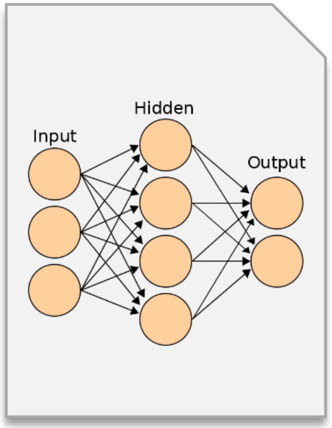
Karl F. A. Friebel[1], Asif A. Khan[1], Lorenzo Chelini[2], Jeronimo Castrillon[1]
[1]Technische Universität Dresden
[2] Intel Switzerland

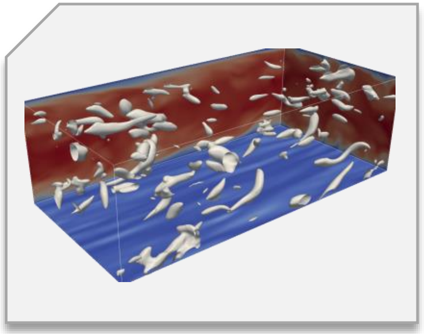
Modelling linear algebra kernels as polyhedral volume operations

IMPACT @ HiPEAC 2023, Toulouse // 2023-01-16

Linear algebra kernels

- Linear algebra as a central abstraction of many domains
  - Neural networks
  - Computational fluid dynamics
  - etc.
- Terse, implicit formulations
  - Einstein notation
  - MLIR `linalg.generic`
- Multi-phase compilation





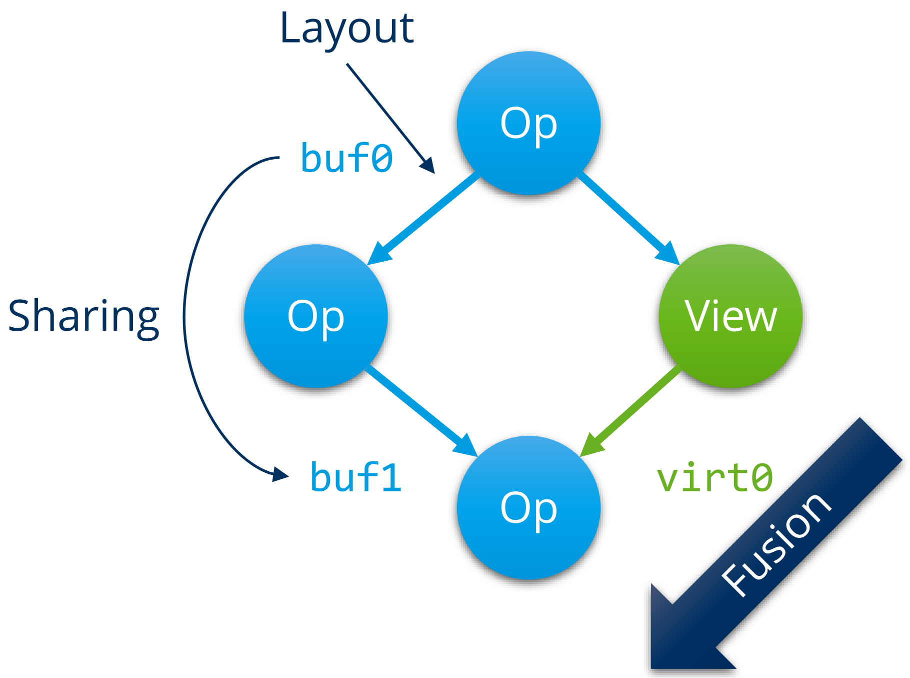
```
D[i, j] += A[i, k] * B[k, j]
E[i, j] += D[i, j] + C[i, j]
```

```
E1[i, j] = fuse(D[i, j], E[i, j])
E2[a, b, s, t] = tile(
    E1[i, j],
    [a, b, s, t] -> [16a+s, 16b+t])
```

Common techniques

- Tile & fuse is order-dependent
- „Physical“ vs „virtual“ tensors
- Layout & bufferization may be device-specific

→ Affine pattern matching
→ Polyhedral dataflow analysis
→ Liveness range splitting



$$\mathit{live}(\texttt{buf0}) \cap \mathit{live}(\texttt{buf1}) = \emptyset \rightarrow \texttt{buf1} = \texttt{buf0}$$

Target problems

- (feed-forward) Shape inference
  - Dynamically sized data
  - Feasability constraints

Partial specialization?

