Chapter 12

Using the Array Processors

12.1 Overview

Many of the more important of the AIPS tasks do a great deal of computation. The traditional approach to increasing the performance of a cpu is by means of hardware arithmetic units called Array Processors. These array processors (or APs) have their own memory and high speed, pipelined arithmetic hardware enabling them to run much faster than the host for certain specialized operations. Since not all computers running AIPS will have, or need, array processors attached, there is a library of Fortran routines which emulate the functions of the array processor; these routines, and a common in the host memory, constitute the "pseudo-array processor". Since the details of the implementation of these routines will depend on the hardware on which the software is run, these routines are explicitly machine dependent and have names beginning with the letter "Q"; thus the "Q-routines". This chapter will describe the use AIPS makes of array processors and explain how to use APs. At the end of this chapter is a list of the major Q routines with detailed comments on the call sequences.

Modern scientific computers are increasingly including vector hardware as an integral part of the CPU, eliminating the need for array processors. The compiler on vector computers knows about the vector capabilities, which greatly simplifies using the vector hardware. The "Q-routines" developed for array processors — or, more properly, the pseudo AP routines — have been successfully adapted to vector computers. Considerations for vectorization are discussed in a later section.

12.1.1 Why use the Array Processor?

The principal reason for using an array processor is speed. The design of most array processors optimizes its performance for repetitive arithmetic operations, making it much faster at vector arithmetic than the host CPU. Since most APs operate asynchronously from the host CPU, they constitute a co-processor which increases the capacity of the system.

A second advantage of using an array processor is that it contains its own local memory. On systems with limited physical memory, or address space, this can be an important consideration. It will be possible in the near future to get array processors, or fast CPUs, with many megawords of local memory. Such large memories will allow the use of more efficient methods of processing data.

12.1.2 When to Use, and Not to Use, the AP

The array processor is most efficient at very repetitive operations such as doing FFTs and multiplying large vectors. Its efficiency is greatly degraded for non-repetitive operations, or operations requiring a great number of decisions based on the results of computations. In fact, most array processors have very limited capability to make decisions based on the results of computations. Since the APs have their own program and data memory, the AP instructions and the data must be transferred to, and the results transferred from, the AP. These I/O operations may cost more cpu time than the amount saved by using the array processor. As a general rule, use of the AP is more efficient than the CPU when multiple or complex (such as FFT) operations, which are highly repetitious, are going to be done on relatively large amounts of data (thousands of words or more). In other cases, using the AP will probably not help much and will keep other processes from using this valuable resource.

12.2 The AIPS Model of an Array Processor

The model of an array processor used is colored strongly by our early use of Floating Point Systems FPS AP-120B array processors. However, expressed in general terms, this model can be emulated on other real or virtual (pseudo) array processors. It should be noted that use of the APs requires vectorized programming; hence, implementation on super computers or other vector machines should be relatively efficient. The following describes the fundamental features of the AIPS model of array processors.

1. AIPS uses APs as detached vector arithmetic units. That is, data is sent into the AP, some (usually vector) operation is done, and the results are returned to the host CPU. The principal difficulty in the implementation of AIPS on other array or vector processors is that our concept of a vector operation is rather more general than that of most computing hardware manufacturers. Many of the more complex of the AIPS operations are better described as pipelined scalar operations. In the AIPS usage, high level control and use of disk storage is done in the host CPU and only arithmetic operations are done in the AP.

2. AIPS considers the AP to be a device which can be assigned via QINIT and deassigned via QRLSE. Basically, this means that data will not disappear from the task's assigned AP data memory between these calls. This concept has little meaning for virtual APs, except that the data memory is cleared after a QINIT call.

3. An AP should have a relatively large local data memory. The size of the AP data memory is obtained from a common set by ZDCHIN, which reads it from a disk file. The value in this disk file can be modified by the AIPS utility program SETPAR. In the case of pseudo (virtual) AP's, this memory is physically in the host CPU. A similar implementation could be done for an AP with significantly less capacity than an FPS AP-120B. In some vector implementations of the Pseudo AP the size of the memory given in the DDCH.INC include common is reset in QINIT.

4. In addition to data memory, the AP is assumed to have an array of 16 integer registers (SPAD) which can be read from the host CPU. These are used to communicate the addresses of maxima, minima, etc. This capability is not extensively used.

5. AIPS assumes that the array processor is programmable in that functions are used which are not now, or likely ever to be, in a standard library. If the AP is not programmable or is otherwise incapable of emulating one of the AIPS functions, then these functions must be performed in the host CPU and hidden from the AIPS routines. Alternately, these functions may be reformulated in terms of the functions available; this will be necessary for efficient implementation of long-vector super computers.

6. Communication with the AP by AIPS is via Fortran call statements which specify the data in the AP memory and other control information, transfer data between the AP and host CPU, or synchronize the operation of the AP and host CPU.

7. Data in the AP memory is specified by a base address and an increment. In current implementations these addresses are absolute, but this is not assumed. The calling process is assumed to have absolute control over an address space beginning at address 0 and extending to the address indicated in the device characteristic common (include DDCH.INC) as (1024*KAPWRD-1). Word addressing only is used. In pseudo AP implementations, further memory may be available; this additional memory is indicated by KAP2WD.

8. Many of the most crucial functions used by AIPS routines depend on data-dependent address generation and logic flow. As mentioned above, implementation of AIPS on an array processor without this capability will require reformulation of several of the algorithms (especially gridding and the in-core

CLEAN) in terms of vector operations. This reformulation will likely require vector logical operations, i.e., gather, scatter, merge, and compress operations.

9. AIPS assumes that the AP can handle either integer or real data values (with the same word size). Complex values consist of a pair of real values in adjacent locations, the first being the real part and the second being the imaginary part.

12.3 How to Use the Array Processor

Since the array processors used by AIPS have their own program and data memories, the instructions must be loaded in to the AP and data sent to, and results returned from, the AP. Since the AP runs asynchronously from the host cpu there must also be ways to synchronize the operations. The general operations are given in the following list, with the name of the subroutine AIPS uses for the given operation:

1. Assign / Initialize the AP. (QINIT)
2. Transfer data to the AP. (QPUT)
3. Wait for transfer to complete. (QWD, QWAIT)
4. Load and execute the AP program. (many)
5. Wait for computations to finish. (QWR, QWAIT)
6. Transfer data back to host cpu. (QGET)
7. Wait for transfer to complete. (QWD, QWAIT)
8. Release AP. (QRLSE)

12.3.1 AP Data Addresses

The AIPS convention for specifying data in the AP memory, which follows the Floating Point Systems (FPS) conventions, is to specify data by the zero-relative memory address of the first element in an array, the memory address increment between the elements of an array, and the number of elements in the array. On FPS APs, the memory address is an absolute address, but in implementations on other APs, the address may be a relative address, but this should be hidden from the programmer.

Q Routine Arguments

The call arguments to the Q routines (AP-routines) are integers. The exceptions to this are the host array names passed in QPUT and QGET. The FPS Q routines convert these to 16 bit unsigned integers.

Array Processor Memory Size

Since different array processors will have different memory sizes, the memory size of the AP is carried in the Device Characteristics Common which is obtained by the include DDCH.INC. The size of the AP is in the integer value KAPWRD as the multiple of 1024 words of AP data memory. Any operation with the AP should check that enough data memory is available and, if possible, scale the operation to make full use of the available memory.

Several Pseudo AP implementations have more memory available for certain operations. The amount of this secondary memory is given by KAP2WD. This memory is used as work space by some routines, but is generally available when these routines are not in use.

12.3.2 Assigning the AP

The array processor is assigned to the calling task using the AIPS routine QINIT. QINIT incorporates the AIPS priority system and provides for smooth use of the AP for batch tasks. The AIPS AP priority scheme is to give tasks with lower POPS numbers (the number at the end of the task name when it is running) higher priority. This is done by keeping a list of AP tasks in QINIT. When a task asks for an AP, QINIT then checks to see if any AP tasks with a lower POPS number are running; if so, QINIT suspends the task for a short period and then checks again. The number of times a task goes through the check/suspend loop before asking for the AP at the next opportunity is proportional to its POPS number. QINIT also sets values in common /BPROLC/ (include DBPR.INC which control the AP roller subroutine QROLL. The text of this include is shown at the end of this chapter and the use of the values is described in the detailed description of QROLL given at the end of this chapter. In some vector implementations of the Pseudo-AP QINIT also sets the size of the AP memory (KAPWRD, KAP2WD) in the DDCH.INC included common.

On some systems, batch AIPS tasks present more of a problem. AIPS batch tasks are usually run at lower priority than interactive tasks, so they may grab the AP and then not get enough cpu cycles to finish that AP operation for a very long time. To avoid this problem, QINIT increases the priority of the batch task to that of an interactive task while it has the AP.

QRLSE is used to deassign the AP. QRLSE also lowers the priority of batch tasks after the AP is released.

In the interest of a smooth and friendly system for users, it is important not to hog the AP for long periods of time. The priority system should then work to give lower POPS numbered AIPS users a larger fraction of the time, if they need the AP. A task should in general not keep the AP tied up for more than 5 to 10 minutes at a time, less if that is practical. For tasks which may need to keep the same data in the AP for long periods of time, such as tasks which compute models based on CLEAN components, there is an AP roller subroutine QROLL.

QROLL determines if it is time to roll out the AP based on values set by QINIT, and, if so, will create a scratch file (using the DFIL.INC system), copy the specified contents of the AP memory to a scratch file, release the AP, wait a short period of time, re-assign the AP, and load the previous contents back into the AP memory. Details of the call sequence to QROLL are found at the end of this chapter. IMPORTANT NOTE: QROLL (and APROLL) work properly only for floating point data. Integer values rolled will not be restored correctly.

12.3.3 Data Transfers To and From the AP

The fundamental routines for getting data to and from the Array Processor memory are QPUT and QGET; details of the call sequences can be found at the end of this chapter. In addition, for image-like data, there is the routine APIO.

