

ストリーミング SIMD 拡張命令 2(SSE2)を使用した ビデオ・デコード/エンコードのためのモーション補正

バージョン 2.0

2000 年 7 月

資料番号: 248607J-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写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 資料センター

〒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	モーション補正アルゴリズム	5
2.1	モーション補正を使用するアプリケーション	6
2.2	モーション補正アルゴリズムの実現	8
2.2.1	命令選択と精度に関する手法	9
3	パフォーマンス	12
3.1	パフォーマンス向上	12
3.2	その他の考察	12
3.2.1	ソフトウェア・プリフェッチ命令の使用	12
3.2.2	データ・アクセスとアライメント	13
4	結論	13
5	C/C++によるコード例	14
6	SSE2 を使用したアセンブリ・コード例	18
付録 A	パフォーマンス・データ	24
	パフォーマンス・データの改訂履歴	24
	テスト・システム構成	26

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。

- 『Information Technology - Generic Coding of Moving Pictures and Associated Audio Information』、International Standard ISO/IEC DIS 13818-2、Part 2: Video、1994年
- 『MPEG Video Compression Standard』、Joan L. Mitchell、William B. Pennebaker、Chad E. Fogg、Didier J. LeGall 著、C&H、ITP、1996年、28～29、237～262ページ
- 『Image and Video Compression Standards, Algorithms and Architectures』、Vasudev Bhaskaran、Konstantinos Konstantinides 著、Kluwer Academic Publishers、1995年、91～92、171～174ページ
- 『Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback』、インテル・アプリケーション・ノート、AP-529、<http://developer.intel.com/drg/mmx/appnotes/ap529.htm>

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 インテル® 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、MPEG(Moving Picture Expert Group)ビデオ・デコード/エンコードで使用するモーション補正(MC)アルゴリズムについて説明し、SSE2 を使用してこのアルゴリズムを実現するコード例を示す。

以前のアプリケーション・ノート(AP-529)「Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback」では、モーション補正アルゴリズムでインテル® MMX® テクノロジを使用する方法を示し、同じ機能を実行するスカラ・コードに対するパフォーマンス向上について述べた。このアプリケーション・ノートでは、インテル・アーキテクチャ、SSE2、Pentium 4 プロセッサ・アーキテクチャにより、MMX テクノロジをしのぐ大幅なパフォーマンス向上が得られることを説明する。

2 モーション補正アルゴリズム

このセクションでは、モーション補正(MC)アルゴリズムについて簡単に説明する。MPEG 圧縮、および MC の概念とアルゴリズムの詳細については、「参考資料」に示した資料を参照してほしい。AP-529 のセクション 2 では、MPEG フレーム構造について簡単に説明している。参考資料[2]には、MPEG-2 圧縮の標準規格について、基礎から仕様まで詳しく説明している。

以下の説明は参考資料[2]と参考資料[3]に基づいて行う。ビデオ・シーケンスを構成する各フレームは、フレームごとにわずかに異なるいくつかの領域で構成される。MPEG のようなビデオ圧縮スキームでは、フレーム内容の類似性を利用するフレーム間コーディングを採用している。MC とは、「フレーム間で移動するオブジェクトの変異を補正するプロセス」と定義できる(参考資料[3]、91 ページ)。MC を適用する基本単位は、マクロブロック・レベルで定義される。マクロブロックは、16×16 ピクセルの 1 つの輝度コンポーネント(Y コンポーネントともいう)と、8×8 の 2 つのクロミナンス・コンポーネント(Cb および Cr コンポーネントともいう)で構成される。

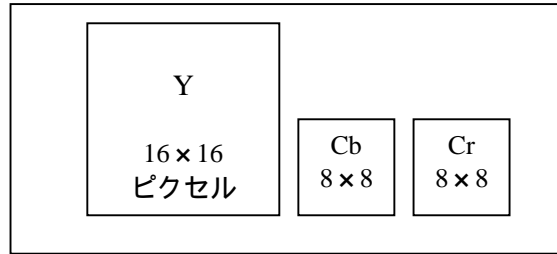


図 1: マクロブロック構造 – 16×16 の Y、8×8 の Cb、8×8 の 3 つの Cr コンポーネント

カレント・フレーム内の指定されたマクロブロックに MC を実行するには、あらかじめ計算した 1 つまたは 2 つのリファレンス・フレームと対応するモーション・ベクトル(オフセット)を使用する。モーション・ベクトルとは、カレント・マクロブロックに相対的なリファレンス・マクロブロック位置を示すものである。MC を実行するには、単純にリファレンス・マクロブロックをコピーする方法と、リファレンス・マクロブロックの隣接ピクセルの平均化による補間で、変わり目を滑らかにする方法がある。

