Signal Processing 82 (2002) 643–648 www.elsevier.com/locate/sigpro

Extendible look-up table of twiddle factors and radix-8 based fast Fourier transform

Qing Li[a,b,*], Nengchao Wang[c], Baochang Shi[c], Chuguang Zheng[b]

[a] *School of Computer Eng. & Sci., Shanghai University, Shanghai 200072, People's Republic of China*
[b] *State Key Laboratory of Coal Combustion, Huazhong University of Science and Technology, Wuhan 430074, People's Republic of China*
[c] *Parallel Computation Institute, Huazhong University of Science and Technology, Wuhan 430074, People's Republic of China*

Received 25 February 2001

Abstract

An extendible look-up table of the *twiddle factors* for implementation of fast Fourier transform (FFT) is introduced in this paper. In fact, this twiddle factors table is independent of the length of sequence. It need not be recomputed for shorter sequences. And for longer sequences, the table can be extended easily. A radix-8 based FFT algorithm for 2^m-FFT with this table is presented. Experimental comparisons between our algorithm and FFTW software package have be done. And the results indicate that our FFT scheme is effective. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Fast Fourier transform; Twiddle factors; Bit-reversed order

1. Introduction

The Fourier transform (FT) arises in many fields of science, like signal processing, image processing, bioinformatics, applied mathematics, etc. It is the basic kernel to compute convolution of two given sequences. The most popular algorithm for computing a FT is the fast Fourier transform (FFT) algorithm.

The discrete Fourier transform (DFT) of a sequence $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ of size n is the sequence $\boldsymbol{X} = (X_0, X_1, \ldots, X_{n-1})$, of size n, defined by

$$X_j = \sum_{k=0}^{n-1} x_k W_n^{jk}, \quad j = 0, 1, \ldots, n-1, \tag{1}$$

where $W_n = \mathrm{e}^{-2\mathrm{i}\pi/n}$, and $\mathrm{i} = \sqrt{-1}$.

The first algorithm of FFT was proposed by Cooley and Tukey [3]. The FFT algorithm uses a greatly reduced number of arithmetic operations as compared to the brute force computation of the DFT. Since then, a large number of variations of the original algorithms have been proposed. They only differ in the way of storing intermediate data. The basic idea of these algorithms is the splitting of data entry $\boldsymbol{x}$ into two subsets at each step of the algorithm, and combine them using a butterfly scheme.

* Corresponding author. School of Computer Eng. & Sci., Shanghai University, Shanghai 200072, People's Republic of China.

E-mail address: qingli@public.wuhan.cngb.com (Q. Li).

PII: S0165-1684(01)00211-0

In recent years, fast transforms of interest are the fractional DFT [1], split-radix algorithms, vector-radix, mixed-radix algorithms [2,4–6], multi-dimension FFT algorithms, parallel FFT [10], efficient FFT software package [7] and high-performance FFT processors [8].

In this paper, we are interested in implementation of FFTs. We propose a new extendible look-up table of the twiddle factors for implementation of FFT in Section 2. Some efficient FFT algorithms using this look-up table are discussed in Section 3. In Section 4, we provide some numerical results.

2. Look-up tables of twiddle factors

For implementation of FFT algorithms, we create an array to store the twiddle factors, and this array is called look-up table. Let the number of points in the data sequence be a power of 2; that is, $n = 2^m$, where m is an integer. It is clear that $W_n^{n/2} = -1$, $W_n^n = 1$ and $W_{kn}^{kj} = W_n^j$. So the size of the array only needs $n/2$ complex units.

