卒業論文

題目

動画像圧縮符号化規格 H.265/HEVC研究用エンコーダの 高速化

指導教員

近藤　利夫 教授

2014年

三重大学 工学部 情報工学科
計算機アーキテクチャ研究室

梶川　恭祐 (410808)

内容梗概

近年，従来のハイビジョンの 16 倍の画素数を有するスーパーハイビジョンが 2020 年の本放送を目指し，開発が進められているなど，動画像の高精細化が進んでいる．その高精細化に伴って，現在主流の H.264/AVC の約 2 倍の圧縮性能を持つ新しい符号化複合化規格である H.265/HEVC の標準化が昨年 1 月に完了している．このような状況の中，当研究室では，高効率の H.265 エンコーダ実現にむけて，JCT-VC によって提供されている HM(HEVC Test Model) と呼ばれる参照ソフトウェアを用いて，圧縮処理手法の実装，評価を行っている．しかし現在の HM はわずか 5 フレーム程度のエンコードに 5 分程と試行的なエンコードを繰り返すにはあまりに遅く，研究効率低下の原因となっている．そこで圧縮特性不変の条件で大幅な高速化を目指し，SIMD 命令による並列処理を実装した．その結果，実装前の HM と比べて約 33%の高速化を達成した．

Abstract

In recent years, high resolution video technology has been developed in order to start broadcast of UHD having 16 times definition of HD in 2020. In fact, in January last year, standardization of the new video compression standard H.265/HEVC havings about twice the compression performance of the conventional standard H.264/AVC was completed. At this laboratory, for and evaluation of efficient HEVC encording teqnique, we must use the reference software HM(HEVC Test Model) provided by JCT-VC. But original HM takes about 5 minutes for encording only 5 frames. Thus HM is too late for repeating experimental evaluation. Therefore, it is necessary to optimiz HM code, and raise evaluation efficiency. I examined performance profilling to detect the bottleneck of HEVC encording in the HM, and implement SIMD parallel processing at the conditions that compression performance is invariable. As a result, compared with original HM, the execution time was reduced by about 33 percent.

目 次

図 目 次

表 目 次

1　まえがき

1.1　背景

近年，従来のハイビジョン (1920x1080) の 16 倍の画素数を有するスーパーハイビジョン (7680x4320) の本放送の開始が 2020 年に予定されるなど，動画像の高精細化が進んでいる．その高精細化によって，動画像の圧縮符号化は，処理量が大幅に増大し，その高効率化が必要不可欠となっている．そして H.264/AVC(Advanced Video Coding) は，これまで最も高い圧縮率を備えた動画像符号化規格として利用されてきたものの，10 年も前に標準化された規格であり，最近の超高精細化に見合う圧縮性能を得ることは期待出来ない．このような状況の中，動画像符号化の開発機関である MPEG(Moving Picture Experts Group) と VCEG(Video Coding Experts Group) の共同で設立された JCT-VC(Joint Collaborative Team on Video Coding) によって新規格の制定が進み，2013 年に H.264/AVC の約 2 倍の符号化効率を実現する H.265/HEVC(High Efficiency Video Coding) が標準化された (参考文献 [1])．これはモバイル端末から，超高精細のスーパーハイビジョンまでの利用が想定されており，伝送帯域の効率的な利用や急増する動画トラフィックの緩和に大きく貢献することが期待されている．すでに Apple 社の iPad は H.265 に対応しており，その他の製品へ

の普及も急速に進むものと考えられる．しかし H.265 ではその高精細化に対応するため，動画像の圧縮符号化に要する処理量が大幅に増える問題がある．よって今後の高精細化に対応するために，H.265 を遵守しながら，高効率を達成する処理方式の研究，開発が必要不可欠となる．

1.2 研究目的

JCT-VC によって HM(HEVC Test Model) と呼ばれている標準ソフトウェアが提供されており，H.265 規格に基づく，高効率の符号化方式，処理方式などの技術提案を行う場合，互いの優劣を比較するための基準の Codec ソフトウェアとして，この HM を多くの研究者が，内部に提案手法を組み込むことで，有効性の評価を行っている．当研究室でも，H.265 を準拠する高効率の符号化処理方式を研究しており，HM を提案方式の評価に広く利用している．しかし現在の HM ではわずか 5 フレーム程度のエンコードに 5 分ほどと試行的なエンコードを繰り返すにはあまりに遅い．エンコードが遅い理由として H.265 の特性評価のために高速処理より，規格への忠実な準拠を優先してきたことが挙げられる．すでに H.265 準拠の高速エンコーダ x265 の開発が進行しているが，x265 は圧縮性能の多少の低下を許容しているため，新しい符号化方式の研究する為のエン

コーダには適していない．また学会に発表する場合，HM の圧縮特性の維持は不可欠となる．以上の理由から x265 では無く HM を高速化することにより，その試行時間を削減し研究の効率を高めることを目指す．

2　HM 参照ソフトウェア

2.1　処理量の分析

HM は C++言語で記述されているソフトウェアエンコーダである．そして高速化を行うに当たって，どの関数で時間がかかっているか調べる必要があるため，実装環境として用いている Visual Studio のプロファイル機能を用いて処理量の多い関数を調査した．(図 2.1)

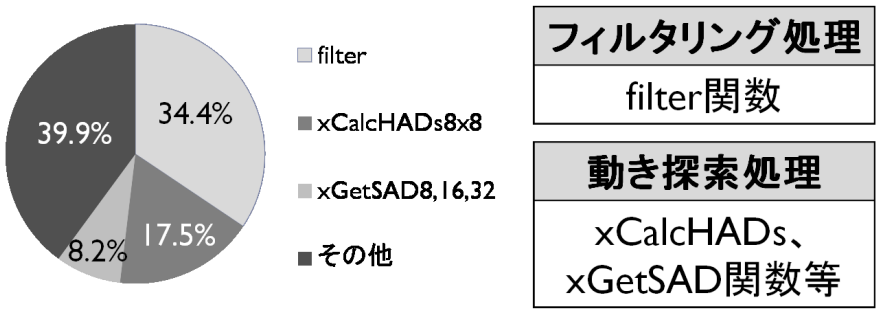

図 2.1: 関数ごとの処理割合

グラフに示すように，HM の符号化処理では SAD 演算などを含む動き探索処理やフィルタリング処理が処理量の大部分を占めており，高速化のボトルネックとなっていることがわかる．そのことから当研究室ではそ

