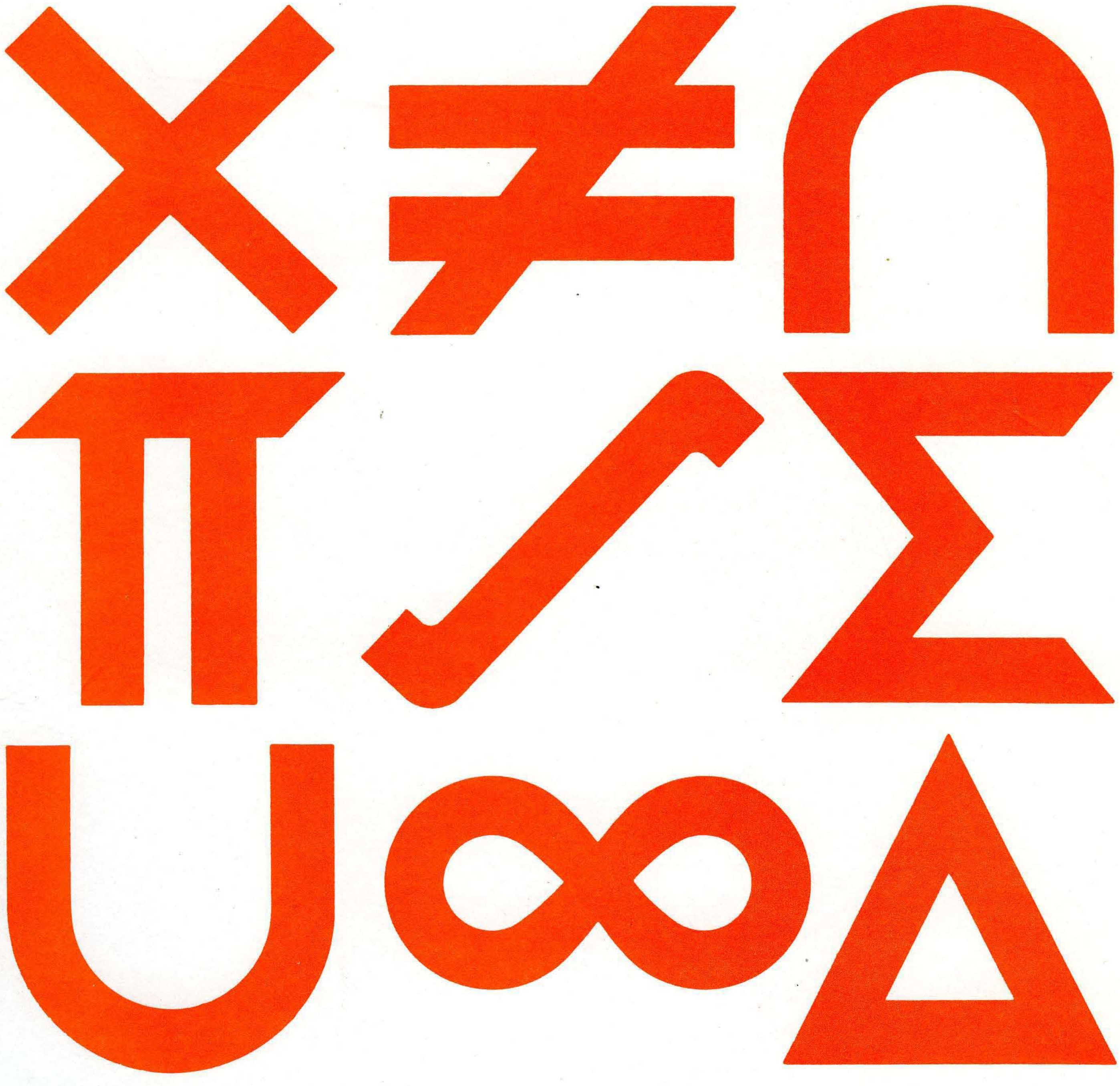
IBM PALO ALTO SCIENTIFIC CENTER
G320-3504, November 1987

Exploiting the IBM 3090 Vector Facility in Image Processing Applications

H. J. Myers
A. H. Karp



EXPLOITING THE IBM 3090 VECTOR FACILITY IN IMAGE PROCESSING APPLICATIONS

Document Number G320-3504

October 15, 1987

H. Joseph Myers
Alan Karp

IBM Scientific Center
P.O. Box 10500
Palo Alto, CA 94304

Exploiting the IBM 3090 Vector Facility in Image Processing Applications

Abstract

Image processing requires computationally intensive manipulation of very large amounts of byte-oriented data. It would be desirable to take advantage of a vector processor to reduce computation time when solving image processing problems. An obstacle to achieving this is the tendency of image data to consist of 1-byte data elements, while the Vector Facility offers at a minimum a 2-byte load/store capability. The purpose of this study was to implement several common image processing applications to take advantage of the Vector Facility offered on the IBM 3090 in order to determine the degree to which vectorization could be accomplished, and to gauge the performance benefits which could be derived.