- Tile shape inference
  - Apply tiling on outputs
  - Tile inputs → partial reductions

Memory & FU constraints → output tiling

Reuse & locality → input tiling

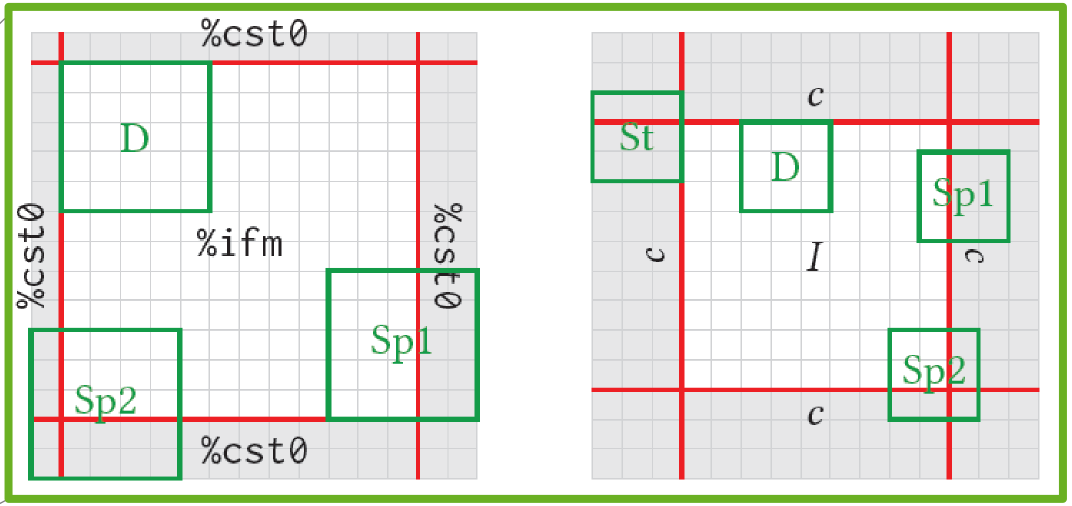
- Structural sparsity
  - Static inspector / specialized executor

