VECTORIZATION TECHNIQUES FOR ALGEBRAIC FRACTALS

by

V.C. Bhavsar, U.G. Gujar, N. Vangala

TR90-058, September 1990

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B. E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

Vectorization Techniques for Algebraic Fractals

Virendra C. Bhavsar
Uday G. Gujar
Nagarjuna Vangala

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B., Canada E3B 5A3

Abstract

Algebraic fractals generated from the self-squared transformation function $z \leftarrow z^2 + c$, where z and c are complex quantities, have been discussed extensively in the literature. The process of generating these fractal images, being iterative in nature, is computationally intensive. In this paper we propose and study three vectorization techniques for generating algebraic fractals from $z \leftarrow z^2 + c$, namely, use of long vectors, short vectors and short vectors with replenishment. The speedups obtained by vectorization of all these techniques on IBM 3090-180VF, which has a vector facility, are presented. It is observed that the technique of using short vectors with replenishment is the best.

1 Introduction

Fractals are objects with fractional dimensions. They have a high degree of visual complexity and possess a significant potential for modelling many natural complex objects such as mountains, clouds, plants and trees. The basic principle of generating fractals involves repetitive application of a specified transformation function to points within a region of space [3,4,5].

The algebraic fractals are generated using iterations of an algebraic transformation function. In this paper we consider the well known self-squared transformation function $z \leftarrow z^2 + c$, introduced by Mandelbrot [3]. Peitgen and Richter devote almost the entire book to illustrate and study a large number of images generated from this function [4].

The process of generating fractal images is computationally intensive since (a) this function is used iteratively, (b) complex quantities are involved, and (c) the computations have to be carried out for a large number of pixels. In order to reduce the execution time, we consider the vectorization of this

process. We propose three different techniques for vectorizing the generation of fractal images. These three techniques are implemented on IBM 3090-180VF and vectorization speedups are studied [6].

2 Generation of Algebraic Fractals

The principle of generating fractals employs the iterative formula :

$$z_{n+1} \leftarrow f(z_n),$$

where, z_0 = the initial value of z, and
z_i = the value of the complex quantity at the i^{th} iteration.

For the self-squared function,

$$f(z) = z^2 + c,$$

where z and c are both complex quantities. The Mandelbrot set is obtained by choosing z_0 as a constant and varying the value of c. The Julia set is obtained from the same self-squared function by choosing a constant value for c and varying the value of z_0 [4].

A fractal image consisting of the Mandelbrot set is constructed as a two-dimensional array of pixels wherein each pixel is represented as a pair of (x, y) co-ordinates. The values of the pixel co-ordinates x and y are based on the real and imaginary parts of c. The correspondence between a region of c-plane and the pixel positions in the fractal image is shown in Figure 1. The value of c for $(K, J)^{th}$ pixel, $C(K, J)$ is given as

$$C(K, J) = (CX + iCY)$$

where, $i = \sqrt{-1}$,
$CX = CXMIN + CSIZE * (K - 1)/(SIZE - 1)$, and
$CY = CYMIN + CSIZE * (J - 1)/(SIZE - 1)$.

The initial value of z, z_0, is kept constant for all the pixels.

For each pixel in the image, the value of z is computed iteratively. This iterative process terminates when the absolute value of z diverges beyond a

