Vol.2017-HPC-161 No.16
2017/9/20

FFTカーネルを用いたKNLでのスケーラビリティに関する調査

青木 聖陽[1]　廣田 悠輔[2]　今村 俊幸[2]　横川 三津夫[1]

概要：FFT は音声・画像の信号処理や数値シミュレーションなど幅広い分野で用いられるデータ解析アルゴリズムである．また，3 次元 FFT は流体シミュレーションで用いられ，現実的なシミュレーションコードの実行のためには FFT カーネルと共にハードウェアにも高い並列計算性能が要求される．FFT は内部に自明な高並列処理を有するためメニイコアや GPU などでの並列処理には向いているが，連続アドレスのアクセスとストライドアクセスが同時に現れるためメモリ周りに高い性能を要求することが知られている．本稿では，近年注目されている，Intel Xeon Phi の第 2 世代 Knights Landing（KNL) 上での 3 次元 FFT 応用を視野に入れて，STREAM ベンチマークや FFTE の C 言語移植版，FFTW などの基本カーネルコードを使用し KNL のメモリ周りの性能測定をまず調査する．更に，KNL 上での 3D-FFT 処理の並列性能評価（特にスケーラビリティ）に関する性能分析を行う．

Aoki Masaaki[1]　Hirota Yusuke[2]　Imamura Toshiyuki[2]　Yokokawa Mitsuo[1]

1. はじめに

高速フーリエ変換 (Fast Fourier Transform; FFT)[1] は音声・画像の信号処理や数値シミュレーションなど幅広い分野で用いられるデータ解析アルゴリズムである．また，3 次元 FFT は流体シミュレーションで用いられ，現実的なシミュレーションコードの実行のためには FFT カーネルと共にハードウェアにも高い並列計算性能が要求される．近年，Intel Xeon Phi の第 2 世代 Knights Landing（以下 KNL) を代表とする新たなプロセッサが出現し，アプリケーション利用者ならびに数値計算ルーチン開発者は高い性能を得られることを期待している．メニイコアをターゲットハードウェアとしたときの FFT の性能についてレビューをしたところこれまでに数件の学会発表が見られる．Deslippe らは Xeon Phi アーキテクチャにおける BerkeleyGW コードの最適化を行い，KNL における予備的な結果を示している [2]．Wende らは VASP における 3 次元 FFT をメニイコア向けに最適化を行い，Haswell CPU および Xeon Phi の第 1 世代 Knights Corner 上で性能を評価している [3]．高橋は 1 次元 FFT を Intel Advanced Vector Extensions 512 (以下 AVX-512) を用いてベクトル化し，Xeon Phi 7250 上で 91Gflops 以上の性能を達成している [4]．また，文献内で FFTW[6] および MKL[7] との性能比較も行っている．

我々は，高橋によって作成されたオープンソースの FFT プログラムである FFTE[5] を C 言語化し，組み込み SIMD 関数などの高度な最適化を施したいと考えている．そのために，従来発表の報告と比較して，我々の開発した FFTE の C 言語移植版がどの程度の性能を発揮するかを把握しておく必要がある．また，併せて各種 FFT カーネルを同様の条件で測定することで，KNL の性能をある程度評価することは可能である．本稿では，KNL について FFT の利用・開発の視点から理解するため，最良のハードウェアかどうかの判断材料となるベンチマークを実施する．KNL についてはまだ手探りで，性能最適化技術について不明な点があるため，いくつかの項目について最適化の将来指針となる情報の蓄積を行う．

本稿の構成は以下の通りである．2 章では Six-step FFT および 3 次元 FFT のアルゴリズムについて述べる．3 章では，KNL の基本的な構成について述べる．4 章では，実装した FFTE の C 言語版とその他に用いたベンチマークコードについて述べ，その性能評価を行った結果を示す．5 章はまとめの章とする．

[1] 神戸大学大学院システム情報学研究科
Graduate School of System Informatics, Kobe University
[2] 理化学研究所計算科学機構
RIKEN Advanced Institute for Computational Science

