VECTORIZATION OF PATTERN RECOGNITION AND MONTE CARLO CODE

D. Levinthal

Florida State University, Tallahassee, USA

In the two lectures I will attempt to review some of the work done by the experimental HEP group at SCRI/FSU. The first lecture will be on pattern recognition, the second on simulation techniques. It is first necessary to describe how a vector machine works briefly and particularly what constructs are used in building algorithms which run effectively on modern supercomputers. The machines which are used at FSU are a CYBER 205 and a four processor ETA-10, which is just coming on-line now. These machines run an extended FORTRAN which contains most of the constructs of FORTRAN 8. The syntax is unfortunately not the same. We have found that to write effective code on a vector machine the algorithms must be constructed from explicitly vector constructs. This is completely at odds to the view expressed by many colleagues insisting on transportability down to the smallest PC. The codes we have worked on do not rely on a vector optimizing compiler. In fact we disable it completely. This is not the technique normally used on Cray and Fujitsu machines but code conversion from the CDC V200 FORTRAN to something that the compilers can handle is straightforward.

Vector Fortran

The IVERSON operators are a set of logical vector operations and are the basic tools for constructing algorithms. It is these functions which in my mind distinguish vector supercomputers from array processors. They are listed in figure 1 with the associated FORTRAN which an optimizer would convert. The GATHER and SCATTER operations work on central memory organizing data so it can be fed to the vector pipeline hardware. Vector machines operate on sequentially stored numbers so indirectly addressed variables must be reorganized to be useful. GATHER and SCATTER operations are comparatively slow on a CDC machine and in fact timing depends on the instantaneous machine load taking between 2 and 5 machine cycles/word. I'm told on the XMP series they run a one cycle/word. Depending on the fraction of an array which is desired the COMPRESS and MERGE functions can serve the same purpose running at one result per clock cycle per vector pipe , so between 4 and twenty times faster than the GATHER and SCATTER,but must look at every array element.

The pipeline units on a 205 execute addition, subtraction, multiplication, square root and logical or boolean instructions at one 64 bit word per clock cycle per pipeline. Thus in 32 bit mode (half precision) arithmetic can produce 4 results per cycle on our 205 and 128 boolean results per cycle.

Essentially all of the standard FORTRAN functions (sin, cos, exp ...) have vector versions. There is a vector random number generator but M. Hodous of ETA wrote a couple of new ones including RNDM32 of CERNLIB which produce a number/clockcycle.

The FORTRAN supports two interesting data types that are not standard. BIT vectors use every bit in a word and are used to drive the COMPRESS, MERGE, and BOOLEAN instructions. They are also used for masked arithmetic function, ie. only execute the function if the associated bit is true. A very elegant data type is the DESCRIPTOR. This is a data type which has two components, a starting location and a length. I'll illustrate it. Say you have a DO LOOP of the following:

```
      DO 10 I=1, LENGTH
      A(ISTRT1 + I-1) = B(ISTRT2 + I-1) + C(ISTRT3 + I-1)
10    CONTINUE
```

you could declare 3 DESCRIPTORS AD, BD and CD with a declaration statement

```
DESCRIPTOR AD, BD, CD
```

and define them with a DATA statement at compilation time.

```
DATA AD/A(ISTRT1; LENGTH)/
DATA BD/B(ISTRT2; LENGTH)/
DATA CD/C(ISTRT3; LENGTH)/
```

with ISTRT1, ISTRT2, ISTRT3, and LENGTH being PARAMETERS, or instead of using data statements and parameters assign their definitions dynamically at run time with an ASSIGN statement.

```
ASSIGN AD, A(ISTRT1; LENGTH)
ASSIGN BD, B(ISTRT2; LENGTH)
ASSIGN CD, C(ISTRT3; LENGTH)
```

with ISTRT1, ISTRT2,ISTRT3 and LENGTH being variables. Then the DO LOOP can be written.

```
AD = BD + CD
```

Thus a DESCRIPTOR can be viewed as an implied loop over a single array. This is a particularly useful trick for converting scaler code to vector code and leaves a much more readable source code when you're done.

There is also a vector IF-THEN-ELSE structure which uses the form

```
WHERE (DESCRIPTOR1.LOGICALOPERATION.DESCRIPTOR2)
  DESCRIPTOR3 = DESCRIPTOR4*DESCRIPTOR5
OTHERWISE
DESCRIPTOR3=DESCRIPTOR6
END WHERE
```

the lengths of all of the DESCRIPTORS must be the same. This structure can easily be constructed from the MERGE operation.

One of the unique features on a 205 is that all of the assembler instructions can be executed from FORTRAN through the use of callable "Q8" functions. Many of the Q8 functions are packaged as simpler FORTRAN functions (fewer arguments and not so many modes of use) but for optimum speed the Q8 functions are more versatile and typically faster. This is clearly only interesting for the vector instructions of which there are approximately 100.

One of the critical features of vector machines is the start up time for vector instructions. This particular aspect forces you to maximize the vector lengths, with asymptoric speeds being reached at vector lengths greater than 1000 on a 205 or ETA machine. CRAY machines are register to register machines so the performance is maximized when all of the registers (64) are used. Consequently vector lengths which are multiples of 64 are the optimum. As a scaler machine a CYBER 205 performs at a level between 8 and 12 VAX 11/780's.

Algorithms

So we use a factor of 10 for this conversion for timing numbers. So with that introduction we'll start discussing pattern recognition algorithms. Much of the discussion will center on the FERMILAB experiment 711 which is a double arm open geometry magnetic spectrometer with two calorimeters for triggering (see figure 2). The experiment is measuring the massive dihadron continuum to determine the energy angle and flavor dependence of parton-parton scattering. It is run at an intensity of 5 x 10^6 interactions/sec

on an assortment Nuclear targets corresponding to luminosities of up to $\sim$ 5 x 10^{32} cm $^{-2}$ sec $^{-1}$.