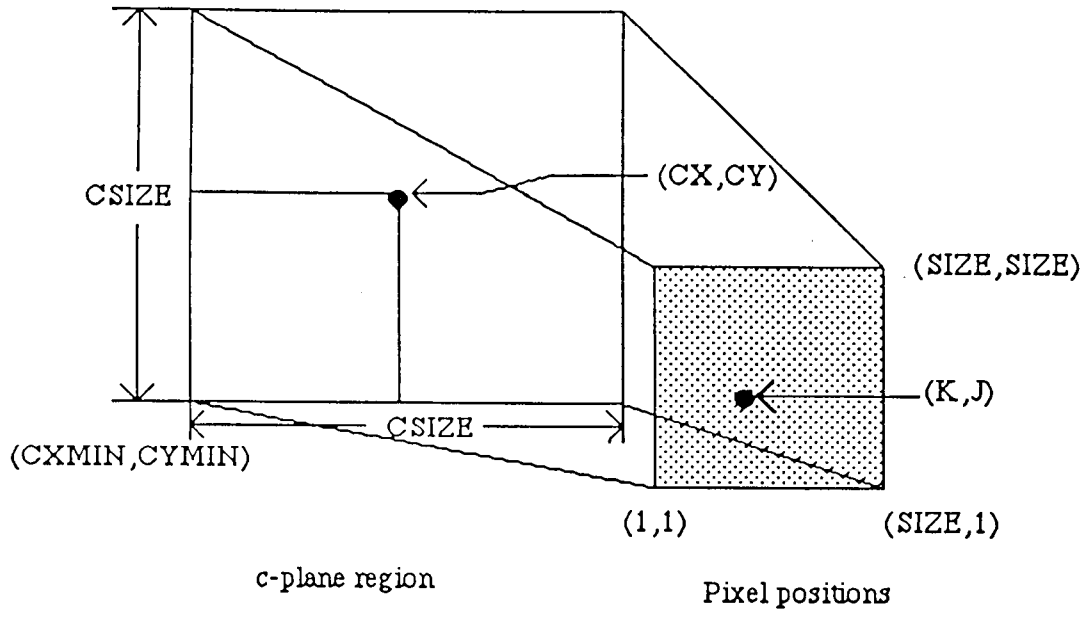

Figure 1: Correspondence between c-plane and pixel positions.

certain preset limit, LIM, or when the number of iterations exceeds the maximum allowable number of iterations, $NITER$. The number of iterations, $ITER$, carried out for the pixel, is used to determine the color of that pixel in the image; usually a color look-up table is used for this purpose.

Two obvious methods exist for rendering the image. The first method involves pixel-wise rendering wherein as soon as the computations for a pixel are complete it is rendered; this implies that no arrays are necessary for storage of the values of z, c and $ITER$. The second method consists of carrying out computations for all the pixels and then rendering the image; naturally, in this method two-dimensional arrays are required for storing the values of z, c and $ITER$. It should be noted that the array $ITER$ contains all the necessary information for rendering the image based on a given color table. Clearly, different images can be obtained from the same $ITER$ array by using different color tables without repeating the time consuming iterative computations; such computations are required for the first method. For these reasons, the second method is preferred and is used in this paper. A typical fractal image from the self-squared function is given in Figure 2.

