$f(x) = m * x + c$

(Quasi-)Affine transformations for array access

Lennart Augustsson
Google Research
May 2019

Warning!

This talk contains nothing new.

Affine array operations

- Some array operations are purely structural, i.e., no computation involved
  - E.g., transpose, reshape, reverse, rotate
- A nice subclass of those are the *affine* ones, i.e., those that can be described by an affine transformation on the index.
  - E.g., transpose, reverse
- An affine transformation can be described by

  $$f(x) = m * x + o$$

  where x and o are vectors, and m is a matrix.
- The arrays considered here are APL style arrays (cf. Remora).
- Affine transformation is very popular in the polyhedral compilation community.

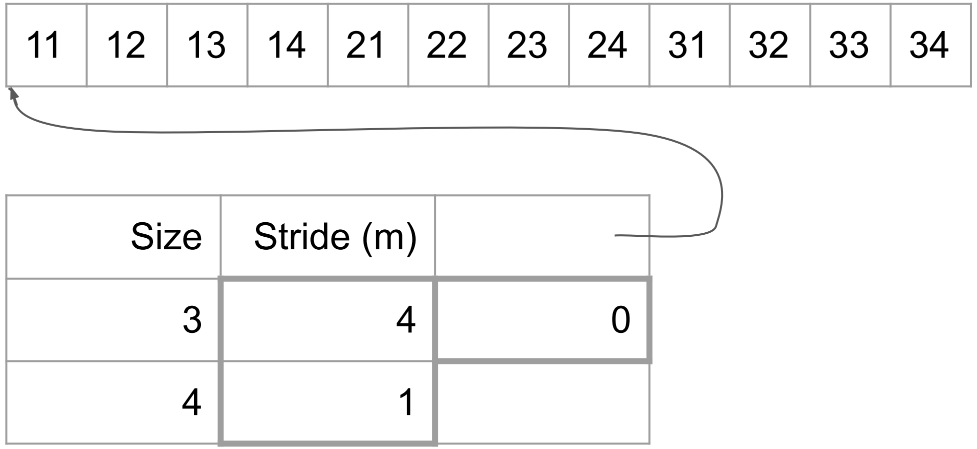
Dope vectors

A dope vector (an old idea) describes the size and strides of an array.

Example array:

11	12	13	14
21	22	23	24
31	32	33	34

Memory layout:



11	12	13	14	21	22	23	24	31	32	33	34

Dope vector:

Size	Stride (m)	
3	4	0
4	1	

Indexing with dope vectors

The index into a 2-dimensional array is a vector of length 2.

This is actually an instance of an affine transformation from 2-vectors to 1-vectors.

$$ix(i) = \begin{array}{|c|c|}\hline 4 & 1 \\ \hline\end{array} * i + \begin{array}{|c|}\hline 0 \\ \hline\end{array}$$

Example: $i = \begin{array}{|c|}\hline 1 \\ \hline 3 \\ \hline\end{array}$

ix(i) = 4*1 + 1 * 3 + 0 = 7

Example, transposition with dope vectors

Transposition of a 2-dimensional array just swaps the indices.

Size	Stride	
3	4	0
4	1	

becomes

Size	Stride	
4	1	0
3	4	

This is also an affine transformation

$$\text{transpose}(i) = \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & 0 \\ \hline\end{array} * i$$

A change of notation

Instead of using n-dimensional arrays we will use a functional notation

Size	Stride	
3	4	0
4	1	

```
ix :: A(3, 4) -> A(12)
ix (i0, i1) = (i0 * 4 + i1 * 1 + 0)

transpose :: A(4, 3) -> A(3, 4)
transpose (i0, i1) = (i1, i0)

f :: A(4, 3) -> A(12)
f = ix • transpose = (i1 * 4 + i0 * 1 + 0)
```

