Structured Code Generation

Alex Zinenko - Google DeepMind
Nicolas Vasilache - Google Research

Google Research

Structured Code Generation is…

- … too complicated.
- … difficult to generalize.
- … too “researchy”.
- … is a dogmatic all-or-nothing approach.
- … not ready yet.
- …

Structured Code Generation

(you are already using it)

Alex Zinenko - Google DeepMind
Nicolas Vasilache - Google Research

Google Research

Structured Code Generation

Google Research

Finding Structure

f32 = f32 + f32

Not much structure here...

Code: c = a + b

AVX2: vaddss

MLIR: arith.addf : f32

Disclaimer: all code in the slides is pseudo code.
Disclaimer 2: our code is in MLIR, but the concepts generalize.

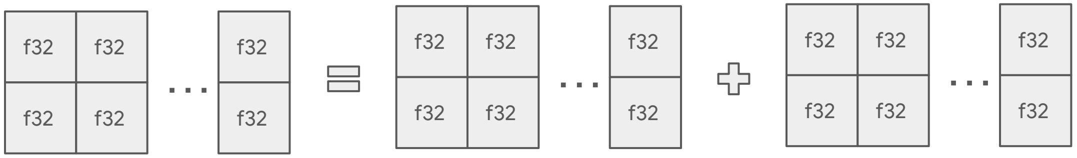
Finding Structure: Vectors

Code:	c[**0:8**] = a[0:8] + b[0:8]
AVX2:	vadd**ps**
MLIR:	arith.addf : **vector**<**8**xf32>

```
for i in 0:8
  c[i] = a[i] + b[i]
```

Structure: repetition

Finding Structure: Vectors



Code: c[0:8][**0:32**] = a[0:8][0:32] + b[0:8][0:32]

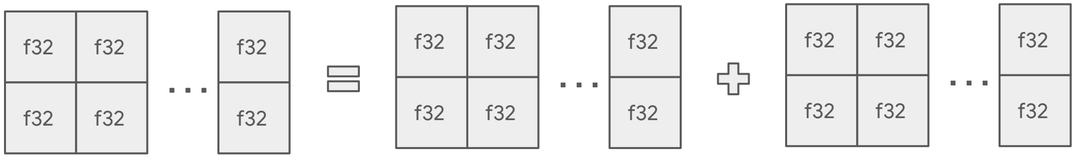
AVX2:

vaddps
vaddps
… 29 more … also, splitting
vaddps

MLIR: arith.addf : vector<**8x32**xf32>

```
for i in 0:8
  for j in 0:32
  c[i][j] = a[i][j] + b[i][j]
```

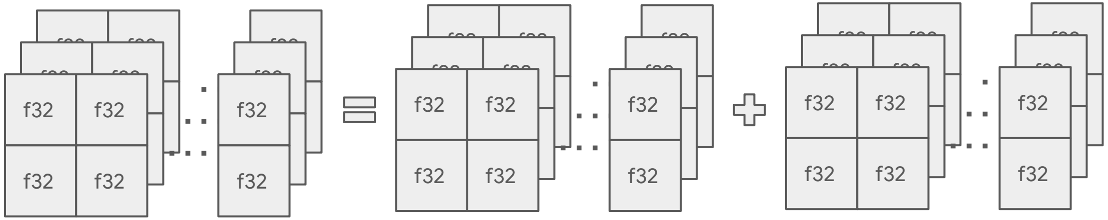
Finding Structure: Vectors



Code:	c[0:8][**0:32**] = a[0:8][0:32] + b[0:8][0:32]
AVX2:	vaddps vaddps **… 29 more …** also, splitting vaddps
MLIR:	arith.addf : vector<**8x32**xf32>
LLO:	vaddf32

```
for i in 0:8
  for j in 0:32
  c[i][j] = a[i][j] + b[i][j]
```

Finding Structure: Vectors



Code: c[0:8][0:32][**0:4**] = a[0:8][0:32][0:4] + b[0:8][0:32][0:4]

```
for i in 0:8
  for j in 0:32
    for k in 0:4
      c[i][j][k] = a[i][j][k] + b[i][j][k]
```

AVX2:
vaddps
vaddps
… 125 more … also, shuffle
vaddps

MLIR: arith.addf : vector<**8x32x4**xf32>

LLO: vaddf32 plus some reshuffling

Structure 1:
Uniform Repetition

Finding Structure: Vector Broadcast

Code: c[0:8] = a[0:8] + b

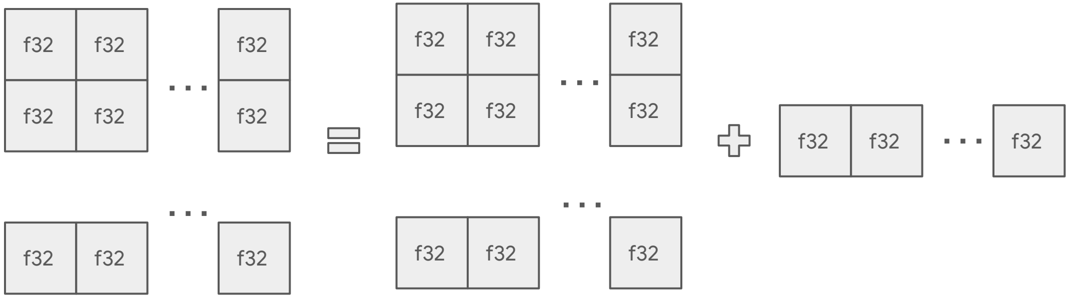
```
for i in 0:8
  c[i] = a[i] + b
```

AVX2: **vbroadcastss**
vaddps

Naming things: broadcast

MLIR: vector.**broadcast** : f32 to vector<8xf32>
arith.addf : vector<8xf32>

Finding Structure: Vector Broadcast



Code: c[0:8][0:8] = a[0:8][0:8] + b[0:8]

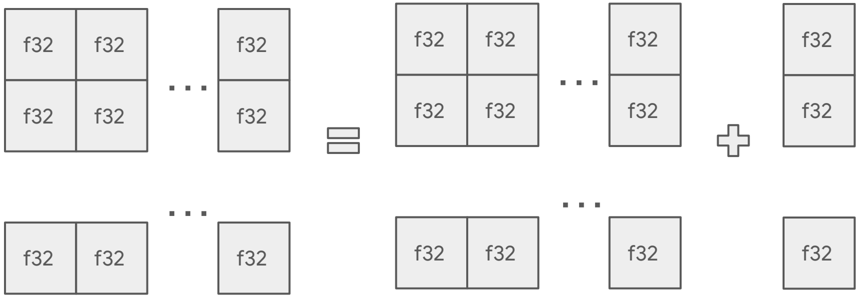
```
for i in 0:8
  for j in 0:8
    c[i][j] = a[i][j] + b[i]
```

AVX2: ~~vbroadcastss~~
vaddps
… 7 more …

MLIR: vector.broadcast : vector<**8x**f32> to vector<**8x**8xf32>
arith.addf : vector<8x8xf32>

Google Research

Finding Structure: Vector Broadcast



Code: c[0:8][0:8] = a[0:8][0:8] + b[0:8] Ouch...

