Fermi National Accelerator Laboratory

FERMILAB-Pub-87/72
2300.000

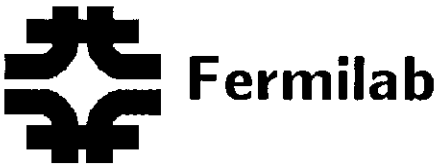
Evaluation of the FPS-164 Computer for High Energy Physics Pattern Recognition Problems*

L. Roberts
Computing Department
Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

May 1987

*Submitted to Computer Physics Communications.

Operated by Universities Research Association Inc. under contract with the United States Department of Energy



FERMILAB–Pub–87/72
May 1987

Evaluation of the FPS-164 Computer for High Energy Physics Pattern Recognition Problems

Lee Roberts

Computing Department
Fermi National Accelerator Laboratory
P. O. Box 500, Batavia, IL 60510

ABSTRACT

Vectorized pattern recognition algorithms have been compared to traditional scalar algorithms in single-view detectors on the FPS-164 Scientific Computer. The vectorization has been limited to the track recognition phase of the data analysis. Results show that the traditional scalar algorithm outperforms the vectorized algorithms on the FPS-164 in chamber multiplicities of physical interest.

Evaluation of the FPS-164 Computer for High Energy Physics Pattern Recognition Problems

I. Introduction

The application of modern computer architectures to pattern recognition problems in high energy physics has recently become a topic of considerable interest. Several recent articles, including those written by Georgiopoulos[1] *et al.*, Becker[2] *et al.*, and Zacharov[3] have outlined algorithms for data analysis programs on machines with vector and/or parallel architectures. Georgiopoulos *et al.* claim a substantial speed advantage for their (Cyber 205) vector architecture over scalar architecture (VAX-11) in the data analysis of Fermilab experiment E-711. Their conclusions run counter to the commonly-held belief among high energy physicists that *vector computers are not suitable for high energy physics pattern recognition problems.* This has stimulated interest in applying these new architectures to high energy physics problems.

The Floating Point Systems FPS-164 Scientific Computer was chosen for study because it offers an advanced computer architecture—multiple functional units—with features expected to be similar to supercomputers (Cray, Cyber 205) and because of its availability at Fermilab. The FPS-164 functions as an attached processor to a general-purpose front-end computer. At Fermilab, the front-end computer is a VAX 8600. Two methods of program execution are available on the FPS-164. These are:

1) execution of the complete job in the FPS-164
2) execution of a main program on the front-end computer (VAX) with computationally-intensive subroutines run on the FPS-164.

Programs for the FPS-164 may be written in FORTRAN (APFTN64 cross-compiler) or assembler (APAL64 cross-assembler) and are linked with the APLINK64 cross-linker. The APMATH64 subroutine library provides efficient mathematical routines for use in FORTRAN or assembly programs.

The FPS-164 architecture has three major features:

1) multiple functional units
2) multiple interconnections
3) multi-operation instructions.

There are a total of ten functional units, which include two memory units, three register

units, three computational units and two control units. Simultaneous operations can occur in each unit, and each unit may initiate a new operation each CPU cycle. Effective utilization of the FPS-164 therefore requires code that can utilize all of the functional units during each machine cycle. Efficient code can be produced by using appropriate APMATH64 routines and through careful FORTRAN programming (or APAL64 assembly programming). The APFTN64 compiler provides several levels of optimizations, including software pipelining. Software pipelining involves overlapping operations performed during one iteration of a loop with other iterations of the loop so as to utilize all of the CPU functional units during each cycle.

II. Algorithm types

A. Scalar algorithm

The scalar algorithm for high energy physics pattern recognition is the *traditional* method, i.e., the algorithms employed on the scalar computers that have been available to the experimenters. Events are analyzed sequentially by these algorithms—the contents of each event is fully examined before the next event is input. These algorithms generally require large quantities of floating point arithmetic and memory accesses during the track recognition process.

A scalar algorithm for track recognition in a typical fixed-target experiment would behave as follows. Numerical constants for the position of each detector plane, wire spacings in each plane and orientation angle of each plane would be calculated and/or stored for future use. Each view of the detector would then be searched for tracks—often several passes (searches) through the chambers are required to find tracks of varying pedigree. For example, tracks which contain hits in all planes of the detector are often found first, and the data (hits) corresponding to these tracks may be removed from the data set to eliminate confusion when searching for tracks having missing hits or different signatures. Each track consists of required *defining* hits and *confirming* hits. The number of defining hits is the minimum number required to establish a unique track. For example, a straight track requires two defining hits, whereas a track with a single bend (due to a magnet) requires three defining hits. A scalar algorithm for finding straight tracks would use the hits present in two (usually predefined) planes as possible defining hits. Using loops over

the hits in the defining planes, all combinatoric possibilities of defining hits for straight tracks would be considered. For each possible combination, the coordinates of confirming hits within the detector plane would be calculated—and the data would be searched for these hits. Confirming hits are "found" if a hit exists within an experimentally-determined distance of the computed coordinate. If a sufficient number of confirming hits is found, this combination of defining and confirming hits becomes a track. After tracks have been independently found in the each of the views, track matching must be performed on the one-dimensional view tracks to form the *real* three-dimensional tracks. The track matching operation may include some amount of track fitting to determine valid matches. After the three-dimensional tracks have been identified, the physics analysis of the event can begin.

The scalar track-finding algorithm uses a substantial number of floating-point operations. For each set of possible defining hits, the coordinates of all possible confirming hits must be calculated. Equations for track coordinates are usually linearized to reduce the amount of computation—calculations require simple multiplications and additions rather than special mathematical functions. Often-used quantities such as distances between detector planes or their ratios should be calculated once and stored, but the track coordinates are recomputed as necessary for each event. Scalar algorithms can require nearly minimum amounts of memory—memory for the raw event data, the pattern recognition output and detector position and alignment constants is necessary, but little else. Memory use is limited at the expense of CPU time for repeated floating-point operations.

