Vol.2015-HPC-152 No.17
2015/12/17

Xeon PhiにおけるDSYRKのスレッド並列化手法

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．近年，Xeon Phi などの多数のコアを持つ高速な CPU が現れており，このようなアーキテクチャに対する BLAS の実装手法が重要となっている．本発表では，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，計算時間が均等になるよう計算領域を分割することは容易ではない．そこで，並列化可能な 3 つの軸すべてを並列化に使う動的分割法と，1 つまたは 2 つのみを使う静的分割法の 2 種類の手法を示し，それぞれを用いたときの実行性能を示す．

キーワード：DSYRK，Xeon Phi，Level-3 BLAS，スレッド並列化，動的スケジューリング

Static and Dynamic Scheduling Methods for DSYRK on Xeon Phi Coprocessor

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

Abstract: BLAS is a basic linear algebra library which includes matrix multiplication subroutines. Recently, it becomes a problem how to optimize BLAS for manycore architecture like Xeon Phi which consists of large number of cores. In this talk, we introduce multithreading methods for DSYRK. DSYRK is a subroutine of BLAS peforms matrix multiplication results a self-adojoint matrix, thus it can reduce the half of computations by skipping the upper/lower trianguler part. On the other hands, it's not easy to get best work-balance by parallelizing this triangular area. We developed two types of methods, one is a dynamic scheduling method which parallelizes all the three outer loops of the blocked matrix multiplication, and another one is a static scheduling method which parallelizes one or two of theme. Then we show the performance results of those methods on Xeon Phi.

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

1. はじめに

BLAS は行列積などの基本的な行列計算を行うサブルーチンを集めたライブラリであり，広く科学技術計算において利用されている．そのため，各 CPU やハードウェアにチューニングされたものが多く存在し，MKL や BLIS, GotoBLAS などがある．しかし中には十分にチューニングされていないサブルーチンが存在する場合があり，期待されるほどの性能が出ない，または，非常に大きなサイズの行列でないと性能が出ないことがある．

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 CPU が登場している．メニーコア CPU 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 50 以上集積することで，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 CPU 上で高い性能を実現するために

[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

は，高度な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くてはならない．

これまで，多くのBLASのサブルーチンは自明的な並列性を持つため，BLASの並列化は容易なものと考えられてきた．現状のマルチコアCPUでは並列度が4から8程度であるから，複雑な並列化をせずとも，十分にワークバランスが取れ，高い性能を実現できた．しかし，並列数が大きくなると，単に数が増えたということで，問題の複雑性は増大し，困難なものとなる．

本研究では，BLASのサブルーチンの中でもLevel-3 BLASの1つであるDSYRKについて扱う．DSYRKは倍精度実数の行列Aが与えられたとき，$C = AA^\top$を計算するものである．よってDSYRKの計算結果は必ず対称行列となり，行列の上（下）三角部分のみを計算することで，全体を計算してしまうのに比べて計算量を約半分にすることができる．したがって，高速なDSYRKを実装するにはこの対称性を利用しなければならないが，これは対称性を利用しない場合と比べてやや複雑となる．とくに，コアごとの計算時間が均等になるよう三角行列を分割する手法は，自明的ではなく，考慮すべき多くの手法が存在し得る．そこで本研究ではDSYRKのスレッド並列化手法について検討する．行列積における並列性は，結果の行列の要素ごとの並列性（2つの軸）と，内積が持つ並列性の，合わせて3つの軸がある．ここではそれら3つの軸を使った，3種類のスレッド並列化手法について調査する．

既存研究としては，後藤のGotoBLASにおける実装手法を解説したものがあり[4]，現代的なマルチコアCPUに対するLevel-3 BLASの構成論を示している．他に，本研究と同じくXeon Phiに対する行列積の実装手法について解説したBLISの論文[5][6]があるが，本研究と異なり，DGEMMとSGEMMという，結果の対称性がない行列積を行う関数を対象としている．本研究ではこれらの論文をベースとしてDSYRKの実装に用いている．

