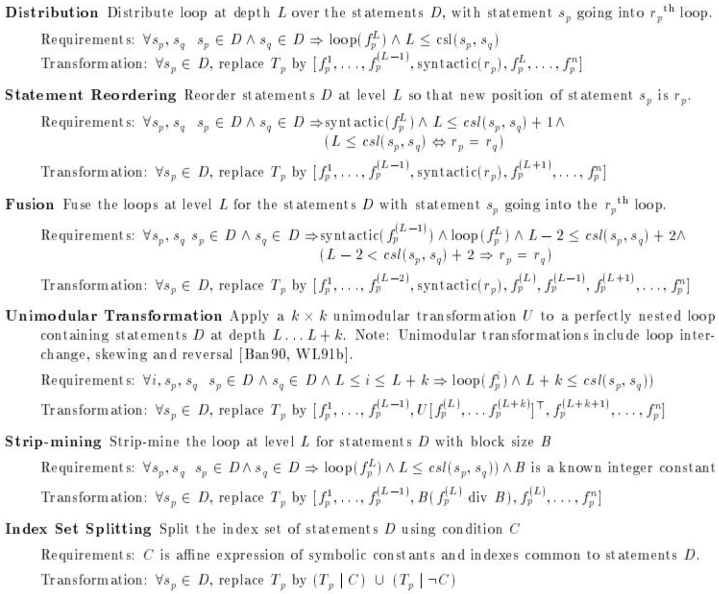
Scheduling DSLs in the Wild are Time-Tested



```
# Avoid spurious versioning
addContext(C1L1,'ITMAX>=9')
addContext(C1L1,'doloop_ub>=ITMAX')
addContext(C1L1,'doloop_ub<=ITMAX')
addContext(C1L1,'N>=500')
addContext(C1L1,'M>=500')
addContext(C1L1,'MNMIN>=500')
addContext(C1L1,'MNMIN<=M')
addContext(C1L1,'MNMIN<=N')
addContext(C1L1,'M<=N')
addContext(C1L1,'M>=N')

# Move and shift calc3 backwards
shift(enclose(C3L1),{'1','0','0'})
shift(enclose(C3L10),{'1','0'})
shift(enclose(C3L11),{'1','0'})
shift(C3L12,{'1'})
shift(C3L13,{'1'})
shift(C3L14,{'1'})
shift(C3L15,{'1'})
shift(C3L16,{'1'})
shift(C3L17,{'1'})
motion(enclose(C3L1),BLOOP)
motion(enclose(C3L10),BLOOP)
motion(enclose(C3L11),BLOOP)
motion(C3L12,BLOOP)
motion(C3L13,BLOOP)
motion(C3L14,BLOOP)
motion(C3L15,BLOOP)
motion(C3L16,BLOOP)
motion(C3L17,BLOOP)
```

```
# Peel and shift to enable fusion
peel(enclose(C3L1,2),'3')
peel(enclose(C3L1_2,2),'N-3')
peel(enclose(C3L1_2_1,1),'3')
peel(enclose(C3L1_2_1_2,1),'M-3')
peel(enclose(C1L1,2),'2')
peel(enclose(C1L1_2,2),'N-2')
peel(enclose(C1L1_2_1,1),'2')
peel(enclose(C1L1_2_1_2,1),'M-2')
peel(enclose(C2L1,2),'1')
peel(enclose(C2L1_2,2),'N-1')
peel(enclose(C2L1_2_1,1),'3')
peel(enclose(C2L1_2_1_2,1),'M-3')
shift(enclose(C1L1_2_1_2_1),{'0','1','1'})
shift(enclose(C2L1_2_1_2_1),{'0','2','2'})

# Double fusion of the three nests
motion(enclose(C2L1_2_1_2_1),TARGET_2_1_2_1)
motion(enclose(C1L1_2_1_2_1),C2L1_2_1_2_1)
motion(enclose(C3L1_2_1_2_1),C1L1_2_1_2_1)

# Register blocking and unrolling (factor 2)
time_stripmine(enclose(C3L1_2_1_2_1,2),2,2)
time_stripmine(enclose(C3L1_2_1_2_1,1),4,2)
interchange(enclose(C3L1_2_1_2_1,2))
time_peel(enclose(C3L1_2_1_2_1,3),4,'2')
time_peel(enclose(C3L1_2_1_2_1_2,3),4,'N-2')
time_peel(enclose(C3L1_2_1_2_1_2_1,1),5,'2')
time_peel(enclose(C3L1_2_1_2_1_2_1_2,1),5,'M-2')
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,2))
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,1))
```

URUK (Girbal et.al. 2006)



Distribution Distribute loop at depth L over the statements D, with statement s_p going into $r_p{}^{\text{th}}$ loop.

Requirements: $\forall s_p, s_q \;\; s_p \in D \wedge s_q \in D \Rightarrow \text{loop}(f_p^L) \wedge L \leq \text{csl}(s_p, s_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, \text{syntactic}(r_p), f_p^L, \ldots, f_p^n]$

Statement Reordering Reorder statements D at level L so that new position of statement s_p is r_p.

Requirements: $\forall s_p, s_q \;\; s_p \in D \wedge s_q \in D \Rightarrow \text{syntactic}(f_p^L) \wedge L \leq csl(s_p, s_q) + 1 \wedge$
$(L \leq csl(s_p, s_q) \Leftrightarrow r_p = r_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, \text{syntactic}(r_p), f_p^{(L+1)}, \ldots, f_p^n]$

Fusion Fuse the loops at level L for the statements D with statement s_p going into the $r_p{}^{\text{th}}$ loop.

Requirements: $\forall s_p, s_q \; s_p \in D \wedge s_q \in D \Rightarrow \text{syntactic}(f_p^{(L-1)}) \wedge \text{loop}(f_p^L) \wedge L - 2 \leq csl(s_p, s_q) + 2 \wedge$
$(L - 2 < csl(s_p, s_q) + 2 \Rightarrow r_p = r_q)$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-2)}, \text{syntactic}(r_p), f_p^{(L)}, f_p^{(L-1)}, f_p^{(L+1)}, \ldots, f_p^n]$

Unimodular Transformation Apply a $k \times k$ unimodular transformation U to a perfectly nested loop containing statements D at depth $L \ldots L + k$. Note: Unimodular transformations include loop interchange, skewing and reversal [Ban90, WL91b].

Requirements: $\forall i, s_p, s_q \;\; s_p \in D \wedge s_q \in D \wedge L \leq i \leq L + k \Rightarrow \text{loop}(f_p^i) \wedge L + k \leq csl(s_p, s_q))$

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, U[f_p^{(L)}, \ldots f_p^{(L+k)}]^\top, f_p^{(L+k+1)}, \ldots, f_p^n]$

Strip-mining Strip-mine the loop at level L for statements D with block size B

Requirements: $\forall s_p, s_q \;\; s_p \in D \wedge s_q \in D \Rightarrow \text{loop}(f_p^L) \wedge L \leq csl(s_p, s_q)) \wedge B$ is a known integer constant

Transformation: $\forall s_p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, B(f_p^{(L)} \text{ div } B), f_p^{(L)}, \ldots, f_p^n]$

Index Set Splitting Split the index set of statements D using condition C

Requirements: C is affine expression of symbolic constants and indexes common to statements D.

Transformation: $\forall s_p \in D$, replace T_p by $(T_p \mid C) \cup (T_p \mid \neg C)$

Omega (Pugh, 1991)

Motivation for Schedules in MLIR

- Many successful systems rely on some sort of *schedule representation* to produce state-of-the-art results.
- Schedules allow for *declarative* specification of transformations with arbitrary granularity.
- Schedules are *separable* and can be shipped independently.
- Schedules can support multi-versioning with runtime dispatch.
- Focus transformation on parts of IR ("vertical" sequencing rather than "horizontal" as with passes).

Schedules in MLIR

In MLIR, everything is an op.

So are schedules.

Such ops live in the Transform dialect.

02

Simple Transformation Chain

Google Research

~~Dis~~claimer

All operations and APIs exist upstream as of LLVM 8bf7f86d7966ff835 (Apr 17) unless explicitly specified otherwise.

Their syntax is revised on the slides for brevity.

Simple Transformation Chain

Source IR: fully connected layer + ReLU

Objective: fuse matmul and addition so it can be replaced by an efficient BLAS gemm call for 32x32 size, keep ReLU apart and vectorize it.

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence {









}
```

Perform transformations one after another.

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
















}
```

Perform transformations one after another.

Abort the transform and complain to the user if any transformation fails.

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Google Research

Simple Transformation Chain

Transform IR

```
transform.sequence failures(suppress) {



















}
```

Perform transformations one after another.

Abort the sequence but do not complain to the user. Next one can be attempted.

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(%root:
     %matmul:
     %elemwise:                                        ):




}
```

The sequence applies to some payload operations associated with transform IR values, or *handles*.

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(%root: !pdl.operation,
     %matmul: !transform.op<"linalg.matmul">,
     %elemwise: !transform.op<"linalg.elemwise_binary">):
```

The sequence applies to some payload operations associated with transform IR values, or *handles*.

Handles are typed. The type describes properties of the associated payload operations.

```
}
```

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Typing in Transforms Leads to Better Errors

Transform IR

```
transform.sequence failures(propagate) {
^bb0(%root: !pdl.operation,
     %matmul: !transform.op<"linalg.matmul">,
     %elemwise: !transform.op<"linalg.elemwise_unary">):







}
```

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

```
matmul.mlir:21:70: error: incompatible payload operation name
^bb0(%root: !pdl.operation, %matmul: !transform.op<"linalg.matmul">, %elemwise: !transform.op<"linalg.elemwise_unary">)
                                                                    ^

matmul.mlir:10:13: note: payload operation
 %biased = linalg.elemwise_binary { fun = #linalg.binary_fn }
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(%root: !pdl.operation,
     %matmul: !transform.op<"linalg.matmul">,
     %elemwise: !transform.op<"linalg.elemwise_binary">):

  %bias, %relu = transform.split_handles %elemwise in [2]
    : (!transform.op<"linalg.elemwise_binary">)
```

Handles are associated with *lists* of payload ops.

```
}
```

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(%root: !pdl.operation,
     %matmul: !transform.op<"linalg.matmul">,
     %elemwise: !transform.op<"linalg.elemwise_binary">):

  %bias, %relu = transform.split_handles %elemwise in [2]
    : (!transform.op<"linalg.elemwise_binary">)
   -> (!pdl.operation, !transform.op<"linalg.elemwise_binary">)
```

Handles are associated with *lists* of payload ops.

Handles can be casted to a different type, the verification happens dynamically.

```
}
```

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Google Research

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  %bias, %relu = transform.split_handles %elemwise in [2]
    ...

  %loop, %tiled = transform.structured.tile_to_forall_op %bias
    tile_sizes [32, 32]









}
```

Transformations apply to the payload ops associated with handles, tweaked by attributes.

Payload IR

```
%matmul = linalg.matmul ...
%biased = linalg.elemwise_binary {#add} (%matmul, ...)
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  %bias, %relu = transform.split_handles %elemwise in [2]
    ...

  %loop, %tiled = transform.structured.tile_to_forall_op %bias
    tile_sizes [32, 32]
```

Transformations apply to the payload ops associated with handles, tweaked by attributes.

```
}
```

Payload IR

```
%matmul = linalg.matmul ...
%biased = scf.forall (%i, %j) in (.../32, .../32) {
   %slice = tensor.extract_slice %matmul
   %part = linalg.elemwise_binary {#add} (%matmul, ...)
   “scf.forall.yield_slice” %slice
}
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  %bias, %relu = transform.split_handles %elemwise in [2]
    ...

  %loop, %tiled = transform.structured.tile_to_forall_op %bias
    tile_sizes [32, 32]
```

Transformations apply to the payload ops associated with handles, tweaked by attributes.

Transform ops define new handles for payload ops produced as the result.

```
}
```

Payload IR

```
%matmul = linalg.matmul ...
%biased = scf.forall (%i, %j) in (.../32, .../32) {
   %slice = tensor.extract_slice %matmul
   %part = linalg.elemwise_binary {#add} (%matmul, ...)
   “scf.forall.yield_slice” %slice
}
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Handle Consumption and Invalidation

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  %bias, %relu = transform.split_handles %elemwise in [2]
    ...

  %loop, %tiled = transform.structured.tile_to_forall_op %bias
    tile_sizes [32, 32]
```

Transform ops may *consume* handles that should no longer be used (associated payload was rewritten).

```
}
```

Payload IR

```
%matmul = linalg.matmul …
~~%biased = linalg.elemwise_binary {#add} (%matmul, ...)~~
%biased = scf.forall (%i, %j) in (.../32, .../32) {
   %slice = tensor.extract_slice %matmul
   %part = linalg.elemwise_binary {#add} (%matmul, ...)
   “scf.forall.yield_slice” %slice
}
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Handle Consumption and Invalidation

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  %bias, %relu = transform.split_handles %elemwise in [2]
    ...

  %loop, %tiled = transform.structured.tile_to_forall_op %bias
    tile_sizes [32, 32]

  transform.test_print_remark_at_operand %bias, "help!"
  : !pdl.operation
```

Payload IR

```
%matmul = linalg.matmul …
~~%biased = linalg.elemwise_binary {#add} (%matmul, ...)~~
%biased = scf.forall (%i, %j) in (.../32, .../32) {
   %slice = tensor.extract_slice %matmul
   %part = linalg.elemwise_binary {#add} (%matmul, ...)
   "scf.forall.yield_slice" %slice
}
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

```
matmul.mlir:27:3: error: op uses a handle invalidated by a previously executed transform op
 transform.test_print_remark_at_operand %bias, "help!" : !pdl.operation
 ^
matmul.mlir:26:19: note: invalidated by this transform op that consumes its operand #0 and invalidates all handles to payload IR
entities associated with this operand and entities nested in them
 %loop, %tiled = transform.structured.tile_to_forall_op %bias tile_sizes [32, 32]
                 ^
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
  %loop, %tiled = transform.structured.tile_to_forall_op %bias

  %cast_matmul = transform.cast %matmul
    : !transform.op<"linalg.matmul"> to !pdl.operation

  %fused_matmul = transform.structured.fuse_into_containing_op
    %cast_matmul into %loop
```

Transformations can be chained and *precisely targeted* by applying them to specific handles.

