社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－ARC－157　(28)
2004－HPC－ 97　(28)
2004／3／3

走査透過電子顕微鏡の実時間収差補正システムの性能予測

高橋　翔†　　市川　周一†

走査透過電子顕微鏡の分解能は，対物レンズの球面収差のため理論的限界の 1/100 程度に制約されている．生田は，多数の小面積検知器を用いて実時間信号処理を施し，無収差の振幅像と位相像を得る手法を提案した．本研究では，生田の手法を PC クラスタで実装するため，信号処理部のチューニングと性能評価を行い，最終システムの処理性能を予測する．

Performance Estimation of a Real-time Aberration Correction System for Scanning Transmission Electron Microscopes

Sho Takahashi † and Shuichi Ichikawa†

The resolution of a scanning transmission electron microscope remains 100 times worse than its theoretical limit owing to the spherical aberration of objective lens. Ikuta proposed a system to generate an aberration-free amplitude image and phase image using multiple small detectors with real-time signal processing. This study shows the performance optimization results of its signal processing procedure, and estimates the overall performance of Ikuta's aberration correction system.

1. はじめに

透過電子顕微鏡 (TEM; Transmission Electron Microscope) は，試料に高速の電子線を照射して，透過電子と散乱電子およびそれらの干渉を観察する装置である．特に，対物レンズで電子線を細く収束して試料上を走査する方式の透過電子顕微鏡を，走査透過電子顕微鏡 (STEM; Scanning Transmission Electron Microscope) と呼ぶ．

TEM の分解能を向上させるためには，加速電圧の増加，色収差の低減，対物レンズの球面収差の低減などが必要である．しかし磁界型電子レンズでは原理的に凹レンズが実現できないため，対物レンズの球面収差を補正することが困難である．そのため TEM の分解能は電子の波長 (0.01〜0.04Å) に基づく回折限界よりも低い 1Å 程度に制限されている．電子光学系とは別の補正系で電子レンズの球面収差を取り除く方法として，電子線ホログラムによる収差補正や，デフォーカスの異なる複数の観察像をフーリエ面で画像処理する収差補正法なども提案されている．最近では，多極子電子光学系を用いた球面収差補正 TEM[1]，STEM[2] が実用化されているが，無収差位相像を得ることはできない．

† 豊橋技術科学大学 知識情報工学系
Dept. Knowledge-based Information Engineering, Toyohashi University of Technology

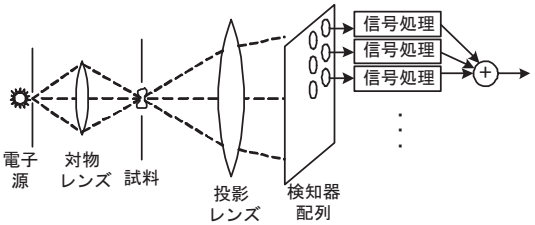

図 1　システム構成図[4)5)]

生田[3] は，動的ホローコーン照明法と焦点深度拡大を併用することで，光学系の球面収差に影響されない結像方式が実現できることを示した．この手法では，2 次元フーリエ面上で収差補正バンドパスフィルタを適用することにより，振幅像・位相像の双方に対して無収差結像が可能である．すなわち，通常顕微鏡としても位相差顕微鏡としても使用できる．生田はこの手法の有効性をシミュレーションで検証し[4]，システムの構築方法についても提案している[4)5)]．

図 1 は，生田の収差補正法を実現するシステムの構成例である．本システムと通常の STEM の相違点は，検知器〜信号処理部にある．通常の STEM では単一の大面積検知器を用いて観察像を生成する．一方，生田のシステムでは多数の小面積検知器からなる検知器配列を用い，各検知器の出力に対して信号処理を施した後，出力を積算して観察像を生成する．

