Structured Ops in MLIR

Compiling Loops, Libraries and DSLs

MLIR Open Design Meeting - Dec 5th 2019

Albert Cohen, Andy Davis, Nicolas Vasilache, Alex Zinenko

Outline

MLIR Dialects for Codegen

Structured Ops Abstraction - Buffers

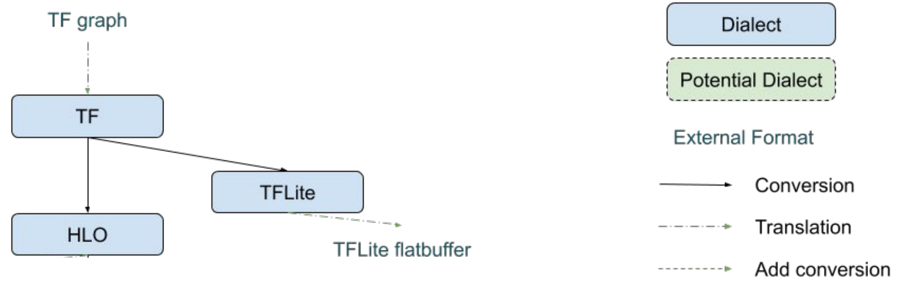
Structured Ops Abstraction - Vectors

Declarative Transformations with Composable Patterns

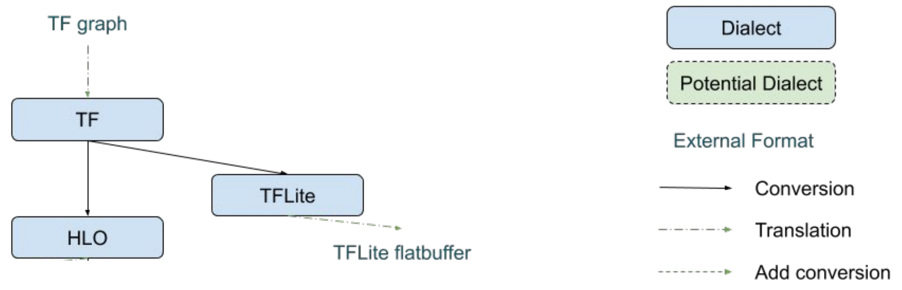
Conclusion

MLIR Dialects for Codegen

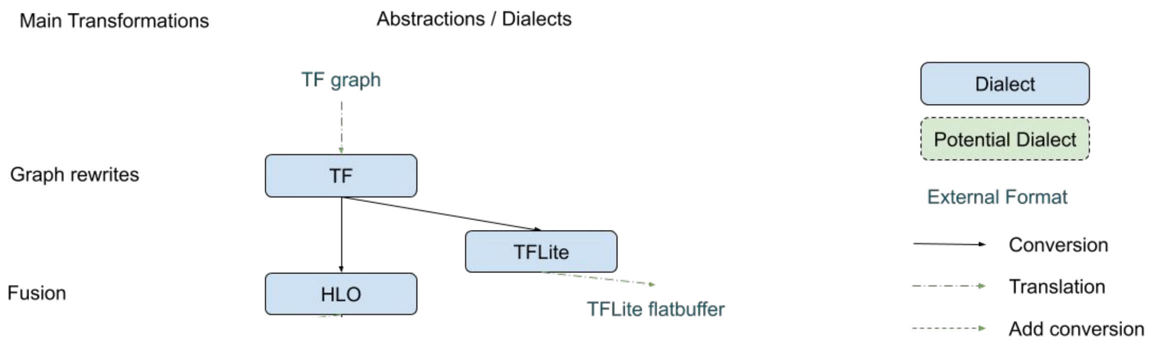
Abstractions / Dialects

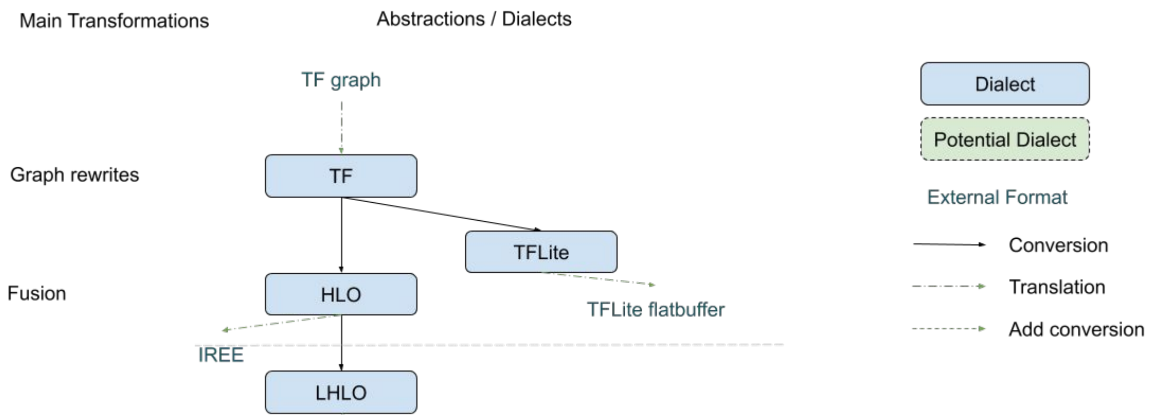


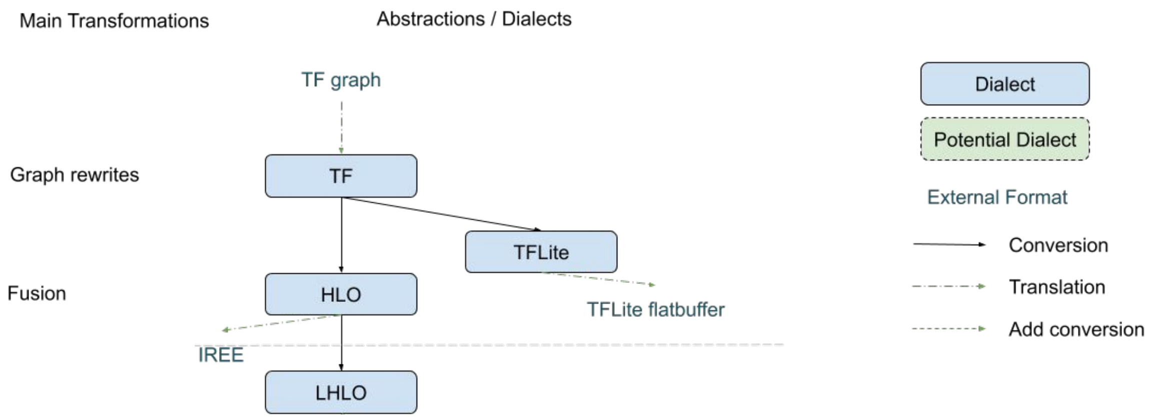
Abstractions / Dialects




```
%0 = "xla_hlo.dot"(%arg0, %arg1) : (tensor<2x2xi32>, tensor<2x2xi32>) -> tensor<2x2xi32>
```

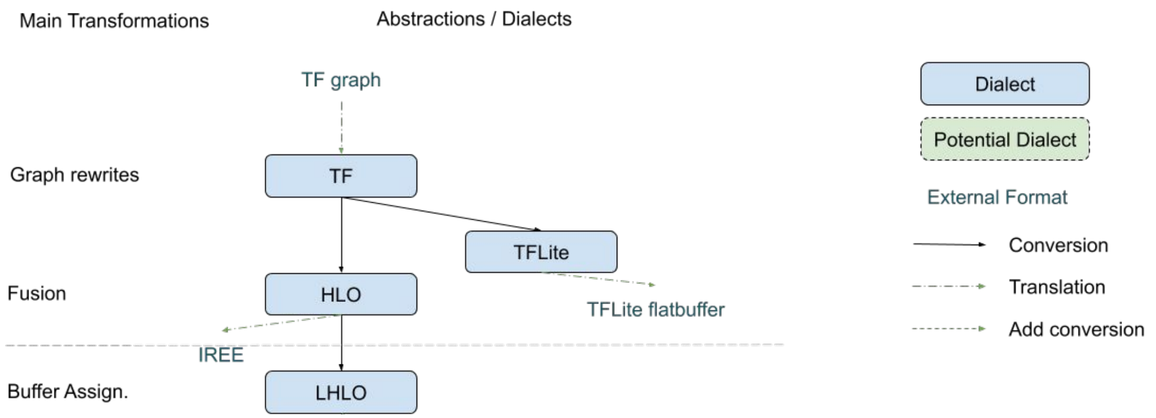

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion


Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
LHLO
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

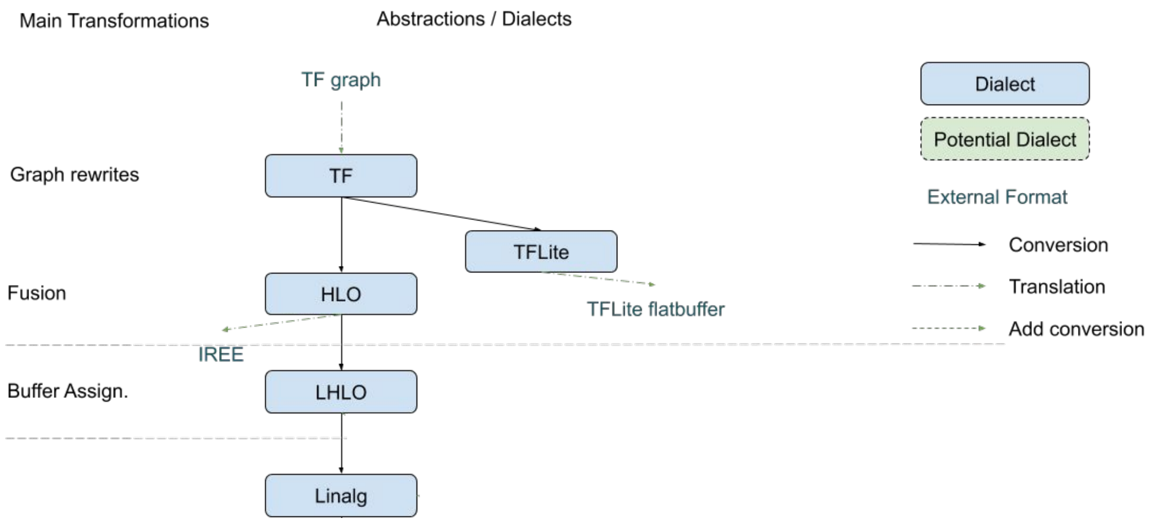


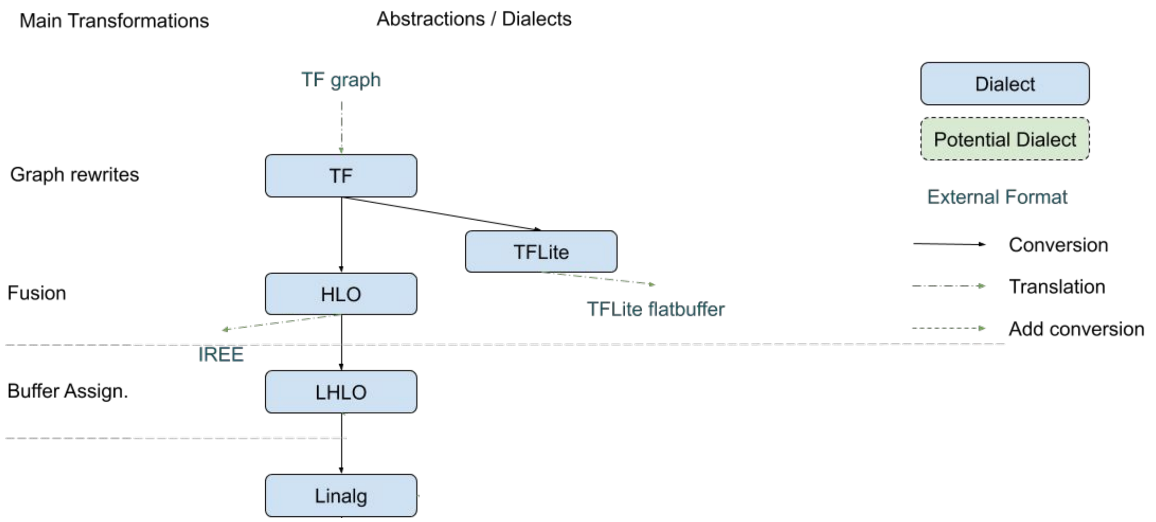

```
"xla_lhlo.mul"(%lhs, %rhs, %out) : (memref<10xf32>, memref<10xf32>, memref<10xf32>) -> ()
```

```
"xla_lhlo.fusion"() ( {
  %0 = tensor_load %input1 : memref<10xf32>
  %1 = tensor_load %input2 : memref<10xf32>
  %2 = "xla_hlo.add"(%0, %1) {name = "add"} : (tensor<10xf32>, tensor<10xf32>) -> tensor<10xf32>
  %3 = tensor_load %input3 : memref<10xf32>
  %4 = "xla_hlo.mul"(%2, %3) {name = "multiply"} : (tensor<10xf32>, tensor<10xf32>) -> tensor<10xf32>
  tensor_store %4, %out : memref<10xf32>
  "xla_lhlo.terminator"() : () -> ()
} ) : () -> ()
```