APIO transfers image-like data between disk files and the array processor. The file open and close and initialization logic are all contained in this routine. Information about the file and the the desired properties of the I/O are passed to APIO in the array FLIST. APIO can access either cataloged 'MA' type files or scratch files using the DFIL.INC common system. APIO can handle arbitrary row lengths. This is done by breaking up the logical records if they are larger than the buffer size. It is **IMPORTANT** to always use the same size buffer when accessing a given file. Usage notes for APIO:

1. Opening the file.

   If APIO determines that the file is not open, it will open it. The file can be either a cataloged file or a scratch file using the DFIL.INC common system. If the catalog slot number given in FLIST is 0 or less, the file is assumed to be a scratch file. File open assumes that the file type is 'MA' (if cataloged); the file is opened patiently without exclusive use.

2. Initialization.

   APIO initializes the I/O using the values in FLIST when it opens the file. It may be initialized again at any time using OPCODE 'INIT'. Also, switching between 'READ' and 'WRIT' will force a flushing of the buffer ('WRIT') and initialization. Any initialization when the current operation is 'WRIT' will cause the buffer to be flushed.

3. Closing the file.

   The file may be closed with a call with opcode 'CLOS'. If the file is being written and a 'CLOS' call is issued, APIO will flush the buffer. This means that, if APIO is being used to write to a disk, it MUST be called with OPCODE='CLOS', 'READ', or 'INIT' to flush the buffer. NOTE: all pending AP operations MUST be complete before calling APIO with opcode 'CLOS'.

4. AP timing calls.

   APIO calls QWD before getting data from, or sending data to, the AP, but does not call QWR. The calling routine should call QWR as appropriate.

More details about the call arguments are found at the end of this chapter, and an example of the use of APIO is given in a later section.

12.3.4 Loading and Executing AP Programs

Loading and executing AP programs is done in a single call to the relevant routine. The call argument also includes the specification of the data, location of the output array, and any processing flags. A list of the AP routines currently supported in AIPS is found at the end of this chapter. If the function desired is not available, then it is possible to write it for the AP.

12.3.5 Timing Calls

Since array processors normally run asynchronously from the host, CPU timing calls are necessary. The subroutine calls basically suspend the operation of the calling program until the specified AP operation is completed. FPS claims that data transfers and computations (not involving the same AP memory) may be overlapped; however, the results of doing this are erratic and this practice should be avoided. On occasion, there appear to be timing problems whose symptoms are erratic and which produce very wrong results, which go away when apparently unnecessary timing calls are added, e.g., calls to QWR between calls to computation routines. We use three timing calls:

1. QWD suspends the calling program until data transfers to or from the AP are complete.
2. QWR suspends the calling program until the AP completes all computations.
3. QWAIT suspends the calling program until all data transfers and computations are complete.

12.3.6 Writing AP Routines

If the current library of AP routines does not contain the desired function, there are two possibilities for coding the function: (1) microcoding the routine or (2) using the Vector Functor Chainer (or equivalent on non-FPS APs) to combine existing functions to create the desired function. If either of these is chosen, the programmer should also write the corresponding pseudo-AP routines, if the task is likely to have general use. The name of the routine should start with the letter Q and be placed in the appropriate libraries.

For routines to be run on real array processors, there should be an intermediate routine that converts the integer addresses etc. to unsigned 2 byte integers. This routine should then call the actual AP routine.

In order to use microcode or Vector Function Chainer (VFC) routines, the following steps must be performed:

1. Compile VFC (or other high level language routines) to assembly (microcode) language. For FPS code, this is done by the FPS routine VFC.
2. Assemble microcode into machine code. For FPS code, this is done using APAL.
3. Link edit microcode routines together to make an executable module. For FPS code, this is done using APLINK, which creates a Fortran or host assembly language routine with the executable module in a data statement. Make sure the integer type declarations are correct. All integers to be passed to FPS handlers should be explicitly declared INTEGER*2.

4. Compile/assemble the Fortran/assembly language module and put in the appropriate subroutine link edit library.

It is beyond the scope of this manual to describe the use of the FPS or other AP software; the reader is referred to the appropriate manual provided by the AP vendor.

Microcoding Routines

It is beyond the scope of this manual to give details about microcoding for array processors; see the AP manuals for these details. The general principles of efficient microcoding are that several of the hardware units - address computation, floating add, floating multiply, and memory access - may be given instructions in a given cycle. In addition, the floating point hardware is pipelined. That is, even though it takes several cycles for an operation, it is broken up into several, single cycle steps and a new operation can be initiated each cycle.

This architecture allows for very efficient loops. The loop may be broken into several sections and one section from each of several passes through the loop may be processed in parallel. Efficient coding of loops may become very complicated, but careful coding may speed up the process by a factor of several to many. The source code for NRAO written microcode is kept in the file FPSSUB:NRAO.AP.

Vector Function Chainer

The principal purpose of the Vector Function Chainer is to combine a number of microcoded routines into a single AP call. This can greatly reduce the overhead of the host cpu talking to the AP; and, if the individual AP operations are relatively numerous and short, chaining routines can make a dramatic improvement in the speed of the overall process.

The Vector Function Chainer uses source code that looks vaguely like Fortran, but has very limited capabilities and essentially no access to the data memory. Hopefully, in the future, there will be efficient Fortran compilers for APs. (FPS has such a compiler for the 120B, but NRAO doesn't have a copy).

12.3.7 FFTs

One of the more common operations using the array processor is the Fast Fourier Transform (FFT). We have adopted the FPS convention for real-to-complex FFTs in packing the real part of the last complex value into the imaginary part of the first value in the array. This is allowed because the imaginary part of the first and last values are always zero. This convention allows the use of the same AP memory space for the input and output arrays from a real-to-complex FFT. For 2-D FFTs this convention is not followed and the Complex FFT of a Real N×M image is (M×2) × (N/2+1) in size.

We also adopt the convention for FFTs that the second half of a one dimensional array comes first, and that the center is N/2+1, where N is the number of elements in the array (always a power of two). In two dimensions, this means basically that the center of the array is at the corners with the first element of an NX x NY array being (NX/2+1, NY/2+1). An exception to this is that the AIPS two-dimensional FFT routine DSKFFT expects the normal order when transforming from the sky plane to the aperture plane (reverse transform).

The AIPS utility routine DSKFFT will FFT a two-dimensional array kept in a DFIL.INC system scratch file. Real-to-complex, complex-to-real, or full complex transforms can be done in either direction. Real-to-complex and complex-to-real transforms return the maximum and minimum values in the output array and real-to-complex transforms can return either the amplitude, real part, or full complex version of the result. Details of the call sequence for DSKFFT are given at the end of this chapter.

The FFT routines require data without blanking, in an array which is a power of two on a side. In addition, the center of an image in a cataloged file may not be in the required (NX/2+1, NY/2+1) position which will produce a phase ramp in the transformed array. Two AIPS utility routines are useful in this case (1) PEAKFN which finds the location of the peak of an image near the center (say of a dirty beam) and (2) PLNGET which will subimage a cataloged file, zero fill the excess, and rotate the center of the image. Detailed descriptions of these routines are given at the end of this chapter.

12.4 Pseudo-Array Processor and Vector Computers

Many modern scientific computing systems are becoming available which have integrated vector hardware and vectorizing compilers; there is little need for array processors on these machines. In addition, many older, scalar systems running AIPS have no array processor either. For these reasons, it is necessary to have software emulations of the functions of the array processor; this emulation of an array processor is called the Pseudo array processor (or Pseudo AP).

The pseudo AP consists of a Fortran common containing memory, which is used as the AP memory, and a set of routines, which perform the same operations on the contents of the "AP memory" as are done by the corresponding true AP routines. Because of the layering into the "Q-routines", the higher level routines need not know if they are using a true or pseudo AP.

There are a number of differences in programming an array processor and writing the same software in Fortran, especially using vectorizing compilers. Microcoding an AP is more flexible than using a vectorizing compiler, but is very much more difficult. On the other hand, memory is very limited on APs, but is more available to the CPU. The following sections will discuss some of the aspects of Q-routines on a vector computer; particular attention will be paid to issues relating to the performance of software.

12.4.1 Vectorization

What is Vectorization?

A vector operation is an operation on a one-dimensional array of values as a whole rather than as individual elements. In practice, the operations are done sequentially using pipelined hardware, but it is useful to think of the operations as being simultaneous. Vectorization is an operation performed by a compiler which takes code describing scalar operations and compiles instructions to do the same operation in vector hardware. In general, compilers will only vectorize explicit loops, so software to be vectorized needs to be cast into a form that the compiler will recognize.

Vector Hardware

The actual hardware used for vector operations differs greatly from one vendor to another, but the backbone of any vector unit is a pipelined arithmetic unit. The operations in this unit are divided into discrete stages (typically around 5) and, in each machine cycle, another operation is begun. Thus, once "the pipe gets rolling", a result comes out each cycle. In addition, different pipelines may run in parallel, either independently or with the output of one going into the other, a process called chaining (or a linked triad on CDC machines). The combination of these various forms of parallelism may result in a speedup of typically 10 to 20 over scalar operations (on IBM vector units this factor is about 4).

Another common, but not universal, feature of vector machines is the vector register (CDC machines don't have vector registers). A vector register is an array of high speed memory elements residing in the CPU, which are the source or destination of vector operations. Vector registers may be either fixed length (32 - 128 elements) or variable, with the number of vector registers being traded off against the length of each.

Vector Operations

There is no unique and universal set of vector operators, but there are a number which are sufficiently common and useful to warrant discussion. Vector operations can be classified into a number of categories: editing functions, arithmetic/logical functions, and reduction functions.

For this discussion, editing functions will include the operations of loading from, or storing data to, memory or of manipulating the elements of a vector. Operations in this class are summarized in the following:

1. LOAD: Load an array of values from memory into a vector register. There may be a constant stride or increment between elements in memory.

2. STORE: Store the contents of a vector register into memory. There may be a constant stride allowed.

3. GATHER: Load an array of values from memory whose addresses are specified into a register; the addresses are contained in another register.
4. SCATTER: Store the contents of a vector register in memory in a specified list of addresses.
5. COMPRESS: Remove elements from a vector based on an array of flags called the vector mask.
6. MERGE: Form a vector from two input vectors based on a vector mask.

