卒業論文

題目

補間画素生成に対応するロード命令の提案

指導教員

近藤　利夫教授

2018年

三重大学 工学部 情報工学科
コンピュータアーキテクチャ研究室

長戸　翔平(414848)

内容梗概

画素数が，現在主流のフルハイビジョン映像の 4 倍の 4K 映像，更にその 4 倍の 8K 映像へと動画像の高精細化が進み，4K テレビが普及し始める中，2016 年 8 月には 8K スーパーハイビジョンの試験放送が開始されている．このような，動画像の高精細化の進展に伴い，動画像データ量の増大と符号化に要する演算量の増加への対応が迫られ，H.265/HEVC[1] が動画像符号化の国際規格として 2013 年に標準化された．

この H.265/HEVC は，圧縮率をこれまで最高であった H.264/AVC の約 2 倍にまで高めているものの，符号化に必要となる演算量が大幅に増加し，符号化時間の増加を招いている．符号化に必要な総演算量のうち，小数画素精度での動き補償に必要な演算量が約 40%を占め，ボトルネックとなっている [2]．さらに，この小数画素精度での動き補償の中で，補間画素生成が半分程度の演算量を占め，H.265/HEVC の参照ソフトウェアである HM(HEVC Test Model)[3] では，1/2 および 1/4 位置の補間画素を生成する補間画素生成クラス TComInterpolationFilter はエンコード時間の約 20%を占めている [4]．このエンコード時間増の大きな要因は，水平方向に並ぶフィルタ対象画素に対して読み出しと演算を高並列実行可能であるのに対し，垂直方向の対象画素に対しては，メモリからデータを一度に読み出せず，複数回に分けてロードする必要があるために低並列での実行にならざるを得ないのに加え，画素とフィルタ係数との積和演算を並列化するためにデータの並び替えが別途必要になることにある．

そこで本研究では，画素の読み出しとデータの並び替えを高効率に実行する補間画素生成対応の新しいロード命令を提案する．そしてハードウェア追加面積と消費電力増加率を 1%未満に抑えつつ，SIMD に対応した拡張命令セット SSE と AVX2 と比較して最大で 30%のサイクル数削減が実現可能であることを示す．

Abstract

In recent years, the pixels of image has been improved to 4K video, which is four times to that of full high vision video, Moreover, in the same period, 8K full high vision video which 4 times to 4K and moving pictures with high definition is progressing eg.. In August 2016 8K Super Hi-Vision trial broadcasting started while 4K television begins to spread. With the progress of higher definition of moving images, H.265/HEVC[1] is standardized in 2013 to solve the problem on an increase in the amount of moving image data and an increase in the amount of computation required for coding.

Although the compression rate of H.265/HEVC has been improved to about twice as high as that of H.264/AVC, which was the highest so far, the amount of computation required for encoding and coding time largely increased. It becomes a bottleneck that for motion compensation with decimal pixel precision occupies about 40% among the total computation required for encoding[2]. Moreover, interpolation pixel generation occupies about half of the computation amount among the motion compensation with subpixel precision. And, in HM(HEVE Test Model)[3] which is the reference software of H.265/HEVC, interpolated pixel generation class TComInterpolation Filter that generates interpolated pixels at 1/2 and 1/4 positions occupies about 20% of the encoding time[4].The major reason for this increase in the encoding time is concluded: reading and calculation can be executed in parallel for filter target pixels aligned in the horizontal direction, for vertical target pixels, since sometimes data cannot be read out from the memory in parallel, we have to execute the operation in a low parallel manner due to the necessary for loading the pixels in a plurality of times, and furthermore, in order to parallelize the product-sum operation of the pixel and the filter coefficient, other ways for rearranging the data is necessary.

In this research, we propose a new load instruction for interpolation pixel generation which performs pixel readout and data rearrangement with high efficiency. And it shows that it is possible to reduce the number of cycles by up to 30% through comparing to the extended instruction set SSE and AVX2 compatible with SIMD, while keeping the added area of hardware and the power consumption increase rate less than 1%.

