Vol.2019-HPC-170 No.35
2019/7/26

ヤコビ回転カーネルを用いたヤコビ固有値計算手法の性能評価

工藤 周平[1,a)]　今村 俊幸[1]

概要：ヤコビの固有値計算手法（ヤコビ法）は計算精度の面で他手法と比べ優れるが，演算量が大きいため，極めて並列度が高いような特殊な状況を除けば，速度面で他手法に大きく劣る．我々はこれまで，ヤコビ法の主要な計算を計算カーネル（ヤコビ回転カーネル）として構成する手法について提案し，計算カーネル自体は高性能に実装可能であることを示したが，ヤコビ法全体を高性能化できるかどうかについては未検証であった．そこで本稿ではヤコビ法全体での性能評価を行い，ヤコビ回転カーネルを用いた実装が用いないものと比べて 30%から 40%程度，実行時間を削減することを示す．また，この結果，他の固有値計算手法と比べても 2 倍程度の実行時間にまで抑えられることが可能となり，我々の手法がヤコビ法の弱点を大きく改善することを示す．

1. はじめに

ヤコビ法は実対象行列の固有値分解手法の一つであり，行列分解（三重対角化）を前処理として用いる他手法と比べて良い誤差上界をもつ点 [2] で優れているが，一方で計算時間についてはそれらの手法と比べて一般的に大きいという問題をもつ．

ヤコビ法の主要な計算は行列に対するヤコビ回転（平面回転，または Givens 回転）を繰り返し適用することであり，この部分が計算時間の大半を占める．そこで通常は，回転を部分的に小行列へと集積し，計算を行列積へと変換することで，高速な行列積実装を活用する．ただしこの手法は演算量の増大という問題がある．そこで我々は既報 [15] において，ヤコビ回転を含むループ構造を再構築し，行列積に似た計算カーネル構造を抽出することで，ヤコビ回転自体の高性能実装（ヤコビ回転カーネル）を作成した．この手法を用いれば，演算量はそのままに，行列積に近い高い実行効率を実現することが可能となる．また高速平面回転 (FPR) を適用すれば，演算量自体も削減でき，ヤコビ法全体の高性能化につながると考えられる．しかしながら，ヤコビ回転カーネルの実装は高度なチューニングを要し，実装の手間が大きいため，そのコストに見合うほどの効果が得られるのかどうかについて，検討する必要がある．

そこで，本稿では既報で議論できていなかった，ヤコビ回転カーネルを実際にヤコビ法へ組み込んだときの性能について評価する．ただし，単なる性能評価ではなく，次の点で拡充されている．第一に，既報では実装できていなかった，FPR を用いないヤコビ回転カーネルを作成し評価に含んでいる点，第二に，ヤコビ回転を集積した行列の持つ非零構造を利用した行列積高速化手法についての議論を行っている点，第三に，並列化実装手法の議論を含んでいる点である．

本稿での性能測定は**表 1** の環境を用いて行う．これは既報と比べて数が減っているが，コンパイラーのサポートする言語バージョンと，プログラミングや性能測定の容易さから選ばれたものである．しかし，KNL や SKX といった最新のメニーコア CPU が含まれるため，十分有益な結果が得られていると考えている．

本稿は 6 つに分かれている．1 節では本稿の概要と，既報との関係について示した．2 節では，ヤコビ法のアルゴリズムと，既報のヤコビ回転カーネル構成手法，また上述の非零構造の利用法について解説する．3 節では本稿で比較する実装手法について，演算量を用いた性能比較を行う．4 節では本稿の実装で用いる並列化手法について議論する．5 節では上記の計算機環境における性能評価結果について示す．そして最後の節で本稿をまとめる．

2. ヤコビ法のアルゴリズム

2.1 ヤコビ法

簡単のため，行列 X の i, j 要素を小文字の $x_{i,j}$ と表記する．対角化する n 次元の実対称行列を A とおく．ヤコビ

[1] 理化学研究所 計算科学研究センター
R-CCS, Kobe, Hyogo 657–0046, Japan
[a)] shuhei.kudo@riken.jp

表 1 The specifications of tested platforms and CPUs

	model	freq.	SIMD	cores	$ size	GFlop/s	compiler
HSW	Core i7-4790	3.6 GHz	256	4	9 MiB	230.0	icc
KNL	Xeon Phi 7210	1.1 GHz	512	64	32 MiB	2252.8	19.0.1.
SKX	Xeon Gold 6126	2.3 GHz	512	12×2	64 MiB	1766.4	144

```
procedure JevSweep(n, {a_·,·}, {v_·,·}, P, {r_·,·,·})
  for (p, q) ∈ P
    if a_{p,q} has large magnitude then
      Compute c and s using Eq.2.
      r_{1,p,q} ← c;  r_{2,p,q} ← s;
      for i = 1 to p − 1
        t ← a_{i,p}, a_{i,p} ← ct − sa_{i,q}, a_{i,q} ← st + ca_{i,q};
      for I = p + 1 to q − 1
        t ← a_{i,p}, a_{i,p} ← ct − sa_{q,i}, a_{q,i} ← st + ca_{q,i};
      for i = q + 1 to n
        t ← a_{p,i}, a_{p,i} ← ct − sa_{q,i}, a_{q,i} ← st + ca_{q,i};
      for i = 1 to n
        t ← v_{i,p}, v_{i,p} ← ct − sv_{i,q}, v_{i,q} ← st + cv_{i,q};
```

図 1 A pseudo code for a single sweep of the Jacobi method

法では初期値 $A^{(0)} := A$, 初期固有ベクトル $V^{(0)} := I_{n,n}$ とおき，ある回転軸 $(p^{(k)}, q^{(k)})$ と角度 $\theta^{(k)}$ に関するヤコビ回転 $G^{(k)} := G(p^{(k)}, q^{(k)}, \theta^{(k)})$ を反復的に適用する：

$$A^{(k+1)} = \left(G^{(k)}\right)^\top A^{(k)} G^{(k)}, \quad V^{(k+1)} = V^{(k)} G^{(k)}. \tag{1}$$

