Xeon PhiにおけるDSYRKの並列化手法と性能解析

工藤 周平[1,a)]　山本 有作[1,2,b)]

概要：BLAS は行列積などの基本的な行列計算を行う計算ライブラリである．本稿では，BLAS の中でも DSYRK の並列化手法について議論する．DSYRK は結果が対称行列となるような行列積であり，上（下）三角部分のみを計算する．そのため並列化をした場合，分割した計算領域が不均等な形になりやすく，単純な行列積と比べてワークインバランスを起こしやすい．とくに近年，高い並列度を持った高速な CPU であるメニーコアプロセッサが登場しており，そのような環境ではこの問題が顕著に表れると考えられる．そこで，メニーコアプロセッサの 1 つである Xeon Phi（Knights Corner）を対象にして，様々な並列化手法を用いた DSYRK を実装し，性能解析を行った．本稿では実装手法の詳細と性能解析結果を示す．

キーワード：DSYRK，Xeon Phi，Level-3 BLAS，スレッド並列化

Parallelization Methods of DSYRK on Xeon Phi and Their Performance Analysis

Shuhei Kudo[1,a)]　Yusaku Yamamoto[1,2,b)]

Abstract: BLAS is a basic linear algebra library which includes matrix-matrix multiplication subroutines. In this paper, we discuss parallelization methods for a BLAS routine DSYRK. DSYRK is a variant of matrix-matrix multiplication which results in a symmetric matrix, so it can skip the computation of the upper or lower triangular part of the resulting matrix. On the other hand, when parallelizing DSYRK, it is more difficult to achieve good load blance than in the case of simple (non-symmetric) matrix-matrix multiplication because divided computational regions have more irregular shapes. This becomes more problematic on recent high-performance many-core architecture CPUs which have much larger degree of parallelism. Therefore, to investigate efficient parallelization methods for DSYRK on such CPUs, we implemented DSYRK using a couple of parallelization methods for Xeon Phi (Knights Corner), and analyzed the performance results. In this paper, we will describe the implementation details and the results of the performance analysis.

Keywords: DSYRK, Xeon Phi, Level-3 BLAS, Multithreading

1. はじめに

Basic Linear Algebra Subprograms (BLAS) は行列積などの基本的な行列計算を行う関数 146 種を含む計算ライブラリであり，広く科学技術計算において利用されている．そのため，CPU アーキテクチャやハードウェアに合わせた高性能実装手法の研究開発が行われており，CPU メーカーが自社のハードウェアに向けてチューニングしたものとして，Intel MKL[1]，他にオープンソースの実装として，GotoBLAS[5] や BLIS[6] などの BLAS 実装が利用できるようになっている．しかし，これらの実装においても，多種多様な BLAS の関数すべてが十分に最適化されているわけではなく，実装が簡単なものや，特定のベンチマークなどに用いられるものなど重要なものが優先的に最適化されるため，中には期待したほどの性能がでない関数が存在する場合がある．

本研究では，BLAS の関数の中でも DSYRK を扱う．DSYRK は Cholesky 分解や Cholesky-QR 法など [8] で使

[1] 電気通信大学
1-5-1 Chofugaoka, Chofu, Tokyo 182-8585
[2] JST CREST
[a)] k1541013@edu.cc.uec.ac.jp
[b)] yusaku.yamamoto@uec.ac.jp

われる形の行列積であり，ある行列とそれ自身の転置との積を行う．類似した形の行列積として，DSYR2K，DSYMM，DTRMM などがあり，どれも行列の対称性や非零構造を利用することで，単純な行列積を行う DGEMM と比べて，演算量やデータ移動量を減らすことができる一方で，並列化やキャッシュ利用のためのブロック化などの処理がより複雑となる．そのため，DGEMM と比べてチューニングが行き届いていないことがあり，各ハードウェアに合わせた実装手法の研究が必要とされている．

近年，計算効率を上げるために CPU のマルチコア化が進んでおり，それをさらに発展させたアーキテクチャとして Xeon Phi のようなメニーコアプロセッサが登場している．メニーコアプロセッサは，単体では遅い速度のコアを多数集積することで，全体としては高い性能を実現する．Xeon Phi においては，1 つのコア当たりの速度は約 17GFlops であるが，これを 60 個程度，集積することで，1CPU では約 1TFlops の高い理論ピーク性能を持つ．このように多数のコアを持つため，メニーコアプロセッサ上で高い性能を実現するためには，高い並列性を持つプログラムが必要であり，BLAS の実装においてもこのことを考慮しなければならない．

そこで本稿では，Xeon Phi に向けて高性能な DSYRK の開発を行うために，DSYRK の並列化手法について検討する．Xeon Phi に対する行列積 (DGEMM) の実装手法については Smith ら [6][7] の研究があり，高性能な実装も公開されているが，DSYRK については十分な研究がされていない．そこで，Smith らの研究や，他にマルチコアプロセッサ向けの Level-3 BLAS 実装についての後藤ら [5] の研究をベースに，Xeon Phi への DSYRK の並列化手法の実装と，性能解析を行う．

以下の議論で対象としているアーキテクチャは，Xeon Phi の中でも，Knights Corner（KNC）のコードネームで呼ばれるものである．また，KNC の中でもコア数やメモリチャネルの数など，性能の異なるものが数種類あるが，そのうち型番が 3120A のものを用いている．

2. 行列積の構造

DSYRK は Fortran 形式で格納された倍精度実数の $m \times n$ 行列 A について

$$C = AA^\top \tag{1}$$

を計算するものである．DSYRK は別の計算パターンとして $C = A^\top A$ を計算することもできるが，本稿ではこの形のものについて考える．また，C の上三角・下三角部分のどちらを計算するか，を選択することもできるが，本稿では上三角部分のみを計算するものを考える．この式からすぐに C は $m \times m$ 行列であって，$C = AA^\top = (AA^\top)^\top = C^\top$ が成り立つことがわかる．しかし，DSYRK は結果が対称となることを除けば単なる行列積であり，DGEMM と多くの点で同じ議論ができる．

2.1 ブロック行列積

高性能な行列積の実装には並列化だけでなく，キャッシュブロック化や，データアクセス性能向上のためのデータパッキング，SIMD 命令などの CPU 機能の活用が不可欠である．これらを意識した実装をするうえでベースとなる考え方は，行列積をブロック行列積として書き直すことである．

