Indexing Analysis in XLA:GPU

Indexing Map

AffineMap + Domain metadata

```
(d0)[s0, s1] -> (s1 mod 3, 2 * d0, s1, s0),
domain:
d0 in [0, 9],
s0 in [0, 69],
s1 in [0, 19],
d0 + s1 in [0, 20],
d0 mod 8 in [0, 0],
s0 mod 3 in [1, 1],
is_simplified: true
```

ranges of variables

constraints

Dimension Variables

Tensor indices or IDs of GPU threads/blocks

Broadcast tensor<20xf32> -> tensor<10x20x30xf32>

```
(d0, d1, d2) -> (d1)
domain:
d0 in [0, 9]
d1 in [0, 19]
d2 in [0, 29]
```

Range Variables

Slices of the input, loops.

Reduce tensor<256x10xf32> -> tensor<10xf32>

```
(d0)[s0] -> (s0, d0)
domain:
d0 in [0, 9]
s0 in [0, 255]
```

For every fixed d0 we need 256 values of s0.

Runtime Variables

Dynamic IndexType parameters.

dynamic_slice %src[%i] [16] [1] : tensor<64xf32> to tensor<16xf32>

```
(d0)[s0] -> (d0 + s0)
domain:
d0 in [0, 15]
s0 in [0, 47]
  hlo: %i = s32[] parameter(1)
  (d0)  -> ()
```

Ranges

For the given ranges of the variables, the map is valid.

concat(tensor<2x5xf32>, tensor<2x11xf32>) -> tensor<2x16xf32>

```
operand 0: (d0, d1) -> (d0, d1)
domain:
d0 in [0, 1]
d1 in [0, 4]

operand 1: (d0, d1) -> (d0, d1)
domain:
d0 in [0, 1]
d1 in [5, 15]
```

Constraints

When the constraints are satisfied, the map is valid.

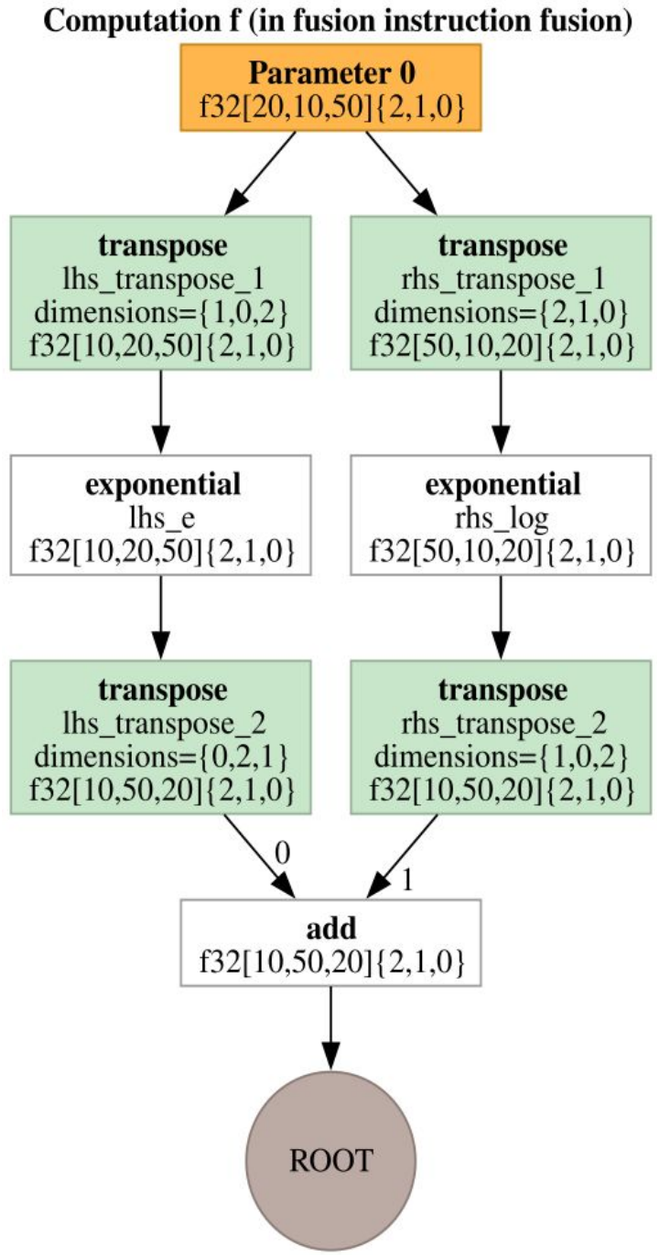
pad tensor<4x4xf32> -> tensor<12x16xf32>
low [1, 4], high [4, 8], interior [1, 0]

```
(d0, d1) -> ((d0 - 1) floordiv 2, d1 - 4)
domain:
d0 in [1, 7]
d1 in [4, 7]
(d0 - 1) mod 2 in [0, 0]
```

