Controllable Transformations in MLIR

Alex Zinenko
Denys Shabalin
Nicolas Vasilache

Google Research

Agenda

01 Why?

02 How?

03 So what?

Google Research

01

Why?

Scheduling 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_0
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 …
```

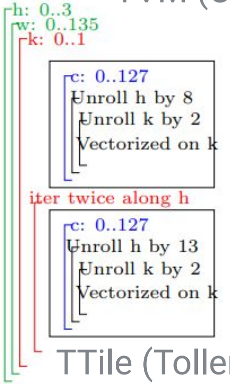
+ Map to Accelerator Tensor Instructions

```
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 in the Wild

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

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

- 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).

02

How?

Dialect for schedules in MLIR

- In MLIR, everything is a dialect. So are schedules.
- Extended the dialect extension mechanism to allow clients to inject ops.
- Combine transformations in various ways: “then”, “repeat foreach”, “try-catch-try-something-else”, …
- Schedules-as-IR allow us to decouple the “schedule generator” from the “schedule applicator”.

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op):

    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [42, 32] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    %retiled, %inner_loop = tile_to_scf_for %op { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

May be the same module a separate module.

Top-level operation applies to the payload “root” supplied in C++.

The combinator is also an op (top-level is always a single op).

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op):

    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [42, 32] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    %retiled, %inner_loop = tile_to_scf_for %op { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

The “handle” value corresponds to an ordered list of payload ops.

Handle types describe constraints on payload ops.

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op):

    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [42, 32] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    %retiled, %inner_loop = tile_to_scf_for %op { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

The “handle” value corresponds to an ordered list of payload ops.

Handle types describe constraints on payload ops.

Op semantics may require types or list (e.g., singleton) properties.

Google Research

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op):

    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [42, 32] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    %retiled, %inner_loop = tile_to_scf_for %tiled { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

Transformations can be *chained* as opposed to globally applied.

Google Research

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op):

    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [42, 32] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    %retiled, %inner_loop = tile_to_scf_for %tiled { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

Transformations can be *chained* as opposed to globally applied.

Google Research

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op):

    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [42, 32] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    %retiled, %inner_loop = tile_to_scf_for %tiled { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

Transformations can be *chained* as opposed to globally applied.

Google Research

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op):

    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [42, 32] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    %retiled, %inner_loop = tile_to_scf_for %tiled { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

Transformations can be *chained* as opposed to globally applied.

Google Research

Transform dialect by example

```
module @payload {
 // Some computational IR.
}

module @transform {
  transform.sequence {
  ^bb0(%arg0: !transform.any_op, %arg1: !transform.param<i64>,
       %arg2: !transform.param<i64>):
    %op = match #transform.interface<Tile> in %arg0
      : !transform.interface<Tile>

    %tiled, %outer_loops:2 = tile_to_scf_for %op { sizes = [%arg1, %arg2] }
      : !transform.interface<Tile>, !transform.param<i64>, !transform.param<i64>

    %retiled, %inner_loop = tile_to_scf_for %tiled { sizes = [0, 4] }
      : !transform.interface<Tile>, !transform.op<"scf.for">

    loop.vectorize %inner_loop : !transform.op<"scf.for">

    loop.unroll %outer_loops#1 : !transform.op<"scf.for">
  }
}
```

Parameters contain values known at transform dialect execution time, such as sizes.

Implementation

Transform ops implement an interface that describes *how* to apply the transform.

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;

};
```

Google Research

Implementation

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;

};
```

Transform ops implement an interface that describes *how* to apply the transform.

Tri-state result type (definite/recoverable error, success) + optional diagnostic.

Implementation

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;

};
```

Transform ops implement an interface that describes *how* to apply the transform.

Tri-state result type (definite/recoverable error, success) + optional diagnostic.

The implementation sets up the mapping between result handles and payload ops.

Implementation

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;

};
```

Transform ops implement an interface that describes *how* to apply the transform.

Tri-state result type (definite/recoverable error, success) + optional diagnostic.

The implementation sets up the mapping between result handles and payload ops.

The mapping is maintained by the state; custom state extensions are supported.

Examples of transforms

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;
```

Take the “iteration space” ops associated with the first handle/operand and tile them using the sizes provided as attribute.

`transform.structured.tile`

Google Research

```
};
```

Examples of transforms

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;
```

Go over all (transform) ops in the attached single-block region and apply them one by one; fail if any of them fails.

`transform.sequence`

Google Research

```
};
```

Examples of transforms

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;
```

Clone the payload IR, apply the transform from the first region. If failed, take the original IR, clone it, apply the transform from the second region, etc until succeeds.

`transform.alternatives`

```
};
```

Examples of transforms

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;

};
```

Apply the transform from another "symbol" op depending on the properties of the payload op, such as rank or shape.

`TBD: transform.dispatch`

Examples of transforms

```
class TransformOpInterface : public OpInterface<...> {

 virtual DiagnosedSilenceableFailure apply(
   TransformResults &results,
   /*const*/ TransformState &state) const;
```

Create a new handle pointing to the same payload operations as the handles given as operands.

`transform.merge_handles`


```
};
```

Handle invalidation: a data flow problem

```
transform.sequence {
  %handles = merge_handles %h1, %h2
  erase_op %h1
  print %h1

}
```

Use-after-free are possible with handles, but luckily we are in the SSA world and can just run data flow analysis over side effects.

Handle invalidation: a data flow problem

```
transform.sequence {
  %handles = merge_handles %h1, %h2
  erase_op %h1

  print %handles
}
```

May also need aliasing and be aware of the nesting relationship between payload IR ops.