AVX2:

```
vbroadcastss
... 7 more ...
vaddps
... 7 more ...
```

MLIR:

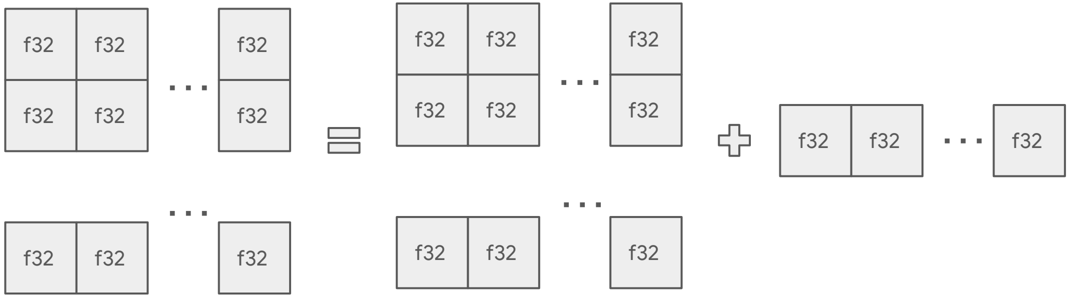
```
vector.broadcast : vector<8xf32> to vector<8x8xf32>
vector.transpose : vector<8x8xf32>
arith.addf : vector<8x8xf32>
```

```
for i in 0:8
  for j in 0:8
    c[i][j] = a[i][j] + b[j]
```

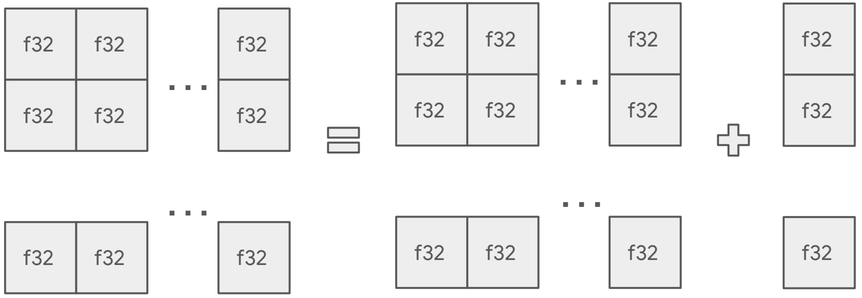
Google Research

Leveraging Structure for Representation

```
for i in 0:8
  c[i] = a[i] + b
```



```
for i in 0:8
  for j in 0:8
    c[i][j] = a[i][j] + b[i]
```

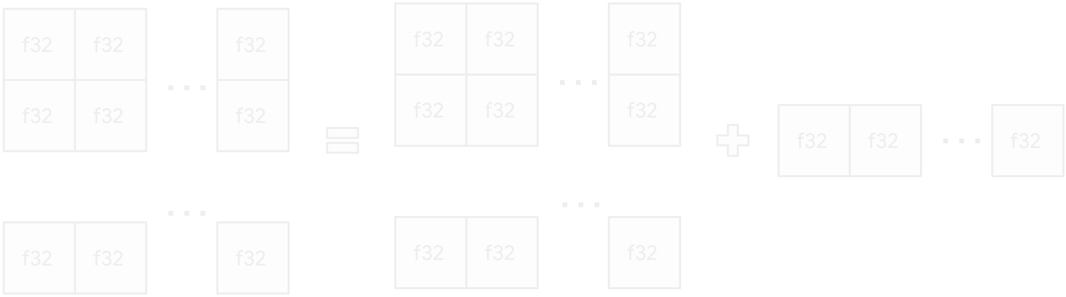


```
for i in 0:8
  for j in 0:8
    c[i][j] = a[i][j] + b[j]
```

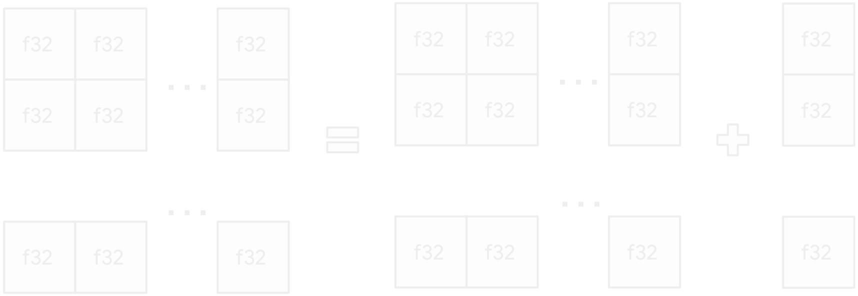
Google Research

Leveraging Structure for Representation

```
for i
  c[i] = a[i] + b
```



```
for i
  for j
    c[i][j] = a[i][j] + b[i]
```



```
for i
  for j
    c[i][j] = a[i][j] + b[j]
```

Google Research

Leveraging Structure for Representation

```
for i
  c[i] = a[i] + b
```

```
c, a:   (i) -> (i)
b:      (i) -> ()
```

```
for i
  for j
    c[i][j] = a[i][j] + b[i]
```

```
c, a:   (i,j) -> (i,j)
b:      (i) -> (i)
```

```
for i
  for j
    c[i][j] = a[i][j] + b[j]
```

```
c, a:   (i,j) -> (i,j)
b:      (i) -> (j)
```

