モダンコード - ベクトル化されたリダクション 2: コンパイラーのベクトル化を活用する

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「Vectorized Reduction 2: Let the Compiler do that Voodoo that it do so well」の日本語参考訳です。

以前の記事「インテルのベクトル組込み関数からベクトル化されたリダクション・コードを記述する」で説明したコードは、複素数の 2 乗差を計算するループの仕上げの部分でした。C++ でコードを記述したとき、インテル® コンパイラーは主要な計算をベクトル化することができました。この計算は、ベクトルレジスターに格納された部分値を加算して最終的な解を生成しました (リダクション操作)。生成されたアセンブリー言語コードに目を通すことで、計算がどのように行われたか確認することにしました。

私は、大学 2 年のときにアセンブリー言語のコースをとりました。このコースは、6502 プロセッサー (Apple* II デスクトップ・システム) を使用して行われました。その後、後続のプロセッサー・アーキテクチャーのアセンブリー言語では、大幅に拡張された異なる命令セットとなったことを覚えています。私は、現在でもアセンブリー言語コードと高水準プログラミング言語のステートメントを区別することができます。また、アセンブリー・コードのいくつかの基本的な命令については何を行うためのものか分かります。しかし、ヘルプなしでできるのはここまでです。(これは私が高校で 2 年間学んだドイツ語に通じるものがあります。いくつかの単語の意味は正確に覚えていませんが、それらの単語を話すことはできます。)

そのため、生成された数百行のアセンブリー言語に目を通すことは、非常に大変な作業に思えました。幸い、**-S** オプションを指定してコードをコンパイルすると、インテル® コンパイラーはソースのアセンブリー・コードを生成します。また、そのアセンブリー・コード・ファイルにはソースコードの行番号で注釈が付けられます。このおかげで、リダクション操作を実装するコードの部分を比較的容易に見つけることができました。オリジナルの C++ コードに明示的なリダクションは含まれていませんでした。その部分は計算ループの最後であり、ここですべての部分解を最終的に計算された値を含むように指定された 1 つの変数に合計する必要があります。

私が行った作業の細かい説明は省き、4 つの倍精度浮動小数点値のリダクションを実行して (インテル® AVX を使用)、生成されたアセンブリー・コード・ファイルで指定された 1 つの値にする部分のコードの紹介に移ります。

```
..LN138:
   .loc    1  54  is_stmt 1
        vextractf128 $1, %ymm0, %xmm1                          #54.17
..LN139:
        vaddpd    %xmm1, %xmm0, %xmm2                          #54.17
..LN140:
        vunpckhpd %xmm2, %xmm2, %xmm3                          #54.17
..LN141:
        vaddsd    %xmm3, %xmm2, %xmm0                          #54.17
```

加算操作の 2 つは簡単に分かりますが、ほかの 2 つはやや分かりにくいでしょう。確かに、これらの操作は、私が記述したリダクション操作のデータを処理しているようには見えません。Google™ で検索すると、これらの不

明な命令の目的が明らかになるでしょう。しかし、奇妙なことに、私が見たサイトでは、すべてのパラメーターが、生成されたアセンブリー・コード・ファイルで示されている順序と逆になっていました。実際のコンテキストから、また、どうすべきか知っていたことため、私はその点を無視して、正確に解釈しました。

以前のブログで記述したバージョンで行ったように、このリダクション計算で 4 つの命令がそれぞれどのように使用されているか、順に説明します。また、どの値がプロセッサー内で保持および操作されているかを示す、視覚的な表現を追加しました (これらの名前は私が勝手に付けたものではなく、実際のベクトルレジスター名です)。倍精度値の表現には、ビリヤードの球の写真を使用しました。(なお、私は某ビリヤード場とタトゥーパーラーから生涯出入禁止になったため、読者の皆さんがこの写真を喜んでいただけると幸いです。)

リダクションの説明に移る前に、読者の皆さんに、一部の値を **%ymm0** (インテル® AVX レジスター) に移動している命令があることを伝えておかなければなりません。この移動は、私が確認しなかった形式のレジスターとアドレッシング・オフセット・モードを使用して、どこかからデータを操作しています。最終的な 4 つの部分和は **%ymm0** レジスターに格納されていると仮定します。この例では、私が記述したバージョンの説明で使用した同じ 4 つの値で始めます。 つまり、値 2.0、3.0、2.0、5.0 が **%ymm0** レジスターに格納されています。ビリヤードの球で示すと次のようになります。

部分 2 乗差の最後の計算が完了した後、最初に実行する命令は、次の命令です。

```
vextractf128 $1, %ymm0, %xmm1
```

この命令は、256 ビットのインテル® AVX レジスター **%ymm0** から 2 つの倍精度値 (128 ビット) を抽出して 128 ビットの SSE ベクトルレジスター **%xmm1** に格納します。**$1** パラメーターは抽出する値を制御するビットマスクです。(定数値を示すために “$” 表記を使用しています。) ビットマスクの値 0 は下位半分 (ビット 127:0) を、1 は上位半分 (ビット 255:128) を処理します。つまり、2 つの上位の値 (2.0 および 3.0) は **%xmm1** レジスターへ移動されます。