Since transforms are just IR, we can analyze and transform them!

03

So what?

“Deep” and “wide” transformations

Introduce loops everywhere

Tile all loops

Bufferize everything

Vectorize all loops

Vectorize straight-line code

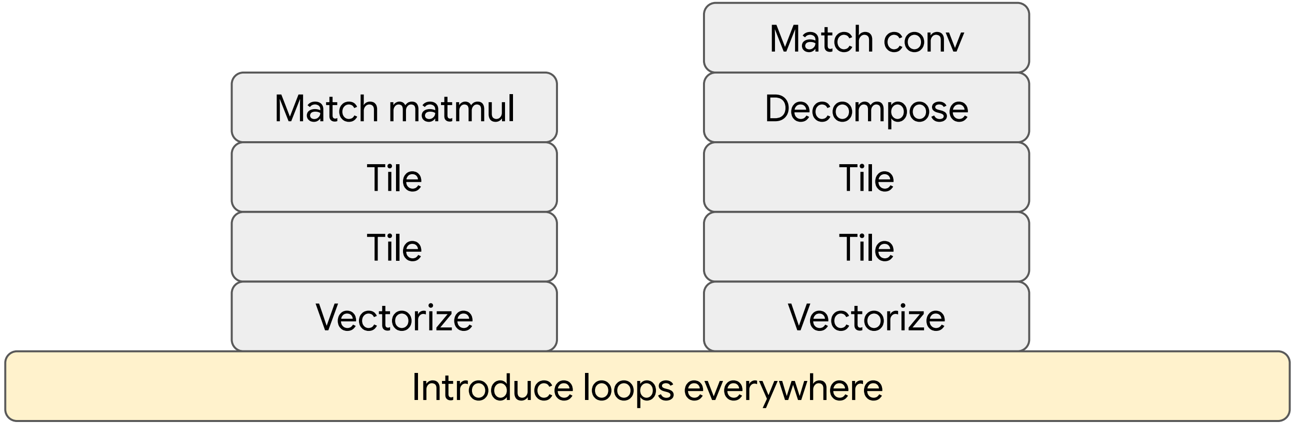
What if we only want to tile some loops?

... if we want to vectorize only the loops produced by tiling?

“Deep” and “wide” transformations

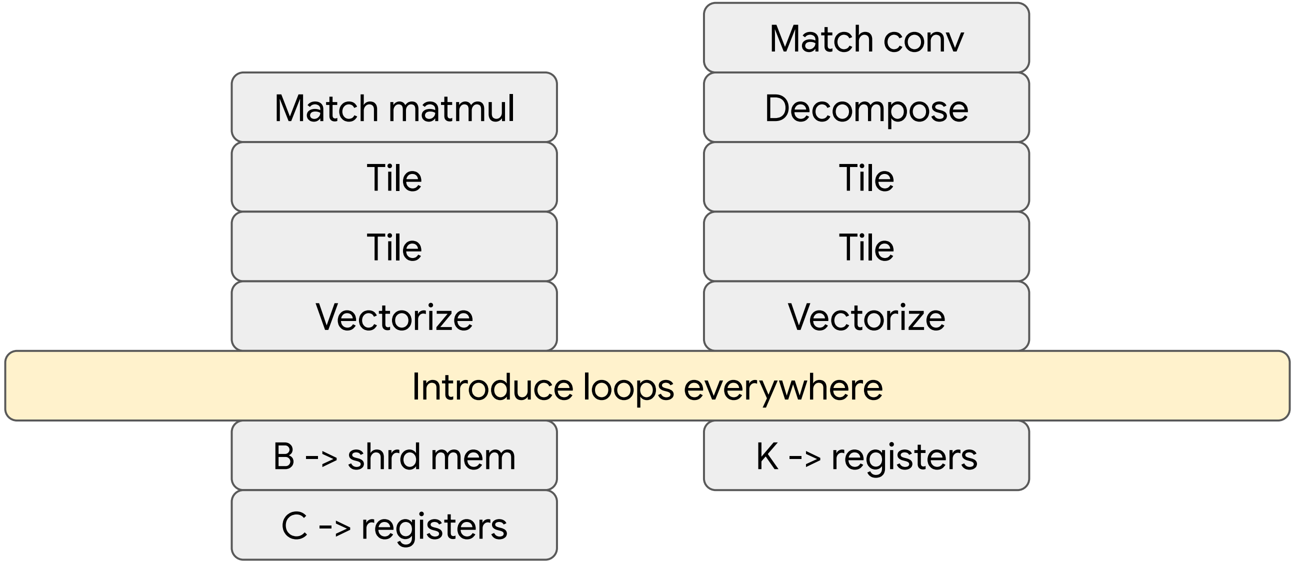
	Match conv
Match matmul	Decompose
Tile	Tile
Tile	Tile
Vectorize	Vectorize

