Short Vector SIMD 命令を用いた並列 FFT の実現と評価

高橋大介[†] 朴泰祐[†] 佐藤三久[†]

本論文では，Short Vector SIMD 命令を用いて並列一次元 FFT を実現し評価した結果について述べる．Short Vector SIMD 命令の 1 つである，Intel の Streaming SIMD Extensions 2 （SSE2）命令を用いて，FFT カーネルをベクトル化するとともに，ブロック six-step FFT アルゴリズムと組み合わせることで，性能が改善されることを示す．実現した FFT を dual Xeon PC および dual Opteron PC 上で性能評価を行った結果，2^{24} 点 FFT において，dual Xeon 3.06 GHz PC では約 809 MFLOPS，dual Opteron 1.8 GHz PC では約 927 MFLOPS の性能を得ることができた．

Implementation and Evaluation of Parallel FFT Using Short Vector SIMD Instructions

Daisuke Takahashi,[†] Taisuke Boku[†] and Mitsuhisa Sato[†]

In this paper, we propose an implementation of a parallel one-dimensional fast Fourier transform (FFT) using short vector SIMD instructions. We vectorized FFT kernels using Intel's Streaming SIMD Extensions 2 (SSE2) instructions. We show that a combination of the vectorization and block six-step FFT algorithm improves performance effectively. Performance results of one-dimensional FFTs on a dual Xeon PC and dual Opteron PC are reported. We successfully achieved performance of about 809 MFLOPS on dual Xeon 3.06 GHz PC and about 927 MFLOPS on dual Opteron 1.8 GHz PC for 2^{24}-point FFT.

1. はじめに

高速 Fourier 変換（fast Fourier transform，以下 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．FFT において大量のデータを高速に処理するために，これまでに多くの FFT アルゴリズムが提案されている．

浮動小数点演算をより高速に処理するために，最近のプロセッサでは Intel Pentium4 の SSE2 や，AMD Athlon の 3DNow!，Motorola PowerPC の AltiVec など，Short Vector SIMD 命令を搭載しているものが多い．

しかし，これらの Short Vector SIMD 命令を使ったとしても，最近のプロセッサのデータ供給能力は，キャッシュに頼っているのが現状であり，メモリアクセスの最適化もあわせて行う必要がある．

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収ま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．

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，Short Vector SIMD 命令を搭載したプロセッサで FFT を計算する際には，Short Vector SIMD 命令を有効に活用しつつ，かつキャッシュミスの回数を減らすということが性能を引き出すうえでの大きな鍵となる．

これまでに，Short Vector SIMD 命令を用いた FFT の実装[2)~6)] がいくつか行われているが，これらの FFT は，データがキャッシュに収まるような場合を想定しているものが多く，キャッシュに収まりきらないような大規模なデータを扱う場合には必ずしも性能が発揮できていないのが現状である．

多くのプロセッサにおいて高速な FFT ライブラリとして知られている FFTW [6),7)] の最新版である Version 3.0.1 では，Short Vector SIMD 命令である SSE，SSE2，3DNow!，AltiVec の各命令に対応しているとともに，プロセッサ内でデータを再帰的に二分木状に分割することによって，最終的に小さな点数の DFT に帰着させるというアプローチをとることで，メモリ

† 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

アクセスの局所性を引き出している．

一方，今回実現した並列 FFT では，SSE2 のベクトル命令を用いて FFT カーネル部分の性能を向上させているのは FFTW と同様であるが，ブロック six-step FFT アルゴリズム[8] を用いて，さらなるキャッシュの有効利用を図っているのが特徴である．

実現した並列 FFT を dual Xeon PC および dual Opteron PC 上で，性能評価を行う．なお，本論文では倍精度浮動小数点演算の場合について取り扱うことにする．

以下，2 章で Short Vector SIMD 命令の 1 つである，Intel の SSE2 命令について説明する．3 章で FFT カーネルのベクトル化について述べる．4 章でブロック six-step FFT アルゴリズムについて述べる．5 章でデータがキャッシュに載る場合の FFT アルゴリズムおよび並列化について述べる．6 章で性能評価の結果を示す．最後の 7 章はまとめである．

2. Intel SSE2 命令

SSE2 命令[9] とは，Intel Pentium4 から導入された，x87 命令に代わる新しい演算命令である．128bit 長のデータに対して，SIMD 処理を行うことができる．Intel Pentium4 および Xeon プロセッサには 128 bit の XMM レジスタが `XMM0`〜`XMM7` の 8 個，x86-64 アーキテクチャの AMD Opteron プロセッサには 128 bit の XMM レジスタが `XMM0`〜`XMM15` の 16 個搭載されている．

SSE2 命令には，ベクトル命令とスカラ命令がある．SSE2 のベクトル命令では，XMM レジスタの全 128 bit を使うことができ，1 つの命令で複数の要素に対して演算を行うことができるため，多くの場合，SSE2 のスカラ命令よりも高い性能が得られる．

SSE2 のベクトル命令を用いることによって，32 bit の単精度浮動小数点演算ではベクトル長が 4 の，64 bit の倍精度浮動小数点演算ではベクトル長が 2 のベクトル演算（加減乗除，平方根，論理演算）を行うことができる．

2.1 SSE2 命令の利用方法

SSE2 のベクトル命令の利用方法としては，大きく分けて 3 つあげられる．

(1) コンパイラによりベクトル化する方法
(2) SSE2 組み込み関数を使用する方法
(3) アセンブラを使用する方法

(1)〜(3) の順にコーディングが複雑になるが，性能という観点からは有利になる．

まず，(1) のコンパイラによりベクトル化する方法であるが，Intel Compiler や PGI Compiler では，自動ベクトル化を行って SSE2 のベクトル命令を生成することが可能となっている．ただし，ベクトル化の条件が限られている場合があることや，ベクトル化のためのディレクティブを用いる必要がある場合もある．

