ハイパフォーマンス
コンピューティング 76−10
（1999. 5. 14）

積和演算に向いた8基底FFT Kernelの提案

高橋大介† 金田康正†

本論文では，積和演算に向いた8基底FFTカーネルを提案する．このFFTカーネルは積和演算命令を持つプロセッサにおいて，従来の8基底FFTカーネルに比べて演算回数が削減される．提案する8基底FFTカーネルをワークステーションIBM RS/6000 590および共有メモリ型ベクトル並列計算機NEC SX-4に実現し，性能評価を行った．その結果，従来の8基底FFTカーネルや，Goedeckerが提案している積和演算に向いた4基底FFTカーネルに比べて高い性能が得られた．

Fast Radix-8 FFT Kernel Suitable for Multiply-Add Operations

DAISUKE TAKAHASHI† and YASUMASA KANADA†

In this paper, we propose a new radix-8 fast Fourier transform (FFT) kernel suitable for multiply-add operations. The proposed radix-8 FFT kernel requires less multiply-add operations than the conventional radix-8 FFT kernel. We implemented this radix-8 FFT kernel on the IBM RS/6000 590 workstation and NEC SX-4 shared-memory vector parallel computer. The result shows that our radix-8 FFT kernel is faster than the conventional radix-8 FFT kernel or Goedecker's radix-4 FFT kernel.

1. はじめに

高速Fourier変換（fast Fourier transform, FFT）[1]は，科学技術計算において今日広く用いられているアルゴリズムである．

$n = 2^m$ 点のFFTを計算するにあたって，これまでに2基底のFFT[1]や4基底[2]，8基底[3]のFFTが提案されてきた．基底を大きくすることにより，演算量，特に実数の乗算回数が減ることが知られている[2)~4)]．

最初にFFTが提案された1960年代においては，浮動小数点加算は浮動小数点乗算に比べてずっと高速であった．したがって，FFTにおいては実数の乗算回数を減らすようなアルゴリズムが多く提案されてきた[5),6)]．

しかし，今日では多くのプロセッサが浮動小数点加算と浮動小数点乗算を同じ速さで実行できる．さらに，加算と乗算を同時に行える，積和演算命令を持つプロセッサも多い．

積和演算命令を持たないプロセッサにおいては，実数の加算回数と乗算回数の和が演算回数となるが，積和演算命令を持つプロセッサにおいては，加算回数と乗算回数の比によって積和演算回数が変化する．

FFTにおいて，積和演算に着目した研究としては，Goedeckerによる2，3，4，5基底FFTカーネルにおける積和演算に向いた手法[7]が知られている．しかし，同様の手法を用いた8基底FFTカーネルは提案されていない．理由としては，積和演算に向いた手法を8基底の場合に適用しても4基底の場合に比べて積和演算回数を減らすことができないからであるとされている[7]．

しかし，実際のFFTの性能は演算回数だけではなく，ロードとストアの回数も大きく影響する．

近年のプロセッサにおいては，演算速度に対するメモリのアクセス速度が相対的に遅くなってきていることから，メモリアクセス回数を減らすことは，より重要になっている．特に，最近はSMP構成の計算機が増えてきていることから，演算速度に対するメモリのアクセス速度の差は，さらに大きくなると予想される．

したがって，これからのFFTアルゴリズムにおいては，演算回数だけではなく，メモリアクセス回数を減らすことが今まで以上に重要である．

8基底のFFTは，2基底や4基底のFFTと比べて演算量が減るばかりでなく，2基底のFFTと比べてトータルのロードとストアの回数が1/3で済み，4基底のFFTと比べても，ロードとストアの回数が2/3で済むという利点がある[2)]．これは，基底を大きくするに従ってデータを再利用できる回数が増えるためにメモリのロードとストアの回数が減るからである[2)]．

