

ストリーミング SIMD 拡張命令 2(SSE2)を使用した ブロック・マッチング、動体予測アルゴリズム

バージョン 2.0

2000 年 7 月

資料番号: 248605J-001

【輸出規制に関する告知と注意事項】

本資料に掲載されている製品のうち、外国為替および外国為替管理法に定める戦略物資等または役務に該当するものについては、輸出または再輸出する場合、同法に基づく日本政府の輸出許可が必要です。また、米国産品である当社製品は日本からの輸出または再輸出に際し、原則として米国政府の事前許可が必要です。

【資料内容に関する注意事項】

・本ドキュメントの内容を予告なしに変更することがあります。

・インテルでは、この資料に掲載された内容について、市販製品に使用した場合の保証あるいは特別な目的に合うことの保証等は、いかなる場合についてもいたしかねます。また、このドキュメント内の誤りについても責任を負いかねる場合があります。

・インテルでは、インテル製品の内部回路以外の使用にて責任を負いません。また、外部回路の特許についても関知いたしません。

・本書の情報はインテル製品を使用できるようにする目的でのみ記載されています。

インテルは、製品について「取引条件」で提示されている場合を除き、インテル製品の販売や使用に関して、いかなる特許または著作権の侵害をも含み、あらゆる責任を負わないものとします。

・いかなる形および方法によっても、インテルの文書による許可なく、この資料の一部またはすべてを複製することは禁じられています。

本資料の内容についてのお問い合わせは、下記までご連絡下さい。

インテル株式会社 資料センタ

〒305-8603 筑波学園郵便局 私書箱115号

Fax: 0120-47-8832

*一般にブランド名または商品名は各社の商標または登録商標です。

Copyright © Intel Corporation 1999, 2000

目次

1	はじめに	5
2	ブロック・マッチング	5
2.1	ブロック・マッチングを使用するアプリケーション	5
2.2	ブロック・マッチング・アルゴリズムの実現	6
2.2.1	高速化の手法	7
3	パフォーマンス	8
3.1	測定されたパフォーマンス向上	8
3.2	考察	9
4	結論	9
5	C/C++によるコード例	10
6	SSE2 固有機能を使用したコード例	11
付録 A	パフォーマンス・データ	A-1
	パフォーマンス・データの改訂履歴	A-1
	テスト・システム構成	A-2

改訂履歴

改訂	改訂履歴	日付
2.0	インテル® Pentium® 4 プロセッサに関する改訂	2000年7月
1.0	初版	1999年9月

参考資料

このアプリケーション・ノートでは次の資料を参考にした。次の資料には、ここで取り上げた事項を理解するための有用な情報が含まれている。

1. 「Digital Television」、Benoit, H. 著、Arnold Publishing Co.、1997 年
2. 「MPEG Video Compression Standard」、Mitchell, Joan L. 他著、Chapman and Hall Publishing、1995 年

1 はじめに

ストリーミング SIMD 拡張命令 2(SSE2、Streaming SIMD Extensions 2)では、SIMD(Single Instruction Multiple Data)倍精度浮動小数点命令、および SIMD 整数命令が IA-32 Intel[®] アーキテクチャに新しく導入された。倍精度 SIMD 命令による機能拡張の方法は、ストリーミング SIMD 拡張命令(SSE)で導入された単精度 SIMD 命令による機能拡張とよく似ている。128 ビットの整数 SIMD 拡張命令は、64 ビットの整数 SIMD 拡張命令の完全なスーパーセットで、より多くの整数データ型、整数と浮動小数点間のデータ型変換、キャッシュとシステム・メモリの効果的使用をサポートする命令が追加されている。これらの命令は、3D グラフィックス、リアルタイムの物理的な現象、空間的(3D)オーディオ、ビデオ・エンコーディング/デコーディング、暗号化、および科学計算アプリケーションによく使用される演算を高速化する。このアプリケーション・ノートでは、特に 128 ビット整数 SIMD 拡張命令を使用するアプリケーションについて説明し、SSE2 を利用するコード例を示す。

このアプリケーション・ノートでは、128 ビット・バージョンの `psadbw` および `paddw` 命令の使用例として、高速ブロック・マッチング・アルゴリズムについて詳しく説明する。MPEG および MPEG2 エンコーダでは、実行時間の 40 ~ 70% をこの種のコードが占めている。