図 2 は，図 1 の信号処理部の内容を示したものである．信号処理は以下の手順で行われる．

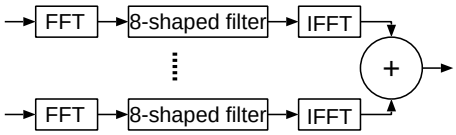

図 2 信号処理部の概念図

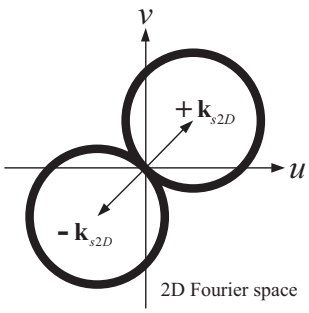

図 3 8 の字フィルタ[4)]

(1) 各検出器からのデータを，1 フレーム分まとめて 2 次元配列に格納する．
(2) 2 次元配列に対して複素 FFT 演算を行い，2 次元フーリエ像を得る．
(3) 2 次元フーリエ面で，収差除去フィルタ (8 の字フィルタ) を適用する．
(4) 逆 FFT (IFFT) により，1 検知器あたりの再生画像を得る．
(5) 各検出器の再生画像を積算して，最終処理像を求める．

図 3 に，上記ステップ (3) の収差除去フィルタを図示した．ベクタ $\mathbf{k}_{s2D}$ は，試料と小面積検知器の位置関係で定まる 2 次元ベクタである．生田[3)] は，2 次元フーリエ面上で，$\pm\mathbf{k}_{s2D}$ を中心とした半径 $|\mathbf{k}_{s2D}|$ の円上には，球面収差成分が含まれないことを示した．従って図 3 に示す形状の狭帯域バンドパスフィルタを用いれば，信号から球面収差成分を取り除くことができる．ただし，球面収差だけでなく他の多くの周波数成分も失ってしまうため，単独の検知器の再生像は現実とかけ離れたものになる．そこで位置関係 $\mathbf{k}_{s2D}$ の異なる多くのセンサを用い，得られた多くの再生像を積算すると，周波数空間が広く覆われて自然な像が得られる．これが上記ステップ (5) の意味である．なお，通常顕微鏡像を得たい場合は 8 の字フィルタの強調係数を $+\mathbf{k}_{s2D}$ 側と $-\mathbf{k}_{s2D}$ 側で同符号にし，位相差顕微鏡像を得たい場合は逆符号とする．

非線形結像成分を除去するためには，さらにステップ (5) の後，デフォーカスを連続的に変えながら最終処理像を積算する必要がある (デフォーカス移動平均)．しかしながら，電子光学系に色収差が含まれていれば，陽にデフォーカス移動平均を行わなくとも，非線形結像成分は相当部分除去されると考えられる[4)]．そこで本研究では，以下，デフォーカス移動平均に関しては考慮しないことにする．

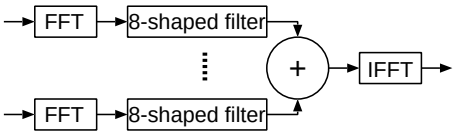

図 4 現実の信号処理部

FFT や IFFT は線形演算であるので，検知器毎に IFFT を実行してから総和を取る (図 2) かわりに，周波数成分の総和を取ってから IFFT を行うと演算回数を削減することができる (**図 4**)．そこで本研究では，以下，図 4 の処理方法を前提として議論を進める．

使用目的や観察対象にもよるが，一般に STEM では実時間での観察が望まれる．実時間画像出力を得るには，上に述べた収差補正処理もビデオレートで実行する必要がある．生田[4)] は，各小面積検知器の信号処理が並列処理可能であることを指摘し，並列処理によってビデオフレームレートに近い速度の収差補正が可能であると予測した．しかし具体的なシステム設計や現実的評価に関しては言及していない．

本研究では，生田の収差補正システムを具体的に検討し，現在の技術で実時間収差補正処理が可能であるか検討する．そのために，まず 2 章で 2 次元複素 FFT の実行時間を検討し，次に 3 章で 8 の字フィルタ処理の実行時間を調査する．これらの調査結果を踏まえて，4 章で並列信号処理による実時間収差補正システムの性能を予測する．