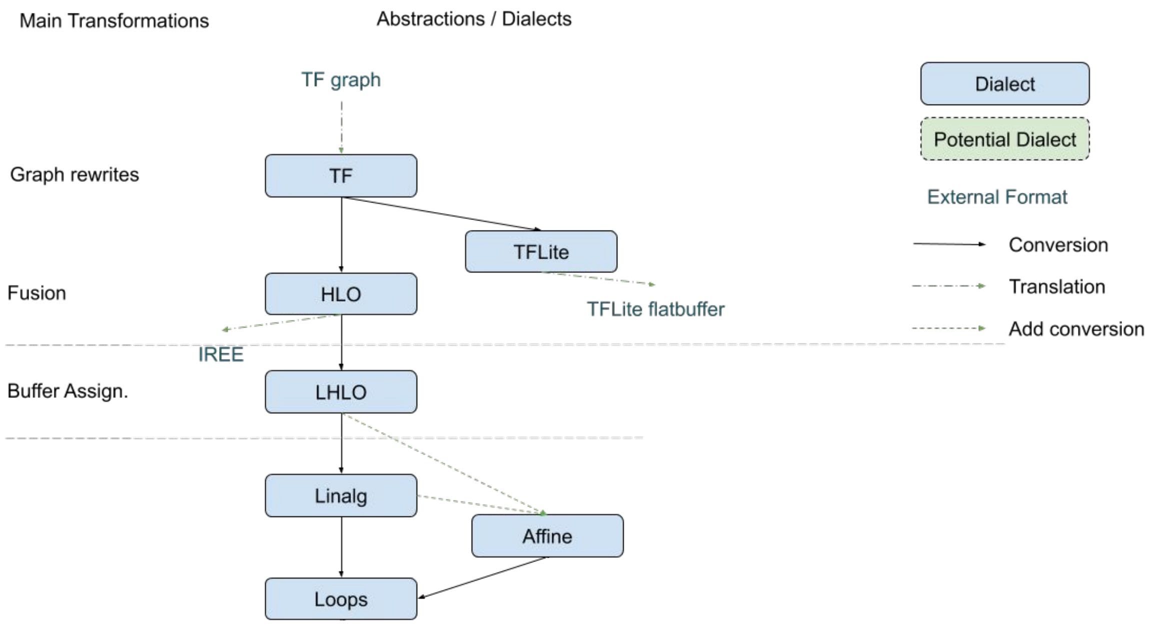

```
"xla_lhlo.mul"(%lhs, %rhs, %out) : (memref<10xf32>, memref<10xf32>, memref<10xf32>) -> ()
```

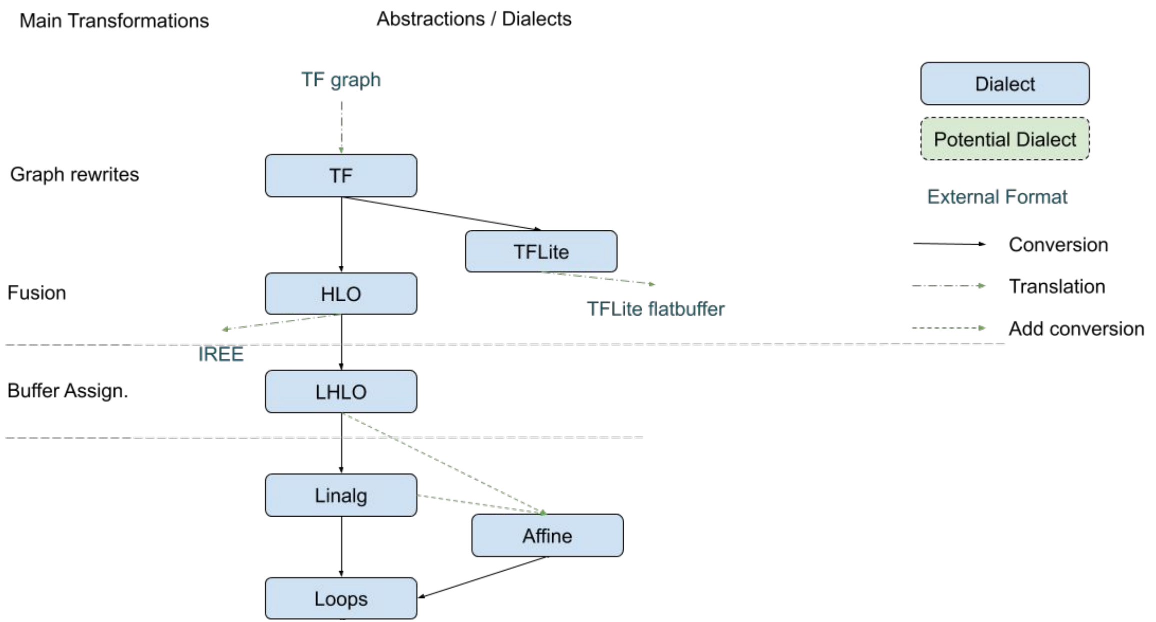

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion



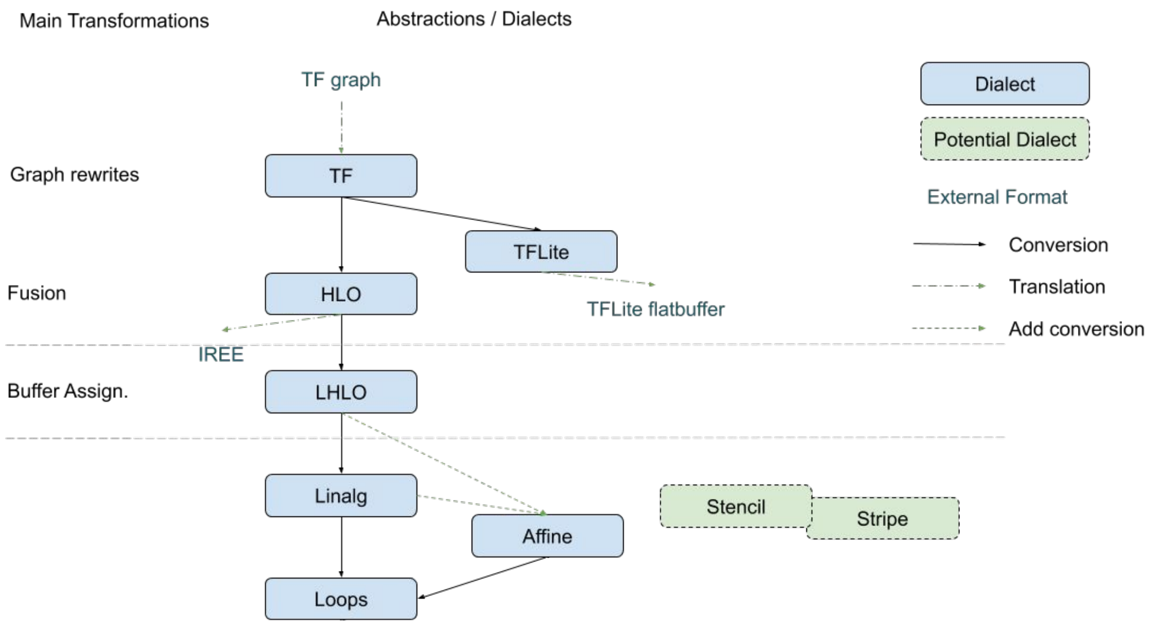

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  n_views = [2, 1],
  iterator_types = ["parallel", "parallel", "reduction"],
  indexing_maps = #matmul_accesses,
  library_call = "external_outerproduct_matmul"
}
```

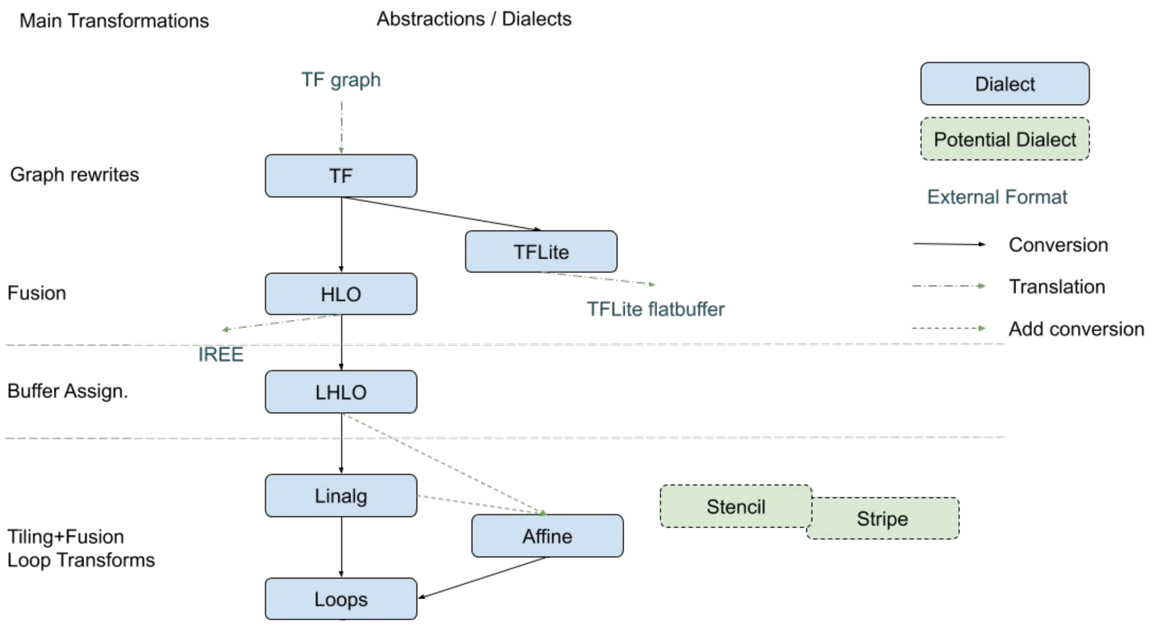
```
!vector_type_A = type vector<4xf32>
!vector_type_B = type vector<4xf32>
!vector_type_C = type vector<4x4xf32>

!matrix_type_A = type memref<?x?x!vector_type_A>
!matrix_type_B = type memref<?x?x!vector_type_B>
!matrix_type_C = type memref<?x?x!vector_type_C>

func @matmul_vec_impl(%A: !matrix_type_A, %B: !matrix_type_B, %C: !matrix_type_C) {
  linalg.generic #matmul_trait %A, %B, %C {
    ^bb0(%a: !vector_type_A, %b: !vector_type_B, %c: !vector_type_C):
      %d = vector.outerproduct %a, %b, %c: !vector_type_A, !vector_type_B
      linalg.yield %d: !vector_type_C
  } : !matrix_type_A, !matrix_type_B, !matrix_type_C

  return
}
```

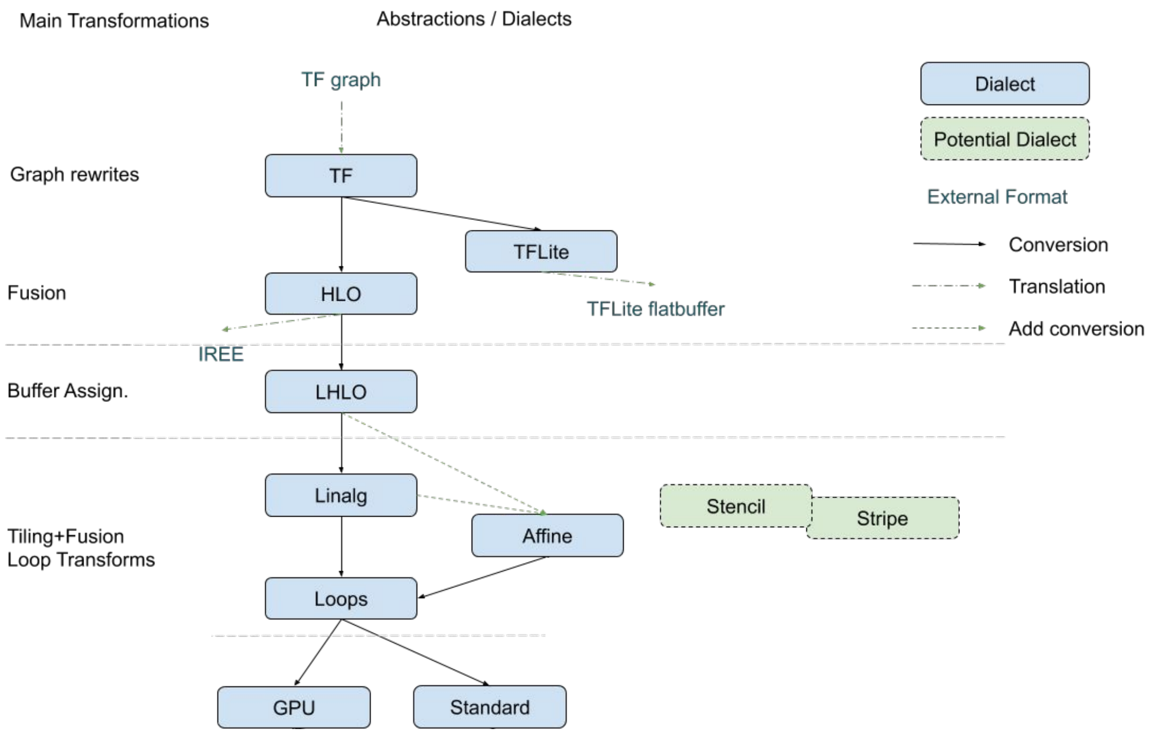

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Affine
Loops
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

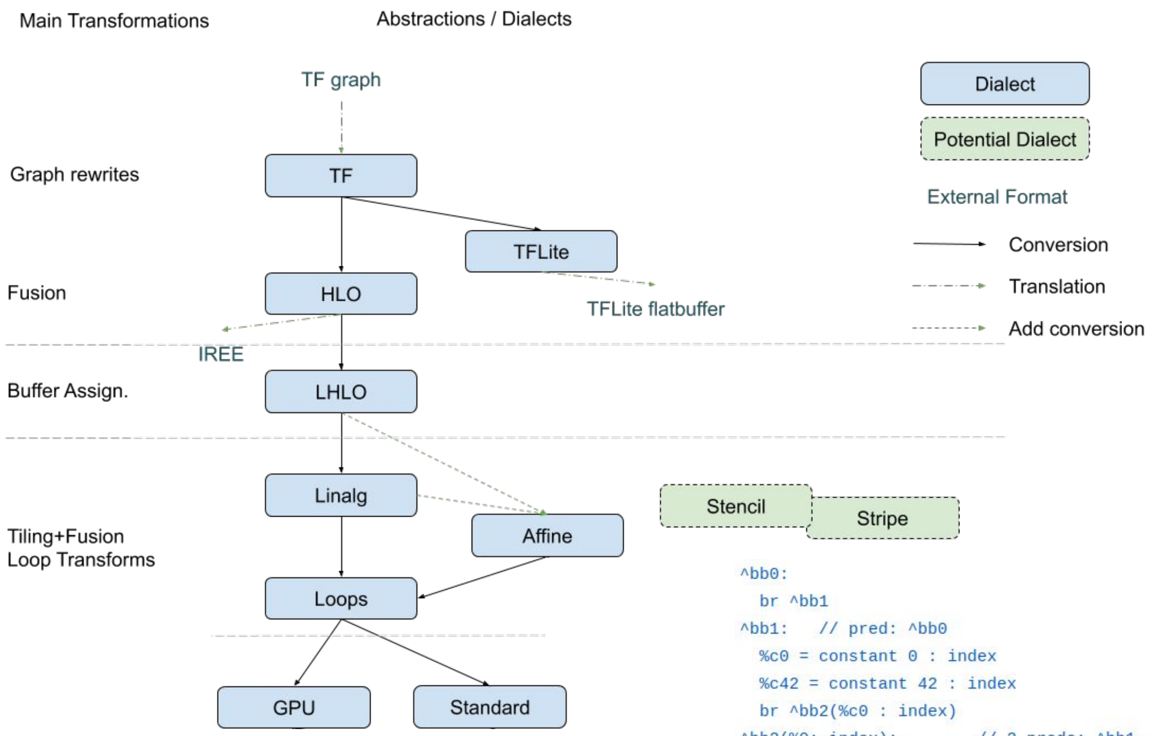