Arithmetic/logical vector functions are arithmetic or logical operations on the elements of a vector for unary operations and between the corresponding elements for binary operations. Common operations in this class are given in the following:

1. + : Add the elements of two vectors.
2. − : Subtract the elements of two vectors.
3. ∗ : Multiply the elements of two vectors.
4. AND : Logical AND the elements of two vectors.
5. OR : Logical OR the elements of two vectors.

Reduction operations are those which produce a scalar from a vector. The common reduction operations include the following:

1. SUM : Sum the elements of a vector
2. PROD : Multiply the elements of a vector
3. MIN : Find the minimum value in an vector
4. MAX : Find the maximum value in an vector

None of these operations are explicitly specified in Fortran (maybe in Fortran-8x), but must be recognized by the compiler. This requires both smart compilers and programmers who write in a style recognizable to the compiler. Fortunately, most vector compilers give reasonably understandable diagnostics when they don't vectorize a loop.

Difficulties with Vectorization

There are a number of typical barriers to efficient vectorization of which the programmer should be aware. Brief descriptions of some of these follow.

Dependencies A very common and serious problem in vectorization is the presence of dependencies. There is an implicit assumption in the function of vector hardware that each elemental operation in a vector operation is independent of all the others. An example where this is not true is the following:

```
      DO 10 LOOP = 2,LIMIT
         A(I) = A(I-1) + B(I)
 10      CONTINUE
```

In this example, each input value of A is the result of the previous operation. This loop cannot vectorize as written.

A problem in vectorizing the AIPS Q routines is that all "AP memory" values are in the same array with pointers and increments being passed to each routine. Most compilers, when faced with this, will declare that an apparent dependency exists; there could, in fact, be dependencies if incorrect values of the pointers were passed. This problem is solved by including the ZVND.INC include before each loop not containing a real dependency. The contents of this include declares to the compiler that no dependency exists. All vectorizing compilers have such directives and several examples are given later in this chapter. It is very important not to declare that no dependency exists when one does; if you do, the compiler will believe you and you'll get what you've got coming. Another include, ZVD.INC may be used to inhibit vectorization if necessary.

Short Loops The efficiency of vectorization depends on the length of the vectors being operated on. There is a fixed time cost for starting an operation before the first result is available. For very long loops this cost is mostly hidden, for short loops this cost can dominate. The length of a vector which results in an effective speed per element of half of the maximum is referred to as the vector half length of the machine. This value varies from about 7 (Cray XMP) to 100 or more (some configurations of CDC Cyber). This parameter of the system is generalized to the description of short vector (Cray XMP, Convex C1) and long vector (CDC Cyber, Cray 2) machines.

For reasons given above, it is desirable to have loop lengths as long as possible. For nested loops, it is frequently possible to move the longest loop to the innermost loop, although smart compilers, such as the Convex compiler, will attempt to vectorize nested loops. The gather and scatter operations can frequently be used to make loops longer. Gather or scatter operations can usually be triggered using a Fortran array containing the indices in an array. A value named NSHORT in the DDCH.INC included common gives the shortest vector length that should be vectorized on the current hardware.

Branches in Loops Logic complications, such as branches inside of loops, may cause difficulties to compilers. Some older compilers refused to vectorize a loop which contained any IF statements; more recent compilers can vectorize statements of the form:

```
IF (A(I).GT.0.0) B(I) = A(I)
```

IF statements which may cause a branch are likely to inhibit vectorization. Whenever possible, such complications should be moved out of loops or replaced with Gather/Scatter and Compress operations.

Tuning

Most systems running on vector computers contain performance analysis tools (PROFILE on Convex, FLOWTRACE on Cray) which allow the programmer to determine how the CPU time in a program is being spent. Only routines which take a significant fraction of the total program execution time need to have much effort expended on them. A performance analyzer is an extremely useful tool for making a program run faster, although some care needs to be taken in interpreting the output of the performance analysis tools. The analysis process itself may distort the results; very short routines which are called very many times may appear to use significantly more time than is actually the case.

12.4.2 Memory Use

The following sections discuss the efficient use of memory both in general terms and in the AIPS pseudo-AP in particular.

Memory Heirarchy

All computer systems contain several types of data storage elements with different access times. The amount of storage in each category increases as the access time increases. The efficiency of a program will be seriously affected by the efficiency of use of these storage elements; the more often the desired data is in one of the shorter access time memory elements, the faster the program will run. After vectorization, efficient use of memory is the most important factor in determining the speed of the CPU for a given program. This issue is complicated by the fact that the distinction between different speed memories will be hidden by the compiler and operating system.

Not all systems have all types of memory described below, but all have some hierarchal structure. The various types of memory will be discussed in order of decreasing speed.

1. Registers

   The registers in the CPU are the highest speed memory in any system; usage of the registers is controlled by the compiler or by hand coding a routine in assembly language. Careful coding of loops can result in much of the work being done using data already in the registers, rather than all operations needing to access slower speed memory.

Many vector compilers use a "strip mining" technique in which a pass is made through entire loop, processing a number of elements equal to the vector register length each pass. This increases the amount of chaining possible and reduces the time spent waiting for data from slower access memory. The programmer can assist this by putting all related work not requiring logic flow control in one loop rather than several.

2. Cache memory

   Some computers have a limited amount of high speed memory called cache memory. When the cpu requests a value from memory, a block of data (typically 8 words) containing that value is read from the main memory into cache memory and then the referenced value is passed to the cpu. If the value referenced is already in cache memory, it is simply sent to the CPU. The computer system will assure that the contents of main memory track those in cache.

   A parameter used to describe the efficiency of the use of cache is the "cache hit rate", which is the percentage of the time the datum desired resided in the cache. The cache hit rate can be increased by noting that an entire block of memory was copied to cache; thus, sequential access of memory will increase the cache hit rate. In Fortran arrays, the first axis varies the most rapidly in memory; therefore, inner loops over the first axis are the most efficient.

3. Memory

   All systems contain relatively large amounts of storage elements called, simply, the memory. These are mostly semiconductor devices, although older systems used magnetic doughnuts called cores; memory is sometimes referred to as "core memory". This type of storage element will be referred to as physical memory in the following.

4. Virtual Memory

   A number of systems increase the apparent size of a program's memory by means of virtual memory. Virtual memory consists of data storage elements which reside outside of the physical address space of the program. Such storage is referenced in units of pages (typically 512 to 4096 bytes) and is transferred into the programs addressable physical memory when first referenced (called a "page fault"). When the maximum allowable amount of physical memory is reached by a program, the page of memory least recently accessed is removed from physical to virtual memory.

   The physical storage of data in virtual memory may be one or more of several forms. If the requested page has been used before and still exists in the systems semiconductor memory, that page will be restored to the program's physical memory. Some systems (IBM vector machines) may contain a large amount of semiconductor memory exclusively for this use. Virtual memory pages not kept in semiconductor memory will be stored on a mass storage device such as a magnetic disk.

   Virtual memory is best used in the way overlaying is used on machines without virtual memory; once a page is read in, it is used many times and is only removed from physical memory after its use has (at least temporarily) ended. Badly coded routines may page fault on every line of code; this is likely to occur in multi-dimensional arrays larger than will fit in the program's physical memory and the inner loop is over the last axis. *Programs with a very high page fault rate will run very slowly and may destroy the response of the system for all users.* Virtual memory abuse is one of the most serious and antisocial crimes a programmer can commit.

Work Vectors

In synthesis data processing, there are a number of operations which do not vectorize in the forms they are originally specified. A prime example of this is the gridding of uv data, which is discussed in detail in AIPS Memo No. 33. For these difficult cases, it is frequently possible to reformulate the algorithm into a vectorizable form using temporary arrays, such as addresses for gather/scatter operations. These arrays are referred to as work vectors in the following. This technique has been heavily used in the specialized Q routines for various machines. There are several WKVECn work vectors located in the common containing the "AP memory" (DAPC.INC. In all vector machine pseudo APs, the memory in these work arrays greatly exceeds the memory in the primary "AP memory".

Extension of AP Memory

Since there is considerable memory allocated to the work vectors, which are used only for specialized routines, it is desirable to use this memory for other purposes. To this end, the work arrays are placed in the same common after the main "AP memory" in the DAPC.INC include. The amount of this memory (referred to as secondary AP memory) is given in the DDCH.INC include common as KAP2WD in units of 1024 real words. Before using this secondary AP memory, the programmer should be sure that it is not also being used as work vectors; none of the routines described in this chapter use work vectors overlapping the secondary AP memory.

Memory allocation

On computers without virtual memory, all of a program must fit in physical memory; on some of these, Crays in particular, memory is an expensive resource. In addition, programs requiring large amounts of memory will spend most of the time rolled out, if the machine is busy (which Crays usually are). For these reasons, it is desirable to be able to reduce or expand the amount of memory used for the AP on Cray XMPs. This is done using blank common, which Cray will allow to be dynamically allocated, for the "AP" memory. The routine QMEMSZ is then used to allocate or deallocate memory. Since this common will be loaded in its fully expanded form, it is a good idea to call QRLSE as the first executable statement in tasks using the AP to release the AP memory.