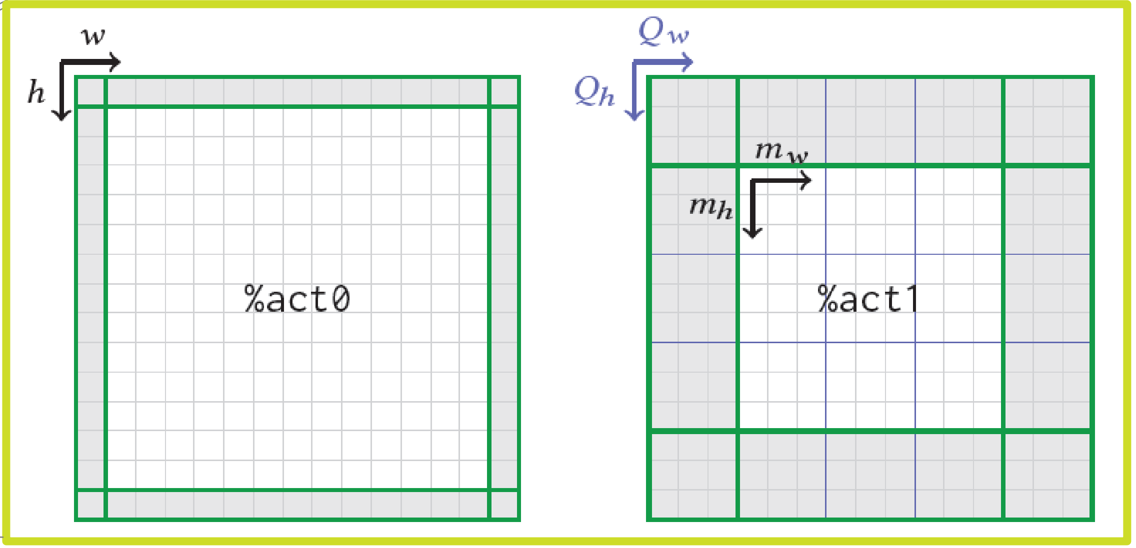
Library matching

Kernel extraction & generation

Linear algebra in MLIR

```
^krnl0(%ifm: tensor<1x3x512x512xf32>):
①%pad0 = tensor.pad %ifm low[0,0,1,1] high
      ↪ [0,0,2,2] {
    ^bb0(%i0: index, %i1: index, %i2: index, %i3:
        ↪ index):
      tensor.yield %cst0 : f32
  } : tensor<1x3x512x512xf32> to tensor<1
      ↪ x3x515x515xf32>
②%conv0 = linalg.conv_2d_nchw_fchw {
      dilations = dense<1> : tensor<2xi64>,
      strides = dense<2> : tensor<2xi64>
    }
    ins(%pad0, %wgt0: tensor<1x3x515x515xf32>,
        ↪ tensor<8x3x5x5xf32>)
    outs(%conv0.init: tensor<1x8x256x256xf32>)
    -> tensor<1x8x256x256xf32>
③%act0 = linalg.generic #elementwise_traits
      ins(%conv0: tensor<1x8x256x256xf32>)
      outs(%act0.init: tensor<1x8x256x256xf32>) {
    ^bb0(%a: f32, %b: f32):
      %0 = arith.maxf %a, %cst0 : f32
      %1 = arith.mulf %0, %cst0_01 : f32
      %2 = arith.addf %a, %1 : f32
      linalg.yield %2 : f32
  } -> tensor<1x8x256x256xf32>
```





State of the art

- Live-out-based transformations

  Jie Zhao and Albert Cohen. 2019. Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation. ACM Trans. Archit. Code Optim. 16, 4, Article 47 (dec 2019), 25 pages. https: //doi.org/10.1145/3369382

- Fusion ordering problem

  Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, Ponnuswamy Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: convexity, pruning and optimization. ACM SIGPLAN Notices 46, 1 (2011), 549–562

- Physical and virtual tensors (lifting)

  Norman A. Rink and Jeronimo Castrillon. 2019. TeIL: A Type-Safe Imperative Tensor Intermediate Language. In Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming - ARRAY 2019. ACM Press, Phoenix, AZ, USA, 57–68. https://doi.org/10.1145/3315454.3329959

- Inspector / Executor specialization

  Mahdi Soltan Mohammadi, Kazem Cheshmi, Ganesh Gopalakrishnan, Mary Hall, Maryam Mehri Dehnavi, Anand Venkat, Tomofumi Yuki, and Michelle Mills Strout. 2018. Sparse matrix code dependence analysis simplification at compile time. ArXiv e-prints (2018), arXiv–1807

Outline

- Introduction
- Volume-based dataflow analysis
- Usage & Implementation
- Conclusions