```
loop.for %i0 = %arg0 to %arg1 step %arg2 {
  loop.for %i1 = %arg0 to %arg1 step %arg2 {
    %min_cmp = cmpi "slt", %i0, %i1 : index
    %min = select %min_cmp, %i0, %i1 : index
    %max_cmp = cmpi "sge", %i0, %i1 : index
    %max = select %max_cmp, %i0, %i1 : index
    loop.for %i2 = %min to %max step %i1 {
    }
  }
```

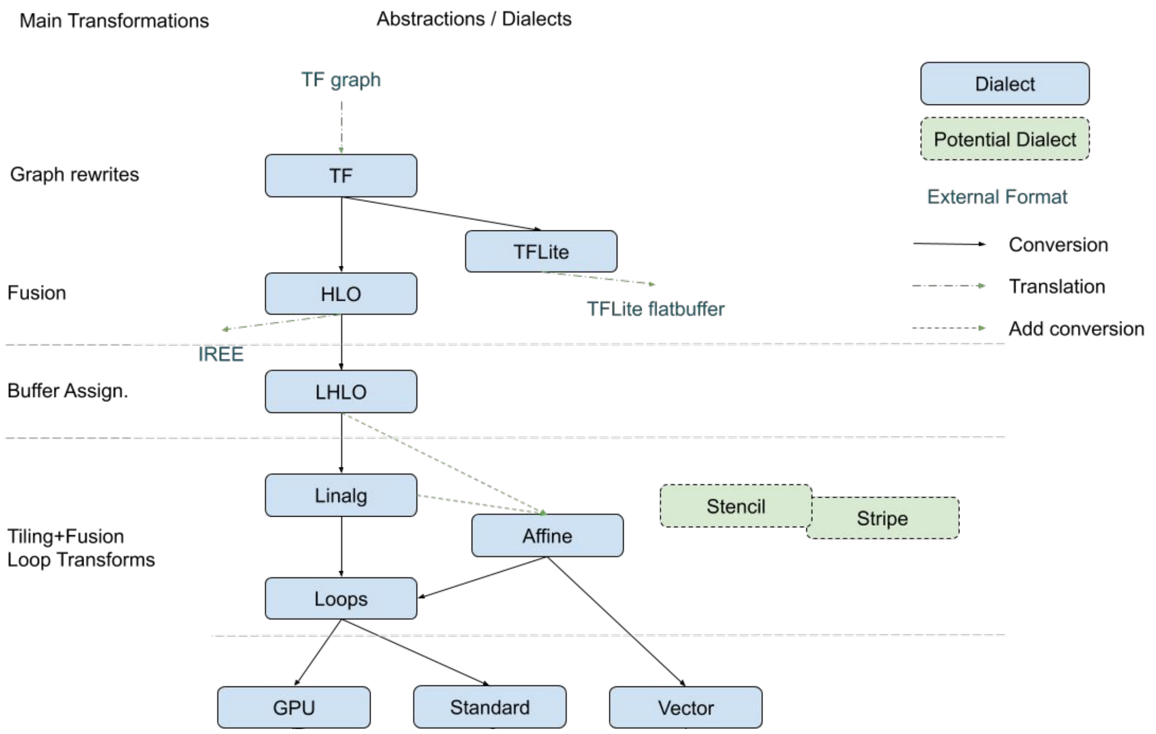

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Affine
Stencil
Stripe
Loops
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

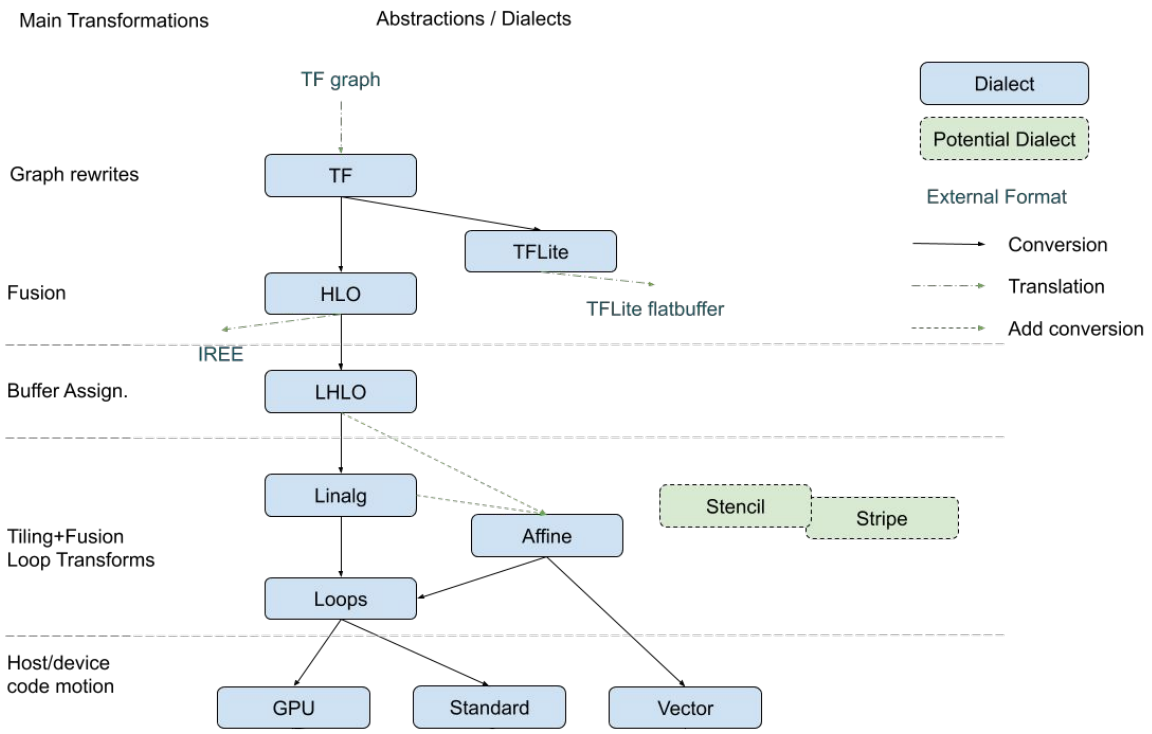

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Affine
Stencil
Stripe
Tiling+Fusion
Loop Transforms
Loops
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

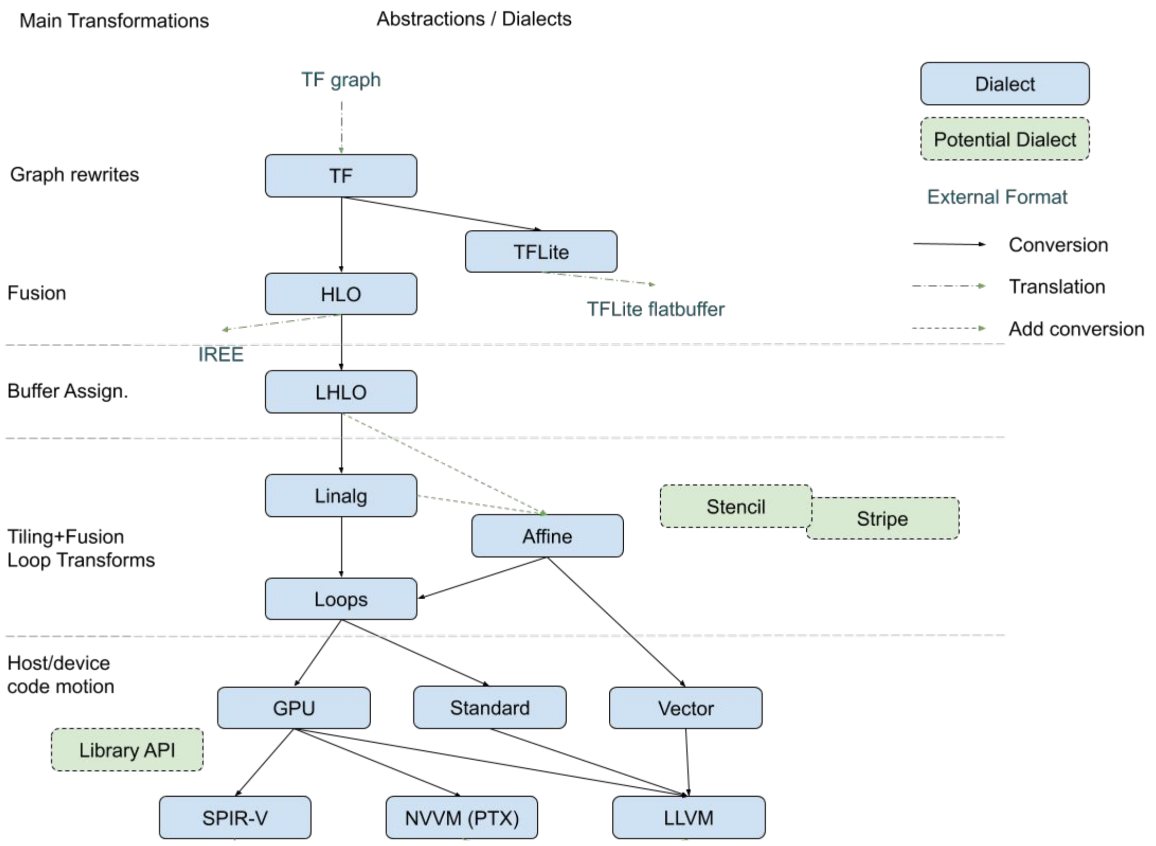

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Affine
Stencil
Stripe
Tiling+Fusion
Loop Transforms
Loops
GPU
Standard
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion



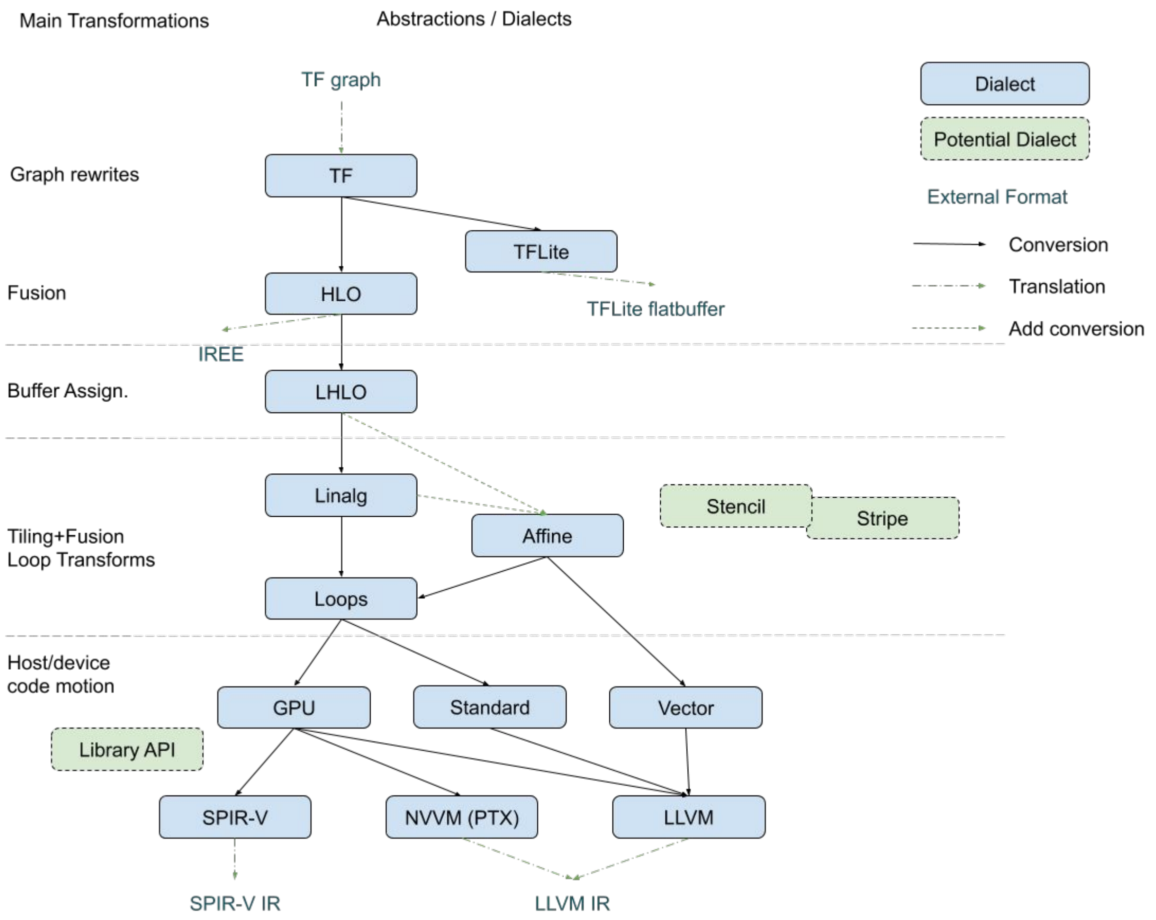

