A RADIX-16 FFT ALGORITHM SUITABLE FOR MULTIPLY-ADD INSTRUCTION BASED ON GOEDECKER METHOD

Daisuke Takahashi

Institute of Information Sciences and Electronics, University of Tsukuba
1-1-1 Tennodai, Tsukuba-shi, Ibaraki 305-8573, Japan
daisuke@is.tsukuba.ac.jp

ABSTRACT

A radix-16 fast Fourier transform (FFT) algorithm suitable for multiply-add instruction is proposed. The proposed radix-16 FFT algorithm requires fewer floating-point instructions than the conventional radix-16 FFT algorithm on processors that have a multiply-add instruction. Moreover, this algorithm has the advantage of fewer loads and stores than either the radix-2, 4 and 8 FFT algorithms or the split-radix FFT algorithm. We use Goedecker's method to obtain an algorithm for computing radix-16 FFT with fewer floating-point instructions than the conventional radix-16 FFT algorithm. The number of floating-point instructions for the proposed radix-16 FFT algorithm is compared with those of conventional power-of-two FFT algorithms on processors with multiply-add instruction.

1. INTRODUCTION

For computing an $N = 2^m$-point FFT, radix-2, 4, 8 and 16 FFT algorithms and split-radix FFT algorithms have been proposed [1, 2, 3, 4, 5].

Until several years ago, floating-point addition was faster than floating-point multiplication on most processors. For this reason, FFT algorithms that reduced real multiplications, e.g., the Winograd Fourier transform algorithm (WFTA) [6] and the prime factor FFT algorithm (PFA) [7, 8], have been intensively studied. These algorithms show an advantage over processors that require more time for multiplication than addition. Today, floating-point multiplication is as fast as floating-point addition on the latest processors. Moreover, many processors have a multiply-add instruction.

As for related works, Linzer and Feig [9] have shown radix-2, 4 and 8 FFT algorithms and split-radix FFT algorithm for fused multiply-add architectures. These FFT algorithms are based on the Cooley-Tukey FFT algorithm [1] and the split-radix FFT algorithm [3, 4]. On the other hand, radix-2, 3, 4 and 5 FFT algorithms on computers with overlapping multiply-add instructions have been proposed by Goedecker [10] and Karner et al. [11].

The higher radices are more efficient in terms of both memory and floating-point operations. A high ratio of floating-point instructions to memory operations is particularly important in a cache-based processor. In view of the high ratio of floating-point instructions to memory operations, the radix-16 FFT is more advantageous than the radix-2, 4 and 8 FFTs. Thus the FFTW [12], is known as one of the fastest FFT libraries for many processors, contains a radix-16 FFT routine.

This work was supported by the Grant-in-Aid for Young Scientists (B) (No. 14780185) of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

Although higher radix FFTs require more floating-point registers to hold intermediate results, some processors have sufficient floating-point registers (e.g., Intel Itanium processor has 128 floating-point registers [13]).

However, efficient implementations of a radix-16 FFT algorithm suitable for multiply-add instruction have not yet been presented.

In this paper, we propose a radix-16 FFT algorithm suitable for multiply-add instruction based on Goedecker method.

Throughout this paper, we use a multiply-add instruction, which computes $d = \pm a \pm bc$, where a, b, c and d are floating-point registers. Also, we assume that an addition, a multiplication, or a multiply-add each requires one machine cycle on processors that have a multiply-add instruction. We will call any of these computations a floating-point instruction, and assign a unit cost to each.

2. A RADIX-16 FFT ALGORITHM SUITABLE FOR MULTIPLY-ADD INSTRUCTION

The DFT of N points is given by

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}, \quad k = 0, \cdots, N-1 \tag{1}$$

where $W_N = \exp(-j2\pi/N)$, X_k and x_n are sequences of complex numbers.

An FFT kernel [10, 14] calculates the innermost part in a transformation, which has the form [10]

$$Z_{out}(k) = \sum_{n=0}^{P-1} Z_{in}(n)\Omega^n W_P^{nk} \tag{2}$$

for $k = 0, 1, \cdots, P-1$. The radix of the kernel is given by the prime factor P which is 16 in this paper. Ω^n is called the twiddle factor and $W_P = \exp(-j2\pi/P)$.