Volumes

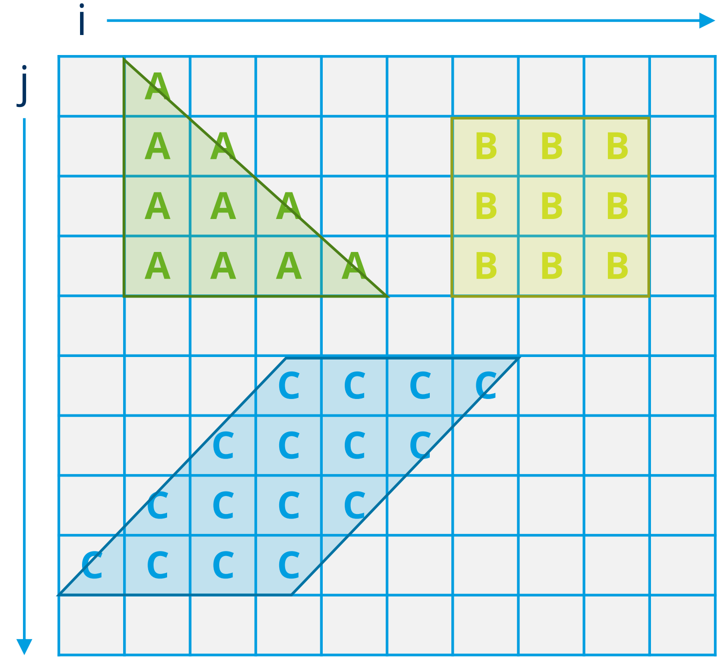
- Distinct, unambiguous term: Volume
- Affine generalization of structured, tuple-indexed aggregate

MLIR discourse: „*Structured Codegen Beyond Rectangular Arrays* “

- Scalars are also volumes
- Typically: Volume ←→ SSA value

Definition 4.1 (Volume). A *volume* V is an indexable aggregate of values with a polyhedral index domain $\operatorname{dom} V \subset \mathbb{Z}^N$, where N is the *rank* of the volume.

Definition 4.2 (Element). An *element* $V[i_1, \ldots, i_N] = V[\mathbf{i}]$ is the value of volume V at index $\mathbf{i} \in \operatorname{dom} V$.



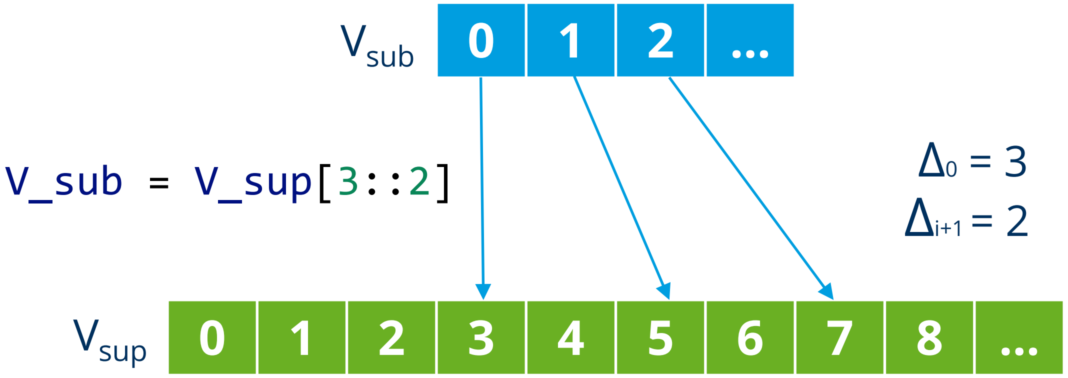
Arrays, Views & Slices

- Some concepts only apply to hyperrectangular volumes
  - Padding [convex poly. halo]
  - Offset-size-stride views
  - etc.
- Volumes should also model memory (minus mutability)
  - Memory is modelled by multi-dimensional arrays
- Layout map: Volume → Array

Definition 4.7 (Array). An *array* A is a volume with a hyperrectangular index domain such that $\Delta_0 \, \mathrm{dom}\, A = \mathbf{0}$.

Definition 4.8 (View). A *view* $\Pi : \mathrm{dom}\, V_{\mathrm{sub}} \longrightarrow \mathrm{dom}\, V_{\mathrm{sup}}$ is an affine map that defines a subvolume $V_{\mathrm{sub}}[\mathbf{i}] = V_{\mathrm{sup}}[\Pi(\mathbf{i})]$ of the supervolume V_{sup}.