Production scalar track-finding algorithms often use FORTRAN "data structure" memory-management packages (HYDRA, ZBOOK, ZEBRA) as well as other common high energy physics library routines (CERNLIB). (Histogramming packages (HBOOK) are often used as part of the physics analysis portion of the data analysis code.) These packages are not presently available on the FPS-164. The scalar algorithms tested on the FPS-164 cannot mimic the production codes in this respect. The lack of FORTRAN "data structure" packages may be an advantage for the scalar codes on the FPS-164. APFTN64 guidelines[4] for *efficient FORTRAN programs* make the following recommendations:

- Avoid using scalar variables in loops if the variables share a storage location with another variable (through the use of the EQUIVALENCE statement). Optimizations cannot be performed on such variables.

- Avoid using arrays which share storage locations with another array or variable (through the use of the EQUIVALENCE statement), especially in loops on which software pipelining can be performed (pipelinable loops).

In general, equivalence statements complicate the global data flow analysis of optimizing compilers—limiting the optimizations that can be performed.

B. Multi-scalar algorithm

The multi-scalar algorithm is derived from the above scalar algorithm using the invariant coding techniques presented by Zacharov.[3] An algorithm that is invariantly coded requires the same number of CPU cycles to process any event. This procedure may also involve restructuring of the data so that the data itself is invariant in form. Conversion of scalar variable code to scalar invariant code often requires assuming "worst" cases or performing sometimes unnecessary steps to maintain code length for all events. The scalar invariant code thus obtained can be run in parallel—many events can be processed simultaneously. Such algorithms may be useful on pipelinable machines or on SIMD (single-instruction multiple-data) parallel processors.

The computational resources required by a multi-scalar algorithm are closely related to the scalar algorithm. Memory requirements per event may be smaller—only one copy of the detector specifications may be necessary—of course, memory for the raw data and pattern recognition output of each event are still necessary. The amount of floating point computation per event will usually be larger due to the "unnecessary" work performed in order to obtain the invariant code. The goal of this technique is to keep this inefficiency small so that the speed advantages of parallel processing will yield a significant speed increase. Horizontal parallelism (or vectorization) is the goal of the invariant coding technique—efficiency can be obtained by processing many events simultaneously, using algorithms closely related to the traditional scalar algorithms described above.

The implementation of the algorithm on the FPS-164 used APMATH64 subroutines and pipelinable loops to achieve this horizontal vectorization. Most of the scalar algorithm could be made invariant. The length of vectors in APMATH64 subroutines and the length of pipelinable loops is a variable factor. APMATH64 vector routines can be most useful when replacing loops, except when short (iteration count less than 60) loops are involved. Since the FPS-164 pipelining is done in software, rather than hardware, speed

advantages for loops of special lengths (lengths corresponding to multiples of hardware pipeline lengths) are not expected. However, longer loops reduce the amount (per event) of overhead involved in APMATH64 calls and in starting pipelinable loops. In this case, the length of the loops corresponds to the number of simultaneously processed events—and memory constraints will provide a limit on the loop size.

C. Vector algorithms

Vector algorithms for pattern recognition generally use *new* approaches and techniques, as efforts to vectorize the "slow" parts of the traditional scalar algorithms are unproductive. Dictionary look-up approaches have been used in vectorizable analysis codes for experiments Fermilab E-711[1] and SLAC Mark III.[2] Both algorithms utilize a precalculated dictionary containing all tracks of physical interest through the chamber and the hits that create these tracks. A non-numerical approach can then compare the data hits to the dictionary in order to find the existing tracks. Existing data analysis codes have been successful in vectorizing the pattern recognition algorithm and the event data unpacking and packing routines. Vector algorithms for track matching and physics analysis remain to be developed, if such vectorization can be achieved. In the case of Fermilab E-711, which has the only vectorizable data analysis code actually running on a vector processor, the published program data[1] do not detail the proportion of the speed increase due to vectorization of the event data unpacking and that due to the vectorized pattern recognition algorithm.

Creation of the track dictionary is a non-trivial endeavor. All possible hit combinations that can represent tracks of interest must be enumerated. Track dictionaries for all desired track pedigrees must be created—straight tracks, one-bend tracks, etc. One quickly realizes that an enormous number of tracks can pass through a typical detector—and in a typical fixed-target detector, the dependence of track numbers on wire numbers is quadratic (twice as many wires$\Longrightarrow$four times as many tracks). The approach taken in the E-711 analysis code is to define a *virtual* detector with one-third as many wires as the real detector. (The number of straight-line tracks in the *virtual* views of E-711 is still $\{31\,744,\ 22\,528,\ 38\,912,\ 38\,912\}$[5].)

The approach taken by Georgiopoulos[1] *et al.* in the E-711 analysis code is to create

a vectorized loop through the track dictionary for each of the detector planes. On each iteration, the wire specified by the dictionary entry is checked for a hit; if the wire has been hit, a counter for the corresponding dictionary entry (track) is incremented. After this has been done for all detector planes, the counters contain the number of hits on each track through the detector. Since the E-711 detector has four planes, track counters having values of 3 or 4 are accepted as candidate tracks. These candidate tracks are then cleaned, fitted and matched in three dimensions by scalar-type code before physics analysis can take place.

A more sophisticated algorithm is used in the Mark III analysis code.[2] This algorithm was designed to be vectorizable, even though the target machine was an IBM 3081K. This algorithm uses a track dictionary that has been sorted by wire for each detector plane. Plus, an auxiliary track list is kept—this list identifies tracks that become indistinguishable when one or more of their hits are missing. Rather than loop over the track dictionary, loops over detector wires are performed. For each detector wire that has been hit, counters corresponding to all tracks that pass through that wire are incremented. In addition, a bit-mapped array where each entry corresponds to the hits in 32 tracks is also kept. Searches for found tracks now proceed by checking the bit-mapped array for banks of 32 tracks that contain hits in all planes; tracks corresponding to such an entry are then searched for entries with no missing hits—and these are accepted as candidate tracks. Then, the tracks on the auxiliary lists for these candidate tracks are eliminated from the possible tracks, and a search is performed for tracks with one missing hit. This procedure continues until all desired tracks are found.