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（KC）のコードネームで呼ばれるものである．また，KC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．本稿の構成は，行列積の実装手法における一般的な話題からはじめて，KCに最適化した実装手法を述べたのち，DSYRKの並列化手法について示す．そして，それぞれの並列化手法について性能測定した結果を比較する．

2. 行列積の構造

DSYRKはある実数の$m \times n$行列Aについて

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

を計算するものである．DSYRKにはオプションパラメータがあって，似たような形のいくつかのパターンがあるが，本稿ではこの形のものについて考える．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，多くの点で同じ議論ができる．

行列積を実装するための基本となる考え方は，ブロック行列積として書き直すことである．このときのブロック幅をうまく調整することで，キャッシュ再利用性が高く，SIMD化に適した計算順序にできる．

いま$C = \{C_{i,j}\}$，$A = \{A_{i,k}\} = \{A_{j,k}\}$とブロック分割する．それぞれのブロックの大きさは，$C_{i,j}$が$b_i \times b_j$，$A_{i,k}$が$b_i \times b_k$，$A_{j,k}$が$b_j \times b_k$となっているものとする．このとき，$C = AA^\top$は

$$C_{i,j} = \sum_k A_{i,k} A_{j,k}^\top \tag{2}$$

と書ける．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

$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} A_{j,k}^\top \tag{3}$$

における，メモリからキャッシュへのデータ移動量d_{L}(Byte)は

$$d_{\mathrm{L}} = 8(b_i b_j + b_i b_k + b_j b_k) \tag{4}$$

であり，キャッシュからメモリへのデータ移動量d_{S}(Byte)は

$$d_{\mathrm{S}} = 8 b_i b_j \tag{5}$$

また，演算量f(Flop)は

$$f = 2 b_i b_j b_k \tag{6}$$

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，

$$B = \frac{4}{b_i} + \frac{4}{b_j} + \frac{8}{b_k} \tag{7}$$

となり，キャッシュのサイズが許す限りブロック幅を増加させることで，B/F値を小さくすることができる．

3. Level-3 BLASの実装戦略

後藤の論文[4]で示されている設計を要約すると，Level-3 BLASを構成するための主要な部品が3つある．1つが，計算カーネルであり，一部のブロック幅が固定サイズのブロック行列積を行う部品である．このブロック幅をSIMDレジスタの幅に合わせるなどすることで，計算機アーキテクチャに最適化した計算カーネルを作る．次に，コピーカーネルであり，部分行列のデータを計算カーネルに合わ

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 B/c	32KB
L2 cache	24 cycle	12 GB/s	512KB
Remote L2	250 cycle		
Memory	302 cycle	164 GB/s	

せた順序に変換する部品である．近年の計算機においてはデータアクセス順序によってメモリバンド幅が大きく変わり，適切なデータ配置が重要となる．データの並び替えにはコストがかかるが，ブロック行列積によって，並び替えたデータを再利用できるので，データ並び替えのコストは隠蔽できる．最後に，ドライバーであり，適切な順序で計算カーネルとコピーカーネルを呼び出して，キャッシュ再利用性を高めてかつ，コピーカーネルの計算コストが隠ぺいされるように制御する部品である．

これらの部品を数種類ずつ作成し，適切に組み合わせることで Level-3 BLAS の多種類のサブルーチンを構成することができる．例えばドライバーは，単精度のルーチンと倍精度のルーチンとでほぼ同じものが使えるし，行列のデータ順序ごとにコピーカーネルを作成すれば，1 つの計算カーネルで複数のデータ順序に対応できる．

また，このように部品ごとに考えることは，性能を理解する上で役立つ．高い性能を実現するには，計算に直接かかわらない部分（コピーカーネルとドライバー）の実行時間を小さくし，計算カーネルが実行時間に占める割合を増やせばよく，これらの性能比が理想的な値になるようチューニングしていくことが高性能な Level-3 BLAS の構築する作業となってくる．

4. Knights corner と計算カーネルの詳細

以上の Level-3 BLAS の部品を計算機アーキテクチャに合わせて作成する．そこで KC に対して DSYRK をチューニングする上で必要な KC のアーキテクチャについて示す．

KC について特に重要なことは，57 以上のコアを持つマルチコア CPU であること，512bit 幅の SIMD 演算（FMA を含む）をもつこと，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があることである [1][2][3].