In the radix-P FFT kernel, an input data $Z_{in}(n)$ multiplied by the twiddle factor Ω^n is performed with "short DFT" [15].

The adaptability of the conventional radix-16 FFT algorithm on processors that have a multiply-add instruction is here discussed.

The conventional radix-16 FFT (decimation-in-time) is split into the first step and the remaining part. The first step is the complex multiplication of $Z_{in}(n) \times \Omega^n$ $(n = 1, 2, \ldots, 15)$. Then 15 complex multiplications are necessary. We assume that a complex multiplication in the $(1, j)$ plane is done with four real multiplications and two real additions. In the first step, since the ratio of

Table 1. Number of Floating-Point Instructions for FFT Algorithms of N Points with Multiply-Add Instruction

Algorithm	Floating-point instructions
Linzer and Feig radix-4 [9]	$(11/4)N\log_2 N - (13/6)N + (8/3)$
Linzer and Feig radix-8 [9]	$(11/4)N\log_2 N - (57/28)N + (16/7)$
Linzer and Feig split-radix [9]	$(8/3)N\log_2 N - (16/9)N + 2 - (2/9)(-1)^{\log_2 N}$
Conventional radix-16	$(55/16)N\log_2 N - (241/60)N + (64/15)$
Proposed radix-16	$(87/32)N\log_2 N - (241/120)N + (32/15)$

Table 2. Number of Floating-Point Instructions for FFT Algorithms of N Points with Multiply-Add Instruction

N	Linzer and Feig radix-4 [9]	Linzer and Feig radix-8 [9]	Linzer and Feig split-radix [9]	Conventional radix-16	Proposed radix-16
8		52	52		
16	144		144	160	144
32			372		
64	920	928	912		
128			2164		
256	5080		5008	6016	5056
512		11632	11380		
1024	25944		25488		
2048			56436		
4096	126296	126832	123792	152512	125408

multiplications to additions is two to one, the addition unit cannot be exploited on processors with multiply-add instruction. Then a 16-point DFT is performed in the remaining part. Since the conventional radix-16 FFT has 24 real multiplications and 144 real additions in the remaining part, the multiply unit also cannot be exploited. As a result, the conventional radix-16 FFT algorithm has 220 floating-point instructions on processors with multiply-add instruction.

We conclude that the conventional radix-16 FFT algorithm is therefore not suitable for the multiply-add instruction.

As we mentioned in the above, the multiply-add unit cannot be exploited in the conventional radix-16 FFT algorithm. We will make full use of the multiply-add unit to transform the conventional radix-16 FFT.

Goedecker's method [10] consists of repeated transformations of the expression:

$$ax + by \rightarrow a(x + (b/a)y) \tag{3}$$

where $a \neq 0$.

Applying repeated transformations of (3) to the conventional radix-16 FFT, a radix-16 FFT algorithm suitable for multiply-add instruction is obtained.

The proposed radix-16 FFT algorithm suitable for multiply-add instruction is shown in Fig. 1. Here, the real part of the array Z_{in} is denoted by zinr, the imaginary part by zini, and correspondingly for Z_{out}. The real part and imaginary part of the twiddle factor Ω^n are cr_n and ci_n, respectively.

In the proposed radix-16 FFT algorithm, a table for twiddle factors of ci_1–ci_{15}, cr_{31}–cr_{157}, cos_{81}–cos_{8381} and cr_{181}–cr_{282} is needed. Since these values can be computed in advance, the overhead of making the table is negligible.

The proposed radix-16 FFT algorithm has only 174 floating-point instructions on processors with multiply-add instruction. We can see that the multiply-add unit can be exploited in the proposed radix-16 FFT algorithm from Fig. 1.

3. EVALUATION

In order to evaluate the effectiveness of power-of-two FFT algorithms, we compare the number of floating-point instructions, loads, and stores.

The number of floating-point instructions for various FFT algorithms of N points with multiply-add instruction is shown in Tables 1 and 2. The proposed radix-16 FFT algorithm asymptotically saves about 21% of the floating-point instructions over the conventional radix-16 FFT on processors that have a multiply-add instruction.