これらの事実から，積和演算に向いた8基底FFTカーネルを構築することにより，Goedeckerによる積和演算に向いた2基底や4基底のFFTカーネルに比

† 東京大学情報基盤センター
Computer Centre, University of Tokyo

べて高速にFFTが計算できると予想される．したがって本論文では，積和演算に向いた8基底FFTカーネルを提案し，その評価を行う．

なお，特に断わらない限り本論文で取り扱うFFTは複素FFTを意味することとし，積和演算とは，$x = y + z * w$ のような，4オペランドの積和演算を意味するものとする．

このような積和演算命令を持つプロセッサを搭載した，ワークステーションIBM RS/6000 590および共有メモリ型ベクトル並列計算機NEC SX-4で提案するFFTカーネルの評価を行う．

以下，2章で高速Fourier変換について，3章で従来の8基底FFTカーネルについて，4章で本論文で提案する8基底FFTカーネルを示す．5章で演算回数の比較を行い，6章で誤差の評価を行う．7章で性能評価結果を示す．最後の8章はまとめである．

2. 高速Fourier変換

FFTは，離散Fourier変換（discrete Fourier transform, DFT）を高速に計算するアルゴリズムとして知られている．DFTは次式で定義される．

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, \quad 0 \le k \le n-1 \qquad (1)$$

ここで，$\omega_n = e^{-2\pi i/n}$，$i = \sqrt{-1}$ である．

FFTカーネル[2),4)]はFFTにおいて，最内側のループで計算される処理であり，FFTカーネルの基底（radix）を p で表すと，

$$Y(k) = \sum_{j=0}^{p-1} X(j) \Omega^j \omega_p^{jk} \qquad (2)$$

ここで，$\omega_p = e^{-2\pi i/p}$ である．Ω はひねり係数（twiddle factor）[4)]と呼ばれる1の原始根であり，複素数である．

基底 p のFFTカーネルにおいては入力データにひねり係数 Ω^j を掛けたものに対して p 点のショートDFT[8)]が実行される．

式(2)を計算するために，今までにさまざまな手法が提案されている[9),10)]．

本論文では，$p = 8$ の場合に積和演算に向いたFFTカーネルを提案する．

3. 従来の8基底FFTカーネル

従来の8基底FFTカーネル[2),3)]を以下に示す．以後簡単のため，$X(j)$ の実数部，虚数部をそれぞれ $X_R(j)$，$X_I(j)$ とし，$Y(k)$ についても同様に $Y_R(k)$，$Y_I(k)$ とする．また，ひねり係数 Ω^j においても，実数部と虚数部をそれぞれ wr_j と wi_j とする．