Performance improvements resulting from the techniques used ranged from zero to a factor of 3.57 when the vector instructions were compared to simple scalar algorithms. However, over half of the gain was due to the better coding techniques alone. The vectorized algorithms demonstrated a vectorizability around 90%. It is concluded that the addition of 1-byte load and store instructions to the vector instruction set would not provide benefit beyond the methods described here.

Keywords
Image Processing
Vector Facility
IBM 3090
Algorithms
Optimization
Performance

Overview

Image processing of Landsat, medical and other byte-mapped data requires operations on very large Megabyte arrays. (A "standard" Landsat Scene requires 320 megabytes to store.) The unit of information in these arrays, called a "pixel" is normally stored as one byte. Typical image displays can present 1024x1024 or 1024x1280 pixel images, each of which may be represented by three bytes (3.6 megabytes total). Because of the size of these images, unit increases in performance can have a significant effect.

We have selected three different image processing functions for algorithmic construction.

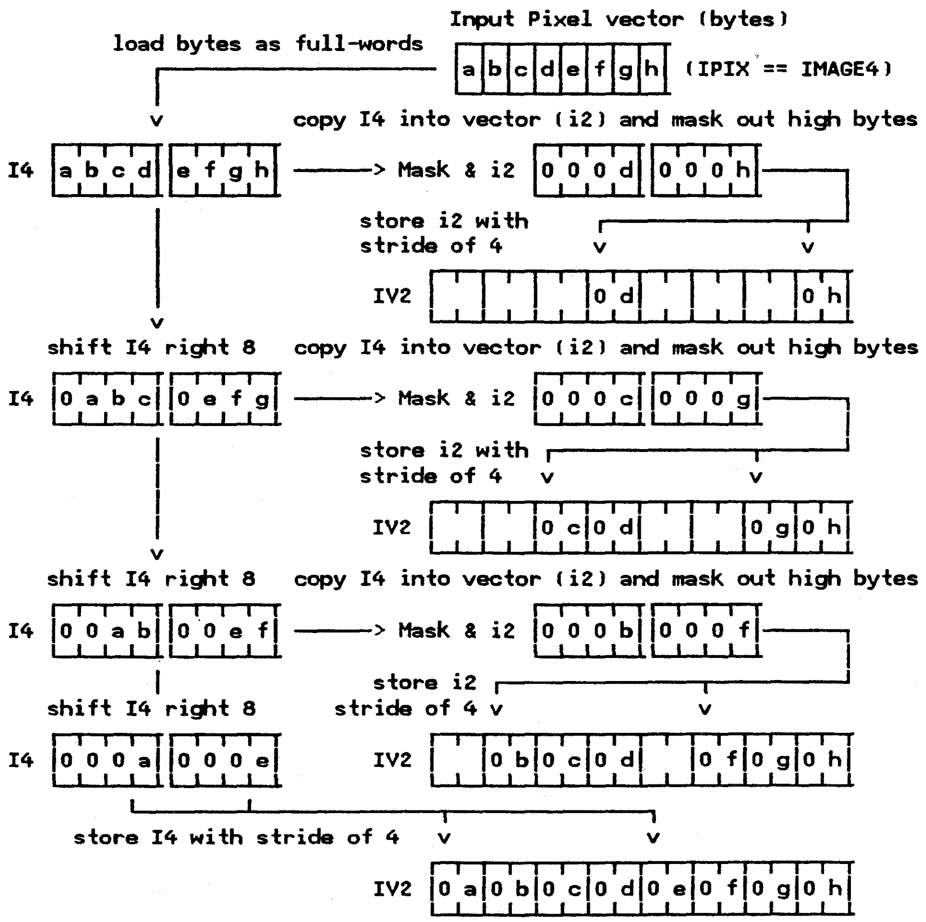
- Point operations
- Histogram collection
- Local Intensity Enhancement

Each of these applications presents a different set of vectorizing problems to be solved. However, they all shared a single main problem: handling 1-byte data. We will, therefore, first discuss a method for unpacking 1-byte data into 2-byte data as a general technique. The analogous function of packing 2-byte data is similar. With this pair of functions one could simply unpack an image, work on it with vector operations, then pack the result. Another approach is to process the data as it is being unpacked. This latter approach is the best, and will be implemented in the various algorithms, but the unpacking process is shown separately here for expository purposes. Naturally, for this technique to pay off, the computational cost of unpacking and the additional loop overhead has to be more than absorbed by the gain in vector processing the results. This technique uses the vector facility to do the unpacking. This reduces the cost of this technique to a minimum. Note that the method also can be further enhanced if parallel computation is available. The method assumes that images have a multiple of four pixels.