12.5 Example of the Use of the AP

In the following example of the use of the array processor, the elements of two scratch files containing arrays N x M using the DFIL.INC system (numbers ISCRA and ISCRB) are added and returned to the file ISCRC. This makes very inefficient use of the AP, but illustrates the basic features. This example also illustrates use of APIO.

```
      SUBROUTINE FILADD (ISCRA, ISCRB, ISCRC, N, M, IRET)
C-----------------------------------------------------------------------
C   FILADD adds two N x M arrays in the DFIL.INC scratch files
C   ISCRA and ISCRB and writes the result in scratch file ISCRC.
C   Inputs:
C      ISCRA     I  DFIL.INC scratch file number of first input file.
C      ISCRB     I  DFIL.INC scratch file number of second input file.
C      ISCRC     I  DFIL.INC scratch file number of output file.
C      N         I  Length of a row in the array
C      M         I  Number of rows in the array.
C   Output:
C      IRET      I  Return error code 0=>OK, otherwise APIO error code.
C-----------------------------------------------------------------------
      INTEGER ISCRA, ISCRB, ISCRC, N, M, IRET
C
      INTEGER   INCR, FLIST(22,3), LOOP, APLOCA, APLOCB, APLOCC, LEN,
      REAL      BUFF1(4096), BUFF2(4096), BUFF3(4096)
C----------------------------------------------------------------------
C                                     Setup for APIO
      CALL FILL (22, 0, FLIST)
C                                     Size of array
      FLIST(5,1) = N
      FLIST(6,1) = M
C                                     Buffer size (4096 words)
      FLIST(13,1) = 4096 * 2
C                                     Copy for other files
```

```
      CALL COPY (22, FLIST(1,1), FLIST(1,2))
      CALL COPY (22, FLIST(1,1), FLIST(1,3))
C                                   Set LUNs
      FLIST(1,1) = 16
      FLIST(1,2) = 17
      FLIST(1,3) = 18
C                                   Set DFIL.INC file numbers
      FLIST(2,1) = ISCRA
      FLIST(2,2) = ISCRB
      FLIST(2,3) = ISCRC
C                                   Set AP pointers,
      APLOCA = 0
      LEN = N
C                                   Address for B file
      APLOCB = APLOCA + LEN
C                                   Address for C file
      APLOCC = APLOCB + LEN
C                                   Grab AP
      CALL QINIT (0, 0, KAP)
C                                   Start loop.
      DO 100 LOOP = 1,M
C                                   File A to AP
         CALL APIO ('READ', FLIST(1,1), APLOCA, BUFF1, IRET)
C                                   Check for error
         IF (IRET.NE.0) GO TO 999
C                                   File B to AP
         CALL APIO ('READ', FLIST(1,2), APLOCB, BUFF2, IRET)
C                                   Check for error
         IF (IRET.NE.0) GO TO 999
C                                   Wait for data transfer
         CALL QWD
C                                   Add
         CALL QVADD (APLOCA, 1, APLOCB, 1, APLOCC, 1, LEN)
C                                   Wait for operation to finish
         CALL QWR
C                                   Write result to disk.
         CALL APIO ('WRIT', FLIST(1,3), APLOCC, BUFF3, IRET)
C                                   Check for error
         IF (IRET.NE.0) GO TO 999

 100     CONTINUE
C                                   Release the AP
      CALL QRLSE
C                                   Close files.
      CALL APIO ('CLOS', FLIST(1,1), APLOCA, BUFF1, IRET)
C                                   Check for error
      IF (IRET.NE.0) GO TO 999
      CALL APIO ('CLOS', FLIST(1,2), APLOCB, BUFF2, IRET)
C                                   Check for error
      IF (IRET.NE.0) GO TO 999
      CALL APIO ('CLOS', FLIST(1,3), APLOCC, BUFF3, IRET)
C
 999  RETURN
      END
```

12.6 INCLUDEs

12.6.1 DAPC.INC

Several versions of this include exist for the pseudo AP implementations on different systems. This INCLUDE defines the pseudo AP memory.

```
C                                                           Include DAPC
C                                       'Vanilla' Pseudo AP version.
      REAL   APCORE(65536), RWORK(4096)
      INTEGER   APCORI(1), IWORK(4096), SPAD(16)
      COMPLEX CWORK(2048)
      COMMON /APFAKE/RWORK, APCORE
      COMMON /SPF/ SPAD
      EQUIVALENCE (APCORE, APCORI), (RWORK, IWORK, CWORK)
```

```
C                                       Convex C1 Version
      INTEGER   APSIZE, PKPWRD, PKPWD2
C                                       "1 MWord" size
      PARAMETER (APSIZE=262144)
C                                       "256 KWord size"
C      PARAMETER (APSIZE=65536)
C                                       PKPWRD="primary" AP size
      PARAMETER (PKPWRD=64)
C                                       PKPWD2="secondary" memory
      PARAMETER (PKPWD2=((APSIZE*5)-PKPWRD*1024)/1024)
      REAL   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   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)
      COMMON /APFAKE/ APCORE, WKVEC1, WKVEC2, WKVEC3, WKVEC4, WKVEC5,
     *   WKVEC6, WKVEC7, WKVEC8, WKVEC9
      COMMON /SPF/ SPAD
      EQUIVALENCE (APCORE, APCORI),
     *   (WKVEC1, IWVEC1), (WKVEC2, IWVEC2), (WKVEC3, IWVEC3),
     *   (WKVEC4, IWVEC4), (WKVEC5, IWVEC5), (WKVEC6, IWVEC6),
     *   (WKVEC7, IWVEC7), (WKVEC8, IWVEC8), (WKVEC9, IWVEC9)
```

```
C                                       Alliant FX Version
      INTEGER   APSIZE, FFTSZE
      PARAMETER (APSIZE=65536)
      REAL   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   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)
      COMMON /APFAKE/ APCORE, WKVEC1, WKVEC2, WKVEC3, WKVEC4, WKVEC5,
     *   WKVEC6, WKVEC7, WKVEC8, WKVEC9
      COMMON /SPF/ SPAD, FFTSZE
      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 DAPC
```

12.6.2 DBPR.INC

```
C                                                           Include DBPR
C                                          AP roller common
      REAL      DELAY
      DOUBLE PRECISION XTLAST, DELTIM
      LOGICAL   TRUEAP
      COMMON /BPROLC/ XTLAST, DELTIM, DELAY, TRUEAP
C                                                           End DBPR.
```

12.6.3 DDCH.INC

```
C                                                           Include DDCH.
C                                          AIPS system parameters
      CHARACTER SYSNAM*20, VERNAM*4, RLSNAM*8, DEVNAM(10)*48,
     *   NONNAM(8)*48, MAPNAM(12)*48, SYSTYP*4, SYSVER*8
      HOLLERITH HBLANK
      DOUBLE PRECISION DBLANK
      REAL   XPRDMM, XTKDMM, TIMEDA(15), TIMESG, TIMEMS, TIMESC, TIMECA,
     *   TIMEBA(4), TIMEAP(3), FBLANK, RFILIT(14)
      INTEGER   NVOL, NBPS, NSPG, NBTB1, NTAB1, NBTB2, NTAB2, NBTB3,
     *   NTAB3, NTAPED, CRTMAX, PRTMAX, NBATQS, MAXXPR(2), CSIZPR(2),
     *   NINTRN, KAPWRD, NWDPDP, NBITWD, NCHLIN, NTVDEV, NTKDEV, BLANKV,
     *   NTVACC, NTKACC, UCTSIZ, BYTFLP, USELIM, NBITCH, NCHPRT,
     *   KAP2WD, MAXXTK(2), CSIZTK(2), DASSGN(8,15), SPFRMT, DPFRMT,
     *   NSHORT, TTYCAR, DEVTAB(50), FTAB(1024)
      COMMON /DCHCHM/ SYSNAM, VERNAM, SYSTYP, SYSVER, RLSNAM,
     *   DEVNAM, NONNAM, MAPNAM
      COMMON /DCHCOM/ DBLANK, XPRDMM, XTKDMM, TIMEDA, TIMESG, TIMEMS,
     *   TIMESC, TIMECA, TIMEBA, TIMEAP, FBLANK, RFILIT, HBLANK,
     *   NVOL, NBPS, NSPG, NBTB1, NTAB1, NBTB2, NTAB2, NBTB3, NTAB3,
```

```
     *   NTAPED, CRTMAX, PRTMAX, NBATQS, MAXXPR, CSIZPR, NINTRN,
     *   KAPWRD, NWDPDP, NBITWD, NCHLIN, NTVDEV, NTKDEV, BLANKV,
     *   NTVACC, NTKACC, UCTSIZ, BYTFLP, USELIM, NBITCH, NCHPRT,
     *   KAP2WD, MAXXTK, CSIZTK, DASSGN, DEVTAB, SPFRMT, DPFRMT,
     *   NSHORT, TTYCAR
      COMMON /FTABCM/ FTAB
C                                                           End DDCH.
```

12.6.4 ZVND.INC

This include is a vectorizing compiler directive to ignore the apparent dependency in the following loop; several versions are given:

```
C                                       Include for compiler directive
C                                       to ignore apparent dependency.
C                                       Dummy version (nonvectorizing)

C                                       Convex version
C$DIR NO_RECURRENCE

C                                       COS version
CDIR$  IVDEP

C                                       Alliant version
CVD$L NODEPCHK
CVD$L NOSYNC
```

12.6.5 ZVND.INC

This include is a vectorizing compiler directive to assert that there is a dependency in the following loop; several versions are given:

```
C                                       Force scalar compilation
C                                       compiler directive
C                                       Dummy version (nonvectorizing)

C                                       Convex version
C$DIR SCALAR

C                                       COS version
CDIR$  NEXTSCALAR

C                                       Alliant version
CVD$L NOVECTOR
```

12.7 Routines

12.7.1 Utility Routines

APCONV

Is a disk based, two dimensional convolution routine. The image to be convolved and the FFT of the convolving function are passed to APCONV along with two scratch files. NOTE: Uses AIPS LUNs 18, 23, 24, 25.

```
APCONV (NX, NY, LI, LW1, LW2, LO, LC, FACTOR, JBUFSZ, BUFF1, BUFF2,
  *   BUFF3, SMAX, SMIN, IERR)
Inputs:
   NX       I     The number of columns in the input image (must be
                  a power of 2).
   NY       I     The number of rows in the input image.
   LI       I     File number in /CFILES/ of input.
   LW1      I     File number in /CFILES/ of work file no. 1
                     size = (4*NX x NY+2).
   LW2      I     File number in /CFILES/ of work file no. 1
                     size = (4*NX x NY+2).
   LO       I     File number in /CFILES/ of output.
   LC       I     File number in /CFILES/ of FFT of convolving fn.
                     size = (4*NX x NY+2).
   FACTOR   R     Normalization factor for convolving function; i.e.
                  is multiplied by the transform of the convolving
                  function
   JBUFSZ   I     Size of BUFF1,2,3 in AIPS bytes.  Should be
                  large, at least 8192 words.
Output:
   BUFF1    R(*)  Working buffer
   BUFF2    R(*)  Working buffer
   BUFF3    R(*)  Working buffer
   SMAX     R     Maximum value in the output file.
   SMIN     R     Minimum value in the output file.
   IERR     I     Return error code, 0 => OK, otherwise error.
```

APIO

Transfers image-like data between disk files and the array processor. The file open and close and initialization logic are all contained in this routine. Information about the file and the the desired properties of the I/O are contained in the array FLIST. APIO can access either cataloged 'MA' type files or scratch files using the /CFILES/ common (DFIL.INC) system.