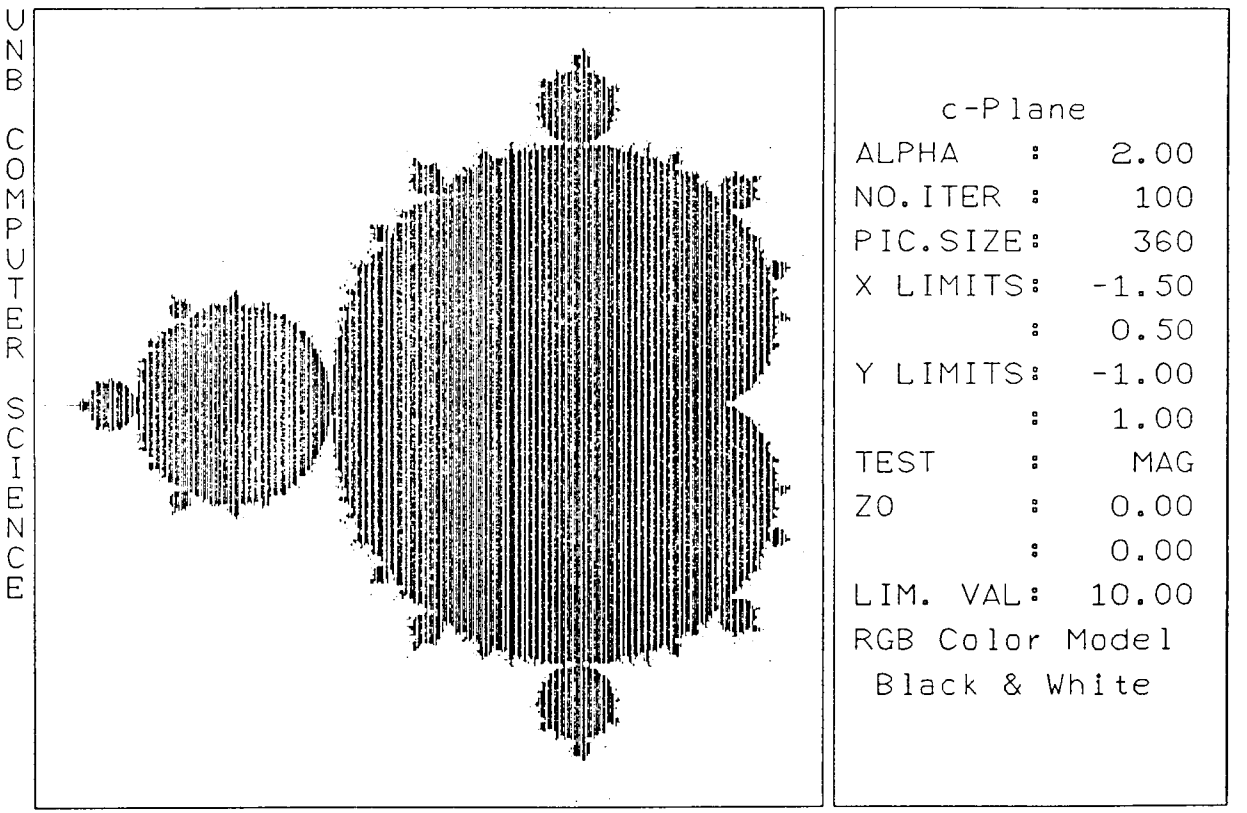

Figure 2: A Fractal Image from the Self-squared Function.

```
      COMPLEX C(65536),Z(65536)
      INTEGER SIZE,PITER(65536)
C     This loop is vectorizable
      DO I = 1,NPIX
         Z(I) = CMPLX(0.,0.)
         K = (I-1)/SIZE + 1
         CX = CXMIN + CSIZE*(K-1)/(SIZE-1)
         J = I - (I-1)/SIZE*SIZE
         CY = CYMIN + CSIZE*(J-1)/(SIZE-1)
         C(I) = CMPLX(CX,CY)
      CONTINUE
```

Figure 3: Initialization.

3 Vectorization Techniques

We propose the following three techniques for designing the code for generating a fractal image :

1. Use of long vectors.
2. Use of short vectors.
3. Use of short vectors with replenishment.

Vector processing on machines, such as Cray X-MP and IBM 3090-180VF, is efficient when the memory references are contiguous. In order to facilitate contiguous memory references and minimize the index calculations, the two dimensional arrays z, c and $ITER$ are converted into one dimensional arrays Z, C and $PITER$ respectively. The length of each of these arrays is equal to the total number of pixels, $NPIX$, in the image. For all the three techniques, the initialization of the z and c arrays is carried out by the program segment given in Figure 3. The initialization of the array $PITER$ is not required since it is set when the processing of a pixel is completed.

3.1 Technique 1: Use of Long Vectors

In this technique all the pixels are processed in every iteration. A code segment for obtaining the number of iterations required for each pixel ($PITER$) is given in Figure 4; for this figure $NPIX$ is equal to 65536. There are two DO

```
      COMPLEX C(65536),Z(65536)
      INTEGER PITER(65536)
      REAL LIM
C     Generate the Fractal image
      DO K = 1,NITER
C        This loop is vectorizable
         DO I = 1,NPIX
            IF(CABS(Z(I)).LT.LIM) THEN
               Z(I) = Z(I)*Z(I) + C(I)
               PITER(I) = K
            ENDIF
         CONTINUE
      CONTINUE
```

Figure 4: Use of Long Vectors.

loops in this code. The outer loop is for the number of iterations while the inner loop takes care of processing all the pixels in the fractal image. The iterations for a pixel are carried out until the absolute value of z exceeds the limit or the NITER iterations are finished. When vectorization is carried out the inner loop gets vectorized; the outer loop cannot be vectorized due to inherent dependencies in the computations for z values. As the iterations proceed, computations on more and more pixels get completed and the effective vector lengths for vector operations reduce. Thus, the efficiency of vector processing reduces as iterations proceed. Moreover, since the pixels for which computations are required are no longer contiguous, the use of masked vector operations further reduces the efficiency of vector processing.

3.2 Technique 2: Use of Short Vectors

In this technique we process only a fixed small number of elements at a time from the C, Z and $PITER$ arrays. The idea is that the memory references get localized to a small part of the memory so that the efficiency of vector processing may be improved.