2. 高速フーリエ変換

2.1 Six-step FFT

FFT は離散フーリエ変換 (Discrete Fourier Transform; DFT) を高速に計算するアルゴリズムである．DFT は以下で定義される．

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

ここで，$\omega_n = \mathrm{e}^{-\frac{2\pi i}{n}}$，$i = \sqrt{-1}$ である．式 (1) は入力 x_j を並べた列ベクトルを x，出力 y_k を並べた列ベクトルを y，j 行 k 列目の要素が ω_n^{jk} となる n 次正方行列を F_n とすると，次式のような行列ベクトル積に書き換えられる．

$$y = F_n x \tag{2}$$

$n = n_1 \times n_2$ と分解できるとすると，x と y は以下のような 2 次元配列で表現できる．

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

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

F_n が持つ周期性から，式 (2) はクロネッカー積を用いて次式のように変形される [8]．

$$y = \Pi_{n_1,n}^{\top}(I_{n_2} \otimes F_{n_1})\Pi_{n_2,n}^{\top} \mathrm{diag}\left(I_{n_2}, \Omega_{n_2,n}, \ldots, \Omega_{n_2,n}^{n_1-1}\right)(I_{n_1} \otimes F_{n_2})\Pi_{n_1,n}^{\top} x \tag{5}$$

ただし，I_n は n 次単位行列，$\Omega_{k,n} = \mathrm{diag}\left(\omega_n^0, \ldots, \omega_n^{k-1}\right)$ を表す．また，Π は置換行列を表し，$\Pi_{n_1,n}^{\top} x(j_1, j_2) = x(j_2, j_1)$ を満たす．

式 (5) より，以下のアルゴリズムが導出される．

Step 1: $n_1 \times n_2$ 行列の転置
$$x_1 = \Pi_{n_1,n}^{\top} x$$

Step 2: n_1 組の n_2 点 FFT
$$x_2 = (I_{n_1} \otimes F_{n_2}) x_1$$

Step 3: ひねり係数の乗算
$$x_3 = \mathrm{diag}\left(I_{n_2}, \Omega_{n_2,n}, \ldots, \Omega_{n_2,n}^{n_1-1}\right) x_2$$

Step 4: $n_2 \times n_1$ 行列の転置
$$x_4 = \Pi_{n_2,n}^{\top} x_3$$

Step 5: n_2 組の n_1 点 FFT
$$x_5 = (I_{n_2} \otimes F_{n_1}) x_4$$

Step 6: $n_1 \times n_2$ 行列の転置
$$y = \Pi_{n_1,n}^{\top} x_5$$

```
#pragma omp for collapse(2) private(i,j,j_,m_,n_)
  for(i_=0; i_<m; i_+=NB) {
    for(j_=0; j_<n; j_+=NB) {
      m_=int_min(i_+NB,m) - i_;
      n_=int_min(j_+NB,n) - j_;
      for(i_=0; i_<m; i_+=NB) {
        for(j_=0; j_<n; j_+=NB) {
          *(Dst+i+m*j) = *(Src+j+n*i);
        }
      }
    }
  }
```

図 1 キャッシュブロッキングを用いた転置操作のコード
Fig. 1 Code of transpose using cache blocking techniques.

このアルゴリズムは Six-step FFT と呼ばれる．Step2 および 5 の FFT はメモリ参照の局所性が高く，キャッシュを有効に活用できる．一方，Step1，4，6 の転置操作はキャッシュミスが頻発し，ボトルネックとなる．これを改善するため，キャッシュブロッキング手法を用いる．キャッシュブロッキングを用いた転置操作の実装例を**図 1** に示す．NB はブロック幅を定めるパラメータで，データがキャッシュ内に全て収まるように設定することで演算器の利用効率を向上させることが可能である．

2.2 3 次元 FFT

3 次元 DFT は以下で定義される．

$$Y(\beta_1, \beta_2, \beta_3) = \sum_{\alpha_1=0}^{n_x-1} \sum_{\alpha_2=0}^{n_y-1} \sum_{\alpha_3=0}^{n_z-1} \omega_{n_1}^{\beta_1\alpha_1} \omega_{n_2}^{\beta_2\alpha_2} \omega_{n_3}^{\beta_3\alpha_3} X(\alpha_1, \alpha_2, \alpha_3) \tag{6}$$