Conventional Radix-8

$cos_4 = cos(\pi/4)$
$u0 = X_R(0)$
$v0 = X_I(0)$
$r = X_R(1)$
$s = X_I(1)$
$u1 = r * wr_1 - s * wi_1$
$v1 = r * wi_1 + s * wr_1$
$r = X_R(2)$
$s = X_I(2)$
$u2 = r * wr_2 - s * wi_2$
$v2 = r * wi_2 + s * wr_2$
$r = X_R(3)$
$s = X_I(3)$
$u3 = r * wr_3 - s * wi_3$
$v3 = r * wi_3 + s * wr_3$
$r = X_R(4)$
$s = X_I(4)$
$u4 = r * wr_4 - s * wi_4$
$v4 = r * wi_4 + s * wr_4$
$r = X_R(5)$
$s = X_I(5)$
$u5 = r * wr_5 - s * wi_5$
$v5 = r * wi_5 + s * wr_5$
$r = X_R(6)$
$s = X_I(6)$
$u6 = r * wr_6 - s * wi_6$
$v6 = r * wi_6 + s * wr_6$
$r = X_R(7)$
$s = X_I(7)$
$u7 = r * wr_7 - s * wi_7$
$v7 = r * wi_7 + s * wr_7$
$r0 = u0 + u4$
$s0 = v0 + v4$
$r1 = u2 + u6$
$s1 = v2 + v6$
$r2 = u1 + u5$
$s2 = v1 + v5$
$r3 = u3 + u7$
$s3 = v3 + v7$
$r4 = u0 - u4$
$s4 = v0 - v4$
$r5 = u2 - u6$
$s5 = v2 - v6$
$r6 = u1 - u5$
$s6 = v1 - v5$
$r7 = u3 - u7$
$s7 = v3 - v7$
$u0 = r0 + r1$
$v0 = s0 + s1$
$u1 = r0 - r1$
$v1 = s0 - s1$
$u2 = r2 + r3$
$v2 = s2 + s3$
$u3 = r2 - r3$
$v3 = s2 - s3$
$Y_R(0) = u0 + u2$
$Y_I(0) = v0 + v2$
$Y_R(4) = u0 - u2$
$Y_I(4) = v0 - v2$
$Y_R(2) = u1 + v3$
$Y_I(2) = v1 - u3$
$Y_R(6) = u1 - v3$
$Y_I(6) = v1 + u3$
$u0 = r4 + s5$
$v0 = s4 - r5$
$u1 = r4 - s5$
$v1 = s4 + r5$
$u4 = r6 + s7$
$v4 = s6 - r7$
$u5 = s7 - r6$
$v5 = s6 + r7$
$u2 = cos_4 * (u4 + v4)$
$v2 = cos_4 * (v4 - u4)$
$u3 = cos_4 * (u5 + v5)$
$v3 = cos_4 * (u5 - v5)$
$Y_R(1) = u0 + u2$
$Y_I(1) = v0 + v2$
$Y_R(5) = u0 - u2$
$Y_I(5) = v0 - v2$
$Y_R(3) = u1 + u3$
$Y_I(3) = v1 + u3$
$Y_R(7) = u1 - u3$
$Y_I(7) = v1 - v3$

従来の8基底FFTカーネルは，前半部分と後半部分に分けることができる．前半部分は入力データ $X(j)$ とひねり係数 Ω^j の積の計算であり，$j = 1, 2, \ldots, 7$ に対して行うので7回の複素数の積が必要になる．複素数の積は，通常の計算方法では実数の積が4回と実数の和が2回必要になる．したがって，乗算と加算の

比が2:1となり，積和演算命令を持つプロセッサでは，加算器が半分遊んでしまうことになる．

また，変換の後半部分は4回の実数の積に対して52回の実数の加算となっている．つまり，後半部分では逆に乗算器が有効に使われていないことが分かる．このように，従来の8基底FFTカーネルは積和演算に適しているとはいいがたい．

4. 提案する8基底FFTカーネル

3章で述べたように，従来の8基底FFTカーネルでは積和演算命令が有効に使われていない．そこで，8基底FFTカーネルを変形して，積和演算命令を有効に活用することを考える．

式(2)において$p=2$の場合，つまり2基底FFTカーネルを例に，以下説明する．

$$
\begin{aligned}
&u0 = X_R(0)\\
&v0 = X_R(0)\\
&r = X_R(1)\\
&s = X_R(1)\\
&u1 = r * wr_1 - s * wi_1\\
&v1 = r * wi_1 + s * wr_1\\
&Y_R(0) = u0 + u1\\
&Y_I(0) = v0 + v1\\
&Y_R(1) = u0 - u1\\
&Y_I(1) = v0 - v1
\end{aligned}
$$

このカーネルでは，$u1$および$v1$を計算するのに積和演算が合計4回必要であり，$Y_R(0)$，$Y_I(0)$，$Y_R(1)$，$Y_I(1)$の計算において積和演算が合計4回必要である．つまり，FFTカーネル全体では積和演算が合計8回必要となる．

ところが，FFTにおいては$wr_1 \neq 0$であるために，上式においてwr_1をくくり出すことができ，その結果以下のように変形できる[7]．

$$
\begin{aligned}
&wi_1 = wi_1/wr_1\\
&u0 = X_R(0)\\
&v0 = X_R(0)\\
&r = X_R(1)\\
&s = X_R(1)\\
&Y_R(0) = u0 + wr_1 * (r - s * wi_1)\\
&Y_I(0) = v0 + wr_1 * (r * wi_1 + s)\\
&Y_R(1) = u0 - wr_1 * (r - s * wi_1)\\
&Y_I(1) = v0 - wr_1 * (r * wi_1 + s)
\end{aligned}
$$

