Coordinate Transformations in AMD's rocMLIR

Krzysztof Drewniak[1]

Advanced Micro Devices

Oct 06, 2022

$$(\mathrm{d}_0, \mathrm{d}_1, \mathrm{d}_2) \to (\mathrm{d}_0 \,/\, 9, \mathrm{d}_2 + 2\mathrm{d}_1, (\mathrm{d}_0 \,\%\, 9) \,/\, 3, \mathrm{d}_0 \,\%\, 3)$$

$$\mathrm{d}_0 \leftarrow \mathrm{Merge}\{9, 3, 1\}(\mathrm{o}_0, \mathrm{o}_2, \mathrm{o}_3)$$

$$\mathrm{d}_1, \mathrm{d}_2 \leftarrow \mathrm{Embed}\{2, 1\}(\mathrm{o}_1)$$

[1]Krzysztof.Drewniak@amd.com

Pop quiz

What's the difference between these two maps?

$$f = (\mathrm{d}_0, \mathrm{d}_1) \to (\mathrm{d}_0, \mathrm{d}_1)$$
$$g = (\mathrm{d}_0, \mathrm{d}_1) \to (\mathrm{d}_0, \mathrm{d}_1)$$

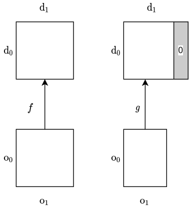
Pop quiz

What's the difference between these two maps?

$$f = (\mathrm{d}_0, \mathrm{d}_1) \to (\mathrm{d}_0, \mathrm{d}_1)$$
$$g = (\mathrm{d}_0, \mathrm{d}_1) \to (\mathrm{d}_0, \mathrm{d}_1)$$

What if I told you?

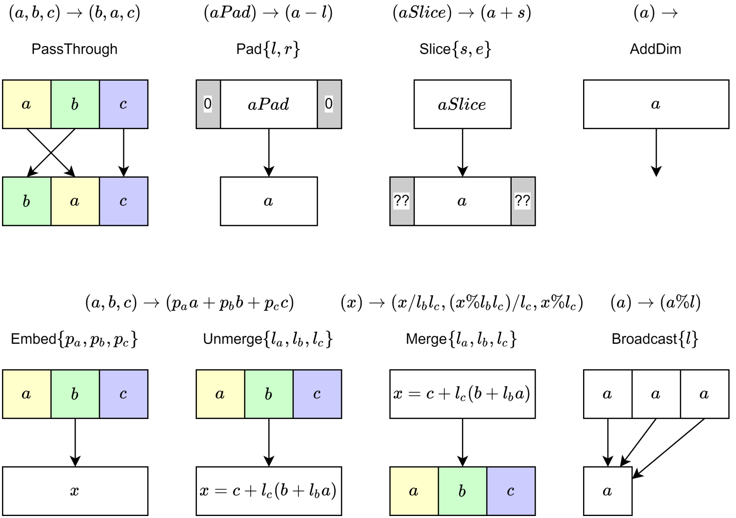
$$f = [\mathrm{d}_0 \leftarrow \mathrm{o}_0, \mathrm{d}_1 \leftarrow \mathrm{o}_1]$$
$$g = [\mathrm{d}_0 \leftarrow \mathrm{o}_0, \mathrm{d}_1 \leftarrow \mathrm{Pad}\{0, 1\}(\mathrm{o}_1)]$$



What and why

- `transform_map` — affine maps + metadata
  - Can represent implicit padding (no clear upstream analogue)
  - Conversely, restricts set of available expressions
  - Comes from declarative builder
- Improves reasoning about maps (ex. more precise bounds checking)
- Index diffs — more efficient loop unrolling
  - The `transforming_for` loop
- System arose from other AMD code + less upstream infrastructure early in development

Coordinate transformations



Building a transform_map

To get the map

$$(\mathrm{d}_0, \mathrm{d}_1) \to (\mathrm{d}_0, \mathrm{d}_1 \,/\, 9, (\mathrm{d}_1 \,\%\, 9) \,/\, 3, \mathrm{d}_1 \,\%\, 3)$$
$$M@\mathrm{d}_0 \leftarrow \mathrm{PassThrough}(O@\mathrm{o}_0)$$
$$K@\mathrm{d}_1 \leftarrow \mathrm{Merge}\{2, 3, 3\}(I@\mathrm{o}_1, H@\mathrm{o}_2, W@\mathrm{o}_3)$$

