Controllable Transformations in MLIR

Alex Zinenko

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 …
```

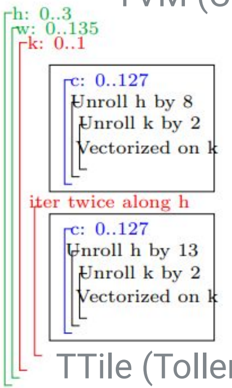
+ Map to Accelerator Tensor Instructions

```
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_2,2))
fullunroll(enclose(C3L1_2_1_2_1_2_1_2,1))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 {
}
```

Payload IR

Perform transformations one after another.

```
%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.)
```

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.)
```

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.)
```

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<...>
```

Handle Invalidation Continued

Transform IR

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

```
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
```

```
}
```

Payload IR

```
    %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<...>
```

03

Let’s generalize!

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.

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.

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.

Things to know:

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

.td

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


}
```

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.

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.

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.

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.

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

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.

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.

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.

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

```
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.

Thank you!

```
%deck = transform.deck.create {name = "Controllable Transformations in MLIR",
                                author = ["Alex Zinenko ]}
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).
```