Efficient Bit and Digital Reversal Algorithm Using Vector Calculation

Soo-Chang Pei and Kuo-Wei Chang

***Abstract*—This correspondence describes an efficient bit and digital reversal algorithm using vector calculation. It is much more efficient and simple than calculating the bit and digital reversal sequentially one by one using for-loop. An auxiliary small-size seed table can also be used for building up larger table in our algorithm to speed up the computation time. It will be very useful for data shuffling in radix-2 and radix-4 fast Fourier transforms.**

***Index Terms*—Bit reversal, digital reversal, fast Fourier transform.**

I. Introduction

The well-known radix2 fast Fourier transform (FFT) algorithm [1] can be classified into two major classes, one is decimation-in-time (DIT) FFT, the other is decimation-in-frequency (DIF) FFT. The bit reversal data shuffling is in the first stage of DIT-FFT or the last stage of DIF-FFT to speed up the transform. A number of bit-reversal algorithm have been investigated and published in the open literature[1]–[11]. We can classify into two classes, the first class needs an auxiliary small size table [3]–[5], the second class performs efficient bit-reversal algorithm without tables.

The conventional algorithms calculate the bit reversal one by one using for-loop. The new idea is to calculate the bit-reversal index in vector form. Vector calculation is much more efficient and simple than calculating the bit reversal sequentially one by one using for-loop, especially taking the advantage of MATLAB's vector characteristics. It can be implemented very effectively using not more than four lines MATLAB procedures.

Meanwhile table look-up methods are generally much faster than calculating directly, but need a lot of memory storage. An auxiliary small size seed table [3]–[5] can be used to build a large table look-up for efficient bit-reversal. This approach can speed up the computation time with a small size table for paying the price. The similar idea can also be used in our method for table look-up using a small sized seed table. This new method for bit-reversal can be easily generalized to nonbinary digit reversal. For example, in radix-4 FFT, it will need 4-nary digital reversal process for data shuffling to speed up the transform.

Calculating bit reversal in the vector way can also produce an in-place algorithm. This algorithm is very similar to the Walker's [5], but it can take the advantage of premultiplying thus contains only logical OR operations, which makes it faster than the Walker's [5].

II. Using Vector Calculation

Conventional bit reversal algorithm uses for-loop. But we know that in some languages, such as MATLAB, calculation in vectors is faster than using for-loop. The conventional method is calculating the bit reversal one by one, and can't take the advantage of MATLAB's characteristics. So a new observation is made. Note that

$$\{\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3}\} \overset{\text{bitReverse}}{\longrightarrow} \{\mathbf{0},\mathbf{2},\mathbf{1},\mathbf{3}\} \tag{1}$$

$$\{\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{7}\} \overset{\text{bitReverse}}{\longrightarrow} \{\mathbf{0},\mathbf{4},\mathbf{2},\mathbf{6},\mathbf{1},\mathbf{5},\mathbf{3},\mathbf{7}\}. \tag{2}$$

The first four numbers in (2) are exactly what we just find in (1) multiply 2. And the last four numbers, ***they are the first four numbers multiply 2 plus one, respectively!*** Moreover

$$\{0,1,2,3,4,5,6,7\} \overset{\text{bitReverse}}{\longrightarrow} \{0,4,2,6,1,5,3,7\} \tag{3}$$

$$\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\} \overset{\text{bitReverse}}{\longrightarrow} \{0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15\}. \tag{4}$$

It is very similar to the first one, so we can conclude a MATLAB procedure as follows:

Step 1) $x(1) = 0$

Step 2) for 1: $\log_2 N\mathbf{x} = [2^*\mathbf{x}2^*\mathbf{x}+1]$; end

Step 3) Now x is the vector of bit reversal of $[0:N-1]$, but in MATLAB, index begins from 1, so if a is the input, we should write $a = a(x+1)$ to complete bit reversal.

