FAST BIT-REVERSAL ALGORITHMS

Anne Cathrine Elster

School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Abstract

Several numerical computations, including the Fast Fourier Transform (FFT), require that the data is ordered according to a bit-reversed permutation. In fact, for several standard FTT programs, this pre or post computation is claimed to take 10-50 percent of the computation time [1]. In this paper, a *linear* sequential bit-reversal algorithm is presented. This is an improvement by a factor of $\log_2 n$ over the standard algorithms. Even at the register level (where additions and multiplications are not considered to be constant operations), the algorithm presented is shown to be linear with a low constant factor,

The recursive method presented extends nicely to *radix−r* permutations; *mixed−radix* permutations are also discussed. Most importantly, however, the method is shown to provide an efficient *vectorizable* bit-reversal algorithm.

1. Introduction

The bit-reversal permutation is a common data ordering, *its most prominent application being the pre-computation step* of the Cooley-Tukey Fast Fourier Transform (FFT) algorithm [2,3]. Other applications include image transpositions [4] and generalized sorting of multidimensional arrays [5].

Bit-reversal might be defined for $n = 2^t$ as the $n \times n$ permutation matrix P_n such that:

$$z = P_n^T x \implies z(k) = x(r_n(k)) \qquad k = 0 \cdots n-1, \quad (1)$$

where $r_n(k)$ is the integer obtained by reversing the bit-order in k's t-bit representation:

$$\begin{aligned}(k)_2 &= b_0 \cdots b_{t-1} \\ \implies (r_n(k))_2 &= b_{t-1} \cdots b_0\end{aligned} \quad (2)$$

Example: n = 8

$(k)_{10}$	$(k)_2$	$(r_8(k))_2$	$(r_8(k))_{10}$
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

The question is how fast such a permutation can be obtained for a given sequence $0 \cdots n-1$? Parallelization?

2. Previous Work

Several algorithms have been developed to compute bit-reversed indices. The most common are derivatives of techniques testing each bit of the binary representation of each index. Consequently, the most common technique for computing the bit-reversed ordering of a sequence is by a series of shifts and additions, e.g. the technique included in Cooley and Tukey's original paper [2,6].

In 1969, Fraser [7] demonstrated that by working entirely in bit-reversed form (no regular integer increments), a sequence of bit-reversed integers could be generated by finding the leftmost **0**, replacing it by **1** and clearing any leading ones. A possible order of magnitude speed-up was achieved by taking advantage of floating point hardware. If the bit-reversed number, treated as a normal fraction was negative (from a corresponding odd non-reversed integer), a conversion to floating point and back automatically gave the desired leftmost **0**. Interchanging it and performing shifts equal to the magnitude of the exponent yielded the desired result. Bit-reversed integers corresponding to the even entries may then be generated by flipping the most significant bit of the preceding integer in the bit-reversed sequence generated.

In 1984, Johnson and Burrus [8] presented an in-order, in-place radix-2 FFT algorithm which eliminated the need for the bit-reversal permutation of data (which they claimed takes 10-50% of the computation time). A double butterfly operation was, however, needed to avoid permuting the data.

In 1985, Fraser [5] proposed a bit-reversal algorithm attempting to minimize main memory accesses for large memory systems. He achieved this by noting that the bit-reversal permutation can be achieved by a series of cyclic shifts:

e.g. $n = 2^4 = 16$

$$R_4 = ({}_0S_4 \cdot {}_1S_4 \cdot {}_2S_4), \quad (3)$$

where R_4 is the bit-reversal permutation on 4 bits, and ${}_0S_4$, ${}_1S_4$, and ${}_2S_4$ cyclically shift 4,3, and 2 bits of each number respectively. Combining cyclic shifts and direct bit-reversal permutations, fewer main-memory accesses are required, though the time complexity of this algorithm still remains at $[O(n \log_2 n)]$.

CH2673-2/89/0000-1099 $1.00 © 1989 IEEE

In 1987, Evans [9] proposed a linear digit-reversal permutation algorithm (which is bit-reversal for base=2) which uses a *seedtable* of pre-calculated digit-reversed numbers.

Finally, in 1988, Burrus [1] showed that any radix-2^t-FFTs and mixed-radix FFTs can be written to scramble the data in a *bit*-reversed order.

