インテル® Xeon® プロセッサーおよび インテル® Xeon Phi™ コプロセッサーで 共通のコードを使用した正方行列のマルチスレッド転送

Colfax Research
Andrey Vladimirov

2013 年 8 月 12 日

概要

線形代数の基本的な操作であるインプレースの行列転置は、メモリー帯域幅に制限される操作です。転置の理論的な最大パフォーマンスはメモリーコピー帯域幅ですが、転置操作で連続していないメモリーアクセスが行われるため、実際のパフォーマンスは一般に低くなります。転置レートとメモリーコピー帯域幅の比率は、転置アルゴリズムの効率を示します。

このホワイトペーパーでは、並列のインプレース正方行列置換の効率良い C 言語実装について説明します。この実装は大きな行列で、インテル® Xeon® プロセッサーでは 49GB/秒 (効率 82%)、インテル® Xeon Phi™ コプロセッサーでは 113GB/秒 (効率 67%) の転置レートを達成しました。コードは、さまざまなプラグマベースのコンパイラー・ヒントとコンパイラー引数を使用してチューニングされています。スレッド並列処理は OpenMP* で制御し、ベクトル化はインテル® コンパイラーにより自動的に実装されます。このアプローチでは、同じ C コードから、プロセッサーとインテル® MIC アーキテクチャー向けに効率良い実行ファイルを生成することができます。ベンチマークには、インテル® Xeon Phi™ コプロセッサー 7110P を使用しました。

目次

1. 行列の転置はなぜ難しいのか 2
2. 実装 4
2.1. 幌馬車の速度: *最適でない* 行列転置 4
2.2. チームペニング: ベクトル化パターンの調整 5
2.3. スクエアダンス: 行列走査アルゴリズム 6
2.4. ギャロップで駆ける: プリフェッチと一時的でないストア 8
2.5. 荒れ馬を馴らす: スレッド・アフィニティー 9
2.6. 良い行列サイズ、悪い行列サイズ、最悪の行列サイズ 9
3. ベンチマーク 11
3.1. 転置レート 11
3.2. ハードウェア・システム構成 11
3.3. STREAM ベンチマーク 11
3.4. コンパイルと実行 12
3.5. 結果: 良い行列サイズ 13
3.6. 結果: 良い行列サイズと悪い/最悪の行列サイズ 15
3.7. インテル® MKL との比較 16
4. まとめ 17
A. ソースコード 18
参考文献 (英語) 19

Colfax International (http://www.colfax-intl.com/) 社は、ワークステーション、クラスター、ストレージ、パーソナル・スーパーコンピューティング向けの革新的かつ専門的なソリューションを導くリーディング・プロバイダーです。他社では得られない、ニーズに応じてカスタマイズされた、広範なハイパフォーマンス・コンピューティング・ソリューションを提供します。すぐに利用可能な Colfax の HPC ソリューションは、価格/パフォーマンスの点で非常に優れており、IT の柔軟性を高め、より短期間でビジネスと研究の成果をもたらします。Colfax International 社の広範な顧客ベースには、Fortune 1000 社にランキングされている企業、教育機関、政府機関が含まれています。Colfax International 社は、1987 年に創立された非公開企業で、本社はカリフォルニア州サニーベールにあります。

1. 行列の転置はなぜ難しいのか

コンピューター・メモリーで、2 次元配列は通常、行優先形式または列優先形式で格納されます。行優先形式では、行がメモリーに連続して配置され、各行内の隣接する要素も連続しています。行優先行列を列方向に処理する場合、メモリーが行の長さと同じストライドでアクセスされる行方向の処理よりも遅くなります。この場合、データを転置して、より効率的な行方向でメモリーにアクセスしたほうが速くなることがあります。

転置は方程式 1 のように行と列を交換するだけなので、簡単そうに見えますが、コンピューター・メモリーで転置を行う場合、行列の要素を交換する順序がパフォーマンスに大きく影響します。共有メモリー行列置換では、行列データを読み取り、異なる場所に書き込まなければならないため、理論的な最大パフォーマンスはメモリーコピーの帯域幅と等しくなります。しかし、アーキテクチャーを意識しないアルゴリズムでは、実際のパフォーマンスは非常に低くなります。

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \dots & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}, \quad A^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ & \dots & & \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix}. \qquad (1)$$

インテル® MIC アーキテクチャーを含む多くのコンピューター・アーキテクチャーは、メモリーのデータが連続してアクセスされるときに高いメモリー帯域幅を達成します。しかし、行列が行優先形式や列優先形式で格納されている場合でも、転置の際に、連続していないメモリー操作がどこかで発生します。例えば、行優先形式の場合、アルゴリズムが元の行列を行方向に読み取る際、メモリーへの読み取りアクセスは連続していますが、転置したデータを書き込むときは、隣接する要素が行の長さと同じストライドで分かれた列に書き込みます。逆に、(行方向に) 連続して書き込む場合は、(列方向に) 1 ストライドで読み取りを行います。図 1 は、この問題点を示しています。

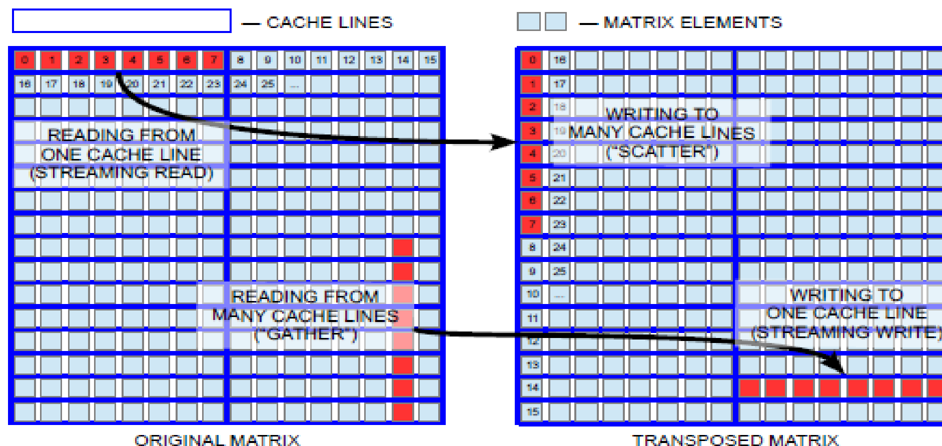

図 1: キャッシュラインに 8 つの行列の要素を格納できるコンピューター・アーキテクチャーで行優先形式による正方行列の転置。アルゴリズムが連続して読み取りを行う場合、分散して書き込みを行う必要がある。逆も同様。

連続するメモリーアクセスの必要性は、メモリー・アーキテクチャーにより生じます。最高の帯域幅を達成するため、インテル® Xeon Phi™ コプロセッサーを含む多くのコンピューター・アーキテクチャーでは、メモリーはキャッシュラインの粒度でアクセスされます。コアがメモリーの 1 つのデータ要素を読み書きする場合、メモリー・コントローラーはその要素を含む**キャッシュライン**全体を RAM からキャッシュまたはコアに転送します。インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーのキャッシュラインは 64 バイトです。つまり、行列の倍精度 (8 バイト) の要素 a_{11} をメモリーから取得した後、a_{12} から a_{18} にアクセスする場合はコストがかかりませんが、a_{19} や a_{21} にアクセスする場合は追加のメモリーアクセスが必要になります [1]。図 1 に示すように、行列転置の性質により、アルゴリズムは、1 つのキャッシュラインから読み取って複数のキャッシュラインに書き込むか、その逆のいずれかです。さらに、Linux* では仮想メモリーが仮想メモリーページにマップされるため、大きなストライドアクセスが多すぎると、ページマップのルックアップによる遅延も発生します。

非ローカルメモリーへのアクセス問題に対処するには、分散されたメモリーアクセスの時間的局所性を高める必要があります。そうすることで、複数のキャッシュラインに書き込んだり (スキャッター) または複数のキャッシュラインから読み取る (ギャザー) 場合でも、これらのキャッシュラインがコアのベクトルレジスター内または高レベルのキャッシュ内にある間に処理が行われます。結果として、データが高速なメモリーでシャッフルされ、メインメモリーのアクセスが合理化されます。このような時間のデータ局所性の最

[1] この例は、a_{11} がキャッシュラインの先頭にあり、行列が行優先順で格納されていると仮定しています。

© Colfax International, 2013 — http://research.colfaxinternational.com/

適化は、ループタイリング (ループ・ブロッキングとしても知られる) や再帰分割統治 (キャッシュを意識しないアルゴリズムとしても知られる) によって達成することができます。これらのアプローチは、複数の著者 ([1]、[2]、[3]) により発表されています。要約は [4] を参照してください。