```
}
```

Payload IR

```
%matmul = linalg.matmul ...
%biased = scf.forall (%i, %j) in (.../32, .../32) {
   %slice = tensor.extract_slice %matmul
   %part = linalg.elemwise_binary {#add} (%matmul, ...)
   “scf.forall.yield_slice” %slice
}
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
  %loop, %tiled = transform.structured.tile_to_forall_op %bias

  %cast_matmul = transform.cast %matmul
    : !transform.op<"linalg.matmul"> to !pdl.operation

  %fused_matmul = transform.structured.fuse_into_containing_op
    %cast_matmul into %loop
```

Transformations can be chained and *precisely targeted* by applying them to specific handles.

This will *only* tile and fuse matmul with addition, and *not* relu, even though addition and relu are identical except for the attribute.

Payload IR

```
%biased = scf.forall (%i, %j) in (.../32, .../32) {
   tensor.extract_slice
   %slice = linalg.matmul ...
   %part = linalg.elemwise_binary {#add} (%matmul, ...)
   “scf.forall.yield_slice” %slice
}
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Precise Error Messages on Failure

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
  %loop, %tiled = transform.structured.tile_to_forall_op %bias

  %cast_matmul = transform.cast %matmul
    : !transform.op<"linalg.matmul"> to !pdl.operation

  %fused_matmul = transform.structured.fuse_into_containing_op
    %cast_matmul into %tiled
```

Payload IR

```
%biased = scf.forall (%i, %j) in (.../32, .../32) {
   tensor.extract_slice
   %slice = linalg.matmul ...
   %part = linalg.elemwise_binary {#add} (%matmul, ...)
   "scf.forall.yield_slice" %slice
}
%relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

```
matmul.mlir:28:19: error: could not find next producer to fuse into container
 %fused_matmul = transform.structured.fuse_into_containing_op %cast_matmul into %tiled
                 ^
matmul.mlir:10:13: note: container
 %biased = linalg.elemwise_binary { fun = #linalg.binary_fn }
```

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
  %loop, %tiled = transform.structured.tile_to_forall_op %bias
  ...




  transform.loop.outline %loop {func_name = "loop"}
    : (!pdl.operation) -> !pdl.operation








}
```

Payload IR

```
func.call @loop {
 %biased = scf.forall (%i, %j) in (.../32, .../32) {
    tensor.extract_slice
    %slice = linalg.matmul ...
    %part = linalg.elemwise_binary {#add} (%matmul, ...)
    “scf.forall.yield_slice” %slice
 }
 func.return %biased
}

 %biased = func.call @loop
 %relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

Google Research

Simple Transformation Chain

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
  %loop, %tiled = transform.structured.tile_to_forall_op %bias
  ...
  %parent = transform.get_closest_isolated_parent %loop
    : (!pdl.operation) -> !pdl.operation

  transform.loop.outline %loop {func_name = "loop"}
    : (!pdl.operation) -> !pdl.operation

  transform.structured.vectorize %parent




}
```

Payload IR

```
func.call @loop {
 %biased = scf.forall (%i, %j) in (.../32, .../32) {
    tensor.extract_slice
    %slice = linalg.matmul ...
    %part = linalg.elemwise_binary {#add} (%matmul, ...)
    “scf.forall.yield_slice” %slice
 }
 func.return %biased
}
```

```
%biased = func.call @loop
%relued = arith.maxf (%biased, 0.) : vector<...>
```

Handle Invalidation Continued

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
  %loop, %tiled = transform.structured.tile_to_forall_op %bias
  ...
  %parent = transform.get_closest_isolated_parent %loop
    : (!pdl.operation) -> !pdl.operation

  transform.loop.outline %loop {func_name = "loop"}
    : (!pdl.operation) -> !pdl.operation

  transform.structured.vectorize %parent



}
```

Payload IR

```
func.call @loop {
 %biased = scf.forall (%i, %j) in (.../32, .../32) {
    tensor.extract_slice
    %slice = linalg.matmul ...
    %part = linalg.elemwise_binary {#add} (%matmul, ...)
    “scf.forall.yield_slice” %slice
 }
 func.return %biased
}

 %biased = func.call @loop
 %relued = arith.maxf (%biased, 0.) : vector<...>
```

Handle Invalidation Continued

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
  %loop, %tiled = transform.structured.tile_to_forall_op %bias

  %fused_matmul = transform.structured.fuse_into_containing_op
    %cast_matmul into %loop

  transform.loop.outline %loop {func_name = "loop"}
    : (!pdl.operation) -> !pdl.operation

  transform.test_print_remark_at_operand %fused_matmul, "help!"
  : !pdl.operation
```

Consuming a handle invalidates *all other handles* associated with any of the payload ops nested in the payload ops associated with the consumed handle.

```
}
```

Payload IR

```
func.call @loop {
 %biased = scf.forall (%i, %j) in (.../32, .../32) {
    tensor.extract_slice
    %slice = linalg.matmul ...
    %part = linalg.elemwise_binary {#add} (%matmul, ...)
    "scf.forall.yield_slice" %slice
 }
 func.return %biased
}

 ~~%biased = scf.forall (%i, %j) in (.../32, .../32) {~~
    ...
    ~~%slice = linalg.matmul ...~~
    ...
 ~~}~~
 %biased = func.call @loop
 %relued = arith.maxf (%biased, 0.) : vector<...>
```

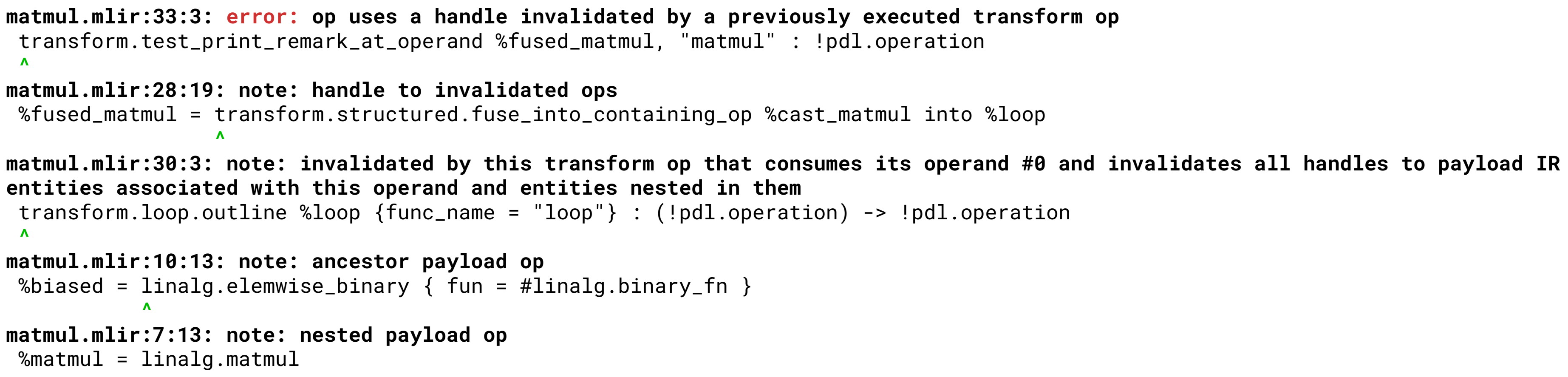
Google Research

Handle Invalidation Continued

Transform IR

```
transform.sequence failures(propagate) {
^bb0(...):
  ...
```

Payload IR



```
matmul.mlir:33:3: error: op uses a handle invalidated by a previously executed transform op
 transform.test_print_remark_at_operand %fused_matmul, "matmul" : !pdl.operation
 ^
matmul.mlir:28:19: note: handle to invalidated ops
 %fused_matmul = transform.structured.fuse_into_containing_op %cast_matmul into %loop
                 ^
matmul.mlir:30:3: note: invalidated by this transform op that consumes its operand #0 and invalidates all handles to payload IR entities associated with this operand and entities nested in them
 transform.loop.outline %loop {func_name = "loop"} : (!pdl.operation) -> !pdl.operation
 ^
matmul.mlir:10:13: note: ancestor payload op
 %biased = linalg.elemwise_binary { fun = #linalg.binary_fn }
           ^
matmul.mlir:7:13: note: nested payload op
 %matmul = linalg.matmul
```

```
}
```

~~%slice = linalg.matmul ...~~

```
    ...
~~}~~
%biased = func.call @loop
%relued = arith.maxf (%biased, 0.) : vector<...>
```

Simple Transformation Chain

Source IR: fully connected layer + ReLU

Objective: fuse matmul and addition so it can be replaced by an efficient BLAS gemm call for 32x32 size, keep ReLU apart and vectorize it.

```
 %matmul = linalg.matmul ...
 %biased = linalg.elemwise_binary {#add} (%matmul, ...)
 %relued = linalg.elemwise_binary {#maxf} (%biased, 0.)
```

```
func.call @loop {
 %biased = scf.forall (%i, %j) in (.../32, .../32) {
    tensor.extract_slice
    %slice = linalg.matmul ...
    %part = linalg.elemwise_binary {#add} (%matmul, ...)
    “scf.forall.yield_slice” %slice
 }
 func.return %biased
}

 %biased = func.call @loop
 %relued = arith.maxf (%biased, 0.) : vector<...>
```

Google Research

03

Let’s generalize!

Google Research

Transform Dialect

Transformations of the IR are described as a separate piece of IR where:

- Operations describe individual transformations to apply.
- Values (handles) are associated with operations that are being transformed.
- Transform operations may read or “consume” operands.
- Transform operations “produce” operands.
- Consuming a handle invalidates other handles to the same or nested IR.

Transform Dialect Interpreter

- Maintains the mapping between transform IR values and payload IR operations.
- Drives the application of transformations, including control flow.
- Maintains extra state if desired via the extension mechanism.
- Performs verification and tracks invalidation (expensive, similar to ASAN, disabled by default).
- Can be embedded into passes similarly to pattern application: `applyTransforms`.

Transform Dialect Interfaces

Transform operation interface:

- Specifies how a transform operation applies to payload IR (the interpreter dispatches to this), this may include dispatching to other operations from nested regions.
- Specifies the effects a transform has on handles and payload (reads, consumes, etc.)

Transform type interface:

- Specifies the conditions the payload must satisfy so it can be associated with the handle of this type (checked by the interpreter when a handle is produced).

Transform Dialect Entry Point

The application starts from a transformation op with a `PossibleTopLevelTransformOpTrait` that:

- Has no operands and no results (at least, the current instance of the op).
- Has a region with at least one argument of `TransformHandleTypeInterface` type.

The call to applyTransforms takes as arguments:

- The payload op to be associated with the first region argument.
- An optional list of lists of objects (ops, values, attributes) to be associated with the following region arguments.

Google Research

Trying it out

Interpreter can be *embedded* into passes similarly to pattern application: `applyTransform`.

The mlir-opt test pass “test-transform-dialect-interpreter” applies the first top-level transform op in the module to the module itself, and can bind extra arguments to payloads of the given kind.