の動き探索部分とそれに加えてフィルタリング処理にも有効なキャッシュメモリの高速化に励んでいる．そして今回の高速化の対象として，このグラフで大きく占める処理部分を取り上げる．

2.2 動き探索処理

動き探索処理は，符号化対象画像と参照画像内のブロックを探索し，ブロックマッチングで探索部分の類似度を評価することでそのブロックごとの動きベクトルを算出する．探索法として，HM では追跡型の拡張ダイヤモンド探索やラスタ探索などが実装されている．拡大ダイヤモンド探索は探索回数が少なく済むが，圧縮性能を高める場合，ラスタ探索も使用する．しかしその分探索回数が多くなってしまう．また探索ブロックの類似度の評価値として，主に xGetSAD 関数で算出される SAD(差分絶対値和) と xCalcHADs 関数で算出される SATD(アダマール変換絶対値和) が使用されている．SAD，SATD の用途の違いについては 4 章で説明する．それらの値を求める例を図 2.2 に示す.

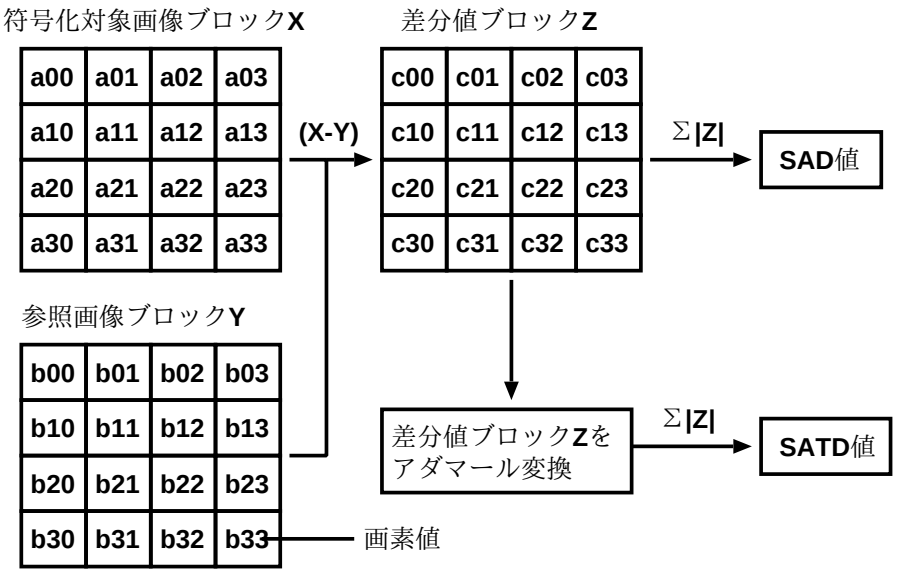

図 2.2: ブロック間での SAD 演算と SATD 演算

xGetSAD 関数は主に 8x8，16x16，32x32，64x64 で，xCalcHADs 関数は主に 4x4，8x8 などのブロックサイズから評価値を算出する．図 2.2 のような 4x4 ブロックの SAD 演算の場合，減算，絶対値化，加算を合計し 48 回の演算処理が行われる．そしてラスタ探索のように探索回数が多くなる場合，その演算量が高速化のボトルネックとなる．

2.3 フィルタリング処理

フィルタリング処理として filter 関数にて FIR(Finite Impulse Response) フィルタを行う．主な処理として filter 関数では画像ブロック内の各画素

に式 (1) の処理を行う．

$$d[n_0] = c_0 s[n_0] + c_1 s[n_1] + ... + c_7 s[n_7] \tag{1}$$

8 個ずつの画素を対象とし，d はフィルタリング後の画素値，c はフィルタ係数，s は元の画素値である．FIR フィルタは有限インパルス応答とも呼ばれ，入力信号である s の様に有限個の画素値を定数 c で重み付けし，その総和を出力信号 d とすることで雑音を取り除き，画質を安定させる処理である．SAD 演算や SATD 演算に無い乗算処理も多く，全体に占める処理割合が特に大きくなっている．

2.4 高速化の問題点

前述の処理を含む関数ごとに着目し高速化を目指す．そのためにそれらの計算量を削減する必要がある．しかし HM は，2013 年に Ver10 まで進められており，アルゴリズムの変更だけでは劇的な高速化は期待しにくい．また HM は比較の基準となる参照ソフトウェアとしての役割を果たす必要があるため，圧縮特性が犠牲になるような高速化を行うわけにはいかない．よってアルゴリズムの変更のみでない他の側面から高速化を考える必要がある．

3　並列化による高速処理法

3.1　並列化

当研究室の HM には並列化が実装されていない．そのため，並列化することで高速化が見込め，また既存のアルゴリズムに沿って実装することで，画像の劣化も抑えやすい．並列化にも種類があり，マルチスレッドと SIMD 命令を用いた並列化があるが，この内，改良する関数と相性の良い方式を選択し、その方式による高速化を目指す.

3.2　マルチスレッド

マルチスレッドは一つのタスクを複数のスレッドに分けて平行処理することで，そのタスクでの処理速度を向上させる．しかし HM は一度の関数の実行に多くの時間がかかることは無く，何度もそれらの関数が実行されることで処理時間が積み重なることで，多くの時間がかかる．今回は高速化のボトルネックとなる処理部分の関数を個別に高速化をすることを目指しているため，マルチスレッドによる高速化手法が最適とは言えない.

