データとコードの並べ替え: 最適化とメモリー – パート 1

この記事は、インテル® デベロッパー・ゾーンに公開されている「Putting Your Data and Code in Order: Optimization and Memory – Part 1」の日本語参考訳です。

このシリーズ (全 2 パート) のパート 1 では、データとメモリーレイアウトのパフォーマンスへの影響を説明し、ソフトウェア・パフォーマンスを向上するためのステップを紹介します。ここで示す基本的なステップに従って、パフォーマンスを大幅に向上できる可能性があります。ソフトウェア・パフォーマンスの最適化に関する多くの記事は、分散メモリー型並列処理 (MPI など)、共有メモリー型並列処理 (スレッドなど)、SIMD (ベクトル化) のいずれかを取り上げていますが、並列処理を検討する場合、これらすべてを考慮する必要があります。メモリーも、見過ごされがちですが、これら 3 つの項目と同じぐらい重要です。ソフトウェア・アーキテクチャーと並列設計の変更は、メモリーとパフォーマンスに影響します。

このシリーズは中級者向けです。一般的な C、C++、Fortran プログラミングを使用してソフトウェア・パフォーマンスを最適化したいと考えている方を対象にしています。アセンブリーと組込み関数の使用は上級者向けなので、このシリーズでは取り上げません。プロセッサーの命令セット・アーキテクチャー (ISA) に関する資料や、さまざまな学術雑誌に掲載されているデータの配置とレイアウトについての分析と設計に関する資料に目を通して、理解を深めることをお勧めします。

データとコードの並べ替えは、データの移動を最小限に抑え、データを使用される場所の近くに配置するという 2 つの基本原則に基づいて行います。データがプロセッサー・レジスターにロードされるか、近くに配置されたら、そのデータが退避またはプロセッサーの実行ユニットから遠い場所に移動される前に、できるだけ活用する必要があります。

データの配置

データは異なるメモリー階層に格納することができます。このことについて考えてみましょう。実行ユニットに最も近いのは、プロセッサー・レジスターです。レジスターに格納されているデータは、インクリメント、乗算、加算、比較、ブール演算にすぐに利用できます。通常、マルチコア・プロセッサーの各コアには、プライベートの 1 次キャッシュ (L1) があります。L1 キャッシュからレジスターへのデータ移動は素早く行うことができます。キャッシュには複数のレベルがありますが、通常、少なくとも最終レベルキャッシュ (LLC) はプロセッサーのすべてのコアによって共有されます。中間レベルキャッシュの数と、それらが共有かプライベートかは、プロセッサーに依存します。インテル® プラットフォームでは、複数のソケットがある場合でも、単一プラットフォーム内でキャッシュの一貫性が保持されます。キャッシュからレジスターへのデータ移動のほうが、メインメモリーからのデータフェッチよりも高速です。

データの配置、プロセッサー・レジスターとの距離、アクセスにかかる相対時間を図 1 に示します。レジスターに近いブロックのほうが、移動時間が短く、実行のためデータをレジスターにロードするレイテンシーも低くなります。キャッシュは、レイテンシーが最も低く、最も高速です。次に高速なのはメインメモリーです。メモリーには複数のレベルが存在する可能性があります。メモリーの異なるレベルについては、パート 2 (英語) で取り上げます。ページがハードディスクまたは SSD にスワップアウトされると、仮想メモリーが非常に遅くなる可能性があります。従来の MPI のインターコネクト・ファブリック (イーサネット、InfiniBand* など) を介した送受信は、ローカルデータを取得する場合よりもレイテンシーが高くなります。MPI によりリモートシステムからデータを移動する場合、アクセス時間はインターコネクト・ファブリック (イーサネット、InfiniBand*、インテル® True Scale、インテル® Omni-Path) に依存します。