Definition 4.9 (Slice). A *slice* $\Xi : \mathbf{i} \mapsto \Delta_0 \, \Xi + \mathbf{i} \odot \Delta_{\mathbf{i}+1} \, \Xi$ is a view defined by an offset $\Delta_0 \, \Xi$ and a stride $\Delta_{\mathbf{i}+1} \, \Xi$ vector.



Operations

- Volumes are defined by operations
- Operations combine operand volumes to produce result volumes
- Dataflow in an operation is described by a volume element map:

  result elements → operand elements

- Shape inference through implied context

Definition 4.10 (Operation). An *operation* X is a side-effect free function producing M result values from N operand values

$$X(O_1, \ldots, O_N) \mapsto R_1, \ldots, R_M$$

Volume element map:

$$\mathcal{M}_X : \bigcup_j \bigcup_k \{R_j \to O_k\}$$

Implied context:

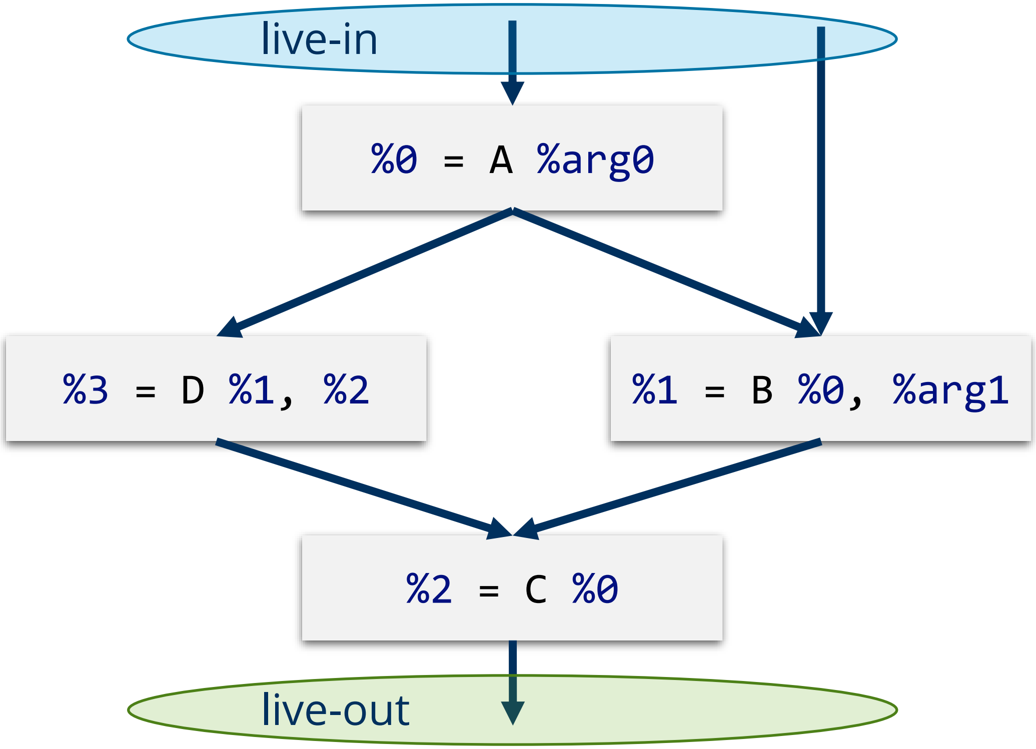
$$\mathcal{M}_X \subseteq \bigcup_j \operatorname{dom} R_j \times \bigcup_k \operatorname{dom} O_k$$

Programs

- Side-effect free SSA basic blocks
- Volume dataflow through assignments (ops) is acyclic def-use graph
- Element dataflow is obtained via volume element maps

$$\mathcal{M}_P = \left(\bigcup_{A \in P} \mathcal{M}_A\right)^+ \cap \{o \mapsto i : o \in O, i \in I\}$$

Definition 4.14 (Program). A *program P* is an unordered sequence of assignments, plus a set of incoming definitions and exiting uses. It is well-formed iff the graph associating each use with its unique definition is acyclic.



