積和演算命令に適した新しい8基底FFTカーネル

額　田　　　彰[†,††]　西　田　　　晃[†,††]　小　柳　義　夫[†]

積和演算命令に適した新しい 8 基底の FFT カーネルを提案する．この FFT カーネルは従来の積和演算命令と同じ積和演算命令数であるが，必要なひねり係数のテーブルのサイズが小さく，N 点 FFT を $3N/2$ 個の浮動小数テーブルで計算することができる．またカーネルの計算でロードする必要があるひねり係数の浮動小数も 14 と従来のものより少ない．Itanium2, Power4 で評価したところキャッシュミスが減り，より高い性能を出すことができた．

New Radix-8 FFT Kernel for Fused Multiply-Add Instructions

Akira Nukada ,[†,††] Akira Nishida [†,††] and Yoshio Oyanagi[†]

In this paper, we propose a new radix-8 FFT kernel for fused multiply-add instructions. Although this kernel requires the same number of multiply-add instructions as conventional radix-8 FFT kernels, it requires smaller size of twiddle factor table, that is, $3N/2$ floating point numbers for N-point FFT. Moreover, this kernel needs to load only 14 floating point numbers for twiddle factors to compute a kernel. In the experiments on Itanium2 and Power4, the number of cache miss was reduced, and higher performance was attained.

1. はじめに

現在離散フーリエ変換とその高速アルゴリズムである Fast Fourier Transform(FFT)[1) は，大規模な科学技術計算からマルチメディア関連の圧縮／伸長まで非常に多くの分野で用いられている．FFT のアルゴリズムが発見されてから多くの改良が研究されているが，近年プロセッサアーキテクチャが急激に進歩しており，それに適した新しい FFT カーネルが開発されてきている．

浮動小数演算器の構成は現在では Intel Pentium4, Sun UltraSPARC-III など加減算ユニットと乗算ユニットを 1 つずつ持ち，1cycle あたり 2 つの浮動小数演算が実行できるプロセッサが主流となっているが，Intel IA-64, MIPS R18000, IBM POWER4, PowerPC 970(G5) など積和演算ユニットを 2 つ持ち，1cycle あたり 4 つの浮動小数演算が実行できるプロセッサが増加する傾向にある．

積和演算命令 (Fused Multiply-Add Instruction) は乗算結果に加減算を行う複合命令であり，積和演算命令をサポートするプロセッサの場合乗算だけ，または加減算だけ行いたい時にも積和演算ユニットを使用する．このため，積和演算ユニットを効率よく使用するためにはなるべく乗算と加減算を組み合わせて積和演算命令として実行することが不可欠である．

積和演算命令数が最小となる 8 基底 FFT カーネルが既に研究されているが，本研究ではこれと同じ積和演算数でありながら必要なひねり係数のテーブルが小さく，ロードする必要があるひねり係数の数も少ないという利点を持つような新しい積和演算命令に適した 8 基底 FFT カーネルを提案する．

2. FFT カーネル

N 点離散フーリエ変換は以下のような式で定義される．

$$X(k) = \sum_{j=0}^{N-1} x(j)\omega_N^{jk}$$

ただし ω_N は 1 の N 次原始根であり，$\omega_N = \exp(-2\pi i/N), i = \sqrt{-1}$ であるとする． N が合成数で $N = N_1N_2$ と表わせる場合，$j = j_1 + j_2N_1, k = k_1 + k_2N_2$ とおくと定義式は次のように N_1 点のフーリエ変換,N_2 点フーリエ変換という小さいサイズのフーリエ変換とひねり係数 $\omega_N^{j_1k_1}$ の乗算に分解できる．

$$X(k_1 + k_2N_2) = \sum_{j_1=0}^{N_1} \left(\sum_{j_2=0}^{N_2} x(j_1 + j_2N_1)\omega_{N_2}^{j_2k_1} \right) \omega_N^{j_1k_1}\omega_{N_1}^{j_1k_2}$$

† 東京大学大学院情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, the University of Tokyo
†† 科学技術振興機構 CREST