These vector algorithms obviously require large amounts of memory for storage of the track dictionaries. The necessary storage for the Mark III algorithm is several times larger than for the E-711 algorithm, as the lists of tracks through the detector wires and the auxiliary track lists are quite large. These algorithms require relatively little floating point arithmetic—most of the pattern recognition is done using bit manipulation or logical tests. Vector algorithms may perform some excess work, particularly in regard to tracks with missing hits. Only one pass through the data is used to obtain all the possible tracks—the auxiliary lists (or an equivalent scheme) must be used to eliminate meaningless partial tracks from the "found" candidates.

Variations of both of these algorithms have been implemented on the FPS-164. The vectorized loops of the E-711 algorithm (which are Cyber 205 specific function calls) translate to pipelinable loops on the FPS-164. The Mark III algorithm on the FPS-164 depends upon pipelinable loops for execution efficiency.

III. Studies Undertaken & Results

A. Single-View Detector: 5 planes $\times$ 10 wires

A simple detector with five planes of ten wires served as the first toy model. Wires were equally spaced at 2.5 mm in all detector planes. Detector planes were separated by 90 cm. The input tracks passed through all detector planes and were randomly created (tracks did not originate from a common point). Programs using each of the above algorithms were written to find complete tracks (tracks with no missing hits) in the toy detector. Short tests of these programs were run on both the FPS-164 and various VAX computers where the code was compatible (or nearly so). The effect of APFTN64 compiler optimization level was also briefly evaluated. All programs were tested on the FPS-164 under varying conditions of track multiplicity and detector noise.

Table 1 shows representative CPU times for the pattern recognition programs on a variety of CPUs and under different optimization conditions. The details of the program contents will be described below. CPU times for versions of the programs with and without formatted I/O are also shown. (Timing tests for the various algorithms was done without formatted I/O.) It is clear that a substantial penalty is paid for formatted I/O on the FPS-164.

A single version of the scalar program was developed. Each event was scanned on input to obtain the number of hits in each chamber and to reduce each detector plane to a list of hit wires—consolidating the data into a tight, easily accessed structure. The first and last detector planes were taken as the defining planes for the detector. The traditional scalar algorithm was then applied, requiring confirming hits in all three inner planes to create the candidate five-chamber track. This program was tested briefly on the VAX and was compared to the FPS-164 results in order to gain some feeling for the relative speeds of the machines. The scalar results on the FPS-164 were expected to be of particular interest, especially when compared to the vector algorithms; a difficulty with

computer /compiler	CPU time (no output)	CPU time (output)
Scalar Algorithm		
VAX-11/785	2.22	24.11
VAX 8600	1.10	10.39
VAX 8650	0.67	7.61
FPS/opt=0	1.68	25.73
FPS/opt=4	1.13	25.08
Multi-Scalar/200 Algorithm		
FPS/opt=0	4.42	41.57
FPS/opt=4	3.16	39.96
Vector Algorithm #0		
FPS/opt=0	24.32	65.48
FPS/opt=4	14.12	54.46
Vector Algorithm #1		
FPS/opt=0	26.37	67.62
FPS/opt=4	12.80	53.42
Vector Algorithm #2		
VAX 8650	1.74	12.20
FPS/opt=0	5.20	45.86
FPS/opt=4	3.05	43.89

Table 1 CPU times in seconds for pattern recognition of five-chamber tracks on 1000 events containing 0–5 input tracks. Results with and without output demonstrate the effect of formatted I/O. Compiler optimization options are shown for the FPS-164.

interpretation of the E-711 results is the lack of a scalar algorithm which also executed on the Cyber 205—only crude arguments regarding the scalar speeds of the Cyber 205 and VAX were given. A direct comparison between scalar and vector algorithms would more satisfactorily demonstrate any advantage gained by vectorization.

The multi-scalar program developed for this toy detector had the degree of parallelism (number of simultaneously-processed events) as a parameter. The above scalar algorithm was modified to produce (nearly-)invariant code. The input events are scanned as in the scalar algorithm above, but in each plane, the *maximum* number of hits in any event must be used as the number of hits for all events. Extremely-large computational values are used as dummy hits so that track matches cannot occur for invalid data. In Table 1, the multi-scalar listing is for 200 simultaneously-processed events. The multi-scalar program was tested at several parallelism values. As expected, CPU requirements decreased with

increased parallelism. Figure 1 displays the CPU time versus degree of parallelism for pattern recognition of five-chamber tracks on 20 000 events. Parallelism values were chosen to allow the full number of events to be processed on all iterations.

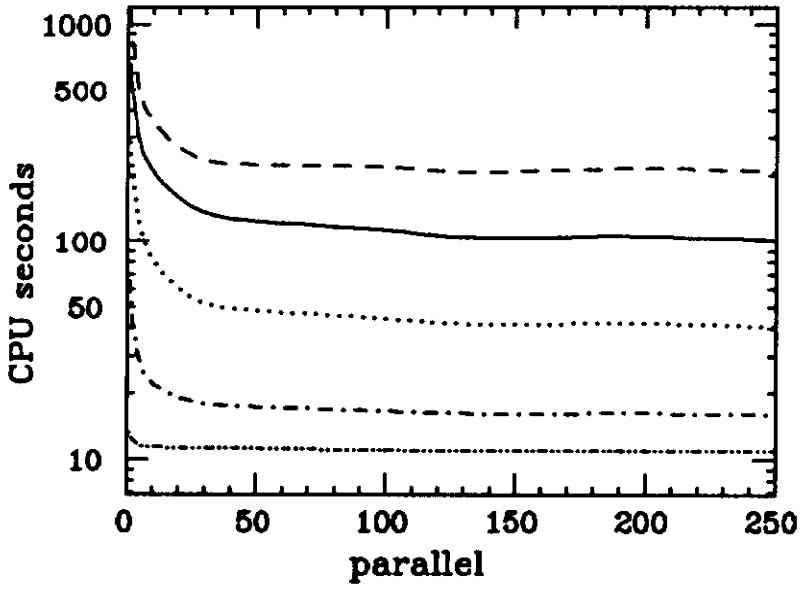


Figure 1 CPU times in seconds versus degree of parallelism for multi-scalar algorithm. Pattern recognition input is 20 000 events containing five-chamber input tracks.