We now prove the method is correct. Let $N = 2^m$, first we know that the bit reversal of 0 is still 0. Assume the bit reversal of $\mathbf{u_k} = [0,1,\ldots,2^k-1]$ is $\mathbf{v_k}$, $k = 1 \sim m-1$. Because we know that $\mathbf{u_m} = [0^*2^{m-1} + \mathbf{u_{m-1}}, 1^*2^{m-1} + \mathbf{u_{m-1}}]$, the bit reversal of $\mathrm{u_m}$ is $[\mathrm{Br}(\mathbf{u_{m-1}})$ with left shift one bit and the added bit is 0, $\mathrm{Br}(\mathbf{u_{m-1}})$ with left shift one bit and the added bit is $1] = [2^*\mathbf{v_{m-1}}, 2^*\mathbf{v_{m-1}}+1]$. Where "+" is the "+" in vector, i.e., $[1,2,3]+1=[2,3,4]$. So from the mathematic induction, the method is correct for all $m \in N$.

III. Table Look-Up Method

Generally speaking, table lookup methods are much faster than calculating directly, but waste a lot of space. So a lot of compromise method has been built, such as [5], [12]. They use smaller size table, cooperate with the characteristic of some symmetry or the algebra and obtain faster speed. In [5] and [12], the table size is $\sqrt{N}$ Similarly, we can use the same size of table and build a new table lookup method suit for MATLAB. Observing that

$$\{0,1,2,3\} \overset{\text{bitReverse}}{\longrightarrow} \{0,2,1,3\} \text{ table of } N=4$$

$$\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\} \overset{\text{bitReverse}}{\longrightarrow} \{0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15\} \text{ table of } N=16.$$

One can easily tell that

$$\{\mathbf{0},\mathbf{8},\mathbf{4},\mathbf{12}\} \text{ is } \{0,2,1,3\}\times 4+0,$$
$$\{\mathbf{2},\mathbf{10},\mathbf{6},\mathbf{14}\} \text{ is } \{0,2,1,3\}\times 4+2,$$
$$\{\mathbf{1},\mathbf{9},\mathbf{5},\mathbf{13}\} \text{ is } \{0,2,1,3\}\times 4+1,$$
$$\text{and } \{\mathbf{3},\mathbf{11},\mathbf{7},\mathbf{15}\} \text{ is } \{0,2,1,3\}\times 4+3$$

where $+$ is the $+$ in MATLAB, i.e., $\{1,2,3\}+1=\{2,3,4\}$. Note that the "add term" $\{0,2,1,3\}$ is added separately. And $\{0,2,1,3\}$ is also the bit reversal of $N=4$ case. So a MATLAB procedure can be written with this method. The details are as follows:

Step 1) get a lookup table of size $\sqrt{N}$, called x.

Step 2) $\mathbf{y} = \sqrt{N}^*\mathbf{x}_*^T$ ones $(1,\sqrt{N}) +$ ones $(\sqrt{N},1)^*\mathbf{x}$;

Manuscript received November 24, 2005; revised April 24, 2006. This work was supported by the National Science Council of Taiwan, R.O.C., under Contracts 93-2219-E-002-004 and NSC 93-2752-E-002-006-PAE. The associate editor coordinating the review of this correspondence and approving it for publication was Dr. Yuan-Pei Lin.

The authors are with the Department of Electrical Engineering, National Taiwan University Taipei, Taiwan, 10617, R.O.C. (e-mail: pei@cc.ee.ntu.edu.tw).

Digital Object Identifier 10.1109/TSP.2006.887567

TABLE I
COMPARISON FOR RUNNING TIME OF BIT-RIVERSAL BY DIFFERENT ALGORITHMS (BY MATLAB 6.0)

	Walker's[5]	Prado's[12]	Section III
Table size = 2^4	1.625 ms	1.797 ms	0.079 ms
Table size = 2^5	5.782 ms	7.602 ms	0.141 ms
Table size = 2^6	22.25 ms	28.91 ms	0.515 ms
Table size = 2^7	88.83 ms	117.1 ms	2.109 ms

Step 3) $\mathbf{y} = y(:)^T$; % rearrange $\mathbf{y}$, then $\mathbf{y}$ is the bit reversal of $[0:N-1]$. Similar to the algorithm in Section II, we should use $a = a(x+1)$ to complete bit reversal.