KC 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 Controler には GDDR5 メモリが接続されている．参考のため，Fang ら [7] による，型番が異なるがアーキテクチャの同じ Xeon Phi における，各種メモリ性能を測定した結果を表 1 に示す．今回用いたものは表の結果と比べて，メモリバンド幅が大きく異なるが，それ以外の値については参考になる．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，同時に 8 つの倍制度実数を計算する VPU を持っており，FMA 命令を使えば，1 サイクルで 16 演算を行うことができる．KC のパイプラインは 2 本あるが，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 FMA 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．

KC の命令デコーダーは 1 コアに複数のスレッドが実行されることを前提としており，1 スレッドのみでは高い性能が出せない設計となっている．要点を説明すると，KC は 1 サイクルに最大 2 命令を実行できるが，命令デコーダーは 2 サイクルに 2 命令までしかデコードできない．ただし，命令デコーダーは 4 並列であり，1 サイクルごとにスレッドを切り替えることができるため，2 つ以上のスレッドを 1 つのコアで実行すれば，1 サイクル 2 命令の命令供給ができる．1 コアに複数スレッドを流す利点は他にもあり，メモリアクセスのレイテンシーを一部隠蔽できることや，演算のレイテンシーも隠蔽できることがある．

4.1 計算カーネルの設計

以上の KC の特徴から，まず計算カーネルがどういったものになるかが決定される．KC での理論ピーク性能は，すべてのサイクルで FMA 命令を実行した場合の実行性能として換算されるから，計算カーネルにおいて u-pipe に流れる命令のうち何割が FMA 命令であるかが，性能の上限となる．検討した結果，この値が 9 割を超えるものとして，筆者は 2 つのパターンを見つけた．1 つ目のパターンは，結果の行列積の大きさが 8×30 となるもの（30 の値はレジスタ本数であり，30 以下の値に変更可能）であり，内積方向の長さは任意である．このパターンは BLIS で実装されているものであり，また，様々な状況証拠から MKL の実装もこの形となっているものと推測している．2 つ目のパターンは，結果の行列サイズは $8 \times b_i$ と，横方向に任意となっているかわりに，内積方向は 30 以下の値（レジスタ本数による）となるものである．両者の違いは，どのデータをレジスタ上の置くかであり，前者のパターンでは結果の行列 C の部分行列をレジスタ上に置くが，後者は，入力の行列 A の部分行列を置く．本稿では他のライブラリに倣って前者のパターンのみを用いている．後者のパターンについては，既存の実装が存在しないものであり，現在，実装と性能解析をしている．またアライメントがとりやすいように，8×30 ではなく，8×24 の結果となるものを作っているが，FMA 命令の比率に対する影響は最大でも

2% 以下の小さな値となる．

結果の行列のサイズが 8×24 の場合の計算順序を次の Pseudo Code に示す．

```
subroutine kernel8x24(C, A, I, J)
    W(1:8,1:24) = 0
    for k=K, K+bk-1
        X = A((J+1:8), k)
        for i=1, 24
            W(1:8, i) += X * A(I+i, k)
    C((I+1:8), (J+1:24)) += W(1:8, 1:24)
```

この計算は 8 要素のベクトル単位で行っており，KC の SIMD 演算に適している．はじめに，W は縦ベクトルの 8 要素が 1 つの SIMD レジスタを使い，合計 24 本のレジスタを使う．外側ループは内積方向に進めるループであり，ループの長さは任意に設定できる．内側ループは固定長のループであり，完全なループアンローリングを行う．再内側処理は，ある SIMD レジスタをスカラー倍して，あるレジスタに足しこむという形の処理となっており，KC の FMA 命令では 1 命令で実行できる演算パターンである．そこで，うまくループ処理のための演算などをスケジュールすると命令実行サイクルに占める FMA 命令の割合は $\frac{24}{26} \approx 0.923$ となる．

このように，結果の行列の横幅 24 は，何本レジスタを使うかで決められており，端数処理を完全に行うなら，使用するレジスタ本数が 1 から 24 に対応するように，24 種類のカーネルが必要となるが，これは大変なので，8 の倍数のもの（8, 16, 24）のみ作成した．これは，KC の SIMD のアライメントやキャッシュラインの幅に一致している点でも都合がよい値である．