The unpacking method is illustrated by the following FORTRAN code. (The annotation to the left of the code is from the FORTRAN compiler vectorizer report.)

```
                    LOGICAL*1 IPIX(N)
                    INTEGER*2   IV2(N)
                    INTEGER*4   IMAGE4(N/4), I4, I, N
                    EQUIVALENCE (IMAGE4(1),IPIX(1))
                    INTEGER*4 I4
 VECT +------- DO 20 I = 0,N/4-1
      |             I4 = IMAGE4(I+1)           !Transfer to 4-byte word
      |             IV2(4*I+4) =IAND(I4,255) !Pick off last byte & store
      |             I4 = ISHFT(I4,-8)          !Shift next byte into place
      |             IV2(4*I+3) =IAND(I4,255) !etc.
      |             I4 = ISHFT(I4,-8)
      |             IV2(4*I+2) =IAND(I4,255)
      |             I4 = ISHFT(I4,-8)
      |_______      IV2(4*I+1) =IAND(I4,255)
               20 CONTINUE
```

The diagram below illustrates vectorized unpacking into 2-byte integers.



In the sections below we will compare timings and give performance figures. All runs were made on an IBM 3090 model 200 with a vector facility having 128-element vectors. The term "percent vectorizable" is discussed in detail in reference [1] . The heading "Disabled" below refers to compiling the vector version of the algorithm with FORTRAN/VS, release 2.2 without selecting the vectorizing option.

Point Operations

Point operations are carried out on all pixels of an image uniformly, and without regard to pixel location, and without regard to the computations on other pixels in the image. For example, doubling all pixel values in an image is a point operation. Point operations are clearly a prime candidate for vectorization. In this operation the shape of an image need not be considered. It can be treated in vector (1-dimensional) form.

A common technique for applying point operations is to compute a table of 256 results from applying the operation to all possible values of a pixel. A simple replacement of each pixel with its corresponding table entry can then ensue as a rapid process. The operation selected was "contrast stretch", calculated as: $p' = a \times p + b$

Below is the entire subroutine for performing a contrast stretch as described above. Note the technique required for fetching a byte into a word which is required because the FORTRAN/VS compiler does not allow a LOGICAL*1 quantity to act as a subscript. (The functions ICHAR and CHAR do not generate in-line conversion code.)

```
      SUBROUTINE POINTO(IPIX,OPIX,N,A,B)
        INTEGER*4 N
        REAL*4 A, B, C
        LOGICAL*1 IPIX(N), OPIX(N)
C
      LOGICAL*1 C41(4), C42(4), C1, C2
      INTEGER*4 I41, I42, LUT(0:255)
      EQUIVALENCE (C1, C41(4)), (I41, C41(1))
      EQUIVALENCE (C2, C42(4)), (I42, C42(1))
      I41 = 0
      I42 = 0

C     Load up LUT for 256 input values
      DO 10 I = 0,255
        C = A * I + B
        IF (C .GT. 255) THEN C = 255
        IF (C .LT. 0) THEN C = 0
        LUT(I) = INT(C+0.5)
10    CONTINUE

C     Apply LUT to IPIX to produce OPIX
      DO 20 I = 1,N
        C1 =IPIX(I)     !fetch pixel into low order part of word (I41)
        I42 = LUT(I41)
        OPIX(I) = C2    !fetch pixel from low order part of word (I42)
20    CONTINUE
      END
```

The code on the next page shows the vectorized subroutine plus the compiler report showing that vectorization was possible for both the program loops. Following that are the timings. Note that the modified algorithm performed 2.7 times faster than the original without the benefit of the vector facility.

```
        SUBROUTINE POINT1(IN4,OUT4,N,A,B)
          INTEGER*4 N
          REAL*4 A, B, C
          INTEGER*4 IN4(N/4), OUT4(N/4)
        INTEGER*4 I41, I42, LUT(0:255)
              C      Load up LUT for 256 input values
 VECT +-------       DO 10 I = 0,255
      |                C = A * I + B
      |                IF (C .GT. 255) THEN C = 255
      |                IF (C .LT. 0) THEN C = 0
      |______          LUT(I) = ISHFT(INT(C+0.5),24)
                 10  CONTINUE

              C      Apply LUT to IPIX to produce OUT4
                     I4=0

              C      Operate on pixels 4 at a time
 VECT +-------       DO 20 I = 0,N/4
      |                I41 = IN4(I)
      |                I42 = LUT(IAND(I41,255))
      |                I41 = ISHFT(I41,-8)
      |                I42 = IOR(ISHFT(I42,-8),LUT(IAND(I41,255)))
      |                I41 = ISHFT(I41,-8)
      |                I42 = IOR(ISHFT(I42,-8),LUT(IAND(I41,255)))
      |                I41 = ISHFT(I41,-8)
      |                I42 = IOR(ISHFT(I42,-8),LUT(IAND(I41,255)))
      |______          OUT4(I) = I42
                 20  CONTINUE
```

