Journal of Zhejiang University SCIENCE A
ISSN 1673-565X (Print); ISSN 1862-1775 (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

High throughput bandwidth optimized VLSI design for motion compensation in AVS HDTV decoder*

Kai LUO†, Dong-xiao LI†‡, Ming ZHANG
(*Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou 310027, China*)
†E-mail: luokai82@gmail.com; lidx@zju.edu.cn

Received Aug. 30, 2007; revision accepted Dec. 14, 2007; published online May 10, 2008

Abstract: In this paper we present a motion compensation (MC) design for the newest Audio Video coding Standard (AVS) of China. Because of compression-efficient techniques of variable block size (VBS) and sub-pixel interpolation, intensive pixel calculation and huge memory access are required. We propose a parallel serial filtering mixed luma interpolation data flow and a three-stage multiplication free chroma interpolation scheme. Compared to the conventional designs, the integrated architecture supports about 2.7 times filtering throughput. The proposed MC design utilizes Vertical Z processing order for reference data re-use and saves up to 30% memory bandwidth. The whole design requires 44.3k gates when synthesized at 108 MHz clock frequency using 0.18-μm CMOS technology and can support up to 1920×1088@30 fps AVS HDTV video decoding.

Key words: Audio Video coding Standard (AVS), Motion compensation (MC), Interpolation, VLSI, Architecture
doi:10.1631/jzus.A071460 **Document code:** A **CLC number:** TN919.8

INTRODUCTION

The Advanced Video coding Standard (AVS) (AVS Workgroup, 2004) is for compression and decompression in digital audio and video multimedia services. With its current two profiles and five levels, AVS can support digital video broadcasting, standard/high-definition television, mobility application and possibly 3D television in the future. The standard was approved and became the national standard in Dec. 2006.

AVS adopts a hybrid video coding framework and several advanced compression techniques (Fan *et al.*, 2004; Yu *et al.*, 2005), such as 2D variable length coding (2D-VLC), pre-scale integer transform (PIT), variable block size (VBS) and two steps four taps (TSFT) (Wang *et al.*, 2004) sub-pixel interpolation for temporal prediction. These advanced techniques improve the overall coding efficiency but also bring problems of intensive data calculation and huge memory access.

For the AVS Jizhun profile level 6.0, when decoding a 1920×1088 resolution 4:2:0 format 30 fps video with 64-bit memory bus under 108 MHz clock frequency, a maximum of 8160×30=244 800 macroblocks (MB) need to be processed in 1 s. Theoretically, the average time budget for compensating one MB is at most $108\times10^6/244\,800=441$ cycles, and the memory bandwidth reaches at least $244\,800\times1.5\times256\times8/2^{20}=661.16$ Mbits/s. In practice, for a pipelined AVS HDTV decoder, 441 cycles may include operations of motion vector (MV) prediction, reference data fetch and interpolation, all of which make the time budget tight. Jia *et al.*(2006) give the running analysis of (AVS Reference Software, 2007) and show that the motion compensation (MC) part can use up to 73% of the total decoding time. Therefore it is necessary to design a hardware accelerator for MC in an AVS HDTV decoder.

Within the MC hardware, the interpolator is the most fundamental component. Existing literature has

‡ Corresponding author
* Project (No. Y106574) supported by the Natural Science Foundation of Zhejiang Province, China

proposed various architectures for a video codec interpolator. Most of them (He *et al.*, 2003; Chen *et al.*, 2004; Deng *et al.*, 2004; Song *et al.*, 2005; Wang R.G. *et al.*, 2005; Hyun *et al.*, 2006; Zhou and Liu, 2007) focus on luma interpolator design but very few of them (Lie *et al.*, 2005; Wang S.Z. *et al.*, 2005) mention chroma interpolator design. As a core part of the decoder, besides high performance interpolation, MC also requires efficient reference data preparation. In the literature few studies (Chen *et al.*, 2004; Tsai *et al.*, 2005; Zhang *et al.*, 2006) include the discussion of an effective data re-use scheme. In this paper, a high throughput bandwidth optimized AVS MC design is presented, which is capable of supporting the Jizhun profile level 6.0 1920×1088@30 fps video decoding at the working frequency of 108 MHz.

Compared to other AVS MC and its components designs (Deng *et al.*, 2004; Wang R.G. *et al.*, 2005; Zheng *et al.*, 2006), our contribution concentrates on two aspects. First, we developed an integrated interpolator with parallel serial filtering mixed data flow, thus improving data throughput. Second, we developed a three-stage multiplication free chroma interpolation scheme with standard compliant calculation, thus eliminating the drift error. In addition we propose a Vertical Z processing order based reference data re-use scheme that saves up to 30% memory access. As a result, our MC design has the characteristics of high throughput and optimized memory bandwidth.

The rest of this paper is organized as follows. Section 2 provides an overview of the AVS MC algorithm. Section 3 proposes two hardware adaptive methods for pixel interpolation and one effective data re-use scheme. The detailed interpolator architecture, filter structure and on-chip memory design are then presented in Section 4. Section 5 analyzes filtering latency and compares implementation results. And finally Section 6 concludes the paper.

OVERVIEW OF AVS MOTION COMPENSATION

This section provides an overview of the AVS MC algorithm. AVS utilizes fractional motion vector (FMV) for temporal prediction. Fig.1 shows the position of integer and fractional pixels. AVS utilizes different 4-tap FIR filters to calculate pixels at half and quarter positions. Table 1 lists the luma pixel filtering methods for 16 possible positions pointed by one pair of FMV.