目 次

図 目 次

表 目 次

1　まえがき

1.1　研究背景

現在主流となっている 1920x1080 画素のフルハイビジョン映像の 4 倍の画素数を持つ 4K 映像，そして 4K 映像の更に 4 倍の画素数を持つ 8K 映像に対応する技術開発が進められるなか，4K 対応テレビの流通や NHK が主導する 8K スーパーハイビジョンの試験放送が 2016 年 8 月に開始され，動画像の高精細化が進んでいる．その高精細化に伴う動画像データ量の増大と符号化に要する演算量の増加への対応が迫られている．

1.2　研究目的

高精細化によるデータ量の増大が進む動画像に対応するため，動画像符号化の国際標準規格 H.264/AVC の後継となる H.265/HEVC[1] が 2013 年に標準規格化された．H.265/HEVC は H.264/AVC と比較して圧縮率を約 2 倍にまで高めたものの，符号化に必要となる演算量が大幅に増加し，符号化速度の低下と符号加時間の増加を招いている．符号化に必要な総演算量のうち，フレーム間予測における小数画素精度での動き補償に必要な演算量が約 40%を占め，ボトルネックとなっている [2]．さらに，この小数画素精度での動き補償の中で，補間画素生成が半分程度の演算量

を占めている．H.265/HEVC の参照ソフトウェアである HM(HEVC Test Model)[3] では，その対応ルーティンであり，整数画素間の 1/2 および 1/4 位置の補間画素を生成する補間画素生成クラス TComInterpolationFilter は総エンコード時間の約 20%を占めている [4].

このエンコード時間増大の大きな要因は，水平方向に並ぶフィルタリング対象画素に対しては読み出しと積和演算を高並列実行可能であるのに対し，垂直方向の対象画素に対しては，低並列での実行にならざるを得ないのに加え，画素とフィルタ係数との積和演算を並列化するためにデータの並び替えが別途必要になることにある．

そこで本研究では，画素の読み出しとデータの並び替えを高効率に実行する補間画素生成対応の新しいロード命令を提案する．そしてハードウェア追加面積を 1%未満に抑えつつ，SIMD に対応した拡張命令セット SSE と AVX2 と比較して最大で 30%のサイクル数削減が実現可能であることを示す.

2　H.265/HEVC について

2.1　H.265/HEVC

現在に至るまで様々な動画像符号化の規格が策定されている．H.265/HEVC は 2013 年に ITU-T の VCEG(Video Coding Experts Group) と ISO/IEC の MPEG(Moving Picture Experts Group) との合同組織である JCT-VC(Joint Collaborative Team on Video Coding) によって策定された動画像符号化の国際標準規格であり，既存の規格である H.264/AVC と比較して約 2 倍の圧縮性能を実現した．しかし，圧縮率を高めるために符号化に必要な演算量が増大し，符号化速度の低下と符号化時間の増大を招いている．符号化に必要な演算量のうち，小数画素精度での動き補償に必要な演算が約 40%を占めており，ボトルネックとなっている [2].

2.2　インター (画面間) 予測

インター予測は，符号化を行うある任意のフレームとその前方，後方，あるいはその両方のフレームの画素情報を利用して動画像を圧縮する方法である．任意のフレームとその次のフレームをすべて符号化するのではなく，任意のフレームとその前方もしくは後方のフレームとの差分を生成すると，任意のフレームと生成した差分だけを符号化するだけで圧

縮した画像の再構成が可能となる．この差分を単純フレーム間予測誤差という．HEVC/H.265 ではインター予測が符号化処理量のおよそ 70%を占めている [5].

2.3 動き補償技術

動画像が切り替わる等の理由で前後のフレームで大きな動きがあった際には二つのフレーム間に類似性がなくなり，2.2 節で述べた単純なフレーム間予測だけを用いて予測誤差を生成して符号化を行うと，符号化前よりもかえって情報量が増えてしまう場合がある．そこで，フレーム内の被写体がどれだけ動いたかの情報 (動きベクトル) と予測誤差を併用することで符号化が非常に効率的になる．この動きベクトルを用いたフレーム間予測を動き補償フレーム間予測という．動き補償フレーム間予測は動きベクトルを探し求める処理に多くの演算を必要とするため，1980 年代初頭頃に実用的になったが，この技術によって動画像の圧縮効率は更に高まった.

動き補償を用いてフレーム間予測を行うためには，画像の動き量を推定する動きベクトル探索が必要になる．1990 年に策定された H.261 では整数画素精度で動きを推定し動き補償を行なっていたが，1993 年に策定

された MPEG-1 からは 1/2 画素精度での動き補償が可能となった．これは，物体の動きがちょうど整数画素単位に収まる確率が高くなく，小数画素精度で動きベクトルを求めることでさらに効率的な動き補償が可能となるためである．H.264/AVC ではさらに細かい 1/4 画素精度での動き補償が可能となり，H.265/HEVC でも 1/4 画素精度での動き補償が採用されている．

2.4 補間画素生成 [2]

小数画素精度で動きベクトルを求める際には整数位置の画素間の小数位置の画素 (小数画素) が必要になる．しかし，元々の動画像には小数位置の画素は存在しない．そのため，小数画素を補間生成する必要がある．