3. A Fast Bit-Reversal Algorithm

Any algorithm which implicitly checks each bit of each number of a regular sequence in order to create a bit-reversed permutation, will clearly be of $O(n \log_2 n)$. The trick is to view the computation as a *mapping* from one sequence to another. The problem is then to search for this mapping function. A linear recursive sequential algorithm was found by factoring out the bit-reversed number sequence with powers of two. Figure 1 shows how this was done for $n = 16 = 2^t = 2^4$.

Notice that $r_{16}(k) = c_k \cdot 2^{t-q}$ with c_k odd.

Definition 1:

If $n = 2^t$, $1 \le q < t$, and $2^{q-1} \le k \le 2^q$

then the odd constant c_k is defined by: (4)

$$r_n(k) \equiv c_k \cdot 2^{t-q}, \quad c_k \quad [\textit{odd integer}]$$

That c_k is an odd integer can be deduced from the definition:

Definition 2:

$$\begin{aligned} &r_n(k) \equiv c_k \cdot 2^{t-q} \\ &k = (0 \cdots 01 b_{q-2} \cdots b_0)_2 \Longrightarrow \\ &r_n(k) = (b_0\, b_1 \cdots b_{q-2}\, 1)_2 \cdot 2^{t-q}. \end{aligned} \tag{5}$$

Hence c_k will always have a 1 in the least significant bit-position, so c_k is odd.

The stippled lines in Figure 1 discern the recursive pattern of the c_ks. This pattern is more visually shown in Figure 2.

Notice that $c_{2k} = c_k$, and $c_{2k+1} = c_k + L$. This is formalized in the following theorem:

Theorem:

Let $n = 2^t$ and $1 \le q < t$.
If k satisfies

$$L_0 \equiv 2^{q-1} \le k < 2^q \equiv L \tag{6}$$

then

$$c_{2k} = c_k \tag{7}$$

$$c_{2k+1} = c_k + L. \tag{8}$$

Proof of Theorem:

Because $L_0 \le k < L$, k's and $r_n(k)$'s binary representations must be of the form:

$$k = (0 \cdots 01 b_{q-2} \cdots b_0)_2 \tag{9}$$

$$r_n(k) = (b_0 b_1 \cdots b_{q-2} 10 \cdots 0)_2. \tag{10}$$

Example:

$$n = 2^t = 2^4 = 16$$

$(k)_2$	k	$r_{16}(k)$		q
0000	0	$0 = 0 \cdot 2^4$	0000	0
0001	1	$8 = 1 \cdot 2^3$	1000	1
0010	2	$4 = 1 \cdot 2^2$	0100	2
0011	3	$12 = 3 \cdot 2^2$	1100	
0100	4	$2 = 1 \cdot 2^1$	0010	
0101	5	$10 = 5 \cdot 2^1$	1010	
0110	6	$6 = 3 \cdot 2^1$	0110	3
0111	7	$14 = 7 \cdot 2^1$	1110	
1000	8	$1 = 1 \cdot 2^0$	0001	
1001	9	$9 = 9 \cdot 2^0$	1001	
1010	10	$5 = 5 \cdot 2^0$	0101	
1011	11	$13 = 13 \cdot 2^0$	1101	4
1100	12	$3 = 3 \cdot 2^0$	0011	
1101	13	$11 = 11 \cdot 2^0$	1011	
1110	14	$7 = 7 \cdot 2^0$	0111	
1111	15	$15 = 15 \cdot 2^0$	1111	

Figure 1: Factorizing $r_{16}(k)$.

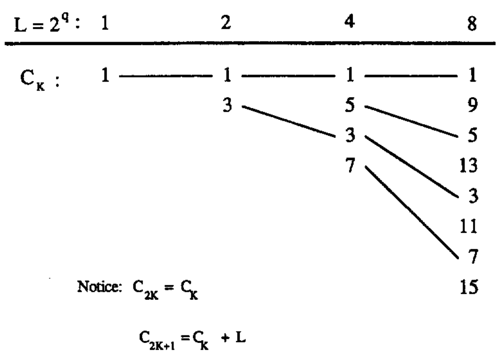

Figure 2: Discerning the recursive pattern of the C_Ks.

