

THE PARALLEL UNIVERSE

現在と将来の OpenMP*

代表的な並列プログラミング言語の進化

マルチコア / メニーコアのインテル[®] アーキテクチャー
におけるパフォーマンスの向上

並列アプリケーションのスケーラビリティの問題を
特定する

目次

編集者からのメッセージ

3

継続した HPC 分野の変化

Henry A. Gabb インテル コーポレーション 主任エンジニア

注目記事

現在と将来の OpenMP* API 仕様

5

代表的な並列プログラミング言語の各バージョンにおける進化

行列 - 行列乗算のパックのオーバーヘッドを減らす

20

マルチコア / メニーコアのインテル® アーキテクチャーにおけるディープ・ニューラル・ネットワークのパフォーマンスの向上

並列アプリケーションのスケーラビリティの問題を特定する

25

インテル® VTune™ Amplifier XE のメモリー解析を使用してインテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ プロセッサー向けにスケーラビリティを向上する方法

インテル® AVX-512 で向上したベクトル化のパフォーマンス

46

インテル® コンパイラーでループをベクトル化してスピードアップするさまざまな例

インテル® Advisor のルーフライン解析

55

パフォーマンス最適化のトレードオフを視覚化する新しい方法

インテルが推進するディープラーニング・フレームワーク

73

さらなる洞察を提供

継続した HPC 分野の変化

Henry A. Gabb インテル コーポレーション 主任エンジニア

HPC と並列コンピューティング分野で長年の経験があり、並列プログラミングに関する記事を多数出版しています。『Developing Multithreaded Applications: A Platform Consistent Approach』の編集者 / 共著者で、インテルと Microsoft® による Universal Parallel Computing Research Centers のプログラママネージャーを務めました。

OpenMP* 20 周年

OpenMP* アプリケーション・プログラミング・インターフェイスが 20 周年を迎えることを記念して、Michael Klemm (OpenMP* Architecture Review Board (ARB) の現 CEO) と彼の同僚が、最新の機能の概要、特にタスクベースの並列処理の拡張と専用アクセラレーターへの計算のオフロードについて話してくれました。

詳しい内容は注目記事「[現在と将来の OpenMP* API 仕様](#)」をご覧くださいとして、ここでは OpenMP* の歴史について簡単に述べたいと思います。私は冗談半分で、1990 年代前半はハイパフォーマンス・コンピューティング (HPC) にとって「嫌な時代」であったと言うことがあります。当時、急速に変わりつつある HPC 分野では、多数の並列プログラミング・モデルと並列アーキテクチャーが点在していました。分散メモリー・アーキテクチャーでは、SHMEM のような低レベルのメッセージパッシング手法、PVM や **MPI** のような高レベルのメッセージパッシング手法、そして High Performance Fortran や Unified Parallel C によるさらに高レベルの抽象化がありました。共有メモリー・アーキテクチャーでは、pthreads のような低レベルのスレッド化手法や高レベルのコンパイラーに指示する並列化がありました。1 つははっきりしていたことは、実際のアプリケーションを自動的に並列化できる魔法のようなコンパイラーはないということでした。次善の策として、並列コンパイラー・ディレクティブがありました。

OpenMP* が登場する前の並列コンパイラー・ディレクティブをご存知の方は、ベンダー (Cray*, SGI*, インテル、Kuck and Associates, Inc. など) ごとに異なるセットがあり、それぞれが同じことを異なる構文で行っていたことを記憶されているでしょう。

その後、いくつかの主要な政府系 HPC 施設が並列コンパイラー・ディレクティブ構文の統一を要求したことで、1997 年に OpenMP* が誕生しました。設立時のベンダーのほとんどが現在も ARB に残っており、その後多数のメンバーが加わりました (ARB は現在 29 のメンバーで構成されています)。OpenMP* は、当初の目的を見失うことなく、可搬性のあるベンダー・ニュートラルな並列プログラミング・ディレクティブとして不動の地位を確立しています。

今日、HPC 分野におけるほとんどのアプリケーション要件は MPI と OpenMP* で対応できます。しかし、まだ課題があります。メモリー・サブシステムはこれまでになく不均衡で、同じシステムに異なるプロセッサ・アーキテクチャーが存在することは珍しくなく、異なる処理要素間でデータの一貫性を保持することが、プログラマーの負担となっています。ただし、これらの課題に対応するべく MPI と OpenMP* は進化を続けており、HPC の未来は明るいと言えるでしょう。

シリアル・パフォーマンスのチューニングを支援する新しいツール

並列化は素晴らしいことですが、適切にチューニングされていないコードを並列化しようとは思わないでしょう。この号では、シリアル・パフォーマンスのチューニングについても取り上げています。私は、最初に扱ったスーパーコンピュータが Cray* X-MP であったため、早くからベクトル化の重要性について学びました。「[Intel® AVX-512 で向上したベクトル化のパフォーマンス](#)」では、新しい Intel® アドバンスド・ベクトル・エクステンション 512 (Intel® AVX-512) 命令セットを利用してコードをチューニングし、以前は不可能だったベクトル化について紹介します。新しい Intel® Advisor のルーフライン機能と Intel® VTune™ Amplifier XE のメモリー解析機能は、パフォーマンス最適化のトレードオフを視覚化し、メモリーアクセスのアプリケーション・パフォーマンスへの影響を把握するのに役立ちます。これらの機能については、「[Intel® Advisor のルーフライン解析](#)」と「[並列アプリケーションのスケラビリティの問題を特定する](#)」で詳しく説明します。そして、「[行列 - 行列乗算のパックのオーバーヘッドを減らす](#)」では、Intel® マス・カーネル・ライブラリー (Intel® MKL) で汎用行列 - 行列乗算を最適化するためのヒントを、「[Intel が推進するディープラーニング・フレームワーク](#)」では、Intel® ソフトウェアによるマシンラーニングのサポートに関する情報を提供します。

新しい編集者について

最後に、簡単に自己紹介したいと思います。この度、The Parallel Universe の編集を担当することになりました Henry A. Gabb です。私は、1990 年頃から HPC 分野に携わっています。最初は、計算生命科学の研究に取り組んでいましたが、研究プロジェクトを重ねるたびに、より多くの計算能力が必要になり、[パフォーマンス・チューニング](#)や並列プログラミングについて学びました。大学では生化学と遺伝子を専攻していたため、コンピューター・サイエンスの科学分野への進出を快く思っていませんでした。しかし、HPC が研究を変え、新しく、より大きな研究課題への取り組みを可能にすることを理解してからは、当初の抵抗がなくなりました。ハードウェアとソフトウェアの進化により、かつては Circa 1995 スーパーコンピュータで数日かかっていたシミュレーションを、ラップトップ上で短時間で実行できるようになりました。以前は、来るべきヘテロジニアス並列コンピューティング時代に不安を感じていましたが、今では大学院生のときのように大いに魅了されています。

Henry A. Gabb

2017 年 1 月





現在と将来の OPENMP* API 仕様

代表的な並列プログラミング言語の各バージョンにおける進化

Michael Klemm Intel Deutschland GmbH シニア・スタッフ・アプリケーション・エンジニア、
Alejandro Duran Intel Corporation Iberia アプリケーション・エンジニア、
Ravi Narayanaswamy インテル コーポレーション シニアスタッフ開発エンジニア、
Xinmin Tian 同シニア主任エンジニア、Terry Wilmarth 同シニア開発エンジニア

OpenMP* API には 20 年の歴史があります。その登場以来、ハードウェアとソフトウェアの進歩とともに、新しいハードウェアのプログラミングに対応するさまざまな機能が追加されてきました。2013 年にリリースされたバージョン 4.0 から、OpenMP* 言語はヘテロジニアス・プログラミングと SIMD プログラミングをサポートしました。同様に、2008 年には、タスク構文の追加により、不規則な並列処理を含むプログラムのサポートが向上しました。OpenMP* Technical Report 4: Version 5.0 Preview 1 (略称 TR4) は、OpenMP* 言語の進化における次のステップです。タスク・リダクションの追加、SIMD 並列処理の拡張が行われ、ヘテロジニアス・プログラミングの生産性が大幅に向上します。この記事では、既存の OpenMP* 機能を確認した後、TR4 をサポートする実装で追加される機能を紹介します。

タスクベースのプログラミング: タスクで表現する

タスクベースのプログラミングは、不規則な並列処理（再帰アルゴリズム、グラフのトラバース、非構造化データを操作するアルゴリズムなど）が必要なアプリケーションにとって重要な概念です。OpenMP* バージョン 3.0 から、OpenMP* ランタイムシステムのスケジューラーに渡される小さな単位のワークの並列実行を容易に表現する方法として、タスク構文が提供されました。

```
void taskloop_example() {
    #pragma omp taskgroup
    {
        #pragma omp task
        long_running_task() // 同時に実行可能

        #pragma omp taskloop collapse(2) grainsize(500) nogroup
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                loop_body();
    }
}
```

1 新しい taskloop 構文で OpenMP* タスクを利用する例

図 1 は、`long_running_task` 関数を実行する OpenMP* タスクを生成し、その後 `taskloop` 構文を使用してループを並列化する例を示しています。この構文は OpenMP* 4.5 で追加されたもので、プログラマーが OpenMP* タスクを使用してループを簡単に並列化できる糖衣構文を提供します。この構文は、ループの反復空間をチャンクに分割し、それぞれのチャンクに対して 1 つのタスクを生成します。細かい制御ができるように、いくつかの節（例えば、タスクあたりのワークの量を制御する `grainsize` や `i` ループと `j` ループを 1 つのループにまとめる `collapse`）をサポートしています。TR4 では、`taskgroup`、`task`、`taskloop` 構文に生成されたタスク間でリダクションを行う新しい節を定義することにより、OpenMP* タスクの表現が拡張されます。

図 2 は、リンクリストを処理してリストの全要素の最小値を調べるタスクの例を示しています。`parallel` 構文は、ワーカーレッドがタスク実行で利用できるように並列領域を作成します。`single` 構文は、リンクリストのトラバースを 1 つのスレッドの実行に制限し、`omp` タスクにより各リスト項目に 1 つのタスクを生成します。これは OpenMP* で生産と消費パターンを実装する一般的な方法です。

TR4 のタスク・リダクションは、OpenMP* バージョン 4.0 で追加された `taskgroup` 構文を使用します。この構文は、タスクを論理的にグループ化し、グループのすべてのタスクの完了を待機する方法を提供します。TR4 では、**図 2** に示すように、`task_reduction` 節によりリダクションを行うように、`taskgroup` 構文が拡張されます。この節が構文に追加されると、`taskgroup` 領域の最後で個々のタスクにより収集された部分結果がすべて集計され、最終的な結果が生成されます。リダクション操作に関係するタスクには、`taskgroup` の `reduction` 節と一致する `in_reduction` 節が必要です。

TR4 から、**taskloop** 構文はタスク・リダクション・セマンティクスによる **reduction** 節および **in_reduction** 節をサポートします。**reduction** 節が **taskloop** 構文に含まれていると、ループの最後で要求されたリダクション操作を実行する、暗黙的なタスクグループが作成されます。**in_reduction** 節が追加されると、**taskloop** 構文により生成されたタスクは外側の **taskgroup** 領域のリダクションに関係します。

```
int find_minimum(list_t * list) {
    int minimum = INT_MAX;
    list_t * ptr = list;
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskgroup task_reduction(min:minimum)
    {
        for (ptr = list; ptr ; ptr = ptr->next) {
            #pragma omp task firstprivate(ptr) in_reduction(min:minimum)
            {
                int element = ptr->element;
                minimum = (element < minimum) ? element : minimum;
            }
        }
    }
    return minimum;
}
```

2 タスク・リダクションを使用したリンクリストのトラバースと最小値の計算

オフロード：コプロセッサを最大限に利用する

OpenMP* API は、ユーザーのフィードバックに基づいてオフロードプラグマが使いやすくなるように努めています。これにより、新機能が TR4 に追加され、いくつかの既存の機能が強化されました。重要な新機能の 1 つは、オフロード領域で呼び出されている関数を自動的に検出して、それらの関数が **declare target** ディレクティブで指定されているかのように扱います。以前は、オフロード領域で呼び出されるすべての関数は、**declare target** ディレクティブによって明示的にタグ付けされている必要がありました。特にプログラマーが関係していないヘッダーファイルにルーチンが含まれている場合 (標準テンプレート・ライブラリーなど)、ヘッダーファイル自体に **declare target** ディレクティブが必要になるため (その結果、一部の関数がオフロード領域で使用されていない場合でもデバイスのヘッダーファイルにすべての関数のコピーを作成する必要があり)、これは大変な作業でした。

```
#pragma omp declare target
void foo() {
    // ...
}
#pragma omp end declare target

void bar() {
    #pragma omp target
    {
        foo();
    }
}
```

```
void foo() {
    // ...
}

void bar() {
    #pragma omp target
    {
        foo();
    }
}
```

3 オフロード領域で使用される関数を自動的に検出する

OpenMP* バージョン 4.5 では、**図 3** の左側のようにコードを記述する必要があります。TR4 では、右側に示すコードでデバイス関数を暗黙的に検出して作成できます。

TR4 では、静的記憶域を含む変数も自動的に検出できます。これにより、**図 4** の 2 つの例は等価となります。

```
int x;
#pragma omp declare target to (x)

void bar() {
    #pragma omp target
    {
        x = 5;
    }
}
```

```
int x;

void bar() {
    #pragma omp target
    {
        x = 5;
    }
}
```

4 静的記憶域を含む変数を自動的に検出する

OpenMP* バージョン 4.5 では、**use_device_ptr** 節が追加されました。**use_device_ptr** の変数は、使用する前にマップする必要があります。変数は 1 つのデータ節にのみ記述できるため、プログラマーは個別の **#pragma target data** 節を記述する必要があります。そのため、**図 5** の OpenMP* ディレクティブが必要になります。


```
#pragma omp target data map(buf)
#pragma omp target data use_device_ptr(buf)
```

5 変数のマップ

TR4 では、[図 6](#) に示すように、単一構文で変数を `map` 節と `use_device_ptr` 節の両方に記述できる例外が追加されました。

```
#pragma omp target data map(buf) use_device_ptr(buf)
```

6 変数を `map` 節と `use_device_ptr` 節の両方に記述できる

スタティック・データ・メンバーが `omp declare target` 構文内のクラスで使用できるようになりました。また、スタティック・メンバーを含むクラス・オブジェクトは `map` 節でも使用できます ([図 7](#))。

```
#pragma omp declare target
class C {
    static int x;
    int y;
}
class C myclass;
#pragma omp end declare target

void bar() {
#pragma omp target map(myclass)
{
    myclass.x = 10
}
}
```

7 `map` 節で使用するスタティック・メンバーを含むクラス・オブジェクト

さらに、仮想メンバー関数を `omp declare target` 構文内のクラスまたは `map` 節で使用されているオブジェクトで使用できるようになりました。唯一の注意点は、オブジェクトが同じデバイスで作成される場合、仮想メンバー関数はそのデバイス上でのみ呼び出すことができる点です。

OpenMP* 4.5 では、最初の構造がターゲットである結合構造の `reduction` 節または `lastprivate` 節で使われるスカラー変数は、ターゲット構造の `firstprivate` として扱われます。ホストの変数が更新されることはありません。ホストの変数を更新するには、プログラマーは結合構造から `omp target` ディレクティブを分離してスカラー変数を明示的にマップする必要があります。TR4 では、これらの変数は自動的に `map(tofrom:variable)` が適用されているかのように扱われます。

名前付き配列のセクションを **omp target** データを使用してマップする場合、配列を参照する **omp target data** 構文内の入れ子の **omp target** は、同じセクションまたは外部の **omp target data map** 節で使用される配列のサブセクションに暗黙的にマップする必要があります。内側の **omp target** 領域で明示的なマップが省略されると、暗黙的なマップ規則が適用され、配列全体が OpenMP* バージョン 4.5 の仕様に従ってマップされます。これにより、配列のサブセクションがすでにマップされている場合、大きなサイズの配列をマップするとランタイムエラーが発生します。同様に、外側の **omp target data** 構文内で構造体変数のフィールドをマップして、内側の入れ子の **omp target** 構文内で構造体変数のアドレスを使用すると、構造体の一部がすでにマップされている場合、構造体変数全体をマップしようとします。TR4 では、プログラマーが想定した動作になるように、これらのケースが修正されました (図 8)。

```
struct {int x,y,z} st;
int A[100];
#pragma omp target data map(s.x A[10:50])
{
    #pragma omp target
    {
        A[20] = ;          // OpenMP* 4.5 ではエラー、TR4 では OK
        foo(&st);          // OpenMP* 4.5 ではエラー、TR4 では OK
    }
    #pragma omp target map(s.x, A[10:50])
    {
        A[20] = ;          // OpenMP* 4.5 と TR4 の両方で OK
        foo(&st);          // OpenMP* 4.5 と TR4 の両方で OK
    }
}
```

8 改良されたマップ処理

TR4 の新機能により、OpenMP* を使用したオフロードのプログラミングの容易性が向上し、アプリケーションに要求される変更点が少なくなります。**target** 領域で使用される変数と関数は自動的に検出されるため、明示的に指定する必要はなくなります。同様に、入れ子の領域の内側で **map** 節を繰り返す必要性がなくなります。**map** 節と **use_device_ptr** 節の両方に変数を記述できるため、必要な OpenMP* ディレクティブの数が減ります。リダクション変数の動作は、プログラマーの想定に沿うように変更されます。全体的に、セマンティクスが簡潔になり、OpenMP* アプリケーション内でより簡単に、さらに直感的にオフロードデバイスを利用できるようになります。

効率的な SIMD プログラミング

繰り返し間の依存関係がある SIMD ループ

OpenMP* バージョン 4.5 では、**ordered** 構文に新しく **simd** 節が追加され機能が拡張されました。**ordered simd** 構造は、SIMD ループまたは SIMD 関数の構造化ブロックを反復順または関数呼び出しの順で実行することを宣言します。**図 9** の左のコード例は、**ordered simd** ブロックを使用して各反復内および反復間の読み取り / 書き込み、書き込み / 読み取り、書き込み / 書き込みの順序を保存する方法を示しています。すべてのループは SIMD 命令を使用して同時に実行できます。最初の **ordered simd** ブロックでは、配列のインデックス **ind[i]** の書き込み / 書き込みが重複する可能性があるため (例えば、**ind[0] = 2, ind[2] = 2**)、すべてのループが**ベクトル化**できるように **ordered simd** でシリアル化する必要があります。2 つ目の **ordered simd** ブロックでは、**myLock(L)** 操作と **myUnlock(L)** 操作を 1 つの **ordered simd** ブロックで行う必要があります。そうしないと、ループベクトル化の一部として (例えば、ベクトル長 2 の場合)、**myLock(L)** と **myUnlock(L)** の呼び出しが **{myLock(L); myLock(L); ...; myUnlock(L); myUnlock(L);}** のように 2 つの呼び出しになります。一般に、ロック関数を入れ子にするとデッドロックが発生します。例で示している **ordered simd** 構文は、適切なシーケンス **{myLock(L); ...; myUnlock(L); ...; myLock(L); myUnlock(L);}** を作成します。

```
#pragma omp simd
for (i = 0; i < N; i++) {
    // ...
    #pragma omp ordered simd
    {
        // 書き込み/書き込みの重複
        a[ind[i]] += b[i];
    }
    // ...
    #pragma omp ordered simd
    {
        // アトミック更新
        myLock(L)
        if (x > 10) x = 0;
        myUnlock(L)
    }
    // ...
}
```

```
#pragma omp simd
for (i = 0; i < N; i++) {
    // ...
    #pragma omp ordered simd
    {
        if (c[i] > 0) q[j++] = b[i];
    }
    // ...
    #pragma omp ordered simd
    {
        if (c[i] > 0) q[j++] = d[i];
    }
    // ...
}
```

9 各反復内および反復間の読み取り / 書き込み、書き込み / 読み取り、書き込み / 書き込みの順序を保存する

ordered 構文で **simd** 節を使用する場合、2 つの **ordered simd** ブロック間の固有の依存関係に違反しないように注意してください。**図 9** の右のコード例は、**#pragma omp ordered simd** の誤った使い方 (シリアル実行時の格納順が SIMD 実行で変更される) を示しています。**c[0] = true** および **c[1] = true** と仮定します。上記のループがシリアルに実行された場合、格納順は **q[0] = b[0], q[1] = d[0], q[2] = b[1], q[3] = d[1], ...** になります。しかし、ループがベクトル長 2 で同時に実行された場合、格納順は **q[0] = b[0], q[1] = b[1], q[2] = d[0], q[3] = d[1], ...** になります。格納順の違いは、ループの 2 つの **ordered simd** ブロック間の変数 **j** の書き込み / 読み取り依存関係の違反によるものです。正しい使い方は、2 つの **ordered simd** ブロックを 1 つの **ordered simd** ブロックにマージすることです。

ref/uval/val 修飾子を linear 節に追加

linear 節は **private** 節で指定される機能のスーパーセットを提供します。**linear** 節が構文で指定されると、関連するループの各反復の新しいリスト項目の値は、構文に入る前のオリジナルのリスト項目の値に対応し、反復の論理数と **linear** ステップを掛けた値が加えられます。そして、関連するループの連続する最後の反復に対応する値がオリジナルのリスト項目に割り当てられます。**linear** 節が宣言型のディレクティブで指定された場合、すべてのリスト項目は各 SIMD レーンで同時に呼び出される関数の仮引数になります。

ref/uval/val 修飾子を **linear** 節に追加した理由は、コンパイラーがギャザー / スキャッターの代わりにユニットストライド方式のロード / ストアを使用して効率的な SIMD コードを生成できるように、アドレスとデータ値に関してメモリー参照の **linear** または **uniform** プロパティを正確に指定する方法をプログラマーに提供するためです。基本的に、暗黙的に参照される **linear** 引数は、**linear** として参照するほうが良いでしょう。**uval/val/ref** のセマンティクスは次のようになります。

- **linear(val(var):[step])** は、**var** が参照で渡されている場合でも値が **linear**であることを示します。アドレスのベクトルは参照渡しで渡されます。この場合、コンパイラーはギャザーまたはスキャッターを生成する必要があります。
- **linear(uval(var):[step])** は、参照で渡される値は **linear** で、参照自体は **uniform**であることを示します。そのため、最初のレーンの参照は渡されますが、ほかの値は **step** を使用して構築します。コンパイラーは、汎用レジスターを使用してベースアドレスを渡し、その **linear** 値を計算します。
- **linear(ref(var):step)** は、パラメーターは参照で渡され、根本的な参照は **linear** で、メモリーアクセスは **step** の値に応じて **linear** ユニットストライドまたは非ストライドであることを示します。コンパイラーは、汎用レジスターを使用してベースアドレスを渡し、その **linear** アドレスを計算します。

図 10 は、関数 **FOO** と引数 **X** および **Y** (Fortran で参照で渡される) を示しています。“**VALUE**” 属性はこの動作を変更しません。Fortran 2008 言語仕様では更新された値が呼び出し元に見えないことを示すだけです。**X** と **Y** の参照は **linear** として示されていないため、コンパイラーは、ベクトル長が 4 であると仮定して、(**X0**, **X1**, **X2**, **X3**) および (**Y0**, **Y1**, **Y2**, **Y3**) をロードするギャザー命令を生成する必要があります。

図 11 では、**X** と **Y** の参照が **linear** として示されているため、コンパイラーはパフォーマンスに優れたユニットストライド方式の SIMD ロードを生成することができます。

```

REAL FUNCTION FOO(X, Y)
!$omp declare simd(FOO)
  REAL, VALUE :: Y      !! 参照渡し
  REAL, VALUE :: X      !! 参照渡し
  FOO = X + Y            !! アドレスのベクトルに
                        !! 基づいて生成された
                        !! ギャザー
END FUNCTION FOO
! ...
!omp$ simd private(X,Y)
DO I= 0, N
  Y = B(I)
  X = A(I)
  C(I) += FOO(X, Y)
ENDDO

```

10 参照 X と Y の直線性がコンパイル時に不明

```

REAL FUNCTION FOO(X, Y)
!$omp declare simd(FOO) linear(ref(X), ref(Y))
  REAL, VALUE :: Y      !! 参照渡し
  REAL, VALUE :: X      !! 参照渡し
  FOO = X + Y            !! ユニットストライド
                        !! SIMD ロード
END FUNCTION FOO
! ...
!omp$ simd private(X,Y)
DO I= 0, N
  Y = B(I)
  X = A(I)
  C(I) += FOO(X, Y)
ENDDO

```

11 X と Y の参照が linear として示されている

図 12 で、関数 `add_one` は SIMD 関数として示され、C++ の参照引数 `const int &p` を含んでいます。`p` が `linear(ref(p))` として示されている場合、コンパイラーはベクトル長を 4 と仮定し、`rax` レジスターのベースアドレス `p` でユニットストライド方式のロード命令を生成して、`p[0]`、`p[1]`、`p[2]`、`p[3]` を `xmm0` レジスターにロードします。この場合、`add_one` 関数で必要な命令は 3 つのみです。

```

#pragma omp declare simd notinbranch // linear(ref(p))
__declspec(noinline)
int add_one(const int& p) {
  return (p + 1);
}

