Solutions for Homework 1

Problem 1

Assume that \( x^T = (x_1^T, \ldots, x_k^T) \) and \( y^T = (y_1^T, \ldots, y_k^T) \) where \( k = n/R \) and \( x_i, y_i \in \mathbb{R}^R \).

After accumulating a partial sum in each element of a vector register, the \( R \) values must be summed to a single scalar. This can be done within the registers without having to store data in memory or create any workspace. The rightshift instruction is for convenience only – this assumes a particular ordering of the bits of the mask register relative to the elements of the vector register. It could also be done with integer divides. Note the use of compress to split the vector into halves. This allows vector processing to be used during the summation of the elements within the vector register. In practice, this would probably not be done down to a vector length of one. It would stop at some minimum vector length and the remaining summation would be done by extracting the vector elements into scalar registers and summing in scalar mode.

```
CVM
MOVSL VLR, R
  do i=1,k
    LEA A1, X(1+(I-1)*R)
    LEA A2, Y(1+(I-1)*R)
    LV V1, A1
    LV V2, A2
    MULTV V3, V1, V2
    ADDV V4, V3, V4
  end do
L1: DIVSS R1, 2, VLR
*** SET MASK TO VLR/2 O’S THEN ALL 1’S
*** ON FIRST PASS AND HALF AS MANY O’S
*** AS BEFORE ON SUBSEQUENT PASSES
*** USING A RIGHT SHIFT (0 SHIFTED IN) OF VM
CVM
  RHTSHFT VM, R1
*** BRING SECOND HALF OF VECTOR OF
*** VLR PARTIAL SUMS IN V4 INTO FIRST
*** PART OF V5 (NOTE VECTOR LENGTH
*** HERE IS STILL VLR)
  CMPRS V5, V4
*** HALVE THE VECTOR LENGTH AND ADD
  DIVSS VLR, 2, VLR
  ADDV V4, V4, V5
*** TEST VECTOR LENGTH AND BRANCH IF NONZERO
  LSS VLR, 1
  BRA L1
```
Problem 2.

We are to compute the function

\[ y_{i,j} = \alpha_1 x_{i-1,j} + \alpha_2 x_{i+1,j} + \alpha_3 x_{i,j} + \alpha_4 x_{i,j-1} + \alpha_5 x_{i,j+1} \]

Given the data structures described in the problem this is easily turned into a sequential code.

do i = 2,n-1
  do j = 2,n-1
    B(i,j) = alpha1 * A(i-1,j)
      + alpha2 * A(i+1,j)
      + alpha3 * A(i,j)
      + alpha4 * A(i, j-1)
      + alpha5 * A(i, j+1)
  end do
end do

The question concerns how to vectorize this code. Given the sequential loop above it is clear that we can vectorize either the i loop or j loop. Typically, it is convention to assume the inner loop may be vectorized. So we can immediately vectorize the j loop above. Note it is important to verify that the dependences allow this. Also note that since we are only updating the interior portion of the mesh, i.e., we are not writing to the locations that contain the boundary conditions, the array B can be used in the assumed next update which uses the data in B to produce the next version of the mesh values in the array A.

do i = 2,n-1
  B(i,2:m-1) = alpha1 * A(i-1,2:m-1)
    + alpha2 * A(i+1,2:m-1)
    + alpha3 * A(i,2:m-1)
    + alpha4 * A(i,1:m-2)
    + alpha5 * A(i,3:m)
end do

The vector length here is \( m - 2 \) which is the maximum possible in the \( j \) direction.

In order to vectorize the \( i \) loop we first must swap the loop nesting order. It is easy to verify that the dependences allow this. After the swap we can vectorize.

do j = 2,m-1
  do i = 2,n-1
    B(i,j) = alpha1 * A(i-1,j)
      + alpha2 * A(i+1,j)
      + alpha3 * A(i,j)
      + alpha4 * A(i,j-1)
      + alpha5 * A(i,j+1)
  end do
end do

do j = 2,m-1
  B(2:n-1,j) = alpha1 * A(1:n-2,j)
    + alpha2 * A(3:n,j)
    + alpha3 * A(2:n-1,j)
    + alpha4 * A(2:n-1,j-1)
    + alpha5 * A(2:n-1,j+1)
end do
The vector length here is $n - 2$ which is the maximum possible in the $i$ direction. The main difference between these codes is that given Fortran column major ordering, the vector accesses in the code that vectorizes the $i$ loop uses stride 1 while the code that vectorizes the $j$ loop uses a stride of LDA. As we will see in the memory discussion this may cause a loss in performance.