いま $C = \{Z_{i,j}\}$，$A = \{X_{i,k}\}$，$A = \{Y_{j,k}\}$ とブロック分割する．簡単のためそれぞれのブロックの大きさは，$Z_{i,j}$ が $b_i \times b_j$，$X_{i,k}$ が $b_i \times b_k$，$Y_{j,k}$ が $b_j \times b_k$ のように固定とする．このとき，$C = AA^\top$ は

$$Z_{i,j} = \sum_k X_{i,k} Y_{j,k}^\top \tag{2}$$

と書ける．つまり全体の行列積を部分行列積の繰り返しとして書ける．ブロック幅は任意に設定できるため，データ量をキャッシュサイズに合った大きさにしたり，並列化のときに演算量を均等化することができる．また部分行列積もまた再帰的にブロック行列積として書くことができるため，さらに小さなブロックに分割することで，SIMD レジスタ幅に合わせた大きさにすることもできる．

ここで，部分行列積における，メモリからキャッシュへのデータ移動量と演算量を計算する．いま，部分行列積における部分和の計算の 1 つ

$$Z_{i,j} \leftarrow Z_{i,j} + X_{i,k} Y_{j,k}^\top \tag{3}$$

における，メモリからキャッシュへのデータ移動量は $8(b_i b_j + b_i b_k + b_j b_k)$ Byte であり，キャッシュからメモリへのデータ移動量は $8 b_i b_j$ Byte である．また，演算量は $2 b_i b_j b_k$ Flop である．よって，部分行列積におけるメモリ・キャッシュ間データ移動量と演算量の比（B/F 値）を B とおくと，

$$B = \frac{4}{b_i} + \frac{4}{b_j} + \frac{8}{b_k} \tag{4}$$

となり，キャッシュのサイズが許す限りブロック幅を大きくすることで，B/F 値を小さくすることができる．実際のブロック幅はキャッシュサイズからおおむね決定される上限の大きさを決めておいて，端数処理や並列化のために，ブロックごとに異なるサイズをとれるようにする．

2.2 ブロック行列積の手順

いま，C が $n_i \times n_j$ ブロックであり，また，A の列ブロック数が n_k であるとする．これはブロック幅が固定の場合，それぞれ $n_i = \lceil m/b_i \rceil$，$n_j = \lceil m/b_j \rceil$，$n_k = \lceil m/b_k \rceil$ である．このとき，ブロック行列積全体は次のようなプログラ

ムとして書くことができる．

```
subroutine blocked-matmul({Z_{i,j}}, {X_{i,k}}, {Y_{j,k}})
  Z_{:,:} ← O
  for i = 1:n_i
    for k=1:n_k
      for j = 1:n_j  if Z_{i,j} overlaps with triu(C)
        Z_{i,j} ← Z_{i,j} + X_{i,k} × Y^⊤_{j,k}
```

blocked-matmul の 3 つ目の j に関するループは，C の上三角部分 (tri(C)) に重なるもののみを処理するために，開始位置が変化する．そのため，j ループの開始位置を常に 1 とすれば，このプログラムを DGEMM の計算に用いることができる．また補足として，本稿での実験では $m \leq b_i$ の範囲でしか性能測定していないため，このプログラムの最外側ループは 1 度のみしか実行しない構造となっている．そこで本稿では最外側ループを無視した形での議論をしているが，$m > b_i$ の領域を考える場合には，以下の議論を多少拡張する必要がある．

blocked-matmul はメモリ上の Fortran 配列として格納された行列データに直接アクセスするが，CPU によっては CPU に合わせたデータ順序でなければ高い性能を引き出せないことがある．そこで，最内側処理のデータアクセス順序に合わせてデータの順序を並び替える処理（データパッキング）を追加したものが次のプログラムである．

```
subroutine blocked-matmul2({Z_{i,j}}, {X_{i,k}}, {Y_{j,k}})
  Z_{:,:} ← O
  for i = 1:n_i
    for k=1:n_k
      X̄ ← X_{i,k}
      for j = 1:n_j  if Z_{i,j} overlaps with triu(C)
        Ȳ ← Y_{j,k}
        Z_{i,j} ← Z_{i,j} + X̄ × Ȳ
```

このプログラムでは $X_{i,k}$ と $Y_{j,k}$ のパッキングを実装している．

2.3 データパッキングの効果

データパッキングを行うと，データアクセスが増える分のコストが発生する．そのため，データ順序が適切になったことによる内部処理の性能向上がデータパッキングのための性能劣化を上回らなければ，実装する価値がない．一方，パッキングを行ったデータはその直後に使用されるため，$\bar{X}$ や $\bar{Y}$ をキャッシュにのる大きさにすれば，性能劣化を小さく抑えることができる．そこで，データパッキングにおいては，パッキングのコストと，パッキングのためのデータ量が重要である．

まず全体の計算コストに対するパッキングの相対的なコストを考えるため，パッキングしたデータの再利用回数を計算する．blocked-matmul2 でのパッキングでは，$\bar{X}$ は 1 要素当たり最大 m 回，再利用されるが，$\bar{Y}$ は 1 要素当たり $\min(b_i, m)$ 回，再利用される．またパッキングのためのデータ量を計算すると，$\bar{X}$ を保存するために $8b_ib_k$ Byte，$\bar{Y}$ を保存するために $8b_jb_k$ Byte の領域が必要である．よって，b_j は $\bar{Y}$ が L2 キャッシュのような高い階層のキャッシュにのるように設定し，一方で b_i は $\bar{Y}$ の再利用回数を増やすため，大きな値とすることになり，$\bar{X}$ はラストレベルキャッシュ，もしくはメモリ上に置くことになる．

実際に KNC に対する実装で用いた値は $b_j = 144$，$b_i = 40,000$ である．これらの値は Smith ら [7] の実装における $b_j = 120$，$b_i = 14,400$ と異なるが，b_j は後述の，レジスタブロック幅の違いと手動チューニングの結果によるものである．b_i は，後に示す内積方向分割のときに，$\bar{Y}$ を分割して利用するため，分割したときでも十分な大きさとなる値としている．b_i の値を $14,400$ と $40,000$ とのどちらの値に設定したとしても，$\bar{X}$ のデータ量は KNC のキャッシュに収まらない大きさとなり，$\bar{X}$ はメモリ上に置くことになる．このため，$\bar{X}$ をパッキングするコストは $\bar{Y}$ のものと比べて大きくなる．そこで，$\bar{X}$ については，パッキングをする場合とそうでない場合の性能差が興味深い．