2. FFT の性能

2.1 ライブラリ

FFT はポピュラーな応用であり，すでに多くの高性能ライブラリが開発されている．2 次元複素 FFT を提供する数値計算ライブラリのうち，広く入手可能なものに以下の 4 つがある．

FFTW2 MIT の Frigo と Johnson が開発した FFT ライブラリ．GPL で配布されている．MPI による並列 FFT が可能である．[6)]

FFTW3 FFTW の新版で，SSE/SSE2 や 3DNow! などの SIMD 命令に対応している．Thread による並列 FFT も可能であるが，MPI による並列処理には対応していない．[7)]

MKL Intel 社の数値計算ライブラリ (Math Kernel Library)．30 日間の試用版は無料．[8)]

ACML AMD Core Math Library．AMD 社から提供されている AMD64 アーキテクチャ用の数値計算ライブラリで，BLAS, LAPACK, FFT を含む．無料で使用可能 (2003 年 12 月現在)．g77/gcc で利用できるが，基本的には Fortran 用である．[9)]

各ライブラリの SIMD 命令および MPI への対応状況を**表 1** にまとめた．ACML を使うには SSE2 に対応

表 1 各ライブラリの特徴

	SIMD	MPI	環境の制限
FFTW2	×	○	無し
FFTW3	○	×	無し
MKL	○	×	無し
ACML	○（SSE2 のみ）	×	SSE2 必須

表 3 測定環境

	Pentium4 環境	Athlon 環境
CPU	Pentium4 2.4 GHz	Athlon XP 2600+
RAM	512 MB	1024 MB
OS	RedHatLinux 9	RedHatLinux 8.0
コンパイラ	gcc 3.2.2-5	gcc 3.2
	（ACML のみ f77 3.2.2）	
FFT の種類	複素 FFT，順方向	複素 FFT，順方向

した CPU が必要で，Pentium4，Opteron，Athlon64 などで動作可能である．

2.2 ライブラリの性能比較

4 種類のライブラリについて，データサイズ 256×256〜1024×1024 の 2 次元複素 FFT と，データサイズ 1000000 の 1 次元複素 FFT の実行時間を測定した．精度は倍精度とした．測定結果を**表 2** に示す．測定は Pentium4 システムと Athlon システムで行った．各システムの構成を**表 3** に示す．測定プログラムは基本的に C 言語で作成したが，ACML のみ Fortran で作成した．各ライブラリのバージョンと，コンパイル時のオプションは**表 4** の通りである．

FFTW3 は SIMD 命令の有効/無効がコンパイル時に指定できるので，Pentium4 では両方の場合を計測した．FFTW2 は SIMD 命令をサポートしていないので，SIMD 命令は利用されない．ACML，MKL は SIMD 演算が強制的に利用される (使わないという選択肢がない)．Athlon XP は SSE2 をサポートしていないため，SSE2 必須の ACML は測定対象から除外した．同様に Athlon XP の FFTW3 では SIMD 演算が行われていない (Athlon XP には倍精度の SIMD 命令が存在しないため)．

FFTW2，FFTW3，MKL にはプラン作成という概念がある☆．プラン作成では，使用するアルゴリズムを選択し，アルゴリズムに応じた領域を確保し，領域を初期化する．同じ配列を使う限り，プランを毎回作成する必要はないため，FFTW2，FFTW3，MKL ではプラン作成の時間を除いた実行時間を測定している．ACML の場合はプランが存在していないので，実行時にプラン作成に相当する処理を毎回行っている．つまり ACML の測定時間には，プラン作成相当の時間が含まれている．