4.2 キャッシュブロック化

KC はコア間で共有するキャッシュがないため，コアごとに独立のキャッシュブロックを処理するよう順番を決める．コア内のスレッドではキャッシュを共有するから，キャッシュブロックの処理はコア内のスレッドで並列化する．

基本的な手順は，ブロック行列積の式（2）における右側の行列 $A_{j,k}$ をキャッシュにコピーし，$A_{i,k}$ の 1 行を読み出すたびにその行と $A_{j,k}^{\top}$ とのベクトル行列積を計算する，という順序である．

実際の手順は，計算カーネルのブロック幅に合わせる必要があるから，1 行ごとに処理するのではなく，8 行 24 列ごとに処理しなければならない．いま $A_{i,k}$ と $A_{j,k}$ をブロック分割して

$$A_{i,k} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_x \end{bmatrix}, \quad A_{j,k} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_y \end{bmatrix} \tag{8}$$

と表記する．ただし $X_{\cdot}$ は $8 \times b_k$ 行列，$Y_{\cdot}$ は $24 \times b_k$ 行列である．またこれに合わせて $C_{i,j} = \{W_{i_r,j_r}\}$ のように分割する．このようにあまりなく分割するために $b_i = 8x$, $b_j = 24y$ が成り立つものとする．計算カーネルを 1 回呼び出すことはある i_r, j_r に対して $W_{i_r,j_r} \leftarrow W_{i_r,j_r} + X_{i_r} Y_{j_r}^{\top}$ を計算することである．そこで，キャッシュブロックの計算を次のようなブロック列優先の計算順序を行う．

```
subroutine cblock(W, X, Y)
    for j_r = 1:y
        for i_r = 1:x
            W_{i_r,j_r} ← W_{i_r,j_r} + X_{i_r} Y_{j_r}^T
```

この手順を用いれば，W_{i_r,j_r} は最内側処理のみに使い，X_{i_r} は内側ループの間だけ，Y_{j_r} は処理全体で繰り返し使われる構造になるため，Y_{j_r} のみを L2 キャッシュに常駐させ，他の 2 つは一時的にしか L2 キャッシュに乗せずに計算ができる．

実際に上のプログラムの想定通りにキャッシュ内のデータをコントロールするために，Prefetch 命令とストリームストア命令を適宜使う．またブロックサイズを適切に決める必要があるが，様々なパラメータを手動で探索したとき良い性能が得られたものは，$b_j = 144$，$b_k = 200$ であり，512KB の L2 キャッシュサイズの半分に近い 225KB の大きさとなった．

`cblock` の並列化は外側ループで行う．`cblock` では省略しているが，DSYRK では上三角部分のみ計算するため，`cblock` の内側ループは，j_r の値に応じて，短くなる場合がある．そのため，スレッド間で処理の量が大きく変わらないように，`cblock` の外側ループはサイクリック分割にする．

データ並び替えについては，`cblock` を呼び出す直前のタイミングに，A の部分行列を Y にコピーすることで行う．X の並び替えについては，コア間で共有のラストレベルキャッシュがある場合には，より大きなブロックサイズでのブロック行列積を考えて，

```
for k=1:n:b_k
    for i=1:m:b_i
        X ← A(i:b_i, k:b_k) // X の並び替え
        for j=1:m:b_j
            Y ← A(j:b_j, k:b_j) // Y の並び替え
            cblock(C(i:b_i, j:b_j), X, Y)
```

のようにすれば X についての最大限のデータ再利用性が得られる．ただし，KC には共有のラストレベルキャッシュが

なく，X はメモリ上に置かなければならないため，キャッシュ再利用性の意味では，あまり効果がない．本稿での実験結果は Y への並び替えだけ行い，X については直接 A を使う形のものとなっている．

5. DSYRK の並列化

高い性能で並列計算をするためには，単体性能と合わせてワークバランスを向上することが必要である．計算カーネルのブロック幅や，キャッシュブロック幅，そして並列化軸によっては同期処理などがあり，これらの影響を考えて計算領域を分割しなくてはならない．