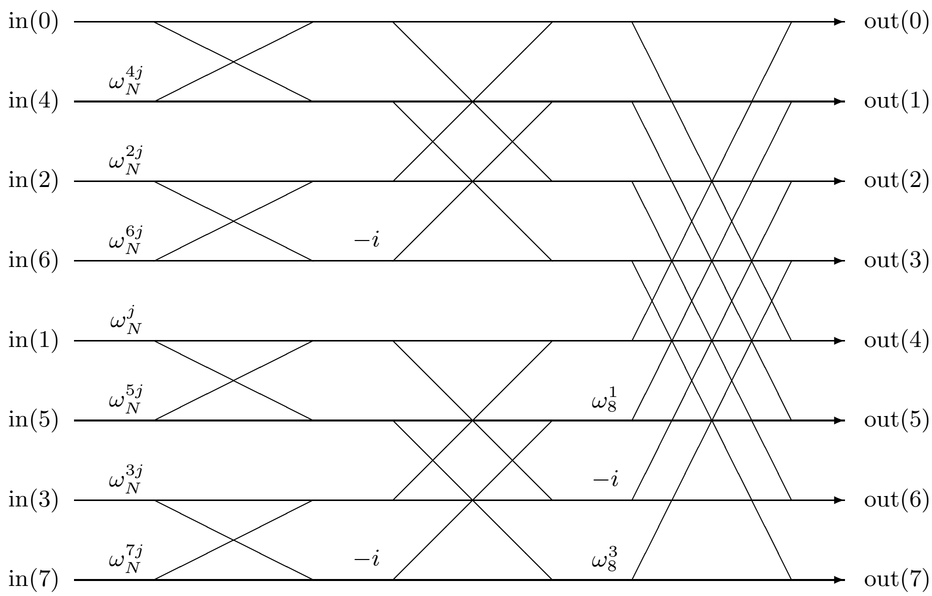

図 1　従来の 8 基底 FFT カーネル
Fig. 1　Conventional Radix-8 FFT Kernel

このような変形を繰り返してフーリエ変換をカーネルと呼ばれる小さいサイズのフーリエ変換に帰着させる．

FFT カーネル[2) は FFT の計算において最内側のループで行われる演算であり, p 基底の FFT カーネルは,

$$X_{t+1}(k) = \sum_{l=0}^{p-1} X_t(l)\omega_N^{jl}\omega_p^{kl}$$

と表わすことができる． 入力に対してひねり係数 $\omega_N^{jl}(0 \leq j < N/8)$ を掛けた後 p 点 FFT の計算が行われる．

2^m 点 FFT の計算では, 2,4,8 基底や Split-Radix[3) などの FFT カーネルにより構成することができる．これらのカーネルは基底によって**表 1** のように演算数とメモリアクセス回数の比が変化する．大きい基底を用いることによって演算数, 特に乗算回数が少なくなり, またメモリアクセス回数も少なくなることが知られている[4)．Split-Radix FFT では演算パターンがやや不規則になるが, 演算数は最小となることが知られている．大きな基底を用いることにより演算数やメモリアクセス数は減少するが, 大きな基底ほど多くのレジスタを必要とするため, レジスタが足りない場合にはメモリアクセス数が増加する．よって最適となる基底は利用可能なレジスタ数及びプロセッサの各命令の処理性能比により異なる．最近の計算機では 2 基底や Split-Radix を用いた場合メモリアクセスに要するサイクル数で実行時間が決まることが多く, そのような場合メモリアクセス率の低減のため 4,8 基底などの大きい基底を用いた方が高速に計算できる．

表 1　2,4,8 基底及び Split-Radix FFT カーネルの演算数とメモリアクセス回数
Table 1　Number of operations for radix-2,4,8 and Split-Radix FFT kernels

	radix-2	radix-4	radix-8	Split-Radix
load	4	8	16	8
store	4	8	16	8
fmul	4	12	32	8
fadd	6	22	66	16

本論文では $p = 8$, すなわち 8 基底 FFT カーネルについて考える．

3. 従来の 8 基底 FFT カーネル

様々な基底に対して積和演算命令に適した FFT カーネルが研究されてきている．Linzer は 2,4,8 基底及び Split-Radix の FFT カーネル[5), Karner は 2,3,5 基底の FFT カーネル[6), Goedecker は 2,3,4,5 基底の FFT カーネル[7) を提案している．

従来の 8 基底 FFT カーネルは**図 1** のように表わされる．入力 in(1) から in(7) にひねり係数をかけた後, 8 点 FFT を行うものである．Linzer の提案する積和演算命令向けの 8 基底 FFT カーネルはこれを以下のようなアイデアで積和演算命令に適した形式に変換したものである．

FFT カーネルでひねり係数の乗算部分では複素数の乗算のために $wr * x \pm wi * y$ のような計算が行われる．この計算には積和演算命令が 2 つ必要となる．これに対して以下のような変換を考える．