Composition

Deduplication of indexing

```
(d0, d1, d2) -> (d2, d0, d1)
domain:
d0 in [0, 9]
d1 in [0, 49]
d2 in [0, 19]
```



Simplification

Use domain info to simplify mods and divs

```
(d0, d1) -> (-((d0 * -11 - d1 + 109) floordiv 11) + 9,
   d0 * 11 + d1 + ((d0 * -11 - d1 + 109) floordiv 11) * 11 - 99)
```

```
domain:
d0 in [0, 7]
d1 in [0, 8]
```

after simplification becomes

```
(d0, d1) -> (d0, d1)
```

Folding of RTVars

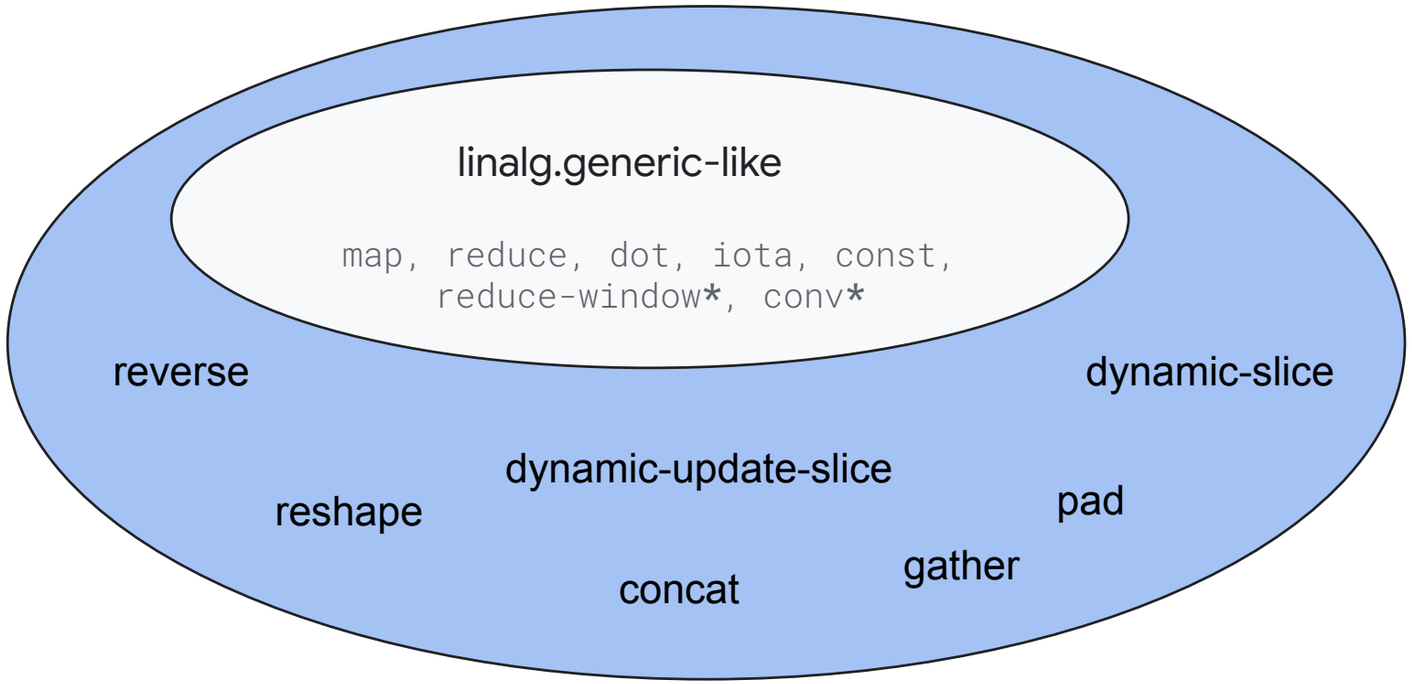
Indexing map with a runtime variable

```
(d0)[s0] -> (d0, s0)
domain:
d0 in [0, 11]
s0 in [0, 47]
  hlo: %add = s64[12,13,24] add(s64[12,13,24] %broadcast,
                                s64[12,13,24] %iota)
  (d0) -> (d0, 7, 2 * d0)
```

after folding

```
(d0) -> (d0, d0 * 2 + 42),
domain:
d0 in [0, 11]
```

Supported Ops



* dilation and interior padding are also supported

Thread ID map (grid->tensor)

