インテル® AVX-512 でベクトル化のパフォーマンスを向上する

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「Improve Vectorization Performance with Intel® AVX-512」の日本語参考訳です。

この記事では、データの依存関係により、これまでインテル® C++ コンパイラーによってベクトル化されず、インテル® Xeon Phi™ プロセッサーのインテル® 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) 命令を利用することでベクトル化されるようになったループの例を示します。新しい命令セットを使用することで、コンパイラーがループをベクトル化可能と自動認識する理由を調査し、ベクトル化のパフォーマンスの問題について述べます。

はじめに

コードを最適化する場合、最初にベクトル化に注目すべきです。現代のプロセッサーのリソースを効率良く活用する最も根本的な方法は、ベクトルレジスターや SIMD (Single Instruction Multiple Data) 命令などのハードウェア機能を利用してベクトル化モードで実行可能なコードを記述することです。最適化プロセスの段階では、アルゴリズムやコードでデータ並列処理を実装します。

ベクトル化により細粒度の並列性を最大限に引き出すことで、マルチスレッドやマルチタスクにより、プロセッサー・コア数に応じてソフトウェア・アプリケーションのパフォーマンスをスケールすることができます。ベクトル化とスレッド化の併用は相乗効果をもたらすため、マルチスレッド・アプリケーション全体のパフォーマンスにおいて、シングルコア・リソースを効率良く利用することは不可欠です。

新しいインテル® Xeon Phi™ プロセッサーは、512 ビット幅のベクトルレジスターを備えています。インテル® Xeon Phi™ プロセッサー (および将来のインテル® プロセッサー) でサポートされるインテル® AVX-512 命令セット・アーキテクチャー (ISA) は、ベクトルレベルの並列処理を提供するため、ソフトウェアはコアごとに 2 つのベクトル・プロセシング・ユニット (VPU) を使用できます (各 VPU は同時に 16 個の単精度 (32 ビット) または 8 個の倍精度 (64 ビット) 浮動小数点数を処理できます)。これらのハードウェアおよびソフトウェア機能を利用することが、インテル® Xeon Phi™ プロセッサーを最適に使用するための鍵です。

この記事では、インテル® Xeon Phi™ プロセッサーでインテル® AVX-512 ISA を利用する方法を説明します。イメージ処理アプリケーションの例を用いて、インテル® C++ コンパイラーが、インテル® AVX2 ではベクトル化しなかったループをインテル® AVX-512 では自動的にベクトル化する仕組みを示します。ベクトル化されたコードにより生じるパフォーマンスの問題についても述べます。

インテル® AVX-512 ISA は、いくつかのサブセットで構成されています。一部のサブセットは、インテル® Xeon Phi™ プロセッサーで利用できます。新しいインテル® Xeon® プロセッサーで利用可能なサブセットもあります。インテル® AVX-512 のサブセットとさまざまなインテル® プロセッサーでの対応状況については、(Zhang, 2016) に詳しい説明があります。

この記事では、インテル® Xeon Phi™ プロセッサーとインテル® Xeon® プロセッサーで利用可能なベクトル化機能を提供するインテル® AVX-512 ISA のサブセットに注目します。これらのサブセットには、インテル® AVX-512 基本命令 (インテル® AVX-512F) サブセット (ベクトル命令と新しい 512 ビット・ベクトル・レジスターを利用するためのコア機能を提供する) とインテル® AVX-512 競合検出命令 (インテル® AVX-512CD) サブセット (ベクトルのデータ競合を検出する命令を追加し、データの依存関係が存在する特定のループをベクトル化する) が含まれます。

ベクトル化手法

インテル® Xeon Phi™ プロセッサー・コアのベクトル化機能を利用する方法はいくつかあります。

- 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/ベクトル化されたライブラリーを使用します。
- コンパイラーが、ハードウェアで利用可能なベクトル命令を使用して対応するバイナリーコードを生成するように (これは、一般に自動ベクトル化と呼ばれる)、ベクトル化可能な高水準コードを記述します。
- 言語拡張 (コンパイラーの組込み関数) を使用したり、アセンブリー言語でベクトル命令を直接呼び出します。