The times (in milliseconds) to contrast stretch a 1 Mbyte image are shown below:

```
                Scalar     Vectorized   Disabled
 Virtual CPU     0.596       0.167        0.220 (Ts)
 Vector CPU      0.000       0.150        0.000
                            ------
 Non-vectorizable time       0.017 (s)

 Percent vectorizable: 100 x (1 - s/Ts) =  92.3%
 Percent of time vector facility in use = 100 x .150/.167 = 89.8%
 Performance improvement (.220/.167) = 1.32 Disabled/Vectorized
 Performance improvement (.596/.167) = 3.57 Scalar/Vectorized
 Performance improvement (.596/.220) = 2.71 Scalar/Disabled
```

The Histogram Algorithm

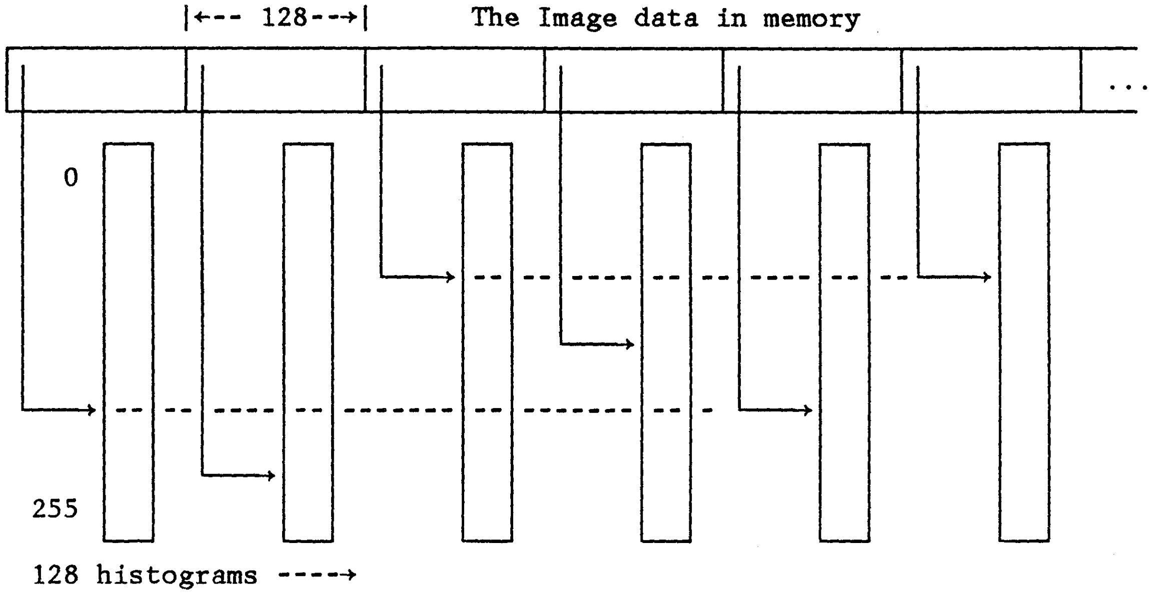
In addition to the problem of accessing 1-byte data, histogramming introduces the problem of order dependency. The normal histogram calculation is performed in FORTRAN as follows:

```
        LOGICAL*1 IPIX(N)
        INTEGER*4 I4, I, N, HGM(0:255)
        DO 10 I = 1,N
          I4 =IPIX(I)            ! Copy 1 byte of image data to I4
          HGM(I4) = HGM(I4) + 1 ! Generate the histogram counts
 10     CONTINUE
```

This program cannot be vectorized. We focus on the case when several pixels have the same value. If two pixels have the same value, then two vector elements will refer to the same word in storage. When this happens, copies of the same word will be loaded into separate vector elements, incremented, and returned to memory. Because there is no control to assure this process will be properly synchronized, there is no guarantee that the histogram will be computed properly. In fact, there is every reason to believe that it won't. Therefore, the code must be run in scalar mode.