2 ブロック・マッチング

ブロック・マッチング・アルゴリズムとは、16 バイト × 16 バイトのメモリ・ブロック同士を比較するアルゴリズムで、対応するピクセルの絶対差の総和(Sum of the Absolute Differences : SAD)、または差の二乗和(Squared Sum of Differences : SSD)から一致度を計算する。最終的に得られた差の総和が、ビデオの 2 つのブロックの一致度を表す尺度になる。このアルゴリズムでは、各ピクセル・ペアの絶対差を計算するときに、次のような分岐が必要になるのに注意する。

```
if(difference < 0) difference *= -1;
running_total += difference;
```

このアプリケーション・ノートでは、よく使用されている SAD アルゴリズムについてのみ説明する。

2.1 ブロック・マッチングを使用するアプリケーション

MPEG エンコーダの圧縮方法には、フレームを個々に圧縮する方法と、複数フレームからなるシーケンスを圧縮する方法の 2 種類がある。JPEG 圧縮と同様に、ビデオ・シーケンスを構成する個々のフレームを個別に圧縮できる。このように圧縮したデータは、一般に元の 1/3 ~ 1/4 の大きさで保存できる。

一般に、ビデオ内の連続フレームは内容がよく似ていることに注目すると、圧縮率を劇的に高められる。各フレームの違いは、いくつかのピクセル・ブロックがフレーム内を規則正しく動いているだけのことが多い。テニスの試合のビデオを思い浮かべてほしい。大半のピクセルはフレームごとに前後に移動するが、内容は同じである。ボールもやはり同じボールであり、フレームごとに位置が異なるだけである。

ブロックを圧縮して保存するかわりに、モーション・ベクトルで保存できる。例えば、 16×16 のブロックが2ピクセル左、1ピクセル下に移動した場合、そのような移動情報を保存する方が、256ピクセル全体を圧縮して保存するより効率がよい。

2.2 ブロック・マッチング・アルゴリズムの実現

一般に、ブロック・マッチング・アルゴリズムは動体予測(Motion Estimation : ME)検索アルゴリズムに組み込まれている。ME 検索には種々の方法があるが、最も単純なのは完全検索である。これは、特定範囲のすべてのブロックをピクセル単位でくまなく比較する。例えば、 16×16 のブロック単位で完全検索すると、カレント・ブロックが256個のリファレンス・ブロックと比較される。リファレンス・ブロックは開始点からピクセル単位でカレント・ブロックと比較される。次に1ピクセルだけリファレンス・ブロックを移動して、同じカレント・ブロックと比較する。検索領域全体に対して、この作業を繰り返す。SAD 値が最も小さいブロックがベスト・マッチと見なされる。