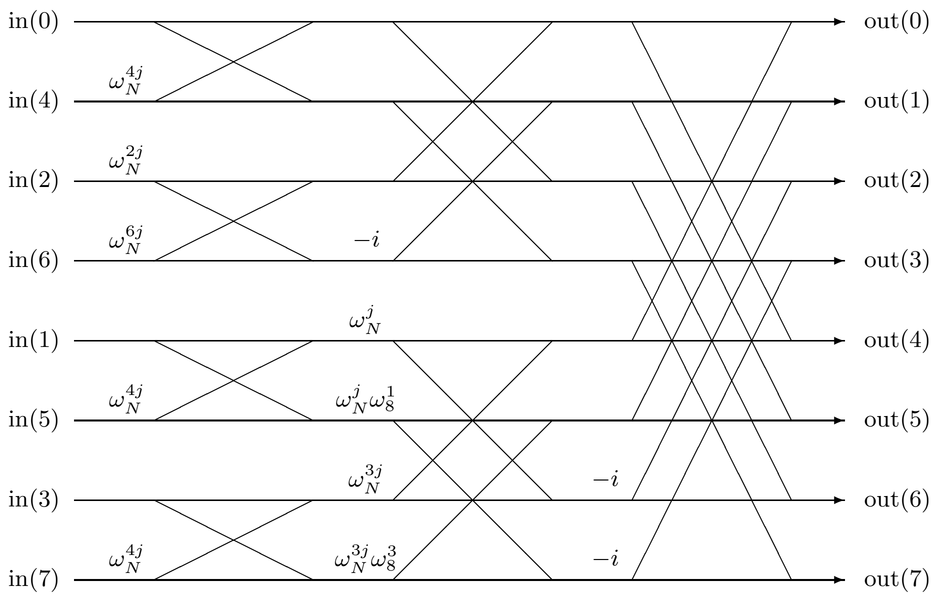

図 2 提案する 8 基底 FFT カーネル
Fig. 2 New Radix-8 FFT Kernel

$$wr * x \pm wi * y \rightarrow wr * [x \pm (wi/wr) * y]$$

ひねり係数の虚部と実部の比を用いて wr でくくり出すことによって [] 内を積和演算化することができる. wr の乗算は後続のバタフライ部分で行うことで積和演算に帰着させることができる.

この変換を従来の 8 基底 FFT カーネルに対して繰り返し適用することによって全ての乗算が積和演算として実行されるようになり, 積和演算数は加減算数と同じ 66 となる.

4. 提案する 8 基底 FFT カーネル

我々の提案する FFT カーネルは従来の 8 基底カーネルとは異なる 8 基底 FFT カーネルをベースとし, これを積和演算命令に適した形に変換するものである. ベースとなる 8 基底 FFT カーネルは**図 2** のように表わされる.

この 8 基底 FFT カーネルは 2 基底 FFT カーネルが 2 つ, 4 基底 FFT カーネルが 1 つ, Split-Radix FFT カーネルが 2 つを組み合わせてつくられている. 8 点 FFT に対して Split-Radix FFT を用いた状況に似ている.

まず in(0), in(2), in(4), in(6) に対しては 4 基底 FFT カーネルで計算を行う. また in(1) と in(5), in(3) と in(7) に対しては 2 基底 FFT カーネルで計算を行う. これらの出力に対して 2 組の時間間引き Split-Radix FFT の計算を行う.

図 1 と比較してみると, ひねり係数の乗算が移動していると言うこともできる. in(1),in(5) でのひねり係数から ω_N^j を, in(3),in(7) でのひねり係数から ω_N^{3j} をそれぞれ後方 (図では右方向) へ移動し, また ω_8^1 を前方 (図では左方向) へ移動している.($\omega_8^3 = -i\omega_8^1$)

表 2 8 基底 FFT カーネルの演算数の比較
Table 2 Number of operations of Radix-8 FFT Kernels

	Conventional Radix-8	New Radix-8
load	16	16
store	16	16
fmul	32	36
fadd	66	66

この 8 基底 FFT カーネルの演算数を従来の 8 基底 FFT カーネルと比較したものが**表 2** である. ひねり係数の複素数乗算回数は両方 9 回で等しい. 加減算数も従来のものと等しいが, 乗算数では増加している. これは従来の 8 基底 FFT カーネルの ω_8^1, ω_8^3 の乗算では実部と虚部の絶対値が等しいため乗算数が少なくなるためである. このため積和演算ユニットではなく, 乗算ユニットと加減算ユニットを搭載するアーキテクチャを用いる場合にはこのカーネルは効率が悪くなる可能性がある. しかしながら現在ほとんどのプロセッサでは浮動小数の乗算命令と加減算命令を同数同時に実行でき, このような場合には命令数の多い方である加減算の数が実行時間を決定する. このような場合少ない方の乗算命令数が増えることは問題にならない.

