AIPS MEMO NO. 47

National Radio Astronomy Observatory

Edgemont Road, Charlottesville, Virginia 22903-2475
804-296-0211 (FTS 938-1271); TWX 910-997-0174

22 August 1986

To: Vector-computer analysts

From: Don Wells

Subject: Installing NRAO's AIPS on Vector Computers

About 15 AIPS tasks were originally (circa 1980) designed to utilize a Floating Point Systems AP-120B array processor when one is available. In order to use such tasks on machines which do not have an FPS AP, NRAO developed the concept of a "pseudo-AP", in which a Fortran `COMMON` is used as the AP memory, and a library of Fortran and assembly language routines operate on data in this `COMMON`. These routines have exactly the same names, arguments, and functionality as the corresponding routines in the subroutine library which invokes the FPS AP. There is only one version of each of the tasks which use the AP; the choice of whether it uses a true AP or the pseudo-AP is made by link editing it with the appropriate subroutine library. Thus, the "pseudo-AP" concept in AIPS amounts to defining a "virtual device interface" for vector processing.

The pseudo-AP interface in AIPS is often called the "Q-routines", because the names of the subroutines all start with `Q`. The formal description of the interface is in Chapter 12 (Using the Array Processors) in Volume Two of "Going AIPS: A Programmer's Guide to the NRAO Astronomical Image Processing System", which is available from the AIPS Group at the address given above. Knowledge of the content of this chapter is *not* a prerequisite for an AIPS installation on a new vector architecture.

The vectorizing compilers of vector computers can generally recognize and exploit the inherent parallelism of the Fortran code of the pseudo-AP, which emulates the original AP microcode. Special features or limitations of particular architectures or compilers can be handled by creating custom versions of the Q-routines. NRAO currently supports Q-routine implementations for the FPS 38-bit APs, the Cray X-MP, and the Convex C-1. NRAO also developed Q-routines for the Alliant FX/8 during its 1985 benchmarking campaign, and several AIPS user sites have developed Q-routine implementations for other architectures. For example, the PSAIPS project at Pennsylvania State Univ. has developed an implementation for the Sky Warrior AP. The Cray, Convex, and Alliant implementations are nearly identical, because most of the adaptation to each of these vector computer architectures is handled by the vectorizing compilers.

Analysts should be aware that almost all of the heavy computing burden of AIPS image synthesis, self-calibration, and deconvolution is concentrated precisely in the Q-routine library. In fact, a satisfactory installation of AIPS on a vector computer can be made by vectorizing *only* the Q-routines, leaving the main programs and main application libraries compiled in scalar. The current Q-routine library has 97 Fortran subroutines. Most are simple vector operators or utility routines which vectorize immediately and optimally on all architectures, requiring no analyst effort at all. A few of the routines present special problems, and because of their great importance in the synthesis mapping application, need some customization. The purpose of this memo is to expose the character of these routines and the level of customization required.

The Q-routine library includes Fourier Transforms; AIPS needs 1-D complex and real-to-complex routines, with lengths up 8192. The distributed AIPS code contains an FFT which will vectorize reasonably well, but custom implementations are likely to improve upon it by as much as a factor of two. Analysts should note that overall AIPS performance is not dominated by FFTs—the FFT supplied with AIPS is generally good enough for an initial implementation, but should be replaced with a custom routine eventually (by calling the local routine from within a subroutine with the same name, arguments and functionality as the AIPS Q-routine).

AIPS uses very few standard signal processing algorithms, other than the FFT, and AIPS does not assume the existance of any libraries such as IMSL, NAG, SSL, LINPACK, etc., although specific routines from several public domain libraries (LINPACK and MINPACK in particular) are included in the bodies of certain AIPS tasks. None of these are really critical to the overall performance of AIPS implementations on vector architectures (if they were critical they would be in the Q-routine library!). The signal processing algorithms which are critical in AIPS are peculiar to radio astronomy. Two of these, "gridding" and "CLEANing", are especially important and are discussed below.