APIO can handle arbitrary row lengths with any size buffer larger than one disk block. This is done by breaking up the logical records. NOTE: it is important that data read with APIO have been written by APIO using the same buffer if the buffer is shorter than the row size. The problem is that APIO will break up logical records if they are longer than the buffer size and MDISK may leave blank space on the disk if the shorter logical record does not fill a disk sector.

Useage notes:

1. Opening the file. If APIO determines that the file is not open it will do so. The file can be either a cataloged file or a scratch file using the /CFILES/ common system. If the catalog slot number given in FLIST is 0 or less the file is assumed to be a scratch file. File open assumes that the file type is 'MA' (if cataloged), file is opened patiently without exclusive use.

2. Initialization. APIO initializes the I/O using the values in FLIST when it opens the file. It may be initialized again at any time using OPCODE 'INIT'. Also switching between 'READ' and 'WRIT' will force flushing the buffer ('WRIT') and initialization. Any initialization when the current operation is 'WRIT' will cause the buffer to be flushed.

3. Closing the file. The file may be closed with a call with opcode 'CLOS'. If the file is being written and a 'CLOS' call is issued, APIO will flush the buffer. This means that if APIO is being used to write to a disk it MUST be called with OPCODE='CLOS','READ', or 'INIT' to flush the buffer. NOTE: All pending AP operations MUST be complete before calling APIO with opcode 'CLOS'.

4. AP timing calls. APIO calls APWD before getting data from or sending data to the AP but does not call APWR. The calling routine should call APWR as appropriate.

```
APIO (OPCODE, FLIST, APLOC, BUFFER, IRET)
 Inputs:
    OPCODE   C*4  Code for the desired operation.
                  'INIT' forces the initialization of the I/O.
                  'READ' reads a logical record from the disk and
                         sends it to the specified AP location.
                  'WRIT' Gets data from the AP and writes it to
                         disk.
                  'CLOS' Closes the file and flushes the buffer if
                         necessary.
    FLIST(22) I   An array containing information about the file
                  and the I/O. Parts are to be filled in by the
                  calling routine and are for use by APIO.
                    1 = LUN, must be filled in,
                    2 = disk number for catalogs files or
                        /CFILES/ number for scratch files.
                    3 = catalog slot number for cataloged files,
                        .LE. 0 indicates that the file is a scratch
                        file.
                    4 = Unused
                    5 = Length of a logical record (row) in pixels.
                    6 = Number of rows in a plane.
                    7 = value to be added to 1 for the block offset.
                    9-12 = the window desired in the image, 0's=>
                        all of image.  The logical records must fit
                        in the buffer and be smaller than BUFSZ
                        bytes to subimage rows.
                    13 = Buffer size in bytes.
        Used by APIO:
                    14 = FTAB pointer
                    15 = Number of MDISK calls per logical record.
                    16 = Current OPCODE,
                         0 = none, INIT on next call
                         1 = READ
                         2 = WRITE
                    17 = actual length of logical row.
                    18-22 = Spare.
    APLOC      I   Base address in AP for data.
    BUFFER(*)  R   Working buffer.
 Output:
    IRET       I   Return code, 0 => OK or
                                1 = Bad OPCODE,
                                2 = Attempt to window too large
                                    a file.
                                3 = Buffer too small (<NBPS bytes)
                                MDISK error codes + 10, or
                                MINI3 error codes + 20, or
                                ZOPEN error codes + 30.
```

DSKFFT

Is a disk based, two dimensional FFT. If the FFT all fits in AP memory then the intermediate result is not written to disk. Input or output images in the sky plane are in the usual form (i.e. center at the center, X the first axis). Input or output images in the uv plane are transposed (v the first axis) and the center-at-the-edges convention with the first element of the array the center pixel. NOTE: Uses AIPS LUNs 23, 24, 25.

```
DSKFFT (NR, NC, IDIR, HERM, LI, LW, LO, JBUFSZ, BUFF1, BUFF2,
  *   SMAX, SMIN, IERR)
Inputs:
   NR      I     The number of rows in input array (# columns in
                 output).  When HERM is TRUE and IDIR=-1, NR is
                 twice the number of complex rows in the input file
   NC      I     The number of columns in input array (# rows in
                 output).
   IDIR    I     1 for forward (+i) transform, -1 for inverse (-i)
                 transform.
                 If HERM = .TRUE. the follwing are recognized:
                    IDIR=1 keep real part only.
                    IDIR=2 keep amplitudes only.
                    IDIR=3 keep full complex (half plane)
   HERM    L     When HERM = .FALSE., this routine does a complex to
                 complex transform.
                 When HERM = .TRUE. and IDIR = -1, it does a
                 complex to real transform.  When HERM = .TRUE. and
                 IDIR = 1, it does real to complex.
   LI      I     File number in /CFILES/ of input.
   LW      I     File number in /CFILES/ of work file (may equal LI)
   LO      I     File number in /CFILES/ of output.
   JBUFSZ  I     Size of BUFF1, BUFF2 in bytes.  Should be large
                 at least 4096 words.
Output:
   BUFF1   R(*)  Working buffer
   BUFF2   R(*)  Working buffer
   SMAX    R     For HERM=.TRUE. the maximum value in output file.
   SMIN    R     For HERM=.TRUE. the minimum value in output file.
   IERR    I     Return error code, 0 => okay, otherwise error.
```

PEAKFN

Searches a region around the center of an image to locate the pixel location of the maximum. Will handle data cubes.

```
PEAKFN (LUN, VOL, CNO, IDEPTH, CATBLK, BUFFER, JBUFSZ, PEAKX, PEAKY,
  *   IRET)
Inputs:
   LUN      I        Logical unit number to use.
   VOL      I        Disk on which image resides.
   CNO      I        Catalog slot number of image.
   IDEPTH   I(5)     Depth in image of desired plane.
   CATBLK   I(256)   Catalog header block for image.
   JBUFSZ   I        Size of the BUFFER in bytes
Output:
   BUFFER   R(*)     Real work buffer
```

```
   PEAKX    R        X coordinate of peak pixel location.
   PEAKY    R        Y coordinate of peak pixel location.
   IRET     I        Return code, 0=> OK, otherwise error.
```

PLNGET

Reads a selected portion of a selected plane parallel to the front and writes it into a specified scratch file. The output file will be zero padded and a shift of the center may be specified. If the input window is unspecified (0's) and the output file is smaller than the input file, the NX x NY region about position (MX/2+1-OFFX, MY/2+1-OFFY) in the input map will be used where MX,MY is the size of the input map. NOTE: If both XOFF and/or YOFF and a window (JWIN) which does not contain the whole map, XOFF and YOFF will still be used to end-around rotate the region inside the window. The image header is taken from the disk catalog AND explicitly will not handle blanked images.

```
PLNGET (IDISK, ICNO, CORN, JWIN, XOFF, YOFF, NOSCR, NX, NY,
  *   BUFF1, BUFF2, BUFSZ1, BUFSZ2, LUN1, LUN2, IRET)
Inputs:
   IDISK    I      Input image disk number.
   ICNO     I      Input image catalog slot number.
   CORN     I(7)   BLC in input image (1 & 2 ignored)
   JWIN     I(4)   Window in plane.
   XOFF     I      offset in cells in first dimension of the center
                   from MX/2+1 (MX 1st dim. of input win.)
   YOFF     I      offset in cells in second dimension of the center
                   from MY/2+1 (MY 2nd dim. of input win.)
   NOSCR    I      Scratch file number in common /CFILES/ for outpu.
   NX       I      Dimension of output file in X
   NY       I      Dimension of output file in Y
   BUFF1    R(*)   Work buffer
   BUFF2    R(*)   Work buffer.
   BUFSZ1   I      Size in AIPS bytes of BUFF1
   BUFSZ2   I      Size in AIPS bytes of BUFF2
   LUN1     I      Logical unit number for input file
   LUN2     I      Logical unit number to use for output
Output:
   IRET     I      Return error code, 0 => OK,
                    1 = couldn't copy input CATBLK
                    4 = couldn't open output map file.
                    5 = couldn't init input map.
                    6 = couldn't init output map.
                    7 = read error input map.
                    8 = write error output map.
                    9 = error computing block offset
                    10 = output file too small.
Common: (DCAT.INC)
   /MAPHDR/ CATBLK  is set to the input file CATBLK.
```

QROLL

If it is time for the current task to roll out of the AP then QROLL copies the first NWORDs of AP MD memory to a scratch file, gives up the AP, does a task delay for DELAY, grabs an AP and loads the scratch file back into the AP. If NWORD ≥ 0, then the AP is not rolled and the AP is given up and then reassigned.

NOTE: APROLL is called by QROLL and uses commom /CFILES/ for the scratch file. NOTE: LUN 8 is used for I/O and a AIPS "map" I/O slot is opened if the AP memory is actually rolled.

No action is taken if the task is using a "pseudo" AP.