Leveraging Structure for Representation

```
for i
 c[i] = a[i] + b
```

c, a: (i) -> (i)
b: (i) -> ()

```
vector.broadcasting_elementwise {
 indexing_maps = [affine_map<(i, j) -> (i, j)>,   // a
                 affine_map<(i, j) -> (i)>,      // b
                 affine_map<(i, j) -> (i, j)>],  // c
 iterator_types = [“parallel”, “parallel”],
 kind = #vector.kind<add>
} : (vector<8x8xf32>, vector<8xf32>) -> vector<8x8xf32>
```

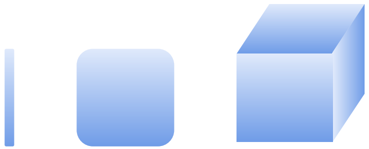
```
for i
 for j
  c[i][j] = a[i][j] + b[i]
```

c, a: (i,j) -> (i,j)
b: (i) -> (i)

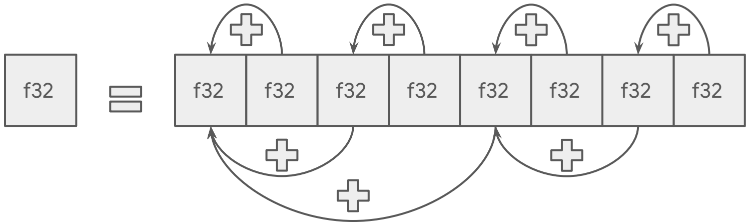
```
for i
 for j
  c[i][j] = a[i][j] + b[j]
```

c, a: (i,j) -> (i,j)
b: (i) -> (j)

Structure 2:
Dimensionality Increase



Finding Structure: Vector Reduction



Code: `c += a[0:8][0:2]`

```
for i in 0:2
  c += a[i]
```

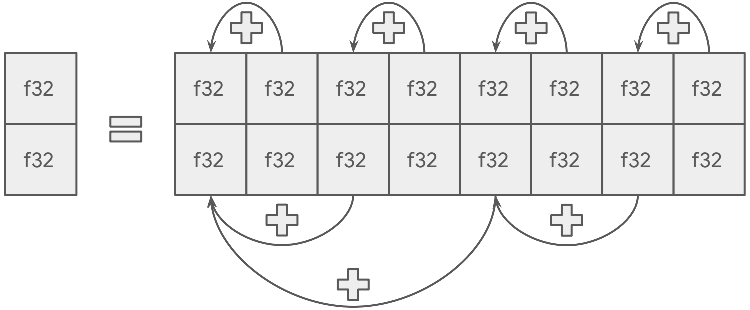
AVX2:

```
vhaddps
vhaddps + vpshufd
vhaddps
```

MLIR: vector.**reduction**<add> : vector<8xf32> into f32

Google Research

Finding Structure: Vector Reduction



Code: `c[0:2] += a[0:8][0:2]`

```
for i in 0:2
  for j in 0:8
    c[j] += a[i][j]
```

AVX2:

```
vhaddps
... 4 more ...
vhaddps
```

Meh...

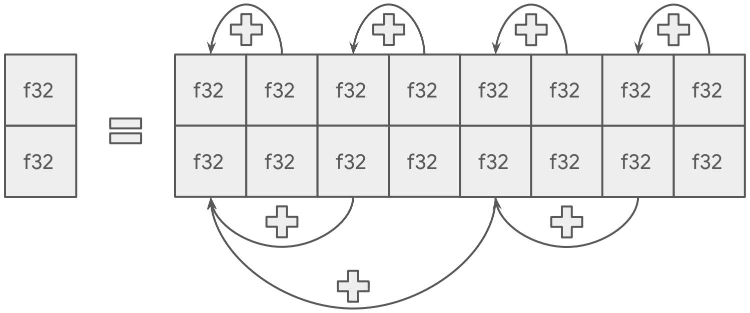
MLIR:

vector.**reduction**<add> : vector<8xf32> into f32
vector.**reduction**<add> : vector<8xf32> into f32

Google Research

What if I told you that we can reuse the same structure?

Finding Structure: Vector ~~Red~~ Contraction

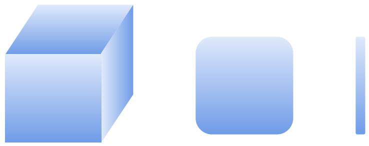


```
vector.contract {
 indexing_maps = [affine_map<(i, j) -> (i, j)>,   // a
                  affine_map<(i, j) -> ()>,       // b
                  affine_map<(i, j) -> (j)>],     // c
 iterator_types = [“reduction”, “parallel”],
 kind = #vector.kind<add>
} : (vector<8x8xf32>, vector<f32>) -> vector<8xf32>
```

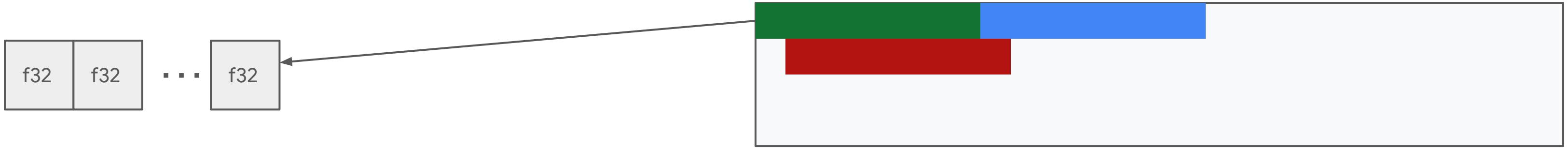
```
for i in 0:2
  for j in 0:8
    c[j] += a[i][j] * 1
```