A change of notation

$f :: A(i_1, \dots i_n) \text{ -> } A(o_1, \dots o_m)$

$f(i_1, \dots i_n) = (e_1, \dots e_m)$

e ::= i * k + … + k
  where k is an integer

Example:

```
bcast :: A(3, 2) -> A(2)
bcast (i0, i1) = (i1)
```

5	9

appears as

5	9
5	9
5	9

Some properties

These functions, just like the affine transformations, have some nice properties:

- The functions are closed under composition
- There is a very simple normal form
- They are very simple and efficient to use in practice (ignoring caching issues)

Useful functions

There is a surprising (to me) number of useful functions that are affine transformations. (In reading these, beware contravariance!)

index i :: A(s_2,…,s_n) -> A(s_1,s_2,…,s_n)
index i (i_2,…,i_n) = (i,i_2,…,i_n)

unflatten :: A(s_1,…,s_n) -> A(s_1*…*s_n)
unflatten (i_1,i_2,$i_{3,}$…,i_k) = ((i_1*s_2 + i_2)*s_3 … + i_k)

transpose p :: A(s_1,…,s_n) -> A(s_{p1},…,s_{pn})
transpose (i_1,…,i_k) = (i_{q1},i_{q2},…,i_{qk})
   where *p* is some permutation, and *q* its inverse

reverse j :: A(s_1,…,s_n) -> A(s_1,…,s_n)
reverse (i_1,…,i_n) = (…,s_j-1-i_j,…)

Useful functions

Take every t_i element of an array.

$\texttt{stride :: A}(s_1/t_1, \ldots, s_n/t_n) \texttt{ -> A}(s_1, \ldots, s_n)$

$\texttt{stride }(i_1, \ldots, i_n) = (i_1 * t_1, \ldots, i_n * t_n)$

Duplicate the values along some dimensions

$\texttt{broadcast b :: A}(s_1, \ldots, s_n) \texttt{ -> A}(r_1, \ldots, r_m)$

$\texttt{broadcast }(i_1, \ldots, i_n) = (j_1, \ldots, j_m)$

Where $r_1 \ldots r_m$ is a subset $s_1 \ldots s_n$ and $j_1 \ldots j_m$ is a subset of $i_1 \ldots i_n$, but keeping the order.

```
slice :: ...
```

Useful functions

Create windows of the *k* outermost dimensions (this smells of convolution).

$$
\begin{aligned}
&\texttt{window } [w_1, \dots, w_k] \; :: \\
&\qquad A(s_1, \dots, s_k, s_{k+1} \dots, s_n) \; \texttt{->} \\
&\qquad A(s_1 - w_1 + 1, \dots, s_k - w_k + 1, w_1, \dots, w_k, s_{k+1} \dots, s_n) \\
&\texttt{window } (i_1, \dots, i_k, j_1, \dots, j_k, i_{k+1}, \dots, i_n) = (i_1 + j_1, \dots, i_k + j_k, i_{k+1}, \dots, i_n)
\end{aligned}
$$

`window [3]`

a	b	c	d	e	f

=

a	b	c
b	c	d
c	d	e
d	e	f

Flies in the ointments

Anyone having used APL is familiar with reshape. It’s the operation that changes the dimensions of an array, but keeping the elements in their canonical order.

$$\texttt{reshape :: A(}s_1\texttt{,}\ldots\texttt{,}s_n\texttt{) -> A(}r_1\texttt{,}\ldots\texttt{,}r_m\texttt{)}$$

Where $s_1 * \ldots * s_n == r_1 * \ldots r_m$

reshape

1	2	3	4
5	6	7	8
9	10	11	12

=

1	2	3	4	5	6
7	8	9	10	11	12

There is no way to express this as an affine transformation.

Flies in the ointments

A simpler function which is inexpressible

$\texttt{flatten :: A(}s_1\texttt{*…*}s_n\texttt{) -> A(}s_1\texttt{, …, }s_n\texttt{)}$

Using this function we have

```
reshape = flatten • unflatten
```

Quasi-affine transformations

To cover functions like flatten we extend the expression language for affine transformations by also allowing *division* and *modulus* with a positive integer.