計算領域の 3 つの軸のうち 2 つは結果の行列の要素ごとの並列性であり，処理間に依存性がないため並列化のプログラミング自体は容易であるが，領域の形状が三角であり，ワークバランスをとることが難しい．もう 1 つの軸は内積計算における総和方向の並列性であり，並列化をした場合には，同期をとって総和を計算しなければならず，並列化のプログラミング自体は複雑となる．

このように困難なことが異なる 3 つの軸をうまく利用するため，2 つの異なるアプローチに基づく 3 つの並列化手法を作成した．1 つ目のアプローチは，三角部分のワークバランスをとることを放棄する代わりに，内積の並列性を用いて，並列プログラミングの手法を工夫することで並列化するものである．もう 1 つは逆に，内積の並列性を放棄して，三角部分のワークバランスをとることだけに注力したものである．前者の手法は，結果的に 3 つの軸すべてを用いるため，3 次元動的分割法と呼び，後者は，三角部分の並列性のうち 1 つを使うものと 2 つを使うものとで，1 次元静的分割法，2 次元静的分割法と呼ぶことにする．

5.1 3 次元動的分割法

3 次元動的分割法の基本的な構造は，スレッドプールパターンである．つまり，問題を独立に実行可能な小さな塊（タスク）に分割し，遊んでいるスレッド（KC においてはコア）へタスクを割り当てていく．タスクの管理にはキュー構造が用いられ，今の仕事が終わったスレッドは，キューから新しいタスクを取得（Pop）する，という動作を繰り返す．スレッドプールパターンは，十分多くのタスクがあればスレッド間の実行時間のばらつきが少なく，また，タスクごとの実行時間の違いも自動的に吸収する．

3 次元動的分割法は，単体性能を高めつつ多くのタスクを作ること，また，同期の必要性を減らすことを目標として，次の 3 つの特徴を有する．1 つは，内積方向も含めた 3 つの軸すべてを並列化に用いること，次に，内積方向の並列化によって生まれる同期の回数を減らすため，複数のキューを用いること，そして，キャッシュブロック幅を分割の基準とすることである．

キャッシュブロック幅は，データ再利用回数の上限であるから，これ以上大きなブロックで分割しても，性能はあまり向上しない．そのため，計算領域をちょうどタスクブロック幅で切ることは単体性能の面では理に適っている．しかし，キャッシュブロック幅は大きな値であるため，内積方向の軸もキャッシュブロック幅で切り，並列化に利用することで，タスク数を大きくする．

しかし内積方向も利用したことで同期処理が問題となる．そこで，総和の回数を減らすように，優先的にとるタスクを決める．これは，一度あるコアにあるタスクを割り当てれば，そこから内積方向に進んだタスクを優先的にそのコアへ割り当てる方策である．すると，同じ内積方向に進んでいる間は部分和をローカルキャッシュに保存しておくことができ，内積の終端にたどり着いたときにはじめて同期をとってローカルキャッシュの部分和を結果の行列に足しこむということができる．

このようなスケジューリング処理を行うために次のようなスケジュール機構を作成した．まずデータ構造として，複数のタスクキューを持つ．タスクキューが保持する情報は 4 つの整数値である．まず，どの $C_{i,j}$ に対応しているか，i と j の値，そして，内積がどのブロックまで進行しているかを表す k の値がある．つまり，$k = 10$ ならば，$A_{i,1}A_{j,1}^{\top}, A_{i,2}A_{j,2}^{\top}, \ldots, A_{i,9}A_{j,9}^{\top}$ までの計算が割り当て済みであることを表す．タスクキューは $C_{i,j}$ ごとに必要であるが，コア数分しか同時に利用しないから，データ量は m によって変化しない．k はアトミック変数であり，この変数をアトミックに 1 増やす（fetch-and-add）ことが，タスクキューから新しいタスクを Pop することに相当する．例として，いま k の値が 10 のときにあるコアが Pop を行うと，そのコアが次の処理するのは $A_{i,10}A_{j,10}^{\top}$ の計算であり，k は次のタスクである 11 を指すことになる．

