Abstract—The computational complexity of H.264 video coding standard is two or three times higher than that of H.263 and MPEG-4. Especially, the operations of the entropy coding and deblowing filter are the most complex parts in the decoder. In order to reduce the computational complexity of these operations, we propose a fast algorithm for H.264 decoder implementation, which containing a group-based CAVLC decoding method for H.264 entropy code-tables and an optimized pre-judging Bs method for H.264 deblowing filter to reduce the computational complexity. The computer simulation results indicate that the entropy decode algorithm can decrease 82.3%~87.9% decoding time than the original H.264 reference software and the improved deblowing filter algorithm can decrease 69.4%~76.9% computing time compared to the original H.264 reference software.

Keywords—H.264; CAVLC; Deblowing Filter

I. INTRODUCTION

The H.264 video coding standard provides enhanced coding efficiency for a wide range of applications. However, the computational complexity of the H.264 is two or three times higher than that of H.263 and MPEG-4. The increased computational complexity causes effectiveness problems in developing H.264 based video solutions.

For the sake of further increasing the data compression rate, the H.264 adopts the Context-adaptive variable length coding (CAVLC) entropy coding method to encode the residual data. Compared with previous entropy coding methods, CAVLC introduces the concept of context model to model the probability of symbols more accurately so that the compression ratio can be further increased. However, the trade-off for high compression ratio of CAVLC is its high computational complexity and long encoding or decoding time.

The deblowing filter, which is used on both encoder and decoder sides, is designed to reduce blocking artifacts that are produced by block-based video coding. On encoder side, by using a filtered image as a reference picture, efficient motion estimation is available. On the other hand, the deblowing filter improves the decoding video quality. However, the deblowing filter, which is applied to all the vertical and horizontal edges of blocks, is a very complex process. Thus, decreasing computation of the deblowing filter is necessary for real-time processing of H.264 encoder/decoder.

Though efficient techniques have been proposed to reduce the implementation complexity recently, the complexity is still high because of the flow of algorithm itself. Many efficient hardware implementations for H.264 have been proposed [1]-[2], but few consider the algorithmic optimization for overall speedup in H.264. Thus, the development of an effective optimized algorithm for H.264 is inevitable.

This paper is organized as follows. In Section II, we simply review the principle of CAVLC decoding and a new group-based lookup table algorithm is introduced. In Section III, the principle of the deblowing filter is first introduced, and a new optimized deblowing filter algorithm is then proposed. The implementation results are shown in Section IV. And the conclusion will be given in Section V.

II. THE OPTIMIZED CAVLC DECODING

CAVLC is used to code residual data, which is obtained after transform and quantization. The residual data is usually sparse with a large number of zeros and is run length encoded. The coefficients obtained after run length coding are sent using CAVLC. In CAVLC coding, the total number of non-zero coefficients and the number of trailing ones are coded into a single variable length code: coeff_token. The sign of each trailing one is coded as a single bit: Sign of Ts. The value of each non-zero coefficient except for Ts: level, the total number of zeros preceding the last non-zero coefficient: total_zeros and the number of successive zero coefficients preceding non-zero coefficients: run_before are also coded as variable length codes and follow the order [3].

H.264 uses the lookup tables method depending on the context for coding above mentioned elements. CAVLC decoder selects the tables depending on the number of non-zero coefficients in neighboring blocks.

(1) coeff_token decoding process: First, the process is started by decoding the TotalCoeffs and the TrailingOnes (Ts). These two values are decoded by coeff_token table which is divided into five sub-tables according to the variable nC, i.e., Num-VLC0 (nC = -1), Num-VLC1 (0 <= nC < 2), Num-VLC2 (2 <= nC <4), Num-VLC3 (4 <= nC < 8), FLC (8 <= nC). The nC means the number of coefficients in neighbor blocks.

(2) Sign of Ts decoding process: According to the Ts, the corresponding bits are taken to decode the sign of trailing values. A single bit is used to decode the sign (0= +1, 1= -1) of each Ts in reverse order.

