FFTにおけるキャッシュ最適化方式

高橋大介

筑波大学大学院システム情報工学研究科／
計算科学研究センター

1 はじめに

高速 Fourier 変換（fast Fourier transform、以下 FFT）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。

多くの FFT アルゴリズムは処理するデータがキャッシュメモリに載っている場合には高い性能を示す。しかし、問題サイズがキャッシュメモリのサイズより大きくなった場合においては著しい性能の低下をきたす。FFT アルゴリズムにおける 1 つの目標は、いかにしてキャッシュミスの回数を減らすかということにある。

近年のプロセッサの演算速度に対する主記憶のアクセス速度は相対的に遅くなってきており、主記憶のアクセス回数を減らすことは、より重要になっている。したがって、キャッシュメモリを搭載したプロセッサにおける FFT アルゴリズムでは、演算回数だけではなく、主記憶のアクセス回数も減らすことが重要になる。ここで、キャッシュミスの回数を減らすことができれば、主記憶のアクセス回数を減らすうえで非常に効果があるといえる。

本稿では、FFT におけるキャッシュ最適化方式について述べる。

2 高速 Fourier 変換

2.1 離散 Fourier 変換

FFT は、離散 Fourier 変換（discrete Fourier transform、以下 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。

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

ここで、入力 x_j および出力 y_k は複素数の値であり、$\omega_n = e^{-2\pi i/n}$、$i = \sqrt{-1}$ である。

式 (1) に従ってそのまま n 点 DFT を計算すると、$O(n^2)$ の計算量が必要になるが、FFT の手法を用いることで、$O(n \log n)$ の計算量で n 点 DFT を計算することが可能である。FFT については、非常に多くの参考書が出版されているが、文献 [4, 16] 等を参考にされたい。

FFT アルゴリズムとしては Cooley-Tukey アルゴリズム [6] や、その変形である Stockham アルゴリズム [5, 13] が良く知られている。

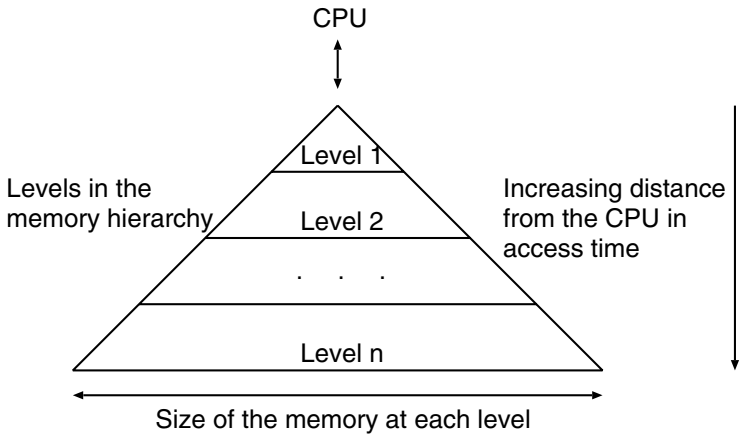

図 1: メモリ階層の例

2.2 FFT カーネル

FFT カーネル [16, 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 カーネルの基数（radix）を p で表すと、次式で表される。

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

ここで 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。

基数 p の FFT カーネルでは、入力データ $X(j)$ 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[10] が実行される。

式 (2) を計算するために、これまでにさまざまな手法が提案されている [12, 15]。

3 メモリアクセスの局所性

3.1 メモリ階層

メモリ階層（memory hierarchy）の例を図 1 に示す。メモリ階層は記憶域に対するアクセスパターンの局所性（locality）を前提に設計されている。局所性には時間的局所性と空間的局所性があり、前者は、ある一定のアドレスに対するアクセスは、比較的近い時間内に再発するという性質、後者は、ある一定時間内にアクセスされるデータは、比較的近いアドレスに分布するという性質である。

これらの傾向は、事務計算などの非数値計算プログラムには当てはまることが多いが、数値計算プログラムでは一般的ではない。特に大規模な科学技術計算においては、データ参照に時間的局所性がないことが多い。これが、科学技術計算でベクトル型スーパーコンピュータが有利であった大きな理由である。