Consequently, from the definition of c_k we have:

$$\begin{aligned} &r_n(k) = (b_0 b_1 \cdots b_{q-2} 1)_2 \cdot 2^{t-q} \\ &\Longrightarrow c_k = (b_0 b_1 \cdots b_{q-2} 1)_2. \end{aligned} \tag{11}$$

However, since $2k$ and $2k+1$ are as follows:

$$2k = (0 \cdots 01 b_{q-2} \cdots b_0 0)_2 \tag{12}$$

$$2k + 1 = (0 \cdots 01 b_{q-2} \cdots b_0 1)_2 \tag{13}$$

We thus have the factorizations:

$$r_n(2k) = (0b_0 \cdots b_{q-2}1) \cdot 2^{t-q-1} \tag{14}$$

and

$$r_n(2k+1) = (1b_0 \cdots b_{q-2}1) \cdot 2^{t-q-1}. \tag{15}$$

The theorem follows. □

It can be deduced from the above theorem that the c_ks can be generated *recursively*; i.e. having generated $c_{L_0}, \cdots, c_{L-1}$, then $c_L, \cdots, c_{2L-1}$ may readily be computed from $c_{L_0}, \cdots, c_{L-1}$. The *linear* sequential algorithm follows by noticing that the 2^{t-q} factors follow a similar *recursive* pattern.

In the advent of the increasing number of parallel computers, the most interesting case is, perhaps, how the method presented in the previous section can help improve parallel versions of the bit-reversal algorithm. Algorithm 1a with its "vector" notion, shows, in fact, the parallelization of the method. The inner loop operates on independent data and can hence be performed in one parallel step. This yields an $O(\log_2 n)$ algorithm for the parallel case (order of main loop).

Parallelizing the "standard" algorithm, one could look at the corresponding bits of the representation in parallel. The same performance in the *big–Oh* sense can thus be achieved. However, n processors would be required throughout the computation, whereas the proposed algorithm requires only $n/2$ processors, and that occurs only during the last computational step.

Algorithm 1a: Fast Bit-Reversal

$x \in C^n$, $n = 2^t$

$x \longleftarrow P_n \cdot x$

```
z := x ; c(1) := 1 ; x(1) := z(n/2)
For q = 2 to t
  L := 2^q , r := n/L, L0 := L/2
  (* Find (Pn x) *)
  For j = 0 to L0
    c(L + 2j) := c(L0 + j)
    c(L + 2j + 1) := c(L0 + j) + L
    x(L + 2j) := z(c(L + 2j) · r)  (* r = 2^(t-q) *)
    x(L + 2j + 1) := z(c(L + 2j) + L) · r)
  end
end
```

Notice that this algorithm in the sequential setting requires only $O(n)$ memory accesses and $O(n)$ integer arithmetic. However, an integer n-vector workspace (c) is needed for storing the bit-reversed index, and a complex n-vector workspace (z) is needed as a temporary array. The integer workspace may be reduced to $n/2$ by noticing the following relation:

$$\begin{aligned} &c(n/2) \cdots c(n-2)\; c(n-1) = \\ &c(0) + 1 \cdots c(n/2-2) + 1 \quad c(n/2-1) + 1 \end{aligned} \tag{16}$$

Hence, the second-half indices can be generated by adding one to the corresponding first-half indices.

Though factorizing $r_n(k)$ was helpful to discern its recursive nature, factorization is not necessary for obtaining a *linear* sequential algorithm. A bit-reversed sequence may also be generated recursively directly by following the same method since the factors follow the same logarithmic pattern (both follow q). Algorithm 1b shows such a subroutine implemented in Fortran for the IBM 3090.

Algorithm 1b: Linear Bit-Reversal

```
  SUBROUTINE BITREV (t,c)

  IMPLICIT INTEGER*4  (A-C), (a-c)
  DIMENSION          c(0:*)
  n = 2 ** t
  L = 1
  c(0) = 0
  DO  1   q = 0, t-1
     n = n/2
     DO  2   j = 0, L-1
        c(L+j) = c(j) + n
2    CONTINUE
  L = L * 2
1 CONTINUE
  RETURN
  END
```

4. O(n) Register-Level Algorithm

Having shown that sequential bit-reversal takes $O(n)$ in an algorithmic sense, the question of whether this can be achieved at the register level remains. Also, by considering the method at this level, a better understanding of the magnitude of the constant related to the linear factor can be achieved.

In order to obtain a true linear register-level algorithm, additions and multiplications cannot be allowed since these operations are not constant at the register level. (Fast addition uses carry look-ahead adders, fast multiplications, carry-save adder trees -- both operations of about $O(\log_2(\text{no.ofbitsinrepresentation}))$. The following operations are, however, considered linear at the register level:

LOAD
STORE
SHIFT (left or right)
MASK (e.g. OR each elem. in reg. with a given mask)

By studying Figure 1 and relating the recursive pattern discerned to what is happening on the bit-level the following algorithm is achieved for generating the coefficients c_k for $k = 0 \cdots n-1$ (Algorithm 2).