Lenient composition

- Transitive hull can be avoided

$$R_{A,i} \leftarrow A(R_{B,j}, \ldots, O_{A,k})$$

$$\mathcal{M}_{A\circ B} := \{R_{A,i} \mapsto O_{B,j}\} \cup \{R_{A,s} \mapsto O_{A,t}\}$$

1. Start with empty subgraph, initialize VEM with identity on live-out
2. Visit every assignment once in <u>any topological use-def order</u>
   a) Add assignment to subgraph
   b) Update VEM via lenient composition

Definition 4.15 (Lenient composition). The *lenient composition* $A \circ^{\mathrm{id}} B$ extends composition of unions of affine maps

$$A \circ^{\mathrm{id}} B = A \circ (B \cup \mathrm{id}(\mathrm{range}\, A \setminus \mathrm{dom}\, B))$$

where $\mathrm{id}\, X$ is the identity over X.

Let $G_0 = \emptyset$ and $\mathcal{M}_{G_0} = \mathrm{id}\, O$

$$\nexists A_j \in P : A_j \notin G_i, \mathrm{range}\, \mathcal{M}_{A_j} \cap \mathrm{dom}\, \mathcal{M}_{A_{i+1}} \neq \emptyset$$

$$G_{i+1} = G_i \cup \{A_{i+1}\}$$

$$\mathcal{M}_{G_{i+1}} = \mathcal{M}_{G_i} \circ^{\mathrm{id}} \mathcal{M}_{A_{i+1}}$$

Outline

- Introduction
- Volume-based dataflow analysis
- Implementation & examples
- Conclusions

Example kernel

- Simple program involving padding, convolution & activation operators
- MLIR operations define VEM

$$\mathcal{M}_{\mathrm{krnl0}} = \mathcal{M}_3 \circ^{\mathrm{id}} \mathcal{M}_2 \circ^{\mathrm{id}} \mathcal{M}_1$$

- MLIR types define the volumes

$$\mathrm{dom}\,\mathtt{act0} := \{\mathtt{act0}[\mathbf{i}] : \mathbf{0} \leq \mathbf{i} < [1, 8, 256, 256]\}$$

$$\mathrm{dom}\,\mathtt{ifm} := \{\mathtt{ifm}[\mathbf{i}] : \mathbf{0} \leq \mathbf{i} < [1, 3, 512, 512]\}$$

```
^krnl0(%ifm: tensor<1x3x512x512xf32>):
(1)%pad0 = tensor.pad %ifm low[0,0,1,1] high
    ↪ [0,0,2,2] {
  ^bb0(%i0: index, %i1: index, %i2: index, %i3:
      ↪ index):
    tensor.yield %cst0 : f32
  } : tensor<1x3x512x512xf32> to tensor<1
    ↪ x3x515x515xf32>
(2)%conv0 = linalg.conv_2d_nchw_fchw {
    dilations = dense<1> : tensor<2xi64>,
    strides = dense<2> : tensor<2xi64>
  }
  ins(%pad0, %wgt0: tensor<1x3x515x515xf32>,
      ↪ tensor<8x3x5x5xf32>)
  outs(%conv0.init: tensor<1x8x256x256xf32>)
  -> tensor<1x8x256x256xf32>
(3)%act0 = linalg.generic #elementwise_traits
    ins(%conv0: tensor<1x8x256x256xf32>)
    outs(%act0.init: tensor<1x8x256x256xf32>) {
  ^bb0(%a: f32, %b: f32):
    %0 = arith.maxf %a, %cst0 : f32
    %1 = arith.mulf %0, %cst0_01 : f32
    %2 = arith.addf %a, %1 : f32
    linalg.yield %2 : f32
  } -> tensor<1x8x256x256xf32>
```

Padding

- Semantically sound for convex polyhedra
- `tensor.pad` only deals with hyperrectangular volumes
- Other relevant variants exist
  - Periodic boundary conditions
  - Generative boundary
  - etc.