表 2 を見ると，2 次元 FFT において MKL の実行時間が非常に大きいことがわかる．そこで MKL に関して更に調査したところ，gcc の代わりに Intel の icc[10] を使い，コンパイルオプション-xW を指定して Pentium4 に特化したコードを生成すると，速度が向上することがわかった．測定環境に表 3 の Pentium4 と icc-7.1 を用いたときの処理時間を**表 5** に示す．icc を用いても，単純な最適化 (-O2) だけでは表 2 (gcc) と同程度の性能しか得られない．ちなみにこの環境 (icc -O2 -xW) を他のライブラリに適用してみたが，特に処理速度の向上は見られなかった．

表 2 において，データのサイズと演算時間の関係は単純でない．一般に FFT ではデータサイズを 2 の冪乗にとるのが自然であるが，キャッシュのライン衝突などによって性能が低下する可能性がある．さらに，現実の FFT ライブラリには高度なチューニングが施されており，高性能を達成できる問題サイズは自明ではない．

そこで，データサイズ $x \times x$ $(950 \leq x \leq 1050)$ について，FFTW3 の 2 次元複素 FFT の処理時間を調べてみた．実行環境は表 3 の通りで，SSE2 は使用していない．その結果を**図 5** に示す．この範囲内でも，実行時間に 10 倍程度のばらつきがあることがわかる．適切な x を選択することは，高速処理のために非常に重要である．

表 4 各ライブラリの条件

	バージョン	コンパイルオプション
FFTW2	2.1.5	-O2
FFTW3	3.0.1	-O2
MKL	6.1	-O2
ACML	1.0r2	-O2

表 5 icc と MKL による処理時間 [ms]

問題サイズ	-O2 -xW	-O2
256×256	1.1×10^{1}	3.3×10^{3}
640×480	7.0×10^{1}	6.1×10^{3}
1000×1000	2.2×10^{2}	2.8×10^{4}
1024×1024	2.3×10^{2}	1.3×10^{4}
1000000	4.8×10^{2}	4.8×10^{2}

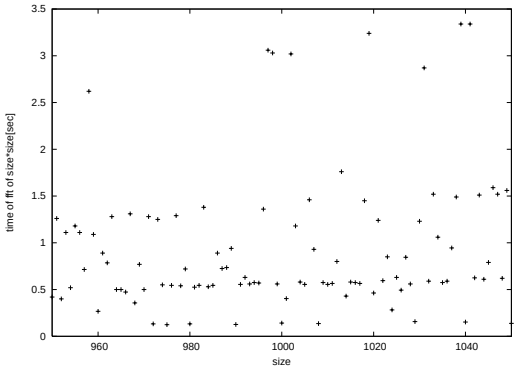

図 5 データサイズ 1000×1000 付近の処理時間

☆ 正確には，プランは FFTW の用語であり，MKL では Descripter という表現を使う．

表 2 測定結果（単位 ms）

環境	ライブラリ	256×256	512×512	640×480	1000×1000	1024×1024	1000000
Pentium4	FFTW2	3.1×10^1	1.2×10^2	4.8×10^1	1.9×10^2	5.4×10^2	3.1×10^2
	FFTW3 (†)	1.1×10^1	7.1×10^1	5.5×10^1	1.4×10^2	3.3×10^2	3.1×10^2
	FFTW3	1.1×10^1	7.0×10^1	4.5×10^1	1.4×10^2	2.9×10^2	2.8×10^2
	MKL	3.5×10^1	3.3×10^3	6.1×10^3	2.8×10^4	1.3×10^4	4.0×10^2
	ACML	2.1×10^1	9.3×10^1	1.3×10^2	4.0×10^2	4.0×10^2	5.2×10^2
Athlon	FFTW2	2.3×10^1	1.20×10^2	6.5×10^1	1.72×10^2	5.20×10^2	2.7×10^2
	FFTW3 (‡)	1.4×10^1	8.3×10^1	7.9×10^1	1.7×10^2	3.3×10^2	2.9×10^2
	MKL	9.6×10^0	3.9×10^2	5.2×10^2	2.3×10^3	1.4×10^3	4.7×10^2

(†) コンパイル時に SIMD を無効化したもの.
(‡) Athlon XP には倍精度の SIMD 命令はないので，SIMD 演算は行われていない.