To compute the histogram in a vectorized manner, we divided the image into some number of sectors. We used, for example, a number matching the machine vector size. This allowed each sector to have its own histogram, and set up the computation so that the vectors were loaded with elements from different sectors at any time. This was done by setting the vector stride to span each sector so that only one element from each sector was processed at a time. This ensured that only one element in any histogram was affected on any vector add cycle. At the end of the main accumulation, the histograms were accumulated into a single histogram, again in vector mode.

The method for accumulating counts into separate histograms is diagrammed below. Note that even though multiple pixels may have the same value, there is no contention because different histograms are being incremented.



Below is the code used to accumulate counts into 128 histograms. (Our vector facility has 128 elements.) It assumes that the image has been unpacked into the vector IV2. Note that the J-loop is vectorized, and that all accumulations on each iteration of the I-loop are into separate histogram accumulators.

```
VECT +------- DO 30 J = 1, 128
RECR |+------    DO 30 I = 0,N-128,128
     ||______       HGMS(IV2(J+I),J) = HGMS(IV2(J+I),J) + 1
     |_______
           30 CONTINUE
```

This is the code needed to combine the results into a single histogram.

```
VECT +------- DO 50 I = 0, 255
     |           I4 = 0                    !I4 was introduced to allow
RECR |+------    DO 40 J = 1, 128          !the complier to vectorize the
     ||______       I4 = I4 + HGMS(I,J)    !outer loop. Had HGM(I) been used
     |     40    CONTINUE                  !the complier would have inferred
     |_______    HGM(I) = I4               !recursion in both loops.
           50 CONTINUE
```

It is possible to combine the unpacking process with the DO 30 I loop in order to avoid allocating space to the unpacked vector, and eliminate the corresponding store instructions. The following code illustrates the method. Note that the inner loop which unpacks four pixels is "unrolled" avoiding additional overhead of loop control, and allowing for customization of the first and fourth extractions.

```
              INTEGER*4 IVX               !additional declaration
VECT +------- DO 70 J = 1, 128
RECR |+------   DO 60 I = 0,N/4-1,128
     ||            I4 = IMAGE4(I+J)
     ||            IVX =IAND(I4,255)
     ||            HGMS(IVX,J) = HGMS(IVX,J) + 1
     ||            IVX =IAND(ISHFT(I4,-8),255)
     ||            HGMS(IVX,J) = HGMS(IVX,J) + 1
     ||            IVX =IAND(ISHFT(I4,-16),255)
     ||            HGMS(IVX,J) = HGMS(IVX,J) + 1
     ||            IVX =IAND(ISHFT(I4,-24),255)
     ||______      HGMS(IVX,J) = HGMS(IVX,J) + 1
     |     60   CONTINUE
     |_______
         70   CONTINUE
```

As noted earlier, there is a trade-off between the time to unpack the image data and the speed-up resulting from vectorization. The assembler results below show the times to be about equal. Below are the times in seconds to process a 1 Mbyte image.

```
Algorithm         Scalar   Vectorized Scalar   Vectorized  Disabled
Times            Assembler Assembler  FORTRAN   FORTRAN    FORTRAN
Virtual CPU       0.208      0.200     0.593     0.285      0.375 (Ts)
Vector CPU        0.000      0.188     0.000     0.265      0.000
                             ------              ------
Non-vectorizable time        0.012 (s)           0.020 (s)

Percent vectorizable: 100 x (1 - s/Ts) =  94.2% (asm)
Percent vectorizable: 100 x (1 - s/Ts) =  94.7% (ftn)
Percent of time vector facility in use = 100 x .188/.200 = 94% (asm)
Percent of time vector facility in use = 100 x .265/.285 = 93% (ftn)
Performance improvement (.208/.200) = 1.02 (asm)
Performance improvement (.375/.285) = 1.32 (ftn) Disabled/Vectorized
Performance improvement (.593/.285) = 2.08 (ftn) Scalar/Vectorized
Performance improvement (.593/.375) = 1.58 (ftn) Scalar/Disabled
```

When the FORTRAN versions of the program are compared, the performance improvement is about 58%. When optimized fully by hand (assembler code) both scalar and vector algorithms perform at approximately the same speed. We should not have been surprised by this result (but we were).

