Fast multiple \textit{run\_before} decoding method for efficient implementation of an H.264/advanced video coding context-adaptive variable length coding decoder

Dae Wook Ki
Jae Ho Kim
Fast multiple run_before decoding method for efficient implementation of an H.264/advanced video coding context-adaptive variable length coding decoder

Dae Wook Ki
Jae Ho Kim
Pusan National University
Department of Electronics Engineering
San 30, Jangjeon-Dong
Kumjeong-Gu, Pusan, 609-735 Republic of Korea
E-mail: jhkim@pusan.ac.kr

Abstract. We propose a fast new multiple run_before decoding method in context-adaptive variable length coding (CAVLC). The transform coefficients are coded using CAVLC, in which the run_before symbols are generated for a 4 × 4 block input. To speed up the CAVLC decoding, the run_before symbols need to be decoded in parallel. We implemented a new CAVLC table for simultaneous decoding of up to three run_befores. The simulation results show a Total Speed-up Factor of 205% ~ 144% over various resolutions and quantization steps. © The Authors. Published by SPIE under a Creative Commons Attribution 3.0 Unported License. Distribution or reproduction in any form proceeding this work in whole or in part requires full attribution of the original publication, including its DOI.

Subject terms: run_before, context-adaptive variable length coding; H.264/advanced video coding; multiple decoding.

Paper 120601 received Apr. 25, 2012; revised manuscript received Oct. 6, 2012; accepted for publication Oct. 11, 2012; published online Mar. 4, 2013.

1 Introduction

Efficient decoding of H.264/advanced video coding (AVC) is quite important in personal computer (PC)-based decoding as well as for application specific integrated circuit (ASIC) implementation. Low power implementations are more important for mobile devices such as smart phones and digital multimedia broadcasting.

H.264/AVC uses context-adaptive variable length coding (CAVLC) for encoding the transformed coefficients of 4 × 4 blocks. The method is composed of five syntax elements: TotalCoeff (coeff_token), TrailingOnes (coeff_token) and trailing_ones_sign_flag, level, total_zeros, and run_before. For TrailingOnes (or T1s), the trailing_ones_sign_flag is assigned to 0 or 1 to represent the “+” or “−” sign of the trailing ones, respectively. For this syntax element, there is no need for decoding in the calculation of the table access. There is only one value of TotalCoeff(coeff_token), T1(coeff_token), and total_zeros for one 4 × 4 block. But in many cases there may be more than one “level” and “run_before” value for the block.

Moon3 proposed a method to reduce the large number of memory access operations in the decoding process for coeff_token. He used new variable-length decoding (VLD)s for coeff_token and a state machine instead of the run_before decoding table. This method reduced 95% of the memory access, which is quite a good improvement for a low power hardware implementation. Kim et al.4 used conventional digital signal processor (DSP) or general purpose processor (GPP) instructions for CAVLC decoding. As shown in Fig. 1 the run_before table is divided into five groups and their decoding algorithms are proposed separately. They removed most of the table accesses. This has advantages for DSP- or GPP-based systems; however, the hardware implementation is complex as the equation uses complicated calculation logic.

For implementing the 1-5 groups of Fig. 1, hardware blocks such as shifters, adders, and multiplexers are needed. For the outer “if then else” statements, the condition calculation logics and multiplexers are needed.

Many researchers have contributed different methods to the efficient decoding of CAVLC in H.264/AVC.5-8 Key to these methods is noting the bottleneck that inhibits decoding speed. The authors propose to increase the speed of run_before decoding since in many cases there are multiple run_befores for one 4 × 4 block. This is also the case for the levels in CAVLC; however, as the multiple parallel decoding of levels is quite complex, this paper only focuses on the multiple parallel decoding of run_before.

Wen6 earlier proposed parallel multiple run_befores, using parallel multiple hardware decoders. It is shown in Fig. 2. A total of six decoding units were used in the actual design of the hardware. Each decoding unit had a different starting position for the input bit stream, and they produced decoding results or failures according to each bit stream input. Therefore, each decoder needed its own variable-length coding (VLC) table. The hardware complexity was quite high, as was the power consumption since the units ran in parallel. This did not improve the decoding algorithm and it also increased the degree of hardware complexity.