Do

\# Output space is $O \times I \times H \times W = 128 \times 2 \times 3 \times 3$

```
BottomUpBuilder b(
  rw, {"O", "I", "H", "W"}, {128, 2, 3, 3}, loc);
```

\# $M@\mathrm{d}_0 \leftarrow O@\mathrm{o}_0$

```
b.passThrough("M", 0, "O");
```

\# $K@\mathrm{d}_1 \leftarrow \mathrm{Merge}\{2, 3, 3\}(I@\mathrm{o}_1, H@\mathrm{o}_2, W@\mathrm{o}_3)$

```
b.merge("K", 1, {"I", "H", "W"});)
b.get();
```

Bounds checks

- Read/write $T[x_1, \ldots, x_n]$ with $\mathbf{x} = f(\mathbf{t})$
- Some $\mathbf{x}$ are invalid for in-bounds $\mathbf{t}$
  - SIMD size is 64, but matrix size is 128×18
  - Implicit padding of input tensor
  - Hardware load with bound check instead of if
- Can we avoid always testing $0 \leq x_i < \mathrm{size}(i)$?
- Hard to determine from general affine maps (ex. right side padding looks like pass through)

Bounds checks with `transform_maps`

If $f = (f_1 \circ f_2 \circ f_l)$ is a composition of `transform_maps`, have rules for when to check bounds.

Example

Pad K dimension of matrix for SIMD

$$f_1 = [M@\mathrm{d}_0 \leftarrow M@\mathrm{o}_0, K_p@\mathrm{d}_1 \leftarrow \mathrm{Pad}\{0, 64 - 18\}(K@\mathrm{o}_1)]$$

OIHW filter tensor as matrix

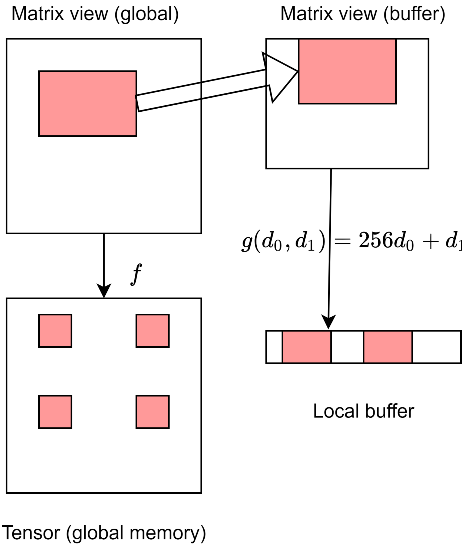
$$f_2 = [M@\mathrm{d}_0 \leftarrow O@\mathrm{o}_0, K@\mathrm{d}_1 \leftarrow \mathrm{Merge}\{2, 3, 3\}(I@\mathrm{o}_1, H@\mathrm{o}_2, W@\mathrm{o}_3)]$$

We know

- No need to check O
- K (f_1 output, unpadded K_p) needs bounds check on the right
- If that overflows, I is too large, H and W in bounds (modulo)
- Only need to check I dimension, and only on the right

transforming_for

`transforming_for`: The context



Note: coordinate transforms are from before `linalg.generic`

transforming_for: The context in code

```
for %d0 = 0 to 4 by 1 {
  for %d1 = 0 to 8 by 1 {
    %global0 = add %s0, %d0
    %global1 = add %s1, %d1
    %buffer0 = add 0, %d0
    %buffer1 = add 8, %d1
    %arg1, %arg2, %arg3 = f(%global0, %global1)
    %tmp0 = mul 256, %buffer0
    %arg4 = add %tmp0, %buffer1
    %0 = load %T[%arg1, %arg2, %arg3]
    store %0 -> %buffer[%arg4]
  }
}
```

transforming_for

```
transforming_for
  (%arg1, %arg2, %arg3) = [#tmap1, #tmap2](%s0, %s1)
  (%arg4) = [Embed{256, 1}](0, 8) # 256 * d0 + d1
  bounds [4, 8] strides [1, 1] {
    %0 = load %T[%arg1, %arg2, %arg3]
    store %0 -> %buffer[%arg4]
}
```

Loads from:	Stores to:
#tmap2(#tmap1(%s0, %s1))	%buffer[8]
...	...
#tmap2(#tmap1(%s0 + 0, %s1 + 7))	%buffer[15]
#tmap2(#tmap1(%s0 + 1, %s1 + 0))	%buffer[256 + 8]
...	...
#tmap2(#tmap1(%s0 + 3, %s1 + 7))	%buffer[768 + 15]

`transforming_for`: Index diffs — why?