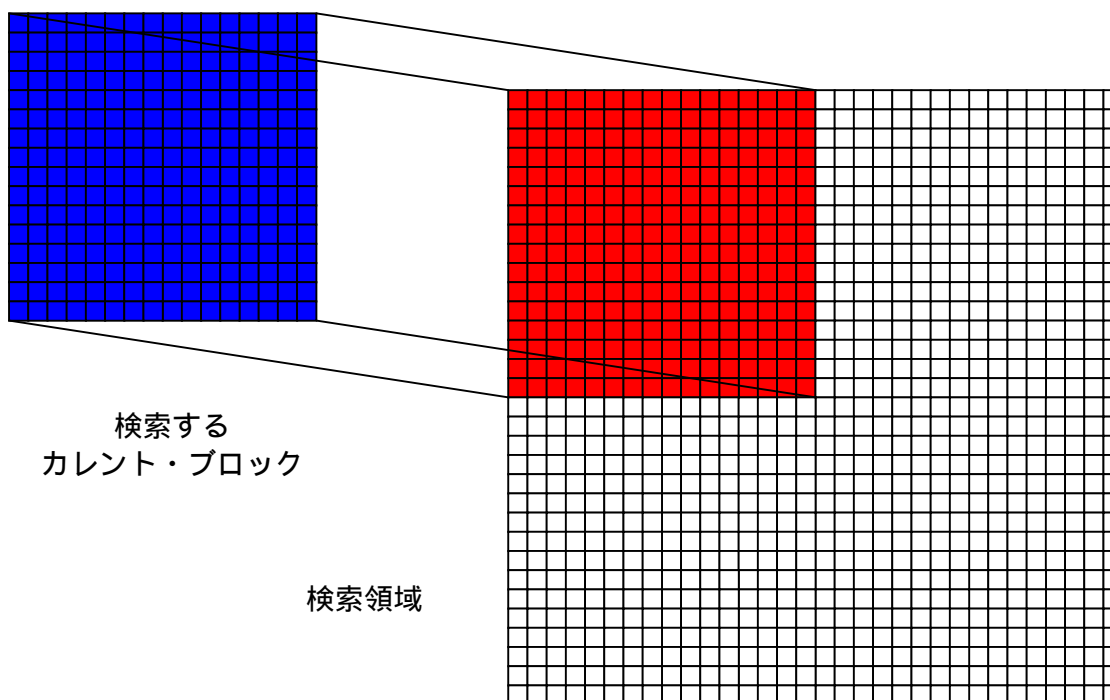


図1: 最初のブロックとの比較

図1に、最初に比較されるブロック・ペアを示す。動体予測検索は、検索範囲の左上隅から開始する。2つのブロック内で対応するピクセル値を減算し、その絶対値を累積する。絶対差の総和が、2つのブロックの一致度を表す相対的な尺度となる。

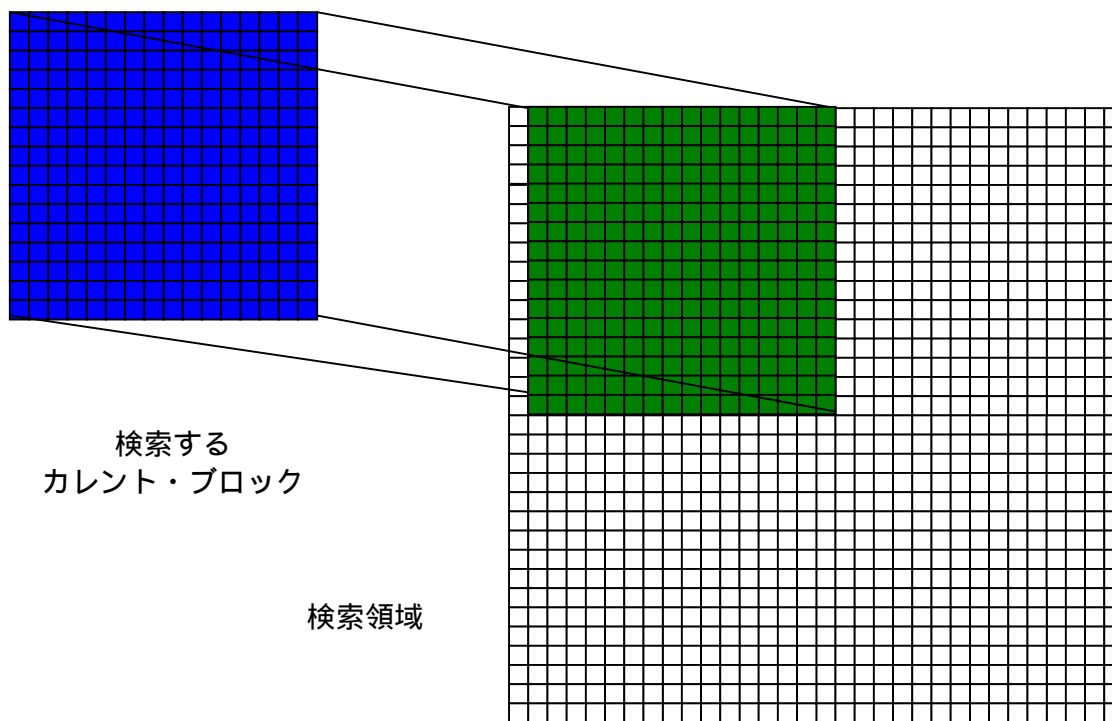


図2: 2 番目のブロックとの比較

図2に、2番目に比較されるブロック・ペアを示す。比較するピクセル・ペアは異なるものの、大部分が最初のブロック・ペアと重なっているのに注意すること。このような比較作業が、検索領域全体に渡って繰り返される。256個の値を持つ2つの配列を256回比較するため、すぐに数万回の比較サイクルに到達してしまうのは明らかである。もちろん、比較サイクルはフレーム内の全ブロックで繰り返す必要がある(MPEG2の場合は一般に720×480ピクセルなので、45×30=1350ブロック/フレーム)。これがエンコーダの実行時間の大部分が動体予測に費やされている理由である。