The most time consuming part of the bulk reconstruction code is the track finding and fitting. To explain the three vector algorithms that have been developed to date a description of the wire chamber system is necessary. The magnetic spectrometer consists of a point target alleviating the need for tracking upstream of the magnets. The target is followed by two large magnets producing a total magnetic deflection of 1.35 GeV/c. The downstream tracking system consists of five stations of finely spaced wire chambers (four stations in the 1985 run). Each station has four views measuring X,Y and $\pm$ 10^0 to X, the bend view. In X and Y all of the stations have the same number of wires (256 and 544 for X and Y respectively) producing a projective system. In the two angled views the number of wires is 320 in the first two stations and 356 in the downstream three stations, thus not being purely projective.

The tracking is done in each view separately and the two dimensional projections are the coupled into three dimensional tracks and fit. While this technique is fast it suffers from being very sensitive to the hit multiplicity/plane. This is due to the low rejection of false tracks at the two dimensional level. The chamber inefficiency (85% in 1985 90% in 1987) further requires accepting two dimensional tracks with only three hits. On the 1985 data the two scaler algorithms used in the on-line system required 2 and 12 sec/event with the slower algorithm having a significantly higher track finding ability. The data taken in 1985 had an average of 2.25 found 3D tracks/event $\sim$ 10-15 hits/plane and $\sim$40 2D tracks/view found each event. In the 1987 data the fifth station was added the electronics modified and the intensity raised by a factor of 5. This resulted in the chamber multiplicity increasing by 30% to 15 to 20 hits/plane and the CPU time of the second algorithm increasing to 50 sec/event. (Scaler algorithm times are on a VAX 11/780). Scaler algorithms of this type connecting hits pairwise and searching consume CPU time in proportion to the number of hits/plane raised to some large power due to the redundancy needed for efficiency.

Vector Algorithms

We have developed three algorithms for doing the pattern recognition on the CYBER 205 we can access. All of the algorithms work by looking for every possible track in the spectrometer defined from a Monte Carlo generated list. The first takes explicit advantage of projective geometry and is described in reference 1 in detail. I'll give only a brief description of how the algorithm works here. Figure 3 illustrates the principle involved. In a projective geometry manifested as a projective grid there exist families of valid tracks which have the property that the members of the family can be generated successively by incrementing all of the cell numbers by one from the previous member. The technique then is to define a list of seed tracks on the grid and for each seed track to do a vector loop of one cell displacements and checking if a sufficient number of those cells are occupied to declare a track found. In X for example the hit wires are projected onto a perfectly projective software grid. To ensure finding efficiency the cells of each grid are in fact overlapping, mapping onto an integer number of wire spacings. Thus a single hit wire will "light" several cells. This is handled with predefined mappings and inverse mappings. This was necessary because the small angle views were not projective having varying numbers of wires. In a grid structure with 180 cells and 4 planes, you start with

plane 1 cell 1 and plane 4 cell 1 and have a family of 180 tracks like.

(1,1,1,1) (2,2,2,2) (3,3,3,3) .... (180,180,180,180) a second family of 179 tracks.
(1,1,2,2) (2,2,3,3) (3,3,4,4) ... (179,179,180,180) and a related family of 179 tracks
(1,2,2,2) (2,3,3,3) (3,4,4,4) .... (179,180,180,180)

and so on.

Each of these families can be checked in a vector loop because the data of on/off for each successive track on a fixed plane is by construction adjacent.

Consider a single view with 180 cells and 4 planes with the data stored in a 2 dimensional array PLANE (180,4). What is stored is a bit pattern for which wires actually lit each cell. There are NTRK families of tracks with starting cell numbers stored in the 2 dimensional array ISTRT(NTRK,4). Further, there is a single array LENGTH(NTRK) defining the number of tracks in each family, a status array NSTATUS(NTOT), where NTOT is the total number of tracks in all families i.e., the sum of LENGTH, and a BIT array IGOOD to define whether a track is found or not. NSTATUS is used to count how many lit cells are on each track.

```
Thus NSTATUS (1;NTOT) = 0
   ITRK = 1
   DO 100 JFAM = 1, NTRK
   LEN = LENGTH (JFAM)
   DO 50 IPLN = 1,4
   ASSIGN DESC, PLANE (ISTRT(JFAM,IPLN),IPLN;LEN)
C.... now the vector loop....
   WHERE (DESC.NE.0)
   NSTATUS(ITRK;LEN) = NSTATUS(ITRK;LEN) + 1
   END WHERE
C....
 50 CONTINUE
   ITRK = ITRK + LEN
100 CONTINUE
   IGOOD(1;NTOT) = NSTATUS(1;NTOT) .GE.3
```

While the 2 dimensional finding was simple and very fast ($\sim$ msec/event) it operated at the level of 3 wire wide roads and required a sorting algorithm which was not brought to a high enough efficiency to satisfy us, particularly as a second algorithm was developed.

The second algorithm is extremely general and is based on the use of the GATHER and SCATTER operations. It is possible, in fact quite easy, to generate a list of all possible tracks at the wire level. This list consists of the wire numbers for each plane for every track. Assume there are NTOT unique tracks (for example) in a four plane system in a single view. A track list can be constructed ITRKLST (NTOT, 4) where the value of ITRKLST(ITRK,IPLN) is the wire number on plane IPLN for track number ITRK. Starting with a compressed list of NHIT lit wires in IWIRE(NHIT) a UNIT vector ONE is expanded into array HITS(NWIRE,4) where NWIRE is the total number of wires in a plane in that view (assumed here to be the same for all planes but not essential). The SCATTER function is used on an array which is initially zeroed. This array can now be used as the source of a GATHER function to assemble the data in the format of ITRKLST in an array ITRKDAT. The valves on a given row are the zeroes or ones corresponding to the wires associated with that track, being off or on respectively. The algorithm now looks something like this:

Note: The first and third descriptors (first and third arguments) are only used for a starting location in GATHER and SCATTER operations, the lengths are ignored.

```
C.... initialize HITS and ONE....
      HITS(1,1;4*NWIRE) = 0.
      ONE(1;NWIRE) = 1.
      NSTATUS(1;NTOT) = 0.
      DO 100 IPLN = 1,4
C.... define some DESCRIPTORS for neatness....
      NHT = NHIT(IPLN)
      IFRST = IFIRST(IPLN)
      ASSIGN DHITS, HITS(1,IPLN;NWIRE)
      ASSIGN DIWIRE, IWIRE(IFRST;NHT)
      ASSIGN DONE, ONE(1;NWIRE)
C.... SCATTER Unit Vector....
      DHITS = Q8VSCATR(DONE,DIWIRE;DHITS)
C.... Some more DESCRIPTORS to assemble TRKDAT
      ASSIGN DTRKDAT,TRKDAT(1,IPLN;NTOT)
      ASSIGN DTRKLST,ITRKLST(1,IPLN;NTOT)
      DTRKDAT = Q8VGATHR(DHITS,DITRKLST;DTRKDAT)
C.... fill NSTATUS....
      WHERE (DTRKDAT.NE.0) NSTATUS(1;NTOT) = NSTATUS(1;NTOT) + 1
  100 CONTINUE
      IGOOD(1;NTOT) = NSTATUS(1;NTOT).GE.3
```

One should note there are 13 vector instructions used in the algorithm beyond the 3 initialization loops, making timing in fact calculable! The resulting timing on this algorithm including tape unpacking, finding, fitting and DST writing was 27.5 msec/event on the CYBER 205 as opposed to ~10 sec/event on a VAX 11/780 for the scaler code of comparable efficiency.

This brings us to algorithm number 3.

The GATHER and SCATTER operations are fairly slow and a three hit track will show up in several places in the list and must be sorted out. It is possible to construct the core of the algorithm completely out of boolean operations which could run up to 700 times faster per individual result. This was investigated and in fact became the final technique.

The basic trick here is to reconstruct the track list in an expanded form using bit vectors. To store the track number vs wire number information, consider again the view with 180 cells, 4 planes, and NTOT possible tracks. Make a 3-dimensional BIT array, BITX,(NTOT, 180,4), the last index referring to each of the four X planes. Say for example track 3251 has cell 57 on it in plane one, then the value of BITX (3251,57,1) will be one and all other value on the row BITX (3251,J,1) will be zero. This defines the construction of the track list. More space is used but far less CPU time. The construction of the list of possible tracks with a hit in a given plane can now be constructed by simply executing an OR function (Q80RV) on those columns whose cells are on. Assume IWIRE (NWIRE,4) and NHIT(4) (the lit wire identities and number of lit wires), and define a 2 dimensional BIT vector for all tracks that have hits in each given plane TRKLST(NTOT,4) initialized to all zero.

The boolean vector instructions execute on the whole number of words used. So the

lists are increased in length from NTOT by the integer calculation.

```
      NWORDS = NTOT/64 + 1
      NTOTP = 64 * NWORDS
```

The additional dummy tracks (NTOTP-NTOT) have zeros for all the cell numbers and can therefore never be found.

So algorithm 3 starts with the construction of the BIT vector defining which tracks could exist with a hit in a given chamber.

```
C....Initialize TRKLST to zero.... use intrinsic function
      CALL Q8MKZ(4* NWORDS,0,TRKLST(1,1;4*NWORDS))
C....Construct TRKLST for each plane.
      DO 100 IPLN = 1,4
      NHT = NHIT(IPLN)
      ASSIGN DTRKLST,TRKLST(1,IPLN;NWORDS)
      DO 50 IHT = 1, NHT
      IW = IWIRE(IHT,IPLN)
      ASSIGN DBITX, BITX(1,IW,IPLN;NWORDS)
C .... Use assembler version for speed hence extra commas
C .... Hex code defines OR function
      CALL Q8ORV (X'02',,DTRKLST,,DBITX,,DTRKLST)
   50 CONTINUE
  100 CONTINUE
```

This block of code is equivalent to the GATHER loop of algorithm 2 but TRKLST is now a vector of BIT type. The Q8ORV function is equivalent logically to something like

```
      DTRKLST = OR (DTRKLST,DBITX)
```

of a FORTRAN 8X type of syntax. Each column of TRKLST is now a list of true/false values indicating whether a particular predefined tracks' wire in that plane is on. A four hit track would appear as a row of four TRUE values.

Explicitly then one constructs the four fold (1111) ANDs of the four columns (one per plane) and the four three fold AND corresponding to a missing plane (1110,1101,1011, 0111) producing five track list bit vectors. These can all be constructed with a total of 7 uses of the Q8ANDV instruction. The original loop takes one OR instruction per lit wire. An event with a multiplicity of ten hits on each plane would require 40 OR instructions to assemble the list and 7 AND instructions to determine which tracks existed. This would then be done four times, once for each projection.

The elimination of 3/4 tracks which are superseded by 4/4 tracks and duplicate versions of the 3/4's is the next problem (a single 3/4 corresponds to several 4/4). An encoded array is precalculated , defining the wires used on each plane for each track (thus NTOT long). We compress this array 5 times using the 5 bit vectors (1111,1110,1101,1011,0111), creating 5 encoded lists. For the first three (1110,1101,1011) which lack one plane, the tracklist was constructed in a way which the duplicate 3/4's would be adjacent. Thus by comparing the compressed encoded vector with itself, shifted by plus and minus one location and tagging, the duplicate 3/4 can be eliminated.

The remaining missing chamber case (0111) has to be handle differently due to the numbering sequence. In the end we found that at most all of the duplicates were within 30 locations of each other so a search over the limited range was done comparing the encoded track numbers. Prior to this realization a system based on the binary radix sort was used which will now be illustrated.

If the encoded numberrs can be put in numerically increasing or decreasing order the duplicate track candidates will automatically be adjacent. On a vector machine this is a surprisingly easy operation. Consider a list of integer numbers INUM(LEN) which require NBITS to represent the largest value. (I don't usually need 48 bits)

Do a bit compress using the lowest order bit, saving first all the words with 1 bits, then all the words with zero bits into a new array. Copy to the original (not nescessary) and repeat the process from lowest order bit to highest used (NBITS). The final result is in numerically decreasing order.

This can be coded as follows.