ヤコビ回転は次の 4 つの要素を除き単位行列と一致する直交変換である：$g^{(k)}_{p^{(k)},q^{(k)}} = g^{(k)}_{q^{(k)},q^{(k)}} = c^{(k)} := \cos\theta^{(k)}$, $g^{(k)}_{p^{(k)},q^{(k)}} = -g^{(k)}_{q^{(k)},p^{(k)}} = s^{(k)} := \sin\theta^{(k)}$. $\theta^{(k)}$ は $A^{(k)}$ の $(p^{(k)}, q^{(k)})$ 要素を消去するよう設定する．具体的には：

$$\tan 2\theta^{(k)} = \frac{2a^{(k)}_{p^{(k)},q^{(k)}}}{a^{(k)}_{p^{(k)},p^{(k)}} - a^{(k)}_{q^{(k)},q^{(k)}}}, \quad \left|\theta^{(k)}\right| \le \frac{\pi}{4}. \tag{2}$$

このとき，ヤコビ回転の適用前後で非対角要素の二乗和が $2\left(a^{(k)}_{p^{(k)},q^{(k)}}\right)^2$ だけ減少するため，$A^{(k)}$ は対角行列へと収束していく．回転軸の選び方は次小節にて議論する．

ヤコビ回転は行列の 2 列（行）のみを変化させ，具体的には右側回転のとき $i = 1, \ldots, n$ について

$$v^{(k+1)}_{i,p^{(k)}} = c^{(k)} v^{(k)}_{i,p^{(k)}} - s^{(k)} v^{(k)}_{i,q^{(k)}}, \tag{3}$$
$$v^{(k+1)}_{i,q^{(k)}} = s^{(k)} v^{(k)}_{i,p^{(k)}} + s^{(k)} v^{(k)}_{i,q^{(k)}}. \tag{4}$$

左側についても同様となる．そこで，ヤコビ回転適用の演算量は，$V^{(k)}$ の更新については $6n$，$A^{(k)}$ の更新については，$A^{(k)}$ の対称性を用いると $6(n-2)$ となる．これは，高速平面回転 (Fast Plane Rotation, FPR) [4] によって削減可能である．

FPR のアイデアは, 回転を縮小成分と平行四辺形ゆがみ成分とに分解し, 両者による変換を個別に追跡することである．いま対角スケーリング成分 $D^{(k)} = \mathrm{diag}(d^{(k)}_1, d^{(k)}_2, \ldots, d^{(k)}_n)$ によって $V^{(k)} = U^{(k)} D^{(k)}$ と分解する．ただし初期値は

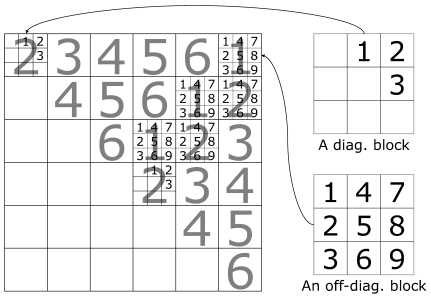

図 2 An example of the ordering for $n = 8$ and $b = 3$.

$D^{(0)} = I_{n,n}$．このとき，$V^{(k)}$ に対する回転 $G^{(k)}$ を，$U^{(k)}$ に対する変換 $\hat{G}$ と $D^{(k)}$ の対角成分に対する変換とに分離することで，前者の演算量を減らす：

$$U^{(k)} D^{(k)} G^{(k)} = \left(U^{(k)} \hat{G}\right) D^{(k+1)} = U^{(k+1)} D^{(k+1)}. \tag{5}$$

$D^{(k+1)}$ は $\hat{G}$ の対角成分が 1 となるものを選ぶ:

$$\begin{bmatrix} d^{(k)}_{p^{(k)}} & \\ & d^{(k)}_{q^{(k)}} \end{bmatrix} \begin{bmatrix} c^{(k)} & s^{(k)} \\ -s^{(k)} & c^{(k)} \end{bmatrix} = \begin{bmatrix} 1 & \hat{g}_{p^{(k)},q^{(k)}} \\ \hat{g}_{p^{(k)},q^{(k)}} & 1 \end{bmatrix} \begin{bmatrix} d^{(k+1)}_{p^{(k)}} & \\ & d^{(k+1)}_{q^{(k)}} \end{bmatrix}, \tag{6}$$

ここで $\hat{g}_{p^{(k)},q^{(k)}} = \frac{s^{(k)} d^{(k)}_{q^{(k)}}}{c^{(k)} d^{(k)}_{p^{(k)}}}$, $\hat{g}_{q^{(k)},p^{(k)}} = \frac{-s^{(k)} d^{(k)}_{p^{(k)}}}{c^{(k)} d^{(k)}_{q^{(k)}}}$, $d^{(k+1)}_{p^{(k)}} = d^{(k)}_{p^{(k)}} c^{(k)}$, そして $d^{(k+1)}_{q^{(k)}} = d^{(k)}_{q^{(k)}} c^{(k)}$．この変形によって $U^{(k)}$ の更新に必要な演算量は $4n$ となり，元の演算量の 2/3 倍となる．また演算パターンは xAXPY となり，積和演算を利用しやすい．ただし FPR には値の拡大によるオーバーフローの危険性が知られている．我々の既報 [15] では，オーバーフローの直前で安全な動作に切り替える簡易的な手法についての議論がある．

図 1 にヤコビ法の疑似コードを示す．疑似コードは与えられた回転軸の列 P に従い計算を進め，同時に回転角を R に格納する．収束判定などを含めたより現実的なコード例については Demmel [2], Alg. 3.1 などを参照されたい．このコードを FPR 用に変更することは容易である．また，V が不必要な場合はその計算をしないように変更すると演算量をさらに半減できる．

