Coordinate Transformations in AMD’s rocMLIR

Krzysztof Drewniak

Advanced Micro Devices

Oct 06, 2022

\[(d_0, d_1, d_2) \rightarrow (d_0 / 9, d_2 + 2d_1, (d_0 \mod 9) / 3, d_0 \mod 3)\]

\[d_0 \leftarrow \text{Merge}\{9, 3, 1\}(o_0, o_2, o_3)\]

\[d_1, d_2 \leftarrow \text{Embed}\{2, 1\}(o_1)\]

\(^1\text{Krzysztof.Drewniak@amd.com}\)
Pop quiz

What’s the difference between these two maps?

\[
f = (d_0, d_1) \rightarrow (d_0, d_1)
\]

\[
g = (d_0, d_1) \rightarrow (d_0, d_1)
\]
Pop quiz

What’s the difference between these two maps?

\[ f = (d_0, d_1) \mapsto (d_0, d_1) \]
\[ g = (d_0, d_1) \mapsto (d_0, d_1) \]

What if I told you?

\[ f = [d_0 \leftarrow o_0, d_1 \leftarrow o_1] \]
\[ g = [d_0 \leftarrow o_0, d_1 \leftarrow \text{Pad}\{0, 1\}(o_1)] \]
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

\[(a, b, c) \rightarrow (b, a, c)\]
PassThrough

\[(aPad) \rightarrow (a - l)\]
Pad\{l, r\}

\[(aSlice) \rightarrow (a + s)\]
Slice\{s, e\}

\[(a) \rightarrow\]
AddDim

\[(a, b, c) \rightarrow (p_a a + p_b b + p_c c)\]
Embed\{p_a, p_b, p_c\}

\[(x) \rightarrow (x / l_c(a), (x % l_c(b))/l_c, x % l_c)\]
Unmerge\{l_a, l_b, l_c\}

\[(a) \rightarrow (a % l)\]
Merge\{l_a, l_b, l_c\}

\[x = c + l_c(b + l_b a)\]
Broadcast\{l\}
To get the map

$$(d_0, d_1) \rightarrow (d_0, d_1 / 9, (d_1 \% 9) / 3, d_1 \% 3)$$

$M@d_0 \leftarrow \text{PassThrough}(O@o_0)$

$K@d_1 \leftarrow \text{Merge}\{2, 3, 3\}(I@o_1, H@o_2, W@o_3)$

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);

# $M@d_0 \leftarrow O@o_0$

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

# $K@d_1 \leftarrow \text{Merge}\{2, 3, 3\}(I@o_1, H@o_2, W@o_3)$

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

b.get();
Bounds checks

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

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

Example

Pad $K$ dimension of matrix for SIMD

$$f_1 = [M@d_0 \leftarrow M@o_0, K_P@d_1 \leftarrow \text{Pad}\{0, 64 - 18\}(K@o_1)]$$

OIHW filter tensor as matrix

$$f_2 = [M@d_0 \leftarrow O@o_0, K@d_1 \leftarrow \text{Merge}\{2, 3, 3\}(I@o_1, H@o_2, W@o_3)]$$

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

```assembly
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:
#tmap2(#tmap1(%s0, %s1))
...
#tmap2(#tmap1(%s0 + 0, %s1 + 7))
#tmap2(#tmap1(%s0 + 1, %s1 + 0))
...
#tmap2(#tmap1(%s0 + 3, %s1 + 7))

Stores to:
%buffer[8]
...
%buffer[15]
%buffer[256 + 8]
...
%buffer[768 + 15]
Suppose we unrolled our loop

\[
%\text{args}_0:3 = \text{tmap2}(\text{tmap1}(s0, s1))
\]

%arg4_0 = 8

... 

\[
%\text{args}_1:3 = \text{tmap2}(\text{tmap1}(s0, s1 + 1))
\]

%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

\[%\text{args}_0:3 = \text{tmap2}(\text{tmap1}(\%s0, \%s1))\%
\]

\[%\text{arg4}_0:0 = 8\%
\]

\[\ldots\]

\[%\text{args}_0:1:3 = \text{tmap2}(\text{tmap1}(\%s0, \%s1 + 1))\%
\]

\[%\text{arg4}_0:1 = 9\%
\]

\[\ldots\]

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

What if?

\[%\text{args}_0:0:3 = \text{tmap2}(\text{tmap1}(\%s0, \%s1))\%
\]

\[%\text{arg4}_0:0 = 8\%
\]

\[\ldots\]

\[%\text{args}_0:1:3 = u(\%\text{args}_0:0:3, (0, 1))\%
\]

\[%\text{arg4}_0:1 = \%\text{arg4}_0:0 + 256 \times 0 + 1 = 9\%
\]

\[\ldots\]
Problem: given $x = g(t)$, we want $x' = g(t + \delta)$

With transforms, can get: $x' = u_g(x, \delta)$

Example

$$g = (d_0, d_1) \rightarrow (256d_0 + d_1)$$

$$= [d_0, d_1 \leftarrow \text{Embed\{256, 1\}(o_0)]]$$

$$u_g(x, \delta) = x_0 + 256\delta_0 + \delta_1$$

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_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],
    bounds = [1, 18, 128] -> [1, 128, 2, 3, 3]>

16 / 17
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>
}