A.2.6 Padding. Adding a boundary around an array A using values from a volume B, selected based on index, is called *padding*

$$\mathcal{M}_{\text{pad},\mathbf{lo},\mathbf{hi},\Pi} := R[\mathbf{r}] \mapsto \begin{cases} A[\mathbf{r}-\mathbf{lo}] & \mathbf{lo} \le \mathbf{r} < (\Delta_\square R - \mathbf{hi}) \\ B[\Pi(\mathbf{r})] & \text{otherwise} \end{cases}$$

where **lo** and **hi** indicate the size of the boundary in all dimensions, $\Pi : \operatorname{dom} R \longrightarrow \operatorname{dom} B$ and $\Delta_\square R = \Delta_\square A + \mathbf{hi} + \mathbf{lo}$.

$$\begin{aligned} \mathcal{M}_1 := \{\, & \texttt{pad0}[n,c,h,w] \mapsto \texttt{ifm}[n,c,h-1,w-1] \\ & : 1 \le h < 513 \land 1 \le w < 513\} \\ \cup \{\, & \texttt{pad0}[n,c,h,w] \mapsto \texttt{cst0} \\ & : h < 1 \lor w < 1 \lor h \ge 513 \lor w \ge 513\} \end{aligned}$$

MLIR `linalg.generic`

- Implicit perfect loop nest
- Implicit iteration domain (`ShapesToLoops`)
- Inter-iteration dependencies
  - None (`parallel`)
  - Sequential (`reduction`)
- Point-wise expression body

Access relations

```
#elementwise_traits = {
  indexing_maps = [
    affine_map<(d0,d1,d2,d3) -> (d0,d1,d2,d3)>,
    affine_map<(d0,d1,d2,d3) -> (d0,d1,d2,d3)>
  ],
  iterator_types = [
    "parallel", "parallel", "parallel", "parallel"
  ]
}
```

Inter-iteration dependencies

```
%act0 = linalg.generic #elementwise_traits
    ins(%conv0: tensor<1x8x256x256xf32>)
    outs(%act0.init: tensor<1x8x256x256xf32>) {
  ^bb0(%a: f32, %b: f32):
    %0 = arith.maxf %a, %cst0 : f32
    %1 = arith.mulf %0, %cst0_01 : f32
    %2 = arith.addf %a, %1 : f32
    linalg.yield %2 : f32
  } -> tensor<1x8x256x256xf32>
```

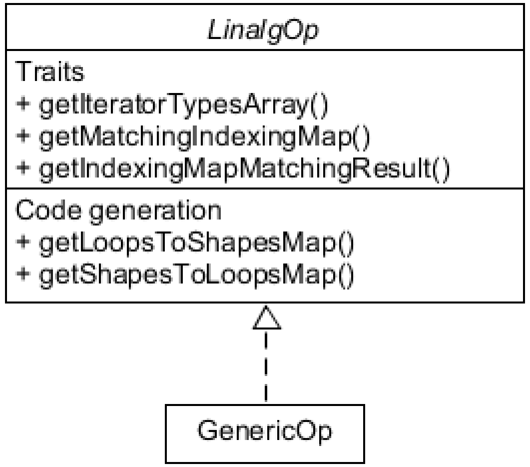
MLIR `linalg::LinalgOp`

- All structured ops implement `LinalgOp`
- Provides access to traits & operands

→ Compute the VEM directly

- ATM, built-in `ShapesToLoops` (domain) can only deal with permuted identities

```
%R0, ..., %RM = "linalg.op" (%O0, ..., %ON)
```



Iteration domain:

$$\operatorname{dom} X = \bigcap_j \operatorname{dom}\left(\mathcal{I}_{R_j} \underset{\mathrm{rg}}{\cap} \operatorname{dom} R_j\right)$$

Volume element map:

$$\mathcal{M}_X := \bigcup_j \bigcup_k \left(\left(\mathcal{I}_{R_j}^{-1} \underset{\mathrm{rg}}{\cap} \operatorname{dom} X\right) \circ \mathcal{I}_{O_k}\right)$$