Properties:

- The functions are closed under composition.
- There is a normal form (no longer simple).
- They can be less efficient to use in practice.

Examples

Flatten turns into a series of div and mod.

```
flatten :: A(12) -> A(3,4)
flatten ($i_1$) = ($i_1$ / 4, $i_1$ % 4)

reshape :: A(2,6) -> A(3,4)
reshape ($i_1$,$i_2$) = (($i_1$*6+$i_2$) / 4, ($i_1$*6+$i_2$) % 4)
```

More functions

Replicate (or truncate) data along each dimension

repl :: $A(s_1,…,s_n)$ -> $A(r_1,…,r_n)$
repl $(i_1,…,i_n)$ = $(i_1 \% r_1, …, i_n \% r_n)$

rotate r j :: $A(s_1,…,s_n)$ -> $A(s_1,…,s_n)$
rotate $(i_1,…,i_n)$ = $(…,(s_j+r)\%s_j,…)$

And then...

Most array operations are about enumerating the array elements in some way (sequential, parallel, …). The affine transform and other concerns guides how to do this in an efficient way.

- Loop fusion
- Loop tiling
- Loop interchange
- ...

A real example

This code performs 2-d convolution of an image using a single matrix multiply, using a stride:

```
aconv2D :: forall sy sx y x ky kx c f a ty tx . (_) =>
  Arr [y, x, c] a ->
  Arr [ky, kx, c, f] a ->
  Arr [(y-ky+1) // sy, (x-kx+1) // sx, f] a
aconv2D i k =
  let i' :: Arr [ty * tx, ky * kx * c] a
      i' = imageP @sy @sx @ky @kx i
      k' :: Arr [ky * kx * c, f] a
      k' = reshape k
      r :: Arr [ty * tx, f] a
      r = matMul i' k'
      r' :: Arr [ty, tx, f] a
      r' = reshape r
  in  r'
```

A real example

Munging the image:

```
imageP ::
  forall sy sx ky kx y x c a ty tx .
  (ty ~ ((y-ky+1) // sy),
   tx ~ ((x-kx+1) // sx),
   _) =>
  Arr [y, x, c] a ->
  Arr [ty * tx, ky * kx * c] a
imageP i =
  let w :: Arr [y-ky+1, x-kx+1, ky, kx, c] a
      w = window @[ky, kx] i
      s :: Arr [ty, tx, ky, kx, c] a
      s = stride @[sy, sx, 1, 1, 1] w
      r :: Arr [ty * tx, ky * kx * c] a
      r = reshape s
  in  r
```

A real example

Looking at a particular example:

```
image :: Arr [289, 289, 3] Int  -- stored as a vector of size 250563

image’ :: Arr [20164, 147] Int
image’ = imageP @2 @2 @7 @7 image
```

The affine transform:

```
f :: (20164, 147) -> (250563)
f (d0, d1) =
  (d1 / 21 * 867 + (d1 + d0 * 147) / 20874 * 1734 +
   d0 % 142 * 6 + d1 / 3 % 7 * 3 + d1 % 3)
```

A real example

The affine transform:

```
f :: (20164, 147) -> (250563)
f (d0, d1) =
  (d1 / 21 * 867 + (d1 + d0 * 147) / 20874 * 1734 +
   d0 % 142 * 6 + d1 / 3 % 7 * 3 + d1 % 3)
```

This rather ugly affine transform can actually access the elements in order in a nice way:

```
for d0 = 0 to 246228 by 1734   -- 142 iterations
  for d1 = 0 to 852 by 6       -- 142 iterations
    for d2 = 0 to 6069 by 867  --   7 iterations
      for d3 = 0 to 21 by 1    --  21 iterations
        v[d0 + d1 + d2 + d3]
```

Memory allocation

Given that all memory access go through an affine transform we don’t have to allocate linear segments of memory for an array.

- Record concatenation is (sometimes) possible without moving data.
- Useful for allocation in multi-dimensional memories.

Conclusions

Quasi-affine transformations are simple and very useful for man structural array operations.

Arrays are awesome!