それぞれの手法にはメリットとデメリットがあり、どの手法を使用するかは特定のケースに依存します。ここでは、移植性を高め、将来のプロセッサーに対応できるベクトル化可能なコードの記述に注目します。新しいインテル® AVX-512 命令セットを利用して、インテル® Xeon Phi™ プロセッサーのベクトルモードで実行するコードを作成する単純な例 (ヒストグラム) を検証します。この例の目的は、インテル® AVX-512 ISA を利用することで、コンパイラーがインテル® AVX2 などの以前の命令セットではベクトル化できなかったデータの依存関係を含むソースコードをベクトル化できるようになった理由を明らかにすることです。インテル® AVX-512 ISA の詳細は、(Intel, 2016) を参照してください。

言語拡張とコンパイラーの組込み関数を利用して明示的にベクトル化する手法については、今後の記事で述べる予定です。これらの手法は、複雑なフローやデータの依存性によりコンパイラーが安全にベクトル化できないループで役立ちます。ここで使用する比較的単純な例は、コンパイラーがインテル® AVX-512 ISA の新機能を使用していくつかの一般的なループ構造のパフォーマンスを向上する方法を理解するのに適しています。

例: イメージのヒストグラム計算

インテル® AVX-512F およびインテル® AVX-512CD サブセットの新機能を理解するため、イメージのヒストグラムを計算する例を使用します。

イメージのヒストグラムは、イメージのピクセル値の分布を視覚的に表現したものです (Wikipedia, n.d.)。ピクセル値は、グレースケール値を表すスカラー、または RGB イメージ (赤、緑、青の 3 つの値を組み合わせてカラーを表現する) のカラーを表す値を含むベクトルです。

ここでは、3024 x 4032 のグレースケール・イメージを使用します。このイメージの合計ピクセル数は 12,192,768 です。オリジナルイメージと対応するヒストグラム (1 ピクセル 256 階調のグレースケール強度で計算) を図 1 に示します。