メモリー・トラフィックの最適化に加えて、効率良い行列転置には並列処理が必要です。複数のコアが協調して動作する場合に限り、インテル® MIC アーキテクチャーのメモリー・コントローラーと各コアのローカルキャッシュをすべて利用できます。しかし、次のような条件を満たしていなければなりません。

a) すべてのコアでバランスよく処理できる十分なワークがあること (つまり、十分な細粒度の並列処理が可能であること)。

b) フォルス・シェアリングが発生しないように、一度に 1 つのコアのみ指定されたキャッシュラインにアクセスすること (つまり、十分な疎粒度の並列処理が可能であること)。

c) コア間の同期がほとんど (あるいは全く) 必要ないこと。

既存のライブラリーの多くは、分散メモリー転置 (行列データがメモリーを共有しない複数の計算ノードに格納される場合) に適した効率良い関数を提供しています ([5]、[6])。分散メモリー (アウトオブコアとも呼ばれる) 転置は重要であり、多くの点で難しい問題ですが、共有メモリーで効率良いマルチスレッド転置を必要とするアプリケーションには対応できません。インテル® MKL ライブラリー [7] には、インプレースの行列転置とスケーリングのマルチスレッド実装 (`mkl_?imatcopy` 関数) が用意されています。`mkl_?imatcopy` は非正方行列をサポートしていますが、そのパフォーマンスは特定のマルチスレッド化と行列サイズ用に高度にチューニングされています [2]。セクション 3.7 では、より広範囲の行列サイズで、インプレース正方行列転置を `mkl_?imatcopy` の最大 1.5 倍で実行できることを示します。GPU の転置についても、これまで研究されてきました (例 [8]、[9])。しかし、マルチコアおよびメニーコア・アーキテクチャーでの (キャッシュよりも) 大きな行列の効率良い共有メモリー転置は、ソフトウェアで過小評価されています。これが、ここで説明するアプリケーションを開発した理由です。

正方行列転置の重要性に加えて、この問題は、マルチコア・プロセッサーとメニーコア・コプロセッサーの一般的な最適化手法を示す良い例です。スレッド並列処理の実装と同時にキャッシュ・トラフィックを最適化することは、多次元配列の処理を含む一般的な問題でよくあることです。

これらの点を考慮して、インテル® MIC アーキテクチャー向けの効率良い行列転置アルゴリズムを実装するにあたり、次の 2 つの目標を設定しました。

1) 行列転置は線形代数アプリケーションで広く使用されているため、さまざまな用途に対応できる行列転置用のツールを作成すること。

2) 汎用プロセッサーおよびインテル® MIC アーキテクチャー・コプロセッサー向けにメモリー・トラフィックを最適化する一般的な手法を習得して共有すること。

[2] インテル® MKL 11.0.5 (ビルド日 20130612) 時点。

© Colfax International, 2013 — http://research.colfaxinternational.com/

2. 実装

Colfax Research の別のホワイトペーパー [4] で、インテル® MIC アーキテクチャー向け転置アルゴリズムを初めて実装したケースについて説明しました。この実装では、2 つの最適化手法 (入れ子のループタイリングおよびキャッシュを意識しない再帰) を調査し、各手法について 2 つの並列フレームワーク (インテル® Cilk™ Plus および OpenMP*) をテストしました。[4] で報告した結果は量的には不十分でしたが、将来の作業のロードマップを提供できました。

ここでは、インプレースの正方行列転置アルゴリズムの新しい実装について説明します。この実装は、[4] で見つかった問題を修正したもので、すべての行列サイズで優れたパフォーマンスを実現します。このセクションでは、この実装の詳細と、この結果をもたらした最適化プロセスについて解説します。新しい実装のコードは、リスト 9 を参照してください。ベンチマークのコードは、Colfax Research の Web サイト [3] から入手できます。

2.1. 幌馬車の速度: *最適でない* 行列転置

リスト 1 の単純な転置アルゴリズムは、セクション 1 で説明した最適でないメモリー・アクセス・パターンによりパフォーマンスが妨げられます。インテル® Xeon® プロセッサーを搭載したホストシステムでは、このアルゴリズムの転置レートは、後述する最適化されたアルゴリズムの 60 ～ 70% になります。インテル® Xeon Phi™ コプロセッサーでは、最適化された実装の 25 ～ 30% に過ぎません。

リスト 1: 最適化されていない並列転置アルゴリズムでは十分なパフォーマンスが得られない。FTYPE は単精度では float、倍精度では double。

```
1  #pragma omp parallel for
2  for (int i = 0; i < n; i++)
3    for (int j = 0; j<i; j++) {
4     const FTYPE c = A[i*n + j];
5      A[i*n + j] = A[j*n + i];
6      A[j*n + i] = c;
7    }
```

転置レートを向上するための鍵は、このアルゴリズムをタイリングまたは再帰を使用して変更することです。この最適化では、行列を多くの部分行列 (タイル) に分割し、タイルのセットを走査する並列アルゴリズムが選択されています。各タイルは、転置マイクロカーネルと呼ばれるコードで (1 つのスレッドにより) シリアルに転置されます。このアプローチをリスト 2 に示します。

リスト 2: タイリングによりデータ局所性を向上させた並列転置アルゴリズム

```
1  #pragma omp parallel for
2  for (int tile = 0; tile < nTiles; tile++) {
3    // Traversal of the set of tiles:
4    const int ii = // ... choose the x-location of the tile
5    const int jj = // ... choose the y-location of the tile
6
7    // Tile transposition microkernel:
8    for (int i = ii; i < ii+TILE; ii++)
9      for (int j = jj; j < jj+TILE; jj++) {
10       // ... swap A_ij with A_ji
11       const FTYPE c = A[i*n +
12       j]; A[i*n + j] = A[j*n +
13       i]; A[j*n + i] = c;
14     }
15 }
```

ここでは、走査アルゴリズムの選択と転置マイクロカーネルの最適化について説明します。

[3] http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx

2.2. チームペニング: ベクトル化パターンの調整

以前のホワイトペーパー [4] のタイル走査アルゴリズムは、行列の主対角線上および行列の端のタイルを含む、行列のすべてのタイルを走査しました。[4] の 1 タイルの転置マイクロカーネルの実装をリスト 3 に示します。

リスト 3: [4] の最適でない転置マイクロカーネルに主対角線または行列の端に達したことの確認を追加

```
const int iMax = (n < ii+TILE ? n : ii+TILE);
for (int i = ii; i < iMax; i++) {
  const int jMax = (i < jj+TILE ? i : jj+TILE);
#pragma loop count avg(TILE)
#pragma simd
  for (int j = jj; j<jMax; j++)
    {
    const FTYPE c = A[i*n + j];
    A[i*n + j] = A[j*n + i];
    A[j*n + i] = c;
  }
}
```

ここで、TILE はコンパイル時に既知の定数で、経験的に TILE=32 を選択しています。j の内部ループには可変上限 jMax を指定し、内部ループのインデックス j が外部ループのインデックス i を超えないように制御します。この値を指定しないと、対応する行列の要素は 2 回転置されます。通常は、jMax==jj+TILE です (タイルが行列の主対角線上にない場合)。また、i の外部ループには可変上限 iMax を指定し、タイルが行列の端にある場合、マイクロカーネルが境界外の行列にアクセスしないようにします。ほとんどの場合、iMax==ii+TILE です (タイルが行列の端にない場合)。

このマイクロカーネルは一般的ですが、j の内部ループの反復数がコンパイル時に既知でないため、最適ではありません。結果として、コンパイラーはこのループに対して複数の実行コードを実装し、実行時にループカウントのランタイム値が SIMD ベクトル長の倍数かどうかに基づいて適切なコードパスを実行するランタイムチェックを生成します。コンパイラー・ヒント "#pragma loop count avg(TILE)" は、最大ループカウントが jMax==jj+TILE (主対角線上と行列の端のタイルを含まない場合) であることをコンパイラーに知らせます。しかし、これだけではまだ、転置されるタイルごとにランタイムチェックと実行パスの選択を行う必要があります。

ループカウントが一定になるようにマイクロカーネルを変更すると、ベクトル化パターンが調整され、コンパイラーが効率良い実行コードを生成できるようになり、転置のパフォーマンスを向上できます。そのためには、マイクロカーネルが「本体」タイル (主対角線上および行列の端でないタイル) にのみ適用されるように、タイル走査アルゴリズムを変える必要があります。

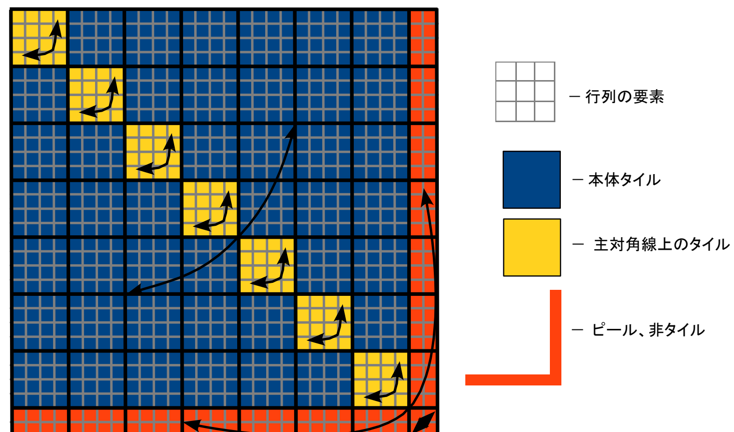

図 2: 1 タイルの転置マイクロカーネルを簡素化するためのタイル走査アルゴリズムの変更。主対角線上のタイルと行列の端のピールを含まない本体タイル (青で表示) は最適化されたマイクロカーネル (リスト 4) で処理される。残りのタイル (主対角線上のタイルと行列の端のピール) は別々に処理される。