Figure 5 gives a code segment to carry out the processing of pixels using this technique. In this code, sub-vectors of length VLW are processed at a time; for simplicity, we assume that $NPIX$ is divisible by VLW so that the number of sub-vectors, NV, is given by $NV = NPIX/VLW$. When this

```
      COMPLEX C(65536),Z(65536)
      INTEGER PITER(65536)
      REAL LIM
C     Generate the Fractal image
      NV = NPIX/VLW
      DO K = 1,NITER
         DO J = 1,NV
C           This loop is vectorizable
            DO I=1+(J-1)*VLW,J*VLW
               IF(CABS(Z(I)).LT.LIM) THEN
                  Z(I) = Z(I)*Z(I) + C(I)
                  PITER(I) = K
               ENDIF
            CONTINUE
         CONTINUE
      CONTINUE
```

Figure 5: Use of Short Vectors.

code is vectorized, only the innermost loop gets vectorized due to the data dependencies in the outer two DO loops. It should be noted that the effective vector lengths of the sub-vectors reduce as the iterations proceed.

3.3 Technique 3: Use of Short Vectors with Replenishment

In the previous two techniques the effective vector lengths reduce as the iterations proceed and masked vector operations are required. Consequently, the efficiency of the vector processing reduces as iterations proceed. To alleviate these problems, we propose the use of short working vectors for pixel processing. Further, whenever processing is complete for a pixel, its data is transferred from the working vectors to the long vectors (which are used to store data for all the pixels) and the elements of the working vectors are initialized to the values for a unprocessed pixel. In this case, all the elements of the working vectors participate in the processing operations (except for the trailing end of the processing wherein the total number of pixels to be processed is smaller than the length of the working vectors). Thus, for most of the iterations, no masked vector operations are used and we expect higher efficiency for vector processing.

```
      COMPLEX C(65536),Z(65536),CW(256),ZW(256)
      INTEGER PITER(65536),INDEX(256),PITERW(256),VLW
      LOGICAL FLAG(256)
C     Further Initialization
      IDX = VLW
C     This loop is vectorizable
      DO I = 1,VLW
         ZW(I) = Z(I)
         CW(I) = C(I)
         PITERW(I) = 0
         FLAG(I) = .FALSE.
         INDEX(I) = I
      CONTINUE
```

(a) Initialization.

Figure 6: Use of Short Vectors with Replenishment.

A code segment using this technique is given in Figure 6. Here, we assume that the lengths of the working vectors ZW, CW ,$PITERW$ and $FLAG$ are equal to 256. An index vector, $INDEX$, of the same length is used to indicate the positions of the elements in the working vectors as they correspond to the long vectors. The flag vector, $FLAG$, is used to indicate the status of the pixels. A pointer, IDX, gives the index of the next pixel to be processed.

In addition to the initialization given in Figure 3, the work vectors are initialized by the code given in Figure 6(a). Subsequently, after every processing of iteration, all the pixels are checked to see if the computations are complete for any of the pixels (see Figure 6(b)). If so, then the corresponding elements from the work vectors, CW and ZW, are stored in the main vectors, C and Z, and the values for new pixel to be processed are transferred to the working vectors. This process continues until all the pixels are processed. It should be noted that checking and replenishment of working vectors is done after every iteration and the working vectors are always full except when the last few pixels have to be processed. The remaining pixels are processed by using masked vector operations as shown in Figure 6(c).

```
C     Process all the pixels
      DO WHILE(IDX .LE. NPIX)
C        Process the work vectors
C        This loop is vectorizable
         DO J = 1,VLW
            ZW(J) = ZW(J)*ZW(J) + CW(J)
            PITERW(J) = PITERW(J) + 1
            IF (PITERW(J) .EQ. NITER) FLAG(J) = .TRUE.
            IF (CABS(ZW(J)) .GE. LIM) FLAG(J) = .TRUE.
         CONTINUE
C        If flag set,
C              transfer work vector data,
C              update pointer to the long vectors, and
C              replenish the work vectors
C        This loop is not vectorizable due to indirect addressing
         DO J = 1,VLW
            IF (FLAG(J)) THEN
               PITER(INDEX(J)) = PITERW(J)
               Z(INDEX(J)) = ZW(J)
               IDX = IDX + 1
               INDEX(J) = IDX
               CW(J) = C(INDEX(J))
               ZW(J) = Z(INDEX(J))
            ENDIF
         CONTINUE
C        Reset iteration counts and flags
C        This loop is vectorizable
         DO J = 1,VLW
            IF (FLAG(J)) THEN
               PITERW(J) = 0
               FLAG(J) = .FALSE.
            ENDIF
         CONTINUE
      ENDWHILE
```

(b) Processing of Short Vectors with Replenishment.