In comparison with the Linzer and Feig radix-4 and 8 FFT algorithms [9], the proposed radix-16 FFT algorithm asymptotically saves about 1% of the floating-point instructions. On the other hand, the proposed radix-16 FFT algorithm asymptotically increases about 2% of the floating-point instructions over the Linzer and Feig split-radix FFT algorithm [9].

The number of loads, stores, floating-point instructions used for various FFT butterflies is given in Table 3. In calculating the number of loads and the number of stores, we assume that enough registers are available to perform an entire butterfly in the registers without using any intermediate stores or loads.

In Table 4, the asymptotic number of loads, stores, and floating-point instructions used by each algorithm is given. The proposed radix-16 FFT algorithm requires fewer loads and stores than the Linzer and Feig radix-4 and 8 FFT algorithms or the Linzer and Feig split-radix FFT algorithm. In particular, in comparison with the Linzer and Feig radix-8 FFT algorithm, the proposed radix-16 FFT algorithm asymptotically saves 25% of the loads and stores.

Table 3. Number of Loads, Stores, and Floating-Point Instructions for General Butterflies Used in FFT Algorithms with Multiply-Add Instruction. The Number of Loads Does Not Include Loading All of the Constants

Algorithm	Loads	Stores	Floating-point instructions
Linzer and Feig radix-4 [9]	8	8	22
Linzer and Feig radix-8 [9]	16	16	66
Linzer and Feig split-radix [9]	8	8	16
Conventional radix-16	32	32	220
Proposed radix-16	32	32	174

Table 4. Number of Loads, Stores, and Floating-Point Instructions Divided by $N \log_2 N$ Used by FFT Algorithms to Compute an N Point DFT. Lower Order Terms Have Been Omitted

Algorithm	Loads	Stores	Floating-point instructions
Linzer and Feig radix-4 [9]	1	1	$11/4$
Linzer and Feig radix-8 [9]	$2/3$	$2/3$	$11/4$
Linzer and Feig split-radix [9]	$4/3$	$4/3$	$8/3$
Conventional radix-16	$1/2$	$1/2$	$55/16$
Proposed radix-16	$1/2$	$1/2$	$87/32$

4. CONCLUSION

A radix-16 FFT algorithm suitable for multiply-add instruction has been presented. We reduced the number of floating-point instructions necessary for a radix-16 FFT algorithm by maximizing the use of multiply-add instructions.

The proposed radix-16 FFT algorithm requires fewer floating-point instructions than the conventional radix-16 FFT algorithm on processors that have a multiply-add instruction.

If the FFTs are being computed on a machine that has enough registers to perform an entire radix-16 FFT algorithm, those FFTs will use fewer loads and stores than the radix-2, 4 and 8 FFT algorithms or the split-radix FFT algorithm.

5. REFERENCES

[1] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.*, vol. 19, pp. 297–301, Apr. 1965.

[2] G. D. Bergland, “A fast Fourier transform algorithm using base 8 iterations,” *Math. Comput.*, vol. 22, pp. 275–279, Apr. 1968.

[3] P. Duhamel and H. Hollmann, “Split radix FFT algorithm,” *Electron. Lett.*, vol. 20, pp. 14–16, Jan. 1984.

[4] H. V. Sorensen, M. T. Heideman, and C. S. Burrus, “On computing the split-radix FFT,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 152–156, Feb. 1986.

[5] D. Takahashi, “An extended split-radix FFT algorithm,” *IEEE Signal Processing Lett.*, vol. 8, pp. 145–147, May 2001.

[6] S. Winograd, “On computing the discrete Fourier transform,” *Math. Comput.*, vol. 32, pp. 175–199, Jan. 1978.

[7] D. P. Kolba and T. W. Parks, “A prime factor FFT algorithm using high-speed convolution,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-25, pp. 281–294, Aug. 1977.

[8] C. S. Burrus and P. W. Eschenbacher, “An in-place, in-order prime factor FFT algorithm,” *IEEE Trans. Audio Electroacoust.*, vol. AU-29, pp. 806–817, Aug. 1981.