Vector processing is faster than scalar for three reasons. First, only one instruction is needed to process many operands. On modern machines, like the 3090, the instruction fetch and decode is overlapped with the execution of previous instructions so there is little gain here. Second, there is only one branch for each vector register full of operands instead of one branch per operand in the scalar case. We unrolled the scalar loop to handle 4 pixels at at time which reduces this advantage of the vector unit. Third, vectors run faster because the operations can be pipelined (i.e., a multiple cycle operation can be broken down into steps that are overlapped) so that asymptotically the machine produces one result per cycle. In the histogram algorithm virtually every scalar operation takes one cycle negating this advantage of the vector unit. In other words, the vector unit does not speed up this calculation because the scalar unit is so efficient at handling byte data.

The assembler code below shows the algorithm operating on pixels in both scalar and vector implementations. Examination of the assembler code shows that, contrary to the appearance of the FORTRAN code, the scalar code loop must include load and shift instructions in the loop, which consumes the same number of cycles as the ISHIF and IAND in the vector loop. Therefore, the ISHIFT and IAND do not actually make the loop longer.

The improvement seen in the FORTRAN implementation is due to the inability of the compiler to efficiently handle 1-byte data, particularly in applying it as a subscript. This problem is not manifest in the vectorized solution because we always are dealing with integers in that case. As can be seen, neither FORTRAN program can compete favorably with the assembly code.

```
* Second pixel (scalar)
         LA       8,0          Clear register to get pixel values
         IC       8,1(7,5)     Load pixel value into register
         SLL      8,2          Convert word address to bytes
         L        9,0(6,8)     Load HGM(IPIX(I))
         AR       9,0          Increment counter
         ST       9,0(6,8)     Store updated value

* Second pixel (vector)
         VSRL     2,2,8        Shift next pixel into position
         VNQ      4,1,2        Pick off next pixel value
         VAR      3,4,1        Add pixel values to column offsets
         VLI      0,3,0(8)     Load HGM(IPIX(I))
         VAQ      0,9,0        Update all histograms
         VSTI     0,3,0(8)     Store updated value
```

Note, however, that because the vector algorithm is almost completely vectorizable, the algorithm speed will be nearly that of the the vector unit. The following table shows the expected timing for corresponding vector facility speed ups.

Vector/Scalar Speed	1	2	4	10
Algorithm Time	0.200	0.106	0.059	0.031

Clearly, this algorithm will prove useful on a machine with a vector facility substantially faster than its scalar unit. Note also, that providing a vector version of a 1-byte fetch instruction would not lead to a significant speed up of the process.

Local Intensity Enhancement (LIE)

It often happens that the lighting over an image is uneven. This can be corrected for by an algorithm which performs localized contrast stretching. This algorithm [2] is carried out as follows: Move a W x W window over the image, first vertically, moving one row at a time. At the end of each row, start at the top, one column over. For each set of pixels under the window at each position, compute the mean and standard deviation. Use these values to contrast stretch the value of the central pixel under the window at each position, forcing a desired mean and standard deviation.

This algorithm typifies a "neighborhood" process in which the value of a pixel is determined from the values of pixels that surround it. This algorithm must take into account the 2-dimensional aspect of the image.

Formulas:

Mean $M = \frac{1}{n}\sum_{i=1}^{n} p_i$

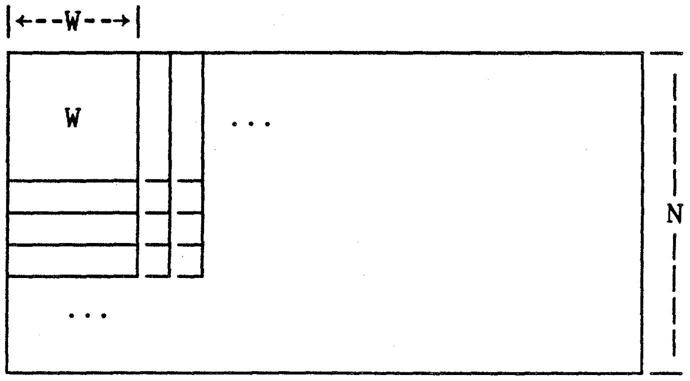
Standard Deviation: $S = \sqrt{\frac{1}{n}\sum_{i=1}^{n} p_i^2 - M^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} p_i^2 - (\frac{1}{n}\sum_{i=1}^{n} p_i)^2}$