Step 2 seems difficult, so we explain this in figure: Use our previous example, assume we have table of $\sqrt{N} = 4$, i.e., $\mathbf{x} = [0213]$. And we want to calculate $N = 16$.

$$\mathbf{y} = 4^* \begin{bmatrix} 0 \\ 2 \\ 1 \\ 3 \end{bmatrix} \times [1\ 1\ 1\ 1] + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \times [0\ 2\ 1\ 3]$$
$$= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 8 & 8 & 8 & 8 \\ 4 & 4 & 4 & 4 \\ 12 & 12 & 12 & 12 \end{bmatrix} + \begin{bmatrix} 0 & 2 & 1 & 3 \\ 0 & 2 & 1 & 3 \\ 0 & 2 & 1 & 3 \\ 0 & 2 & 1 & 3 \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 2 & 1 & 3 \\ 8 & 10 & 9 & 11 \\ 4 & 6 & 5 & 7 \\ 12 & 14 & 13 & 15 \end{bmatrix}.$$

Step 3 is to convert the above matrix to [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15], which is exactly the bit reversal of $N = 16$.

The proof of this method is very similar to the method discussed in Section II. But this time we do not need mathematical induction. Let $\mathbf{v_m} = [v_{m0}, v_{m1}, \ldots, v_{m(2^m-1)}]$ be the bit reversal of $\mathbf{u_m} = [0, 1, \ldots, 2^m - 1]$. We want to prove that

$$\mathbf{v_{m2}} = [\mathbf{v_m^*} 2^m + v_{m0}, \mathbf{v_m^*} 2^m + v_{m1}, \ldots, \mathbf{v_m^*} 2^m + v_{m(2^m-1)}].$$

To begin with, rewrite $\boldsymbol{u}_{\mathbf{m2}} = [0, 1, \ldots, 2^{m^2} - 1] = [u_m + 0^* 2^m, u_m + 1^* 2^m, \ldots, u_m + (2^m - 1)^* 2^m]$. Then we know that $v_{m^2} = [v_m$ shift m bits and add $\mathrm{v_{m0}}, v_m$ shift m bits and add $\mathrm{v_{m1}}, \ldots, v_m$ shift m bits and add $\mathrm{v_{m(2^m-1)}}] = [v_m^* 2^m + v_{m0}, v_m^* 2^m + v_{m1}, \ldots, v_m^* 2^m + v_{m(2^m-1)}]$. Thus, the proof is completed.

IV. GENERALIZATION TO DIGIT REVERSAL

In modern FFT, we want to calculate non-binary reversal, or digit reversal. Unfortunately, the method in [12] uses operation like XOR, which is hard to do in nonbinary case. However, the new views of bit reversal discussed above can easily generalize to nonbinary case, like radix-4. Take $N = 4^2$ as an example.

$$\{0,1,2,3\} \overset{\mathrm{digit-Reverse}}{\longrightarrow} \{0,1,2,3\} (\mathrm{Radix-4\ case}) \quad (5)$$
$$\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$$
$$\overset{\mathrm{digit-Reverse}}{\longrightarrow} \{\mathbf{0},\mathbf{4},\mathbf{8},\mathbf{12},1,5,9,13,2,6,10,14,3,7,11,15\}. \quad (6)$$

One can easily observe that the first four numbers of (6) are $4^*(5)$. We can say that

$$(6) = \{4^*(5)+0, 4^*(5)+1, 4^*(5)+2, 4^*(5)+3\}$$

or in general:

If $N = k^m$, $\mathbf{v_m}$ is the k-nary digit reversal of $[0:N-1]$, then

$$\mathbf{v_m} = [k^* \mathbf{v_{m-1}} + 0, k^* \mathbf{v}_{m-1} + 1, k^* \mathbf{v}_{m-1} + 2, \ldots, k^* \mathbf{v_{m-1}} + k - 1]. \quad (7)$$

Not surprisingly, the new table lookup method can also be generalized.

If $N = k^m$, $\mathbf{v_m}$ is the k-nary digit reversal of $[0:N-1]$, and $\mathbf{v_m} = [v_0, v_1, \ldots, v_{N-1}]$. Then

$$\mathbf{v_{m2}} = [N^* \mathbf{v_m} + v_0, N^* \mathbf{v_m} + v_1, \ldots, N^* \mathbf{v_m} + v_{N-1}]. \quad (8)$$

(It can be easily checked that the discussions here are special case of $k = 2$.)

Notice that (7) and (8) are also powerful when using vector calculation, so the implementation in MATLAB is straightforward.