この FFT カーネルを Linzer や Goedecker らと同じ手法によって積和演算命令に適した演算形式に変換することができる. この変換は 2 基底 FFT カーネル,4 基底 FFT カーネル, Split-Radix FFT カーネルのそれぞれに対して独立に積和演算命令向けに変換を行っ

表 3　N 点 FFT の計算に要するひねり係数用テーブル
Table 3　Table size of twiddle factors for computing N-point FFT

	Normal		Optimized for FMA		
	Radix-4	Radix-8	Radix-4	Conventional Radix-8	New Radix-8
$\cos(x)$	3N/4	7N/8	N/2	3N/8	N/2
$\sin(x)$	3N/4	7N/8	0	0	0
$\sin(x)/\cos(x)$	0	0	3N/4	7N/8	3N/4
$\cos(3x)/\cos(x)$	0	0	N/4	N/4	N/4
$\cos(5x)/\cos(x)$	0	0	0	N/8	0
$\cos(7x)/\cos(x)$	0	0	0	N/8	0
$\cos(x)/\sqrt{2}$	0	0	0	N/8	0
Total	3N/2	7N/4	3N/2	15N/8	3N/2

たものと同じである．これにより積和演算命令の数は加減算数と同じ 66 にすることができる．

付録 A.1 に積和演算命令向けに変換を行った 8 基底 FFT カーネルを示す．

ここで wnr, wni, $wn31$ はひねり係数のためのテーブルであり，それぞれ次の値をあらかじめ計算して格納しておくものとする．

$$
\begin{aligned}
wnr(k) &= \Re(\omega_N^k) \\
&= \cos(-2\pi k/N) \\
&\quad 0 \le k < N/2 \\
wni(k) &= \Im(\omega_N^k)/\Re(\omega_N^k) \\
&= \sin(-2\pi k/N)/\cos(-2\pi k/N) \\
&\quad 0 \le k < 3N/4 \\
wn31(k) &= \Re(\omega_N^{3k})/\Re(\omega_N^k) \\
&= \cos(-6\pi k/N)/\cos(-2\pi k/N) \\
&\quad 0 \le k < N/4
\end{aligned}
$$

$N = 2^p$ 点 FFT の計算に必要とするひねり係数をいくつかの積和演算向けのカーネルについて比較したものが**表 3** である．提案する 8 基底 FFT カーネルが必要とするテーブルは 4 基底のそれと同じである．

N 点 FFT の計算を行う場合，従来の 8 基底 FFT カーネルでは $15N/8$ のテーブルが必要であるのに対し，我々の提案する 8 基底 FFT カーネルでは $3N/2$ だけでよい．我々が提案する 8 基底 FFT カーネルは，従来の 8 基底で必要である 2, 4 基底になく 8 基底固有な $\cos(5x)/\cos(x), \cos(7x)/\cos(x)$ のテーブルを必要としない．このため，4 基底と 8 基底を混在して用いる場合でも 4 基底用のテーブルを用意するだけでよいという利点がある．一般に入力サイズ N が 2 のべき乗であることは多いが，$N = 8^p \times 4^q (q = 1, 2)$ である場合にはこの利点が特に有効である．

また，カーネル中でロードするひねり係数関係の浮動小数の数は，Linzer の提案する 8 基底カーネルでは 15 であるのに対して，我々の提案する 8 基底カーネルでは 14 と少ない．multicolumn FFT[2] の計算を行う場合にはひねり係数関係のロードが性能に影響を及ぼすことがあり，そのような場合には必要なひねり係数が少ないこととテーブルが小さいことはキャッシュミスを低下させるということが我々のカーネルの利点である．

表 4　Itanium2 の計算環境
Table 4　Itanium2 Computing Environment

CPU Clock	1.3GHz
L2 cache	256kB,8-way,WB/WA☆, 128B line
L3 cache	3MB, 12-way,WB/WA, 128B line
Compiler	Intel C Compiler 7.1(ecc -O3)

表 5　Power4 の計算環境
Table 5　Power4 Computing Environment

CPU Clock	1.3GHz
L1D cache	64kB, 2-way,WT/NWA☆☆,128B line
L2 cache	1440kB,8-way,WB/WA, 128B line
L3 cache	128MB, 8-way,WB/WA, 512B line
Compiler	IBM XLC compiler (xlc -q64 -O5 -qarch=pwr4 -qtune=pwr4)

表 6　8^4 点 FFT の計算のパフォーマンスモニタによる計測
Table 6　Performance monitor data in computing 8^4-point FFT