“Deep” and “wide” transformations

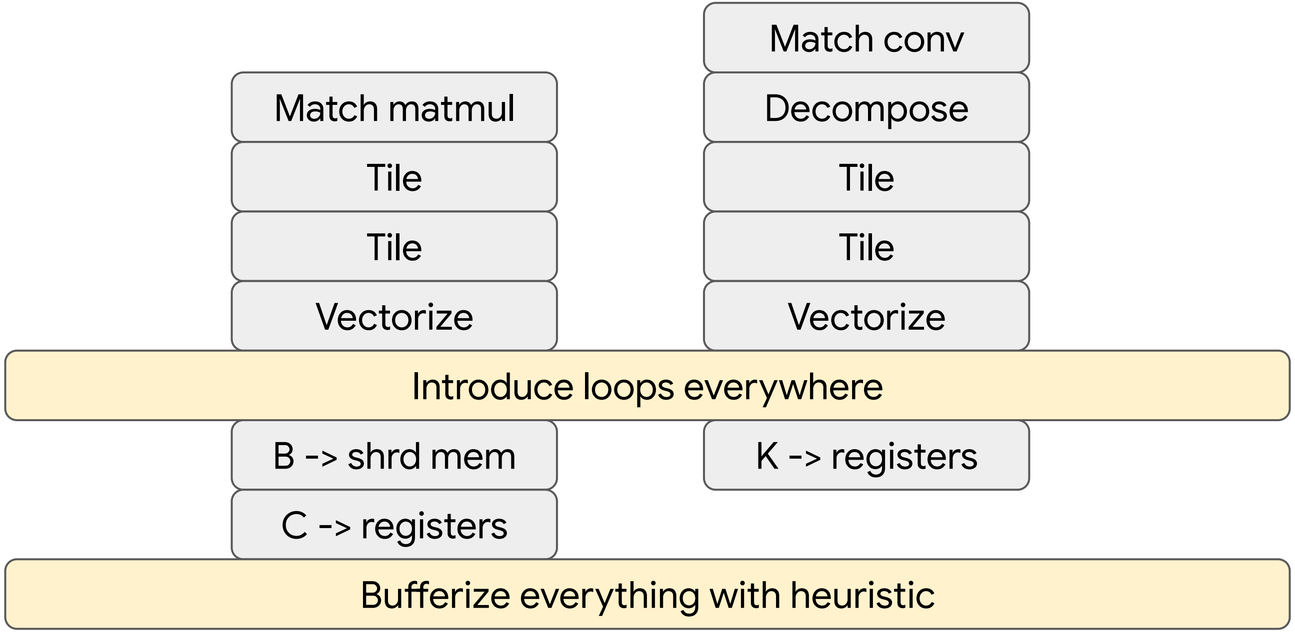


Google Research

“Deep” and “wide” transformations

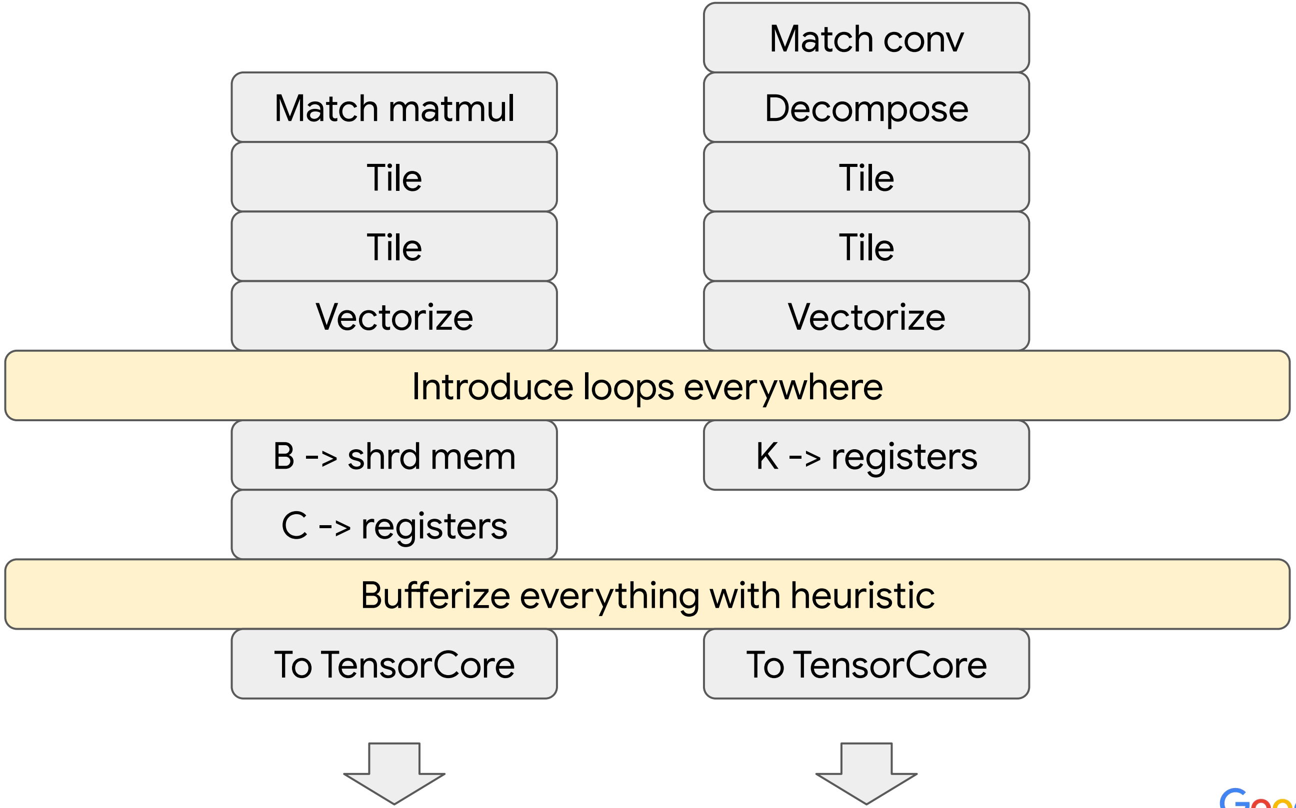


“Deep” and “wide” transformations

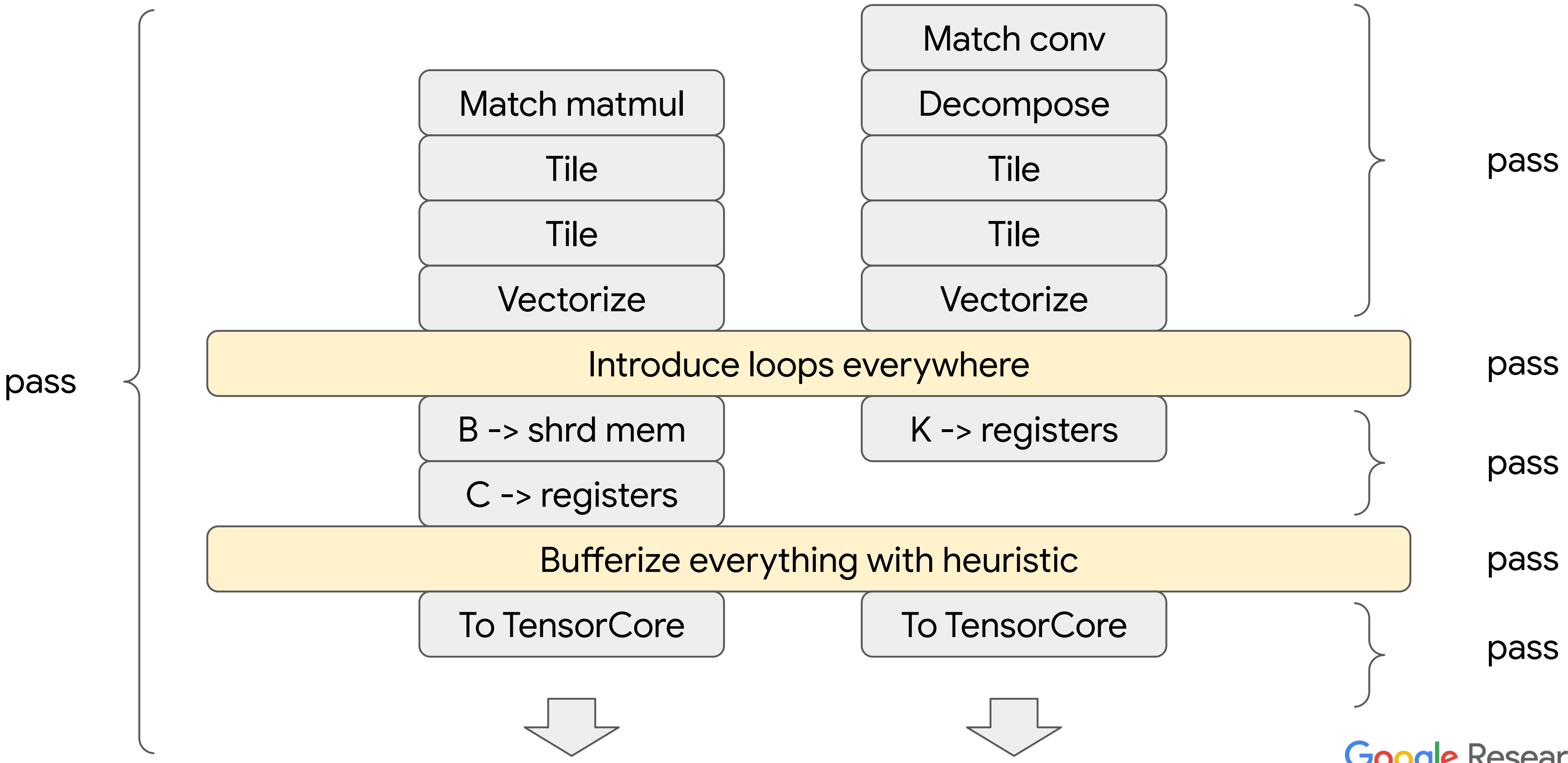


Google Research

“Deep” and “wide” transformations



Where does this leave us with passes?



Google Research

Where does this leave us with passes?

	Match conv	pass
Match matmul	Decompose	
Tile	Tile	
Tile	Tile	
Vectorize	Vectorize	
Introduce loops **everywhere**		pass
B -> shrd mem	K -> registers	pass
C -> registers		
Bufferize **everything** with heuristic		pass
To TensorCore	To TensorCore	pass

Google Research

“States” of the IR

Graph-level IR on tensors

Introduce loops everywhere

IR with loops on tensors

Bufferize everything with heuristic

IR with loops on buffers

“States” of the IR

Have transforms/patterns describe effects on the state of the IR, and register them.

Automatically “construct” passes to have the desired effect such as introducing loops.