V. NEW IN-PLACE ALGORITHM

All of the methods aforementioned calculate the indices first, and then do the data bit reversal. Such algorithms use $O(N)$ memories, and many of the memories are wasted because there are $\sqrt{N}$ data after bit reversal are themselves. In order to use less memories and develop an in-place algorithm, let us consider the example of $N = 16$ again. Suppose the input $\mathbf{A}$is a matrix.

$$\mathbf{A} = \begin{bmatrix} a0 & a4 & a8 & a12 \\ a1 & a5 & a9 & a13 \\ a2 & a6 & a10 & a14 \\ a3 & a7 & a11 & a15 \end{bmatrix} \overset{\mathrm{bitReverse}}{\longrightarrow} \begin{bmatrix} a0 & a2 & a1 & a3 \\ a8 & a10 & a9 & a11 \\ a4 & a6 & a5 & a7 \\ a12 & a14 & a13 & a15 \end{bmatrix}. \quad (9)$$

This can be replaced by three vector-matrix operations.

Step 1)

$$\mathbf{A} = \begin{bmatrix} a0 & a4 & a8 & a12 \\ a1 & a5 & a9 & a13 \\ a2 & a6 & a10 & a14 \\ a3 & a7 & a11 & a15 \end{bmatrix}$$
$$= [\mathbf{a[0]}\ \mathbf{a[1]}\ \mathbf{a[2]}\ \mathbf{a[3]}] \overset{\mathrm{bitRofvectors}}{\longrightarrow} [\mathbf{a[0]}\ \mathbf{a[2]}\ \mathbf{a[1]}\ \mathbf{a[3]}]$$
$$= \begin{bmatrix} a0 & a8 & a4 & a12 \\ a1 & a9 & a5 & a13 \\ a2 & a10 & a6 & a14 \\ a3 & a11 & a7 & a15 \end{bmatrix}.$$

Step 2) Transpose the above matrix.

$$\begin{bmatrix} a0 & a8 & a4 & a12 \\ a1 & a9 & a5 & a13 \\ a2 & a10 & a6 & a14 \\ a3 & a11 & a7 & a15 \end{bmatrix} \xrightarrow{\text{Transpose}} \begin{bmatrix} a0 & a1 & a2 & a3 \\ a8 & a9 & a10 & a11 \\ a4 & a5 & a6 & a7 \\ a12 & a13 & a14 & a15 \end{bmatrix}.$$

Step 3) Do the same thing as in Step 1. And the result is the right-hand side of (9).

An in-place algorithm must use for-loop. While using for-loop, the step 1 and 3 can be avoided technically. Note transposing a matrix means that the $\sqrt{N}$ diagonal elements are unchanged, as we discussed above, so the step 2 only exchanges $((N-\sqrt{N})/2)$ numbers. The pseudo code is given below. Let B is the input array and $\boldsymbol{x}$ is the look-up table.

$$\text{For } i = 0 \; to \; \sqrt{N} - 2$$
$$\text{For } j = i + 1 \text{ to } \sqrt{N} - 1.$$

$a = \sqrt{N}^* \boldsymbol{x}(i) + j; b = \sqrt{N}^* \boldsymbol{x}(j) + i$; exchange the elements $\boldsymbol{B}$(a) and $\boldsymbol{B}$(b).

The proof of the correctness of this algorithm is similar to the Walker's [5], so it is omitted here. Looking at this algorithm closely, one can easily find the multiplying is unnecessary if the $\boldsymbol{x}$ is premultiplied by $\sqrt{N}$. Furthermore, since i and j are less than $\sqrt{N}$, and $\boldsymbol{x}$ is larger than $\sqrt{N}$ if we premultiply it, the plus addition can be done by logical OR operation, if $\sqrt{N}$ is a power of 2. In summary, this algorithm can be implemented only by 'read', 'write' and 'OR' operations, which makes it faster than the Walker's [5]. The revised algorithm is given below. Again, Let $\boldsymbol{B}$ is the input array and $\boldsymbol{x}$ is the look-up table.

$$\boldsymbol{x} = \sqrt{N}^* \boldsymbol{x}$$
$$\text{For } i = 0 \text{ to } \sqrt{N} - 2$$
$$\text{For } j = i + 1 \text{ to } \sqrt{N} - 1.$$

$a = \boldsymbol{x}(i)|j; \; b = \boldsymbol{x}(j)|i$; exchange the elements $\boldsymbol{B}$(a) and $\boldsymbol{B}$(b).