IMPORTANT NOTE: QROLL (and APROLL) work properly only for floating point data. Integer values rolled will not be restored correctly.

```
QROLL (NWORD, BUFFER, BUFSZ, IRET)
Inputs:
   NWORD    I      Number of words of AP memory to save.
                   If <= 0 the contents of the AP memory are
                   not saved.
   BUFFER   R(*)   Work buffer.
   BUFSZ    I      Size of BUFFER in bytes.
Inputs from COMMON /BPROLC/ (DBPR.INC)  (set by QINIT)
   TRUEAP   L      True if a real AP (to be rolled)
   XTLAST   D      Real time AP assigned (min).
   DELTIM   D      Time interval between rolls (min).
   DELAY    R      Time to delay task (seconds).
Output:
   IRET     I      Return error code, 0 => OK
                     2 => couldn't reload AP.
```

12.7.2 Array Processor Routines

The names and functions of the general purpose AP routines are given in the following brief list. A number of specialized routines for CLEANing, gridding uv data and model computations have been omitted.

1. QGET (HOST, AP, N, TYPE) Transfers data from AP to host.
2. QGSP (I, NREG) Reads the value of an SPAD register (FPS and pseudo).
3. QPUT (HOST, AP, N, TYPE) Transfers data from host to AP.
4. QRFT (UDATA, UFT, UPH0, NFT, NDATA) Computes real, inverse Fourier transform from arbitrarily spaced data.
5. QWAIT (no arguments) Suspends host until all transfers and computations are complete.
6. QWD (no arguments) Suspends host until all transfers of data are complete.
7. QWR (no arguments) Suspends host until all computations are complete.
8. QBOXSU (A, I, NB, C, J, N) Does a boxcar sum on a vector.
9. QINIT (I1, I2, I3) Assigns and initializes AP.
10. QRLSE (no arguments) Releases the AP.
11. QCFFT (C, N, F) Complex FFT.
12. QCRVMU (A, I, B, J, C, K, N) Complex - real vector multiply.
13. QCSQTR (CORNER, SIZE, ROW) In-place transpose of square complex matrix.
14. QCVCMU (A, I, B, C, J, N) Multiplies a complex scalar times the complex conjugate of a complex vector producing a real vector.
15. QCVCON (A, I, C, K, N) Takes complex conjugate of complex vector.

16. QCVEXP (A, I, C, K, N) Takes complex exponential of real vector.
17. QCVJAD (A, I, B, J, C, K, N) Adds a complex vector to the complex conjugate of another complex vector.
18. QCVMAG (A, I, C, K, N) Complex vector magnitude squared.
19. QCVMMA (A, I, C, N) Finds the maximum square modulus of a complex vector.
20. QCVMOV (A, I, C, K, N) Copies one complex vector to another.
21. QCVMUL (A, I, B, J, C, K, N, F) Multiplies two complex vectors.
22. QCVSDI (A, I, B, C, J, N) Divides a weighted complex vector by a complex scalar, weight is multiplied by the amplitude of the scalar.
23. QCVSMS (A, I, B, C, J, D, K, N, FLAG) Subtracts a real vector times a complex scalar from a complex vector.
24. QDIRAD (A, IA, B, N) Complex directed add.
25. QHIST (A, I, C, N, NB, AMAX, AMIN) Computes histogram of a vector.
26. QLVGT (A, I, B, J, C, K, N) Logical vector greater than.
27. QMAXMI (A, I, MAX, MIN, N) Finds maximum and minimum values in a vector.
28. QMAXV (A, I, C, N) Finds maximum in an array.
29. QMEMSZ (NWORDS) Expands or contracts the size of the Pseudo AP memory.
30. QMINV (A, I, C, N) Finds minimum in an array.
31. QMTRAN (A, I, C, K, MC, NC) Matrix transpose.
32. QPHSRO (A, I, B, J, PHAS0, DELPHS, N) Imposes a phase gradient on a complex vector.
33. QPOLAR (A, I, C, K, N) Rectangular to polar conversion.
34. QRECT (A, I, C, K, N) Polar to rectangular conversion.
35. QRFFT (C, N, F) Real to complex, or vice versa, Fast Fourier Transform.
36. QSVE (A, I, C, N) Sum of vector elements.
37. QSVESQ (A, I, C, N) Sum of the square of the elements of a vector.
38. QVABS (A, I, C, K, N) Vector absolute value.
39. QVADD (A, I, B, J, C, K, N) Vector add.
40. QVCLIP (A, I, B, C, D, L, N) Vector clip.
41. QVCLR (C, K, N) Vector clear.
42. QVCOS (A, I, C, K, N) Vector cosine.
43. QVDIV (A, I, B, J, C, K, N) Vector division.
44. QVEXP (A, I, C, K, N) Vector exponentiation.
45. QVFILL (A, C, K, N) Vector fill.
46. QVFIX (A, I, C, K, N) Vector real to integer.

47. QVFLT (A, I, C, K, N) Vector integer to real.
48. QVIDIV (A, I, D1, D2, B, J, N) Divides a vector by the product of two scalar integers.
49. QVLN (A, I, C, K, N) Vector natural logarithm.
50. QVMA (A, I, B, J, C, K, D, L, N) Vector multiply and add.
51. QVMOV (A, I, C, K, N) Copies one vector to another.
52. QVMUL (A, I, B, J, C, K, N) Vector multiply.
53. QVNEG (A, I, C, K, N) Takes negative of a vector.
54. QVRVRS (C, K, N) Reverses a vector.
55. QVSADD (A, I, B, C, K, N) Vector scalar add.
56. QVSIN (A, I, C, K, N) Vector sine.
57. QVSMA (A, I, B, C, K, D, L, N) Vector scalar multiply and add.
58. QVSMAFX (A, I, B, C, D, L, N) Vector scalar multiply, add and fix.
59. QVSMSA (A, I, B, C, D, L, N) Vector scalar multiply, scalar add.
60. QVSMUL (A, I, B, C, K, N) Vector scalar multiply.
61. QVSQ (A, I, C, K, N) Vector square.
62. QVSQRT (A, I, C, K, N) Vector square root.
63. QVSUB (A, I, B, J, C, K, N) Subtracts two vectors.
64. QVSWAP (A, I, C, K, N) Swaps two vectors.
65. QVTRAN (M, N, IAD, LV) Transposes a row-stored, M x N array of row vectors of length LV.

12.7.3 AP Routine Call Sequences

A note should be made about the conventions used in the description of the routines. Data addresses are normally denoted by A, B, C, or D and their increments (stride) by I, J, K, L and an element count by N. In the descriptions of the routines, many of the values in AP memory are referred to by the name given to the variable giving the address, e.g., A(mI) is used to denote the value in memory location A + m*I. All input variables are integer unless otherwise marked.

QGET

Transfer data from AP memory to host core.

```
    QGET (HOST, AP, N, TYPE)

 Inputs:
    AP      I     Target area in AP; 0-relative, increment=1
    N       I     Number of elements
    TYPE    I     Data type:
                     0   data is I in host
                     2   data is R in host
 Output:
    HOST(*) R/I Data array in "host"
```

QGSP

Read contents of SPAD register: FPS and Pseudo AP only.

```
QGSP (I, NREG)

Inputs:
   NREG  I  SPAD register number desired
Outputs:
   I     I  Contents of the SPAD register.
```

QPUT

Transfer data from host memory to AP memory.

```
QPUT (HOST, AP, N, TYPE)

Inputs:
   AP      I     Target area in AP; 0-relative, increment=1.
   N       I     Number of elements
   TYPE    I     Data type:
                   0   data is I in host
                   2   data is R in host
   HOST(*) R/I Data array in "host"
```

QWAIT

Suspend host task until all AP I/O and computations are complete.

```
QWAIT
```

QWD

Suspend host task until all AP I/O is complete.

```
QWD
```

QWR

Suspend host task until all AP computations are complete.

```
QWR
```

QINIT

Implements AIPS AP priority for true AP, increases the task priority for AIPS batch tasks using a true AP, and assigns an AP.

```
QINIT (I1, I2, I3)

Inputs:
   I1   I    Dummy
```

```
      I2   I     Dummy
   Outputs:
      I3   I     AP number (Neg. to indicate virtual AP, i.e.,
                 not to be rolled.
```

QRLSE

Releases the AP and lowers task priority for AIPS batch tasks using a true AP.

```
QRLSE
```

QBOXSU

Do a boxcar sum on a vector; values at the ends of the vector are the sum of the values within one boxcar length of the ends.

```
      QBOXSU (A, I, NB, C, J, N)

   Inputs:
      A          input vector base address
      I          input vector increment
      NB         boxcar width
      C          output vector base address; output vector
                     should not overlap input
      J          output increment
      N          number of elements
```

QCFFT

Do an in-place complex fast Fourier transform.

```
      QCFFT (C, N, F)

   Inputs:
      C      Base address (0-rel) of complex array to transform
      N      Number of points in array (must be power of two.)
      F      Transform direction; 1 -> Forward
                                 -1 -> Backward
```

QCRVMU

Multiply the elements of a complex vector by the elements of a real vector.

```
     C(mK)   = A(mI)   * B(mJ)            m = 0 to N-1
     C(mK+1) = A(mI+1) * B(mJ)

      QCRVMU (A, I, B, J, C, K, N)

   Inputs:
      A       Source complex vector base address.
      I       Increment of A (normally 2 * integer)
      B       Source real vector base address
```

```
J      Increment of B
C      Destination vector base address
K      Increment of C (normally 2 * integer)
N      Element count
```

QCSQTR

Do an in-place transpose of square matrices of complex values.

```
   QCSQTR (CORNER, SIZE, ROW)

Inputs:
   CORNER    AP location of first corner of matrix encountered.
   SIZE      Size (number of reals) of a row or column.
   ROW       Number of locations in AP between beginnings
             of the rows.
```

QCVCMU

Multiply a scalar complex value times the complex conjugate of a vector, producing a real vector.

```
C(mJ) = B(0) * A(mI) + B(0+1) * A(mI+1)   for m = 0 to N-1

   QCVCMU (A, I, B, C, J, N)

Inputs:
   A     Source complex vector base address.
   I     Increment of A (normally 2 * integer)
   B     Address of scalar (real part)
   C     Destination real vector base address.
   J     Increment of C
   N     Element count (reals)
```

QCVCON

Take complex conjugate of a vector.

```
   C(mK)    =  A(mI)             for m = 0 to N-1
   C(mK+1)  = -A(mI+1)

   QCVCON (A, I, C, K, N)

Inputs:
   A      Source vector base address.
   I      Increment of A (normally 2 * integer)
   C      Destination vector base address
   K      Increment of C (normally 2 * integer)
   N      Element count
```

QCVEXP

Complex exponential of a vector.

```
C(mK)   = COS (A(mI))         for m = 0 to N-1
C(mK+1) = SIN (A(mI))

QCVEXP (A, I, C, K, N)
```

```
Inputs:
   A      Source vector base address.
   I      Increment of A
   C      Destination vector base address
   K      Increment of C (normally 2 * integer)
   N      Element count
```

QCVJAD

Add the elements of one complex vector to the complex conjugate of the elements of another complex vector.

```
C(mK)   = A(mI)   + B(mJ)          for m = 0 to N-1
C(mK+1) = A(mI+1) - B(mJ+1)

QCVJAD (A, I, B, J, C, K, N)
```

```
Inputs:
   A      Source vector base address.
   I      Increment of A (normally 2 * integer)
   B      Source vector base address (conjugate)
   J      Increment of B (normally 2 * integer)
   C      Destination vector base address
   K      Increment of C (normally 2 * integer)
   N      Element count
```

QCVMAG

Square the magnitude of the elements of a complex vector.

```
C(mK) = A(mI)**2 + A(mI+1)**2      for m = 0 to N-1

QCVMAG (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment (normally 2 * integer)
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QCVMMA

Find the maximum of the square modulus of a complex vector.