このカーネルでは，積和演算が6回で済むことが分かる．ここで，$wi_1 = wi_1/wr_1$の値はあらかじめ計算しておくものとする．この変形を従来の8基底FFTカーネルに適用すると，積和演算に向いた8基底FFTカーネルが導かれる．

なお，これらの変形はくくり出す変数が0でないことが明らかでないと行えないために，最適化コンパイラでは不可能な変形であることに注意する．

New Radix-8

$$
\begin{aligned}
&cos_4 = cos(\pi/4)\\
&wi_1 = wi_1/wr_1\\
&wi_2 = wi_2/wr_2\\
&wi_3 = wi_3/wr_3\\
&wi_4 = wi_4/wr_4\\
&wi_5 = wi_5/wr_5\\
&wi_6 = wi_6/wr_6\\
&wi_7 = wi_7/wr_7\\
&wr_{31} = wr_3/wr_1\\
&wr_{51} = wr_5/wr_1\\
&wr_{62} = wr_6/wr_2\\
&wr_{73} = wr_7/wr_3\\
&wr_{121} = wr_1 * cos_4\\
&u0 = X_R(0)\\
&v0 = X_I(0)\\
&r = X_R(1)\\
&s = X_I(1)\\
&u1 = r - s * wi_1\\
&v1 = r * wi_1 + s\\
&r = X_R(2)\\
&s = X_I(2)\\
&u2 = r - s * wi_2\\
&v2 = r * wi_2 + s\\
&r = X_R(3)\\
&s = X_I(3)\\
&u3 = r - s * wi_3\\
&v3 = r * wi_3 + s\\
&r = X_R(4)\\
&s = X_I(4)\\
&u4 = r - s * wi_4\\
&v4 = r * wi_4 + s\\
&r = X_R(5)\\
&s = X_I(5)\\
&u5 = r - s * wi_5\\
&v5 = r * wi_5 + s\\
&r = X_R(6)\\
&s = X_I(6)\\
&u6 = r - s * wi_6\\
&v6 = r * wi_6 + s\\
&r = X_R(7)\\
&s = X_I(7)\\
&u7 = r - s * wi_7\\
&v7 = r * wi_7 + s\\
&r0 = u0 + u4 * wr_4\\
&s0 = v0 + v4 * wr_4\\
&r1 = u2 + u6 * wr_{62}\\
&s1 = v2 + v6 * wr_{62}
\end{aligned}
$$

$$
\begin{aligned}
&r2 = u1 + u5 * wr_{51}\\
&s2 = v1 + v5 * wr_{51}\\
&r3 = u3 + u7 * wr_{73}\\
&s3 = v3 + v7 * wr_{73}\\
&r4 = u0 - u4 * wr_4\\
&s4 = v0 - v4 * wr_4\\
&r5 = u2 - u6 * wr_{62}\\
&s5 = v2 - v6 * wr_{62}\\
&r6 = u1 - u5 * wr_{51}\\
&s6 = v1 - v5 * wr_{51}\\
&r7 = u3 - u7 * wr_{73}\\
&s7 = v3 - v7 * wr_{73}\\
&u0 = r0 + r1 * wr_2\\
&v0 = s0 + s1 * wr_2\\
&u1 = r0 - r1 * wr_2\\
&v1 = s0 - s1 * wr_2\\
&u2 = r2 + r3 * wr_{31}\\
&v2 = s2 + s3 * wr_{31}\\
&u3 = r2 - r3 * wr_{31}\\
&v3 = s2 - s3 * wr_{31}\\
&Y_R(0) = u0 + u2 * wr_1\\
&Y_I(0) = v0 + v2 * wr_1\\
&Y_R(4) = u0 - u2 * wr_1\\
&Y_I(4) = v0 - v2 * wr_1\\
&Y_R(2) = u1 + v3 * wr_1\\
&Y_I(2) = v1 - u3 * wr_1\\
&Y_R(6) = u1 - v3 * wr_1\\
&Y_I(6) = v1 + u3 * wr_1\\
&u0 = r4 + s5 * wr_2\\
&v0 = s4 - r5 * wr_2\\
&u1 = r4 - s5 * wr_2\\
&v1 = s4 + r5 * wr_2\\
&u4 = r6 + s7 * wr_{31}\\
&v4 = s6 - r7 * wr_{31}\\
&u5 = s7 * wr_{31} - r6\\
&v5 = s6 + r7 * wr_{31}\\
&u2 = u4 + v4\\
&v2 = v4 - u4\\
&u3 = u5 + v5\\
&v3 = u5 - v5\\
&Y_R(1) = u0 + u2 * wr_{121}\\
&Y_I(1) = v0 + v2 * wr_{121}\\
&Y_R(5) = u0 - u2 * wr_{121}\\
&Y_I(5) = v0 - v2 * wr_{121}\\
&Y_R(3) = u1 + u3 * wr_{121}\\
&Y_I(3) = v1 + v3 * wr_{121}\\
&Y_R(7) = u1 - u3 * wr_{121}\\
&Y_I(7) = v1 - v3 * wr_{121}
\end{aligned}
$$