リファレンス・マクロブロックにおける隣接ピクセルの平均化(ハーフピクセル予測ともいう)は、次のどれかに対して行われる。

- 水平方向に隣接する 2 つのピクセル
- 垂直方向に隣接する 2 つのピクセル
- 4 つの隣接ピクセル

独立した 2 つのリファレンス・マクロブロックを使用する場合は、実行結果の 2 つのマクロブロック間でさらに平均化を行う。

Y 平面と Cb および Cr 平面の両方で同じモーション・ベクトルと同じモーション・モード(例えば、ハーフピクセル方向タイプ)が使用される。このアプリケーション・ノートに示すコード例では、Y ピクセルしか計算していないが、同様のコードで Cb および Cr ピクセルを計算できる。

SSE2 と SSE が登場するまでは、MC は複雑で多大なメモリ・オーバーヘッドをもたらすものだった。

SSE2 と SSE を使用すると、`pavgb` という特別な平均化命令が利用できること、また SIMD 並列処理が 8 要素から 16 要素に拡張されたことにより、計算の複雑さを大きく軽減できる。

MC は、プロセッサ・キャッシュから消えてしまったフレーム内のマクロブロックを 1 つまたは 2 つ参照するため、メモリへの負荷が大きいモジュールである。プリフェッチ命令を使用すると、このようなメモリ・オーバーヘッドを大きく軽減できる可能性がある。

2.1 モーション補正を使用するアプリケーション

モーション補正(MC)アルゴリズムは、動画をビデオ圧縮するのに使用される。MPEG(MPEG-1、MPEG-2、MPEG-4)、H.261 などの標準規格でも使用している。このアプリケーション・ノートでは、特に MPEG-2 デコード/エンコードにおける使用方法を説明する。

MC アルゴリズムは、MPEG-2 デコードの主要モジュールで、デコーダの実行時間の 14 ~ 20% を占めている。MPEG-2 エンコードでも実行されるが、エンコーダの実行時間に占める割合は小さい。

次のブロック・ダイアグラムは、MPEG エンコード/デコード・プロセスにおける、MC アルゴリズムと他のアルゴリズムの関係を示す。

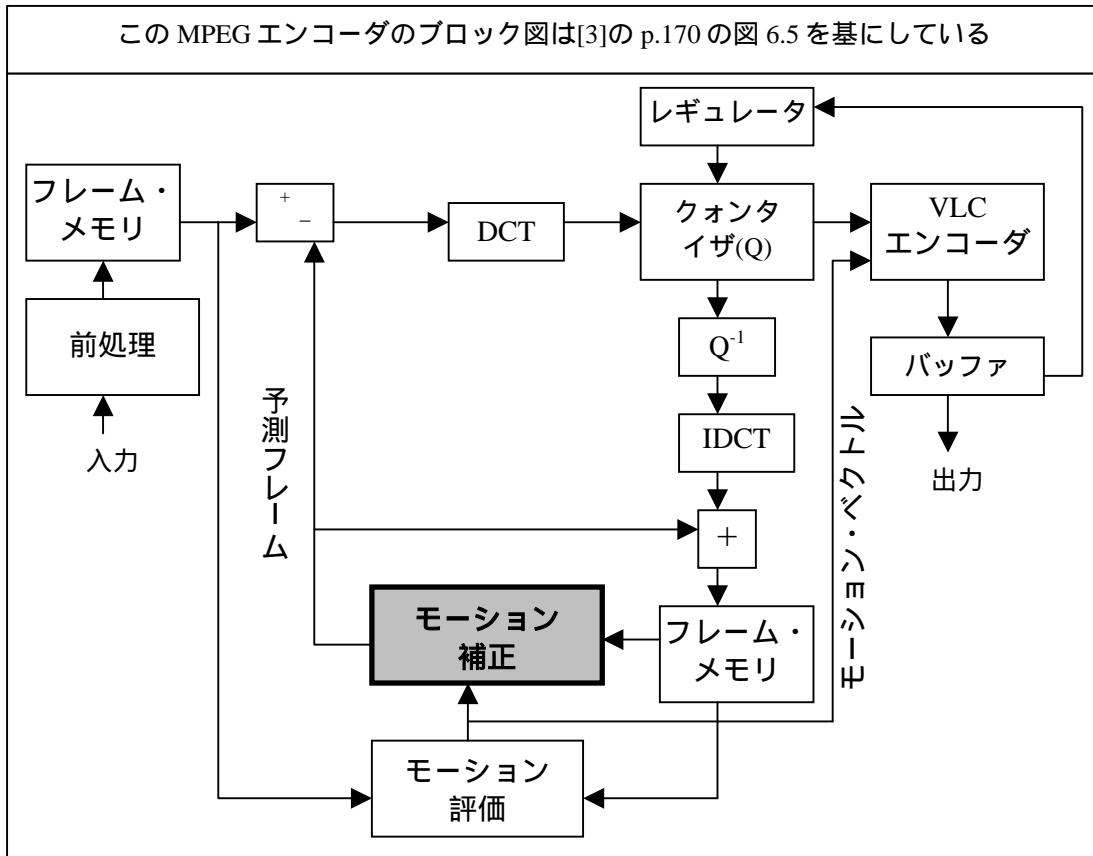


図 2: MPEG エンコーダのブロック図

2.2.1 命令選択と精度に関する手法

パックド平均化命令の `pavgb` を使用すると、AP-529 に示したアルゴリズムより高速かつ正確に MC アルゴリズムを実現できる。SSE2 では、`pavgb` 命令が拡張され、16 ピクセル(バイト)を並列に平均化できるようになった。