表 6 データ数と処理時間

問題サイズ	時間 [ms]	因子数	素因数
960×960	2.7×10^2	8	$2^6\times3\times5$
972×972	1.3×10^2	7	$2^2\times3^5$
975×975	1.2×10^2	4	$3\times5^2\times13$
980×980	1.3×10^2	5	$2^2\times5\times7^2$
990×990	1.3×10^2	5	$2\times3^2\times5\times11$
1000×1000	1.4×10^2	6	$2^3\times5^3$
1008×1008	1.4×10^2	7	$2^4\times3^2\times7$
1024×1024	2.8×10^2	10	2^{10}
1029×1029	1.6×10^2	4	3×7^3
1040×1040	1.5×10^2	6	$2^4\times5\times13$
1050×1050	1.4×10^2	5	$2\times3\times5^2\times7$

FFTW3 のマニュアル[7] によれば，FFTW はデータサイズが $2^a3^b5^c7^d11^e13^f$ のとき高速に計算できるとある (ただし $e+f=0$ or 1). 従ってデータサイズ 1000×1000=$2^35^3\times2^35^3$ が比較的高速に処理できることは納得できる．しかしデータサイズ 1024×1024=$2^{10}\times2^{10}$ は，データサイズ 1000×1000 に比べ 2 倍以上の時間がかかっている．

図 5 の範囲で $x=2^a3^b5^c7^d11^e13^f$ を満たすデータサイズ x の実行時間をまとめたものが，**表 6** である．データサイズを素因数分解して，因数が多くなる場合に処理速度が遅くなる傾向がみられる．今回の応用では，1000×1000 程度の画像が必要であるが，偶然にも 1000×1000 は高速化の条件を満たしていることがわかる．

2.3 FFTW2 による並列 FFT

FFTW2 は MPI を利用した並列 FFT に対応している．本研究では，Pentium-III 866 MHz のデュアルプロセッサ (メモリ共有)，Athlon 1.33 GHz 2 台 (100base-TX 接続) などの構成で性能測定を行った．測定結果はページ数の制約上省略するが，並列 FFT による性能上のメリットはほとんど見られなかった．そのため，以下，MPI による FFT の並列処理は検討の対象外とした．

2.4 FFTW3 のチューニング

以上の結果をふまえ，以下の議論では FFTW3 を FFT ライブラリとして採用する．本章では更に FFTW3 のオプションを調査し，性能改善の余地があるか検討する．測定は表 3 の Pentium4 環境で行い，データサイズは高速な演算が可能な 1000×1000 とした (2.2 節参照).

2.4.1 プ ラ ン

2.2 節で述べたように，FFTW3 では FFT 実行に先立ってプラン作成を行う．FFTW3 には以下の 4 つのプラン作成オプションが用意されており，それぞれ目的や性能に相違がある[7].

ESTIMATE プランを高速に作ることが出来るが，計算は遅い．入力/出力配列は保護される．

MEASURE プランを作るのに数秒かかるが計算を高速に実行できる (**デフォルト**).

PATIENT MEASURE と似ているが，さらに高速に計算できるプランを作る．ただしプラン作成に長時間かかる．(1024×1024 で 5 分程度必要)

EXHAUSTIVE PATIENT よりもさらに広範囲のアルゴリズムを調べて，高速なプランを作成する．(1024×1024 で 20 分程度必要)

表 7 各プランでの測定結果

オプション	FFT [ms]	IFFT [ms]
ESTIMATE	1.7×10^2	1.7×10^2
MEASURE	1.4×10^2	1.7×10^2
PATIENT	1.4×10^2	1.7×10^2
EXHAUSTIVE	1.8×10^2	1.8×10^2