2.2.1 高速化の手法

ME アルゴリズムを高速化する第1ステップとして、必要なブロック比較回数を減らす。サブサンプリング、対数検索、絞込み検索のような手法で、エンコーダの効率をあげられる。ビデオ品質は落とすことなく、比較回数削減により3~8倍の効率アップが可能である。これらの手法の実例については、「参考資料」に示した資料を参照のこと。

ME アルゴリズムを高速化する第2ステップは、SSE (特に `psadbw` 命令) を使用できる Pentium® III プロセッサの導入である。`psadbw` 命令は、両ブロックから8バイトずつまとめて比較して、1つのSAD値を返す。8つの減算と累積計算を1つの命令で行うだけでなく、分岐なしで絶対値を決定できる。このため、Pentium III プロセッサでMMX®テクノロジーを使用するときの1.7倍の高速化を実現できる。

SSE2における機能拡張により、この命令は同時に16ピクセルを処理できるようになった。そのため、ブロックの各行全体のSAD計算を1つの命令で実行できる。

アルゴリズムによっては、「早期リターン」しきい値を使用するものがある。決まった行数だけ比較し、その時点でのSAD値がしきい値を超えていれば、そこで中止する。すでに比較し

たブロックよりはるかに差違の大きいブロックであれば、それ以上比較を続ける意味はないので、このやり方は賢明である。8バイト幅または16バイト幅の `psadbw` 命令を使用する場合は、多くのアルゴリズムにおいて、この手法の有効性が低くなる。中止するかどうかを判定する(分岐の予測誤りのペナルティが発生する)方が、ブロック・マッチングを最後まで行うより時間がかかる上に、正確さが低下するからである。

3 パフォーマンス

3.1 測定されたパフォーマンス向上

128ビット・バージョンの `psadbw` 命令を使用すると、SSEバージョンのコード(ここでは64ビット・バージョンの `psadbw` 命令を使用)と比較して、大きなパフォーマンス向上が見られた。また、各行で一度しかSAD計算が必要ないため、関数サイズも半分近くに減らせた。

`Psadbw` は下位8バイトと上位8バイトの2つのSAD値を返す。この2つを加えて最終的なSAD値が得られる。この計算は行ごとに行う必要はなく、ブロック・マッチング全体で一度でよいことに注意する。

Pentium III プロセッサ(733 MHz)に対する、Pentium 4 プロセッサの全般的パフォーマンス向上も非常に大きかった。これには、次の3つの要因がある。

- メモリ帯域幅の拡大(Pentium III プロセッサの800MB/秒に対して3.2GB/秒)により、外部メモリからのデータのロードが高速になった。
- 符号なし演算(uops)のスケジュールの改良
- カウント付きループ終了時のトラッキング用ビット数が増えたことによる、外側検索ループの分岐予測の改良

上記のような新しいアーキテクチャの利点は、単に新しく導入された128ビット命令を使用する利点より大きい。

このアルゴリズムは膨大なメモリを使用し、常にキャッシュを使い尽くす(各フレームが720×480、すなわち300K以上ある)ため、バスの帯域幅やCPUメモリなどサブシステム・リソースを大量に必要とする。プリフェッチはより優先度が高いロードを追い出す傾向にあり、プリフェッチ命令を使用する利点はなかった。計算に費やされる時間はロード回数に比べて少ないので、このアプリケーションではプリフェッチは不適切だった。検索中は、検索領域内を1ピクセルずつ移動するため、データ・キャッシュが分断された。これは、ロードしようとするデータが別の2つのキャッシュ・ラインに分かれて入っているために起こる。以前のインテル・プロセッサでは、2つのキャッシュ・ラインをロードし、手動でデータを構築する方が、高いペナルティに耐えるより速い場合があった。Pentium 4 プロセッサで `movdqu` 命令を使用すると、同じことをアルゴリズムで実現するより高速である。