The ordinal number j, k $(0 \leqslant j, k < n)$ can be expressed as m-bit binary form:

$$j = (j_{m-1} j_{m-2} \cdots j_1 j_0) = \sum_{r=0}^{m-1} j_r 2^r, \quad j_r = 0 \text{ or } 1, \tag{2}$$

$$k = (k_{m-1} k_{m-2} \cdots k_1 k_0) = \sum_{r=0}^{m-1} k_r 2^r, \quad k_r = 0 \text{ or } 1. \tag{3}$$

Then the twiddle factors can be written as follows:

$$W_n^{jk} = \prod_{l=1}^{m} W_n^{j_{l-1} 2^{l-1} \sum_{r=0}^{m-l} k_r 2^r}. \tag{4}$$

Besides of the *natural order* look-up table

$$\boldsymbol{U}_n \overset{\text{def}}{=} (W_n^0, W_n^1, \ldots, W_n^{n/2-1}), \tag{5}$$

we recommend the following *bit-reversed order* (BRO) look-up table:

$$\boldsymbol{V}_n \overset{\text{def}}{=} (W_n^{\langle 0 \rangle_{m-1}}, W_n^{\langle 1 \rangle_{m-1}}, \ldots, W_n^{\langle n/2-1 \rangle_{m-1}}), \tag{6}$$

where

$\langle j \rangle_{m-1} = (j_0 j_1 \cdots j_{m-3} j_{m-2})$ if $j = (j_{m-2} j_{m-3} \cdots j_1 j_0)$.

Theorem 1. *For* $n = 2, 4, 8, 16, \ldots$, *we have the following recursion formula*:

$$\begin{aligned} &\boldsymbol{V}_2 = (1), \\ &\boldsymbol{V}_{2n} = (\boldsymbol{V}_n, W_{2n} \times \boldsymbol{V}_n) = (1, W_{2n}) \otimes \boldsymbol{V}_n. \end{aligned} \tag{7}$$

Proof. For $0 \leqslant j < n/2$, $j = (0 j_{m-2} j_{m-3} \cdots j_1 j_0)$; we have

$$\langle j \rangle_m = (j_0 j_1 \cdots j_{m-3} j_{m-2} 0) = 2\langle j \rangle_{m-1}$$

and

$$\langle j + n/2 \rangle_m = (j_0 j_1 \cdots j_{m-3} j_{m-2} 1) = 2\langle j \rangle_{m-1} + 1.$$

Consider the components of $\boldsymbol{V}_{2n}$, we can get

$$v_{2n}(j) = W_{2n}^{\langle j \rangle_m} = W_{2n}^{2\langle j \rangle_{m-1}} = W_n^{\langle j \rangle_{m-1}} = v_n(j)$$

and

$$\begin{aligned} v_{2n}(j + n/2) &= W_{2n}^{\langle j+n/2 \rangle_m} = W_{2n}^{2\langle j \rangle_{m-1}+1} \\ &= W_{2n} v_n(j). \qquad \square \end{aligned}$$

The meaning of Theorem 1 is that the BRO twiddle factors look-up table is extendible. On the other hand, we can compute all n-FFT $(n \leqslant N)$ if $\boldsymbol{V}_N$ is ready. We can get the BRO table using clever schemes (see [9]). Note, the first 4 components of $\boldsymbol{V}_n$ are $1, -i$, $(1-i)/\sqrt{2}, -(1+i)/\sqrt{2}$. It is trivial for a complex number multiplies $v(i)$ $(i = 0, 1, 2, 3)$.

3. FFT algorithms based on BRO twiddle factors look-up table

With (2)–(4), Eq. (1) can be rewritten in the following form:

$$\begin{aligned} &X(j_{m-1} \cdots j_0) \\ &\quad = \sum_{k_0=0}^{1} \cdots \sum_{k_{m-1}=0}^{1} x(k_{m-1} \cdots k_0) W_n^{j_0 \sum_{r=0}^{m-1} k_r 2^r} \cdots \\ &\qquad W_n^{j_{m-1} 2^{m-1} k_0}. \end{aligned} \tag{8}$$

Eq. (8) can be computed from the inside to the outside, and we get the following well-known FFT algorithm.