図 2 に示すように、現在の実装では行列を 3 つの領域に分割しています。

1. 本体タイル (青で表示) は、行列データの大部分を含んでいます。これらのタイルには、簡素化された転置マイクロカーネル (リスト 4 の上) が使用されます。主対角線と行列の端を確認する必要はありません。
2. 主対角線上のタイル (黄色で表示) は、異なるマイクロカーネル (リスト 4 の左下) を使用して、別のループで転置されます。i の内部ベクトルループの終了条件は i>=j です。このループのベクトル形式は効率良くありませんが、正しい転置結果を生成するために必要です。
3. 最後に、行列の端のタイル (ピール) は、3 つ目のマイクロカーネル (リスト 4 の右下) を使用して 3 つ目のループで転置されます。境界外のメモリーアクセスを回避するため、端に接近しているかどうかを確認しています。このループでは、#pragma simd が使用されていないため、自動ベクトル化は必須ではありません。内部ループの長さが短いため、ベクトル化は逆効果になる可能性があります。

タイルサイズが行列サイズ n に依存しないため、行列本体のタイルの数は $O(n^2)$ でスケーリングします。主対角線上のタイルとピールタイルの数は $O(n)$ でスケーリングします。そのため、n が大きい場合、本体タイルのワークの割合は漸近的に 100% に近くなります。

```
1  // Tile transposition microkernel
2  // for main body tiles
3  for (int j = jj; j < jj + TILE; j++) {
4  #pragma simd
5    for (int i = ii; i < ii + TILE; i++) {
6      const FTYPE c = A[i*n +
7      j]; A[i*n + j] = A[j*n +
8      i]; A[j*n + i] = c;
9    }
10 }
```

```
1  // Tile transposition microkernel
2  // for tiles on the main diagonal
3  for (int j = jj; j < jj+TILE; j++) {
4  #pragma simd
5    for (int i = ii; i < j; i++)
6      {
7      const FTYPE c = A[i*n + j];
8      A[i*n + j] = A[j*n + i];
9      A[j*n + i] = c;
10   }
11 }
```

```
1  // Transposition algorithm
2  // for elements at the edges of the matrix
3  const int nEven = n - n%TILE;
4  for (int j = 0; j < nEven; j++) {
5    for (int i = nEven; i < n; i++)
6      {
7      const FTYPE c = A[i*n + j];
8      A[i*n + j] = A[j*n + i];
9      A[j*n + i] = c;
10   }
11 }
```

リスト 4: 最適化された転置マイクロカーネル。上: 大部分のタイルに使用 (ループカウントはコンパイル時に既知で一定)。
左下: 主対角線上のタイルに使用。右下: 行列の端の要素 (ピール) に使用。

1 つの汎用マイクロカーネルですべてのタイルを転置した場合 ([4]) に比べて、行列を 3 つの領域に分割して「本体」領域に最適化されたマイクロカーネルを使用すると、コードの量は増加し、抽象的でなくなります。しかし、パフォーマンスは約 20% 向上します。

2.3. スクエアダンス: 行列走査アルゴリズム

リスト 4 のコードのインデックス ii および jj は、1 つのタイルの左上要素の列と行です。ii と jj の値を選択するアルゴリズムにより、タイルのセットが走査される順序が決まります。このアルゴリズムで重要なのは、複数のプロセッサー・コアにわたってワークを並列化することです。

以前のワーク [4] では、次の 2 つの行列走査方法を実装しました。

1. 2 つの入れ子のループを使って、タイルの行 (外部ループ) と列 (内部ループ) をシーケンシャルにインクリメントする方法。この方法では、外部ループは並列化されますが、内部ループは各スレッドでシリアルに実行されるため、小さな行列サイズでは十分な並列処理が行われません。この場合、並列ワークアイテムの数 (タイルの行数) は ≈n/TILE です。TILE の経験的な最適値は 32 で、インテル® Xeon Phi™ コプロセッサーは (モデルに応じて) 240 または 244 のスレッドをサポートします。8000 × 8000 未満の行列サイズでは、n/TILE は 240 未満であるため、すべてのスレッドを占有するのに十分なワークがありません。また、小さな値の ii は大きな値の ii よりもワークが少ないため、ロード・インバランスが発生することがあります。このアプローチを図 3 の左に示します。

2. 部分行列のサイズが十分小さくなるまで、行列を再帰的に横または縦に分割する、再帰分割統治 (キャッシュを意識しない) 方法。分割後、最小部分行列 (タイル) が個別に転置されます。再帰手法では、ワーク量の不足は見られず、並列処理の粒度は 1 つのタイルであるため (図 3 の右を参照)、並列ワークアイテム (タイル) の数は (n*n)/(2*TILE) とほぼ等しくなります。TILE=32 で 700 × 700 よりも大きな行列の場合、少なくとも 240 のワークアイテムがあるため、コプロセッサーの 240 の論理コアにわたってワークロードをバランスよく分配できます。しかし、大量の並列タスクをスケジュールする必要があるため、並列スケジュールのオーバーヘッドが非常に大きくなります。

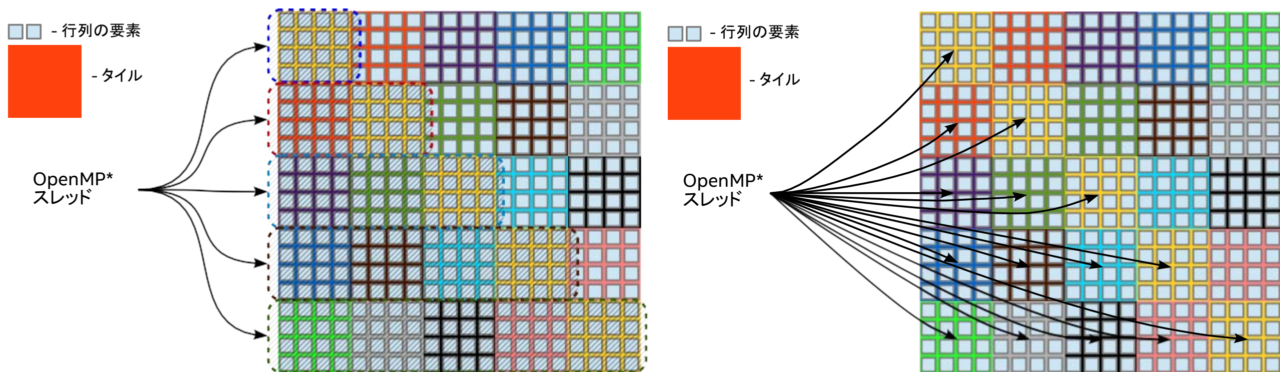

図 3: 左: 入れ子のループ [4] を使用した行列走査。タイルの行がスレッドに分配される (不十分な並列処理、ロード・インバランス)。右: タイルサイズの粒度の並列処理を使用した行列走査 (より多くの並列処理、より優れたロードバランス)。

ここで説明する新しい実装では、入れ子のループと再帰タイル走査アルゴリズムを再度実装します。ただし、[4] で発生したスケジュール問題を緩和するため、新しい実装では次の対策を行います。

a) タイルサイズの粒度で入れ子のループ・アルゴリズムを並列化します。

b) 転置を実行する前にタイル走査の順序を生成する計画ルーチンを使用して、並列化のオーバーヘッドを少なくします。

転置手法を呼び出す前に、指定された行列サイズで計画ルーチンを呼び出す必要があります。このルーチンは、行列のすべての本体タイル (図 2 の青い領域) の位置を計算して、要求される順にあらかじめ割り当てられている配列に書き込みます。転置ルーチンは、この配列からタイルの位置を読み取ります (例えば、リスト 1 の 5 行目と 6 行目を参照)。計画ルーチンの例である、入れ子のループ走査アルゴリズムをリスト 5 に示します。再帰アルゴリズムの計画ルーチンは非常に長いため、ここでは掲載していません。ベンチマークのコードは、Colfax Research の Web サイトから入手できます。

```
1   void NestedLoopTranspositionPlan(int* const plan, const int n) {
2
3     // Number of complete tiles in each dimension
4     const int wTiles = n / TILE;
5     int i = 0;
6
7     // Tiled plan
8     for (int j = 1; j < wTiles; j++)
9     for (int k = 0; k < j; k++) {
10        plan[2*i + 0] = j*TILE; // Value
          of                          ii
11        plan[2*i + 1] = k*TILE; // Value
          of                          jj
12        i++;
13    }
14
```

リスト 5: 入れ子のタイル走査アルゴリズムの計画ルーチン。主対角線上や行列の端でないタイルのみ走査される。

図 4 は、入れ子のタイル走査アルゴリズムと再帰タイル走査アルゴリズムです。入れ子のループ・アルゴリズム (左) では、アルゴリズムは主行列本体の左上隅から開始して、左から主対角線の方向にタイルの各行を走査します。選択されたタイルは、主対角線上で対称的に位置するタイルと転置されます。再帰アルゴリズムも左上隅から開始しますが、より複雑な (前のタイルの横方向に加えて縦方向にも近いタイルを選択する) 方法でタイルを走査します。

