ベクトル化されたリダクション操作を記述できますか?

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「Can You Write a Vectorized Reduction Operation?」の日本語参考訳です。

ある開発者に、構造体配列 (AoS) にあるインデックス付きの複素数のコレクション (実部と虚部を含む) を配列構造体 (SoA) に変更しなければならないことを示すため、差分の 2 乗和を計算する hotspot コードを単純化したものを作成しました。このサンプルコードは 2 つの同じサイズの複素数ベクトルを受け取り、対応する実部と虚部の差分をそれぞれ計算し、2 つの差分値の 2 乗和を求めて、結果を集計済みの合計に足す for ループです。

要点を明確にするため、開発者から提供された複素数型の**構造体**配列を使用して for ループ計算を実装し、別のバージョンとして、実部と虚部の値を格納する 2 つの配列からなる構造体を使用するループ計算を追加しました。どちらのバージョンでも、C++ コードで倍精度浮動小数点値を使用しました。予想どおり、データレイアウトが改善されたことで、配列構造体 (SoA) のほうがオリジナルのコードよりもパフォーマンスが向上しました。さらに、インテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) のベクトル化を利用するようにサンプルコードをコンパイルしてみたところ、データ構造を変更する利点がより明確になりました。

更なる可能性を追求するため、インテル® AVX のベクトル組込み関数を利用する、2 つサンプルコードの別のバージョンを作成しました。計算で行っている処理は非常に単純です: ベクトルレジスターにデータをコピーし、ベクトルレジスターの内容を複製・置換して、いくつかの乗算と加算を行います。これらの操作がサポートされていることは分かっていたので、オンラインの Intel Intrinsics Guide (インテルの組込み関数ガイド) ページ (英語) でどの組込み関数を使用すべきかを確認しました。Technology セクションでターゲットシステム上でサポートされているテクノロジーと、Categories セクションで操作を選択すると、必要なすべての組込み関数の情報を入手できます。この作業中に、最終ベクトルレジスター内の個々の和を加算するリダクション操作が必要なことに気付きました。

これは簡単に追加できると考えられました。並列計算においてリダクションは基本的な操作です。OpenMP* とインテル® スレッディング・ビルディング・ブロック (インテル® TBB) はどちらもビルトインのリダクション操作を提供しています。また、『The Art of Concurrency (日本語訳: 並列コンピューティング技法)』の第 6 章の半分は、並列和 (リダクション) の実装方法について説明しています。『The Art of Concurrency (並列コンピューティング技法)』のアドバイスに従って (そして、できるだけ簡単に済ませられるように)、オンラインでサンプルコードを探してみました。すでに誰かがどこかでリダクションのベクトル組込み関数コードを作成していると思ったのです。しかし、Google™ で 20 分費やしても何も見つかりませんでした。そこで、Intel Intrinsics Guide ページに戻り、自分でコードを記述することにしました。

結論から言うと、以下が私の考案したアルゴリズムのスタンドアロン実装です。このコードは、インテル® AVX ベクトルレジスター内の 4 つの倍精度値のリダクション操作を行い、(結果をチェックし、正しく行われたことを示すため) 最終結果を出力します。アルゴリズムは 4 つのステップで構成されます: **ymm0** ベクトル変数の値を受け取り、4 つの倍精度値の和を計算し、それぞれの結果を最終ベクトル変数 (**ymm4**) の 4 つの要素に格納します。4 つのステップと使用した組込み関数については後で詳しく述べます。

```
#include <iostream>
// get AVX intrinsics
#include <immintrin.h>

double  indata[4] = {2.0, 3.0f, 2.0f, 5.0f};
double outdata[4] = {0.0, 0.0, 0.0, 0.0};

__m256d ymm0, ymm1, ymm2, ymm3, ymm4;
```

```
int main()
{
    ymm0 = _mm256_loadu_pd(indata);
    ymm1 = _mm256_permute_pd(ymm0, 0x05);
    ymm2 = _mm256_add_pd(ymm0, ymm1);
    ymm3 = _mm256_permute2f128_pd(ymm2, ymm2, 0x01);
    ymm4 = _mm256_add_pd(ymm2, ymm3);
    _mm256_storeu_pd(outdata, ymm4);
    std::cout << "Out vector: " << outdata[0] << " " << outdata[1] << " " <<
 outdata[2] << " " << outdata[3] << std::endl;
}
```