```
mlir-opt matmul.mlir  --pass-pipeline="builtin.module(test-transform-dialect-interpreter{
        bind-first-extra-to-ops=linalg.matmul
        bind-second-extra-to-ops=linalg.elemwise_binary
        enable-expensive-checks})"
```

Trying it out

```
transform.sequence failures(propagate) {
^bb0(%root: !pdl.operation,
     %matmul: !transform.op<"linalg.matmul">,
     %elemwise: !transform.op<"linalg.elemwise_binary">):
```

```
mlir-opt matmul.mlir  --pass-pipeline="builtin.module(test-transform-dialect-interpreter{
        bind-first-extra-to-ops=linalg.matmul
        bind-second-extra-to-ops=linalg.elemwise_binary
        enable-expensive-checks})"
```

Semantic Trick for Early Exit

```
transform.sequence failures(propagate) {
^bb0(%root: !pdl.operation,
     %matmul: !transform.op<"linalg.matmul">,
     %elemwise: !transform.op<"linalg.elemwise_binary">):
```

How do we abort in the middle of a transformation sequence when an op is not a terminator?

- When a transformation fails, it sets the “has-failed” flag.
- Any transformation has the (implicit) semantics of doing nothing and associating result handles with empty lists of payload if the “has-failed” flag is set .
- Can be modeled as side effects to control reordering of transform ops.

04

Replicating Halide

Google Research

Halide Computation DSL

Conv2D Layer application (https://github.com/halide/Halide/tree/294f80c49bf3bb8582446613c25fcce03b82bcd8/apps/conv_layer)

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;

Func conv("conv");
```

Various constants

Halide Computation DSL

Conv2D Layer application (https://github.com/halide/Halide/tree/294f80c49bf3bb8582446613c25fcce03b82bcd8/apps/conv_layer)

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");
```

Various constants

Named dimensions (loops)

Halide Computation DSL

Conv2D Layer application (https://github.com/halide/Halide/tree/294f80c49bf3bb8582446613c25fcce03b82bcd8/apps/conv_layer)

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");

RDom r(0, CI, 0, 3, 0, 3);
    // x in 0:CI, y in 0:3, z in 0:3
```

Various constants

Named dimensions (loops)

Reduction dimensions r.{x,y,z} (loops)

Halide Computation DSL

Conv2D Layer application (https://github.com/halide/Halide/tree/294f80c49bf3bb8582446613c25fcce03b82bcd8/apps/conv_layer)

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");

RDom r(0, CI, 0, 3, 0, 3);

conv(c, x, y, n) = bias(c);
```

Various constants

Named dimensions (loops)

Reduction dimensions r.{x,y,z} (loops)

Pure statement

Halide Computation DSL

Conv2D Layer application (https://github.com/halide/Halide/tree/294f80c49bf3bb8582446613c25fcce03b82bcd8/apps/conv_layer)

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");

RDom r(0, CI, 0, 3, 0, 3);

conv(c, x, y, n) = bias(c);
conv(c, x, y, n) += filter(c, r.y, r.z, r.x)
                  * input(r.x, x + r.y, y + r.z, n);
```

Various constants

Named dimensions (loops)

Reduction dimensions r.{x,y,z} (loops)

Pure statement

Update Statement

Halide Computation DSL

Conv2D Layer application (https://github.com/halide/Halide/tree/294f80c49bf3bb8582446613c25fcce03b82bcd8/apps/conv_layer)

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");

RDom r(0, CI, 0, 3, 0, 3);

conv(c, x, y, n) = bias(c);
conv(c, x, y, n) += filter(c, r.y, r.z, r.x)
                  * input(r.x, x + r.y, y + r.z, n);
relu(c, x, y, n) = max(0, conv(c, x, y, n));
```

Various constants

Named dimensions (loops)

Reduction dimensions r.{x,y,z} (loops)

Pure statement

Update Statement

Halide Equivalent Loop Form

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
```

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");

RDom r(0, CI, 0, 3, 0, 3);

conv(c, x, y, n) = bias(c);
conv(c, x, y, n) += filter(c, r.y, r.z, r.x)
                  * input(r.x, x + r.y, y + r.z, n);
relu(c, x, y, n) = max(0, conv(c, x, y, n));
```

Google Research

Halide Equivalent Loop Form

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");

RDom r(0, CI, 0, 3, 0, 3);

conv(c, x, y, n) = bias(c);
conv(c, x, y, n) += filter(c, r.y, r.z, r.x)
                  * input(r.x, x + r.y, y + r.z, n);
relu(c, x, y, n) = max(0, conv(c, x, y, n));
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
```

Halide Equivalent Loop Form

```
const int N = 5, CI = 128, CO = 128, W = 100, H = 80;
Var x("x"), y("y"), c("c"), n("n");
Func conv("conv");

RDom r(0, CI, 0, 3, 0, 3);

conv(c, x, y, n) = bias(c);
conv(c, x, y, n) += filter(c, r.y, r.z, r.x)
                  * input(r.x, x + r.y, y + r.z, n);
relu(c, x, y, n) = max(0, conv(c, x, y, n));
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        relu[n, y, x, c] = max(0, conv[n, y, x, c])
```

Google Research

Halide Equivalent Loop Form

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        relu[n, y, x, c] = max(0, conv[n, y, x, c])
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
for n in ...:
  for y in ...:
    for x in ...:
      for co in 0:max(c)/vec*tile_w:
        for ci in 0:vec*tile_w:
          relu[n, y, x, co+ci] = max(0, conv[n, y, x, co+ci])
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
 .split(x, xo, xi, tile_h)
```

```
for n in ...:
 for y in ...:
  for x in ...:
   for c in ...:
    conv[n, y, x, c] = bias[c]
    for r.x in 0:CI:
     for r.y in 0:3:
      for r.z in 0:3:
       conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                * input[n, y + r.z, x + r.y, r.x]
for n in ...:
 for y in ...:
  for xo in ...:
   for xi in 0:tile_h:
    for co in 0:max(c)/vec*tile_w:
     for ci in 0:vec*tile_w:
      relu[n,y,xo+xi,co+ci] = max(0, conv[n,y,xo+xi,co+ci])
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
for co in 0:max(c)/vec*tile_w:
  for n in ...:
    for y in ...:
      for xo in ...:
        for xi in 0:tile_h:
          for ci in 0:vec*tile_w:
            relu[n,y,xo+xi,co+ci] = max(0, conv[n,y,xo+xi,co+ci])
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
for co in 0:max(c)/vec*tile_w:
  for n in ...:
    for y in ...:
      for xo in ...:
        for xi in 0:tile_h:
          for ci in 0:tile_w:
            relu[n,y,xo+xi,[co + ci*vec : co + (ci+1)*vec]] = ...
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
for co in 0:max(c)/vec*tile_w:
  for n in ...:
    for y in ...:
      for xo in ...:
        for xi in 0:tile_h:
          ~~for ci in 0:tile_w:~~
          relu[n,y,xo+xi,[co:co + vec]] = ...
          relu[n,y,xo+xi,[co + vec : co + 2*vec]] = ...
          ...
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
for co in 0:max(c)/vec*tile_w:
  for n in ...:
    for y in ...:
      for xo in ...:
        ~~for xi in 0:tile_h:~~
        relu[n,y,xo,[co:vec]] = ...
        relu[n,y,xo,[co + vec : co + 2*vec] = ...
        ...
        relu[n,y,xo+1,[co:vec]] = ...
```

Google Research

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                * input[n, y + r.z, x + r.y, r.x]
pfor co in 0:max(c)/vec*tile_w:
  pfor n in ...:
    pfor y in ...:
      for xo in ...:
        relu[n,y,xo,[co:vec]] = ...
        relu[n,y,xo,[co + vec : co + 2*vec] = ...
        ...
        relu[n,y,xo+1,[co:vec]] = …
        relu[n,y,xo+1,[co + vec : co + 2*vec]] = ...
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
```

```
for n in ...:
  for y in ...:
    for x in ...:
      for c in ...:
        conv[n, y, x, c] = bias[c]
        for r.x in 0:CI:
          for r.y in 0:3:
            for r.z in 0:3:
              conv[n, y, x, c] += filter[r.x, r.z, r.y, c]
                                  * input[n, y + r.z, x + r.y    .x]
pfor co in 0:max(c)/vec*tile_w:
  pfor n in ...:
    pfor y in ...:
      for xo in ...:
        relu[n,y,xo,[co:vec]] = ...
        relu[n,y,xo,[co + vec : co + 2*vec] = ...
        ...
        relu[n,y,xo+1,[co:vec]] = …
        relu[n,y,xo+1,[co + vec : co + 2*vec]] = ...
```

Google Research

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
```

```
pfor co in 0:max(c)/vec*tile_w:
  pfor n in ...:
    pfor y in ...:
      for xo in ...:
        for x in ...updated...:
          for c in ...updated...:
            conv[n, y, x, c] = bias[c]
            for r.x in 0:CI:
              for r.y in 0:3:
                for r.z in 0:3:
                  conv[n, y, x, c] += ...
        relu[n,y,xo,[co:vec]] = ...
        relu[n,y,xo,[co + vec : co + 2*vec] = ...
        ...
        relu[n,y,xo+1,[co:vec]] = …
        relu[n,y,xo+1,[co + vec : co + 2*vec]] = ...
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
    .split(x, xo, xi, tile_h)
    .reorder(ci, xi, xo, y, n, co)
    .vectorize(ci, vec)
    .unroll(ci)
    .unroll(xi)
    .parallel(y)
    .parallel(n)
    .parallel(co);
conv.compute_at(relu, xo)
    .vectorize(c, vec)
    .unroll(c)
```

```
pfor co in 0:max(c)/vec*tile_w:
 pfor n in ...:
  pfor y in ...:
   for xo in ...:
    for x in ...updated...:
     conv[n, y, x, 0:vec] = bias[0:vec]
     conv[n, y, x, vec:2*vec] = bias[vec:2*vec]
      ...
     for c in ...updated...:
      for r.x in 0:CI:
       for r.y in 0:3:
        for r.z in 0:3:
         conv[n, y, x, 0:vec] += ...
    relu[n,y,xo,[co:vec]] = ...
    relu[n,y,xo,[co + vec : co + 2*vec] = ...
    ...
    relu[n,y,xo+1,[co:vec]] = …
    relu[n,y,xo+1,[co + vec : co + 2*vec]] = ...
```

Google Research

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
  .split(x, xo, xi, tile_h)
  .reorder(ci, xi, xo, y, n, co)
  .vectorize(ci, vec)
  .unroll(ci)
  .unroll(xi)
  .parallel(y)
  .parallel(n)
  .parallel(co);
conv.compute_at(relu, xo)
  .vectorize(c, vec)
  .unroll(c)
  .unroll(x)
  .unroll(y)
```

```
pfor co in 0:max(c)/vec*tile_w:
 pfor n in ...:
  pfor y in ...:
   for xo in ...:
    conv[n, y, 0, 0:vec] = bias[0:vec]
    conv[n, y, 0, vec:2*vec] = bias[vec:2*vec]
    ...
    for x in ...updated...:
     for c in ...updated...:
      for r.x in 0:CI:
       for r.y in 0:3:
        for r.z in 0:3:
         conv[n, y, x, 0:vec] += ...
    relu[n,y,xo,[co:vec]] = ...
    relu[n,y,xo,[co + vec : co + 2*vec] = ...
    ...
    relu[n,y,xo+1,[co:vec]] = …
    relu[n,y,xo+1,[co + vec : co + 2*vec]] = ...
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   ~~.unroll(y)~~
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
```

```
pfor co in 0:max(c)/vec*tile_w:
  pfor n in ...:
    pfor y in ...:
      for xo in ...:
        conv[n, y, 0, 0:vec] = bias[0:vec]
        conv[n, y, 0, vec:2*vec] = bias[vec:2*vec]
        ...
        for r.z in 0:3:
          for r.y in 0:3:
            for r.x in 0:CI:
              for x in ...updated...:
                for c in ...updated...:
                  conv[n, y, x, 0:vec] += ...
        relu[n,y,xo,[co:vec]] = ...
        relu[n,y,xo,[co + vec : co + 2*vec] = ...
        ...
        relu[n,y,xo+1,[co:vec]] = …
        relu[n,y,xo+1,[co + vec : co + 2*vec]] = ...
```

Halide Schedule DSL

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .unroll(r.x, 2);
```

```
pfor co in 0:max(c)/vec*tile_w:
  pfor n in ...:
    pfor y in ...:
      for xo in ...:
        LOTS OF REPETITION
```

Two Possible Approaches

Introduce identical transformation ops on loops.

Map to existing transform ops on another MLIR abstraction.

Two Possible Approaches

Introduce identical transformation ops on loops.

Map to existing transform ops on another MLIR abstraction.

05

Primer on Structured Ops

Google Research

Primer on Structured Ops for Dense Algebra

```
linalg.generic {
  indexing_maps = [affine_map<(i,j)->(i,j)>, ..., affine_map<(i,j)->()>, ...],
  iterator_types = [“parallel”, “parallel”],
} ins(memref<4x8xf32>, memref<4x8xf32>, f32)
 outs(memref<4x8xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %3: f32, %out_init: f32):
  %3 = arith.addf %0, %1 : f32
  %4 = arith.maxf %2, %3 : f32
  linalg.yield %4 : f32
}
```

Primer on Structured Ops for Dense Algebra

```
linalg.generic {
  indexing_maps = [affine_map<(i,j)->(i,j)>, ..., affine_map<(i,j)->()>, ...],
  iterator_types = [“parallel”, “parallel”],
} ins(memref<4x8xf32>, memref<4x8xf32>, f32)
 outs(memref<4x8xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %3: f32, %out_init: f32):
  %3 = arith.addf %0, %1 : f32
  %4 = arith.maxf %2, %3 : f32
  linalg.yield %4 : f32
}
```

Structure of Iteration Space