3.3 SIMD 命令 (Single Instruction Multiple Data)

SIMD 命令は一つの命令で複数のデータに対する処理が可能な並列処理である．つまりその一つの命令によって，例えば 32 ビットデータを使用した 4 回の同一の演算を繰り返すような操作の場合，逐次的に行うのではなく，一度にそれらの演算を行うことができる．SIMD 命令を使用した処理として，インテルが開発した CPU の SIMD 拡張命令セットである SSE(Streaming SIMD Extensions) を使用することで，最大で 128 ビット同士の演算ができる．SSE では x86 アーキテクチャを 32 ビットモードで動作させることで，8 本の xmm と呼ばれる 128 ビットレジスタが使用可能であり，32 ビットである 4 個の int 型データを一つの xmm レジスタに格納，また 16 ビットである short 型データの場合，8 個を格納することが可能である．このように xmm レジスタに要素を格納し，一括の SIMD 処理を行う．SSE を使用した SIMD 命令の動作を図 3.3 に示す．

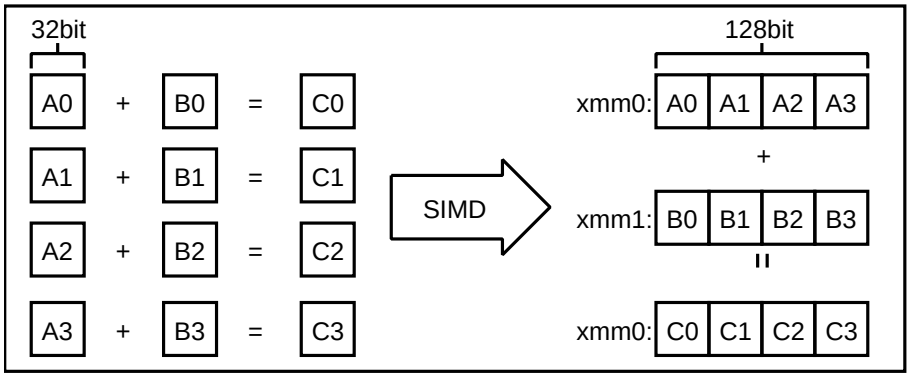

図 3.3: SIMD 演算例

またSSEの機能を追加したSSE2，SSE3，SSSE3(Supplemental Streaming SIMD Extensions 3) がある．SSE を含めたそれらの命令の一部を表 3.1 に示す.

表 3.1: SSE 命令の一部

命令名	オペランド	動作
MOVDQU	xmm1, xmm2/m128	xmm2 内の 128 ビット整数値を xmm1 に移動する.
PADDW	xmm1, xmm2/m128	xmm1 に xmm2 の 16 ビット単位で整数値を加算する.
PSUBW	xmm1, xmm2/m128	xmm1 に xmm2 の 16 ビット単位で整数値を減算する.
PMULLW	xmm1, xmm2/m128	xmm1 に xmm2 の 16 ビット単位で整数値を乗算する.
PABSW	xmm1, xmm2/m128	xmm2 の 16 ビット整数単位で絶対値を xmm1 に格納する.

HM のコードの一部を図 3.4 に示す.

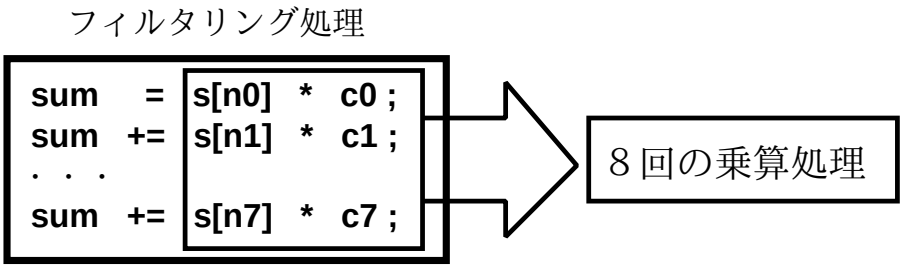

図 3.4: HM のフィルタリング処理コードの一部

フィルタリング処理として、この図のような 8 回ずつの乗算などの演算処理が繰り返されている．HM のコードはこのように同一の演算処理を繰り返す記述が多い．よってマルチスレッドでなく前述の特性から SIMD 命令による高速化を目指す．また SIMD 拡張命令セットとして SSE だけで無く，AVX も存在する．

3.3.1 Intel AVX(Intel Advanced Vector eXtensions) と AVX2

AVX はインテルが開発した SSE 後継の SIMD 拡張命令セットである．SSE と比べ，SIMD 演算幅が 2 倍の 256 ビットに拡張され，1 命令で 8 つの単精度浮動小数点演算もしくは 4 つの倍精度浮動小数点演算を実行することができる．そのため，AVX から xmm レジスタが 256 ビットに拡張された ymm レジスタが使用可能となり，そのため，ymm レジスタの下位 128 ビットが同じレジスタ番号の xmm レジスタとなっている．AVX

においては，従来の SSE で採用されていた 2 オペランド書式から 3 オペランド書式に改良された．そしてインテルによって Haswell マイクロアーキテクチャから AVX2 が搭載された．AVX2 では浮動小数点演算だけで無く，整数演算の演算幅も 256 ビットに拡張された．

AVX の命令の一部を表 3.2 に示す．

表 3.2: AVX 命令の一部

命令名	オペランド	動作
VMOVDQU	ymm1, ymm2/m256	ymm2 内の 256 ビット整数値を ymm1 に移動する．
VPADDW	xmm1, xmm2, xmm3/m128	xmm2 と xmm3 の 16 ビット単位整数値の加算値を xmm1 に格納する．
VPSUBW	xmm1, xmm2, xmm3/m128	xmm2 と xmm3 の 16 ビット単位整数値の減算値を xmm1 に格納する．
VPMULLW	xmm1, xmm2, xmm3/m128	xmm2 と xmm3 の 16 ビット単位整数値の乗算値を xmm1 に格納する．

そして SIMD 命令をコードに記述する手法として，インラインアセンブラが存在する．

3.3.2 インラインアセンブラ

インラインアセンブラは最も低レベルなアセンブリ言語で記述されたコードを C や C++などの高級言語のソースコードに埋め込む機能である．SIMD 命令は組み込み関数として使用可能であるが，アセンブリ言