PATTERN)	zero input tracks	DOTS)	four input tracks
DOTDASH)	two input tracks	SOLID)	six input tracks
		DASHES)	eight input tracks

Several variations of vector algorithms were developed and tested for this toy detector. There are 526 possible five-chamber tracks in this toy detector. Vector algorithm #0 listed in Table 1 is an E-711 type algorithm. This algorithm uses pipelinable loops over the planes in the track dictionary. The dictionary was stored in 526 64-bit words by packing each wire number of the track into a 12-bit field. The algorithm then used the extract† APFTN64 intrinsic function to extract the wire numbers for comparison with the event data. The program loops over each plane in the track dictionary, comparing the track wire numbers to the event data and incrementing hit counters when hit wires are found. After the loops through all planes have been completed, hit counters with a value of five correspond to the five-chamber tracks.

† The first version of this program used an APMATH64 library subroutine, **extru**, to extract the bit fields. The **extru** version of the program required approximately 76% more CPU time than the **extract** version presented as vector algorithm #0.

A slight modification of vector algorithm #0 yielded vector algorithm #1. Algorithm #1 stores each track of the track dictionary in five 64-bit words rather than packing all five wire numbers into one 64-bit word. This yielded some advantage in program speed, as the APFTN64 intrinsic function `extract` was not necessary on each dictionary loop iteration. However, the dictionary is five times larger than in vector algorithm #0. All other aspects of vector algorithms #0 and #1 are identical.

Vector algorithm #2 is a Mark III type algorithm. This algorithm contains the 526-track dictionary (used only for reference upon track output via formatted I/O) and a dictionary of track lists sorted by wire for each detector plane. The track lists are stored in a three-dimensional array indexed by track index, wire number, and detector plane. The maximum track index is 101, as this is the maximum number of tracks through any wire (computationally observed). An additional array (indexed on wire number and detector plane) provides the number of tracks through each detector wire. This algorithm loops over the detector planes, checking each wire for a hit. When a hit is found, the track list for that wire, which contains all tracks which pass through that wire, is scanned and the hit counters for the corresponding tracks are incremented. After the loops through the detector have been completed, hit counters with a value of five correspond to the five-chamber tracks. This algorithm requires a substantial amount of memory—the track list dictionary is approximately twice as large as the unpacked track dictionary (which is also part of this algorithm‡).

All programs read the same input file containing the events to be analyzed. Events were generated by a Monte Carlo generator which output the events as an unformatted array of logical values with a value for each detector wire (`.TRUE.`$\Longrightarrow$hit wire). Files containing unformatted dumps of the track dictionaries were prepared for efficient input by the vector algorithms. In this way, the timing results for the test programs correspond as closely as possible to the pattern recognition CPU requirements. Figure 2 presents results for CPU times versus number of input tracks in the events. *NOTE: The CPU*

‡ A version of this algorithm with a packed dictionary was produced and tested. However, the raw track dictionary is only used during the formatted output phase, so the pattern recognition differences are insignificant.

times are plotted against the theoretical number of tracks created in the Monte Carlo detector—NOT the number of found tracks, which can become quite large for high chamber multiplicities. Results for all of the above algorithms are shown. The events contain only five-chamber tracks; no noise has been simulated. All tests have been performed with no formatted I/O and with APFTN64 optimization level 4. The vector algorithms contain *no* tunable parameters. The tunable parameters were adjusted for the scalar and multi-scalar algorithms so as to yield results consistent with the output of the vector algorithms.

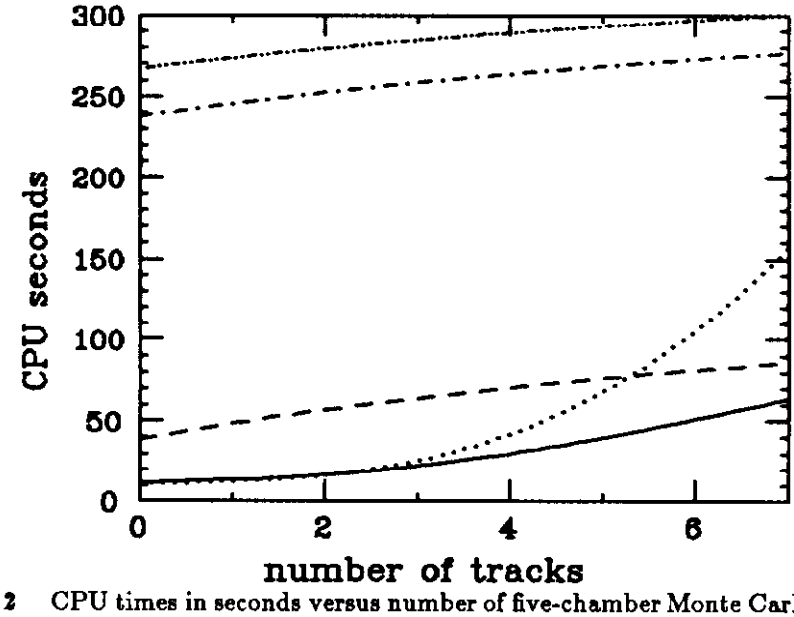


Figure 2 CPU times in seconds versus number of five-chamber Monte Carlo tracks present in input events. CPU times represent the analysis of 20 000 events.

SOLID) Scalar Algorithm	PATTERN) Vector Algorithm #0
DOTS) Multi-Scalar Algorithm	DOTDASH) Vector Algorithm #1
	DASHES) Vector Algorithm #2

As expected, the CPU requirements of the scalar algorithm increase considerably with increasing chamber multiplicity. The multi-scalar algorithm, because it always behaves as a "worst case" scalar algorithm, has CPU requirements that rise much more rapidly than the scalar algorithm. The vector algorithms #0 and #1 are expected to behave nearly-independently of chamber multiplicity—Figure 2 appears to confirm this. Vector algorithm #2 should behave approximately linearly with the total number of detector hits. The decrease in slope of its curve in Figure 2 as the chamber multiplicity increases is evidence of this (overlapping tracks at high multiplicity$\Longrightarrow$fewer additional hits per track). Clearly, none of the vector algorithms outperforms the scalar algorithm for reasonable chamber multiplicities.