最も高い性能を得るという観点からは，(3) のアセンブラを使用するのが有利であると考えられるが，その反面，アセンブラで書かれた部分はコンパイラによる最適化の対象から外れてしまうという欠点がある．

(2) の SSE2 組み込み関数は，SSE2 命令を使いやすくするためにインラインアセンブラをマクロ化したものであり，API が Intel から提供されている[10]．SSE2 組み込み関数を用いたプログラムは，アセンブラで書かれたプログラムに近い性能が得られること，コンパイラによりレジスタの割当てと命令のスケジューリングを可能な限り最適化することができるという利点がある．また，これらの組み込み関数には，コンパイラがサポートしているすべてのインテル・アーキテクチャ・ベースのプロセッサの間で移植性がある．

SSE2 組み込み関数を使用すると，アセンブラを使用した場合に比べて，プログラムの作成と保守のコストを大幅に軽減できることから，Intel は SSE2 組み込み関数の使用を勧めている．

また，この SSE2 組み込み関数は Intel Compiler だけでなく，gcc 3.3 以降でも使用が可能になっている．

x86-64 アーキテクチャ向けの gcc 3.3 以降でも，この SSE2 組み込み関数がサポートされており，Intel Pentium4 および Xeon と同じプログラムを用いても，`XMM0`〜`XMM15` の 16 個の XMM レジスタを使用するようなオブジェクトを出力できることが確認できたため，今回は SSE2 組み込み関数を用いてコーディングを行った．

2.2 SSE2 命令による複素数演算

SSE2 命令を用いて倍精度複素数演算を行うにあたって，用いるデータ構造としては次の 2 つが考えられる．

(1) 複素数の実部と虚部を別の配列に格納する方法
(2) 複素数の実部と虚部を交互に格納する方法

(1) の方法では，倍精度の場合にベクトル長が 2 となるが，実部と虚部でそれぞれ 1 個ずつ XMM レジスタを用いる必要があるため，8 個の XMM レジスタでは 4 個の倍精度複素数配列しか格納できない．また，Fortran の double complex 型配列を用いたプログラムをベクトル化する場合には，データ構造を大きく変更する必要がある．

一方，(2) の方法は，データ構造が Fortran の double complex 型配列と同じであり，倍精度の場合には

```
#include <emmintrin.h>

static __inline __m128d ZMULADD(__m128d a, __m128d b, __m128d c)
{
  __m128d br, bi;

  br = _mm_unpacklo_pd(b, b);            /* br = [b.r b.r] */
  br = _mm_mul_pd(br, c);                /* br = [b.r*c.r b.r*c.i] */
  a = _mm_add_pd(a, br);                 /* a = [a.r+b.r*c.r a.i+b.r*c.i] */
  bi = _mm_unpackhi_pd(b, b);            /* bi = [b.i b.i] */
  bi = _mm_xor_pd(bi, _mm_set_sd(-0.0)); /* bi = [-b.i b.i] */
  c = _mm_shuffle_pd(c, c, 1);           /* c = [c.i c.r] */
  bi = _mm_mul_pd(bi, c);                /* bi = [-b.i*c.i b.i*c.r] */

  return _mm_add_pd(a, bi);              /* [a.r+b.r*c.r-b.i*c.i a.i+b.r*c.i+b.i*c.r] */
}
```

図 1 倍精度複素数の積和演算（$a + b \times c$）を SSE2 組み込み関数で記述した例
Fig. 1 An example of double-precision complex multiply-add using SSE2 intrinsics.

```
typedef struct { double r, i; } doublecomplex;

void zaxpy(int n, doublecomplex a, doublecomplex *x,
           doublecomplex *y)
{
  int i;

  if (a.r == 0.0 && a.i == 0.0) return;

#pragma unroll(8)
#pragma vector aligned
  for (i = 0; i < n; i++) {
    y[i].r += a.r * x[i].r - a.i * x[i].i,
    y[i].i += a.r * x[i].i + a.i * x[i].r;
  }
}
```

図 2 C で記述した倍精度複素ベクトルの積和（ZAXPY）
Fig. 2 ZAXPY written by C.

ベクトル長が 1 となるが，8 個の XMM レジスタで 8 個の倍精度複素数配列を格納できるという利点がある．また，(2) の方法では実部と虚部が交互に格納されているため，メモリアクセスの局所性という観点からは (1) の方法に比べて有利となるとともに，ベクトル長が 1 となるので，ループの端数処理が不要であるという利点がある．

そこで，今回は (2) の方法を採用することにした．ただし，(2) の方法で複素数の乗算を行う際には，実部と虚部で演算が異なるため，XMM レジスタの上位と下位の内容を入れ替えたり，符号を付け替えたりする必要がある．**図 1** に，倍精度複素数の積和演算（$a + b \times c$）を行うインライン関数 `ZMULADD` を SSE2 組み込み関数で記述した例を示す．

なお，図 1 のプログラムで，`__m128d` は SSE2 組み込み関数のオペランドとして倍精度浮動小数点 SIMD 演算に使用される 128 bit のオブジェクトを表す新しいデータ・タイプである．

```
#include <emmintrin.h>

typedef struct { double r, i; } doublecomplex;
__m128d ZMULADD(__m128d a, __m128d b, __m128d c);

void zaxpy(int n, doublecomplex a, doublecomplex *x,
           doublecomplex *y)
{
  int i;
  __m128d a0;

  if (a.r == 0.0 && a.i == 0.0) return;

  a0 = _mm_loadu_pd(&a);
#pragma unroll(8)
  for (i = 0; i < n; i++)
    _mm_store_pd(&y[i], ZMULADD(_mm_load_pd(&y[i]), a0,
                                _mm_load_pd(&x[i])));
}
```

図 3 SSE2 組み込み関数で記述した倍精度複素ベクトルの積和（ZAXPY）
Fig. 3 ZAXPY using SSE2 intrinsics.

