ArrayView Index Math

The math behind `ArrayView` indexing in arkouda

In [ ]:
```python
import numpy as np
from itertools import product
```

Say we have a $m \times n$ matrix A

$$\mathbf{A} := \underbrace{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}}_{n} \left.\vphantom{\begin{bmatrix} a \\ a \\ \vdots \\ a \end{bmatrix}}\right\} m$$

We are logically treating A as if it is multi-dimensional, but it's being stored in memory as a flat array

$$A_{\text{flat}} = \underbrace{\begin{bmatrix} a_{11}, & \cdots & a_{1n}, & a_{21}, & \cdots & a_{2n}, & \cdots & a_{m1}, & \cdots & a_{mn} \end{bmatrix}}_{m \times n}$$

What is actually being provided is a multi-dimensional view of a 1-d array. So we need to convert multi-dimensional coordinates into the corresponding coordinate in the flat array.

Integer only indexing

Let's saying we have the following: (Note Numpy is row-major by default)

```
>>> a = np.arange(15).reshape(3,5)
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.base
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

We see that a single step in the first dimension of `a`, increases the corresponding index into the flat array by 1

$$a = \begin{bmatrix} 0 & \color{red}{1} & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \end{bmatrix}$$

$$\text{a.base} = \begin{bmatrix} 0 & \color{red}{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \end{bmatrix}$$

```
>>> a[0,1]
1
>>> a.base[1]
1
```

But taking a single step in the second dimension `a`, increases the corresponding index into the flat array by 5 because we are skipping an n long row

$$a = \begin{array}{c} \overbrace{\hphantom{0 \quad 1 \quad 2 \quad 3 \quad 4}}^{n} \\ \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ \color{red}{5} & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \end{bmatrix} \end{array}$$

$$\text{a.base}=[\overbrace{0 \quad 1 \quad 2 \quad 3 \quad 4}^{n} \quad \color{red}{5} \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14]$$

```
>>> a[1,0]
5
>>> a.base[5]
5
```

Following this logic of 1 step in first dimension of `a` is 1 in a.base and 1 step in second dimension is n in `a.base`, we can work out the formula

$$a[i,j] = \text{a.base}[(n \cdot i) + j]$$

This intuition caries forward to higher dimensions:

```
>>> a = np.arange(24).reshape(2,3,4)
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> a.base
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23])
```

A single step in the third dimension of `a` results in a stride of 3×4 steps in `a.base`, because we are skipping a size $m \times n$ matrix

$$a = \begin{bmatrix} \left. \overbrace{\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}}^{n} \right\} m \\ \begin{bmatrix} \color{red}{12} & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \end{bmatrix} \end{bmatrix}$$

$$\text{a.base}=[\overbrace{0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11}^{m \times n} \quad \color{red}{12} \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20$$

```
>>> a[1,0,0]
12
>>> a.base[(3*4)]
12
```

From this we can work out,

$$a[i, j, k] = \text{a.base}[((m \cdot n) \cdot i) + (n \cdot j) + k]$$

So the idea is we're multiplying each additional higher dimension by the product of the previous dimensions and then summing

To phrase this more technically, let's take a look at this block borrowed from address calculation section of the `Row- and column-major order` Wikipedia entry:

> For a d-dimensional $N_1 \times N_2 \times \cdots \times N_d$ array with dimensions $N_k (k = 1...d)$, a given element of this array is specified by a tuple $(n_1, n_2, \ldots, n_d)$ of d (zero-based) indices $n_k \in [0, N_k - 1]$.In row-major order, the ''last'' dimension is contiguous, so that the memory-offset of this element is given by:
>
> $$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$
>
> In column-major order, the ''first'' dimension is contiguous, so that the memory-offset of this element is given by:
>
> $$n_1 + N_1 \cdot (n_2 + N_2 \cdot (n_3 + N_3 \cdot (\cdots + N_{d-1} n_d) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=1}^{k-1} N_\ell \right) n_k$$

Looking just at the `column_major` we see this is the sum of the coordinates times the product of the previous dimensions. If we define

$$P_k := \prod_{\ell=1}^{k-1} N_\ell$$

Then the `column_major` formula becomes

$$\sum_{k=1}^{d} P_k n_k$$

Note: the `row_major` equivalent is just the product of reversed dimension and reversed coordinates

We can cache this since the shape and order won't change between indexing (otherwise a new ArrayView object would need to be created)

```
dim_prod = cumprod(shape)//shape if order is COLUMN_MAJOR else
cumprod(reverse_shape)//reverse_shape
```

with `dim_prod` cached, for any given set of coords, our address calculation for the correct index in the base array is

```
# column major
sum(dim_prod * coords)
# row major
sum(dim_prod * coords[::-1])
```

Let's verify this formula matches numpy

In [ ]:
```
base = np.arange(30)
# F is column_major
f = base.reshape(5, 3, 2, order='F')
f_dim_prod = np.cumprod(f.shape)//f.shape
print(f"f_dim_prod = {f_dim_prod}")
print(f"f =\n{f}")
```