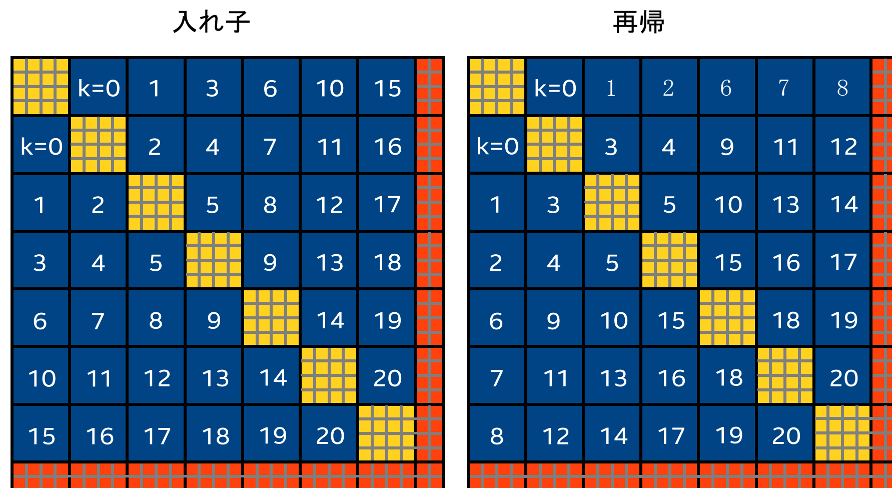

図 4: 左: 入れ子のタイル走査アルゴリズムを使用した行列転置。右: 再帰分割統治アルゴリズムを使用した行列転置。タイルの番号は、タイルが走査され転置マイクロカーネル (リスト 4 の上) が適用される順序を示している。

セクション 2.2 で示したように、どちらのアルゴリズムも、主対角線上や行列の端でないタイルのみ走査し、前のタイルに近いタイルを選択することでデータの局所性を維持しようとしています。[3] によれば、再帰アルゴリズムの漸近的キャッシュヒット率が最も優れています。しかし、インテル® Xeon Phi™ コプロセッサーで実際にテストした多くのケースでは、入れ子のアルゴリズムのほうが優れていました (セクション 3 を参照)。

2.4. ギャロップで駆ける: プリフェッチと一時的でないストア

タイル走査アルゴリズムを選択して、転置マイクロカーネルを最適化したら、コンパイラー引数と環境変数で残りの最適化を行います。このセクションでは、メモリーとのデータの読み書きの効率を向上させる手法について説明します。

読み取りパフォーマンスの最適化には、プリフェッチを使用します。プリフェッチは、プロセッサーおよびインテル® MIC アーキテクチャーで利用可能なキャッシュの機能です。データがコアで使用される前に RAM からキャッシュへのキャッシュラインの移動を要求することにより、メモリーアクセスのレイテンシーを隠蔽します。プリフェッチ命令は、専用のハードウェア・ユニットにより (ハードウェア・プリフェッチ)、またはアプリケーション自身により (ソフトウェア・プリフェッチ) 発行できます。インテル® Xeon® プロセッサーには L1 キャッシュ・ハードウェア・プリフェッチャーと L2 キャッシュ・ハードウェア・プリフェッチャーがありますが、インテル® Xeon Phi™ コプロセッサーには L2 キャッシュ・ハードウェア・プリフェッチャーしかありません。このため、インテル® Xeon Phi™ アプリケーションでは、ソフトウェア・プリフェッチ命令がパフォーマンス向上の鍵です。プリフェッチ命令は通常、コンパイラーにより自動的に実行コードに挿入されます。しかし、コンパイラーが選択したプリフェッチの距離 (つまり、どのくらい前のベクトルループの反復でプリフェッチ命令を発行するか) を無視することもできます。コンパイラー引数 “`-opt-prefetch-distance=8`” を使用すると、パフォーマンスがさらに 1 ～ 2% 向上します。

書き込みパフォーマンスの最適化には、転置ルーチンで一時的でない (Non-temporal) ストアを使用します。一時的でないストアでは、書き込まれたデータはキャッシュをバイパスして RAM にフラッシュされます。そのため、不要なデータがキャッシュに混入されるのを防ぎ、読み取り操作でより多くのキャッシュを利用できます。転置操作では、転置したタイルを書き込んだ後、そのデータを再利用しないため、一時的でないストアが適しています。この書き込み操作は、コンパイラー引数 “`-opt-streaming-stores always`” でリクエストできます。あるいは、ループの前に “`#pragma vector nontemporal`” を配置して、そのループに一時的でないストアを実装することもできます。現在の実装では、前者の手法 (コンパイラー引数) が使われています。一時的でないストアにより、転置パフォーマンスは約 2% 向上します。

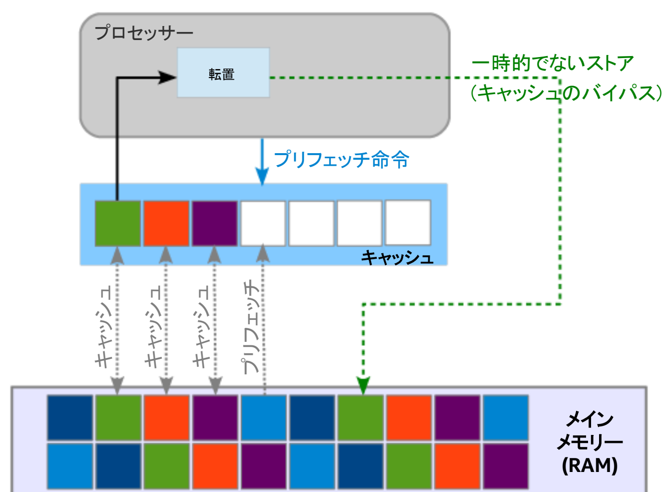

図 5: ソフトウェア・プリフェッチ: プロセッサーは、あらかじめメインメモリーからデータを取得しておくようにキャッシュにリクエストする。プリフェッチの距離はコンパイラーのヒューリスティックに基づいて予測されるが、コンパイラー引数 "-opt-prefetch-distance=n" を指定して無視することができる。一時的でないストア: プロセッサーは、キャッシュをバイパスしてメインメモリーにデータを直接書き込む。コンパイラー引数 "-opt-streaming-stores always" を指定すると、コンパイルしたオブジェクト・ファイルで一時的でない書き込み操作が行われる。

2.5. 荒れ馬を馴らす: スレッド・アフィニティー

デフォルトでは、OpenMP* スレッドは、アプリケーションの実行中に、ある論理/物理コアから別のコアに移動することができます。スレッド (並列コードのインスタンス) が物理コアから別のコアに移動すると、そのコアのローカルキャッシュに含まれるデータへのアクセスが失われるため、パフォーマンスの低下につながります。OpenMP* スレッドを論理コアまたは物理コアにバインドすることにより、アプリケーションのパフォーマンスを向上して、パフォーマンスの変動を抑えられます。OpenMP* スレッドの論理コアや物理コアへのマッピングを**スレッド・アフィニティー**と呼びます。

現在の実装では、環境変数 "KMP_AFFINITY=compact" を使用して、実行時にスレッド・アフィニティーを設定しています。このアフィニティー型は、できるだけ近くに連続する OpenMP* スレッドを配置します。現在の物理コアがフルでない場合、スレッドはその物理コアに配置され、フルの場合は次の物理コアに配置されます。マルチプロセッサーでは、1 つのプロセッサー・ソケットがフルになった場合にのみ、次のプロセッサーにスレッドが配置されます。デフォルトでは (アフィニティー・マスクで引数 "granularity" が指定されない限り)、OpenMP* アフィニティー "compact" の粒度は 1 つの物理コアです。つまり、スレッドを 1 つの物理コアの論理コア間で移動できます。この設定では、行列転置アプリケーションのパフォーマンスへの影響はありません。

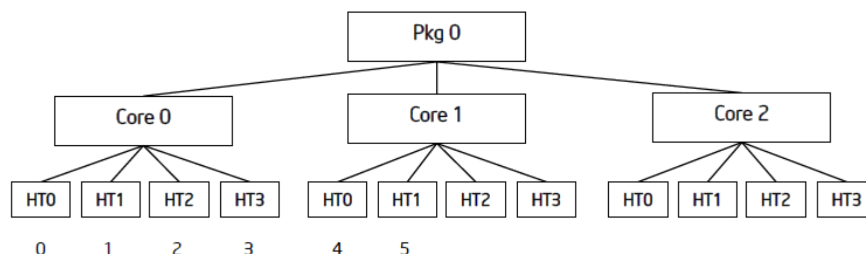

図 6: OpenMP* スレッド・アフィニティー "compact" は、できるだけ近くに連続する OpenMP* スレッドを配置して複数の物理コアにわたるスレッド移動を防ぐ。イメージ出典: インテル コーポレーション

2.6. 良い行列サイズ、悪い行列サイズ、最悪の行列サイズ

最高のパフォーマンスは、行列の行の長さがキャッシュラインのサイズ (インテル® Xeon® アーキテクチャーとインテル® Xeon Phi™ アーキテクチャーでは 64 バイト) の倍数のときに得られます。そのため、行列サイズ n を 64/4=16 (単精度) または 64/8=8 (倍精度) にします。この条件を満たす行列サイズは、「最悪」の条件 (以下を参照) を同時に満たさない限り、「良い」サイズと呼びます。「良い」行列サイズの例は、n=960、8000、22000 です。