3 次元 FFT は各次元方向についての 1 次元 FFT によって計算される．その実装は，以下の手順で実現される．

Proc. 1: x 軸方向に対して $n_y n_z$ 組の 1 次元 FFT

Proc. 2: y 軸方向についてデータ連続化

Proc. 3: y 軸方向に対して $n_x n_z$ 組の 1 次元 FFT

Proc. 4: z 軸方向についてデータ連続化

Proc. 5: z 軸方向に対して $n_x n_y$ 組の 1 次元 FFT

Proc. 6: x 軸方向についてデータ連続化

3 次元 FFT は Six-step FFT と同様に複数組の FFT と転置操作によって計算される．

3. Intel Xeon Phi KNL の構成

3.1 基本構成

KNL は，高メモリ帯域幅と高い並列処理性能を持ったメニイコアプロセッサである．KNL は 2D メッシュ・インターコネクトで最大 36 個のタイルが接続され，各タイルは 2 つの物理コアをもつ．各コアはハイパースレッディングにより最大 4 スレッドを同時に実行でき，タイル上の 2

つのコアで 1MB の L2 キャッシュを共有する．L2 キャッシュは 16way のアソシアティブで，64B のキャッシュラインで構成される．また，各コアは 2 つのベクトル演算ユニット (VPU) に接続され，1 サイクルごとに 64 の単精度 (SP) または 32 の倍精度 (DP) 演算が実行できる．KNL がサポートするベクトル命令セットには x87，SSE，AVX と AVX2 に加え，AVX-512 が導入されている．

3.2 クラスタモード

KNL では，各タイル上のコアは 2 コア間で共有する L2 キャッシュにアクセス可能であるが，他の L2 キャッシュや MCDRAM または DDR 内のデータにアクセスするためにはインターコネクトを介して通信を行う必要がある．コアは要求するデータがどのタイル上に存在するかを Cache Home Agent (CHA) によって追跡する．クラスタモードはチップを仮想領域に分割する方法であり，通信を局所的にする．クラスタモードには 3 つの主要なモードがあり，それぞれ All-to-All モード，Quadrant (4 分割) モード，SNC-4 モードである．それぞれの概要を以下にまとめる．

(1) All-to-All モード
タイル・CHA・メモリの間にアフィニティがなく，アドレスがすべてのメモリ上に一様にハッシュされる．メモリパフォーマンスは悪いが，他のモードと異なり DDR DIMM の容量が同一でなくとも使用可能である．

(2) Quadrant モード
チップを仮想的に 4 つの領域に分割し，同一領域内にデータが存在するようアドレスをハッシュする．

(3) SNC-4 モード
チップを 4 つの NUMA に分割する．同一領域内では低レイテンシな near メモリアクセス，それ以外では高レイテンシな far メモリアクセスとなる．

また，Quadrant モードの派生として Hemisphere (2 分割) モード，SNC-4 モードの派生として SNC-2 モードがあるが，分割数が 4 つでなく 2 つであること以外は同じである．

3.3 メモリモード

KNL はコア数の向上に伴い，DP および SP の性能向上は言うまでもないが，前世代の KNC に比べてメモリ周りの変化がアプリケーション性能に大きく影響する．KNL はいわゆる大規模主記憶である DDR4 と中規模な高速記憶領域 MCDRAM を 3 つのモードで使い分けられる構成をとっている．カタログ性能上，DDR4 は 90GB/s，MCDRAM は 400GB/s 以上の性能ができると期待されている．3 つのモードはそれぞれ Cache モード，Flat モード，Hybrid モードと呼ばれる．それぞれの概要を以下にまとめる．

(1) Cache モード
MCDRAM を Last Level Cache (LLC) として使用する．ハードウェアが MCDRAM の使用を制御する．MCDRAM はダイレクト・マッピングキャッシュで，64B のキャッシュラインで構成される．