2.4 ブロック行列積の並列化

blocked-matmul2 に対する並列化の議論は Smith ら [7] によって詳しく解説されている．まず，最内側処理である部分行列積（$Z_{i,j} \leftarrow Z_{i,j} + \bar{X} \times \bar{Y}$）は，$b_i$ が十分に大きいため十分な演算量があり，並列化に適している．また，j に関するループは，b_j が小さな値であるから，ループ回数が大きいため，並列化に適している．一方，k に関する 2 番目のループは，b_k が小さな値であるためループ回数は大きいが，総和のための同期の問題や，スレッドごとに $\bar{X}$ の領域を確保する必要があるなどの問題があるため，あまり並列化に適していない．i に関する最外側ループは，$\bar{X}$ の領域に関する問題があり，また，最内側処理の並列化と並列化軸が同じであるため，積極的に採用する理由がない．よって Smith らは Xeon Phi に対しては，再内側処理と j に関するループの並列化を行っている．本研究ではこれを DSYRK に拡張すること，さらに，k に関するループの並列化との組み合わせを考える．

3. Knights Corner に対する部分行列積の実装

3.1 Knights Corner の詳細

KNC については Jeffers[4] に概要があり，CPU アーキテクチャに関しては Rahman[3] や Intel の命令リファレンス [2] に詳細がある．ここでは行列積の実装のために必要なことに焦点をあて，簡単に説明する．

KNC は，57 以上のコアを持つマルチコア CPU であり，512bit 幅の SIMD 演算（FMA を含む）を毎サイクル行う

表 1　Xeon Phi 51xx のメモリ性能
Table 1　The memory performance of Xeon Phi 51xx.

	Latency	Bandwidth	Capacity
L1D cache	3 cycle	64 Bytes/cycle	32KB
L2 cache	24 cycle	12 GB/s	512KB
Remote L2	250 cycle		
Memory	302 cycle	164 GB/s	

演算器を持ち，そして，1 コアあたり 4 つのスレッドを 1 サイクルごとに切り替えて実行する機構を持つ．

KNC は多数のコアが 1 つの Bus につながれており，キャッシュコヒーレンシが自動的にとられる．そのため，通常のマルチコア CPU のように使用できる．それぞれのコアが L1I/L1D キャッシュと L2 キャッシュをもち，Bus につながれる Memory Controller には GDDR5 メモリが接続されている．参考のため，Fang ら [10] による，Xeon Phi 5100 シリーズにおける，各種メモリ性能を測定した結果を表 **1** に示す．今回用いた Xeon Phi 3120A は表の Xeon Phi 5100 シリーズと比べて，メモリーチャネルの幅が 3/4 倍であり，理論ピークメモリバンド幅も約 3/4 倍となっているため，メモリバンド幅は表で示した値よりも小さな値となるが，それ以外の値は参考になる．

要点を書くと，コア内部にあるキャッシュメモリへのアクセスは高速であるが，コアの外部にあるデータへのアクセスは一様に遅い．そのため，コア間で共有するデータが少ないようにプログラムすることが重要である．

SIMD 演算は，8 要素の SIMD レジスタを 32 本持つ構成となっており，1 サイクルに最大 1 つの SIMD 演算命令を実行できる．そのため積和命令を使えば，1 サイクルで 16 演算を行うことができる．KNC のパイプラインは 2 本（uv-pipe）あるが，SIMD 演算は基本的に片方のパイプライン（u-pipe）にしか流せないため，SIMD 積和命令の機能をうまく活用して，少ない命令数で行列積を作らなければならない．

KNC の命令デコーダーは 1 コアにつき，2 つ以上 4 つまでのスレッドが実行されることを前提としており，1 スレッドのみでは最大の命令供給性能（2 命令/cycle）の半分の性能しか出せない．そこで，1 コアに 2 つ以上のスレッドを走らせなければならないが，これらのスレッドでコア内のキャッシュを共有することになる．そのため，コア内のキャッシュにはこれらのスレッド間で共有するデータを入れるようにすると，キャッシュをスレッドで分断することがなくなり，都合がよい．一方，コア外部のキャッシュについては，複数のコアでデータを共有すると，同じデータがそれらのコアのキャッシュに配置されるため，コア間でデータを共有することに利点がない．よってコア内部での並列化とコア間での並列化とでは共有データの違いを意識する必要がある．

3.2　計算カーネルの設計

$Z_{\cdot,\cdot}$，$X_{\cdot,\cdot}$，$Y_{\cdot,\cdot}$ についての部分行列積をさらに分割して，SIMD 演算に適した大きさにする．SIMD レジスタは 8 要素，32 本あるから，ワークレジスタの本数を最小限にし，部分行列を SIMD レジスタ上に乗せることで，メモリ・キャッシュとレジスタ間のデータ移動命令を最小限にする．ここでは，

$$Z_{\cdot,\cdot} = \left[\begin{array}{c|c|c|c} w_{1,1} & w_{1,2} & \cdots & w_{1,s_j} \\ \hline w_{2,1} & w_{2,1} & \cdots & w_{2,s_j} \\ \hline \vdots & & \ddots & \vdots \\ \hline w_{s_i,1} & \cdots & & w_{s_i,s_j} \end{array}\right] \tag{5}$$

$$X_{\cdot,\cdot} = \left[\begin{array}{c|c|c|c} u_1^\top & u_2^\top & \cdots & u_{s_i}^\top \end{array}\right]^\top \tag{6}$$

$$Y_{\cdot,\cdot} = \left[\begin{array}{c|c|c|c} v_1 & v_2 & \cdots & v_{s_j} \end{array}\right] \tag{7}$$

のように分割し，$w_{i,j}$ を SIMD レジスタ上に置く方法を採用する．$w_{i,j}$ を 8×24 行列とすると，$w_{i,j}$ の 1 列に 1 つのレジスタを用いることで，24 本の SIMD レジスタで表現できる．このとき計算の主要部分は $8 \times b_k$ 行列 u_i と $b_k \times 24$ 行列の v_j との積 $w_{i,j} \leftarrow u_i \times v_j$ であり，これを KNC に最適化する．

次のプログラムは KNC 上で高速な $w_{i,j} \leftarrow u_i \times v_j$ の計算手順である．

```
subroutine kernel8x24(w, u, v)
  r(1:8, 1:24) ← O
  for k = 1:b_k
    s ← u(1:8, k)
    for j = 1:24
      r(1:8, j) ← r(1:8, j) + v(k, j) × s
  w ← w + r
```