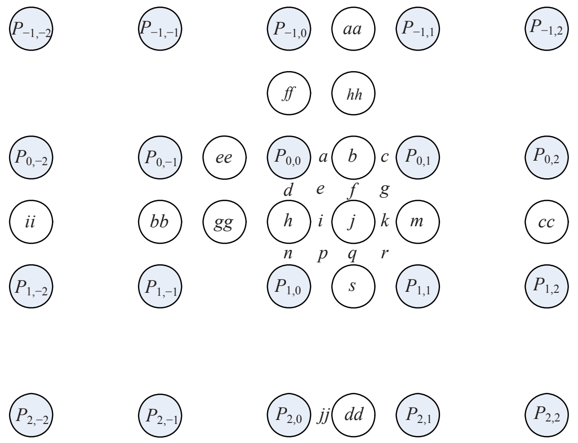

Fig.1 Position of luma integer, 1/2 and 1/4 pixels

Table 1 Luma pixel filtering methods

Pixel position	Pixels pointed by one MV	Filter type
Integer	$P_{0,0}$	
Half	b, h, j	[−1 5 5 −1]
Quarter	a, c, d, f, i, k, n, q	[1 7 7 1]
Diagonal	e, g, p, r	[1 1]

For example, half pixel ‘*b*’ is calculated as

$$b=5(P_{0,0}+P_{0,1})-(P_{0,-1}+P_{0,2}). \quad (1)$$

Quarter pixel ‘*a*’ is a bit more complex:

$$a=ee+7\times(8P_{0,0})+7b+8P_{0,1}. \quad (2)$$

The integer pixels are bypassed and directly output. The remaining diagonal pixels are generated by bilinear filters.

Eqs.(1) and (2) show that when interpolating an $X\times Y$ block (X, Y=8 or 16. X denotes the horizontal width and Y denotes the vertical height), it needs at most $(2+X+2)\times(2+Y+2)$ integer pixels. For the chroma components, AVS adopts a 1/8 precision interpolation method (Fig.2) denoted by

$$o=(8-\mathrm{d}x)\times(8-\mathrm{d}y)\times A+\mathrm{d}x\times(8-\mathrm{d}y)\times B$$
$$+\mathrm{d}x\times\mathrm{d}y\times C+(8-\mathrm{d}x)\times\mathrm{d}y\times D. \quad (3)$$

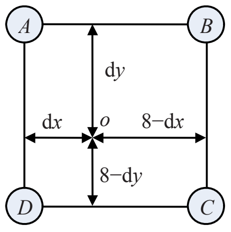

Fig.2 1/8 precision chroma pixel interpolation

Note that the filtered pixels are rounded and clipped into the range of 0 to 255 before output, except for integer pixels.

AVS supports VBS. For the inter-predicted MBs, a 16×16 MB can be partitioned into four 8×8 blocks, two 16×8 blocks or two 8×16 blocks. The grey area in Fig.3 illustrates the overlapped reference data region when decoding a 16×16 partitioned MB utilizing an 8×8-based interpolator. Without a data re-use scheme, the overlapped reference data will be reloaded, which leads to even higher memory bandwidth, as calculated in Section 1.

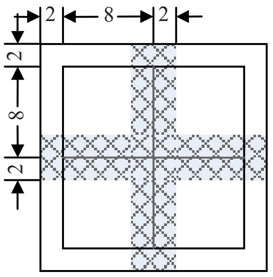

Fig.3 Overlapped reference data region. The grey area illustrates the overlapped reference data region when decoding a 16×16 partitioned MB utilizing an 8×8-based interpolator

Therefore, during the designing of AVS MC hardware, to improve data throughput and reduce memory bandwidth, we need to propose an efficient interpolation data flow and an effective data re-use scheme. The following section describes in detail our proposed methods.

PROPOSED METHODS

Parallel serial filtering mixed luma interpolation data flow

For luma pixel interpolation, several architectures have been proposed. They can be classified into three classes: Serial 1D approach (He *et al.*, 2003; Song *et al.*, 2005; Hyun *et al.*, 2006), Circular 1D approach (Deng *et al.*, 2004; Wang R.G. *et al.*, 2005; Zhang *et al.*, 2006) and Separate 1D approach (Chen *et al.*, 2004; Tsai *et al.*, 2005; Wang S.Z. *et al.*, 2005). Serial 1D approach has the lowest cost but is less efficient for data re-use and memory access. Circular 1D and its improved approach use parallel and pipeline structure to improve the throughput of a 1D structure at the cost of complex internal data multiplexing and additional register array. Compared with Circular 1D approach, Separate 1D approach has the advantage of straightforward data flow and achieves the balance between throughput and memory access.

Due to the feature of AVS TSFT (Wang *et al.*, 2004) interpolation algorithm that fractional pixel is dependent on not two but four or five neighboring integer pixels, thus conventional Separate 1D architectures cannot be directly applied. We propose a parallel serial filtering mixed data flow for AVS luma pixel interpolation.

We first use sample simplification as in (Wang R.G. *et al.*, 2005) to remove data dependency and multiplier usage.

Refer to Section 2: Eq.(2) shows that 1/4-pel ‘*a*’ is dependent on neighboring 1/2-pel ‘*ee*’ and ‘*b*’. Similar to 1/2-pel ‘*b*’ in Eq.(1), 1/2-pel ‘*ee*’ can be written as

$$ee=5(P_{0,-1}+P_{0,0})-(P_{0,-2}+P_{0,1}). \tag{4}$$

By substituting ‘*b*’ [Eq.(1)] and ‘*ee*’ [Eq.(4)] into Eq.(2), we get:

$$a=-P_{0,-2}-2P_{0,-1}+96P_{0,0}+42P_{0,1}-7P_{0,2}, \tag{5}$$

or

$$a=(P_{0,2}-P_{0,-2})+2(P_{0,1}-P_{0,-1})+8(P_{0,1}-P_{0,2}) +32(P_{0,0}+P_{0,1})+64P_{0,0}. \tag{6}$$

Compared to Eq.(2), Eqs.(5) and (6) have desirable features for hardware implementation. Both the equations remove the filtering dependency of 1/4-pel ‘*a*’ on 1/2-pel ‘*ee*’ and ‘*b*’, indicating that samples ‘*a*’ and ‘*b*’ can be filtered simultaneously. Furthermore, all the multiply coefficients of Eq.(6) are two’s power. This removes the use of the multiplier and can be simply realized with shift operation.