2.2 巡回順序

回転軸の選択はヤコビ法の性質を大きく変える．Jacobi による初期の手法 [8] では局所最大に行列を対角へと近づける軸を探索するが，探索自体に計算量がかかる．軸の選択順序を事前に決め，同じ順序を繰り返し用いることで探

```
procedure BJevSweep(N, b, {A_{I,J}}, {V_{I,J}}, P_B)
  for (P, Q) ∈ P_B
    if P == Q then
      V̂ ← I_{b,b};
      ♣ JevSweep(b, A_{P,P}, V̂, R̂, P_in);      ▷ on triu(A_{P,P})
      ♦ A_{I,P} ← A_{I,P}V̂ for I = 1 to P − 1;
      ♦ A_{P,I} ← V̂^⊤ A_{P,I} for I = P + 1 to N;
      ♦ V_{I,P} ← V_{I,P}V̂ for I = 1 to N;
    else
      Â_{1,1} ← A_{P,P}; Â_{1,2} ← A_{P,Q}; Â_{2,2} ← A_{Q,Q}, V̂ ← I_{2b,2b};
      ♣ JevSweep(b, Â, V̂, R̂, P_out);            ▷ on Â_{1,2}
      A_{P,P} ← Â_{1,1};   A_{P,Q} ← Â_{1,2};   A_{Q,Q} ← Â_{2,2};
      ♦ [A_{I,P}, A_{I,Q}] ← [A_{I,P}, A_{I,Q}]V̂ for I = 1 to P − 1;
      ♦ [A^⊤_{P,I}, A_{I,Q}] ← [A^⊤_{P,I}, A_{I,Q}]V̂ for I = P + 1 to Q − 1;
      ♦ [A^⊤_{P,I}, A^⊤_{Q,I}] ← [A^⊤_{P,I}, A^⊤_{Q,I}]V̂ for I = Q + 1 to N;
      ♦ [V_{I,P}, V_{I,Q}] ← [V_{I,P}, V_{I,Q}]V̂ for I = 1 to N;
```

図 3　A pseudo code for a single sweep of the blocked Jacobi method

索の手間を省く手法があり，巡回ヤコビ法と呼ばれる．この順序のことを巡回順序と呼ぶ．巡回順序はヤコビ法の収束性や計算性能に影響を与える．並列巡回順序は連続する 2 つの回転軸が重ならないものを含む順序であり，そのような回転軸に対する計算を並列に行える．ブロック指向順序は，ブロック行列のブロック内部に対する消去が連続して行われる順序である．

我々が用いた順序の例を図 **2** に示す．この順序はブロック指向順序の 1 つであり，ブロック内部に列巡回順序 [3]，ブロックの選択に並列巡回順序の 1 つであるモジュロ順序 [11] を用いたものである．これによって異なる 2 つ以上のブロックを同時に巡回可能とする．この組み合わせは Schroff ら [13] のブロック弱ウェーブフロント順序となっており，これはもっとも単純な巡回順序である列巡回順序と収束性を共有する [13], Th. 5. また，並列化の必要のない単体コア上での実行の場合，ブロックの選択はモジュロ順序に代えて行巡回順序を用いている．これは列巡回順序と厳密に同値なブロックウェーブフロント順序となっている [13], Th. 3.

2.3　ブロックヤコビ法

ブロック指向順序において，ブロック内部の計算と，他のブロックの計算を分離することができる；非対角要素の消去や行・列の更新におけるデータ依存は，一部のブロック中に閉じているためである．いま行列 $A^{(k)}$ と $V^{(k)}$ を $N \times N$ 個ブロックへ分割し，それぞれ $A^{(k)} = \{A^{(k)}_{I,J}\}$, $V^{(k)} = \{V^{(k)}_{I,J}\}$ と表記する．個々のブロックの大きさは簡単のためすべて $b \times b$ とし，$n = Nb$ が成り立つ場合を考える．ブロック指向順序においてあるブロック軸 (P, Q) を選択したとき，(P, Q) に関するピボット行列 $\hat{A} := \begin{bmatrix} A_{P,P} & A_{P,Q} \\ A_{Q,P} & A_{Q,Q} \end{bmatrix}$ を定義する．ブロック指向順序では $\hat{A}$ の内部を巡回し，$\hat{A}$ を更新すると同時に，回転を $\hat{V}$ へ蓄積する．つまり内部で用いたヤコビ回転を $\hat{G}_1, \hat{G}_2, \ldots$ とおくと，$\hat{V} = \hat{G}_1\hat{G}_2\cdots$ が成り立つ．この計算をピボット巡回と呼ぶ．$\hat{V}$ は後に残りのブロックへと適用する．この計算をブロック更新と呼ぶ．

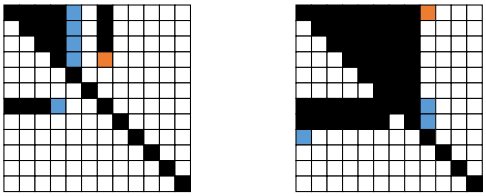

図 4　An illustration of the fill-ins in the computation of $\hat{V}$

図 **3** にブロックヤコビ法の疑似コードを示す．コード中，“♣” がピボット巡回，“♦” がブロック更新に相当する．コードでは非対角ブロックと対角ブロックの処理を分けている[*1]が，これは巡回順序の違いにより手順が大きく異なるためである．コードにおいて P_B はブロックの選択順序，P_{in} や P_{out} はブロック内部の巡回順序であり，図 2 に対応するものとなっている．

ブロックヤコビ法の演算時間はピボット巡回とブロック更新に支配される．前者の演算量自体は全体の $O(\frac{b}{n})$ の割合であり，N が十分大きければ無視できる．実際にはピボット巡回の演算効率が低いため，影響は小さくない．後者は演算量自体が大きいため，高性能化が不可欠である．通常はコード中のように行列積 xGEMM を使えるため，簡単に高性能実装を得られる．ただし詳細に解析すれば $\hat{V}$ の非零構造を用いて演算量を削減できることがわかる．これについては次小節で議論する．また，$\hat{V}$ を生成した後に行列積を用いるよりも，$\hat{G}_1, \hat{G}_2, \ldots$ を行列に直接適用する方が性能面での利点がいくつかある．そこでヤコビ回転適用の高性能化について § 2.5 で議論する．また，手法の組み合わせによる性能への影響を次節で議論する．