このプログラムは，外側ループが 1 つ進むたびに u の 1 列と v の 1 行を読み込んで，r のすべての要素を更新する形となっている．r をレジスタ上に格納するため，内側ループは固定長になっており，完全なループアンロールを行う．最内側処理は，r の 1 列（1 つのレジスタ）に，s（1 つのレジスタ）を $v(k,j)$（メモリ・キャッシュ上の 1 要素）でスカラー倍したものを足しこむ操作となっている．これは KNC の命令 `vfmadd231pd` のアドレッシングのオプションを使えば，1 命令で実行できる．よって，内側ループは 24 命令で 24 回の積和を行うことになる．そして，外側ループのループ制御命令などをうまく配すれば，外側ループの 1 ステップあたり 26 サイクルで実行できる．よって外側ループの実行効率は，すべての命令が 1cycle ごとに投入される最も理想的な状態においても最大で $24/26 \approx 0.923$ となる．実際には，w への結果の足しこみ処理や，データパッキングのような計算カーネル以外の処理も行わなければならないため，DSYRK 全体の実行性能は 0.923 を上限

としてこれよりも小さな値となる．

レジスタのうち w の格納以外に 2 本のレジスタを使うため，内側ループの長さは最大で 30 まで伸ばすことができる．逆に内側ループ長の短いものも作ることで端数処理に対応できる．BLIS では幅 30 のもののみを実装しており，インラインアセンブラのコードが公開されている [9]．本研究では，縦と横の大きさの比が単純となり実装が簡単となるため，8 の倍数の幅のもの（8, 16, 24）を作成した．

また，ここでは w をレジスタ上においたが，分割の仕方を変えて，u や v をレジスタ上においたときの計算カーネルを考えることもできる．とくに u を 8×24 の大きさにした計算カーネルも，同じ積和命令比率を達成できる．

3.3 キャッシュブロック化

前節の計算カーネルを用いると，部分行列積は次のようなプログラムとなる．

```
subroutine subblock-matmul({w_{i,j}}, {u_i}, {v_j})
  for i = 1:b_i/8
    for j = 1:b_j/24
      kernel8x24(w_{i,j}, u_i, v_j)
```

実際にはこの 2 重ループのうち内側ループは上三角の構造に合わせて開始位置を調整し，計算量を抑える．このプログラムにおいて u_i は内側ループで繰り返し使われる．$\bar{X}$ のパッキングを行わない場合は，この u_i へのアクセスは行列 A へ直接アクセスすることになる．v_j は外側ループで繰り返し使われるから，キャッシュに乗せる．前者の格納に必要なデータ量は $64b_k$ Byte であり，後者は $8b_jb_k$ Byte である．よってデータ量だけを考えて後者を L2 キャッシュに収めるようなブロック幅を計算すると，b_j, b_k を約 181 にすればよいが，計算カーネルのブロック幅や，b_j と b_k の性能への影響の違い，キャッシュ制御機構の影響を考慮したうえで最適な b_j, b_k を理論的に求めることは困難である．そこで，手動でチューニングした結果，性能の良かった $b_j = 144, b_k = 200$ を用いる．この場合，v_j のデータ量は 225KB，u_i は 12.5KB となり，u_i は十分 L1D キャッシュに収まる大きさとなるが，次に示すコア内での並列化のため，u_i も L2 キャッシュに収めるようにする．

このように部分行列積ではキャッシュの使用量を考えるため，キャッシュを共有するコア内部での並列化を同時に取り扱うと都合がよい．`subblock-matmul` において，外側ループを並列化すると，u_i は共有できないが，v_j は共有でき，L2 キャッシュのデータの共有ができる．この並列化においては，内側ループの長さの違いや，スレッド間の多少の実行時間の違いを吸収するため，外側ループ番号の i をアトミック変数として動的な並列化を行う．

4. DSYRK の並列化

以上の通りレジスタ・キャッシュブロック化やデータパッキングを行うので，DSYRK の並列化では，ブロックサイズやコア間での共有データを考慮する．

行列積の並列性には 2 種類あり，1 つは結果の行列 C の要素ごとの並列性であり，もう 1 つは内積計算の並列性である．前者は処理間に依存性がない並列性であるが，データパッキングをする場合には結局同期が必要となる．また，C の上三角部分のみ計算するため，分割したときに処理のバランスをとることが難しい．後者を並列化した場合には，どのタイミングで同期をとり総和を計算するかが問題となる．また，部分和を保存しておくためのメモリ領域が必要となる．

そこで，まず内積計算の並列性を用いた分割を考える．これは，単独で用いた場合，必要なメモリ量が非現実的な値になりやすいという問題があるため，他の並列化手法と組み合わせて利用する．次に，結果の行列の要素ごとの並列性を用いた手法 2 つを示す．これらの 2 手法は計算時間を均等化することが難しいが，内積方向の分割と組み合わせると，この問題を緩和できる．

4.1 内積方向分割

内積方向分割は，$A = \begin{bmatrix} A_1 & A_2 & \cdots & A_{N_\mathrm{I}} \end{bmatrix}$ と N_I 個に分割しておき，より小さな DSYRK である $C_l = A_l A_l^\top$ をすべて並列に行った後，最後に総和 $C = \sum_l C_l$ を計算するものである．これはプログラム `blocked-matmul2` において，中央の k に関するループを並列化したことに相当する．

この並列化手法の利点は，演算量や計算時間を均等化しやすい点である．A の分割を 1 列単位で行えば，各 C_l 間の演算量の違いは 1 列分以下となるため，n が N_I と比べて十分大きければ，演算量の違いは小さくなる．また，各 C_l の計算領域は A_l の列数の違いを除けば同じとなるため，計算時間の違いも小さくなると予想される．

一方，問題点は，分割によって内積方向のデータ再利用性が低下することと，総和の計算のためのコストが増えること，そして C_l のためのメモリ領域が必要になることである．A_l の列数がキャッシュブロックの大きさ b_k よりも小さくなると，その分だけデータ再利用性が低下するため，性能が劣化する．また，A_l の列数が b_k よりも多少大きい場合でも，端数処理のために b_k よりも小さい列数のブロックを計算するため，同じ問題が起きる．総和の計算は，行列積の内積のうちいくつかの足し算を最後に処理することであるから，行列積の演算量を変化させない．しかし積和命令を持つような CPU では，積と和を同時に行えないため演算量が増えることと同然となる．また，総和の