2 つ目に実行する命令は、単純な加算です。

```
vaddpd    %xmm1, %xmm0, %xmm2
```

加数の 1 つは、部分解の半分をロードした **%xmm1** レジスターで、和のデスティネーション・レジスター は **%xmm2** です。別の加数は、もう 1 つの SSE レジスター **%xmm0** です。これはどこから来たのでしょうか。

私は、このレジスターがどこでロードされ、何を含んでいるか調べるために、再びアセンブリー言語コードに目を通しましたが、何も分かりませんでした。

そこで、最後の 2 つのステートメントから、おそらく **%ymm0** レジスターと **%xmm0** レジスターの間に何らかの関係があると推測して、インテル® コンパイラーのサポートチームの知人に質問しました。すぐに、**%xmm0** レジスターは **%ymm0** レジスターの下位半分であるという答えがありました。また、『Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture』の 326 ページ (第 14 章 - Programming with AVX, FMA and AVX2) の図 14-1 に、**%xmm0** レジスターは **%ymm0** レジスターの下位半分であるという説明があることを教えてくれました。

この関係が明らかになったことで、この加算がオリジナルの部分和の 2 つと別の 2 つを合計して、さらに 2 つの部分解を生成していることが分かりました。下記の例は、**%xmm1** レジスターと **%xmm2** レジスターをインテル® AVX レジスター (**%ymm0** レジスターの **%xmm0** 部分) に拡張しています (**ymm** レジスターと **xmm** レジスターで背景色が異なっていることに注意してください)。計算に関連していない部分は灰色になるか手球 (無地の球) に置換されています。

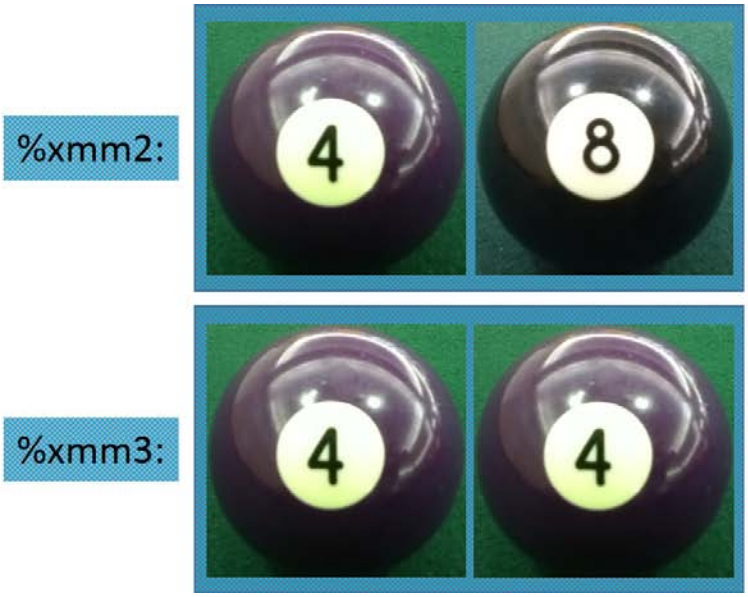
目標の半分まできました。次の命令は、別のデータ移動命令です。

```
vunpckhpd %xmm2, %xmm2, %xmm3
```

この命令は、あるレジスターから別のレジスターへの内容の並べ替えや単純な移動ではなく、2 つのレジスターの一部をデスティネーション・レジスターにインターリーブする「アンパック」命令です。このケースでは、2 つの **xmm** レジスターの上半分の単一値がデスティネーション **xmm** レジスターの上位および下位部分に配置されます。命令名の 'h' は「上位」ビット ('l' は「下位」ビット) を表すため、上位であることを確認します。

上記の命令の 2 つのソースレジスターはどちらも **%xmm2** で、デスティネーション・レジスターは **%xmm3** です。**%xmm2** の上位半分に格納された値は、**%xmm3** の 2 つの値に複製されます。この命令を実行した後、

ソースレジスターとデスティネーション・レジスターの内容は次のようになります。(この命令は 128 ビット **xmm** レジスターのみ処理するため、レジスターの該当部分のみ示しています。)



