Array Spooling

A sewing machine bobbin is a cylindrical object onto which thread is *spooled* before the machine is operated.

In computing, there are many cases where your data is multi-dimensional. But multi-dimensional arrays are unwieldy, especially in distributed memory systems. One dimensional arrays are best.

If you choose to *spool* your data into a one-dimensional array, you ought to know where your data ends up, and ideally, be able to access it efficiently. These notes help me do that.

To spool an n-dimensional array into a 1-d array, we need to map an arbitrary number of indices into a single index y:

$$\text{spool}(y) \leftarrow \text{array}[i, j, k, l, m, n, o, p, ...], \tag{1}$$

To create a one-to-one correspondence, we have to assign an order to the n indices:

$$x_n > x_{n-1} > ... > x_1. \tag{2}$$

This ordering, or hierarchy, is arbitrary; it doesn't have anything to do with the meaning of the indices or their ranges, but it does have consequences, discussed later.

Let's further suppose that each index x runs from x^{min} to x^{max}.
With these constraints, we can use the following mapping:

$$y = x_n + \sum_{i<n}(x_i - x_i^{min}) \prod_{j>i}(x_j^{max} - x_j^{min} + 1). \tag{3}$$

The inequalities under the sum and product symbols are with respect to the ordering. For example, $i < n$ means "those indices x_i with lesser order than x_n".

With this mapping, incrementing x_n will change y by the least amount. In fact, it will be incremented by one. Going up one index in the ordering we have defined will increase the amount y is incremented.

Our choice of ordering should therefore be informed by the *access pattern* we expect for this array. Indices which will be incremented more frequently (e.g. deeper inside a nested-loop structure) should appear higher in the ordering.

2-d example

Let's take a two dimensional array array$[x_2, x_1]$ where $x_1 = 1, ..., m$, and $x_2 = 0, ..., n-1$. We have to choose an ordering, so let's take $x_2 > x_1$.

Applying the formula, we have

$$y = x_2 + (x_1 - x_1^{min}) \times (x_2^{max} - x_2^{min} + 1) \tag{4}$$

Substituting our particulars we have:

$$\begin{aligned} y &= x_2 + (x_1 - 1)(n - 1 - 0 + 1) && (5)\\ &= x_2 + (x_1 - 1)n. && (6) \end{aligned}$$

If instead we take $x_1 > x_2$ then,

$$y = x_1 + (x_2 - x_2^{min}) \times (x_1^{max} - x_1^{min} + 1), \tag{7}$$

which is:

$$\begin{aligned} y &= x_1 + (x_2 - 0)(m - 1 + 1) && (8)\\ &= x_1 + x_2 m. && (9) \end{aligned}$$

3-d example

Suppose a three dimensional array array$[x_1, x_2, x_3]$ where $x_1 = 1, ..., r$, $x_2 = 0, ..., s$, and $x_3 = 1, ..., 2t$. We take the ordering $x_2 > x_3 > x_1$.

$$\begin{aligned} y &= x_2 + (x_3 - 1)(s - 0 + 1) + (x_1 - 1)(2t - 1 + 1)(s - 0 + 1) && (10)\\ &= x_2 + (x_3 - 1)(s + 1) + (x_1 - 1)2t(s + 1). && (11) \end{aligned}$$

Again, changing the ordering will change the expression. This ordering would be most efficient for a nested loop such as:

```
do x1 = 1, r:
  do x3 = 1, 2t:
    do x2 = 0, s:
      y = x2 + (x3-1)(s+1) + (x1-1)*2*t*(s+1)
      access spool[y]
```

This requires $10(s + 1)(2t)r$ integer operations to compute.

Sectors

The 3-d example in the previous section can be improved by pre-computing parts of the index further up in the loop hierarchy.

$$y = x_n + \text{sector}(x_{n-1}, ..., x_1), \tag{12}$$

where

$$\text{sector}(x_{n-1}, ..., x_1) \equiv \sum_{i<n} (x_i - x_i^{min}) \text{sectorsize}(i). \tag{13}$$

The sectorsize quantities do not depend on the loop variable x_i, and can therefore be pre-computed beforehand:

$$\text{sectorsize}(i) \equiv \prod_{j>i} (x_j^{max} - x_j^{min} + 1). \tag{14}$$

Or, recursively:

$$\text{sectorsize}(i) = (x_i^{max} - x_i^{min} + 1) \text{sectorsize}(i-1), \tag{15}$$

with $\text{sectorsize}(1) = 1$.