計算は演算量に対してデータ移動量が大きいため，データ移動時間を演算時間の中に隠蔽することができない．このため総和の計算の実行効率は行列積自体と比べて小さくなり，相対的に総和のコストが大きく見えやすい．以上 2 つの問題点は n が N_{I} と比べて十分大きければ無視できるようになる．

C_l を格納するために必要なメモリ量は，ユーザーから与えられた C を活用すると 1 つ分だけ省略できるので，$8m^2(N_{\mathrm{I}}-1)$ Byte となる．ただし，今回は実装していないが，C_l が対称性を利用すればメモリ量を約半分にすることも可能である．この式を用いて実際に必要なメモリ量を計算すると，例えば $m = 5,000$，$N_{\mathrm{I}} = 56$ の場合，約 10GB のメモリ領域を必要とすることになり，現実的でない．またデータパッキングを行う場合，$\bar{X}$ や $\bar{Y}$ もコアごとに用意しなければならない．とくに $\bar{X}$ はもとから大きな領域となっているため，問題である．

そこで，内積方向の分割は単独では利用せず，別の並列化手法（以下で説明する 1 次元，2 次元分割）と組み合わせて利用する．例えば $N_{\mathrm{I}} = 10$ としておいて，各 C_l を別の並列化手法を用いて，5 または 6 並列で実行する．こうすれば，必要なメモリ量を大幅に小さくでき，なおかつ，以下の並列化手法においても現れる，並列度が大きい場合のデメリットを緩和できる．

ここで，N_{I} の決定基準が問題となる．N_{I} の値は，メモリ量や計算時間にも影響するが，ここでは，使用するメモリ量の最大値を決めて，その範囲で最も大きな N_{I} を選択する，固定量メモリ基準を使う．いま全体のコア数を N_{C}，C_l のために用意するメモリ量の最大値を要素数として $M = 25 \times 10^6$ のように決めたとき，

$$\bar{m} = 8\lceil m/8 \rceil \tag{8}$$

とおき，

$$N_{\mathrm{I}} = \min\left(\left\lfloor \frac{M}{\bar{m}^2} \right\rfloor + 1, N_{\mathrm{C}}\right) \tag{9}$$

と決める．$\bar{m}$ はアライメントを調整した m であり，$\bar{m} \times \bar{m}$ の大きさの行列が M の中に何個入るか計算して，それに 1 を足した値を N_{I} としている．ただし N_{C} より大きくならないように調整する．これは $N_{\mathrm{C}} = 56$ だとすると，$m = 672$ のときの最大値 56 をとり，m が大きくなるにつれ段階的に減少し，$m \geq 5,000$ のとき最小値 1 をとる．このとき $\bar{X}$ を格納するために必要な領域は，$8\bar{m}N_{\mathrm{I}}b_k$Byte である．$\bar{m}N_{\mathrm{I}}$ の最大値は M によって決まるため，$b_i \geq \max(\bar{m}N_{\mathrm{I}})$ として $\bar{X}$ を 1 つ用意しておき，$N_{\mathrm{I}} > 1$ の場合は $\bar{X}$ のメモリ領域を N_{I} 個に分割して用いるようにすれば，$\bar{X}$ のためのメモリ領域も固定サイズになる．

固定量メモリ基準によって N_{I} を m^2 に反比例するように決めることは，使用するメモリ量を一定にできるだけでなく，次の観点で合理性がある．まず，m が十分大きい場合は，以下の並列化手法を用いても十分高い性能が実現できるため，小さな N_{I} でよいが，逆に m が小さい場合は，以下の並列化手法の性能が落ちるため，N_{I} を大きくすべきである．ただし，この基準は n を計算に使用していないため，「n が小さく，総和のコストが相対的に大きくなる状態では，m が小さくても N_{I} を小さくする」などの調整は行えない．このような計算時間の最小化を行うためには，計算時間のモデル化が必要であるから，今後の課題とする．

4.2 1 次元分割

以上のように内積方向分割を行った場合，それぞれの C_l を $N_{\mathrm{O}} = \lfloor N_{\mathrm{C}}/N_{\mathrm{I}} \rfloor$ または $N_{\mathrm{O}} = \lceil N_{\mathrm{C}}/N_{\mathrm{I}} \rceil$ 並列で計算する．（内積分割を行わない場合は $C = C_1$ を N_{C} 並列で計算する．）1 次元静的分割は，C_l を N_{O} 個の列ブロックに分割し：

$$C_l = \left[\begin{array}{cccc} D_1 & D_2 & \cdots & D_{N_{\mathrm{O}}} \end{array} \right], \tag{10}$$

それぞれの列ブロックを 1 コアに割り当て，並列に処理するものである．これはプログラム `blocked-matmul2` において最内側ループの j に関するループを並列化したものである．ここでの目標は，各列ブロックの列幅を調整して，コアごとの実行時間を均等にすることである．そこで 1 次元静的分割では，演算量がなるべく均等になるよう，演算量を最小最大化する手法を用いる．

いまそれぞれの列ブロックの列幅を $c_1, c_2, \ldots, c_{N_{\mathrm{O}}}$ とおき，列ブロックの開始位置 $j_s = \sum_{t=1}^{s} c_t$（ただし $j_0 = 0$）とおく．j_s は本来，整数だが，実数の範囲で考えると最適値は簡単に計算でき，

$$\frac{j_{s+1}^2 - j_s^2}{2} = D \tag{11}$$

と漸化式で求められる．ただし，$D = \frac{m^2}{2N_{\mathrm{O}}}$ であり，上三角部分の面積をコア数で割った値である．GotoBLAS や BLIS で実装されている手法は，単純にこれを整数で丸めたものであり，次の漸化式を使っている．

$$j_{s+1} = \mathrm{round}(\sqrt{j_s^2 + 2D}). \tag{12}$$

ただし，終端は行列サイズと一致するよう $j_{N_{\mathrm{O}}} = m$ とする．round は丸め関数で，計算カーネルのブロック幅 8 へ丸める．このように求めた c_s は b_j より大きくなりうるが，その場合は b_j ごとに分割して，同じコアで計算させればよい．

このように分割を行うと丸めがどのように起こるかわからないため，演算量の最小最大化を行える保証はない．本来求めたい値は，

$$\begin{array}{ll} \min. & \max\{\sum_{i=i_{s-1}}^{i_s-1} i | s = 1, 2, \ldots, N_{\mathrm{O}}\}, \\ \text{sub. to.} & 0 = i_0 \leq i_1 \leq i_2 \leq \cdots \leq i_{N_{\mathrm{O}}} = m, \end{array} \tag{13}$$

の最適解である．これは整数最適化問題であるが，実は簡

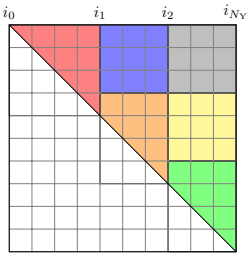

図 1　2 次元静的分割の例
Fig. 1　An illustration of the 2D static blocking.