The subroutine QGRD4.FOR interpolates complex visibility samples onto a regular grid. The grid is required in order to allow the use of the Fast Fourier Transform (FFT) to compute the map. QGRD4 uses an interpolating kernel which is typically 7×7 in size, although the size is either larger (up to 11-square) or smaller (down to 3-square) in some special applications. The key technical problem of QGRD4 for vector architectures is that the kernel dimensions are small, comparable to $n_{\frac{1}{2}}$ (the vector half-length), thus tending to prevent achievement of the full performance of vector/parallel architectures. The outer DO-loop (label 300) has a long length (typically NVIS > 100), but the co-addition of the visibilities to the grid array produces a nasty *dependency* which prevents permuting the order of the DO-loops. In the implementation shown below, an index vector is used to access the 49 cells of a 7-square kernel using gather/scatter. This allows the multiply and add operations to execute with a vector length of 49, and makes sense if gather/scatter operations are fast enough. The code shown here presumes that the vectorizing compiler is able to vectorize the indirect addressing in the DO-200 loop (both the Convex and the Alliant do so). An alternative QGRD4 design concept for the case of "long-vector" architectures ($n_{\frac{1}{2}} > 100$) is discussed in AIPS Memo No. 33, "Gridding Synthesis Data in Vector Machines" (30 January 1985). This long-vector gridding algorithm has not yet been implemented by NRAO because the Cray, Convex and Alliant vector architectures all have relatively small values of $n_{\frac{1}{2}}$.

CLEAN is an iterative non-linear deconvolution algorithm which is used extensively in aperture synthesis radio mapping. Its purpose is to decompose an image into an ensemble of δ-functions which, when convolved with the beam (the point source response of the interferometer array) will be equal to the original data. When the iterative decomposition process has converged, the δ-functions are reassembled into a new map in the which the confusing sidelobe structures of the original "dirty" map are eliminated. Radio astronomers refer to the new map as the "clean map". The most efficient version of CLEAN was designed by Barry Clark of NRAO for the FPS AP-120B. The heart of his algorithm is contained in a microcode subroutine called CLNSUB, which subtracts one component from the current residual map and then finds the next component to be subtracted. It is called many thousands of times during the iterative decomposition process for a typical map. In the AIPS implementation of Clark's algorithm CLNSUB is renamed QCLNSU, and pseudo-AP Fortran versions are provided to complement Clark's microcode version for the 120B.

The original pseudo-AP version of the beam subtraction loop of QCLNSU (the DO-200 loop) had an IF-statement inside the loop. Unfortunately, the "truth-ratio" of this IF is low, and because most current vectorizing compilers compile such a loop using masking register logic, efficiency is low. Therefore, the implementation shown here calls a highly optimized subroutine WHNALT to construct an index vector of loop indices for which the IF succeeded, and then the beam subtraction loop uses indirect addressing (gather/scatter) to access the proper elements. Note that the search for the new largest residual is also performed by a highly optimized custom code, which is LINPACK's ISAMAX.

Subroutine WHNALT is quite similar to WHENILT in the Cray library (the only exception being the absolute value). If a WHENILT is available, the analyst can get a fairly good implementation of WHNALT by computing a scratch array of absolute value of the argument array, and then calling WHENILT. For peak efficiency a custom assembly language variation on WHENILT can be constructed; a version for the Convex C-1 is shown below as an example.

Analysts should note the presence of the statement "INCLUDE 'INCS:ZVND.INC'" before many of the DO-loops. The purpose of this is to insert the appropriate host-dependent version of the "no-vector-dependency" directive (for the Convex the statement is "C$DIR NO_RECURRENCE"; for the Cray it is "CDIR$ IVDEP"). Almost *all* DO-loops in the AIPS Q-routine library contain *apparent* vector dependencies because data in the pseudo-AP COMMON block is addressed using symbolic offsets whose values are not known at compile-time. The ZVD.INC directive before DO-300 in QGRD4 orders the compiler to compile that loop in *scalar,* because of the *real* dependency contained within that loop. An analyst doing a new vector machine implementation of AIPS needs to put his compiler's directives into the ZVND and ZVD files in the INCS directory so that the compiler (or an INCLUDE preprocessor) can insert them.