語で SIMD 命令を使用することもできる．このインラインアセンブラによってコードを記述することで，その動きもわかりやすく，コンパイラの制約を受けることなく自由なコーディングが可能である為，高速化に繋がりやすい．インラインアセンブラを使用したコード例を図 3.5 に示す．

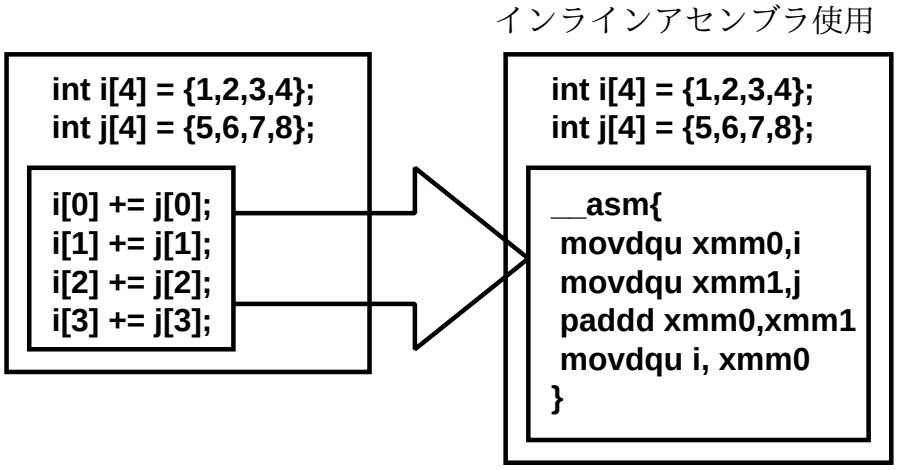

図 3.5: インラインアセンブラ使用例

4 SIMD 命令による並列化の実装

4.1 実装環境

Sandy Bridge 以降の新型の x86 プロセッサでは，SSE より最新の命令セットである AVX が使用が可能となる．また x86 プロセッサを 64 ビットモードで動作させることで，xmm レジスタが 16 個まで使用可能となる．しかし今回は実装部分の移植性を高めるため，前述の最新の機能は使用

せず，移植に適した環境で実装を行った．そのため，本研究ではSSEのみを使用し，そしてx86プロセッサを32ビットモードで動作させ，使用可能となる8個のxmmレジスタを使用する．またx86プロセッサに対し，前述のインラインアセンブラを使用してコードを記述することでSIMD処理を実装した．

4.2 filter関数

4.2.1 転置処理を使用する実装

filter関数の主な処理を図4.6に示す．

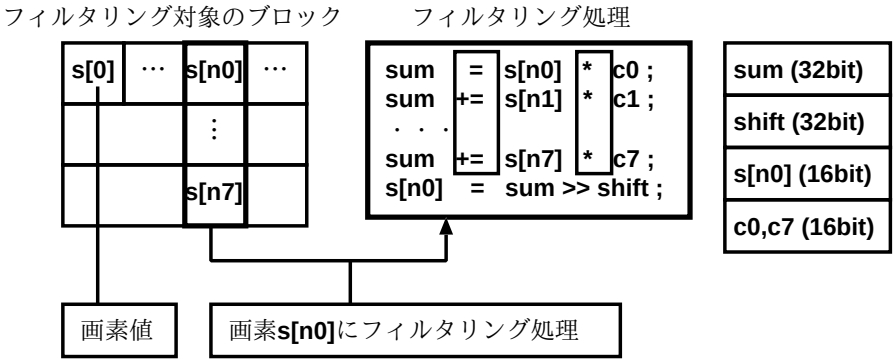

図 4.6: filter関数の主な処理

この図の例として画素s[n0]にフィルタリング処理を行っており，このような処理をブロック内の全画素に行う．フィルタリング処理の内，sumの値が16bitを超えるとき，その値によってshiftの値が変化し，sumが16bitに収まるように右シフトを行っている．shiftの値は0，6，12に変化する．

SIMD 処理として，フィルタリング処理の加算，乗算部分を pmaddwd 命令を使用して実装する．pmaddwd 命令の動作を図 4.7 に示す.

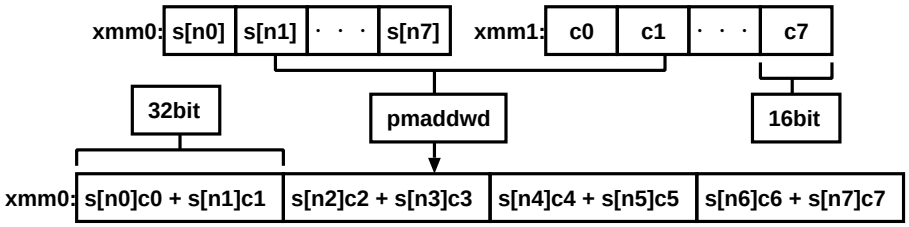

図 4.7: pmaddwd 命令の動作

動作として，レジスタ間で乗算を行い、その値を水平に加算し，その加算値を 32bit に拡張した領域に格納している．よって 1 命令で 8 回の乗算と 4 回の加算を同時に演算処理することができる.

しかし図 4.6 より，フィルタリング処理に使用する s[n0] から s[n7] の要素のように画素 s を垂直にアクセスされている．SIMD 命令には垂直に読み込む命令がないため，転置処理が必要になる.

転置処理に使用した SIMD 命令の punpcklwd，punpckhwd 命令の動作を図 4.8 に示す.