The method of Nikara7 is an MPEG2 version of Wen6 using multiple decoders. Each decoder has a complete decoding logics and a code detector (CD) table. Their starting bit positions in the input bit stream are different. They are trying to decode their own bit streams simultaneously. Some result in success_flags, the decoded bits, and the code lengths. Others result in failures. This method is used to speed up the hardware decoding process. Before this is proposed, a simple decoding is iterated many times. Compared to that, this method reduces the overall hardware decoding time. But there is no algorithmic improvement in Nikara7 and the required hardware size is increasing as the multiple decoding logics are used in parallel.

As an alternative, the authors propose fast multiple run_befores decoding with reduced memory access. The main
idea is that there is a higher probability of run_befores with shorter coded bits. During our research, we found many cases in which three run_before codes can be decoded using only one 8-bit table access. In those cases, one table access can produce three decoded values of run_before. This has the benefit of a low power implementation due to fewer table accesses. This is a big achievement for implementation compared to Nikara.5 There is no need for many decoders running in parallel in the proposed idea.

This paper is organized as follows. Section 2 reviews the general CAVLC algorithm and briefly describes the contributions from earlier papers. Section 3 explains the proposed method, and Sec. 4 analyzes the experimental results. Section 5 provides the conclusions regarding the proposed algorithm.

2 H.264/AVC CAVLC
CAVLC decoding is the inverse of encoding and operates on coded information from the compressed bit stream.

In the encoding side, zigzag scanning is applied after quantization of the transformed coefficients, and then CAVLC is used to encode the transformed coefficients. The following syntax elements are used for coding efficiency:

1. The total number of coefficients (TotalCoef or Tc): Tc for the neighborhood blocks are used in selecting the current VLC look-up table.
2. TrailingOnes (T1s): After quantization, most of the nonzero coefficients are +1 or −1. They are called T1s. Note that their signs are coded separately as trailing_ones_sign_flag. If there are more than three T1s, they are regarded as level.
3. level: Coefficient values above 1 are coded as level. Note that the value “1” after counting three “1”s is also regarded as level. For coding these levels, the VLC look-up table is selected adaptively using information from the most recently coded level.
4. total_zeros: Encodes the total number of zeros occurring after the first nonzero coefficient.
5. run_before: The quantized block typically has many zero coefficients. CAVLC uses run_before for the efficient coding of zeros.

Figure 3 shows the five decoding process for the five syntaxes of CAVLC. The quantized 4 × 4 coefficients are reordered by zigzag scanning and encoded by the CAVLC encoder. When the encoded bit stream is provided to the

![Fig. 1 The single run-before decoding method of Kim](image)

![Fig. 2 An example of multiple symbol decoding](image)
decoder, coeff_token, signs of TIs, level, total_zero, and run_before are decoded sequentially.

3 Proposed Multiple run_before Decoding Algorithm

In the proposed method, up to three run_before can be decoded simultaneously. The number “zerosLeft” indicates how many “0”s are not yet decoded. The code length of run_before is a maximum 3 bits if zerosLeft is between 1 and 6. In the entropy coding, a shorter code length means a higher occurrence probability of the code.

Figure 4 shows the run_before decoding table. The codes in region (1) provide the target where multiple decoding is possible. The codes in region (2) are decoded by the normal decoding method because they are less likely. If zerosLeft is not greater than 6, the proposed multiple decoding is applied. For various test sequences, we found that the average probability of regions (1) and (2) are 87.3% and 12.7%, respectively.

3.1 Proposed Multiple Decoding Table

Region (1) in the run_before table needs to be reorganized for multiple decoding. The revised table is shown in Fig. 5(a). The 8-bit buffer holding the input bit stream is used for the column address of the table, and zerosLeft is used for the row address. The memory output of the table shows three run_before values and the corresponding run_before code bits consumed.