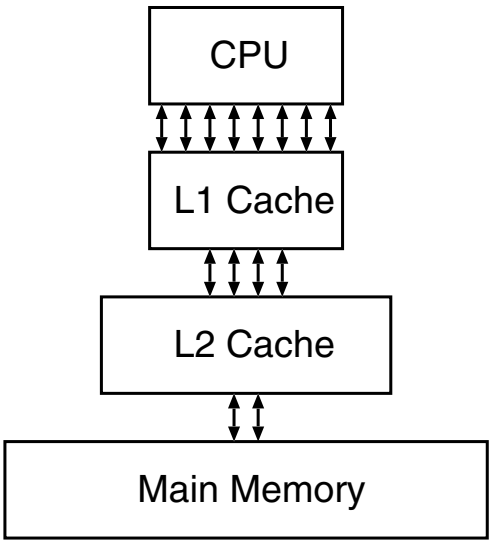

図 2: キャッシュメモリ

ベクトル型スーパーコンピュータはデータに関してキャッシュを用いないが、RISC プロセッサのようなスカラプロセッサは性能をキャッシュに深く依存している。したがって、RISC プロセッサで高い性能を得るためにはブロック化を行うなどして、意図的にアクセスパターンに局所性を与えることが必要になる。

多くの計算機では、メモリとレジスタの間にキャッシュメモリが入っている。キャッシュメモリはメモリとレジスタの中間的な性質を持ち、メモリ内のデータや機械語命令のうち使用頻度の高い部分がキャッシュメモリに入り、さらに使用頻度の高い部分がレジスタに入る。

キャッシュメモリは通常データが入るデータキャッシュと機械語命令が入る命令キャッシュに分かれている。そのうちチューニングに特に関係があるのはデータキャッシュであるので、本稿ではデータキャッシュについて説明する。

3.2 キャッシュミス

メモリ階層のうち、上位階層の小容量高速メモリを、通常キャッシュメモリあるいはキャッシュと呼ぶ。

キャッシュについても、階層的に構成することができる。つまり、より高速な 1 次キャッシュ（L1 Cache）の下に、アクセス速度は 1 次キャッシュよりも遅いが、容量は 1 次キャッシュよりも大きい 2 次キャッシュ（L2 Cache）を設けるのである。このようにすることにより、メモリ階層の局所性をより生かすことが可能になる。さらに 3 次キャッシュまで備えたプロセッサもある。

演算に必要なデータがキャッシュメモリにないため、メモリから一旦キャッシュメモリに転送せざるを得ないことをキャッシュミスという。キャッシュメモリよりもメモリの方が低速な半導体を使用しているため、データをメモリからキャッシュメモリに転送する時間はキャッシュメモリから

```
      SUBROUTINE ZAXPY(N,A,X,Y)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMPLEX*16 A,X(*),Y(*)
C
      DO I=1,N
        Y(I)=Y(I)+A*X(I)
      END DO
      RETURN
      END
```

図 3: ZAXPY のプログラム例

レジスタに転送する時間の数倍かかる。したがってパフォーマンスを向上させるためにはキャッシュミスをできるだけ少なくする必要がある。

多階層のキャッシュを用いる場合には、1 次キャッシュのヒット時の速度を高速化するようにすること、2 次キャッシュのミスの割合を最小化するようにすること、が重要である。

3.3 ZAXPY ルーチンの性能

FFT カーネルは、主に複素数演算から構成されている。ここで、FFT カーネルに類似した ZAXPY（A X plus Y、倍精度複素ベクトルの積和）と呼ばれる演算を用いて、ベクトル長を変化させた場合の性能を測定した結果を図 4 に示す。

使用した計算機は Intel Xeon 3.06 GHz（FSB 533 MHz、512 KB L2 cache、PC2100 DDR-SDRAM）であり、コンパイラは Intel C Compiler 8.0 を用いている。

実行においては、Intel Pentium4 などに搭載されている SIMD（Single Instruction Multiple Data）命令である、SSE2 命令 [8] を用いたものと、従来の x87 命令を用いたもの、さらに Intel が提供している MKL（Math Kernel Library、Version 6.1.1）[9] の BLAS（Basic Linear Algebra Subprograms）で比較を行った。