```

12 `linear(ref(p))` アノテーションあり / なしの SIMD コードの比較

しかし、`p` が `linear(ref(p))` として示されていない場合、コンパイラーは 4 つの異なるアドレス `p0`、`p1`、`p2`、`p3` が 2 つの `xmm` レジスターで渡され、ギャザー操作がスカラーロードとパック命令のシーケンスでエミュレートされていると仮定します。その結果、`add_one` 関数に必要な命令は 3 つではなく 16 になります。

全体的に、OpenMP* バージョン 4.5 で追加された SIMD 機能を利用すると、ユーザーは、より詳細な情報をコンパイラーに提供できます。その結果、コンパイラーは、多くの状況で、より多くのループをベクトル化し、より優れたベクトルコードを生成します。

アフィニティー：スレッドの配置が簡単に

OpenMP* バージョン 4.0 仕様では、スレッド・アフィニティーを制御する標準的な方法が初めて提供され、言語に 2 つの新しい概念をもたらしました。

- 1. バインドポリシー
- 2. プレース・パーティション

バインドポリシー (`bind-var` 内部制御変数 (ICV) で指定) は、チームのスレッドがどこで親スレッドのプレースにバインドされるか決定します。プレース・パーティション (`place-partition-var` ICV で指定) は、スレッドをバインドできる場所のセットです。スレッドは、いったん指定されたチームのプレースにバインドされたら、そのプレースから移動すべきではありません。

仕様では、`master`、`close`、`spread` の 3 つのバインドポリシーが定義されています。これらのポリシーの説明では、4 つのプレース (それぞれ 1 コアと 2 スレッド) のセットについて考えます。これらのプレースに 3 スレッドおよび 6 スレッドを配置する例を示します。親スレッドは常に 3 目目のプレースに配置されると仮定します。`master` ポリシーでは、マスタースレッドは親スレッドのプレースにバインドされ、チームの残りのスレッドはマスタースレッドと同じプレースに割り当てられます (表 1)。

	プレース 1: {0,1}	プレース 2: {2,3}	プレース 3: {4,5} (親)	プレース 4: {6,7}
3 スレッド			0、1、2	
6 スレッド			0、1、2、3、4、5	

表 1. `master` ポリシーのスレッド配置

close ポリシーは、親スレッドのプレースにマスタースレッドを配置して開始した後、チームの残りのスレッドをラウンドロビン方式で処理します。**P** 個のプレースに **T** スレッドを配置するには、マスターのプレースで最初の **T/P** スレッドを取得した後、プレース・パーティションの次のプレースで次の **T/P** スレッドを取得し（以後同様、必要に応じてラップアラウンドされます）、スレッドを分散します (表 2)。

	プレース 1: {0,1}	プレース 2: {2,3}	プレース 3: {4,5} (親)	プレース 4: {6,7}
3 スレッド	2		0	1
6 スレッド	4	5	0,1	2,3

表 2. **close** ポリシーのスレッド配置

spread ポリシーでは、非常に興味深い処理になります。スレッドの配置は利用可能なプレースに拡散されます。プレース・パーティションのサブパーティション (**T >= P** の場合は **P** パーティション) で **T** がほぼ均等になるようにします。**T <= P** の場合、マスタースレッドから順に各スレッドは独自のサブパーティションを取得し、親スレッドがバインドされるプレースを含むサブパーティションを取得します。各後続スレッドは、各後続サブパーティションの最初のプレースにバインドされます (必要に応じてラップアラウンドされます)。**T > P** の場合、連続するスレッドのセットが同じサブパーティション (このケースでは 1 つのプレース) を取得します。そのため、セットのすべてのスレッドは同じプレースにバインドされます。中括弧形式のサブパーティションを表 3 に示します。入れ子の並列処理が使用される場合、各入れ子の並列領域で使用される利用可能なリソースに影響するため、これらは重要です。

	プレース 1: {0,1}	プレース 2: {2,3}	プレース 3: {4,5} (親)	プレース 4: {6,7}
3 スレッド	1 {{0,1}}	2 {{2,3}}	0 {{4,5},{6,7}} 注: 0 は {4,5} にバインドされる	
6 スレッド	4 {{0,1}}	5 {{2,3}}	0,1 {{4,5}}	2,3 {{6,7}}

表 3. **spread** ポリシーのスレッド配置とサブパーティション

OpenMP* バージョン 4.0 では、スレッド・アフィニティのバインドポリシーのクエリー関数 **omp_proc_bind_t omp_get_proc_bind()** も提供されました。この関数は、次の **parallel** 領域で使われるバインドポリシーを返します (その領域で **proc_bind** 節が指定されないと仮定します)。

spread ポリシーの興味深い点はサブパーティションで起こることです。**master** および **close** ポリシーでは、暗黙的なタスクはそれぞれ、親の暗黙的なタスクのプレース・パーティションを継承します。しかし、**spread** ポリシーでは、暗黙的なタスクは代わりにサブパーティションの **place-partition-var** ICV セットを取得します。これは、入れ子の **parallel** 構文ではすべてのスレッドが親のサブパーティション内に配置されることを意味します。

bind-var ICV の値は **OMP_PROC_BIND** 環境変数で初期化することができます。**bind-var** ICV の値は **proc_bind** 節を **parallel** 構文に追加してオーバーライドすることもできます。**place-partition-var** ICV の値は **OMP_PLACES** 環境変数で指定します。プレースは、ハードウェア・スレッド、コア、ソケット (特定数を含む) になります。明示的なプロセッサ・リストの場合もあります。詳細は、OpenMP* API 仕様を参照してください。

OpenMP* 4.5 仕様では、プレース・パーティションを照会して現在のスレッドのプレースをバインドする機能のセットが提供され、言語のアフィニティ機能が強化されました。これらの新しい API 関数は、希望するスレッド・アフィニティを達成するため、設定の正確さを確認するのに役立ちます。コードが複雑で入れ子の並列処理が **spread** バインドポリシーとともに使用され、低レベルキャッシュを共有する入れ子の並列領域にスレッドを配置する場合は特に重要です。次の API 関数があります。

- **int omp_get_num_places():** 初期タスクの実行環境で **place-partition-var** のプレースの数を返します。
- **int omp_get_place_num_procs(int place_num):** プレース・パーティションの **place_num** で指定されたプレースの実行環境で利用可能なプロセッサの数を返します。
- **void omp_get_place_proc_ids(int place_num, int *ids):** プレース・パーティションの **place_num** で指定されたプレースの実行環境で利用可能なプロセッサを取得し、それらを保持する配列を割り当てて、その配列を **ids** に格納します。
- **int omp_get_place_num(void):** 到達スレッドがバインドされるプレース・パーティションのプレースの数を返します。
- **int omp_get_partition_num_places(void):** 最も内側の暗黙的なタスクのプレース・パーティションのプレースの数を返します。常にオリジナルのプレース・パーティションを示す **omp_get_num_places()** とは異なり、プレース・パーティションをサブパーティションに分割する **spread** バインドポリシーの効果を示すことに注意してください。
- **void omp_get_partition_place_nums(int *place_nums):** 最も内側の暗黙的なタスクのプレース・パーティションに対応するプレース番号のリストを取得し、**place_nums** の配列を割り当てて格納します。プレース番号はオリジナルのプレース・パーティションのプレースの番号であることに注意してください。この関数は、**spread** バインドポリシーで生成されるサブパーティションに含まれるオリジナルのプレース・パーティションのプレースを確認する際に特に役立ちます。

OpenMP* API 仕様バージョン 5.0 の概要

OpenMP* ARB (Architecture Review Board) では、OpenMP* 仕様のバージョン 5.0 で追加される可能性のある機能について議論しています。最も可能性の高い候補をこのセクションで紹介します。

メモリー管理のサポート

OpenMP* アプリケーション内でますます複雑になるメモリー階層をサポートする方法は、活発に議論されています。この複雑さは、異なる特性の新しいメモリー (インテル® Xeon Phi™ プロセッサの MCDRAM やインテル® 3D XPoint™ メモリーなど)、優れたパフォーマンスを保証するため割り当てメモリーの特性に関する要求 (特定のアライメントやページサイズなど)、いくつかのメモリーに対する特別なコンパイラ・サポート、NUMA による影響など、さまざまな要因によるものです。さらに、多くの新しいメモリー・テクノロジーが調査中であるため、将来のテクノロジーに対応できるように拡張可能なようにする必要があります。

現在取り組んでいる方法は、異なるテクノロジーと操作をモデル化する 2 つの重要な概念 (メモリー空間およびアロケータ) に基づいています。メモリー空間は、プログラマーがプログラムで使用するメモリーを見つけるために指定できる、システムメモリーとメモリー特性のセット (ページサイズ、容量、帯域幅など) を表します。アロケータは、メモリー空間からメモリーを割り当てるオブジェクトで、動作 (アロケータのアライメントなど) を変更する特性を含むこともできます。

新しい API は、メモリー空間とアロケータの操作、およびメモリーの割り当てと割り当て解除を行うように定義されています。アロケータの作成とメモリーの割り当ての呼び出しを分けることにより、メモリーを割り当てる場所に関する決定が共通の「決定」モジュールで得られる、管理しやすいインターフェイスを構築できます。**図 13** は、2MB ページを使用してメモリーから異なる 2 つのアロケータ (割り当てが 64 バイトでアライメントされることを保証するアロケータと保証しないアロケータ) を定義するシステムで最も高い帯域幅のメモリーを選択する際に、どのように提案を使用するかを示しています。

```
omp_memtrait_set_t trait_set;
omp_memtrait_t traits[] = {{OMP_MTK_BANDWIDTH, OMP_MTK_HIGHEST},
                           {OMP_MTK_PAGESIZE, 2*1024*1024}};
omp_init_memtrait_set(&trait_set, 2, traits);

omp_memspace_t *amemspace = omp_init_memspace(&trait_set);

omp_alloctrail_t trait = {{OMP_ATK_ALIGNMENT}, {64}};
omp_alloctrail_set_t trait_set;
omp_init_alloctrail_set(&trait_set, 1, &trait);

omp_allocator_t *aligned_allocator = omp_init_allocator(amemspace,
                                                         &trait_set);
omp_allocator_t *unaligned_allocator = omp_init_allocator(amemspace, NULL);

double *a = (double *)omp_alloc( aligned_allocator, N * sizeof(double) );
double *b = (double *)omp_alloc( unaligned_allocator, N * sizeof(double) );
```

13 最も高い帯域幅のメモリーを選択

新しい **allocate** ディレクティブは、API 呼び出しで割り当てられない変数の割り当てを制御するために提案されたものです (自動変数または静的変数など)。新しい **allocate** 節は、OpenMP* ディレクティブによる割り当て操作に使用できます (変数のプライベート・コピーなど)。図 14 は、ディレクティブを使用して 2MB ページを使用する最も高い帯域幅のメモリに変数 **a** と **b** の割り当てを変更する方法を示しています。並列領域で **b** のプライベート・コピーは、レイテンシーが最も低いメモリに割り当てられます。

```
int a[N], b[M];
#pragma omp allocate(a,b) memtraits (bandwidth=highest, pagesize=2*1024*1024)

void example() {
#pragma omp parallel private(b) allocate(memtraits(latency=lowest):b)
{
    // ...
}
}
```

14 2MB ページを使用する最も高い帯域幅のメモリに変数 **a** と **b** の割り当てを変更

ヘテロジニアス・プログラミングの向上

OpenMP* のデバイスサポートを向上するために次のような機能が検討されています。

- 現在、**map** 節の構造は、構造のポインターフィールドを含めて、ビット単位でコピーされます。ポインターフィールドが有効なデバイスメモリを指すようにプログラマーが要求した場合、デバイス上にメモリを確保してデバイスのポインターフィールドを明示的に更新する必要があります。委員会は、構造のポインターフィールドのサポートを拡張することにより、プログラマーが **map** 節を使用して構造のポインターフィールドの自動割り当て / 割り当て解除を指定できるようにする拡張について議論しています。
- ARB は、関数ポインターを **target** 領域で使用できるようにすること、および関数ポインターを **declare target** に記述できるようにすることを検討しています。
- 非同期に実行できる新しいデバイス **memcpy** ルーチンのサポート。
- **target** 構文の「デバイスで実行または失敗」セマンティクスのサポート。現在、デバイスが利用できない場合、ターゲット領域はホストで実行されます。
- デバイスのみに存在し、ホストベースのコピーでない変数や関数のサポート。
- 単一アプリケーションでの複数のデバイスタイプのサポート。

タスクの改良

この記事で説明した機能以外に、OpenMP* 5.0 では次の機能が検討されています。

- **taskloop** 構文間のデータ依存関係の有効化。
- **task** 構文と **taskloop** 構文の両方でのデータ依存関係の有効化。1 つの **depend** 節からより多くの依存関係を生成する複数の値に拡張できる式を含む。
- **OMP_PROC_BIND** に似たパターンのスレッド・アフィニティーでのタスク表現のサポート。

その他の変更

OpenMP* 5.0 の追加機能として、次のような機能が議論されています。

- OpenMP* のベース言語の仕様を C11、C++11、C++14、Fortran 2008 にアップグレードする。
- 非矩形のループ形状、入れ子のループ間のコード記述がそれぞれ可能になるように、[collapse](#) 節の制限を緩和する。
- ワークシェアリング構造に関連付けられていない並列領域の内部でのリダクションを可能にする。

OpenMP* API は、ハイパフォーマンス・コンピューティングの C、C++、Fortran アプリケーションの共有メモリ並列化に適した、可搬性のあるベンダー・ニュートラルな並列プログラミング言語として不動の地位を確立しています。バージョン 5.0 の今後の開発では、開発者が最近のプロセッサの機能を最大限に活用できるように、さらに多くの機能が提供される予定です。

BLOG HIGHLIGHTS

コードの現代化 : CERN における科学的発見と全体的な革新の促進 (パート 2)

[RUSS BEUTLER \(INTEL\)](#) >

インテルは、CERN openlab CTO の Dr. Maria Girone と、CERN とインテルが協力して処理速度を向上する方法や CERN で行われている物質の基本成分の研究に与える影響について対談を行いました。この対談は、「モダンコード」アプローチが研究の進歩やブレイクスルーにどのように役立つか、分かりやすくプログラマーに説明することが目的でした。

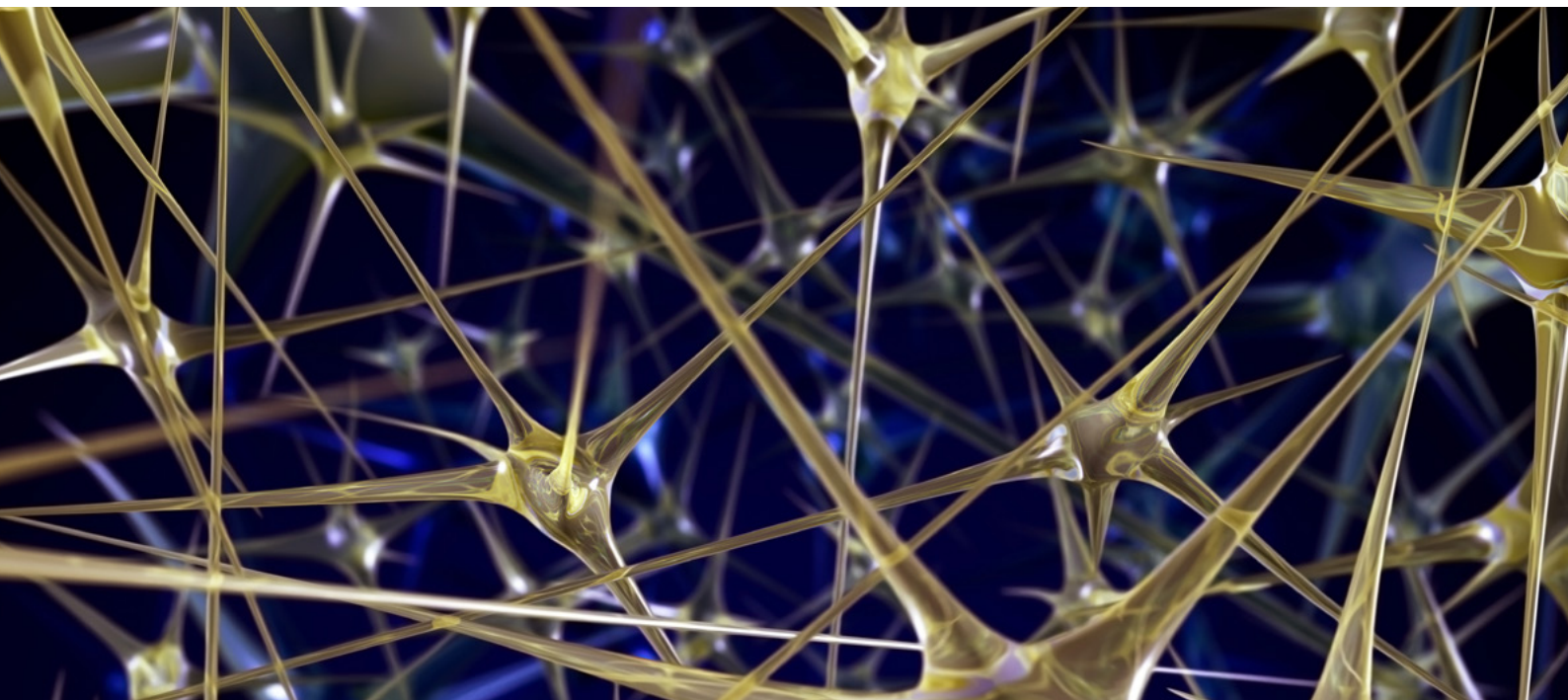
対談の第 2 部で、Dr. Maria は、コードの現代化戦略に関する開発者へのアドバイスを述べ、開発者と学生がスキルを伸ばし、キャリアをアップさせて、取り組んでいるアプリケーションに大規模な改良をもたらすことができるプログラムについて紹介しました。

コードの現代化戦略を構築し、アプリケーションが最近のサーバー・ハードウェアを確実に活用することを求めている開発者と企業に対するアドバイスを聞かせてください。

まず、古いコードを利用している場合、大幅な改良の余地があると認識することが重要です。これは、古いコードの更新が得意な開発者が職を得られる機会があることを意味します。

具体的には、レガシー・ソフトウェアと現在のパフォーマンス標準との差、および効率的な並列化とベクトル化により達成可能なゲインを理解する必要があります。次に、コードの現代化に取り組んだ場合に達成できるパフォーマンスの向上を分かりやすく示すことが重要です。

[この記事の続きはこちら \(英語\) でご覧になれます。>](#)



行列 - 行列乗算のパックのオーバーヘッドを減らす

マルチコア / メニーコアのインテル® アーキテクチャーにおけるディープ・ニューラル・ネットワークのパフォーマンスの向上

Kazushige Goto、Murat Efe Guney、Sarah Knepper インテル コーポレーション ソフトウェア開発エンジニア

汎用行列 - 行列乗算 (GEMM) は、多くの科学、工学、マシンラーニング・アプリケーションで利用される基本操作であり、BLAS (Basic Linear Algebra Subprograms) ドメインの重要なルーチンの 1 つです。GEMM には 4 つの精度 (実数単精度、実数倍精度、複素数単精度、複素数倍精度) があります。この記事では、SGEMM (実数単精度) について取り上げます。

SGEMM の Fortran API は次のとおりです。

`SGEMM(transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)`

この API は次の計算を実行します。

$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * \text{op}(C)$

($\text{op}(X)$ は X または X^T 、`transx` パラメーターの値に依存)

配列 A と B は入力で、C は入力および出力です。配列 A は $m \times k$ 行列、配列 B は $k \times n$ 行列、配列 C は $m \times n$ 行列をそれぞれ含みます。リーディング・ディメンジョン (`lda`、`ldb`、`ldc`) は、GEMM が大きな行列の一部を処理できるように、ある列から次の列へのストライドを決定します。リーディング・ディメンジョンは、後続列をキャッシュラインにアライメントしたり、8 ウェイ 1 次キャッシュの同じセットにマップすると、パフォーマンスにも影響します。

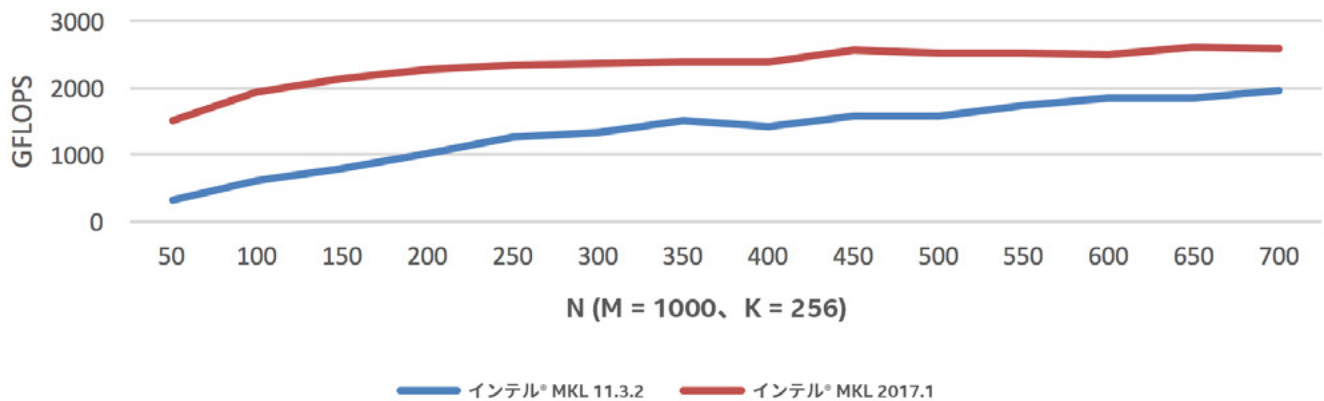
インテル® マス・カーネル・ライブラリー (インテル® MKL) は、ハイパフォーマンスな GEMM 実装を提供します。行列 - 行列乗算を最適化する一般的なアプローチは、オリジナルの入力行列のブロックを内部データ形式 (パックド形式など) に変換して、手書きのアセンブリ・カーネルで変換されたブロックを乗算した後、出力行列を更新します。¹ ブロックサイズはキャッシュとレジスターの使用率が最大になるように選択されます。パックする理由はさまざまです。

- A および B から多くのデータをキャッシュに格納できる (ブロックを大きくするとより多くのデータを再利用できる)。
- 連続し、アライメントされた、予測可能なアクセス (キャッシュおよびデータ・トランスレーション・ルックアサイド・バッファ (DTLB) のミスを最小限に抑えることができる)。
- ループのオーバーヘッドの軽減。

ハイパフォーマンス・コンピューティングの従来のサイズでは、このパックベースのアプローチは適切に動作します。一般に、 m および n は比較的大きく、 k は中程度 (外積) または同様に比較的大きい (正方) ため、入力行列のパックで費やされる時間は計算カーネルで費やされる時間に比べるとわずかです。しかし、一部のマシンラーニング・アプリケーションで一般的な、 m または n のサイズが比較的小さい場合は、パックのオーバーヘッドが大きく影響します。その結果、明示的なパックに依存しない GEMM 実装のほうが、従来のパックベースの GEMM 実装よりもパフォーマンスが高くなる場合があります。インテル® MKL 11.3 Update 3 には、**インテル® アドバンスド・ベクトル・エクステンション 2 (インテル® AVX2)**、インテル® アドバンスド・ベクトル・エクステンション 512 (インテル® AVX-512)、第 2 世代 **インテル® Xeon Phi™** プロセッサ向けに、これらのいくつかのサイズを最適化した {S,D}GEMM カーネルが含まれています。以降のインテル® MKL バージョンでも、これらのカーネルは引き続き改良されています。

これらの新しいカーネルが有効な例を示すため、**図 1** では、マシンラーニングで利用されるサイズで、インテル® MKL 2017 Update 1 とインテル® MKL 11.3 Update 2 の SGEMM のパフォーマンスを比較しています。ここで、 m と k はそれぞれ 1000 と 256 に固定されていますが、 n は可変です。パフォーマンスは GFLOPS (1 秒間に 10 億回の浮動小数点演算を実行) で表されるため、数値が高いほうが優れています。

小さなサイズにおけるインテル® MKL の SGEMM パフォーマンスの向上 M = 1000、N 多様、K = 256



構成情報 – バージョン: インテル® MKL 11.3.2 およびインテル® MKL 2017.1。ハードウェア: インテル® Xeon® プロセッサ E5-2699 v4、2 x 22 コア CPU (55MB LLC、2.20GHz)、64GB RAM。オペレーティング・システム: RHEL 7.2 GA x86_64。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサ用に最適化されていることがあります。SYSmark® や MobileMark® などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。ベンチマーク出典: インテル コーポレーション

最適化に関する注意事項: インテル® コンパイラーでは、インテル® マイクロプロセッサに限定されない最適化に関して、他社製マイクロプロセッサ用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサ依存の最適化は、インテル® マイクロプロセッサでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサ用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。注意事項の改訂 #20110804

1 小さなサイズにおけるインテル® MKL の SGEMM パフォーマンスの向上

SGEMM 呼び出しの情報 (プロセッサの種類とスレッド数、問題サイズとリーディング・ディメンジョン、転置パラメーターを含む) に基づいて、インテル® MKL は従来のカーネルと新しいパックフリー・カーネルのどちらかを使用するか決定します。インテル® MKL の SGEMM パフォーマンスを利用するディープラーニング・フレームワークは、フレームワークの修正を行うことなく、これらの最適化の恩恵を受けることができます。