表 1　積和演算命令を持つプロセッサにおける各 FFT カーネルの必要演算回数

	積和演算	ロード	ストア
Conventional Radix-4	28	8	8
Goedecker Radix-4[7]	22	8	8
Conventional Radix-8	84	16	16
New Radix-8	66	16	16

表 2　積和演算命令を持たないプロセッサにおける各 FFT カーネルの必要演算回数

	乗算	加算	ロード	ストア
Conventional Radix-4	12	22	8	8
Goedecker Radix-4[7]	14	22	8	8
Conventional Radix-8	32	66	16	16
New Radix-8	38	66	16	16

表 3　積和演算命令を持つプロセッサにおける各 FFT カーネルを用いた場合の n 点 FFT の必要演算回数の $n\log_2 n$ に対する比

	積和演算	ロード	ストア
Conventional Radix-4	3.5	1	1
Goedecker Radix-4[7]	2.75	1	1
Conventional Radix-8	3.5	0.667	0.667
New Radix-8	2.75	0.667	0.667

表 4　積和演算命令を持たないプロセッサにおける各 FFT カーネルを用いた場合の n 点 FFT の必要演算回数の $n\log_2 n$ に対する比

	乗算 + 加算	ロード	ストア
Conventional Radix-4	4.25	1	1
Goedecker Radix-4[7]	4.5	1	1
Conventional Radix-8	4.083	0.667	0.667
New Radix-8	4.333	0.667	0.667

提案する 8 基底 FFT カーネルでは，wi_1 から wi_7 の値は $\sin\alpha$ ではなく，$\tan\alpha$ の形になっており，wr_{31} から wr_{73} の値は $\cos\alpha/\cos\beta$ の形になっていることが分かる．これらの値はあらかじめテーブルとして作成しておけば良いので，何回も FFT を実行する場合においてはオーバーヘッドは無視できる．

5. 演算回数の比較

演算回数の比較にあたっては，積和演算に向いていない従来の 4 基底 FFT カーネル，Goedecker が提案している積和演算に向いた 4 基底 FFT カーネル [7]，従来の 8 基底 FFT カーネル，そして提案する 8 基底 FFT カーネルの 4 種類の各 FFT カーネルの必要演算回数を比較した．

5.1 積和演算命令を持つプロセッサの場合

積和演算命令を持つプロセッサにおける，各 FFT カーネルの必要演算回数を**表 1** に示す．

表 1 から分かるように，提案した 8 基底 FFT カーネルは，従来の 8 基底 FFT カーネルに比べて積和演算回数が 84 回から 66 回に削減されている．これは約 27%の積和演算回数の削減になる．