Map thread and block IDs to the fusion roots

Map for a GPU kernel for an cwise op on tensor<20x40x300xf32>

```
(th_x, th_y, th_z, bl_x, bl_y, bl_z)[vector_elem] -> (
  (bl_x * 128 + th_x) floordiv 3000,
  ((bl_x * 128 + th_x) floordiv 75) mod 40,
  ((bl_x * 128 + th_x) mod 75) * 4 + vector_elem
),
domain:
  th_x in [0, 127], th_y in [0, 0], th_z in [0, 0],
  bl_x in [0, 468], bl_y in [0, 0], bl_z in [0, 0],
  vector_elem in [0, 3],
  bl_x * 128 + th_x in [0, 59999],
  is_simplified: true
```

Coalescing analysis

Thread ID -> linearized physical layout of operands

For kernels generated with XLA:GPU Emitters we know the following indexing maps:

- **`Thread ID -> tensor`** `within the fusion`
- **`Output -> input`** `for the fused ops`
- **`Logical -> linearized physical layout`** `of operands`

We compose them to get **`Thread ID -> linearized physical layout`**.

Analyze the result to check if the reads are coalesced or not.

MLIR-based emitters

IndexingMapAttr & xla::gpu::ApplyIndexingOp

```
#map = #xla_gpu.indexing_map<(d0, d1, d2) -> (d0 * 4 + d1 * 512 + d2),
  domain:
  d0 in [0, 127],
  d1 in [0, 468],
  d2 in [0, 3],
  is_simplified: true
>


%index = xla_gpu.apply_indexing #map(%thread_id_x, %block_id_x, %c0)
```

MLIR-based emitters

Vectorization (before)

```
#map = #xla_gpu.map<(d0, d1, d2) -> (d0 * 4 + d1 * 512 + d2),
  domain: d0 in [0, 127], d1 in [0, 468], d2 in [0, 3]>

%loop = scf.for %arg2 = %c0 to %c4 step %c1 iter_args(%arg3 = %arg1)
    -> (tensor<240000xf32>) {
  %i = xla_gpu.apply_indexing #map(%thread_id_x, %block_id_x, %arg2)
  %extracted = tensor.extract %arg0[%i] : tensor<240000xf32>
  …
}
```

MLIR-based emitters

Vectorization (after)

```
#map = #xla_gpu.map<(d0, d1, d2) -> (d0 * 4 + d1 * 512 + d2),
  domain: d0 in [0, 127], d1 in [0, 468], d2 in [0, 3]>

%i = xla_gpu.apply_indexing #map(%thread_id_x, %block_id_x, %c0)
%vec = vector.transfer_read %arg0[%i], %cst {in_bounds = [true]}
  : tensor<240000xf32>, vector<4xf32>
%loop = scf.for %arg2 = %c0 to %c4 step %c1 iter_args(%arg3 = %arg1)
    -> (tensor<240000xf32>) {
  %extracted = vector.extract %vec[%i] : vector<4xf32>
  …
}
```

MLIR-based emitters

xla::gpu::LoopOp

```
%loop = xla_gpu.loop
  (%th_x,%th_y,%th_z,%bl_x,%bl_y,%bl_z)[%i, %j]
   -> (%ra, %rb, %rc) in #indexing_map
   iter_args(%iter = %out) -> (tensor<20x40x300xf32>) {
  %elem = <compute_element_of the input>
  %inserted = tensor.insert %elem into %iter[%ra, %rb, %rc]
    : tensor<20x40x300xf32>
  xla_gpu.yield %inserted : tensor<20x40x300xf32>
}
```

Equivalent to an `scf.for` loop nest with an `scf.if` constraint check.
Peeling == splitting the domain to simplify constraints.

MLIR-based emitters

Elemental emitters (HLO->Loops)

Dot, **Pad**, **Reduce**, **ReduceWindow**, **Conv** are converted to loop nests based on the indexing maps.

Block-level tiling

Symbolically compute expressions for offsets-sizes-strides for a fusion.

Compute constraints that allow to tiling propagation through reshapes, concats.

Use it for lowering to Triton and tile sizes tuning.

Future plans

- Support stable_hlo ops
- Compute utilization in cost model
- Stop using AffineMap to improve compile time
- More metadata for variables, e.g. variable type to make the maps more readable
- Debugging tools, e.g. visualization of access patterns for every warp/thread/block

Links

- Indexing Map
- Indexing Analysis
- Symbolic Tiling
- MLIR-based emitters (reduction, scatter, ...)