例えば、再帰型ニューラル・ネットワークのように入力行列 (A または B) が複数の行列乗算で再利用される場合、別の方法でパックのオーバーヘッドを最小限に抑えます。この方法では、再利用される行列を 1 回パックし、複数の SGEMM 計算でパックドバージョンを使用します。インテル® MKL 2017 では、複数の行列乗算でパックのオーバーヘッドを相殺できる、{S,D}GEMM のパックド API が追加されました。単精度の場合、新しいパックド API は次の 4 つです。

```
dest = sgemm_alloc (identifier, m, n, k)
```

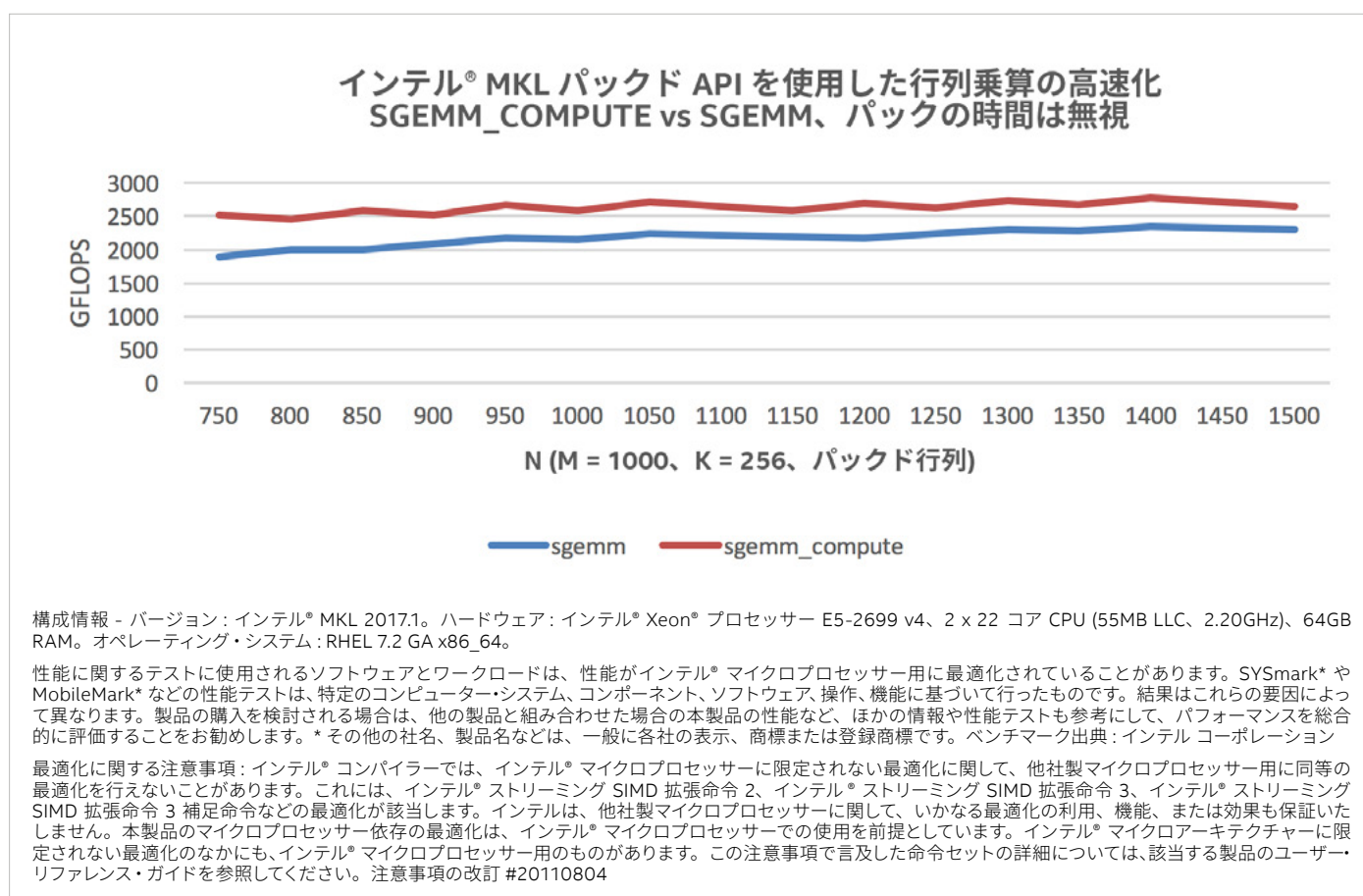
```
sgemm_pack (identifier, trans, m, n, k, alpha, src, ld, dest)
```

```
sgemm_compute (transa, transb, m, n, k, A, lda, B, ldb, beta, C, ldc)
```

```
sgemm_free (dest)
```

パラメーターは SGEMM とほぼ同じで、パックする行列 (A または B) を示すパラメーター **identifier** が追加されています。**sgemm_compute** の **transa** および **transb** パラメーターは、通常どおり、“T” (転置する) または “N” (転置しない) に設定します。値 **P** は、対応する行列が内部パックド形式であることを示します。パックド API は、入力行列が複数回使用される場合に恩恵が得られます。そのため、**sgemm_alloc** と **sgemm_pack** をそれぞれ呼び出して、メモリーの割り当てと行列のパックを行い内部パックド形式にした後、**sgemm_compute** を複数回呼び出して、パックド行列を入力行列の 1 つとして渡します。最後に、**sgemm_free** を呼び出してメモリーを解放します。(詳細は、[インテル® MKL デベロッパー・リファレンス](#) (英語) を参照してください。)

図 2 は、**sgemm** と **sgemm_compute** のパフォーマンスの比較を示しています (A 行列はパックド形式、パックの時間は無視)。



2 インテル® MKL パックド API を使用した行列乗算の高速化

GEMM 呼び出し間で大きな A および B 入力行列を再利用するアプリケーションは、パックド API の恩恵を最も得られる可能性があります。m と n パラメーターがともに大きな場合、あるいは 2 つの小さな入力行列を再利用する場合は、パックド API に変更して得られるパフォーマンスの向上がプログラミングの労力と釣り合わない可能性があります。このため、アプリケーションのコードを変更してパックド API を使用する前に、特定のユースケースでのパフォーマンスを測定することを推奨します。

これまで説明したように、特に 1 つ以上の行列のサイズが小さい場合、従来の GEMM 実装で行われている内部パック操作では大きなオーバーヘッドが発生します。このオーバーヘッドは、入力行列を明示的にパックしないカーネルを使用するか、複数の GEMM 計算でパックのコストを相殺することにより減らすことができます。インテル® MKL 11.3 Update 3 から、{S,D}GEMM は、最初に内部バッファにパックしないで入力行列を直接操作するカーネルを選択できるようになりました。使用するカーネルは、問題の特性とプロセッサ情報に基づいて実行時に決定されます。別の方法として、インテル® MKL 2017 では、入力行列の 1 つまたは両方を明示的にパックした後、複数の行列 - 行列乗算で再利用できるパックド API が追加されました。これらの 2 つのアプローチは、特にディープ・ニューラル・ネットワークで利用されるサイズを利用するときに、マルチコア / メニーコアのインテル® アーキテクチャーで高い GEMM パフォーマンスを達成するのに役立ちます。

1. Kazushige Goto and Robert A. van de Geijn. 2008. "Anatomy of High-Performance Matrix Multiplication." ACM Transactions on Mathematical Software, 34, 3, Article 12, May 2008.

インテル® MKL を評価する >

並列アプリケーションのスケラビリティの問題を特定する

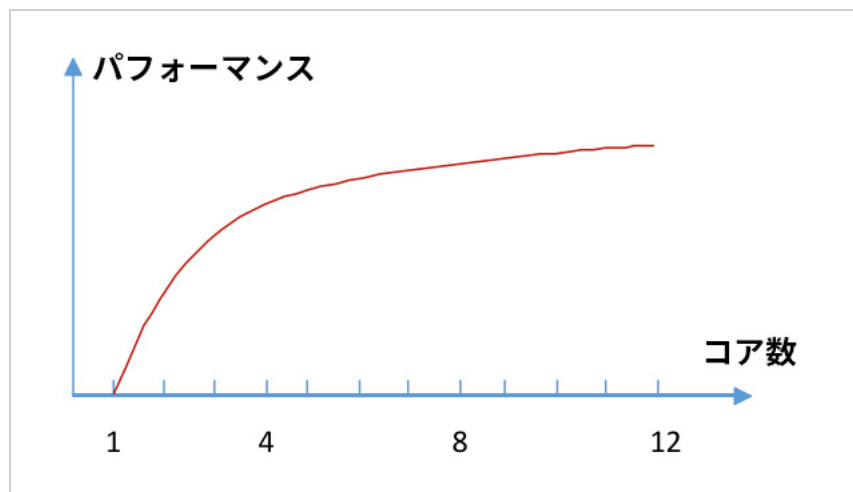
インテル® VTune™ Amplifier XE のメモリ解析を使用してインテル® Xeon® プロセッサおよびインテル® Xeon Phi™ プロセッサ向けにスケラビリティを向上する方法

Vladimir Tsybal インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

システムの計算コア数が増え続ける中、効率良く並列化されたソフトウェアでは、コア数の増加に伴ってパフォーマンスが直線的に向上することが望まれます。しかし、いくつかの要因により、マルチコアシステムの並列性とスケラビリティが妨げられることが分かっています。この記事では、ほとんどのケースで原因となる、並列処理の実装について取り上げます。

- **ロードインバランス**により、スレッドと CPU コアがアイドル状態になります。
- **過度の同期**により、スピン待機やその他の非生産的なワークに CPU 時間が費やされます。
- **並列ランタイム・ライブラリーのオーバーヘッド**が、適切でないライブラリー API の使用により発生します。

これらの要因を排除することで、並列処理の効率が大幅に向上し、すべての CPU コアが有用なワークを実行するためビジー状態になります。STREAM や LINPACK のような最適なチューニングが行われているベンチマークでは、ほぼ直線的なスピードアップを達成できます。しかし、システムのコア数の増加に伴い（または、コア数の多い新しいシステムでコードを実行した場合に）、アプリケーションのパフォーマンスが直線的に向上しなかったり、並列パフォーマンスが伸び悩むことがあります (図 1)。



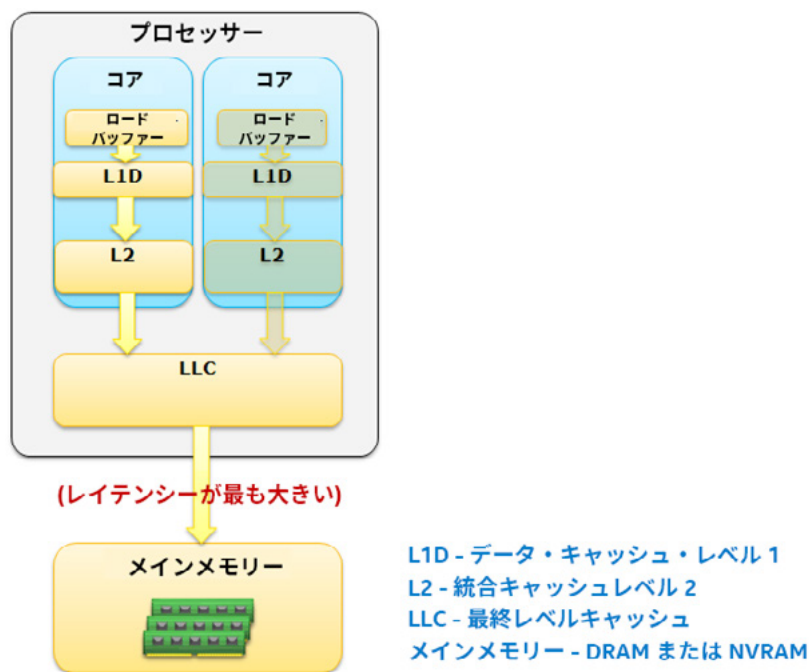
1 コア数とパフォーマンスの変化

トップダウンのパフォーマンス解析アプローチ¹では、最初にほかのコンポーネントによりパフォーマンスが妨げられていないかチェックすべきです。次のことを確認します。

- **ほかのプロセスによってシステムが常にビジー状態でない**：リソースを消費するほかのアプリケーションやサービスに計算時間が費やされていることがあります。
- **アプリケーションがシステムの I/O に依存していない**：例えば、ディスク、ほかのファイルシステム、ネットワーク・システムの処理が完了するのを待っていないか確認します。
- **システムに十分な物理メモリー容量がある**：ハードディスク・ドライブとのスワップが頻発するのを防ぎます。

一般に、ハードウェアが適切に構成され、メモリー・サブシステムが想定するパフォーマンス特性を提供しているかどうか把握しておくといいでしょう。例えば、すべてのメモリースロットにマザーボードの特性（チャンネル数、メモリー速度など）に対応した DIMM が搭載されている場合、一般的なベンチマークを使用してハードウェアのパフォーマンスを簡単にチェックできます。問題の修正は、ソフトウェアを最適化するよりもハードウェアで行ったほうが容易なため、このようなチェックを行うことが重要です。

これらのチェックが終わったら、並列処理がスケーリングしない主な原因の 1 つであるメモリー・レイテンシーを確認します。x86 システム・アーキテクチャーでは、CPU はキャッシュ・サブシステムからデータを読み取ります。命令が必要ときに、CPU に最も近いキャッシュ (L1 データキャッシュ) にデータがあることが理想的です (図 2)。要求されたデータが CPU から離れた場所にある場合、データが CPU コアの実行ユニットに到達するのに長い時間がかかります。CPU ハードウェア・プリフェッチャーは、迅速にデータを取得できるように支援しますが、常に適切に動作するとは限りません。データの遅延はよくあることで、CPU のストールを引き起こします。²



基本的に、データが遅延する理由は 2 つあります。

1. CPU の実行ユニットで実行される命令がデータを要求し、プリフェッチャーが適切に動作しない場合、メインメモリーやほかのキャッシュから CPU の L1D にデータが読み込まれます。これにより、メモリー・レイテンシーの問題が生じます。
2. プリフェッチャーが適切に動作し、データが事前に要求されても、転送インフラストラクチャーの処理能力が原因で CPU への転送が遅れることがあります。この場合、メモリー帯域幅の問題が生じます。

2 メモリー・サブシステムからのデータの読み取り

複数のソースから複数の要求があった場合、両方の問題が発生する可能性があります。これらの問題を回避するには、データを上手く利用することが重要です。メモリー・レイテンシーの問題を解決するには、アドレスを使用してインクリメンタルにデータにアクセスします。シーケンシャル・データ・アクセス (または一定の短い距離のユニットストライド) を使用すると、プリフェッチャーによって適切に処理され、迅速にデータにアクセスできます。メモリー帯域幅の問題を解決するには、データを再利用し、できる限りキャッシュに保持するようにします。そのためには、データ・アクセス・パターンの見直しや、場合によってはアルゴリズム全体の実装の見直しが必要になります。

アプリケーションのスケーラビリティを妨げる要因

CPU でコードが効率良く実行されず、ほとんどのストールがメモリー依存であることが分かったら、問題の具体的な原因を明らかにする必要があります。問題への対応方法は、原因がメモリー・レイテンシーによるものか、メモリー帯域幅によるものかに応じて異なります。ここでは、**インテル® VTune™ Amplifier XE** の組込みメモリーアクセス解析を使用して、メモリーの問題を詳しく調査します。

単純な行列乗算ベンチマークを向上するいくつかの反復について考えてみます。この例は単純ですが、アルゴリズムの実装方法に応じてメモリーの問題が発生する可能性を効果的に示しています。パフォーマンスの測定には、**インテル® Xeon® プロセッサー E5-2697 v4** (開発コード名 Broadwell、36 コア) ベースのシステムを使用します。このシステムの理論メモリー帯域幅は 76.8GB/ 秒で、1 秒あたりの倍精度 (DP) 浮動小数点演算の回数 (FLOPS) は 662GFLOPS/ 秒です。

行列乗算アルゴリズムのナイーブ実装

ナイーブ行列乗算実装 (図 3 の multiply1) は、CPU コア数が多い場合は直線的にスケーリングしません。しかし、パフォーマンスが悪い原因の特定方法を示す良い例です。ここでは、**ベクトル化**が行われるように、`-no-alias` コンパイラー・オプションを追加します。そうしないと、スカラー実装のパフォーマンスは約 1/10 に低下します。サイズ 9216 x 9216 でベクトル化されているベンチマーク (multiply1) を実行した結果が表 1 です。最高のパフォーマンスが理論最大 FLOPS を大きく下回っていることが分かります。

```
void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE c[][NUM], TYPE t[
[NUM])
{
    int i,j,k;

    // ループ交換
    for(i=tidx; i<msize; i=i+numt) {
        for(k=0; k<msize; k++) {
            #pragma ivdep
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}
```

3 行列乗算アルゴリズムの最適化された実装 (multiply2)

スレッド数	実行時間 (秒)	DP FLOPS (GFLOPS/ 秒)
4	208	7.7
8	102	15.1
16	59	26.8
36	42	37.8
72 HT	24	66.1

表 1. ナイブ行列実装のパフォーマンスとスケーリング (36 コア、インテル® Xeon® プロセッサー E5-2697 v4、2 ソケット @ 2300MHz)

表 1 に示すように、並列ベンチマークは、スレッド数の増加に伴ってほぼ直線的にスケーリングしています。そして、30 コアに達すると横ばいになります。表 1 のデータは、multiply1 ベンチマークのパフォーマンスとスケーリングについて誤った認識を招く恐れがあります。ベンチマークがマシンの計算能力をどれぐらい使用しているか理解することは非常に重要です。この例では、(ベンチマークで測定された) FLOPS がマシンの理論値を大きく下回っています (約 1/10)。並列処理のスケーラビリティではなく、シリアル・パフォーマンスが制限されています。インテル® VTune™ Amplifier XE は、ループ内のコード実行が効率的でないことを示しています (図 4)。リタイアした命令数が低く、CPI レートが高いことから、実際の制限からどれぐらいかけ離れているのか予測できます。

S. Li. ▲	Source	★			☒	☒	☒	☒
		Cloc...	Inst... Ret...	CPI Rate	Front-... Bound	Bad Specu...	Back-... Bound	Retiring
66	void multiply1(int msize, int tid, int numt, TYPE a[][NUM])							
67	{							
68	int i,j,k;							
69								
70	// Naive implementation							
71	for(i=tid; i<msize; i=i+numt) {							
72	for(j=0; j<msize; j++) {	38,488...	35,...	1.077	1.1%	2.2%	76.6%	20.1%
73	for(k=0; k<msize; k++) {	71,300...	69,...	1.033	0.0%	0.0%	100.0%	14.7%
74	c[i][j] = c[i][j] + a[i][k] * b[k][j];	1,867,...	903...	2.067	0.7%	0.0%	87.1%	12.1%
75	}							
76	}							
77	}							
78	}							

4 ナイブ並列行列乗算ベンチマークのパフォーマンス

次に、行列乗算アルゴリズムの最適化された実装 (図 3 の multiply2) について見てみます。アルゴリズムが単純で、コンパイラーが理解できれば、インデックス・ストライドの効率が悪いことを認識し、ループ交換したバージョンを自動的に生成します (あるいは手動で生成できます)。

スレッド数	実行時間 (秒)	DP FLOPS (GFLOPS/ 秒)
4	208.8	7.8
8	103.3	15.1
16	58.8	26.4
36	38.4	40.5
72 HT	24.7	63.0

表 2. 最適化された行列乗算のパフォーマンスとスケーリング (36 コア、インテル® Xeon® プロセッサー E5-2697 v4、2 ソケット @ 2300MHz)

表 2 から、絶対数はやや向上しましたが、まだ理想からはかけ離れていることが分かります。

パフォーマンスとスケーラビリティを妨げている要因について考えてみましょう。General Exploration (全般解析) プロファイルの結果 (図 5) は、CPU マイクロアーキテクチャー向けの別のトップダウン解析アプローチです。³ いくつかの興味深いことが分かります。

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。

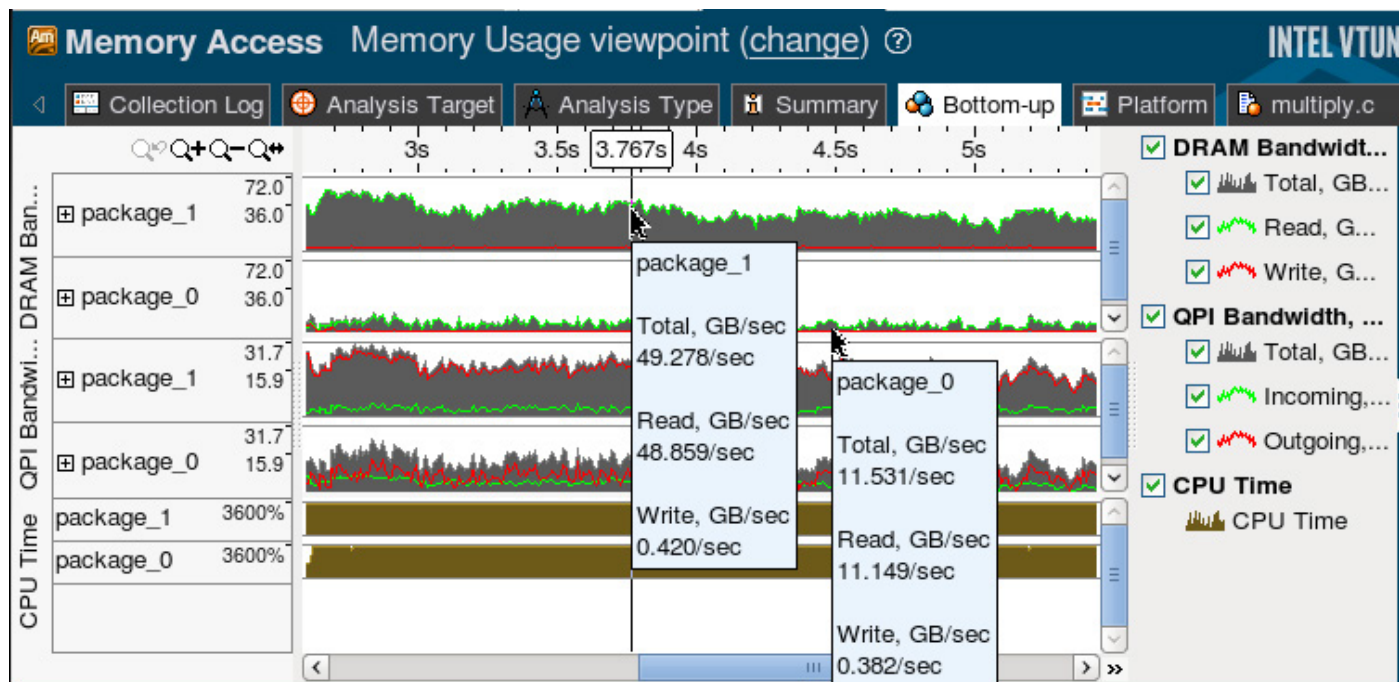
Function / Call Stack	Clockticks ▼	Instructions Retired	Front-End Bound	Bad Specul...	Back-End Bound	
					Memory Bound	Core Bound
▶ multiply2	3,536,756.00...	1,160,060.2...	5.6%	0.1%	71.8%	14.3%

Back-End Bound								
Memory Bound								
L1 Bound	L2 Bound	L3 Bound	Memory Bandwidth	Memory Latency			Store Bound	
				Local DRAM	Remote DRAM	Remote Cache		
7.9%	0.0%	33.3%	82.2%	29.0%	41.9%	0.0%	0.1%	