それぞれのオプションでプランを作成し，1000×1000 の 2 次元倍精度複素 FFT と IFFT の実行時間を測定した (**表 7**). プランはファイルに保存することが可能で，使用時のプラン読み込み時間は数十 ms 程度である．従って PATIENT でプランを作成し，結果を保存しておいて実行時に読み込むのが得策と思われる．また，FFT と IFFT の時間は最大で 2 割ほど異なる場合があるが，基本的には大差ないことがわかる．

2.4.2 SIMD 命令

FFTW では SSE/SSE2，3DNow!がサポートされている．SIMD 命令の効果を確認するため，Pentium4 の SIMD 命令 (SSE/SSE2) を有効化／無効化して測定を行った．結果を**表 8** に示す．なお，SSE 命令は単精度演算，SSE2 命令は倍精度演算なので，単精度と倍精度の FFT を測定した．単精度 FFT では SSE により一定の高速化が得られたが，倍精度 FFT では SSE2 の効果は小さかった．検知器入力 (ADC の精

表 8 SIMD 命令の効果

オプション	FFT [ms]	IFFT [ms]
float (SSE 有効)	9.0×10^1	9.5×10^1
float (SSE 無効)	1.5×10^2	1.5×10^2
double (SSE2 有効)	1.4×10^2	1.4×10^2
double (SSE2 無効)	1.4×10^2	1.7×10^2

表 9 複数 FFT を一括処理

howmany	単精度 [ms]	倍精度 [ms]
1	7.0×10^1	1.4×10^2
2	6.1×10^1	1.1×10^2
4	5.4×10^1	8.9×10^1
8	4.3×10^1	7.2×10^1
16	4.2×10^1	6.9×10^1

度) は 12〜16 bit であるから，演算精度は単精度とし，SSE を利用するのが効果的と考えられる．

2.4.3 複数 FFT の同時計算

FFTW3 でプランを作成する場合には，ここまで利用してきた fftw_plan_dft() 関数の他に，複数データを一括して扱う fftw_plan_many_dft() 関数を利用することができる．fftw_plan_many_dft() 関数では，引数 howmany に与えた個数分の FFT をまとめて計算する．howmany を 1〜16 として，1000×1000 の単精度・倍精度の FFT を実行した場合の実行時間 (総実行時間を howmany で割った値) を**表 9** に示す．プロセッサは表 3 の Pentium4，プランには PATIENT を用い，SIMD 命令は有効にした．

パラメータ howmany に n を与えるとき，複数 FFT の総実行時間 $T_{Fs}(n)$(単精度) と $T_{Fd}(n)$(倍精度) は以下の一次式で近似できる (単位 ms)．

$$T_{Fs}(n) = 39.3n + 40.9 \quad (1)$$

$$T_{Fd}(n) = 63.3n + 86.7 \quad (2)$$

3. 8 の字フィルタ

本章では，8 の字フィルタの性能チューニングを試みる．まず，生田の Fortran プログラム stemmda2.f[4] から 8 の字フィルタ部分を抜き出し，C 言語に移植した (プログラム P1)．P1 を表 3 の Pentium4 で動作させたところ，サイズ 1000 × 1000 の実行時間が 1.1 秒であった．これは FFT と比べて遅すぎるので，以下のチューニングを施した．

(1) P1 では変数を単精度で定義しているが，標準ライブラリは倍精度で計算を行うため，キャストが多数発生する．そこで全ての変数を倍精度にかえて，キャストを減らす．

(2) 繰り返し使われる共通部分式を変数に記憶して，計算回数を減らす．

(3) 命令列を Pentium4 用に最適化するため，gcc のコンパイルオプションを -O3 -march=pentium4 -msse2 -mfpmath=sse にする．

表 10 各チューニングを組み合わせた結果

測定条件	コンパイラ	実行時間 [ms]
単精度	gcc	6.6×10^2
(2)(3)(4)(5)(6)(7)(8)	icc	3.0×10^2
倍精度	gcc	6.0×10^2
(1)(2)(3)(4)(7)(8)	icc	4.1×10^2