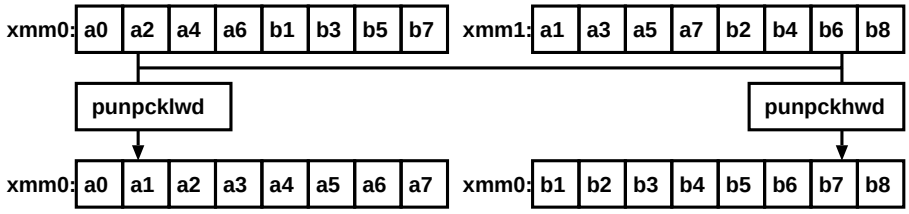

図 4.8: punpcklwd，punpckhwd 命令の動作

punpcklwd 命令の場合，レジスタの下位 64bit 同士で並び替えを行う．図の例として，xmm0 の下位 64bit は下位から順に a0，a2，a4，a6 で，そして xmm1 は a1，a3，a5，a7 とする．その要素を図のように順番に格納していく．また punpckhwd 命令はレジスタの上位 64bit 同士を対象としている．

そして punpcklwd，punpckhwd 命令を使用した転置処理の動作を図 4.9 に示す．

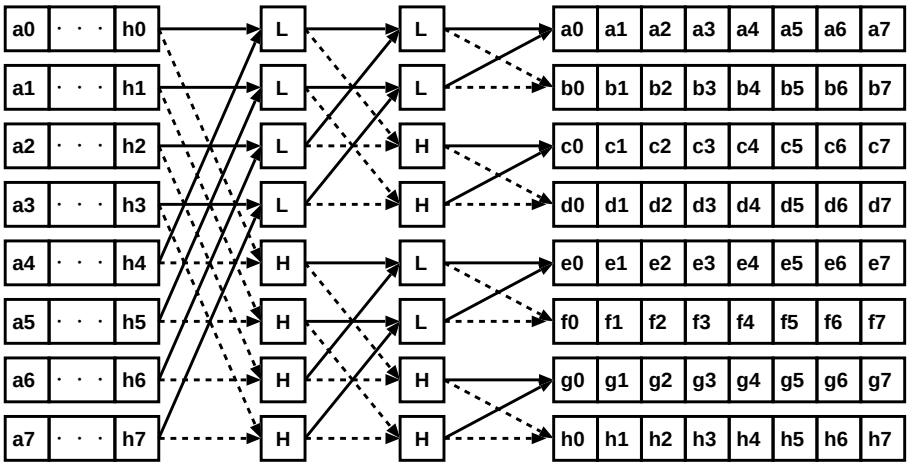

図 4.9: punpckhwd，punpcklwd 命令を使用した転置処理

実線が punpcklwd 命令 (L)，点線が punpckhwd 命令 (H) を示す．a から h は xmm レジスタに格納された要素であり a，b，c，d，e，f，g，h の順で格納されている．図のように転置したレジスタの要素を先ほどの pmaddwd 命令のような SIMD 演算に使用している．

このように転置処理を使用した SIMD 処理でも高速化は可能であるが，転置処理を使用しない SIMD 処理の手法も考案し，実装した．

4.2.2 転置処理を使用しない実装

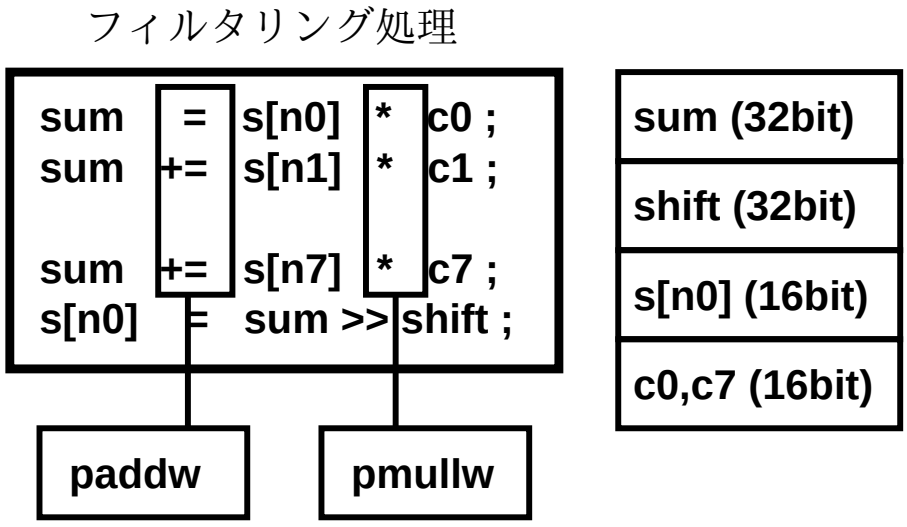

図 4.10: 実装 2 の場合の filter 関数の加算，乗算部分の SIMD 化

図 4.10 のように，SIMD 命令の paddw(16bit 単位の加算)，pmullw(16bit 単位の乗算) 命令を使用して SIMD 処理を実装する．その手法として，図 4.11 のように 8 画素ずつ同時にフィルタリング処理を行う．

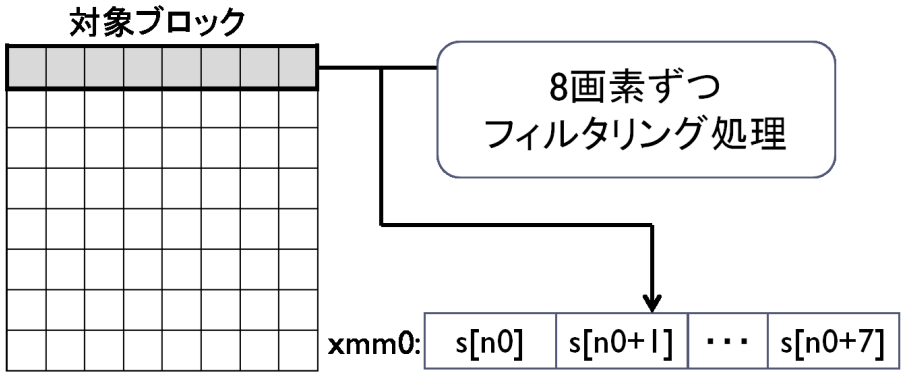

図 4.11: 8 画素ずつのフィルタリング処理

そのため，その水平の 8 個の画素値を xmm0 レジスタに読み込む.

図 4.12，4.13，4.14 に以降の一連の SIMD 処理の動作を示す.