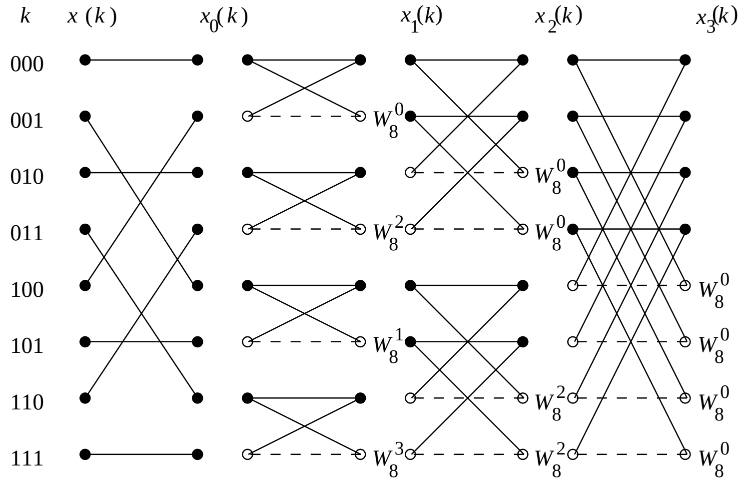

Fig. 1. Flow diagram for Algorithm 1.

Algorithm 1. Let $x_0(k_0k_1\cdots k_{m-2}k_{m-1})=x(k_{m-1}k_{m-2}\cdots k_1k_0)$, for $l=1,2,\ldots,m$, do

$$x_l(k_0\cdots k_{m-l-1}j_{l-1}\cdots j_0) = \sum_{k_{m-l}=0}^{1} x_{l-1}(k_0\cdots k_{m-l}j_{l-2}\cdots j_0) \times W_n^{j_{l-1}2^{l-1}\sum_{r=0}^{m-l} k_r 2^r},$$

then $X(j_{m-1}j_{m-2}\cdots j_1j_0)=x_m(j_{m-1}j_{m-2}\cdots j_1j_0)$.

If $n=8$, Algorithm 1 becomes

Step 1: (*Data input in bit-reversed order*) $x_0(k_0k_1k_2)=x(k_2k_1k_0)$.

Step 2: (*Compute*)

$$x_1(k_0k_1\mathbf{j}_0)=\sum_{k_2=0}^{1} x_0(k_0k_1\mathbf{k}_2)W_8^{j_0(4k_2+2k_1+k_0)},$$

$$x_2(k_0\mathbf{j}_1j_0)=\sum_{k_1=0}^{1} x_1(k_0\mathbf{k}_1j_0)W_8^{2j_1(2k_1+k_0)},$$

$$x_3(\mathbf{j}_2j_1j_0)=\sum_{k_0=0}^{1} x_2(\mathbf{k}_0j_1j_0)W_8^{4j_2k_0}.$$

Step 3: (*Data output*)

$X(j_2j_1j_0)=x_3(j_2j_1j_0)$.

The flow diagram of Algorithm 1 for $n=8$ is shown in Fig. 1.

It is easy to count that the computational complexity of Algorithm 1 is $5n\log_2 n-10n+16$.

We merge all three steps in Algorithm 1 to one step, that is, x_{l+2} can be expressed with x_{l-1}. Then we get the following radix-8 based Algorithm 2.

Algorithm 2. Let $x_0(k_0k_1\cdots k_{m-2}k_{m-1})=x(k_{m-1}k_{m-2}\cdots k_1k_0)$, for $l=1,4,7,10$, until $l\leqslant m-2$, do

$$x_{l+2}(k_0\cdots k_{m-l-3}j_{l+1}\cdots j_0) = \sum_{k_{m-l-2}=0}^{1}\sum_{k_{m-l-1}=0}^{1}\sum_{k_{m-l}=0}^{1} P_l \times x_{l-1}(k_0\cdots k_{m-l}j_{l-2}\cdots j_0),$$

where

$$P_l=(-1)^{j_{l-1}k_{m-l}+j_lk_{m-l-1}+j_{l+1}k_{m-l-2}} \times(-i)^{j_{l-1}k_{m-l-1}+j_lk_{m-l-2}}W_8^{j_{l-1}k_{m-l-2}} \times v^{j_{l+1}}(K_l)v^{j_l}(2K_l)v^{j_{l-1}}(4K_l)$$

and $K_l=(k_0\cdots k_{m-l-3})$.