An example illustrates the usage of the new table, shown in Fig. 5(b). The zigzag scanned coefficients arrive from right to left. The relationship of zerosLeft and run_before is depicted in Fig. 5(b), which shows that zerosLeft is reduced by the value of a decoded run_before. Assume that an encoded bit stream is “10011.” This means that run_before and zerosLeft are (1,2,0,1) and (4,3,1,1) in the coding process, respectively. Of course, this is unknown until they are subsequently decoded correctly.

Total_zeros is decoded as “4” just before the run_before decoding starts. The coded bit stream “10011xxx” is stored in the input bit buffer of the decoder, and zerosLeft is initialized with total_zeros at the start of run_before decoding. The bit data of the input buffer and zerosLeft are used for the column and row addresses of the table. Then “001 010 000 101” is the output of the table. The first three 4-bit numbers represent the run_before value 1, 2, and 0, respectively. The last 4-bit number shows the consumed number of bits, which is used to shift the buffer for the next decoding process. Because three run_before are decoded, the next starting zerosLeft becomes 1.

If there are more than three run_before, then additional iterations of this process are needed. The run_before decoding table size in Fig. 5(a) is 6 (row size) × 256 (column size) × 12 bits (output bits size), i.e., 18,432 bits. Our aim is to
decode a maximum of three run_befores at once. We need three tables to support either one, two, or three run_befores (see the following section). Therefore, the required table size is 18,432 bits $\times 3 = 55,296$ bits. This is too large for a hardware implementation.

### 3.2 Reducing the Multiple Decoding Table

Figure 6(a) shows the special case in which the coefficients before $C_n$ are all "0," called the leading zero case (LZC). Note that $C_n$ is a nonzero coefficient. In LZC, after decoding $C_n$, the decoder knows that all nonzero coefficients are decoded and that the remaining coefficients are all zeros. The encoder does not assign any bits for the run_before of the leading zeros. In the leading zero case, the multiple run_befores decoding table needs to be separated; an explanation follows.

Figure 6(b) and 6(c) shows how the decoding is performed regardless of whether there are leading zeros.

In the encoding process, zerosLeft is set to 6 for both Fig. 7(a) and 7(b). Consider the run_before values for Fig. 7(a). The first (before $C_1$), second (before $C_2$), and third (before $C_3$) run_befores are 1, 2, and 3, respectively. But the third run_before is not coded because it is the leading zeros. Therefore, the run_before stream for (a) is "000011." The next syntax element (xx) will be concatenated, so that "000011xx" will be the encoded stream. On the other hand, in the case of Fig. 7(b), the first, second, and third run_befores are 1, 2, and 3, respectively, and the run_before stream is "00001100." On the decoding side, when "00001100" arrives we cannot distinguish between the Fig. 7(b) case "00001100" or the Fig. 7(a) case "000011xx" with the following syntax element $xx = 00$. The number of consumed bits for Fig. 7(a) and 7(b) are 6 and 8, respectively.

![Fig. 6](image)

![Fig. 7](image)

![Fig. 8](image)
In other words, the same bit stream "00001100" should produce different table outputs according to whether there is a leading zero case, and we need to separate the table. Considering how many run_befores are decoded at once, the above two examples Fig. 7(a) and 7(b) can be decoded with the tables having two and three run_befores. These tables are called run_before-table (RBT)3 and RBT2. For the same reason, it is quite easy to understand that we need another table having only one run_before, called RBT1.

<table>
<thead>
<tr>
<th>zerosLeft</th>
<th>RBT1</th>
<th>RBT2</th>
<th>RBT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>Total</td>
<td>804</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the multiple run_before process, loop_count is initialized as Total coefficient - 1. This loop_count indicates how many coefficients have not yet been decoded. The number of decoded run_befores (# run_before) is the same as the number of decoded coefficients. In the next iteration loop_count is decreased by # run_before. If loop_count is greater than or equal to 3, equals 2, or equals 1, then RBT3, RBT2, or RBT1 will be selected, respectively.