(2) Flat モード
MCDRAM を NUMA ノードとして使用する．ユーザが MCDRAM の使用を制御する．

(3) Hybrid モード
Flat モードと Cache モードを組み合わせる．メモリ比率は設定可能である．

4. 実装と性能評価

4.1 FFTE-C

本稿では，高橋によって作成されたオープンソースの FFT プログラムである FFTE[5] の C 言語への移植を行った．このプログラムを FFTE-C と呼ぶこととする．C 言語に移植するに当たり，長さ n の複素数データを配列 a に格納する場合，$a[2k], 0 \leq k \leq n-1$ に実数部，$a[2k+1], 0 \leq k \leq n-1$ に虚数部を格納するものとした．FFTE は，基数 2，3，4，5，8 の FFT を用いてデータ長 $2^p3^q5^r$ の複素数データに対する FFT および逆 FFT を計算する．

今回は，FFTE-C を用いて 1 次元の実数データに対する Six-step FFT と 3 次元の複素数データに対する 3 次元 FFT を実装し，AVX-512 によるベクトル化および OpenMP による並列化を行った．AVX-512 によるベクトル化は ivdep プラグマを用いたコンパイルによる自動ベクトル化である．

4.2 ベンチマークコード

ベンチマークコードとして，FFTW(version 3.3.6-pl2)[6]，MKL(version 2017 update 2)[7]，FFTE-C，Transpose ベンチマークコードおよび STREAM ベンチマークコードを用いる．各 FFT は事前にひねり係数を生成し，変換部分のみを測定対象とする．また，FFTW については planner に “measure” を使用する．本稿では，以下の 4 つのプログラムについて性能を測定した．

- MKL，FFTW，FFTE-C による 1 次元 FFT
- FFTE-C による 3 次元 FFT
- Transpose (転置操作) ベンチマーク
- STREAM ベンチマーク

4.3 実行環境

本稿では，Xeon Phi 7210 を用いて性能評価を行った．その仕様を**表 1** に示す．

クラスタモードは Quadrant，メモリモードは Flat モー

表 1　Intel Xeon Phi KNL の仕様
Table 1　Specification of Intel Xeon Phi KNL.

Processor Number	7210
Cores	64
Operating frequency	1.3 GHz
L1 cache	Instruction cache : 32KB Data cache : 32KB
L2 cache	1MB (shared between 2 cores)
Main memory	MCDRAM 16GB DDR4 48GB

ドとする．コンパイラは Intel コンパイラ (icc, 17.0.2 20170213) としコンパイラオプションは`-O3 -qopenmp -fma -xMIC-AVX512` を使用した．GNU CC コンパイラは 4.9 以上で AVX512 をサポートするが，今回は使用しない．環境変数 `KMP_AFFINITY=granularity=fine,balanced` に設定し，`numactl -m 1` によりデータはすべて MCDRAM に割り当てた．また，KNL に直接は関係しないが，AVX でメモリ周りの実験を行うため`-qopt-streaming-stores` オプションを使用した．このオプションは Streaming Store 命令を生成するか否かを指定するもので，`#pragma vector nontemporal` に相当する．

4.4　1 次元 FFT の性能

FFTW，MKL，FFTE-C を用いて，以下のパラメータの範囲で 1 次元 FFT の性能を測定した．

- 問題サイズ $n = 2^{16}, 2^{17}, \ldots, 2^{28}$
- コア数は 64 で固定
- 1 コアあたりのスレッド数は 4 で固定

測定結果を**図 2** に示す．1 次元 FFT の FLOPS は，内部で展開する基数で異なるが，今回は基数 2 のみの利用を想定し，問題サイズが n，実行時間が t のとき $5n\log_2 n/t$ で求める．