Suppose we unrolled our loop

```
%args_0_0:3 = #tmap2(#tmap1(%s0, %s1))
%arg4_0_0 = 8
...
%args_0_1:3 = #tmap2(#tmap1(%s0, %s1 + 1))
%arg4_0_1 = 9
...
```

- All of those maps need to be recomputed
- Generic reasoning not always enough

transforming_for: Index diffs — why?

Suppose we unrolled our loop

```
%args_0_0:3 = #tmap2(#tmap1(%s0, %s1))
%arg4_0_0 = 8
...
%args_0_1:3 = #tmap2(#tmap1(%s0, %s1 + 1))
%arg4_0_1 = 9
...
```

- All of those maps need to be recomputed
- Generic reasoning not always enough

What if?

```
%args_0_0:3 = #tmap2(#tmap1(%s0, %s1))
%arg4_0_0 = 8
...
%args_0_1:3 = u(%args_0_0:3, (0, 1))
%arg4_0_1 = %arg4_0_0 + 256 * 0 + 1 = 9
...
```

`transforming_for`: Index diffs

- Problem: given $\mathbf{x} = g(\mathbf{t})$, we want $\mathbf{x}' = g(\mathbf{t} + \delta)$
- With transforms, can get: $\mathbf{x}' = u_g(\mathbf{x}, \delta)$

Example

$$
\begin{aligned}
g &= (\mathrm{d}_0, \mathrm{d}_1) \to (256\mathrm{d}_0 + \mathrm{d}_1) \\
&= [\mathrm{d}_0, \mathrm{d}_1 \leftarrow \mathrm{Embed}\{256, 1\}(\mathrm{o}_0)] \\
u_g(\mathbf{x}, \delta) &= x_0 + 256\delta_0 + \delta_1
\end{aligned}
$$

- Removes repetitive recomputations when unrolling
- Often improves performance

Summary

- Coordinate transformations: extra data about maps
- Incorporate info not available in affine, mainly padding
- Restrict available maps, enabling more precise reasoning (ex. bounds check elimination)
- More efficient loop unrolling — index diffs and `transforming_for`
- Most parts can be done with current MLIR core, but not all

Questions?

Bonus slides

transform_map syntax

```
#transform_map1 = #rock.transform_map<
  affine_map<(d0, d1, d2) ->
    (d0, d2, d1 floordiv 9, (d1 mod 9) floordiv 3, d1 mod 3)>
  by [<PassThrough ["gemmG"] at [0] -> ["g"] at [0]>,
      <Merge{2, 3, 3} ["gemmK"] at [1]
        -> ["c", "y", "x"] at [2, 3, 4]>,
      <PassThrough ["gemmM"] at [2] -> ["k"] at [1]>]
  bounds = [1, 18, 128] -> [1, 128, 2, 3, 3]>
```

transforming_for syntax

```
%17 = rock.transforming_for {forceUnroll, useIndexDiffs}
    (%arg3, %arg4, %arg5, %arg6, %arg7) =
      [#transform_map0, #transform_map1](%2, %8, %11),
    (%arg8) = [#transform_map2](%c0, %c0, %c0)
    iter_args (%arg9 = %cst_1 : vector<2xf32>)
    bounds [1, 1, 2] strides [1, 1, 1] {
  %28 = rock.buffer_load
    %arg0[%arg3, %arg4, %arg5, %arg6, %arg7]
    {leftOobDims = [], rightOobDims = [2 : i32]}
    : memref<1x128x2x3x3xf32>,
    index, index, index, index, index -> f32
  %29 = vector.insertelement %28, %arg9[%arg8] : vector<2xf32>
  rock.yield %29 : vector<2xf32>
}
```