(3) level decoding process: The level of each non-zero coefficient is decoded in reverse order. There are seven choices of the tables in this part, table Level_VLC0 to Level_VLC6. The choice of VLC table to decode each level...
is decided according to the magnitude of each successive coded level. Each item in these VLC tables is represented as “0…01x…xs”. The “0…01” is the prefix, and the “x…xs” is the suffix in which the “s” means the sign of the level. The suffix decoding process depends on the prefix and the level decoding is decided by previous decoded level.

(4) total_zeros decoding process: The total_zeros tables are applied for decoding AC 4×4 blocks or DC 2×2 blocks. The VLC tables to decode the total zeros is decided according to the total number of the non-zero coefficients in the current AC 4×4 or DC 2×2 blocks.

(5) run_before decoding process: In this process, the number of zeros between two adjacent coefficients is decoded in reverse order. The VLC tables for each run of zeros is chosen on the zerosLeft which is calculated by subtracting previous run from previous zerosLeft and initialized with TotalZeros. When zerosLeft is equal to 0, the remaining runs of rest coefficients is set to 0, which means the order of rest coefficients is the same as the decoding order in level stage.

It’s known that the sign of T1s and level elements are usually decompressed by some arithmetic operations owing to the well-structured codewords. But the decoding of the other syntax elements is based on the lookup tables, which are implemented with the memory. So the most important thing for the CAVLC decoding is the way to build the code-tables and to look up the tables. In the H.264/AVC standard, the code-tables have already been defined in advance, so the optimization is how to look up the code-tables.

In H.264/AVC standard, the code-tables of CAVLC are 2D tables, in which the indexes stand for the two syntax elements: TotalCoeffs and T1s and the contents of these tables are the codewords. While decoding, it’s difficult to get the indexes by looking up the context, because it would search the whole table and cost high computation and memory accesses. When decoding the VLC codewords, because the length of the codeword is variable and there is no partition mark between the codewords, the first and most important steps is to judge the length of each codeword.

In H.264 reference software, the CAVLC decoding method is to retrieve the codeword from the code-table by matching the consecutive coding stream bit by bit, which is very slow and consumes much computation because of heavy memory accesses and many conditional statements due to sequential search[4].

To improve the speed of entropy decoding, we develop a new group-based searching algorithm, which is based on the features in the proposed codewords and symbols memory mapping.

We allocate the codewords into two classes according to their bit width, i.e. the no more than 8-bit codewords and the more than 8-bit codewords. Then, we build the 8-bit auxiliary table and first put the no more than 8-bit codewords into it, which mapping the codewords and the range of the reading datas. Table 1 gives an example to show the auxiliary table for the no more than 8-bit codewords (0<= nC <2).

<table>
<thead>
<tr>
<th>Codeword</th>
<th>Data of reading</th>
<th>Range of the data</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1******</td>
<td>00000000</td>
</tr>
<tr>
<td>01</td>
<td>01******</td>
<td>00000000</td>
</tr>
<tr>
<td>001</td>
<td>001*******</td>
<td>00000000</td>
</tr>
<tr>
<td>00011</td>
<td>00011***</td>
<td>00000100</td>
</tr>
<tr>
<td>000011</td>
<td>000011**</td>
<td>00001100</td>
</tr>
<tr>
<td>000100</td>
<td>000100**</td>
<td>00010000</td>
</tr>
<tr>
<td>0000100</td>
<td>0000100*</td>
<td>00001000</td>
</tr>
<tr>
<td>0000101</td>
<td>0000101*</td>
<td>00001010</td>
</tr>
<tr>
<td>00001111</td>
<td>00001111</td>
<td>00000111</td>
</tr>
</tbody>
</table>

Then, for these more than 8-bit codewords, we put those which have the same 8-bit prefix codewords into the 8-bit auxiliary table as a parent table and build up auxiliary sub-tables recursively, which the width of the sub-table is subtracted 8 from the length of the longest codeword in the sub-table. The code-length of the prefix in the parent table will be a negative value that means the codeword is a more than 8-bit code and it points to its corresponding sub-table. In a word, the structure of each code-table can be divided into groups based on leading prefixes. These codewords, which have the same lengths and the same prefixes, can be grouped together into the sub-tables.