ピボット巡回の計算は，部分行列の近似固有値分解だととらえることもできる．そこで，ピボット巡回の代わりに，厳密な固有値分解を行うことも考えられる．この手法をフルブロックヤコビ法と区別して呼ぶ．フルブロックヤコビ法はヤコビ法のアナロジーとなっているが，数値的には別物であり，収束性の議論や収束のための修正など，複雑となるため [6]，本稿では対象としていない．

2.4　$\hat{V}$ の非零構造を用いる手法

$\hat{V}$ は単位行列に対して限られた回数のヤコビ回転を特定の順序でかけたものとなっており，正確に計算手順を追えば，規則的な非零構造を見つけることができる．図 **4** は $\hat{V}$ の計算途中で発生するフィルインの様子を図示している．図中，橙色のブロックは回転軸に対応する要素，水色のブロックはフィルインが発生した要素を表している．右上ブロックにおける列巡回の最初の 1 列の処理では左上ブロッ

[*1] 対角ブロックは非対角要素を含むことに注意．

```
procedure ApplyR(b, {x_{·,·}}, {y_{·,·}}, {r_{·,·,·}})
  for i = 1 to b
    for j = 1 to b
      for k = 1 to b
        t ← x_{k,j};
        x_{k,j} ← r_{1,j,i} t − r_{2,j,i} y_{k,i};
        y_{k,i} ← r_{2,j,i} t + r_{1,j,i} y_{k,i};
```

図 5 A pseudo code of the application of the rotations.

クの上三角が満たされ，左下ブロックは 1 行のみ満たされる．2 列目以降の巡回によって 1 列ずつ右下ブロックの上三角部分が満たされていく．この結果，巡回の最後には上三角行列の左下を密にした形を得る．この構造をブロックヤコビ法に用いた例は筆者の知る範囲にはないが，Ltaiefa らの二段階 Hessenberg 分解を Givens QR を基礎として行う手法において出現している [10], fig. 2.

この非零構造を用いることで演算量をほぼ 3/4 倍に削減できる．ただしこのような構造を直接扱う BLAS ルーチンは存在しないため，上三角部分と小正方部分とに分けて計算する必要がある．前者は Level-3 BLAS の xTRMM を使えば計算でき，後者は単純な xGEMM となる．また演算量の削減分ほど実際の性能を向上させるかについては別問題であり，xTRMM のチューニング度合いに依存する．

2.5 ヤコビ回転カーネル

ブロック更新において直接ヤコビ回転を適用する場合，その計算は図 **5** のように記述できる．これは長さ b の三重ループであり，行列積と類似した構造を持つ．再内側計算は積+積和であるが，FPR を用いる場合は単純な積和となり，より行列積と似る．そこで我々は既報 [15] において，Goto らの行列積高性能実装手法 [5] をブロック更新へ応用することを提案した．

図 **6** は最適化後の疑似コードを示している．このコードは 2 つに分けられており，ループブロック化後の外側ループ (Driver) と，固定サイズの内側ループ (Kernel) である．Driver は，ブロックをさらに細かく分割した部分行列であるパネルを選択し，バッファへと並び替えた後に，並び替えたデータを用いて Kernel を呼び出す．並び替え時は，ブロック更新の位置に応じて転置も行う．Kernel は与えられたパネルのペアに対して回転を適用する．Kernel は計算時間の大部分を占めることになるため，高度なチューニングに値する．チューニング戦略の核は，パネルをレジスターに置くことで演算回数に対するキャッシュやメモリーへのアクセス回数を減らすことである．ここではデータストアの回数を減らすために，図中 x をレジスターに置く．

以上の構成により，ブロック幅に比例するデータ再利用回数を実現でき，演算密度を高められ，行列積に似た高性能を実現できる．さらに FPR を用いれば演算量の面でも利点がある．表 **2** に 3 つの CPU 上での Kernel 部分の性能を示す．表では FPR を用いたときの性能と，通常のヤコビ回転を用いたときの性能 (Regular) の両方を示している．どの CPU においても 1 コアの理論性能の 8 割以上に達している．ただし，Regular では積と和の比率の問題から，理論性能が CPU の演算性能の 2/3 となっていることに注意．また，KNL 上の FPR のみ 6 割程度の性能となっているが，これは命令実行幅が他の 2 つの CPU と比べて狭いことが原因である．詳細なチューニングや性能解析については既報 [15] を参照されたい．

```
procedure Driver(b, X_{·,·}, Y_{·,·}, R_{·,·,·})
  for J = 0 to n_J(⌈b/b_J⌉ − 1), step n_J
    for i = 1 to ⌈b/b_i⌉
      for k = 1 to ⌈b/b_k⌉
        Load Y_{k,i} into y;
        for j = J + 1 to J + n_J
          Load X_{k,j} into x;
          kernel(x, y, R_{·,i,j})
          Store x back to X_{k,j};
        Store y back to Y_{k,i};
procedure Kernel(x_{·,·}, y_{·,·}, r_{·,·,·})
  Load x_{·,·} to the SIMD registers;
  for i = 1 to b_i        ▷ unrolled by 1–4. b_i can be a variable
    Load y_{·,i} to the SIMD registers;
    for j = 1 to b_j                                  ▷ full-unrolled
      for k = 1 to b_k                ▷ full-unrolled and SIMDized
        t ← x_{k,j}
        x_{k,j} ← r_{1,j,i} t − r_{2,j,i} y_{k,i}
        y_{k,i} ← r_{2,j,i} t + r_{1,j,i} y_{k,i}
    Store y_{·,i} back;
  Store x_{·,·} back;
```