3.2 考察

理論的には、2 倍の幅を持つ `psadbw` 命令と適切な命令パイプラインの使用により、2 倍の高速化が実現できるはずである。ところが、16 バイト境界にアライメントされていないデータをロードしなければならない場合は、そうはいかない。すべての暗黙的ロードと `movdqa` 命令は、16 バイト境界にアライメントされていなければ、正しく機能しない。動体予測検索を説明した図からわかるように、検索ブロックは検索領域内を 1 ピクセルずつ移動するため明らかに、ロードするすべてのデータを 16 バイト境界にアライメントするのは不可能である。

この動作(配列内を 1 ピクセルだけ移動する動作)が、データ・キャッシュ・ラインの分断の原因となっている。一度のロードに必要なデータが、2 つのキャッシュ・ラインに分かれてしまうのである。この場合、ロードを 2 度行い、2 つのデータを組み合わせて目的のデータにする余分な操作が必要になるので、非常に遅くなる。わかりやすくするために、アドレスがキャッシュ・ラインの最終バイトから始まる 1 つの整数(32 ビット/4 バイト)をロードする場合を考えてほしい。1 バイトは最初のキャッシュ・ラインから取り出し、残りの 3 バイトは 2 番目のキャッシュ・ラインから取り出すことになる。この 2 つの部分のプロセッサ内部で組み合わせてからでないと、計算できない。

命令セットには、アライメントされていないデータをロードする `movdqu` 命令がある。`movdqu` 命令は、データ・キャッシュ分断も含めて、あらゆるタイプのミスアライメントに対応するが、その分時間がかかる。実際には、一連の符号なし命令(`uops`)でアライメント済みデータを生成している。この命令を使用すると、高速化の妨げとなる。

重要なスケジューリング問題の 1 つに、`psadbw` 命令と `paddw` 命令の処理がある。この 2 つの命令は実行に何サイクルかを要し、コード内で連続してスケジューリングされている 2 つの `psadbw` 命令(または 2 つの `paddw` 命令)はオーバーラップしない。ところが、1 つの `psadbw` 命令と 1 つの `paddw` 命令を続けて実行するときは、連続したクロックで実行を開始し、オーバーラップして実行できる。その結果、コードの計算部分のパフォーマンスが向上する。

4 結論

SSE2 は、MPEG エンコーダで最もプロセッサへの負荷が大きい部分のパフォーマンスを大きく向上させると同時に、コードの大きさや複雑さを軽減することが明らかになった。動体予測検索における制限事項により、理論的に達成できるはずの最高パフォーマンスは得られなかった。

結論として、Pentium 4 プロセッサ・アーキテクチャは、動体予測アルゴリズムの高速化に大きく貢献する。通常の MPEG エンコーダで CPU の 50% を動体予測検索に使用していたら、Pentium 4 プロセッサではその比率をずっと小さくできる。あるいは、実行時間はそのままより正確な検索を行い、エンコードしたビデオの品質を向上させることができる。

5 C/C++によるコード例

```
// These outer loops control the movement through the search range
// pucC points to the current (stationary) block, pucR
// points to the block in the search area.
// iTop, iBottom, iLeft, iRight are the edges of the search range.
// iWidth is the number of pixels across a frame.
for(iY = iTop; iY <= iBottom; iY++){
    pucRefLeft = pucRef + (iY * iWidth);    // Setup pointer into search
                                            // range for the current Y.
    for(iX = iLeft; iX <= iRight; iX++){
        iTmpAd = 0;                        // Initialize accumulator
        pucC = pucCurLeftTop;            // Set pointers for current X
        pucR = pucRefLeft + iX;

        // These inner loops control the movement across the block
        // as we do the SAD operation pixel by pixel.
        for(iV = 0; iV < 16; iV++){        // For each pixel in 16X16 block,
            for(iH = 0; iH < 16; iH++){    // calculate the abs difference
                iTmpAd += abs((ME_INT32)*(pucC++) - (ME_INT32)*(pucR++));
            }
            pucC += iDown; // Move pointers to next row of block
            pucR += iDown;
        }

        /* Check for a new minimum SAD. If it is a new min, store
           position information */
        if(iMinAd > iTmpAd){
            iMinAd = iTmpAd;
            *piMvPos = iX;
            *(piMvPos+1) = iY;
        }
    }
}
return iMinAd; // Return best SAD value from search
```