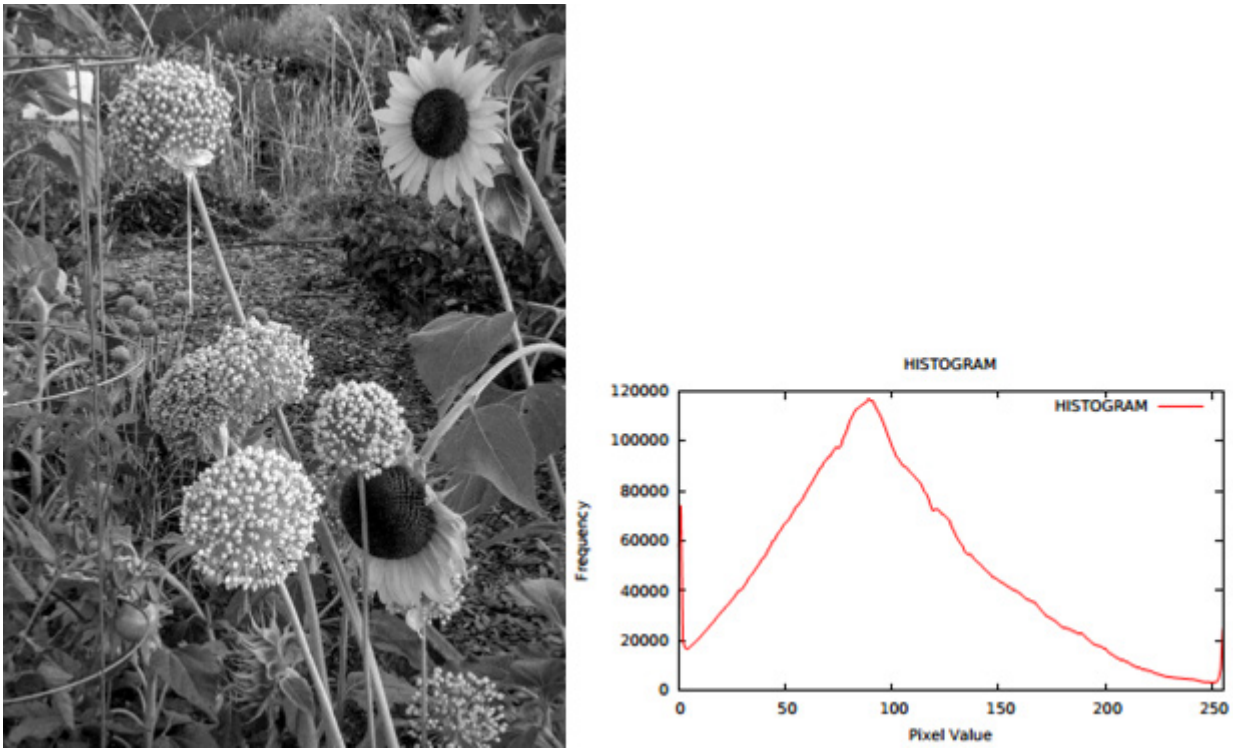

図 1: 使用イメージ (出典: Alberto Villarreal) と対応するヒストグラム

ヒストグラムを計算する基本アルゴリズムは、次のとおりです。

1. イメージを読み取ります。
2. イメージの行数と列数を取得します。
3. イメージ配列 image [1: rows x columns] をイメージのピクセル値に設定します。
4. ヒストグラム配列 histogram [0: 255] をゼロに設定します。
5. イメージの各ピクセルに対して、以下の処理を行います。

```
{
        histogram [ image [ pixel ] ] = histogram [ image [ pixel ] ] + 1
}
```

基本アルゴリズムでは、イメージ配列がヒストグラム配列へのインデックスとして使用されています (整数への型変換が推定されます)。イメージの隣接するピクセルが同じ強度の場合、ループの複数の反復を同時に処理すると正しい結果が得られない可能性があるため、このような間接参照は無条件に並列化できません。

次のセクションでは、このアルゴリズムを C++ で実装して、インテル® AVX-512 ISA を利用する場合、コンパイラーがこの構造を安全にベクトル化できることを示します (部分的にのみベクトル化され、パフォーマンスはイメージデータに依存します。)。

ここで示す実装は、この記事での説明を目的としたものであり、ヒストグラム計算の効率良い実行方法を保証するものではありません。パフォーマンス向上が目的の場合は、効率良いライブラリーが提供されています。また、ここでは単純なコードを使用して、新しいインテル® AVX-512 ISA がベクトル化の可能性を追加する方法を示し、新しいインテル® AVX-512 ISA の新機能を理解することを目的としています。

ヒストグラム計算の特定のケースに応じて、並列処理を実装する方法はほかにもあります。例えば、(Colfax International, 2015) では、ストリップマイニング手法によりコードを変更して、同様のアルゴリズム (ビニング・アプリケーション) を自動ベクトル化する方法を説明しています。

ハードウェア

アプリケーションのテストには、以下のシステムを使用します。

プロセッサー: インテル® Xeon Phi™ プロセッサー 7250 (1.40GHz)
コア数: 68
スレッド数: 272

Linux* システムでは、次のコマンドでこの情報を確認できます。

```
cat /proc/cpuinfo.
```

コマンドを実行すると、出力の "flags" セクションに "avx512f" と "avx512cd" プロセッサー・フラグが表示されます。これらのフラグは、このプロセッサーでインテル® AVX-512F およびインテル® AVX-512CD サブセットがサポートされていることを示します。"avx2" プロセッサー・フラグも表示されます。これは、インテル® AVX2 ISA もサポートされていることを示します (ただし、インテル® AVX2 ISA は、このプロセッサーの 512 ビット・ベクトル・レジスター・サポートを利用できません)。

インテル® C++ コンパイラーによるベクトル化の結果

このセクションでは、ヒストグラム・コードの一部の基本ベクトル化を解析します。具体的には、このコードの 2 つの異なるループを解析します。

ループ 1: ヒストグラム計算のみ実装するループ。入力イメージのヒストグラムを計算して、単精度浮動小数点形式で配列 image1 に格納します。

ループ 2: 畳み込みフィルターとヒストグラム計算を実装するループ。配列 image1 にあるオリジナルイメージにフィルターを適用して、配列 image2 に格納されているフィルター済みイメージの新しいヒストグラムを計算します。

以下は、上記の 2 つのループから成るコードです (イメージとヒストグラム・データはアライメントされた配列に配置済みです)。

```
// ループ 1
#pragma vector aligned
for (position=cols; position<rows*cols-cols; position++)
{
        hist1[ int(image1[position]) ]++;
}

(…)
```

```
// ループ 2

#pragma vector aligned
for (position=cols; position<rows*cols-cols; position++)
{
           if (position%cols != 0 || position%(cols-1) != 0)
{
image2[position] = ( 9.0f*image1[position]
                                                - image1[position-1]
                                                - image1[position+1]
                                                - image1[position-cols-1]
                                                - image1[position-cols+1]
                                                - image1[position-cols]
                                                - image1[position+cols-1]
                                                - image1[position+cols+1]
                                                - image1[position+cols]) ;
                }
               if (image2[position] >= 0 && image2[position] <= 255)
               hist2[ int(image2[position]) ]++;
 }
```