Figure 6: Use of Short Vectors with Replenishment (continued).

```
C     Process the pixels remaining in the work vectors
      DO K = 1,NITER
C        This loop is vectorizable
         DO J = 1,VLW
            IF (.NOT. FLAG(J)) THEN
               ZW(J) = ZW(J)*ZW(J) + CW(J)
               PITERW(J) = PITERW(J) + 1
               IF (PITERW(J) .EQ. NITER) FLAG(J) = .TRUE.
               IF (CABS(ZW(J)) .GE. LIM) FLAG(J) = .TRUE.
            ENDIF
         CONTINUE
      CONTINUE
C        This loop is not vectorizable due to indirect addressing
         DO J = 1,VLW
            PITER(INDEX (J)) = PITERW(J)
            Z(INDEX(J)) = ZW(J)
         CONTINUE
```

(c) Processing of Remaining Pixels.

Figure 6: Use of Short Vectors with Replenishment (continued).

4 Results and Discussion

We have developed three FORTRAN programs based on the three techniques discussed in the previous section. These programs have been implemented on IBM 3090-180VF which has a vector facility. The vector facility consists of 16 vector registers, each having 128 scalar registers. The peak processing speed is 108 MFLOPS. The programs are compiled and executed in both scalar and vector modes for varying number of values for the maximum number of iterations ($NITER$); the compiler options used for the VS FORTRAN compiler are the highest scalar optimization level, OPT(3), the highest vectorization level, LEVEL(2).

The execution times of these programs are given in Table 1, where T_{ri} represents the execution time, where, $r \in \{s, v\}$, $i \in \{1, 2, 3\}$, s stands for scalar processing, v represents vector processing and i identifies the technique used.

From the Table 1 it can be seen that the execution time increases almost linearly as the number of pixels ($NPIX$) in the fractal image are increased.

Since Technique 2 needs additional housekeeping operations, it requires higher execution time in scalar mode compared to that for Technique 1. However, the incorporation of replenishment operations in Technique 2, which then really becomes Technique 3, results in such a significant amount of reduction in the scalar execution time that Technique 3 requires the least amount of execution time.

Technique 2 requires the highest execution time, compared to Technique 1 and 3, in the vector processing mode. Technique 3 requires the least vector execution time for all the image sizes, except for 16×16 and 32×32 pixels. For these sizes, Technique 1 is superior. This can be attributed to the fact that since the work vector lengths are equal to 256, no replenishment takes place for 16×16 image while the number of replenishment operations are small for 32×32 image.

We present the various speedup ratios between the three techniques and the two processing modes in Table 2. The significance of each of the speedup ratios is as follows :

1. T_{s1}/T_{s3} - scalar processing gain due to the use of short vectors and replenishment.
2. T_{v2}/T_{v3} - advantage of replenishment in vector processing mode.
3. T_{s2}/T_{v2} - vectorization speedup for Technique 2.
4. T_{s2}/T_{v3} - best possible speedup.
5. T_{s3}/T_{v3} - vectorization speedup for Technique 3.
6. T_{s3}/T_{v2} - ratio of the best scalar processing time and worst vector processing time.
7. T_{s1}/T_{v3} - speedup with Technique 3 using vectorization compared to the Technique 1 in scalar mode.

These speedup ratios are plotted in Figure 7 to illustrate the various trends.

Since the speedup T_{s1}/T_{s3} is greater than one (about 1.6), the Technique 3 is advantageous even when a machine does not have a vector facility. Although Technique 2 is the worst in both processing modes, vectorization does reduce the execution time as evidenced by T_{s2}/T_{v2} which is always greater than 1