補間画素生成のプロセスを図 2.1 を用いて説明する．符号化対象フレームのある整数画素に着目し，まず水平方向に着目画素を含めた 4 または 8 画素分の整数画素のデータ (図 2.1 中の灰色で塗りつぶされた四角形) と予め用意されている FIR タップフィルタを用いて積和演算 (式 (1)) を行い水平方向の補間画素 (図中の星が入っている四角形) を生成する．そして整数画素と生成された小数画素，タップフィルタを用いて積和演算 (式 (2)〜(5)) を垂直方向に実行して，残りの小数画素 (図中の白色の四角形)

を生成する．

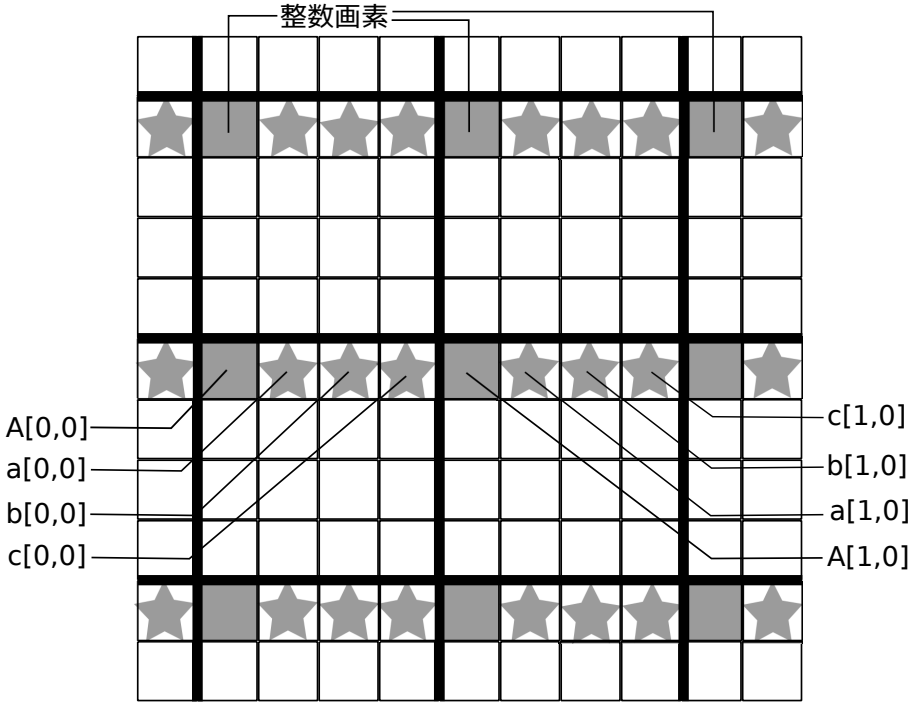

図 2.1: 補間画素の生成プロセス

整数位置の画素データを A，補間生成された $1/4$，$1/2$，$3/4$ 位置の画素をそれぞれ a,b,c，タップフィルタを f で表し，動画像フレームの左上を始点，着目画素の座標を [0,0] とすると，補間画素の生成プロセスは以下の 5 式で表される．このとき，shift は動画像のビット深度 (BitDepth) を用いて $shift = BitDepth - 8$ によって定められる．また，タップフィルタの詳細を表 2.1 に示す．

$$pic\_horiz = \sum_{k=-3}^{4} (A[k,0] \times f[k]) \div 2^{shift} \tag{1}$$

$$pic\_vert_A = \sum_{k=-3}^{4} (A[0,k] \times f[k]) \div 2^{shift} \tag{2}$$

$$pic\_vert_a = \sum_{k=-3}^{4} (a[0,k] \times f[k]) \div 2^{6} \tag{3}$$

$$pic\_vert_b = \sum_{k=-3}^{4} (b[0,k] \times f[k]) \div 2^{6} \tag{4}$$

$$pic\_vert_c = \sum_{k=-3}^{4} (c[0,k] \times f[k]) \div 2^{6} \tag{5}$$

表 2.1: タップフィルタ

入力画素インデックス	-3	-2	-1	0	1	2	3	4
1/4 画素	-1	4	-10	58	17	-5	1	N/A
1/2 画素	-1	4	-11	40	40	-11	4	1
3/4 画素	N/A	1	-5	17	58	-10	4	-1

3　提案手法

3.1　従来法

従来法では，大量のデータに対してデータ読み出し･積和演算･加算等の同一処理を繰り返している．よって，逐次的に演算を行うよりも単一の命令で単一のデータに演算を適用させた方が効率的である．そこで，単一の命令列で複数のデータに演算が可能な SIMD(Single Instruction Multiple Data) という処理を用いて並列に演算を実行している．