```
max (A(mI)**2 + A(mI+1)**2)      over m = 0 to N-1

QCVMMA (A, I, C, N)
```

```
Inputs:
   A     Source vector base address
   I     Increment of A (normally 2 * integer)
   C     Destination vector.
           0 = MAX(A ** 2) (real)
           1  = location of max (integer)
   N    Element count
Also:
        SPAD(15) = index of max.
```

QCVMOV

Copy one complex vector to another.

```
C(mK)   = A(mI)              for m = 0 to N-1
C(mK+1) = A(mI+1)

QCVMOV (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment (normally 2 * integer, >= 2)
   C    Destination vector base address
   K    C address increment (normally 2 * integer)
   N    Element count
```

QCVMUL

Multiply the elements of two complex vectors.

```
C(mK)    = A(mI) * B(mJ)   - F * A(mI+1) * B(mJ+1)
C(mK+1)) = A(mI) * B(mj+1) + F * A(mI+1) * B(mJ)
                                  for  m = 0 to N-1

QCVMUL (A, I, B, J, C, K, N, F)
```

```
Inputs:
   A    Source vector base address
   I    A address increment (normally 2 * integer)
   B    Source vector base address
   J    B address increment (normally 2 * integer)
   C    Destination vector base address
   K    C address increment (normally 2 * integer)
   N    Element count
   F    Conjugate flag, 1 => normal complex multiply
                   -1 => multiply with conjugate of A
```

QCVSDI

Divide the elements of a complex vector with weights by a complex scalar. The complex vector is expected to have data in the order real, imaginary, weight. The weight is multiplied by the amplitude of the complex scalar. This is used for AIPS uv data.

```
C(mJ)   = (1./(B(1)**2+B(2)**2)) * (A(mI)  *B(1) + A(mI+1)*B(2))
C(mJ+1) = (1./(B(1)**2+B(2)**2)) * (A(mI+1)*B(1) - A(mI)  *B(2))
C(mJ+2) = A(mI+2) * SQRT(B(1)**2+B(2)**2)       for m = 0 to N-1

   QCVSDI (A, I, B, C, J, N)

Inputs:
   A      Source vector base address.
   I      Increment of A (normally 3)
   B      Source scalar address.
   C      Destination vector base address
   J      Increment of C (normally 3)
   N      Element count
```

QCVSMS

Subtract the elements of a real vector times the elements of a complex scalar from a complex vector, alternately i (SQRT(-1)) times the real vector times the complex scalar is subtracted from the complex vector. Since the element count is expected to be small, the looping is not very efficient.

```
If FLAG > 0
   D(mK)   = A(mI)   - B(1) * C(mJ)
   D(mK+1) = A(mI+1) - B(2) * C(mJ)     for m = 0 to N-1

If FLAG < 0
   D(mK)   = A(mI)   + B(2) * C(mJ)
   D(mK+1) = A(mI+1) - B(1) * C(mJ)     for m = 0 to N-1

   QCVSMS (A, I, B, C, J, D, K, N, FLAG)

Inputs:
   A      Source complex vector base address.
   I      Increment of A (normally 2 * integer)
   B      Source  complex scalar address.
   C      Source real vector base address
   J      Increment of C
   D      Destination complex vector base address
   K      Increment of D (normally 2 * integer)
   N      Element count
   FLAG   Flag, if < 0 multiply complex scalar by i
```

QDIRAD

Do a complex directed add: adds a complex vector to a complex vector whose addresses are given in the first vector.

```
B(A(mIA))   = B(A(mIA))   + A(mIA+1)   for m = 0 to N-1
B(A(mIA)+1) = B(A(mIA)+1) + A(mIA+2)
```

```
QDIRAD (A, IA, B, N)

Inputs:
   A   Source vector base address
          0 => address (integer) to be added to
               (address is zero relative)
          1,2 => complex value (reals)
   IA  Increment for A (normally 3)
   B   Destination vector base address
   N   Element count
```

QHIST

Compute the histogram of a vector: histogram element (NB-1)∗(DATA-MIN)/(MAX-MIN), where DATA is the data value, is incremented.

```
QHIST (A, I, C, N, NB, AMAX, AMIN)

Inputs:
   A        Source vector base address.
   I        A address increment.
   C        Histogram base address
               Histogram must be cleared before first call.
   N        Element count for A
   NB       Number of bins in histogram
   AMAX     Address of histogram maximum.
   AMIN     Address of histogram minimum.
```

QLVGT

Logical vector greater than.

```
C(mK) = 1.0    if A(mI) >  B(mJ)
C(mK) = 0.0    if A(mI) =< B(mJ)   for m = 0 to N-1

QLVGT (A, I, B, J, C, K, N)

Inputs:
   A    Source vector base address
   I    A address increment
   B    Source vector base address
   J    B address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QMAXMI

Search the given vector for maximum and minimum values.

```
QMAXMI (A, I, MAX, MIN, N)
```

```
Inputs:
   A        Source vector base address
   I        Increment of A
   MAX      Location for maximum.
   MIN      Location for minimum.
   N        Element count.
```

QMAXV

Find maximum value of a vector and address of the maximum.

```
      QMAXV (A, I, C, N)

   Inputs:
      A    Source vector base address
      I    A address increment
      C    Destination  base address
              C(0) = Max (A(mI))  m = 0 to N-1
              C(1) = address. also in SPAD 15.
      N    Element count
```

QMEMSZ

manipulates the size of the blank common used for the "AP" memory and vector work space. The size of the blank common is expanded to NWORDS, if that is larger than the current size, or is reduced to zero, if NWORDS is zero.
Note: this routine should be called only from Q routines. At present, it is only available for Cray XMPs under COS.

```
      QMEMSZ (NWORDS)

   Inputs:
      NWORDS  I  The number of words desired in the blank common
   Input from common /SPF/ (include D/CAPC.INC)
      MEMSIZ  I  The current size of the blank common.
                 -1 => minimum size.
   Output to common /SPF/
      MEMSIZ  I  The current size of the blank common.
                 -1 => minimum size.
```

QMINV

Find minimum value of a vector and address of the minimum.

```
      QMINV (A, I, C, N)

   Inputs:
      A    Source vector base address
      I    A address increment
      C    Destination  base address
              C(0) = Max (A(mI))  m = 0 to N-1
              C(1) = address. also in SPAD 15
```

```
N    Element count
```

QMTRAN

Transpose a matrix.

```
C((p+qMC)K) = A((q+pNC)I)
                  for  p = 0 to MC-1
                  and  q = 0 to NC-1

QMTRAN (A, I, C, K, MC, NC)

Inputs:
   A    Source matrix base address
   I    A address increment
   C    Destination  matrix base address
   K    C address increment
   MC   Number of columns of A
   NC   Numbers of rows of A
```

QPHSRO

Add a phase gradient to a complex array.

```
B(j) = A(j)*EXP(-i*(PHAS0+j*DELPHS))    for j = 0 to N-1

   or

B(mJ)   = A(mI) * cos(P0+mDP) - A(mI+1) * sin(P0+mDP)
B(mJ+1) = A(mI) * sin(P0+mDP) + A(mI+1) * cos(P0+mDP)
                              for m = 0 to N-1
     where cos(P0) = PHAS0(0),   sin(P0) = PHAS0(0+1)
           cos(DP) = DELPHS(0),  sin(DP) = DELPHS(0+1)

   QPHSRO (A, I, B, J, PHAS0, DELPHS, N)

Inputs:
   A         Source vector base address.
   I         Increment of A (normally 2 * integer)
   B         Destination base address.
   J         Increment of B (normally 2 * integer)
   PHAS0     Address of complex unit vector with
                phase PHAS0
   DELPHS    Address of complex unit vector with
                phase DELPHS
   N         Element count
```

QPOLAR

Rectangular to polar conversion.

```
C(mK)   = SQRT   (A(mI)**2 + A(mI+1)**2)
C(mK+1) = ARCTAN (A(mI+1) / A(mI))        for m = 0 to N-1

QPOLAR (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment (normally 2 * integer)
   C    Destination vector base address
   K    C address increment (normally 2 * integer)
   N    Element count
```

QRECT

Polar to rectangular vector conversion:

```
COS (tab+fract) = COS(tab)*COS(fract) - SIN(tab)*SIN(fract)
SIN (tab+fract) = SIN(tab)*COS(fract) + COS(tab)*SIN(fract)

C(mK)   = A(mI) * COS (A(mI+1))
C(mK+1) = A(mI) * SIN (A(mI+1))    for m = 0 to N-1

QRECT (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment (normally 2 * integer)
   C    Destination vector base address
   K    C address increment (normally 2 * integer)
   N    Element count
```