5 全般解析プロファイルの結果

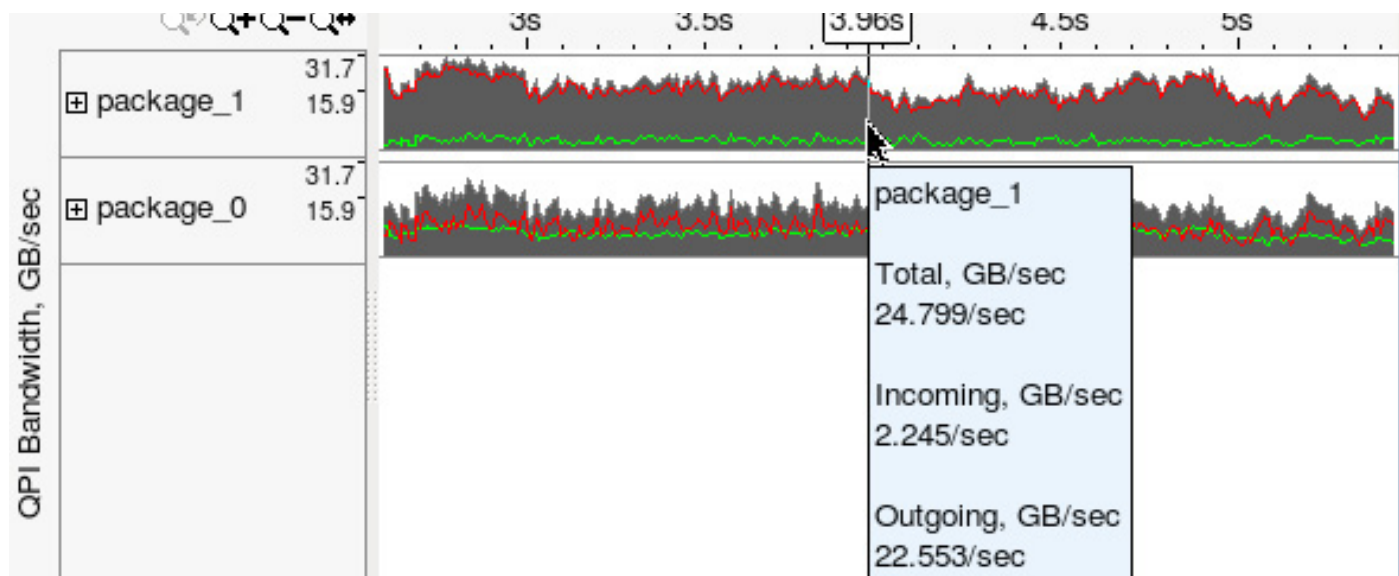
最初に、DRAM から CPU へのデータの読み取りメモリー・レイテンシーが減っています。アルゴリズムで連続するアドレスへのアクセスを実装しているため、これは想定内です。しかし、メモリー帯域幅メトリックが非常に高くなっているため、DRAM コントローラーと**インテル® QuickPath インターコネクト (インテル® QPI) (英語)** がボトルネックになっていないか、メイン・データ・パスの帯域幅を確認すべきです。2 つ目に、データアクセスが連続したパターンにもかかわらず、L3 レイテンシーも大きくなっていることが分かります。これは調査する必要があります。L3 レイテンシーが大きいということは、L2 ミスが頻繁に発生しているということです。連続したアクセスで DRAM レイテンシーが減っていないことから、ハードウェア L2 プリフェッチャーは適切に動作しているはずであり、これは奇妙です。3 つ目に、リモート DRAM レイテンシーが大きいことが分かります。これは、不均等メモリーアクセス (NUMA) による影響で、各ノードに対してデータの一部がリモート DRAM からフェッチされることを示しています。データ転送の全体像を掴むには、ソケット間の DRAM メモリー・コントローラーとインテル® QPI バスのデータ・トラフィックを測定する必要があります。これには、インテル® VTune™ Amplifier XE のメモリー・アクセス・プロファイルを使用します。

図 6 は、72 スレッドでプロファイルした結果です。1 つの DRAM コントローラー (package_1) のみでデータがロードされ、平均データ転送速度は約 50GB/ 秒、つまり最大帯域幅の約 2/3 であることが分かります。package_0 のメモリー・コントローラー上にはほとんどトラフィックがありません。



6 72 スレッドで multiply2 のメモリー・アクセス・プロファイルを収集

同じ期間のインテル® QPI 出力レーンのデータ・トラフィックの半分は package_1 が占めています。これで、package_1 DRAM から package_0 CPU コアへのデータの流れの説明がつきます (図 7)。このインテル® QPI のクロストラフィックが原因で、プリフェッチャーによってリモート DRAM からフェッチされるデータと CPU コアによってリモート LLC からフェッチされるデータには、追加のレイテンシーがかかります。ベンチマークでは、データが適切に構造化されており、スレッド間で均等に分配されるため、NUMA による影響を簡単に排除できます。スレッド・アフィニティーを CPU コアに変更し、各スレッドで a、b、c 行列を初期化するだけです。ただし、各スレッドでデータを割り当てることで NUMA による影響がすべて排除されるとは限りません。



7 インテル® QPI のクロストラフィック

例えば、**図 8** はパフォーマンスが向上しない例です。インテル® VTune™ Amplifier XE でこの問題を検出できます。ベンチマークのソースコードに、列挙された CPU にスレッドをピンングする関数を追加します。**図 8** は、コードの一部です。

```
CreateThreadPool( ... )
{
    pthread_t ht[NTHREADS];
    pthread_attr_t attr;
    cpu_set_t cpus;
    pthread_attr_init(&attr);

    for (tidx=0; tidx<NTHREADS; tidx++) {
        CPU_ZERO(&cpus);
        CPU_SET(tidx, &cpus);
        pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpus);
        pthread_create( &ht[tidx], &attr, (void*)start_routine, (void*) &par[tidx]);
    }
    for (tidx=0; tidx<NTHREADS; tidx++)
        pthread_join(ht[tidx], (void **)&status);
}
```

8 列挙された CPU にスレッドをピンング

データ初期化関数では、配列は乗算関数で乗算される順序でスレッド間に分配されるべきです。**図 9** は、NUMA 対応を容易にするため、コードを変更した後の関数です。初期化関数では、データ配列は行列のサイズをスレッド数で割った、サイズ **msize/numt** のチャンクに分割されます。**図 10** に示すように、乗算関数でも同様の処理を行います。驚くべきことに、ベンチマークの実行時間は、NUMA 対応でないバージョンとあまり変わりありません。インテル® VTune™ Amplifier XE のメモリー・アクセス・プロファイルで解析してみましょう (**図 11**)。

```
InitMatrixArrays (int msize, int tidx, int numt, ... )
{
    int i,j,k,ibeg,ibound,istep;
    istep = msize / numt;
    ibeg = tidx * istep;
    ibound = ibeg + istep;

    for(i=ibeg; i<ibound; i++) {
        for (j=0; j<msize;j++) {
            a[i][j] = 1.0*i+2.0*j+3.0;
            b[i][j] = 2.0*i+1.0*j+3.0;
            c[i][j] = 0.0;
        }
    }
}
```

9 NUMA 対応の単純化

```

multiply2(int msize, int tid, int numt, ... )
{
    int i,j,k,ibeg,ibound,istep;
    istep = msize / numt;
    ibeg = tid * istep;
    ibound = ibeg + istep;

    for(i=ibeg; i<ibound; i++) {
        for(k=0; k<msize; k++) {
            for(j=0; j<msize; j++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
}

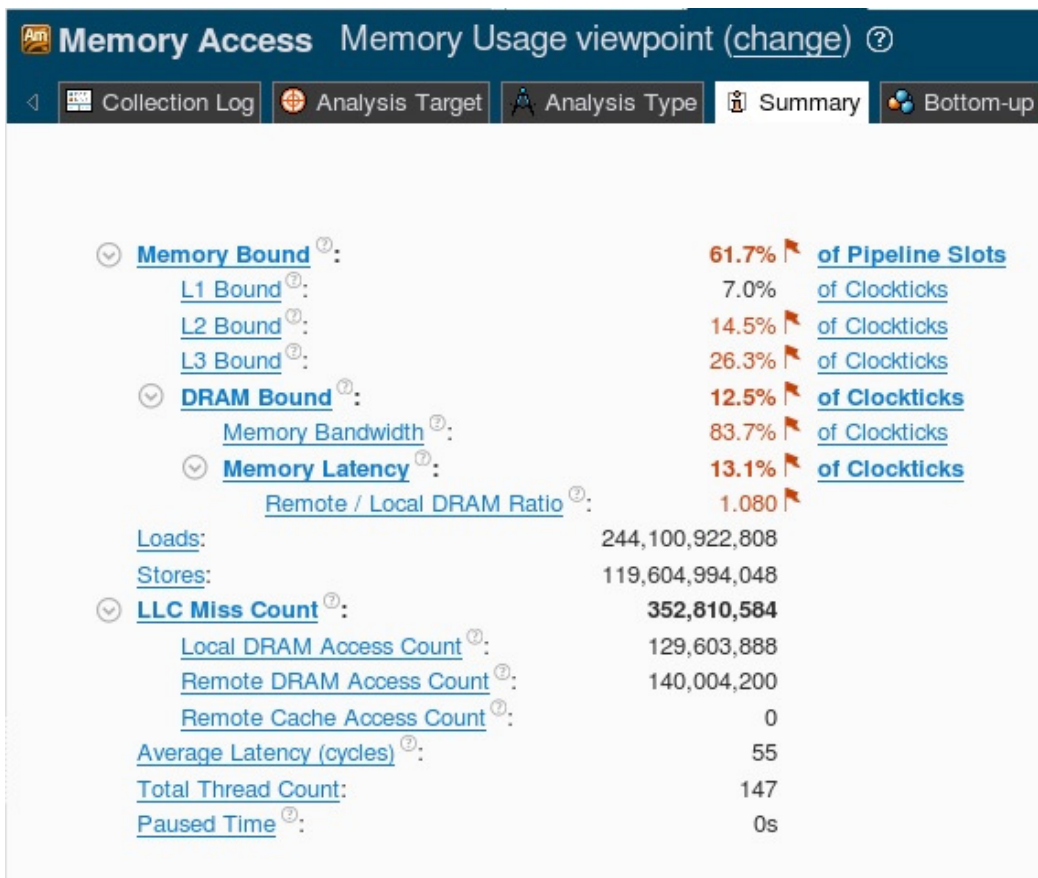
```

```

Threads #: 72 Pthreads
Matrix size: 9216
Using multiply kernel: multiply2
Freq = 2.30100 GHz
Execution time = 20.162 seconds
MFLOPS: 72826.877 mflops

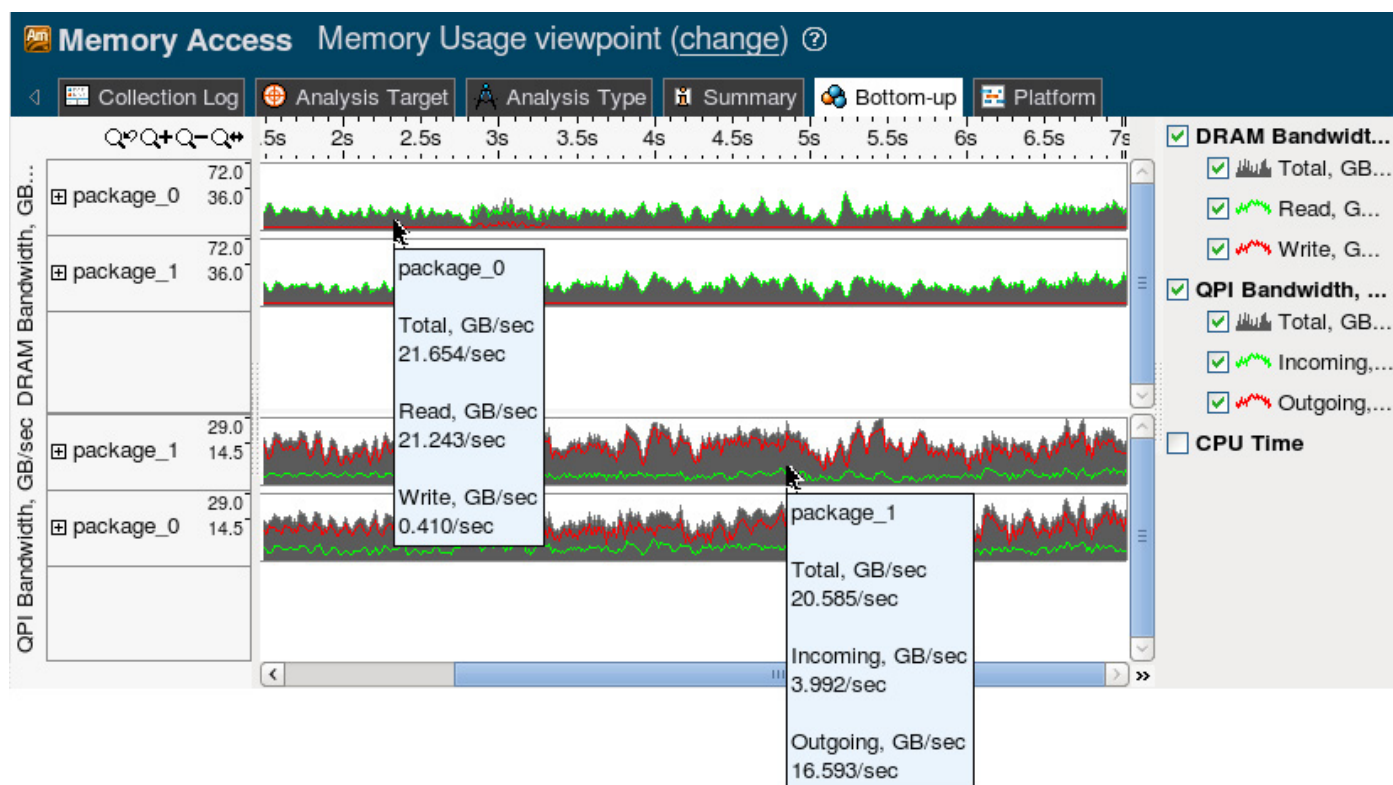
```

10 乗算関数



11 メモリー・アクセス・プロファイル

[Summary] ページは、アプリケーションがまだメモリー依存であることを示しています (メモリーからのデータ・レイテンシーとデータ・トラフィックによるストールがあります)。ただし、レイテンシーのほとんどは LLC によるもので、DRAM によるものはわずからです。ローカルアクセスとリモートアクセスの比率が非常に高いことも分かります。これは、NUMA 対応のアプローチが適していないことを意味します。DRAM コントローラーとインテル® QPI のトラフィックのタイムライン (図 12) を確認すると、DRAM からのデータストリームがピーク帯域幅の 30% に達することはほとんどなく、一方でインテル® QPI の帯域幅は各方向で約 90% に達しており、飽和状態といえます (このシステムにおけるインテル® QPI の実際的な制限は 29.2GB/ 秒です)。



12 DRAM コントロールとインテル® QPI のトラフィックをタイムラインでチェック

リモートアクセス (DRAM または LLC) は、メモリーブロック読み取りレイテンシーと CPU ストールを増加させます。インテル® VTune™ Amplifier XE のメモリーアクセスでこれらのレイテンシーを測定することで、非効率的なリモートアクセスのデータ (行列) を特定できます。メモリー解析サマリー (図 13) から、レイテンシーの大きいメモリー・オブジェクトが分かります。

Top Memory Objects by Latency

This section lists memory objects that introduced the highest latency to the overall application execution.

Memory Object	Total Latency	Loads	Stores	LLC Miss Count [?]
_mm_malloc (648 MB)	63.5%	112,553,776,512	25,600,384	339,210,176
_mm_malloc (648 MB)	36.2%	129,952,698,464	118,683,380,224	8,000,240
[Unknown]	0.2%	1,149,634,488	606,409,096	5,600,168
[Stack]	0.0%	120,803,624	201,603,024	0
_mm_malloc (648 MB)	0.0%	276,808,304	88,001,320	0
[Others]	0.0%	47,201,416	0	0

13 レイテンシー別上位メモリー・オブジェクト

上位 3 つのメモリー・オブジェクト (割り当て関数によって表現される) のうち、1 つのレイテンシーが最も大きく、多くのロード操作を行っていることが明らかです (図 14)。平均レイテンシーから、LLC のリモート DRAM からデータを読み取っているのは、1 つのオブジェクトのみであることが分かります。これは、[Remote DRAM Access] 列の値によって裏付けることができます。

Grouping: Memory Object / Function / Allocation Stack				
Memory Object / Function / Allocation Stack	Loads	Stores	LLC Miss Count	
			Local DRAM Acces...	Remote DRAM Access Count
▶ [Unknown]	1,149,634,488	606,409,096	3,200,096	3,200,096
▶ _mm_malloc (648 MB)	129,952,698,464	118,683,380,224	8,800,264	0
▶ _mm_malloc (648 MB)	112,553,776,512	25,600,384	117,603,528	136,804,104
▶ _mm_malloc (648 MB)	276,808,304	88,001,320	0	0

14 割り当て関数別メモリー・オブジェクト

この 3 つのオブジェクトが a、b、c 行列であることは明らかです。[Stores] の値が最も大きいのは行列 c です。レイテンシーが大きい行列データを特定するには、インテル® VTune™ Amplifier XE の [Stack] ペイン (図 15) でメモリー・オブジェクトのスタックを確認する必要があります。ユーザーコードのスタックを確認して、インテル® VTune™ Amplifier XE の [Source] ビューでデータ割り当てのソース行にドリルダウンできます (図 16)。この例では、レイテンシーとロード操作の増加の原因は行列 b のデータにあります。ピンングしたスレッド内でデータ配列の割り当てと初期化を行っているにもかかわらず、これらが増加している原因を理解する必要があります。

Accesses (Memory Allocation)

Viewing 1 of 1 selected stack(s)

100.0% (41607124905 of 41607124905)

matrix.iccl! [_mm_malloc](#) - [unknown source file]

matrix.iccl![main](#)+0x55 - matrix.c:106

matrix.iccl! [start](#)+0x28 - [unknown source file]

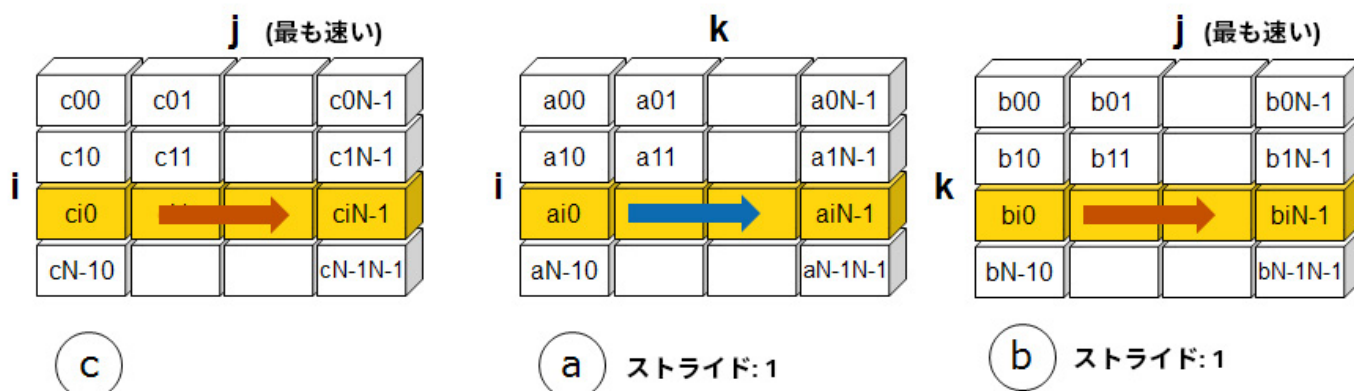
15 [Stack] ペインのメモリー・オブジェクト

Source Assembly Assembly grouping:

S. Li. ^	Source
98	#else
99	buf1 = _aligned_malloc((sizeof (double))*NUM*NUM, 64);
100	buf2 = _aligned_malloc((sizeof (double))*NUM*NUM, 64);
101	buf3 = _aligned_malloc((sizeof (double))*NUM*NUM, 64);
102	buf4 = _aligned_malloc((sizeof (double))*NUM*NUM, 64);
103	#endif //ICC
104	#else // WIN32
105	buf1 = _mm_malloc((sizeof (double))*NUM*NUM, 64);
106	buf2 = _mm_malloc((sizeof (double))*NUM*NUM, 64);
107	buf3 = _mm_malloc((sizeof (double))*NUM*NUM, 64);
108	buf4 = _mm_malloc((sizeof (double))*NUM*NUM, 64);
109	#endif //WIN32
110	addr1 = buf1;
111	addr2 = buf2;
112	addr3 = buf3;
113	addr4 = buf4;
114	

16 インテル® VTune™ Amplifier XE の [Source] ビュー

転置行列のアルゴリズムを調査したところ、データ・アクセス・パターンの根本的な問題が見つかりました (図 17)。行列 **a** の各行において、行列 **b** 全体をメモリーから読み取らなければなりません。



```
for(i=0; i<N; i++){
    for(k=0; k<N; k++){
        for(j=0; j<N; j++){
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
    }
}
```

k++ \Rightarrow 2N 要素
i++ \Rightarrow 2N+N² 要素

Float = 8 Byte, N = 9K
 \Rightarrow 72K / k++
 \Rightarrow 648MB / i++

17 転置行列を使用するアルゴリズム

行列の列 / 行には、約 9,000 要素が含まれています。そのため、行列全体のメモリーブロックが CPU キャッシュに収まらず、キャッシュデータの退避と DRAM からのリロードが繰り返されます。行列 **c** と **a** の分散された行は、それらが割り当てられた CPU コアのスレッドによってアクセスされますが、行列 **b** は違います。アルゴリズムのこの実装では、行列 **b** のデータの半分はスレッドによってリモートソケットから読み取られます。さらに、行列 **a** の各行で行列 **b** 全体を読み取ることで、冗長なデータのロード操作が (N 回も余分に) 発生し、リモートデータにアクセスするためインテル® QPI トラフィックが増加します。

同様に、インテル® Xeon Phi™ プロセッサ・ベースのシステムでは、DRAM や MCDRAM のトラフィックを増加させているデータ・オブジェクトを特定できます。解析するメモリー・ドメイン・トラフィックを選択するだけで、オブジェクトへの参照と割り当てスタックの情報が得られます (図 18)。また、帯域幅ドメインと帯域幅使用率のタイプでグループ化すると、L2 ミスのカウントが最も大きいオブジェクトを特定できます (図 19)。

Bandwidth Utilization

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution.

Bandwidth Domain:

DRAM, GB/sec

Bandwidth Utilization Type:

DRAM, GB/sec

MCDRAM Flat, GB/sec

This histogram displays the bandwidth utilization by certain value. Use sliders at the bottom of the histogram to filter the data by a particular utilization type. To learn bandwidth capabilities, refer to your system specific bandwidth.

Elapsed Time

20s

15s

10s

5s

0s

0

10

20

30

Low

Top Memory Objects with High Bandwidth Utilization

This section shows top memory objects, sorted by LLC Misses, that were accessed when bandwidth utilization was high.

Memory Object	LLC Miss Count
new_allocator.h:104 (61 MB)	42.6%
new_allocator.h:104 (1 GB)	23.3%
new_allocator.h:104 (836 MB)	12.4%
[Unknown]	10.1%
new_allocator.h:104 (61 MB)	6.2%
[Others]	2.3%

18 メモリー・ドメイン・トラフィックの解析

Grouping:

Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack

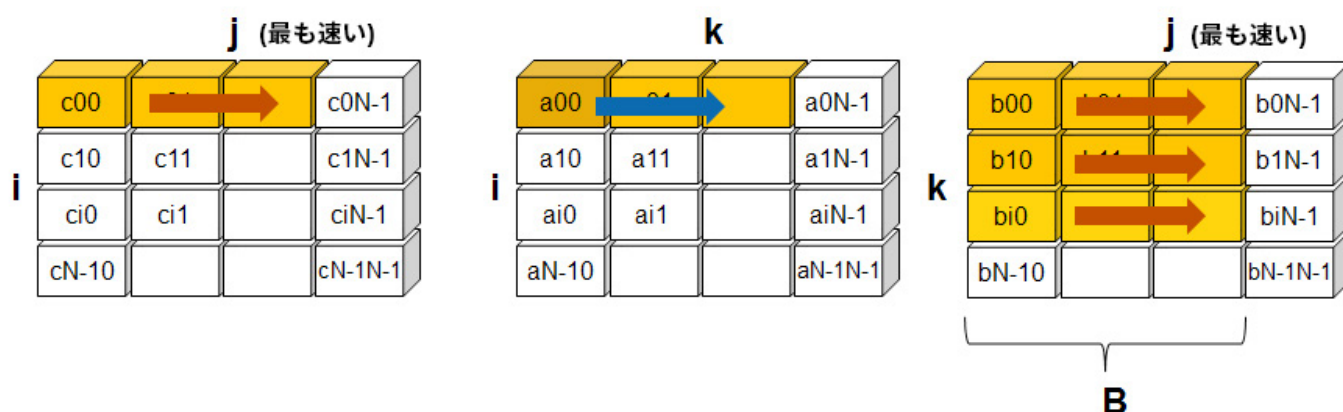
Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack	CPU Time	Memory Bound					L2 Miss Count
		L2 Hit Rate	L2 Hit Bound	L2 Miss Bound	MCDRAM Flat B...	DRAM Bandwidth Bound	
▼ DRAM, GB/sec	1738.026s	88.3%	1.9%	3.4%	0.0%	27.6%	387,011,610
▼ High	1221.252s	79.5%	1.4%	4.8%	0.0%	27.6%	384,011,520
▶ [Unknown]							39,001,170
▶ new_allocator.h:104 (61 MB)							24,000,720
▼ new_allocator.h:104 (61 MB)							165,004,950
▶ <code>__gnu_cxx::new_allocator<double>::allocate</code> ← <code>miniFE::Vector<double, int, int> Vector ← miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>, miniFE::matvec_std<miniFE::CSRMatrix<double, int, int>, miniFE::Vector<double, int, int>> ← miniFE::driver<double, int, int> ← main ← _start</code>							82,502,475
▶ new_allocator.h:104 (61 MB)							9,000,270
							3,000,090

19 帯域幅ドメイン

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。

データ・ブロッキング

乗算アルゴリズムをさらに変更して、CPU ストールを排除することで、データ・レイテンシーを減らすことができます。ローカルソケットで実行中のスレッドが 3 つの行列のすべてのデータにアクセスできるようにします。一般的に良く使用される手法の 1 つはデータ・ブロッキングです (図 20)。各配列の小さなブロックを処理することで、キャッシュにデータを保持し、CPU がそれらを再利用できるようにします (CPU キャッシュサイズのブロックを最適化することで、パフォーマンスがさらに向上する可能性があります)。また、スレッド間のブロックの配布が容易になり、過度のリモートアクセスとリロードを防ぐことができます。



```
for (i = ii; i < ii + B; i++) {
    for (k = kk; k < kk + B; k++) {
        for (j = jj; j < jj + B; j++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

Float = 8 Byte

B = 64 要素

- L1D キャッシュに収まる

i++ ⇒ 2N + N² 要素

N = 4K ⇒ 256K / i++

- L2 キャッシュに収まる

20 行列データ・ブロッキング

キャッシュブロック変更後の結果から (図 21)、NUMA による影響を修正しなくても、メモリー・レイテンシーが大幅に減り、高速に実行されるようになったことが分かります。