インテル® C++ コンパイラーで次のオプションを指定してこのコードをコンパイルし、最適化レポートを生成します。

```
icpc histogram.cpp -o histogram -O3 -qopt-report=2 -qopt-report-phase=vec -xCORE-AVX2  …
```

ここでは、-xCORE-AVX2 コンパイラー・オプションを使用して、コンパイラーにインテル® AVX2 ISA を利用して実行コードを生成するように指示しています。

以下は、コンパイラーにより生成される最適化レポートのループに関連する部分の抜粋です。
ループの開始 histogram.cpp(92,5)
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。最初の依存関係を以下に示します。詳細については、レベル** 5 **のレポートを使用してください。**
リマーク #15346: **ベクトル依存関係:** FLOW **の依存関係が行** 94 **と行** 94 **の間に仮定されました。**
ループの終了

ループの開始 histogram.cpp(92,5)
<剰余>
ループの終了

ループの開始 histogram.cpp(103,5)
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。最初の依存関係を以下に示します。詳細については、レベル** 5 **のレポートを使用してください。**
リマーク #15346: **ベクトル依存関係:** FLOW **の依存関係が行** 118 **と行** 118 **の間に仮定されました。**
ループの終了

上記の最適化レポートの抜粋から、ヒストグラム計算のコード行 (行 94 と行 118) に依存関係が存在するため、コンパイラーはどちらのループもベクトル化しなかったことが分かります。

次に、コンパイラーにインテル® AVX-512 ISA の使用を指示するため、-xMIC-AVX512 コンパイラー・オプションを指定してコードをコンパイルします。

```
icpc histogram.cpp -o histogram -O3 -qopt-report=2 -qopt-report-phase=vec -xMIC-AVX512  …
```

両方のループがベクトル化されたことを示す、次のような最適化レポートが出力されます。

```
ループの開始 histogram.cpp(92,5)
    リマーク #15300: ループがベクトル化されました。

    ループの開始 histogram.cpp(94,8)
        リマーク #25460: ループの最適化はレポートされませんでした。
    ループの終了
ループの終了

ループの開始 histogram.cpp(92,5)
<ベクトル化の剰余ループ>
    リマーク #15301: 剰余ループがベクトル化されました。

    ループの開始 histogram.cpp(94,8)
        リマーク #25460: ループの最適化はレポートされませんでした。
    ループの終了
ループの終了

ループの開始 histogram.cpp(103,5)
    リマーク #15300: ループがベクトル化されました。

    ループの開始 histogram.cpp(118,8)
        リマーク #25460: ループの最適化はレポートされませんでした。
    ループの終了
ループの終了

ループの開始 histogram.cpp(103,5)
<ベクトル化の剰余ループ>
    リマーク #15301: 剰余ループがベクトル化されました。
```

コンパイラー・レポートの結果は、次のように要約できます。

- ヒストグラム計算を実装するループ 1 は、インテル® AVX2 コンパイラー・オプションでは、依存関係の可能性が存在するためベクトル化されませんでした (前述のセクション 3 を参照)。しかし、インテル® AVX-512 コンパイラー・オプションではベクトル化されました。これは、コンパイラーがインテル® AVX-512 ISA を利用して依存関係を解決したことを意味します。
- ループ 2 もループ 1 と同じ診断になりました。2 つのループの違いは、ループ 2 ではヒストグラム計算に加えて、フィルター処理を行っています。このフィルター処理は、依存関係が存在しないため、ベクトル化を妨げません。インテル® AVX2 コンパイラー・オプションを指定した際にループ 2 がベクトル化されない理由は、ヒストグラム計算がループ全体のベクトル化を妨げているからです。

注: 上記の最適化レポートから、コンパイラーはループをメインループと剰余ループの 2 つに分割することが分かります。剰余ループには、ループの最後のいくつかの反復 (ベクトルユニットを満たすことができなかった反復) が含まれます。ループの総反復数がベクトル長の倍数であることが分かっている場合を除き、コンパイラーは通常この分割を行います。

ここでは、剰余ループについては触れません。剰余ループを排除してパフォーマンスを向上する方法は、文献で紹介されています。

コードのパフォーマンスを解析する

インテル® AVX2 とインテル® AVX-512 ISA を利用する実行ファイル間の各ループの実行時間を比較するため、各ループの始めと終わりにタイミング命令を追加して、コードのパフォーマンスを解析しました。