$n = p^t$ と表される場合，n 点 FFT の演算回数 T は，p 基底 FFT カーネルの演算回数を T_p とすると，

$$T = T_p \cdot \frac{1}{p}\log_p n \qquad (3)$$

で表される [2]．

表 1 に基づいて式 (3) より算出した，積和演算命令を持つプロセッサにおける各 FFT カーネルを用いた場合の n 点 FFT の必要演算回数の $n\log_2 n$ に対する比を**表 3** に示す．

表 3 から分かるように，提案した 8 基底 FFT カーネルの積和演算回数は従来の 8 基底 FFT カーネルに比べて積和演算回数は削減されているものの，Goedecker が提案している積和演算に向いた 4 基底 FFT カーネルと積和演算回数は同一になっている．

ところが，ロードとストアの回数は Goedecker の 4 基底 FFT カーネルに比べてそれぞれ 2/3 になっていることが分かる．したがって，積和演算回数およびロードとストア回数のトータルで考えると，提案した 8 基底 FFT カーネルは Goedecker の 4 基底 FFT カーネルに比べて有利であることが分かる．

5.2 積和演算命令を持たないプロセッサの場合

積和演算命令を持たないプロセッサにおける各 FFT カーネルの必要回数を**表 2** に示す．

表 2 に基づいて式 (3) より算出した，積和演算命令を持たないプロセッサにおける各 FFT カーネルを用いた場合の n 点 FFT の必要演算回数の $n\log_2 n$ に対する比を**表 4** に示す．

表 4 から分かるように，提案した 8 基底 FFT カーネルの乗算と加算の合計演算回数は Goedecker の 4 基底 FFT カーネルに比べて，約 4%の演算回数の削減になっているが，従来の 8 基底 FFT カーネルに比べると演算回数が約 6%増えている．これは，提案した 8 基底 FFT カーネルでは，乗算と加算のバランスを改善するために，乗算を増やしているからである．

つまり，提案した 8 基底 FFT カーネルは，演算回数を増やして積和演算回数を減らすという，一見相反することを行って，高速化を計っているといえる．

6. 誤差の評価

図 1 に従来の 8 基底 FFT カーネルの相対誤差と提案した 8 基底 FFT カーネルの相対誤差を IBM RS/6000 590 において比較したものを示す．誤差の評価にあたっては，$x_k = (k+1) + (n-k)i$，$0 \le k \le n-1$ のデータに対する FFT と逆 FFT を IEEE 表現の倍精度計算で行ったデータを FFT を計算する前のデータと比較することにより行った．図 1 から分かるように，提案した 8 基底 FFT カーネルは従来の 8 基底 FFT

表 5 Radix-4, 8 の FFT カーネルの性能 (IBM RS/6000 590)

	64 組の 64 点 FFT		1024 組の 64 点 FFT	
	Time (m sec)	MFLOPS	Time (m sec)	MFLOPS
DCFT (ESSL)	0.5157	238.26	17.139	114.71
Goedecker Radix-4[7]	0.4913	250.10	16.094	122.16
Conventional Radix-8	0.5304	231.68	12.031	163.41
New Radix-8	0.4865	252.61	11.465	171.49

表 6 Radix-4, 8 の FFT カーネル (65536 組の 64 点 FFT) の性能 (NEC SX-4)

# CPU	Conventional Radix-4		Goedecker Radix-4		Conventional Radix-8		New Radix-8	
	Time (sec)	GFLOPS	Time (sec)	GFLOPS	Time (sec)	GFLOPS	Time (sec)	GFLOPS
1	0.07844	1.604	0.06931	1.815	0.07458	1.687	0.06817	1.846
2	0.03942	3.192	0.03479	3.617	0.03750	3.355	0.03410	3.690
4	0.01974	6.373	0.01745	7.209	0.01873	6.717	0.01710	7.359

表 7 Radix-4, 8 の FFT カーネル (1024 組の 4096 点 FFT) の性能 (NEC SX-4)