参考文献 [4] では FFTE は $n = 2^{24}$ あるいは 2^{25} のとき

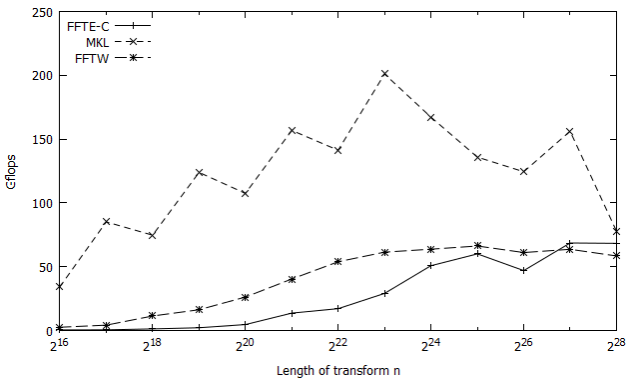

図 2　1 次元 real FFT の性能 (64 コア 各 4 スレッド)
Fig. 2　Performance of 1d real FFTs
(64 cores, 4 threads per core)

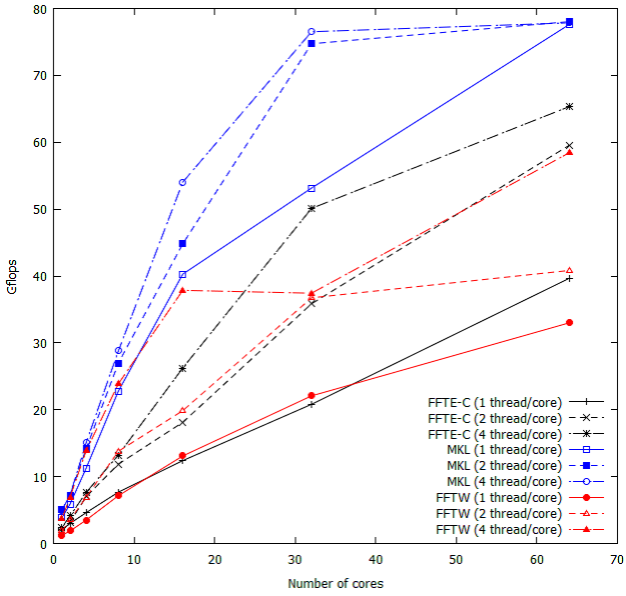

図 3　1 次元 real FFT の性能 ($n = 2^{28}$)
Fig. 3　Performance of 1d real FFTs ($n = 2^{28}$)

100Gflops 近い性能を達成しているが，FFTE-C は最大で 68Gflops 程度だった．FFTE-C では基数 16 の FFT が未実装であることと，転置操作のメモリアクセスに最適化の余地があることが考えられる．転置操作の詳細については Transpose ベンチマークで議論する．MKL については，参考文献と比較するとピーク性能はかなり高かったが，概ね等しい傾向が確認された．FFTW については，バージョンの違いによるものか参考文献よりも高い性能であることが確認された．

また，コア数およびハイパースレッディングの影響を見るため，以下のパラメータの範囲で FFTE-C，MKL および FFTW の 1 次元 FFT の性能を測定した．

- 問題サイズ n は 2^{28} で固定
- コア数=1,2,4,8,16,32,64
- 1 コアあたりのスレッド数=1,2,4

測定結果を**図 3** に示す．

FFTE-C ではハイパースレッティングが効果的に作用していることが確認された．

4.5　3 次元 FFT の性能

FFTE-C を用いて，以下のパラメータの範囲で 3 次元 FFT の性能を測定した．

- 問題サイズ $n = 32^3, 64^3, \ldots, 512^3$
- コア数は 64 で固定
- 1 コアあたりのスレッド数は 4 で固定

測定結果を**図 4** に示す．3 次元 FFT の FLOPS は，1 次元の場合と同様に基数 2 のみの利用を想定し，問題サイズが $n_x \times n_y \times n_z$，実行時間が t のとき，$5n_x n_y n_z \log_2(n_x n_y n_z)/t$ で求める．

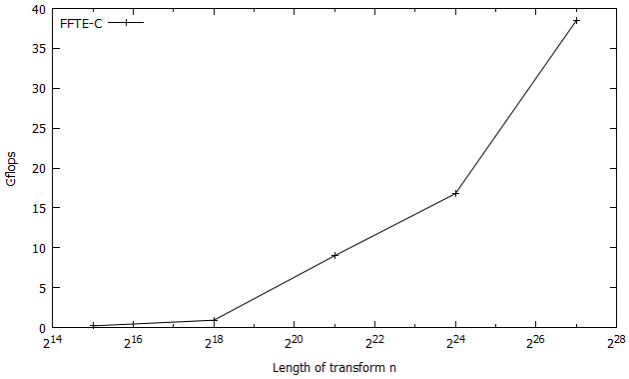

図 4　3 次元 real FFT の性能 (64 コア 各 4 スレッド)
Fig. 4　Performance of 3d FFT
(64 cores, 4 threads per core)