	Conv. radix-8	New radix-8
CPU_CYCLES	192026	179051
L2_REFERENCES	82025	81401
L2_MISSES	675	563
L3_MISSES	293	173

5. 性能評価

我々の提案する 8 基底 FFT カーネルと従来の 8 基底 FFT カーネルの性能を比較する．Linzer の提案するカーネルとして用いたコードを**付録 A.2** に示す．これは文献[5] の $0 \le k < n/40$ の場合に修正を加えたものである．

積和演算命令を持つ Intel Itanium2, IBM Power4 を搭載するシステム上で評価を行う．これらのシステムの仕様を**表 4**, **表 5** に示す．

FFT プログラムの実装としてはまず**図 3** のような

☆ WriteBack/WriteAllocate
☆☆ WriteThrough/NoWriteAllocate

```
radix8(src,dst,ls,ns) {
  for (i = 0; i < ls; i++) {
    /* ひねり係数を読み込む */
    for (j = 0; j < ns; j++) {
      /* 入力データを読み込む */
      /* 8 基底 FFT カーネルを計算する */
      /* 出力データを書き込む */
    }
  }
}

fft(src,dst,n) {
  ls = 1;
  ns = n/8;
  for (i = 0; i < logn; i++) {
    radix8(src,dst,ls,ns);
    ls = ls * 8;
    ns = ns / 8;
    SWAP(dst,src);
  }
}
```

図 3 FFT の実行コード 1
Fig. 3 FFT Program 1

ひねり係数関係のメモリアクセスが少ないコードを用いた (実行コード 1).

また, Stockham FFT アルゴリズム[2),8)] による自動ソート機能を用いた.

Itanium2 の Performance Monitor を使用し, 8^4 点 FFT 計算時の実行サイクル数 (CPU_CYCLES), L2 キャッシュの参照 (L2_REFERENCES), L2 キャッシュのミス (L2_MISSES), 及び L3 キャッシュのミス (L3_MISSES) を計測したものが**表 6** である. 使用するメモリ領域は我々の提案する FFT カーネルでは 176kB, 従来の FFT カーネルでは 188kB であり, Itanium2 ではともに L2 キャッシュの容量に収まる量である. Itanium2 では浮動小数点データは L1 データキャッシュを経由しないため, L2 キャッシュに対して計測を行った.

まず, 我々が提案する FFT カーネルが従来の FFT カーネルより必要なひねり係数の数が少ないため, メモリアクセス (L2_REFERENCES) が少ないことが分かる. メモリアクセスが少ないことと, またひねり係数のテーブルが小さいことにより L2 キャッシュおよび L3 キャッシュのミスが低減している.

実行コード 1 を用いて N を変えて演算性能を比較したものが**表 7**, **表 8** である. Mflops 値は実際に実行された積和演算命令数より算出した. 1 積和演算を 2 浮動小数演算として数え, N 点 FFT の演算数は $(11/2)N \log_2 N$ であるとする.

N が小さい場合には性能差が大きいが, N が大きくなるにつれてキャッシュミス率が上がっていくためほとんど差がなくなる.

表 7 Itanium2 での演算性能 (実行コード 1)
Table 7 Mflops on Itanium2(FFT Program 1)

N	Conventional radix-8	New radix-8
8^3	1402 Mflops	1494 Mflops
8^4	1748 Mflops	1792 Mflops
8^5	906 Mflops	919 Mflops
8^6	280 Mflops	294 Mflops
8^7	184 Mflops	184 Mflops

表 8 Power4 での演算性能 (実行コード 1)
Table 8 Mflops on Power4(FFT Program 1)

N	Conventional radix-8	New radix-8
8^3	1545 Mflops	1659 Mflops
8^4	1488 Mflops	1572 Mflops
8^5	689 Mflops	693 Mflops
8^6	280 Mflops	286 Mflops
8^7	299 Mflops	314 Mflops
8^8	239 Mflops	239 Mflops

```
radix8(src,dst,ls,ns) {
  if (ns != 1) {
    for (i = 0; i < ls; i++) {
      /* ひねり係数を読み込む */
      for (j = 0; j < ns; j++) {
        /* 入力データを読み込む */
        /* 8 基底 FFT カーネルを計算する */
        /* 出力データを書き込む */
      }
    }
  } else {
    for (j = 0; j < ls; j++) {
      /* ひねり係数を読み込む */
      /* 入力データを読み込む */
      /* 8 基底 FFT カーネルを計算する */
      /* 出力データを書き込む */
    }
  }
}
```

図 4 FFT の実行コード 2
Fig. 4 FFT Program 2

次に**図 4** のように最内側ループのループ回数 ns が 1 である場合を特別に扱っている実装を考える (実行コード 2). ひねり係数のロード回数などは実行コード 1 と変わらない. この実行コード 2 を用いて評価を行ったところ**表 9**, **表 10** のような結果が得られた. 実行コード 2 では $ns = 1$ の場合のコードが高速化されているため, 実行コード 1 よりも高い性能が出ている. $ns = 1$ の時が一番多くのひねり係数をロードするため, 提案するカーネルの利点であるひねり係数のロード回数の少なさが与える影響が大きくなっている.