In general, obtaining a good implementation of the AIPS Q-routine library on a new vector computer, especially one with small or modest $n_{\frac{1}{2}}$, will probably be easy; almost all of the work has already been done in the previous ports. Much of the code will also perform well on long-vector architectures, but the gridding subroutine shown here is likely to require some work in order to achieve ultimate performance.

```
      SUBROUTINE QGRD4 (UV, VIS, WT, GRID, CONX, CONY, NO2, M, LROW,
     *   ROW, INC, NVIS)
C-----------------------------------------------------------------------
C   Vector compiler version.
C   Depends on gather/scatter and will probably be slower than the
C   scalar version on scalar machines.
C   The vector length in the vectorized loop here will typically be 49
C   but will never be less than 9.
C   Convolves visibility data onto a grid.
C   A single channel is gridded at a time.
C   It assumes that NO points lie within one half the convolving
C   function support size of the outside edge.
C   Also assumes that the convolving support function contains no more
C   than 512 pixels (32*16).
C    Inputs:
C     UV     I*4   Location of (u,v) values in cells.
C     VIS    I*4   Location of (complex) visibilities.
C     WT     I*4   Weight for data. Assumes any tapering
C                  has already been done.
C     GRID   I*4   address of base address of gridded data.
C                  Order assumed to be the following
C                  for each of the M rows:
C                   1) 2 * LROW visibilities
C     CONX   I*4   address of base address of X convolving fn.
C     CONY   I*4   address of base address of Y convolving fn.
C     NO2    I*4   INT( (# cells used on a row) / 2 )
C     M      I*4   number of rows kept in the AP.
C     LROW   I*4   length of a row ( max. X).
C     ROW    I*4   address of lowest central row number.
C                  (contents of ROW subtracted from Y)
C     INC    I*4   increment for UV, VIS and WT
C     NVIS   I*4   number of visibilities to grid.
C      In the above, X refers to rows and y to columns
C      in the gridded data, NOT on the sky.  The total
C      numbers of rows and cells used on a row should
C      be odd.
C     All AP memory I/O values are assumed floating.
C     It is assumed that all values of v correspond to row M/2.
C     Uses work vector WKVEC7
C-----------------------------------------------------------------------
      INTEGER*4 UV, VIS, WT, GRID, CONX,  CONY, NO2, M, LROW, ROW, INC,
     *   NVIS,  N, INCR, HAF, IX, IY, LOOP, LIMIT
      INTEGER*4 JUV, JVIS, JWT, JGRID, JCONX, JCONY, JCX, JCY,
     *   JG, JJCX, JJLOOP, IFIX, IRND, ICX, ICY, IG
      INTEGER*4 INCX(512), INCY(512), INCG(512), OLDM, OLDN, OLDLRO
      REAL*4    AIM, RE, RRE, AAIM, X, XX, XWT, Y, RHALF, SIGN, ZERO,
     *   WWT, RROW
      INCLUDE 'INCS:DAPC.INC'
      INCLUDE 'INCS:CAPC.INC'
      INCLUDE 'INCS:EAPC.INC'
C                                             Store increment table in WKVEC7
      EQUIVALENCE (OLDM, IWVEC7(1)),      (OLDN, IWVEC7(2)),
     *            (OLDLRO,IWVEC7(3)),     (INCX, IWVEC7(515)),
     *            (INCY, IWVEC7(1028)),   (INCG, IWVEC7(1541))
      DATA ZERO, RHALF /0.0,0.5/
C-----------------------------------------------------------------------
      IRND(XX) = IFIX (XX + SIGN (RHALF, XX))
C                                             Convert addresses to 1 rel.
```

```
      JUV = UV + 1
      JVIS = VIS + 1
      JWT = WT + 1
      JGRID = APCORE(GRID+1) + 1.5
      JCONX = APCORE(CONX+1) + 1.5
      JCONY = APCORE(CONY+1) + 1.5
      RROW = APCORE(ROW+1)
      N = NO2 * 2 + 1
      HAF = LROW / 2 - NO2
      INCR = 2 * LROW
      LIMIT = N * M
      IF ((OLDN.EQ.N) .AND. (OLDM.EQ.M) .AND. (OLDLRO.EQ.LROW)) GO TO 70
C                                            Fill increment tables
      OLDN = N
      OLDM = M
      OLDLRO = LROW
      DO 50 LOOP = 1,LIMIT