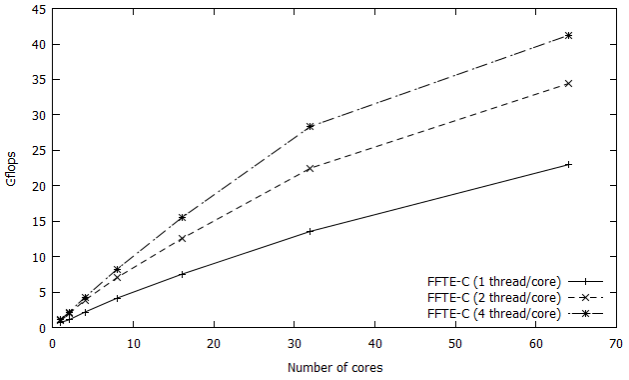

図 5　3 次元 real FFT の性能 ($n = 512^3$)
Fig. 5　Performance of 3d real FFTs ($n = 512^3$)

3 次元 FFT の計算は複数組の 1 次元 FFT と転置操作によって構成されるため，Six-step FFT と基本の動作は同じである．よって，Six-step FFT と同様に転置操作のメモリアクセスに最適化の余地があることが考えられる．

また，コア数およびハイパースレッディングの影響を見るため，以下のパラメータの範囲で FFTE-C の 3 次元 FFT の性能を測定した．

- 問題サイズ n は 512^3 で固定
- コア数=1,2,4,8,16,32,64
- 1 コアあたりのスレッド数=1,2,4

測定結果を**図 5** に示す．

3 次元 FFT においても，ハイパースレッティングが効果的に作用していることが確認された．

4.6　Transpose ベンチマーク

Six-step FFT の実装では 2 次元に配置した複素数を転置する操作が 3 回必要となる．一般的に転置操作はメモリアクセスで性能が抑えられることが知られている．特に，実装上，連続アクセスとストライドアクセスを併用するため，キャッシュメモリ帯域を使い切ることが難しい．また，SIMD 命令やスレッド並列などを効率よく組み合わせた実装方法にも議論の余地がある．高橋の研究では，単純な 2 重ループ構造では不足する並列度に対して，ブロック化手法により 4 重ループ化し外側 2 重ループの一重化により KNL などのメニイコアでのスレッド並列化に対応する手法を紹介している [4]．我々は，同様の手法でメニイコア向けの高スレッド並列化を進めるとともに，個々のスレッド単位での転置操作でのキャッシュと SIMD 処理が適切になされるように，4 重ループ構造に対して 6 重ループ化を施す．6 重ループ化を施した転置操作のコードを**図 6** に示す．

ここで，外側 2 重ループでのブロック幅 (NBX) と内側 2 重ループのブロック幅 (NB)，さらに再内側ループでのベクトル処理の有無の自由度がある（ブロック化は正方でなく長方形でもよいが，今研究ではパラメータ探索のコストを削減するため正方に限定する）．

以下のパラメータの範囲で性能を測定した．

- 行列サイズ $n = 16384^2, 1024^2$
- NBX=8,16,24,32,40,48,56
- NB は 8 で固定
- Directive novector | vector (ivdep+simd) | streaming (ivdep+simd+nontemporal)

$n = 16384^2$ の場合の測定結果を**図 7** に，$n = 1024^2$ の場合測定結果を**図 8** に示す．

$n = 1024^2$ は，LLC に収まる最大サイズである．SIMD 化の有無では性能は変わらず，MBX の値が性能に大きく寄与する結果となった．測定結果から，LLC に収まる範囲では 200GB/s 収まらない場合は 120GB/s 程度までメモリ帯域を使用することができることが分かった．MCDRAM の帯域は 400GB/s であるので，transpose では MCDRAM の 30 %程度しか利用できておらず最適化の余地は残っている可能性はある．