Table 1: Execution Times (in milliseconds) for $NITER = 20$

Size	NPIX	T_{s1}	T_{s2}	T_{s3}	T_{v1}	T_{v2}	T_{v3}
16	256	19	19	12	9	18	13
32	1024	76	77	48	32	49	34
48	2304	171	174	108	71	92	62
64	4096	302	307	190	128	150	101
80	6400	472	479	295	199	222	148
96	9216	681	692	429	285	304	204
112	12544	928	941	583	388	402	268
128	16384	1204	1225	756	508	513	341
144	20736	1529	1555	964	641	727	473
160	25600	1883	1911	1178	794	875	569
176	30976	2284	2318	1431	958	1035	673
192	36864	2716	2758	1702	1139	1206	788
208	43264	3183	3237	1993	1336	1392	910
224	50176	3699	3742	2306	1550	1593	1042
240	57600	4239	4312	2659	1717	1807	1185
256	65536	4817	4901	3023	2024	2032	1327

Table 2: Execution Time Ratios for $NITER = 20$

Size	NPIX	$\frac{T_{s1}}{T_{s3}}$	$\frac{T_{v2}}{T_{v3}}$	$\frac{T_{s2}}{T_{v2}}$	$\frac{T_{s2}}{T_{v3}}$	$\frac{T_{s3}}{T_{v3}}$	$\frac{T_{s3}}{T_{v2}}$	$\frac{T_{s1}}{T_{v3}}$
16	256	1.52	1.36	1.05	1.43	0.94	0.69	1.46
32	1024	1.57	1.45	1.57	2.29	1.44	0.99	2.24
48	2304	1.58	1.49	1.88	2.80	1.75	1.17	2.76
64	4096	1.59	1.50	2.04	3.05	1.89	1.26	2.99
80	6400	1.60	1.50	2.16	3.24	2.00	1.33	3.19
96	9216	1.59	1.50	2.27	3.40	2.11	1.41	3.34
112	12544	1.59	1.50	2.34	3.51	2.18	1.45	3.46
128	16384	1.59	1.50	2.39	3.59	2.22	1.48	3.53
144	20736	1.59	1.54	2.14	3.29	2.04	1.33	3.23
160	25600	1.60	1.54	2.18	3.36	2.07	1.35	3.31
176	30976	1.60	1.54	2.24	3.44	2.13	1.38	3.39
192	36864	1.59	1.53	2.30	3.50	2.16	1.41	3.45
208	43264	1.60	1.53	2.33	3.56	2.19	1.43	3.50
224	50176	1.60	1.53	2.35	3.60	2.21	1.45	3.55
240	57600	1.59	1.52	2.39	3.64	2.24	1.47	3.58
256	65536	1.59	1.53	2.41	3.69	2.28	1.49	3.63