表 9　Itanium2 での演算性能 (実行コード 2)
Table 9　Mflops on Itanium2(FFT Program 2)

N	Conventional radix-8	New radix-8
8^3	1490 Mflops	1749 Mflops
8^4	1872 Mflops	1960 Mflops
8^5	843 Mflops	909 Mflops
8^6	429 Mflops	445 Mflops
8^7	397 Mflops	409 Mflops

表 10　Power4 での演算性能 (実行コード 2)
Table 10　Mflops on Power4(FFT Program 2)

N	Conventional radix-8	New radix-8
8^3	1749 Mflops	1949 Mflops
8^4	1729 Mflops	1844 Mflops
8^5	765 Mflops	835 Mflops
8^6	303 Mflops	316 Mflops
8^7	342 Mflops	355 Mflops
8^8	231 Mflops	233 Mflops

以上のようにひねり係数のロード回数の少ない実装を用いた場合にも Itanium2 では最大 17%, Power4 では最大 11%の性能向上がみられ, 提案するカーネルの優位性が十分に示された. 他のひねり係数のロード回数がより多い実装を用いた場合にはさらに大きな性能向上を生むと考えられる.

6. ま と め

積和演算命令に適した新しい 8 基底の FFT カーネルを提案した. この FFT カーネルは従来の積和演算命令向けの FFT カーネルと比べて必要なひねり係数のテーブルが小さく, N 点 FFT を $3N/2$ の浮動小数テーブルで計算することができる. またロードするひねり係数の数も 14 と既存の積和演算命令向けの 8 基底 FFT カーネルよりも少なく, キャッシュミスの低下により高い性能が得られる. Itanium2, Power4 で評価を行い, 従来のカーネルより適していることを示した.

謝辞　本研究を進めるに当たり貴重なアドバイスを頂きました方々に感謝致します. なお, 本研究の一部は科学技術振興機構戦略的創造研究推進事業によるものである.

参 考 文 献

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 (1965).
2) C. Van Loan: *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA (1992).
3) P. Duhamel and H. Hollmann: Split-Radix FFT Algorithm, *Electron. Lett.*, Vol. 20, pp. 14–16 (1984).
4) G. D. Bergland: A Fast Fourier Transform Algorithm Using Base 8 Iterations, *Math. Comp.*, Vol. 22, pp. 275–279 (1968).
5) E. N. Linzer and E. Feig: Implementation of Efficient FFT Algorithms on Fused Multiply-Add Architectures, *IEEE Trans. Signal Processing*, Vol. 41, pp. 93–107 (1993).
6) H. Karner, et al.: Multiply-Add Optimized FFT Kernels, *Math. Models and Methods in Appl. Sci.*, Vol. 11, pp. 105–117 (2001).
7) S. Goedecker: Fast Radix 2,3,4 and 5 Kernels for Fast Fourier Transformations on C omputers with Overlapping Multiply-Add Instructions, *SIAM J. Sci. Comput.*, Vol. 18, pp. 1605–1611 (1997).
8) P. N. Swarztrauber: FFT algorithms for vector computers, *Parallel Computing*, Vol. 1, pp. 45–63 (1984).

付　　録

A.1　積和演算命令に適した形式に変換した提案する新しい 8 基底 FFT カーネル

$\omega r_1 = wnr(j)$
$\omega r_2 = wnr(2j)$
$\omega r_4 = wnr(4j)$
$\omega r_{81} = wnr(j + N/8)$
$\omega i_1 = wni(j)$
$\omega i_2 = wni(2j)$
$\omega i_3 = wni(3j)$
$\omega i_4 = wni(4j)$
$\omega i_6 = wni(6j)$
$\omega i_{81} = wni(j + N/8)$
$\omega i_{83} = wni(3j + 3N/8)$
$\omega r_{31} = wn31(j)$
$\omega r_{831} = wn31(j + N/8)$
$\omega r_{62} = wn31(2j)$
$u0 = inr(0)$
$v0 = ini(0)$
$u1 = inr(1)$
$v1 = ini(1)$
$r = inr(2)$
$s = ini(2)$
$u2 = r - s * \omega i_2$
$v2 = r * \omega i_2 + s$
$u3 = inr(3)$
$v3 = ini(3)$
$r = inr(4)$
$s = ini(4)$
$u4 = r - s * \omega i_4$
$v4 = r * \omega i_4 + s$
$r = inr(5)$
$s = ini(5)$
$u5 = r - s * \omega i_4$
$v5 = r * \omega i_4 + s$
$r = inr(6)$
$s = ini(6)$
$u6 = r - s * \omega i_6$
$v6 = r * \omega i_6 + s$
$r = inr(7)$
$s = ini(7)$
$u7 = r - s * \omega i_4$
$v7 = r * \omega i_4 + s$