Further refinements of the vector codes yielded some gain in efficiency. No profiling program was available on the FPS-164; such a program might have been very useful. An important question was whether the vector algorithms were using the FPS-164 as effectively as possible. The APFTN64 compiler can output the generated machine code into its listing file. Machine code listings were studied for the scalar codes and the vector codes. These studies were non-quantitative. Nonetheless, the vector codes appeared to have a higher frequency of NOP instructions and a higher percentage of instructions appeared to address a small number of the CPU's functional units. Loop unrolling* is suggested[4] as a means of increasing program speed in APFTN64 when compiling at optimization levels 1 or 2. At optimization levels above 2, loop unrolling is expected to give little improvement or even slow execution. Several loops with high iteration counts but with short loop bodies were unrolled to varying degrees. As forewarned, unrolling of some loops caused slower execution. However, unrolling of several major loops resulted in speed increases of 19.5% to 22.7% for vector algorithm #1 and 6.7% to 15.4% for vector algorithm #2. (Speed increases were approximately 54 seconds for #1 and 6 seconds for #2 in the tests presented in Figure 2.) On loops which responded to unrolling, tests were performed to determine the optimum level for unrolling. These loops showed decreasing marginal improvement as unrolling increased. All of the tested loops showed saturation of the speed increases at unrolling levels of three to six iterations.

The above algorithms were also tested with random noise present in the Monte Carlo events. Noise was created by selecting wires at random throughout the detector and placing hits on those wires. No inefficiencies were included in the simulation. Noise was measured in terms of the number of wires randomly hit (regardless of whether a track hit the wire) throughout the detector. Studies were performed with varying amounts of noise on samples which included one, three or five Monte Carlo tracks as input. These results are shown

---

* Loop unrolling is demonstrated by the following loop transformation. Consider the loop below.

```
      DO 100 I = 1 , 600
100   A( I ) = B( I ) * C( I )
```

Unrolling this loop to calculate three elements of A during one iteration yields the following loop.

```
      DO 100 I = 1 , 600 , 3
        A( I ) = B( I ) * C( I )
        A( I + 1 ) = B( I + 1 ) * C( I + 1 )
100     A( I + 2 ) = B( I + 2 ) * C( I + 2 )
```

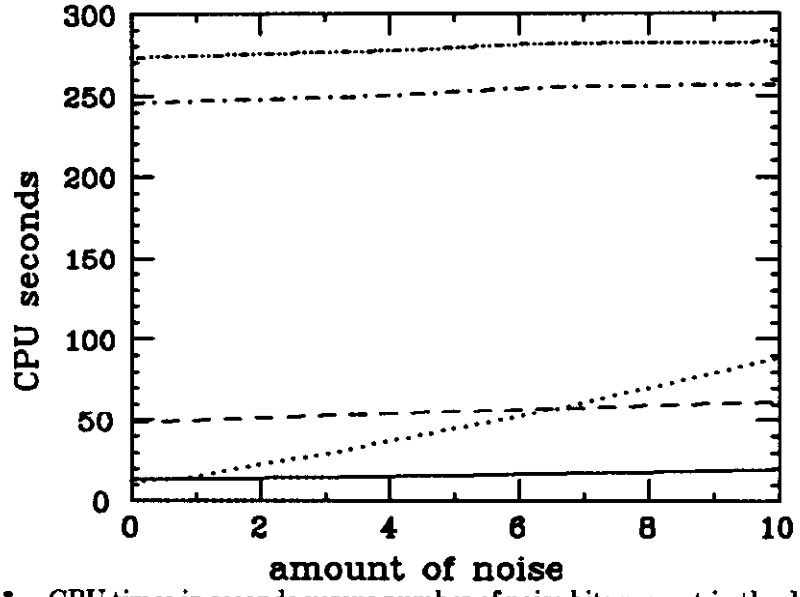


Figure 3 CPU times in seconds versus number of noise hits present in the detector. One five-chamber Monte Carlo track is the underlying event. CPU times represent the analysis of 20 000 events.

SOLID) Scalar Algorithm
DOTS) Multi-Scalar Algorithm
PATTERN) Vector Algorithm #0
DOTDASH) Vector Algorithm #1
DASHES) Vector Algorithm #2

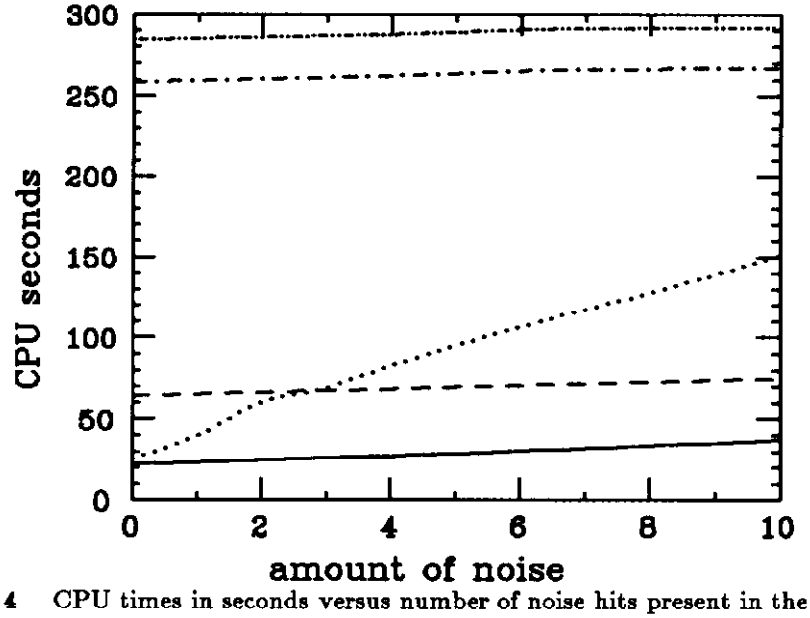


Figure 4 CPU times in seconds versus number of noise hits present in the detector. Three five-chamber Monte Carlo tracks are the underlying event. CPU times represent the analysis of 20 000 events.