単に解けるクラスの問題であり，具体的には，式（12）の D を二分探索することで，最適解が得られる．アルゴリズムとしては word wrapping の逆問題としても見ることができる [11].

1 次元分割の問題点は，列ブロックの幅が小さくなり，データの再利用性が低くなりやすいことである．式（11）で概算すると，$N_O = N_C = 56$ の場合 $m = 5000$ のとき c_{N_O} は約 45 となる．これはキャッシュブロック幅 b_j の約 3 分の 1 である．この問題は内積方向分割との組み合わせで緩和できる．

4.3　2 次元分割

1 次元分割では，1 列ブロック当たり 1 コアを割り当てるので，右端の列ブロックの幅が小さくなった．2 次元分割では，右の列ブロックに行くほど演算量を増やす代わりに多くのコアを割り当てることで，列ブロックの幅を均等にする．1 つの列ブロックをさらに行ブロックに分割することで，列ブロック内での並列化を行う．これはプログラム `blocked-matmul2` において，最内側処理を並列化したことに相当する．例として，図 **1** に分割の様子を示す．

まず，列ブロックの個数を N_X，s 番目の列ブロックに割り当てるコアの数を m_s とおく．このとき，1 コアあたりの演算量がなるべく等しくなるように分割したいので，1 次元静的分割の問題を拡張した次の問題の最適解を，列ブロックの分割位置とする．

$$\begin{array}{ll} \text{min.} & \max\{\sum_{j=j_{s-1}}^{j_s-1} \frac{j}{m_s} | s = 1, 2, \ldots, N_X\}, \\ \text{sub. to.} & 0 = j_0 \le j_1 \le j_2 \le \cdots \le j_{N_X} = m, \end{array} \quad (14)$$

この問題に対しても 1 次元分割の時と同じアルゴリズムが使えるので効率的に計算できる．N_X や m_s の設定の仕方が問題であるが，分割した領域が正方形となることを目指して，$N_X \approx 2\sqrt{N_O}$，$m_s \approx s$ を整数に丸めた値とする．

5.　性能測定の結果

本実験のプログラムは C++と SIMD 演算の intrinsic 命令を用いて作成した．コンパイラは icpc の 15.0.1 である．Xeon Phi はコプロセッサとしての実行形態として，ホストマシンと独立した計算機として動作する native モードや，ホストマシンから呼び出される形で動作する offload モードなどがあるが，このプログラムは native モードのみを対象としている．また，並列化のためのスレッド生成や affinity の設定などに OpenMP を用いている．また，同期機構については，OpenMP や Pthread のバリアではなく，独自に実装したバタフライバリアを用いている．これは，コア内のスレッド間での同期や内積方向分割における各 C_l ごとの同期など，複雑な制御が必要であるから，OpenMP のバリアでは実装が難しく，また，Pthread のバリアは性能上の問題があったためである．

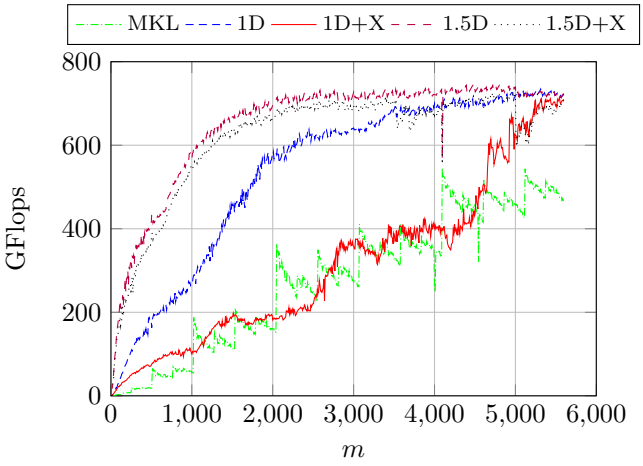

図 2　n を固定したときの実行性能 (1D)
Fig. 2　The performance results of DSYRK with fixed n using 1D parallelization.

性能測定は Xeon Phi 3120A を用いた．これは 57 個のコアを持つが，Jeffer[4] などで解説されているとおり，1 つのコア OS やホストマシンとの通信で使われる分としてを残し，56 コアのみを計算に用いた．このときの理論ピーク性能は 985.6GFlops である．スレッド数は，各コア 4 スレッドずつの 224 とし，環境変数を `KMP_AFFINITY=granularity=fine,compact` と設定し affinity を指定した．実行性能は演算量 $E_{\text{all}} = m(m+1)(2n-1)/2$ を実行時間で割ったものとして計算しており，実際に CPU で行った浮動小数点演算の量とは異なる．

測定の対象は，まず比較対象として MKL の version 11.2.1 に含まれる DSYRK を用いた．また，今回実装したものとして，1 次元分割のみを使うもの（1D），1 次元分割で $\bar{X}$ のパッキングを行うもの (1D+X)，それぞれについて，内積方向分割を組み合わせたもの（別の次元を補助的に使うという意味でそれぞれ 1.5D, 1.5D+X），それらの 2 次元分割に置き換えたもの（2D, 2D+X, 2.5D, 2.5D+X）を対象とした．行列 A と C はどちらもそれぞれの列のアライメントがそろっているものを用いている．

5.1　実行性能

計算する行列の大きさに関して 2 つのパラメータ m，n があるから，それぞれのパラメータの実行性能への影響を

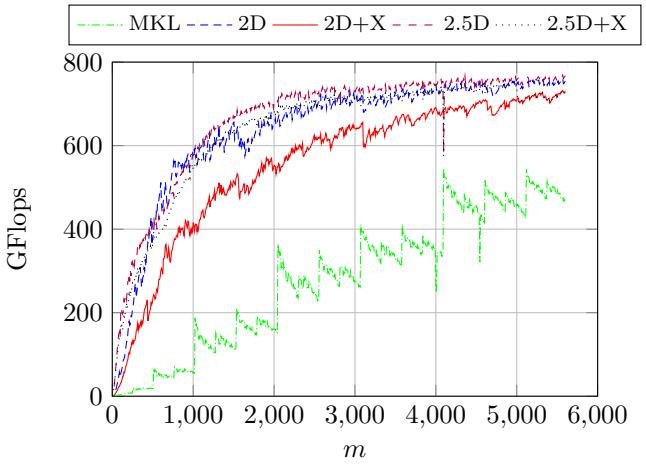

図 3　n を固定したときの実行性能 (2D)

Fig. 3　The performance results of DSYRK with fixed n using 2D parallelization.