3.2　従来法の問題点

動画像に限らず従来のメモリに格納されている二次元データは行方向 (ラスタ走査順) には必要なデータを高速かつ並列に読み出すことが可能であるのに対し，列方向 (非ラスタ走査順) のデータ読み出しでは高速かつ並列に読み出すことは不可能であり，符号化高速化のボトルネックとなっている．

補間画素の生成においても，高速にメモリアクセスできる方向の制約によって垂直方向に並ぶデータを一回のロード命令で読み出すことは不可能であり，水平方向のデータ読み出しを複数回に分割して演算対象データを読み出さざるを得ないため，演算に必要なサイクル数が増加する．

また，メモリからデータを読み出した後に演算に適した並びになるよう画素の並び替えを行う必要があり，それによってもサイクル数が増加する．実際，H.265/HEVC ビデオエンコーダアプリケーションライブラリである x265[6] では，8 タップのフィルタ係数との積和演算を行う際に，図 3.2 に示すようにメモリから読み出したデータをレジスタに格納した後に 16 ビットごとに区切って並び替える処理を行なっている．そのオーバーヘッドは，付録 A.1 と付録 A.3 に示したように，SSE において発行命令数の約 20%を，AVX2 において発行命令数の約%35 を占めている．

3.3 提案手法

本研究の提案手法では，メモリからデータを読み出してレジスタに格納する前に 16 ビットごとにあらかじめ並び替えを行うことで既存の並び替え命令を削除し，補間画素生成に必要な演算サイクルを削減する．SSE におけるレジスタに格納されたデータの並び替えの一例を図 3.2 に示す．

また，当研究室ではライン (行方向)・タイル (列方向) 両アクセスに対応したキャッシュメモリ [7] が提案されており，ラインモードではラスタ走査順に 512 ビット，タイルモードでは 64 ビット x8 行を並列に読み出すことができる．このキャッシュメモリを搭載したプロセッサを想定し，必

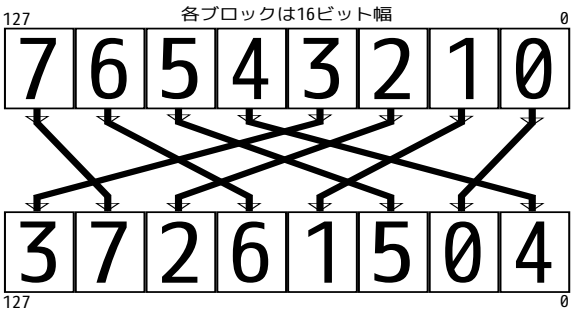

図 3.2: SSE におけるデータ並び替えの
模式図

要なデータが一度に並列に読み出せることを前提に，補間画素生成に対応した新規ロード命令を考案し，符号化に必要な命令数とサイクル数を削減する．

新しく考案したロード命令では，演算に必要なデータの並列読み出しと読み出したデータに対する 16 ビットごとの並び替えを同時に行うことでサイクル数の低減をはかる．データ並び替えのためのモジュールの模式図を図 3.3 に示す。

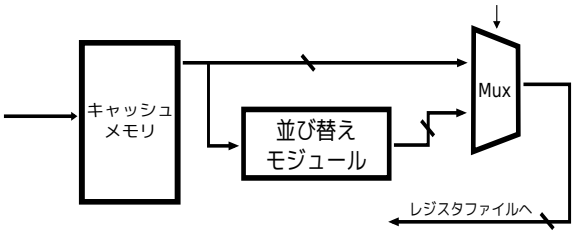

図 3.3: データの並び替えモジュールの
模式図

このデータ並び替えのための新規モジュールはデータに対して演算を実行せず，バス線と選択器 (マルチプレクサ) のみで構成することが可能であり，追加前のハードウェア全体の面積と比較しても面積増加率は 1%未満とわずかなオーバーヘッドで済むと推測される．

さらに，ハードウェア全体の消費電力と比較してもマルチプレクサ数個の消費電力は無視できるほど小さく，消費電力の増加率も 1%未満に抑えられると推測される．

3.4 提案命令セット

本研究では，Intel が開発した SIMD の拡張命令セットである SSE と AVX2 それぞれに提案する新規ロード命令を組み込むことを想定する．A.2 小々節と A.4 小々節にそれぞれ提案ロード命令の仕様を記す．