Improved 3-d example

Taking the same 3-d example as before, we can improve the performance by computing sectors in shallower loops and by pre-computing sector sizes. Recall that:

$$y = x_2 + (x_3 - 1)(s+1) + (x_1 - 1)2t(s+1). \tag{16}$$

In this case:

```
secsize1 = 1
secsize2 = (s+1) # * secsize1
secsize3 = (2*t) * secsize2

do x1 = 1, r:
  sector1 = (x1 - 1) * secsize3
  do x3 = 1, 2t:
    sector31 = (x3 - 1) * secsize2 + sector1
    do x2 = 0, s:
      y = x2 + sector31
      access spool[y]
```

This requires $(2t)(s+1)r + 3(2t)r + 2r + 3$ operations to compute. Assuming $2t$, s, and r are all similarly large, this is a speedup of approximately $\frac{20str}{2str} = 10$.

Upper-triangular

Suppose we have some symmetry which allows us to only store the upper triangular part of a two-dimensional array. I.e. $i <= j$ for all rows i and columns j. Then the access pattern of the array might look like the following:

```
  column j
r  1  2  4  7 11
o     3  5  8 12
w        6  9 13
            10 14
i               15
```

and so on. In this case, the spooling is different. Importantly, we no longer need to know the size of the array, because the number of labeled rows in each column is always equal to the column number. Further, we always know that the largest index in the previous column $j' = j - 1$ is the sum of all integers up to j', which is $j'(j'+1)/2$. Thus:

$$y = i + j(j-1)/2. \tag{17}$$

This can be combined with higher dimensional square blocks.

Un-spooling

Suppose we have a single do-loop which addresses a two-dimensional array. How do we invert the map?

It depends on the access pattern. We could have a row-major access pattern:

```
1 4 7
2 5 8
3 6 9
```

In which case the mapping is

$$x_1 = (y-1)/s_{x_1} + 1 \tag{18}$$
$$x_2 = \mod(y-1, s_{x_1}) + 1 \tag{19}$$

Upper triangular

Suppose the desired access pattern is upper triangular:

```
1 2 4 .
  3 5 .
    6 .
      .
```

Then the mapping is:

$$j = \text{ceiling}((\sqrt{8y+1} - 1)/2) \tag{20}$$
$$i = y - j(j-1)/2 \tag{21}$$

Here, i is fast and j is slow. A nice aspect of this layout is that the mapping does not depend on the size of the array. However, it does rely on real-type arithmetic (but there's a way around that).

The compute cost of the j index is at least 5 FLOPS plus the cost of the ceiling function. The cost of the i index is 4 FLOPS. The former is reduced to 4 FLOPS with:

$$j = \text{ceiling}(\sqrt{2y + 0.25} - 0.5) \tag{22}$$
$$i = y - j(j-1)/2 \tag{23}$$

To see why this seemingly arbitrary map works, notice from the label pattern above that we are trying to map the first N integers onto the first n columns, where the j-th column has j consecutive labels. The maximum label which n rows can support is:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}. \tag{24}$$

The index y maps to the column with at least as many labels:

$$y \leq \frac{j(j+1)}{2}. \tag{25}$$

The positive solution for j is:

$$j \leq \frac{\sqrt{8y+1} - 1}{2}. \tag{26}$$

Finally, by demanding that j be an integer, we find the first part of the mapping:

$$j = \text{ceiling}\left(\frac{\sqrt{8y+1} - 1}{2}\right). \tag{27}$$

Since the label y increases by one when increasing the row index i, all we need is to subtract the maximum label from the previous column from y to get i. But this is just the sum of all integers up to the previous column number $j' = j - 1$. A simple substitution gives us the final result for the row i:

$$i = y - \frac{j(j-1)}{2}. \tag{28}$$

A word of caution: Since this map relies on real numbers, it is susceptable to rouding errors. In particular, if $1/j$ is small compared to the precision of the real data type, then the ceiling function may go the wrong way!