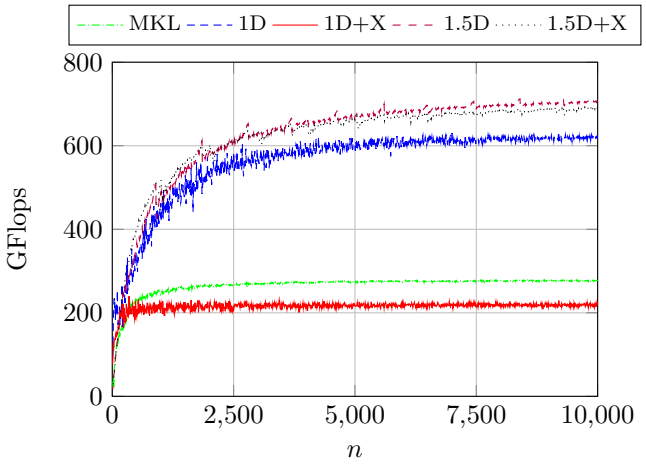

図 4　m を固定したときの実行性能 (1D)

Fig. 4　The performance results of DSYRK with fixed m using 1D parallelization.

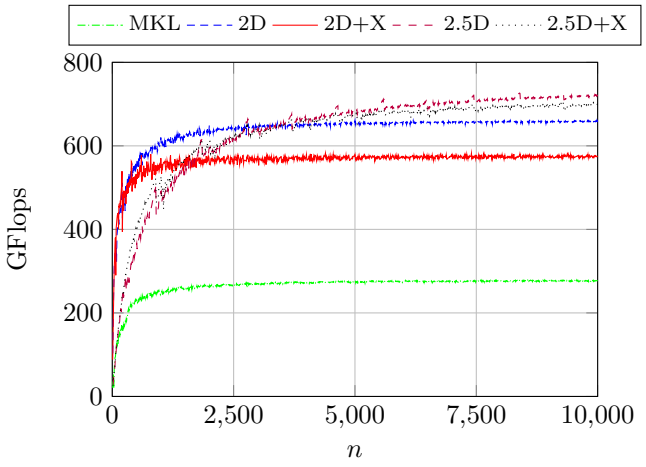

図 5　m を固定したときの実行性能 (2D)

Fig. 5　The performance results of DSYRK with fixed m using 2D parallelization.

見るために，$n = 10,000$ に固定して m を 8 から 5,600 まで変化させたときの性能と，$m = 2,400$ に固定して n を 8 から 10,000 まで変化させたときの性能をそれぞれ調べた．

まず，$n = 10,000$ と固定して m を変化させたときの性能を図 **2** と図 **3** に示す．図 2 は 1 次元分割と，それに内積方向分割を組み合わせたもの，図 3 は 2 次元分割と内積方向分割を組み合わせたものである．1 次元分割で最も速いものは，1.5D であり，小さな m においても高い性能となり，最大 746GFlops となっている．内積方向分割を組み合わせないものは m が小さい時の性能が低くなっている．2 次元分割では，内積方向分割を組み合わせなくても，小さな m で高い性能となるが，全体として最も高速であるのは 2.5D であり，最大 769GFlops に到達している．

$\bar{X}$ のデータパッキングを行ったものは，むしろ性能が減少している．これはデータパッキングのコストがデータアクセス速度向上の効果より大きくなっているためだと思われる．しかし，この実験では A のアライメントがそろっているものを用いているが，そうでない場合，A へのアクセスコストが上昇するため，この差が逆転する可能性がある．

次に，$m = 2,400$ と固定して n を変化させたときの性能を図 **4** と図 **5** に示す．図 4 は 1 次元分割と，それに内積方向分割を組み合わせたもの，図 5 は 2 次元分割と内積方向分割を組み合わせたものである．この図から，内積方向分割を行わないものは，$n > 1,000$ 程度の領域でほぼ性能が一定となっていることがわかる．一方で，内積方向分割を組み合わせたものは，n によって性能が大きく変動しており，総和などのオーバーヘッドの影響が見て取れる．ただし例外として 1D については，内積方向分割を行うものと同じように n によっての性能変化が大きい．

5.2　性能解析

次にプログラムを各ステップに分けて時間測定を行うことで実行時間の内訳を示したものが図 **6** である．ここでは行列サイズとして，$m = 1,200$ または $2,500$，$n = 2,500$，または $5,000$ の組み合わせとして 4 種類用いた．ステップは，並列化手法によって多少異なるが，計算カーネル部分（`kernel8x24` に相当）と，$\bar{X}$ のパッキング（行うもののみ），$\bar{Y}$ のパッキング，C の総和（内積方向分割のみ），その他（同期や初期化処理）に分けている．

図 2 などの結果と対応して，図 6 においても 1D+X がとくに遅い．実行時間の内訳をみると，1D と比べて，単に同期や $\bar{X}$ のパッキングの時間が増えているだけではなく，計算カーネルの実行時間自体が増えていることがわかる．1D と 1D+X とでは計算領域の形状が一致するが，$\bar{X}$ をコア間で共有する 1D+X は 1D より頻繁に同期を行う．この結果，複数のコアで共通のデータへのアクセスが頻繁に発生し，単純なメモリアクセスより低速となるデータブロードキャストが頻繁に発生したことが原因ではないかと