The maximum coded bit length for various combinations of multiple run_befores varies, and it is also a function of zerosLeft. In other words, the maximum coded bit length determines the table size. Figure 8 shows how the table size can be reduced.

The following discussion explains how the size of the tables can be reduced for decoding multiple run_befores (please refer to Fig. 8).

1. This is an example of decoding the “00001100” run_before bit stream when zerosLeft is 6. With table access the 8-bit input can be decoded to have a

![Diagram](https://www.spiedigitallibrary.org/journals/Optical-Engineering)
sequence of 1, 2, and 3 run_befores. We need to use RBT3 to simultaneously decode three run_befores. After analyzing all RBT3 cases, the authors found that the maximum bit stream is 8, so the table size can be 256.

2. This is two examples of decoding “000000” and “001000” when zerosLeft is 6. They are decoded to have a sequence of (1,5) and (2,4) run_befores, respectively. In these cases, RBT3 is needed to simultaneously decode two run_befores. After analyzing all RBT2 cases, it is found that the maximum bit stream is 6. This means the table size can be 64.

3. This shows the case of RBT1, for which the maximum bit stream is 3. This means the table size can be 8.

We further reduced the table for the case of Fig. 8(a). As in the above example, we only considered zerosLeft equals 6. If the zerosLefts are 5, 4, 3, 2, and 1, the maximum bit streams become 8, 7, 7, 6, 4, and 3 bits, respectively. This tells us the table size can be reduced further.

Similarly, we can summarize the necessary memory table sizes for various cases as shown in Table 1.

4 Simulation Results
Simulations are used to compare the proposed multiple run_befores decoding and conventional single decoding. The gain is calculated for various test sequences and quantization parameter (QP)s. Simulation conditions are shown in Table 2.

<table>
<thead>
<tr>
<th>QP</th>
<th>Foreman CIF (%)</th>
<th>Mobile CIF (%)</th>
<th>Paris CIF (%)</th>
<th>Tempete CIF (%)</th>
<th>City 4CIF (%)</th>
<th>Crew 4CIF (%)</th>
<th>Harbour 4CIF (%)</th>
<th>Soccer 4CIF (%)</th>
<th>Pedest HD (%)</th>
<th>Rushhour HD (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>151.71</td>
<td>163.98</td>
<td>186.72</td>
<td>179.76</td>
<td>156.76</td>
<td>164.32</td>
<td>158.54</td>
<td>173.68</td>
<td>163.92</td>
<td>148.07</td>
</tr>
<tr>
<td>37</td>
<td>151.48</td>
<td>160.05</td>
<td>170.19</td>
<td>175.78</td>
<td>157.98</td>
<td>165.91</td>
<td>153.97</td>
<td>172.76</td>
<td>165.13</td>
<td>151.34</td>
</tr>
</tbody>
</table>

Fig. 10 The trend of the total speed-up factor (TSF) according to QPs for two typical video sequences.
4.1 Comparison of the Number of Table Accesses

We define the proposed decoding method as multiple run_before (MRB) and the conventional decoding method as single run_before (SRB). The number of table accesses for MRB and SRB is defined as \( nTBL(MRB) \) and \( nTBL(SRB) \), as illustrated in Fig. 9. In the case of MRB, two decoding steps are needed as shown in the bold line boxes. Here, run before 1, 2, 0 can be decoded in the first decoding and run before 1 can be decoded in the second decoding. In the case of SRB, run before 1, 2, 0 and 1 are decoded with four iterations.

The processing gain of the proposed MRB compared to SRB can be defined as

\[
\text{Gain} = \frac{nTBL(SRB)}{nTBL(MRB)}.
\]

Table 4 shows the gain for all possible cases of one 4 × 4 block.

4.2 Ideal Speed-Up Gain of the Proposed Method

Simulations using the baseline profile for the common intermediate format (CIF), 4CIF, and high definition (HD) sequences are performed. In MRB, there are nine distinct cases of gain in the decoding. Table 5 presents the simulation results and the probability of the gain.

