**Notes on index ranges and striding in Chapel: dense linear algebra**

In today’s standard scientific languages the model for passing (arithmetic) array arguments to procedures is a combination of “call by reference” with re-indexing. That is, a base address for the argument is passed, together with information that permits reconstructing the association between indexed entries of the array and their addresses. The original index ranges of the actual arguments are lost; additional arguments would be required to retain the original index ranges. The formal arguments within the procedure are re-indexed to a standard base index for each dimension / rank (zero for C, one for Fortran). (The base index can be overridden with an explicitly chosen index.) The model used in most standard libraries, including LAPACK and ScaLAPACK, is that actual arguments will be stored contiguously and each dimension is indexed from the Fortran base index of 1.

Other models, such as passing strided array sections in Fortran, lead to complicated and confusing rules on array association and entry sequence association. They may also lead to explicit copying of arguments to contiguous storage.

In contrast, Chapel arrays carry their indexing (domain) information with them. For example, consider

a block partitioned square array, with each dimension indexed as 1..m, m+1 .. 2m. Passing the block to a procedure results in an array whose first dimension is range 1..m and second dimension is range m+1..m. Chapel does not re-index by default. If the actual argument is strided, the argument in the procedure is strided.

There are contexts in which it is important or useful to retain the original indexing information. The Chapel AMR code is an example. There are other contexts, such as writing dense linear algebra libraries, in which the standard unstrided, indexed from a standard base, model simplifies coding.

The major point of this note is that Chapel permits us to have our cake and eat it too. We do need to keep track of which we are trying to do. But we can have the best of both worlds.

In the following we will use a simple dense matrix-matrix multiply kernel to illustrate the point. Suppose we want to compute. The following Chapel code looks similar to what we would write in Fortran or C, with the conformality condition required by this code written explicitly in the assertion statement. To be conformable, A and C must have the same number of rows, B and C must have the same number of columns and A must have the same number of columns as B has rows. The assertion looks similar to the usual condition for languages with a fixed base index and unit stride. However, in those languages “same number of” translates into “the same range from base index to high index by 1”.

def gemm\_matching\_ranges ( C : [] , A : [], B : [] )

const C\_rows = C.domain.dim (1);

const C\_cols = C.domain.dim (2);

const A\_cols = A.domain.dim (2);

assert ( A.domain.dim (1) == C\_rows &&

B.domain.dim (1) == A\_cols &&

B.domain.dim (2) == C\_cols );

for (i, j, k) in [C\_rows, C\_cols, A\_cols] do

C (i,j) += A (i,k) \* B (k,j);

}

In this Chapel code, the test for conformality is simply that certain domains must match exactly. But there is no requirement on the low and high indices and no requirement that the domains be unstrided. So the Chapel code is arguably more general than code in conventional languages.

Is this useful in linear algebra? I believe it will whenever dealing with submatrices leads to situations where the natural indexing does not begin at the usual base index. One such place is computing a Schur complement during a factorization; the matrix-matrix multiplication has the form. The index sets match transparently, provided that the larger matrix has the same indices for rows as for columns.

Note the Chapel code is identical for unstrided and strided input, provided only that the domains match. How useful this will be is difficult to judge. Strided linear algebra codes are unusual for two reasons, difficulty of coding and loss of performance from lack of contiguity. Chapel removes the first obstacle. In this case, writing a strided code is no more difficulty than writing an unstrided code. The second obstacle is inherent in current processor architectures. Chapel does not solve that potential performance problem. Note however while discontiguous memory accesses may affect performance, the indexing calculations Chapel needs to make are almost identical for strided and unstrided matrices.

This is not a complete picture of the generality that Chapel can create for a library routine. One can write Chapel matrix-matrix multiplication code that accepts three matrix arguments, provided only that the conformality condition on *the number of rows / columns* is met. The domains do not have to match. The index sets of one may be offset fromanother; one may be strided and the other not; they may both be strided with different strides.

There are two solutions. I will begin here with the easy, more general, solution because it comes nearly for free and will be generally useful. I will then illustrate a more direct, and more complicated, approach which is not the right solution here, but whose principles may be useful elsewhere.