3.4.1 readunpck 命令

readunpck xmm1, r64/m64

readunpck 命令は，メモリの r64/m64 で指定した番地からタイル単位でデータを 128 ビット分だけ読み出し，並び替えモジュールを通して並び替えを実行した後に xmm1 に格納する命令である．データの並び替えを 26 ページの付録 B.1 に示す．

3.4.2 readunpck256 命令

readunpck256 ymm1, r64/m64

readunpck256 命令は，メモリの r64/m64 で指定した番地からタイル単位でデータを 256 ビット分だけ読み出し，並び替えモジュールを通して並び替えを実行した後に ymm1 に格納する命令である．データの並び替えを 26 ページの付録 B.2 に示す．

3.4.3 スループットとレイテンシ

提案命令は，従来法で使われている movq 命令と比較して機能が類似しているため，レイテンシとスループットは同程度であると見做した．表 3.2 に提案ロード命令のレイテンシとスループットを示す．

表 3.2: 提案ロード命令のレイテンシとスループット

命令	レイテンシ (Cycle)	スループット (Cycle)
movq(既存命令)	1.0	0.33
readunpck	1.0	0.33
readunpck256	1.0	0.33

4　性能評価

4.1　評価

提案命令セットの有用性を評価するため，SSE と AVX2 が動作するデータパスのブロック構成を明らかにし，x265 内の 4x4 画素に対する補間画素を生成する演算マクロの実行に要する発行命令数，所要サイクル数および二つの削減率を算出し評価を行う．また，今回の命令セットの演算ステージ数は演算マクロ内の命令の中でレイテンシが一番長い pmaddwd 命令に合わせて 5 段と設定した．演算マクロに使用される命令のレイテンシとスループットを表 4.3 に示す．

従来法である SSE，AVX2 と提案手法を組み込んだアセンブリプログラムは 19 ページの付録 A にそれぞれ示してある．

表 4.3: 従来法で使用される命令のレイテンシとスループット

命令	レイテンシ (Cycle)	スループット (Cycle)
[SSE，AVX2 共通]		
movq	1.0	0.33
punpcklwd	1.0	1.0
pmaddwd	5.0	0.5
lea	3.0	1.0
paddd	1.0	0.33
[AVX2 のみ]		
vinserti128	3.0	1.0

表 4.4 と表 4.5 にそれぞれ SSE，AVX2 とそれぞれに提案手法を組み込んだ場合を比較した発行命令数，所要サイクル数および削減率を示す．

表 4.4: 従来法 (SSE) と提案手法との比較

	SSE	SSE+提案手法	削減率
発行命令数	53	42	20.75%
所要サイクル数	30	27	10%

表 4.5: 従来法 (AVX2) と提案手法との比較

	AVX2	AVX2+提案手法	削減率
発行命令数	42	20	52.38%
所要サイクル数	27	16	30.43%

表 4.4 と表 4.5 に示したように，提案手法を組み込んだ命令セットによって発行命令数を SSE 比で 20.75%，AVX2 比で 52.38%，所要ステップ数を SSE 比で 10%，AVX2 比で 30.43%削減することができる．

また，3.3 小節で述べたように，追加した並び替えモジュールはハードウェア全体と比較しても面積・消費電力共に充分に小さく，ハードウェアの面積増加率は 1%未満，消費電力の増加率も 1%未満であると推測される．

4.2 考察

付録 A に示したプログラムを見ると，付録の A.1 と A.2 を比較してもわかるようにデータのロード命令 (movq 命令) と並び替え命令 (punpcklwd 命令) の発行数が従来法よりも削減されていることがわかる．そして，SSE よりも AVX2 の方が発行命令数の削減率が大きいのは，AVX2 は SSE と比較してデータを読み出して並び替えるステップに多くの命令が必要となっており，提案手法によってその命令を削減できたことが考えられる．

また，命令数の削減率の割にサイクル数の削減率が高くないのは，表 4.3 からも分かるように補間画素生成に必要な命令の中で一番サイクル数が掛かる pmaddwd 命令がボトルネックとなり，データのロードが高速に完了したとしても pmaddwd 命令の完了を待たなければならず，その結果命令数削減率に見合ったサイクル数削減率が達成できなかったと推測される．

5　あとがき

本研究では，動画像符号化における補間画素生成に対応する新規ロード命令とデータ並び替えモジュールを提案し，ハードウェアの追加面積および消費電力増加量を 1%未満に抑えつつ演算の実行に必要な命令の発行数とサイクル数を低減し，従来法である SSE や AVX2 と比較して提案手法の有用性を示した．

今後の課題としては，実際にデータパスに提案命令と並び替えモジュールを実装してサイクル数を厳密に測定すること，並び替えモジュールにおいてパラメータ指定による並び替えの可変化に対応させること，既存命令よりもステップ数を削減した積和演算の考案などが挙げられる．