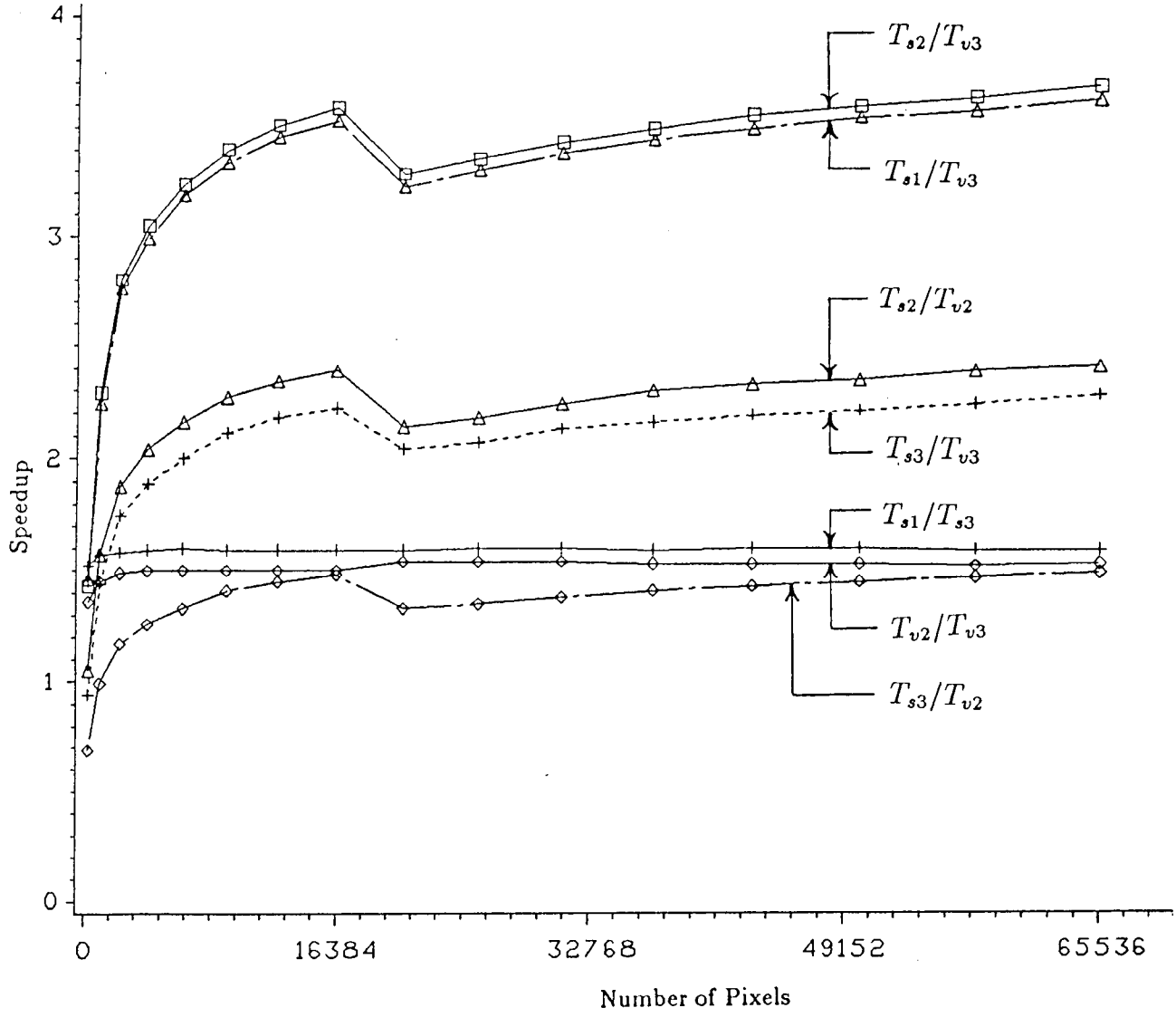

Figure 7: Speedup Ratios.

(ranging between 1.05 to 2.4). Since Technique 2 with scalar processing gives the highest execution times and Technique 3 in vector processing mode results in the least execution times, the ratio T_{s2}/T_{v3} represents the best possible speedup (range 1.4 - 3.7). The ratio T_{s3}/T_{v2} is almost always greater than 1; this implies that the vectorization of even the worst technique is advantageous over the best technique in scalar mode. Technique 1 is a straight forward method while Technique 3 uses both short vectors and replenishment. The speedup T_{s1}/T_{v3}, therefore, gives the speedup due to vectorization and program reorganization.

5 Conclusion

In this paper, we have proposed three techniques for designing programs for generating algebraic fractal images. These techniques are : (a) the use of long vectors, (b) the use of short vectors, and (c) the use of short vectors with replenishment. We have implemented these techniques on IBM 3090-180VF and studied the scalar and vector execution times for image sizes ranging from 16×16 pixels to 256×256 pixels. The technique which uses short vectors with replenishment gives the best performance both in the scalar as well as vector processing modes, even though the program based on this technique is very complex compared to the other techniques.

References

[1] Gujar, U.G., Bhavsar, V.C. and Kalra, P.K., *Vectorization and Visualization of Fractals*, Supercomputing Symposium '89, Toronto, June 19-21, 1989.

[2] Kalra, P.K., Wu, Liya, Bhavsar, V.C. and Gujar, U.G., *Vectorization of Some Computer Graphics Algorithms on IBM 3090-180VF*, Proc. of Third International Conference on Supercomputing, May 15-20, 1988, Boston, MA, International Supercomputing Institute, St.Petersburg, FL, pp.232-238, May 1988.

[3] Mandelbrot, B.B.,*The Fractal Geometry of Nature*, W.H.Freeman Company, San Fransisco, CA, 1982.

[4] Peitgen, H.O. and Richter, P., *The Beauty of Fractals*, Springer-Verlag, Berlin, 1986.

[5] Pickover, C.A., *Pattern Formation and Chaos in Networks*, Communications of the ACM, Vol. 31, No. 2, pp. 136-151, Feb.1988.

[6] Vangala, Nagarjuna, *Vectorization and Visualization of Algebraic Fractals*, M.Sc(C.S) Thesis, Faculty of Computer Science, University of New Brunswick, Fredericton, Canada, May 1990.