VEM from MLIR

- Single-pass over operation DAG
- VEM produced by interface

$$\mathcal{M}_2 := \{\ \texttt{conv0}[n, f, y, x] \mapsto \texttt{pad0}[n, c, 2y + a, 2x + b] : 0 \leq a < 5 \wedge 0 \leq b < 5\} \cup \{\texttt{conv0}[n, f, y, x] \mapsto \texttt{wgt}[f, c, a, b]\}$$

Pitfalls:

- Overapproximation of non-isolated uses
- Selection of subgraph (live-ins)
- Loss of 1:1 statement correspondence

```
{ act0[n, f, h, w] -> ifm[n, c, y, x]
    : y >= 2h - 1 and 0 <= y <= 511 and y <= 3 + 2h
      and x >= 2w - 1 and 0 <= x <= 511
      and x <= 3 + 2w;
  act0[n, f, h, w] -> cst0[]
    : h >= 255 or w >= 255 or h <= 0 or w <= 0 }
```

Fact-based partitioning

- Inspect partitionings of the index domain

- In MLIR
  - Side-effect free & point-wise
  - Hyperrectangular only

  → Disjoint slices
  → Cloning

$$I_c = I_{\mathtt{Sp1}} \cup I_{\mathtt{Sp2}}$$
$$= \{\mathtt{act0}[n, f, h, w] : h \geq 255 \lor w \geq 255 \lor h \leq 0 \lor w \leq 0\}$$

$$I_{\mathrm{D}} = \operatorname{dom} \mathtt{act0} \setminus I_c$$
$$= \{\mathtt{act0}[n, f, h, w] : 0 < h \leq 254 \land 0 < w \leq 254\}$$

Algorithm 1: Generating code for hyperrectangular partitions.

```
%res′ ← Uninitialized(dom %res);
foreach disjoint hyperrect out_rect in I_x do
    in_rects ← range(M ∩_dom out_rect)^2;
    %op:N′ ← ExtractSlice(%op:N,in_rects);
    %part ← CloneOps(%op:N′);
    %res′ ← InsertSlice(%part,%res′,out_rect);
end
return %res′;
```

Optimistic tiling

- Tile volumes not loops
- Expecting hyperrectangular slices

- Existentially-quantified variables → redution → partial reduction

Optimistic slice matching:

- Fix tile indices as parameters
- Recursively determine offset & stride affine expressions

$$\mathcal{M}_T := \{\ \texttt{act1}[n, f, q_h, m_h, q_w, m_w] \mapsto \texttt{act0}[n, f, T_h q_h + m_h, T_w q_w + m_w] : 0 \leq m_h < T_h \wedge 0 \leq m_w < T_w\}$$

Algorithm 2: Generating tiled loops.

$\mathcal{M}' \leftarrow$ tile_map $\circ\ \mathcal{M}$;
%res′ ← Uninitialized(dom %res);
foreach *tile dim pair* q_i, m_i *in* tile_map **do**
  iv ← CreateAndEnterLoop(min Q_i **to** max Q_i);
  iv_dom ← $[Q_i = \text{iv}] \rightarrow \{[\ldots, q_i = Q_i, \ldots]\}$;
  $\mathcal{M}' \leftarrow \mathcal{M}' \underset{\text{dom}}{\cap}$ iv_dom;
end
in_rects ← MatchSlice(range $\mathcal{M}'$);
out_rect ← MatchSlice(dom $\mathcal{M}'$);
%op:N′ ← ExtractSlice(%op:N,in_rects);
%tile ← CloneOps(%op:N′);
%res′ ← InsertSlice(%tile,%res′,out_rect);
return %res′;

Conclusions & future work

- ✓ High-level decision making
  - ✓ Identify known subproblems
  - ✓ Satisfy fixed library / hardware constraints
  - ✓ Bridge immutable and bufferized gap
- ✓ Starting point for constrained fine-grained code generation

- Subgraph choice problem & associated overestimations
- Reduction indices & partial reductions
- More general affine matching
- Baseline implementation

TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR FOR COMPILER CONSTRUCTION

DRESDEN concept