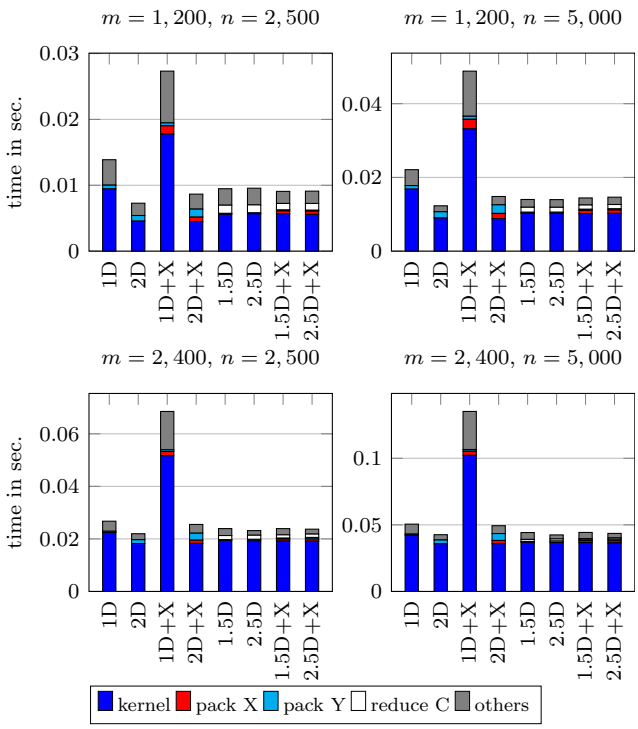

図 6　実行時間の内訳
Fig. 6　The break down of the time of DSYRK.

考えられる．

1D+X 以外についてみると，まず 1D は 1D+X ほどではないが計算カーネルの実行時間が大きい．とくに $m = 1,200$ のときは 2D のものと比べて約 2 倍となっているが，$m = 2,400$ のときは約 1.2 倍まで低下する．これは純粋に，1D の欠点である，列ブロックの幅が小さくなりやすく，データ再利用性が低下する問題が表れているものと考えられる．

2D や 2D+X は他と比べて顕著に $\bar{Y}$ のパッキングにかかる時間が大きい．これらは `blocked-matmul2` の再内側処理を複数のコアで並列化するが，このとき $\bar{Y}$ のパッキングについては並列化していないことが影響していると考えられる．

また，全体として，m が増加したとき $\bar{X}$ や $\bar{Y}$ のパッキングの割合が減少し，n が増加したとき，C の総和の割合が減少していることが確認できる．C の総和の割合は m が増加したときも減少している．これは，$m = 1,200$ のときは内積方向の分割数 $N_{\mathrm{I}} = 18$ であるのに対して，$m = 2,400$ のときは $N_{\mathrm{I}} = 5$ となり，総和の演算量の占める割合が減少することを反映していると考えられる．

6. まとめと今後の課題

6.1 まとめ

本稿では，Xeon Phi の KNC に対して，DSYRK の並列化手法を複数実装し，性能測定した．性能の詳細な解析のために，まずマルチコア CPU 向けの既存の行列積実装手法を解説し，実行時間に影響するプログラムの構造として，計算カーネルとデータパッキングを示した．そして，並列化手法として 1 次元分割と 2 次元分割，そしてそれに組み合わせるものとして，内積方向分割を示した．

最後に，それぞれの並列化手法を KNC 上で性能測定した．その結果，n が十分大きい場合は，2 次元分割手法と内積方向分割を組み合わせた手法が高速であった．ただし，内積方向分割を用いると n が小さい場合に性能劣化するので，2 次元分割が高速となることがあった．

6.2 今後の課題

本稿で実験の対象とした実装は，BLAS の DSYRK と完全互換なものではなく，特定の条件でしか動かない，不完全なものである．例えば，入力行列の転置操作であったり，入出力行列のアライメントといった部分を切り替えた場合，データアクセスの速度が変わってくるため，今回は見えなかった $\bar{X}$ のパッキングをする効果が現れる可能性がある．よって今後，実装を強化したうえで，調査する必要がある．

今回の並列化手法では演算量が均一となるように N_{I} や j_s を決定して計算領域を分割したが，本当にしたいことは計算時間の最小最大化である．計算時間は計算機アーキテクチャや計算領域の形状によって変化するため，何らか手法で性能モデルを構築する必要がある．そこで，このモデル化手法と，そのモデル上で最適な N_{I} や j_s を決定する手法といった，性能最適化手法が考えられる．

謝辞　本研究は, 科学技術振興機構 戦略的創造・研究推進事業 (CREST)「ポストペタスケールに対応した階層モデルによる超並列固有値解析エンジンの開発」および科学研究費補助金（No. 26286087, 15H02708, 15H02709）の支援を受けている．

参考文献

[1] Intel Corporation: *Developer Reference for Intel Math Kernel Library 11.3 - C*, available from ⟨https://software.intel.com/en-us/mkl-reference-manual-for-c-pdf⟩ (Sept., 2015).
[2] Intel Corporation: *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*, available from ⟨https://software.intel.com/sites/default/files /forum/278102/327364001en.pdf⟩ (Sept., 2012).
[3] Rahman, R.: *Intel Xeon Phi Coprocessor Vector Microarchitecture*, available from ⟨https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture⟩ (May 31, 2013).
[4] Jeffers, J. and Reinders, J.: *Intel Xeon Phi Coprocessor High Peformance Programming*, Morgan Kaufmann (2013). (すがわらきよふみ, エクセルソフト 訳: インテル Xeon Phi コプロセッサー ハイパフォーマンス・プログラミング, カットシステム (2014)).
[5] Goto, K and Geijin, R.A..: “High-Performance Implementation of the Level-3 BLAS,” *ACM TOMS*, Vol. 35, Issue 1, No. 4 (2008).
[6] Field, G., Van Zee and et. al,: “The BLIS Framework:

Experiments in Portability," *ACM TOMS* (2015).
[7] Smith, T.M., Geijin, R.A. Hammond, J.R. and Van Zee, F.G.: "Anatomy of High-Performance Many-Threaded Matrix Multiplication," *IPDPS 2014* (2014).
[8] Golub, G.H. and Van Loan, C.F.: *Matrix Computations 3rd Edition*, Johns Hopkins Univ. Press (1996).
[9] FLAME group: "BLAS-like Library Instantiation Software Framework," available from ⟨https://github.com/flame/blis⟩ (Feb. 4, 2016).
[10] Fang, J., Varbanescu, A.L., Sips, H., Zhang, L., Che, Y. and Xu, C.: "An Empirical Study of Intel Xeon Phi," available from ⟨http://arxiv.org/abs/1310.5842⟩ (2013).
[11] An answer at stackoverflow: available from ⟨http://stackoverflow.com/questions/6426017/word-wrap-to-x-lines-instead-of-maximum-width-least-raggedness⟩ (Nov. 12, 2015).