```
./matrix.icc
Threads #: 72 Pthreads
Matrix size: 9216
Using multiply kernel: multiply3
Freq = 2.3 GHz
Execution time = 12.08 seconds
MFLOPS: 128710.367 mflops
```

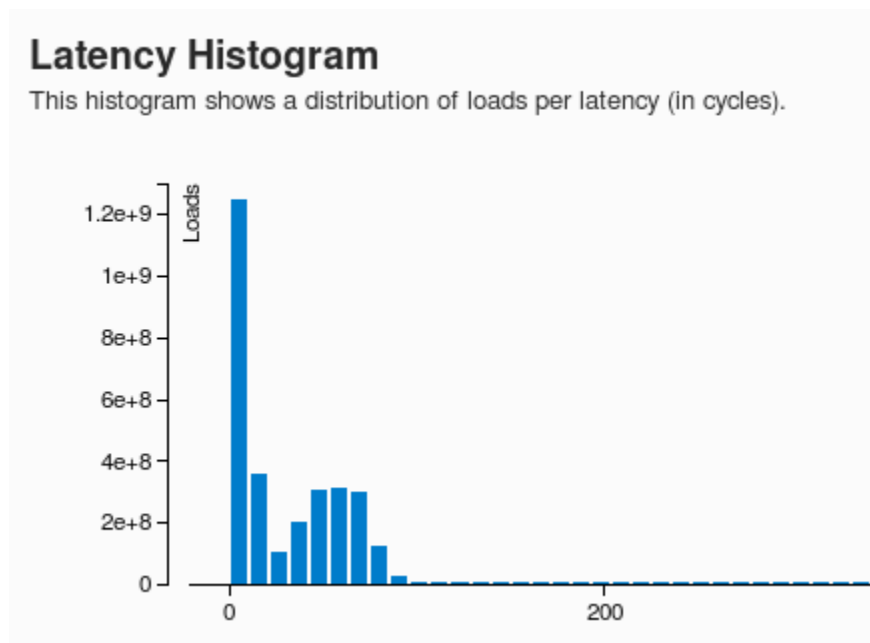
21 キャッシュ・ブロッキング変更後 (multiply3) のパフォーマンス

General Exploration (全般解析) プロファイル (図 22) は、リタイアしたパイプライン・スロットが最大 20% 増加し、残りの CPU ストールがメモリー依存とコア依存の実行で共有されることを示しています。

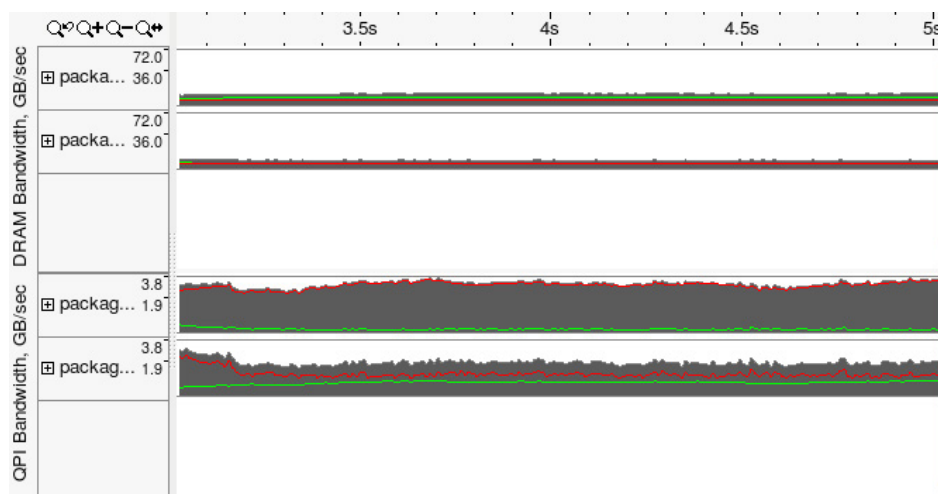
Clockticks:	2,268,184,100,000	
Instructions Retired:	1,851,470,100,000	
CPI Rate ^⑦ :	1.225	🚩
MUX Reliability ^⑦ :	0.994	
③ Front-End Bound ^⑦ :	9.4%	of Pipeline Slots
③ Bad Speculation ^⑦ :	0.0%	of Pipeline Slots
③ Back-End Bound ^⑦ :	70.9%	🚩 of Pipeline Slots
③ Memory Bound ^⑦ :	45.6%	🚩 of Pipeline Slots
③ L1 Bound ^⑦ :	0.7%	🚩 of Clockticks
③ L2 Bound ^⑦ :	5.6%	of Clockticks
③ L3 Bound ^⑦ :	31.9%	🚩 of Clockticks
Contested Accesses ^⑦ :	0.0%	of Clockticks
Data Sharing ^⑦ :	0.0%	of Clockticks
L3 Latency ^⑦ :	100.0%	🚩 of Clockticks
SQ Full ^⑦ :	0.0%	🚩 of Clockticks
③ DRAM Bound ^⑦ :	3.0%	of Clockticks
Memory Bandwidth ^⑦ :	26.9%	of Clockticks
③ Memory Latency ^⑦ :	69.5%	of Clockticks
Local DRAM ^⑦ :	36.6%	of Clockticks
Remote DRAM ^⑦ :	6.5%	of Clockticks
Remote Cache ^⑦ :	0.0%	of Clockticks
③ Store Bound ^⑦ :	1.2%	of Clockticks
③ Core Bound ^⑦ :	25.3%	🚩 of Pipeline Slots
③ Retiring ^⑦ :	19.6%	of Pipeline Slots
Total Thread Count:	73	
Paused Time ^⑦ :	0s	

22 全般解析プロファイル (multiply3)

レイテンシー・ヒストグラム (図 23) は、レイテンシーのほとんどが L2 アクセスに関連するもので、残りは 50 ~ 100 サイクルの LLC ヒット・レイテンシーであることを示しています。帯域幅タイムライン・ダイアグラム (図 24) では、データのほとんどがローカル DRAM から読み取られ、インテル® QPI トラフィックがわずかに増えたことが分かります。これでもまだ、**インテル® マス・カーネル・ライブラリー (インテル® MKL)** の倍精度行列乗算の実装 (dgemm) のパフォーマンスには及びませんが、この行列サイズでは、同程度のパフォーマンスが得られるでしょう (図 25)。このほかに最適化としてできることは、アルゴリズムがブロックされるようにし、完全に NUMA 対応にすることです。最終的なパフォーマンスを表 3 と図 26 に示します。



23 レイテンシー・ヒストグラム (multiply3)



24 帯域幅タイムライン・ダイアグラム (multiply3)

```

$./matrix.mkl
Threads #: 72 requested OpenMP threads
Matrix size: 9216
Using multiply kernel: multiply5
Freq = 2.799980 GHz
Execution time = 2.897 seconds
MFLOPS: 540032.936 mflops
    
```

25 インテル® MKL ベースの multiply5 のパフォーマンス

スレッド数	実行時間 (秒)	DP FLOPS (GFLOPS/ 秒)
4	104.8	14
8	60.1	25
16	31.3	49
36	17.85	87
72 HT	12.08	128

表 3. 最後の最適化を適用した matrix3 のパフォーマンス (36 コア、インテル® Xeon® プロセッサー E5-2697 v4、2 ソケット @ 2300MHz)

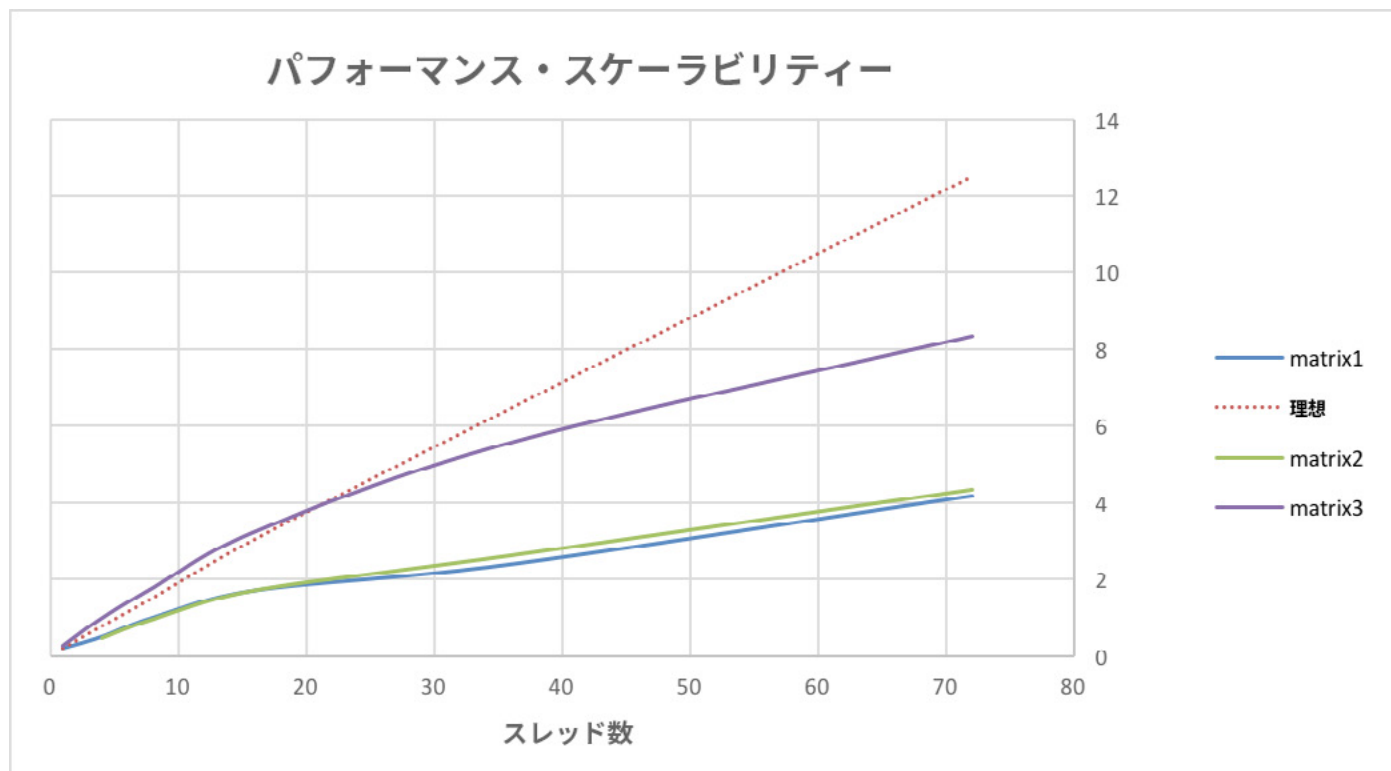
BLOG HIGHLIGHTS

AI 開発者としてはじめにすべきこと

NIVEN SINGH (INTEL) >

人工知能の可能性は、少なくとも 1950 年代から我々の想像力を掻き立ててきました。コンピューター科学者を新しくより複雑なテクノロジーの創造へと駆り立て、一般消費者の将来への関心を盛り上げました。身体的なリスクを負わずに海底を散歩したり、スマートウェイで無人自動車に乗ることができたらどうしますか？ AI とその可能性に対する理解はこの数十年の間に変わりましたが、ついに AI 時代が到来すると信じるに足る根拠が見つかりました。開発者として、我々は何をすべきでしょうか？ この記事では、AI の基本について説明し、役立つツールとリソースを紹介します。

[この記事の続きはこちら \(英語\) でご覧になれます。>](#)



26 行列乗算ベンチマークの結果

スケーラビリティ・グラフに関する補足事項：

- matrix3 は、キャッシュ・ブロッキングにより理想的なスケーラビリティを超えています。そのため、シングルスレッド・バージョンはナイーブ実装よりも高速に実行できます。
- スレッド数が物理コア数と等しくなるまで、matrix3 は理想的なスケーラビリティとほぼ同じです。[ハイパースレッディング](#)を追加しても、スケーリングは向上しません。

まとめ

一部のメモリー・アクセス・パターンは、CPU マイクロアーキテクチャーの制限により、並列アプリケーションにおいてスケーラビリティを妨げているように見えるかもしれません。これらの制限を回避するには、データの待機中に CPU ストールを引き起こすデータ配列を特定する必要があります。インテル® VTune™ Amplifier XE のメモリー・アクセス・プロファイルを利用することで、最も大きな遅延を引き起こすデータ・オブジェクト、遅延の長さ (CPU クロック数)、データが配置されているキャッシュ・サブシステムのレベル、データ・オブジェクトの割り当てと遅延アクセスのソースコードを識別することができます。この情報は、アルゴリズムを見直し、メモリー・アクセス・パターンを改善するのに役立つでしょう。

参考資料

1. Charlie Hewett. [Top Down Methodology for Software Performance Analysis \(英語\)](#).
2. [Intel® 64 and IA-32 Architectures Optimization Reference Manual \(英語\)](#).
3. Ahmad Yasin. "A Top-Down Method for Performance Analysis and Counters Architecture." IEEE Xplore: 26 June 2014. Electronic ISBN: 978-1-4799-3606-9.
4. [インテル® VTune™ Amplifier XE の General Exploration \(一般解析\) がどのように動作するかを理解する](#)
5. [マイクロアーキテクチャーのトップダウン解析法を使用してアプリケーションをチューニングする](#)

インテル® VTUNE™ AMPLIFIER XE を
評価する >



インテル® AVX-512 で向上したベクトル化のパフォーマンス

インテル® コンパイラーでループをベクトル化してスピードアップするさまざまな例

Martyn Corden インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

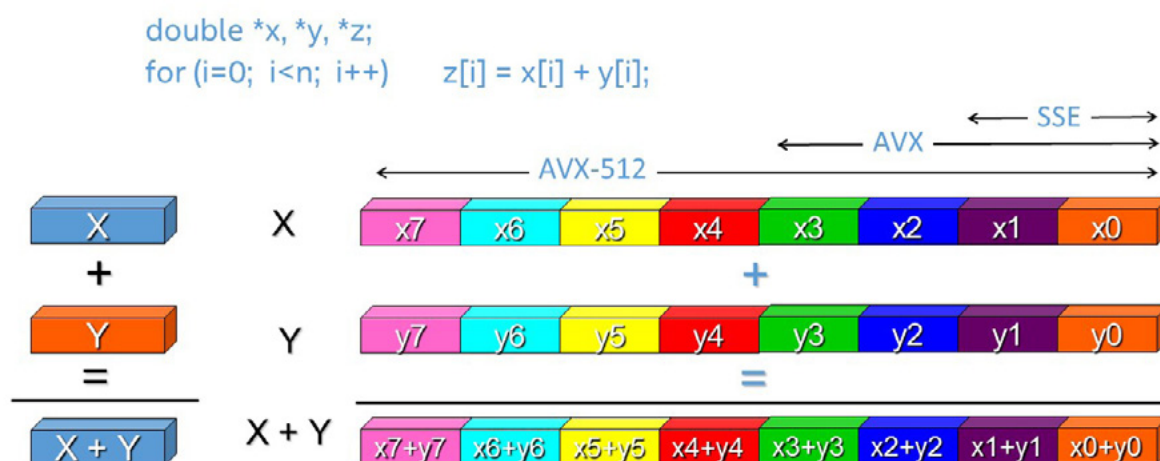
編集者から：前号の The Parallel Universe では、インテル® Parallel Studio XE 2017 のベクトル化サポートについて取り上げました。今号では、Martyn Corden が、インテル® アドバンスド・ベクトル・エクステンション 512 (インテル® AVX-512) 命令についてさらに詳しく取り上げ、以前は不可能だったベクトル化を開発者が利用する方法を説明します。

クロック速度を上げるだけで簡単にパフォーマンスを向上できる時代は遠い過去のものとなりました。ムーアの法則は、代わりに、追加のコアと SIMD レジスターの帯域幅の増加による並列処理の向上に適用されます。インテル® AVX-512 (英語) では、SIMD ベクトル幅が 512 ビットに拡張され、以前の命令セットではベクトル化できなかったループのベクトル化や、より効率的なベクトル化を行うことができる新しい命令が含まれています。

この記事では、潜在的なアドレス競合の問題がある、配列を圧縮 / 展開するループや、ヒストグラムの生成とスキャッターを実行するループを、**インテル® C/C++ コンパイラー**および**インテル® Fortran コンパイラー**でベクトル化してスピードアップする方法を、例とともに説明します。さらに、構造体配列の特定のループ形式について、コンパイラーでストライドロードやギャザーをより効率的なユニットストライド方式の SIMD ロードに変換する方法を説明します。最後に、**インテル® Parallel Studio XE 2017** に含まれるインテル® コンパイラー 17.0 の最適化レポートの新しい機能を使用して、この変換を認識する方法を説明します。これらの最適化は、**インテル® Xeon Phi™ プロセッサー** x200 製品ファミリーだけでなく、インテル® AVX-512 命令セットをサポートする将来のインテル® Xeon® プロセッサーで動作するさまざまなアプリケーションにも役立ちます。

ベクトル化の利点

図 1 は、単純な倍精度浮動小数点ループを示しています。スカラーモードでは、1 つの命令で 1 つの結果が生成されます。ベクトル化を行うと、1 つのインテル® AVX-512 命令で 8 つ (インテル® AVX では 4 つ、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) では 2 つ) の結果が生成されます。インテル® Xeon Phi™ プロセッサー x200 製品ファミリーは、32 ビットおよび 64 ビットの整数と浮動小数点データを扱うさまざまな演算でインテル® AVX-512 命令をサポートします。将来のインテル® Xeon® プロセッサーでは、8 ビットおよび 16 ビット整数にも対応する予定です。



1 スカラーおよびインテル® SSE、インテル® AVX、インテル® AVX-512 でベクトル化したループ

インテル® コンパイラーは、デフォルトで自動ベクトル化を有効にしますが、インテル® AVX-512 命令をターゲットにするには、**図 2** のいずれかのコンパイラー・オプションを指定する必要があります。

Linux* および OS X*	Windows®	ターゲット
-xmic-avx512	/Qxmic-avx512	インテル® Xeon Phi™ プロセッサ x200 製品ファミリー
-xcore-avx512	/Qxcore-avx512	将来のインテル® Xeon® プロセッサ
-xcommon-avx512	/Qxcommon-avx512	両方に共通のインテル® AVX-512 のサブセット。 ファットバイナリーでは <u>ありません</u> 。
-axmic-avx512	/Qaxmic-avx512	インテル® Xeon Phi™ プロセッサ x200 製品ファミリー およびインテル® Xeon® プロセッサ。 ファットバイナリーです。
-xhost	/Qxhost	コンパイルを行うホストマシンに搭載されるプロセッサ

表 1. インテル® AVX-512 命令を有効にするコンパイラー・オプション

ループの圧縮と展開

図 2A の Fortran の例は、配列を圧縮します。大きなソース配列から条件を満たす要素のみ、小さなターゲット配列にコピーされます。図 2B の C の例は、逆の操作 (つまり、配列の展開) を行います。小さなソース配列の要素が大きなスパース配列にコピーされます。

```
nb = 0
do ia=1, na
  if (a(ia) > 0.) then
    nb = nb + 1
    b(nb) = a(ia)
  endif
enddo
```

! 行 23
! 依存関係
! 圧縮

```
int j = 0
for (int i=0; i <N; i++) {
  if (a[i] > 0) {
    c[i] = a[k++];
  }
}
// j と k の繰り返し間の依存関係
```

// 展開

2A 配列の圧縮

2B 配列の展開

密配列インデックスの条件付きインクリメントを行うと、ループ反復間の依存関係が発生します。以前は、この依存関係が自動ベクトル化を妨げていました。例えば、インテル® AVX2 向けに図 2A のループをコンパイルすると、次のような最適化レポートが表示されます。

```
ifort -c -xcore-avx2 -qopt-report-file=stderr -qopt-report=3 compress.f90
...
ループの開始 compress.f90 (23,3)
  リマーク #15344: ループはベクトル化されませんでした: ベクトル依存関係がベクトル化を妨げています。
  リマーク #15346: ベクトル依存関係: ANTI の依存関係が nb (25:7) と nb (25:7) の間に仮定されました。
ループの終了
```

図 2B の C の例も同じようなレポートが出力されます。依存関係により不正な結果が引き起こされる可能性があるため、OpenMP* SIMD ディレクティブは使用できません。

インテル® AVX-512 では、1 つの SIMD レジスターの特定の要素を別の SIMD レジスターの連続する要素またはメモリーに書き込む新しい `vcompress` 命令により、この依存関係の制約を解消しました。同様に、`vexpand` 命令は、ソースレジスターまたはメモリーの連続する要素をディスパージョン SIMD レジスターの特定の (スパース) 要素に書き込みます。これらの新しい命令により、インテル® AVX-512 が有効な場合、コンパイラーは圧縮の例 (**図 2A**) をベクトル化することができます。

```
ifort -c -xmic-avx512 -qopt-report-file=stderr -qopt-report=3 compress.f90
...
ループの開始 compress.f90 (23,3)
  リマーク #15300: ループがベクトル化されました。
  リマーク #15450: マスクなし非アライン・ユニット・ストライド・ロード : 1
  リマーク #15457: マスク付き非アライン・ユニット・ストライド・ストア : 1
...
  リマーク #15497: ベクトル圧縮 : 1
ループの終了
```

3 インテル® AVX-512 を有効にした場合の配列の圧縮

ソース配列のすべての要素がロードされるため、ロードはマスクなしです。選択された要素のみストアされるため、有効なストアはマスク付きです。最適化レポートのリマーク #15497 は、圧縮表現が認識されベクトル化されたことを示しています。-S オプションを指定して取得できるアセンブリー・リストには、次のような命令が表示されます。

```
vcompressps %zmm4, -4(%rsi,%rdx,4){%k1}
```

図 2B の配列展開ループについても同様の結果が得られます。

1,000,000 の乱数を含む単精度配列を 1/2 に圧縮する操作をインテル® Xeon Phi™ プロセッサー 7250 で 1000 回繰り返した結果、インテル® AVX-512 はインテル® AVX2 よりも約 16 倍高速でした。このスピードアップは SIMD レジスターと命令の幅の違いによるものと言えます。

インテル® AVX-512 競合検出命令

間接アドレス指定のストアを含むループには、ベクトル化を妨げる潜在的な依存関係があります。次に例を示します。

```
for (i=0; i<n; i++) a[index[i]] = ...
```

2 つの異なる `i` の値で取得される `index[i]` の値が同じ場合、データ競合が発生し、同時に (安全に) 実行することはできません。インテル® AVX-512 の `vpconflict` 命令は、競合が発生しない (`index[i]` の値が重複しない) SIMD レーンのマスク (`i` の値) を提供することにより、この競合を回避します。これらのレーンで SIMD 計算が安全に実行された後、マスクアウトされたレーンでループが再実行されます。

ヒストグラム

ヒストグラムの生成は、多くのアプリケーション (画像処理など) で一般的な操作です。**図 4** のコード例は、入力値の配列 **x** について、配列 **h** で **sin(x)** のヒストグラムを生成します。2 つの入力値が同じヒストグラム・ビン **ih** に関連付けられ、依存関係が発生する可能性があるため、インテル® AVX2 では、このループはベクトル化されません。

```
for (i=0; i<n; i++) {
    y    = sinf(x[i]*twopi);
    ih    = floor((y-bot)*invbinw);
    ih    = ih > 0      ? ih : 0;
    ih    = ih < nbins-1 ? ih : nbins-1;
    h[ih] = h[ih] + 1;           // 行 25
}
```

4 sin(x) のヒストグラムを生成するループ

```
icc -c -xcore-avx2 histo.c -qopt-report-file=stderr -qopt-report-phase=vec
...
ループの開始 histo2.c(20,4)
  リマーク #15344: ループはベクトル化されませんでした: ベクトル依存関係がベクトル化を妨げています。 ...
  リマーク #15346: ベクトル依存関係: FLOW の依存関係が h[ih] (25:7) と h[ih] (25:7) の間に仮定されました。
ループの終了
```

インテル® AVX-512 競合検出命令を使用すると、このループを安全にベクトル化できます。