図 6 A pseudo code for an optimized code for the rotations

表 **2** The performance of the kernels for the rotations

	GFlop/s (FPR)	GFlop/s (Regular)
HSW	49.5 ± 0.0	36.5 ± 0.0
KNL	20.6 ± 0.3	23.3 ± 0.2
SKX	99.0 ± 0.3	81.7 ± 0.7

3. ブロック更新手法の比較

ここでは 4 つのブロック更新手法の比較を行う．1 つは行列積を使うもの (*MM*)，もう 1 つは非零構造を使うもの (MMsp)，そして残り 2 つはヤコビ回転カーネルを用いるものであり，通常の回転を用いるもの (*Regular*) と FPR を用いるもの (*FPR*) とがある．これらの手法はブロック更新における演算量が異なると同時に，ピボット巡回における $\hat{V}$ の計算の要不要も異なる．ブロック更新は演算量の大部分を占めるためより重要であるが，ピボット巡回の実行効率は低いため，単純な演算量の数値以上に影響を及ぼす．そこで両者を分けて比較を行う

表 **3** に各手法の 1 ステップあたりの演算量を示す．ここでは演算量の大部分を占める非対角ブロックの更新 ($P \neq Q$ の場合) のものを示している．表中の各列では，ピボット

表 3 Computational costs for a single sweep over an off-diagonal block

	sweep $\hat{A}$	update A	update V	total
FPR	$8b^3 + O(b^2)$	$4nb^2 - 8b^3$	$4nb^2$	$8nb^2 + O(b^2)$
Regular	$12b^3 + O(b^2)$	$6nb^2 - 12b^3$	$6nb^2$	$12nb^2 + O(b^2)$
MM	$24b^3 + O(b^2)$	$8nb^2 - 16b^3 + O(nb)$	$8nb^2 + O(nb)$	$16nb^2 + 8b^3 + O(nb)$
MMsp	$24b^3 + O(b^2)$	$6nb^2 - 12b^3 + O(nb)$	$6nb^3 + O(nb)$	$12nb^2 + 12b^3 + O(nb)$

巡回 (sweep $\hat{A}$) と A に関するブロック更新 (update A), V に関するブロック更新 (update V), そしてそれらの合計 (total) を示している．FPR はピボット巡回の演算量が少なく，また合計の演算量も少ない．そこで他手法と同程度の実行効率を実現できれば最も速くなると予測できる．Regular は FPR に続いて，ピボット巡回の演算量と合計の演算量が小さい．ただし Regular は積と和の比率の問題に影響されるため，演算量の数値ほどの性能とはならない．MM や MMsp は $\hat{V}$ の計算が必要であるため，ピボット巡回の演算量が大きい．この結果，演算量の合計にオーバーヘッド項 $O(b^3)$ として表れている．MM は演算量が最も大きく，FPR の 2 倍以上となる．MMsp は非零構造のため MM の 3/4 程度となるが，実際の性能は Level-3 BLAS のチューニング状況に依存する．

以上の通りヤコビ回転を行列に直接かける 2 手法，FPR と Regular は演算量削減に効果があるだけでなく，オーバーヘッド項の削減にも役立つ．しかしながら，FPR や Regular が実際に高性能となるためにはヤコビ回転カーネルの高度なチューニングが不可欠である．一方で MM や MMsp は BLAS のルーチンを利用可能という利点があり，ポータビリティーの面で優れている．そこで，FPR や Regular が実装の手間に匹敵するほどの性能面での利点があるかどうかについて調査することが本研究の目的である．

4. ブロックヤコビ法のタスク並列化

我々は単体コアでの性能測定だけでなく，より現実的な結果を得るために，スレッド並列化した実装を作成した．ブロックヤコビ法はブロック単位で記述された明快な構造を持っているため，並列化もブロック単位で行えばわかりやすい．我々は，ブロック単位の処理をタスクとみなし，OpenMP のタスク並列化 [12], § 2.10, とくに task depends 節 [12], § 2.17.11 を用いたランタイムの DAG スケジューリングによって並列化を行った．このスケジューリング手法は，記述が単純になるにもかかわらず，利用の難しい隠れた並列性を自動的に発見し性能向上につなげられるというメリットがある．

図 7 にブロックヤコビ法における処理の依存関係の例を示す．この図ではピボット巡回と，A についてのブロック更新という 2 種類の計算について示しており，ピボット巡回を行うブロックを橙色，ブロック更新を行うブロックを青色に表示している．ピボット巡回は $\hat{V}$ または回転角の

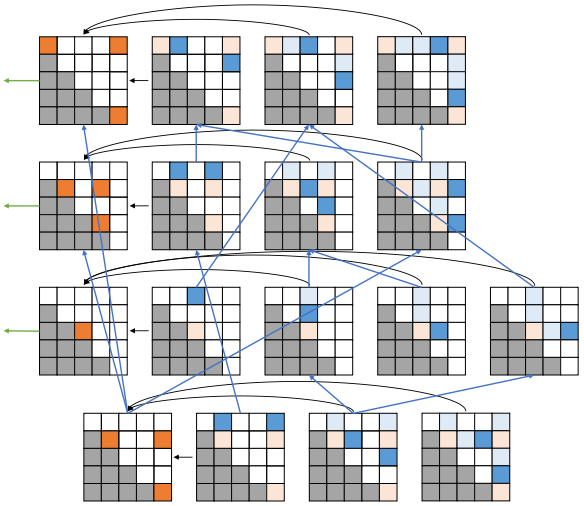

図 7 An illustration of the task dependency for the first 4 steps of the blocked Jacobi sweep on 5×5 blocks

列 R を生成し，ブロック更新はそれをもとに非対角ブロックを更新する．また次のステップのピボット巡回は，更新済みのブロックを用いることがある．図中，黒矢印が $\hat{V}$ または R についての，青矢印が非対角ブロックについての Flow 依存を示している．また端点のない緑矢印は処理の開始点を表す．図からわかるように，ブロックヤコビ法では巡回順序によっては複数のピボット巡回を同時実行可能である（並列巡回順序）．ただし複数のピボットに関するブロック更新の間には依存関係が一部に生じる．