Contrast stretch: $p' = a \times p + b$

where $a = S_{desired} / S_{window}$

$b = M_{desired} - a \times M_{window}$

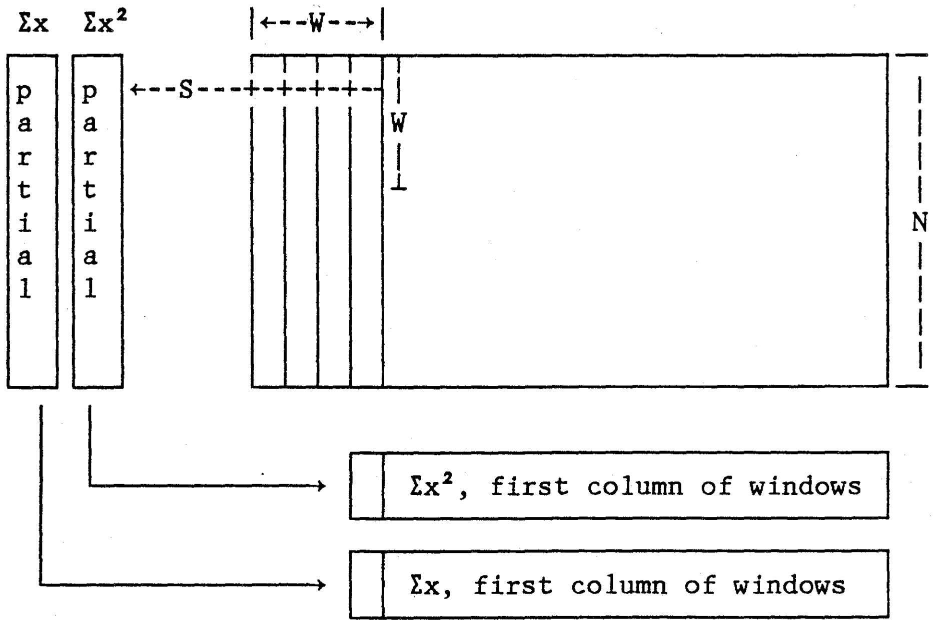
The figure below illustrates placement of some windows.



General approach:

1. Compute partial sum (and partial sum of squares) of 1st W columns. (Note: use of the word "sums" in this section will mean both sums collectively.)

The following diagram illustrates steps 2 and 3.



2. Compute sums for first column of windows.
   Note: Once the sums for the first window are computed, subsequent window sums can be computed by differences. That is, the sum for a window is its predecessor's sum, minus its predecessor's first row, plus the first row below the predecessor.

3. Compute a, b and by formulas above using Σx and Σx^2. Apply the formula to the center pixel of each window.

4. For each subsequent column, compute window partial sums. The new partial sums can be efficiently done by subtracting the left column of each window and adding the column to its right.

5. Repeat steps 2, 3 and 4 until all columns of windows are processed.

Vector Approach:

To provide good vector performance (in FORTRAN) we rewrite the actions to:

a) operate on pixels in groups of 4
b) start vectorized operations on word boundaries
c) operated on data in column-major order (adjacent pixels)

Only action 3 (computing central pixel values from the summations) requires (b) to be applied if we assume images have a multiple of 4 rows. (Images typically come in sizes that are powers of two.) We divide the rows into three sections shown below. Only operations on the middle section are vectorized.

```
        |←----------- multiple of 4 -----------→|
|_______|________________________________________|________|

head    |                 middle                      tail
        └ start at 4x
|←--------------------- L ----------------------→|
```

Thinking in terms of pixels (bytes):
a) the middle section starts at word boundary (multiple of 4 bytes)
b) length of the head (LH) is 0, 1, 2 or 3
c) length of the middle (LM) is the largest multiple of 4 less than L- LH.
d) length of the tail (LT) is L - LH - LM (will be 0, 1, 2 or 3).

Considerations and methods for vectorizing are as follows:

1. Allow equivalent views of an image, LOGICAL*1 and INTEGER*4. This has to be done by passing the image arguments twice, but declaring them differently. (Note that because of FORTRAN's column-major order, this means 4 rows of the byte-image are equivalent to one row of the word-image.)

```
SUBROUTINE LIE1(IPIX,IWRDS,OPIX,OWRDS,N,W,MEAN,STD)
  INTEGER*4 N, W
  REAL*4 MEAN, STD
  LOGICAL*1 IPIX(N,N), OPIX(N,N)
  INTEGER*4 IWRDS(N/4,N), OWRDS(N/4,N)

CALL       LIE1(IPIX,IPIXS,OPIX,OPIXS,N,W,MEAN,STD)
```

2. Collect partial sums and sums of squares into INTEGER*4 vectors This is done while unpacking. As illustrated earlier, the inner loop of unpacking four pixels is unrolled.

3. Computation of sums and sums of squares for the current column of windows vectorizes automatically, being summations of integer vectors.

4. The computation of mean and standard deviation for each window, and adjustment factors (a and b) therefor, can be calculated as a separate loop and stored into REAL*4 vectors, or it can be incorporated into the extraction loops of the center pixels. If the latter is done, the intermediate variables (standard deviation, mean, a and b) will be maintained in the vector unit and not be stored in memory.