「悪い」行列サイズとは、行列の行の長さがキャッシュラインの長さの倍数でないものです。この場合、行列の行の最初の要素は常にキャッシュラインの先頭にマップされるとは限りません。そのため、転置マイクロカーネルがキャッシュラインと同じサイズの要素ブロックを読み取るときに、(1 つではなく) 2 つのキャッシュラインからデータをロードしなければならないことがあります。さらに、キャッシュラインが 2 つのタイル間で共有されており、2 つのスレッドがそのキャッシュラインの内容を同時に変更すると、フォルス・シェアリングが発生します。「悪い」行列サイズの例は、n=962、8051、22004 です。

最後に、「最悪」の行列サイズとは、行列の行の長さ (バイト数) がキャッシュセットの競合サイズの倍数、またはそれに近い場合です。キャッシュセットの競合は、指定されたメモリーアドレスをキャッシュの小さな領域 (セット) にのみマップできる、連想キャッシュで発生します。メモリーアドレスと許容キャッシュセット間のマッピングは、キャッシュセットの競合サイズで周期的に繰り返されます。インテル® Xeon Phi™ アーキテクチャーの場合、キャッシュセットの競合サイズは 4 キロバイトです。これは、メモリーで 4 キ

© Colfax International, 2013 — http://research.colfaxinternational.com/

ロバイト間隔のデータ要素が同じキャッシュセットにマップされることを意味します。アプリケーションが 4 キロバイトの倍数のストライドでメモリーのデータを走査すると、キャッシュの一部のみが使用され、パフォーマンスが低下します。「最悪」の行列サイズの例は、n=1024、8192、21504 です。

このホワイトペーパーで説明した実装は、任意の行列サイズで正しい結果を出力します。しかし、パフォーマンスは「良い」サイズの場合に最高になります。転置を行う実際のアプリケーションで行列サイズを選択できる場合は、「良い」行列サイズを選択してください。必要に応じて、未使用の行や列を行列にパディングして「良い」行列サイズにすることができます。

3. ベンチマーク

3.1. 転置レート

このホワイトペーパーでは、転置のパフォーマンスを帯域幅 (GB/秒) で表示しています。転置レートは、行列サイズ (GB) を転置時間 (秒) で割った結果に 2 を掛けて計算します。

$$\text{転置レート} = \frac{N \times N \times \text{sizeof(TYPE)}}{2^{30} \times \text{時間}} \text{x2} \qquad (2)$$

ここで、N は行列サイズです。これは転置パフォーマンスを求める従来の方法です。係数 2 は、行列を転置するにはデータを読み取って書き込まなければならず、1 つの行列要素あたり 2 回のメモリーアクセスが行われるためです。サイズと時間の比率に 2 を掛けることで、同様にサイズと時間の比率に係数 2 を掛ける STREAM ベンチマーク [10] ("copy" テスト) と結果を直接比較することもできます。具体的に言うと、方程式 (2) の理論的な最大転置レートは STREAM "copy" テストの帯域幅と等しくなります。

現在の結果を以前の結果 [4] と比較する場合、今回の結果に含まれている係数 2 が以前の結果には含まれていないことに注意してください。

3.2. ハードウェア・システム構成

このホワイトペーパーのテストはすべて、CX2265i-XP5 サーバー[4] で実行しました。

1) ホストシステム: 2 x インテル® Xeon® プロセッサー E5-2680 (8 コア、2 ウェイ・ハイパースレッディング有効)、64GB DDR3 RAM 1,333MHz、Cent OS 6.4、コードのコンパイルにはインテル® C++ コンパイラー 13.1.3.192 (Build 20130607) を使用。

2) コプロセッサー: 1 x インテル® Xeon Phi™ コプロセッサー SKU B1QS-7110P (61 コア)、16GB GDDR5 RAM、インテル® MPSS (インテル® Xeon Phi™ コプロセッサー用ドライバー) 2.1.6720-13 を実行。

コプロセッサーとホスト・プロセッサーには、RAM のデータ (ビット) の誤りをランタイムに保護する ECC (誤り訂正符号) 機能があり、このホワイトペーパーのテストはすべて、この機能を有効にして行われました。ECC 機能を無効にすると、帯域幅に制限されるベンチマークのパフォーマンスが向上することは良く知られています。しかし、実際のアプリケーションでは ECC モードが一般に使用されているため、ここでは ECC 機能を有効にしました。

3.3. STREAM ベンチマーク

転置パフォーマンスを測定するため、STREAM ベンチマーク [10] をホスト・プロセッサーとコプロセッサーで実行しました。STREAM "copy" テストの結果は、行列転置のパフォーマンスの上限を提供します。STREAM ベンチマークは、[11] で推奨されているように、次のコマンドを使用してホストおよびコプロセッサー用にコンパイルしました。

```
andrey@dublin$ # Compile CPU version:
andrey@dublin$ icpc -O3 -openmp -DSTREAM_ARRAY_SIZE=64000000 -o stream
stream.c
andrey@dublin$ #
andrey@dublin$ # Compile MIC version:
andrey@dublin$ icpc -O3 -openmp -DSTREAM_ARRAY_SIZE=64000000 \
               -mmic -opt-prefetch-distance=64,8 -opt-streaming-cache-evict=0 -ffreestanding \
               -o stream-MIC stream.c
```

リスト 6: STREAM ベンチマークのコンパイル

STREAM ベンチマークは、ホストでは通常どおり、コプロセッサーではネイティブ実行モードで実行しました。ネイティブ実行では、実行コードとライブラリーをコプロセッサーの仮想ファイルシステムにコピーした後、SSH セッションによりデバイスで直接アプリケーションを開始します。ホスト・プロセッサーはネイティブ・アプリケーションの実行に関与しません。転置ベンチマークのネイティブ実行については、セクション 3.4 も参照してください。

[4] http://www.colfax-intl.com/xeonphi/CX2265i-XP5.html

最適な結果を達成するため、環境変数 OMP_NUM_THREADS を使って、ホストのスレッド数を 16、コプロセッサーのスレッド数を 60 に設定しました。この設定により、各デバイスの物理コアあたり 1 つのスレッドが使用されます [5]。さらに、環境変数 KMP_AFFINITY=scatter を設定して、OpenMP* スレッド・アフィニティー型を "scatter" にしました。

表 1 に STREAM ベンチマークの結果を示します。

Test	Two Intel Xeon E5-2680	Intel Xeon Phi 7110P
KMP_AFFINITY	scatter	scatter
OMP_NUM_THREADS	16	60
ECC	on	on
Copy	60.4 GB/s	169.2 GB/s
Scale	66.0 GB/s	166.5 GB/s
Add	65.8 GB/s	174.3 GB/s
Triad	66.3 GB/s	174.1 GB/s
Theoretical Peak Bandwidth	102.4 GB/s	352 GB/s

表 1: STREAM ベンチマークの結果

2 つのインテル® Xeon® プロセッサー E5-2860 を搭載したホストシステムの理論的なピーク帯域幅は 102.4GB/秒 [12] なので、STREAM ベンチマークの効率は 60 ～ 65% です。インテル® Xeon Phi™ コプロセッサー 7110P の理論的なピーク帯域幅は 352GB/秒 [13] なので、STREAM ベンチマークの効率は 48 ～ 50% です。

3.4. コンパイルと実行

リスト 7 は、行列転置アプリケーションの倍精度バージョンをコンパイルしたコマンドラインです。コンパイラー・オプションを含む、これらのコマンドについては、セクション 2.4 で説明しています。

```
andrey@dublin$ # CPU version:
andrey@dublin$ icpc -c Transpose.cc -o Transpose-dp-CPU.o \
               -DDOUBLE -O3 -openmp -opt-prefetch-distance=8
andrey@dublin$ icpc -c Main.cc -o Main-dp-COU.o -DDOUBLE -O3 -openmp
andrey@dublin$ icpc -o runme-dp-CPU Main-dp-CPU.o Transpose-dp-CPU.o -O3 -openmp
andrey@dublin$ #
andrey@dublin$ # MIC version:
andrey@dublin$ icpc -c Transpose.cc -o Transpose-dp-MIC.o \
               -DDOUBLE -O3 -openmp -mmic -opt-prefetch-distance=8 -opt-streaming-stores always
andrey@dublin$ icpc -c Main.cc -o Main-dp-MIC.o -DDOUBLE -O3 -openmp -mmic
andrey@dublin$ icpc -o runme-dp-MIC Main-dp-MIC.o Transpose-dp-MIC.o -O3 -openmp -mmic
```

リスト 7: 転置ベンチマークの倍精度バージョンをコンパイルしたプロセッサー (実行ファイル runme-dp) とコプロセッサー (実行ファイル runme-dp-MIC) 用のコマンドライン

runme-dp はインテル® Xeon® アーキテクチャー用の実行ファイルで、runme-dp-MIC はインテル® Xeon Phi™ コプロセッサーのネイティブ実行用の実行ファイルです。コード (リスト 8 を参照) の実行には、セクション 2.5 で説明したように、環境変数 KMP_AFFINITY=compact を使用しました。OMP_NUM_THREADS の値として、ホストで 32 (16 コア、2 ウェイ・ハイパースレッディング)、コプロセッサーで 244 (61 コア、4 ウェイ・ハイパースレッディング) を指定しました。ネイティブ実行では、scp コマンドで実行ファイルとライブラリーをコプロセッサーの仮想ファイルシステムに転送してから、ssh クライアントでシェルを起動し、環境変数を設定して、インテル® Xeon Phi™ コプロセッサーで直接アプリケーションを起動しました。