```
^bb0:
  br ^bb1
^bb1:   // pred: ^bb0
  %c0 = constant 0 : index
  %c42 = constant 42 : index
  br ^bb2(%c0 : index)
^bb2(%0: index):        // 2 preds: ^bb1, ^bb11
  %1 = cmpi "slt", %0, %c42 : index
  cond_br %1, ^bb3, ^bb12
^bb3:   // pred: ^bb2
  call @pre(%0) : (index) -> ()
  br ^bb4
^bb4:   // pred: ^bb3
  %c7 = constant 7 : index
  %c56 = constant 56 : index
  br ^bb5(%c7 : index)
```


Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Affine
Stencil
Stripe
Tiling+Fusion
Loop Transforms
Loops
GPU
Standard
Vector
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion


Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Affine
Stencil
Stripe
Tiling+Fusion
Loop Transforms
Loops
Host/device
code motion
GPU
Standard
Vector
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

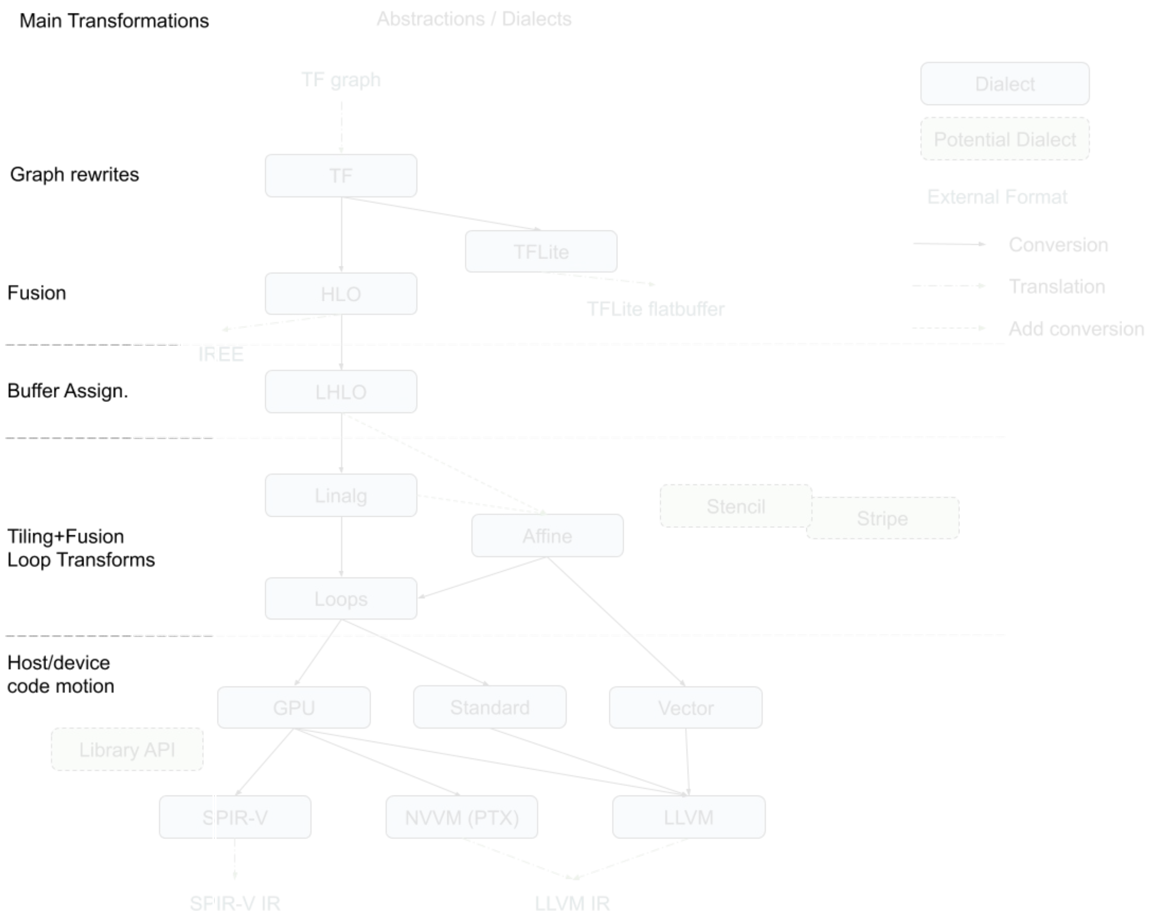

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Linalg
Affine
Stencil
Stripe
Tiling+Fusion
Loop Transforms
Loops
Host/device
code motion
GPU
Standard
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

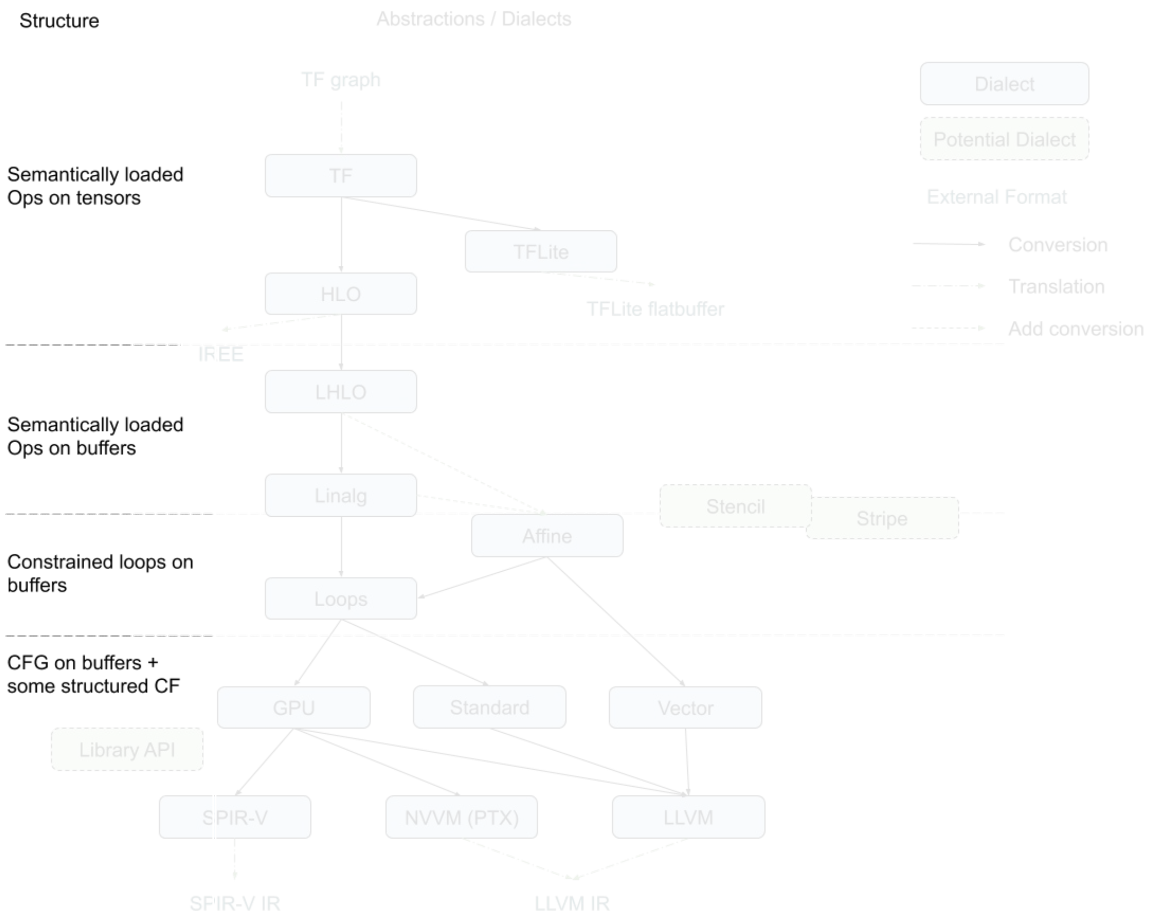
Google

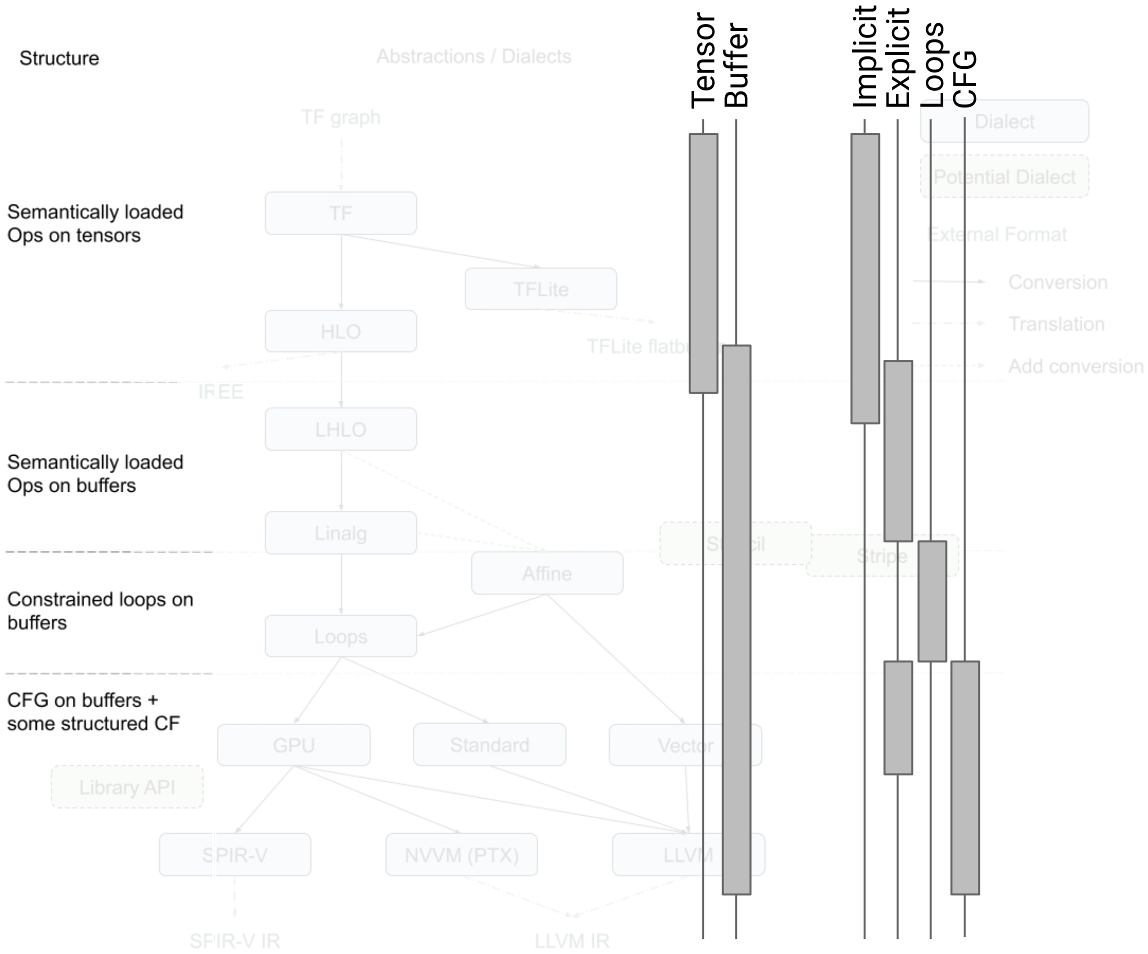

Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
TFLite
TFLite flatbuffer
Fusion
HLO
IREE
Buffer Assign.
LHLO
Tiling+Fusion
Loop Transforms
Linalg
Affine
Stencil
Stripe
Loops
Host/device
code motion
GPU
Standard
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
SPIR-V IR
LLVM IR
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

The lowering is not so much about transformations as it is about structure.

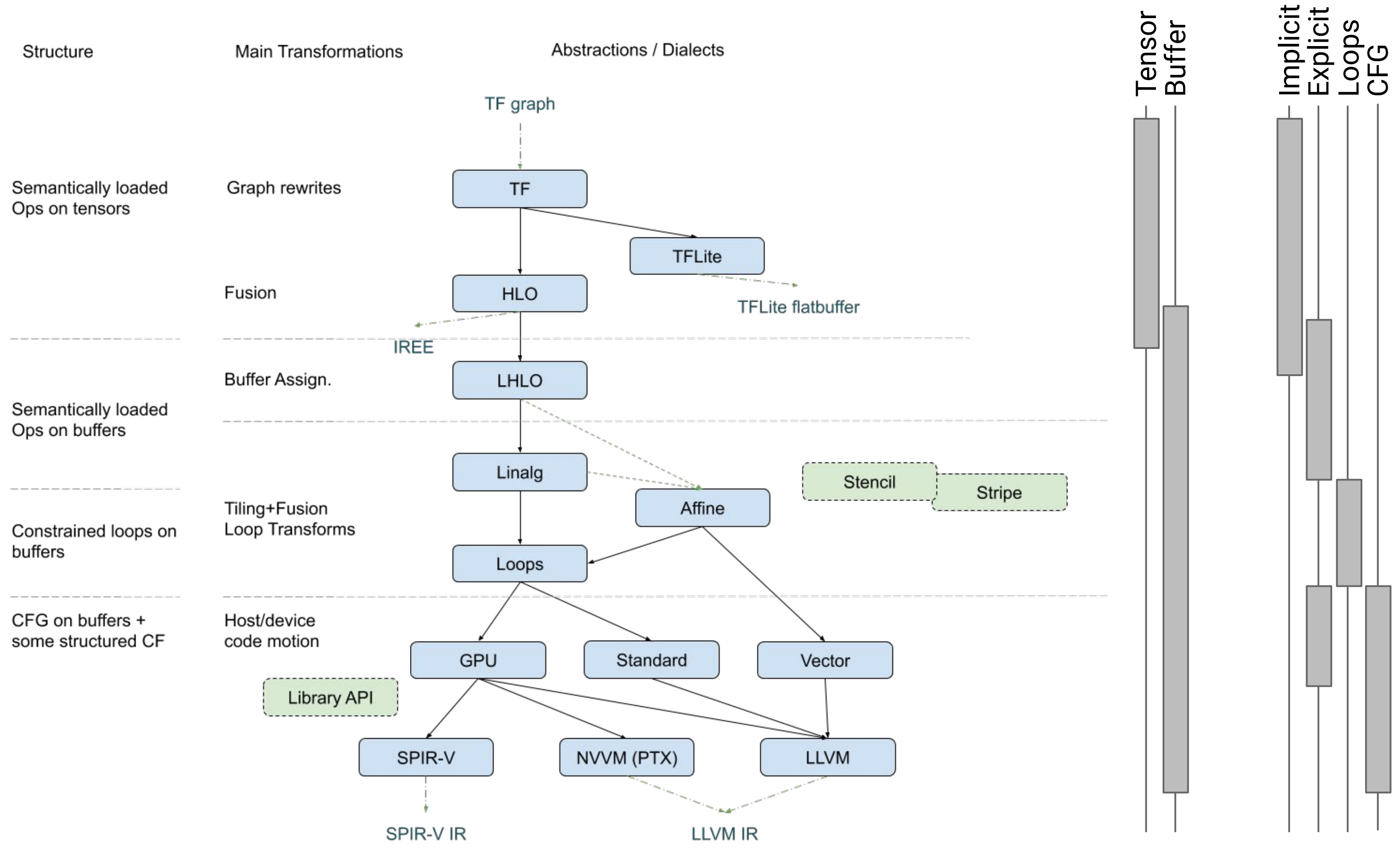
Or rather having the right structure at the right time.

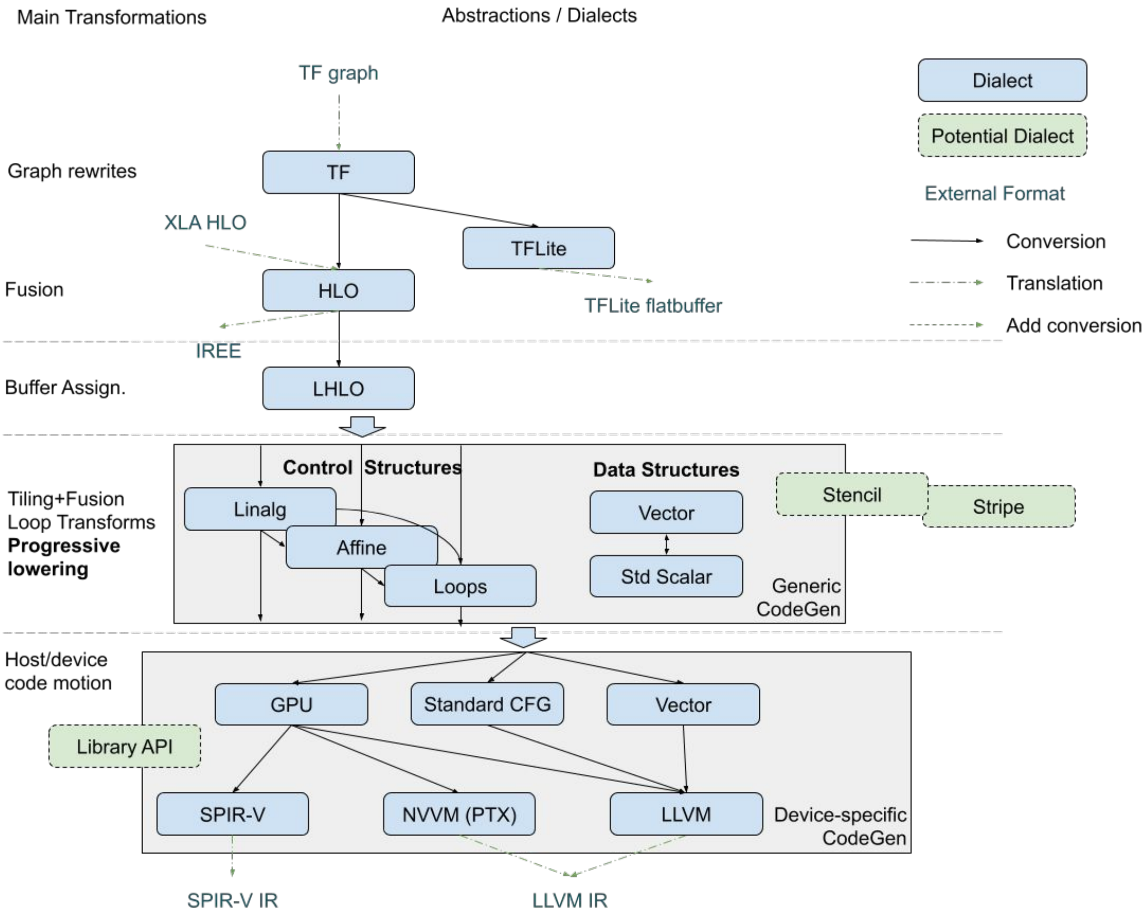

Main Transformations
Abstractions / Dialects
TF graph
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion
Graph rewrites
TF
TFLite
TFLite flatbuffer
Fusion
HLO
IREE
Buffer Assign.
LHLO
Linalg
Affine
Stencil
Stripe
Tiling+Fusion
Loop Transforms
Loops
Host/device
code motion
GPU
Standard
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
SPIR-V IR
LLVM IR


Structure
Abstractions / Dialects
TF graph
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion
Semantically loaded
Ops on tensors
TF
TFLite
HLO
TFLite flatbuffer
IREE
LHLO
Semantically loaded
Ops on buffers
Linalg
Stencil
Stripe
Affine
Constrained loops on
buffers
Loops
CFG on buffers +
some structured CF
GPU
Standard
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
SPIR-V IR
LLVM IR


Structure
Abstractions / Dialects
Semantically loaded Ops on tensors
Semantically loaded Ops on buffers
Constrained loops on buffers
CFG on buffers + some structured CF
TF graph
TF
TFLite
HLO
IREE
LHLO
Linalg
Affine
Loops
GPU
Standard
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
SPIR-V IR
LLVM IR
Stencil
Stripe
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion
Tensor
Buffer
Implicit
Explicit
Loops
CFG

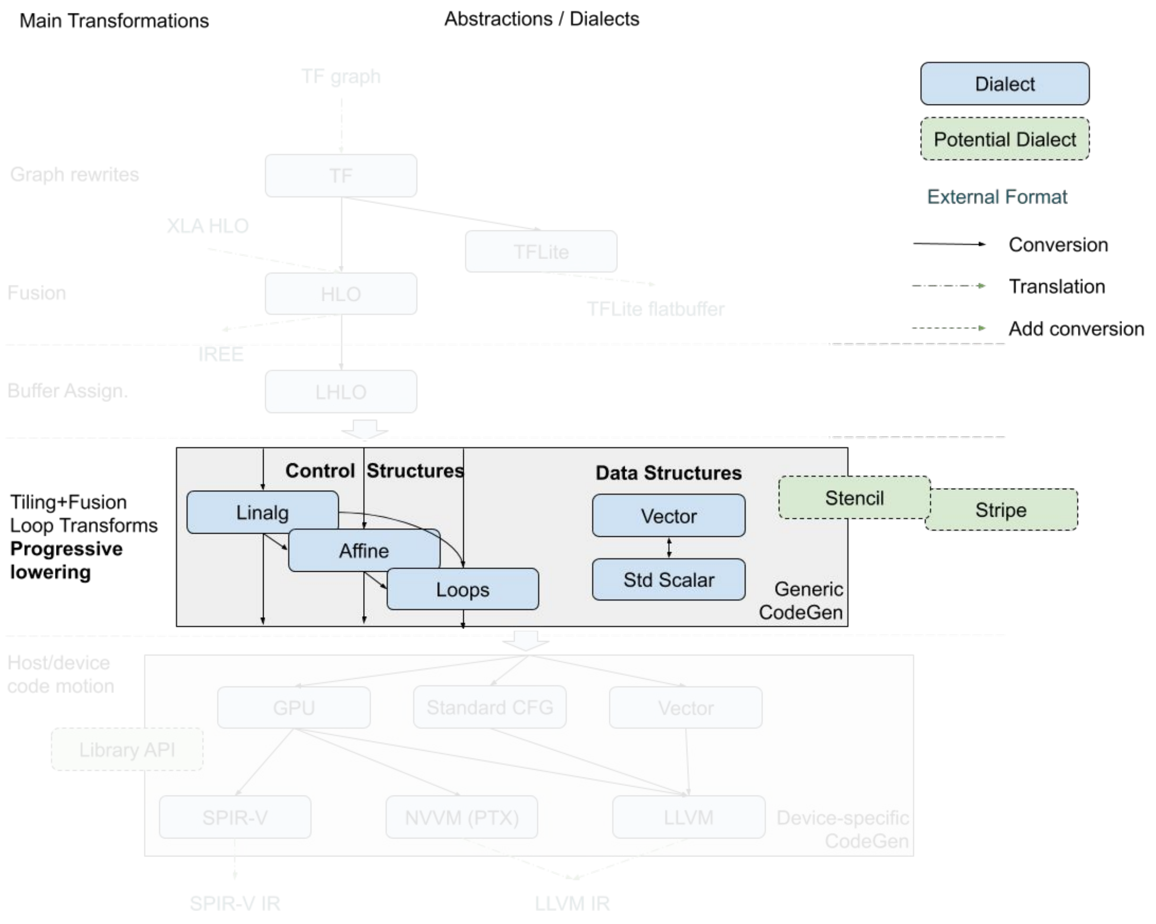
Google


Structure
Main Transformations
Abstractions / Dialects
TF graph
Semantically loaded
Ops on tensors
Graph rewrites
TF
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Semantically loaded
Ops on buffers
Linalg
Stencil
Stripe
Affine
Tiling+Fusion
Loop Transforms
Constrained loops on
buffers
Loops
CFG on buffers +
some structured CF
Host/device
code motion
GPU