If $m\equiv 1 \pmod 3$ then

$$x_m(j_{m-1}j_{m-2}\cdots j_1j_0) = \sum_{k_0=0}^{1} x_{m-1}(k_0j_{m-2}\cdots j_1j_0)(-1)^{k_0j_{m-1}}.$$

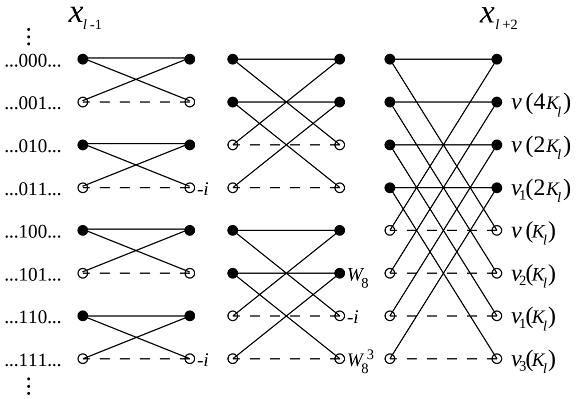

Fig. 2. Flow diagram for Algorithm 2 (part K_l in step l).

If $m \equiv 2 \pmod 3$ then

$$x_m(j_{m-1}j_{m-2}\cdots j_1j_0) = \sum_{k_0=0}^{1}\sum_{k_1=0}^{1} x_{m-2}(k_0k_1j_{m-3}\cdots j_1j_0) \times(-1)^{j_{m-1}k_0+j_{m-2}k_1}(-i)^{j_{m-2}k_0}.$$

The result is

$$X(j_{m-1}j_{m-2}\cdots j_1j_0) = x_m(j_{m-1}j_{m-2}\cdots j_1j_0).$$

Fig. 2 shows the flow diagram of the part K_l in step l of Algorithm 2. $l = 1,4,7,10,\ldots$, until $l \leqslant m-2$, and $K_l = 0,1,\ldots,n/2^{l+2}-1$.

For performance of Algorithm 2, we employ other three arrays $\boldsymbol{V}^{(1)}$, $\boldsymbol{V}^{(2)}$ and $\boldsymbol{V}^{(3)}$ to store $v_1(k) \stackrel{\text{def}}{=} v(k)v(2k)$, $v_2(k) \stackrel{\text{def}}{=} v(k)v(4k)$ and $v_3(k) \stackrel{\text{def}}{=} v(k)v(2k)v(4k)$, respectively. $\boldsymbol{V}$ (of size $n/2$), $\boldsymbol{V}^{(1)}$ (of size $n/4$), $\boldsymbol{V}^{(2)}$ (of size $n/8$) and $\boldsymbol{V}^{(3)}$ (of size $n/8$) are all independent of n. Therefore, we need n complex units to store those twiddle factors. That is, in our FFT scheme we need $n/2$ extra complex units to store $\boldsymbol{V}^{(1)}$, $\boldsymbol{V}^{(2)}$ and $\boldsymbol{V}^{(3)}$.