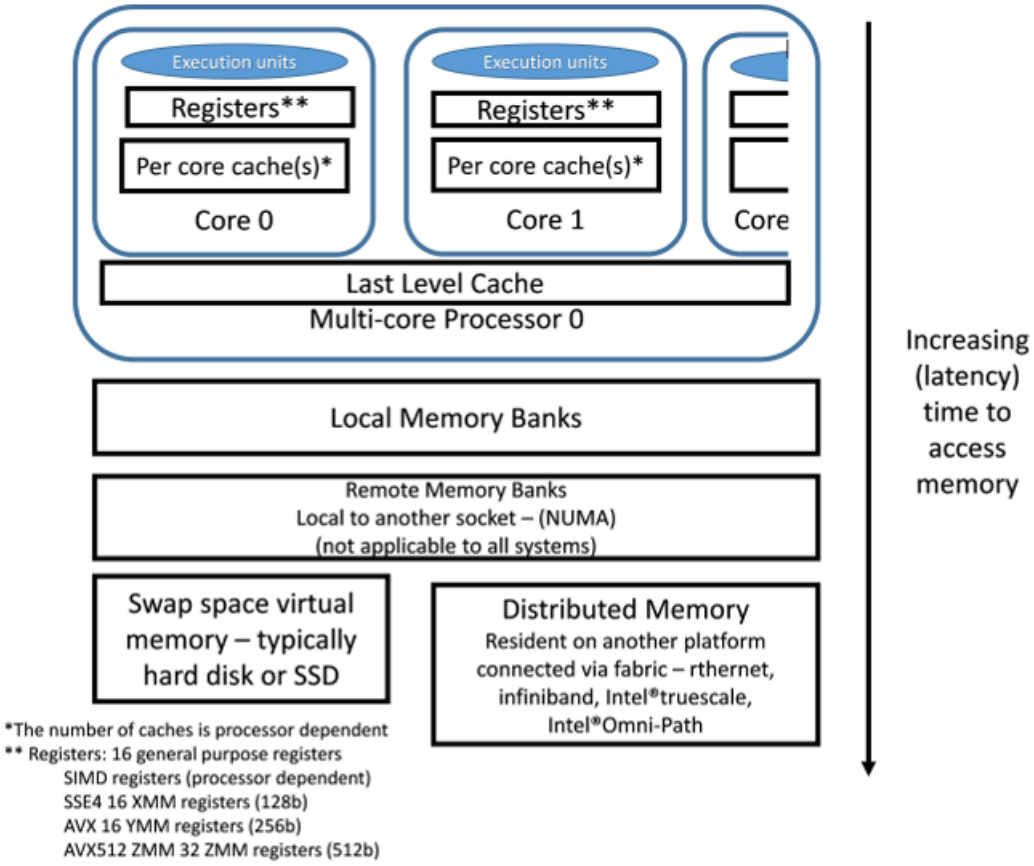

図 1. メモリーアクセスのレイテンシー (データアクセスの相対時間)

実行ユニットに最も近いのは、プロセッサー・レジスターです。レジスター数、データをレジスターへロードするレイテンシー、メモリー操作のキューの幅を考慮すると、データのロードには時間がかかるため、レジスターの値をそれぞれ一度だけ使用して、すべての実行ユニットをビジー状態に維持することは不可能です。データが実行ユニットの近くに配置されたら、キャッシュから退避されたり、レジスターから移動される前に、そのデータをできるだけ再利用するようにします。一部の変数は、レジスターにのみ存在し、メインメモリーには格納されません。コンパイラーは、変数のスコープがレジスターのみかどうか認識できるため、C/C++ の "register" キーワードの使用は推奨しません。"register" キーワードは、コンパイラーによって無視されることがあります。

ここで重要なことは、ソフトウェア開発者がコードを確認し、データがどのように使用され、どれぐらいの期間存続しなければならないのかを考えることです。一時変数や一時配列を作成すべきかどうか、あるいはこれほど多くの一時変数が必要かどうかについて考えます。パフォーマンスを向上するには、最初にソフトウェアのパフォーマンス・メトリックを収集し、多くの実行時間が費やされているモジュールやコード領域でデータの局所性に注目すべきです。よく使用されているデータ収集ユーティリティーには、インテル® VTune™ Amplifier XE、gprof、Tau があります。

データの使用と再利用

この 2 つのステップを理解するため、行列乗算について考えてみます。n × n の 3 つの正方行列について、A = A + B * C を計算する場合、3 つの単純な入れ子の for ループで表すことができます。

```
for (i=0;i<n; i++)                      // 行 136
   for (j = 0; j<n; j++)                // 行 137
     for (k=0;k<n; k++)                 // 行 138
         A[i][j] += B[i][k]* C[k][j] ;  // 行 139
```

この順序付けでは、リダクション操作 (行 138 と行 139) が含まれていることが問題になります。行 139 の左辺は単一値です。コンパイラーは、行 138 の一部をアンロールして SIMD レジスターの幅を B と C の要素の 4 つまたは 8 つの積で埋めようとしますが、それらの積を合計して単一値にする必要があります。これはリダクション操作です。リダクション操作では、SIMD 実行ユニットのすべての幅を効率良く活用することができないため、並列パフォーマンスが得られません。リダクション操作を軽減または排除することで、並列パフォーマンスが向上します。ループ内の左辺に単一値がある場合、リダクション操作の可能性があります。行 137 の 1 つの反復 (`i = 2`、`j = 2`) のデータ・アクセス・パターンを図 2 に示します。