In fact we can write this in a data parallel form using the HPF or Fortran 90 syntax described in class. This demonstrates that the basic computation is in fact embarrassingly parallel.

$$B(2:n-1,2:m-1) = \alpha_1 A(1:n-2,2:m-1)$$
$$+ \alpha_2 A(3:n,2:m-1)$$
$$+ \alpha_3 A(2:n-1,2:m-1)$$
$$+ \alpha_4 A(2:n-1,1:m-1)$$
$$+ \alpha_5 A(2:n-1,3:m)$$

The second part of the question demonstrates that the Fortran level of coding and its associated data structures are convenient in terms of expression for this problem but they do impose artificial constraints on the vector length.

We can use the supplied assembly routines to extend the vector length beyond the $O(m)$ or $O(n)$ limit of the versions above. The only complication is that we must take into account the padding implied by the value of LDA and the fact that the B array contains boundary conditions in certain elements that we wish to preserve. If we protect these then we can exploit the fact that the arrays $B$ and $A$ are in fact stored in column major ordering and can be processed as vectors with an $O(mn)$ vector length.

Setting the mask to control the operations at first may seem problematic. However, here we can use the convenience of the Fortran data structure expression by treating the mask as a two dimensional array in Fortran and passing it to the assembler routine where it will be viewed as a one dimensional vector defined by a column major ordering.

The mask must be 0 in the positions of the boundary conditions and if we wish also to leave the padding due to LDA untouched (which is probably wise since it may be used to store other information) in the padding positions as well.

This is easily done at the fortran level.

mask(1,1:2:m) = 0
mask(n,1:2:m) = 0
mask(2:n,1) = 0
mask(2:n,m) = 0
mask(n+1:LDA,1:m) = 0

Consider the following diagram of the $A$ and $B$ arrays where LDA is 6, $m$ is 5 and $n$ is 4

$$
\begin{array}{cccccccc}
S & S & S & S & S & & & \\
S & I & I & I & S & & & \\
S & I & I & I & S & & & \\
S & S & S & S & S & & & \\
P & P & P & P & & & & \\
P & P & P & P & & & & \\
\end{array}
$$

The $I$'s are internal elements of the array $A$ that are updated by the loop and receive the corresponding elements in the array $B$. The $S$'s are positions that have specified boundary values and the $P$'s are elements that are padding due to LDA being larger than $n$. Since we assume column major ordering so the vector that receives the update values starts at $B(2,2)$ and continues continuously through memory until $B(n,m)$. Note that there are several boundary points and padding points in between that are protected by the mask setting. The vector length is easily determined. In column 2 elements 2,2 through LDA,2 are included (this is LDA-1 elements). In column m-1, elements 1,m-1 through n-1,m-1 are included (this is $n$ elements). In each of columns 3 through m-2, elements 1,i to LDA,i are included (this is LDA elements per column).
Therefore the total vector length is \( LDA - 1 + n + (m - 2 - 3 + 1) LDA = (m - 3) LDA + n - 2 \) which is \( O(mn) \) (or 14 for the example above). Each of the vectors that are scaled and then added to this vector have the same length and are determined by a shifted starting point. These starting points are determined by setting the \( i \) and \( j \) indices to their starting values in the sequential loop form of the code. If we assume that the operation field in the provided subroutine provides a VMUL (vector multiply) and a VMUADD (a vector multiply add) the code is easily generated. If you only assume vector multiply and vector add then the code is still easily generated by you will need a work vector defined. Assuming VL and the MASK have been set appropriately we have

\[
\begin{align*}
call & \text{ vectorop}(\text{alpha}3,1,1,\text{A}(2,2),1,\text{VL},\text{B}(2,2),1,\text{VL},\text{VMUL},\text{MASK}) \\
call & \text{ vectorop}(\text{alpha}4,1,1,\text{A}(1,2),1,\text{VL},\text{B}(2,2),1,\text{VL},\text{VMUADD},\text{MASK}) \\
call & \text{ vectorop}(\text{alpha}2,1,1,\text{A}(3,2),1,\text{VL},\text{B}(2,2),1,\text{VL},\text{VMUADD},\text{MASK}) \\
call & \text{ vectorop}(\text{alpha}5,1,1,\text{A}(2,3),1,\text{VL},\text{B}(2,2),1,\text{VL},\text{VMUADD},\text{MASK}) \\
\end{align*}
\]