Theorem 2. *The number of required arithmetic operations of the Algorithm 2 with $n = 2^m$ is*

$$A(n) = \begin{cases} \frac{98}{24}nm - \frac{25}{4}n + 8, & m \equiv 0 \pmod 3, \\ \frac{98}{24}n(m-1) - \frac{7}{4}n + 8, & m \equiv 1 \pmod 3, \\ \frac{98}{24}n(m-2) + 2n + 8, & m \equiv 2 \pmod 3). \end{cases}$$

Proof. (i) For $m \equiv 0 \pmod 3$, there are $m/3$ steps and $n/8$ 8-component groups in each step. In each group there are not more than $2\times3\times8+2\times2+7\times2 = 66$ real additions and $2\times2+7\times4 = 32$ real multiplications (see Fig. 2). So, the computational complexity of Algorithm 2 is not more than

$$T = (66+32)\times\frac{n}{8}\times\frac{m}{3} = \frac{98}{24}nm.$$

Consider those "trivial" multiplications which are involved the part $K_l = 0,1,2,3$ in each step, and the following number of the arithmetic operations should be removed from all steps:

$$K_l = 0:\ 7\times6(1+8+8^2+\cdots+8^{m/3-1}) = 6n-6.$$

$$K_l = 1,2,3:\quad (10+2+2)(1+8+8^2+\cdots+8^{m/3-2}) = n/4-2.$$

So

$$A(n) = \tfrac{98}{24}nm - \tfrac{25}{4}n + 8.$$

(ii) For $m \equiv 1 \pmod 3$,

$$T = 98\times\frac{n}{8}\times\frac{m-1}{3} + 2n = \frac{98}{24}n(m-1)+2n,$$

$$K_l = 0,1:\quad (7\times6+10)(1+8+8^2+\cdots+8^{(m-1)/3-1}) = \frac{52}{7}\left(\frac{n}{2}-1\right),$$

$$K_l = 2,3:\quad (2+2)(1+8+8^2+\cdots+8^{(m-1)/3-2}) = \frac{4}{7}\left(\frac{n}{16}-1\right).$$

So

$$A(n) = \frac{98}{24}n(m-1) - \frac{7}{4}n + 8.$$

(iii) For $m \equiv 2 \pmod 3$,

$$T = 98\times\frac{n}{8}\times\frac{m-2}{3} + 4n = \frac{98}{24}n(m-2)+4n,$$

$$K_l = 0,1,2,3:\quad (7\times6+10+2+2)\times(1+8+8^2+\cdots+8^{(m-2)/3-1}) = 8\left(\frac{n}{4}-1\right) = 2n-8.$$

So

$$A(n) = \tfrac{98}{24}n(m-2) + 2n + 8. \qquad \square$$

Table 1
Computational complexity and implementation complexity of Algorithm 2

n	a	b	c	d	n	a	b	c	d
8	56	32	2.33	3.67	4096	175 112	32 125	3.56	4.22
16	176	83	2.75	4.05	8192	387 080	74 546	3.63	4.33
32	464	152	2.90	3.85	16 384	835 592	149 135	3.64	4.29
64	1176	323	3.06	3.90	32 768	1 802 248	298 346	3.67	4.27
128	2920	824	3.26	4.18	65 536	3 899 400	678 687	3.72	4.37
256	6792	1653	3.32	4.12	131 072	8 290 312	1 357 436	3.72	4.33
512	15 624	3344	3.39	4.12	262 144	17 629 192	2 714 967	3.74	4.31
1024	35 848	8005	3.50	4.28	524 288	37 617 672	6 085 388	3.78	4.39
2048	79 368	16 034	3.52	4.23	1 048 576	79 167 496	12 170 857	3.78	4.36

Table 2
The comparison of elapsed times between Algorithm 2 and FFTW

n	t_1	t_2	n	t_1	t_2	n	t_1	t_2
8	34×10^{-6}	219×10^{-6}	512	0.001794	0.003909	32 768	0.28	0.32
16	72×10^{-6}	312×10^{-6}	1024	0.004224	0.007945	65 536	0.63	0.67
32	128×10^{-6}	468×10^{-6}	2048	0.010288	0.015984	131 072	1.34	1.40
64	215×10^{-6}	735×10^{-6}	4096	0.023614	0.031618	262 144	2.96	3.03
128	395×10^{-6}	1159×10^{-6}	8192	0.054832	0.069192	524 288	6.31	7.99
256	809×10^{-6}	2201×10^{-6}	16 384	0.117800	0.152433			

From Theorem 2, we have $A(n) < 4n \log_2 n$ for $n < 2^{75}$. Theorem 2 is also verified by a program which can count the number of the arithmetic operations of Algorithm 2 and the results are listed in Table 1column a.