Dummy constant 1, because contraction is always c += a * b

Structure 2b:
Dimensionality Decrease



Finding Structure In Memory Accesses



Code: c[0:8] = load(&p)

```
for i in 0:8
  c[i] = load(&p + i)
```

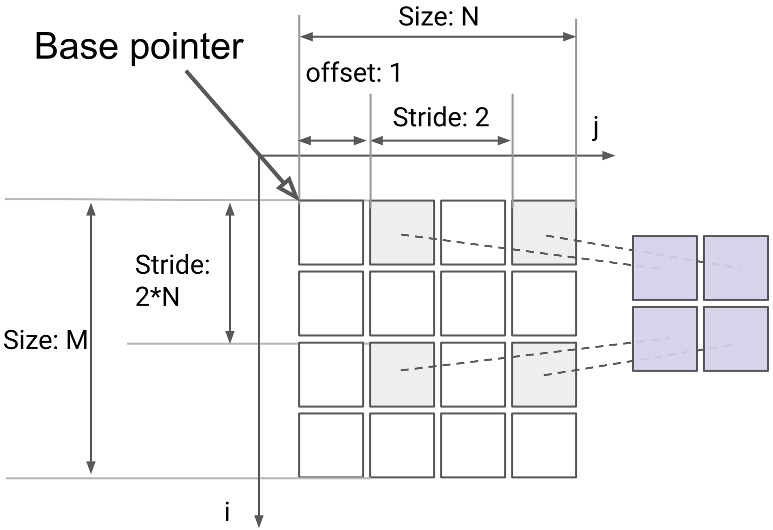
AVX2: vmovaps

or vmovups

MLIR: memref.load : **memref**<?x**vector**<8xf32>>

or vector.load : memref<?x**f32**>, vector<8xf32>

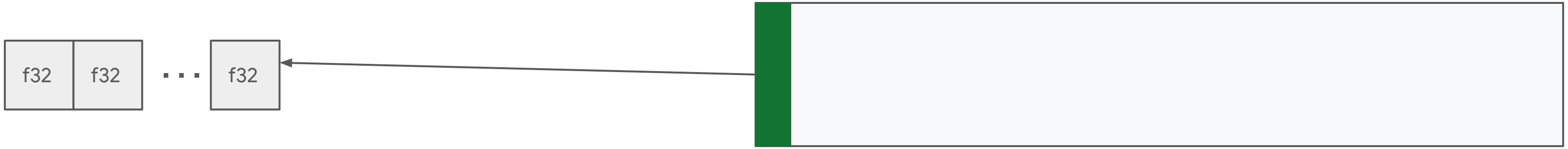
Intermezzo: Memory Reference Type



Base pointer, offset, sizes along each dimension, strides (# of elements) along each dimension.
Strides allow for transposed access.
Elemental types may be vectors to guarantee contiguity.

Google Research

Finding Structure In Memory Accesses



Code: c[0:8] = load(&p)

AVX2: vgatherqps

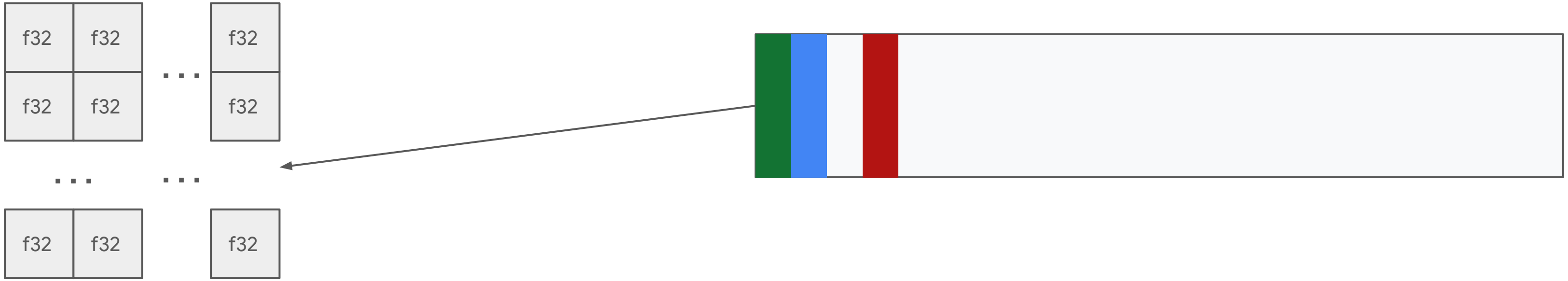
MLIR: ~~memref.load : memref<?xvector<8xf32>>~~

or ~~vector.load : memref<?xf32>, vector<8xf32>~~

vector.**transfer_read**
: memref<?x?xf32>, vector<8xf32>
{ permutation_map = affine_map<(i,j)->(j,i)> }

```
for i in 0:8
  c[i] = load(&p + 42*i)
```

Finding Structure In Memory Accesses

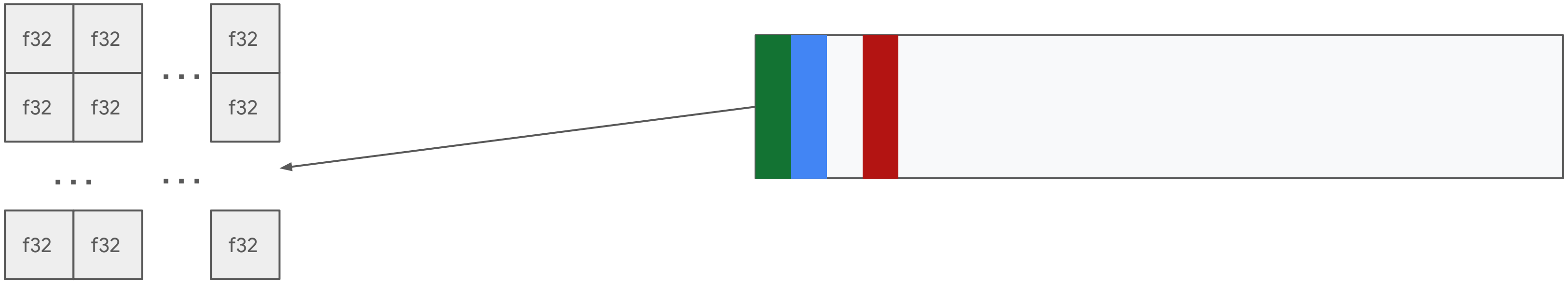


vector.**transfer_read**
: memref<?x?xf32>, vector<8x8xf32>
{ permutation_map = affine_map<(i,j)->(j,i)> }

vector.gather

vgatherqps
… 6 more …
vgatherqps

Finding Structure In Memory Accesses



vector.**transfer_read**
: memref<?x?xf32>, vector<8x8xf32>
{ permutation_map = affine_map<(i,j)->(j,i)> }

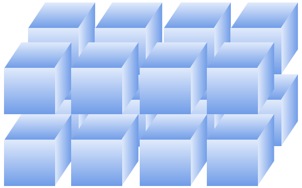
→ vector.load → vmovups *... 6 more ...* vmovups

vector.transpose → many vblendps + vpshufd

→ vector.gather → vgatherps *... 6 more ...* vgatherps

Google Research

Structure 3:
Multidimensional memory



Recap: Structure in Computations

- Elementwise extension to nD vectors.
- Dimensionality mismatch (broadcast or reduction) with explicit access maps and combinators.
- Similar structures in memory access.