```
for i in 0:4:
  for j in 0:8:
```

Google Research

Primer on Structured Ops for Dense Algebra

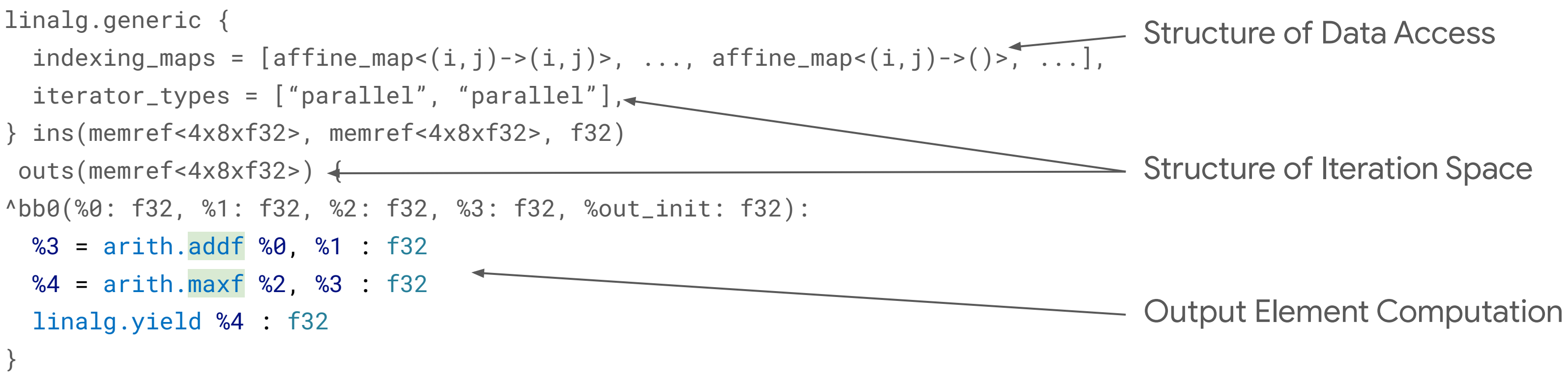
```
linalg.generic {
  indexing_maps = [affine_map<(i,j)->(i,j)>, ..., affine_map<(i,j)->()>, ...],
  iterator_types = [“parallel”, “parallel”],
} ins(memref<4x8xf32>, memref<4x8xf32>, f32)
 outs(memref<4x8xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %3: f32, %out_init: f32):
  %3 = arith.addf %0, %1 : f32
  %4 = arith.maxf %2, %3 : f32
  linalg.yield %4 : f32
}
```

Structure of Data Access

Structure of Iteration Space

```
for i in 0:4:
  for j in 0:8:
              in1[i,j]   in2[i,j]  in3
```

Primer on Structured Ops for Dense Algebra



```
linalg.generic {
  indexing_maps = [affine_map<(i,j)->(i,j)>, ..., affine_map<(i,j)->()>, ...],
  iterator_types = [“parallel”, “parallel”],
} ins(memref<4x8xf32>, memref<4x8xf32>, f32)
 outs(memref<4x8xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %3: f32, %out_init: f32):
  %3 = arith.addf %0, %1 : f32
  %4 = arith.maxf %2, %3 : f32
  linalg.yield %4 : f32
}
```

Structure of Data Access

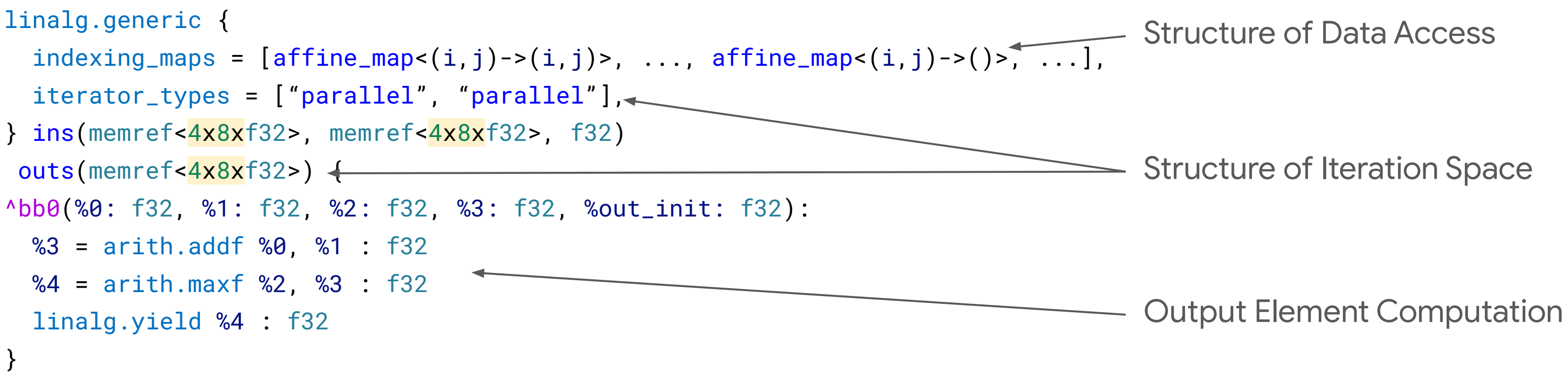
Structure of Iteration Space

Output Element Computation

```
for i in 0:4:
  for j in 0:8:
    yield max(in1[i,j] + in2[i,j], in3)
```

Google Research

Primer on Structured Ops for Dense Algebra



```
linalg.generic {
  indexing_maps = [affine_map<(i,j)->(i,j)>, ..., affine_map<(i,j)->()>, ...],
  iterator_types = [“parallel”, “parallel”],
} ins(memref<4x8xf32>, memref<4x8xf32>, f32)
 outs(memref<4x8xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %3: f32, %out_init: f32):
  %3 = arith.addf %0, %1 : f32
  %4 = arith.maxf %2, %3 : f32
  linalg.yield %4 : f32
}
```

Structure of Data Access

Structure of Iteration Space

Output Element Computation

These are not *actual* loops, but an implicit iteration space

```
for i in 0:4:
  for j in 0:8:
    yield max(in1[i,j] + in2[i,j], in3)
```

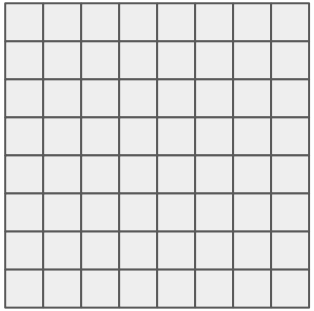
Structured Ops for Dense Linear Algebra

- Structured ops implicitly iterate over the input data (similar to Halide).
- The data access patterns are captured in closed forms (affine maps, similar to polyhedral).
- Support accumulation via a “reduction” dimension (also similar to Halide).
- Avoids the need for analysis, all information is readily available (tiling+vectorization is cheap).
- Keep the loop structure implicit (different from polyhedral).

Note: the concept of structured ops is *not* restricted to dense linear algebra. E.g., one can consider sparse access patterns and looplets.

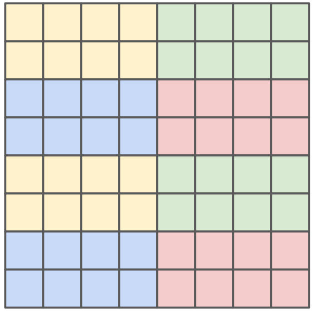
Loop Materialization by Tiling

```
linalg.generic {
  indexing_maps = ...,
  iterator_types = ...,
} ins(memref<8x8xf32>, memref<8x8xf32>, f32)
 outs(memref<8x8xf32>) {
  ...
}
```



Loop Materialization by Tiling

```
scf.forall (%i, %j) in (2, 4) {
  linalg.generic {
    indexing_maps = ...,
    iterator_types = ...,
  } ins(memref<4x2xf32>, memref<4x2xf32>, f32)
   outs(memref<4x2xf32>) {
    ...
  }
}
```



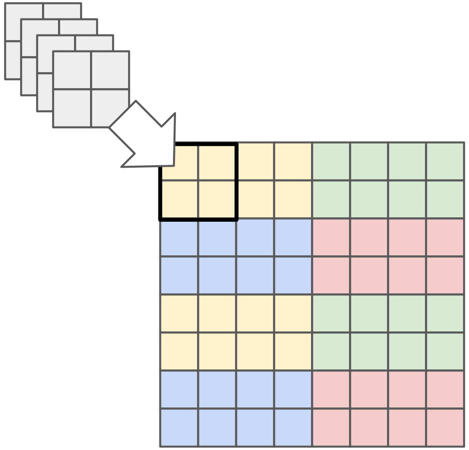
```
transform.structured.tile_to_forall_op
  tile_sizes [4, 2] %structured
```

Caveat: cannot reorder materialized and implicit loops.

But can tile repeatedly with size 1 to achieve the desired loop order.

Fusion into Loops

```
linalg.generic
scf.forall (%i, %j) in (2, 4) {
  linalg.generic {
    indexing_maps = ...,
    iterator_types = ...,
  } ins(memref<4x2xf32>, memref<4x2xf32>, f32)
   outs(memref<4x2xf32>) {
    ...
  }
}
```

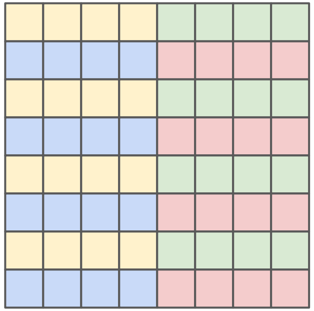


```
transform.structured.fuse_into_containing_op
  %structured into %loop
```

Similar to `compute_at` as long as the loop has been materialized.

Vectorization

```
scf.forall (%i, %j) in (2, 4) {
  arith.addf : vector<4xf32>
  arith.maxf : vector<4xf32>




}
```



```
transform.structured.vectorize %container
```

Caveat: loop vectorization is hard, so vectorize the structured op instead.

Caveat 2: must update types, so vectorize a whole container such as a function.

Loop Unrolling

```
scf.forall (%i, %j) in (2, 4) {
  arith.addf : vector<4xf32>
  arith.maxf : vector<4xf32>




}
```

```
transform.loop.unroll { factor = 4 } %loop
```

Loop unrolling is actually a loop transformation.
Transform dialect use case: unroll exactly the loops materialized by tiling.

06

Recreating the Schedule

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  %co_loop, %relu2 = transform.structured.tile_to_forall_op %relu
    tile_sizes [0, 0, 0, vec * tile_w]
// Loop order: [n, y, x, c]
// Generated:  xo, [n, y, xi, c] ; brackets denote implicit loops.
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  %co_loop, %relu2 = transform.structured.tile_to_forall_op %relu
    tile_sizes [0, 0, 0, vec * tile_w]
  %xo_loop, %relu4 = transform.structured.tile_to_forall_op %relu2 tile_sizes
    [0, tile_h]




// Loop order:            xo, [n, y, xi, c]
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  %co_loop, %relu2 = transform.structured.tile_to_forall_op %relu
    tile_sizes [0, 0, 0, vec * tile_w]
  %xo_loop, %relu4 = transform.structured.tile_to_forall_op %relu2 tile_sizes
    [0, tile_h]
```

```
// Loop order:               xo, [n, y, xi, c]
// Desired after reorder: co, n, y, xo, [(n), (y), xi, ci]
                         ; parentheses denote dimensions with size=1.
// If we were to tile the x dimension now,
// we would get:             co, xo, [n, y, xi, ci]
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  %co_loop, %relu2 = transform.structured.tile_to_forall_op %relu
    tile_sizes [0, 0, 0, vec * tile_w]
  %multi_loop, %relu3 = transform.structured.tile_to_forall_op %relu2
    tile_sizes [1, 1, 0, 0]
  %xo_loop, %relu4 = transform.structured.tile_to_forall_op %relu3 tile_sizes
    [0, 0, tile_h]

// Loop order:            xo, [n, y, xi, c]
// Desired after reorder: co, n, y, xo, [(n), (y), xi, ci]
                        ; parentheses denote dimensions with size=1.

// Instead, materialize the n and y loops by tiling with size 1 first
//              to obtain co, n, y, [(n), (y), x, ci]
// Then, tile x to obtain co, n, y, xo, [(n), (y), xi, ci] as desired
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  %co_loop, %relu2 = transform.structured.tile_to_forall_op %relu
    tile_sizes [0, 0, 0, vec * tile_w]
  %multi_loop, %relu3 = transform.structured.tile_to_forall_op %relu2
    tile_sizes [1, 1, 0, 0]
  %xo_loop, %relu4 = transform.structured.tile_to_forall_op %relu3 tile_sizes
    [0, 0, tile_h]

// Vectorization applies to the entire function, postpone.
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);

conv.compute_at(relu, xo)
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  %co_loop, %relu2 = transform.structured.tile_to_forall_op %relu
    tile_sizes [0, 0, 0, vec * tile_w]
  %multi_loop, %relu3 = transform.structured.tile_to_forall_op %relu2
    tile_sizes [1, 1, 0, 0]
  %xo_loop, %relu4 = transform.structured.tile_to_forall_op %relu3 tile_sizes
    [0, 0, tile_h]

  // Let's look at the convolution instead.
  // Must fuse into loops progressively, in order:
  %conv2 = transform.structured.fuse_into_containing_op %conv into %co_loop
  %conv3 = transform.structured.fuse_into_containing_op %conv2 into %multi_loop
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop
}
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);

conv.compute_at(relu, xo)
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  %co_loop, %relu2 = transform.structured.tile_to_forall_op %relu
    tile_sizes [0, 0, 0, vec * tile_w]
  %multi_loop, %relu3 = transform.structured.tile_to_forall_op %relu2
    tile_sizes [1, 1, 0, 0]
  %xo_loop, %relu4 = transform.structured.tile_to_forall_op %relu3 tile_sizes
    [0, 0, tile_h]



  %conv2 = transform.structured.fuse_into_containing_op %conv into %co_loop
  %conv3 = transform.structured.fuse_into_containing_op %conv2 into %multi_loop
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop

  // In MLIR, the bias is a separate operation, fuse it too.
  %bias2 = transform.structured.fuse_into_containing_op %bias into %co_loop
  %bias3 = transform.structured.fuse_into_containing_op %bias2 into %multi_loop
  %bias4 = transform.structured.fuse_into_containing_op %bias3 into %xo_loop
}
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop
  ...
  %bias4 = transform.structured.fuse_into_containing_op %bias3 into %xo_loop

 // Ensure we produce the desired order: n, r.z, r.y, r.x, y, x, c
 // Existing:                          [n, y, x, c, r.z, r.y, r.x]
 ... = transform.structured.tile_to_forall_op % tile_sizes [1]




}
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
    .split(x, xo, xi, tile_h)
    .reorder(ci, xi, xo, y, n, co)
    .vectorize(ci, vec)
    .unroll(ci)
    .unroll(xi)
    .parallel(y)
    .parallel(n)
    .parallel(co);
conv.compute_at(relu, xo)
    .vectorize(c, vec)
    .unroll(c)
    .unroll(x)
    .unroll(y)
    .update()
    .reorder(c, x, y, r.x, r.y, r.z, n)
    .vectorize(c, vec)
    .unroll(c)
    .unroll(x)
    .unroll(y)
    .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop
  ...
  %bias4 = transform.structured.fuse_into_containing_op %bias3 into %xo_loop

 // Ensure we produce the desired order: n, r.z, r.y, r.x, y, x, c
 // Existing:                           [n, y, x, c, r.z, r.y, r.x]
 ... = transform.structured.tile_to_forall_op % tile_sizes [1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 0, 0, 0, 1, 1, 1]




}
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);

conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop
  ...
  %bias4 = transform.structured.fuse_into_containing_op %bias3 into %xo_loop

 // Ensure we produce the desired order: n, r.z, r.y, r.x, y, x, c
 // Existing:                           [n, y, x, c, r.z, r.y, r.x]
 ... = transform.structured.tile_to_forall_op % tile_sizes [1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 0, 0, 0, 1, 1, 1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 1, 1]




}
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);

conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop
  ...
  %bias4 = transform.structured.fuse_into_containing_op %bias3 into %xo_loop

 // Ensure we produce the desired order: n, r.z, r.y, r.x, y, x, c
 // Existing:                            [n, y, x, c, r.z, r.y, r.x]
 ... = transform.structured.tile_to_forall_op % tile_sizes [1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 0, 0, 0, 1, 1, 1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 1, 1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 0, 0, 16]



}
```