2.3 ZAXPY による性能比較

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，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，2.1 節で述べた各コーディング手法について評価を行った．

ZAXPY は，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．

以下の 5 つの手法について，性能比較を行った．

(1) **図 3** のプログラムのように SSE2 組み込み関数を用いた場合（with SSE2 (intrinsic)）
(2) **図 2** のプログラムをコンパイラによりベクトル化した場合（with SSE2 (vector)）

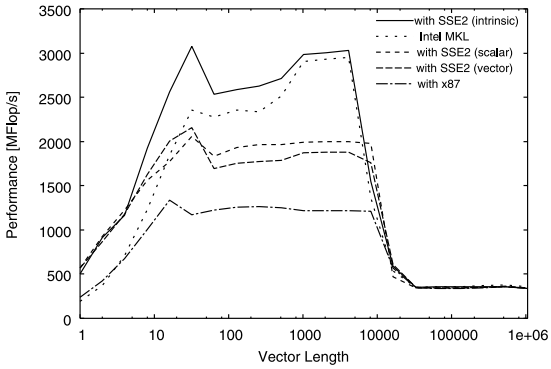

図 4 ZAXPY の性能（Intel Xeon 3.06 GHz）
Fig. 4 Performance of ZAXPY (Intel Xeon 3.06 GHz).

(3) 図 2 のプログラムでベクトル化を行わず☆，SSE2 のスカラー命令を用いた場合（with SSE2 (scalar)）
(4) 図 2 のプログラムで SSE2 命令を生成せず，x87 命令を用いた場合（with x87）
(5) Intel が提供している MKL（Math Kernel Library，Version 6.1.1）[11] の BLAS の ZAXPY ルーチンを用いた場合（Intel MKL）

性能評価に使用した計算機環境は，以下のとおりである．

- プロセッサ：Intel Xeon 3.06 GHz（FSB 533 MHz, 512 KB L2 cache，PC2100 DDR-SDRAM）
- OS：Linux 2.4.20smp
- コンパイラ：Intel C++ Compiler 8.0（`icc`）
- コンパイルオプション：`icc -O3 -xW`（with SSE2, Intel MKL），`icc -O3`（with x87）

ここで，コンパイルオプションの “`-xW`” は SSE2 命令の使用を指示するオプションである．なお，図 2 および図 3 のプログラムでは，性能向上のために `#pragma unroll(8)` のディレクティブを用いて 8 回アンローリングを行っている．

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 組み込み関数を使ったプログラム（with SSE2 (intrinsic)）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．これは，ZAXPY では 1 回の iteration につき倍精度実数の加算，乗算がそれぞれ 4 回であるのに対し，load が 4 回，store が 2 回必要であることが主な原因であると考えられる．

図 2 のプログラムをコンパイラによりベクトル化を行った場合のオブジェクトを見ると，64 bit の load 命令が `x[i].r` と `x[i].i` や `y[i].r` と `y[i].i` のそれぞれに対して 1 回ずつ発行され，`x[i].r` と `x[i+1].r` のペアが 1 個の XMM レジスタに格納されている．64 bit の store 命令も同様に `y[i].r` と `y[i].i` に対してそれぞれ 1 回ずつ発行されている．

それに対して図 3 のように，SSE2 組み込み関数で記述した場合，実部と虚部のペアを 1 回の 128 bit の load および store 命令で行うことが可能である．これが，SSE2 組み込み関数を使ったプログラムが最も高速であった理由である．

また，図 4 からも分かるように，図 3 の SSE2 組み込み関数を用いたプログラムは，Intel が提供している MKL の BLAS の ZAXPY ルーチンよりも高速である．このことから，SSE2 組み込み関数を用いた手法は十分高速であるといえる．

なお，図 1 から分かるように，たとえ SSE2 組み込み関数で記述した場合でも，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性能を得るうえで不可欠であることを示している．

3. FFT カーネルのベクトル化

FFT カーネルのベクトル化に際しては，図 2 の `ZMULADD` と同様に，**図 5** に示すような倍精度複素数の乗算を行うインライン関数 `ZMUL` を作成した．

倍精度複素数の加算，減算や実数倍の演算は，SSE2 組み込み関数の `_mm_add_pd`，`_mm_sub_pd`，`_mm_mul_pd` を用いることで，実部と虚部の計算を 1 命令で行うことができる．このように，今回行った実装方法では，ベクトル化といっても，倍精度複素数データに対するベクトル長は 1 となっているので，実際には SSE2 のベクトル命令を倍精度複素数演算のスカラー命令のように使っていることになる．

図 6 に示すような C で記述されたプログラムを，**図 7** のようにベクトル化を行った．SSE2 組み込み関数を使うとともに，倍精度複素数の乗算をインライン関数化することで，見通し良くベクトル化を行えていることが分かる．今回の FFT の実装では，基数 2 の FFT カーネルだけでなく，基数 3，4，5，8 の FFT カーネルも含んでおり，ソースプログラムの変更は数百行に及んだ．

4. ブロック Six-Step FFT アルゴリズム

この章では，本論文で用いたブロック six-step FFT

☆ `#pragma vector aligned` をコメントアウトする．

```
#include <emmintrin.h>

static __inline __m128d ZMUL(__m128d a, __m128d b)
{
  __m128d ar, ai;

  ar = _mm_unpacklo_pd(a, a);            /* ar = [a.r a.r] */
  ar = _mm_mul_pd(ar, b);                /* ar = [a.r*b.r a.r*b.i] */
  ai = _mm_unpackhi_pd(a, a);            /* ai = [a.i a.i] */
  ai = _mm_xor_pd(ai, _mm_set_sd(-0.0)); /* ai = [-a.i a.i] */
  b = _mm_shuffle_pd(b, b, 1);           /* b = [b.i b.r] */
  ai = _mm_mul_pd(ai, b);                /* ai = [-a.i*b.i a.i*b.r] */

  return _mm_add_pd(ar, ai);             /* [a.r*b.r-a.i*b.i a.r*b.i+a.i*b.r] */
}
```