[5] インテル® Xeon Phi™ コプロセッサー 7110P には 61 のコアが含まれていますが、STREAM は 60 スレッドで実行しています。残りのコアは、オフロード・アプリケーションでオフロードタスクを管理するのに利用できます。

```
andrey@dublin$ # Benchmark on the CPU-based host
andrey@dublin$ export KMP_AFFINITY=compact
andrey@dublin$ export OMP_NUM_THREADS=32
andrey@dublin$ ./runme-dp-CPU
...
andrey@dublin$ #
andrey@dublin$ # Benchmark on the MIC-based coprocessor
andrey@dublin$ # Copy required libraries to the coprocessor:
andrey@dublin$ scp /opt/intel/composer_xe_2013.5.192/compiler/lib/mic/libiomp5.so mic0:~/
andrey@dublin$ # Copy the code to the coprocessor:
andrey@dublin$ scp runme-dp-MIC runme-sp-MIC mic0:~/
andrey@dublin$ # Log in to the coprocessor to run the
code:andrey@dublin$ ssh mic0
andrey@dublin-mic0$ export KMP_AFFINITY=compact
andrey@dublin-mic0$ export OMP_NUM_THREADS=244
andrey@dublin-mic0$ export LD_LIBRARY_PATH=.
andrey@dublin-mic0$ ./runme-dp-MIC
...
andrey@dublin-mic0$ exit
andrey@dublin$ #
```

リスト 8: ホストとコプロセッサーでネイティブ実行モードにより転置ベンチマークを実行

ベンチマークの出力を収集し、解析した結果をセクション 3.5 および 3.6 の図と表に示します。STREAM ベンチマークと同様に、すべてのテストでホストとコプロセッサーの ECC 機能を有効にしました。

3.5. 結果: 良い行列サイズ

「良い」行列サイズ (定義はセクション 2.6 を参照) の転置ベンチマークを図 7 に示します。ホスト・プロセッサーの結果で、30MB 以下のサイズの行列は、インテル® Xeon® プロセッサー E5-2680 の L3 キャッシュに収まることに注意してください。転置の開始時に行列がプロセッサーのキャッシュに収まっていない現実的な状況をシミュレーションするため、このベンチマークでは、ベンチマーク・テスト間で大きなダミー配列を使って読み取り/書き込み操作を行い、キャッシュの内容を排除しています。

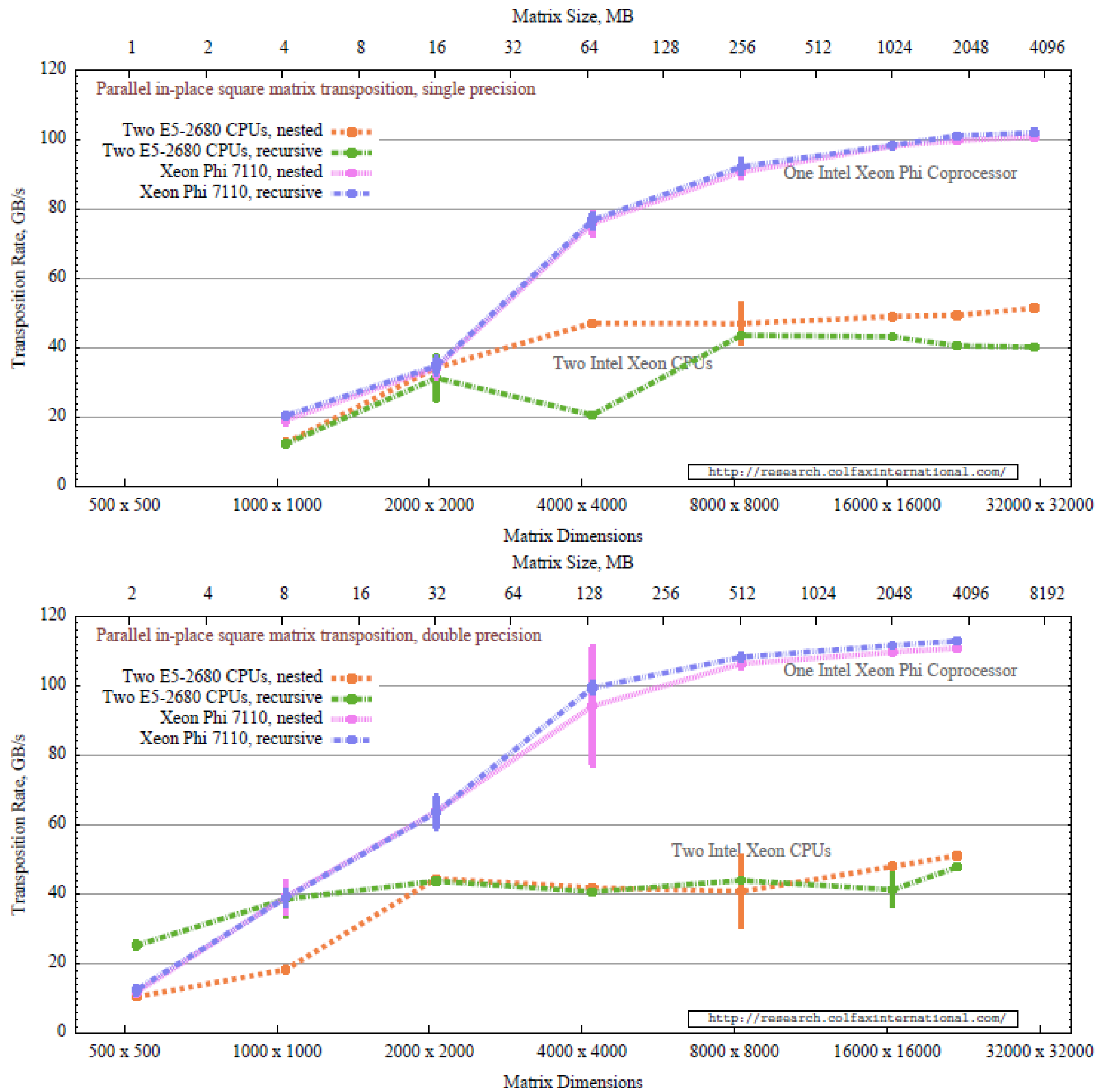

図 7:「良い」行列サイズの並列インプレース正方行列転置レート。誤差バーは、試行回数 20 回の平均二乗偏差。詳細はテキストを参照

3.6. 結果: 良い行列サイズと悪い/最悪の行列サイズ

		Performance in Single Precision					
		Host			Coprocessor		
#		"Good"	"Bad"	"Ugly"	"Good"	"Bad"	"Ugly"
	Size	1040×1040	1030×1030	1024×1024	1040×1040	1030×1030	1024×1024
1	Nested	42.3 ± 1.9	36.4 ± 2.8	29.8 ± 1.2	23.2 ± 3.2	21.2 ± 0.2	18.0 ± 0.4
	Recursive	44.0 ± 1.0	33.0 ± 1.2	35.1 ± 0.9	21.9 ± 0.6	18.6 ± 2.9	18.3 ± 0.2
	Size	2064×2064	2060×2060	2048×2048	2064×2064	2060×2060	2048×2048
2	Nested	55.8 ± 2.6	33.1 ± 8.4	22.0 ± 6.5	37.5 ± 0.6	32.5 ± 0.5	26.7 ± 0.6
	Recursive	55.5 ± 2.7	31.7 ± 8.5	38.1 ± 0.5	36.0 ± 3.0	28.4 ± 0.2	22.9 ± 1.4
	Size	4160×4160	4100×4100	4096×4096	4160×4160	4100×4100	4096×4096
3	Nested	22.2 ± 0.2	21.7 ± 0.3	15.1 ± 0.3	77.9 ± 5.2	60.2 ± 1.6	20.3 ± 1.3
	Recursive	26.6 ± 4.5	24.6 ± 2.3	14.9 ± 0.0	77.6 ± 2.1	57.5 ± 0.9	20.4 ± 2.7
	Size	8240×8240	8210×8210	8192×8192	8240×8240	8210×8210	8192×8192
4	Nested	33.4 ± 0.7	33.7 ± 0.3	27.2 ± 0.8	93.1 ± 1.1	71.4 ± 1.2	8.1 ± 0.0
	Recursive	36.1 ± 0.6	46.1 ± 0.3	28.2 ± 0.7	90.8 ± 0.7	69.0 ± 0.7	7.5 ± 0.0
	Size	16400×16400	16390×16390	16384×16384	16400×16400	16390×16390	16384×16384
5	Nested	44.9 ± 0.2	37.3 ± 0.2	29.1 ± 0.1	98.0 ± 0.2	71.2 ± 0.3	4.9 ± 0.0
	Recursive	50.4 ± 0.4	48.6 ± 0.2	30.2 ± 0.0	98.1 ± 0.1	68.3 ± 0.2	4.8 ± 0.0
	Size	22000×22000	22004×22004	21504×21504	22000×22000	22004×22004	21504×21504
6	Nested	41.9 ± 0.2	36.6 ± 0.1	36.0 ± 0.1	100.9 ± 0.3	78.9 ± 0.1	47.2 ± 0.1
	Recursive	52.7 ± 0.2	50.8 ± 0.2	40.2 ± 0.0	99.5 ± 0.3	79.5 ± 0.2	46.5 ± 0.1
	Size	31200×31200	31100×31100	30720×30720	31200×31200	31100×31100	30720×30720
7	Nested	40.5 ± 0.2	34.7 ± 0.1	35.1 ± 0.1	102.0 ± 0.1	78.5 ± 0.0	46.5 ± 0.0
	Recursive	51.4 ± 0.2	48.7 ± 0.1	38.9 ± 0.1	100.8 ± 0.1	79.3 ± 0.1	45.9 ± 0.0