- 1D and 2D vector operations are a special case of structured computations!
- Various HLO flavors are a special case of structured computations!

Extracting Common Structure

```
%0 = vector.load : memref<4x8xf32>, vector<4x8xf32>
%1 = vector.load : memref<4x8xf32>, vector<4x8xf32>
%2 = vector.broadcast 0 :    f32 to vector<4x8xf32>
%3 = arith.addf %0, %1 :            vector<4x8xf32>
%4 = arith.maxf %2, %3 :            vector<4x8xf32>
vector.store %4 :  memref<4x8xf32>, vector<4x8xf32>
```

Extracting Common Structure

```
%0 = vector.load : memref<4x8xf32>, vector<4x8xf32>
%1 = vector.load : memref<4x8xf32>, vector<4x8xf32>
%2 = vector.broadcast 0 :    f32 to vector<4x8xf32>
%3 = arith.addf %0, %1 :            vector<4x8xf32>
%4 = arith.maxf %2, %3 :            vector<4x8xf32>
vector.store %4 :  memref<4x8xf32>, vector<4x8xf32>
```

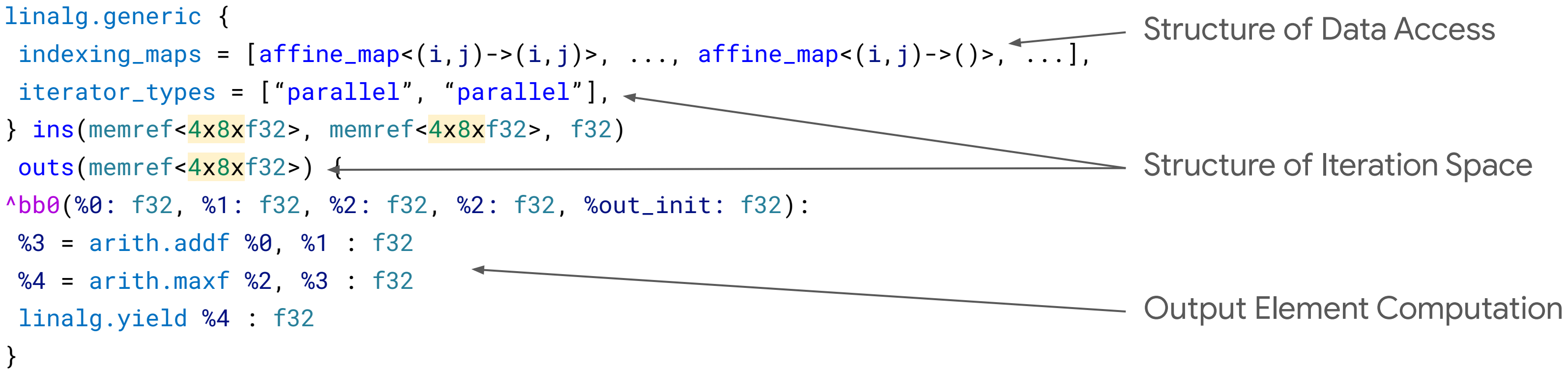
```
for i, j
  %0[i][j] = load(%p1 + f(i,j))
for i, j
  %1[i][j] = load(%p1 + g(i,j))
for i, j
  %2[i][j] = 0
for i, j
  %3[i][j] = %0[i][j] + %1[i][j]
for i, j
  %4[i][j] = maxf(%2[i][j], %3[i][j])
for i, j
  store(%p3 + h(i,j), %4[i][j])
```

```
for i, j
  %0[i][j] = load(%p1 + f(i,j))
  %1[i][j] = load(%p1 + g(i,j))
  %2[i][j] = 0
  %3[i][j] = %0[i][j] + %1[i][j]
  %4[i][j] = maxf(%2[i][j], %3[i][j])
  store(%p3 + h(i,j), %4[i][j])
```

```
for i, j
  %0 = load(%p1 + f(i,j))
  %1 = load(%p1 + g(i,j))
  %2 = 0
  %3 = %0 + %1
  %4 = maxf(%2, %3)
  store(%p3 + h(i,j), %4)
```

Google Research

Extracting Common Structure



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

Structure of Data Access

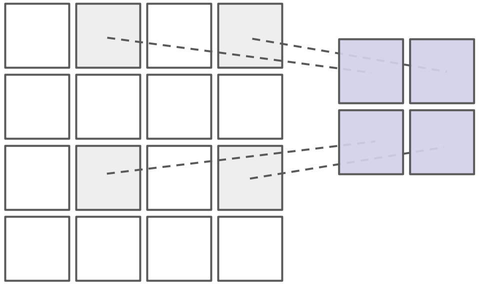
Structure of Iteration Space

Output Element Computation

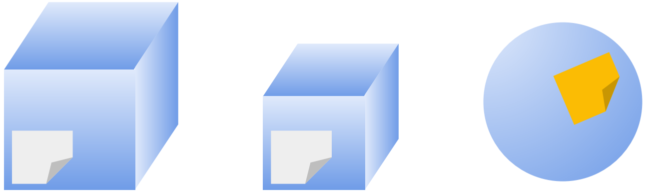
- Indexing can be elementwise, expansions, contractions, combinations (i+j).
- Iterators can be parallel or reduction.
- Output element is provided to allow for accumulation.

Operating on Subsets

```
%in1 = memref.subview %source1[offsets][sizes][strides] : memref<...xf32> to memref<...xf32>
%in2 = memref.subview %source2[offsets][sizes][strides] : memref<...xf32> to memref<...xf32>
linalg.generic {...}
  ins(memref<...xf32>, memref<...xf32>, f32)
 outs(memref<...xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %2: f32, %out_init: f32):
 %3 = arith.addf %0, %1 : f32
 %4 = arith.maxf %2, %3 : f32
 linalg.yield %4 : f32
}
```



Reify common structure



We identify and name different forms of structure.
(Naming things is one of the two hard problems in computer science.)