`pavgb` 命令は、MPEG 標準に正確に準拠して、2 つのピクセルを平均化するので、符号なしデータが最も近い整数に切り上げられる。最も正確かつ高速に実行できるのは、`pavgb` を使用して 2 つのピクセルを平均化するモーション・モードである。

4 つのピクセルを平均化する場合は、単純に平均化を 3 回行うと、+1 の誤差が生じる可能性がある。すなわち、 $\text{avg}(a, b, c, d) = \text{avg}[\text{avg}(a, b), \text{avg}(c, d)]$ のように、4 つのデータ要素 a, b, c, d に対して、単純な 2 段階の平均化を行うとする。第 1 段階の 2 つの平均化、すなわち $\text{avg}(a, b)$ と $\text{avg}(c, d)$ で +0.5 の誤差が生じる可能性がある。次に、第 1 段階の結果を平均化する第 2 段階の平均化、すなわち $\text{avg}[\text{avg}(a, b), \text{avg}(c, d)]$ でさらに +0.5 の誤差が生じる可能性がある。したがって、最終結果は第 1 段階の誤差 +0.5 と第 2 段階の誤差 +0.5 により、最大 +1 の誤差が生じる可能性がある。例えば、5、2、0、1 の 4 つの値を平均化する場合を考えてみよう。 $(5+2+0+1)/4=2$ なので、正しい結果は 2 である。ところが、上記のような 2 段階の方法で平均化すると、 $\text{avg}[\text{avg}(5, 2), \text{avg}(0, 1)] = \text{avg}[4, 1] = 3$ となり、+1 の誤差が生じる。このとき、0.5 は切り上げられることに注意する。 $(5+2)/2=3.5$ なので $\text{avg}(5, 2)=4$ となり、同様に $\text{avg}(0, 1)=1$ 、 $\text{avg}(4, 1)=3$ となる。

例 1 に、4 つのピクセルを単純に平均化するコードを示す。

```

movdqa xmm0, XMMWORD PTR [ a16 ]      ; load 16 pixels from src "a"
movdqa xmm1, XMMWORD PTR [ b16 ]      ; load 16 pixels from src "b"
movdqa xmm2, XMMWORD PTR [ c16 ]      ; load 16 pixels from src "c"
movdqa xmm3, XMMWORD PTR [ d16 ]      ; load 16 pixels from src "d"
pavgb  xmm0, xmm1 ; avg_1=simd_avg(a16,b16); // 1st stage averaging #1
pavgb  xmm2, xmm3 ; avg_2=simd_avg(c16,d16); // 1st stage averaging #2
pavgb  xmm0, xmm2 ; result=simd_avg(avg_1,avg_2) ); // 2nd stage avg

```

例 1: 単純な 4 ピクセル平均化、16 の結果を並列計算する

上記のような単純な 2 段階平均化を行うと、約 37% のピクセルで +1 の誤差が生じる。この誤差を最小にするには、次のコード例のように、第 1 段階で求めた平均値のどちらかから 1 を引いておく。

```

movdqa xmm0, XMMWORD PTR [ a16 ]      ; load 16 pixels from src "a"
movdqa xmm1, XMMWORD PTR [ b16 ]      ; load 16 pixels from src "b"
movdqa xmm2, XMMWORD PTR [ c16 ]      ; load 16 pixels from src "c"
movdqa xmm3, XMMWORD PTR [ d16 ]      ; load 16 pixels from src "d"
pavgb  xmm0, xmm1                      ; 1st stage averaging #1
pavgb  xmm2, xmm3                      ; 1st stage averaging #2
      ; avg_1 = sat_sub( avg_1, const_1_16_bytes )
psubusb xmm0, XMMWORD PTR [const_1_16_bytes] ; compensate errors
pavgb  xmm0, xmm2                      ; 2nd stage avg

```

例 2: より正確な 4 ピクセル平均化、16 の結果を並列計算する

このような簡単で高速な修正を行うと、-1 の誤差が生じる可能性が 13% のピクセルになる。このアプリケーション・ノートに示すコード例でも、この方法を使用している。もちろん、他の正確なモードを使用した場合の方が、MC アルゴリズム全体の誤差分布率は低くなる。