タスクキュー内で処理を進めた結果，最終的にキューが空となる（k の値が $k_{\max}$ を超える．）そうすると，そのタスクキューを使っていたコアは別のタスクキューを探索する．このとき，競合を減らすため，まだどのコアも使っていないタスクキューがあれば，それを選択し，もしなければ，使用中のタスクキューのうち，“残りの計算量” が最も多いものを選択する．残りの計算量は，そのタスクキューが担当する $C_{i,j}$ の要素数に，タスクキューの現在の長さ $k_{\max} - k$ をかけて，$n_{\mathrm{q}} + 1$ で割ったものとして計算する．これは，1 つのタスクキューに過剰な数のコアが割り当てられないようにするためヒューリスティックに決められた基準である．

以上を行うスケジューラーができれば，他の処理は次のように書ける．

```
(i, j, k) = (-1, -1, -1)
C̄ ← O
for
```

```
(ni, nj, nk) = scheduler()
if(ni > maxi) break
if(ni != i || nj != j)
    C_{i,j} ← C_{i,j} + C̄,  C̄ ← O
(i, j, k) = (ni, nj, nk)
C̄ ← C̄ + A_{i,k} A^⊤_{j,k}
```

つまり，i, j の値が前回と変わらなければローカルキャッシュ $\bar{C}$ に対して部分和を追加し，異なっていれば，$C_{i,j}$ に $\bar{C}$ を書き戻して，次の部分和の計算に移る．

書き戻し処理においてはロックが必要であるが，$C_{i,j}$ の各列の先頭アドレスのハッシュ値を計算し，その値に対応するロックをとるようにする．これは，細粒度なロックをとりながら，m にロックの数が比例しないために行っている．ハッシュ値の最大値を適切に決めることで，ロックの数と，本来不要な同期の割合を調整できる．

5.2 1 次元静的分割

1 次元静的分割は，結果の行列 C をコア数 N_{C} の列ブロックに分割し：

$$C = \left[\begin{array}{cccc} C_1 & C_2 & \cdots & C_{N_{\mathrm{C}}} \end{array} \right], \tag{9}$$

それぞれの列ブロックをコアに割り当て，並列に処理するものである．ここでの目標は，各列ブロックの列幅のみを調整して，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，列ブロックのうち上三角部分に属する要素数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 $c_1, c_2, \ldots, c_{N_{\mathrm{C}}}$ とおき，列ブロックの開始位置 $i_s = \sum_{t=1}^{s} c_t$（ただし $i_0 = 0$）とおく．まず i_s が実数の場合に演算量が均等にする手法を考えると，次を満たすように決めればいいことがわかる．

$$\frac{i_{s+1}^2 - i_s^2}{2} = D \tag{10}$$

ただし，$D = \frac{m^2}{2N_{\mathrm{C}}}$ であり，上三角部分の要素数をコア数で割った値である．OpenBLAS 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．

$$i_{s+1} = \mathrm{round}(\sqrt{i_s^2 + 2D}). \tag{11}$$

ただし，終端は行列サイズと一致するよう $i_{N_{\mathrm{C}}} = m$ とする．round は丸め関数で，計算カーネルのブロック幅に丸める．

この手法の問題点は，実数での解と整数での解が異なる点，そして，逐次的に丸め処理を行っているため，丸め誤差が累積する可能性がある点である．要素数がなるべく一致するようにすることを目標にするならば，本来求めたい値は

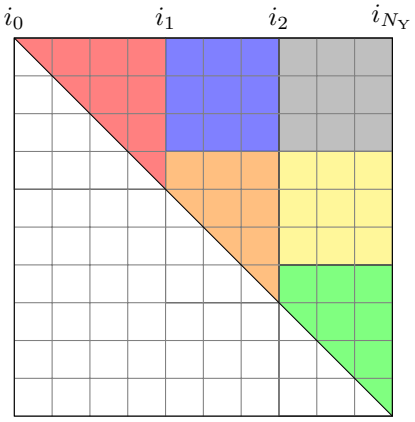

図 1　2 次元静的分割の例
Fig. 1　An example of the 2D static blocking.