# CPU	Conventional Radix-4		Goedecker Radix-4		Conventional Radix-8		New Radix-8	
	Time (sec)	GFLOPS	Time (sec)	GFLOPS	Time (sec)	GFLOPS	Time (sec)	GFLOPS
1	0.16720	1.505	0.14463	1.740	0.16147	1.559	0.14119	1.782
2	0.08399	2.996	0.07239	3.476	0.08054	3.125	0.07067	3.561
4	0.04205	5.985	0.03628	6.936	0.04044	6.224	0.03548	7.092

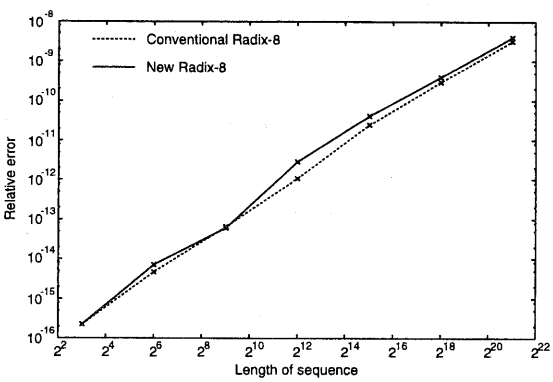

図 1 8 基底 FFT カーネルの相対誤差

カーネルに比べて若干誤差が増えるものの，極端な精度の低下は見られない．誤差が増える原因としては，提案した 8 基底 FFT カーネルにおいては，$wi_1 \sim wi_7$ は tan 関数をそれぞれ 1 回呼び出してテーブルを作成できるが，$wr_{31} \sim wr_{73}$ は $\cos\alpha/\cos\beta$ の形になっているため，cos 関数をそれぞれ 2 回呼び出す必要があり，さらに除算が 1 回必要になるためであると考えられる．

7. 性能評価

性能評価にあたっては，FFT カーネルのみの性能を比較するために，複数 (m) 組の n 点 FFT を同時に計算し，実行時間を比較した．最内側ループが m 回繰り返される，このような実行形態は，"four step" FFT[2),11)] や多次元 FFT に見ることができる．MFLOPS 値および GFLOPS 値の算出にあたっては，n 点 FFT の演算量を $5n\log_2 n$ とし，m 組の n 点 FFT の演算量は $5m \cdot n\log_2 n$ とした．

なお FFT の計算は倍精度複素数で行い，三角関数のテーブルはあらかじめ作り置きとしている．

計算機としては，ワークステーション IBM RS/6000 590 (POWER2 66MHz，ピーク性能 266MFLOPS) および共有メモリ型ベクトル並列計算機 NEC SX-4 (1CPU 当たりのピーク性能 2GFLOPS) を用いた．これら 2 つの計算機には，積和演算を行える命令を持つプロセッサが搭載されている．

7.1 IBM RS/6000 590 による測定結果

IBM RS/6000 590 においては，IBM のライブラリである ESSL (version 2.2.1) の FFT ルーチン (`DCFT`) および，Goedecker が提案している積和演算に向いた 4 基底 FFT カーネル [7)]，従来の 8 基底 FFT カーネル，そして提案する 8 基底 FFT カーネルの 4 種類の性能の比較を行った．

コンパイラは IBM の XL Fortran (version 3.2) を用い，最適化オプションとして `-O3 -qarch=pwr2 -qhot -qtune=pwr2` を指定した．測定に関しては，CPU 時間を測定した．IBM RS/6000 590 における 64 組の 64($=4^3=8^2$) 点 FFT および 1024 組の 64 点 FFT の実行時間を**表 5** に示す．

表 5 から分かるように，64 組の 64 点 FFT でも 1024 組の 64 点 FFT においても，提案する 8 基底 FFT カーネルは，ESSL の `DCFT` ルーチン，Goedecker が提案

している積和演算に向いた4基底FFTカーネルや従来の8基底FFTカーネルに比べて，高い性能が得られている．

以下，理由を考察する．今回評価に用いたIBM RS/6000 590ではデータキャッシュの大きさが128KBである．したがって，データの大きさが64KBとなる64組の64点FFTでは，キャッシュにデータが入ってしまうために，4基底FFTカーネルでも8基底FFTカーネルでも性能はあまり変わらないと推測できる．