もう 1 つの解決方法として、4 ピクセルを平均化する正確なコードを使用する方法がある。この方法は、正確ではあるが計算量が多く、パフォーマンス上は最適とはいえないので、正確さが最も要求される場合にのみ使用する。インテル® C/C++ コンパイラ向けの、SIMD 演算のための C++ クラス・ライブラリを使用して、正確な平均化を行うコード例を次に示す。

```
Iu8vec16 Accurate_4pels_average ( Iu8vec16 a16, Iu8vec16 b16,
                                   Iu8vec16 c16, Iu8vec16 d16 )
{
    Iu8vec16 avg_1, avg_2, result;
    Iu8vec16 half_err_1st, half_err_2nd, fixup_mask;

    avg_1 = simd_avg(a16, b16);          // 1st stage averaging #1
    avg_2 = simd_avg(c16, d16);          // 1st stage averaging #2
    result = simd_avg(avg_1, avg_2);     // 2nd stage averaging

    // record 1 in lsb <=> some avg in the 1st stage carries +0.5 error
    half_err_1st = (a16 ^ b16) | (c16 ^ d16);
    // record 1 in lsb <=> 2nd stage avg carries at least +0.5 error
    half_err_2nd = ( avg_1 ^ avg_2 );
    // record 1 <=> result carries +1 error
    fixup_mask = half_err_1st & half_err_2nd & (*(Iu8vec16*)const_1_16_bytes);

    return ( result - fixup_mask );     // compensate error where needed
}
```

例 3: pavgb 命令(SIMD 幅は 16)を使用する正確な 4 ピクセル平均化

3 パフォーマンス

SSE2 を使用して、MC アルゴリズムを実現すると、従来のテクノロジーを使用した場合と比べてパフォーマンスが大きく向上する。このセクションでは、SSE2 を使用して得られるパフォーマンス向上を示し、関連するプログラミング上の考察を行う。

3.1 パフォーマンス向上

AP-529 では、MMX テクノロジーによる MC アルゴリズムの最適化を行った。このアプリケーション・ノートでは、SSE2 を使用してさらなる最適化を行う。

高速化の要因を次に示す。

- 8 ピクセルではなく 16 ピクセルの同時処理を可能にする、128 ビットの SIMD 整数命令
- 高速で正確な `pavgb` 命令
- 次のような Pentium 4 プロセッサのマイクロアーキテクチャの改良
 - ハードウェア・プリフェッチャ
 - キャッシュとメモリに対する高速かつ幅の広いデータ・アクセス
 - 分岐予測メカニズムの改良

3.2 その他の考察

3.2.1 ソフトウェア・プリフェッチ命令の使用

MC アルゴリズムでは、キャッシュに入っていないデータにアクセスすることが多い。通常は、ソフトウェア・プリフェッチ命令をループに組み込むことで、メモリ・レイテンシの影響を最小にできる。ところが、Pentium 4 プロセッサを使用して、MC アルゴリズムのパフォーマンス評価を行ったところ、MC カーネルでプリフェッチ命令を使用する効果は見られなかった。

これは、Pentium 4 プロセッサのマイクロアーキテクチャがハードウェア・プリフェッチ・メカニズムをサポートしているためと考えられる。MC アルゴリズムで行われるような、一定間隔でロードを繰り返すような操作は、Pentium 4 プロセッサのプリフェッチ・メカニズムの恩恵を最も受けやすい。

また、MC アルゴリズムで行う計算が短かすぎて、プリフェッチ・レイテンシが完全には隠れないのも理由の 1 つである。ただし、プリフェッチ命令を MC カーネルの外側で使用すれば、プリフェッチ・レイテンシをより隠蔽できる。例えば、iDCT カーネル内部では、計算が主体でメモリ・トラフィックが低いため、プリフェッチが全般的パフォーマンスに貢献する。同様に、デコード/エンコード・アルゴリズムを実行するアプリケーションでいくつかのマクロブロックをまとめて処理する場合は、MC カーネルの外側でプリフェッチを行うことでパフォーマンスを向上できる。

3.2.2 データ・アクセスとアライメント

モーション・ベクトルは任意のアドレスをポイントする可能性があるため、アライメントは一切保証されない。このアプリケーション・ノートに示すコード例は、すべてのモーション・ベクトルがアライメントされていないものと見なし、アライメントされていないデータのロード命令である `movdqu` を使用している。

この他に、アライメントされていないポインタをアライメントする方法がある。その場合は、目的の(アライメントされていない)データを含む(アライメントされた)データ範囲を2度に分けてロードし、`shift` または `or` 命令を使用して、必要なデータを取り出す。この方法は、シフト量が一定か、64 ビット・レジスタの上位または下位半分に限定されているときに使用できる。モーション・ベクトルは、このケースに当てはまらない。