Notice that no actual additions, multiplications, or divisions were used (division and multiplication by 2 are simple shift operations).

Since $2^0 = 1$, by computing the c_ks, $r_n((n-1)/2) \cdots r_n(n-1)$ have been generated. The rest of the bit-reversed indices may be generated recursively by using the even entries of the subsequent q-range and left-shifting them (multiply by 2) as shown in Algorithm 3.

Algorithm 2: Register-Level Generation of c_ks:

```
Assume n = 2^t

c(0) := 0
c(1) := 1     (* base case *)
L := 2
WHILE L < n
    FOR i = L TO 2*L - 1 (* expand*)
        IF {integer is odd}
            c(i) := c(i - i/2)   (* load *)
        ELSE {even} (* load and OR in bit set by L: *)
            c(i) := c(i-1) + L
    END (* for*)
    L := L * 2  (* left-shift *)
END (* while *)
```

Algorithm 3: Linear Register-Level Bit-Reversal

```
L := n/2
FOR q = 1 TO t-1
    FOR i = L/2 TO L by 2  (* L/2 : 2 : L *)
        x(i) : = x(i/2)
        x(i) := left-shift(x(i))
        L := L/2
    END
END
```

Notice how low the constant factor for this linear method is. The total number of operations required was $n/2$ loads (odd), $n/2$ transfers with *OR*-masking, and $n/2$ transfers followed by shifts. Comparing this to the n loads and $n \cdot \log_2 n$ shifts required for the "standard" case, it was shown how the constant is kept very small.

5. Radix-r and Mixed-Radix Algorithms

The extension to radix representations other than the radix-2 (binary) case, is easily achieved by discerning a similar recursive pattern. By factoring the reversed integers as powers of r rather than as powers of 2, a radix-r index-reversal is achieved. For the mixed-radix case, the patterns are discerned in groups for each radix type. In the case of FFTs, the plain bit-reversal algorithm could be used if care is taken [1].

6. Conclusions

A novel fast algorithm for computing a sequence of bit-reversed integers was presented. By finding a mapping function from a sequence of integers to a sequence of their bit-reverse, a recursive approach was taken to overcome the logarithmic factor burdening the standard scheme. The associated constant for the timing factor was also shown to be very low--even at the register level. Most importantly, however, the method generalized for radix-r and mixed radix cases, as well as provided an efficient vectorizable scheme with the same low constant. Algorithmic details for radix-r and mixed radix permutations are outlined in the technical report associated with this paper.

Acknowledgements

The author wishes to thank her advisor, Prof. Charles Van Loan, for his encouragement and support, and for the help with formalizing Algorithm 1a. Gratitude is also extended to Dr. James W. Cooley and Prof. C. Sidney Burrus for pointing out references, and to Dr. Fred Gustavson for demonstrating how the recursive linear approach might be encoded on the IBM 3090 (Alg. 1b).

References

1. C. S. Burrus, "Unscrambling for Fast DFT Algorithms," *IEEE Transactions on ASSP*, **36**,(7), pp. 1086-1087, (July 1988).
2. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier Series," *Math. of Comput.*, **19**,(90), pp. 297-301, (Apr. 1965).
3. W. T. Cochran and J. W. Cooley et al., "What is the Fast Fourier Transform?," *Proceedings of the IEEE*, **55**,(10),(Oct. 1967).
4. D. Fraser, R. A. Schowegert, and I. Briggs, "Rectification of Multichannel Images in Mass Storage Using Image Transposition," *Computer Vision, Graphics, and Image Processing*, **29**, pp. 23-36, (1985).
5. D. Fraser, "Bit-Reversal and Generalized Sorting of Multidimensional Arrays," *Signal Processing*, **9**,(3), pp. 163-176, North-Holland, (Oct. 1985).
6. J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Fast Fourier Transform and Its Applications," *IEEE Transactions on Education*, **E-12**,(1), pp. 27-34, (Mar. 1969).
7. D. Fraser, "Incrementing a Bit-Reversed Integer," *IEEE Transactions on Computers*, p. 74, (Jan. 1969).
8. H. W. Johnson and C. S. Burrus, "An In-Place, In-Order Radix-2 FFT," *ICASSP 1984 Proceedings*, (1984).
9. D. M. W. Evans, "An Improved Digit-Reversal Permutation Algorithm for the Fast Fourier and Hartley Transforms," *IEEE Transactions on ASSP*, **35**, pp. 1120-1125, (Aug. 1987).