High-Performance VLSI Architecture of H.264/AVC CAVLD by Parallel Run_before Estimation Algorithm

JONGWOO BAE1 AND JINSOO CHO2,*
1Department of Information and Communication Engineering
Myongji University
Gyeonggi-Do, 449-728 Korea
2Department of Computer Engineering
Gachon University
Gyeonggi-Do, 461-701 Korea

A high-performance VLSI architecture for the H.264/AVC context-adaptive variable-length decoder (CAVLD) is proposed in order to reduce the computation time. The overall computation is pipelined, and a parallel processing is employed for high performance. For the run_before computation, the values of input symbols are estimated in parallel to check if their computation can be skipped in advance. Experimental results show that the performance of run_before is improved by 134% on average when four symbols are estimated in parallel, while the area of the VLSI implementation is only increased by 12% compared to a sequential method. The degree of parallelism employed for the estimation module is 4, and it can be changed easily. H.264/AVC is an essential technology for the multimedia engines of many consumer electronics applications, such as D-TVs and mobile devices. The proposed method contributes to the performance improvement of those applications.

Keywords: H.264/AVC, CAVLD, run_before, skip estimation, VLSI design

1. INTRODUCTION

H.264/AVC has replaced MPEG-2, and it is widely used for terrestrial, cable, and internet broadcasting in many countries. The frame rates and resolutions for TVs and mobile devices have been increasing rapidly, and interest in 3D movies has also developed recently. These factors have led to an explosive boost in the required amount of video data. Therefore, the performance of H.264/AVC must improve to meet the new requirements. Due to the sequential bottleneck of the entropy decoder, it is hard to improve the performance of H.264/AVC and other video codecs. Especially for real-time applications with a low-latency requirement, sufficient buffering is not guaranteed, and the bit rates can suddenly escalate sharply, which may cause the decoding to fail.

H.264/AVC CAVLD consists of five steps: coeff_token, trailing_ones, level, total_zeroes, and run_before [1]. Among these steps, level and run_before computations are the most time-consuming. In general, if the number of levels is large in a macroblock, so is the number of run_befores; they are proportionally related. For the encoding of complex video scenes, both the computational load and the bit-rate of the encoded stream are increased. This leads to an increased number of macroblocks with a large number of
levels and run_befores. In most applications, the decoder can tolerate the temporary increase of computational load by buffering if the additional stress does not continue for too long. However, for real-time applications with a low-latency condition, the number of levels is increased in a macroblock, as with the number of zeroes of run_before values. The sudden escalation of the computational load is difficult for the decoder to handle. In order to overcome such problems, this paper proposes a method to improve the performance of CAVLD for H.264/AVC.

Various methods have been presented for the computation of CAVLD. A method to modify the arithmetic computations for run_before was proposed by Chang [2]. He observed the regularity in the run_before look-up table (LUT) and changed the look-up table into four-step arithmetic calculations, which reduced computation time and power consumption. However, this method is basically sequential, and computation time is much slower than it would be for parallel methods. Nikara [3] proposed a parallel architecture for multi-symbol, variable-length decoding. An advanced memory reference technique was employed, and the on-chip memory was reduced. The proposed architecture offers performance that is proportional to the degree of parallelism provided; unfortunately, the price to pay for the increased parallelism is very high, as the area of run_before substantially increases. For this reason, a design with a high degree of parallelism seems to be impracticable. The method of skipping run_before with a value of zero was proposed by Tsa [4]. However, this method is also sequential, and the detailed architecture and experimental results were not clearly provided. A lot of research studies have been conducted on H.264/AVC recently [5-10].

Herein, we introduce a new algorithm and VLSI architecture to reduce the run_before computation time drastically at the cost of small area addition. The first four steps of the CAVLD explained before are implemented by employing the best known technology so far, which includes the parallel processing and pipelining techniques, introduced briefly at the beginning of the proposed work. We focus on the improvement of the final step of run_before computation. Since run_before is one of the most time-consuming steps of CAVLD, our work is an important contribution toward the improvement of CAVLD.

This paper is organized as follows. In section 2, the related works are explained. In section 3, the proposed architecture is explained in detail. In section 4, experimental results are provided. Section 5 concludes this paper.

2. RELATED WORKS

The flowchart of CAVLD for the hardware implementation is shown in Fig. 1. It consists of five computation steps: coeff_token, trailing_ones, level, total_zeroes, and run_before. Each step is generally executed sequentially.

Step 1: Coeff_token

The numbers of the total coefficients and the trailing ones are computed first, according to the look-up table and algorithm in [1]. This can be executed in parallel to the computation of the previous macroblock, thereby saving computation time.
Step 2: Trailing ones

The trailing ones are computed next. They are either 1 or \(-1\), based on the corresponding input bit of zero or 1. The computation of trailing ones can be computed immediately by a simple logic design.

Step 3: Level

The remaining non-zero symbols are decoded next. The level value is computed after the computation of level prefix, level suffix, and suffix length. The level computation is the most time-consuming, as well as the run before computation. It is possible to compute two level symbols in parallel according to [5], and the computation speed can be doubled. The level decoding of one symbol takes more than one cycle for real-time HD video decoding. We can speed up the computation by pipelining the level computation into two stages, as well as the parallel computation.