```
f_dim_prod = [ 1  5 15]
f =
[[[ 0 15]
  [ 5 20]
  [10 25]]

 [[ 1 16]
  [ 6 21]
  [11 26]]

 [[ 2 17]
  [ 7 22]
  [12 27]]

 [[ 3 18]
  [ 8 23]
  [13 28]]

 [[ 4 19]
  [ 9 24]
  [14 29]]]
```

In [ ]:
```
# Check for every coordinate our result matches
coords = list(product(range(5), range(3), range(2)))
print([f[i] for i in coords])
print([base[sum(f_dim_prod * i)] for i in coords])
```

```
[0, 15, 5, 20, 10, 25, 1, 16, 6, 21, 11, 26, 2, 17, 7, 22, 12, 27, 3, 18, 8, 23, 13, 28,
4, 19, 9, 24, 14, 29]
[0, 15, 5, 20, 10, 25, 1, 16, 6, 21, 11, 26, 2, 17, 7, 22, 12, 27, 3, 18, 8, 23, 13, 28,
4, 19, 9, 24, 14, 29]
```

This also works for `row_major` with `dim_prod=cumprod(c.shape[::-1])//c.shape[::-1]` and coords reversed

In [ ]:

```
# C is row_major (which is the default for numpy)
c = base.reshape(5, 3, 2)
c_dim_prod = np.cumprod(c.shape[::-1])//c.shape[::-1]
print(f"c_dim_prod = {c_dim_prod}")
print(f"c =\n{c}")
```

```
c_dim_prod = [1 2 6]
c =
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]

 [[12 13]
  [14 15]
  [16 17]]

 [[18 19]
  [20 21]
  [22 23]]

 [[24 25]
  [26 27]
  [28 29]]]
```

In [ ]:

```
# Check for every coordinate our result matches
print([c[i] for i in coords])
print([base[sum(c_dim_prod * i[::-1])] for i in coords])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 2
4, 25, 26, 27, 28, 29]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 2
4, 25, 26, 27, 28, 29]
```

Mixed Type indexing

The next type of indexing to contend with is when we have slices as indicies (pontentailly mixed with integer indicies). We'll walk through how we calculate the indices into the flat array for these slices in an example

In [ ]:

```
a = np.arange(27).reshape(3,3,3)
a[1:3, 0:3:2, 0:3:2]
```

Out[ ]:
```
array([[[ 9, 11],
        [15, 17]],

       [[18, 20],
        [24, 26]]])
```

Since `a.shape = (3,3,3)` . We get `dim_prod = (1,3,9)`

In [ ]:
```
dim_prod = np.cumprod(a.shape[::-1])//a.shape[::-1]
print(f"a.shape = {a.shape}")
print(f"dim_prod = {dim_prod}")
```

```
a.shape = (3, 3, 3)
dim_prod = [1 3 9]
```

We then calculate the set of indicies desired at every dimension, so

In [ ]:
```
dim1_coords = np.arange(a.shape[0])[1:3]
dim2_coords = np.arange(a.shape[1])[0:3:2]
dim3_coords = np.arange(a.shape[2])[0:3:2]
print(f"dim1_coords = {dim1_coords}")
print(f"dim2_coords = {dim2_coords}")
print(f"dim3_coords = {dim3_coords}")
```

```
dim1_coords = [1 2]
dim2_coords = [0 2]
dim3_coords = [0 2]
```

From here we can calculate the coordinates scaled by the `dim_prod` so only a sum is required

In [ ]:
```
# Note we are reversing the dim_prod because we are row_major by default
scaled1 = dim3_coords * dim_prod[0]
scaled2 = dim2_coords * dim_prod[1]
scaled3 = dim1_coords * dim_prod[2]
scaled_coords = np.concatenate([scaled1, scaled2, scaled3])
print(f"scaled_coords = {scaled_coords}")
user_dims = [scaled1.size, scaled2.size, scaled3.size]
print(f"user_dims = {user_dims}")
offsets = np.cumsum(user_dims) - user_dims
print(f"offsets = {offsets}")
user_dim_prod = np.cumprod(user_dims[::-1]) // user_dims[::-1]
print(f"user_dim_prod = {user_dim_prod}")
ndims = 3
```