Finding Structure in SSA / Functional*

* SSA *is* functional programming

Values are immutable. Mutation (such as inserting an element) produces a new value.

f32 f32 ··· f32 = insert f32 into f32 f32 ··· f32

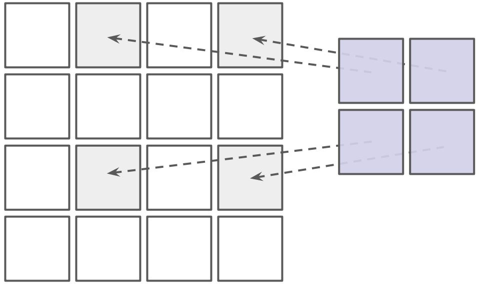
Code:	c[1] = 42
AVX512:	vinsertf32x8
MLIR:	vector.insert : f32 into vector<8xf32>
LLVM IR:	insertelement <8 x f32>, f32, i32

Finding Structure in SSA / Functional

Same works on MLIR tensors combined with “strided subset” abstraction from memref.

MLIR:

```
tensor.insert_slice %small, %big[offsets][sizes][strides]
  : tensor<...xf32> into tensor<...xf32>
```



Structured Everything on Tensors

```
%in1 = memref.subview %source1[offsets][sizes][strides] : memref<...xf32> to memref<...xf32>
%in2 = memref.subview %source2[offsets][sizes][strides] : memref<...xf32> to memref<...xf32>


       linalg.generic {...}
  ins(tensor<...xf32>, tensor<...xf32>, f32)
 outs(tensor<...xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %2: f32, %out_init: f32):
 %3 = arith.addf %0, %1 : f32
 %4 = arith.maxf %2, %3 : f32
 linalg.yield %4 : f32
} : tensor<...xf32>
```

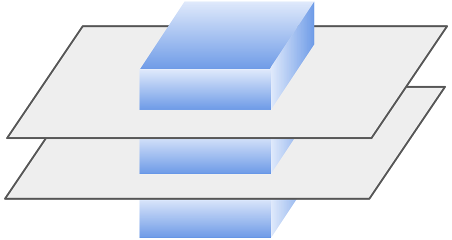
Google Research

Structured Everything on Tensors

```
~~%in1 = memref.subview %source1[offsets][sizes][strides] : memref<...xf32> to memref<...xf32>~~
~~%in2 = memref.subview %source2[offsets][sizes][strides] : memref<...xf32> to memref<...xf32>~~
tensor.extract_slice %source1[offsets][sizes][strides] : tensor<...xf32>
tensor.extract_slice %source2[offsets][sizes][strides] : tensor<...xf32>
%out = linalg.generic {...}
  ins(tensor<...xf32>, tensor<...xf32>, f32)
 outs(tensor<...xf32>) {
^bb0(%0: f32, %1: f32, %2: f32, %2: f32, %out_init: f32):
 %3 = arith.addf %0, %1 : f32
 %4 = arith.maxf %2, %3 : f32
 linalg.yield %4 : f32
} : tensor<...xf32>
%full_result = tensor.insert_slice %out into %result[offset][sizes][strides]
  : tensor<...xf32> into tensor<...xf32>
```

Google Research

Structure 4:
Immutable sliceable objects



Google Research

Structured Code Generation

Leveraging Structure for Code Generation

Recall how “generic” and “contraction” are explained as pseudo-code with loops.
We can reify these loops as IR!

```
linalg.generic {
 indexing_maps = [...],
 iterator_types = [“parallel”, “parallel”],
} ins(tensor<4x8xf32>, tensor<4x8xf32>, f32)
 outs(tensor<4x8xf32>) {
  ...
}
```

Google Research

Leveraging Structure for Code Generation

Recall how “generic” and “contraction” are explained as pseudo-code with loops.
We can reify these loops as IR!

```
linalg.generic {
 indexing_maps = [...],
 iterator_types = [“parallel”, “parallel”],
} ins(tensor<4x8xf32>, tensor<4x8xf32>, f32)
 outs(tensor<4x8xf32>) {
  ...
}
```

```
scf.forall %i, %j in (0:4, 0:8)
                                      {

                                    }
```

Google Research

Leveraging Structure for Code Generation

Recall how “generic” and “contraction” are explained as pseudo-code with loops.
We can reify these loops as IR!

```
linalg.generic {
 indexing_maps = [...],
 iterator_types = [“parallel”, “parallel”],
} ins(tensor<4x8xf32>, tensor<4x8xf32>, f32)
 outs(tensor<4x8xf32>) {
  ...
}
```

```
                          scf.forall %i, %j in (0:4, 0:8)
                                                        {
tensor.extract_slice %source1[%i, %j][1, 1][1, 1]
tensor.extract_slice %source2[%i, %j][1, 1][1, 1]






}
```

Leveraging Structure for Code Generation

Recall how “generic” and “contraction” are explained as pseudo-code with loops.
We can reify these loops as IR!

```
linalg.generic {
 indexing_maps = [...],
 iterator_types = [“parallel”, “parallel”],
} ins(tensor<4x8xf32>, tensor<4x8xf32>, f32)
 outs(tensor<4x8xf32>) {
  ...
}
```

```
%full_result = scf.forall %i, %j in (0:4, 0:8)
               shared_outs(%shared = %result) {
 tensor.extract_slice %source1[%i, %j][1, 1][1, 1]
 tensor.extract_slice %source2[%i, %j][1, 1][1, 1]


 scf.forall.in_parallel {
   tensor.parallel_insert_slice ... into %shared[%i, %j][1, 1][1, 1]
 }
}
```

Google Research

Leveraging Structure for Code Generation

Recall how “generic” and “contraction” are explained as pseudo-code with loops.
We can reify these loops as IR!

```
linalg.generic {
 indexing_maps = [...],
 iterator_types = [“parallel”, “parallel”],
} ins(tensor<4x8xf32>, tensor<4x8xf32>, f32)
 outs(tensor<4x8xf32>) {
  ...
}
```

```
%full_result = scf.forall %i, %j in (0:4, 0:8)
               shared_outs(%shared = %result) {
 tensor.extract_slice %source1[%i, %j][1, 1][1, 1]
 tensor.extract_slice %source2[%i, %j][1, 1][1, 1]
 ...

 scf.forall.in_parallel {
   tensor.parallel_insert_slice ... into %shared[%i, %j][1, 1][1, 1]
 }
}
```