VI. Comparison

The methods in Section II and III need N shifts, N additions, and an index adjusting. Both of them use $O(N)$ memories. The major difference of these two methods is that the table lookup method can be applied for *parallel computing*. For example, for $\mathrm{N} = 16$ and table $\mathrm{size} = 4$, if we just want to calculate the output $y[12] \sim y[15]$, we need not wait until the calculation of $y[0] \sim y[11]$ is completed. We can directly compute $\{0,2,1,3\} \times 4 + 3 = \{3,11,7,15\}$ and output these four values $x[3], x[11], x[7]$,and $x[15]$. Table I shows the comparison of the running time between three different algorithms using MATLAB 6.0. Each of the time is averaged on 1000 runs. One can notice that the new method of Section III is significantly faster than the others, because of MATLAB's characteristics.

Table II is, the running time using C-program as mex-function and running in MATLAB 6.0. Because C usually runs faster than MATLAB, the time is averaged on 10 000 runs. As we know, C doesn't have vector calculation. So we compare the proposed method discussed in Section V with the others.

TABLE II
Comparison for Running Time of Bit-Riversal by Different Algorithms (Using C-Program as Mex-Function, Running in Matlab 6.0)

	Walker's[5]	Prado's[12]	Proposed
Table size = 2^4	0.0109 ms	0.0094 ms	0.0078 ms
Table size = 2^5	0.0266 ms	0.0281 ms	0.0266 ms
Table size = 2^6	0.0968 ms	0.1063 ms	0.0938 ms
Table size = 2^7	0.3884 ms	0.4047ms	0.3562 ms

From Tables I and II, one can easily find that C version runs much faster than the MATLAB one. This is due to the fact that the C version can make much better use of the CPU data cache than the MATLAB version, when the table is very large. Also, the C version in Section V needs fewer operations than the MATLAB version, as discussed in Section V. Besides, although it has been proven in [12] that the proposed algorithm could run twice faster than the algorithm in [5], the running times do not reflect this fact because the mex-function implementation.

VII. Conclusion

In this correspondence, we showed that bit reversal algorithm can be presented in vector way, which is suit for some languages like MATLAB. We also provide the method of table lookup method, with similar math structure and proof. Besides, the new points of view make it possible to generalize bit reversal, which can be used in FFT transforms. Finally, a new fast in-place algorithm is derived by the vector view.

References

[1] A. Karp, "Bit reversal on uniprocessors," *SIAM Rev.*, vol. 38, no. 1, pp. 1–26, Mar. 1996.

[2] A. Elster, "Fast bit-reversal algorithms," in *Proc, IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, 1989, pp. 1099–1102.

[3] D. Evans, "An improved digital-reversal permutation algorithm for the fast Fourier transforms," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-35, pp. 1120–1125, Aug. 1987.

[4] D. Evans, "A second improved digital-reversal permutation algorithm for the fast fourier transforms," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 37, pp. 1288–1291, Aug. 1989.

[5] J. Walker, "A new bit-reversal algorithm," *IEEE Trans. Signal Process.*, vol. 38, no. 8, pp. 1472–1473, Aug. 1989.

[6] P. Duhamel and J. Prado, "A connection between bit-reverse and matrix transpose, hardware and software consequences," in *Proc, IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP)*, 1988, pp. 1403–1406.

[7] B. Gold and B. Rader, *Digital Processing of Signal*. New York: McGraw-Hill, 1969.

[8] A. Yong, "A better FFT bit-revesal algorithm without tables," *IEEE Trans. Signal Process.*, vol. 39, no. 10, pp. 2365–2367, Oct. 1991.

[9] M. Orchard, "Fast bit-reversal algorithms based on index representations in GF(2^b)," *IEEE Trans. Signal Process.*, vol. 40, no. 4, pp. 1004–1008, Apr. 1992.

[10] J. Rius and R. D. Porrata-Dorin, "New FFT bit-reversal algorithm," *IEEE Trans. Signal Process.*, vol. 49, no. 1, pp. 251–254, Jan. 2001.

[11] K. Drouiche, "A new efficient computational algorithm for bit reversal mapping," *IEEE Trans. Signal Process.*, vol. 49, no. 1, pp. 251–254, Jan. 2001.

[12] J. Prado, "A new fast bit-reversal permutation algorithm based on a symmetry," *IEEE Signal Process. Lett.*, vol. 11, pp. 933–936, Dec. 2004.