		Performance in Double Precision					
		Host			Coprocessor		
#		"Good"	"Bad"	"Ugly"	"Good"	"Bad"	"Ugly"
	Size	528×528	524×524	512×512	528×528	524×524	512×512
1	Nested	21.3 ± 1.4	13.2 ± 0.6	10.9 ± 0.5	14.7 ± 0.2	16.8 ± 0.4	14.5 ± 0.4
	Recursive	18.7 ± 6.9	18.0 ± 0.6	12.2 ± 0.5	14.1 ± 0.3	15.2 ± 5.4	16.1 ± 0.4
	Size	1040×1040	1030×1030	1024×1024	1040×1040	1030×1030	1024×1024
2	Nested	26.4 ± 1.5	24.7 ± 1.1	20.8 ± 0.6	40.6 ± 0.6	38.0 ± 4.0	32.3 ± 3.3
	Recursive	26.8 ± 1.5	23.8 ± 1.0	21.5 ± 0.6	38.4 ± 0.6	39.2 ± 0.9	27.8 ± 3.6
	Size	2064×2064	2060×2060	2048×2048	2064×2064	2060×2060	2048×2048
3	Nested	40.1 ± 1.5	80.7 ± 5.0	51.5 ± 1.8	66.4 ± 1.8	60.5 ± 1.1	39.3 ± 3.0
	Recursive	39.6 ± 1.1	62.0 ± 13.8	56.4 ± 1.5	65.7 ± 2.0	58.5 ± 1.5	34.3 ± 0.8
	Size	4160×4160	4100×4100	4096×4096	4160×4160	4100×4100	4096×4096
4	Nested	42.3 ± 0.1	24.7 ± 0.1	39.3 ± 0.2	98.5 ± 2.3	87.9 ± 2.2	16.6 ± 0.3
	Recursive	44.4 ± 0.1	49.3 ± 0.5	29.1 ± 9.2	97.3 ± 1.7	85.0 ± 1.0	15.2 ± 0.1
	Size	8240×8240	8210×8210	8192×8192	8240×8240	8210×8210	8192×8192
5	Nested	44.4 ± 0.2	45.9 ± 0.6	36.5 ± 2.3	108.0 ± 1.1	92.3 ± 0.4	16.0 ± 0.0
	Recursive	48.1 ± 0.1	49.4 ± 0.7	29.3 ± 7.8	106.5 ± 0.2	90.4 ± 0.1	15.5 ± 0.0
	Size	16400×16400	16390×16390	16384×16384	16400×16400	16390×16390	16384×16384
6	Nested	45.0 ± 0.2	41.9 ± 0.1	33.1 ± 0.2	111.5 ± 0.1	92.8 ± 0.2	16.2 ± 0.0
	Recursive	48.4 ± 0.2	47.0 ± 0.1	34.9 ± 0.1	109.6 ± 0.1	90.1 ± 0.2	16.1 ± 0.0
	Size	22000×22000	22004×22004	21504×21504	22000×22000	22004×22004	21504×21504
7	Nested	47.5 ± 0.4	46.8 ± 0.2	43.9 ± 0.3	112.9 ± 0.1	100.4 ± 0.1	83.0 ± 0.1
	Recursive	49.3 ± 0.2	50.3 ± 0.6	47.5 ± 0.2	110.8 ± 0.1	99.6 ± 0.1	81.7 ± 0.0

表 2: パフォーマンス結果。上の表は単精度、下の表は倍精度。各セルの 1 行目は行列サイズ、2 行目と 3 行目は転置レートと試行回数 20 回の平均二乗偏差 (GB/秒)。詳細はセクション 2.6 を参照

3.7. インテル® MKL との比較

インテル® マス・カーネル・ライブラリー (インテル® MKL) [7] の mkl_?imatcopy 関数は、このホワイトペーパーで実装したインプレース行列転置と同じ操作を行います [14]。mkl_?imatcopy には、ここで説明したルーチンよりも多くの機能があり、非正方行列を転置し、転置と同時にデータのスケーリングを実行できます。さらに、2 の累乗に比例する行列サイズ用に高度にチューニングされています。インテル® MKL 11.0.5 の mkl_?imatcopy の実装は、スレッド数を行列サイズで調整したときに優れた結果を達成します。*n* はスレッドの倍数とキャッシュラインの要素数を掛けた数です。

図 8 は、ここで説明した転置ルーチンと mkl_dimatcopy をインテル® Xeon Phi™ コプロセッサーで実行した比較結果です。この比較では、2 セットの行列サイズを使用しました。

1. 左のグラフでは、転置ルーチンとインテル® MKL 関数の両方に最適なサイズとして、1952、3904、5856、7808、9760、13664、17568、21472 を使用しています。これらは「良い」サイズであり、スレッド数 (244) とキャッシュラインの要素数 (倍精度では 8) を掛けた数の倍数です。

2. 右のグラフでは、インテル® MKL 関数では最適ですが、転置ルーチンでは最適でないサイズとして、2048、3072、4096、6144、8192、12288、16384、21504 を使用しています。これらは「最悪」のサイズですが、2 の累乗または 2 の累乗の和です。ここでは 128 スレッドで実行しました。

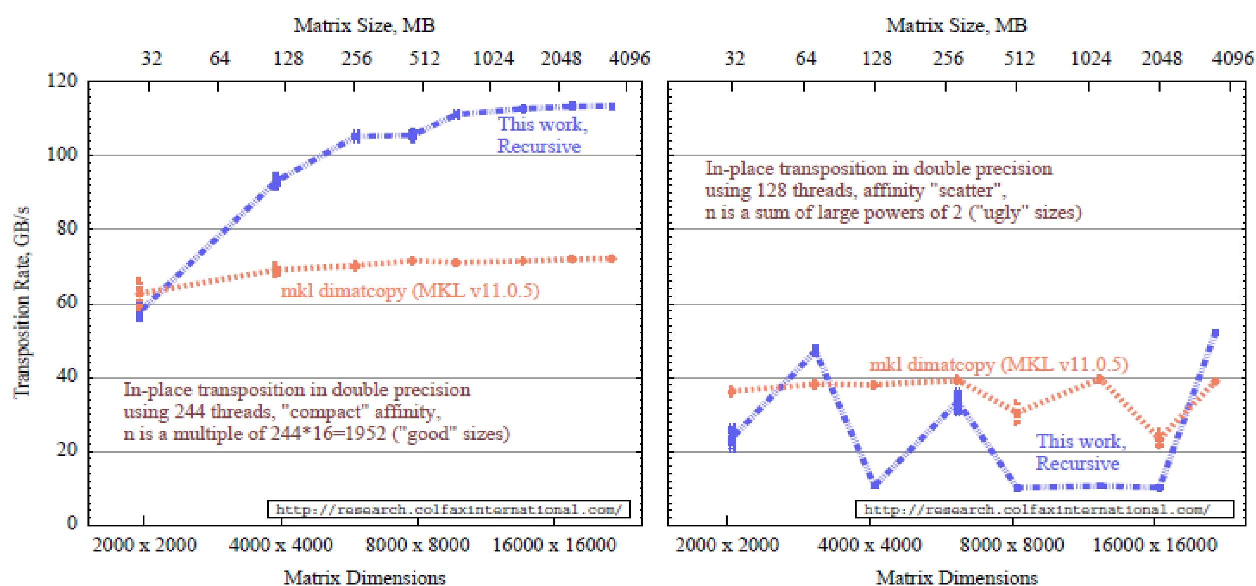

図 8: 現在の実装とインテル® MKL 関数のパフォーマンスの比較

図 8 では、行列サイズが「良い」場合は、ここで説明した実装のほうがインテル® MKL 関数よりも転置レートが高くなっています。しかし、*n* が 2 の累乗と等しい場合は、インテル® MKL 関数のほうが優れた結果を達成しています。

実際のアプリケーションにおける転置手法の選択は、アプリケーションの要件に依存します。アプリケーションで非正方行列転置を行う場合、あるいは転置行列のサイズが 2 の累乗で、スレッド数を行列サイズで調整できる場合、mkl_?imatcopy が最適です。しかし、行列が正方行列で、そのサイズを「良い」値にパディングできる場合、あるいは転置ルーチンのスレッド数を行列サイズで調整できない場合、ここで説明したルーチンのほうがより高速に実行できます。

これらのコードをアプリケーションで使用する場合は、インテル® MKL の将来のバージョンで実装が変更される可能性があるため、最新バージョンをダウンロードしてテストすることを推奨します。