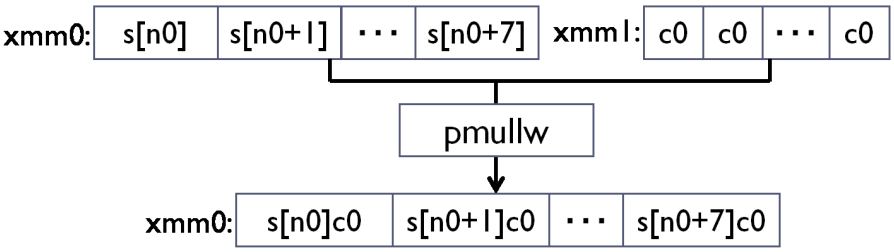

図 4.12: pmullw 命令を使用した 1 ラインの乗算処理

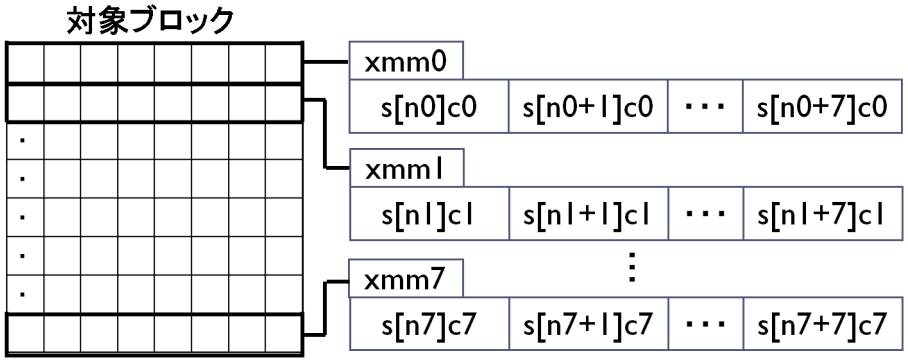

図 4.13: 図 4.12 の 8 ライン分の演算処理

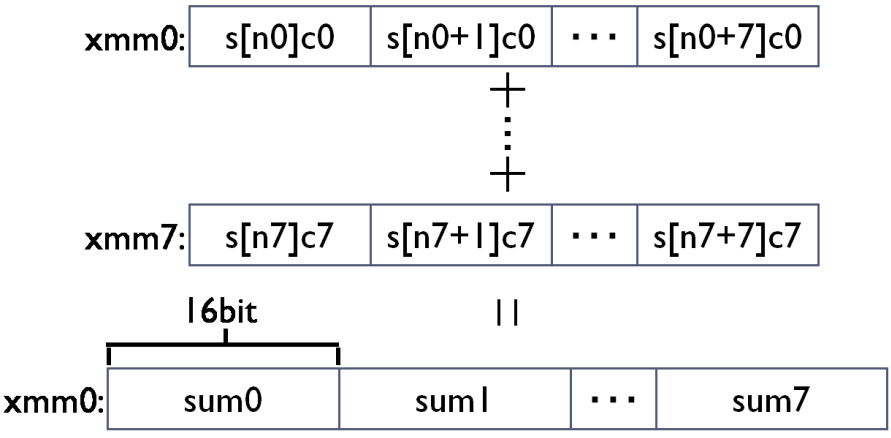

図 4.14: 8 画素を対象とした SIMD 処理

図の通り，フィルタリング処理の乗算部分を pmullw 命令を使用した乗算処理 (図 4.12) で縦方向に 8 ライン分行い (図 4.13)，その後，加算部分を先ほどの乗算値を含んだレジスタを paddw 命令を使用した加算処理で総和を算出する (図 4.14)．結果として，その総和は 8 画素分の sum となる．しかしこの sum は 16bit であり，実際のフィルタリング処理の sum は 32bit であるため，この SIMD 処理の場合，オーバーフローが発生し，圧縮性能が変化してしまう危険がある．そのため，転置処理を使用する実装と併用し，shift=0 の sum が 16bit に収まる時のみ，この手法で処理するように実装した．

4.3 xGetSAD8 関数

4.3.1 SAD 演算

SAD(Sum of Absolute Difference) とは差分絶対値和のことであり，下記の式で定義される．

$$SAD = \sum_{x,y} |Diff(x,y)| \tag{2}$$

式中の Diff(x,y) は，座標 (x,y) における符号化対象信号と予測画像信号との誤差 (予測歪) を示している．SAD は符号化歪の算出に必要な直交変換処理や量子化処理，乗算処理を必要としていない為，符号化歪を性格に

表現することは出来ないが，動き探索などの計算頻度の高い歪評価時に，演算付加を低減させる目的で使用する．

4.3.2 実装

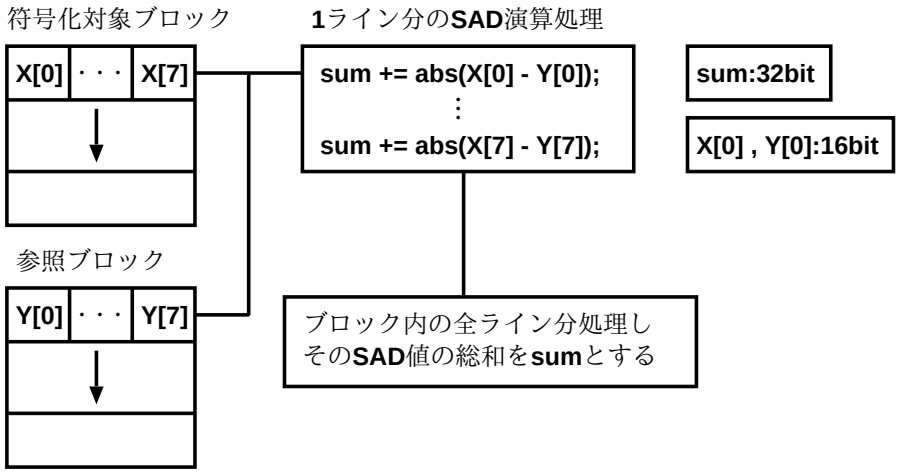

図 4.15: SAD 演算のコード

xGetSAD8 関数の SAD 演算処理の動作を図 4.15 に示す．符号化対象画像ブロックと参照画像ブロックの SAD 演算を 1 ライン (8 画素) ずつ行っており，この処理をブロック内の全ライン分行う．そして全処理が終了したときの sum がそのブロックの SAD 値となる．この 1 ライン分の SAD 演算処理を SIMD 命令で実装する．その実装部分を図 4.16 に示す．