(4) C99 規格☆の単精度用の数学ライブラリを使う．

(5) C99 規格の複素数データ型である _Complex 型を導入する．_Complex 型は単精度，倍精度で定義可能である．複素数演算の記述が簡潔になるだけでなく，高速化も期待できる．

(6) コンパイルオプションで計算精度を 24bit にする．gcc では–allow-single-precision オプションを指定する．

(7) 8 の字領域に入っていない部分は計算しなくても結果に影響が出ないため，条件を判定して処理を省略する．

(8) 8 の字領域に入っているかどうか判別する際に，平方根を利用しているので，除去する．(例えば $a < \sqrt{b}$ という比較式を $a^2 < b$ に変更)

上記各手法について実行時間を測定し，効果を調べた (個別の結果は紙数不足のため省略する)．各手法は単精度と倍精度で効果が異なるので，それぞれに適すると考えられるオプションを組み合わせて測定を行った．その結果，単精度では (2)(3)(4)(5)(6)(7)(8)，倍精度では (1)(2)(3)(5)(7)(8) を組み合わせると良い結果が得られた．実行時間を**表 10** に示す．gcc では単精度の実行時間が倍精度より長いが，icc-7.1 では単精度の方が短い．いずれにせよ，本章の性能改善の結果，8 の字フィルタの実行時間は FFT, IFFT より短くなった．

4. おわりに

本章では，これまで述べてきた性能改善策を生田の収差補正システムに適用し，システムの性能を予測する．

まず，生田の収差補正シミュレーションプログラム stemmda2.f[4] を C 言語に移植し，アルゴリズムを図 2 から図 4 に変更して，プログラム stemmda2.c を作成した．stemmda2.f (stemmda2.c) は，17 × 17 に配列された 289 検知器について，解像度 256 × 256 の (球面収差を含む) 試験画像を生成し，収差補正処理のシミュレーションを行うプログラムである．次に stemmda2.c に 2 章と 3 章で検討した性能改善手法を実装し，プログラム custom.c を作成した．stemmda2.c と custom.c の出力画像を生田の結果と比較して，動作に問題がないことも確認した．

次に表 3 の Pentium4 で stemmda2.c と custom.c の実行時間を測定した (**表 11**)．それぞれ単精度版と

☆ ISO/IEC 9899:1999 - Programming Language C の略称

表 11 比較結果

	実行時間 [sec.]		
	stemmda2.c	custom.c	性能比
単精度	4.4	2.2	2.0
倍精度	8.1	6.2	1.3

表 12 実行時間内訳

	時間 [sec]	割合
FFT, IFFT	1.4	62%
8 の字フィルタ	0.72	31%
その他処理	0.17	7%

倍精度版を作成し，演算精度に合わせた最適化手法を適用している (2 章，3 章参照)．表 11 の実行時間は，FFT，8 の字フィルタ，加算，IFFT の時間である．サンプル画像の生成時間は除外した．画像が 256×256 と小さいこともあり，289 検知器のデータを 1 台の Pentium4 で処理して，実行時間は 2.2〜8.1 秒である．custom.c では stemmda2.c と比べて，単精度で 2.0 倍，倍精度で 1.3 倍の性能向上が得られる．custom.c の実行時間の内訳を**表 12** に示す．FFT，IFFT，8 の字フィルタで実行時間の 90%以上を占めているので，本研究で無視した部分の最適化は実用上も不要であると判断される．

次に実用的システムを構築する場合の性能予測を試みる．入力画像の画素数は，ユーザの立場では多いほど良いが，画素数が多いと処理時間が長くなるので，ここでは 1000×1000 とする．各画素は 12〜16 ビット (ADC の精度) であるから，演算精度は単精度とする．検知器数は出力画像の品質に影響するため，あまり減らすことはできない．ここでは 16×16 とする．