私は視覚的に覚えるほうなので、各インテル® AVX 組込み関数の操作を理解しやすいように、パイガオの牌を使ってインテル® AVX ベクトルレジスター内の 4 つの値を表します。ここでは、各牌の名称、点の色、点の配置は無視し、点の数がベクトルの各要素の値を表すものとします。(サンプルコードでは倍精度浮動小数点値を使用していますが、牌の点は整数値を表します。)

リダクションを行うデータは、**ymm0** 変数にロードします。(プロセッサーのベクトルレジスターの名前にちなんで変数名を付けていますが、これらの変数名はコンパイル済みコード内で使用される実際のレジスターにマップされません。また、ここではこれらの変数を「ベクトル変数」と呼ぶ代わりに、「レジスター」と呼んでいます。) 最初に、プログラムは **indata** の値を **ymm0** にロードします。このアルゴリズムを差分の 2 乗和を計算するコードに統合する場合、ベクトル化された計算の結果が初期データになります。どのようなケースでも、リダクション対象のデータは、まず **ymm0** レジスターにロードされます。初期データの 4 つの値を図で表してみましょう。

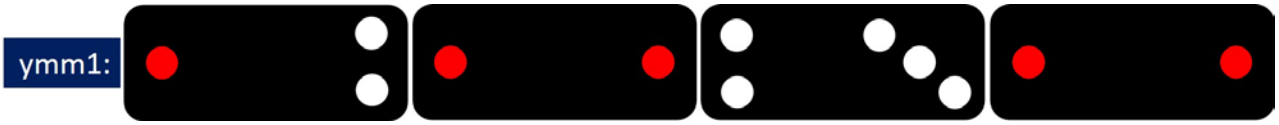
リダクションの最初のステップは、**ymm0** レジスター内のデータをコピーし、ベクトル要素の順番を置換して **ymm1** レジスターに格納します。

```
    ymm1 = _mm256_permute_pd(ymm0, 0x05);
```

第 1 引数はデータのソース (**ymm0**)、第 2 引数はソースの要素の置換を制御する整数ビットマスクです。マスクの最下位ビット (index [0]) はベクトルの最下位要素 (ビット 63:0)、次のビット (index [1]) はベクトルの次の 64 ビット要素 (ビット 127:64) を指定し、残りの上位 2 つのベクトル要素も同様になります。マスクの最下位ビットが '0' の場合は、最下位要素の現在の値をそのままデスティネーションにコピーします。'1' の場合は、次の要素 (ビット 127:64) をデスティネーションの最下位要素にコピーします。マスクの下から 2 番目のビットでは逆で、'1' の場合は値をデスティネーションの下から 2 つ目の要素にコピーし、'0' の場合は最下位要素の値をデスティネーションにコピーします。この処理は 128 ビット・レーンで行われ、ベクトルレジスターの上位 2 つの要素も同様に置換できます (すべてのビット・インデックスに 128 を足すだけです)。

(ここまでの複雑な説明で理解できた方は、私よりも読解力が優れている方でしょう。私を含め、そうでない方のために **ymm1** に格納された値を図に表してみます。長く複雑な説明よりも図にしたほうが分かりやすいでしょう。ここで使用している組込み関数、その引数と操作については、Intel Intrinsics Guide を確認してください。)

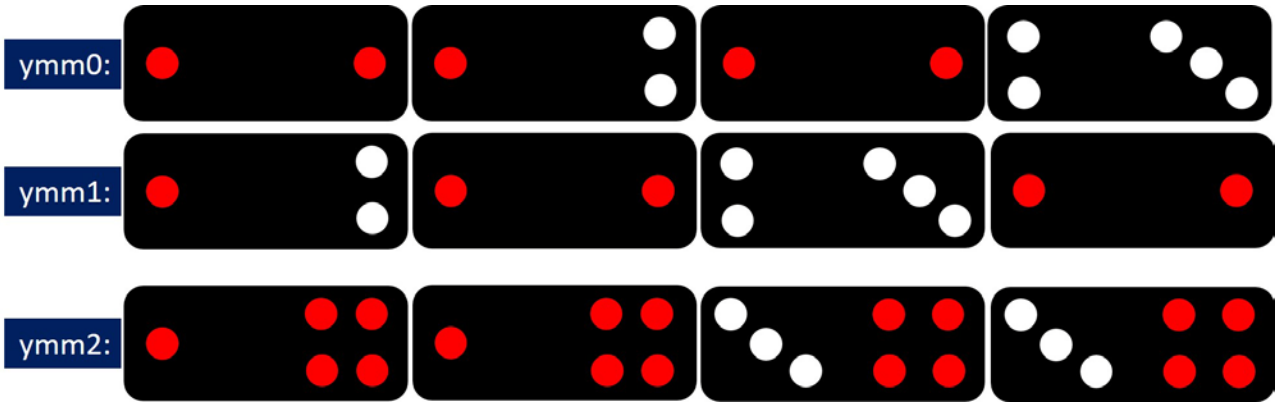
私のリダクション操作では、ビットマスクに '0101' (16 進数では 0x05) を使用して、ベクトル要素の下位のペアの値の位置と、ベクトル要素の上位のペアの値の位置をそれぞれ交換します。下の図と上の図を比較して、**ymm1** レジスター内の要素がどのように置換されたか確認してみてください。