Then consider the reference data input. Integer samples of the reference picture are stored in row order in the external memory. It is natural that the interpolation module receives one row of integer samples at a time, so horizontal parallel filtering (HPF) of samples ‘*a*’ [Eq.(6)] and ‘*b*’ [Eq.(1)] is a fit choice. But for the samples ‘*d*’ and ‘*h*’, the situation is different. Their dependent data are fed into the interpolator one at a time. So serial filtering (SF) is a proper choice. Fig.4 shows the data flow of a simplified 4×4-based interpolator with HPF and VSF (vertical serial filtering).

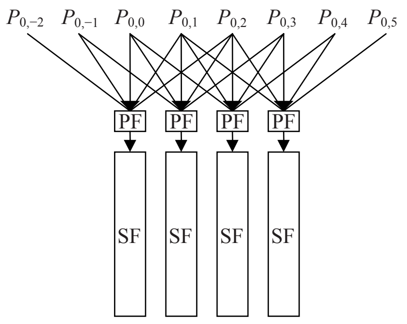

Fig.4 Data flow of a 4×4-based interpolator with mixed parallel serial filtering

It is worth noting, refer to Fig.1, that by adopting VSF we can calculate samples '*i*' and '*j*' as soon as their dependent data are available, i.e., samples '*ii*', '*bb*', '*h*', '*m*' and '*cc*'. In other words, as cycle time allows, we can get samples '*h*', '*i*' and '*j*' in one cycle. The scheme also reduces the register array size, for we do not need to store sample '*h*' for the calculation of samples '*i*' and '*j*'.

Three-stage multiplication free chroma interpolation scheme

Compared to luma interpolation, fewer works in the literature include a discussion on chroma interpolator design. Wang S.Z. *et al.*(2005) utilize each bit of an FMV (AVS denotes a 6-bit FMV pair as [*xFrac*, *yFrac*]) as the control bit and translate the multiplication form as $A\times xFrac\times yFrac$ into the accumulation process. Lie *et al.*(2005) propose a three-stage recursive interpolation algorithm. We simulate the algorithm but find it produces a slight drift error, for the algorithm prematurely truncates the intermediate pixel value. Compared to the accumulation process (Wang S.Z. *et al.*, 2005), the three-stage recursive technique has the advantage of regular data flow and shorter combinational logic latency. Therefore we modify the technique and propose a standard compliant drift error free three-stage recursive scheme for AVS chroma pixel interpolation.

Fig.5a shows 63 possible 1/8 precision chroma pixel positions pointed by one pair of FMV. Every black dot denotes a 1/8-pel except integer pixel *A*. Blue lines indicate that every four neighboring pixels constitute one square and the total 64 pixels constitute 16 squares. That is to say, each pixel belongs to one square.

Figs.5b to 5d describe the three-stage recursive scheme, taking pixel '*o*' as an example. Fig.5b illustrates that the process begins by first finding the 1st level square [a_1, b_1, c_1, d_1] where pixel '*o*' is in, followed by finding the 2nd level square [a_2, b_2, c_2, d_2] in Fig.5c where there is only one blue square. Then it determines that pixel '*o*' is located at c_3, and clips c_3 and finally outputs the value.

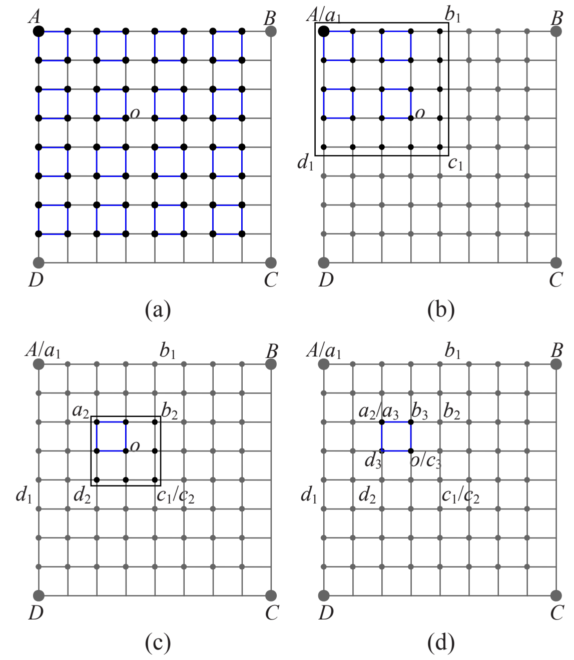

Fig.5 Illustration of a three-stage recursive scheme
(a) Position of chroma 1/8-pels; (b) 1st level square; (c) 2nd level square (d) 3rd level square

The detailed calculation process of 1/8-pel '*o*' is as follows:

For the 1st square [a_1, b_1, c_1, d_1]:

$$a_1=A<<2;\ b_1=(A+B)<<1;$$
$$c_1=(A+B+C+D);\ d_1=(A+D)<<1.$$

For the 2nd square [a_2, b_2, c_2, d_2]:

$$a_2=(a_1+b_1+c_1+d_1);\ b_2=(b_1+c_1)<<1;$$
$$c_2=c_1<<2;\ d_2=(c_1+d_1)<<1.$$

For the 3rd blue square [a_3, b_3, c_3, d_3]:

$$a_3=a_2<<2;\ b_3=(a_2+b_2)<<1;$$
$$c_3=(a_2+b_2+c_2+d_2);\ d_3=(a_2+d_2)<<1.$$

Finally, we get: $o=(c_3+32)>>6$.

This three-stage decomposition recursive process can be applied to chroma pixels at any position. The scheme utilizes addition and shift, thus eliminating multiplication. It does not truncate intermediate data in the calculation process, thus avoiding the drift error. It also shortens process latency between stages and provides a regular data flow.

Vertical Z processing order based reference data re-use scheme

Because of the VBS feature of AVS that there may be four MVs in an MB, the MC hardware design requires great effort in dealing with irregular memory access and interpolator utilization. Refer to Fig.3: a 16×16-based interpolator design has a minimal memory access when processing a 16×Y (Y=16 or 8) block because it can cover the vertical grey area of horizontal overlapped reference data without reloading them, but its hardware utilization is obviously low when processing an 8×Y block (since there is only 8 pixels in the horizontal, and the interpolator utilization is 50%). So we adopt an 8×8-based interpolator since one 16×Y block can be decomposed into two 8×Y blocks.