このような複雑な依存関係を解決する専用の機構を作成することは大きな労力を要するが，DAG スケジューリングのような一般解法に任せてしまえば簡単である．ただし DAG スケジューリングはスケジューリングコストが大きいデメリットがあるため，最適なスケジューリング手法の開発は今後の課題である．

5. 性能測定

我々はヤコビ回転カーネルを用いたブロックヤコビ法の性能評価を行うため，次の 3 つのテストを行った．1 つは，単体コア上で各ブロック更新手法を用いたブロックヤコビ法の 1 巡回を行ったときの実行時間の測定であり，純粋な演算性能を比較する目的がある．ここではブロック化していない単純なヤコビ法実装との性能比較も行う．もう 1 つは，複数コアを用いたときの同様の実験であり，より現実的な状況下での性能を測定する目的がある．最後は，収束まで反復を繰り返した場合の実行時間測定で

ある．ここでは他の固有値計算手法との性能比較も行うことで，我々の実装の実用性を検証する．すべてのテストは倍精度浮動小数点数で行い，テスト行列は minij 行列を用いた：$a_{i,j} := \min(i,j)$．BLAS 実装は Intel MKL のものを使用しており，性能比較においては Intel MKL の LAPACK 実装から DSYEV と DSYEVD を用いている．(ブロック) ヤコビ法の収束判定では相対的基準を用いている：$|a_{i,j}^{(k)}| \leq \epsilon\sqrt{n}|a_{i,i}^{(k)}a_{j,j}^{(k)}|$, ただし $\epsilon \approx 1.01 \times 10^{-16}$ は計算機イプシロン．これは Demmel [2] の推奨する基準であるが，今回のテスト行列については正定値ではないため，過剰な基準となっている．すべてのテストは同じ設定を用い 7 回繰り返し，観測値を点，その平均値を折れ線として表示している．表示上の工夫として，はじめの 2 つのテストでは n^3 を実行時間で割った値（正規化性能, n^3/s），最後のテストでは DSYEV の実行時間の平均値との比をプロットしている．前者は適切な係数をかけると GFlop/s 値に換算できる値である；ただし各手法の 1 反復当たりの演算量が異なることに注意．後者はとくに比較に重きを置いたためである．

5.1 1 巡回での正規化性能

図 **8** に単体コアを用いたときの正規化性能を示す．正規化性能は大きな値ほど高速であることを意味する．ここでは行列サイズ $n = 10$ から $2,000$ を用い，ブロック幅 $b = 8$ から 128 の中で最も平均性能が良いものを表示している．図中 serial はブロック化していないが，巡回順序としては他と同じものを使用したヤコビ法である．

serial は小さな行列（$n < 100$ 程度）を除いては最も遅い．逆に最も速いものは FPR であり，HSW や SKX では MM の 2 倍近い性能に達している．ただし KNL の場合はヤコビ回転カーネルの性能を反映して，他の CPU と比べて差が小さくなっている．Regular は FPR と比べて遅いが，MM や MMsp よりも高性能となっている．MMsp は演算量では MM よりも優れているはずだが，実際の実行時間では，ほとんど差がない (HSW) かむしろ MM よりも遅い結果となっている (KNL, SKX)．どの結果においても $n = 2,000$ になっても性能上昇を続けているものがあるが，単体コアでは実行時間が大きく，これ以上大きな行列サイズでのテストはできていない．

図 **9** に CPU 上のすべてのコアを用いた時の正規化性能を示す．ここでは行列サイズ $n = 100$ から $4,000$(HSW), または $10,000$ (KNL, SKX) を用い，ブロック幅 $b = 16$ から 400 までの中で最も平均性能が良いものを表示している．

複数コアを用いた場合でも単体コアを用いたときと性能の傾向は似通っており，FPR が最も速く，続いて Regular, そして MM と MMsp はほぼ同程度だが，HSW を除いては MMsp の方が遅くなっている．

5.2 他の固有値計算手法との性能比較

図 **10** に DSYEV との実行時間の比を示す．この値は小さいほど高速であることを意味する．ここでは行列サイズ $n = 800$ から $10,000$(HSW) または $16,000$(KNL, SKX) を用い，ブロック幅は，2 つ目のテストで最もよかった値を用いている．ただし，行列サイズが 2 つ目のテストの範囲を超えるものについては最も近い行列サイズに対応するブロック幅を用いている．

このテストでは DSYEVD が最も速く，DSYEV の半分以下の実行時間となっている点もある．FPR はブロックヤコビ法の中では最も速く，KNL では DSYEV よりも高速となる点が多数ある．それ以外の CPU においても，HSW では約 2 倍から 3 倍，SKX では 2 倍以下の範囲に入っている．他のブロックヤコビ法の性能傾向は 2 つ目のテストと類似しており，DSYEV からはさらに差を広げられる．ブロックヤコビ法の曲線が $n = 8,000$ 付近を境にして急激に変化しているように見える．これは逆であり，DSYEV や DSYEVD の実行時間がその付近においてなめらかでない変化をしている．簡単な調査によれば，MKL は $n = 7,800$ を境に，内部で呼び出す関数が大きく変化しており，何らかのアルゴリズムの切り替えをしていることが伺えるが，詳細は不明である．

表 **4** に各 CPU ごとの最も大きな行列サイズにおける実行時間を示す．HSW では，FPR によって MM や MMsp から実行時間を 47% 削減している．また KNL でのその数値は 33%, SKX では 41% となっており，演算量の削減幅に近い大きな削減となっている．