次の命令は、オリジナルデータと置換したコピーを加算します。

```
ymm2 = _mm256_add_pd(ymm0, ymm1);
```

これは基本的な命令で、2 つのソースレジスター (**ymm0**、**ymm1**) の対応する要素を加算し、結果をデスティネーション・レジスターの対応する要素に格納します。次の図は、2 つのソースレジスターと **ymm2** デスティネーション・レジスターの内容を示したものです。



2 つ目の操作が完了すると、リダクション操作の半分が終わったことになります。上の図から、オリジナルレジスターの上位 2 つの要素と下位 2 つの要素の合計が計算され、それぞれ上位 2 つの要素と下位 2 つの要素に複製されたことが分かります。

リダクション操作を完了するには、置換と加算の 2 つの操作を実行するだけです。置換操作では、ベクトルデータの 128 ビットをコピーする必要があります。これは次の組込み関数で行えます。

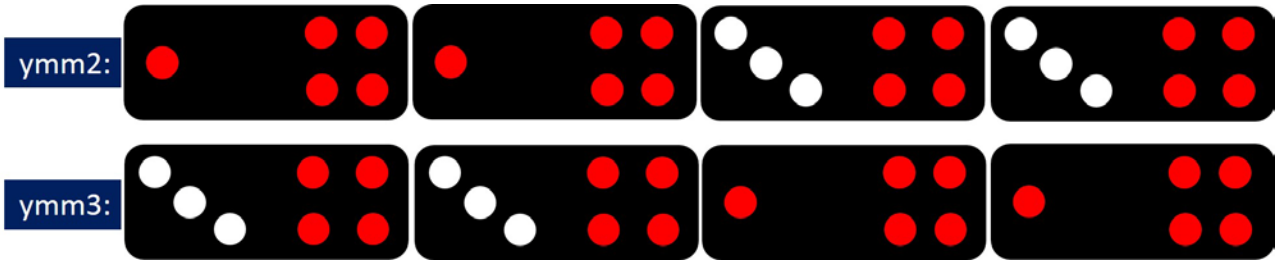
```
ymm3 = _mm256_permute2f128_pd(ymm2, ymm2, 0x01);
```

Intel Intrinsics Guide にあるこの命令の説明では、第 1 引数と第 2 引数が異なるベクトル・ソース・レジスターになっています。ここでは、1 つのソースのデータを置換 (またはコピー) したいだけなので、同じレジスターを使用しています。以前使用した置換命令と同様に、最後の引数はデスティネーション・レジスターにコピーする値を制御するビットマスクです。

Intel Intrinsics Guide の説明では、ビットマスクは 8 ビットで、下位 4 ビットが 256 ビットのデスティネーション・レジスターの下位 128 ビットにコピーされる値を制御し、上位 4 ビットが同レジスターの上位 128 ビットにコピーされる値を制御します。入力ソースが 2 つある場合、128 ビット・レジスターにコピーされる可能性がある値は 4 つです: 第 1 ソースの下位半分 (ビット 127:0) (‘0’)、第 1 ソースの上位半分 (ビット 255:128) (‘1’)、第 2 ソースの下位半分 (‘2’)、および第 2 ソースの上位半分 (‘3’)。