ZAXPY は、図 3 に示すプログラムで記述することができ、1 回の iteration につき倍精度実数の加算、乗算がそれぞれ 4 回、load が 4 回、store が 2 回から成っている。

図 4 から分かるように、配列が L2 キャッシュに収まる領域（$N \leq 8192$）では、SSE2 命令を使ったプログラム（with SSE2）が最も高速である。この場合の最高性能は約 3 GFLOPS と、Xeon 3.06 GHz のピーク性能（6.12 GFLOPS）の約半分程度である。

ところが、L2 キャッシュを外れた場合には、x87 命令で実行した場合とほとんど同じ性能に低下してしまう。これは、メモリアクセスを少なくし、キャッシュの再利用性を高めることが高い性能を得るうえで不可欠であることを示している。

4 Six-Step FFT アルゴリズム

本章では、キャッシュメモリを有効に活用することのできる、six-step FFT アルゴリズム [3, 16] について説明する。six-step FFT アルゴリズムでは、一次元 FFT を二次元表現で表して計算することにより、キャッシュミスを少なくできるのが特徴である。

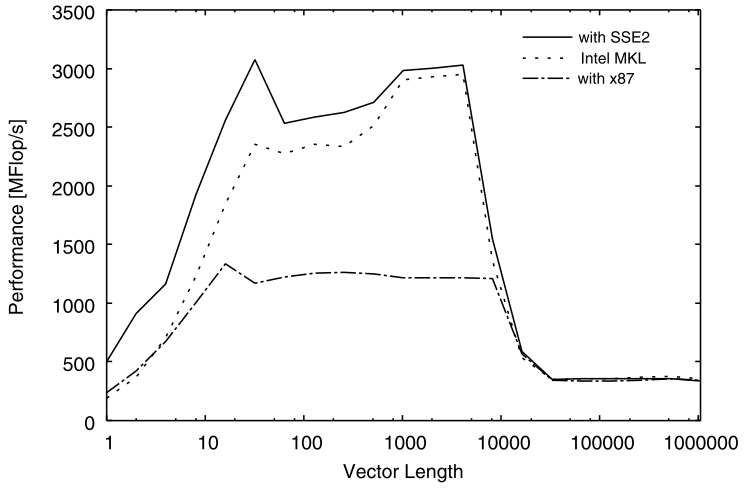

図 4: ZAXPY の性能（Intel Xeon 3.06 GHz）

n 点 DFT を計算する際に $n = n_1 \times n_2$ と分解できるものとすると、式 (1) における j および k は、

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2 \tag{3}$$

と書くことができる。そのとき、式 (1) の x と y は次のような二次元配列（columnwise）で表すことができる。

$$x_j = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, 0 \le j_2 \le n_2 - 1 \tag{4}$$

$$y_k = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, 0 \le k_2 \le n_2 - 1 \tag{5}$$

したがって、式 (1) は式 (6) のように変形できる。

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1} \tag{6}$$

式 (6) から次に示されるような、six-step FFT アルゴリズム [3, 16] が導かれる。

Step 1: 転置

$$x_1(j_2, j_1) = x(j_1, j_2)$$

Step 2: n_1 組の n_2 点 multicolumn FFT

$$x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1) \omega_{n_2}^{j_2 k_2}$$

Step 3: ひねり係数の乗算

$$x_3(k_2, j_1) = x_2(k_2, j_1) \omega_{n_1 n_2}^{j_1 k_2}$$

Step 4: 転置

$$x_4(j_1, k_2) = x_3(k_2, j_1)$$

Step 5: n_2 組の n_1 点 multicolumn FFT

$$x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2)\omega_{n_1}^{j_1 k_1}$$

Step 6: 転置

$$y(k_2, k_1) = x_5(k_1, k_2)$$

Step 3 における $\omega_{n_1 n_2}^{j_1 k_2}$ は、ひねり係数 [4] と呼ばれる 1 の原始根であり、複素数である。

図 5 に従来の six-step FFT アルゴリズムの疑似コードを示す。ここで、ひねり係数 $\omega_{n_1 n_2}^{j_1 k_2}$ は配列 U に格納されている。