図 5 倍精度複素数の積（$a \times b$）を SSE2 組み込み関数で記述した例
Fig. 5 An example of double-precision complex multiplication using SSE2 intrinsics.

```
void fft(double *a, double *b, double *w, int m, int l)
{
  int i, i0, i1, i2, i3, j;
  double u, v, wi, wr;

  for (j = 0; j < l; j++) {
    wr = w[j << 1];
    wi = w[j << 1 + 1];
    for (i = 0; i < m; i++) {
      i0 = (i << 1) + (j * m << 1);
      i1 = i0 + (m * l << 1);
      i2 = (i << 1) + (j * m << 2);
      i3 = i2 + (m << 1);
      u = a[i0] - a[i1];
      v = a[i0 + 1] - a[i1 + 1];
      b[i2] = a[i0] + a[i1];
      b[i2 + 1] = a[i0 + 1] + a[i1 + 1];
      b[i3] = wr * u - wi * v;
      b[i3 + 1] = wr * v + wi * u;
    }
  }
}
```

図 6 基数 2 の FFT カーネル
Fig. 6 An example of a radix-2 FFT kernel.

アルゴリズム[8] について説明する．

離散 Fourier 変換（discrete Fourier transform）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．

$n = n_1 \times n_2$ と分解できるものとすると，式 (1) における j および k は，

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2 \qquad (2)$$

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．

```
#include <emmintrin.h>

__m128d ZMUL(__m128d a, __m128d b);

void fft(double *a, double *b, double *w, int m, int l)
{
  int i, i0, i1, i2, i3, j;
  __m128d t0, t1, w0;

  for (j = 0; j < l; j++) {
    w0 = _mm_load_pd(&w[j << 1]);
    for (i = 0; i < m; i++) {
      i0 = (i << 1) + (j * m << 1);
      i1 = i0 + (m * l << 1);
      i2 = (i << 1) + (j * m << 2);
      i3 = i2 + (m << 1);
      t0 = _mm_load_pd(&a[i0]);
      t1 = _mm_load_pd(&a[i1]);
      _mm_store_pd(&b[i2], _mm_add_pd(t0, t1));
      _mm_store_pd(&b[i3], ZMUL(w0, _mm_sub_pd(t0, t1)));
    }
  }
}
```

図 7 基数 2 の FFT カーネルを SSE2 組込み関数を使ってベクトル化した例
Fig. 7 An example of a vectorized radix-2 FFT kernel using SSE2 intrinsic.

$$x_j = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1 \qquad (3)$$

$$y_k = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1 \qquad (4)$$

したがって，式 (1) は式 (5) のように変形できる．

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1} \qquad (5)$$

式 (5) から，six-step FFT アルゴリズム[12),13)] が導

かれる．この 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[8] 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．

図 8 にブロック six-step FFT アルゴリズムの疑似コードを示す．

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．

提案するブロック six-step FFT アルゴリズムは，いわゆる *two-pass* アルゴリズム[12),13)] となる．つまり，提案するブロック 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．

このブロック six-step FFT アルゴリズムでは Step 2 および Step 4 の各 column FFT が L1 キャッシュに収まると仮定しているが，これが成り立つ条件について検討する．

$n = n_1 n_2$ とすると，n 点 FFT をブロック six-step FFT アルゴリズムを用いて計算する際に，Step 2 および Step 4 ではそれぞれ n_2 点および n_1 点の column FFT を計算することになる．この column FFT に out-of-place アルゴリズム（たとえば Stockham アルゴリズム[14)]）を用いるとすると，入力と出力でそれぞれ $\max(n_1, n_2)$ の領域が必要になる．また，三角関数のテーブルの領域については実装により異なるが，今回の実装では $0.5 \max(n_1, n_2)$ である．簡単のために，$n_1 = n_2 = \sqrt{n}$ と仮定すると，倍精度複素数で計算する場合，各 column FFT が L1 キャッシュに収まる条件は，

L1 キャッシュの容量 (byte) $\leq 16 \times 2.5\sqrt{n}$

となる．

本論文で評価に用いた Xeon プロセッサおよび Opteron プロセッサでは，L1 キャッシュがそれぞれ 8 KB および 64 KB であるので，上記の条件が成り立つ n は，Xeon プロセッサでは $n \leq 41943$，Opteron プロセッサでは $n \leq 2684354$ となる．

しかし，図 4 から分かるように，Xeon プロセッサでは L1 キャッシュから外れても，L2 キャッシュに収まる場合には高い性能が維持できることから，Xeon プロセッサにおけるブロック six-step FFT アルゴリズムでは L1 キャッシュの容量は事実上 L2 キャッシュの容量と同じと見なしてよい．