QRFFT

Does an in-place real-to-complex forward or complex-to-real inverse FFT.

```
QRFFT (C, N, F)
```

```
Inputs:
   C    Base address of source and destination vector
   N    Real element count (power of 2)
   F    flag, 1=>forward FFT, -1=> reverse FFT.
```

QRFT

Compute a real, inverse fourier transform from arbitrarily, but uniformly, spaced data.

```
QRFT (UDATA, UFT, UPHO, NFT, NDATA)
```

```
Inputs:
   UDATA   AP base address of input data.
   UFT     AP base address of output F. T.
   UPHO    AP base address of phase information for F. T.
              0 = COS((TWOPI/(NG*NFT))*(1-ICENT)(1-BIAS))
```

```
              1 = SIN((TWOPI/(NG*NFT))*(1-ICENT)(1-BIAS))
              2 = COS((TWOPI/(NG*NFT))*(1-ICENT))
              3 = SIN((TWOPI/(NG*NFT))*(1-ICENT))
              4 = COS((TWOPI/(NG*NFT))*(1-BIAS))
              5 = SIN((TWOPI/(NG*NFT))*(1-BIAS))
              6 = COS((TWOPI/(NG*NFT)))
              7 = SIN((TWOPI/(NG*NFT)))
              ICENT = center pixel of grid
              BIAS = center of data array (1 rel)
              NG = No. tabulated points per cell.
   NFT     Number of FT points
   NDATA   Number of data points.
```

QSVE

Sum the elements of a vector

```
   C = SUM (A(mI))  m = 0 to N-1

   QSVE (A, I, C, N)

Inputs:
   A      Source vector base address.
   I      Increment of A
   C      Destination scalar address
   N      Element count
```

QSVESQ

Sum the squares of the elements of a vector

```
   C = SUM (A(mI) * A(mI)) for m = 0 to N-1

   QSVESQ (A, I, C ,N)

Inputs:
   A      Source vector base address.
   I      Increment of A
   C      Destination scalar address
   N      Element count
```

QVABS

Take the absolute value of the elements of a vector.

```
   C(mK) = ABS (A(mI))  for m = 0 to N-1

   QVABS (A, I, C, K, N)

Inputs:
   A    Source vector base address
   I    A address increment
```

```
C    Destination vector base address
K    C address increment
N    Element count
```

QVADD

Add the elements of two vectors.

```
C(mK) = A(mI) + B(mJ)  for m = 0 to N-1

QVADD (A, I, B, J, C, K, N)
```

```
Inputs:
   A    First source vector base address
   I    A address increment
   B    Second source vector base address
   J    B address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVCLIP

Limits the values in a vector to a specified range.

```
D(mL) = B      if A(mI) <  B
      = A(mI)  if B     <= A(mI)  <  C
      = C      if C     <= A(mI)    for m = 0 to N-1

QVCLIP (A, I, B, C, D, L, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   B    Address of lower limit
   C    Address of upper limit
   D    Destination vector base address
   L    D address increment
   N    Element count
```

QVCLR

Fill a vector with zeroes.

```
C(mK) = 0   for m = 0 to N-1

QVCLR (C, K, N)
```

```
Inputs:
   C     Destination vector base address
   K     C address increment
   N     Element count
```

QVCOS

Take the cosine of elements in a vector.

```
C(mK) = COS (A(mI))   for m = 0 to N-1

QVCOS (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVDIV

Divide the elements of two vectors

```
C(mK) = B(mJ) / A(mJ)   for m = 0 to N-1

QVDIV (A, I, B, J, C, K, N)
```

```
Inputs:
   A    First source vector base address
   I    A address increment
   B    Second source vector base address
   J    B address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVEXP

Exponentiate the elements of a vector.

```
C(mK) = EXP (A(mI))   for m = 0 to N-1

QVEXP (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVFILL

Fill a vector with a constant.

```
C(mK) = A   for m = 0 to N-1
```

```
QVFILL (A, C, K, N)
```

```
Inputs:
   A    Source scalar base address
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVFIX

Convert the elements of a vector from real to integer.

```
C(mK) = FIX (A(mI))   for m = 0 to N-1

QVFIX (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVFLT

Convert the elements of a vector from integer to real.

```
C(mK) = FLOAT (A(mI))   for m = 0 to N-1

QVFLT (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVIDIV

Divide the given vector by the product of two integers.

```
B(mJ) = A(mI) / (D1 * D2)     for m = 0,N-1

QVIDIV (A, I, D1, D2, B, J, N)
```

```
Inputs:
   A      Source vector base address.
   I      Increment for A
   D1     First dividend. Actual value, not an address.
   D2     Second dividend. Actual value, not an address.
   B      Destination vector base address.
   J      Increment for B
```

```
N       Element count.
```

QVLN

Take the natural logarithm of the elements of a vector.

```
C(mK) = LOGe (A(mI))    for m = 0 to N-1

QVLN (A, I, C, K, N)
```

```
Inputs:
   A      Source vector base address.
   I      Increment of A
   C      Destination vector base address
   K      Increment of C
   N      Element count
```

QVMA

Multiply two vectors and add a third.

```
D(mL) = (A(mI) * B(mJ)) + C(mK)   for m = 0 to N-1

QVMA (A, I, B, J, C, K, D, L, N)
```

```
Inputs:
   A    First source vector base address
   I    A address increment
   B    Second source vector base address
   J    B address increment
   C    Third source vector base address
   K    C address increment
   D    Destination vector base address
   L    D address increment
   N    Element count
```

QVMOV

Copy the elements of one vector to another.

```
C(mK) = A(mI)   for m = 0 to N-1

QVMOV (A, I, C, K, N)
```

```
Inputs:
   A      Source vector base address.
   I      Increment of A
   C      Destination vector base address
   K      Increment of C
   N      Element count
```

QVMUL

Multiply the elements of two vectors.

```
C(mK) = A(mJ) * B(mJ)   for m = 0 to N-1

QVMUL (A, I, B, J, C, K, N)

Inputs:
   A    First source vector base address
   I    A address increment
   B    Second source vector base address
   J    B address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVNEG

Take the negative of the elements of a vector.

```
C(mK) = - A(mI)   for m = 0 to N-1

QVNEG (A, I, C, K, N)

Inputs:
   A    Source vector base address
   I    A address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVRVRS

Reverse the elements in a vector.

```
C(mK) = C((N-m)K)  for m = 0 to N-1

QVRVRS (C, K, N)

Inputs:
   C    Source and destination vector base address
   K    C address increment
   N    Element count
```

QVSADD

Add a scalar to the elements of a vector

```
C(mK) = B + A(mI)   for m = 0 to N-1

QVSADD (A, I, B, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   B    Adding scalar address
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVSIN

Take the sine of the elements of a vector.

```
C(mK) = SIN (A(mI))    for m = 0 to N-1  (A in radians)

QVSIN (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVSMA

Multiply the elements of a vector by a scalar and add to the elements of another vector.

```
D(mL) = (A(mI) * B) + C(mK)    for m = 0 to N-1

QVSMA (A, I, B, C, K, D, L, N)
```

```
Inputs:
   A    First source vector base address
   I    A address increment
   B    Source scalar base address
   C    Second source vector base address
   K    C address increment
   D    Destination vector base address
   L    D address increment
   N    Element count
```

QVSMAFX

Multiply the elements of a vector by a scalar, add a scalar and round to an integer.

```
D(mL) = FIX (ROUND((A(mI)*B)+C))  for m = 0 to N-1

QVSMAFX (A, I, B, C, D, L, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   B    Multiplying scalar address
```

```
  C    Adding scalar address
  D    Destination vector base address
  L    D address increment
  N    Element count
```

QVSMSA

Multiply the elements of a vector by a scalar and add a second scalar.

```
D(mL) = (A(mI)*B)+C    for m = 0 to N-1

QVSMSA (A, I, B, C, D, L, N)
```

```
Inputs:
  A    Source vector base address
  I    A address increment
  B    Multiplying scalar address
  C    Adding scalar address
  D    Destination vector base address
  L    D address increment
  N    Element count
```

QVSMUL

Multiply the elements of a vector by a scalar.

```
C(mK) = A(mI) * B   for m = 0 to N-1

QVSMUL (A, I, B, C, K, N)
```

```
Inputs:
  A    Source vector base address
  I    A address increment
  B    Multiplying scalar address
  C    Destination vector base address
  K    C address increment
  N    Element count
```

QVSQ

Square the elements of a vector

```
C(mK) = A(mI)**2   for m = 0 to N-1

QVSQ (A, I, C, K, N)
```

```
Inputs:
  A    Source vector base address
  I    A address increment
  C    Destination vector base address
  K    C address increment
  N    Element count
```

QVSQRT

Take the square root of the elements of a vector.

```
C(mK) = SQRT (A(mI)) for m = 0 to N-1

QVSQRT (A, I, C, K, N)
```

```
Inputs:
   A    Source vector base address
   I    A address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVSUB

Subtract the elements of two vectors.

```
C(mK) = B(mJ) - A(mI)    for m = 0 to N-1

QVSUB (A, I, B, J, C, K, N)
```

```
Inputs:
   A    First source vector base address
   I    A address increment
   B    Second source vector base address
   J    B address increment
   C    Destination vector base address
   K    C address increment
   N    Element count
```

QVSWAP

Swap the elements of a vector.

```
A(mI) = C(mK) and C(mK) = A(mI)   for m = 0 to N-1

QVSWAP (A, I, C, K, N)
```

```
Inputs:
   A    First source/destination vector base address
   I    A address increment
   C    Second source/destination vector base address
   K    C address increment
   N    Element count
```

QVTRAN

Transpose a (row-stored) M X N array of row vectors of length LV. The starting address is given by IAD. The algorithm works in place. It is adapted from Boothroyd's CACM ALG.#302. Other, probably better, algorithms, are CACM #'S 380 and 467, but they're not as simple to program.

```
  QVTRAN (M, N, IAD, LV)

Inputs:
  M     First dimension of the vector array
  N     Second dimension of the vector array
  IAD   Base address of the array
  LV    Length of the vectors.
```