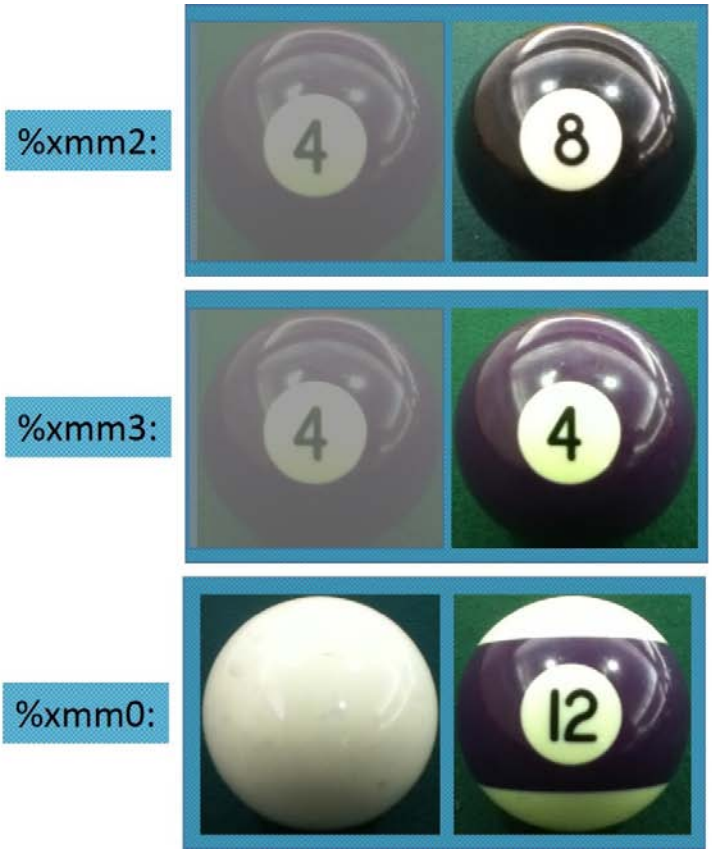
最後の命令は、2 つの部分解を加算して最終値を計算します。命令は次のようになります。

```
        vaddsd    %xmm3, %xmm2, %xmm0
```

一見すると、このベクトル加算が 2 つの加算を行うように思えるかもしれません。私は当初、1 つ目の加算で求めていた値が得られ、2 つ目の加算はベクトルレジスターのほかの値で無駄な演算を行うと考えていました (値を後で無視すればペナルティーがないため)。この加算と前の加算には微妙な違いがあります。

アセンブリー・コードの 4 行のうち、2 つの加算の「ニーモニック」に違いがあることに注意してください。最初の命令は「pd」で終わっています。'd' はオペランドが倍精度浮動小数点であることを示し、'p' はデータがパックされていること、または使用するレジスターに複数の値がパックされていることを示します。一方、リダクションの最後の命令は、命令の最後から 2 つ目の文字が「s」になっています。これはスカラー操作を示し、下位 64 ビット、または 1 つの倍精度値がソースレジスター (ここでは **%xmm2** および **%xmm3**) の加算に使用されます。無駄な演算は行われていません。

この最後の計算をビリヤードの球で示すと次のようになります。各レジスターの上位 64 ビットは、無視しています。



このようにして、インテル® コンパイラーは倍精度浮動小数点値のインテル® AVX ベクトル・リダクションを「記述して」いるのです。これは魔法や手品ではありません。ツールとデバイスについてすべてのことを知っているため、落とし穴を避けながらすべてのトリックを活用できるのです。

ほかのバージョンよりも「優れている」バージョンを判断したい場合、1 つの測定基準は実行時間です。実行時間を知るには、両方のバージョンで使用される個々のアセンブリー命令のクロックに関するドキュメントを調べる必要があるでしょう。たとえすべての命令について調べたとしても (私はしませんが)、アセンブリー・コード命令は 4 つのみです。独自のコードに変更を加えると、アプリケーションは膨大な数のリダクションを計算する必要があります。それらのすべてのリダクションのデータをセットアップするためには膨大なループの計算が必要であることから、独自のコードバージョンのほうが実行時間がかかります。

また、使いやすさも重要です。これは明らかに上記のコードのほうが優れています。上記のコードを生成するため、私は for ループをコード化して、ループがベクトル化されることを確認しました。記述したループをベクトル化できることをコンパイラーが認識しない場合、アノテーション・プラグマを追加するか OpenMP* 4.0 SIMD プラグマを使用します。(ループに 1 つの行を挿入するよりも、最後の文を書くほうが時間がかかりました。) スレッド化と同様に、コンパイラーと OpenMP* は、正しくない答えになる場合でも、プログラマーがベクトル化について知っていると仮定して、要求されたとおりにベクトル化を行います。

コンパイラーのベクトル化 (安全であることが分かっている場合のプログラマーからのヒントを含む) を使用する利点は、将来のアーキテクチャーやベクトル化命令セットでもコードが動作することが「保証される」ことです。インテル® モダンコード開発者コミュニティーには、コンパイラーがアプリケーションをより適切にベクトル化できるように開発者を支援する、さまざまなアドバイスや例が用意されています。あるベクトル化手法を実装するために独自の組込み関数を作成した場合、そのコードを次世代のベクトル・ハードウェアに移植するときに、すべてのコードを再度作成する必要があるでしょう。個人的には、バリ島のビーチから会社にいる同僚に「新しいベクトルオプションを指定して再コンパイルしておいて」と電話で指示するほうがいいですね。(それでは、私の下手なドイツ語で「*Ausgezeichnet!*」)

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。