[9] E. Linzer and E. Feig, “Implementation of efficient FFT algorithms on fused multiply-add architectures,” *IEEE Trans. Signal Processing*, vol. 41, pp. 93–107, Jan. 1993.

[10] S. Goedecker, “Fast radix $2, 3, 4$, and 5 kernels for fast Fourier transformations on computers with overlapping multiply-add instructions,” *SIAM J. Sci. Comput.*, vol. 18, pp. 1605–1611, Nov. 1997.

[11] H. Karner, M. Auer, and C. W. Ueberhuber, “Multiply-add optimized FFT kernels,” *Math. Models. Methods Appl. Sci.*, vol. 11, pp. 105–117, Feb. 2001.

[12] M. Frigo and S. G. Johnson, “FFTW: An adaptive software architecture for the FFT,” in *Proc. 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP98)*, Apr. 1998, pp. 1381–1384.

[13] Intel Corporation, *Intel Itanium Architecture Software Developer’s Manual Volume 1: Application Architecture Revision 2.0 (245317-003)*, Dec. 2001.

[14] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA, 1992.

[15] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, New York, second corrected and updated edition, 1982.

```
/* twiddle factors can be
   computed in advance */
$ci_1 = ci_1/cr_1$
$ci_2 = ci_2/cr_2$
$ci_3 = ci_3/cr_3$
$ci_4 = ci_4/cr_4$
$ci_5 = ci_5/cr_5$
$ci_6 = ci_6/cr_6$
$ci_7 = ci_7/cr_7$
$ci_8 = ci_8/cr_8$
$ci_9 = ci_9/cr_9$
$ci_{10} = ci_{10}/cr_{10}$
$ci_{11} = ci_{11}/cr_{11}$
$ci_{12} = ci_{12}/cr_{12}$
$ci_{13} = ci_{13}/cr_{13}$
$ci_{14} = ci_{14}/cr_{14}$
$ci_{15} = ci_{15}/cr_{15}$
$cr_{31} = cr_3/cr_1$
$cr_{51} = cr_5/cr_1$
$cr_{62} = cr_6/cr_2$
$cr_{73} = cr_7/cr_3$
$cr_{91} = cr_9/cr_1$
$cr_{102} = cr_{10}/cr_2$
$cr_{113} = cr_{11}/cr_3$
$cr_{124} = cr_{12}/cr_4$
$cr_{135} = cr_{13}/cr_5$
$cr_{146} = cr_{14}/cr_6$
$cr_{157} = cr_{15}/cr_7$
$cos_{81} = \cos(\pi/8)$
$cos_{82} = \cos(2\pi/8) = 1/\sqrt{2}$
$cos_{83} = \cos(3\pi/8)$
$cos_{8381} = cos_{83}/cos_{81}$
$cr_{181} = cr_1 * cos_{81}$
$cr_{182} = cr_1 * cos_{82}$
$cr_{282} = cr_2 * cos_{82}$
/* 15 complex multiplications
   by twiddle factors */
$u0 = zinr(0)$
$v0 = zini(0)$
$r = zinr(1)$
$s = zini(1)$
$u1 = r - s * ci_1$
$v1 = r * ci_1 + s$
$r = zinr(2)$
$s = zini(2)$
$u2 = r - s * ci_2$
$v2 = r * ci_2 + s$
$r = zinr(3)$
$s = zini(3)$
$u3 = r - s * ci_3$
$v3 = r * ci_3 + s$
$r = zinr(4)$
$s = zini(4)$
$u4 = r - s * ci_4$
$v4 = r * ci_4 + s$
$r = zinr(5)$
$s = zini(5)$
$u5 = r - s * ci_5$