Graph-level IR on tensors

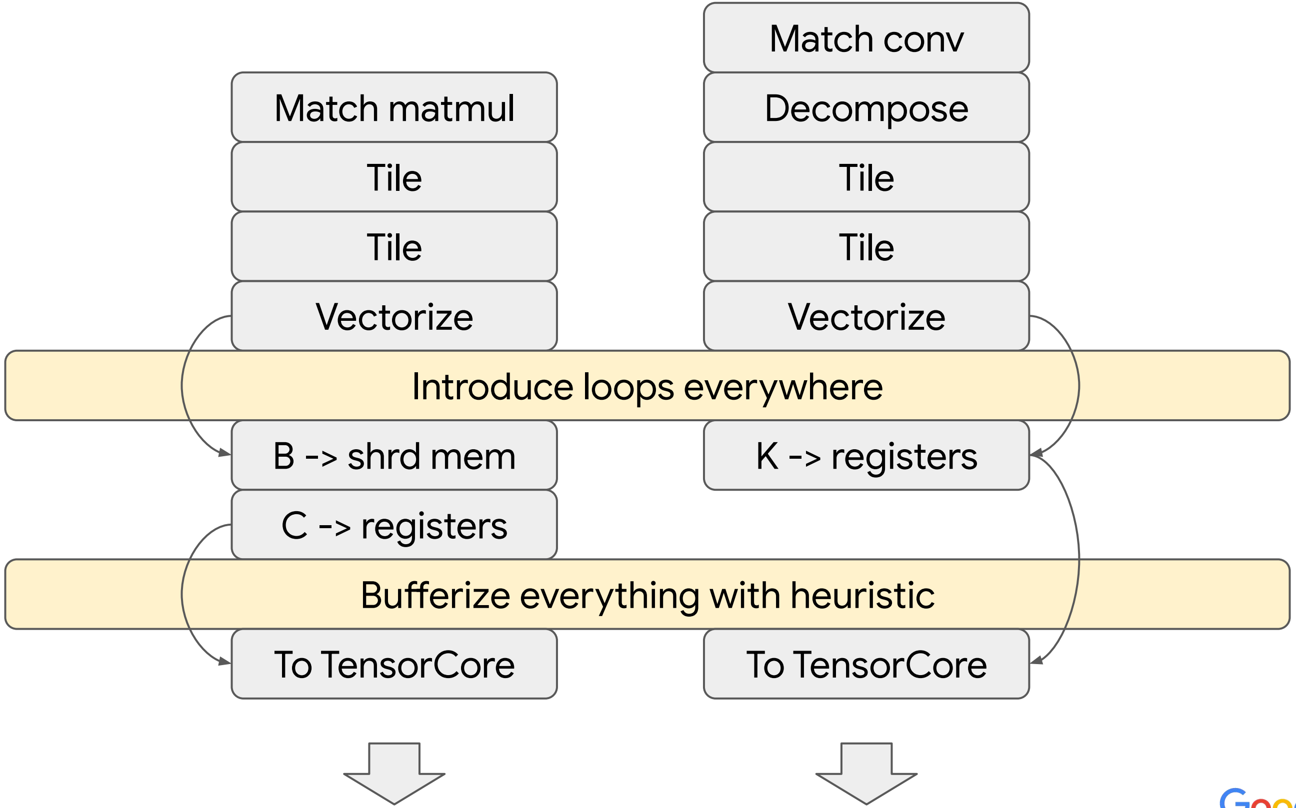
Introduce loops everywhere

IR with loops on tensors

Bufferize everything with heuristic

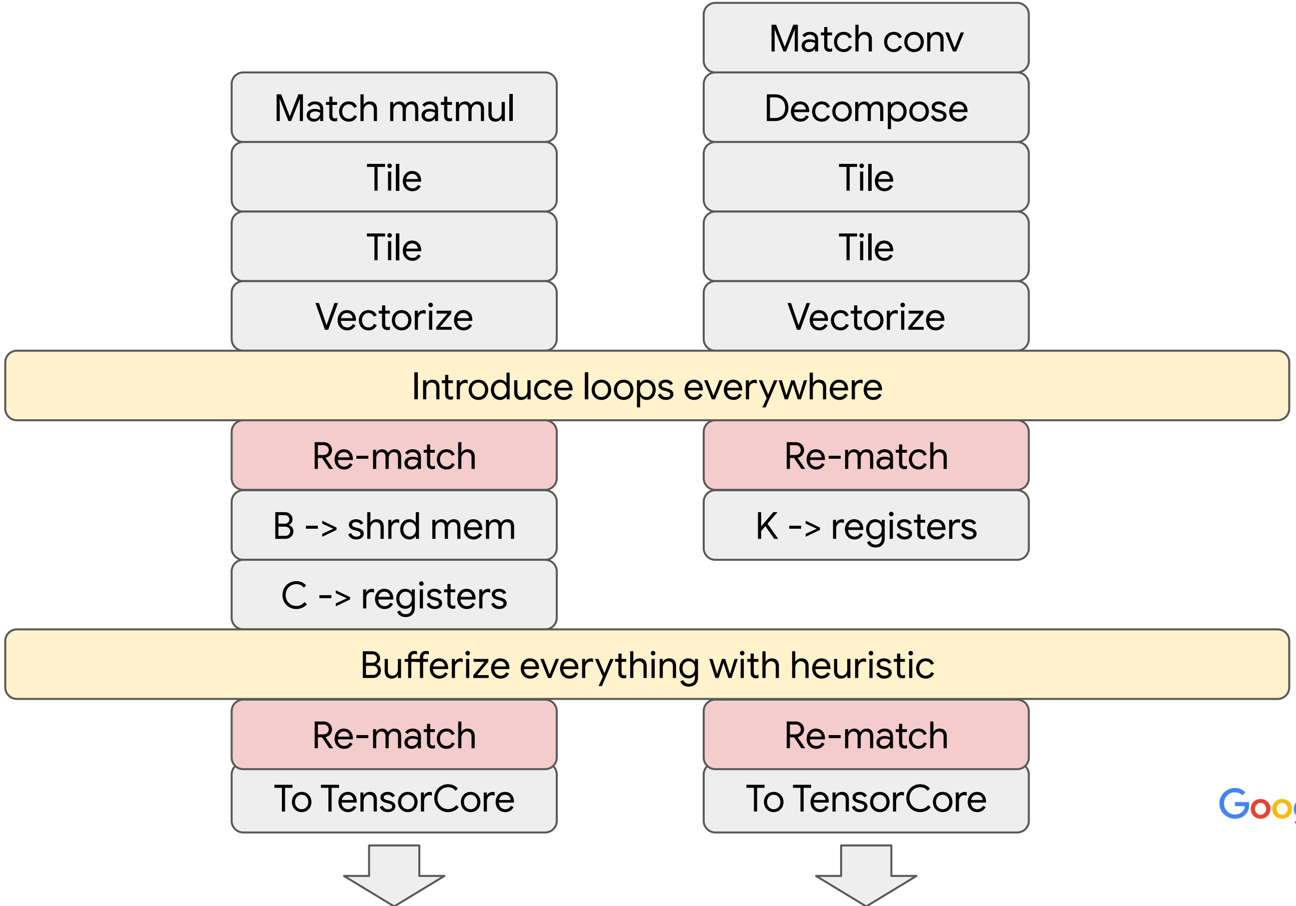
IR with loops on buffers

Preserve transformation continuity



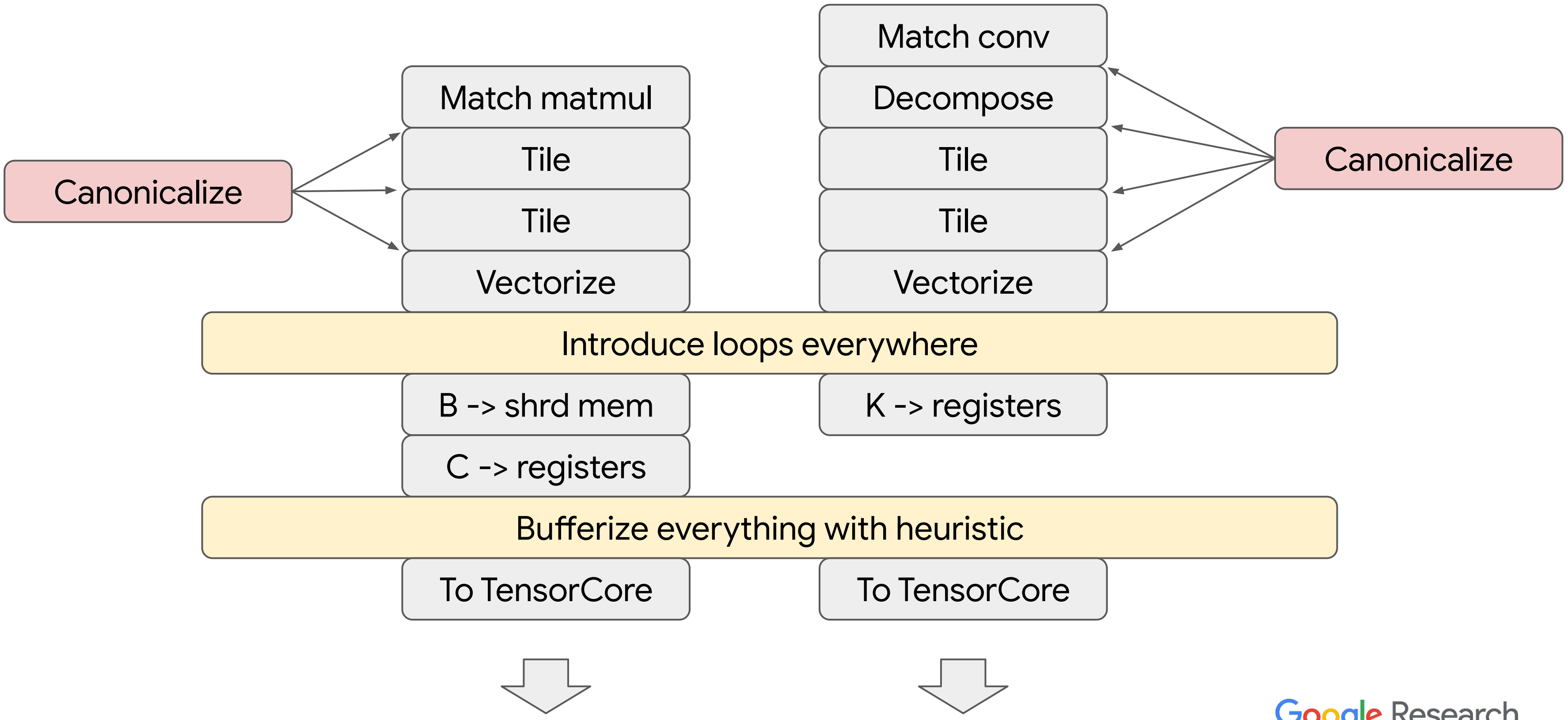
Google Research

Preserve transformation continuity



Google Research

Local canonicalization or cleanup



Google Research

Transformation-driven compiler design

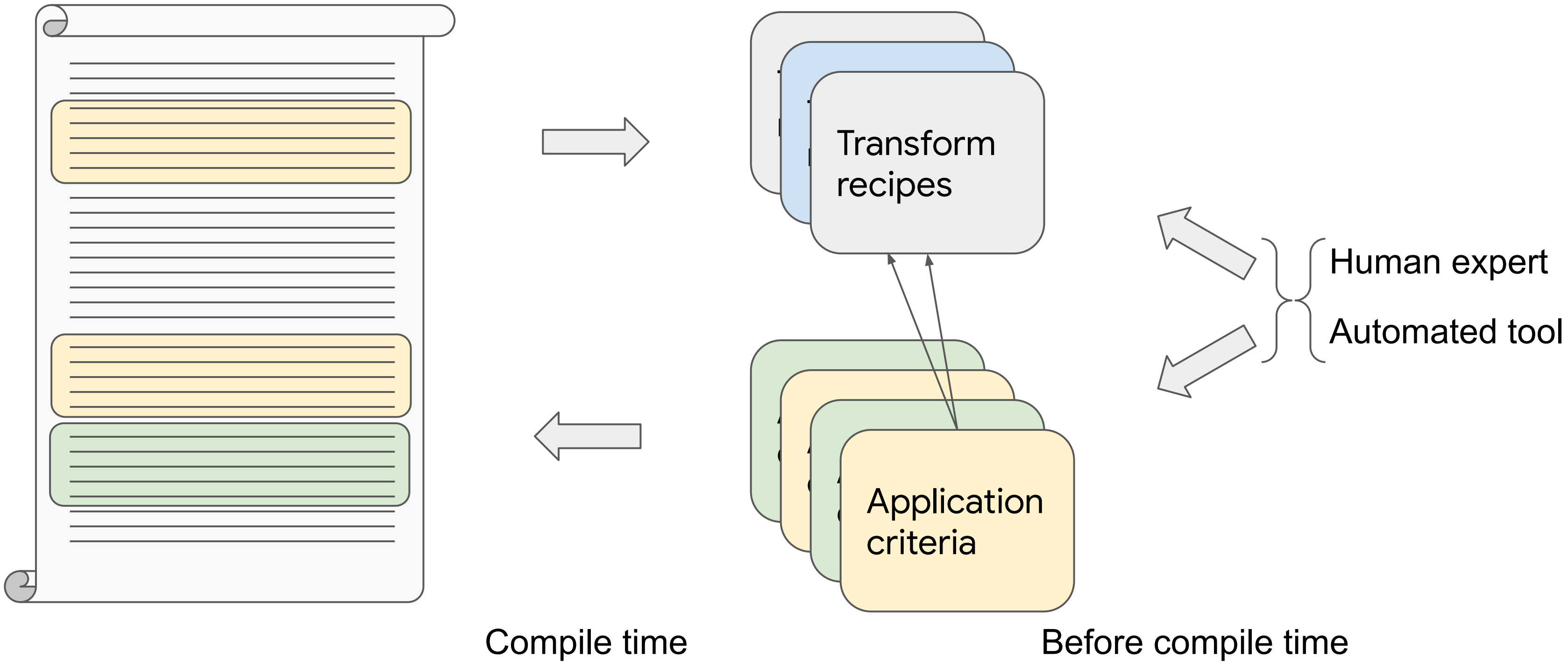
- Rewrite-like transformations should have *results* not only status, e.g., the loops introduced by tiling, the op itself after bufferization.