The 8×8-based interpolator has full hardware utilization but memory bandwidth overhead caused by overlapped reference data reloading becomes a serious problem. Wang S.Z. *et al.*(2005) propose a horizontal-switch approach that utilizes context switch buffers for temporary overlapped data storage. This approach only considers the horizontal overlapped region data re-use and partially solves the repetitive memory access at the cost of a double size register array inside the interpolator. Chen *et al.*(2004) and Tsai *et al.*(2005) propose a vertical integrated data flow to solve the vertical data re-use and classify the interpolation window using five cases to reduce memory access, which is complex but effective. We adopt this strategy and propose an AVS adaptive Vertical Z processing order based reference data re-use scheme.

Fig.6 shows our Vertical Z processing order. In the vertical direction, we use a Single Vertical Z (SVZ) processing order. If two vertical neighboring 8×8 blocks share the same MV, they are interpolated prior to the processing of horizontal neighboring blocks. SVZ applies to MB partitioned as one 16×16 block or two 8×16 blocks. The register array inside the interpolator acts as a data buffer for overlapped reference data and these data are processed sequentially. Refer to Fig.3: four rows of reference data are overlapped, thus the technique can save 4/[2×(2+8+2)]=16.7% memory access.

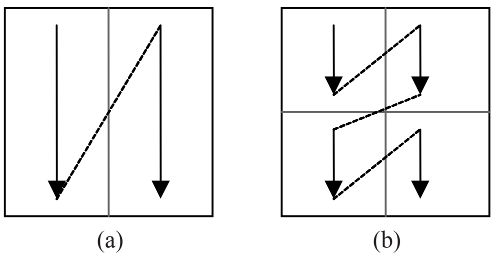

Fig.6 Vertical Z processing order
(a) Single Vertical Z (SVZ) applied to 16×16 or 8×16 partitioned MB; (b) Double Vertical Z (DVZ) applied to 16×8 or 8×8 partitioned MB

For the MB partitioned as two 16×8 blocks, the overlapped region is in the horizontal direction. The register array cannot play as buffer any more, thus we use a Double Vertical Z (DVZ) processing order instead of SVZ and add an on-chip memory to save horizontal re-usable data. DVZ is more effective and the details of this on-chip memory design will be given in the following section.

As to the MB partitioned as four 8×8 blocks, neighboring 8×8 blocks do not share any re-usable data. It seems that SVZ and DVZ are indifferent to the processing. But the fact that MV of the top right block is decoded prior to the MV of the bottom left block makes the DVZ a suitable choice.

ARCHITECTURE DESIGN

Fig.7 shows the data flow of the overall MC design. The 64-bit reference data input from the external memory is in row order. Address generation unit (AGU) generates memory controller compliant address parameters and the controller calculates the real address for memory access. We utilize an 88-bit 20-word register file for horizontal data re-use. The data align buffer selects the data from the external memory and register file, and packs them into aligned 12 pixels for luma interpolation or aligned 5 pixels for chroma interpolation. The 8×8-based interpolator supports simultaneously filtering 8 luma pixels or 4 chroma pixels. The interpolated pixels are sent to the

MC buffer to await combining with residues to complete MB reconstruction. The following subsections detailedly describe our interpolator architecture and register file design.

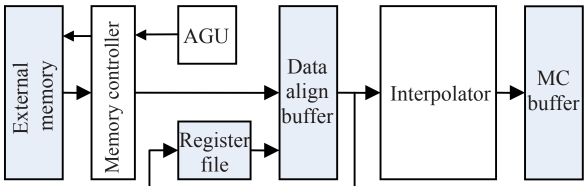

Fig.7 Simplified motion compensation data flow with reference data re-use scheme

Luma interpolator design

This subsection presents the most fundamental luma interpolator design. Fig.8 shows our proposed 8×8-based integrated interpolator architecture. We focus on its circuit and function for luma pixel interpolation. The solid arrow denotes data flow. The input is 2+8+2=12 luma integer pixels in row order, and the output is 8 interpolated pixels.

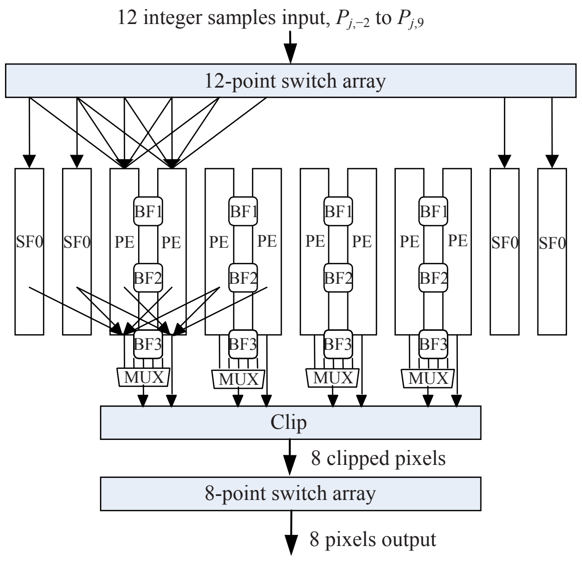

Fig.8 Proposed luma and chroma pixel integrated interpolator architecture

Input pixels first go through a 12-point switch array similar to that shown in Fig.9 and get reversed according to the quarter MV. Then these pixels, i.e., $I_{j,i}$, are fed into the register array. They are parallel filtered on the horizontal, and serially filtered in the vertical. After that, interpolated pixels are clipped, switched back to their original direction and output.

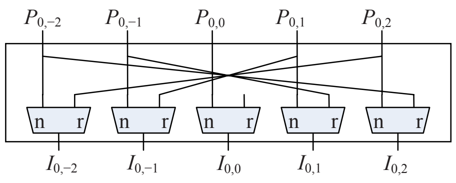

Fig.9 Structure of a 5-point switch array

To simplify the internal data flow and reduce the register array size, we utilize a technique called MV selective data input.