$v5 = r * ci_5 + s$
$r = zinr(6)$
$s = zini(6)$
$u6 = r - s * ci_6$
$v6 = r * ci_6 + s$
$r = zinr(7)$
$s = zini(7)$
$u7 = r - s * ci_7$
$v7 = r * ci_7 + s$
$r = zinr(8)$
$s = zini(8)$
$u8 = r - s * ci_8$
$v8 = r * ci_8 + s$
$r = zinr(9)$
$s = zini(9)$
$u9 = r - s * ci_9$
$v9 = r * ci_9 + s$
$r = zinr(10)$
$s = zini(10)$
$u10 = r - s * ci_{10}$
$v10 = r * ci_{10} + s$
$r = zinr(11)$
$s = zini(11)$
$u11 = r - s * ci_{11}$
$v11 = r * ci_{11} + s$
$r = zinr(12)$
$s = zini(12)$
$u12 = r - s * ci_{12}$
$v12 = r * ci_{12} + s$
$r = zinr(13)$
$s = zini(13)$
$u13 = r - s * ci_{13}$
$v13 = r * ci_{13} + s$
$r = zinr(14)$
$s = zini(14)$
$u14 = r - s * ci_{14}$
$v14 = r * ci_{14} + s$
$r = zinr(15)$
$s = zini(15)$
$u15 = r - s * ci_{15}$
$v15 = r * ci_{15} + s$
/* 16-point DFT */
$r0 = u0 + u8 * cr_8$
$s0 = v0 + v8 * cr_8$
$r1 = u0 - u8 * cr_8$
$s1 = v0 - v8 * cr_8$
$r2 = u4 + u12 * cr_{124}$
$s2 = v4 + v12 * cr_{124}$
$r3 = v4 - v12 * cr_{124}$
$s3 = u12 * cr_{124} - u4$
$r4 = u1 + u9 * cr_{91}$
$s4 = v1 + v9 * cr_{91}$
$r5 = u1 - u9 * cr_{91}$
$s5 = v1 - v9 * cr_{91}$
$r6 = u5 + u13 * cr_{135}$
$s6 = v5 + v13 * cr_{135}$
$r7 = v5 - v13 * cr_{135}$
$s7 = u13 * cr_{135} - u5$
$r8 = u2 + u10 * cr_{102}$
$s8 = v2 + v10 * cr_{102}$
$r9 = u2 - u10 * cr_{102}$
$s9 = v2 - v10 * cr_{102}$
$r10 = u6 + u14 * cr_{146}$
$s10 = v6 + v14 * cr_{146}$
$r11 = v6 - v14 * cr_{146}$
$s11 = u14 * cr_{146} - u6$
$r12 = u3 + u11 * cr_{113}$
$s12 = v3 + v11 * cr_{113}$
$r13 = u3 - u11 * cr_{113}$
$s13 = v3 - v11 * cr_{113}$
$r14 = u7 + u15 * cr_{157}$
$s14 = v7 + v15 * cr_{157}$
$r15 = v7 - v15 * cr_{157}$
$s15 = u15 * cr_{157} - u7$
$u0 = r0 + r2 * cr_4$
$v0 = s0 + s2 * cr_4$
$u1 = r1 + r3 * cr_4$
$v1 = s1 + s3 * cr_4$
$u2 = r0 - r2 * cr_4$
$v2 = s0 - s2 * cr_4$
$u3 = r1 - r3 * cr_4$
$v3 = s1 - s3 * cr_4$
$u4 = r4 + r6 * cr_{51}$
$v4 = s4 + s6 * cr_{51}$
$r = r5 + r7 * cr_{51}$
$s = s5 + s7 * cr_{51}$
$u5 = r + s * cos_{8381}$
$v5 = -r * cos_{8381} + s$
$r = r4 - r6 * cr_{51}$
$s = s4 - s6 * cr_{51}$
$u6 = r + s$
$v6 = s - r$
$r = r5 - r7 * cr_{51}$
$s = s5 - s7 * cr_{51}$
$u7 = r * cos_{8381} + s$
$v7 = -r + s * cos_{8381}$
$u8 = r8 + r10 * cr_{62}$
$v8 = s8 + s10 * cr_{62}$
$r = r9 + r11 * cr_{62}$
$s = s9 + s11 * cr_{62}$
$u9 = r + s$
$v9 = s - r$
$u10 = s8 - s10 * cr_{62}$
$v10 = r10 * cr_{62} - r8$