$r0 = u0 + u4 * \omega r_4$
$s0 = v0 + v4 * \omega r_4$
$r1 = u2 + u6 * \omega r_{62}$
$s1 = v2 + v6 * \omega r_{62}$
$r2 = u1 + u5 * \omega r_4$
$s2 = v1 + v5 * \omega r_4$
$r3 = u3 + u7 * \omega r_4$
$s3 = v3 + v7 * \omega r_4$
$r4 = u0 - u4 * \omega r_4$
$s4 = v0 - v4 * \omega r_4$
$r5 = u2 - u6 * \omega r_{62}$
$s5 = v2 - v6 * \omega r_{62}$
$r6 = u1 - u5 * \omega r_4$
$s6 = v1 - v5 * \omega r_4$
$r7 = u3 - u7 * \omega r_4$
$s7 = v3 - v7 * \omega r_4$
$u0 = r0 + r1 * \omega r_2$
$v0 = s0 + s1 * \omega r_2$
$u1 = r0 - r1 * \omega r_2$
$v1 = s0 - s1 * \omega r_2$
$r = r2 - s2 * \omega i_1$
$s = r2 * \omega i_1 + s2$
$p = r3 - s3 * \omega i_3$
$q = r3 * \omega i_3 + s3$
$u2 = r + p * \omega r_{31}$
$v2 = s + q * \omega r_{31}$
$u3 = r - p * \omega r_{31}$
$v3 = s - q * \omega r_{31}$
$outr(0) = u0 + u2 * \omega r_1$
$outi(0) = v0 + v2 * \omega r_1$
$outr(4) = u0 - u2 * \omega r_1$
$outi(4) = v0 - v2 * \omega r_1$
$outr(2) = u1 + v3 * \omega r_1$
$outi(2) = v1 - u3 * \omega r_1$
$outr(6) = u1 - v3 * \omega r_1$
$outi(6) = v1 + u3 * \omega r_1$
$u0 = r4 + s5 * \omega r_2$
$v0 = s4 - r5 * \omega r_2$
$u1 = r4 - s5 * \omega r_2$
$v1 = s4 + r5 * \omega r_2$

$r = r6 - s6 * \omega i_{81}$
$s = r6 * \omega i_{81} + s6$
$p = r7 - s7 * \omega i_{83}$
$q = r7 * \omega i_{83} + s7$
$u2 = r + p * \omega r_{831}$
$v2 = s + q * \omega r_{831}$
$u3 = r - p * \omega r_{831}$
$v3 = s - q * \omega r_{831}$
$outr(1) = u0 + u2 * \omega r_{81}$
$outi(1) = v0 + v2 * \omega r_{81}$
$outr(5) = u0 - u2 * \omega r_{81}$
$outi(5) = v0 - v2 * \omega r_{81}$
$outr(3) = u1 + v3 * \omega r_{81}$
$outi(3) = v1 - u3 * \omega r_{81}$
$outr(7) = u1 - v3 * \omega r_{81}$
$outi(7) = v1 + u3 * \omega r_{81}$

A.2 Linzer の 8 基底 FFT カーネル

$\omega r_1 = wnr(j)$
$\omega r_2 = wnr(2j)$
$\omega r_4 = wn4(j)$
$\omega i_1 = wni(j)$
$\omega i_2 = wni(2j)$
$\omega i_3 = wni(3j)$
$\omega i_4 = wni(4j)$
$\omega i_5 = wni(5j)$
$\omega i_6 = wni(6j)$
$\omega i_7 = wni(7j)$
$\omega r_{31} = wn31(j)$
$\omega r_{62} = wn31(2j)$
$\omega r_{51} = wn51(j)$
$\omega r_{71} = wn71(j)$
$\omega r_{81} = wn81(j)$
$u0 = inr(0)$
$v0 = ini(0)$
$r = inr(1)$
$s = ini(1)$
$u1 = r - s * \omega i_1$
$v1 = r * \omega i_2 + s$
$r = inr(2)$
$s = ini(2)$
$u2 = r - s * \omega i_2$
$v2 = r * \omega i_2 + s$
$r = inr(3)$
$s = ini(3)$
$u3 = r - s * \omega i_3$
$v3 = r * \omega i_3 + s$
$r = inr(4)$
$s = ini(4)$
$u4 = r - s * \omega i_4$
$v4 = r * \omega i_4 + s$
$r = inr(5)$
$s = ini(5)$
$u5 = r - s * \omega i_5$
$v5 = r * \omega i_5 + s$
$r = inr(6)$
$s = ini(6)$
$u6 = r - s * \omega i_6$