         IY = (LOOP-1) / N
         IX = (LOOP-1) - IY * N
         INCX(LOOP) = IX * 100
         INCY(LOOP) = IY * 100
         INCG(LOOP) = IX * 2 + IY * INCR
 50      CONTINUE
C                                            Loop over visibilities.
C                                            This loop contains dependencies.
      INCLUDE 'INCS:ZVD.INC'
 70   DO 300 JJLOOP = 1,NVIS
C                                             Check weight.
         XWT = MAX (ZERO, APCORE(JWT))
C                                             Determine location.
         X = APCORE(JUV+1)
         Y = APCORE(JUV)
C                                             Deter. conv. fn loc.
         JCX = JCONX + IRND (100. * (IRND (X) - X - 0.5)) + 100
         JCY = JCONY + IRND (100. * (IRND (Y) - Y - 0.5)) + 100
C                                             Determine grid loc.
         Y = Y - RROW
         JG = JGRID + 2 * (IRND (X) + HAF) + IRND (Y) * LROW * 2
C                                             Get visibility
         RE = APCORE(JVIS) * XWT
         AIM = APCORE(JVIS+1) * XWT
C                                             Gridding loop
      INCLUDE 'INCS:ZVND.INC'
         DO 200 LOOP = 1,LIMIT
            ICX = JCX + INCX(LOOP)
            ICY = JCY + INCY(LOOP)
            IG = JG + INCG(LOOP)
C                                             Sum to grid.
            WWT = APCORE(ICX) * APCORE(ICY)
            APCORE(IG) = APCORE(IG) + WWT * RE
            APCORE(IG+1) = APCORE(IG+1) + WWT * AIM
 200        CONTINUE
C                                            Update for next vis.
         JUV = JUV + INC
         JVIS = JVIS + INC
         JWT = JWT + INC
 300     CONTINUE
C
```

```
_UMAO:[AIPS.15JAN87.Q.DEV.PSAP.CVEX]QGRD4.FOR;3                     11-AF

 999  RETURN
      END



_UMAO:[AIPS.15JAN87.INC.NOTST.CVEX]DAPC.INC;1                      31-JAI

C                                                   Include DAPC
C                                       Include for AP memory and work
C                                       C1 Version
      INTEGER*4 APSIZE
      PARAMETER (APSIZE=65536)
      REAL*4 APCORE(APSIZE+1), WKVEC1(APSIZE/2+1), WKVEC2(APSIZE/2+1),
     *   WKVEC3(APSIZE/2+1), WKVEC4(APSIZE/2+1), WKVEC5(APSIZE/2+1),
     *   WKVEC6(APSIZE/2+1), WKVEC7(APSIZE/2+1), WKVEC8(APSIZE/2+1),
     *   WKVEC9(APSIZE/2+1)
      INTEGER*4 APCORI(1), SPAD(16),
     *   IWVEC1(APSIZE/2+1), IWVEC2(APSIZE/2+1), IWVEC3(APSIZE/2+1),
     *   IWVEC4(APSIZE/2+1), IWVEC5(APSIZE/2+1), IWVEC6(APSIZE/2+1),
     *   IWVEC7(APSIZE/2+1), IWVEC8(APSIZE/2+1), IWVEC9(APSIZE/2+1)
      COMPLEX
     *   CWVEC1(APSIZE/4+1), CWVEC2(APSIZE/4+1), CWVEC3(APSIZE/4+1),
     *   CWVEC4(APSIZE/4+1), CWVEC5(APSIZE/4+1), CWVEC6(APSIZE/4+1),
     *   CWVEC7(APSIZE/4+1), CWVEC8(APSIZE/4+1), CWVEC9(APSIZE/4+1)
C                                                   End DAPC



_UMAO:[AIPS.15JAN87.INC.NOTST.CVEX]CAPC.INC;1                      31-JAI

C                                                   Include CAPC
C                                       Include for AP memory and work
C                                       C1 Version
      COMMON /APFAKE/ APCORE, WKVEC1, WKVEC2, WKVEC3, WKVEC4, WKVEC5,
     *   WKVEC6, WKVEC7, WKVEC8, WKVEC9
      COMMON /SPF/ SPAD
C
                                                    End CAPC



_UMAO:[AIPS.15JAN87.INC.NOTST.CVEX]EAPC.INC;1                      31-JAI

C                                                   Include EAPC
C                                       Include for AP memory and work
C                                       C1 Version
      EQUIVALENCE
     *   (APCORE, APCORI),
     *   (WKVEC1, IWVEC1, CWVEC1), (WKVEC2, IWVEC2, CWVEC2),
     *   (WKVEC3, IWVEC3, CWVEC3), (WKVEC4, IWVEC4, CWVEC4),
     *   (WKVEC5, IWVEC5, CWVEC5), (WKVEC6, IWVEC6, CWVEC6),
     *   (WKVEC7, IWVEC7, CWVEC7), (WKVEC8, IWVEC8, CWVEC8),
     *   (WKVEC9, IWVEC9, CWVEC9)