5. Vectorized evaluation of the central pixel of each of the windows is carried out in three segments as indicated in the diagram above. The "head" and "tail" segments are performed in scalar mode, processing no more than 6 pixels per column. The central segment is processed four pixels at a time, with the code being completely vectorized by the compiler.

The resulting timings (in seconds) are:

```
              Scalar     Vectorized   Disabled
Virtual CPU    7.754       2.697        6.528 (Ts)
Vector CPU     0.000       2.097        0.000
                          ------
Non-vectorizable time      0.600  (s)

Percent vectorizable: 100 x (1 - s/Ts) = 90.8 %
Percent of time vector facility in use = 100 x 2.097/2.697 = 77.8%
Performance improvement (6.528/2.697) = 2.42 Disabled/Vectorized
Performance improvement (7.754/2.697) = 2.87 Scalar/Vectorized
Performance improvement (7.754/6.528) = 1.19 Scalar/Disabled
```

Conclusions

Performance Improvement Summary

Algorithm	Recoding Gain	Vector Facility	Total Gain	Percent Vectorizable	Percent In Use
POINT	2.71	1.32	3.57	92.3	90
HISTOGRAM	1.58	1.32	2.08	94.7	93
HIST (asm)	n/a	1.02	1.02	94.2	94
LIE	1.19	2.42	2.87	90.8	78

Above is a summary of the performance improvements for each of the FORTRAN algorithms. It separates out improvements due to recoding, from those due to the vector facility. From it we see that the improvements in the algorithms from their simple, straightforward form yielded gains that were more significant than those achieved by activating the vector facility. This is attributable more to the efficiency of the scalar operations of the 3090 than to any shortcoming in the vector facility.

The bulk of the savings came from loop unrolling when we processed four pixels as a group. So little computation was required in the loop, the loop indexing and branching took a significant part of the time.

The use of the cache memory was "perfect" in both vector and non-vector cases. That is, every byte loaded into the cache was used. In a machine without a cache, but with a broad data path to memory, fetching one byte at a time would leave much of the data path unused. With the cache, data is loaded from memory 128 bytes at a time. So the cache architecture kept single byte processing from being a great liability.

We note that the percent vectorizability for these algorithms is high. This means that the speed of the algorithms is almost completely dictated by the speed of the vector facility. Therefore, if the vector facility speed is doubled, the algorithms will run almost twice as fast.

Finally, we note that some of these algorithms would have benefitted from an extension to the FORTRAN compiler which would allow 1-byte data (CHARACTER*1 or LOGICAL*1) to be used as a number, particularly as a subscript.

References

[1] Program Migration Notebook for 3090 Vector Facility, IBM ZZ81-0170

[2] Ralph Bernstein, et. al, "Earth Imaging and Data Processing for Mapping Analysis", IEEE 1984 Proceedings of the 4th JCIT, pp 144-157

[3] Duda, R. O., and Hart, Peter E., "Pattern Classification and Scene Analysis", John Wiley & Sons, 1973

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHORS:
H. J. Myers and A. H. Karp

2. TITLE:
Exploiting the IBM 3090 Vector Facility in Image Processing Applications

3. ORIGINATING DEPARTMENT
Palo Alto Scientific Center

4. REPORT NUMBER
G320-3504

5a. NUMBER OF PAGES
11

5b. NUMBER OF REFERENCES
3

9. SUBJECT INDEX TERMS
Image Processing
Vector Facility
IBM 3090
Algorithms
Optimization
Performance

6a. DATE COMPLETED
October 1987

6b. DATE OF INITIAL PRINTING
November 1987

6c. DATE OF LAST PRINTING

7. ABSTRACT

Image processing requires computationally intensive manipulation of very large amounts of byte-oriented data. It would be desirable to take advantage of a vector processor to reduce computation time when solving image processing problems. An obstacle to achieving this is the tendency of image data to consist of 1-byte data elements, while the Vector Facility offers at a minimum a 2-byte load/store capability. The purpose of this study was to implement several common image processing applications to take advantage of the Vector Facility offered on the IBM 3090 in order to determine the degree to which vectorization could be accomplished, and to gauge the performance benefits which could be derived.

Performance improvements resulting from the techniques used ranged from zero to a factor of 3.57 when the vector instructions were compared to simple scalar algorithms. However, over half of the gain was due to the better coding techniques alone. The vectorized algorithms demonstrated a vectorizability around 90%. It is concluded that the addition of 1-byte load and store instructions to the vector instruction set would not provide benefit beyond the methods described here.

8. REMARKS