1. MV selective data input

The AVS interpolation scheme has symmetry in horizontal, vertical and diagonal directions. For example, horizontal 1/4-pel 'c' can be written as

$$c=(P_{0,-1}-P_{0,3})+2(P_{0,0}-P_{0,2})+8(P_{0,0}-P_{0,-1}) +32(P_{0,0}+P_{0,1})+64P_{0,1}. \quad (7)$$

Note that Eq.(7) has a similar form to Eq.(6). The correlation indicates that when filtering sample 'c', by reversely inputting data $P_{0,-1}$ to $P_{0,3}$, identical parallel filters can be used. Fig.9 shows the structure of a 5-point switch array to accomplish this task.

As to the vertical and diagonal cases, the situation is similar. Fig.10 shows 16 luma pixels classified into four squares by the quarter MV range. Fig.11 and Table 2 summarize data fetch and input operation when filtering samples in different squares.

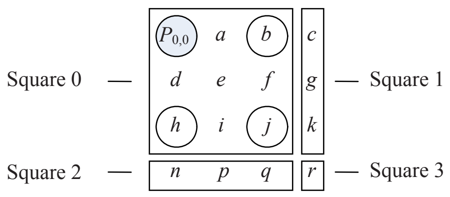

Fig.10 Sample classification by the quarter MV range

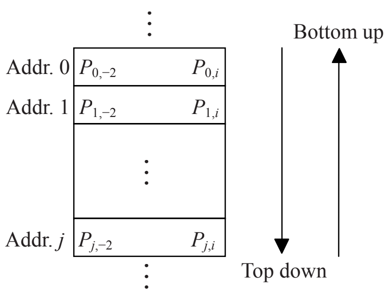

Fig.11 Reading operation from external memory

Table 2 Data fetch and input operation

Pixel from	Memory reading	Switch array
Square 0	Top down	Normal
Square 1	Top down	Reverse
Square 2	Bottom up	Normal
Square 3	Bottom up	Reverse

2. Filter design

The interpolator (Fig.8) consists of 4 serial filters (SF0) and 8 processing elements (PE). Bilinear filters (BF*n*, *n*=1, 2, 3) are used for chroma pixel interpolation. There are 5 integer pixels $I_{j,i}$, and 4 intermediate data from neighboring PE or SF0, fed into every PE. Fig.8 has illustrated the data flow of two PEs. The data flows of remaining PEs are similar.

Fig.12a shows the structure of parallel filter PF0 for Eq.(1). Fig.12b shows PF1 structure for Eq.(6). PF0 and PF1 constitute PF2 (Fig.12c), which is the component of PE as shown in Fig.13. It is worth noting that the output of PF2 is selected by the quarter MV, thus making the horizontal 1/2-pel and 1/4-pel share one register.

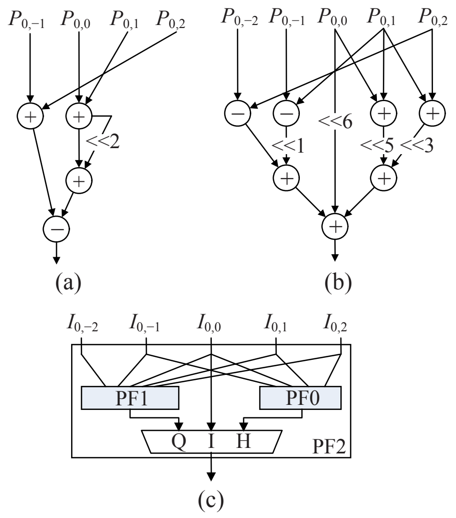
$P_{0,-1}$ $P_{0,0}$ $P_{0,1}$ $P_{0,2}$
<<2
(a)
$P_{0,-2}$ $P_{0,-1}$ $P_{0,0}$ $P_{0,1}$ $P_{0,2}$
<<6
<<1
<<5 <<3
(b)
$I_{0,-2}$ $I_{0,-1}$ $I_{0,0}$ $I_{0,1}$ $I_{0,2}$
PF1
PF0
Q I H
PF2
(c)

Fig.12 Parallel filter structure. (a) PF0; (b) PF1; (c) PF2

PE consists of four components (Fig.13a). Its structure is shown in Fig.13b. Registers D1 to D3 with their left combinational logic in Fig.13b constitute the structure of serial filter SF0 for Eq.(1), and registers D0 to D3 with their right combinational logic constitute the structure of SF1 for Eq.(5). Register D5 functions as a shift buffer for integer pixel storage. The adder at extreme bottom right functions as a bilinear filter (BF) for 1/4-pel in a diagonal direction.

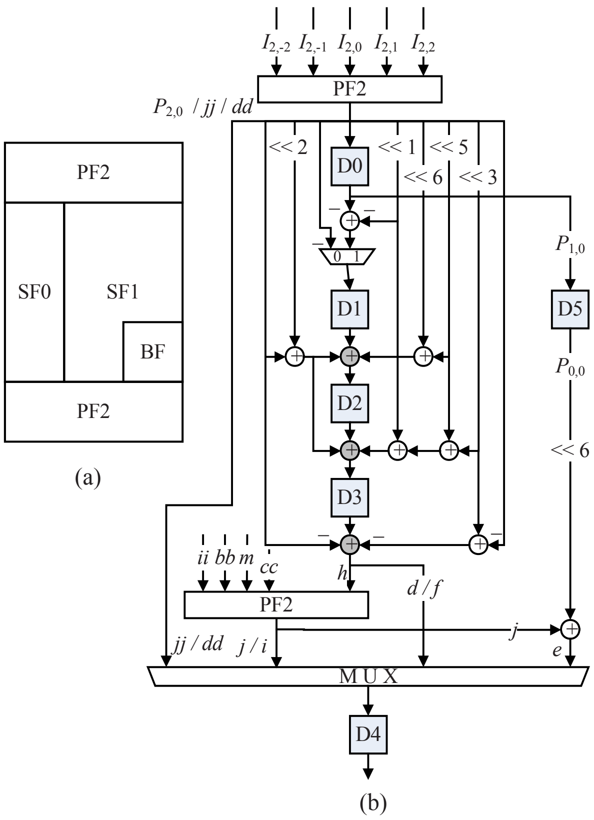

Fig.13 Processing element (PE) architecture
(a) Components of PE; (b) PE structure