Assume an array MASK(LEN,64) which is just a 1 bit in the appropriate column,

```
      MASK(1,I;LEN) = 2**(I-1)
```

a dummy array for working DUMMY(LEN), a bit vector IBIT(LEN), and a place to store the masked data DMASK(LEN).

```
      DESCRIPTOR DLEN
      DO 10 IBT = 1, NBITS
      CALL Q8ANDV(X'00',,INUM(1;LEN),,MASK(1,I;LEN),,DMASK(1;LEN))
      IBIT(1;LEN) = DMASK(1;LEN).GT.0
C .... This is a trick to get the stopping point of the compress
      ASSIGN DLEN, DUMMY(1;0)
C .... Compress on 1 bit; ..., use assembler called version hence the extra commas ...
C .... hex code defines compress on 1
      CALL Q8CPSV(X'00',,INUM(1;LEN),,,IBIT(1;LEN),DLEN)
C .... Execute length to register instruction
      NTRUE = Q8LTOR(DLEN)
C .... Assign output to start at the next location
      ASSIGN DLEN, DUMMY(NTRUE+1;0)
C .... Compreson 0 bit hence different hex code
      CALL Q8CPSV(X'40',,INUM(1;LEN),,,IBIT(1;LEN),DLEN)
      INUM(1;LEN) = DUMMY(1;LEN)
   10 CONTINUE
```

at the end of such loop INUM will be in decreasing order

Calorimeter pattern recognition.

As an exercise we studied pattern recognition techniques for fly's eye calorimeter. There are some very straight forward algorithms based on gathers and scatters with predefined lists much alone the lines of the tracking codes which prove to be very effective. They again illustrate the point that a vectorizing optimizer (compiler) cannot tell you when to use a look up table.

The algorithm is based on building a chain, where all cells of the calorimeter are associated with their largest pulse height neighbor. A reduction on the chain will point at the root, which is the largest pulse height cell. The algorithm thus has two parts, assembling the chain and reducing the chain to a cluster with a root cell.

To assemble the chain you need to establish the largest pulse height cell in the immediate vicinity of each cell with a read out pulse height (this could be the cell itself). In the rectangular geometry we chose, each cell has 9 neighbors (including itself). A predefined list NEIGHBOR(NCELLS,9) is made specifying the identities of 9 neighbors for every

one of the NCELLS in the array. Edge cells have null cells created around them to make the sides of the array look like the interior. Assume the data comes in two ordered lists PULSEHEIGHT(*), ICELLID(*) and there are NHIT cells with pulse height. The first thing done is to scatter the PULSEHEIGHT array into a full array DATA(NCELLS)

```
      DATA(1;NCELLS) = 0
      DATA(1;NCELLS) = Q8VSCATR(PULSEHEIGHT(1;NHIT),
     *                    ICELLID(1;NHIT);DATA(1;NCELL))
```

For each element of ICELLID we want the corresponding row of NEIGHBOR. This is done with 9 gather instructions.

```
      DO 10 I = 1,9
      NEARLIST(1;I;NHIT) = Q8VGATHR(NEIGHBOR(1,I;NHIT),
     *                    ICELLID(1;NHIT);NEARLIST(1,I;NHIT))
   10 CONTINUE
```

followed by gathering the data for each of these neighbors.(accomplishing a sort of nested gather)

```
      DO 20 I=1,9
      NEARPH(1,I;NHIT)=Q8VGATHR(DATA(1;NCELLS),NEARLIST(1,I;NHIT);
     *                    NEARPH(1,I;NHIT))
   20 CONTINUE
```

You then find the largest pulse height neighbor.

```
      PHMAX(1;NHIT) = 0
      DO 30 I=1,9
      WHERE(NEARPH(1,I;NHIT).GE.PHMAX(1;NHIT))
      PHMAX(1;NHIT) = NEARPH(1,I;NHIT)
      MAXNEIGHBOR(1;NHIT) = NEARLIST(1,I;NHIT)
      END WHERE
   30 CONTINUE
```

We now have four ordered lists

```
      PULSEHEIGHT(1;NHIT)
      IDCELL(1;NHIT)
      MAXNEIGHBOR(1;NHIT)
      PHMAX(1;NHIT)
```

where MAXNEIGHBOR(I) is the identifying address of the largest pule height neighbor of IDCELL(I) and so on. These lists make a set of chaining pointers which define the clusters. There are several ways to reduce the chains to clusters perhaps the easiest to understand is as follows

Pack PULSEHEIGHT, IDCELL and MAXNEIGHBOR into a single array (called PACKDAT) with one word containing all three numbers after converting PULSEHEIGHT to integers first (note: if it is an ADC output, it already is), and with Pulseheight in the low order n bits

This array (PACKDAT) can then be sorted with a binary radix sort such that it is in decreasing order of pulse height. After unpacking, the neighbor pointers will always point to an entry higher in the list. As we progress down the list if a cell points to itself (or to

one further down, which can occur if two blocks have equal pulse height and are adjacent) we know we are at the root of a new cluster. This reduction is aided by the scatter instruction and the list of integers IZ(I)=I and can be as follows (including construction of data banks).

```
      IPOSITION(1;NHIT)=Q8VSCATR(IZ(1;NHIT),IDCELL(1;NHIT);
     *                    IPOSITION(1;NHIT))
      NCLUST=0
      DO 50 I=1,NHIT
      IF (IPOSITION(MAXNEIGHBOR(I)).GE.I) THEN
C.... NEW CLUSTER ....
         NCLUST = NCLUST + 1
C.... CORRECT THE UNUSUAL CASE ....
         MAXNEIGHBOR(I) = IDCELL(I)
         IPOSITION(I) = I
C.... INITIALIZE CLUSTER DATA BANK AND INVERTED TABLES ....
         ICLUSTER(IDCELL(I)) = NCLUST
             etc.
      ELSE
C.... SET THE NEIGHBOR TO THE NEIGHBORS' NEIGHBOR
         MAXNEIGHBOR(I) = MAXNEIGHBOR(IPOSITION(MAXNEIGHBOR(I)))
         IROOT = MAXNEIGHBOR(I)
         ICLUST = ICLUSTER(IROOT)
C.... ADD TO DATA BANK ....
      ENDIF
   50 CONTINUE
```

At this point the pattern recognition is finished as the cluster data banks have been assembled.