6 SSE2 固有機能を使用したコード例

```

pucRef = (pucRef + (iTop * iWidth) + iLeft);
pucC = pucCur + (iVpos * iWidth) + iHpos;

// Block loop
for(iY = 0; iY <= (iBottom - iTop); iY++){

    // Set start point for Reference window
    pucR = pucRef + iY * iWidth;

    for(iX = 0; iX <= (iRight - iLeft); iX++){
        sum = _mm_xor_si128(sum, sum);           // Clear accumulator
        sum2 = _mm_xor_si128(sum2, sum2);       // Clear accumulator

        // Get SAD for block pair
        row2 = _mm_loadu_si128((__m128i *)pucR);
        row4 = _mm_loadu_si128((__m128i *) (pucR + iWidth));
        row6 = _mm_loadu_si128((__m128i *) (pucR + 2*iWidth));
        row8 = _mm_loadu_si128((__m128i *) (pucR + 3*iWidth));
        row1 = _mm_load_si128((__m128i *) pucC);
        row3 = _mm_load_si128((__m128i *) (pucC + iWidth));
        row5 = _mm_load_si128((__m128i *) (pucC + 2*iWidth));
        row7 = _mm_load_si128((__m128i *) (pucC + 3*iWidth));

        row1 = _mm_sad_epu8(row1, row2);
        row3 = _mm_sad_epu8(row3, row4);
        sum = _mm_add_epil6(sum, row1);
        sum2 = _mm_add_epil6(sum2, row3);

        row5 = _mm_sad_epu8(row5, row6);
        row8 = _mm_sad_epu8(row7, row8);
        sum = _mm_add_epil6(sum, row5);
        sum2 = _mm_add_epil6(sum2, row7);

        row2 = _mm_loadu_si128((__m128i *) (pucR + 4*iWidth));
        row4 = _mm_loadu_si128((__m128i *) (pucR + 5*iWidth));
        row6 = _mm_loadu_si128((__m128i *) (pucR + 6*iWidth));
        row8 = _mm_loadu_si128((__m128i *) (pucR + 7*iWidth));
        row1 = _mm_load_si128((__m128i *) (pucC + 4*iWidth));
        row3 = _mm_load_si128((__m128i *) (pucC + 5*iWidth));
        row5 = _mm_load_si128((__m128i *) (pucC + 6*iWidth));
        row7 = _mm_load_si128((__m128i *) (pucC + 7*iWidth));

```

```
row1 = _mm_sad_epu8(row1, row2);
row3 = _mm_sad_epu8(row3, row4);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);

row5 = _mm_sad_epu8(row5, row6);
row7 = _mm_sad_epu8(row7, row8);
sum = _mm_add_epi16(sum, row5);
sum2 = _mm_add_epi16(sum2, row7);

row2 = _mm_loadu_si128((__m128i *) (pucR + 8*iWidth));
row4 = _mm_loadu_si128((__m128i *) (pucR + 9*iWidth));
row6 = _mm_loadu_si128((__m128i *) (pucR + 10*iWidth));
row8 = _mm_loadu_si128((__m128i *) (pucR + 11*iWidth));
row1 = _mm_load_si128((__m128i *) (pucC + 8*iWidth));
row3 = _mm_load_si128((__m128i *) (pucC + 9*iWidth));
row5 = _mm_load_si128((__m128i *) (pucC + 10*iWidth));
row7 = _mm_load_si128((__m128i *) (pucC + 11*iWidth));

row1 = _mm_sad_epu8(row1, row2);
row3 = _mm_sad_epu8(row3, row4);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);

row5 = _mm_sad_epu8(row5, row6);
row7 = _mm_sad_epu8(row7, row8);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);

row2 = _mm_loadu_si128((__m128i *) (pucR + 12*iWidth));
row4 = _mm_loadu_si128((__m128i *) (pucR + 13*iWidth));
row6 = _mm_loadu_si128((__m128i *) (pucR + 14*iWidth));
row8 = _mm_loadu_si128((__m128i *) (pucR + 15*iWidth));
row1 = _mm_load_si128((__m128i *) (pucC + 12*iWidth));
row3 = _mm_load_si128((__m128i *) (pucC + 13*iWidth));
row5 = _mm_load_si128((__m128i *) (pucC + 14*iWidth));
row7 = _mm_load_si128((__m128i *) (pucC + 15*iWidth));

row1 = _mm_sad_epu8(row1, row2);
row3 = _mm_sad_epu8(row3, row4);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);
```

```
row5 = _mm_sad_epu8(row5, row6);
row7 = _mm_sad_epu8(row7, row8);
sum = _mm_add_epi16(sum, row1);
sum2 = _mm_add_epi16(sum2, row3);
sum = _mm_add_epi16(sum, sum2);

tmp = sum;
sum = _mm_srli_si128(sum, 8);
sum = _mm_add_epi32(sum, tmp);

// Check for new minimum AD
iTmpAd = _mm_cvtsi128_si32(sum);
if(iTmpAd < iMinAd){
    iMinAd = iTmpAd;
    *piMvPos = iX + iLeft;
    *(piMvPos+1) = iY + iTop;
}
pucR++;
}
}
return iMinAd;
```