また，ブロック six-step FFT アルゴリズムでは

```
 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 8 ブロック six-step FFT アルゴリズム
Fig. 8 A block six-step FFT algorithm.

$\sqrt{n} \times n_b$ の複素数配列が L2 キャッシュに収まることを仮定している．ここで，ある n が与えられた場合に，最適なブロックサイズ n_b は L2 キャッシュの容量に依存する．

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か，または $n_b = 1$ としても $\max(n_1, n_2) \times n_b$ の複素数配列が L2 キャッシュに収まらないことも考えられる．このような場合は二次元表現ではなく，多次元表現[15] 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，ブロックサイズ n_b も大きくすることができる．三次元表現を用いたブロック FFT アルゴリズムとして，ブロック nine-step FFT アルゴリズム[16] を適用することができる．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，たと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．

5. 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[14] を用いた．

2 点 FFT を除く 2 べきの FFT では，基数 4 と基数 8 の組合せにより 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[17]．具体的には，$n = 2^p$ $(p \geq 2)$ 点 FFT を $n = 4^q 8^r$ $(0 \leq q \leq 2,\ r \geq 0)$ 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 $n \geq 4$ 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[12),13]．図 8 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 OpenMP のディレクティブを用いて並列化した．なお，配列 `WORK` はプライベート変数にする必要がある．

6. 性能評価

性能評価にあたっては，ブロック six-step FFT を SSE2 組み込み関数を用いて今回実現した FFT ライブラリである FFTE（Version 3.2）☆ と，FFTW（Version 3.0.1）[6]，そして Intel が提供している MKL（Math Kernel Library，Version 6.1.1），AMD が提供している ACML（AMD Core Math Library，Version 1.5）[18] の性能を比較した．

表 1 評価環境
Table 1 Specification of machines.

Platform	Intel Xeon PC	AMD Opteron PC
Number of CPUs	2 or 4 (with HT)	2
CPU Type	Xeon 3.06 GHz	Opteron 1.8 GHz
L1 Cache	I-Cache: 12 K uops D-Cache: 8 KB	I-Cache: 64 KB D-Cache: 64 KB
L2 Cache	512 KB	1 MB
Main Memory	PC2100 DDR-SDRAM 1 GB	PC2700 DDR-SDRAM 2 GB
OS	Linux 2.4.20smp	Linux 2.4.19smp

$n = 2^m$ の m および CPU 数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．評価環境を**表 1** に示す．

プログラムは SSE2 組み込み関数を用いたカーネル部分は C で，その他の部分は Fortran で記述し，並列化には OpenMP を用いた．

6.1 dual Xeon PC による測定結果

dual Xeon PC においては，FFTE と FFTW，そして MKL の性能の比較を行った．

FFTE および MKL は，Fortran のプログラムから呼び出すとともに，FFTW は C のプログラムから呼び出すことで性能を測定した．

コンパイラは Intel C++ Compiler（`icc`，Version 8.0）および Intel Fortran Compiler（`ifort`，Version 8.0）を用いた．最適化オプションとして，FFTE では “`icc -O3 -xW -fno-alias`” および “`ifort -O3 -openmp -xW -fno-alias`” を用いた．FFTW では，“`icc -O3 -xW`” のオプションを用い，MKL では “`ifort -O3 -xW`” のオプションを用いた．

dual Xeon PC における性能評価では，Intel のハイパー・スレッディング・テクノロジ[19] を用いて，物理 CPU は 2 個であるものの，論理 CPU が 4 個あるものとして評価を行っている．したがって，以下では論理 CPU の個数で示すことにする．なお，2 CPUs と書いてあるのは，論理 CPU，物理 CPU ともに 2 個

☆ `http://www.ffte.jp`

表 2 FFTE 3.2 の一次元 FFT の性能（dual Xeon 3.06 GHz PC，with SSE2）
Table 2 Performance of FFTE 3.2 one-dimensional FFT on dual Xeon 3.06 GHz PC (with SSE2).

n	1 CPU		2 CPUs		4 CPUs	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00013	1907.85	0.00013	1907.51	0.00013	1907.79
2^{13}	0.00042	1270.14	0.00026	2019.57	0.00033	1624.52
2^{14}	0.00102	1122.47	0.00065	1760.20	0.00081	1411.93
2^{15}	0.00298	824.13	0.00199	1237.05	0.00233	1056.54
2^{16}	0.00684	766.80	0.00511	1026.15	0.00569	922.21
2^{17}	0.01456	765.37	0.01079	1032.41	0.01146	971.80
2^{18}	0.03031	778.27	0.02263	1042.44	0.02356	1001.27
2^{19}	0.06013	828.39	0.04350	1144.89	0.04721	1054.93
2^{20}	0.12145	863.37	0.08588	1220.94	0.09597	1092.58
2^{21}	0.25631	859.11	0.19755	1114.65	0.21158	1040.73
2^{22}	0.53133	868.33	0.41048	1124.00	0.55851	826.08
2^{23}	1.32883	725.97	0.93947	1026.85	1.26062	765.25
2^{24}	3.33491	603.69	2.48923	808.79	2.87893	699.31

表 3 FFTE 3.2 の一次元 FFT の性能（dual Xeon 3.06 GHz PC，with x87）
Table 3 Performance of FFTE 3.2 one-dimensional FFT on dual Xeon 3.06 GHz PC (with x87).

n	1 CPU		2 CPUs		4 CPUs	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00018	1337.31	0.00018	1337.62	0.00018	1338.06
2^{13}	0.00060	891.40	0.00038	1395.01	0.00045	1178.42
2^{14}	0.00141	812.97	0.00087	1322.84	0.00115	997.86
2^{15}	0.00444	553.35	0.00327	751.02	0.00370	664.50
2^{16}	0.01168	449.05	0.01000	524.08	0.01037	505.81
2^{17}	0.02390	466.15	0.02016	552.53	0.02088	533.62