従来の six-step FFT アルゴリズムの特徴を以下に示す。

- $n_1 = n_2 = \sqrt{n}$ とした場合、$\sqrt{n}$ 組の $\sqrt{n}$ 点 multicolumn FFT[16] が Step 2 と 5 で行われる。この $\sqrt{n}$ 点 multicolumn FFT はメモリアクセスの局所性が高く、キャッシュメモリを搭載したプロセッサに適している。
- 行列の転置が 3 回必要になる。

Step 1、4、6 の行列の転置および、Step 3 のひねり係数の乗算をブロック化した six-step FFT アルゴリズムが文献 [16] に示されている。しかし、この FFT アルゴリズムでは multicolumn FFT の部分と行列の転置の部分が分離されているため、multicolumn FFT においてキャッシュメモリに載っていたデータが行列の転置の際に有効に再利用されないという問題点がある。

ここで、3 回の行列の転置がキャッシュメモリを搭載した計算機においてボトルネックとなる。

5 ブロック Six-Step FFT アルゴリズム

本章では、さらにキャッシュ内のデータを有効に再利用し、キャッシュミスの回数を少なくするため、従来の six-step FFT では分離されていた multicolumn FFT と行列の転置を統合した、ブロック six-step FFT アルゴリズムについて説明する。前述した従来の six-step FFT において、$n = n_1 n_2$ とし、n_b をブロックサイズとする。ここで、プロセッサは multi-level キャッシュメモリを搭載しているものと仮定する。ブロック six-step FFT アルゴリズムは以下のようになる。

Step 1: $n_1 \times n_2$ の大きさの複素数配列 X に入力データが入っているとする。このとき、$n_1 \times n_2$ 配列 X から n_b 列ずつデータを転置しながら、$n_2 \times n_b$ の大きさの作業用配列 WORK に転送する。ここでブロックサイズ n_b は配列 WORK が L2 キャッシュに載るように定める。

Step 2: n_b 組の n_2 点 multicolumn FFT を L2 キャッシュに載っている $n_2 \times n_b$ 配列 WORK の上で行う。ここで各 column FFT は、ほぼ L1 キャッシュ内で行えるものとする。

Step 3: multicolumn FFT を行った後 L2 キャッシュに残っている $n_2 \times n_b$ 配列 WORK の各要素にひねり係数 U の乗算を行う。そしてこの $n_2 \times n_b$ 配列 WORK のデータを n_b 列ずつ転置しながら元の $n_1 \times n_2$ 配列 X の同じ場所に再び格納する。

Step 4: n_2 組の n_1 点 multicolumn FFT を $n_1 \times n_2$ 配列 X の上で行う。ここでも各 column FFT は、ほぼ L1 キャッシュ内で行える。

Step 5: 最後にこの $n_1 \times n_2$ 配列 X を n_b 列ずつ転置して、$n_2 \times n_1$ 配列 Y に格納する。

```
 1 COMPLEX*16 X(N1,N2),Y(N2,N1),U(N2,N1)
 2 DO I=1,N1
 3   DO J=1,N2
 4     Y(J,I)=X(I,J)
 5   END DO
 6 END DO
 7 DO I=1,N1
 8   CALL IN_CACHE_FFT(Y(1,I),N2)
 9 END DO
10 DO I=1,N1
11   DO J=1,N2
12     Y(J,I)=Y(J,I)*U(J,I)
13   END DO
14 END DO
15 DO J=1,N2
16   DO I=1,N1
17     X(I,J)=Y(J,I)
18   END DO
19 END DO
20 DO J=1,N2
21   CALL IN_CACHE_FFT(X(1,J),N1)
22 END DO
23 DO I=1,N1
24   DO J=1,N2
25     Y(J,I)=X(I,J)
26   END DO
27 END DO
```

図 5: 従来の six-step FFT アルゴリズム

図 6 にブロック six-step FFT アルゴリズムの疑似コードを、図 7 にメモリ配置を示す。図 6 のアルゴリズムにおいて、NB はブロックサイズであり、NP はパディングサイズ、WORK は作業用の配列である。なお、図 7 において、配列 X、WORK、Y の中の 1 から 16 の数字は、配列のアクセス順序を示している。

また、作業用の配列 WORK にパディングを施すことにより、配列 WORK から配列 X にデータを転送する際や、配列 WORK 上で multicolumn FFT を行う際にキャッシュラインコンフリクトの発生を極力防ぐことができる。