MC の場合は、アライメントされていないデータをシフトする負担の方が、アライメント・ロードの利点より大きい。

4 結論

SSE2 は、MPEG-2 デコード/エンコードのための MC アルゴリズムのパフォーマンスを大きく向上させる。

次のようなアーキテクチャ改良が、MC のパフォーマンス向上に貢献している。

- SIMD 幅の拡張
- `pavgb` 命令

この他に、次のマイクロアーキテクチャ改良もパフォーマンス向上に貢献している。

- プリフェッチ・アシスタンス
- 分岐予測の改良

5 C/C++によるコード例

次のコード例は、Yコンポーネントのみに対して、5つのモードのMCアルゴリズムを実行する。

```
#include <windows.h>
#include <dvec.h>
//-----
// Global Declarations
_MM_ALIGN16 BYTE const_1_16_bytes[] =
    { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
//
// my_loadu() is used for unaligned loads, for more readable code than
// currently supported by Intel® C/C++ Compiler.
//
inline Iu8vec16 my_loadu(M128 *p)
{ return ( _mm_loadu_si128( (__m128i*)p ) ); }

//-----
// Full motion (no average) from one frame only
//
void mc_WmtNI_full_b( DWORD stride, DWORD lines,
    BYTE *dst_start, BYTE *bw_start )
{
    DWORD y;
    // num of 128 bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
    Iu8vec16 *src = (Iu8vec16*) bw_start;
    Iu8vec16 *dst = (Iu8vec16*) dst_start;

    // simply copy all the 16-pels lines from reference to dst
    for (y=0; y<lines; y++)
        dst[ y*stride_128 ] = my_loadu( (M128*)src + y*stride_128 );
}
//-----
// Average half-pels horizontally (in the "X" axis),
// from one reference frame only
//
void mc_WmtNI_hx_b( DWORD stride, DWORD lines,
    BYTE *dst_start, BYTE *bw_start )
{
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
```

```
DWORD stride_128 = stride >> 4;
Iu8vec16 *dst = (Iu8vec16*) dst_start;

// avg each ref pixel with its adjacent pel, and store to dst
for (y=0; y<lines; y++)
{
    dst[ y*stride_128 ] = simd_avg(
        my_loadu( (M128*)( bw_start + y*stride ) ),
        my_loadu( (M128*)( bw_start + y*stride + 1 ) ) );
}

}

//-----
// Average half-pels vertically (in the "Y" axis),
// from one reference frame only
//
void mc_WmtNI_hy_b( DWORD stride, DWORD lines,
                   BYTE *dst_start, BYTE *bw_start )
{
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
    Iu8vec16 *dst = (Iu8vec16*) dst_start;

    // avg each ref pixels with the pel from line below, and store to dst
    for (y=0; y<lines; y++)
    {
        dst[ y*stride_128 ] = simd_avg(
            my_loadu( (M128*)( bw_start + y*stride ) ),
            my_loadu( (M128*)( bw_start + (y+1)*stride ) ) );
    }
}

//-----
// Average half-pels both horizontally and vertically ("X" and "Y" axis),
// from one reference frame only
//
void mc_WmtNI_hx_hy_b( DWORD stride, DWORD lines,
                      BYTE *dst_start, BYTE *bw_start )
{
    Iu8vec16 avg_above, avg_below;
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
```

```

Iu8vec16 *dst = (Iu8vec16*) dst_start;

// avg each 2x2 ref pixels, and store to dst
for (y=0; y<lines; y++)
{
    // avg 2 pels from upper line
    avg_above = simd_avg( my_loadu((M128*)(bw_start + y*stride)),
                          my_loadu((M128*)(bw_start + y*stride + 1)));
    // avg 2 pels from line below
    avg_below = simd_avg( my_loadu((M128*)(bw_start + (y+1)*stride)),
                          my_loadu((M128*)(bw_start + (y+1)*stride + 1)));
    // compensate possible error for 4-pels avg accuracy
    avg_above = sat_sub( avg_above, *(Iu8vec16*)const_1_16_bytes );
    // avg the two lines
    dst[ y*stride_128 ] = simd_avg( avg_above, avg_below );
}
}
//-----
// Average from two frames, each frame has 2x2 half-pels avg
//
void mc_WmtNI_hx_hy_b_hx_hy_f( DWORD stride, DWORD lines,
                               BYTE *dst_start, BYTE *fw_start, BYTE *bw_start )
{
    Iu8vec16 bw_avg, fw_avg, avg_above, avg_below;
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
    Iu8vec16 *dst = (Iu8vec16*) dst_start;

    for (y=0; y<lines; y++)
    {
        // 4-pels (2x2) avg from one reference frame
        avg_above = simd_avg( my_loadu((M128*)(bw_start + y*stride)),
                              my_loadu((M128*)(bw_start + y*stride + 1)));
        avg_below = simd_avg( my_loadu((M128*)(bw_start + (y+1)*stride)),
                              my_loadu((M128*)(bw_start + (y+1)*stride + 1)));
        avg_above = sat_sub( avg_above, *(Iu8vec16*)const_1_16_bytes );
        bw_avg = simd_avg( avg_above, avg_below );

        // 4-pels (2x2) avg from another reference frame
        avg_above = simd_avg( my_loadu((M128*)(fw_start + y*stride)),
                              my_loadu((M128*)(fw_start + y*stride + 1)));
        avg_below = simd_avg( my_loadu((M128*)(fw_start + (y+1)*stride)),