6. まとめ

ブロックヤコビ法は部分行列に対するヤコビ回転を小行列 $\hat{V}$ へ集積し，$\hat{V}$ との行列積 xGEMM によって行列を更新することで高性能化を実現するが，ヤコビ回転を直接行列にかける方が，少演算量や低オーバーヘッドなどの利点がある．そこで我々は既報 [15] において，ヤコビ回転自体の高性能化手法（ヤコビ回転カーネル）を検討し，実際に，行列積に近い高い実行効率を実現可能であることを示した．そこで本稿では，ヤコビ回転カーネルをヤコビ法に組み込んだときの性能を検証した．

実験結果から，ヤコビ回転カーネルによって行列積を用いた実装と比べて性能向上することがわかった．とくに，高速平面回転 (FPR) と組み合わせた場合には，行列積をベースとした実装と比べて，30% から 40% 程度の実行時間の削減を実現した．また他の固有値計算手法 (DSYEV, DSYEVD) との性能比較においては，ヤコビ回転カーネルを用いてもブロックヤコビ法はまだそれらの手法と比べて低速であったが，その差を大きく縮め，DSYEV と比べて，2 倍程度の実行時間で計算を完了することに成功している．

我々は，この性能向上はヤコビ法の適用範囲を広げるこ

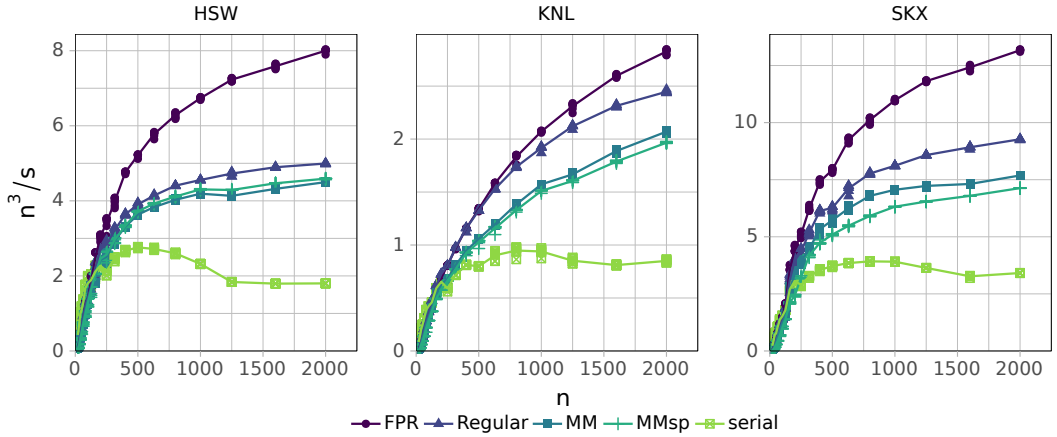

図 8 The performance of implementations using different rotation techniques on the single core of the CPUs

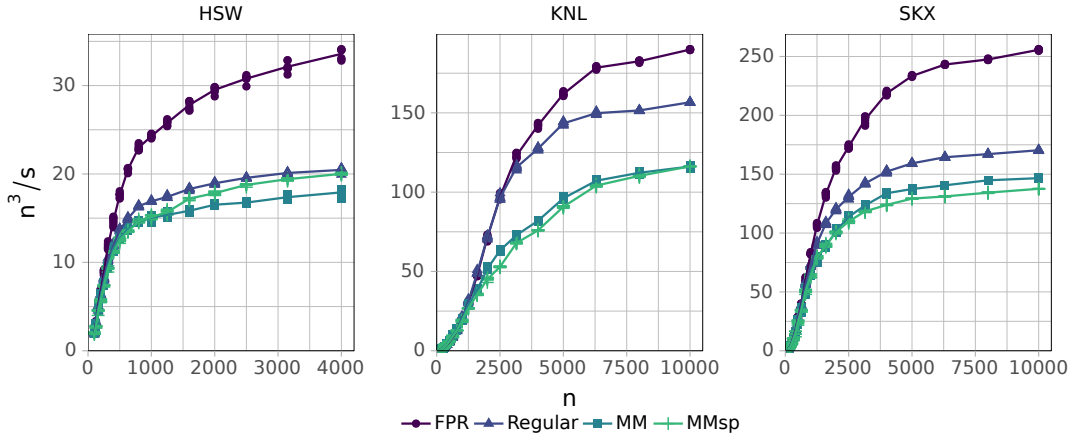

図 9 The performance of implementations using different rotation techniques on the all cores of the CPUs

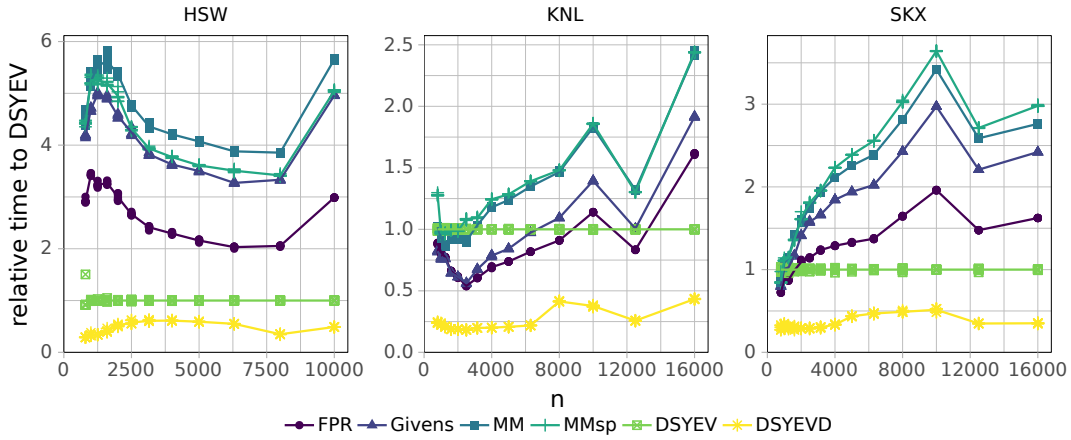

図 10 The relative average time of methods compared with MKL's DSYEV

表 4 The computation timeof the methods in seconds for the largest matrices

	n	FPR	Regular	MM	MMsp	DSYEV	DSYEVD
HSW	$10,000$	360 ± 0.4	597 ± 0.8	681 ± 1.8	608 ± 2.2	120 ± 0.6	58.9 ± 0.5
KNL	$16,000$	296 ± 1.0	352 ± 0.8	446 ± 2.4	448 ± 0.5	184 ± 0.5	79.6 ± 0.5
SKX	$16,000$	211 ± 0.1	314 ± 0.1	358 ± 0.2	386 ± 0.9	130 ± 0.7	45.5 ± 0.5