3. Data flow

This paragraph describes the data flow of Fig.13b. Assume that MV points to one fractional pixel in Square 0 in Fig.10, and that the current input to the first PE are $I_{2,-2}$ to $I_{2,2}$, i.e., non-reversed input $P_{2,-2}$ to $P_{2,2}$. Referring to the pixel position in Fig.1, the output of the top PF2 is $P_{2,0}$, '*jj*' or '*dd*'. The grey adders in Fig.13b select left or right input. If grey adders select the left input, SF0 produces pixel '*h*'; conversely, if they select the right input, SF1 produces pixel '*d*' or '*f*'. The bottom PF2 receives pixels '*ii*' and '*bb*' from two left neighboring SF0, and pixels '*m*' and '*cc*' from two right neighboring PEs. The bottom PF2 produces pixel '*i*' or '*j*', and '*j*' is fed to the extreme bottom right adder with scaled integer pixel $P_{0,0}$ to produce diagonal pixel '*e*'. The bottom MUX selects pixel output according to the quarter MV.

Chroma interpolator design

Our chroma interpolator consists of 3-level 4-point bilinear filters denoted by BF*n* as shown in Fig.8. BF*n* denotes the *n*th-level bilinear filter. One column of BF*n* re-uses five registers (D0 to D4) from each neighboring PE as its input and output. Since chroma block size is half of luma block size, we employ a 4×4-based design thus to fully utilize bilinear filters and re-use register arrays.

Fig.14a shows the three-stage multiplication free interpolation data flow. Fig.14b shows the slightly modified PE which is capable of supporting the recursive scheme. The input to the 1st-level BF is four integer pixels: *A*, *B*, *C* and *D*. Here *A* and *B* come from register D0 of two neighboring PEs; *C* and *D* come from the input of two neighboring PEs. The output of the 1st-level BF is stored in registers D1 and D2, and serves as input to the 2nd-level BF. Registers D3 and D4 serve as input to the 3rd-level BF.

Fig.15 shows a general 4-point BF*n* design. The output MUXs are controlled by the quarter MV. Note that the dynamic range of output data is four times that of input data. This is because we perform truncation at the end of filtering in order to avoid the drift error during recursive computation.

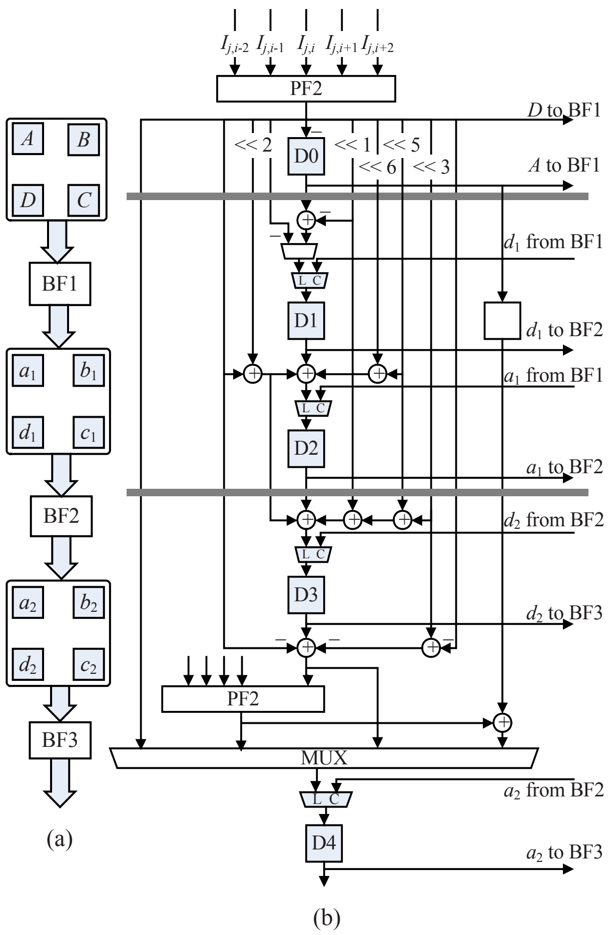

Fig.14 Illustration of chroma pixel interpolation
(a) Data flow of the three-stage multiplication free recursive scheme; (b) Modified PE structure to support chroma pixel interpolation

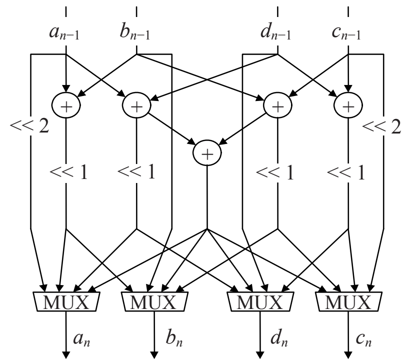

Fig.15 Four-point general BF*n* structure

On-chip memory design

Our integrated interpolator analyzed above processes pixels utilizing Vertical Z processing order as mentioned in Section 3. The horizontal data re-use scheme employing DVZ (Fig.6) needs the cooperation of on-chip memory and data align buffer. With careful analysis, we take advantage of an 88-bit 20-word single port register file, every address of which stores 11 integer pixels in row order.

Fig.16 illustrates all kinds of luma reference data fetches and interpolator input combinations. Our 8×8-based interpolator requires each time 12 pixels fed, which come from 2 or 3 external memory words. Among the 12 pixels, the last 4 pixels are re-usable if the two neighboring 8×8 blocks share the same MV. The remaining pixels are temporarily invalid, but they can be deemed as preloaded data for the horizontal neighboring 8×8 block. It is obvious that in Case 6 (Fig.16) there are a maximum of 7 preloaded data, so the maximum number of possible re-usable data becomes 4+7=11. For a 16×16 partitioned MB, there would be 2+16+2=20 rows of re-usable data. We also note that the left 8×8 block always stores re-usable data in the memory, and the right 8×8 block always fetches re-usable data from the memory. In other words, there would be no read-write conflict at the same time. As to the chroma reference data, the analysis is similar. For the 4×4-based chroma interpolator, each time it requires 5 pixels fed and the maximum re-usable data are 8+1=9 rows. Therefore, an 88-bit 20-word single port register file is capable of managing the re-usable data storage.