```

```
        my_loadu((M128*)(fw_start + (y+1)*stride + 1));
    avg_above = sat_sub( avg_above, *(Iu8vec16*)const_1_16_bytes );
    fw_avg = simd_avg( avg_above, avg_below );

    // avg result from both reference frame
    dst[ y*stride_128 ] = simd_avg( bw_avg, fw_avg );
}
}
```

6 SSE2 を使用したアセンブリ・コード例

次のコードは、前セクションと同じコードを、SSE2 を使用してアセンブリ言語で書いたものである。

```
//-----
// Full motion (no average) from one frame only
//
void mc_WmtNI_full_b( DWORD stride, DWORD lines,
                    BYTE *dst_start, BYTE *bw_start )
{
    __asm
    {
        mov     edx, bw_start      ;
        mov     ecx, dst_start    ;
        mov     esi, lines        ;
        mov     eax, stride       ;
        lea    edi, [eax + eax]   ; stride*2

        ; simply copy all the 16-pels lines from reference to dst
        ; each loop iteration copies 2 lines
nextLinesLoop:
        movdqu  xmm0, XMMWORD PTR [edx]      ; copy 2 lines from ref frame
        movdqu  xmm1, XMMWORD PTR [edx+eax] ;

        movdqa  XMMWORD PTR [ecx], xmm0     ; and store to dst lines
        movdqa  XMMWORD PTR [ecx+eax], xmm1 ;

        add     edx, edi                ; advance 2 lines ahead
        add     ecx, edi
        sub     esi, 2
        jg     nextLinesLoop
    }
}

//-----
// Average half-pels horizontally (in the "X" axis),
// from one reference frame only
//
void mc_WmtNI_hx_b( DWORD stride, DWORD lines,
                  BYTE *dst_start, BYTE *bw_start )
{
    __asm
```

```

{
    mov     edx, bw_start      ;
    mov     ecx, dst_start    ;
    mov     eax, stride       ;
    mov     esi, lines        ;
    lea     edi, [eax + eax]   ; stride*2

    ; avg each ref pixel with its adjacent pel, and store to dst
    ; each loop iteration computes 2 lines
nextLineLoop:
    movdqu xmm0, XMMWORD PTR [edx]      ; load 16 pels
    movdqu xmm1, XMMWORD PTR [edx+1]    ; and their adjacent pels
    movdqu xmm2, XMMWORD PTR [edx+eax]  ; do the same for line below it
    movdqu xmm3, XMMWORD PTR [edx+eax+1]

    pavgb  xmm0, xmm1           ; avg one line
    pavgb  xmm2, xmm3           ; avg second line

    movdqa XMMWORD PTR [ecx], xmm0     ; and store the results
    movdqa XMMWORD PTR [ecx+eax], xmm2

    add    edx, edi               ; advance 2 lines ahead
    add    ecx, edi
    sub    esi, 2
    jg     nextLineLoop
}
}

```

```

//-----
// Average half-pels vertically (in the "Y" axis),
// from one reference frame only
//
void mc_WmtNI_hy_b( DWORD stride, DWORD lines,
                   BYTE *dst_start, BYTE *bw_start )
{
    __asm
    {
        mov     edx, bw_start      ;
        mov     ecx, dst_start    ;
        mov     eax, stride       ;
        mov     esi, lines        ;
        lea     edi, [eax + eax]   ; stride*2
    }
}

```

```

; avg each ref pixels with the pel from line below, and store to dst
; each loop iteration computes 2 result lines

    movdqu  xmm0, XMMWORD PTR [edx]          ; prepare data for 1st iter
nextLineLoop:
    ; xmm0 already holds the first line pels
    movdqu  xmm1, XMMWORD PTR [edx+eax]     ; load 2nd line
    movdqu  xmm2, XMMWORD PTR [edx+edi]     ; and 3rd line

    pavgb   xmm0, xmm1                      ; avg 1st line with 2nd
    pavgb   xmm1, xmm2                      ; avg 2nd line with 3rd

    movdqa  XMMWORD PTR [ecx], xmm0         ; store result to first line
    movdqa  xmm0, xmm2                     ; save "first" line of next iter
    movdqa  XMMWORD PTR [ecx+eax], xmm1     ; store result to second line

    add     edx, edi                        ; advance 2 lines ahead
    add     ecx, edi
    sub     esi, 2
    jg     nextLineLoop
}
}
//-----
// Average half-pels both horizontally and vertically ("X" and "Y" axis),
// from one reference frame only
//
void mc_WmtNI_hx_hy_b( DWORD stride, DWORD lines,
                      BYTE *dst_start, BYTE *bw_start )
{
    __asm
    {
        mov     edx, bw_start              ;
        mov     ecx, dst_start             ;
        mov     eax, stride                 ;
        mov     esi, lines                  ;
        lea    edi, [eax + eax]            ; stride*2

        ; avg each 2x2 ref pixels, and store to dst

        movdqa xmm7, XMMWORD PTR [const_1_16_bytes] ; used for higher accuracy

        movdqu  xmm0, XMMWORD PTR [edx]          ; prepare data for 1st iter
        movdqu  xmm1, XMMWORD PTR [edx+1]       ; prepare data for 1st iter