```
ifort -c -xm512-avx512 histo2.f90 -qopt-report-file=stderr -qopt-report=3 -S
...
ループの開始 histo2.c(20,4)
  リマーク #15300: ループがベクトル化されました。
  リマーク #15458: マスク付きインデックス (集約) ロード : 1
  リマーク #15459: マスク付きインデックス (分散) ストア : 1
  リマーク #15499: ヒストグラム : 2
ループの終了
```

アセンブリ・コード (この記事には含まれていません) では、**x** および **index** がロードされ、すべての SIMD レーンについて **ih** が計算されます。そして、**vpconflict** 命令を実行し、ユニークなビン (つまり、**h** の要素) をレジスターに格納するマスク付きギャザーを行ってインクリメントします。**ih** の値に重複があった場合、コードはループを巻き戻して、対応するビンを再度インクリメントします。最後に、インクリメントされたビン **h** に戻すマスク付きストア (スキッター) を行います。

0 ～ 1 の 100,000,000 の乱数値を含む単精度配列から 200 ビンのヒストグラムの生成を 10 回繰り返す単純なテストを、インテル® AVX2 向けとインテル® AVX-512 向けにコンパイルして、インテル® Xeon Phi™ プロセッサ 7250 で実行したところ、インテル® AVX-512 はインテル® AVX2 と比べて約 9 倍のスピードアップを達成しました。この結果は、問題の詳細に大きく依存します。スピードアップのほとんどは、ギャザーやスカッターではなく、この例の `sinf` 関数のような、ループ計算のベクトル化によるものです。競合が比較的少ない一般的なケースでは、より大きなスピードアップが得られます。しかし、特異点や狭いスパイクを含むヒストグラムのように、多くの競合がある場合、スピードアップは小さくなります。

ギャザーシャッフル最適化

小さな構造体またはショートベクトルの大きな配列はよく利用されますが、これらのベクトル化は効率的ではありません。構造体要素またはショートベクトルのベクトル化は、トリップカウントが低いいため、不可能または非効率です。大きな配列インデックスのベクトル化も、連続する反復で使用されるデータがメモリーで隣接していない (つまり、連続するデータの効率的な SIMD ロードを使用できない) ため、非効率です。例えば、**図 5** の例は、3 つのベクトルを含む配列の要素の 2 乗和を計算します。

```
struct Point { float x; float y; float z; };

float sumsq( struct Point *ptvec, int n) {
    float    t_sum = 0;

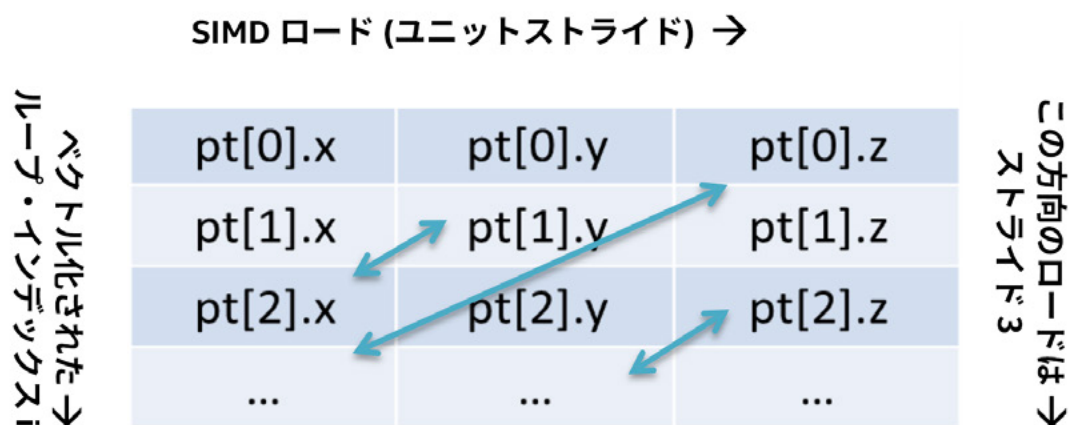
    for (int i=0; i<n; i++) {
        t_sum += ptvec[i].x * ptvec[i].x;
        t_sum += ptvec[i].y * ptvec[i].y;
        t_sum += ptvec[i].z * ptvec[i].z;
    }
    return t_sum;
}
```

5 小さな構造の大きな配列の制御

インテル® コンパイラーの旧バージョン 15 は、構造体の各要素をロードするため、ストライドロードまたはギャザーを使用してこのループをベクトル化します。