We applied several QPs, such as 22, 27, 32, and 37. To simplify the presentation, only one experimental result of QP 22 is summarized in Table 5. In this result, gain 3 appears more frequent with less moving or in lower resolution (CIF) video sequences.

This paper is proposing a new novel algorithm of the multiple run_before decoding for hardware implementation. Therefore, note that comparing with the software implementation efficiency is not necessary.

We define the total speed-up factor as the sum of the product of the gain and the probability of the case. This reflects the proposed algorithmic efficiency compared to the single run_before decoding. Simulation is performed for entire video frames, and the results are summarized in Table 6. This table shows the experimental results of QP 22, 27, 32, and 37.

From Table 4 it can be seen that the proposed method shows higher total speed-up factor (TSF) in lower resolutions or with smaller QP. Figure 10 shows that TSF changes according to the QP. For the mobile and Paris CIF sequences, the proposed algorithm produces more than twice the TSF at QP22.

4.3 Comparison of the Implementation with Wen

For the implementation point of view, it is necessary to compare the hardware size and the operating frequency of the design. Wen\textsuperscript{\textregistered} method is compared because it uses H.264 and proposes efficient decoding of run_before. Both are coded in Verilog HDL and synthesized for a field programmable gate array (FPGA, Xilinx Virtex II XC2V4000-6BF957). The results of synthesis and comparison are shown in Table 6. The proposed method has achieved 6.9 ns maximum period. The critical path delay of the proposed design is about 5.8 times smaller than Wen\textsuperscript{\textregistered} The number of required configurable logic block (CLB) in Wen\textsuperscript{\textregistered} is 654, but we use only 374. The total execution time is equal to the maximum period multiplied by the required decoding cycles. The proposed method’s total execution time gain is about 3.89 compared to Wen\textsuperscript{\textregistered}.

Fewer table accesses means lower power consumption in implemented digital systems. The proposed method is useful for mobile devices such as cell phones, personal digital assistants, and smart phones.

5 Conclusions

CAVLC is used for run_before coding of the quantized residual coefficients in H.264/AVC. The elements of the CAVLC stream are TotalCoef (coeff_token), TrailingOnes (coeff_token), trailing.ones_sign_flag, level, total_zeros, and run_before. Efficient implementation of CAVLC is important for mobile devices.

Prior research has been performed to improve the decoding of CAVLC. Coeff_token decoding has been improved by using new coeff_token VLD and total_zeros decoding has been improved and memory access reduced by 80–90%. Trailing ones_sign_flag is simply assigned as “0” or “1.” Thus, the remaining targets for improving CAVLC decoding are level and run_before. As the level decoding is quite complex, the authors leave this topic for future research.
Here, our target of research is to improve the run_before decoding. Note that Coeff_token and Total_zeros appear only once for a $4 \times 4$ block. But run_before usually appears many times for one $4 \times 4$ block; therefore, it is quite important to improve this run-before decoding for overall efficiency of the CAVLC decoding. Previous research on run_before decoding used an arithmetic operation to reduce the table accesses.[3] But this is efficient when GPP or DSP is used. Our aim is to develop a new algorithm for improving run_before decoding for hardware implementation. In this paper, we applied simultaneous decoding of up to three run_befores. In the simulation results, the Total Speed-up Factor is 144.9% $\sim$ 205.8% for various sequences and QPs.

For the implementation point of view, Verilog HDL is used for hardware coding and synthesized for an FPGA (Xilinx Virtex II XC2V4000BF957). The proposed method is using only about 57% of the CLB count, about 5.79 times faster for the critical delay. Total decoding time gain of the proposed method is about 3.89 times faster for decoding the Foreman video sequence. The proposed method has less memory access and smaller hardware size. It means less power consumption and is good for mobile device application.

Acknowledgments

This work was supported by a Korea Research Foundation Grant funded by the Korean government (MOEHRD, Basic Research Promotion Fund, KRF-2006-511-D00265).

References