以下の表は、前処理をしていない入力イメージを使用して、コードのベクトル化されたバージョンとされていないバージョンをシングルコアで 5 回ずつ実行した結果の平均です。ここでは、インテル® AVX2 コンパイラー・オプションを使用して生成したベクトル化されていないコードの結果をベースライン・パフォーマンスとしています。

テストケース	ループ	ベースライン (インテル® AVX2)	ベクトル化によるスピードアップ (インテル® AVX-512)
入力イメージ	ループ 1	1	2.2
	ループ 2	1	7.0

さらに、入力データの関数としてコードのパフォーマンスを解析するため、ブラーフィルターとシャープフィルターを使用して入力イメージの前処理を行いました。ブラーフィルターはイメージを滑らかにし、シャープフィルターはイメージを鮮明にします。これらのフィルターは、イメージ処理ライブラリーやコンピューター・ビジョン・ライブラリーで提供されています。ここでは、OpenCV* ライブラリーを使用してテストイメージの前処理を行いました。

以下の表は、3 つのテストの結果です。

テストケース	ループ	ベースライン (インテル® AVX2)	ベクトル化によるスピードアップ (インテル® AVX-512)
入力イメージ	ループ 1	1	2.2
	ループ 2	1	7.0
鮮明な入力イメージ	ループ 1	1	2.6
	ループ 2	1	7.4
滑らかな入力イメージ	ループ 1	1	1.7
	ループ 2	1	5.6

上記の結果から、次の疑問が生じます。

1. インテル® AVX-512 コンパイラー・オプションではベクトル化され、インテル® AVX2 コンパイラー・オプションではベクトル化されない理由は?
2. インテル® AVX-512 ISA によりベクトル化されたループ 1 のスピードアップが、512 ビット・ベクトル使用時の理論的なスピードアップと比べて小さい理由は?
3. イメージを前処理した場合、ベクトル化されたコードのパフォーマンス・ゲインが異なる理由は? 具体的には、ベクトル化されたコードのパフォーマンスが、鮮明なイメージでは向上し、滑らかなイメージでは低下する理由は?

次のセクションでは、インテル® AVX-512 ISA のサブセットの 1 つである、インテル® AVX-512CD (競合検出) サブセットの説明を通してこれらの疑問に答えていきます。

インテル® AVX-512CD サブセット

インテル® AVX-512 ISA のインテル® AVX-512CD (競合検出) サブセットは、ベクトルレジスターのデータ競合を検出します。つまり、ベクトルオペランド内の同一要素を検出する機能を提供します。検出結果は、ベクトル演算に使用されるマスクベクトルに格納されるため、ヒストグラム処理 (ヒストグラム配列の更新) は異なる配列要素 (イメージのピクセル値) に対してのみ実行されます。

インテル® AVX-512CD サブセットの動作を検証するため、インテル® C++ コンパイラーの -S オプションを使用してアセンブリー・コードを生成します。

```
icpc example2.cpp -o example2.s -O3 -xMIC-AVX512 -S …
```

上記のコマンドは、実行ファイルの代わりに、C++ ソースコードに対応するアセンブリー・コードを含むテキストファイルを生成します。前述の例のループ 1 のソースコード行 94 (ヒストグラムの更新) を実装するアセンブリー・コードを見てみます。

```
vcvttps2dq (%r9,%rax,4), %zmm5                  #94.19 c1
vpxord     %zmm2, %zmm2, %zmm2                  #94.8 c1
kmovw      %k1, %k2                             #94.8 c1
vpconflictd %zmm5, %zmm3                        #94.8 c3
vpgatherdd (%r12,%zmm5,4), %zmm2{%k2}           #94.8 c3
vptestmd   %zmm0, %zmm3, %k0                    #94.8 c5
kmovw      %k0, %r10d                           #94.8 c9 stall 1
vpaddd     %zmm1, %zmm2, %zmm4                  #94.8 c9
testl      %r10d, %r10d                         #94.8 c11
je         ..B1.165      # Prob 30%             #94.8 c13
```