Standard
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
SPIR-V IR
LLVM IR
Tensor
Buffer
Implicit
Explicit
Loops
CFG



Main Transformations
Abstractions / Dialects
TF graph
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion
Graph rewrites
TF
XLA HLO
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Control Structures
Data Structures
Tiling+Fusion
Loop Transforms
Progressive
lowering
Linalg
Affine
Loops
Vector
Std Scalar
Stencil
Stripe
Generic
CodeGen
Host/device
code motion
GPU
Standard CFG
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
Device-specific
CodeGen
SPIR-V IR
LLVM IR

Google



Main Transformations
Abstractions / Dialects
TF graph
Graph rewrites
TF
XLA HLO
TFLite
Fusion
HLO
TFLite flatbuffer
IREE
Buffer Assign.
LHLO
Tiling+Fusion
Loop Transforms
Progressive lowering
Control Structures
Data Structures
Linalg
Affine
Loops
Vector
Std Scalar
Generic CodeGen
Stencil
Stripe
Host/device code motion
GPU
Standard CFG
Vector
Library API
SPIR-V
NVVM (PTX)
LLVM
Device-specific CodeGen
SPIR-V IR
LLVM IR
Dialect
Potential Dialect
External Format
Conversion
Translation
Add conversion

Structured Ops Rationale

Philosophy: Transformations First

Transformations are the difference between

- a compiler (-O0)
- an optimizing compiler (-Ox)

Transformations are applied statically, they need some static information

Observation: Useful IR Entities That Bring Structure

Structured Types

- N-D vectors, memref, strided memref, memref with layout
- Dynamic evolution (ragged and sparse memref)
- Less regular but structured types: trees, concurrent hash maps, ...

Observation: Useful IR Entities That Bring Structure

Structured Types

- N-D vectors, memref, strided memref, memref with layout
- Dynamic evolution (ragged and sparse memref)
- Less regular but structured types: trees, concurrent hash maps, ...

Structured Iterators

- Defined by data, not just a bag of control-flow
- Properties that enable transformations (parallel, reduction, tilable etc)

Observation: Useful IR Entities That Bring Structure

Structured Types

- N-D vectors, memref, strided memref, memref with layout
- Dynamic evolution (ragged and sparse memref)
- Less regular but structured types: trees, concurrent hash maps, ...

Structured Iterators

- Defined by data, not just a bag of control-flow
- Properties that enable transformations (parallel, reduction, tilable etc)

Structured Ops **combine** 3 “things”

- Tie structured types and structured iterators in a coherent unit
- Properties to simplify analyses and transformations
- High-level correspondence with library calls

Structured Ops Abstraction - Buffers

Structured Ops At The Buffer Level

Encode key properties in the **type system**, consistent with DSL philosophy

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, reduction],
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

Structured Ops At The Buffer Level

Key property: Input and Output Buffer Operands + Types

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
for … {
  …
  for … {
    %a = "load" %A[..., ...]: memref<?x?xf32>
    %b = "load" %B[..., ...]: memref<?x?xf32>
    %c = "load" %C[..., ...]: memref<?x?xf32>
    %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element types
    "store" %cc, %C[..., ...]: memref<?x?xf32>
} … }
```