C                                                   End EAPC
```

```
      SUBROUTINE QCLNSU (COMP, LMAP, L1MAP, L2MAP, IBX, IBY, JNDEX,
     *   INDEX)
C-----------------------------------------------------------------------
C   CRAY version.
C   QCLNSU does a CLEAN on a list of residuals using a given beam
C   patch
C    Inputs:
C      COMP(4)  R*4  Component vector:
C                    0 => intensity
C                    1 => x in cells
C                    2 => Y in cells
C                    3 => CLEAN loop gain (fractional)
C      LMAP     I*4  number of residuals
C      L1MAP    I*4  First residual in Y window
C      L2MAP    I*4  Last residual in Y window
C      IBX,IBY  I*4  Beam patch half width in X and Y
C                    Residuals with delta X or Y from the component
C                    position in COMP .GE. IBX,IBY are to be ignored.
C      JNDEX    I*4  0 - rel Index in FLUX, IX, IY of next residual
C                    to sub.
C   Input From Vector Work common:
C      IWVEC3   I*4  = IX, X pixel location of residuals
C      IWVEC4   I*4  = IY, Y pixel location of residuals
C      WKVEC5   R*4  = FLUX, Residual flux density
C      WKVEC6,7 R*4  = BEAM, BEAM patch.
C   Output:
C      INDEX    I*4  Index in FLUX of next residual.
C   Useage notes:
C     The following common work vectors are used:
C      IWVEC8   = IDXRAY  => array of x pixel offsets
C      IWVEC9   = IB      => indirect address array
C-----------------------------------------------------------------------
      INTEGER*4 ISAMAX, KMAX, INCX, IDXRAY(1), IB(1), IX(1), IY(1)
      INTEGER*4 L21MAP, NIB, JNDEX
      INTEGER*4 LMAP, IBX, IBY, L1MAP, L2MAP, INDEX, BOFF, BEMADR,
     *   LROW, ADDR, LOOP, XCOMP, YCOMP, N1
      REAL*4    SUBT, COMP(4), BEAM(1), FLUX(1)
      INCLUDE 'INCS:DAPC.INC'
      INCLUDE 'INCS:CAPC.INC'
      INCLUDE 'INCS:EAPC.INC'
      EQUIVALENCE (IX, IWVEC3), (IY, IWVEC4), (FLUX, WKVEC5),
     *   (BEAM, WKVEC6)
      EQUIVALENCE (IDXRAY, IWVEC8), (IB, IWVEC9)
      DATA N1 /1/
C-----------------------------------------------------------------------
      LROW = 2 * IBY - 1
      BOFF = (IBX-1) * LROW + IBY
      IF (LMAP.LE.0) GO TO 999
C                                         Get component to be CLEANed
      XCOMP = COMP(2) + 0.5
      YCOMP = COMP(3) + 0.5
      SUBT = COMP(1) * COMP(4)
C                                         Get residual offsets
      INCLUDE 'INCS:ZVND.INC'
      DO 100 LOOP = L1MAP,L2MAP
         IDXRAY(LOOP) = IX(LOOP+JNDEX) - XCOMP
 100     CONTINUE
C                                         Compress x window
```

```
      L21MAP = L2MAP - L1MAP + 1
      CALL WHNALT (L21MAP, IDXRAY(L1MAP), N1, IBX, IB, NIB)
C                                                  Subtraction loop
      INCLUDE 'INCS:ZVND.INC'
      DO 200 LOOP = 1,NIB
         ADDR = IB(LOOP) + L1MAP - 1
C                                                  Get beam address
         BEMADR = BOFF + (IDXRAY(ADDR) * LROW)
     *      + (IY(ADDR+JNDEX) - YCOMP)
         FLUX(ADDR+JNDEX) = FLUX(ADDR+JNDEX) - SUBT * BEAM(BEMADR)
 200     CONTINUE
C                                                  Find largest mag. residual
C                                                 Call Steve Wallach's routine:
      INCX = 1
      KMAX = ISAMAX (LMAP, FLUX(1+JNDEX), INCX)
      COMP(1) = FLUX (KMAX+JNDEX)
      COMP(2) = IX(KMAX+JNDEX)
      COMP(3) = IY(KMAX+JNDEX)
C                                                  Save index of next max.
      INDEX = KMAX
C
 999  RETURN
      END
```

```
        .globl  _whnalt_
;-----------------------------------------------------------------------
; Convex assembly routine
;
; WHNALT returns a list of all locations in an integer array
; for which the integer absolute value is less than a specified
; target value. This is very similar to the Cray library routine
; WHENILT, the only difference being the absolute value. The