```
scaled_coords = [ 0  2  0  6  9 18]
user_dims = [2, 2, 2]
```

```
offsets = [0 2 4]
user_dim_prod = [1 2 4]
```

`scaled_coords` is just multiple arrays flattened into one with `offsets` indicating the start positions of those arrays. These arrays contain the coordinates for each dimension scaled by the appropriate `dim_prod`

The idea here is to take a cartesian product of the scaled coordinates and sum as we go.

$$[0, 2] \times [0, 6] \times [9, 18]$$

We write these values into a defined result array of size $\prod \text{userdims}$. Letting `j0, j1,` and `j2` be our index in the first dim, second, and third dim respectively, we can multiply by `user_dim_prod` and sum to find the correct position in the result array

$$\texttt{Ind} = \sum_i j_i \cdot \text{userdimprod}_i$$

$$\texttt{Sum} = \sum_i \text{scaledcoords}[\text{offsets}_i + j_i]$$

(j_0, j_1, j_2)	`Ind`	`Sum`	`res[Ind] = Sum`	`res`
(0,0,0)	0	$\sum(0, 0, 9) = 9$	$\text{res}[0] = 9$	$[9, 0, 0, 0, 0, 0, 0, 0]$
(0,0,1)	4	$\sum(0, 0, 18) = 18$	$\text{res}[4] = 18$	$[9, 0, 0, 0, 18, 0, 0, 0]$
(0,1,0)	2	$\sum(0, 6, 9) = 15$	$\text{res}[2] = 15$	$[9, 0, 15, 0, 18, 0, 0, 0]$
(0,1,1)	6	$\sum(0, 6, 18) = 24$	$\text{res}[6] = 24$	$[9, 0, 15, 0, 18, 0, 24, 0]$
(1,0,0)	1	$\sum(2, 0, 9) = 11$	$\text{res}[1] = 11$	$[9, 11, 15, 0, 18, 0, 24, 0]$
(1,0,1)	5	$\sum(2, 0, 18) = 20$	$\text{res}[5] = 20$	$[9, 11, 15, 0, 18, 20, 24, 0]$
(1,1,0)	3	$\sum(2, 6, 9) = 17$	$\text{res}[3] = 17$	$[9, 11, 15, 17, 18, 20, 24, 0]$
(1,1,1)	7	$\sum(2, 6, 18) = 26$	$\text{res}[7] = 26$	$[9, 11, 15, 17, 18, 20, 24, 26]$

Now that we have our indicies stored in `res`, we can index into our base array and reshape into `user_dims` to get our answer

```
>>> a.base[res].reshape(user_dims) =
[[[ 9 11]
  [15 17]]

 [[18 20]
  [24 26]]]
```

In [ ]:

```
def recurse(depth=0, ind=0, s=0):
    for j in range(user_dims[depth]):
        if depth == ndims-1:
            res[ind + j*user_dim_prod[depth]] = s + scaled_coords[offsets[depth]+j]
            print(f"res[{ind + j*user_dim_prod[depth]}] = {res[ind + j*user_dim_prod[depth]]}\n===========\n")
        else:
            recurse(depth+1, ind + j*user_dim_prod[depth], s + scaled_coords[offsets[depth]+j])
```

In [ ]:

```
res = np.zeros(np.prod(user_dims), dtype=np.int64)
recurse()
```

```
res[0] = 9
===========

res[4] = 18
===========

res[2] = 15
===========

res[6] = 24
===========

res[1] = 11
===========

res[5] = 20
===========

res[3] = 17
===========

res[7] = 26
===========
```

In [ ]:

```
print(f"a[1:3, 0:3:2, 0:3:2] =\n{a[1:3, 0:3:2, 0:3:2]}\n")
```

```
print(f"a.base[res].reshape(user_dims)
=\n{a.base[res].reshape(user_dims)}")
```

```
a[1:3, 0:3:2, 0:3:2] =
[[[ 9 11]
  [15 17]]

 [[18 20]
  [24 26]]]

a.base[res].reshape(user_dims) =
[[[ 9 11]
  [15 17]]

 [[18 20]
  [24 26]]]
```

Advanced Indexing

numpy calls indexing by arrays "Advanced Indexing" and this behaves differently than a slice with the equivalent indices when there are more than 2 arrays present

In [ ]:

```
n = np.arange(4).reshape(2,2)
# sometimes they line up
n[:,:]
```

Out[ ]:

```
array([[0, 1],
       [2, 3]])
```

In [ ]:

```
n[:,[0,1]]
```

Out[ ]:

```
array([[0, 1],
       [2, 3]])
```

In [ ]:

```
n[[0,1],:]
```

Out[ ]:

```
array([[0, 1],
       [2, 3]])
```

In [ ]:

```
# sometimes they do not
n[[0,1],[0,1]]
```

Out[ ]:

```
array([0, 3])
```

psuedocode for potential solution: (Though `ind` still requires some reworking)

```
def recurse(depth=0, ind=0, s=0, advanced_ind=-1):
    if not advanced[depth] or advanced_ind == -1:
```

```
        for j in range(user_dims[depth]):
            if depth == ndim-1:
                res[ind + j*user_dim_prod[depth]] = s +
scaled_coords[offsets[depth]+j]
            else:
                recurse(depth+1,
                        ind + j*user_dim_prod[depth],
                        s + scaled_coords[offsets[depth]+j],
                        advanced_ind if not advanced[depth] else j)
    else:
        if depth == ndim-1:
            res[ind + advanced_ind*user_dim_prod[depth]] = s +
scaled_coords[offsets[depth]+advanced_ind]
        else:
            recurse(depth+1,
                    ind + advanced_ind*user_dim_prod[depth],
                    s + scaled_coords[offsets[depth]+advanced_ind],
                    advanced_ind)
```