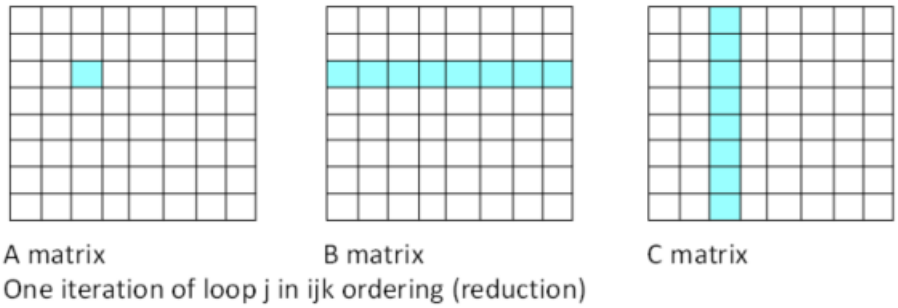

図 2. 順序付け (行列 A の単一値)

場合によっては、操作を並べ替えることでリダクション操作を排除できます。2 つの内部ループをスワップする順序付けについて考えてみます。浮動小数点演算の数は同じですが、リダクション操作 (値を左辺に合計する処理) を排除することで、プロセッサーは SIMD 実行ユニットと SIMD レジスターの幅をすべて利用できるようになります。これにより、パフォーマンスが大幅に向上します。

```
for (i=0;i<n; i++)                    // 行 136
   for (k = 0; k<n; k++)              // 新しい行 137
     for (j=0;j<n; j++)               // 新しい行 138
       a[i][j] += b[i][k]* c[k][j] ;  // 行 139
```

このコードは、A と C の要素に連続してアクセスします。

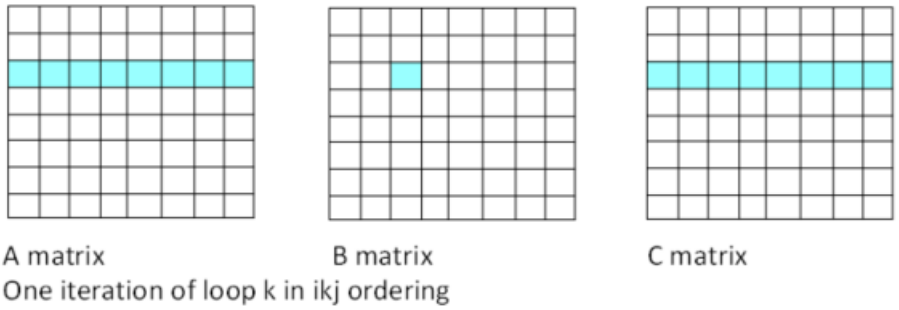

図 3. 更新後の順序付け (連続アクセス)

オリジナルの `i-j-k` 順序はドット積です。2 つのベクトルのドット積を使用して、A の各要素の値を計算します。`i-k-j` 順序は saxpy または daxpy 操作です。1 つのベクトルの倍数が別のベクトルに加算されます。ドット積と axpy 操作はどちらもレベル 1 BLAS (英語) ルーチンです。`i-k-j` 順序では、リダクション操作は不要です。C の行のサブセットと行列 B のスカラーを掛けて、結果を A の行のサブセットに加算します (コンパイラーは、ターゲットの SIMD レジスター (インテル® ストリーミング SIMD 拡張命令 4 (インテル® SSE4)、インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX)、インテル® AVX-512) に応じてサブセットのサイズを決定します)。新しい行 137 のループの 1 反復 (`i = 2`、`j = 2`) のメモリーアクセスを図 3 に示します。

ドット積のリダクション操作を排除したことは大きな改善点です。最適化レベル O2 でコンパイルすると、インテル® コンパイラーと gcc はどちらも、SIMD レジスターと SIMD 実行ユニットを活用するベクトル化されたコードを生成します。さらに、インテル® コンパイラーは、j ループと k ループをスワップします。これは、/Qopt-report (Windows®) または -qopt-report (Linux*) オプションを指定すると生成される、コンパイラーによる最適化レポートで確認できます。デフォルトでは、最適化レポートは filename.optrpt という名前で生成されます。以下は、この例で生成された最適化レポートからの抜粋です。

```
ループの開始 mm.c(136,4)
   リマーク #25444: ループの入れ子の交換: ( 1 2 3 ) --> ( 1 3 2 )
```