Google

Structured Ops At The Buffer Level

Key property: Input and Output Buffer Operands + Types

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
for … {
  …
  for … {
    %a = "load" %A[..., ...]: memref<?x?xf32>
    %b = "load" %B[..., ...]: memref<?x?xf32>
    %c = "load" %C[..., ...]: memref<?x?xf32>
    %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element types
    "store" %cc, %C[..., ...]: memref<?x?xf32>
} … }
```

Google

Structured Ops At The Buffer Level

Key property: Input and Output Buffer Operands + Types

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
for … {
  …
    for … {
      %a = "load" %A[..., ...]: memref<?x?xf32>
      %b = "load" %B[..., ...]: memref<?x?xf32>
      %c = "load" %C[..., ...]: memref<?x?xf32>
      %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element types
      "store" %cc, %C[..., ...]: memref<?x?xf32>
} … }
```

Google

Structured Ops At The Buffer Level

Key property: Input and Output Buffer Operands + Types

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
for … {
  …
  for … {
    %a = "load" %A[..., ...]: memref<?x?xf32>
    %b = "load" %B[..., ...]: memref<?x?xf32>
    %c = "load" %C[..., ...]: memref<?x?xf32>
    %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element types
    "store" %cc, %C[..., ...]: memref<?x?xf32>
} … }
```

Google

Structured Ops At The Buffer Level

Key property: Domain traversal (Reversible mappings from iterations to data type)

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
%M = "dim" %A, 0: index
%K = "dim" %A, q: index
%KK = "dim" %B, 0: index
%N = "dim" %B, 1: index
%MM = "dim" %C, 0: index
%NN = "dim" %C, 1: index
%eq = "eq" %M, %MM: i1   // inferred iteration space is consistent with data
"assert" %eq: (i1) -> ()
            ...
for %m = 0 to %M {
  for %n = 0 to %N {
    for %k = 0 to %K {
      %a = "load" %A[%m, %k]: memref<?x?xf32>
      %b = "load" %B[%k, %n]: memref<?x?xf32>
      %c = "load" %C[%m, %n]: memref<?x?xf32>
      %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element type
      "store" %cc, %C[%m, %n]: memref<?x?xf32>
}}}
```

Google

Structured Ops At The Buffer Level

Key property: Domain traversal (Reversible mappings from iterations to data type)

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
%M = "dim" %A, 0: index
%K = "dim" %A, q: index
%KK = "dim" %B, 0: index
%N = "dim" %B, 1: index
%MM = "dim" %C, 0: index
%NN = "dim" %C, 1: index
%eq = "eq" %M, %MM: i1   // inferred iteration space is consistent with data
"assert" %eq: (i1) -> ()
        ...
for %m = 0 to %M {
  for %n = 0 to %N {
    for %k = 0 to %K {
      %a = "load" %A[%m, %k]: memref<?x?xf32>
      %b = "load" %B[%k, %n]: memref<?x?xf32>
      %c = "load" %C[%m, %n]: memref<?x?xf32>
      %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element type
      "store" %cc, %C[%m, %n]: memref<?x?xf32>
}}}
```

Structured Ops At The Buffer Level

Key property: Domain traversal (Reversible mappings from iterations to data type)

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
%M = "dim" %A, 0: index
%K = "dim" %A, q: index
%KK = "dim" %B, 0: index
%N = "dim" %B, 1: index
%MM = "dim" %C, 0: index
%NN = "dim" %C, 1: index
%eq = "eq" %M, %MM: i1    // inferred iteration space is consistent with data
"assert" %eq: (i1) -> ()
            ...
for %m = 0 to %M {
  for %n = 0 to %N {
    for %k = 0 to %K {
      %a = "load" %A[%m, %k]: memref<?x?xf32>
      %b = "load" %B[%k, %n]: memref<?x?xf32>
      %c = "load" %C[%m, %n]: memref<?x?xf32>
      %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element type
      "store" %cc, %C[%m, %n]: memref<?x?xf32>
}}}
```

Structured Ops At The Buffer Level

Key property: Custom computation with region (or MLIR function)

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
...
for %m = 0 to %M {
  for %n = 0 to %N {
    for %k = 0 to %K {
      %a = "load" %A[%m, %k]: memref<?x?xf32>
      %b = "load" %B[%k, %n]: memref<?x?xf32>
      %c = "load" %C[%m, %n]: memref<?x?xf32>
      %d = mulf %a, %b: f32                          // region body inlined
      %e = addf %d, %c: f32
      "store" %e, %C[%m, %n]: memref<?x?xf32>
}}}
```

Google

Structured Ops At The Buffer Level

Key property: Implicit iterator types (future: explicit mapping to virtual processor id)

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, reduction],
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
// iterator_types used for transformations,
// carry semantic information about dependences
```

Structured Ops At The Buffer Level

Key property: May correspond to an external library call

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, re
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
func @matmul_f32 (memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>) -> ()
call @matmul_f32 (%A, %B, %C): (memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>) -> ()

func @matmul_f32 (!llvm<"{ float*, i64, [2 x i64], [3 x i64] }*">,
                  !llvm<"{ float*, i64, [2 x i64], [3 x i64] }*">,
                  !llvm<"{ float*, i64, [2 x i64], [3 x i64] }*">) -> ()
llvm.call @matmul_f32 (%A, %B, %C): (!llvm<"{ float*, i64, [2 x i64], [3 x i64] }*">...) -> ()
```

Structured Ops At The Buffer Level

For more properties see the backup slides.

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 1, args_out = 1,
  iterator_types = [parallel, parallel, r
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

```
%M = "dim" %A, 0: index
%K = "dim" %A, q: index
%KK = "dim" %B, 0: index
%N = "dim" %B, 1: index
%MM = "dim" %C, 0: index
%NN = "dim" %C, 1: index
%eq = "eq" %M, %MM: i1   // inferred iteration space is consistent with data
"assert" %eq: (i1) -> ()
            ...
for %m = 0 to %M {
  for %n = 0 to %N {
    for %k = 0 to %K {
      %a = "load" %A[%m, %k]: memref<?x?xf32>
      %b = "load" %B[%k, %n]: memref<?x?xf32>
      %c = "load" %C[%m, %n]: memref<?x?xf32>
      %cc = "compute"(%a, %b, %c): (f32, f32, f32) -> (f32) // compute arg types match element type
      "store" %cc, %C[%m, %n]: memref<?x?xf32>
}}}
```

Recap: Structured Ops At The Buffer Level

Encode key properties in the type system, consistent with DSL philosophy

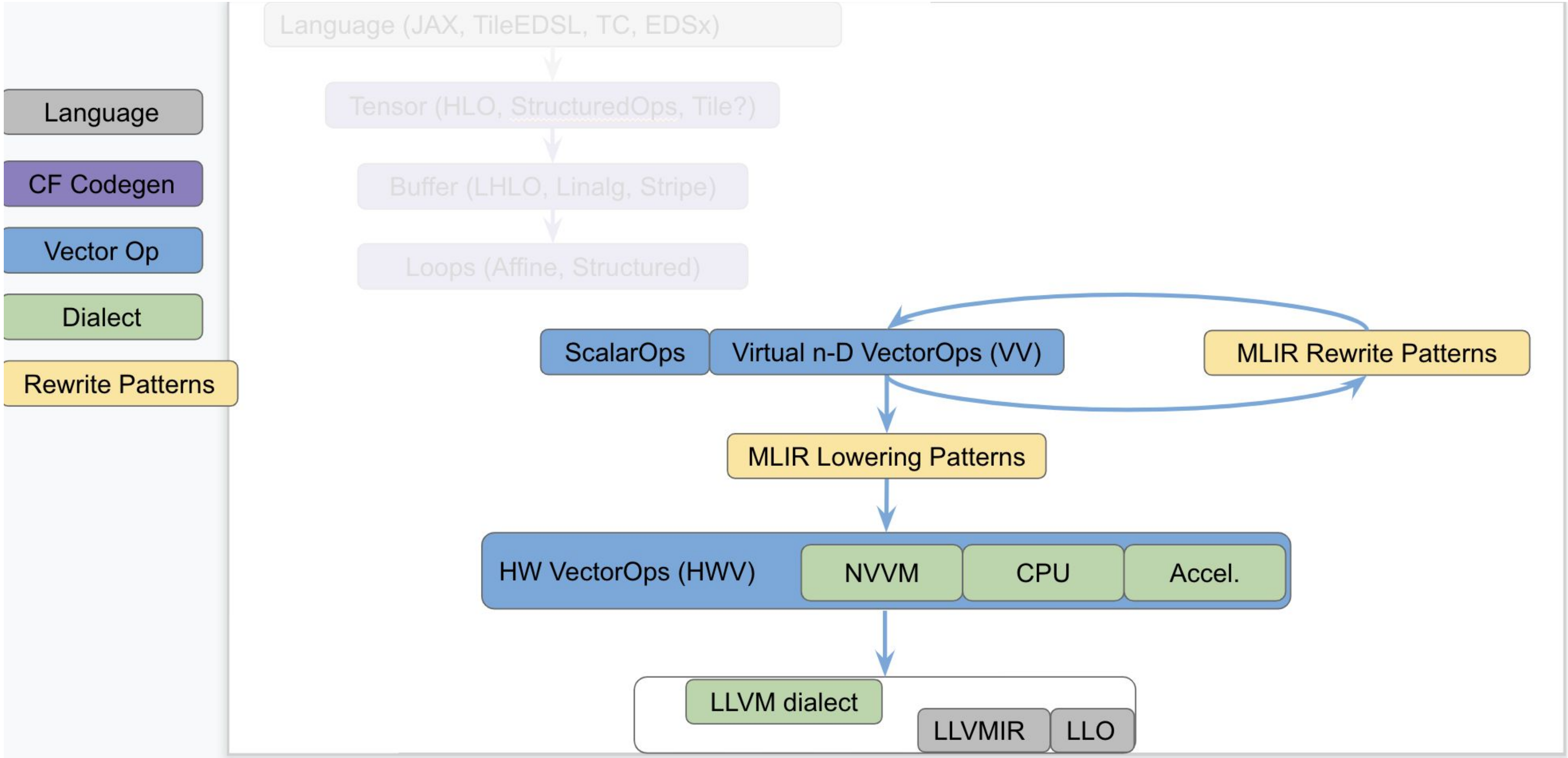
```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  args_in = 2, args_out = 1,
  iterator_types = [parallel, parallel, reduction],
  indexing = #matmul_accesses,
  library_call = "matmul_f32"
}
"s.op" #matmul_trait (%A, %B, %C) {
  ^bb0(%a: f32, %b: f32, %c: f32):
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    "s.yield" %e: f32
} : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

Automatically Sugar Into (NYI)

```
s.matmul (%A, %B, %C) : memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>
```

Structured Ops Abstraction - Vectors

Positioning of Vector Dialects



Structured Ops At The Vector Level

Encode key properties in the **type system**, consistent with DSL philosophy

```
#contraction_accesses = [
  (b0, f0, f1, c0, c1) -> (c0, b0, c1, f0),
  (b0, f0, f1, c0, c1) -> (b0, c1, c0, f1),
  (b0, f0, f1, c0, c1) -> (b0, f0, f1)
]
#contraction_trait = {
   indexing_maps = #contraction_accesses,
   iterator_types = [parallel, parallel, parallel, reduction, reduction]
}
```

```
%0 = vector.contract #contraction_trait %lhs, %rhs, %acc
      : vector<7x8x16x15xf32>, vector<8x16x7x5xf32> into vector<8x15x5xf32>
```

Structured Ops At The Vector Level

Encode key properties in the **type system**, consistent with DSL philosophy

```
#contraction_accesses = [
 (b0, f0, f1, c0, c1) -> (c0, b0, c1, f0),
 (b0, f0, f1, c0, c1) -> (b0, c1, c0, f1),
 (b0, f0, f1, c0, c1) -> (b0, f0, f1)
]
#contraction_trait = {
  indexing_maps = #contraction_accesses,
  iterator_types = [parallel, parallel, parallel, reduction, reduction]
}

%0 = vector.contract #contraction_trait0 %lhs, %rhs, %acc
      : vector<7x8x16x15xf32>, vector<8x16x7x5xf32> into vector<8x15x5xf32>
```

Structure

- Reduction dimensions known
  - enables selective unrolling
- Reduction dimensions separate (e.g. conv spatial dimensions)
  - avoids reshape

Structured Ops At The Vector Level

Encode key properties in the **type system**, consistent with DSL philosophy

```
#contraction_accesses = [
  (b0, f0, f1, c0, c1) -> (c0, b0, c1, f0),
  (b0, f0, f1, c0, c1) -> (b0, c1, c0, f1),
  (b0, f0, f1, c0, c1) -> (b0, f0, f1)
]
#contraction_trait = {
  indexing_maps = #c
  iterator_types = [parallel, parallel, parallel, reduction, reduction]
}

%0 = vector.contract #contraction_trait0 %lhs, %rhs, %acc
      : vector<7x8x16x15xf32>, vector<8x16x7x5xf32> into vector<8x15x5xf32>
```

Structure

- Reduction dimensions known
  - enables selective unrolling
- Reduction dimensions separate (e.g. conv spatial dimensions)
  - avoids reshape
- Operates on large vector aggregates (e.g. multiples of HW vector size).
- Aggregates multiple reduction dimension iterations
  - enables unroll and jam, unroll and pack

Declarative Transformations with Composable Patterns

Google

Structured Ops Transformations

Structured Ops allow the following composable transformations:

- Parametric Tiling
- Promotion to Temporary Buffer
- Producer-Consumer Fusion
- Mapping to Parallel HW
- Rewrite as Loop around finer-grained structured op
- Rewrite in Vector Form
- Call an HPC library (e.g. cuDNN, cuB, cuTLASS, MKL-DNN, BLIS, swizzle inventor)
- Lower to Affine
- Lower to LLVM
- Unroll to smaller vector form
- Vector rewrite patterns

How Are Structured Ops Transformations Composable?