謝辞

本研究を行うにあたって，常日頃から様々なご指導，アドバイスをいただきました近藤 利夫教授，佐々木 敬泰助教，深澤 祐樹技術員に感謝いたします．また，研究室での共同生活等様々な面でお世話になりましたコンピュータアーキテクチャ研究室の皆さまに感謝いたします．最後に，ここまで育ててくれた家族に最大限の感謝の意を表します．

参考文献

[1] 大久保榮，鈴木輝彦，高村誠之，中條健，H.265/HEVC教科書，ISBN-978-4-8443-3468-2，2013年10月

[2] Vladimir Afonso et al.，Hardware Implementation for the HEVC Fractional Motion Estimation Targeting Real-Time and Low-Energy，Journal of Integrated Circuits and Systems 2016; v.11 / n.2:106-120

[3] HM(HEVC Test Model)，https://hevc.hhi.fraunhofer.de/HM-doc/

[4] Chang-Uk JEONG，Computational Complexity Reduction for Motion Estimation in Video Compression，Waseda University，2015

[5] Alex Lee et al.，Efficient Inter Prediction Mode Decision Method for Fast Motion Estimation in High Efficiency Video Coding，ETRI Journal，Volume 36，Number 4，August 2014，http://dx.doi.org/10.4218/etrij.14.0113.0087

[6] x265 HEVC Encoder/H.265 Video Codec，http://x265.org

[7] B.K.Wang et al., Tile/Line Access Cache Memory based on a Multi-level Z-order Tiling Data Layout. Concurrency and Computation: Practice & Experience. 2017; Wiley Press; e4375l;

A　プログラムリスト

A.1　従来法-SSE

以下の演算マクロ PROCESS_LUMA_VER_W4_4R は，4x4 画素のフィルタ対象ブロックにおいて SSE 命令セットを用いて垂直方向に対する演算を行うマクロである．

```
%macro PROCESS_LUMA_VER_W4_4R 0
    movq        m0, [r0]          //データのロード
    movq        m1, [r0 + r1]         //データのロード
    punpcklwd   m0, m1            //データ並び替え
    pmaddwd     m0, [r6 + 0 *16]      //積和演算

    lea         r0, [r0 + 2 * r1]     //アドレス生成
    movq        m4, [r0]          //データのロード
    punpcklwd   m1, m4            //データ並び替え
    pmaddwd     m1, [r6 + 0 *16]      //積和演算

    movq        m5, [r0 + r1]         //データのロード
    punpcklwd   m4, m5            //データ並び替え
    pmaddwd     m2, m4, [r6 + 0 *16] //積和演算
    pmaddwd     m4, [r6 + 1 * 16]     //積和演算
    paddd       m0, m4                //加算

    lea         r0, [r0 + 2 * r1]     //アドレス生成
    movq        m4, [r0]              //データのロード
    punpcklwd   m5, m4            //データ並び替え
    pmaddwd     m3, m5, [r6 + 0 *16] //積和演算
    pmaddwd     m5, [r6 + 1 * 16]     //積和演算
    paddd       m1, m5                //加算

    movq        m5, [r0 + r1]         //データのロード
    punpcklwd   m4, m5            //データ並び替え
```

```
pmaddwd    m6, m4, [r6 + 1 * 16]    //積和演算
paddd      m2, m6                   //加算
pmaddwd    m4, [r6 + 2 * 16]        //積和演算
paddd      m0, m4                   //加算

lea        r0, [r0 + 2 * r1]        //アドレス生成
movq       m4, [r0]              //データのロード
punpcklwd  m5, m4                //データ並び替え
pmaddwd    m6, m5, [r6 + 1 * 16]    //積和演算
paddd      m3, m6                   //加算
pmaddwd    m5, [r6 + 2 * 16]        //積和演算
paddd      m1, m5                //加算

movq       m5, [r0 + r1]            //データのロード
punpcklwd  m4, m5                //データ並び替え
pmaddwd    m6, m4, [r6 + 2 * 16]    //積和演算
paddd      m2, m6                   //加算
pmaddwd    m4, [r6 + 3 * 16]        //積和演算
paddd      m0, m4                //加算

lea        r0, [r0 + 2 * r1]        //アドレス生成
movq       m4, [r0]                 //データのロード
punpcklwd  m5, m4                //データ並び替え
pmaddwd    m6, m5, [r6 + 2 * 16]    //積和演算
paddd      m3, m6                   //加算
pmaddwd    m5, [r6 + 3 * 16]        //積和演算
paddd      m1, m5                   //加算

movq       m5, [r0 + r1]            //データのロード
punpcklwd  m4, m5                //データ並び替え
pmaddwd    m4, [r6 + 3 * 16]        //積和演算
paddd      m2, m4                   //加算

movq       m4, [r0 + 2 * r1]        //データのロード
punpcklwd  m5, m4                   //データ並び替え
```

```
    pmaddwd    m5, [r6 + 3 * 16]    //積和演算
    paddd      m3, m5               //加算
%endmacro
```