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop
  ...
  %bias4 = transform.structured.fuse_into_containing_op %bias3 into %xo_loop

 // Ensure we produce the desired order: n, r.z, r.y, r.x, y, x, c
 // Existing:                          [n, y, x, c, r.z, r.y, r.x]
 ... = transform.structured.tile_to_forall_op % tile_sizes [1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 0, 0, 0, 1, 1, 1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 1, 1]
 ... = transform.structured.tile_to_forall_op % tile_sizes [0, 0, 0, 16]


}
```

Note that we keep the vector size in C.

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...
  %conv4 = transform.structured.fuse_into_containing_op %conv3 into %xo_loop
  ...
  %bias4 = transform.structured.fuse_into_containing_op %bias3 into %xo_loop

 %loops,  ... = transform.structured.tile_to_forall_op ...
 %loops2, ... = transform.structured.tile_to_forall_op ...
 %loops3, ... = transform.structured.tile_to_forall_op ...
 %loops4, ... = transform.structured.tile_to_forall_op ...
 %bias5 = transform.structured.fuse_into_containing_op %bias4 into %loops
 %bias6 = transform.structured.fuse_into_containing_op %bias5 into %loops2
 %bias7 = transform.structured.fuse_into_containing_op %bias6 into %loops3
 %bias8 = transform.structured.fuse_into_containing_op %bias7 into %loops4


}
```

Recreating the Schedule: Unrolling

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
    .split(x, xo, xi, tile_h)
    .reorder(ci, xi, xo, y, n, co)
    .vectorize(ci, vec)
    .unroll(ci)
    .unroll(xi)
    .parallel(y)
    .parallel(n)
    .parallel(co);

conv.compute_at(relu, xo)
    .vectorize(c, vec)
    .unroll(c)
    .unroll(x)
    .unroll(y)
    .update()
    .reorder(c, x, y, r.x, r.y, r.z, n)
    .vectorize(c, vec)
    .unroll(c)
    .unroll(x) // xi after fusion
    ~~.unroll(y)~~ // unit-extent after fusion
    .unroll(r.x, 2);
```

Recreating the Schedule: Unrolling

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x) // xi after fusion
   ~~.unroll(y)~~ // unit-extent after fusion
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x) // xi after fusion
   ~~.unroll(y)~~ // unit-extent after fusion
   .unroll(r.x, 2);
```

Same approach works for other operations.

Recreating the Schedule

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
    .split(x, xo, xi, tile_h)
    .reorder(ci, xi, xo, y, n, co)
    .vectorize(ci, vec)
    .unroll(ci)
    .unroll(xi)
    .parallel(y)
    .parallel(n)
    .parallel(co);

conv.compute_at(relu, xo)
    .vectorize(c, vec)
    .unroll(c)
    .unroll(x)
    .unroll(y)
    .update()
    .reorder(c, x, y, r.x, r.y, r.z, n)
    .vectorize(c, vec)
    .unroll(c)
    .unroll(x)
    .unroll(y)
    .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

  transform.structured.generalize %conv
  %f = transform.get_closest_isolated_parent  %conv
  %f2 = transform.structured.vectorize %f

}
```

Vectorization applies to isolated-from-above operations, such as functions.

Recreating the Schedule: Back to Unrolling

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x) // xi after fusion
   ~~.unroll(y)~~ // unit-extent after fusion
   .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

 // Materialized loops:       [n  y  x  c rz ry rx]
 %loops1, %conv2 = transform.structured.tile_to_forall_op %conv
                   tile_sizes [0, 0, 0, 0, 1, 1, 1]
 %loops2, %conv3 = transform.structured.tile_to_forall_op %conv2
                   tile_sizes [0, 1, 1]
 %loops3, %conv4 = transform.structured.tile_to_forall_op %conv3
                   tile_sizes [0, 0, 0, 16]


}
```

Recreating the Schedule: Back to Unrolling

```
Var co, ci, xo, xi, yo, yi, t;
relu.split(c, co, ci, vec * tile_w)
   .split(x, xo, xi, tile_h)
   .reorder(ci, xi, xo, y, n, co)
   .vectorize(ci, vec)
   .unroll(ci)
   .unroll(xi)
   .parallel(y)
   .parallel(n)
   .parallel(co);
conv.compute_at(relu, xo)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x)
   .unroll(y)
   .update()
   .reorder(c, x, y, r.x, r.y, r.z, n)
   .vectorize(c, vec)
   .unroll(c)
   .unroll(x) // xi after fusion
   ~~.unroll(y)~~ // unit-extent after fusion
   .unroll(r.x, 2);
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

 // Materialized loops:       [n  y  x  c rz ry rx]
 %loops1, %conv2 = transform.structured.tile_to_forall_op %conv
                   tile_sizes [0, 0, 0, 0, 1, 1, 1]
 %loops2, %conv3 = transform.structured.tile_to_forall_op %conv2
                   tile_sizes [0, 1, 1]
 %loops3, %conv4 = transform.structured.tile_to_forall_op %conv3
                   tile_sizes [0, 0, 0, 16]

 transform.loop.unroll %loops1 {factor = ???} : !pdl.operation
}
```

We have handles to loops we produced in tiling!

Google Research

Oops, we have a type mismatch

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

 // Materialized loops:       [n  y  x  c rz ry rx]
 %loops1, %conv2 = transform.structured.tile_to_forall_op %conv
                   tile_sizes [0, 0, 0, 0, 1, 1, 1]
 %loops2, %conv3 = transform.structured.tile_to_forall_op %conv2
                   tile_sizes [0, 1, 1]
 %loops3, %conv4 = transform.structured.tile_to_forall_op %conv3
                   tile_sizes [0, 0, 0, 16]

 transform.loop.unroll %loops1 {factor = ???} : !pdl.operation
}
```

```
scf.forall (%rz, %ry, %rx) in ...
```

≠

```
scf.for %rx = ...
```

07

Extending the Transform Dialect

Defining a Transform Op

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

Defining a Transform Op

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

.td

```
def ForallToFor
  : Op<                 , "tutorial.forall_to_for",
```

Defining a Transform Op

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
```

Things to know:

- Transform ops can be injected into the dialect.

Defining a Transform Op

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [
       DeclareOpInterfaceMethods<TransformOpInterface>]> {





}
```

Things to know:

- Transform ops can be injected into the dialect.
- Must implement the transform interface.

Defining a Transform Op

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {





}
```

Things to know:

- Transform ops can be injected into the dialect.
- Must implement the transform interface.
- Must describe side effects.

Google Research

Defining a Transform Op

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       TransformHandleTypeInterface:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

Things to know:

- Transform ops can be injected into the dialect.
- Must implement the transform interface.
- Must describe side effects.

Base type interface for handles.

Google Research

Defining a Transform Op

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       TransformHandleTypeInterface:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

Things to know:

- Transform ops can be injected into the dialect.
- Must implement the transform interface.
- Must describe side effects.

Google Research

Implementing a Transform Op: Transform Iface

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {









  return DiagnosedSilenceableFailure::success();
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       TransformHandleTypeInterface:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

Would like a transform op that:

- Takes a handle to `scf.forall`.
- Triggers rewriting into a nest of `scf.for`.
- Returns handles to produces ops.

Google Research

Failure Modes

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {









  return DiagnosedSilenceableFailure::success();
}
```

Tri-state result object:

- Success: ~`LogicalResult::success`.
- Definite failure: the diagnostic has been reported to the engine, just propagating `LogicalResult::failure`.
- Silenceable failure: *contains* the *not yet* reported diagnostic. Can be reported to the engine, or silenced and discarded.

Arguments

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {










  return DiagnosedSilenceableFailure::success();
}
```

Transform results:

- Populate this with payload IR objects to be associated with the result handles on success.

Transform state:

- Query this for the payload IR objects associated with operands and other values.
- Access to various extension points.

Implementing a Transform Op: Transform Iface

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {
  ArrayRef<Operation *> payload =
    state.getPayloadOps(getTarget());










  return DiagnosedSilenceableFailure::success();
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       TransformHandleTypeInterface:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. Get the payload ops associated with the operand.

Implementing a Transform Op: Transform Iface

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {
  ArrayRef<Operation *> payload =
    state.getPayloadOps(getTarget());
  if (payload.size() != 1) {
    return emitSilenceableError()
        << "expected a single payload op";
  }



  return DiagnosedSilenceableFailure::success();
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       TransformHandleTypeInterface:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. Get the payload ops associated with the operand.
2. Check well-formedness and report errors.

Google Research

Implementing a Transform Op: Transform Iface

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {
  ArrayRef<Operation *> payload =
    state.getPayloadOps(getTarget());
  if (payload.size() != 1) {
    return emitSilenceableError()
        << "expected a single payload op";
  }
  auto target = dyn_cast<scf::ForallOp>(payload[0]);
  if (!target) {
    return emitSilenceableError()
        << "expected the payload to be scf.forall";
  }




  return DiagnosedSilenceableFailure::success();
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       TransformHandleTypeInterface:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. Get the payload ops associated with the operand.
2. Check well-formedness and report errors.

Implementing a Transform Op: Transform Iface

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {
  ArrayRef<Operation *> payload =
    state.getPayloadOps(getTarget());
  if (payload.size() != 1) {
    return emitSilenceableError()
        << "expected a single payload op";
  }
  ~~auto target = dyn_cast<scf::ForallOp>(payload[0]);~~
  ~~if (!target) {~~
    ~~return emitSilenceableError()~~
        ~~<< "expected the payload to be scf.forall";~~
  ~~}~~



  return DiagnosedSilenceableFailure::success();
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

Specific implementations of the Transform type interface can supply a runtime checks that are performed when payload is associated with the handle, and produce silenceable errors on mismatch

Google Research

Implementing a Transform Op: Transform Iface

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {
  ArrayRef<Operation *> payload =
    state.getPayloadOps(getTarget());
  if (payload.size() != 1) {
    return emitSilenceableError()
        << "expected a single payload op";
  }

  SmallVector<scf::ForOp> loops =
    doActualRewrite(cast<scf::ForallOp>(payload[0]));





  return DiagnosedSilenceableFailure::success();
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. Get the payload ops associated with the operand.
2. Check well-formedness and report errors.
3. Do the actual rewrite.

Google Research

Implementing a Transform Op: Transform Iface

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::apply(
    transform::TransformResults &results,
    transform::TransformState &state) {
  ArrayRef<Operation *> payload =
    state.getPayloadOps(getTarget());
  if (payload.size() != 1) {
    return emitSilenceableError()
        << "expected a single payload op";
  }

  SmallVector<scf::ForOp> loops =
    doActualRewrite(cast<scf::ForallOp>(payload[0]));

  for (auto &&[res, loop]
       : llvm::zip(getTransformed(), loops)) {
    results.set(cast<OpResult>(res), loop);
  }

  return DiagnosedSilenceableFailure::success();
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. Get the payload ops associated with the operand.
2. Check well-formedness and report errors.
3. Do the actual rewrite.
4. Associate result handles with results.