ブロック six-step FFT アルゴリズムは、いわゆる *two-pass* アルゴリズム [3, 16] となる。つまり、ブロック six-step FFT アルゴリズムでは n 点 FFT の演算回数は $O(n \log n)$ であるのに対し、主記憶のアクセス回数は理想的には $O(n)$ で済む。

なお、本稿では Step 2 および Step 4 の各 column FFT は L1 キャッシュに載ると想定しているが、問題サイズ n が非常に大きい場合には各 column FFT が L1 キャッシュに載らないことも十分予想される。このような場合は二次元表現ではなく、多次元表現 [1, 2] を用いて、各 column FFT の問題サイズを小さくすることにより、L1 キャッシュ内で各 column FFT を計算することができる。ただし、三次元以上の多次元表現を用いた場合には *two-pass* アルゴリズムとすることはできず、例えば三次元表現を用いた場合には *three-pass* アルゴリズムになる。このように、多次元表現の次元数を大きくするに従って、より大きな問題サイズの FFT に対応することが可能になるが、その反面主記憶のアクセス回数が増加する。これは、ブロック six-step FFT においても性能

```
 1 COMPLEX*16 X(N1,N2),Y(N2,N1),U(N1,N2)
 2 COMPLEX*16 WORK(N2+NP,NB)
 3 DO II=1,N1,NB
 4   DO JJ=1,N2,NB
 5     DO I=II,II+NB-1
 6       DO J=JJ,JJ+NB-1
 7         WORK(J,I-II+1)=X(I,J)
 8       END DO
 9     END DO
10   END DO
11   DO I=1,NB
12     CALL IN_CACHE_FFT(WORK(1,I),N2)
13   END DO
14   DO J=1,N2
15     DO I=II,II+NB-1
16       X(I,J)=WORK(J,I-II+1)*U(I,J)
17     END DO
18   END DO
19 END DO
20 DO JJ=1,N2,NB
21   DO J=JJ,JJ+NB-1
22     CALL IN_CACHE_FFT(X(1,J),N1)
23   END DO
24   DO I=1,N1
25     DO J=JJ,JJ+NB-1
26       Y(J,I)=X(I,J)
27     END DO
28   END DO
29 END DO
```

図 6: ブロック six-step FFT アルゴリズム

はキャッシュメモリの容量に依存することを示している。

なお、out-of-place アルゴリズム（例えば Stockham アルゴリズム [5, 13]）を Step 2、4 の multicolumn FFT に用いたとしても、余分に必要となる配列の大きさは $O(\sqrt{n})$ で済む。

また、一次元 FFT の結果が転置された出力で構わなければ、Step 5 の行列の転置（図 6 では 24∼28 行目）は省略することができる。この場合、作業用の配列は $O(\sqrt{n})$ の大きさの配列 `WORK` だけで済むことが分かる。

6 In-Cache FFT アルゴリズムおよび並列化

前述の multicolumn FFT において、各 column FFT がキャッシュに載る場合の in-cache FFT には Stockham アルゴリズム [5, 13] を用いた。Stockham アルゴリズムは、Cooley-Tukey アルゴリズム [6] のように入力と出力が重ね書きできないために、入力と出力で別の配列が必要となり、その結果 Cooley-Tukey アルゴリズムの 2 倍のメモリ容量が必要になる。しかし、ビットリバース処理 [6] が不要であるという特徴がある。