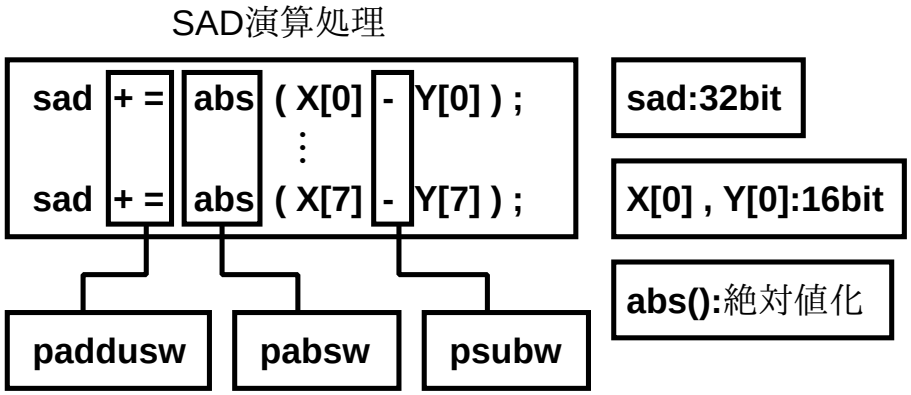

図 4.16: SAD 演算の SIMD 処理

このように，加算，絶対値化，減算部分をそれぞれ paddusw(符号無し 16bit 加算)，pabsw(16bit 絶対値化)，psubw(16bit 減算) 命令を使用して SIMD 処理を実装した．これと同様の SIMD 処理を xGetSAD16，32 関数にも実装した.

4.4 xCalcHADs8x8 関数

4.4.1 SATD 演算

SAD に対して，歪みの評価を高精度にした SATD(Hadamard transformed SAD) と呼ばれるアダマール変換絶対値誤差和を求める関数である．その式は下記に示す.

$$SATD = (\sum_{x,y} |DiffT(x,y)|)/2 \tag{3}$$

DiffT(x,y) は予測歪み Diff(x,y) をアダマール変換したものを示す．その 8x8 ブロックを対象とする関数他にも，2x2，4x4，4x16，16x4 ブロックを対象とする関数もある．その中でも処理時間が長いこの関数に SIMD 処理を実装する．

4.4.2 実装

xCalcHADs8x8 関数内の処理である 8x8 ブロックサイズの 1 ライン分のアダマール変換処理を図 4.17 に示す.

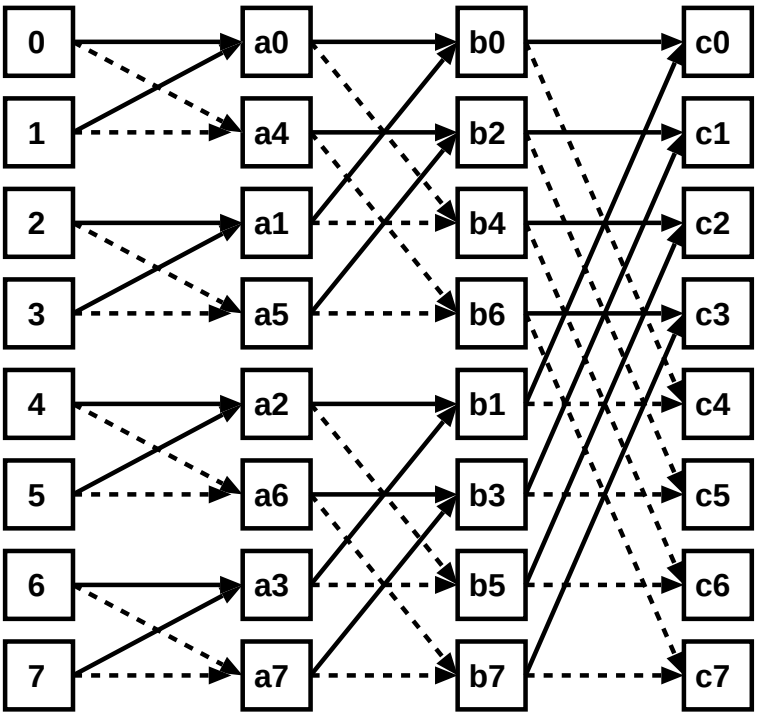

図 4.17: アダマール変換の 1 ライン分の処理

0 から 7 は 1 ライン分である 8 個の要素である．実線は 0+1 のような加算処理，点線は 0-1 のような減算処理を示す．途中の a などの記号は後述の SIMD 処理の説明に使用する段階である．c まで進むことで 1 ライン分の処理が終了する．この 1 ライン分の処理を 8x8 ブロック内で，水平方向の全ラインに処理を行い，その後垂直方向の全ラインに同様の処理を行う．この図 4.17 の処理を SIMD 命令で実装する．