Google Research

Implementing a Transform Op: MemEffect Iface

.cc

```
void transform::TakeAssumedBranchOp::getEffects(
    SmallVectorImpl<MemoryEffects::EffectInstance> &
    effects) {




}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

Implementing a Transform Op: MemEffect Iface

.cc

```
void transform::TakeAssumedBranchOp::getEffects(
    SmallVectorImpl<MemoryEffects::EffectInstance> &
    effects) {
  consumesHandle(getTarget(), effects);


}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. The target handle is consumed because the rewrite replaces the original payload op.

Google Research

Implementing a Transform Op: MemEffect Iface

.cc

```
void transform::TakeAssumedBranchOp::getEffects(
    SmallVectorImpl<MemoryEffects::EffectInstance> &
    effects) {
  consumesHandle(getTarget(), effects);
  producesHandle(getTransformed(), effects);

}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. The target handle is consumed because the rewrite replaces the original payload op.
2. The result handles are produced.

Implementing a Transform Op: MemEffect Iface

.cc

```cpp
void transform::TakeAssumedBranchOp::getEffects(
    SmallVectorImpl<MemoryEffects::EffectInstance> &
    effects) {
  consumesHandle(getTarget(), effects);
  producesHandle(getTransformed(), effects);
  modifiesPayload(effects);
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [DeclareOpInterfaceMethods<MemoryEffectsOpInterface>,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

1. The target handle is consumed because the rewrite replaces the original payload op.
2. The result handles are produced.
3. Also indicate that payload is modified to prevent reordering.

Google Research

Implementing Interfaces with Traits

.cc

```
~~void transform::TakeAssumedBranchOp::getEffects(~~
    ~~SmallVectorImpl<MemoryEffects::EffectInstance> &~~
    ~~effects) {~~
  ~~consumesHandle(getTarget(), effects);~~
  ~~producesHandle(getTransformed(), effects);~~
  ~~modifiesPayload(effects);~~
}
```

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [MemoryEffectsOpInterface,
       FunctionalStyleTransformOpTrait,
       DeclareOpInterfaceMethods<TransformOpInterface>]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

Common patterns are available as *traits* that implement the interface, or make it simpler:

- Functional-style transform op trait: consumes operands, produces results, modifies IR.
- Navigation-only transform op trait: only reads operands, produces results, only reads IR.

Implementing Interfaces with Traits

.cc

```
DiagnosedSilenceableFailure
transform::ForallToFor::applyToOne(
    scf::ForAllOp target,
    transform::ApplyToEachResultList &results,
    transform::TransformState &state) {

  // Target readily available.
  SmallVector<scf::ForOp> loops =
    doActualRewrite(target);

  // Problem: this transform is not one-to-one.
  results.push_back(loops /*expects Operation*/);

  return DiagnosedSilenceableFailure::success();
}
```

Trait not applicable to this op, but useful in general.

.td

```
def ForallToFor
  : Op<Transform_Dialect, "tutorial.forall_to_for",
      [MemoryEffectsOpInterface,
       FunctionalStyleTransformOpTrait,
       TransformOpInterface,
       TransformEachOpTrait]> {
  let arguments = (ins
       Transform_ConcreteOpType<"scf.forall">:$target);
  let results = (outs
       Variadic<TransformHandleTypeInterface>:$transformed);
  // ...
}
```

Common patterns are available as *traits* that implement the interface, or make it simpler:

- Apply-Each transform op trait: iterate over payloads associated with the single operand, apply a one-to-one rewrite, and populate results.

08

Connecting out-of-tree

Putting it All Together

We can't just define an op in somebody else's dialect...

Putting it All Together

We can't just define an op in somebody else's dialect… Actually, we can via dialect extensions.

```
class TutorialTransformDialectExtension
  : public transform::TransformDialectExtension<TutorialTransformDialectExtension> {
public:
  using Base::Base;
```

Putting it All Together

We can’t just define an op in somebody else’s dialect… Actually, we can via dialect extensions.

```
class TutorialTransformDialectExtension
  : public transform::TransformDialectExtension<TutorialTransformDialectExtension> {
public:
  using Base::Base;

  void init() {
    // Indicate that transforms generate ops from SCF dialect, so it is loaded by the interpreter pass.
    declareGeneratedDialect<scf::SCFDialect>();
```

Putting it All Together

We can’t just define an op in somebody else’s dialect… Actually, we can via dialect extensions.

```
class TutorialTransformDialectExtension
  : public transform::TransformDialectExtension<TutorialTransformDialectExtension> {
public:
  using Base::Base;

  void init() {
    // Indicate that transforms generate ops from SCF dialect, so it is loaded by the interpreter pass.
    declareGeneratedDialect<scf::SCFDialect>();

    // Inject additional ops into the dialect, complain if they don’t implement required interfaces.
    registerTransformOps<
    // This is the usual op listing logic from MLIR ODS.
#define GET_OP_LIST
#include "tutorial/Dialect/Tutorial/TransformOps/TutorialTransformOps.cpp.inc"
    >();
  }
};
```

Putting it All Together

We can’t just define an op in somebody else’s dialect… Actually, we can via dialect extensions.

This is not *strictly* necessary for out-of-tree projects, but comes with extra verification and registration logic (otherwise, the pass applying transforms must declare generated dialects).

In-tree, dialect extensions allow us to avoid having 2x dialects.

Creating a Custom Interpreter Pass

Transform Dialect interpreter is a utility, similar to dialect conversion or greedy pattern rewriter.

It can be used within any pass.

It may be useful to provide a standalone pass out-of-tree for, e.g., debugging.

```
class TestTransformDialectInterpreterPass
   : public transform::TransformInterpreterPassBase<TestTransformDialectInterpreterPass, ...> {

  // This provides options to control:
  //   - additional checking (“expensive checks mode”),
  //   - indicate the starting points in payload and transform IR,
  //   - dump reproducers,
  //   - load transforms from files,
  //   etc.

  void runOnOperation() override {
    options = options.enableExpensiveChecks(enableExpensiveChecks);
    if (failed(transform::detail::interpreterBaseRunOnOperationImpl(...))
      return signalPassFailure();
}
```

09

Back to Halide

Google Research

Oops, we have a type mismatch

```
scf.forall (%rz, %ry, %rx) in ...

    ≠

scf.for %rx = ...
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

 // Materialized loops:       [n  y  x  c rz ry rx]
 %loops1, %conv2 = transform.structured.tile_to_forall_op %conv
                   tile_sizes [0, 0, 0, 0, 1, 1, 1]
 %loops2, %conv3 = transform.structured.tile_to_forall_op %conv2
                   tile_sizes [0, 1, 1]
 %loops3, %conv4 = transform.structured.tile_to_forall_op %conv3
                   tile_sizes [0, 0, 0, 16]

 transform.loop.unroll %loops1 {factor = ???} : !pdl.operation
}
```

Oops, we have a type mismatch

```
scf.forall (%rz, %ry, %rx) in ...

        ⇩

scf.for %rz = …
  scf.for %ry = …
    scf.for %rx = ...
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

 // Materialized loops:       [n  y  x  c rz ry rx]
 %loops1, %conv2 = transform.structured.tile_to_forall_op %conv
                   tile_sizes [0, 0, 0, 0, 1, 1, 1]
 %loops2, %conv3 = transform.structured.tile_to_forall_op %conv2
                   tile_sizes [0, 1, 1]
 %loops3, %conv4 = transform.structured.tile_to_forall_op %conv3
                   tile_sizes [0, 0, 0, 16]



 %rz, %ry, %rx = transform.tutorial.forall_to_for %loops1
               : (!pdl.operation) -> !pdl.operation
 transform.loop.unroll %rx {factor = 2} : !pdl.operation
 // …
}
```

Oops, we *still* have a type mismatch

```
scf.forall (%rz, %ry, %rx) in ...

         ⇩

scf.for %rz = …
  scf.for %ry = …
    scf.for %rx = ...
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

 // Materialized loops:       [n  y  x  c rz ry rx]
 %loops1, %conv2 = transform.structured.tile_to_forall_op %conv
                   tile_sizes [0, 0, 0, 0, 1, 1, 1]
 %loops2, %conv3 = transform.structured.tile_to_forall_op %conv2
                   tile_sizes [0, 1, 1]
 %loops3, %conv4 = transform.structured.tile_to_forall_op %conv3
                   tile_sizes [0, 0, 0, 16]



 %rz, %ry, %rx = transform.tutorial.forall_to_for %loops1
               : (!pdl.operation) -> !pdl.operation
 transform.loop.unroll %rx {factor = 2} : !pdl.operation
 // …
}
```

```
  let arguments = (ins
Transform_ConcreteOpType<"scf.forall">:$target);
```

Unrolling with Proper Types

```
scf.forall (%rz, %ry, %rx) in ...
```

⇩

```
scf.for %rz = …
  scf.for %ry = …
    scf.for %rx = ...
```

```
transform.sequence failures(propagate) {
^bb0(%conv, %bias, %relu):
  ...

 // Materialized loops:       [n  y  x  c rz ry rx]
 %loops1, %conv2 = transform.structured.tile_to_forall_op %conv
                   tile_sizes [0, 0, 0, 0, 1, 1, 1]
 %loops2, %conv3 = transform.structured.tile_to_forall_op %conv2
                   tile_sizes [0, 1, 1]
 %loops3, %conv4 = transform.structured.tile_to_forall_op %conv3
                   tile_sizes [0, 0, 0, 16]

 %loops_casted = transform.cast %loops1
               : !pdl.operation to !transform.op<"scf.forall">
 %rz, %ry, %rx = transform.tutorial.forall_to_for %loops_casted
               : (!transform.op<"scf.forall">) -> !pdl.operation
 transform.loop.unroll {factor = 2} : !pdl.operation
 // …
}
```

Putting it All Together

Mapped the following transformations:

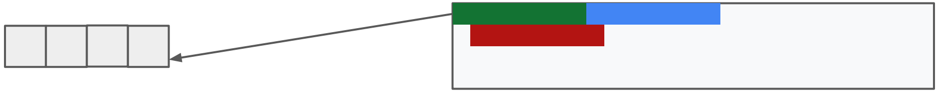
- Split -> `tile_to_forall_op` with one non-zero value.
- Reorder -> sequence of `tile_to_forall_op` to materialize loops.
- Vectorize -> `tile_to_forall_op` of the corresponding dimension to vector size, and the remaining dimensions to 1 to materialize loops, then global vectorize.
- Unroll -> `loop.unroll`.
- ComputeAt -> `fuse_into_containing_op`.

Implemented a custom “forall” to “for” transformation to target sequential loops.

What Was Swept Under the Rug

- All implementation details of actual transformations: transform dialect only *orchestrates* them.

- Bufferization:
  - Structured ops are defined on tensors, Halide works on buffers.
  - The bufferization process is global and is *currently* not controllable (though the placement of loads/stores can be, to some extent, through non-foldable padding).
  - Bufferization invalidates *all* handles, we currently re-match payload IR after it.



10

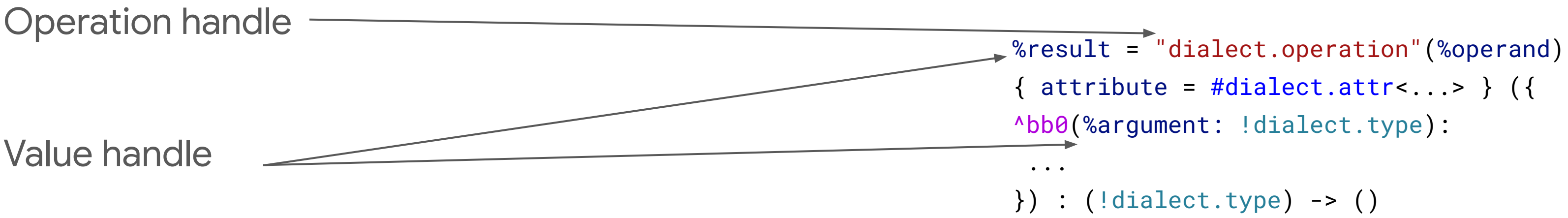
Advanced Topics

Not Only Operation Handles

Operation handle

```
%result = "dialect.operation"(%operand)
{ attribute = #dialect.attr<...> } ({
^bb0(%argument: !dialect.type):
 ...
}) : (!dialect.type) -> ()
```

Not Only Operation Handles

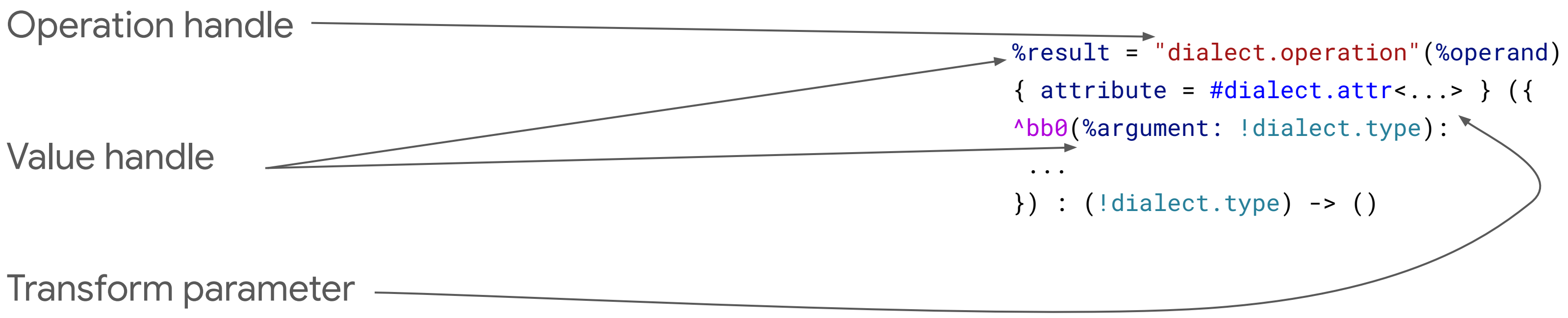


Operation handle

Value handle

```
%result = "dialect.operation"(%operand)
{ attribute = #dialect.attr<...> } ({
^bb0(%argument: !dialect.type):
 ...
}) : (!dialect.type) -> ()
```

Not Only Operation Handles



Operation handle

Value handle

Transform parameter

```
%result = "dialect.operation"(%operand)
{ attribute = #dialect.attr<...> } ({
^bb0(%argument: !dialect.type):
 ...
}) : (!dialect.type) -> ()
```

Not Only Operation Handles

Operation handle	`TransformHandleTypeInterface`
	`getPayloadOps / set`

Value handle	`TransformValueHandleTypeInterface`
	`getPayloadValues / setValues`

Transform parameter	`TransformParamTypeInterface`
	`getParams / setParams`

```
%result = "dialect.operation"(%operand)
{ attribute = #dialect.attr<...> } ({
^bb0(%argument: !dialect.type):
  ...
}) : (!dialect.type) -> ()
```

Not Only Operation Handles

Operation handle	`TransformHandleTypeInterface`
	`getPayloadOps / set`

Value handle	`TransformValueHandleTypeInterface`
	`getPayloadValues / setValues`

Transform parameter	`TransformParamTypeInterface`
	`getParams / setParams`

```
%result = "dialect.operation"(%operand)
{ attribute = #dialect.attr<...> } ({
^bb0(%argument: !dialect.type):
 ...
}) : (!dialect.type) -> ()
```

TBD: block handle

TBD: region handle

Can be mimicked by operation handle + region position + block position.

