Structured Code Generation

Alex Zinenko - Google DeepMind
Nicolas Vasilache - 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
Structured Code Generation

Google Research
Finding Structure

\[ f_{32} = f_{32} + f_{32} \]

Code: \[ c = a + b \]
AVX2: \[ \text{vaddss} \]
MLIR: \[ \text{arith.addf : f32} \]

Not much structure here...

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: \( \text{vaddps} \)

MLIR: \( \text{arith.addf : vector<8xf32>} \)

for \( i \) in 0:8
\[ c[i] = a[i] + b[i] \]

Structure: repetition
Finding Structure: Vectors

Code: \( \begin{array}{ccc}
  c[0:8][0:32] & = & a[0:8][0:32] + b[0:8][0:32] \\
\end{array} \)

AVX2:
- vaddps
- vaddps
- ... 29 more ...
- also, splitting
- vaddps

MLIR:
- arith.addf : vector\(<8×32\)xf32>
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<8x32xf32>`

LLO: `vaddf32`
Finding Structure: Vectors

Code: \[ c_{i,j,k} = a_{i,j,k} + b_{i,j,k} \]

AVX2: 
- vaddps
- vaddps
- ... 125 more ... also, shuffle
- vaddps

MLIR: 
- `arith.addf : vector<8x32x4xf32>`

LLO: 
- vaddf32 plus some reshuffling
Structure 1: Uniform Repetition
Finding Structure: Vector Broadcast

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

AVX2: \texttt{vbroadcastss} \vaddps

MLIR: \texttt{vector.broadcast: f32 to vector<8xf32>} \texttt{arith.addf : vector<8xf32>}

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

Naming things: broadcast
Finding Structure: Vector Broadcast

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

AVX2: 
- \texttt{vbroadcastss}
- \texttt{vaddps}
- ... 7 more ...

MLIR: 
- \texttt{vector.broadcast : vector\<8xf32\> to vector\<8x8xf32\>}
- \texttt{arith.addf : vector\<8x8xf32\>}

for \( i \) in 0:8
- for \( j \) in 0:8
  - \( c[i][j] = a[i][j] + b[i] \)
Finding Structure: Vector Broadcast

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

AVX2:
- vbroadcastss
- vaddps

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
\[
c[i][j] = a[i][j] + b[j]
\]**Ouch...**
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
\[ c[i][j] = a[i][j] + b[j] \]
Leveraging Structure for Representation

\[
\begin{align*}
\text{for } i & \\
c[i] &= a[i] + b \\
\text{for } j & \\
c[i][j] &= a[i][j] + b[i]
\end{align*}
\]
Leveraging Structure for Representation

\[
\begin{align*}
\text{for } i & \quad c[i] = a[i] + b \\
\text{for } j & \quad c[i][j] = a[i][j] + b[i]
\end{align*}
\]

\[
\begin{align*}
\text{vector broadcasting_elementwise} & \quad \{
\text{indexing_maps} = \{\text{affine_map}\langle(i, j) \rightarrow (i, j)\rangle, \quad & \text{a} \\
\text{affine_map}\langle(i, j) \rightarrow (i)\rangle, \quad & \text{b} \\
\text{affine_map}\langle(i, j) \rightarrow (i, j)\rangle, \quad & \text{c}
\}, \quad \text{iterator_types} = \{\text{"parallel"}, \text{"parallel"}\}, \quad \text{kind} = \#\text{vector.kind<add>}
\} : (\text{vector<8x8xf32>, vector<8xf32>}) \rightarrow \text{vector<8x8xf32>}
\end{align*}
\]
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
Finding Structure: Vector Reduction

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

AVX2:
```
vhaddps
... 4 more ...
vhaddps
```

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

Meh...
What if I told you that we can reuse the same structure?
Finding Structure: Vector Red Contraction

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

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>

Dummy constant 1, because contraction is always c += a * b
Structure 2b: Dimensionality Decrease
Finding Structure In Memory Accesses

Code: \( c[0:8] = \text{load}(&p) \)

for i in 0:8
  \( c[i] = \text{load}(&p + i) \)

AVX2: vmovaps
     or vmovups

MLIR: \( \text{memref.load : memref<?xvector<8xf32>>>} \)
      or \( \text{vector.load : memref<?xf32>, vector<8xf32>} \)
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.
Finding Structure In Memory Accesses

Code: \( c[0:8] = \text{load}(&p) \)

AVX2: \( \text{vgatherqps} \)

MLIR: 
- `memref.load : memref<?xvector<8xf32>>`
- `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
\( c[i] = \text{load}(&p + 42*i) \)
vector.transfer_read
: memref<?x?xf32>, vector<8x8xf32>
{ permutation_map = affine_map<(i,j)->(j,i)> }
Finding Structure In Memory Accesses

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

```
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])
store(%p3 + h(i,j), %4[i][j])
```

```
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])
store(%p3 + h(i,j), %4[i][j])
```
Extracting Common Structure

```c
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
}
```

- 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

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

\[
\begin{array}{cccccccc}
| f32 | f32 | \cdots | f32 | \equiv | \text{insert} | f32 | \text{into} | f32 | f32 | \cdots | f32 |
\end{array}
\]

Code: \[c[1] = 42\]

AVX512: \[\text{vinserf32x8}\]

MLIR: \[\text{vector.insert : f32 into vector}<8\text{x}f32>\]

LLVM IR: \[\text{insertelement <8 x f32>, f32, i32}\]
Finding Structure in SSA / Functional

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

MLIR: tensor.insert_slice %small, %big[offssets][sizes][strides] : tensor<...xf32> into tensor<...xf32>
Structured Everything on Tensors

\[
\text{\%in1} = \text{memref.subview \%source1[\text{offsets][sizes][strides]} : memref<...xf32> \to memref<...xf32>}
\]
\[
\text{\%in2} = \text{memref.subview \%source2[\text{offsets][sizes][strides]} : memref<...xf32> \to memref<...xf32>}
\]

\[
\text{linalg.generic {\ldots}}
\]
\[
\text{ins(tensor<...xf32>, tensor<...xf32>, f32)}
\]
\[
\text{outs(tensor<...xf32>) {\ldots}}
\]
\[
^\text{bb0}(\%0: f32, \%1: f32, \%2: f32, \%2: f32, \%\text{out}\_\text{init}: f32):
\]
\[
\text{%3 = arith.addf \%0, \%1 : f32}
\]
\[
\text{%4 = arith.maxf \%2, \%3 : f32}
\]
\[
\text{linalg.yield \%4 : f32}
\]
\[
\text{\ldots : tensor<...xf32>}
\]
Structured Everything on Tensors

```cpp
%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 {...}
  1s(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>
```
Structure 4: Immutable sliceable objects
Structured 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>) {
    ...
}
```
Leveraging Structure for Code Generation

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

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

```plaintext
scf.forall (%i, %j in (0:4, 0:8)) {
    ...
}
```
Leveraging Structure for Code Generation

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

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

```python
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!

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

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

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

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

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

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

```cpp
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]
}

%source1 = linalg.generic {...}
Code Generation is a Choice*

*actually, lots of choices.

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

*actually, lots of choices.

Same as before, we name transformations that are a part of code generation.
Code Generation is Controllable

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

Specifies which of the operations gets fused, scalarized, vectorized, etc. and with what parameters.
With a dialect, because everything in MLIR is a dialect.

Code Generation is Controllable

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
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.)