Leveraging Structure for Tiling

Recall how “generic” and “contraction” are explained as pseudo-code with loops.
We can reify these loops as IR!

```
linalg.generic {
 indexing_maps = [...],
 iterator_types = [“parallel”, “parallel”],
} ins(tensor<4x8xf32>, tensor<4x8xf32>, f32)
 outs(tensor<4x8xf32>) {
  ...
}
```

```
%full_result = scf.forall %i, %j in (0:2, 0:4)
               shared_outs(%shared = %result)
 tensor.extract_slice %source1[%i, %j][2, 2][1, 1]
 tensor.extract_slice %source2[%i, %j][2, 2][1, 1]
 linalg.generic { ... }
   ins(tensor<2x2xf32>, tensor<2x2xf32>, f32) outs(tensor<2x2xf32>)
 scf.forall.in_parallel {
   tensor.parallel_insert_slice ... into %shared[%i, %j][2, 2][1, 1]
 }
}
```

Leveraging Structure for Fusion

Recall how “generic” and “contraction” are explained as pseudo-code with loops.
We can reify these loops as IR!

```
linalg.generic {
 indexing_maps = [...],
 iterator_types = [“parallel”, “parallel”],
} ins(tensor<4x8xf32>, tensor<4x8xf32>, f32)
 outs(tensor<4x8xf32>) {
  ...
}
```

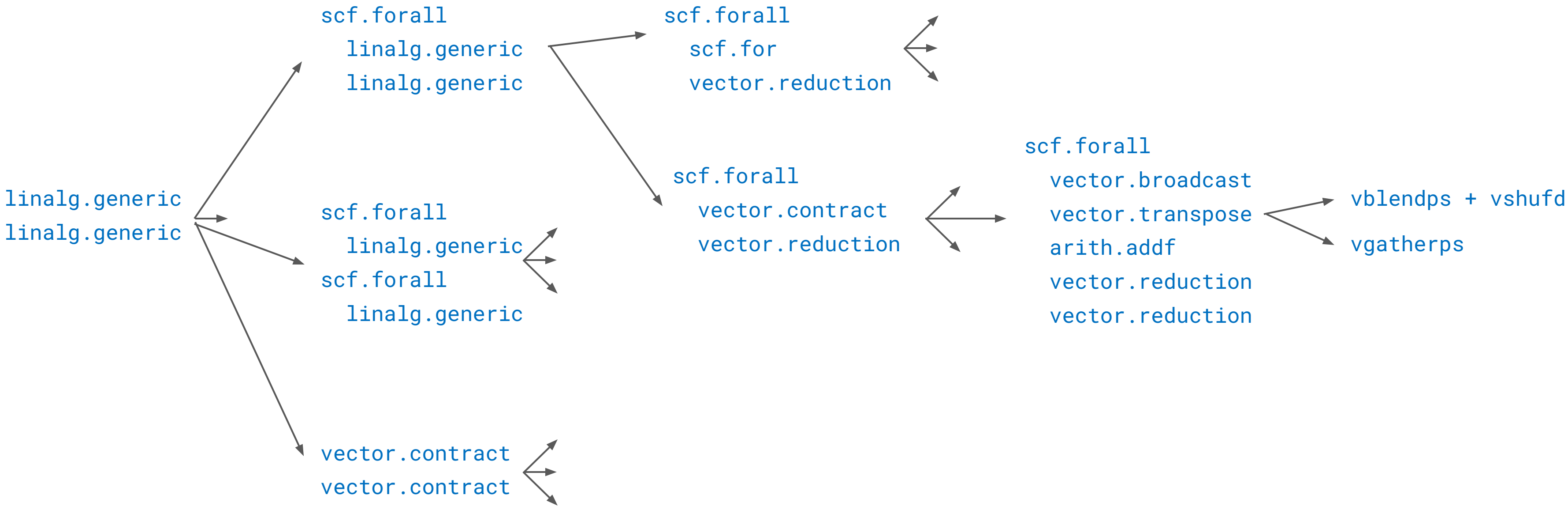
```
 %source1 = linalg.generic {...}


%full_result = scf.forall %i, %j in (0:2, 0:4)
                  shared_outs(%shared = %result)
 tensor.extract_slice %source1[%i, %j][2, 2][1, 1]
 tensor.extract_slice %source2[%i, %j][2, 2][1, 1]
 linalg.generic { ... }
   ins(tensor<2x2xf32>, tensor<2x2xf32>, f32) outs(tensor<2x2xf32>)
 scf.forall.in_parallel {
   tensor.parallel_insert_slice ... into %shared[%i, %j][2, 2][1, 1]
 }
}
```

Google Research

Code Generation is a Choice*

*actually, lots of choices.



```
linalg.generic
linalg.generic
```

```
scf.forall
  linalg.generic
  linalg.generic
```

```
scf.forall
  linalg.generic
scf.forall
  linalg.generic
```

```
vector.contract
vector.contract
```

```
scf.forall
  scf.for
  vector.reduction
```

```
scf.forall
  vector.contract
  vector.reduction
```

```
scf.forall
  vector.broadcast
  vector.transpose
  arith.addf
  vector.reduction
  vector.reduction
```

```
vblendps + vshufd
```