ここで、基数 2 における Stockham のアルゴリズムについて説明する。

$n = 2lm$ とする。ここで l および m は 2 のべきであるものとする。l の初期値は $n/2$ とし、反

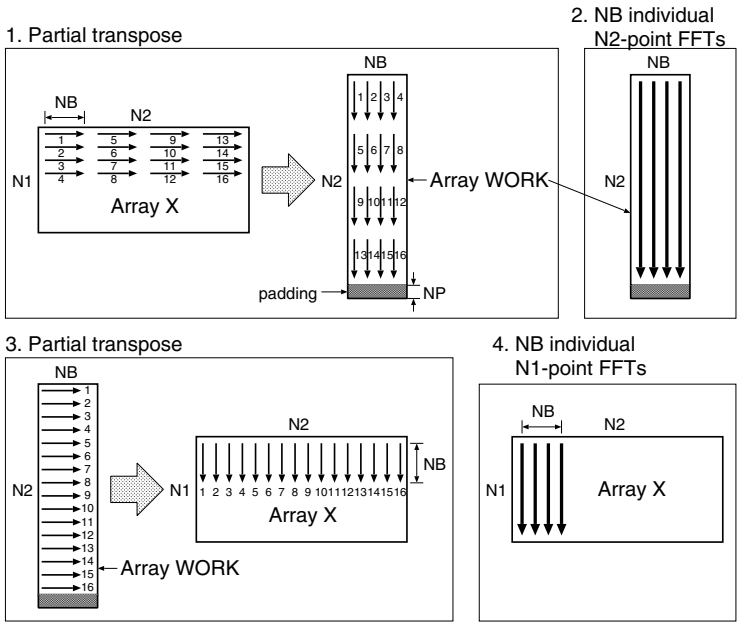

図 7: ブロック six-step FFT アルゴリズムのメモリ配置

復ごとに 2 で割っていく。m の初期値は 1 とし、反復ごとに 2 倍していく。配列 X は入力データであり、Y は出力データである。実際にインプリメントするときには、反復において、X から Y、Y から X に出力されるようにすると、メモリコピーを減らすことができる。ここで $\omega_p = e^{-2\pi i/p}$ である。

$$
\begin{aligned}
c_0 &= X(k + jm) \\
c_1 &= X(k + jm + lm) \\
Y(k + 2jm) &= c_0 + c_1 \\
Y(k + 2jm + m) &= \omega_{2l}^{j}(c_0 - c_1) \\
0 \le j < l &\qquad 0 \le k < m
\end{aligned}
$$

2 点 FFT を除く 2 べきの FFT では、基数 4 と基数 8 の組み合わせにより FFT を計算し、基数 2 の FFT カーネルを排除することにより、ロードとストア回数および演算回数を減らすことができ、より高い性能を得ることができる [14]。具体的には、$n = 2^p$ $(p \ge 2)$ 点 FFT を $n = 4^q 8^r$ $(0 \le q \le 2,\ r \ge 0)$ として計算することにより、基数 4 と基数 8 の FFT カーネルのみで $n \ge 4$ の場合に 2 べきの FFT を計算することができる。

six-step FFT において、multicolumn FFT の各列は独立であるため、並列性が高いことが知られている [3, 16]。

共有メモリ型並列計算機における並列化は以下のように行うことができる。図 5 に示した従来の six-step FFT アルゴリズムにおいては、2、7、10、15、20、23 行目の各 `DO` ループを並列化（OpenMP[11] の `!$OMP DO` ディレクティブを挿入）する。

表 1: デュアルコア Intel Xeon 5150（2.66 GHz）における FFTE 4.0 （SSE3）の性能

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00006	4128.46	0.00006	4128.80	0.00006	4141.40
2^{13}	0.00014	3912.61	0.00014	3900.81	0.00014	3925.46
2^{14}	0.00028	4030.83	0.00029	4020.14	0.00028	4036.37
2^{15}	0.00060	4121.60	0.00060	4113.43	0.00060	4106.24
2^{16}	0.00143	3676.79	0.00141	3713.05	0.00141	3717.98
2^{17}	0.00500	2228.17	0.00380	2931.55	0.00226	4921.67
2^{18}	0.01340	1761.12	0.00747	3159.97	0.00472	4995.93
2^{19}	0.02989	1666.54	0.01678	2968.24	0.01341	3715.39
2^{20}	0.06675	1570.84	0.03735	2807.18	0.03003	3491.69

また、図 6 に示したブロック six-step FFT アルゴリズムにおいては、3、20 行目の各 `DO` ループを並列化する。なお、配列 `WORK` はプライベート変数にする必要がある。MPI を用いて並列 FFT プログラムを記述する際にも、共有メモリ型並列計算機の場合と同様にブロック化することが可能である。

7 ブロック Six-Step FFT の性能

本章では、ブロック six-step FFT を用いた FFT ライブラリである FFTE（version 4.0）[1]と、多くのプロセッサにおいて最も高速な FFT ライブラリとして知られている FFTW（version 3.1.2）[2][7] との性能比較を行った結果を示す。