4. まとめ

セクション 2.1 ～ 2.6 のカウボーイをテーマとしたタイトルは、このホワイトペーパーが行列転置操作で達成できる速度に注目していることを表しています。倍精度 (ECC 有効) の入れ子/再帰アルゴリズムの最高転置レートは、2 つのホスト・プロセッサーで 49GB/秒、コプロセッサーで 113GB/秒に達しています。これらの数値は、それぞれ、STREAM "copy" 帯域幅によって決まる転置パフォーマンスの上限の 82% と 67% に相当します。この操作では、1 つのインテル® Xeon Phi™ コプロセッサー 7110P のほうが、2 つのインテル® Xeon® プロセッサー E5-2680 よりも 2.3 倍高速に実行されています。

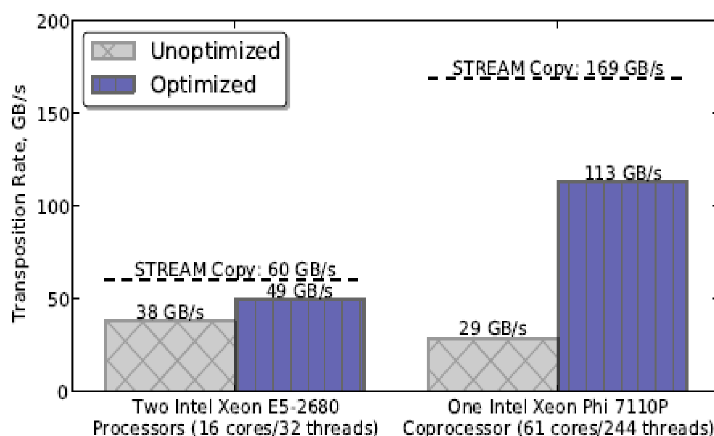

図 9: 最適化されていない手法 (リスト 1) と最適化された手法 (リスト 9) の入れ子/再帰アルゴリズムの最高転置レート。行列サイズ 22000×22000、倍精度、STREAM "copy" テストの帯域幅 (転置レートの上限)、ECC 有効。同じ C 言語コードからプロセッサーとインテル® MIC プラットフォーム用の実行ファイルを生成。

転置操作を高速化する鍵はキャッシュ・トラフィックの最適化です。最適化されていないコード (リスト 1) の転置レートは、ホストで 38GB/秒、コプロセッサーで 29GB/秒です (`KMP_AFFINITY=scatter`、`OMP_NUM_THREADS=16` および `240` を指定)。このホワイトペーパーで述べたように、時間のデータ局所性を向上するループタイリングを使用することで、効率良くインテル® MIC アーキテクチャーのパフォーマンスを引き出せます。また、転置を実行する前にタイル走査パターンを調整し、スケジュール・オーバーヘッドが低い、細粒度のスレッド並列処理を行う最適化も重要です。その他の最適化 (データのアライメント、`#pragma simd` の使用、コンパイラー引数のチューニング) も役立ちます。

コードを最適化すると、一般に、インテル® Xeon® プロセッサー・ベースのシステムよりもインテル® Xeon Phi™ コプロセッサーのパフォーマンスがより大きく向上します。ユニファイド L2 キャッシュや L1 キャッシュのハードウェア・プリフェッチャーなど、プロセッサーには最適でないコードに対応するためのリソースがあるため、プロセッサーはコプロセッサーよりも寛容です。インテル® Xeon Phi™ コプロセッサーが効率良くキャッシュを使用し、利用可能な並列処理をすべて行えるようにコードを最適化すると、コプロセッサーのパフォーマンスは大幅に向上します。

このホワイトペーパーでは、インテル® Xeon Phi™ コプロセッサーが実際に達成可能なパフォーマンスに加えて、このアーキテクチャーのプログラミングに関する重要な特性も示しています。インテル® MIC アーキテクチャー向けアプリケーションは、プロセッサーのベンチマークに使用した C 言語コードを再コンパイルするだけで生成できます。低水準コードは使用されず、最も複雑なタイル転置マイクロカーネルの実装はコンパイラーによって行われます。

ここで示したアプリケーションの効率は、プロセッサーで 82%、コプロセッサーで 67% です。そのため、追加の最適化による理論的な向上値は、ホストで 1.2 倍、コプロセッサーで 1.5 倍になります。ベンチマークのソースコードは http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx からダウンロードできます。

謝辞

インテル® MKL の機能について詳細な情報を提供してくださったインテル コーポレーションの Evarist Fomenko 氏に感謝します。

A. ソースコード

```
1  void Transpose(FTYPE* const A, const int n, const int* const plan ) {// FTYPE is float or double
2     const int TILE = 32;                                    // Tile size
3     const int nEven = n - n%TILE;                           // nEven is a multiple of TILE
4     const int wTiles = nEven / TILE;                        // Complete tiles in each dimens.
5     const int nTilesParallel = wTiles*(wTiles - 1)/2;       // # of complete tiles under the main diag.
6  #pragma omp parallel                                       // Start of parallel region
7     {
8  #pragma omp for schedule(guided)
9        for (int k = 0; k < nTilesParallel; k++) {           // Parallel loop over body tiles
10         const int ii = plan[2*k + 0];                      // Top column of the tile (planned)
11         const int jj = plan[2*k + 1];                      // Left row of the tile (planned)
12
13         for (int j = jj; j < jj+TILE; j++)                 // Tile transposition microkernel:
14 #pragma simd                                               // Ensure automatic vectorization
15           for (int i = ii; i < ii+TILE; i++) {
16             const FTYPE c = A[i*n + j];                    //
17             A[i*n + j] = A[j*n + i];                       // Swap matrix elements
18             A[j*n + i] = c;                                //
19           }
20     }                                                      // End of main parallel for-loop
21
22 #pragma omp for schedule(static)
23       for (int ii = 0; ii < nEven; ii += TILE) {           // Transposing tiles on the main diagonal:
24         const int ii = jj;
25         for (int j = jj; j < jj+TILE; j++)                 // Diagonal tile transposition microkernel:
26 #pragma simd                                               // Ensure automatic vectorization
27           for (int i = ii; i < j; i++) {                   // Avoid duplicate swaps
28             const FTYPE c = A[i*n + j];                    //
29             A[i*n + j] = A[j*n + i];                       // Swap matrix elements
30             A[j*n + i] = c;                                //
31           }
32       }
33
34 #pragma omp for schedule(static)
35       for (int j = 0; j < nEven; j++)                      // Transposing the "peel":
36         for (int i = nEven; i < n; i++) {
37          const FTYPE c = A[i*n + j];                       //
38          A[i*n + j] = A[j*n + i];                          // Swap matrix elements
39          A[j*n + i] = c;                                   //
40         }
41     }                                                      // End of thread-parallel region
42
43     for (int j = nEven; j < n; j++)                        // Transposing bottom-right cornr
44       for (int i = nEven; i < j; i++) {                    //
45         const FTYPE c = A[i*n + j];                        //
46         A[i*n + j] = A[j*n + i];                           // Swap matrix elements
47         A[j*n + i] = c;                                    //
48       }
49 }
```

リスト 9: ループタイリングによって並列インプレース平方行列転置の実装を改良

参考文献 (英語)

[1] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.
http://supertech.csail.mit.edu/papers/Prokop99.pdf.

[2] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science*, 1999.
http://doi.ieeecomputersociety.org/10.1109/SFFCS.1999.814600.

[3] D. Tsifakis, Alistair P. Rendell, and Peter E. Strazdins. Cache Oblivious Matrix Transposition: Simulation and Experiment. In *International Conference on Computational Science*, pages 17–25, 2004.
http://www.springeronline.com/3-540-22115-8.

[4] Andrey Vladimirov. Cache Traffic Optimization on Intel Xeon Phi Coprocessors for Parallel In-Place Square Matrix Transposition with Intel Cilk Plus and OpenMP.
http://research.colfaxinternational.com/post/2013/04/25/Transposition-Xeon-Phi.aspx.

[5] Boost C++ Libraries.
http://www.boost.org/.

[6] FFTW Library.
http://www.fftw.org/.

[7] Intel Math Kernel Library.
http://software.intel.com/en-us/intel-mkl.

[8] Greg Ruetsch and Paulius Micikevicius. Optimizing Matrix Transpose in CUDA, June 2010.
http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf.

[9] Mark Harris. An Efficient Matrix Transpose in CUDA C/C++, Feb 2013.
https://developer.nvidia.com/content/efficient-matrix-transpose-cuda-cc.

[10] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers.
http://www.cs.virginia.edu/stream/.

[11] Karthik Raman. Optimizing Memory Bandwidth on Stream Triad, Feb 2013.
http://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad.

[12] Intel Xeon Processor E5-2860 (20M Cache, 2.70 GHz, 8.00 GT/s Intel QPI).
http://ark.intel.com/ru/products/64583.

[13] Intel Xeon Phi Coprocessor 7120P (16GB, 1.238 GHz, 61 core).
http://ark.intel.com/products/75799.

[14] Intel Math Kernel Library, mkl_imatcopy.
http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/hh_goto.htm#GUID-B4584828-531D-4385-8F63-22C760C9B4B4.htm.