A.2 提案手法-SSE

以下の演算マクロ PROPOSED_LUMA_VER_W4_4R は，4x4 画素のフィルタ対象ブロックにおいて SSE に提案手法を組み込んだ命令セットを用いて垂直方向に対する演算を行うマクロである．

```
%macro PROPOSED_VER_LUMA_W4_4R 0
    readunpck   m0, [r0]         //データのロード&並び替え
    pmaddwd m0, [r6+0*16]        //積和演算

    readunpck   m1, [r0+r1]      //データのロード&並び替え
    pmaddwd m1, [r6+0*16]        //積和演算

    lea     r0, [r0+2*r1]        //アドレス生成
    readunpck   m4, [r0]         //データのロード&並び替え
    pmaddwd m2, m4, [r6+0*16]    //積和演算
    pmaddwd m4, [r6+1*16]        //積和演算
    paddd   m0, m4           //加算

    readunpck   m5, [r0+r1]      //データのロード&並び替え
    pmaddwd     m3, m5, [r6 + 0 *16]    //積和演算
    pmaddwd     m5, [r6 + 1 * 16]   //積和演算
    paddd       m1, m5               //加算

    lea     r0, [r0+2*r1]        //アドレス生成
    readunpck   m4, [r0]         //データのロード&並び替え
    pmaddwd     m6, m4, [r6 + 1 * 16]   //積和演算
    paddd       m2, m6               //加算
    pmaddwd     m4, [r6 + 2 * 16]   //積和演算
```

```
    paddd       m0, m4              //加算

    readunpck   m5, [r0+r1]     //データのロード&並び替え
    pmaddwd     m6, m5, [r6 + 1 * 16]   //積和演算
    paddd       m3, m6              //加算
    pmaddwd     m5, [r6 + 2 * 16]   //積和演算
    paddd       m1, m5          //加算

    lea     r0, [r0+2*r1]       //アドレス生成
    readunpck   m4, [r0]        //データのロード&並び替え
    pmaddwd     m6, m4, [r6 + 2 * 16]   //積和演算
    paddd       m2, m6              //加算
    pmaddwd     m4, [r6 + 3 * 16]   //積和演算
    paddd       m0, m4              //加算

    readunpck   m5, [r0+r1]     //データのロード&並び替え
    pmaddwd     m6, m5, [r6 + 2 * 16]   //積和演算
    paddd       m3, m6              //加算
    pmaddwd     m5, [r6 + 3 * 16]   //積和演算
    paddd       m1, m5          //加算

    lea     r0, [r0+2*r1]       //アドレス生成
    readunpck   m4, [r0]        //データのロード&並び替え
    pmaddwd     m4, [r6 + 3 * 16]   //積和演算
    paddd       m2, m4          //加算

    readunpck   m5, [r0+r1]     //データのロード&並び替え
    pmaddwd     m5, [r6 + 3 * 16]   //積和演算
    paddd       m3, m5              //加算
%endmacro
```