とに大きく役立つと考えている．ヤコビ法は他の固有値計算手法よりも良い誤差評価を持っているため，より小さなコストでそれを実現できるようになったのは大きな利点である．また，ヤコビ法は反復解法であり，反復回数（巡回回数）の小さな行列に対しては高速に動作する．従来は他手法との性能差が大きく，反復回数のきわめて小さな行列でしか性能面でのメリットがなかったが，今回の性能改善によってより多くの行列に対して性能上のメリットが得られるようになったと考えられる．

今後の課題としては，並列化手法の検討があげられる．本稿では単純化のために OpenMP のタスク並列化を用いた DAG スケジューリングを行ったが，これは，メニーコア CPU のような並列化が難しい環境における高性能化手法として，近年の線形計算ソフトウェアに取り入れられているものである（例えば PLASMA [1]）．そこでこのような最新手法がヤコビ法にも適するか調査することは興味深い．また，本稿の結果は実対称行列向けのヤコビ法に限られるが，同じ手法やアイデアは他のヤコビ様アルゴリズムにも適用できる．例として，片側ヤコビ法（Hestenes 法）[7] や Kogbetliantz 法 [9] は SVD 向けのヤコビ法であるが，同様に多数のヤコビ回転を行い，またブロック化も可能であるため，ヤコビ回転カーネルが役立つ．また，不定値行列に対する高精度なヤコビ法である J-Jacobi 法 [14] は，ヤコビ回転の他に Hyperbolic 回転を行うが，これは符号を除いてヤコビ回転と一致するため，ヤコビ回転カーネルを応用できる．また，ヤコビ回転カーネルは分散メモリー並列向けのヤコビ法実装にも役立つと考えられる．とくに，回転角 R のデータ量は $\hat{V}$ の半分または 1/4 にできるため，通信量の削減にもつながることが興味深い．

謝辞　本研究は JSPS 科研費 JP19H04127 の助成を受けたものです．

参考文献

[1] Buttari, A., Langou, J., Kurzak, J. and Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures, *Parallel Comput.*, Vol. 35, No. 1, pp. 38–53 (online), DOI: 10.1016/j.parco.2008.10.002 (2009).

[2] Demmel, J. W. and Veselić, K.: Jacobi's Method is More Accurate than QR, *SIAM J. Matrix Anal. Appl.*, Vol. 13, No. 4, pp. 1204–1245 (online), DOI: 10.1137/0613074 (1992).

[3] Forsythe, G. E. and Henrici, P.: The cyclic Jacobi method for computing the principal values of a complex matrix, *Trans. Am. Math. Soc.*, Vol. 94, No. 1, pp. 1–23 (online), DOI: 10.1090/S0002-9947-1960-0109825-2 (1960).

[4] Gentleman, W. M.: Least squares computations by givens transformations without square roots, *IMA J. Appl. Math.*, Vol. 12, No. 3, pp. 329–336 (online), DOI: 10.1093/imamat/12.3.329 (1973).

[5] Goto, K. and van de Geijn, R. A.: Anatomy of high-performance matrix multiplication, *ACM Trans. Math. Softw.*, Vol. 34, No. 3, pp. 1–25 (online), DOI: 10.1145/1356052.1356053 (2008).

[6] Hari, V.: Convergence to diagonal form of block Jacobi-type methods, *Numer. Math.*, Vol. 129, No. 3, pp. 449–481 (online), DOI: 10.1007/s00211-014-0647-8 (2014).

[7] Hestenes, M. R.: Inversion of Matrices by Biorthogonalization and Related Results, *J. Soc. Ind. Appl. Math.*, Vol. 6, No. 1, p. 51 (1958).

[8] Jacobi, C. G. J.: Über ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen., *J. für die reine und Angew. Math.*, Vol. 30, pp. 51–94 (1846).

[9] Kogbetliantz, E. G.: Solution of linear equations by diagonalization of coefficients matrix, *Q. Appl. Math.*, Vol. 13, No. 2, pp. 123–132 (online), DOI: 10.1090/qam/88795 (1955).

[10] Ltaief, H., Kurzak, J. and Dongarra, J.: Parallel Two-Stage Hessenberg Reduction using Tile Algorithms for Multicore Architectures, pp. 1–26 (online), DOI: 10.1.1.304.8916.

[11] Luk, F. T. and Park, H.: A Proof of Convergence for two Parallel Jacobi SVD Algorithms, *IEEE Trans. Comput.*, Vol. 38, No. 6, pp. 806–811 (online), DOI: 10.1109/12.24289 (1989).

[12] OpenMP Architecture Review Board: OpenMP Application Programming Interface (2018).

[13] Shroff, G. and Schreiber, R. S.: On the convergence of the cyclic Jacobi method for parallel block orderings, *SIAM J. Matrix Anal. Appl.*, Vol. 10, No. 3, pp. 326–346 (1989).

[14] Veselić, K.: A Jacobi eigenreduction algorithm for definite matrix pairs, *Numer. Math.*, Vol. 64, No. 1, pp. 241–269 (online), DOI: 10.1007/BF01388689 (1993).

[15] 工藤周平，今村俊幸：高い演算密度をもつヤコビ回転カーネルの構成手法，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 20, pp. 1–9（オンライン），入手先 ⟨http://id.nii.ac.jp/1001/00194702/⟩ (2019).