```

```

nextLineLoop:
    ; xmm0 already holds the first line pels
    ; xmm1 already holds the first line adjacent pels
    movdqu xmm2, XMMWORD PTR [edx+eax]      ; load 2nd line
    movdqu xmm3, XMMWORD PTR [edx+eax+1]    ; and its adjacent pels
    movdqu xmm4, XMMWORD PTR [edx+edi]      ; load 3rd line
    movdqu xmm5, XMMWORD PTR [edx+edi+1]    ; and its adjacent pels

    pavgb  xmm0, xmm1      ; horizontal avg of 1st line
    pavgb  xmm2, xmm3      ; horizontal avg of 2nd line
    movdqa xmm1, xmm5      ; save "first" adjacent line for next iter
    pavgb  xmm5, xmm4      ; horizontal avg of 3rd line

    psubusb xmm2, xmm7      ; compensate error for accuracy

    pavgb  xmm0, xmm2      ; vertical avg for 1st dst line
    pavgb  xmm2, xmm5      ; vertical avg for 2nd dst line

    movdqa XMMWORD PTR [ecx], xmm0          ; store 1st line to dst
    movdqa xmm0, xmm4                      ; save "first" line for next iter
    movdqa XMMWORD PTR [ecx+eax], xmm2     ; store 2nd line to dst

    add    edx, edi          ; advance 2 lines ahead
    add    ecx, edi
    sub    esi, 2
    jg     nextLineLoop
}
}
//-----
// Average from two frames, each frame has 2x2 half-pels avg
//
void mc_WmtNI_hx_hy_b_hx_hy_f( DWORD stride, DWORD lines,
                               BYTE *dst_start, BYTE *fw_start, BYTE *bw_start )
{
    __asm
    {
        mov    edx, bw_start      ;
        mov    edi, fw_start      ;
        mov    ecx, dst_start     ;
        mov    eax, stride        ;
        mov    esi, lines         ;

        movdqa xmm7, XMMWORD PTR [const_1_16_bytes] ; used for higher accuracy
    }
}

```

```
movdqu xmm0, XMMWORD PTR [edx]      ; prepare data for 1st iter
movdqu xmm1, XMMWORD PTR [edx+1]

movdqu xmm4, XMMWORD PTR [edi]
movdqu xmm5, XMMWORD PTR [edi+1]

nextLineLoop:
    ; xmm0 already holds the first bw line pels
    ; xmm1 already holds the first bw line adjacent pels
movdqu xmm2, XMMWORD PTR [edx+eax]    ; load 2nd bw line
movdqu xmm3, XMMWORD PTR [edx+eax+1]  ; and its adjacent pels

    ; xmm4 already holds the first fw line pels
    ; xmm5 already holds the first fw line adjacent pels
movdqu xmm6, XMMWORD PTR [edi+eax]    ; load 2nd fw line

pavgb xmm0, xmm1      ; horizontal avg of 1st bw line
movdqa xmm1, xmm3     ; save "first" adjacent bw line for next iter
pavgb xmm3, xmm2     ; horizontal avg of 2nd bw line

psubusb xmm3, xmm7   ; compensate error for accuracy

pavgb xmm3, xmm0     ; vertical avg for bw line
movdqa xmm0, xmm2    ; save "first" bw line for next iter

movdqu xmm2, XMMWORD PTR [edi+eax+1] ; load 2nd fw line (adjacent)
pavgb xmm4, xmm5     ; horizontal avg of 1st fw line
movdqa xmm5, xmm2    ; save "first" adjacent fw line for next iter
pavgb xmm2, xmm6     ; horizontal avg of 2nd fw line

psubusb xmm2, xmm7   ; compensate error for accuracy

pavgb xmm2, xmm4     ; vertical avg for fw line
movdqa xmm4, xmm6    ; save "first" fw line for next iter

pavgb xmm3, xmm2     ; final result: avg bw and fw frames

movdqa XMMWORD PTR [ecx], xmm3      ; store to dst

add    edx, eax      ; advance 1 line ahead
add    edi, eax
add    ecx, eax
```

```
    sub    esi, 1
    jg     nextLineLoop
  }
}
```