2^{18}	0.05067	465.62	0.04249	555.20	0.04217	559.45
2^{19}	0.10033	496.43	0.08124	613.06	0.08462	588.63
2^{20}	0.20257	517.63	0.16153	649.17	0.16925	619.55
2^{21}	0.41311	533.03	0.34623	636.00	0.35822	614.71
2^{22}	0.85553	539.28	0.72604	635.47	0.85342	540.61
2^{23}	1.98786	485.29	1.48401	650.06	1.79462	537.55
2^{24}	4.59607	438.04	3.69224	545.27	3.90712	515.28

を用いた評価となっている．

FFTE（with SSE2），FFTE（with x87），FFTW および MKL の性能を**表 2**，**表 3**，**表 4** および**表 5** にそれぞれ示す．ここで，実行時間の単位は秒であり，$n = 2^m$ 点 FFT の MFLOPS 値は $5n \log_2 n$ より算出している．

表 2 および表 4 から，$2^{15} \leq n \leq 2^{23}$（1 CPU），$2^{14} \leq n \leq 2^{23}$（2 CPU），$2^{14} \leq n \leq 2^{15}$ および $2^{18} \leq n \leq 2^{23}$（4 CPU）において，FFTE が FFTW に比べて高い性能が得られていることが分かる．また，表 2 および表 5 から，MKL と比べた場合でも，$2^{16} \leq n \leq 2^{23}$（1 CPU），$n \geq 2^{16}$（2 CPU，4 CPU）において，FFTE が性能が高くなっていることが分かる．

FFTW では，自動チューニングの手法を用いて高速化を図っており，データがキャッシュに入るような場合については高い性能を発揮できていると考えられる．それに対し，FFTE ではキャッシュブロッキングを行っており，すべてのデータがキャッシュに入りきらない場合についても，各 multicolumn FFT はキャッシュに入っているために，ある程度高い性能が維持できていることが分かる．そのために，表 2 および表 3 から，$n = 2^{24}$（1 CPU）でも SSE2 のベクトル命令を用いた場合，x87 命令を用いた場合に比べて約 38%高速であることが分かる．

なお，表 2 において，$n \geq 2^{23}$ で性能が低下しているのは，L2 キャッシュミスが生じているのが原因であると考えられる．

表 2 では 2 CPU の場合の 1 CPU に対する速度向上は，並列化を行っていない $n = 2^{12}$ を除いて，約 1.34 倍から約 1.59 倍となっている．$n = 2^{13}$ においては，1 CPU ではデータが L2 キャッシュから外れるが，2 CPU ではデータが L2 キャッシュに収まるので，並列化の効果が高く出ている．ところが，2 CPU の場合でもデータが L2 キャッシュから外れると，FFT の性能は主記憶からキャッシュへのデータ転送の速度に

表 4 FFTW 3.0.1 の一次元 FFT の性能（dual Xeon 3.06 GHz PC）
Table 4 Performance of FFTW 3.0.1 one-dimensional FFT on dual Xeon 3.06 GHz PC.

n	1 CPU		2 CPUs		4 CPUs	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00008	2980.21	0.00008	2928.67	0.00008	2980.56
2^{13}	0.00020	2664.84	0.00021	2580.55	0.00019	2746.77
2^{14}	0.00088	1307.32	0.00100	1152.81	0.00095	1207.64
2^{15}	0.00276	889.69	0.00263	934.51	0.00250	983.23
2^{16}	0.00663	791.28	0.00615	852.10	0.00558	939.43
2^{17}	0.01525	730.49	0.01224	910.34	0.01139	977.82
2^{18}	0.03255	724.82	0.03373	699.45	0.02456	960.63
2^{19}	0.07234	688.54	0.07492	664.82	0.06410	777.02
2^{20}	0.16086	651.85	0.12872	814.62	0.13805	759.54
2^{21}	0.35453	621.11	0.29581	744.41	0.29562	744.87
2^{22}	0.75349	612.32	0.60865	758.03	0.61773	746.89
2^{23}	1.60795	599.95	1.28141	752.84	1.30157	741.18
2^{24}	3.64585	552.21	3.42129	588.45	2.76877	727.13

表 5 MKL 6.1.1 の一次元 FFT の性能（dual Xeon 3.06 GHz PC）
Table 5 Performance of MKL 6.1.1 one-dimensional FFT on dual Xeon 3.06 GHz PC.

n	1 CPU		2 CPUs		4 CPUs	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00010	2527.64	0.00010	2549.59	0.00012	1988.40
2^{13}	0.00026	2049.90	0.00023	2344.68	0.00029	1811.11
2^{14}	0.00057	2023.92	0.00048	2407.57	0.00057	2024.42
2^{15}	0.00202	1214.12	0.00150	1634.04	0.00178	1380.98
2^{16}	0.00665	788.06	0.00529	990.30	0.00603	869.99
2^{17}	0.01446	770.26	0.01514	735.89	0.01412	789.20
2^{18}	0.03393	695.42	0.03151	748.84	0.03548	664.89
2^{19}	0.06858	726.24	0.07478	666.01	0.07127	698.87
2^{20}	0.15563	673.78	0.14937	701.99	0.16745	626.21
2^{21}	0.31766	693.19	0.34792	632.91	0.33710	653.22
2^{22}	0.71024	649.61	0.69139	667.32	0.77799	593.03
2^{23}	1.42854	675.30	1.53512	628.41	1.55147	621.79
2^{24}	3.19185	630.75	3.07768	654.15	3.51328	573.04

大きく依存するため，高い性能が発揮できず，その結果良い速度向上が得られなかったと推察される．

なお，ハイパー・スレッディングの効果があったのは，FFTW では $2^{12} \leq n \leq 2^{19}$ の場合，MKL では $n = 2^{17}$ および $n = 2^{19}$ の場合であった．FFTE については，ハイパー・スレッディングの効果が確認できなかった．原因としては演算器の利用率が高いことや，複数スレッドによるメモリアクセスの競合が考えられるが，これについては現在調査中である．

6.2 dual Opteron PC による測定結果

dual Opteron PC においては，FFTE と FFTW，そして ACML の性能の比較を行った．

FFTE および ACML は Fortran のプログラムから呼び出すとともに，FFTW は C のプログラムから呼び出すことで性能を測定した．

x86-64 アーキテクチャ用の PGI コンパイラは，64 bit 版であるとともに，`XMM0`〜`XMM15` の 16 個の XMM レジスタを有効に利用したオブジェクトを出力する．しかし，SSE2 組み込み関数には対応していないので，今回は SSE2 組み込み関数を使った C のソースコードに対して gcc 3.3.3 を用いるとともに，Fortran のソースは PGI Workstation Fortran 90 Compiler（`pgf90`，Version 5.1-3）を用いて 64 bit モードでコンパイルした．一方，x87 命令による実行に際しては，`pgf90` が 64 bit モードでは x87 命令を出力できなかったために，逐次 FFT カーネル部分のみ g77 3.3.3 を用いて x87 命令を出力するコンパイラオプション “`-mfpmath=387`” を付けて 64 bit モードでコンパイルし，その他のルーチンを `pgf90` を用いてコンパイルしている．