SOLID) Scalar Algorithm
DOTS) Multi-Scalar Algorithm
PATTERN) Vector Algorithm #0
DOTDASH) Vector Algorithm #1
DASHES) Vector Algorithm #2

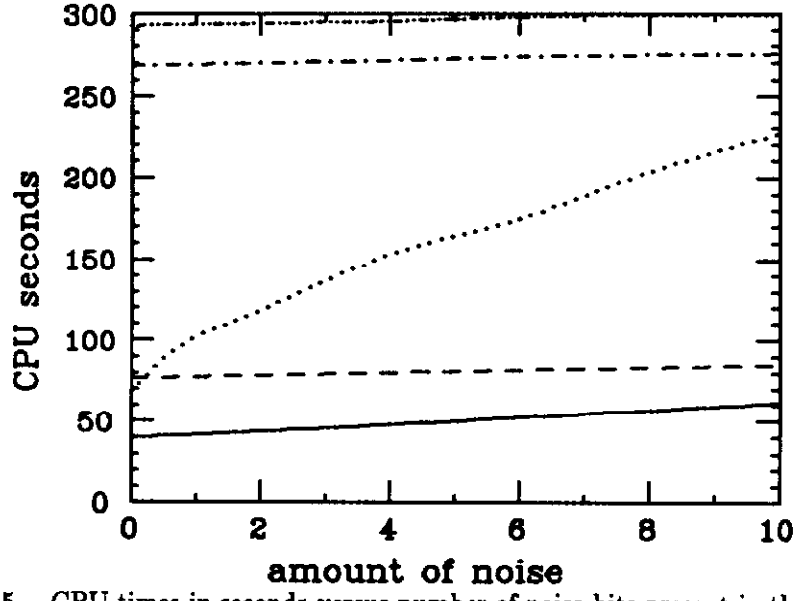


Figure 5 CPU times in seconds versus number of noise hits present in the detector. Five five-chamber Monte Carlo tracks are the underlying event. CPU times represent the analysis of 20 000 events.

SOLID) Scalar Algorithm
DOTS) Multi-Scalar Algorithm
PATTERN) Vector Algorithm #0
DOTDASH) Vector Algorithm #1
DASHES) Vector Algorithm #2

in Figures 3–5. The vector algorithms are relatively insensitive to noise, as expected. The scalar algorithm becomes slightly less efficient with noisy events, but the multi-scalar algorithm is strongly affected by noise in the data. For the scalar and vector algorithms, the effect of five noise hits is comparable to having an additional track in each event (one track$\Longrightarrow$five hits).

B. Single-View Detector: 4 planes × 85 wires

A single-view detector similar in size to a view in the E-711 detector was chosen as the second model for study. The E-711 detector has 256 wires in its X view. Since the E-711 Cyber 205 code uses a virtual detector that maps three real wires to one virtual wire, 85 wires was chosen as the detector size. The single view was set up in a projective geometry, using the data encoded in the E-711 Cyber 205[5] program to match the detector as closely as possible. Events were generated with randomly-placed four-chamber tracks. Random noise was added for some of the tests. A track dictionary for four-chamber tracks was created; 25 052 four-chamber tracks are possible. A dictionary of track lists sorted by wire for each plane was generated for the Mark III vector algorithms—this dictionary had a maximum track index of 432.

The scalar algorithm, vector algorithm #1 and several variations of vector algorithm #2 were tested with this detector. Vector algorithm #1 was found to take six times as long as vector algorithm #2—and was dropped after preliminary testing. Figure 6 displays results for this single-view detector when no noise is present. *NOTE: The CPU times are plotted against the theoretical number of tracks created in the Monte Carlo detector—NOT the number of found tracks, which can become quite large for high chamber multiplicities.* Vector algorithm #3 is a loop-unrolled version of vector algorithm #2. Vector algorithm #4 contains an additional Mark III algorithm element. The track counters (or dictionary) for this algorithm are subdivided into groups of 128 tracks. A bit-mapped array containing an entry for each track subgroup is used to record hits. As the algorithm loops through the detector planes and processes the track lists for hit wires, a bit corresponding to the respective plane is set in each of the bit-mapped elements corresponding to listed tracks. Hit counters are still kept for individual tracks. Rather than search for hit counters with value four to find the candidate tracks, the bit-mapped array is searched for elements with full maps. These elements correspond to track subgroups that may contain a candidate track. These track subgroups are then searched for candidate tracks. This procedure is effective because the density of four-chamber tracks in the dictionary is quite low.

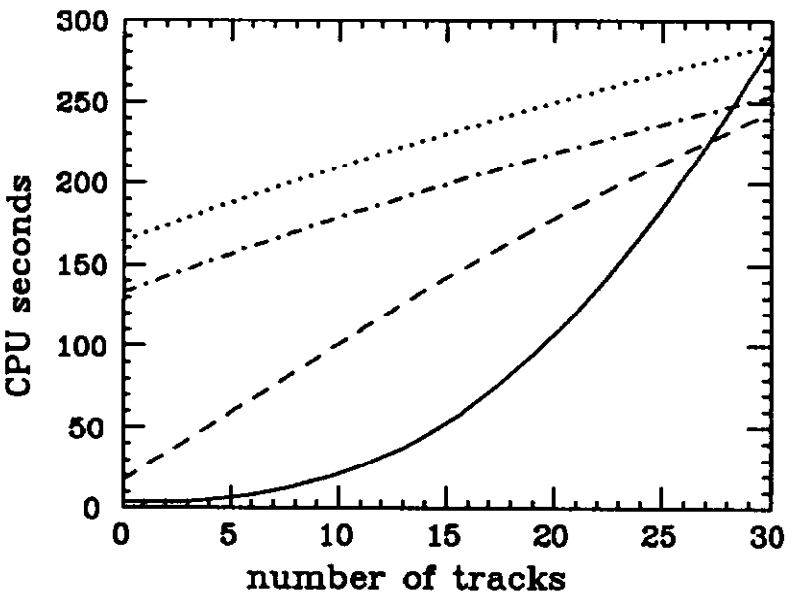


Figure 6 CPU times in seconds versus number of four-chamber Monte Carlo tracks present in input events. CPU times represent the analysis of 2500 events.
SOLID) Scalar Algorithm DOTDASH) Vector Algorithm #3
DOTS) Vector Algorithm #2 DASHES) Vector Algorithm #4