E711 Acceptance Monte Carlo

The acceptance tables for E711 are made in 3 dimensions, mass, rapidity and the scattering angle $\cos\theta^*$. With a typical acceptance of 17% in each good bin and approximately 240 bins (Y vs $\cos\theta^*$) for each mass bin, the required number of general events per charge state is 2 x 10^7. The typical way in which this kind of table is made is to generate the events randomly over the 6 dimensional phase space ($M, Y, \cos\theta^*, \phi, \bar{P}_t$) and integrate over the last 3. The acceptances then are calculated by taking the bin by bin ratios of the accepted/generated numbers of events. The tables are made by histogram packages. In the course of developing this code, 3 "packages" were used, HBOOK with the HFILL routines, HBOOK with the fast fill routines (HF1, HF2), and a homemade package QBOOK which is very fast but has no protections. The flow of the calculation is shown in figure XXX with the evolution of the timing shown below.

On the vector machine the only loop available was the main program loop over events. With all the subroutine calls and event cuts

```
IF (bad event) GO TO End of Loop
```

no vector optimizer would be able to get anything from this program. The basic technique for vectorizing this code was to process more than one event at a time, and if a given event was bad simply set a logical flag (Bit vector) to false, protect the calculations and continue processing even though the event has been rejected. This conversion was particularly easy if one simply declared all variables in a subroutine to be descriptors and added a large

number of declaration and data statements (for the descriptors) at the beginning of each subroutine. Further instead of sampling phase space randomly and histograming, one steps through the 3 dimensional binning of the acceptance table, randomly populating a bin with a vector of events. The acceptance of a bin is just the number of validly flagged events at the end of the tracking code. This removes the need for any histograming ie there are no cumulative random accesses

H(I(J)) = H(I(J)) + WEIGHT(J)

for which pipelining is impossible. The code is entirely vectorized and a speed up factor of $\approx$ 40 was achieved over the fastest of the scaler codes. (Fig 4)

Vectorized Technique for Calorimeter Simulation

Calorimeter Simulation

The simulation of electromagnetic and hadronic showers in calorimeters is usually one of the most time consuming parts of event simultion in HEP detectors. In order to avoid the enormous amounts of time required for detailed particle tracking shower codes (EGS, Geant, Gheisha, etc.) parameterized showers are frequently used. The parameterizations contain some number of randomly generated parameters to generate shower length and width fluctuations. In the example discussed here the Bock parameterization (ref 2) as implemented by CDF (ref 3) was used for simulating the response of the calorimeter in FNAL experiment 711.

The calorimeter

The E711 calorimeter generates the experimental trigger. The goal is to identify interactions producing massive dihadron systems to study high P_t interactions. The device consists of two calorimeters positioned symmetrically above and below the primary proton beam. Each nominally covers the regions between 22 mr to 100 mr in polar angle and $\pm 22.5^0$ in azimuth. There are four segments in depth, a Pb- scintillator EM section and 3 Fe-scintillator hadronic sections. Each segment is read out by ganging horizontal scintillator pieces and placing a PMT at each end. The EM segment has 15 such elements of varying lengths (to approximate constant azimuthal acceptance) made of 14 pieces of scintillator and 31.7 R.L. of Pb. The 3 hadronic segments are constructed similarly but have 16 elements. The front hadronic segment has 14 scintillaors in each element separated by 1.25 inch thick Fe plates. The back two have 7 scintillators in each element separated by 2.5 inch Fe plates. The calorimeter system was designed by M. Crisler of FNAL, constructed and brought into operation by M. Crisler (FNAL) and K. Turner (from FSU). The 80 ton system is illustrated in Figures 5 and 6.

Shower Simulation parameterization (hadronic)

The parameterized hadronic shower function used was of the form

$$dE/dz = wS(z)^a e^{-bS(z)} + (1-w)T(z)^c e^{-dT(z)}$$

where w represents the fraction of the shower which is in EM form from the primary interaction, S(z) the number of radiation lengths since the primary interaction and T(z)

the number of interaction lengths, a, b, c, and d all are fluctuated in accordance with ref. 3. The overall length is also fluctuated with a scaling factor but for simplicity S(z) and T(z) are scaled rather than dE/dz. The electromagnetic and hadronic components each have gaussian radial dependences with the widths of the gaussian depending on shower depth. The parameters used were seeded from ref. 3 and then optimized to agree with test beam data taken with E711 calorimeter.

Shower Generation

The shower generation is done by using two predefined 3D grids (one for the electromagnetic component, one for the hadronic component) of unequal spacing in the horizontal and vertical directions. This is to match the detector segmentation and can be tuned. The third dimension of the grid is constructed such that each layer is positioned at the scintillator planes (i.e. gaps between the absorber). All of the necessary constants describing the material in the grid are predefined (critical-energy(IZ), density(IZ), depth-in- interaction-lengths(IZ), etc...where IZ is the index of the grid defining the position in depth) so for example T(z) becomes

T(IZ) = depth-in-interaction-lengths(IZ) - T(zstart)

where T(zstart) is the interpolated depth in interaction lengths on the grid. The full grids are centered on the particle. Advantage is taken of the explicit azimuthal symmetry of a gaussian by generating only a quarter of the grid and then copying it 3 times. In the vectorized version, two arrays of the gaussian widths are calculated for every point in the generated 3-D grid. In both versions predefined arrays for R^2(IX, IY, IZ) are used. In the vector version the half precision (32 bits) Exp function is used. The copying of the quarter grid is done by the GATHER function as the pointer map can be precalculated. Normalization is done by explicit summation. This entire procedure takes $\approx$ 150 ms/shower in scalar mode and 6.4 ms/shower in vector mode for a speed up of 24. In fact an eighth of the full grid could be used and a faster vector exponential (ref 4) but as will be apparent the procedure is sufficiently fast that the estimated 20 - 40 % speed increase overall was not gone after. The time taken per shower depends on the number of grid points with the granularity being determined by the detector granularity. Consequently this procedure is independent of shower energy.