最適化オプションとして，FFTE では “`gcc -O3 -fomit-frame-pointer`”，“`g77 -O3 -fomit-frame-pointer -mfpmath=387`” および “`pgf90 -fast -mp`” を用いた．FFTW では “`gcc -O3 -fomit-frame-pointer -fstrict-aliasing -mpreferred-stack-boundary=4`” のオプションを用い，ACML では

表 6 FFTE 3.2 の一次元 FFT の性能（dual Opteron 1.8 GHz PC, with SSE2）
Table 6 Performance of FFTE 3.2 one-dimensional FFT on dual Opteron 1.8 GHz PC (with SSE2).

n	1 CPU		2 CPUs	
	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00022	1097.16	0.00022	1096.66
2^{13}	0.00078	686.12	0.00052	1017.62
2^{14}	0.00191	601.37	0.00117	984.34
2^{15}	0.00355	691.78	0.00217	1135.03
2^{16}	0.00751	698.25	0.00432	1212.69
2^{17}	0.01711	651.06	0.00936	1190.86
2^{18}	0.03968	594.56	0.02135	1105.11
2^{19}	0.09349	532.73	0.04922	1012.01
2^{20}	0.21049	498.16	0.11020	951.48
2^{21}	0.43275	508.84	0.22468	980.07
2^{22}	0.83379	553.34	0.43323	1064.96
2^{23}	1.83429	525.92	0.95540	1009.72
2^{24}	4.16131	483.81	2.17202	926.91

表 7 FFTE 3.2 の一次元 FFT の性能（dual Opteron 1.8 GHz PC, with x87）
Table 7 Performance of FFTE 3.2 one-dimensional FFT on dual Opteron 1.8 GHz PC (with x87).

n	1 CPU		2 CPUs	
	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00023	1071.50	0.00023	1071.77
2^{13}	0.00063	844.17	0.00045	1182.97
2^{14}	0.00134	853.72	0.00091	1259.57
2^{15}	0.00299	821.57	0.00185	1328.23
2^{16}	0.00742	706.17	0.00420	1247.39
2^{17}	0.01676	664.83	0.00908	1226.57
2^{18}	0.03910	603.46	0.02086	1130.91
2^{19}	0.08420	591.55	0.04446	1120.32
2^{20}	0.17550	597.47	0.09255	1132.96
2^{21}	0.39187	561.93	0.20453	1076.62
2^{22}	0.81167	568.42	0.42263	1091.67
2^{23}	1.81559	531.34	0.94386	1022.07
2^{24}	4.13064	487.40	2.16151	931.42

“`pgf90 -fastsse`” を指定した．

FFTE（with SSE2），FFTE（with x87），FFTW および ACML の性能を**表 6**，**表 7**，**表 8** および**表 9** にそれぞれ示す．ここで，実行時間の単位は秒であり，$n = 2^m$ 点 FFT の MFLOPS 値は $5n \log_2 n$ より算出している．なお ACML では，1 CPU での実行しかサポートしていないため，表 9 は 1 CPU の場合のみの性能を示している．

表 6 および表 8 から，FFTE と FFTW と比べた場合には，FFTW が高速であった．表 6 および表 9 から，FFTE が，ACML に比べて $n = 2^{16}$ および $n = 2^{17}$ の場合に高い性能が得られていることが分かる．

また，表 6 および表 7 から，Opteron では SSE2 のベクトル命令を用いた場合よりも，x87 命令を用いた方が高い性能が得られていることが分かる．原因としては，Opteron における SSE2 のベクトル命令の実行速度が，x87 命令に比べて遅い場合があるということが考えられる．

表 8 FFTW 3.0.1 の一次元 FFT の性能（dual Opteron 1.8 GHz PC）
Table 8 Performance of FFTW 3.0.1 one-dimensional FFT on dual Opteron 1.8 GHz PC.

n	1 CPU		2 CPUs	
	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00015	1664.13	0.00015	1661.67
2^{13}	0.00042	1260.39	0.00038	1413.97
2^{14}	0.00096	1196.13	0.00077	1484.56
2^{15}	0.00227	1081.90	0.00166	1485.31
2^{16}	0.00568	923.54	0.00388	1352.61
2^{17}	0.01335	834.44	0.00815	1367.56
2^{18}	0.03242	727.66	0.01829	1290.06
2^{19}	0.06891	722.78	0.04076	1222.12
2^{20}	0.15692	668.21	0.08688	1206.90
2^{21}	0.35182	625.89	0.18721	1176.25
2^{22}	0.72557	635.88	0.39930	1155.45
2^{23}	1.52787	631.39	0.85720	1125.40
2^{24}	3.29620	610.78	1.80689	1114.22

表 9 ACML 1.5 の一次元 FFT の性能（dual Opteron 1.8 GHz PC）
Table 9 Performance of ACML 1.5 one-dimensional FFT on dual Opteron 1.8 GHz PC.

n	Time	MFLOPS
2^{12}	0.00017	1450.23
2^{13}	0.00050	1072.97
2^{14}	0.00105	1090.21
2^{15}	0.00328	749.54
2^{16}	0.00752	697.06
2^{17}	0.01765	631.21
2^{18}	0.03860	611.19
2^{19}	0.07863	633.46
2^{20}	0.17788	589.49
2^{21}	0.37355	589.48
2^{22}	0.74513	619.18
2^{23}	1.82019	529.99
2^{24}	3.95176	509.46

7. ま と め

本論文では，Short Vector SIMD 命令を用いた並列 FFT の実現と評価について述べた．Short Vector SIMD 命令の 1 つである，Intel の SSE2 命令を用いて，FFT カーネルをベクトル化するとともに，キャッシュメモリを効果的に利用できるブロック six-step FFT アルゴリズムを組み合わせることで，既存の FFT ライブラリに比べて特にデータがキャッシュに収まらないような場合に対して性能が改善されることを示した．

実現した並列 FFT を dual Xeon PC および dual Opteron PC 上で，性能評価を行った．その結果，2^{24} 点 FFT において，dual Xeon 3.06 GHz PC では約 809 MFLOPS，dual Opteron 1.8 GHz PC では約 927 MFLOPS の性能を得ることができた．

Short Vector SIMD 命令の 1 つである，Motorola PowerPC の AltiVec では，SSE2 のベクトル命令と同様に 32 bit の単精度浮動小数点演算を行うことが可能である．AltiVec では SSE2 と同様に組み込み関数が使える[20] ことから，今回 SSE2 のベクトル命令に適用した手法は，AltiVec にも適用が可能であると考えられる．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金若手研究（B）（課題番号 14780185）による．