Figure 6 shows that at low multiplicities, vector algorithm #4 has a substantial advantage over the other vector algorithms. The effect of loop unrolling is clear from the difference between vector algorithms #2 and #3. The different behavior at high chamber multiplicity between the scalar and vector algorithms is also quite apparent in Figure 6. However, the relevant number of tracks for E-711 is approximately 15 three- and four-hit tracks per view found by the pattern recognition. This would correspond to ten or less four-chamber Monte Carlo input tracks. Timing results for four-chamber tracks with ten or less Monte Carlo input tracks clearly favor the scalar algorithm. However, the E-711 Cyber 205 code searches for three- and four-chamber tracks, so more work needs to be done. Discussion of the effects of inefficiencies and searches for incomplete tracks will be presented in a following section.

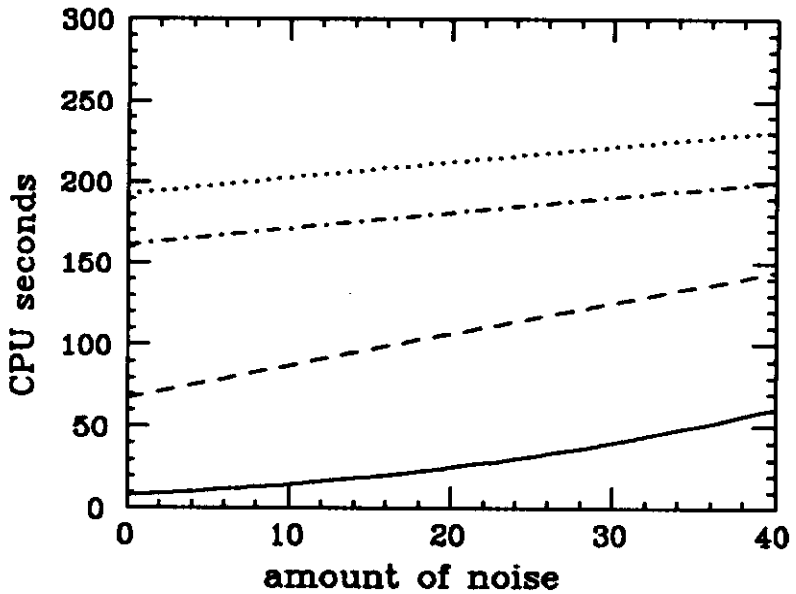


Figure 7 CPU times in seconds versus number of noise hits present in the detector. Six four-chamber Monte Carlo tracks are the underlying event. CPU times represent the analysis of 2500 events.

SOLID) Scalar Algorithm DOTDASH) Vector Algorithm #3
DOTS) Vector Algorithm #2 DASHES) Vector Algorithm #4

Figures 7 and 8 present timing results for the above pattern recognition algorithms in the presence of random noise in the detector. As in the previous model, the number of noise hits is variable over the base event of six or twelve Monte Carlo input tracks. No inefficiencies were included in the simulation. As expected from the smaller model, the vector algorithms #2 and #3 are fairly insensitive to noise. Vector algorithm #3 is slightly more sensitive to noise because its bit-mapped array elements for the track subgroups will

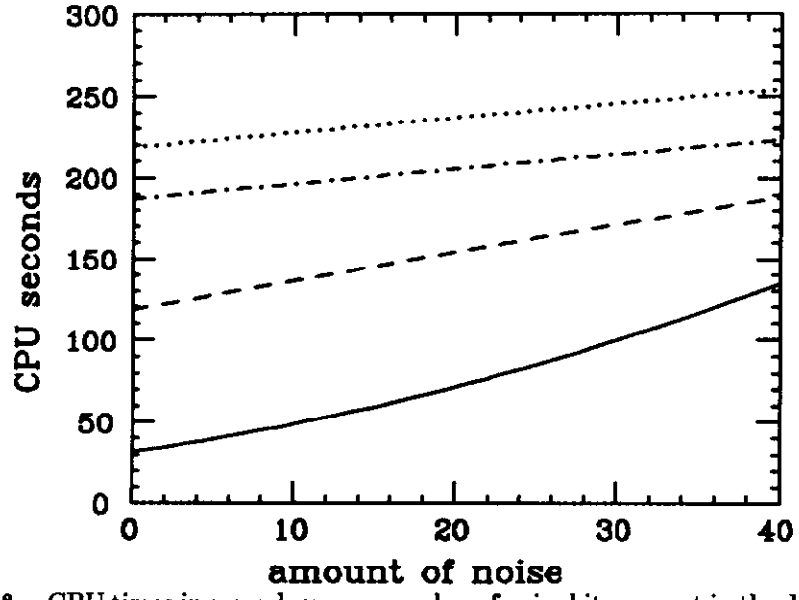


Figure 8 CPU times in seconds versus number of noise hits present in the detector. Twelve four-chamber Monte Carlo tracks are the underlying event. CPU times represent the analysis of 2500 events.

SOLID) Scalar Algorithm
DOTDASH) Vector Algorithm #3
DOTS) Vector Algorithm #2
DASHES) Vector Algorithm #4

be filled by the noise and cause additional track subgroups to be searched. The scalar algorithm continues to be the most sensitive to noise. For all of these algorithms, the effect of four additional noise hits on the pattern recognition timing is comparable to having an additional track in each event (one track$\Longrightarrow$four hits).

C. 3-Dimensional Four-View Detector

The initial intent of this simulation was to use an E-711 look-alike detector. However, it was quickly discovered that the FPS-164 had only 512K 64-bit words—and vector algorithm #4 of the previous section used over one-half of the available memory for only a single detector view. Thus, a detector with four identical views, but oriented at 0°, 90°, +10° and −10°, would substitute. This view was taken to be the single view used in the previous section. The goal of this simulation was to perform pattern recognition in the four views including three- and four-hit tracks and then to perform three-dimensional matching to create the *real* particle tracks.