```
icc -std=c99 -xmic-avx512 -qopt-report=4 -qopt-report-file=stderr -qopt-report-phase=vec,cg
...
ループの開始 g2s.c(6,5)
リマーク #15415: ベクトル化のサポート: 集約 (gather) が生成されました (変数 ptvec): ストライド 3
...
リマーク #15300: ループがベクトル化されました。
リマーク #15460: マスク付きストライドロード: 6
```

しかし、構造体の **x**、**y**、**z** 要素がメモリーで隣接していることが分かっているならば、コンパイラーはより適切な処理を行うことができます。図 6 に示すように、コンパイラーは、コンポーネントの SIMD ロードを実行した後、置換やシャッフルを行ってデータを転置し、**i** のループがベクトル化されるように配置します。



6 ベクトル化が可能になるようにデータを転置する

インテル® コンパイラーの最新バージョン 17 でコンパイルすると、最適化レポートに「コード生成」項目が表示されます。

レポート：コード生成の最適化 [cg]
sumsq.c(10,22)：リマーク #34030：隣接するスパス (ストライド) ロードは速度の最適化が行われました。
詳細：ストライド { 12 }、型 { F32-V512, F32-V512, F32-V512 }、要素数 { 16 }、マスク { 0x000000007 }。

これは、コンパイラーがオリジナルのストライドロードを連続する SIMD ロード (およびシャッフルと並べ替え) に変換できたことを示します。10,000 の乱数に対してループを 100,000 回繰り返す単純なテストを、インテル® AVX-512 向けにコンパイルして、インテル® Xeon Phi™ プロセッサー 7250 で実行したところ、インテル® コンパイラー 17 を使用した場合の速度は、インテル® コンパイラー 15 を使用した場合の 2 倍以上でした。**pt** が 2 次元配列 **pt[10000][3]** の場合 (Fortran では、**pt** が 2 次元配列 **pt(3,10000)** または派生型配列の場合)、同じ最適化が行われます。

この「ギャザーシャッフル」最適化は、ほかの命令セット向けにも行われることがあります。しかし、インテル® AVX-512 をターゲットにする場合、強力な新しいシャッフル命令と置換命令が生成される可能性が高まります。

まとめ

インテル® AVX-512 の強力な新しい命令は、以前はベクトル化できなかったループのベクトル化や、より効率的なベクトル化により、アプリケーションのパフォーマンスを向上します。コンパイラーの最適化レポートは、これらの最適化がいつどこで行われたかを示します。



インテル® コンパイラーを評価する
インテル® PARALLEL STUDIO XE に含まれます >

関連情報

インテル® Xeon Phi™ プロセッサー

インテル® C++ コンパイラーによるベクトル化

Fortran の明示的なベクトル・プログラミング (英語)

ベクトル化の基本 (インテル® Xeon Phi™ コプロセッサー x100 製品ファミリー向けに記述された記事ですが、ほかのプロセッサーにも当てはまります)

Fortran の配列データおよび引数とベクトル化

インテル® コンパイラー・ユーザー・フォーラム (英語)

ウェビナー (英語)

インテル® コンパイラーに導入された新しい最適化レポートを最大限に活用する

インテル® コンパイラーの新しいベクトル化機能

シリアルからその先へ：高度なコードのベクトル化と最適化

データのアライメント、パディング、およびピール / リマインダー・ループ

BLOG HIGHLIGHTS

ダイレクト N 体シミュレーション

MIKE P. (INTEL) >

インテル® Xeon Phi™ プロセッサーを含む、インテル® アーキテクチャーでのパフォーマンス最適化の演習

注：この演習は、『Parallel Programming and Optimization with Intel Xeon Phi Coprocessors Second Edition (2015)』の第 4 章で説明されているさまざまな最適化の概要です。

この書籍は、xeonphi.com/book (英語) から入手できます。

このステップでは、サンプル・アプリケーションを利用してコードを現代化する方法を紹介します。提供されるソースコードは、N 体シミュレーション (重力的または静電的に互いに影響する粒子のシミュレーション) です。各粒子の位置と速度の追跡には、構造体 "Particle" を使用します。シミュレーションは時間ステップに離散化されます。各時間ステップで、最初に、ダイレクト All-to-all アルゴリズム (複雑性 $O(n^2)$) を使用して、(構造体に格納された) 各粒子の力を計算します。次に、明示的なオイラー法を使用して各粒子の速度を変更します。最後に、明示的なオイラー法を使用して粒子の位置を更新します。

この記事の続きはこちら (英語) でご覧になれます。>



インテル® ADVISOR のルーフライン解析

パフォーマンス最適化のトレードオフを視覚化する新しい方法

Kevin O' Leary インテル コーポレーション テクニカル・コンサルティング・エンジニア、
Ilyas Gazizov 同シニア・ソフトウェア・デベロッパー、Alexandra Shinsel 同テクニカル・コンサルティング・エンジニア、
Zakhar Matveev 同製品アーキテクト、Dmitry Petunin 同テクニカル・コンサルティング・エンジニア

現在、そして将来のハードウェアを最大限に活用するためには、ソフトウェアはスレッド化およびベクトル化されていなければなりません。データに基づく**ベクトル化**設計は、長期間にわたって優れたパフォーマンスを達成可能で、少ないリスクで大きな効果が得られます。ベクトルレベルおよびスレッドレベルで完全に並列化されている場合であっても、開発者が CPU/ ベクトル / スレッドの使用状況とメモリー・サブシステムのボトルネックを調整しなければならないことはよくあります。多くの場合、ルーフラインの「依存とボトルネック」パフォーマンス・モデルを使用することで、この最適化に対応できます。

この記事では、**インテル® Advisor 2017** の概要を提供し、新しいルーフライン解析 (Roofline Analysis) 機能を説明します。ルーフライン・モデルは、アプリケーションのパフォーマンスの問題に対応するための最良の方法を、直感的かつ強力に表現します。ケーススタディーでは、実際の例を用いて最適化プロセスを紹介します。

ルーフライン・モデル

ルーフライン・モデルは、カリフォルニア大学バークレー校の研究者 Samuel Williams、Andrew Waterman、David Patterson により、2009 年に「**Roofline: An Insightful Visual Performance Model for Multicore Architectures**」(英語) で提案されました。近年、このモデルは、Aleksandar Ilic、Frederico Pratas、Leonel Sousa の論文「**Cache-Aware Roofline Model: Upgrading the Loft**」(英語) で、メモリー・サブシステムのすべてのレベルに対応するように拡張されました。

ルーフライン・モデルは、開発者が次の質問に答えられるように支援することで、アプリケーションの動作に関する詳細を提供します。

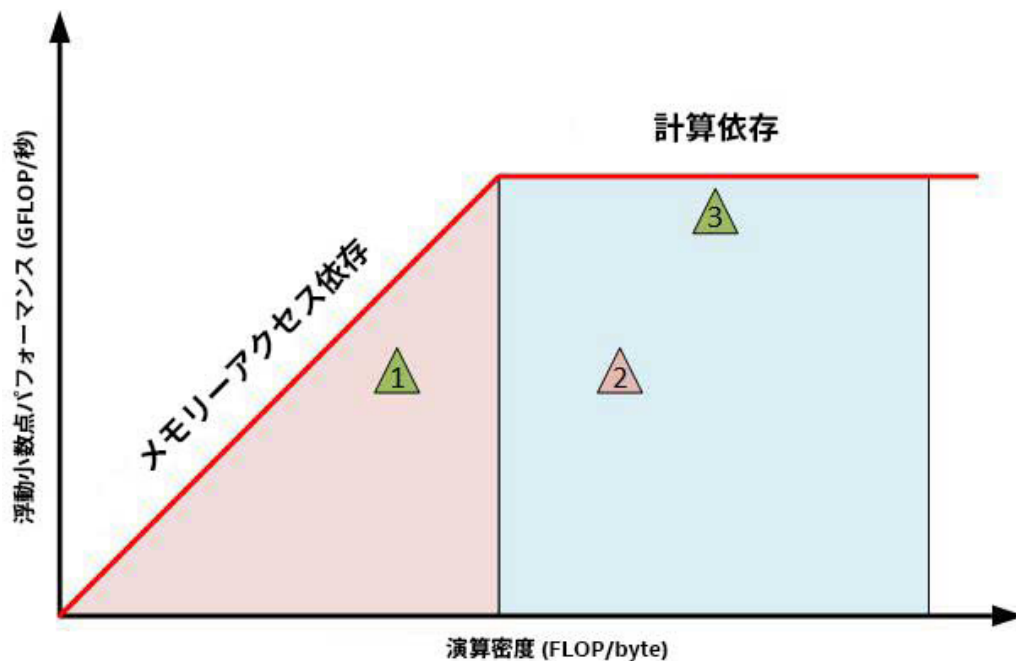
- アプリケーションは現在のハードウェアで最適に動作していますか？ そうでない場合、最も活用されていないハードウェア・リソースは何ですか？
- パフォーマンスを制限している要因は何ですか？ アプリケーションのワークロードはメモリー依存または計算依存ですか？
- アプリケーションのパフォーマンスを向上する適切な方法は何ですか？

モデルは、次の 2 つのパラメーターを測定し、アプリケーションの計算とメモリー帯域幅の上限を視覚化できるようにデータをグラフ化します。

1. 演算密度 (CPU とメモリー間で転送されたバイトあたりの浮動小数点演算数)
2. 浮動小数点パフォーマンス (GFLOPS)

データポイントがモデルライン (ルーフライン) に近いほど、適切に最適化されていることを示します (図 1)。青色の領域にあるカーネルのほうが計算への依存度が高く、Y 軸の上方向にあるカーネルのほうがピーク浮動小数点パフォーマンスに近くなります。これらのカーネルのパフォーマンスは、プラットフォームの計算能力に依存します。カーネル 3 のパフォーマンスを高めるには、**インテル® Xeon Phi™** プロセッサのような計算能力とメモリー・スループットが高い高度な並列プラットフォームへの移行を検討してください。カーネル 2 は、上限からほど遠く離れているため、ベクトル化によりパフォーマンスを向上できるでしょう。

グラフの赤色の領域にあるカーネルはメモリーに依存します。Y 軸の上方向になるほど、カーネルはプラットフォームの DRAM とキャッシュのピーク帯域幅により制限されます。これらのカーネルのパフォーマンスを向上するには、データ項目ごとの計算量を増やして、パフォーマンスの上限が高いグラフの右側に移動するように、アルゴリズムまたはその実装を再検討してください。これらのカーネルも、インテル® Xeon Phi™ プロセッサの高いメモリー帯域幅により、高速に実行できる可能性があります。

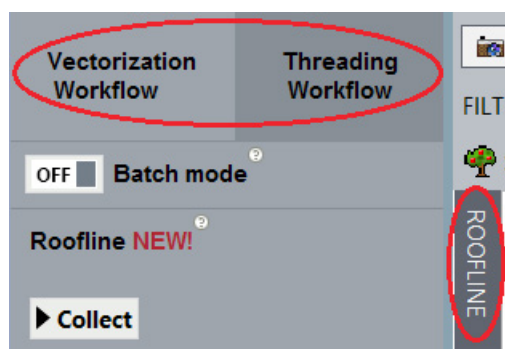


1 ルーフライン・グラフ

インテル® Advisor の概要

インテル® Advisor は、強力なソフトウェア設計とアプリケーション・パフォーマンスの特徴付けを支援するソフトウェア解析ツールです。スレッド並列処理のプロトタイプ生成 (スレッド化アドバイザー)、ベクトル並列処理の最適化 (ベクトル化アドバイザー)、メモリと計算の特徴付け (ルーフライン・オートメーション) 機能を提供します。

この記事では、主にインテル® Advisor のルーフライン解析とベクトル化解析を取り上げます。インテル® Advisor の GUI では、ワークフロー・セクターを使用してベクトル化アドバイザーとスレッド化アドバイザーのワークフローを切り替えることができます。ルーフライン・グラフは、[ROOFLINE] サイドバーからアクセスできます (図 2)。



2 インテル® Advisor のワークフロー・セクター

ベクトル化アドバイザーは、次のステップを使用してコードのパフォーマンスを向上できるように支援します。

- **サーベイ (Survey)** は、最も時間を費やしているループと詳細な SIMD 統計を表示します。
- **FLOPS とトリップカウント (Trip Counts)** は、各ループと関数の反復回数、呼び出し回数、および正確な FLOPS を測定します。
- **推奨事項 (Recommendations)** は、パフォーマンスの問題を解決するための具体的なアドバイスを示します。
- **依存性解析 (Dependencies Analysis)** は、ループにベクトル化や並列化の妨げとなる反復間の依存性がないか、動的な依存性解析を行います。
- **メモリー・アクセス・パターン解析 (Memory Access Patterns Analysis)** は、ベクトル化に適した方法でメモリーを参照しているか確認します。

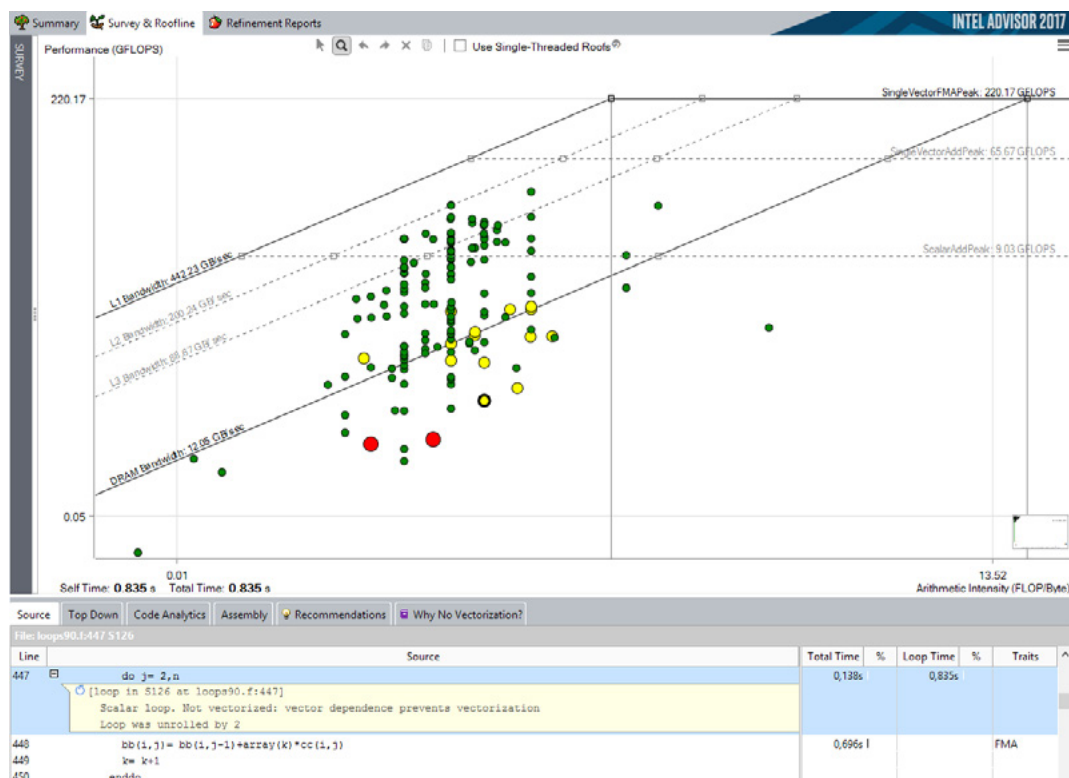
ベクトル化アドバイザーとルーフライン解析を組み合わせることで、パフォーマンスと設計に関する多くの重要な洞察を得ることができます。例えば、ルーフライン・グラフのデータを解釈するには、ベクトル化アドバイザーの [Survey Report (サーベイレポート)] で提供されるベクトル化の効率メトリックを把握しておく必要があります。

インテル® Advisor のルーフライン解析

インテル® Advisor は、「キャッシュを意識した」ルーフライン・モデルを実装し、メモリー / キャッシュ階層のすべてのレベルにわたる詳細を提供します。

- ルーフラインの傾斜部分は、すべてのデータが該当キャッシュに収まる場合のピーク・パフォーマンスを示します。
- 水平線は、ベクトル化やその他の CPU リソースが効率良く利用されている場合に達成可能なピーク・パフォーマンスを示します。

インテル® Advisor は、各ループに対応する点をルーフライン・グラフ上に描画します (図 3)。点の大きさや色は、ループの相対的な実行時間を示します。ほとんどのループは、キャッシュメモリーを効率良く利用できるように、さらなる最適化が必要です。いくつかのループ (例えば、グラフ中央の垂直線の右側で、水平の破線 ScalarAddPeak 上にある緑色の点) は、効率良くベクトル化されていません。このように、ルーフライン・グラフを利用することで、アプリケーション・パフォーマンス向上の可能性を簡単に見つけることができます。



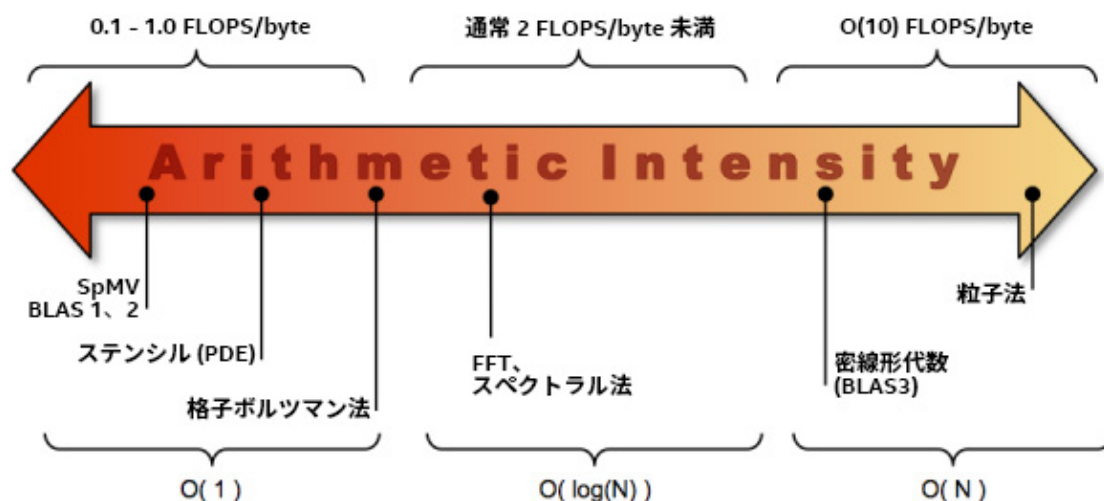
3 インテル® Advisor のルーフライン・グラフと [Source (ソース)] タブ

インテル® Advisor のルーフライン・グラフの解釈方法

ルーフライン・グラフは、役立つ情報を提供しますが、入力と対応する出力を確認するためのリファレンス・テーブルではありません。調査すべき要因を提案するガイドに過ぎないため、その解釈が必要になります。

図 3 に示すようなルーフライン・グラフ上の線は、ホストシステムにおけるベースラインとパフォーマンスの限界を特定するため、インテル® Advisor でベンチマークを実行して得たカーネル・パフォーマンスのハードウェアによる限界を表します。最上部の線 (ルーフ) は、マシンの最大パフォーマンスを示します。図 3 では、「L1 Bandwidth (L1 帯域幅)」と「Single Vector FMA Peak (単一ベクトル FMA ピーク)」です。すべてのカーネルがこのパフォーマンスを達成できるとは限りません。アルゴリズムの性質によっては、最終的により下のルーフで制限される可能性があります (例えば、ベクトル化できないカーネルは、スカラー計算の最大パフォーマンスに制限されます)。

グラフ上のカーネルの水平位置は、演算密度 (オペランドのサイズで測定された CPU とメモリー間で転送されたバイトあたりの浮動小数点演算数) を示します。これは、主にアルゴリズムによって決まりますが、コンパイル時の最適化によって変わります。図 4 は、さまざまなアルゴリズムとその演算密度の例です。カーネルのアルゴリズムを再設計して演算密度を上げると、ルーフライン・グラフ上で右側に移動できます。傾斜部分 (メモリー帯域幅のルーフ) に位置している場合は、この移動により最大パフォーマンスが向上する可能性があります。



4 演算密度

カーネルの垂直位置は、さまざまなボトルネックを示します。カーネルがループよりも上に位置している場合、そのループはパフォーマンスに影響する可能性があります、重要なパフォーマンス・ボトルネックではありません。カーネルがループよりも下に位置している場合、ボトルネックの可能性があり、特定の最適化により解決できます。カーネルがスカラー計算ピークよりも下に位置している場合は、カーネルのベクトル化状況を調査すると良いでしょう。全くあるいは効率良くベクトル化されていない場合、スカラー計算ピークがボトルネックである可能性が高く、慎重にカーネルのベクトル化を改善または実装することを推奨します。効率良くベクトル化されている場合は無視し、カーネルの上にあるほかのループの調査に移ります。

インテル® Advisor を使用してパフォーマンスの問題を解決する

このセクションでは、インテル® Advisor に関連したヒントを提供します。

ヒント 1: [Summary (サマリー)] ビューで最も時間のかかるループ (図 5 と図 6) とチューニングの推奨事項を確認します (図 7)。

Top time-consuming loops

Loop	Source Location	Self Time ^③	Total Time ^③	Trip Counts ^③
flessparseapngtc	lesSparse.f:387	5.9143s	5.9143s	6
flessparseapq	lesSparse.f:232	4.9664s	4.9664s	1; 2; 4; 1; 1
flessparseapkg	lesSparse.f:286	4.3634s	4.3634s	2; 1; 4; 1
fmtxblkdot2	ftools.f:445	1.0166s	1.0166s	9162
fmtxvdimvecmult	ftools.f:45	0.8999s	0.8999s	572

5 最も時間のかかるループのサマリー

- 6

最も時間のかかるループのサマリー
(1 番上のループはベクトル化されている)
- 7

最も時間のかかるループに対する推奨事項のサマリー

Top time-consuming loops^③

Loop	Self Time ^②	Total Time ^②	Trip Counts ^②
[loop in flessparseapngtc at lesSparse.f:389]	5.879s	5.879s	1; 2
[loop in flessparseapg at lesSparse.f:232]	5.247s	5.247s	1; 2; 4; 1; 1
[loop in flessparseapkg at lesSparse.f:287]	4.247s	4.247s	2; 1; 4; 1
[loop in fmtxblkdot2 at ftools.f:445]	1.158s	1.158s	9162; 3; 1
[loop in fmtxblkdmaxpy at ftools.f:874]	1.148s	1.148s	9162; 3; 1

Recommendations^③

Loop	Self Time ^②	Recommendations ^③
[loop in flessparseapngtc at lesSparse.f:389]	5.879s	Enforce vectorized remainder
[loop in flessparseapg at lesSparse.f:232]	5.247s	Add data padding
[loop in flessparseapkg at lesSparse.f:287]	4.247s	Enforce vectorized remainder Add data padding
[loop in flessparseapsclr at lesSparse.f:514]	0.261s	Disable unrolling Enforce vectorized remainder Align data Add data padding
[loop in flessparseapfull at lesSparse.f:448]	0.190s	Enforce vectorized remainder Add data padding

ヒント 2: ルーフラインのカスタマイズで不要なループを削除します (図 8)。

例えば、単精度データのみを扱う場合、ルーフラインから倍精度ピークを削除することができます。

Roof Name	Visible	Selected
DRAM Bandwidth	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
L1 Bandwidth	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
L2 Bandwidth	<input checked="" type="checkbox"/>	<input type="checkbox"/>
L3 Bandwidth	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Scalar Add Peak	<input checked="" type="checkbox"/>	<input type="checkbox"/>
SP Vector Add Peak	<input checked="" type="checkbox"/>	<input type="checkbox"/>
DP Vector Add Peak	<input type="checkbox"/>	<input type="checkbox"/>
SP Vector FMA Peak	<input checked="" type="checkbox"/>	<input type="checkbox"/>
DP Vector FMA Peak	<input type="checkbox"/>	<input type="checkbox"/>

Loop Weight Representation

Cancel

Default

☒ Size

☒ Color

☐ Visible

+

4

green

☒

-

Threshold Value

0.5

%

+

6

yellow

☒

-

Threshold Value

5

%

+

8

red

☒

- ヒント 3:** スマートモード (Smart Mode) を使用して最適化により最も大きな効果が得られる候補を見つけます (図 9)。
- ルーフライン上では、ループは経過セルフ時間順に表示されますが、スマートモードをオンにすると合計時間が多いループを見つけることができます。合計時間が多いループのほうが、最適化により得られる効果が大きくなります。
- 8

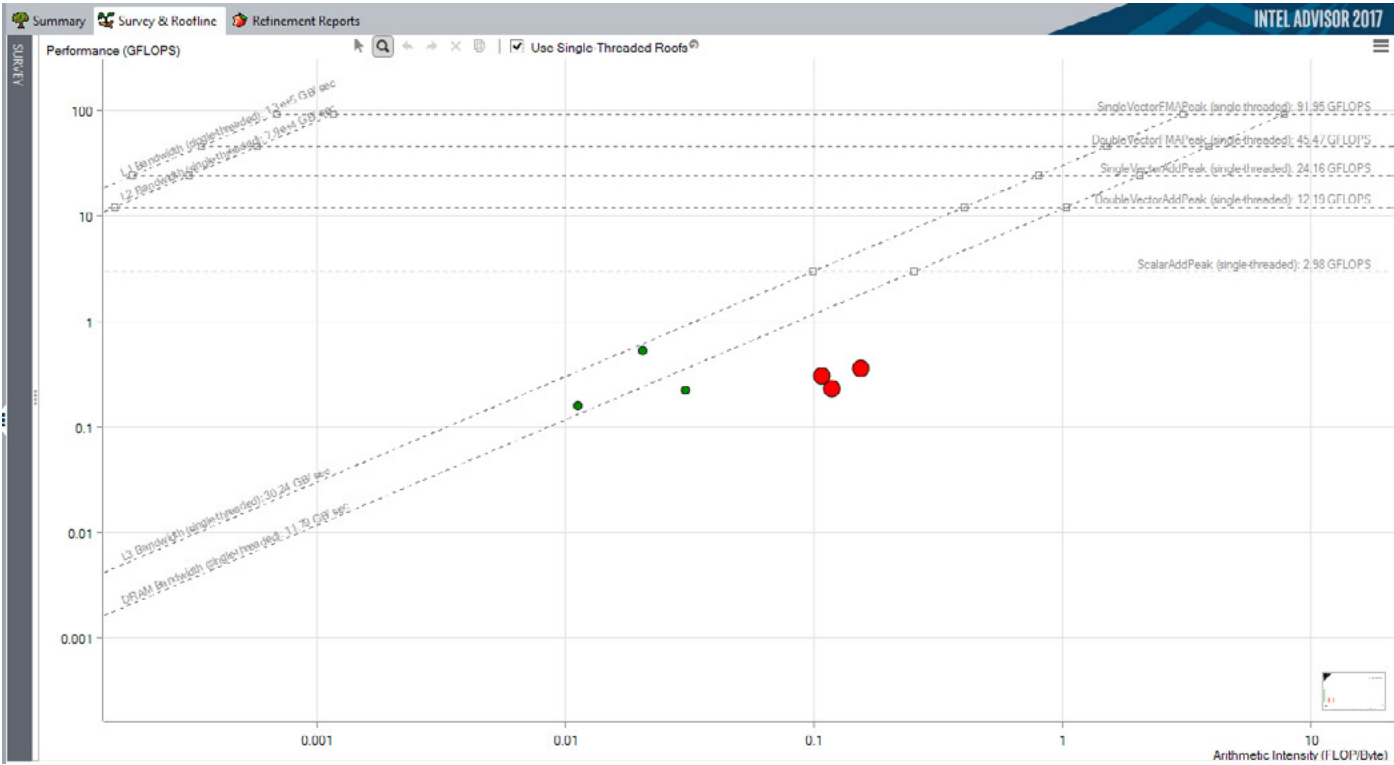
インテル® Advisor のルーフライン・グラフのカスタマイズ

9a

インテル® Advisor の [Smart Mode (スマートモード)] セレクター

OFF

Smart Mode²



9b スマートモードでフィルターされたインテル® Advisor のルーフライン・グラフ

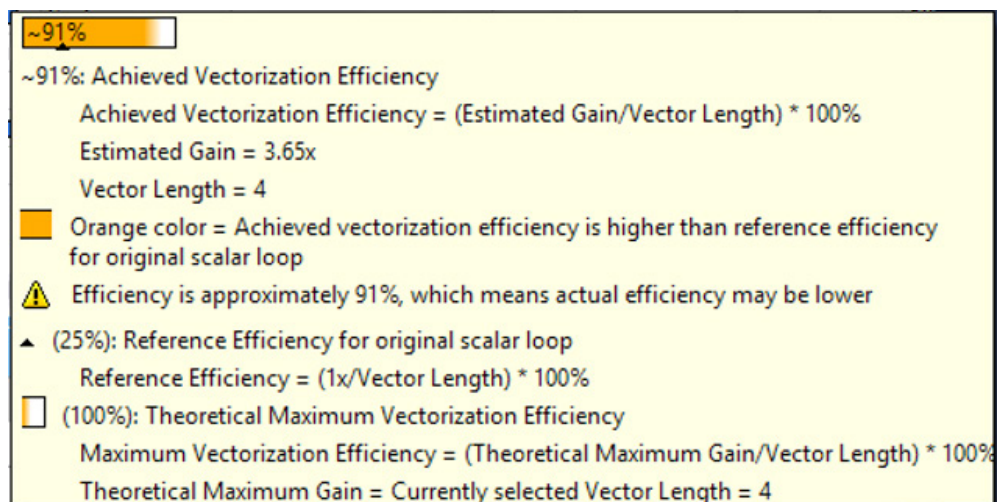
ヒント 4: インテル® Advisor のほかの機能を使用して、ルーフライン・グラフの情報を補完します。

ベクトル化の効率は、ベクトル化の温度計と言えます (図 10 と図 11)。[Instruction Set Analysis (命令セット解析)] の [Traits (特徴)] 列で、ベクトル化に影響している可能性がある要因を確認できます (図 10)。ベクトル化に適していない方法でメモリーを参照している可能性がある場合は、インテル® Advisor のメモリー・アクセス・パターン収集の実行を検討してください。[編集者注: The Parallel Universe 本号の Vladimir Tsymbal の記事「[並列アプリケーションのスケーラビリティの問題を特定する](#)」では、いくつかのメモリーアクセスの解析手法を説明しています。]

Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Trip Counts	Instruction Set Analysis		
					Vect...	Efficiency	Gain...	VL (...)		Traits	Data T...	Num.
1 Inefficient ...	5.281s	5.281s	Vectorized (Bo...		AVX	~55%	2.19x	4	25; 2	Inserts	Float64	3; 7
1 Ineffective ...	2.828s	2.828s	Vectorized (Bo...		AVX	~91%	3.65x	4	25; 2		Float64	3

10 ベクトル化の効率がハイライト表示されたインテル® Advisor のサーベイ

11 インテル® Advisor のベクトル化の効率に関する説明



ヒント 5: ループセクターを使用して、ベクトル化されたループとされなかったループを切り替えることができます (図 12)。

12 インテル® Advisor のベクトル化された / されていないループのセクター



ヒント 6: [Source (ソース)] タブとルーフライン・グラフを併用します (図 13)。

インテル® Advisor は、ソースコードをパフォーマンス・プロファイルにシームレスに統合します。

BLOG HIGHLIGHTS

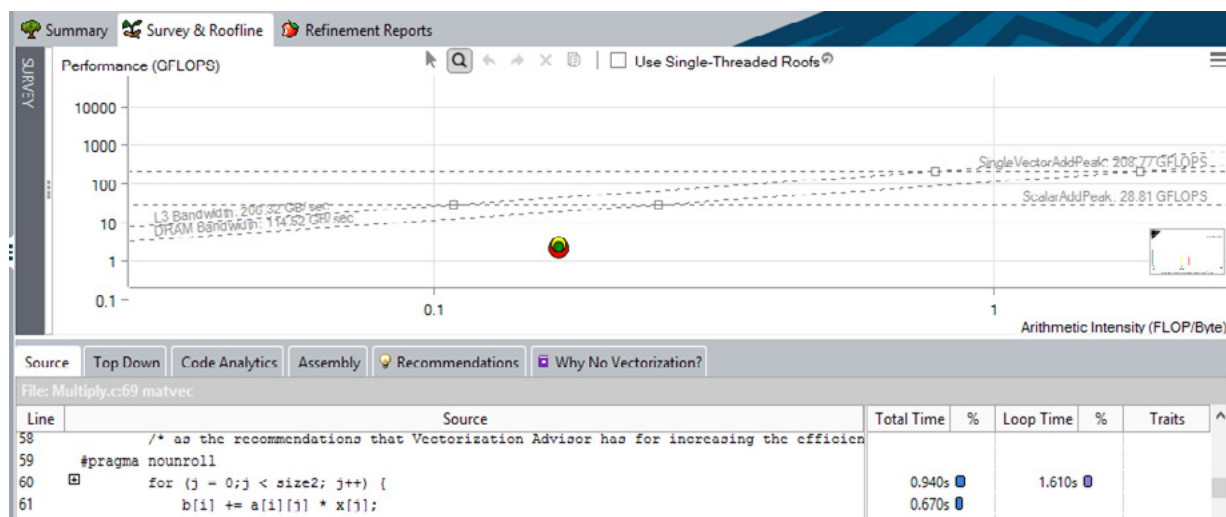
インテル® Xeon Phi™ プロセッサ x200 製品ファミリー (開発コード名 Knights Landing) のユーザーモード (リング 3) の MONITOR と MWAIT

JAMES C. (INTEL) >
BENJAMIN C. (INTEL) >

インテル® Xeon Phi™ プロセッサ x200 製品ファミリー (開発コード名 Knights Landing) には、通常アーキテクチャーでリング 0 (カーネルコード) に制限されている MONITOR および MWAIT¹ 命令を、リング 0 以外のリングで実行可能なモデル固有の機能があります。具体的には、通常ユーザーモードである、リング 3 でこれらの命令の実行を許可します。

この機能を有効にするには、以下に示すように、MSR 140H (MISC_FEATURE_ENABLES モデル固有レジスター) のビット 1 を設定します。このレジスターを読み取ることで、リング 0 以外で命令が有効になっているかどうか判断することもできます。

この記事の続きはこちら (英語) でご覧になれます。 >




13 インテル® Advisor のルーフライン・グラフと [Source (ソース)] タブ

ケーススタディー：ルーフライン解析を使用した MRI 画像再構成ベンチマークのチューニング

514.pomriq SPEC ACCEL* ベンチマークは、Stone ほか (2008) で説明されている MRI 画像再構成カーネルです。MRI 画像再構成は、収集した信号を傾斜磁場に変換します。収集領域は、傾斜磁場の空間 (k 空間) に位置します。MRI 画像再構成の Q 行列は、信号収集軌道 (k 空間における信号の収集方法) に基づいて事前計算可能な値です。アルゴリズムは、対象の MRI スキャン軌道と収集する信号を表す大きな入力セットを処理します。Q 行列の各要素は、すべての軌道の収集信号の加重によって計算されます。各加重は、入力の 3 要素のベクトルドット積、出力の 3D 位置、いくつかの三角関数により計算されます。出力の Q 要素は複素数ですが、入力は多要素ベクトルです。三角関数は重たい処理であるため、カーネルは根本的に計算依存です。問題の正則性により、メモリー帯域幅は容易に管理できます。そのため、タイリングとデータレイアウトにより人為的なメモリー帯域幅のボトルネックを排除したら、低レベルのシーケンシャル・コードを最適化し、ループアンロールなどにより命令ストリームの効率を改善することが重要です。


514.pomriq への入力は、k 空間の値の数、X 空間の値の数、そして k 空間のサンプルの k 空間座標、X 空間座標、Phi フィールドの複素数値のリストを含む 1 つのファイルです。座標と複素数値の各セットは配列に格納され、各フィールドは隣接して記述されます。514.pomriq の出力は、各行に「実部, 虚部」形式で複素数値を格納した結果の Q 行列です。このケーススタディーでは、514.pomriq 計算カーネルを解析し、最適化します。

図 14 は、インテル® Xeon Phi™ プロセッサ 7250 上で 514.pomriq を実行して得られたインテル® Advisor の [Summary (サマリー)] ビューです。Q 行列の計算に関連したループが hotspot であることが分かります。



Vectorization Advisor

Vectorization Advisor is a vectorization analysis tool that lets you identify loops that will benefit most from vectorization.



Program metrics


Elapsed Time: 36.93s

Vector Instruction Set: AVX512

Total GFLOP Count: 19293.90


Number of CPU Threads: 136

Total GFLOPS: 522.51




Loop metrics






Total CPU time	4267.88s	<div></div> 100.0%
Time in 1 vectorized loop	4206.25s	<div></div> 98.6%
Time in scalar code	61.62s	<div></div>




Vectorization Gain/Efficiency (Not available)^②



Top time-consuming loops^②


Loop	Self Time ^②	Total Time ^②	Trip Counts ^②
 [loop in ComputeQCPU at computeQ.c:65]	1957.548s	4206.254s	12500
 [loop in ComputeQCPU at computeQ.c:58]	6.963s	4213.216s	15420
 [loop in outputData at file.c:70]	0.040s	4.160s	2097152
 [loop in start_thread at ?]	0s	49.660s	
 [loop in OpenMP worker at z_Linux_util.c:769]	0s	49.660s	




Refinement analysis data^②

These loops were analyzed for memory access patterns and dependencies:

Site Location	Dependencies	Strides Distribution
[loop in ComputeQCPU at computeQ.c:66]	No information available	<div></div> 96% / 0% / 4%



Collection details



Platform information

CPU Name: Intel(R) Xeon Phi(TM) CPU 7250 @000000 1.40GHz

Frequency: 1.40 GHz

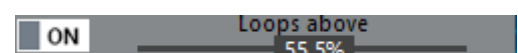
Logical CPU Count: 272

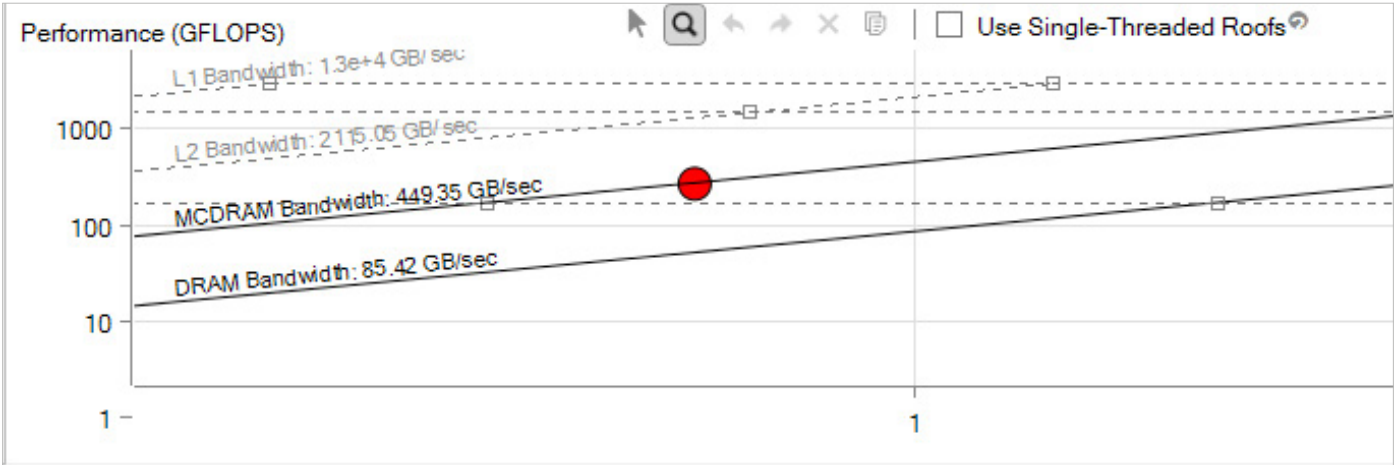
Operating System: Linux

14 インテル® Advisor の [Summary (サマリー)] ビュー

インテル® Advisor のスマートモードを使用して、最適化により最も大きな効果が得られる候補を絞り込みます (図 15a)。最も時間のかかるループはベクトル化されていますが、MCDRAM ループよりも下にあります (図 15b)。これは、メモリーの使用状況に問題がある可能性を示しています。[Code Analytics (コード解析)] タブと [Recommendations (推奨事項)] タブを使用してループを調査します。

15a インテル® Advisor の「Smart Mode (スマートモード)」セレクト





15b スマートモードでフィルターされたインテル® Advisor のルーフライン・グラフ



16 ギャザーがハイライト表示されたインテル® Advisor の [Code Analytics (コード解析)] タブ

負荷の高いギャザー命令があることが分かります (図 16)。また、ループのメモリー・アクセス・パターンの調査を勧めるアドバイスが表示されています (図 17)。メモリー・アクセス・パターン解析を実行したところ、ギャザーストライドはループあたり 4% であることが分かりました。この情報は、最適でないメモリーアクセスの場所を特定するヒントになります (図 18)。詳細を確認すると、ストライドは定数なのでギャザー命令を使用する必要がないことが分かりました (図 19a)。インテル® Advisor の [Recommendations (推奨事項)] タブでも同じことが確認できます (図 19b)。

Issue: Possible inefficient memory access patterns present

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

☒ Recommendation: Confirm inefficient memory access patterns

There is no confirmation inefficient memory access patterns are present. To confirm: Run a [Memory Access Patterns analysis](#).

Confidence: Need More Data

17 インテル® Advisor のメモリー・アクセス・パターンに関する推奨事項

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name	Recon
[loop in ComputeQCPU at computeQ.c:66]	No information available	96% / 0% / 4%	Mixed strides	3MB	loop_site_17	

Memory Access Patterns Report

Dependencies Report

Recommendations

ID		Stride	Type	Source	Nested Function	Variable references	Access Footprint
P1			Gather stride	computeQ.c:66		block 0x7f0045867010 allocated at main.c:99	3MB
P2			Parallel site information	computeQ.c:66			
P4		0	Uniform stride	omriq_exe_nog2s:0x29a8	__svml_sincosf16	__svml_sincosf16_chosen_core_func	8B
P5		0	Uniform stride	omriq_exe_nog2s:0x29cf	__svml_sincosf16_b3		64B
P6		0	Uniform stride	omriq_exe_nog2s:0x29de	__svml_sincosf16_b3		64B
P7		0	Uniform stride	omriq_exe_nog2s:0x29f3	__svml_sincosf16_b3	__svml_ssincos_data	64B
P8		0	Uniform stride	omriq_exe_nog2s:0x29fa	__svml_sincosf16_b3	__svml_ssincos_data	64B
P9		0	Uniform stride	omriq_exe_nog2s:0x2a07	__svml_sincosf16_b3	__svml_ssincos_data	64B
P10		0	Uniform stride	omriq_exe_nog2s:0x2a14	__svml_sincosf16_b3	__svml_ssincos_data	64B
P11		0	Uniform stride	omriq_exe_nog2s:0x2a1b	__svml_sincosf16_b3	__svml_ssincos_data	64B

18 インテル® Advisor のメモリー・アクセス・パターン・レポート

19a インテル® Advisor のメモリー・アクセス・パターン・レポートの詳細

Details View

Gather (irregular) access

Operand Size (bits): 32

Operand Type: bit*16;float32*16

Vector Length: 16

Memory access footprint: 3MB

▼ Gather/scatter details

Pattern: "Constant (non-unit)"

Instruction accesses values with constant offset from the base:

- stride within instruction = X

- stride between iterations = X*vector length

Horizontal stride (bytes): 16

Vertical stride (bytes): 256

Mask is constant

Mask: [1111111111111111]

Active elements in the mask: 100.0%

▼ Variable references

Names: block 0x7f0045867010 allocated at main.c:99

Issue: Inefficient gather/scatter instructions present

The compiler assumes indirect or irregular stride access to data used for vector operations. Improve memory access by alerting the compiler to detected regular stride access patterns, such as.

Pattern	Description
Invariant	The instruction accesses values in the same memory throughout the loop.
Uniform (Horizontal Invariant)	The instruction accesses values in the same memory within the vector iteration.
Vertical Invariant	The instruction accesses the memory locations using the same offset across all vector iterations.
Unit	The instruction accesses values in contiguous memory throughout the loop, and the stride between vector iterations = vector length.

Recommendation: Refactor code with detected regular stride access patterns

The Memory Access Patterns Report shows the following regular stride access(es):

Confidence: Low

19b インテル® Advisor のギャザー / スキャッターに関する推奨事項

```
#pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
for (indexK = 0; indexK < numK; indexK++) {
    expArg = PIx2 * (kVals[indexK].Kx * x[indexX] +
        kVals[indexK].Ky * y[indexX] +
        kVals[indexK].Kz * z[indexX]);

    cosArg = cosf(expArg);
    sinArg = sinf(expArg);

    float phi = kVals[indexK].PhiMag;
    QrSum += phi * cosArg;
    QiSum += phi * sinArg;
}
```

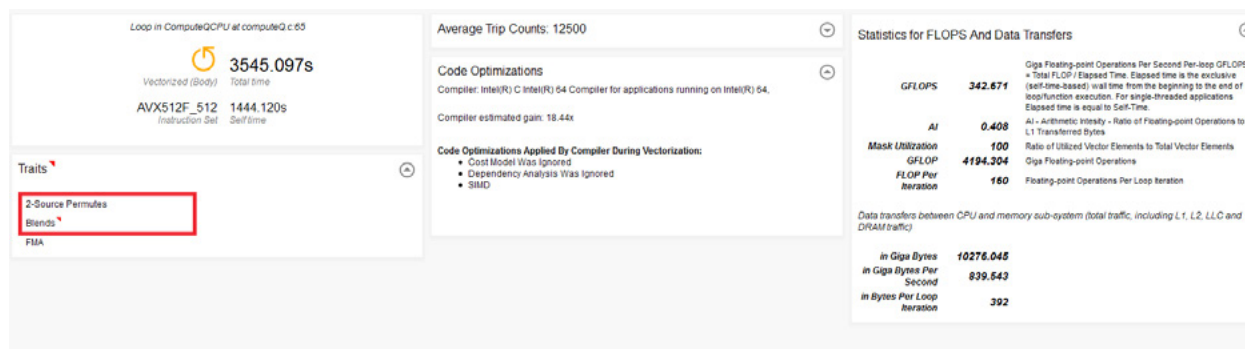
20 速度を制限しているカーネルのソースコード

カーネルのソースコードを確認したところ、原因が分かりました (図 20)。このコードは、ベクトル化すると「ギャザー」される構造体配列を使用しています。新しいバージョンのインテル® コンパイラーは、このアクセスパターンを認識し、より軽量の命令を使用できるようにギャザーを排除する最適化を適用します。「ギャザー」の置換は、「ギャザーシャッフル / 並べ替え」コンパイラー変換により実行され、最近の CPU (特にインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) をサポートするプラットフォーム) では、多くの場合、これにより利点が得られます。[編集者注: The Parallel Universe 本号の Martyn Corden の記事「[インテル® AVX-512 で向上したベクトル化のパフォーマンス](#)」では、インテル® コンパイラー 2017 でインテル® AVX-512 を利用して新たなループベクトル化を可能にする方法について説明しています。]

「ギャザーシャッフル / 並べ替え」をサポートする新しいインテル® コンパイラー (インテル® Parallel Studio XE 2017 Update 1 に含まれるインテル® コンパイラー 17.0 Update 1 など) で再コンパイルして、ルーフラインを見てみましょう。カーネルを示す点が MCDRAM の上に移動し、ギャザー命令がなくなり (インテル® AVX-512 の「2 ソースの並べ替え」に置換され)、FLOPS が向上したことが分かります (図 21 と 図 22)。



21 インテル® Advisor のルーフライン・グラフ: MCDRAM 帯域幅の上にループが移動



22 インテル® Advisor の [Code Analytics (コード解析)] タブ

しかし、AOS (構造体配列) から SOA (配列構造体) への変換を利用することで、この問題を効率良く解決できます。この最適化は、より扱いやすいデータコンテナを使用することで、ベクトル処理の効率を向上します。これまでは、この変換を行う場合、データ構造を手動で変更する必要がありました。しかし、SIMD Data Layout Templates (SDLT) (図 23) を使用することで、**kValues** 構造体の宣言、初期化、k 値の計算位置に数行のコードを追加するだけで、パフォーマンスを向上できるようになりました。