It is worth noting that DVZ notably reduces memory access. In Cases 1 to 5 (Fig.16), two words are fetched for one row interpolation of the left 8×8 block. When applying DVZ, another one word instead of two is required fetching for the same row interpolation of the right 8×8 block. Thus the memory access is reduced from 4 to 3. Similarly, in Cases 6 to 8, the memory access is reduced from 6 to 4. Thus the memory access reduction is up to 30%.

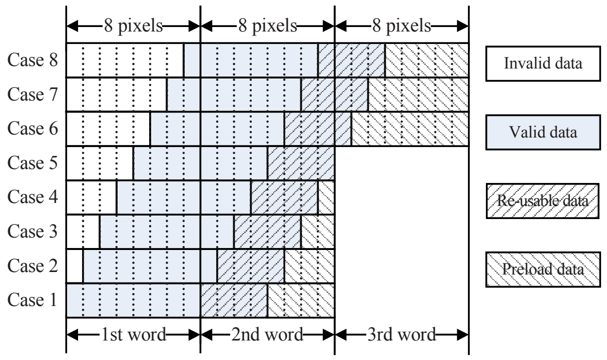

Fig.16 Luma reference data fetches and interpolator input combinations

ANALYSIS AND COMPARISON

Latency analysis

The previous section describes our MC hardware including interpolator architectures and horizontal re-use memory. The SVZ and DVZ processing orders are employed in the pixel filtering process to save memory bandwidth. Because of VBS and our horizontal and vertical data re-use scheme, the filtering latencies for different MB partitions are not the same.

Fig.17 demonstrates the MB filtering latency. We do not consider the pre-charge or activate latency of external memory. There is always one best latency and one worst latency corresponding to every MB partition case. For the 16×16 MB partition case at the bottom of Fig.17a, SVZ is applied. The vertical data re-use is carried out through internal register array, and the horizontal data re-use is accomplished with the cooperation of the on-chip register file and data align buffer. Refer to Fig.16: the best latency is achieved with reference data combination Cases 1 to 5. The interpolator needs 2 words of 12 valid pixels for the left 8×16 block and 1 additional word for the right 8×16 block. That is to say, 3 cycles are required for fetching reference data for each row, and there are in total 20 rows for a 16×16 block. Thus the processing time is 3×20=60 cycles. The worst latency occurs with Cases 6 to 8 in Fig.16. It needs 4 cycles for reference data fetch for each row, so the processing time is 4×20=80 cycles. As to the other luma MB partition cases and corresponding chroma MB partition cases, the latency derivations are similar.

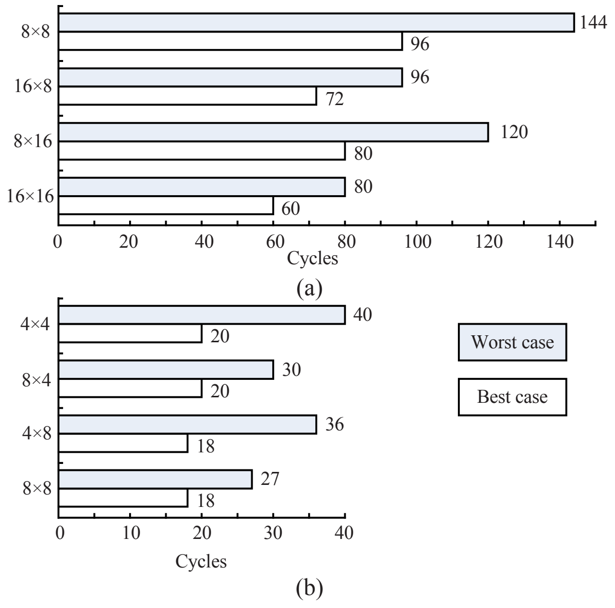

Fig.17 MB filtering latency
(a) Luma MB with different partitions; (b) Chroma MB with different partitions

As we expect, a 16×16 MB partition consumes the least processing cycles, for it utilizes both vertical and horizontal data re-use. An 8×16 MB partition can only utilize vertical data re-use while a 16×8 MB partition can only utilize horizontal data re-use. Fig.17 also shows that horizontal data re-use is more effective than vertical data re-use. At the top of Fig.17a, the 8×8 MB partition consumes the most processing cycles, since in both directions data re-use cannot be employed. The flexible data re-use results in different interpolation latencies. Without the data re-use scheme, the luma MB processing latency of our 8×8-based interpolator has a fixed value of 144 cycles in the worst case and 96 cycles in the best case (Fig.17a).

Implementation comparison

Our proposed MC hardware is described in Verilog and synthesized using 0.18-μm CMOS cell

Table 3 Comparison of 1D architectures proposed by other researchers with ours

	Architecture	Data storage array size	Pipeline level	Register in pipeline	Gate count	Cycles/luma MB
Deng *et al.*(2004)	Circular 1D	13×6	9	84	37.9k*	284
Wang R.G. *et al.*(2005)	Circular 1D	12×5	4	16		280
Ours	Separate 1D	4×4+8×6			26.2k	60~144

* Synthesized under 0.25-μm CMOS technology, with the total chip area being about 379 717 μm^2 (Deng *et al.*, 2004), using the equation that 1 μm^2 is about 100k gates

library. The interpolation results have been verified with that from the reference software and all data are matched. The total gate count is 44.3k excluding the on-chip register file when synthesized at 108 MHz.