A.3　従来法-AVX2

以下の演算マクロ FILTER_VER_LUMA_AVX2_4x4 は，4x4 画素のフィルタ対象ブロックにおいて AVX2 命令セットを用いて垂直方向に対する演算を行うマクロである．

```
%macro FILTER_VER_LUMA_AVX2_4x4 1
    movq            xm0, [r0]           //データのロード
    movq            xm1, [r0 + r1]      //データのロード
    punpcklwd       xm0, xm1            //データ並び替え
    movq            xm2, [r0 + r1 * 2]      //データのロード
    punpcklwd       xm1, xm2                //データ並び替え
    vinserti128     m0, m0, xm1, 1      //データ並び替え
    pmaddwd         m0, [r5]            //積和演算

    movq            xm3, [r0 + r4]      //データのロード
    punpcklwd       xm2, xm3            //データ並び替え
    lea             r0, [r0 + 4 * r1]       //アドレス生成
    movq            xm4, [r0]           //データのロード
    punpcklwd       xm3, xm4            //データ並び替え
    vinserti128     m2, m2, xm3, 1      //データ並び替え
    pmaddwd         m5, m2, [r5 + 1 * mmsize]   //積和演算
    pmaddwd         m2, [r5]                //積和演算
    paddd           m0, m5          //加算

    movq            xm3, [r0 + r1]      //データのロード
    punpcklwd       xm4, xm3            //データ並び替え
    movq            xm1, [r0 + r1 * 2]      //データのロード
    punpcklwd       xm3, xm1                //データ並び替え
    vinserti128     m4, m4, xm3, 1      //データ並び替え
    pmaddwd         m5, m4, [r5 + 2 * mmsize]   //積和演算
    pmaddwd         m4, [r5 + 1 * mmsize]   //積和演算
    paddd           m0, m5              //加算
    paddd           m2, m4          //加算
```

```
    movq            xm3, [r0 + r4]      //データのロード
    punpcklwd       xm1, xm3            //データ並び替え
    lea             r0, [r0 + 4 * r1]       //アドレス生成
    movq            xm4, [r0]           //データのロード
    punpcklwd       xm3, xm4            //データ並び替え
    vinserti128     m1, m1, xm3, 1      //データ並び替え
    pmaddwd         m5, m1, [r5 + 3 * mmsize]   //積和演算
    pmaddwd         m1, [r5 + 2 * mmsize]   //積和演算
    paddd           m0, m5          //加算
    paddd           m2, m1          //加算

    movq            xm3, [r0 + r1]      //データのロード
    punpcklwd       xm4, xm3            //データ並び替え
    movq            xm1, [r0 + 2 * r1]      //データのロード
    punpcklwd       xm3, xm1            //データ並び替え
    vinserti128     m4, m4, xm3, 1      //データ並び替え
    pmaddwd         m4, [r5 + 3 * mmsize]   //積和演算
    paddd           m2, m4              //加算
%endmacro
```

A.4　提案手法-AVX2

以下の演算マクロ PROPOSED_VERT_LUMA_AVX2_version2_4x4 は，4x4 画素のフィルタ対象ブロックにおいて AVX2 に提案手法を組み込んだ命令セットを用いて垂直方向に対する演算を行うマクロである．

```
%macro PROPOSED_VERT_LUMA_AVX2_version2_4x4 1
    readunpck256    xm1, [r0]       //データのロード&並び替え
    pmaddwd         m0, [r5]            //積和演算

    readunpck256    xm1, [r0+2*r1]  //データのロード&並び替え
    pmaddwd         m5, m2, [r5 + 1 * mmsize]   //積和演算
    pmaddwd         m2, [r5]                //積和演算
    paddd           m0, m5          //加算

    lea         r0, [r0+4*r1]       //アドレス生成
    readunpck256    xm3, [r0]       //データのロード&並び替え
    pmaddwd         m5, m3, [r5 + 2 * mmsize]   //積和演算
    pmaddwd         m3, [r5 + 1 * mmsize]   //積和演算
    paddd           m0, m5          //加算
    paddd           m2, m3          //加算

    readunpck256    xm1, [r0+2*r1]  //データのロード&並び替え
    pmaddwd         m4, m1, [r5 + 3 * mmsize]   //積和演算
    pmaddwd         m1, [r5 + 2 * mmsize]   //積和演算
    paddd           m0, m4          //加算
    paddd           m2, m1          //加算

    readunpck256    xm3, [r0+4*r1]  //データのロード&並び替え
    pmaddwd         m5, [r5 + 3 * mmsize]   //積和演算
    paddd           m2, m5              //積和演算
%endmacro
```

B　提案ロード命令によるデータの並び替え

B.1　readunpck 命令

```
//
 dst[15:0] = src[79:64]
 dst[31:16] = src[15:0]
 dst[47:32] = src[95:80]
 dst[63:48] = src[31:16]
 dst[79:64] = src[111:96]
 dst[95:80] = src[47:32]
 dst[111:96] = src[127:111]
 dst[127:112] = src[63:48]
//
```

B.2　readunpck256 命令

```
//
 dst[15:0] = src[207:192]
 dst[31:16] = src[143:128]
 dst[47:32] = src[223:208]
 dst[63:48] = src[159:144]
 dst[79:64] = src[239:224]
 dst[95:80] = src[175:160]
 dst[111:96] = src[255:240]
 dst[127:112] = src[191:176]
 dst[143:128] = src[143:128]
 dst[159:144] = src[76:64]
 dst[175:160] = src[159:144]
 dst[191:176] = src[95:80]
 dst[207:192] = src[175:160]
 dst[223:208] = src[111:96]
 dst[239:224] = src[191:176]
 dst[255:240] = src[127:112]
//
```