Parameterized Transformations

Transform *parameter* (rather than handle) is associated with a list of attributes that are shared by the transform and payload IR, owned by the common MLIR context.

Parameterized Transformations

Transform *parameter* (rather than handle) is associated with a list of attributes that are shared by the transform and payload IR, owned by the common MLIR context.

```
// Analyze the payload op at compile (transform) time to find good tile sizes. [imaginary op]
%tile_sizes = transform.default_sizes_for(%op) : (!transform.any_op) -> !transform.param<i64>
```

```
// Tiling implementation still gets fixed sizes at compile (transform) time.
transform.tile_to_forall_op %op tile_sizes[%tile_sizes : !transform.param<i64>]
```

Parameterized Transformations

Transform *parameter* (rather than handle) is associated with a list of attributes that are shared by the transform and payload IR, owned by the common MLIR context.

```
// Analyze the payload op at compile (transform) time to find good tile sizes. [imaginary op]
~~%tile_sizes = transform.default_sizes_for(%op) : (!transform.any_op) -> !transform.param<i64>~~

// Can use a machine model. [imaginary op]
%tile_sizes = transform.tile_sizes_for(%op, %machine_model)
  : (!transform.machine_model, !transform.any_op) -> !transform.param<i64>




// Tiling implementation still gets fixed sizes at compile (transform) time.
transform.tile_to_forall_op %op tile_sizes[%tile_sizes : !transform.param<i64>]
```

Parameterized Transformations

Transform *parameter* (rather than handle) is associated with a list of attributes that are shared by the transform and payload IR, owned by the common MLIR context.

```
// Analyze the payload op at compile (transform) time to find good tile sizes. [imaginary op]
~~%tile_sizes = transform.default_sizes_for(%op) : (!transform.any_op) -> !transform.param<i64>~~

// Can use a machine model. [imaginary op]
~~%tile_sizes = transform.tile_sizes_for(%op, %machine_model)~~
~~  : (!transform.machine_model, !transform.any_op) -> !transform.param<i64>~~

// Can use advanced search or ML. [imaginary op]
%tile_sizes = transform.use_ml_magic_to_find_tile_sizes(%op)
  : (!transform.any_op) -> !transform.param<i64>

// Tiling implementation still gets fixed sizes at compile (transform) time.
transform.tile_to_forall_op %op tile_sizes[%tile_sizes : !transform.param<i64>]
```

Parameterized Transformations

Transform *parameter* (rather than handle) is associated with a list of attributes that are shared by the transform and payload IR, owned by the common MLIR context.

```
// Analyze the payload op at compile (transform) time to find good tile sizes. [imaginary op]
~~%tile_sizes = transform.default_sizes_for(%op) : (!transform.any_op) -> !transform.param<i64>~~

// Can use a machine model. [imaginary op]
~~%tile_sizes = transform.tile_sizes_for(%op, %machine_model)~~
~~  : (!transform.machine_model, !transform.any_op) -> !transform.param<i64>~~

// Emit IR to compute tile sizes at runtime. [imaginary op]
%tile_sizes = transform.emit_code_for_tile_size_computation_at_runtime(%op)
  : (!transform.any_op) -> !transform.value

// This now requires support for parametric tiling.
transform.tile_to_forall_op %op tile_sizes[%tile_sizes : !transform.value]
```

Did Somebody Mention (Re-)Matching?

Suppression of diagnostics from tri-state results can be used to implement matcher operations.

If any operation fails, the entire sequence fails and produces an empty list of results.

Transformations on an empty list are noops.

Did Somebody Mention (Re-)Matching?

Suppression of diagnostics from tri-state results can be used to implement matcher operations.

If any operation fails, the entire sequence fails and produces an empty list of results.

Transformations on an empty list are noops.

```
// Stop checking, but don't report diagnostics to the user on failure.
%result = transform.sequence failures(suppress) {
^bb0(%op: !transform.any_op):
```

Did Somebody Mention (Re-)Matching?

Suppression of diagnostics from tri-state results can be used to implement matcher operations. If any operation fails, the entire sequence fails and produces an empty list of results. Transformations on an empty list are noops.

```
// Stop checking, but don’t report diagnostics to the user on failure.
%result = transform.sequence failures(suppress) {
^bb0(%op: !transform.any_op):
  // Operation of kind “linalg.generic”,
  transform.match.operation_name %op["linalg.generic"] : !transform.any_op

}
```

Did Somebody Mention (Re-)Matching?

Suppression of diagnostics from tri-state results can be used to implement matcher operations.
If any operation fails, the entire sequence fails and produces an empty list of results.
Transformations on an empty list are noops.

```
// Stop checking, but don’t report diagnostics to the user on failure.
%result = transform.sequence failures(suppress) {
^bb0(%op: !transform.any_op):
  // Operation of kind “linalg.generic”,
  transform.match.operation_name %op["linalg.generic"] : !transform.any_op
  transform.match.structured %op : !transform.any_op {
  ^bb1(%str: !transform.any_op):
    // with all dimensions but the last being parallel,
    transform.match.structured.dim %str[except(-1)] { parallel } : !transform.any_op
```

Did Somebody Mention (Re-)Matching?

Suppression of diagnostics from tri-state results can be used to implement matcher operations. If any operation fails, the entire sequence fails and produces an empty list of results. Transformations on an empty list are noops.

```
// Stop checking, but don’t report diagnostics to the user on failure.
%result = transform.sequence failures(suppress) {
^bb0(%op: !transform.any_op):
  // Operation of kind “linalg.generic”,
  transform.match.operation_name %op["linalg.generic"] : !transform.any_op
  transform.match.structured %op : !transform.any_op {
  ^bb1(%str: !transform.any_op):
    // with all dimensions but the last being parallel,
    transform.match.structured.dim %str[except(-1)] { parallel } : !transform.any_op
    // and the last dimension being a reduction...
    %dim = transform.match.structured.dim %str[-1] { reduction } : (!transform.any_op) -> !transform.param<i64>

}
```

Did Somebody Mention (Re-)Matching?

Suppression of diagnostics from tri-state results can be used to implement matcher operations. If any operation fails, the entire sequence fails and produces an empty list of results. Transformations on an empty list are noops.

```
// Stop checking, but don’t report diagnostics to the user on failure.
%result = transform.sequence failures(suppress) {
^bb0(%op: !transform.any_op):
  // Operation of kind “linalg.generic”,
  transform.match.operation_name %op["linalg.generic"] : !transform.any_op
  transform.match.structured %op : !transform.any_op {
  ^bb1(%str: !transform.any_op):
    // with all dimensions but the last being parallel,
    transform.match.structured.dim %str[except(-1)] { parallel } : !transform.any_op
    // and the last dimension being a reduction...
    %dim = transform.match.structured.dim %str[-1] { reduction } : (!transform.any_op) -> !transform.param<i64>
    // ...of size less than or equal to 32.
    %c32 = transform.param.constant 32 : i64 -> !transform.param<i64>
    transform.match.param.cmpi le %dim, %c32 : !transform.param<i64>
    transform.match.structured.yield
  }
  transform.yield %op : !transform.any_op
}
```

Isn’t PDL Enough?

PDL does provide matching capabilities and is optimized for large sets of patterns.

PDL does *not* support user-defined matchers for custom operations beyond C++ callbacks.

Isn’t PDL Enough?

PDL does provide matching capabilities and is optimized for large sets of patterns.

PDL does *not* support user-defined matchers for custom operations beyond C++ callbacks.

```
%dim = transform.match.structured.dim %str[-1] { reduction } : (!transform.any_op) -> !transform.param<i64>
```

Isn’t PDL Enough?

PDL does provide matching capabilities and is optimized for large sets of patterns.

PDL does *not* support user-defined matchers for custom operations beyond C++ callbacks.

```
%dim = transform.match.structured.dim %str[-1] { reduction } : (!transform.any_op) -> !transform.param<i64>
```

1. Get the list of indexing maps for all operands.
2. Concatenate those lists into a single map.
3. The dimensionality of the op is rank of the map domain (LHS).
4. The last dimension is `rank - 1`.
5. The kind of dimension is given by the enum attribute in `rank - 1` position in the `iterator_types` array attribute.
6. Invert the concatenated map from above.
7. Iterate over the inverted map to find the operand dimension that maps dimension `rank - 1` maps to.
8. If the dimension is static, create the attribute containing its value.

Isn’t PDL Enough?

Transform dialect can interoperate with PDL if desired.

Isn’t PDL Enough?

Transform dialect can interoperate with PDL if desired.

```
transform.with_pdl_patterns {
^bb0(%op: !transform.any_op):
  // A regular PDL pattern with a special rewriting hook.
  pdl.pattern @pattern : benefit(1) {
    %0 = pdl.operation "test.some_op"
    pdl.rewrite %0 with "transform.dialect"
  }







}
```

Isn’t PDL Enough?

Transform dialect can interoperate with PDL if desired.

```
transform.with_pdl_patterns {
^bb0(%op: !transform.any_op):
  // A regular PDL pattern with a special rewriting hook.
  pdl.pattern @pattern : benefit(1) {
    %0 = pdl.operation "test.some_op"
    pdl.rewrite %0 with "transform.dialect"
  }

  transform.sequence %op : !transform.any_op failures(suppress)  {
  ^bb1(%nested: !transform.any_op):
    // Use the pattern within PDL interpreter to perform the match.
    transform.pdl_match @pattern in %nested : (!transform.any_op) -> !pdl.operation
  }
}
```

PDL operations can also be seen as a special case of transform dialect operations.

Google Research

Not Only Structured Ops

Transform Dialect is not limited to structured ops.

There are downstream extensions for affine loop transformations, memref-level optimization.

Not Only Structured Ops

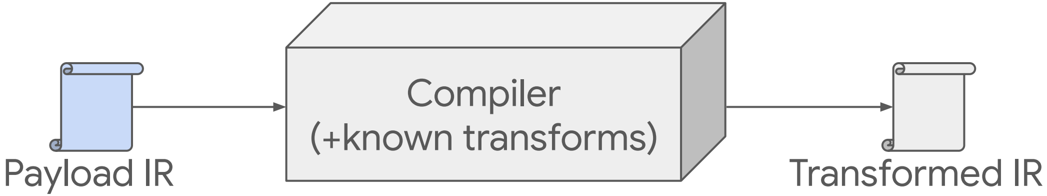
Transform Dialect is not limited to structured ops.

There are downstream extensions for affine loop transformations, memref-level optimization.