- Infrastructure for “wide” transformations needs listeners/tracking. (Reminder: MLIR intro paper advertised the traceability principle).

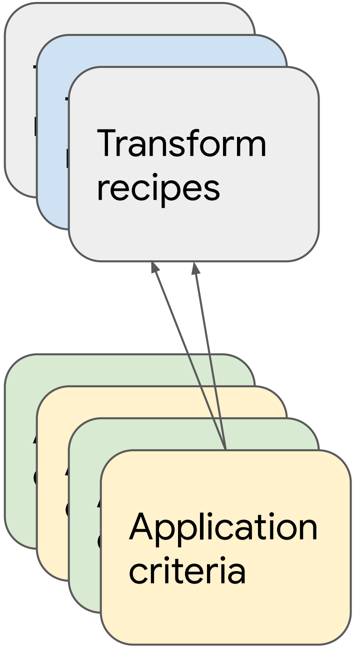
- Local canonicalization / clean-up capabilities, we don’t expect individual transforms to keep track of what is canonical today.

- When tempted to design a dialect customized for transformation, design an interface instead! The dialect is then just a test.

Externalizing compiler heuristics



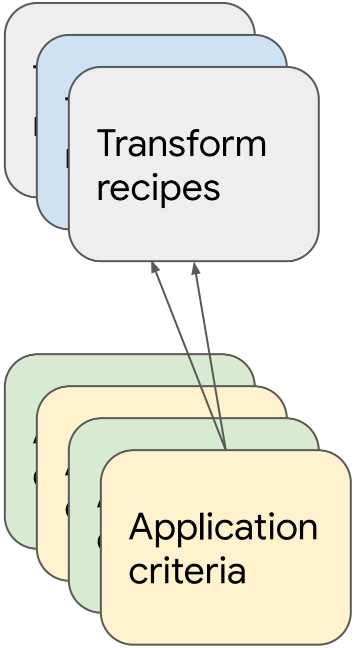
Externalizing compiler heuristics




```
transform.recipe @double_tile_unroll(
   %op: !transform.any_op, %sz:4: !transform.single_result_op) {
 %t1, %l1:2 = tile %op sizes(%sz#0, %sz#1)
 %t2, %l2:2 = tile %t1 sizes(%sz#2, %sz#3)
 include @unroll_and_simplify %l2#1
 yield %t2, %l2#0
}
```

```
transform.strategy {
 %op = pdl_match @matmul : !transform.interface<Structured>
 if (rank(%op) == 2) {
   if (dim(%op, 0) > 512) { @double_tile_unroll(%op, 32, 64, 4, 8) }
   else { @double_tile_unroll(%op, 32, 16, 4, 8) }
 } elif rank(%op) == 3 {
   if (dim(%op, 1) > 512) { @double_tile_unroll(%op, 8, 64, 4, 8) }
   else { @double_tile_unroll(%op, 8, 16, 4, 8) }
 } ...
}
```