Intel Intrinsics Guide の説明では、ビットマスクは 4 ビットであるのに対し、そのうち 2 ビットのみが 4 つのソースの中からコピーするものを決定するのに使用されるため、混乱を生じる可能性があります。ビットマスクを指定する際に、上位 2 ビットは考慮されないため、2 進数値 ‘0001’ は ‘1101’ と同様に扱われるかのような印象を受けます。Intel Intrinsics Guide の説明にあるコード例では、暗黙的にこのことが示されています。(しかし、実際には違います。興味のある方は、上記のプログラムと 2 つ目の置換組込み関数を使用して試してみてください。) ここでは、同じソースレジスター **ymm2** の上位と下位の 128 ビットを交換してデスティネーション・レジスターに格納するだけなので、そのことを心配する必要はありません。

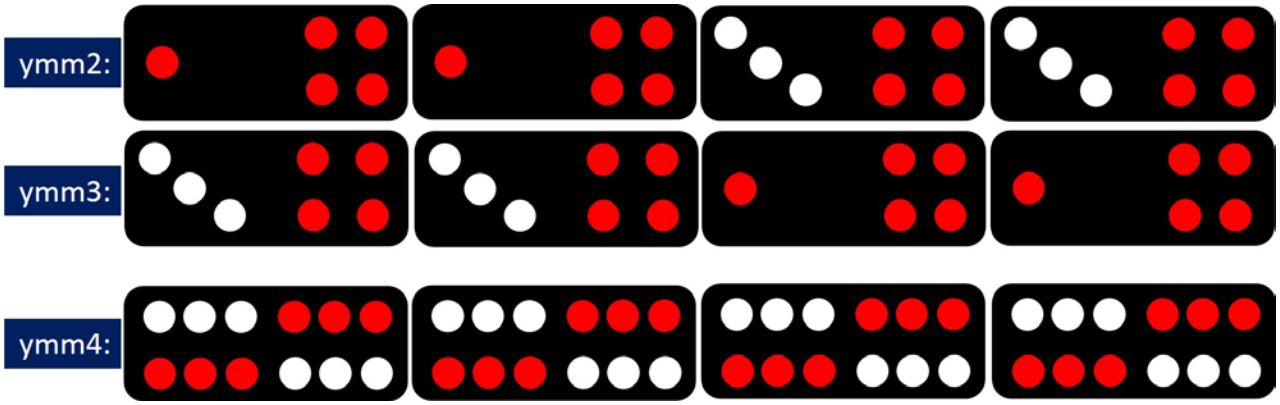
次の例は、ソースレジスター **ymm2** と、置換組込み関数を実行後のデスティネーション・レジスター **ymm3** の内容を示します。(上記の命令のビットマスク '0x21' が同じ結果を生成することが分かりますか?)



最後に、**ymm2** と **ymm3** レジスターのベクトル加算を行って、オリジナル・ベクトル・レジスターからの値のリダクション操作は完了です。

```
ymm4 = _mm256_add_pd(ymm2, ymm3);
```

パイガオの牌で表すと次のようになります。



インテル® AVX の 256 ビット・ベクトル・レジスターの 4 つの倍精度要素に対するリダクション (合計) 操作は、これですべてです。この記事で最初に示した完全なコード例の最後にあるいくつかの命令は、最終ベクトルレジスターから 4 つの結果をコピーし、確認のためそれらを出力します。

このアルゴリズムは、異なるデータ型やほかのベクトル手法でも実装できるのでしょうか?一般的な問題については詳しく見ていませんが、適切な置換組込み関数を利用できれば、異なるサイズやデータ型でも同様のリダクション・アルゴリズムを実装できるでしょう。

皆さんは、組込み関数の調査、操作の詳細の解読、適切な引数とビットマスクの組み合わせを見つけるまでにかかった労力に見合った成果が得られたか気になるでしょう。残念ながら、大きな成果は得られませんでした。差分の 2 乗和を計算するハンドコードのベクトル化バージョン (4 ステップのリダクション操作を含む) とコンパイラーにより生成された単純な for ループコードのベクトル化バージョンを比較したところ、パフォーマンスの差はごくわずかでした。恐らく、コンパイラーを使用している方はすでに同様の経験をされていることでしょう。この経験を通して、よく言われる「ソースコードのベクトル化はコンパイラーに任せ、プログラマーは必要に応じてヒントを与えたり、わずかな変更を加えるだけのほうが良い」ということが正しいことを学びました。

この経験は、決して無駄ではなかったと思います。記事にすることで、Google™ で “vectorized reduction” を検索したら、このアルゴリズムが最初に (そして唯一) 表示されるかもしれません。

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。