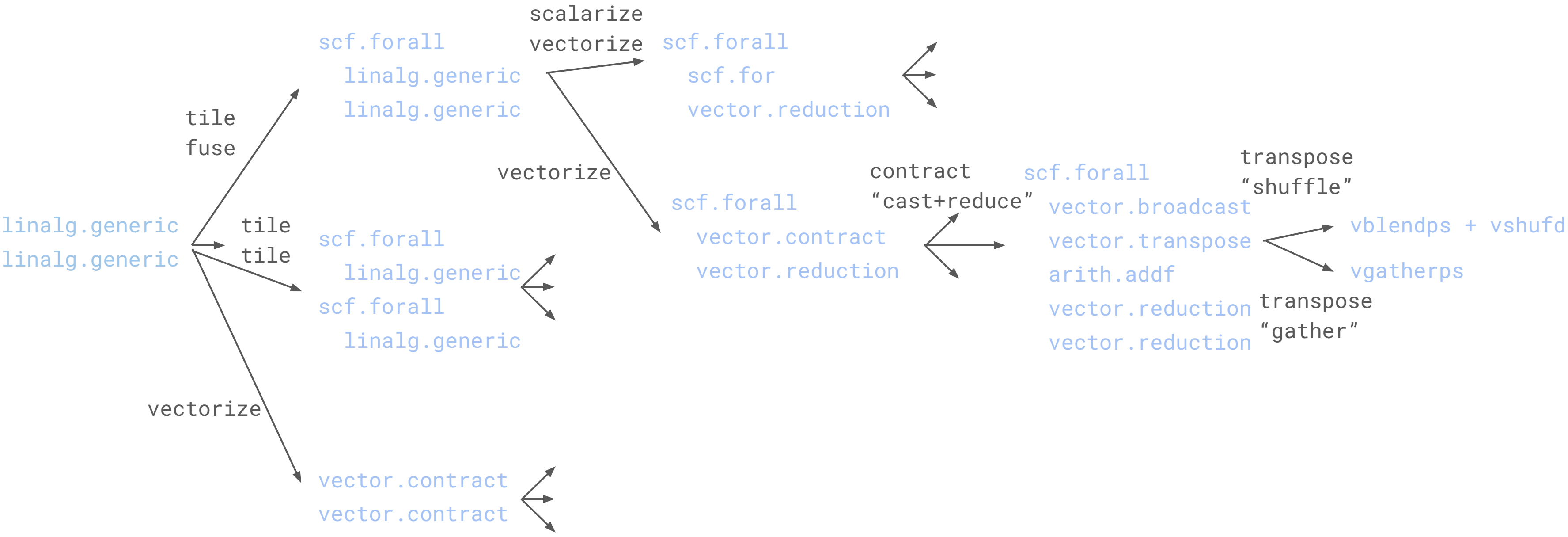
```
vgatherps
```

Structured code generation clearly separates the mechanics from decision making.
Mechanics is simple thanks to abstractions being designed for transformation.

Google Research

Code Generation is a Choice*

*actually, lots of choices.



`linalg.generic`
`linalg.generic`

tile
fuse

`scf.forall`
`  linalg.generic`
`  linalg.generic`

scalarize
vectorize

`scf.forall`
`  scf.for`
`  vector.reduction`

vectorize

`scf.forall`
`  vector.contract`
`  vector.reduction`

contract
“cast+reduce”

`scf.forall`
`vector.broadcast`
`vector.transpose`
`arith.addf`
`vector.reduction`
`vector.reduction`

transpose
“shuffle”

`vblendps + vshufd`

transpose
“gather”

`vgatherps`

tile
tile

`scf.forall`
`  linalg.generic`
`scf.forall`
`  linalg.generic`

vectorize

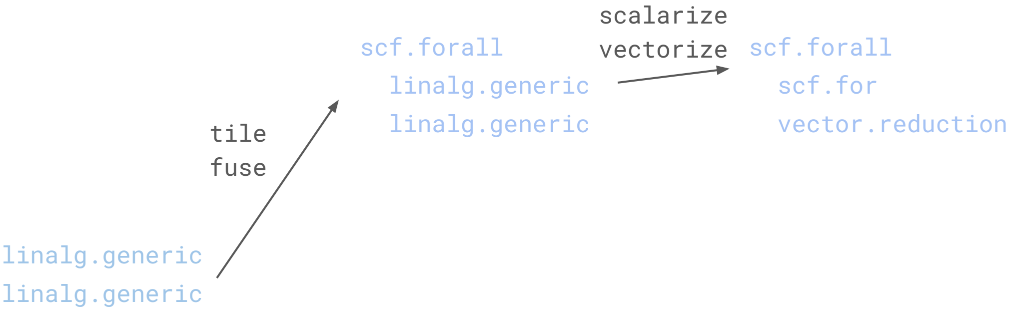
`vector.contract`
`vector.contract`

Same as before, we name transformations that are a part of code generation.

Google Research

Code Generation is Controllable

With a dialect, because everything in MLIR is a dialect.



```
linalg.generic
linalg.generic
```

tile
fuse

```
scf.forall
  linalg.generic
  linalg.generic
```

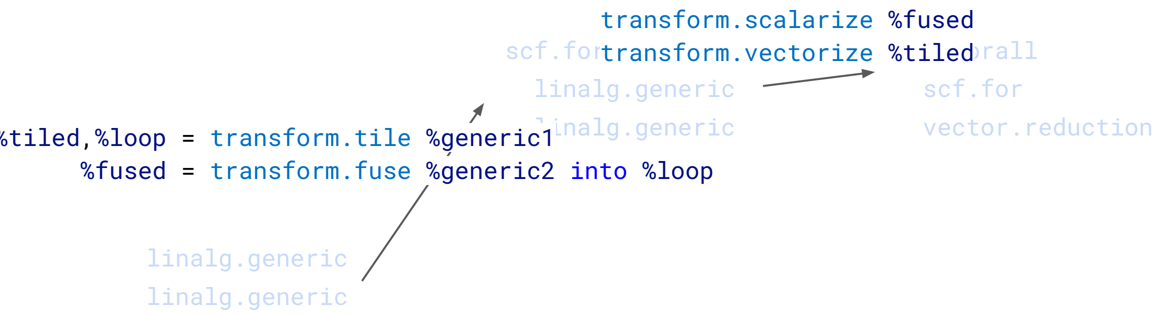
scalarize
vectorize

```
scf.forall
  scf.for
  vector.reduction
```

Specifies which of the operations gets fused, scalarized, vectorized, etc. and with what parameters.

Code Generation is Controllable

With a dialect, because everything in MLIR is a dialect.



```
                      transform.scalarize %fused
                      transform.vectorize %tiled
%tiled,%loop = transform.tile %generic1
      %fused = transform.fuse %generic2 into %loop
```

Specifies which of the operations gets fused, scalarized, vectorized, etc. and with what parameters.
A dialect => exchange/storage format, verifiable, interpretable (no need to recompile the compiler).

Structured Code Generation

Google Research

Structured Code Generation is...

- ... based on observations about *preexisting* structure
  (1d vectors, dimensionality change, non-flat memory, immutability).
- ... generalizing that structure to higher-dimensional objects.
- ... simplifying transformations by preserving the structural information
  (types, operations) and gives more control over them.
- ... is not limited to dense hyper-rectangular computation.
- ... nothing to be afraid of, *you are likely already using a version of it!*
  (in MLIR: llvm, memref, vector, tensor follow the same patterns)
  (outside: various vector programming models, Triton, etc.)