レポートから交換されたループもベクトル化されたことが分かります。

```
ループの開始 mm.c(137,7)
    リマーク #15301: 変換されたループがベクトル化されました。
 ループの終了
```

gcc コンパイラー 4.1.2-55 では、このループのスワップが自動的に行われないため、開発者が手動でスワップする必要があります。

ループをブロックに分割 (ブロッキング) してより多くのデータが再利用されるようにすることで、パフォーマンスがさらに向上します。図 3 では、真ん中のループの各反復で、長さ n の 2 つのベクトル (およびスカラー) が参照されていますが、この 2 つのベクトルの各要素は一度しか使用されていません。n が大数の場合、真ん中のループの反復間でベクトルの各要素がキャッシュから退避される可能性が高いです。ループをブロックに分割してデータが再利用されるようにすると、パフォーマンスが向上します。

最終コードでは、j ループと k ループをスワップし、ブロッキングを追加して、一度にサイズ blockSize の部分行列または行列のブロックを操作します。この単純な例では、blockSize は n の倍数です。

```
for (i = 0; i < n; i+=blockSize)
   for (k=0; k<n ; k+= blockSize)
      for (j = 0 ; j < n; j+=blockSize)
         for (iInner = i; iInner<j+blockSize; iInner++)
            for (kInner = k ; kInner<k+blockSize; kInner++)
               for (jInner = j ; jInner<j+blockSize ; jInner++)
                 a[iInner,jInner] += b[iInner,kInner] *
                    c[kInner, jInner]
```

このモデルでは、ループ j の特定の 1 反復のデータアクセスは次のようになります。

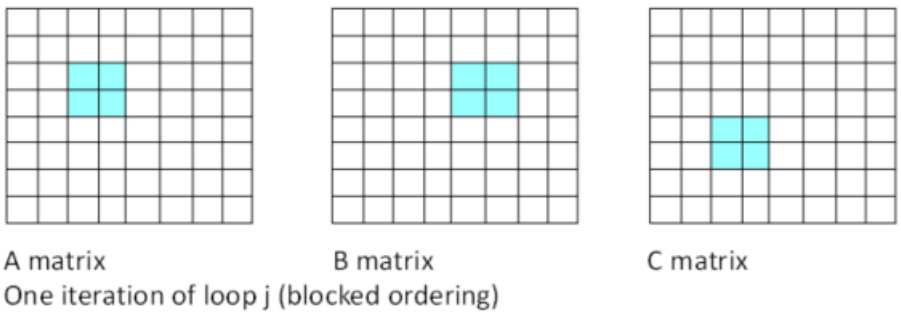

図 4. ブロックモデル

適切な blockSize を使用する場合、3 つの内部ループを処理する間、各ブロックはキャッシュ (または SIMD レジスター) に保持されると仮定できます。A、B、C の各要素または値は、SIMD レジスターから削除されるか、

キャッシュから退避される前に blockSize 回使用されます。これにより、データの再利用が blockSize 倍になります。小さな行列では、ブロッキングによるパフォーマンスの向上はわずかか見込めません。行列のサイズが大きいほうが、パフォーマンスの向上も大きくなります。

次の表は、あるシステムで異なるコンパイラーを使用して測定したパフォーマンス比です。インテル® コンパイラーは、行 137 と行 138 のループを自動的にスワップします。そのため、i-j-k 順序と i-k-j 順序の間に大きな違いは見られません。また、ベースラインと比較したスピードアップが低いように見えますが、これはベースライン・パフォーマンスがすでに高速なためです。

順序	行列/ブロックサイズ	gcc 4.1.2 (-O2 オプションを使用) スピードアップ (ベースラインと比較したパフォーマンス比)	インテル® コンパイラー 16.1 (-O2 オプションを使用) スピードアップ (ベースラインと比較したパフォーマンス比)
i-j-k	1600	1.0 (ベース)	12.32
i-k-j	1600	6.25	12.33
i-k-j ブロック	1600/8	6.44	8.44
i-j-k	4000	1.0 (ベース)	6.39
i-k-j	4000	6.04	6.38
i-k-j ブロック	4000/8	8.42	10.53