; algorithm is equivalent to the following FORTRAN code:
;
;       SUBROUTINE WHNALT (N, IARRAY, INC, ITARGET, INDEX, NVAL)
;C      (arg_pack offsets: 0    4      8     12      16     20 )
;       INTEGER*4          N, IARRAY, INC, ITARGET, INDEX, NVAL
;       INTEGER*4          I, J
;       NVAL = 0
;       J = 1
;       DO I = 1, N
;               IF (IABS(IARRAY(J)).LT.ITARGET) THEN
;                       NVAL = NVAL + 1
;                       INDEX(NVAL) = I
;               ENDIF
;               J = J + INC
;       ENDDO
;       RETURN
;       END
;
; Don Wells, NRAO-CV, 16-18June85.
; minor mod to speed up by Steve Wallach, Convex, Sept85.
; minor bug fix and text cleanup by Don Wells, 29Jan86
; trivial syntactic error fixed, DCW 14Feb86.
; ==============================================================
;                 Register assignments:
; a1 = INC*4 (stride)   s1 = N counter          v1 = "iota" vector
; a2 = IARRAY pointer   s2 = ITARGET            v2 = IARRAY load reg
; a3 = INDEX pointer    s3 = NVAL counter       v3 = INDEX store reg
; a4 = temporary        s4 = pop_cnt last cmprs v4 = negated IARRAY
; a5 = INC*4*128        s5 = "iota" bump const  v5 = IABS() scratch
; ==============================================================
_whnalt_:
        ld.w    @0(ap),s1       ; get N
        ld.w    @12(ap),s2      ; get ITARGET
        ld.w    #0,s3           ; init NVAL
        st.w    s3,@20(ap)      ;  "     "
        lt.w    #0,s1           ; 0.LT.N ?
        jbrs.f  quit            ; if not, go quit.
        ld.w    #4,vs           ; vs = 4 bytes
        ld.w    #128,vl         ; vl = 128
        ld.w    _mth$j_indx,v1  ; init "iota": 1...128
        ld.w    4(ap),a2        ; init IARRAY() pointer
        ld.w    16(ap),a3       ; init INDEX() pointer
        ld.w    #0,s4           ; prev_popcnt=0 initially
        ld.w    @8(ap),a1       ; get INC
        shf     #2,a1           ; INC*4 (stride in bytes)
        mov.w   a1,a5           ;
        shf     #7,a5           ; INC*4*128 (register stride)
        ld.w    #128,s5         ; init "iota" bump constant
looptop:                        ;
        mov.w   s1,vl           ; vl = N<6..0> (usually 128)
        mov.w   a1,vs                 ; vs = INC*4
        ld.w    (a2),v2               ; load next 128 from IARRAY()
        neg.w   v2,v4                 ; get negation of IARRAY(i)
        lt.w    v2,v4                 ; IARRAY(i).LT.negated_IARRAY(i) ?
        mask.t  v2,v4,v5              ; v4 = v4 if T, v2 if F (IABS op)
        le.w    s2,v5                 ; ITARGET.LE.IABS(IARRAY(i)) ?
        ;               following scalar ops execute concurrently
        ;               with the vector compare instruction:
        add.w   a5,a2                 ; IARRAY = IARRAY + INC*4*128
        add.w   s4,s3                 ; NVAL = NVAL + prev_vl
        mov.w   s4,a4                 ;
        shf     #2,a4                 ; words to bytes
        add.w   a4,a3                 ; INDEX = INDEX + prev_vl*4
        sub.w   #128,s1               ; N = N - 128
        ld.w    #4,vs                 ; vs = unity stride for INDEX() store
        ;               following vector compress is inhibited until
        ;               the mask operation above completes:
        cprs.f  v1,v3                 ; compress "iota_of_hits" using vm
        plc.f   vm,s4                 ; get pop count of non-zeroes
        mov.w   s4,vl                 ; pop count is result vector length
        st.w    v3,(a3)               ; store compressed "iota" in INDEX()
        lt.w    #0,s1                 ; 0.LT.N ?
        mov.w   s5,vl                 ; vl = 128
        add.w   v1,s5,v1              ; iota(i) = iota(i) + 128
        jbrs.t  looptop               ; if so, loop back
;
        add.w   s4,s3                 ; NVAL = NVAL + VL
        st.w    s3,@20(ap)            ; store NVAL
quit:   rtn
```