Externalizing compiler heuristics

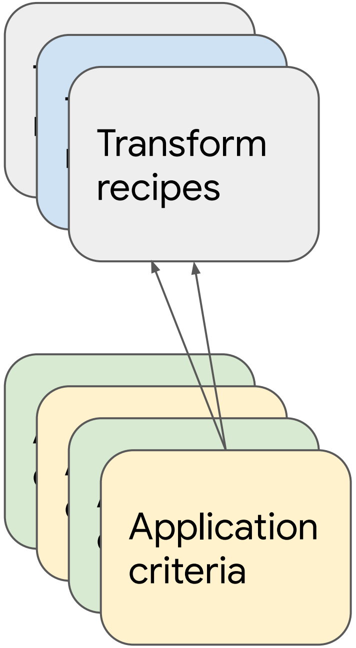



```
transform.recipe @double_tile_unroll(
   %op: !transform.any_op, %sz:4: !transform.single_result_op) {
 %t1, %l1:2 = tile %op sizes(%sz#0, %sz#1)
 %t2, %l2:2 = tile %t1 sizes(%sz#2, %sz#3)
 include @unroll_and_simplify %l2#1
 yield %t2, %l2#0
}
```

Named, reusable snippets

```
transform.strategy {
 %op = pdl_match @matmul : !transform.interface<Structured>
 if (rank(%op) == 2) {
   if (dim(%op, 0) > 512) { @double_tile_unroll(%op, 32, 64, 4, 8) }
   else { @double_tile_unroll(%op, 32, 16, 4, 8) }
 } elif rank(%op) == 3 {
   if (dim(%op, 1) > 512) { @double_tile_unroll(%op, 8, 64, 4, 8) }
   else { @double_tile_unroll(%op, 8, 16, 4, 8) }
 } ...
}
```

Externalizing compiler heuristics

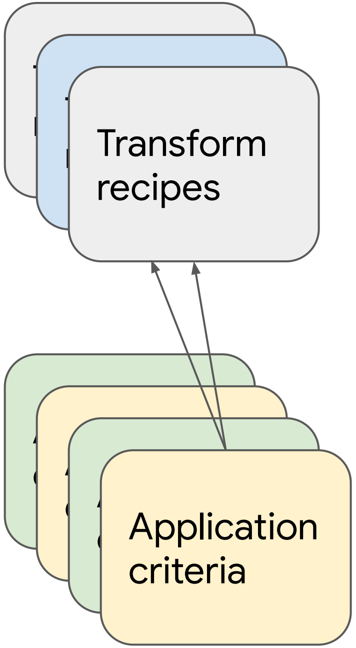



```
transform.recipe @double_tile_unroll(
   %op: !transform.any_op, %sz:4: !transform.single_result_op) {
 %t1, %l1:2 = tile %op sizes(%sz#0, %sz#1)
 %t2, %l2:2 = tile %t1 sizes(%sz#2, %sz#3)
 include @unroll_and_simplify %l2#1
 yield %t2, %l2#0
}
```

```
transform.strategy {
 %op = pdl_match @matmul : !transform.interface<Structured>
 if (rank(%op) == 2) {
   if (dim(%op, 0) > 512) { @double_tile_unroll(%op, 32, 64, 4, 8) }
   else { @double_tile_unroll(%op, 32, 16, 4, 8) }
 } elif rank(%op) == 3 {
   if (dim(%op, 1) > 512) { @double_tile_unroll(%op, 8, 64, 4, 8) }
   else { @double_tile_unroll(%op, 8, 16, 4, 8) }
 } ...
}
```

Google Research

Externalizing compiler heuristics




```
transform.recipe @double_tile_unroll(
   %op: !transform.any_op, %sz:4: !transform.single_result_op) {
 %t1, %l1:2 = tile %op sizes(%sz#0, %sz#1)
 %t2, %l2:2 = tile %t1 sizes(%sz#2, %sz#3)
 include @unroll_and_simplify %l2#1
 yield %t2, %l2#0
}
```

Can be “interpreted” by the compiler or codegen for runtime multiversioning.

```
transform.strategy {
 %op = pdl_match @matmul : !transform.interface<Structured>
 if (rank(%op) == 2) {
   if (dim(%op, 0) > 512) { @double_tile_unroll(%op, 32, 64, 4, 8) }
   else { @double_tile_unroll(%op, 32, 16, 4, 8) }
 } elif rank(%op) == 3 {
   if (dim(%op, 1) > 512) { @double_tile_unroll(%op, 8, 64, 4, 8) }
   else { @double_tile_unroll(%op, 8, 16, 4, 8) }
 } ...
}
```

Ehm?

Summary

- Transform dialect brings the concept of schedules to MLIR.

- It is extensible like anything is expected to be in MLIR.

- We need better infrastructural support for precise controllable transformations.

- Having schedules as IR enables new exciting compiler and compiler/ML-interface work.