表 1. gcc とインテル® コンパイラーで測定したパフォーマンス比

ここで使用したサンプルコードは単純なため、どちらのコンパイラーも SIMD 命令を生成します。ここでは、コンパイラーの比較ではなく、操作の順序とリダクションの影響を理解することが目的のため、古いバージョンの gcc コンパイラーを使用しています。多くのループはより複雑で、すべての並べ替えの可能性を認識できるコンパイラーはありません、そのため、多くの実行時間を費やしているコード領域に注目し、コンパイラーにより並べ替えが行われているかどうかコンパイラー・レポートを確認して、されていない場合は手動で行うことを検討してください。大規模なデータでは、データをブロックに分割することも重要です。ここで使用したような小さな行列では、ブロッキングによるパフォーマンスの向上は見込めませんが、大きな行列ではパフォーマンスの大幅な向上が見込めます。そのため、ブロッキングを行う前に、データのサイズとキャッシュを考慮してよく検討すべきです。いくつかの入れ子のループを追加し、適切なブロックサイズを使用することで、オリジナルコードと比較して 2 ～ 10 倍のスピードアップを達成することが可能です。大した労力を必要としないため、これは大きなパフォーマンス・ゲイトと言えるでしょう。

最適化されたライブラリーの使用

パフォーマンスをさらに向上することができます。上記のブロック化したコードは、インテル® マス・カーネル・ライブラリー (インテル® MKL) などの最適化された LAPACK に含まれるレベル 3 BLAS (英語) ルーチンの DGEMM (英語) を使用することで、さらなるパフォーマンス・ゲインがもたらされます。一般的な線形代数やフーリエ変換では、インテル® MKL のような近代的なライブラリーは、単純なブロッキングや並べ替えよりもはるかに優れた最適化を提供します。可能な場合は、パフォーマンスがチューニングされたこれらの最適化されたライブラリーを利用すべきです。

行列乗算向けの最適化されたライブラリーはありますが、ブロッキングによってパフォーマンスが向上するすべてのケースで最適化されたライブラリーを利用できるわけではありません。行列乗算は、原則を簡単に理解するための一例です。有限差分ステンシルも最適化されたライブラリーによって大きな利点が得られます。

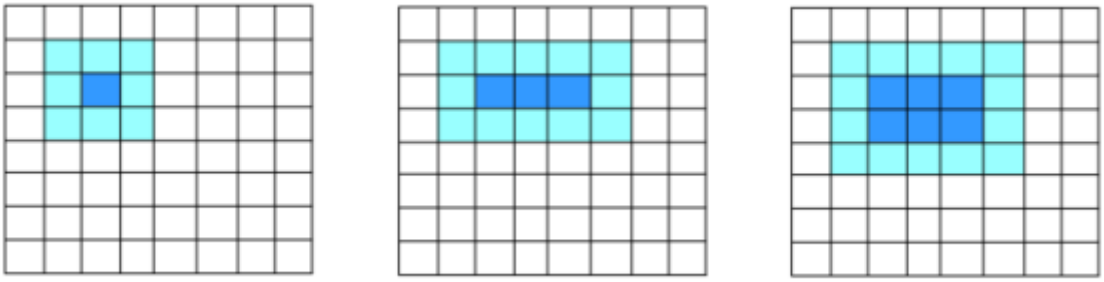

図 5. 2 次元のブロックモデル

単純な 9 ポイントステンシル (図 5 の左の図) は、ハイライトされたブロックを使用して、中央のブロックの値を更新します。1 つの位置を更新するのに、9 つの値がロードされます。隣接する位置の更新では、9 つの値のうち 6 つが再利用されます。コードの実行が進むと、図 5 の中央の図のように、3 つの位置を更新するため 15 の値がロードされます。漸近的に、位置の数とロードされる値の数の比率は 1:3 になります。

2 次元ブロックの場合、ブロック幅が 2 となり、図 5 の右の図のように、6 つの位置を更新する場合、20 の値がレジスターにロードされます。この場合、位置の数とロードされる値の数の比率は、漸近的に 1:2 になります。

有限差分に興味のある方は、Cedric Andreolli の「等方性 3 次元有限差分コード向けの 8 つの最適化」(英語) や「等方性 3 次元有限差分 (3DFD) 波動方程式コード向けの NUMA を理解する」を一読されることを推奨します。ブロッキングに加えて、この記事では、ほかのメモリーの最適化手法についても説明しています。

まとめ

この記事では、開発者がソフトウェアに適用可能な 3 つの主要なステップを示しました。最初に、並列リダクションを回避するように操作を並べ替えます。次に、データの再利用の可能性を見つけ、入れ子のループをブロックに分割してデータが再利用されるようにします。一部の操作では、これによりパフォーマンスが 2 倍向上します。最後に、可能な場合は、最適化されたライブラリーを利用します。最適化されたライブラリーは、通常、単純な並べ替えをはるかに上回るパフォーマンスをもたらします。

サンプルコードのダウンロード

パート 2 (英語) では、SIMD レジスターだけでなく、複数のコアにわたる並列処理について考え、フォルス・シェアリングと構造体配列 (AoS) を説明します。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。