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	1.20 GHz 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

このアプリケーション・ノートに示すコード例は、モーション補正アルゴリズムを次の3つの方法で実現している。

1. SIMD 演算のためのインテル® C++ SIMD クラス・ライブラリ(IVEC)
2. ストリーミング SIMD 拡張命令 2(SSE2)を使用したアセンブリ言語
3. C++ SIMD クラス - SSE2(DVEC)

今回、第3バージョンのパフォーマンス測定は行わなかった。

表 1: モーション補正(MC)プログラムのパフォーマンス・データ(コールド・キャッシュ)

パフォーマンス・データ(反復に要するミリ秒) – コールド・キャッシュ				
	インテル® Pentium® III プロセッサ		インテル® Pentium® 4 プロセッサ	
	full_b	hx_hy_b_hx_hy_f	full_b	hx_hy_b_hx_hy_f
SSE IVEC	2.58	5.17	1.54	3.39
SSE2 ASM	-	-	1.48	3.24

表 2: 表 1 のパフォーマンス・データから求めた高速化(コールド・キャッシュ)

コーディング方法とプラットフォーム	full_b の高速化	hx_hy_b_hx_hy_f の高速化
Pentium 4 プロセッサ(SSE2 ASM vs. SSE IVEC)	1.04	1.05
SSE IVEC(Pentium 4 プロセッサ vs. Pentium III プロセッサ)	1.67	1.52
SSE2 ASM と Pentium 4 プロセッサ vs. SSE と Pentium III プロセッサ	1.74	1.60

表 3: MC プログラムのパフォーマンス・データ(完全キャッシュ)

パフォーマンス・データ(反復に要するミリ秒) - 完全キャッシュ				
	Pentium III プロセッサ		Pentium 4 プロセッサ	
	full_b	hx_hy_b_hx_hy_f	full_b	hx_hy_b_hx_hy_f
SSE IVEC	0.434	1.19	0.373	0.990
SSE2 ASM	-	-	0.260	0.838

表 4: 表 3 のパフォーマンス・データから求めた高速化(完全キャッシュ)

コーディング方法とプラットフォーム	full_b の高速化	hx_hy_b_hx_hy_f の高速化
Pentium 4 プロセッサ(SSE2 ASM vs. SSE IVEC)	1.435	1.181
SSE IVEC(Pentium 4 プロセッサ vs. Pentium III プロセッサ)	1.16	1.20
SSE2 ASM と Pentium 4 プロセッサ vs. SSE と Pentium III プロセッサ	1.67	1.42

パフォーマンス測定は、ランダムなモーション・ベクトルを使用した人工的なテスト環境で行った。また、フレームごとにキャッシュ内容を壊した「コールド・キャッシュ」環境と、「完全キャッシュ」環境の両方で測定した。パフォーマンス測定に使用したのは、733 MHz Pentium III プロセッサと 1.20 GHz Pentium 4 プロセッサである。使用したシステムの詳細は、26ページの「テスト・システム構成」を参照のこと。

表 2 の「コールド・キャッシュ」環境の測定結果からわかるように、Pentium 4 プロセッサで hx_hy_b_hx_hy_f モーション・モードを実行した場合、SSE2 は SSE の 1.05 倍高速になっている。

表 4 の「完全キャッシュ」環境では、メモリの影響がなくなり、パフォーマンスが向上している。Pentium 4 プロセッサで hx_hy_b_hx_hy_f モーション・モードを実行した場合は、SSE2 は SSE の 1.181 倍高速になっている。

hx_hy_b_hx_hy_f モードは、計算とメモリ帯域幅の両方に最も負荷が大きいモードである。full_b モードはメモリをコピーするだけで、計算はいっさい行わない。このモードは、「コールド・キャッシュ」環境では 1.04 倍、「完全キャッシュ」環境では 1.435 倍高速になった。このアプリケーション・ノートに示した他のモードは、hx_hy_b_hx_hy_f モードより計算量が少なく、メモリ・トラフィックは full_b と同じである。それらのモードのパフォーマンスは測定していない。

上記の高速化は、次のような最適化の結果である。

- 拡張された SIMD 幅。整数 SSE2 は、64 ビットの MMX テクノロジ・レジスタではなく、128 ビットの XMM レジスタを使用する。SIMD 幅が拡張されたため、レジスタへの負荷が減り、1 つの命令で処理できるデータ量が 2 倍になった。
- SSE2 を使用したコードも SSE を使用したコードも、pavgb 命令を利用している。この命令は、AP-529 で示した MMX テクノロジ・コードより高速である。Pentium III プロセッサで SSE を実行するプログラムは、prefetch 命令でソフトウェア・プリフェッチを行っている。

テスト・システム構成

表 5: 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	インテル® 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 6: 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(1.20 GHz)
システム	インテル 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 Size	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