表 11 のシミュレーションでも分かるとおり，1 台のプロセッサで全検知器を処理すれば数秒以上の処理時間が必要となり，実時間観察という要求から外れる．生田[4] も述べた通り，複数の PC を用いて並列処理することが適当と思われる．しかし電子顕微鏡 1 台に検知器数 (256 台) の PC を接続することは現実的でない．各 PC に複数の検知器を接続することが妥当であろう．実際，マルチポートの ADC ボード製品を利用して，1 枚の PC に 2〜16 個ほどの検知器を接続することは容易である．

表 3 の Pentium4 PC を $\lceil 256/n \rceil$ 台利用し，各 PC に n 検知器を担当させる場合，システムの信号処理時間 T は以下の式で見積もられる．

$$T = T_{Fs}(n) + nT_8 + T_\Sigma + T_I \qquad (3)$$

ここで，$T_{Fs}(n)$ は n 画像分の FFT 処理時間で，近似式 (1) で与えられる．T_I は画像 1 枚を IFFT で処理する時間であるが，FFT と IFFT の時間は (ほぼ) 同じなので $T_I = T_{Fs}(1)$ で近似できる．T_8 は 1 画像のフィルタ処理時間で，表 10 から 30 ms と予測される．T_Σ は画像の総和を求める時間であるが，本研究では FFT, IFFT, フィルタ処理と比べれば無視できると考え，$T_\Sigma \approx 0$ を仮定する．以上より，

$$T \approx 69n + 121 \quad (ms) \qquad (4)$$

となる．64 プロセッサ ($n = 4$) で 0.4 秒，16 プロセッサ ($n = 16$) で 1.2 秒程度となる．充分に速くはないが，実用可能な処理時間であると思われる．

プロセッサが高速化すれば処理時間 T は削減できる．しかしながら，今度は入力時間が問題になると予想される．例えば $n = 16$ では，16 検知器の画像入力 (各 2MB) を処理時間内に転送しなければならない．現在必要な転送速度は $16 \times 2/1.2 = 27$ MB/s で，PCI バス (32 bit, 33 MHz) の帯域で足りる．しかし検知器のデータをビデオレート (30 frame/s) で転送しようとすれば，256 検知器で 15 GB/s，16 検知器で 960 MB/s の転送速度が必要となり，現在の PC の拡張バス (PCI など) では到底まかなえない．入出力転送幅の観点から見ても，複数 PC による分散入出力が必要であると考えられる．

謝辞 本研究の一部は，科学研究費補助金・基盤研究 (C)(2)13680410，および文部科学省 21 世紀 COE プログラム「インテリジェントヒューマンセンシング」の援助により行われた．

参 考 文 献

1) 細川史夫, 沢田英敬: Cs 補正 TEM, 日本顕微鏡学会第 48 回シンポジウム発表要旨集, pp. 55–58 (2003).
2) 阿部英司, Pennycook, S.: 球面収差 (Cs) 補正 STEM とその応用, 日本顕微鏡学会第 48 回シンポジウム発表要旨集, pp. 63–66 (2003).
3) Ikuta, T.: An Aberration-free Imaging Technique Based on Focal Depth Extention, *Journal of Electron Microscopy*, Vol. 47, No. 5, pp. 427–432 (1998).
4) 生田孝: 画像処理による収束光学系収差補正機構をもつ透過型走査光学顕微鏡の試作, 科学研究費補助金 基盤研究 (B)(2)11555021 研究成果報告書 (2001).
5) 志水隆一, 生田孝: 走査型顕微鏡装置, 特許 3035612 (2000).
6) *FFTW2 Online Manual*. http://www.fftw.org/fftw2_doc/.
7) *FFTW3 Online Manual*. http://www.fftw.org/fftw3_doc/.
8) *Intel Math Kernel Library Reference Manual*. http://developer.intel.com/software/products/mkl/techtopics/mklman60b.pdf.
9) *ACML User Guide*. http://www.developwithamd.com/appPartnerProg/acml/home/.
10) *Intel C++ Compiler for Linux User's Guide*. http://www.intel.com/software/products/compilers/clin/docs/ug/index.htm.