4.7　STREAM ベンチマーク

STREAM ベンチマークはメモリ帯域を測定するものである．PRACE のホームページ上で公開される KNL に関する技術において，Triad の性能が報告されている [9]．我々も同様のパラメータ設定において Triad ベンチマークを実施ほぼ同様の結果を得ている．測定結果を**図 9** に示す．

表中の DDR4|MCDRAM は測定に使用したメモリを示しており，always|never は streaming 命令を常に使用するか，全く使用しないかの指定である．結果からわかるように，MCDRAM に関しては順アクセスを行えばほぼその帯域 400GB/s を使い切ることができることがわかる．LLC 以下のデータに関しては，streaming 命令なしの状況で cache バンド幅が 800GB/s 以上の高い性能を示すことが確認できる．ベンチマーク以前に予測されていはいたが，MCDRAM の容量 16GB に収まる範囲であれば，その高い転送性能を確かに使い切ることができ高速化に大いに

```
void transpose_kernel( int n, int m, complex *Src, int ld_src, complex *Dst, int ld_dst ) {
  for(j=0; j<n; ++j) {
    for(i=0; i<m; ++i) {
      *(Dst+i+ld_dst*j) = *(Src+j+ld_src*i);
    }
  }
}

void transpose_local( int n, int m, complex *Src, int ld_src, complex *Dst, int ld_dst) {
  for(i_=0; i_<m; i_+=NB) {
    for(j_=0; j_<n; j_+=NB) {
      int    m_=int_min(i_+NB,m) - i_;
      int      n_=int_min(j_+NB,n) - j_;
      transpose_kernel( n_, m_, Src+j_*ld_src*i_, ld_src,Dst+i_*ld_dst*j_, ld_dst );
    }
  }
}

void transpose( int n, int m, complex *Src, int ld_src, complex *Dst, int ld_dst ) {
  #omp parallel for collapse(2)
  for(i_=0; i_<m; i_+=NBX) {
    for(j_=0; j_<n; j_+=NBX) {
      int m_=int_min(i_+NBX,m) - i_;
      int n_=int_min(j_+NBX,n) - j_;
      transpose_local( n_, m_, Src+j_*ld_src*i_, ld_src,Dst+i_*ld_dst*j_, ld_dst );
    }
  }
}
```

図 6　転置操作のコード
Fig. 6　Code of transpose.

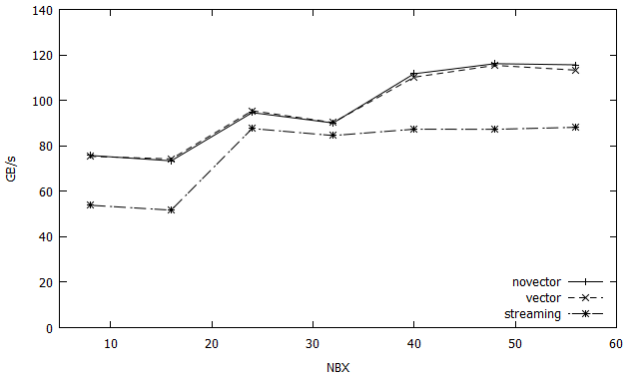

図 7　転置操作の性能 ($n = 16384^2$)
Fig. 7　Performance of transpose ($n = 16384^2$).

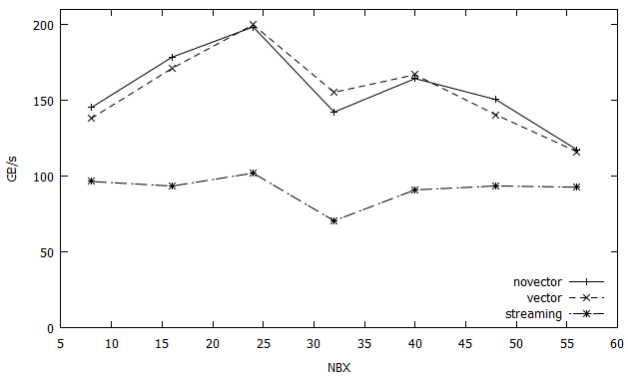

図 8　転置操作の性能 ($n = 1024^2$)
Fig. 8　Performance of transpose ($n = 1024^2$).