$$\begin{array}{ll} \text{min.} & \max\{\sum_{i=i_{s-1}}^{i_s-1} i | s = 1, 2, \ldots, N_{\mathrm{C}}\}, \\ \text{sub.to.} & 0 = i_0 \le i_1 \le i_2 \le \cdots \le i_{N_{\mathrm{C}}} = m, \end{array} \tag{12}$$

の最適解である．この最適解を用いるのが，1 次元静的分割法である．

これは整数最適化問題であるが，実は簡単に解けるクラスの問題であり，具体的には，式（11）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 の逆問題のとしても見ることができる [8]．この計算量は $O(N_{\mathrm{C}} \log_2 m)$ であり，試行した範囲では二分探索は 10 反復程度で終了するから，最適化にかかるコストは十分小さい．

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（10）で概算すると，KC においては $m = 5000$ のとき $c_{N_{\mathrm{C}}}$ 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．

5.2.1 2 次元静的分割

1 次元静的分割を簡単に拡張したものが 2 次元静的分割である．2 次元静的分割は，1 次元静的分割によって列ブロックに分割した後，列ブロックを複数の行ブロックに分ける，という手順で計算領域を分割する．ここで，図 1 のように，割り当てるコア数を列ブロックごとに変化させることで，2 次元分割したブロックが正方形に近い形になるよう調整する．

まず，列ブロックの個数を N_{Y}，それぞれ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．

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

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て，$O(N_{\mathrm{Y}} \log_2 m)$ の計算量となる．N_{Y} や m_s の設定

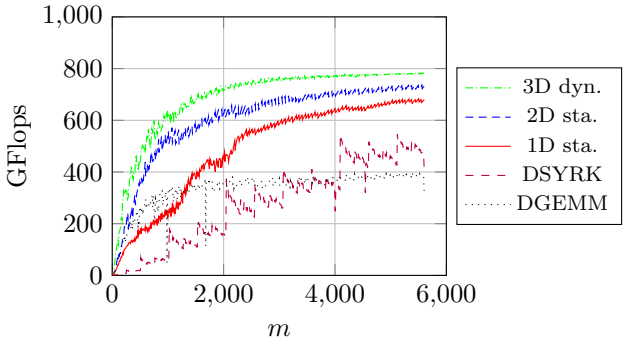

図 2　n を固定したときの実行性能
Fig. 2　The performance results of DSYRK with fixed n.

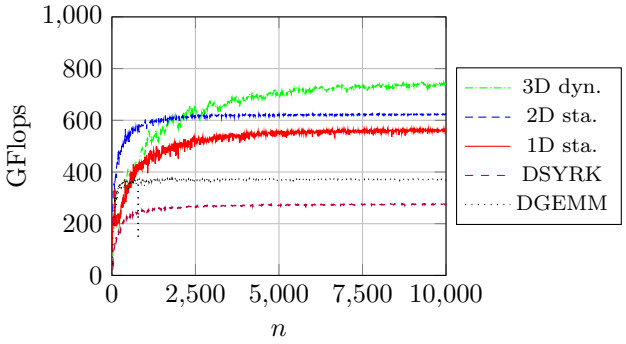

図 3　m を固定したときの実行性能
Fig. 3　The performance results of DSYRK with fixed m.

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，$N_Y \approx \sqrt{N_C}$，$m_s \approx s$ を基本として設定することにする．

列ブロックに分割した後，行ブロックに分割する手法も多く考えられるが，ここでは簡単に，ある列ブロックの行数の平均値 $\frac{i_s+i_{s+1}}{2}$ をコア数で割り，8 の倍数に繰り上げた行数を，1 ブロックあたりの行数とする．

6. 実行性能の測定結果

実行性能の測定は Xeon Phi の 3120A の 57 コア中，56 コアを用いて行った．このときの理論ピーク性能は 985.6GFlops である．実行性能は演算量 E_{all} 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．比較対象として MKL の version 11.2.1 に含まれる BLAS の関数として，DSYRK と DGEMM を用いた．DGEMM は C の対称性を用いないから，上記の定義で計算した実行性能は DGEMM の性能の約半分となる．プログラムは C++と SIMD 演算の intrinsic 命令を用いて作成した．コンパイルは icpc の 15.0.1 である．