使用した SIMD 命令の phaddw 命令の動作を図 4.18 に示す．

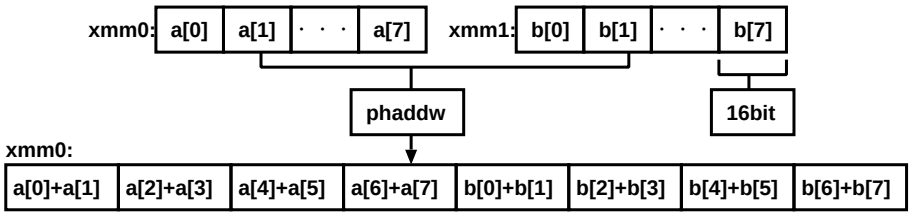

図 4.18: phaddw 命令の動作

動作として二つのレジスタでそれぞれ水平に加算を行い，その結果を繋げて一つのレジスタに格納する処理である．

そして図 4.17 に実装した SIMD 処理の動作を図 4.18 に示す．

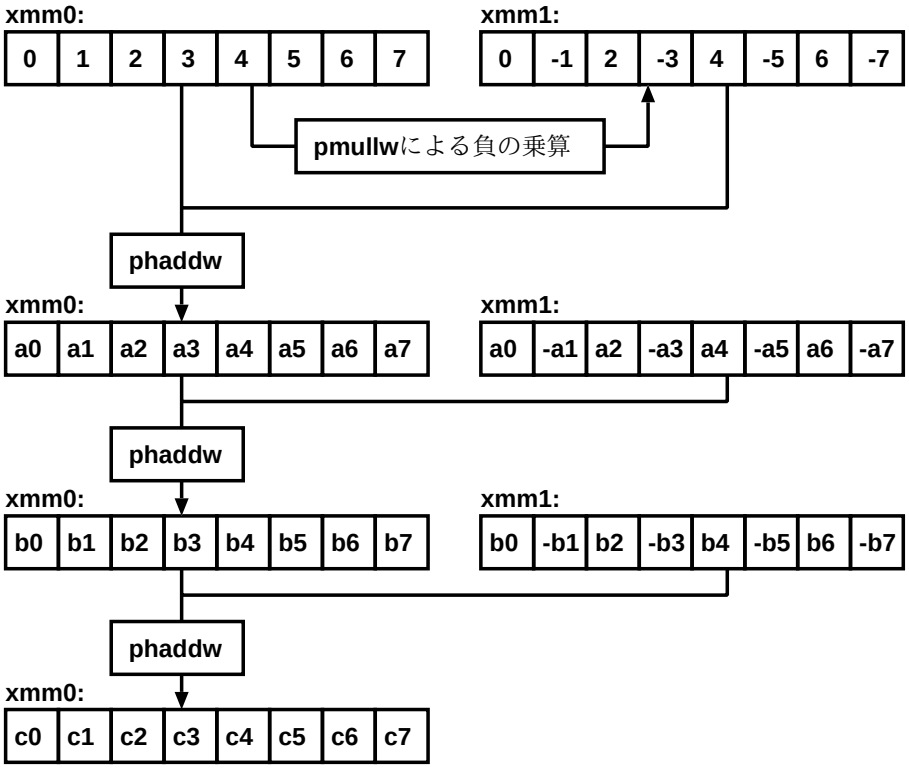

図 4.19: 図 4.17 の SIMD 処理

それぞれの演算に使用している xmm1 レジスタは，phaddw 命令で減算処理を行うために，その隣の xmm0 レジスタに pmullw 命令を使用して負の乗算を行ったものである．図 4.17 のそれぞれの段階の a，b，c と，図 4.19 のそれぞれの段階の xmm0 レジスタと照らし合わせると同様の演算処理を行っていることがわかる．

5　性能評価

5.1　評価結果

HM を使用して HD 画質の動画 (動画名:speed_bag_1080p.yuv) を 30 フレームでエンコードし，その SIMD 命令実装前と実装後の実行時間を表 5.3 に示す.

表 5.3: 並列化後の関数ごとの実行時間 (30 フレーム)

	実装前 (s)	SIMD 命令実装後 (s)	高速化率 (%)
全体の実行時間	2421.9	1601.2	33.9
filter	776.4	258.0	66.8
xCalcHADs8x8	369.9	179.5	51.5
xGetSAD8	65.4	38.8	40.6
xGetSAD16	58.0	20.3	64.9
xGetSAD32	45.4	9.3	79.5

5.2　考察

評価結果より，filter 関数で転置処理など追加の処理を加えたものの SIMD 命令実装により全体の実行時間が約 33%高速化がされた．また filter 関数で転置処理を使用しない並列処理を実装したことで，実装していない場合と比べて約 40 秒程の高速化を実現している．しかしその並列処理の使用の機会が少ないため，そのような小さい結果となったと考えられる．よって，その並列処理の使用可能な条件をさらに調べて，なるべく

その処理で実装することにより，さらなる高速化が期待できる．そしてxGetSAD8，32 関数の高速化率の違いは，xGetSAD32 の場合，xGetSAD8の 4 倍の処理量となり，その分 SIMD 命令の実装が増加し，それに伴って処理量が減少したことが考えられる．PSNR とビットレートに変化は無く，HM を圧縮性能不変で高速化が実現できている．その結果として，従来の HM を 2 回実行する間に，SIMD 命令で実装した HM の場合 3 回程実行出来るようになったことで，試行時間が短縮し，試行回数を増加させることが出来る．そのため，本研究で SIMD 命令を実装した HM を使用することで，研究効率の向上が期待できる．

6　あとがき

本研究では SIMD 命令による HM の高速化を行った．Visual Studio のプロファイル機能を利用して抽出した処理量に多い関数を SIMD 命令を利用することで極力並列化することで，全体の約 33%の高速化を実現した．しかし，この高速化率は可能な SIMD 命令が SSE まで，使用可能なxmm レジスタが 8 個と制約のある環境で実装したため，要求を十分満たせてはいない．今後は，AVX の使用に加え，x86 プロセッサを 64 ビットモードで動作させることで，使用可能な xmm レジスタを 16 個に増加さ

せたり，マルチスレッドによる高速化を施したりすることで，さらなる高速化をはかる必要がある．

謝辞

本研究を進めるにあたり，日頃から様々なご指導，ご助言を頂きました近藤利夫教授，佐々木敬泰助教に感謝いたします．また様々なご指摘を下さいました計算機アーキテクチャ研究室の方々に感謝の意を表します．

参考文献

[1] 大久保榮ほか『H.265/HEVC 教科書』，インプレス，2013.

A　Visual Stdioのプロファイル機能の使用法

下記 URL を参照

pgomgr を使ったパフォーマンス解析

http://d.hatena.ne.jp/Crest/20120108/1326049212