上記のコードで、vpconflictd はソース・ベクトル・レジスター (ピクセル値) の要素を比較して競合を検出し、その結果をビット・ベクトルとしてデスティネーションに書き込みます。マスクベクトルを使用して、この結果をさらにテストして、ヒストグラムの更新でベクトルレジスターのどの要素が同時に使用されるかを定義します (vpconflictd 命令はインテル® AVX-512CD サブセットの一部で、vptestmd 命令はインテル® AVX-512F サブセットの一部です。これらのサブセットの詳細は、インテル® AVX-512 ISA ドキュメント (Intel, 2016) を参照してください)。このプロセスを図 2 と図 3 に示します。

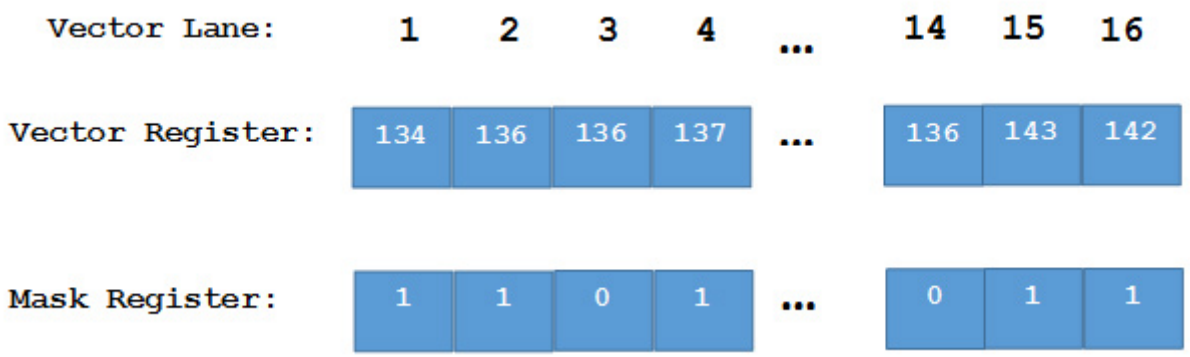

図 2: 配列内のピクセル値 (滑らかなイメージ)

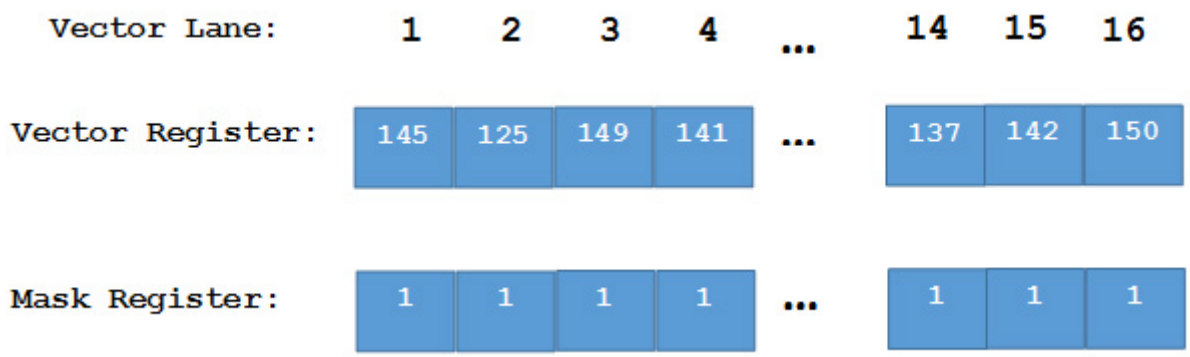

図 3: 配列内のピクセル値 (鮮明なイメージ)

図 2 は、イメージのいくつかの隣接するピクセルの値が同じ場合です。ベクトルレジスターに格納された配列 image1 の値が異なる要素のみ、ヒストグラムの並列更新で使用されます。つまり、競合しない要素のみヒストグラムの並列更新に使用されます。競合する要素は、別の機会のヒストグラムの更新に使用されます。

このケースでは、パフォーマンスはイメージの滑らかさに依存します。最悪のケースは、ベクトルレジスターのすべての要素の値が同じ場合で、ループの最後がスカラーモードで処理されるだけでなく、競合検出とテスト命令のオーバーヘッドにより、パフォーマンスが大幅に低下します。

図 3 は、イメージを鮮明にした場合です。このケースでは、ベクトルレジスターの隣接するピクセルの値は異なる可能性が高いくなります。ベクトルレジスターのほとんどまたはすべての要素がヒストグラムの更新に使用されるため、より多くの要素が並列に処理され、ループのパフォーマンスが向上します。