Note, the computational complexity of Algorithm 2 is close to $4nm - 6n + 8$ which is the computational complexity of the well-known split-radix FFT algorithm [4]. Although the computational complexity of Algorithm 2 is more than that of split-radix FFT algorithm, our Algorithm 2 has a more regular, more symmetric structure. And with the flexible BRO table of the twiddle factors, Algorithm 2 can be coded easily and performed efficiently.

In Table 1, a represents the number of arithmetic operations of Algorithm 2 with different n, and b stands all the auxiliary operations of C subroutine of Algorithm 2. a and b are counted by a C program. c and d stand for $a/(n \log_2 n)$ and $(a+b)/(n \log_2 n)$ respectively. The results show that the number of all operations of our FFT subroutine is less than $5n \log_2 n$ (see Table 1 column d).

4. Experimental results

Our numerical experiments are performed on a PC (Pentium, CPU at 166 Hz, RAM 32 MB) under Linux operation system (Slackware Linux V3.6). The codes are written with C language. We compare the elapsed times between Algorithm 2 and the FFTW (Fastest Fourier Transforms in the West) software package (see (http://www.fftw.org)).

In Table 2, t_1 and t_2 represent the elapsed time (in seconds) of our Algorithm 2 and FFTW software package, respectively. The times in Table 2 include the time of implementation of a forward Fourier transform computation and a backward Fourier transform computation.

We also perform FFT computations for 19 sequences with size from $2^1, 2^2, \ldots, 2^{19}$, and we get $t_{\mathrm{Algo2}} = 11.79(\mathrm{s})$, $t_{\mathrm{FFTW}} = 13.87(\mathrm{s})$.

5. Conclusion

In this paper, we have introduced an extendible look-up table of twiddle factors for implementation of

FFTs and the table is independent of the length of sequences. We also have proposed an efficient FFT algorithm with this table. Experimental comparisons have been done between our algorithm and FFTW software package. The results indicate that our FFT scheme is effective.

Acknowledgements

The authors would like to thank the reviewers for their helpful suggestions and valuable comments. This work is supported by the National 863 Project (Grant 863-306-ZD-01-8), and National Science Foundation (Grant 60073044).

References

[1] L.B. Almeida, The fractional Fourier transform and time frequency representations, IEEE Trans. Signal Process. 42 (1994) 3084–3090.
[2] I.C. Chan, K.L. Ho, Split vector-radix fast Fourier transform, IEEE Trans. Signal Process. 40 (1992) 2029–2040.
[3] J.W. Cooley, J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Comput. 19 (2) (1965) 297–301.
[4] P. Duhamel, H. Hollmann, Split-radix FFT algorithm, Electron. Lett. 20 (1984) 14–16.
[5] P. Duhamel, M. Vetterli, Fast Fourier transforms: a tutorial review and state of the art, Signal Processing 19 (1990) 259–299.
[6] A.M. Grigoryan, S.S. Agaian, Split manageable efficient algorithm for Fourier and Hadamard transforms, IEEE Trans. Signal Process. 48 (1) (2000) 172–183.
[7] (http://www.fftw.org).
[8] Y. Ma, An effective memory addressing scheme for FFT processors, IEEE Trans. Signal Process. 47 (3) (1999) 907–911.
[9] J.M. Rius, R.D. Porrata-Dòria, New FFT bit-reversal algorithm, IEEE Trans. Signal Process. 43 (4) (1995) 991–994.
[10] M. Silvia, R. Giancarlo, S. Gaetano, A parallel fast Fourier transform, Mod. Phys. C 10 (1999) 781–805.