性能比較にあたっては、$n = 2^m$ の m を変化させて順方向 FFT を連続 10 回実行し、その平均の経過時間を測定した。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、デュアルコア Intel Xeon 5150（クロック周波数 2.66 GHz、主記憶 4 GB DDR2-SDRAM、32 KB L1 instruction cache、32 KB L1 data cache、4 MB L2 Cache、Linux 2.6.18-1.2798.fc6）を用いた。

コンパイラは Intel Fortran コンパイラ（version 9.1）および Intel C コンパイラ（version 9.1）を用い、コンパイラの最適化オプションとして、‘‘`-O3 -xP -openmp`’’ を指定した。

表 1 に FFTE（version 4.0）と、FFTW（version 3.1.2）の性能を示す。表 1 から、ブロック six-step FFT を用いている FFTE では、特に 2CPUs、4cores でデータ数 n が大きくキャッシュメモリに収まらない場合に FFTW に比べて性能が高くなっていることが分かる。

8 まとめ

本稿では、FFT におけるキャッシュ最適化方式について述べた。

[1] http://www.ffte.jp
[2] http://www.fftw.org

表 2: デュアルコア Intel Xeon 5150（2.66 GHz）における FFTW 3.1.2 の性能

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00005	5340.65	0.00005	5324.67	0.00005	5370.66
2^{13}	0.00010	5246.95	0.00010	5250.29	0.00010	5321.29
2^{14}	0.00022	5288.20	0.00022	5238.77	0.00022	5331.34
2^{15}	0.00050	4959.85	0.00050	4955.32	0.00049	4971.35
2^{16}	0.00111	4722.97	0.00110	4782.46	0.00119	4405.83
2^{17}	0.00376	2963.07	0.00400	2785.17	0.00396	2811.49
2^{18}	0.00997	2366.58	0.01011	2333.85	0.01000	2358.56
2^{19}	0.02351	2118.26	0.02288	2177.28	0.02166	2299.11
2^{20}	0.05060	2072.33	0.04003	2619.43	0.03698	2835.30

ブロック six-step FFT に基づく並列一次元 FFT では、キャッシュメモリの再利用率を高くすることにより、キャッシュミスを少なくし、その結果主記憶のアクセス回数も少なくすることができる。データがキャッシュに入り切らないような大きな問題サイズの FFT では、ブロック six-step FFT は従来の FFT アルゴリズムに比べて有利である。特に、プロセッサの演算速度と主記憶のアクセス速度との差が大きい場合に、より効果的である。

本稿で述べたブロック化の手法は、他のアプリケーションの高速化にも適用できると考えられる。

参考文献

[1] R. C. Agarwal. An efficient formulation of the mixed-radix FFT algorithm. In *Proc. International Conference on Computers, Systems and Signal Processing*, pages 769–772, 1984.

[2] R. C. Agarwal and J. W. Cooley. Vectorized mixed radix discrete Fourier transform algorithms. In *Proc. IEEE*, volume 75, pages 1283–1292, 1987.

[3] D. H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4:23–35, 1990.

[4] E. O. Brigham. *The Fast Fourier Transform and its Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[5] W. T. Cochrane, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, Jr., D. E. Nelson, C. M. Rader, and P. D. Welch. What is the fast Fourier transform? *IEEE Trans. Audio Electroacoust.*, 15:45–55, 1967.

[6] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93:216–231, 2003.

[8] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 2003.

[9] Intel Corporation. *Intel Math Kernel Library Reference Manual*, 2003.

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

[11] OpenMP. Simple, portable, scalable smp programming. http://www.openmp.org.

[12] R. C. Singleton. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Trans. Audio Electroacoust.*, 17:93–103, 1969.

[13] P. N. Swarztrauber. FFT algorithms for vector computers. *Parallel Computing*, 1:45–63, 1984.

[14] D. Takahashi. A parallel 1-D FFT algorithm for the Hitachi SR8000. *Parallel Computing*, 29(6):679–690, 2003.

[15] C. Temperton. Self-sorting mixed-radix fast Fourier transforms. *J. Comput. Phys.*, 52:1–23, 1983.

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