![Figure 1. The architecture of the tables](image-url)

Figure 1. The architecture of the tables

While decoding, we first read an 8-bit unit data, then through looking up the parent table, if the length of the codeword is a no more than 8-bit, we can get the codeword immediately. But if the length of the codeword is more than
8-bit, we should look up the sub-table to confirm the final codeword. Fig.1 shows the architecture of these tables, in which the symbol “000101” is a no more than 8-bit codeword, whereas the symbol “000000110” is a more than 8-bit codeword, which is divided into two parts. The 8-bit prefix of the symbol “000000110” is stored in the parent table, in which the code-length value is negative so you must read the sub-table. In the parent table, the unit pointes to the sub-table, in which stores the suffix of the symbol “000000110”.

Through these auxiliary tables, we build a symbol mapping module to allocate every possible input symbol into its corresponding class. Thus, a lookup table method using 8-bit shift can be realized for fast exact location.

### III. THE OPTIMIZED DEBLOCKING FILTER METHOD

In H.264/AVC standard, the deblocking filter is divided into 3 parts: The boundary strength of the filter depends on the coding modes of neighbor blocks in the current filtering edge. The filtering decision analyzes whether the filtering should be switched off. And the filtering operation applies on each 4 × 4 luminance and chrominance block edge on a macroblock basis, because the transform coding is operated on 4 × 4 blocks.

1. Boundary Strength (Bs) Decision: Bs decides the strength of the deblocking filter operation, which is associated with each block. The value of Bs, which is divided into different values between 0 and 4, depends on inter/intra prediction, motion vector difference and coded residuals. The result of applying the rules is that: when the Bs equal to 4 represents the strongest mode filtering, from 1 to 3 corresponds to the standard mode filtering and 0 means skip filtering.

2. Filtering Decision: When Bs equals to zero, the filtering doesn’t take place for edges. For edges with nonzero Bs values, a gradient-like analysis is performed to decide whether the filtering should be switched off.

3. Filtering Operation: The edge filtering starts to filter when the input pixels, boundary strength and threshold variables are ready. There is a branch depending on the value of Bs. If Bs is less than 4, there are at most 4 pixels to be modified. Otherwise, there are at most 6 pixels to be modified.

From [5] we could get that more than 90% of the filtering computation resources were spent on boundary strength computation. Thus, we should focus on optimization of the boundary strength algorithm for a significant reduction in computation.

The processing order of the computing Bs in an MB is shown in Fig.2. First, we should process the four vertical edges (a−d), and then process the four horizontal edges (e−h). The major drawback of this direct approach is that the computational times of Bs for each pixel in the edge is as large as 16 times, which results in 128 times computation for a 16 × 16 luminance block. Besides it, the approach doesn’t distinguish the macroblock coding mode (intra/inter), which will lead to different estimate methods. The H.264 reference software directly adopts this approach, and thus requires high computation cost. In order to reduce the computational complexity, we would propose an optimized pre-judging Bs method.

First, according to the H.264 standard, judging Bs 4 or 3 is very simple, which only depends on whether the p or q is intra coded and the boundary is a macroblock boundary. Besides, before filtering we could get some encoding or decoding information, using which we could easily estimate the current macroblock coding mode. Through using this information, we could pre-judge the value of Bs when it’s equal to 4 or 3. In this way, we only use an “if judgment” to compute the Bs when it equals to 4 or 3 and decrease a large amount of loop judgments in the filtering flow.

Second, on each macroblock edge the 16 pixels are highly spatial correlated, especially in the smooth surface. Moreover, in the H.264/AVC standard the basic filtering unit is 4 × 4 block, so the adjacent pixels on an edge may result into similar boundary strengths. To improve the speed of computing Bs, some of the boundary strength computation may be skipped. For the intra macroblock, the 16 pixels in the vertical or horizontal line could be treated as a unit and only the Bs of the first pixel is computed, and the Bs of the rest 15 pixels are directly copied from the first one. For the inter macroblock, the 16 pixels in the vertical or horizontal line could be divided into 4 units. In each unit only the Bs of the first pixel is computed and the Bs of the rest 3 pixels are directly copied from the first one.