Structured Ops provide a natural anchor point for pattern matching and rewriting

- Composes with transformations
  - tiling produces loops + structured ops
  - mapping produces loops + structured ops
  - fusion produces imperfectly nested loops + structured ops
- Structure
  - no need for complex matching rules on non-local pieces of IR

Until we **decide** to lower out of structured ops

⟶ Expose composable building blocks to define an optimization strategy: semi-automatic “schedule” like Halide, or fully automatic heuristic

Declarative Transformations (1) Multi-level Tiling

Match `s.matmul` (any type)

Capture op and arguments by name

Filter ops not marked as "L3"

```
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[2000, 3000, 4000], "L3"> $op), [(Constraint<Or<[HasNoMarker, HasMarker<"MEM">]>> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[200, 300, 400], "L2"> $op), [(Constraint<HasMarker<"L3">> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[20, 30, 40], "L1"> $op), [(Constraint<HasMarker<"L2">> $op)]>;
```

Tile by “sizes” and mark as tiled for "L1"

Transformations: Declarative Multi-level Tiling

```
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[2000, 3000, 4000], "L3"> $op), [(Constraint<Or<[HasNoMarker, HasMarker<"MEM">]>> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[200, 300, 400], "L2"> $op), [(Constraint<HasMarker<"L3">> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[20, 30, 40], "L1"> $op), [(Constraint<HasMarker<"L2">> $op)]>;
```

```
s.matmul (%A, %B, %C) : memref<?x?xf32, offset: ?, strides: [?, 1]> ...
```

Apply Tile<[2000, 3000, 4000], "L3">

Transformations (1) Declarative Multi-level Tiling

```
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[2000, 3000, 4000], "L3"> $op), [(Constraint<Or<[HasNoMarker, HasMarker<"MEM">]>> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[200, 300, 400], "L2"> $op), [(Constraint<HasMarker<"L3">> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[20, 30, 40], "L1"> $op), [(Constraint<HasMarker<"L2">> $op)]>;
```

Apply Tile<[2000, 3000, 4000], "L3">

```
...
for %i = %c0 to %6 step %c2000 {
  for %k = %c0_0 to %7 step %c3000 {
    for %k = %c0_1 to %8 step %c4000 {
      %9 = affine.apply (d0) -> (d0 + 2000)(%i)
      ...
      %16 = s.subview %A[%12, %13, %c1, %14, %15, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
      ...
      %21 = s.subview %B[%17, %18, %c1, %19, %20, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
      ...
      %26 = s.subview %C[%22, %23, %c1, %24, %25, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
      s.matmul (%16, %21, %26) {__internal__ = "L3"} : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>,  ...
```

Apply Tile<[200, 300, 400], "L2">

Transformations (1) Declarative Multi-level Tiling

```
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[2000, 3000, 4000], "L3"> $op), [(Constraint<Or<[HasNoMarker, HasMarker<"MEM">]>> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[200, 300, 400], "L2"> $op), [(Constraint<HasMarker<"L3">> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[20, 30, 40], "L1"> $op), [(Constraint<HasMarker<"L2">> $op)]>;
```

Apply Tile<[200, 300, 400], "L2">

```
...
for %i = %c0 to %0 step %c2000 {
  for %j = %c0 to %2 step %c3000 {
    for %k = %c0 to %1 step %c4000 {
      %3 = affine.apply (d0) -> (d0)(%i)
      ...
      %7 = s.subview %A[%3, %4, %c1, %5, %6, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
      ...
      %12 = s.subview %B[%8, %9, %c1, %10, %11, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
      ...
      %17 = s.subview %C[%13, %14, %c1, %15, %16, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
      ...
      for %ii = %c0_0 to %24 step %c200 {
        for %jj = %c0_1 to %25 step %c300 {
          for %kk = %c0_2 to %26 step %c400 {
            %27 = affine.apply (d0) -> (d0 + 200)(%ii)
            ...
            %34 = s.subview %7[%30, %31, %c1, %32, %33, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
            ...
            %39 = s.subview %12[%35, %36, %c1, %37, %38, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
            ...
            %44 = s.subview %17[%40, %41, %c1, %42, %43, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
            s.matmul (%34, %39, %44) {__internal_linalg_transform__ = "L2"} : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)> ...
```

Apply Tile<[20, 30, 40], "L1">

Google

Transformations (1) Declarative Multi-level Tiling

```
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[2000, 3000, 4000], "L3"> $op), [(Constraint<Or<[HasNoMarker, HasMarker<"MEM">]>> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[200, 300, 400], "L2"> $op), [(Constraint<HasMarker<"L3">> $op)]>;
def : Pat<(MatmulOp:$op $A, $B, $C), (Tile<[20, 30, 40], "L1"> $op), [(Constraint<HasMarker<"L2">> $op)]>;
```

Apply Tile<[20, 30, 40], "L1">

```
...
for %i = %c0 to %0 step %c2000 {
  for %j = %c0 to %2 step %c3000 {
    for %k = %c0 to %1 step %c4000 {
      %3 = affine.apply (d0) -> (d0 + 2000)(%i)
      ...
      %11 = s.subview %C[%D, %9, %c1, %E, %10, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
      ...
      for %arg6 = %c0 to %12 step %c200 {
        for %arg7 = %c0 to %14 step %c300 {
          for %arg8 = %c0 to %13 step %c400 {
            %15 = affine.apply (d0) -> (d0)(%arg6)
            ...
            %29 = s.subview %11[%25, %26, %c1, %27, %28, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
            ...
            for %arg9 = %c0_0 to %36 step %c20 {
              for %B0 = %c0_1 to %37 step %c30 {
                for %B1 = %c0_2 to %38 step %c40 {
                  %39 = affine.apply (d0) -> (d0 + 20)(%arg9)
                  ...
                  %46 = s.subview %19[%42, %43, %c1, %44, %45, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
                  ...
                  %51 = s.subview %24[%47, %48, %c1, %49, %50, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
                  ...
                  %56 = s.subview %29[%52, %53, %c1, %54, %55, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
                  s.matmul (%46, %51, %56) {__internal__ = "L1"} : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>, ...
```

Transformations (2) Declarative Producer-Consumer Fusion

Match `s.matmul` (any type)

Capture op and arguments by name

```
def : Pat<(MatmulOp:$consumer $A, $B, $C), (TileAndFuse<[100, 150], "L2"> $consumer),
        [ (Constraint<HasNoLinalgTransformMarker> $consumer), (Constraint<IsProducedByOpOfType<"MatmulOp">> $consumer, $A)],
        (addBenefit 1)>;
```

Explicitly give higher priority

Filter ops whose first argument is not produced by a `s.matmul` (any type)

Google

Transformations (2) Declarative Producer-Consumer Fusion

```
def : Pat<(MatmulOp:$consumer $A, $B, $C), (TileAndFuse<[100, 150], "L2"> $consumer),
      [ (Constraint<HasNoLinalgTransformMarker> $consumer), (Constraint<IsProducedByOpOfType<"MatmulOp">> $consumer, $A)],
      (addBenefit 1)>;
```

```
// This will not be fused as it would violate dependencies. It will just get tiled for all levels of the memory hierarchy.
s.matmul (%A, %A, %C) : memref<?x?xf32, offset: ?, strides: [?, 1]>, ...

// This will be fused into the last op and bypass s.generic.
s.matmul (%A, %B, %C) : memref<?x?xf32, offset: ?, strides: [?, 1]>, ...

// This will not be fused or transformed at all since there are no patterns on it. However it will be reordered because there are no dependencies.
s.generic #some_generic_trait %A, %D {
  ^bb(%a: f32, %b: f32) :
    "s.yield" %a : f32
} : memref<?x?xf32, offset: ?, strides: [?, 1]>, memref<?x?xf32, offset: ?, strides: [?, 1]>

// This will be fused into and then will continue tiling.
s.matmul (%C, %D, %E) : memref<?x?xf32, offset: ?, strides: [?, 1]>, ...
```

Apply TileAndFuse<[100, 150], "L2">

Transformations (2) Declarative Producer-Consumer Fusion

```
def : Pat<(MatmulOp:$consumer $A, $B, $C), (TileAndFuse<[100, 150], "L2"> $consumer),
    [ (Constraint<HasNoLinalgTransformMarker> $consumer), (Constraint<IsProducedByOpOfType<"MatmulOp">> $consumer, $A)],
    (addBenefit 1)>;
```

```
...
s.matmul (%A, %A, %C) : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>, …

s.generic #some_generic_trait %A, %D {
^bb0(%arg5: f32, %arg6: f32): // no predecessors
  "s.yield" %arg5 : f32
}: memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>, memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
…

for %arg5 = %c0_0 to %6 step %c100 {
  for %arg6 = %c0_1 to %7 step %c150 {
    %9 = affine.apply (d0) -> (d0 + 100)(%arg5)
    ...
    %14 = s.subview %C[%11, %12, %c1, %c0_3, %13, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
    ...
    %18 = s.subview %D[%c0_5, %15, %c1, %16, %17, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
    ...
    %23 = s.subview %E[%19, %20, %c1, %21, %22, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
    ...
    %25 = s.subview %A[%11, %12, %c1, %c0_10, %24, %c11] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
    %26 = s.subview %B[%c0_10, %24, %c11, %c0_3, %13, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
    %27 = s.subview %C[%11, %12, %c1, %c0_3, %13, %c1] : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>
    s.matmul (%25, %26, %27) {__internal__ = "L2"} : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>, ...
    s.matmul (%14, %18, %23) {__internal__ = "L2"} : memref<?x?xf32, (d0, d1)[s0, s1] -> (d0 * s1 + s0 + d1)>, ...
```

Apply whichever of Tile<[2000, 3000, 4000], "L3">, Tile<[200, 300, 400], "L2"> or Tile<[20, 30, 40], "L1"> comes first, iteratively