$r = r9 - r11 * cr_{62}$
$s = s9 - s11 * cr_{62}$
$u11 = -(r - s)$
$v11 = -(r + s)$
$u12 = r12 + r14 * cr_{73}$
$v12 = s12 + s14 * cr_{73}$
$r = r13 + r15 * cr_{73}$
$s = s13 + s15 * cr_{73}$
$u13 = r * cos_{8381} + s$
$v13 = -r + s * cos_{8381}$
$r = r12 - r14 * cr_{73}$
$s = s12 - s14 * cr_{73}$
$u14 = -(r - s)$
$v14 = -(r + s)$
$r = r13 - r15 * cr_{73}$
$s = s13 - s15 * cr_{73}$
$u15 = -r - s * cos_{8381}$
$v15 = r * cos_{8381} - s$
$r0 = u0 + u8 * cr_2$
$s0 = v0 + v8 * cr_2$
$r1 = u0 - u8 * cr_2$
$s1 = v0 - v8 * cr_2$
$r2 = u4 + u12 * cr_{31}$
$s2 = v4 + v12 * cr_{31}$
$r3 = v4 - v12 * cr_{31}$
$s3 = u12 * cr_{31} - u4$
$zoutr(0) = r0 + r2 * cr_1$
$zouti(0) = s0 + s2 * cr_1$
$zoutr(4) = r1 + r3 * cr_1$
$zouti(4) = s1 + s3 * cr_1$
$zoutr(8) = r0 - r2 * cr_1$
$zouti(8) = s0 - s2 * cr_1$
$zoutr(12) = r1 - r3 * cr_1$
$zouti(12) = s1 - s3 * cr_1$
$r0 = u1 + u9 * cr_{282}$
$s0 = v1 + v9 * cr_{282}$
$r1 = u1 - u9 * cr_{282}$
$s1 = v1 - v9 * cr_{282}$
$r2 = u5 + u13 * cr_{31}$
$s2 = v5 + v13 * cr_{31}$
$r3 = v5 - v13 * cr_{31}$
$s3 = u13 * cr_{31} - u5$
$zoutr(1) = r0 + r2 * cr_{181}$
$zouti(1) = s0 + s2 * cr_{181}$
$zoutr(5) = r1 + r3 * cr_{181}$
$zouti(5) = s1 + s3 * cr_{181}$
$zoutr(9) = r0 - r2 * cr_{181}$
$zouti(9) = s0 - s2 * cr_{181}$
$zoutr(13) = r1 - r3 * cr_{181}$
$zouti(13) = s1 - s3 * cr_{181}$
$r0 = u2 + u10 * cr_2$
$s0 = v2 + v10 * cr_2$
$r1 = u2 - u10 * cr_2$
$s1 = v2 - v10 * cr_2$
$r2 = u6 + u14 * cr_{31}$
$s2 = v6 + v14 * cr_{31}$
$r3 = v6 - v14 * cr_{31}$
$s3 = u14 * cr_{31} - u6$
$zoutr(2) = r0 + r2 * cr_{182}$
$zouti(2) = s0 + s2 * cr_{182}$
$zoutr(6) = r1 + r3 * cr_{182}$
$zouti(6) = s1 + s3 * cr_{182}$
$zoutr(10) = r0 - r2 * cr_{182}$
$zouti(10) = s0 - s2 * cr_{182}$
$zoutr(14) = r1 - r3 * cr_{182}$
$zouti(14) = s1 - s3 * cr_{182}$
$r0 = u3 + u11 * cr_{282}$
$s0 = v3 + v11 * cr_{282}$
$r1 = u3 - u11 * cr_{282}$
$s1 = v3 - v11 * cr_{282}$
$r2 = u7 + u15 * cr_{31}$
$s2 = v7 + v15 * cr_{31}$
$r3 = v7 - v15 * cr_{31}$
$s3 = u15 * cr_{31} - u7$
$zoutr(3) = r0 + r2 * cr_{181}$
$zouti(3) = s0 + s2 * cr_{181}$
$zoutr(7) = r1 + r3 * cr_{181}$
$zouti(7) = s1 + s3 * cr_{181}$
$zoutr(11) = r0 - r2 * cr_{181}$
$zouti(11) = s0 - s2 * cr_{181}$
$zoutr(15) = r1 - r3 * cr_{181}$
$zouti(15) = s1 - s3 * cr_{181}$
```

Fig. 1. Proposed radix-16 FFT algorithm suitable for multiply-add instruction.