最高のパフォーマンスは、配列のすべての要素が異なる場合に得られます。ただし、最高のパフォーマンス (ここでは 16 倍) でも、競合検出とテスト命令のオーバーヘッドにより、理論的なスピードアップを下回ります。

ここで述べたことから、セクション 5 の疑問に対する答えを導き出すことができます。

疑問 1 のインテル® AVX-512 コンパイラー・オプションを使用するとコンパイラーがベクトル化されたコードを生成する理由は、インテル® AVX-512CD およびインテル® AVX-512F サブセットには、ループの要素のサブセットの競合を検出して、安全にベクトル化できる競合しないサブセットを作成する新しい命令が含まれているからです。これらのサブセットのサイズはデータに依存します。インテル® AVX2 コンパイラー・オプションを使用する場合、インテル® AVX2 ISA には競合検出機能が含まれていないため、ベクトル化することができません。

疑問 2 のベクトル化されたコードのパフォーマンスが (理論的なスピードアップよりも) 低い理由は、競合検出とテスト命令の実行にはオーバーヘッドが伴うためです。このパフォーマンス・ペナルティーは、ヒストグラム更新のみを行うループ 1 に大きく影響します。

しかし、ループ更新以外の処理も行うループ 2 では、ベースラインよりもパフォーマンスが向上します。インテル® AVX-512 コンパイラー・オプションを使用すると、コンパイラーはヒストグラム計算で生じる依存関係を解決することで、ループ全体のパフォーマンスを向上します。インテル® AVX2 コンパイラー・オプションを使用する場合、ヒストグラム計算の依存関係によって、ループ内のほかの計算はベクトルモードでの実行を妨げられます。これは、インテル® AVX-512CD サブセットの使用によりもたらされる重要な結果と言えます。コンパイラーは、インテル® AVX-512 が登場する前は、ベクトル化されるようにコードを書き直す必要があったヒストグラムのような依存関係を含む複雑なコードをベクトル化できるようになりました。

疑問 3 については、競合検出メカニズムを使用する場合、ベクトル化されたループのパフォーマンスがデータに依存することは明らかです。図 2 と図 3 に示すように、ベクトルモードのスピードアップは、ベクトルレジスター内の異なる (競合しない) 値の数に依存します。ここで紹介した例では、鮮明なイメージは、滑らかなイメージと比較して、隣接するピクセルの値が類似する/同じになる可能性が低くなります。

まとめ

この記事では、メモリー競合により、インテル® C++ コンパイラーによってインテル® AVX2 以前の命令セットではベクトル化されず、インテル® Xeon Phi™ プロセッサーのインテル® AVX-512 ISA を利用してベクトル化されるようになった単純なループの例を示しました。特に、インテル® AVX-512CD およびインテル® AVX-512F サブセットの新機能 (インテル® Xeon Phi™ プロセッサーと将来のインテル® Xeon® プロセッサーでサポート) は、このようなアプリケーションに対し、コードを変更しなくても、コンパイラーがベクトルコードを自動生成できるようにします。しかし、コンパイラーはマスクレジスターを使用してベクトル化し、マスクレジスターの内容は隣接するデータの類似性に依存するため、一般に生成されるベクトルコードのパフォーマンスは、完全なベクトルモードで実行した場合を下回り、データに依存します。

この記事の目的は、インテル® AVX-512CD およびインテル® AVX-512F サブセットの新機能の利用を促すことです。今後の記事では、ベクトル化の効率を向上するため、マスクベクトルの更新ロジックを明示的に制御することでほかの複雑なループのベクトル化の可能性を探っていく予定です。

参考文献 (英語)

Colfax International.(2015)."Optimization Techniques for the Intel MIC Architecture.Part 2 of 3: Strip-Mining for Vectorization." Optimization Techniques for the Intel MIC Architecture.Part 2 of 3: Strip-Mining for Vectorization: http://colfaxresearch.com/optimization-techniques-for-the-intel-mic-architecture-part-2-of-3-strip-mining-for-vectorization/

Intel.(2016, February)."Intel® Architecture Instruction Set Extensions Programming Reference." https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf

Wikipedia.(n.d.).Wikipedia: https://en.wikipedia.org/wiki/Image_histogram

Zhang, B.(2016)."Guide to Automatic Vectorization With Intel AVX-512 Instructions in Knights Landing Processors."

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。

添付ファイル	サイズ
Histogram_Example.zip	2.84KB