ところが，1024組の64点FFTではデータの大きさが1MBになり，キャッシュにデータが入り切らないために，メモリアクセスの少ない8基底FFTカーネルが有利になると考えられる．

また，8基底FFTカーネルにおいては，提案する8基底FFTカーネルの方が積和演算回数が少ないために，従来の8基底FFTカーネルより高い性能が得られていると考えられる．

7.2 NEC SX-4による測定結果

NEC SX-4においては，積和演算に向いていない従来の4基底FFTカーネルおよび，Goedeckerが提案している積和演算に向いた4基底FFTカーネル[7]，従来の8基底FFTカーネル，そして提案する8基底FFTカーネルの4種類のFFTカーネルの性能の比較を行った．

NEC SX-4の評価に関しては，1CPU～4CPUにより経過時間を測定した．コンパイラはNECのFORTRAN77/SX（Rev. 134 1997/07/17）を用い，最適化オプションとして1CPUの実行に際しては`-C hopt -pi`を用い，2CPU，4CPUの実行に際しては`-P auto -C hopt -pi`を用いた．NEC SX-4における65536組の$64(=4^3=8^2)$点FFTおよび1024組の$4096(=4^6=8^4)$点FFTの実行時間を**表6**，**表7**に示す．

表6，表7から分かるように，NEC SX-4では各CPUでベクトル処理を行っているために，データ数が大きくなってもメモリアクセス性能は低下していない．

さらに，NEC SX-4では4CPUの場合でもメモリバンド幅に余裕があるために，IBM RS/6000 590においてキャッシュにデータが入ってしまう場合と同様な傾向が見られるが，メモリアクセスおよび積和演算回数の少ない，提案した8基底FFTカーネルが最も性能が高いことが分かる．

8. まとめ

本論文では，積和演算に向いた8基底FFTカーネルを提案した．今回提案した8基底FFTカーネルに用いた手法は，$n/2$点の複素数FFTでn点の実数FFTを処理する場合[2),4)]の前処理や後処理などにも適用が可能である．また，本手法はDSP (Digital Signal Processor) のように積和演算を高速に行えるハードウェアでFFTを処理する際にも適用可能であると考えられる．

謝辞 本研究の一部は，文部省科学研究費補助金奨励研究（A）（課題番号10780166）の支援を受けた．

参考文献

1) Cooley, J. W. and Tukey, J. W.: An Algorithm for the Machine Calculation of Complex Fourier Series, *Math. Comp.*, Vol. 19, pp. 297–301 (1965).
2) Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA (1992).
3) Bergland, G. D.: A Fast Fourier Transform Algorithm Using Base 8 Iterations, *Math. Comp.*, Vol. 22, pp. 275–279 (1968).
4) Brigham, E. O.: *The Fast Fourier Transform and its Applications*, Prentice-Hall, Englewood Cliffs, NJ (1988).
5) Winograd, S.: On Computing the Discrete Fourier Transform, *Math. Comp.*, Vol. 32, pp. 175–199 (1978).
6) Burrus, C. S. and Eschenbacher, P. W.: An In-Place, In-Order Prime Factor FFT Algorithm, *IEEE Trans. Audio Electroacoust.*, Vol. 29, pp. 806–817 (1981).
7) Goedecker, S.: 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 (1997).
8) Nussbaumer, H. J.: *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, New York, second corrected and updated edition (1982).
9) Temperton, C.: Self-Sorting Mixed-Radix Fast Fourier Transforms, *J. Comput. Phys.*, Vol. 52, pp. 1–23 (1983).
10) Singleton, R. C.: An Algorithm for Computing the Mixed Radix Fast Fourier Transform, *IEEE Trans. Audio Electroacoust.*, Vol. 17, pp. 93–103 (1969).
11) Bailey, D.H.: FFTs in External or Hierarchical Memory, *J. Supercomputing*, Vol. 4, pp. 23–35 (1990).