付録 A - パフォーマンス・データ

パフォーマンス・データの改訂履歴

改訂	改訂履歴	日付
2.0	1.2 GHz インテル® Pentium® 4 のパフォーマンス・データに関する改訂	2000年7月
1.0	初版	1999年9月

表 1: ME プログラムのパフォーマンス・データ

パフォーマンス・データ(単位はミリ秒)		
	インテル® Pentium III プロセッサ (733 MHz)	インテル® Pentium 4 プロセッサ (1.2 GHz)
ストリーミング SIMD 拡張命令 (SSE)	25.6	13.6
ストリーミング SIMD 拡張命令 2 (SSE2)	-	10.8

表 2: 表 1 のパフォーマンス・データから求めた高速化

コーディング方法とプラットフォーム	高速化
Pentium 4 プロセッサ(SSE2 vs. SSE)	1.26
SSE(Pentium 4 プロセッサ vs. Pentium III プロセッサ)	1.89
Pentium 4 プロセッサで SSE2 を使用 vs. Pentium III プロセッサで SSE を使用	2.38

パフォーマンスは、733 MHz Pentium III プロセッサと 1.2 GHz Pentium 4 プロセッサで測定した。測定に使用したシステムの詳細については、A-2 ページの「テスト・システム構成」を参照のこと。

表 2 からわかるように、SSE2 を使用したコードは SSE を使用したコードより 1.26 倍高速になった。どちらのコードも、Pentium 4 プロセッサで実行した。この高速化は下記の最適化の結果である。

- SIMD 幅の拡張。整数 SSE2 は、MMX®テクノロジーの 64 ビット・レジスタのかわりに、128 ビット XMM レジスタを使用している。拡張された SIMD 幅のおかげで、マクロブロックの 1 行全体を 1 つの命令で比較できるようになった。

- 幅を拡張した `psadbw` 命令とロード命令により、コード・サイズを半分近くに減らすことができ、命令数が削減された。
- プロセッサのメモリ帯域幅が大きくなり、同じ時間でより多くのデータをロードできるようになった。このアルゴリズムは頻繁にキャッシュを使い尽くすので、高速化に大きく貢献する。
- アライメントされていないデータのロードを各行で1度行うため、高速化が制限された。これは、検索範囲をフレーム内で1ピクセルずつ移動するため、リファレンス・フレームのデータをアライメントできないために避けられない。この制限事項により、論理的な高速化が1.33倍に制限された。

テスト・システム構成

表3: インテル® Pentium® III プロセッサのシステム構成

プロセッサ	Pentium III プロセッサ(733 MHz)
システム	Intel® Desktop Board VC820
BIOS バージョン	VC82010A.86A.0028.P10
2次キャッシュ	256 KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ドライバ	製品候補 6.00.012
ハード・ディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
ビデオ・ドライバの リビジョン	NVidia Reference Driver 5.22
オペレーティング・ システム	Windows† 2000 ビルド 2195

表 4: インテル® Pentium® 4 プロセッサのシステム構成

プロセッサ	Pentium 4 プロセッサ(1.2 GHz)
システム	Intel Desktop Board D850GB
BIOS バージョン	GB85010A.86A.0014.D.0007201756
2次キャッシュ	256 KB
メモリ・サイズ	128 MB RDRAM PC800-45
Ultra ATA ストレージ・ ドライバ	製品候補 6.00.012
ハード・ディスク	IBM DJNA-371800 ATA-66
ビデオ・コントローラ/ バス	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR -32MB
ビデオ・ドライバの リビジョン	NVidia Reference Driver 5.22
オペレーティング・ システム	Windows 2000 ビルド 2195