Table 3 compares our implemented parallel serial filtering mixed Separate 1D luma interpolator with the Circular 1D approach. Our design requires no additional registers in the pipeline (Deng *et al.*, 2004; Wang R.G. *et al.*, 2005). The vertical serial filtering achieves the task done by the pipeline. Deng *et al.*(2004) employ an unnecessary 9-level long pipeline and utilize one filter for one fractional pixel. These two facts result in higher design costs. Wang R.G. *et al.*(2005) consider sample simplification and filter re-use. Though reducing the pipeline level to 4, Wang's implementation still has complex internal data multiplexing. The drawback of the Circular 1D approach is that it only interpolates one pixel per cycle.

Because of the 8-time parallelism Separate 1D architecture, our design requires the least number of cycles interpolating one MB. The throughput of the Circular 1D approach is close to 1 pixel per cycle. Since our design needs to wait 3 to 4 cycles for one row reference data fetch, the throughput of our design for the luma pixel interpolator is actually about 2.7 times that of the Circular 1D architecture.

In brief, our MC design needs 60 to 144 cycles to interpolate one 256-point luma MB and 18 to 40 cycles to interpolate each 64-point chroma MB. Up to 30% memory access is saved due to the Vertical Z processing order. Therefore, in the worst case, there is still a sufficient time margin left of at least 200 cycles for other tasks. This also means our design can work at lower frequency when decoding at a specified video resolution level, thus achieving lower power consumption.

CONCLUSION

This paper uses a Vertical Z processing order based MC hardware and its integrated interpolator architecture for an AVS HDTV decoder supporting Jizhun profile level 6.0. The vertical and horizontal data re-use scheme reduces by up to 30% memory access, thus alleviating memory bandwidth requirement. The parallel serial filtering mixed data flow and three-stage multiplication free chroma interpolation scheme increase the throughput close to 2.7 times compared with the conventional AVS interpolator design. Our implemented MC architecture also has an advantage in chip area and is suitable for integration in an AVS decoder.

References

AVS Reference Software, 2007. Ftp://159.226.42.57/public/avs_doc/avs_software

AVS Workgroup, 2004. Information Technology—Advanced Audio Video Coding Standard, Part 2: Video.

Chen, T.C., Huang, Y.W., Chen, L.G., 2004. Fully Utilized and Reusable Architecture for Fractional Motion Estimation of H.264/AVC. Proc. ICASSP, p.9-12. [doi:10.1109/ICASSP.2004.1327034]

Deng, L., Gao, W., Hu, M.Z., Ji, Z.Z., 2004. An efficient VLSI implementation for MC interpolation of AVS standard. *LNCS*, **3333**:200-206. [doi:10.1007/b104121]

Fan, L., Ma, S.W., Wu, F., 2004. Overview of AVS Video Standard. Proc. ICME, p.423-426.

He, W.F., Mao, Z.G., Wang, J.X., Wang, D.F., 2003. Design and Implementation of Motion Compensation for MPEG-4 AS Profile Streaming Video Decoding. Proc. ASICON, p.942-945.

Hyun, C.J., Kim, S.D., Sunwoo, M.H., 2006. Efficient Memory Reuse and Sub-Pixel Interpolation Algorithms for ME/MC of H.264/AVC. IEEE Workshop on Signal Processing Systems Design and Implementation, p.377-382. [doi:10.1109/SIPS.2006.352612]

Jia, H.Z., Zhang, P., Xie, D., Gao, W., 2006. An AVS HDTV video decoder architecture employing efficient HW/SW partitioning. *IEEE Trans. on Consumer Electronics*, **52**(4):1447-1453. [doi:10.1109/TCE.2006.273169]

Lie, W.N., Yeh, H.C., Lin, T.C.I., Chen, C.F., 2005. Hardware-Efficient Computing Architecture for Motion Compensation Interpolation in H.264 Video Coding. Proc. ISCAS, p.2136-2139. [doi:10.1109/ISCAS.2005.1465042]

Song, Y., Liu, Z.Y., Goto, S., Ikenaga, T., 2005. A VLSI Architecture for Motion Compensation Interpolation in H.264/AVC. Proc. ASICON, p.279-282. [doi:10.1109/ICASIC.2005.1611300]

Tsai, C.Y., Chen, T.C., Chen, T.W., Chen, L.G., 2005. Bandwidth Optimized Motion Compensation Hardware Design for H.264/AVC HDTV Decoder. Proc. 48th Midwest Symp. on Circuits and Systems, p.1199-1202. [doi:10.1109/MWSCAS.2005.1594322]

Wang, R.G., Huang, C., Li, J.T., Shen, Y.F., 2004. Sub-Pixel motion Compensation Interpolation Filter in AVS. Proc. ICME, p.93-96.

Wang, R.G., Li, J.T., Huang, C., Zhang, Y.D., 2005. A subpixel motion compensation interpolation method and its high performance VLSI design. *Chin. J. Computers*, **28**(12):2052-2058 (in Chinese).

Wang, S.Z., Lin, T.A., Liu, T.M., Lee, C.Y., 2005. A New Motion Compensation Design for H.264/AVC Decoder. Proc. ISCAS, p.4558-4561. [doi:10.1109/ISCAS.2005.1465646]

Yu, L., Yi, F., Dong, J., Zhang, C.X., 2005. Overview of AVS—Video: Tools, Performance and Complexity. Proc. SPIE, **5960**:679-690. [doi:10.1117/12.632515]

Zhang, N.R., Li, M., Wu, C.W., 2006. High Performance and Efficient Bandwidth Motion Compensation VLSI Design for H.264/AVC Decoder. Proc. ICSICT, p.1896-1898. [doi:10.1109/ICSICT.2006.306500]

Zheng, J.H., Deng, L., Zhang, P., Xie, D., 2006. An efficient VLSI architecture for motion compensation of AVS HDTV decoder. *J. Computer Sci. Technol.*, **21**(3):370-377. [doi:10.1007/s11390-006-0370-8]

Zhou, D.J., Liu, P.L., 2007. A Hardware-Efficient Dual-Standard VLSI Architecture for MC Interpolation in AVS and H.264. Proc. ISCAS, p.2910-2913. [doi:10.1109/ISCAS.2007.377858]