The first solution is *re-index* the argument domains so that they do match. This is like Fortran or C re-indexing. It is easier because Chapel’s domains simplify specifying the index sets. It is more general because it handles strided input transparently.

There are two ways to re-index, through *aliasing* arrays and through procedure argument re-indexing. The shell code below illustrates the first in our matrix-matrix multiplication example.

def gemm\_shell ( C : [] , A : [], B : [] )

const C\_rows = C.domain.dim (1);

const C\_cols = C.domain.dim (2);

const A\_rows = A.domain.dim (1);

const A\_cols = A.domain.dim (2);

const B\_rows = B.domain.dim (1);

const B\_cols = B.domain.dim (2);

if C\_rows == A\_rows && C\_cols == B\_cols && A\_cols == B\_rows then

gemm\_matching\_ranges ( C, A, B );

else {

assert ( A\_rows.length == C\_rows.length &&

B\_rows.length == A\_cols.length &&

B\_cols.length == C\_cols.length );

var A\_alias : [ idx\_base .. # A\_rows.length,

idx\_base .. # A\_cols.length ] => A;

var B\_alias : [ idx\_base .. # B\_rows.length,

idx\_base .. # B\_cols.length ] => B;

var C\_alias : [ idx\_base .. # C\_rows.length,

idx\_base .. # C\_cols.length ] => C;

gemm\_matching\_ranges ( C\_alias, A\_alias, B\_alias );

}

}

One could write a more complicated code that aliases fewer arrays in certain cases, but this code illustrates the general point and has quite minimal overhead.

An even simpler scheme, which has been proposed as a minor change in the Chapel language specification, is procedure argument re-indexing, as in the following procedure heading.

def gemm ( C : [idx\_base .., idx\_base ..] ,

A : [idx\_base .., idx\_base ..],

B : [idx\_base .., idx\_base ..] )

What must we do if we want for some reason to deal directly with mismatched indices? First, here is general Chapel code for this matrix-matrix kernel, based on “zippering” together the appropriate sets of indices:

def gemm\_general\_ranges ( C : [] , A : [], B : [] )

const C\_rows = C.domain.dim (1);

const C\_cols = C.domain.dim (2);

const A\_rows = A.domain.dim (1);

const A\_cols = A.domain.dim (2);

const B\_rows = B.domain.dim (1);

const B\_cols = B.domain.dim (2);

for ( (ic, ia), (jc, jb), (ka, kb) ) in

[ (C\_rows, A\_rows), (C\_cols, B\_cols), (A\_cols, B\_rows) ] do

C (ic,jc) += A (ia,ka) \* B (kb,jb);

}

This is also a simple solution. It appears to be of similar complexity to the aliasing solution. This is in part because I omitted the assertions that check on matching row length, knowing that the runtime will make that check when “zippering” pairs of ranges together. I believe that the necessity to keep track of six indices instead of three makes this style more error-prone. Further, this solution is not general. Consider the following fragment from a Cholesky factorization code for a symmetric matrix with equal row and column ranges.

for k in mtx\_indices (..j-1) do

A (j.., j) -= A (j, k) \* A (j.., k);

if A (j, j) > 0.0 then {

// compute the jth column of L

A (j, j) = sqrt ( A (j, j) );

A (j+1.., j ) = A (j+1.., j) / A (j, j);

}

If the domains have other than unit stride, the reference to the sub-ranges via the “next” index j+1 is a reference to an index not in the range, which will result in a memory addressing error. This can be fixed by explicitly keeping track of the stride, as in

const stride = A\_domain.dim (1).stride;

A (j+stride.., j ) = A (j+stride.., j) / A (j, j);

Alternatively, one can use Chapel’s facility to intersect a pair of ranges as shown below.

A (mtx\_index (j+1..), j ) = A (mtx\_index (j+1..), j) / A (j, j);

I believe the aliasing solution is arguably simpler and easier than writing code to manage mismatched indices or strides. It is also simpler for the compiler, which produces faster code for the aliasing solution.

One other general lesson to be taken here is that zippering is simpler and more general than keeping track of offsets between indices.