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影響を見るために，$n = 10,000$ 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．また，2 次元静的分割法においては，列ブロック数のコア数の割り当て方が任意に設定できるが，ここでは，$N_Y = 14$，m_s は m_1 から $1, 1, 2, 2, \ldots, 7, 7$ とした．また 3 次元動的分割法では $b_i = 8b_j = 1,152$ に設定した．

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 に示す．今回実装したどの並列化手法でも MKL の DSYRK より高速となっていることがわかる．また，1 次元，2 次元，3 次元の順で高い性能となっており，並列化軸を増やすことにより性能が向上している．2 次元分割と 3 次元分割とでは，m が十分大きい場所でもまだ性能に差があるが，静的分割法でのワークバランスをとることの難しさが表れているものと思われる．

次に，$m = 2,400$ と固定して n を変化させたときの性能を図 3 に示す．m を変化させたときと異なり，3 次元分割は n の小さいところでは性能を活かせていない．これは，タスク数が少なくて，実行時間のばらつきやローカルキャッシュの書き戻しのコストを隠蔽できていないためだと考えられる．1 次元や 2 次元分割では，n に関して並列化していないため，n が一定以上大きくなると性能の変化が少なくなる．これは MKL の関数にも同じ傾向が言える．

7. まとめと今後の課題

7.1 まとめ

本稿ではまずマルチコア CPU に対する BLAS の実装手法と，Xeon Phi（Knights Corner）に対する行列積の実装について，既存研究の手法ををまとめた．そのうえで，DSYRK を並列化する手法を検討し，DSYRK が持つ 3 つの並列化軸のうち，3 つすべてを使い動的な分割を行う 3 次元動的分割法，1 つまたは 2 つを使い静的な分割を行う 1 次元・2 次元静的分割法を示した．

そして，Xeon Phi 3120A 上で DSYRK の性能測定を行い，3 つの並列化手法と MKL の DSYRK，DGEMM とを比較したところ，内積方向の次元が十分に大きい場合は 3 次元動的分割法が最も高速であったが，逆に内積方向の次元が小さい場合には 2 次元静的分割法が 3 次元動的分割法よりも高速となる場合があった．

このことから，Xeon Phi のように多数のコアを持つ CPU に対しては，2 次元静的分割法や 3 次元動的分割法のような，他のライブラリで使われているものよりも複雑な並列化手法が有効となってくるだろうと考えられる．

7.2 今後の課題

静的分割法では演算量が均一となるように分割したが，本当にしたいことは計算時間の最小化である．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，最適化の対象にすることが難しい．そこで，何らか手法で性能モデル化をすることが考えられる．このとき，

高い精度のモデル化手法を構築すること，そのモデル上で計算時間を均等に分割する手法を構築することが課題である．

現在の実装では，データがアライメント整合していることを条件としている．完全な実装のためには，この条件を無くしたい．その場合には，現在の Y だけ並び替えを行うのではなく，X の並び替えも行う必要がある．このときの実行性能に値する影響を計測することが課題である．

また，計算カーネルのパターンのうち片方は BLIS などでも実装されており，実行性能がよく解析されているが，もう一方のパターンは未知のものであり，実装したうえで詳細な性能測定を行いたい．2 つのパターンは自由度のある軸が異なるため，並列化のしやすさが異なる．そのため，こちらの計算カーネルでも実行性能の測定を行い，スレッド並列化手法の影響を調べることが課題である．

参考文献

[1] 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).
[2] 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).
[3] 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)).
[4] Goto, K and Geijin, R.A..: “High-Performance Implementation of the Level-3 BLAS,” *ACM TOMS*, Vol. 35, Issue 1, No. 4 (2008).
[5] Smith, T.M., Geijin, R.A. Hammond, J.R. and Van Zee, F.G.: “Opportunities for Parallelism in Matrix Multiplication,” *Univ. Texas Techinical Report*, TR-13-20, FLAME Working Note 71 (2013).
[6] 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).
[7] 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).
[8] 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).