Digitization

The preceeding procedure turned the shower generation from the dominant calculation to an insignificant portion (i.e. Amdahl's law came into effect). The digitization dominates the calculation. The digitization is accomplished by simply looping through points, figuring out which detection element a point is in and adding the points' energy to the energy of that detection element. To simplify this, four summations are made over depth at quarter grid level (during shower generation before copying) corresponding to the 4 calorimeter segments. This is done twice (for the two grids) and is part of the reason for rescaling S(z) and T(z) rather than the total shower.

The vertical segmentation of the calorimeter means that the element number depends only on the vertical spatial position of a point. As the widths of the scintillator elements are all multiples of one inch, the element is determined by calculating an integer and then using a table to find the element number. The points' horizontal coordinate is then

compared to the length of its element. If the point is within the active length of its elements' scintillator, the points' energy is added to the element energy.

In scalar mode this calculation is done in a triplely nested loop.

```
Do IZ = 1, 4
Do IY = 1, nygrid
Calculate element number for that grid row
Do IX = 1, nxgrid
if (abs(X(IX)).LT.length(IELEM,IZ))
PH(IELEM,IZ) = PH(IELEM,IZ) + Energy(IX,IY,IZ)
endif
end do
```

In vector mode every point is treated individually independent of which row, column, or layer it is in. Using some precalculated arrays for distances, pointers, etc., the element number for each pointer is calculated with half a dozen vector instructions. A GATHER operation is performed collecting the element length for each point and the comparison on the length performed.

It should be noted that this calculation is essentially histogramming, and the final cumulative random access is extremely time consuming and does not vectorize unless a structure exits on the list of bins. The cumulative random access

```
PH(IBIN(IPOINT)) = PH(IBIN(IPOINT)) + E(IPOINT)
```

if executed in scalar mode will take more time than the showering and element (IBIN) calculation combined (12.5 ms/shower vs 5 ms/shower).

As the points form a grid of regular structure this difficulty can be avoided. All of the points whose energies will be accumlated into a given element are adjacently positioned in memory separated only by points which lie outside of the active area. Operationally what is done is a bit vector (logical) is created for all the points, with the value being true if the horizontal position of the point is within the active region of that points' element. A bit compress function is executed on the element list and the energy list, the resulting compressed vectors can be directly summed in vector mode. The limits of the summations being found by where the element list values change. When the digitization is done in this manner the 12.5 ms/shower taken by the cumulative random access becomes 0.6 ms/shower. At this point the vectorization was declared complete with an overall speed up of 18. Even when the chamber simulation and output routines are added, the Monte Carlo globally takes $\approx$ 22 ms/event.

Event Generation and Particle Tracking

The event generation and the particle tracking through the spectrometer consume a negligible amount of time in this calculation. This is accomplished using the same technique used for the acceptance Monte Carlo (ref 5). Rather than track a single event at a time a vector of 3000 events is tracked through with a Bit vector keeping track of whether a particle should be eliminated (ex. hitting a magnet pole face). The bad tracks are compressed out and a scalar loop goes through the good events and calls the detector simulation for each event in succession. This multi-event technique could easily have been used on the wire chamber simulation but this only takes 1 ms/event in scalar mode and was left alone.

References

ref. 1 - J.J Becker et al., Nucl. Instr. and Methods A235 (1985) 502. C.H. Georgiopoulos et al., Nucl. Instr. and Methods A249 (1986) 451.
ref. 2 - CERN-EP/80-206, R. Bock, et al
ref. 3 - CDF Memo - Description of CDF calorimetry simulation, J. Freedom, M. Eaton, May 13, 1985
ref. 4 - Supercomputer Computations Research Institute preprint - CYBER-205: A Fast Vectorized Expotential Function, B. Berg, A. Billoire, Sept. 1985.
ref. 5 - Supercomputing at F.S.U., D. Levinthal, et al, Proceddings of Asilomar Conference on High energy Computing (1986).

* * *

"IVERSON OPERATORS'

CYBER-205 Data-Handling Vector Operations

GATHER

```
      Do 10 j = 1, n
10    a(j) = b( i(j) )          <==>  A = Q8VGATHR(B , I ; A)
```

SCATTER

```
      Do 10 j = 1 , n
10    a( i(j) ) = b (j )        <==>  A = Q8VSCATR( B , I ; A)
```

COMPRESS

```
      a: 1 2 3 4 5
    bit: 0 1 0 0 1              <==>  C = Q8VCMPRS( A , BIT ; C)

      c: 2 5
```

DECOMPRESS

```
      a: 8 9
    bit: 0 1 1 0 0              <==>  Call Q8MRGV (x'01',,A,,C,,BIT,C)

  Old c: 1 2 3 4 5
  New c: 1 8 9 4 5
```

EXPAND

```
      c: 2 5
    bit: 0 1 0 0 1              <==>  A = Q8VXPND ( C, BIT ; A )

      a: 0 2 0 0 5
```

VECTOR MASK

```
     a1: 1 2 3 4
     a2: 5 6 7 8
    bit: 0 0 1 1                <==>  C = Q8VMASK ( A1, A2, BIT ; C )

      c: 5 6 3 4
```

VECTOR MERGE

```
     a1: 1 2 3 4
     a2: 5 6 7
    bit: 1 1 1 0 0 1            <==>  C = Q8VMERG ( A1, A2, BIT ; C )

      c: 1 2 3 5 6 4
```

SELECTIVE - STORE

```
      a: 1 2 3 4 5
    bit: 0 1 1 0 0              <==>  C = Q8VCTRL ( A, BIT ; C )

  Old c: 6 7 8 9 10
  New c: 6 2 3 9 10
```