Third, from [6] we could see that the large block inter modes(skipp, 16 × 16, 16 × 8, 8 × 16) occupy large proportion of the encoding modes, which the percentage is up to 60%. For these large block inter modes, in the interior of these large blocks we could find that those pixels on both sides of the filtering edge have the same reference pictures and motion vector values. As Fig. 3 shown, for example if the coding mode is 16 × 16 inter block, while filtering vertical edge b, we could see that these pixels on both sides of edge b have the same reference pictures and motion vector values, so for the 16 pixels we might set the same value of Bs in advance; Also, while filtering horizontal edge f, those pixels on both sides of edge f have the same reference pictures and motion vector values, so the values of Bs for the 16 pixels might be set equally in advance too. After this, when going to modifying Bs computation, if these blocks contain coded coefficients, the value might be modified at last. Similarly, the situation happens on 16 × 8 and 8 × 16 inter blocks.

![Figure 2. The order of computing Bs](image-url)
Figure 3. The simplified Bs computing method

So a simplified pre-judging large inter block Bs strategy is proposed.

1. For the skip and 16 × 16 inter mode, the Bs of the b, c, d, f, g, h edges is directly equal to 0;
2. For the 16 × 8 inter mode, while filtering the vertical lines, the Bs of the b, c, d edges is directly equal to 0; For the 16 × 8 inter mode, while filtering the horizontal lines, the Bs of the f, h edges is directly equal to 0; And only the Bs of the first pixel in g edge is computed, the Bs of the rest 15 pixels are directly copied from the first one;
3. For the 8 × 16 inter mode, while filtering the horizontal lines, the Bs of the f, g, h edges is directly equal to 0. For the 8 × 16 inter mode, while filtering the vertical lines, the Bs of the b, d edges is directly equal to 0; And only the Bs of the first pixel in c edge is computed, the Bs of the rest 15 pixels are directly copied from the first one.

In all, we analyze these pixels similar information on both sides of the filtering edge which is caused by the coding modes, use these useful information to do pre-judgments, which decrease the loop computation for Bs. Because we have done the prejudication operation, a mount of computation could be omitted.

IV. EXPERIMENTAL RESULTS

The proposed optimized method is implemented on an Intel Pentium4 Processor 2.4GHz with 512MB of memory hardware environment. In all of our experiments four sequences (300 frames), i.e., foreman, container, coastguard, and mobile are used with QCIF format (176×144). The frame rate is 30 fps, Period of I-frames is 15.

The comparison of the filtering algorithms is taken between the H.264/AVC reference software JM8.6 and our proposed optimized one with QP is 28. We can see that the speed up results between our proposed algorithm and the JM algorithm in table III, from which the percentage of the saving of average filtering time are 72.1%, 76.9%, 69.5% and 69.4% respectively.

TABLE III. THE FILTERING TIME COMPARISON

<table>
<thead>
<tr>
<th>Sequence Name</th>
<th>JM Algorithm</th>
<th>Optimized Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>foreman</td>
<td>Filtering time (ms)</td>
<td>PSNR (dB)</td>
</tr>
<tr>
<td></td>
<td>1581.3</td>
<td>38.86</td>
</tr>
<tr>
<td>container</td>
<td>1001.2</td>
<td>36.69</td>
</tr>
<tr>
<td>coastguard</td>
<td>1662.7</td>
<td>40.35</td>
</tr>
<tr>
<td>mobile</td>
<td>1799.1</td>
<td>34.92</td>
</tr>
</tbody>
</table>

V. CONCLUSION

In order to reduce the computational complexity of H.264, we proposed an effective group-based CAVLC decoding lookup method and propose an optimized pre-judging Bs method for the decoder. The experimental results show that the saving of average CAVLC decoding time is about 82.3%–87.9%, and the saving of average filtering time is from 69.4% to 76.9% according to different sequences and the PSNR achieved by our algorithms are the same to the H.264 reference software.

REFERENCES