$v6 = r * \omega i_6 + s$
$r = inr(7)$
$s = ini(7)$
$u7 = r - s * \omega i_7$
$v7 = r * \omega i_7 + s$
$r0 = u0 + u4 * \omega r_4$
$s0 = v0 + v4 * \omega r_4$
$r1 = u2 + u6 * \omega r_{62}$
$s1 = v2 + v6 * \omega r_{62}$
$r2 = u1 + u5 * \omega r_{51}$
$s2 = v1 + v5 * \omega r_{51}$
$r3 = u3 + u7 * \omega r_{73}$
$s3 = v3 + v7 * \omega r_{73}$
$r4 = u0 - u4 * \omega r_4$
$s4 = v0 - v4 * \omega r_4$
$r5 = v2 - v6 * \omega r_{62}$
$s5 = u6 * \omega r_{62} - u2$
$r6 = u1 - u5 * \omega r_{51}$
$s6 = v1 - v5 * \omega r_{51}$
$r7 = v3 - v7 * \omega r_{73}$
$s7 = u7 * \omega r_{73} - u3$
$u0 = r0 + r1 * \omega r_2$
$v0 = s0 + s1 * \omega r_2$
$u1 = r0 - r1 * \omega r_2$
$v1 = s0 - s1 * \omega r_2$
$u2 = r2 + r3 * \omega r_2$
$v2 = s2 + s3 * \omega r_2$
$u3 = r2 - r3 * \omega r_2$
$v3 = s2 - s3 * \omega r_2$
$outr(0) = u0 + u2 * \omega r_1$
$outi(0) = v0 + v2 * \omega r_1$
$outr(4) = u0 - u2 * \omega r_1$
$outi(4) = v0 - v2 * \omega r_1$
$outr(2) = u1 + v3 * \omega r_1$
$outi(2) = v1 - u3 * \omega r_1$
$outr(6) = u1 - v3 * \omega r_1$
$outi(6) = v1 + u3 * \omega r_1$
$u0 = r4 + r5 * \omega r_{31}$
$v0 = s4 + s5 * \omega r_{31}$
$u1 = r4 - r5 * \omega r_{31}$

$v1 = s4 - s5 * \omega r_{31}$
$u2 = r6 + r7 * \omega r_{31}$
$v2 = s6 + s7 * \omega r_{31}$
$u3 = r6 - r7 * \omega r_{31}$
$v3 = s6 - s7 * \omega r_{31}$
$r4 = u2 + v2$
$s4 = v2 - u2$
$r5 = u3 - v3$
$s5 = v3 + u3$
$outr(1) = u0 + r4 * \omega r_{81}$
$outi(1) = v0 + s4 * \omega r_{81}$
$outr(5) = u0 - r4 * \omega r_{81}$
$outi(5) = v0 - s4 * \omega r_{81}$
$outr(3) = u1 + r5 * \omega r_{81}$
$outi(3) = v1 + s5 * \omega r_{81}$
$outr(7) = u1 - r5 * \omega r_{81}$
$outi(7) = v1 - s5 * \omega r_{81}$

$wnr(k) = \cos(-2\pi k/N)$ $\quad 0 \le k < N/4$
$wni(k) = \sin(-2\pi k/N)/\cos(-2\pi k/N)$ $\quad 0 \le k < 7N/8$
$wn31(k) = \cos(-6\pi k/N)/\cos(-2\pi k/N)$ $\quad 0 \le k < N/4$
$wn4(k) = \cos(-8\pi k/N)$ $\quad 0 \le k < N/8$
$wn51(k) = \cos(-10\pi k/N)/\cos(-2\pi k/N)$ $\quad 0 \le k < N/8$
$wn73(k) = \cos(-14\pi k/N)/\cos(-6\pi k/N)$ $\quad 0 \le k < N/8$
$wn81(k) = \cos(-2\pi k/N)/\sqrt{2}$ $\quad 0 \le k < N/8$