Fig. 1

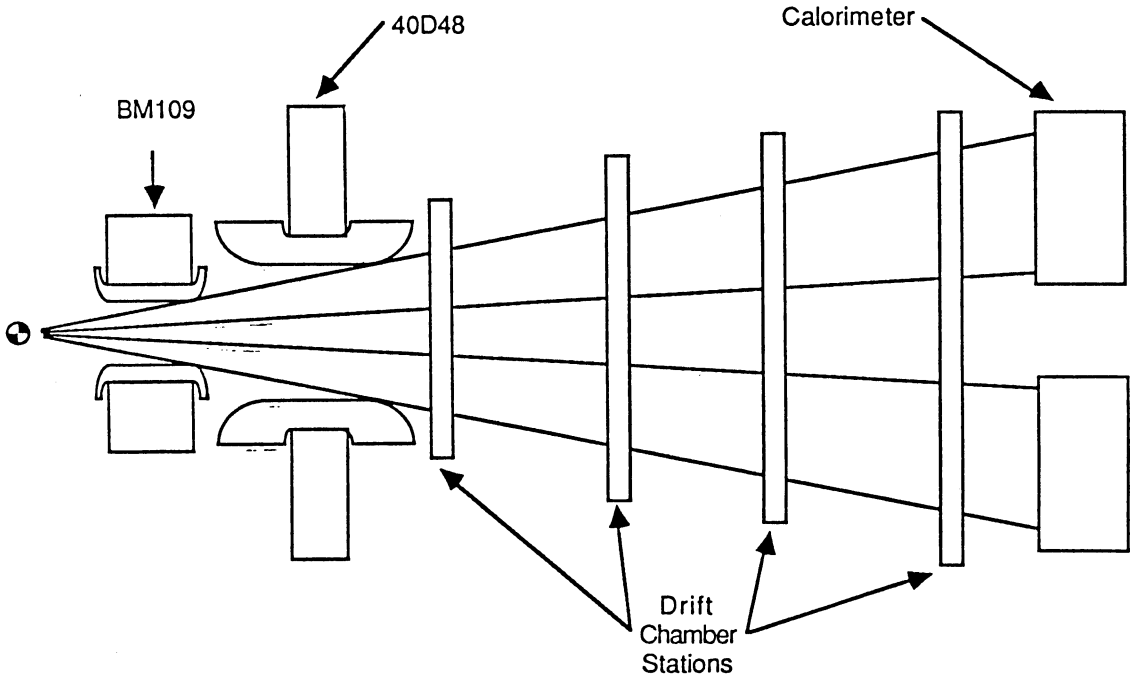

40D48
Calorimeter
BM109
Drift
Chamber
Stations

Fig. 2

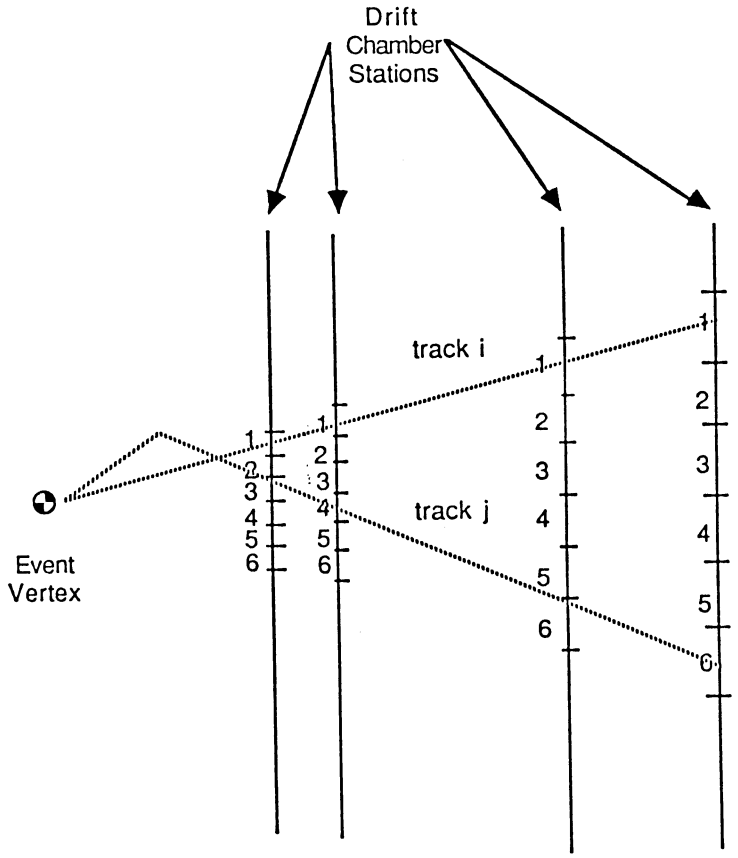
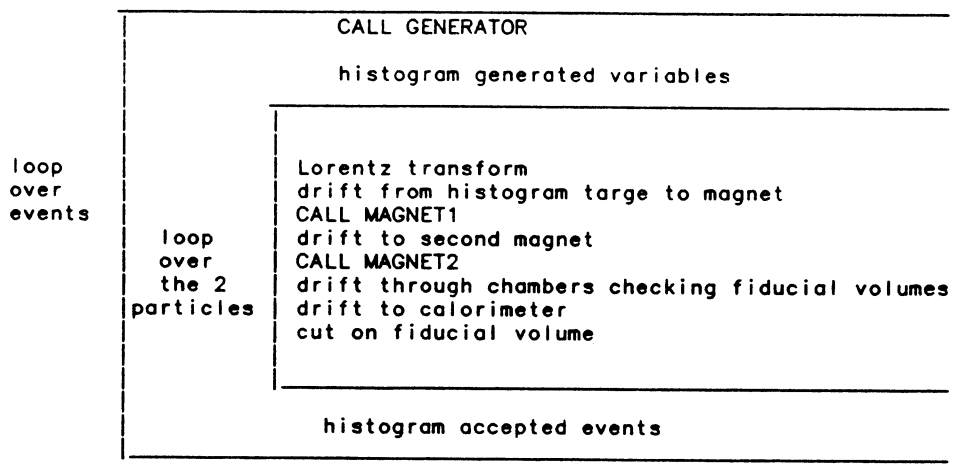

Drift
Chamber
Stations
track i
track j
Event
Vertex

Fig. 3

The logic flow is as follows:



TIMING

ORIGINAL (HBOOK/VAX 11/780)	997.9 SEC/100K EVENTS
MC (QBOOK/VAX 11/780)	383.3 SEC/100K EVENTS
MC (QBOOK/CYB 205)	49.7 SEC/100K EVENTS
MC (SCALER NO HISTOS/CYB 205)	26.3 SEC/100K EVENTS
MC (VECTOR WITH SCALER HISTOS CALLED IN DO LOOP)	23.4 SEC/100K EVENTS
MC (VECTOR WITH SCALER HISTO CODE IN LINE IN LOOP)	11.0 SEC/100K EVENTS
MC (VECTOR WITH VECTOR HISTO)	5.4 SEC/100K EVENTS
MC (VECTOR WITH NO HISTO)	2.2 SEC/100K EVENTS
MC (VECTOR,NO HISTOS HALF PRECISION)	1.3 SEC/100K EVENTS
MC (VECTOR,NO HISTO,HALF PRECISION UP LOOP TO 5K EVENTS)	1.2 SEC/100K EVENTS

Fig. 4

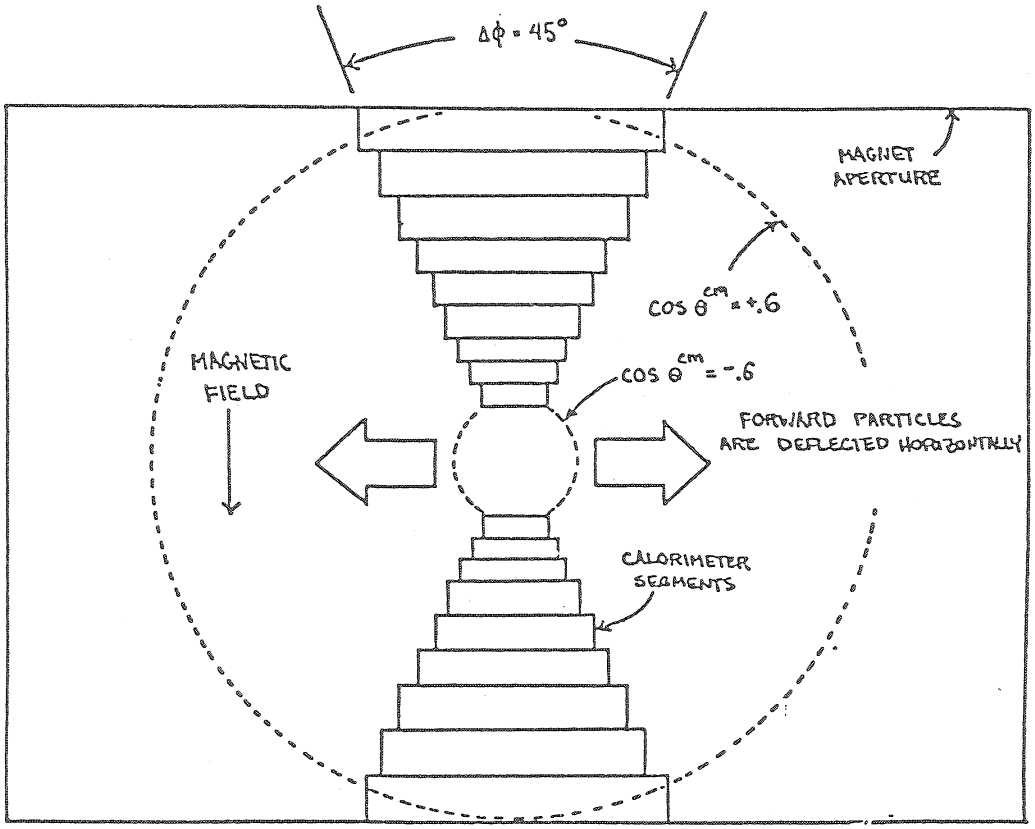


Fig. 5 A beam particles view of the detector. Transverse momentum is measured in the vertical plane, magnetic deflection is in the horizontal.

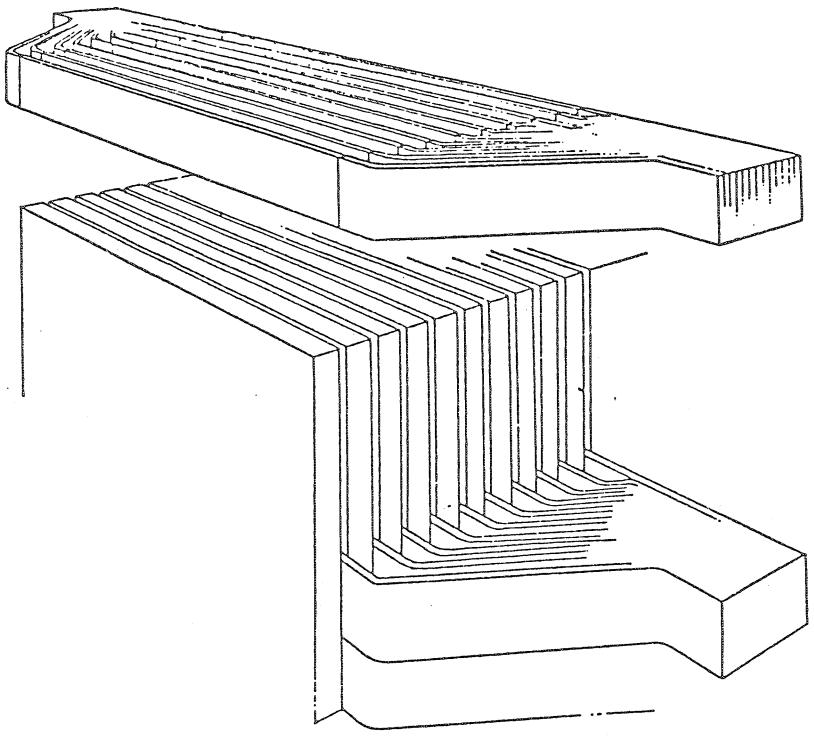


Fig. 6 Calorimeter construction detail.