It’s not even limited to ops, here’s a pass pipeline:

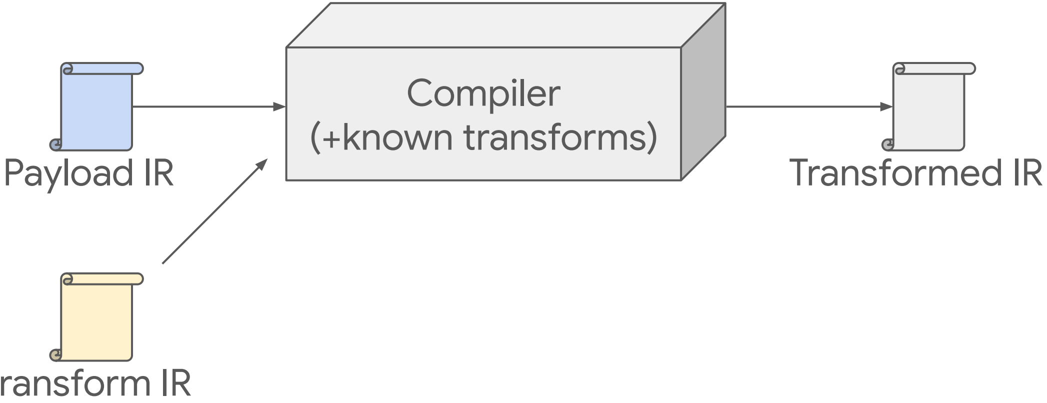
```
transform.pass.pipeline on "builtin.module" {
  transform.pass.pipeline on "func.func" {
    transform.pass "test-transform-dialect-interpreter" {
      bind_first_extra_to_ops="linalg.matmul",
      bind_second_extra_to_ops="linalg.elemwise_binary",
      enable_expensive_checks,
    }
    transform.pass "canonicalize"
    transform.pass "cse"
  }
  transform.pass "test-transform-dialect-drop-schedule"
}
// These ops don’t exist (yet?)
```

Towards Extensible Controllable Compilers

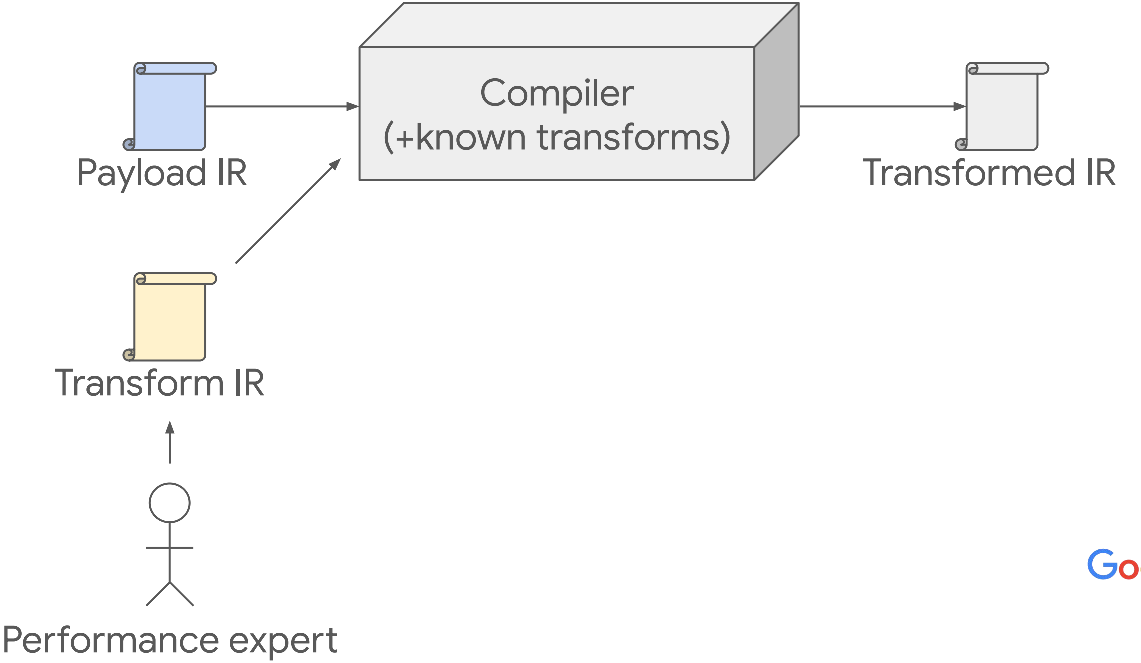


Google Research

Towards Extensible Controllable Compilers

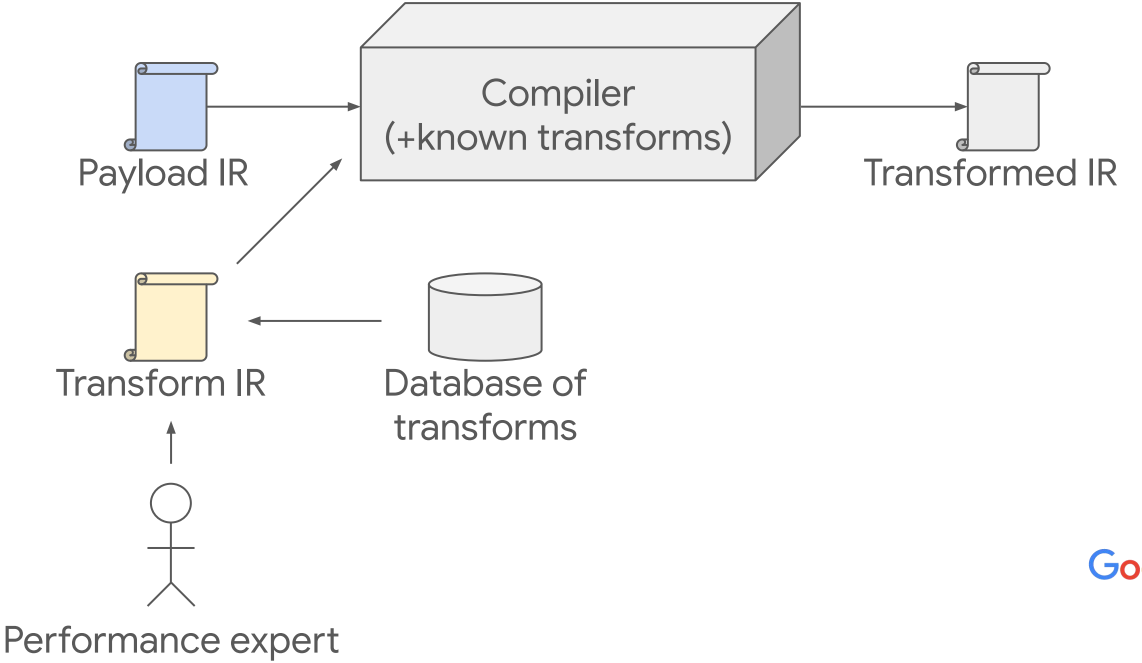


Towards Extensible Controllable Compilers



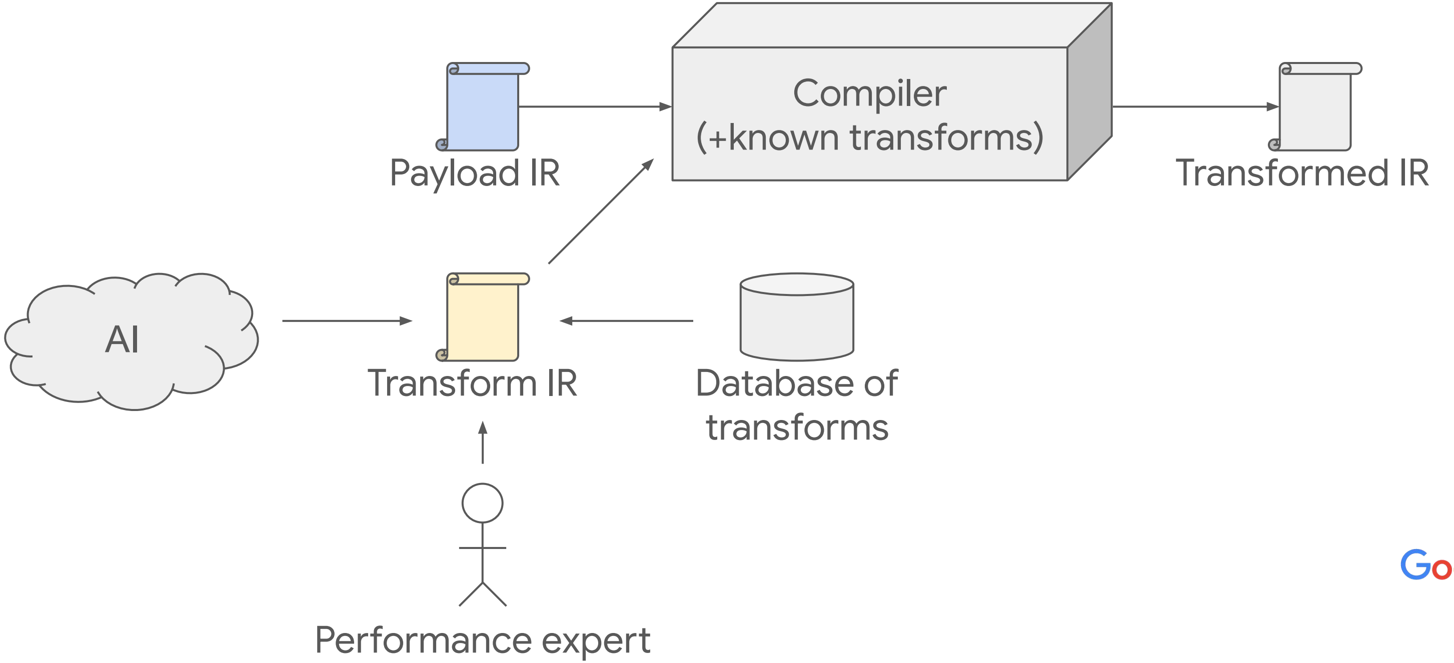
Google Research

Towards Extensible Controllable Compilers

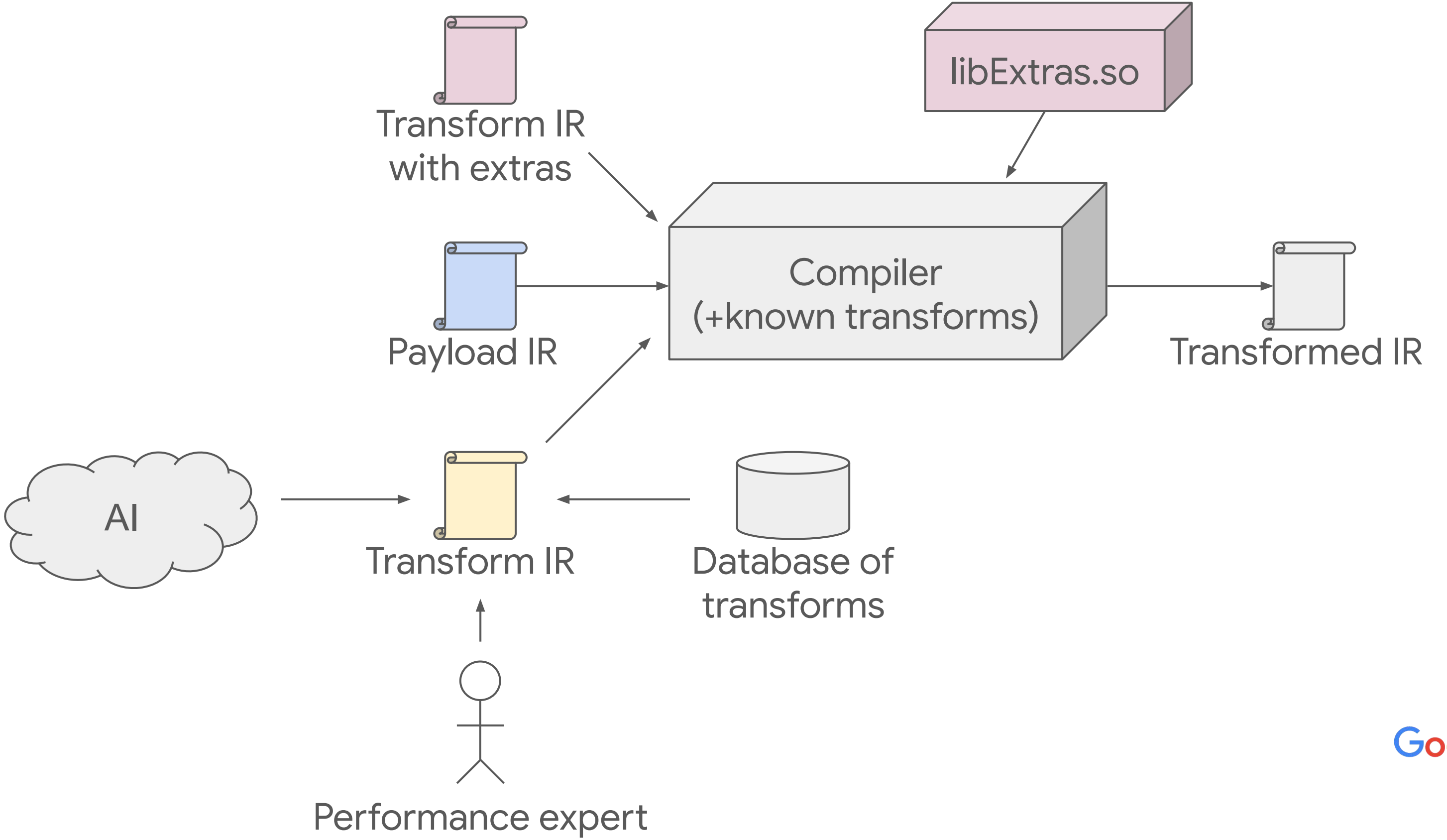


Google Research

Towards Extensible Controllable Compilers



Towards Extensible Controllable Compilers



Thank you!

```
%deck = transform.deck.create {name = "Controllable Transformations in MLIR",
                               author = ["Alex Zinenko <zinenko@google.com>"]}
transform.foreach %case in %interesting_scenarios: !transform.ir {
 %slide = transform.deck.make_a_slide_about %case : !transform.ir -> !slides.slide
 transform.deck.insert %slide, %deck : !slides.deck
}

// * Imaginary ops (I wish these had existed before doing the slides).
```

Google Research