Vector algorithm #4 was adapted to this simulation. A three-dimensional event generator was created, and three- and four-chamber tracks could be found in each of the four detector planes. Only one pass through the data is used in the vector algorithm to find both the three- and four-chamber tracks. The information is stored in the hit counters—

and the search through these counters can easily find both track types. A problem was soon realized, however. For each four-chamber track found, several *adjacent* three-chamber tracks were found. These adjacent tracks had three hits in common with the four-chamber track, so no unique track was specified. A solution was to adapt another component of the Mark III algorithm[2]—the auxiliary track list. The auxiliary track list contains, for each track, a list of the tracks with which it shares all but one hit. Thus, after four-chamber tracks are found, three-chamber tracks on the auxiliary lists of these four-chamber tracks can be eliminated from the data. Memory again proved to be the limiting factor, as the detector had to be cut to 65 wires in order to fit all the dictionaries into the FPS-164 memory.

Due to memory constraints and lack, therefore, of ability to produce a competitive E-711 analysis program on the FPS-164, work on this simulation has been discontinued.

IV. Conclusions

Vectorized pattern recognition algorithms have been compared to traditional scalar algorithms in single-view detectors on the FPS-164. The vectorization has been limited to the track recognition phase of the data analysis. Results show that the traditional scalar algorithm outperforms the vectorized algorithms in chamber multiplicities of physical interest.

This simulation did not have raw data tapes available. The E-711 analysis code of Georgiopoulos[1] *et al.* uses vectorization in event data unpacking as well as in the single view pattern recognition. The simulations on the FPS-164 did not include this vectorized data unpacking. It is not known what portion of the E-711 speed increase is due to the vectorized data unpacking. All other E-711 code—track fitting and three-dimensional matching—is done with scalar (non-vectorized) code.

Another area for consideration is the effect of incomplete tracks on the pattern recognition. The single-view simulations searched for full-detector tracks only. Production analysis codes use incomplete tracks in event reconstruction. This effect has not been tested on the FPS-164, but rough estimates can be presented. Vector algorithms, in general, make only one pass through the data and create an array of hit counters. All incomplete tracks of interest can be found from this array. However, as mentioned above, many adjacent

tracks must be eliminated from this array by using auxiliary lists. The time required for this arbitration via auxiliary lists must be added to the vector algorithm. Incomplete tracks would have a more substantial impact on scalar algorithms, however. An initial pass through the scalar algorithm would be used to find the full-detector tracks. The hits for these tracks would often be deleted from the data in order to limit confusion. Then for a detector such as E-711, two additional passes on the reduced data would need to be made to recognize the tracks with one missing hit. Thus, the times for the scalar pattern recognition would increase by a factor of three or less, while the vector algorithm would have an increase due to the auxiliary list searching (50% or less???). These factors would be used to adjust the figures presented in previous sections. In any case, vectorization does not appear to yield significant gains on the FPS-164.

One might expect that a Floating Point Systems computer would give the best results for a floating-point intensive algorithm. The vector algorithms use very little floating point arithmetic—mostly integer and logical calculations are involved. Such algorithms may not perform well, in general, on the FPS-164. Indeed, the APMATH64 library is oriented toward matrix and other substantially floating point problems. Another indication of this is the observation that the scalar algorithm tended to generate denser machine code for the FPS-164. However, it is also difficult to know whether the algorithms (scalar or vector) employed have utilized the FPS-164 to greatest advantage.

Another advantage of the scalar algorithm is compatibility with VAX FORTRAN. The FPS-164 APFTN64 scalar program was able to run *unchanged* under VAX FORTRAN. APFTN64 is closer (fewer extensions) to ANSI-standard FORTRAN than VAX FORTRAN, but several common and useful extensions are available (user-defined symbols may contain up to 31 characters, including the dollar sign ($) and the underscore (_)). The FPS-164 was quite easy to use—its Single Job Executive (SJE) operating system and its Job Definition Language (JDL) were well documented and the commands were simple and understandable. Program development (editing, compiling, etc.) was performed on the front-end machine in a convenient and familiar environment. Direct access to tape drives for the FPS-164 is not mentioned in the FPS-164 Operating System Manual;[6] the only mention of tape drives is the ability to PRESERVE and RESTORE FPS-164 files using front-end computer disks or tape drives. Copying tapes through the front-end computer onto the

FPS-164 disk subsystem would be impractical for production analysis. Similarly, direct access of front-end computer tape drives or disks from the FPS-164 would probably be too slow. Use of the APEX64 programming method, where a front-end computer passes numerically-intensive computations to the FPS-164 also appears impractical, in view of the amount of data involved (data transfers also involve conversions to and from FPS-164 formats for integers and floating point numbers).

The FPS-164 and Cyber 205 used for the E-711 analysis have substantially different architectures. The Cyber 205 is bit-addressable in vector mode—and this was exploited in the E-711 code in the gathering of the candidate tracks from the dictionary. The FPS-164 deals only with 64-bit words—and algorithms which use smaller quantities can be quite inefficient. The differences in architecture make it difficult to compare results between the FPS-164 and Cyber 205.

Acknowledgements

I wish to thank H. Montgomery and P. Lebrun for many useful discussions, particularly regarding details of the scalar track-finding algorithm. I also wish to thank P. Lucas and the Fermilab Accelerator Division for their cooperation and use of their FPS-164 Scientific Computer.

References

1) C. H. Georgiopoulos, J. H. Goldman, D. Levinthal and M. F. Hodous, *Nuclear Instruments and Methods in Physics Research* **A249**, 451 (1986).
2) J. J. Becker *et al.*, *Nuclear Instruments and Methods in Physics Research* **A235**, 502 (1985).
3) V. Zacharov, CERN DD/82/1, (1982).
4) APFTN64 User's Guide, Floating Point Systems, Inc., Portland, OR, 1985.
5) C. Georgiopoulos, private communication, E-711 Cyber 205 data analysis code.
6) FPS-164 Operating System Manual, Volumes 1–3, Floating Point Systems, Inc., Portland, OR, 1985.
7) P. Lebrun, private communication, TFFTE tracking library code and documentation.