寄与することが示される．

5. おわりに

本稿では，FFTE を C 言語に移植し，それを用いて実装した 1 次元 FFT (Six-step FFT) と 3 次元 FFT のプログラムに AVX-512 によるベクトル化および OpenMP 並列化を実施した．そして，FFTE-C，FFTW および MKL を用いた 1 次元 FFT および FFTE-C を用いた 3 次元 FFT を Intel Xeon Phi 7210 上で実行し，その性能を測定した．また，転置操作について性能を測定する Transpose ベンチマークプログラムおよびメモリ帯域を測定する STREAM ベンチマークプログラムも実行し，性能評価を行った．

結果として，Six-step FFT ならびに多次元 FFT の性能に大きく影響を与える転置操作において最適化の余地の可能性が確認された．今後は，メモリ帯域使用の最適化について検討する予定である．また，組み込み SIMD 関数による最適化についても今後研究を進めていく予定である．

謝辞　本研究は科学研究費補助金 基盤研究 (B)15H02709 の支援を得て実施しています．

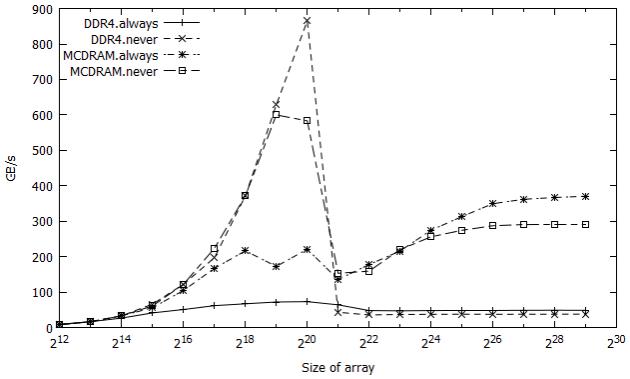

図 9　STREAM ベンチマーク (Triad) の測定結果
Fig. 9　Results of STREAM benchmark (Triad).

参考文献

[1] Cooley J.W. and Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series *Mathematics of Computation*, vol. 19, pp. 297–301 (1965).
[2] Deslippe, J., Jornada, F.H., Vigil-Fowler, D., Barnes, T., Wichmann, N., Raman, K., Sasanka, R. and Louie, S.G.: *Optimizing Excited-State Electronic-Structure Codes for Intel Knights Landing: A Case Study on the BerkeleyGW Software*, High Performance Computing: ISC High Performance 2016 International Workshops, Proceedings of ISC . . . Springer International Publishin, Cham, pp. 402–414 (2016).
[3] Wende, F., Marsman, M. and Steinke, T.: *On Enhancing 3D-FFT Performance in VASP*, CUG Proceedings (2016)
[4] Takahashi, D.: *An Implementation of Parallel 1-D Real FFT on Intel Xeon Phi Processors*, Computational Science and Its Applications – ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part I, Springer International Publishing, Cham, pp. 401–410 (2017).
[5] Takahashi, D.: FFTE: A Fast Fourier Transform Package available from ⟨http://www.ffte.jp/⟩ (updated 2014).
[6] Frigo, M. and Johnson, S.G.: *The Design and Implementation of FFTW3*, Proceedings of the IEEE, vol. 93, pp. 216-231 (2005).
[7] Intel Math Kernel Library Developer Reference (2017). available from ⟨https://software.intel.com/sites/default/files/managed/5e/1b/mkl-2017-developer-reference-c.pdf⟩
[8] Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*, Frontiers in Applied Mathematics, SIAM, Philadelphia (1992).
[9] Best Practice Guide Knights Landing, January 2017 available from ⟨http://www.prace-ri.eu/best-practice-guide-knights-landing-january-2017/⟩ (2017)