参 考 文 献

1) Cooley, J.W. and Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series, *Math. Comput.*, Vol.19, pp.297–301 (1965).
2) Nadehara, K., Miyazaki, T. and Kuroda, I.: Radix-4 FFT Implementation Using SIMD Multimedia Instructions, *Proc. 1999 IEEE International Conference on Acoustics, Speech, and Signal Processing* (*ICASSP '99*), Vol.4, pp.2131–2134 (1999).
3) Franchetti, F., Karner, H., Kral, S. and Ueberhuber, C.W.: Architecture Independent Short Vector FFTs, *Proc. 2001 IEEE International Conference on Acoustics, Speech and Signal Processing* (*ICASSP 2001*), Vol.2, pp.1109–1112 (2001).
4) Rodriguez, V.P.: A Radix-2 FFT Algorithm for Modern Single Instruction Multiple Data (SIMD) Architectures, *Proc. 2002 IEEE International Conference on Acoustics, Speech and Signal Processing* (*ICASSP 2002*), Vol.3, pp.3220–3223 (2002).
5) Kral, S., Franchetti, F., Lorenz, J. and Ueberhuber, C.W.: SIMD Vectorization of Straight Line FFT Code, *Proc. 9th International Euro-Par Conference* (*Euro-Par 2003*), Lecture Notes in Computer Science, Vol.2790, pp.251–260, Springer-Verlag (2003).
6) Frigo, M. and Johnson, S.G.: FFTW. http://www.fftw.org
7) Frigo, M. and Johnson, S.G.: FFTW: An adaptive software architecture for the FFT, *Proc. 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing* (*ICASSP 98*), pp.1381–1384 (1998).
8) Takahashi, D.: A Blocking Algorithm for FFT on Cache-Based Processors, *Proc. 9th International Conference on High Performance Computing and Networking Europe* (*HPCN Europe 2001*), Lecture Notes in Computer Science, Vol.2110, pp.551–554, Springer-Verlag (2001).
9) Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference* (2003).
10) Intel Corporation: *Intel C++ Compiler for Linux Systems User's Guide* (2003).
11) Intel Corporation: *Intel Math Kernel Library Reference Manual* (2003).
12) Bailey, D.H.: FFTs in External or Hierarchical Memory, *The Journal of Supercomputing*, Vol.4, pp.23–35 (1990).
13) Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA (1992).
14) Swarztrauber, P.N.: FFT Algorithms for Vector Computers, *Parallel Computing*, Vol.1, pp.45–63 (1984).
15) Agarwal, R.C.: An Efficient Formulation of the Mixed-Radix FFT Algorithm, *Proc. International Conference on Computers, Systems and Signal Processing*, pp.769–772 (1984).
16) Takahashi, D., Boku, T. and Sato, M.: A Blocking Algorithm for Parallel 1-D FFT on Clusters of PCs, *Proc. 8th International Euro-Par Conference* (*Euro-Par 2002*), Lecture Notes in Computer Science, Vol.2400, pp.691–700, Springer-Verlag (2002).
17) Takahashi, D.: A parallel 1-D FFT algorithm for the Hitachi SR8000, *Parallel Computing*, Vol.29, pp.679–690 (2003).
18) Advanced Micro Devices Inc.: *AMD Core Math Library* (*ACML*) *Version 1.5.0* (2003).
19) Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A. and Upton, M.: Hyper-Threading Technology Architecture and Microarchitecture, *Intel Technology Journal*, Vol.6, pp.1–11 (2002).
20) Motorola, Inc.: *AltiVec Technology Programming Interface Manual* (1999).

(平成 16 年 1 月 31 日受付)
(平成 16 年 5 月 9 日採録)

高橋 大介（正会員）

昭和 45 年生．平成 3 年呉工業高等専門学校電気工学科卒業．平成 5 年豊橋技術科学大学工学部情報工学課程卒業．平成 7 年同大学大学院工学研究科情報工学専攻修士課程修了．平成 9 年東京大学大学院理学系研究科情報科学専攻博士課程中退．同年同大学大型計算機センター助手．平成 11 年同大学情報基盤センター助手．平成 12 年埼玉大学大学院理工学研究科助手．平成 13 年筑波大学電子・情報工学系講師．平成 16 年筑波大学大学院システム情報工学研究科講師．博士（理学）．並列数値計算アルゴリズムに関する研究に従事．平成 10 年度情報処理学会山下記念研究賞，平成 10 年度，平成 15 年度情報処理学会論文賞各受賞．日本応用数理学会，ACM，IEEE，SIAM 各会員．

朴 泰祐（正会員）

昭和 59 年慶應義塾大学工学部電気工学科卒業．平成 2 年同大学大学院理工学研究科電気工学専攻後期博士課程修了．工学博士．昭和 63 年慶應義塾大学理工学部物理学科助手．平成 4 年筑波大学電子・情報工学系講師，平成 7 年同助教授，平成 16 年同大学システム情報工学研究科および計算科学研究センター助教授，現在に至る．超並列処理ネットワーク，超並列計算機アーキテクチャ，クラスタコンピューティング，並列処理システム性能評価等のハイパフォーマンスコンピューティングシステムの研究に従事．日本応用数理学会，IEEE 各会員．

佐藤 三久（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年通産省電子技術総合研究所入所．平成 8 年新情報処理開発機構並列分散システムパフォーマンス研究室室長．平成 13 年より，筑波大学電子・情報工学系教授．同大学計算科学研究センター勤務．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術，グリッドコンピューティング等の研究に従事．IEEE，日本応用数理学会各会員．