```
#include <sdl/sdl.h>
struct kValues {
    float Kx;
    float Ky;
    float Kz;
    float PhiMag;
};

SDLT_PRIMITIVE(kValues, Kx, Ky, Kz, PhiMag)

sdl::soald_container<kValues> inputKValues(numK);
auto kValues = inputKValues.access();

for (k = 0; k < numK; k++) {
    kValues[k].Kx() = kx[k];
    kValues[k].Ky() = ky[k];
    kValues[k].Kz() = kz[k];
    kValues[k].PhiMag() = phiMag[k];
}

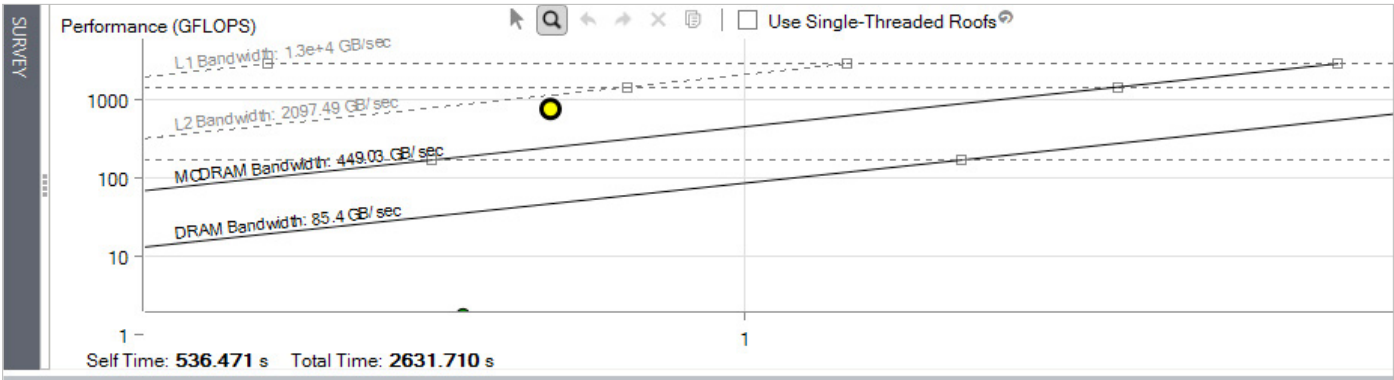
auto kVals = inputKValues.const_access();
#pragma omp simd private(expArg, cosArg, sinArg) reduction(+:QrSum, QiSum)
for (indexK = 0; indexK < numK; indexK++) {
    expArg = PIx2 * (kVals[indexK].Kx() * x[indexX] +
        kVals[indexK].Ky() * y[indexX] +
        kVals[indexK].Kz() * z[indexX]);

    cosArg = cosf(expArg);
    sinArg = sinf(expArg);

    float phi = kVals[indexK].PhiMag();
    QrSum += phi * cosArg;
    QiSum += phi * sinArg;
}
```

23 SIMD Data Layout Templates ライブラリーの使用

新しいルーフライン・グラフを確認してみましょう (図 24)。この最適化を適用後、カーネルを示す点が赤色ではなくなりました。これは、実行時間が短くなり、GFLOPS が向上し、L2 ルーフに近付いたことを示しています。さらに、ループでユニット・ストライド・アクセスを利用することで、特別なメモリー操作がなくなりました (図 25a と図 25b)。最終的には、カーネルでは 3 倍、アプリケーション全体では 1.5 倍のパフォーマンス向上を達成できました。

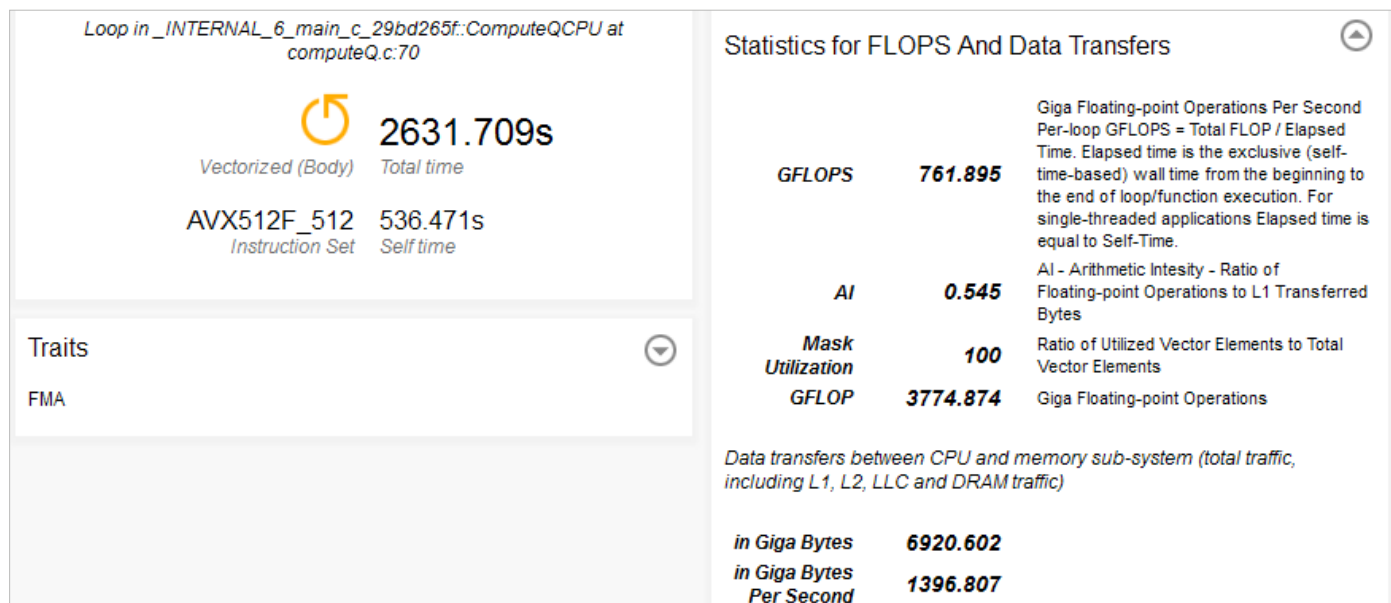


24 最終的な最適化後のインテル® Advisor のルーフライン・グラフ

Site Location	Loop Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name	Recommendations
[loop in_ZN4sd ...	No information available	100% / 0% / 0%	All unit strides	769KB	loop_site_27	

Memory Access Patterns Report		Dependencies Report	Recommendations					
ID	Stride	Type	Source	Nested Function	Variable references	Access Footprint	Modules	Site Name
P1	0	Parallel site information	rvalue_binary_operator_proxy_other_xmacro.h:50				omriq_exe_soa	loop_site_27
P3	0	Uniform stride	omriq_exe_soa:0x2d28	_svml_sincosf16	_svml_sincosf16_chosen_core_func	8B	omriq_exe_soa	loop_site_27
P4	0	Uniform stride	omriq_exe_soa:0x2d4f	_svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P5	0	Uniform stride	omriq_exe_soa:0x2d5e	_svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P6	0	Uniform stride	omriq_exe_soa:0x2d73	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P7	0	Uniform stride	omriq_exe_soa:0x2d7a	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P8	0	Uniform stride	omriq_exe_soa:0x2d87	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P9	0	Uniform stride	omriq_exe_soa:0x2d94	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P10	0	Uniform stride	omriq_exe_soa:0x2d9b	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P11	0	Uniform stride	omriq_exe_soa:0x2da2	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P12	0	Uniform stride	omriq_exe_soa:0x2db6	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P13	0	Uniform stride	omriq_exe_soa:0x2dbb	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P14	0	Uniform stride	omriq_exe_soa:0x2dd2	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P15	0	Uniform stride	omriq_exe_soa:0x2ddf	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P16	0	Uniform stride	omriq_exe_soa:0x2de6	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P17	0	Uniform stride	omriq_exe_soa:0x2df4	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P18	0	Uniform stride	omriq_exe_soa:0x2e13	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P19	0	Uniform stride	omriq_exe_soa:0x2e1a	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P20	0	Uniform stride	omriq_exe_soa:0x2e21	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P21	0	Uniform stride	omriq_exe_soa:0x2e28	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P22	0	Uniform stride	omriq_exe_soa:0x2e40	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P23	0	Uniform stride	omriq_exe_soa:0x2e48	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P24	0	Uniform stride	omriq_exe_soa:0x2e55	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P25	0	Uniform stride	omriq_exe_soa:0x2e62	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P26	0	Uniform stride	omriq_exe_soa:0x2e7b	_svml_sincosf16_b3	_svml_ssincos_data	64B	omriq_exe_soa	loop_site_27
P27	0	Uniform stride	omriq_exe_soa:0x2e96	_svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P28	0	Uniform stride	omriq_exe_soa:0x2e9e	_svml_sincosf16_b3		64B	omriq_exe_soa	loop_site_27
P29	1	Unit stride	computeQ.c:72	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P30	1	Unit stride	computeQ.c:73	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P31	1	Unit stride	computeQ.c:79	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P32	1	Unit stride	computeQ.c:80	ComputeQCPU		769KB	libiomp5.so; omriq_exe_soa	loop_site_27
P33	1	Unit stride	rvalue_binary_operator_proxy_other_xmacro.h:50			769KB	libiomp5.so; omriq_exe_soa	loop_site_27

25a 最終的な最適化後のインテル® Advisor のメモリー・アクセス・パターン・レポート



25b 最終的な最適化後のインテル® Advisor の [Code Analytics (コード解析)] タブ

まとめ

ルーフライン・モデルは、新しく、視覚的に直感的で、強力な方法でアプリケーションのパフォーマンスを表示します。ルーフライン・グラフ上のアプリケーションの位置に応じて適切な最適化手法を利用することで、パフォーマンスに影響しない最適化に貴重な時間を費やさずに済みます。ルーフライン・モデルは、次の質問に対する答えを提供します。

- パフォーマンスを向上できますか？
- 主なパフォーマンス・ボトルネックの原因はメモリーですか？それとも CPU ですか？
- 特定のボトルネックを最適化したら、どれぐらいのスピードアップが得られますか？
- 別のプラットフォームを使用したら、どれぐらいのスピードアップが得られますか？

システムが大規模になり、複雑さが増す中、これらの質問に対する答えを得ることは重要です。ルーフライン解析は、そのための時間と労力を軽減します。

コードの現代化

- ハードウェアの性能を最大限に引き出すため、ベクトル化とスレッド化によりコードを現代化する必要があります。
- この記事で示したような体系的なアプローチを採用し、**インテル® Parallel Studio XE** の強力なツールを利用することで、現代化の作業を劇的に軽減できます。
- インテル® Parallel Studio XE 2017 Update 1 で利用可能なインテル® Advisor のルーフライン解析が役立ちます。
- 現在開発中の新機能に関する最新情報を受け取りたい方は、**vector_advisor@intel.com** までメールにてご連絡ください。

インテル® Advisor の関連情報 (英語)

[インテル® Advisor のルーフライン機能の使い方](#)

[セルフ時間ベースの FLOPS 計算](#)と入れ子のループのルーフライン結果の解釈に関する重要な説明

[インテル® Advisor でインテル® MPI ライブラリーを使用するアプリケーションを解析する](#)

参考資料 (英語)

Stone, S. S.; J. P. Haldar, S. C. Tsao, W. W. Hwu., Z. Liang, and B. P. Sutton. "Accelerating advanced MRI reconstructions on GPUs." In International Conference on Computing Frontiers, pages 261–272, 2008.



インテル® ADVISOR を評価する
インテル® PARALLEL STUDIO XE に含まれます >



インテルが推進するディープラーニング・フレームワーク

さらなる洞察を提供

Pubudu Silva インテル コーポレーション マシンラーニング / コンピューター・ビジョン シニア・ソフトウェア・エンジニア

人工知能 (AI) は、本来は人の知能が必要な、視覚的理解、音声認識、言語処理、意思決定のようなタスクを実行することができる知能機械 (コンピューター) の概念であり、コンピューターの登場以来、次の大きな挑戦であると考えられています。

マシンラーニングは、いくつかの重要な AI タスクを実行するのに非常に有効であることが分かっています。人工ニューラル・ネットワーク (ANN) は、哺乳類の脳皮質の神経細胞構造の数学モデルであり、その遠大な設計とさまざまなタスクに対する汎用性により、特に AI 向きであると考えられていました。ANN の強みは、隠れた過渡状態 (隠れたノード) を学習して保守する能力にあります。この能力により、いくつかの非線形関数をカスケードして、入力から目的の出力まで、広い範囲のマップを学習することが可能です。

学習済みの ANN では、隠れた層は階層的な段階でデータの内部的な抽象化を表します。深い層ほど、より高いレベルの抽象化を表します。哺乳類の脳は複数の階層的な処理層で情報を処理すると考えられています (例えば、霊長類の視覚システムでは、処理が連続した段階で行われ、エッジ検出からプリミティブ形状検出を経て、徐々に複雑な視覚形状に移動します)¹。したがって、AI 研究には多層の「深い」ANN が望まれます。

深いカスケードされた層を備え複数の連続した段階でデータを処理するネットワークは、一般に「ディープ・ネットワーク」と呼ばれます。サポート・ベクトル・マシン (SVM)、混合正規分布 (MoG)、k 近傍法 (kNN)、主成分分析 (PCA)、カーネル密度推定 (KDE) のような、最も広く利用されているマシンラーニング・アルゴリズムには、3 つを超える処理層は含まれていません。したがって、これらは「浅い」アーキテクチャーと見なすことができます。2 層から 3 層の ANN は、うまくトレーニングすることができます。20 世紀の終わりに、より深い ANN をトレーニングするいくつかの試みが行われましたが、次の 2 つの問題に直面したことで失敗に終わりました。

1. 勾配消失問題
2. 多層化による重みの増加から生じる過学習

コンピューティングの進歩とともに、研究者は短時間に膨大なデータサンプルを使用してマシンラーニング・アルゴリズムのトレーニングを行うことが可能になり、過学習問題はほぼ解決しました。畳み込みニューラル・ネットワーク (CNN) は、多層型のディープ・ネットワークですが、重み共有の概念により (同等の深さの全結合 ANN よりも) 小さな多くの重みを含んでいます。このため、CNN は ANN よりもはるかに容易にトレーニングを行うことができます。CNN の理論的なパフォーマンスは ANN にはわずかに及びませんが、教師あり画像認識タスクでは最も普及している手法です。2006 年のジェフリー・ヒントン教授などによる画期的な提案² のおかげで、ディープ・ビリーフ・ネットワーク (DBN) やディープ・オート・エンコーダーのようなディープ・ネットワークの手法は教師なし学習にも利用されています。

一般に、ディープ・ネットワークは、分類、回帰、画像認識、自然言語処理などのほとんどの AI 関連タスクで、ほかのマシンラーニング・アルゴリズムよりも高いパフォーマンスを示します。ディープ・ネットワークの大いなる成功は、特徴階層の自律学習によるものです。

ディープラーニングと従来の統計的学習手法の大きな違いは、前者はローデータを学習するのに対して、後者はデータの間工学的特徴を学習する点です。DNN 学習の初期レベルで、ディープ・ネットワークは、指定されたタスクに適した特徴を自律的に生成します。目的関数に基づく最適化プロセスが、指定されたオリジナルデータを用いてすべての学習タスクを行うため、学習プロセスから推測や人間による判断の偏りが取り除かれます。

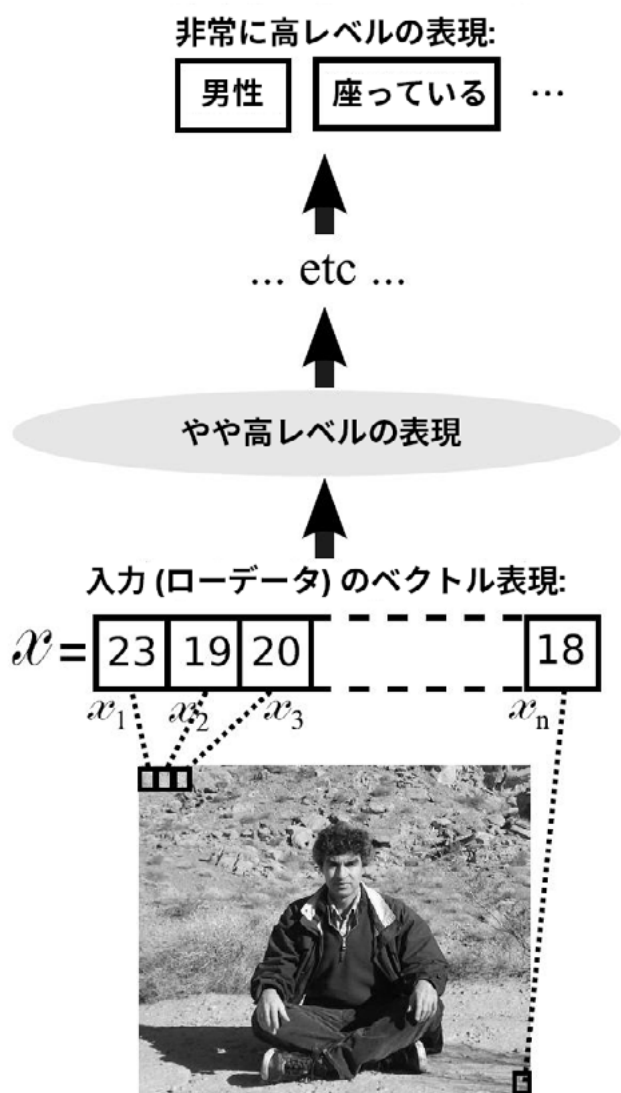
深い階層構造により、特徴の階層的な学習 (ネットワークの前のレベルで学習した低いレベルの特徴に基づいて、深いレベルではより高いレベルの特徴を学習する) が可能になります。

図 1 は、入力画像が徐々にディープ・ネットワークのより高いレベルの表現に変換される様子を示しています。ネットワークの深くに移動するごとに、画像はより抽象的な表現になります。例えば、特徴の階層的な学習は、初期の層から深い層へ、エッジ、形状、オブジェクトの一部、完全なオブジェクト、シーンの順で行われます。しかし、実際には、すべてを学習することなく、これらの階層的な抽象化層の「正しい」特徴ベクトルを推測することは困難です。これは、人間工学的特徴を学習する際に問題となります。一般に、深いネットワークでは、

出力層は非常に高レベルの特徴を処理して、浅いネットワークよりもはるかに高いレベルの概念を学習できるようにします。

さまざまなディープラーニングの手法が開発者、データ・サイエンティスト、研究者の標準ツールになったように、ディープ・ネットワークのトレーニングとスコア付けを支援するため、多くのディープラーニング・フレームワーク (Caffe*、Tensorflow*、Theano*、Torch など) およびライブラリー (MatConvNet、CNTK、Pylearn2、Deeplearning4j) が開発されています。これらのフレームワークとライブラリーは、退屈で単純な作業を減らすのに大いに役立ちます。ユーザーは、個々のコンポーネントの実装ではなく、ディープラーニングのさまざまな面に労力をかけることができます。

さらに、フレームワークとライブラリーは開発者コミュニティの協力の下で、オープンソース・プロジェクトとして開始されることが多いため、ユーザーはフレームワークとライブラリーのコードベースにアクセスすることができます。ディープラーニングでは、完了まで数日から数週間かかる、非常に大きなデータセットでトレーニングが行われるため、一般的に使用されるディープラーニング・ソフトウェアのパフォーマンス最適化が非常に重要です。



1 入力画像を徐々に変換してより高いレベルの表現にするディープ・ネットワーク処理
Source: Yoshua Bengio, Learning Deep Architectures for AI, 2009

インテルは、特にインテル® アーキテクチャー向けに最適化することにより、さまざまなオープンソースのディープラーニング・フレームワークの普及に貢献しています。[マシンラーニング・サイト](#) (英語) では、インテルによるマシンラーニングとディープラーニングの取り組みに関する最新の情報を提供しています。パフォーマンス最適化ツールと手法の詳細も、このサイトで提供しています。最適化に関するいくつかの情報は、独自のディープラーニング・アプリケーションを開発する際、およびフレームワークやライブラリーを開発サイクルで使用する際にソフトウェア開発者の役に立つように、ケーススタディーとして公開されています。

例えば、Caffe* の最適化で行われたプロセスは、[インテル® アーキテクチャー向けに最適化された Caffe*: モダンコード手法の適用](#) (英語) で示されています。[インテル® VTune™ Amplifier XE](#) は、CPU、キャッシュ、CPU コア、メモリーの使用状況、スレッドのロードバランス、スレッドのロックなど、パフォーマンス最適化の初期ガイダンスとして使用できる、さまざまな洞察を提供する強力なプロファイルツールです。[インテル® マスカーネル・ライブラリー \(インテル® MKL\)](#)、[インテル® スレディング・ビルディング・ブロック \(インテル® TBB\)](#)、OpenMP* のようなライブラリーは、ディープラーニング・ソフトウェアの最適化に非常に役立つことが分かっています。

ディープラーニングの開発と研究を促進するため、インテルは、ディープラーニング・ソリューションの開発、トレーニング、配置を支援する、データ・サイエンティストとソフトウェア開発者向けの無料のツールセットとして、[インテル® Deep Learning SDK](#) (英語) を提供しています。インテル® Deep Learning SDK は、Ubuntu*/CentOS* サーバーに接続された Web ベースのクライアントとして設計されています。インストール・ウィザードは、SDK とサーバーのインテル® アーキテクチャー向けに最適化された一般的なディープラーニング・フレームワークをインストールします。トレーニング・ツールは、単純なグラフィカル・ユーザー・インターフェイスと高度な視覚化手法により、トレーニング・データの準備、モデルの設計、モデルの選択を大幅に単純化します。配置ツールは、モデル圧縮と重み量子化手法により、特定のターゲットデバイス向けにトレーニングされたディープラーニング・モデルを最適化します。

参考資料 (英語)

1. Serre, T.; Kreiman, G.; Kouh, M.; Cadieu, C.; Knoblich, U.; and Poggio, T. 2007. “A quantitative theory of immediate visual recognition.” *Progress in Brain Research, Computational Neuroscience: Theoretical Insights into Brain Function*, 165, 33–56.
2. Hinton, G. E.; Osindero, S.; and Teh, Y. 2006. A fast learning algorithm for deep belief nets. *Neural Computation*, 18, 1527–1554.

さらなる洞察を提供

intel.com (英語) では、インテル向けに最適化されたディープラーニング・フレームワーク、ライブラリー、インテル® Deep Learning SDK のようなディープラーニング・ツールについての最新情報を提供しています。

**インテル® DEEP LEARNING SDK と
その他の AI ソフトウェア・ツールの詳細 (英語) >**

インテルの技術ウェビナー (英語) で 最新の情報を入手

インテル® ソフトウェアのエキスパートが提供する最新のヒントとコツでコードのパフォーマンスを向上しましょう。ホットなトピックについて取り上げた無料の技術ウェビナーをぜひご覧ください。

インテル® Parallel Studio XE 2017:
高速なコードを素早く開発、

対応ハードウェアの有無に関係なく
インテル® AVX-512 向けに最適化、

アプリケーションを共有 / 分散
メモリーでスケーリング、

マシンラーニングにおける Python*
パフォーマンスの壁を乗り越える、

インテル® Xeon Phi™ プロセッサ
の高帯域メモリー対応のコード、

ベクトル化: 活用すべき "別の" 並列
処理、

ルーフライン解析: パフォーマンス
最適化のトレードオフを視覚化する
新しい方法、

最新のインテル® プロセッサに
対応したメディア・ソリューション /
アプリケーションの開発、

インテル® ライブラリーによる
ディープラーニングとマシンラー
ニングの促進、

インテル® HPC オーケストレーター:
インテル® スケーラブル・システム・
フレームワークのビルディング・ブ
ロックを提供するシステム・ソフト
ウェア・スタック、

2017 年春に公開予定の新しい技術ウェビナーにご期待ください。
過去の技術ウェビナーは[こちら \(英語\)](#) でご覧になれます。

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。

© 2017 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Xeon、Intel Xeon Phi は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。



THE PARALLEL UNIVERSE

© 2017 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、3D XPoint、Xeon、Intel Xeon Phi、VTune は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。

Microsoft および Windows は、米国 Microsoft Corporation の、米国およびその他の国における登録商標または商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。