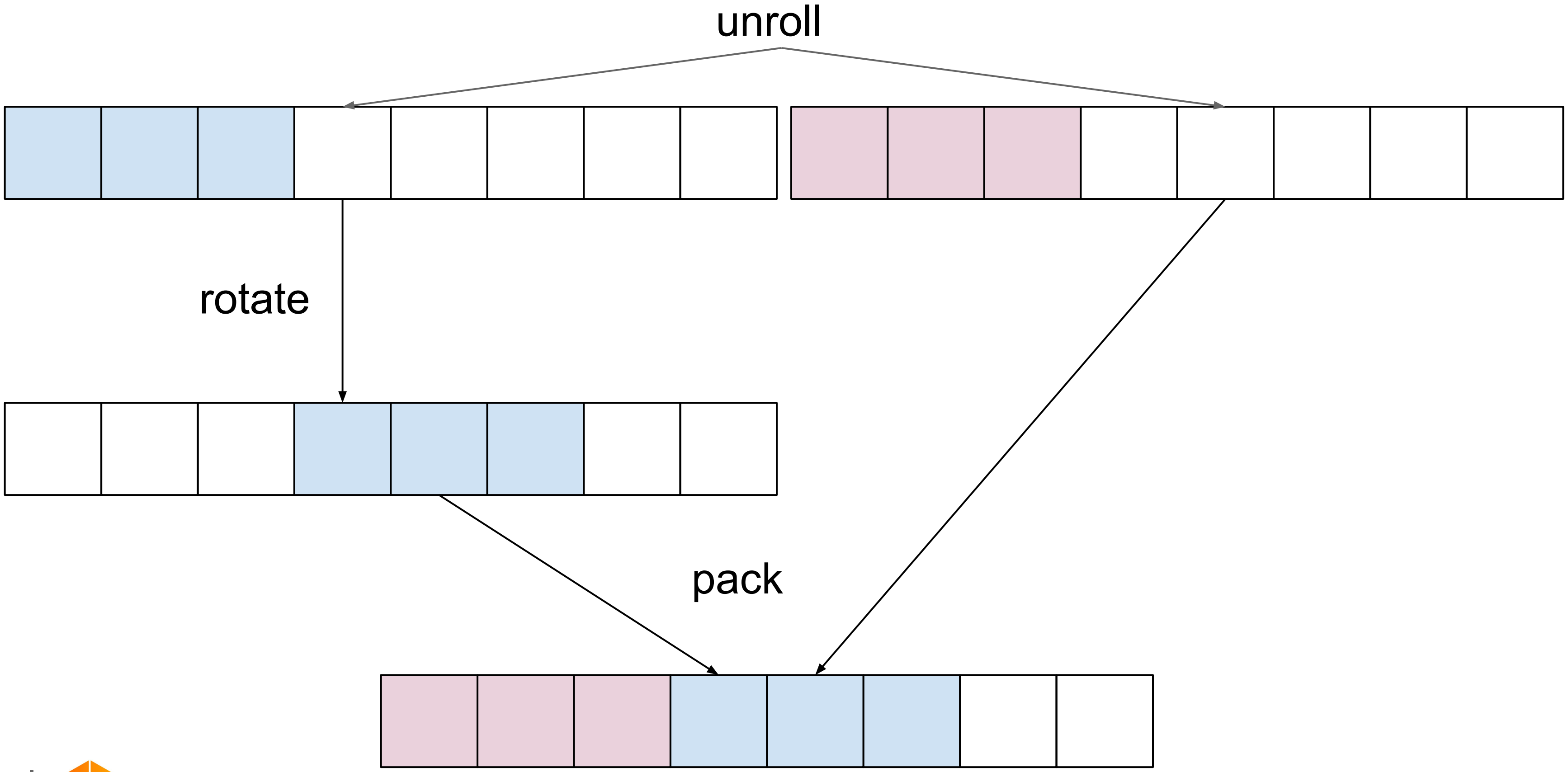
Google

Transformations (2) Declarative Producer-Consumer Fusion

```
def : Pat<(MatmulOp:$consumer $A, $B, $C), (TileAndFuse<[100, 150], "L2"> $consumer),
      [ (Constraint<HasNoLinalgTransformMarker> $consumer), (Constraint<IsProducedByOpOfType<"MatmulOp">> $consumer, $A)],
      (addBenefit 1)>;
```

```
...
for %arg5 = %c0 to %0 step %c2000 {
 for %arg6 = %c0 to %2 step %c3000 {
  for %arg7 = %c0 to %1 step %c4000 {
   ...
   %13 = s.subview %arg2[%arg5, %11, %c1, %arg6, %12, %c1] : memref<?x?xf32, #map3>
   for %arg8 = %c0 to %14 step %c200 {
    for %arg9 = %c0 to %16 step %c300 {
     for %arg10 = %c0 to %15 step %c400 {
      ...
      %25 = s.subview %13[%arg8, %23, %c1, %arg9, %24, %c1] : memref<?x?xf32, #map3>
      for %arg11 = %c0 to %26 step %c20 {
       for %arg12 = %c0 to %28 step %c30 {
        for %arg13 = %c0 to %27 step %c40 {
         %49 = s.subview %37[%arg14, %47, %c1, %arg15, %48, %c1] : memref<?x?xf32, #map3>
         s.matmul (%43, %46, %49) : memref<?x?xf32, #map3>, memref<?x?xf32, #map3>, memref<?x?xf32, #map3>
...
s.generic #some_generic_trait %arg0, %arg3 {
^bb0(%arg5: f32, %arg6: f32):       // no predecessors
 s.yield %arg5 : f32
}: memref<?x?xf32, #map3>, memref<?x?xf32, #map3>
...
for %arg5 = %c0 to %3 step %c100 {
 for %arg6 = %c0 to %4 step %c150 {
  ...
  %17 = s.subview %arg2[%arg5, %5, %c1, %c0, %6, %c1] : memref<?x?xf32, #map3>
  for %arg7 = %c0 to %18 step %c20 {
   for %arg8 = %c0 to %20 step %c30 {
    for %arg9 = %c0 to %19 step %c40 {
     ...
     %32 = s.subview %17[%arg7, %30, %c1, %arg8, %31, %c1] : memref<?x?xf32, #map3>
     s.matmul (%26, %29, %32) : memref<?x?xf32, #map3>, memref<?x?xf32, #map3>, memref<?x?xf32, #map3>
  for %arg7 = %c0 to %21 step %c20 {
   for %arg8 = %c0 to %23 step %c30 {
    for %arg9 = %c0 to %22 step %c40 {
     ...
     %32 = s.subview %13[%arg7, %30, %c1, %arg8, %31, %c1] : memref<?x?xf32, #map3>
     s.matmul (%26, %29, %32) : memref<?x?xf32, #map3>, memref<?x?xf32, #map3>, memref<?x?xf32, #map3>
```

Declarative Transformations (3) Unroll and Pack



Declarative Transformations (3) Unroll and Pack

1. Match: Vector_ContractionOp

1. Match: Shape[6, 4, 5, 3]

2. Unroll: reduction dimension 1 by factor 2

3. Tag: unrolled reduction dimension 1

```
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Unroll<[6, 2, 5, 3], "UnrollReductionDim1"> $op), [(Constraint<HasShape<[6, 4, 5, 3]>> $op)]>;
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Pack<[1, 2, 1, 1], "PackDim1"> $op), [(Constraint<HasMarker<"UnrollReductionDim1">> $op)]>;
```

4. Match: “unrolled” marker

5. Pack: pack 2 iterations of reduction dimension 1 into one vector register

Google

Declarative Transformations (3) Unroll and Pack

```
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Unroll<[6, 2, 5, 3], "UnrollReductionDim1"> $op), [(Constraint<HasShape<[6, 4, 5, 3]>> $op)]>;
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Pack<[1, 2, 1, 1], "PackDim1"> $op), [(Constraint<HasMarker<"UnrollReductionDim1">> $op)]>;
```

```
%0 = vector.contract #contraction_trait %lhs, %rhs, %acc
      : vector<6x4x3xf32>, vector<5x4x3xf32> into vector<6x5xf32>
```

Declarative Transformations (3) Unroll and Pack

```
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Unroll<[6, 2, 5, 3], "UnrollReductionDim1"> $op), [(Constraint<HasShape<[6, 4, 5, 3]>> $op)]>;
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Pack<[1, 2, 1, 1], "PackDim1"> $op), [(Constraint<HasMarker<"UnrollReductionDim1">> $op)]>;
```

```
// Unroll reduction dimension 1 by a factor of 2
%0 = vector.slice %arg0 {offsets = [0, 0, 0], sizes = [6, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%1 = vector.slice %arg1 {offsets = [0, 0, 0], sizes = [5, 2, 3]} : vector<5x4x3xf32> to vector<5x2x3xf32>
%2 = vector.contract #contraction_trait %0, %1, %acc : vector<6x2x3xf32>, vector<5x2x3xf32> into vector<6x5xf32>

%3 = vector.slice %arg0 {offsets = [0, 2, 0], sizes = [6, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%4 = vector.slice %arg1 {offsets = [0, 2, 0], sizes = [5, 2, 3]} : vector<5x4x3xf32> to vector<5x2x3xf32>
%5 = vector.contract #contraction_trait %3, %4, %2 : vector<6x2x3xf32>, vector<5x2x3xf32> into vector<6x5xf32>
```

Declarative Transformations (3) Unroll and Pack

```
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Unroll<[6, 2, 5, 3], "UnrollReductionDim1"> $op), [(Constraint<HasShape<[6, 4, 5, 3]>> $op)]>;
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Pack<[1, 2, 1, 1], "PackDim1"> $op), [(Constraint<HasMarker<"UnrollReductionDim1">> $op)]>;
```

```
// Match and transform first unrolled contraction op
%0 = vector.slice %arg0 {offsets = [0, 0, 0], sizes = [6, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%1 = vector.slice %arg1 {offsets = [0, 0, 0], sizes = [5, 2, 3]} : vector<5x4x3xf32> to vector<5x2x3xf32>
%2 = vector.contract #contraction_trait %0, %1, %acc : vector<6x2x3xf32>, vector<5x2x3xf32> into vector<6x5xf32>

%3 = vector.slice %arg0 {offsets = [0, 2, 0], sizes = [6, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%4 = vector.slice %arg1 {offsets = [0, 2, 0], sizes = [5, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%5 = vector.contract #contraction_trait %3, %4, %2 : vector<6x2x3xf32>, vector<5x2x3xf32> into vector<6x5xf32>
```

Declarative Transformations (3) Unroll and Pack

```
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Unroll<[6, 2, 5, 3], "UnrollReductionDim1"> $op), [(Constraint<HasShape<[6, 4, 5, 3]>> $op)]>;
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Pack<[1, 2, 1, 1], "PackDim1"> $op), [(Constraint<HasMarker<"UnrollReductionDim1">> $op)]>;
```

```
// Slice and pack LHS vectors
%0 = vector.slice %arg0 {offsets = [0, 0, 0], sizes = [6, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%1 = vector.slice %arg1 {offsets = [0, 0, 0], sizes = [5, 2, 3]} : vector<5x4x3xf32> to vector<5x2x3xf32>
%2 = vector.contract #contraction_trait %0, %1, %acc : vector<6x2x3xf32>, vector<5x2x3xf32> into vector<6x5xf32>

%2 = vector.slice %0 {offsets = [0, 0, 0], sizes = [6, 1, 3]} : vector<6x2x3xf32> to vector<6x1x3xf32>
%3 = vector.slice %0 {offsets = [0, 1, 0], sizes = [6, 1, 3]} : vector<6x2x3xf32> to vector<6x1x3xf32>
%4 = vector.rotate %3 : vector<6x1x3xf32>
%5 = vector.pack %2, %4 : vector<6x1x6xf32>
```

Declarative Transformations (3) Unroll and Pack

```
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Unroll<[6, 2, 5, 3], "UnrollReductionDim1"> $op), [(Constraint<HasShape<[6, 4, 5, 3]>> $op)]>;
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Pack<[1, 2, 1, 1], "PackDim1"> $op), [(Constraint<HasMarker<"UnrollReductionDim1">> $op)]>;
```

```
// Slice and pack RHS vectors
%0 = vector.slice %arg0 {offsets = [0, 0, 0], sizes = [6, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%1 = vector.slice %arg1 {offsets = [0, 0, 0], sizes = [5, 2, 3]} : vector<5x4x3xf32> to vector<5x2x3xf32>
%2 = vector.contract #contraction_trait %0, %1, %acc : vector<6x2x3xf32>, vector<5x2x3xf32> into vector<6x5xf32>

%2 = vector.slice %0 {offsets = [0, 0, 0], sizes = [6, 1, 3]} : vector<6x2x3xf32> to vector<6x1x3xf32>
%3 = vector.slice %0 {offsets = [0, 1, 0], sizes = [6, 1, 3]} : vector<6x2x3xf32> to vector<6x1x3xf32>
%4 = vector.rotate %3 : vector<6x1x3xf32>
%5 = vector.pack %2, %4 : vector<6x1x6xf32>

%6 = vector.slice %1 {offsets = [0, 0, 0], sizes = [6, 1, 3]} : vector<5x2x3xf32> to vector<5x1x3xf32>
%7 = vector.slice %1 {offsets = [0, 1, 0], sizes = [6, 1, 3]} : vector<5x2x3xf32> to vector<5x1x3xf32>
%8 = vector.rotate %7 : vector<5x1x3xf32>
%9 = vector.pack %6, %8 : vector<5x1x6xf32>
```

Declarative Transformations (3) Unroll and Pack

def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Unroll<[6, 2, 5, 3], "UnrollReductionDim1"> $op), [(Constraint<HasShape<[6, 4, 5, 3]>> $op)]>;
def : Pat<(Vector_ContractionOp:$op $A, $B, $C), (Pack<[1, 2, 1, 1], "PackDim1"> $op), [(Constraint<HasMarker<"UnrollReductionDim1">> $op)]>;

```
// Create new contraction op operating on packed registers (better vector unit utilization)
%0 = vector.slice %arg0 {offsets = [0, 0, 0], sizes = [6, 2, 3]} : vector<6x4x3xf32> to vector<6x2x3xf32>
%1 = vector.slice %arg1 {offsets = [0, 0, 0], sizes = [5, 2, 3]} : vector<5x4x3xf32> to vector<5x2x3xf32>
%2 = vector.contract #contraction_trait %0, %1, %acc : vector<6x2x3xf32>, vector<5x2x3xf32> into vector<6x5xf32>

%2 = vector.slice %0 {offsets = [0, 0, 0], sizes = [6, 1, 3]} : vector<6x2x3xf32> to vector<6x1x3xf32>
%3 = vector.slice %0 {offsets = [0, 1, 0], sizes = [6, 1, 3]} : vector<6x2x3xf32> to vector<6x1x3xf32>
%4 = vector.rotate %3 : vector<6x1x3xf32>
%5 = vector.pack %2, %4 : vector<6x1x6xf32>

%6 = vector.slice %1 {offsets = [0, 0, 0], sizes = [6, 1, 3]} : vector<5x2x3xf32> to vector<5x1x3xf32>
%7 = vector.slice %1 {offsets = [0, 1, 0], sizes = [6, 1, 3]} : vector<5x2x3xf32> to vector<5x1x3xf32>
%8 = vector.rotate %7 : vector<5x1x3xf32>
%9 = vector.pack %6, %8 : vector<5x1x6xf32>

%10 = vector.contract #contraction_trait %5, %9, %acc : vector<6x1x6xf32>, vector<5x1x6xf32> into vector<6x5xf32>
```

Conclusion

MLIR Codegen: Making Good Use of Structure

This is the vision underlying our design and implementation for MLIR codegen
- Harness hardware and (domain-specific) language heterogeneity
- Simplify writing and maintaining retargetable analyses and transformations

Structure is static information that enables the **composition** of the following steps
- Defining standard and custom ops for HPC and ML
  - capturing op, iterator and type-specific properties
- Progressive lowering to loops, hardware/library blocks, hybrid of both
  - enabling key transformations (tile, fuse, transpose, copy to smaller buffer)
  - lower to control flow & side-effects only when and where it makes sense
  - offer declarative patterns leveraging MLIR rewriting and legalization logic
  - expose hardware constraints and cost model = *f*(hardware)

Feedback welcome!

Questions?

Thank you!