Step 4: Total zeroes

The number of total zeroes is computed next, according to the look-up table in [1]. The computation of total zeroes can be overlapped with the final stage of the level computation, thereby saving computation time.

Step 5: Run before

The final step of the CAVLD is run before. It is the most time-consuming, except for level computation. The run before computation can be implemented in parallel. Various methods have been developed for this step, and we introduce a new method in
The data read and write procedure for CAVLD can be overlapped with the above five steps of CAVLD to further save computation time. By combining all the methods explained above, we can implement a very fast CAVLD with a small size. In this paper, we focus on a new method for run_before, employing a parallel estimation, and further improve the performance of CAVLD.

In a recent work [6], we introduced a run_before estimation method that works as follows. The run_before computation values of the next symbols are estimated in parallel with the computation for the current symbol of a macroblock. Fig. 2 shows the architecture of the method. The block labeled run_before core computes the run_before of the first input symbol. The estimation module shown as the dotted box computes the estimation for the next symbols, with various starting bit positions. If we compute the run_before computation value of the next symbol together with the current symbol, there are two uncertainty factors – the starting bit position and the zeroes left value of the next symbol – as the computation of the first symbol is not finished yet. Note that the zeroes left are defined as the number of remaining zeroes to insert after the current level is decoded [1]. In order to consider every possible combination of input symbols, the design of a parallel run_before estimation module is very complicated, albeit much simpler than other parallel design of a general run_before computation, such as Nikara’s architecture [3]. The real parallel run_before computation has multiple LUTs for various starting bit positions, and zeroes left values for the next symbols. This leads to a large design with an accompanying long delay due to many LUTs and multiplexor layers. The large parallel-computation module is replaced with a small estimation module.

![Diagram of run_before with parallel estimation](image)

**Fig. 2. Run_before with parallel estimation in [6].**

### 3. PROPOSED ARCHITECTURE

In this paper, we further improve the performance of run_before by estimating the run_before computation values of the input symbols in parallel while the estimated run values are zeroes. If the estimation fails, the real run value is computed by another module.
• We only check if the run_before of the next symbol is zero or not.
• If it is zero, we set the run_before value to zero and move to the next symbol.
• We repeat this until there is a non-zero run_before value, then stop.
• The non-zero run_before value is computed next.

To continue, the check of the run_before with zero value is simple, because the input symbol is either ‘1’, ‘11’, or ‘111’, depending on the zeroes left, as shown in Fig. 3. Note that the symbol is ‘1’ if the zeroes left value is 1 or 2. The symbol is ‘11’ if the zeroes left value is from 3 to 6. The symbol is ‘111’ if the zeroes left value is greater than 6. Fig. 4 shows the flowchart of the proposed method.

<table>
<thead>
<tr>
<th>run_before</th>
<th>zeroes left</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>&gt; 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
</tbody>
</table>

Fig. 3. Run_before decoding table.

The operation of the flowchart is explained as follows. There are three different cases corresponding to each column of the flowchart. Each estimation starts at a different bit (bit-13 for Case 1, bit-12 for Case 2, and bit-11 for Case 3). The size of the input bitstream register implemented by a barrel shifter is assumed as 15 bits, and bit-14 is the most significant bit.

The run_before core module does not execute before the estimation module. The estimation is performed until there is no skipping left, and the run_before core module works on the non-skipping run. This way, we can eliminate the wasted cycle of the run_before core by running it at the front (before the estimation). The run_before core does not support parallel execution, and one cycle is consumed for one run symbol only, whether it is a skip or not. In addition to this, the whole architecture of the estimation becomes much simpler than the previous one because of the uncertainty of the zeroes left value by running the run_before core module beforehand. That is, the estimation module only depends on the zeroes left value given, and the estimation always starts at the first bit position, bit-14. The following cases provide a more detailed explanation.
Case 1: The leftmost column of the flowchart in Fig. 2 shows the case where the zeroes left value is 1 or 2. The estimation starts at bit-14. According to the run_before decoding table in Fig. 3, the number of used bits is 1 only when the zeroes left value is 1 or 2. If input bit is ‘1’, the run_before value is zero, the estimation result is skip, and the estimation of the next symbol continues. If input bit is ‘0’, then the estimation stops, and the symbol with starting bit ‘0’ is immediately computed by the run_before core module.

In the flowchart in Fig. 2, the estimation of multiple symbols is shown as a sequential loop. When the estimation result is skip and the count is less than MAX, it goes back to the next input bit comparison. For parallel implementation, using the estimation of 4 symbols as an example, we check to see if the input bits are either ‘1’, ‘11’, ‘111’, or ‘1111’ at the same time. When the comparison result is false for the first time, the number of used input bits and the number of computed symbols of the position are returned. For example, if the comparison result is false at the position of ‘111’, there are two used bits and two computed symbols. If all the comparison results are true, 4 symbols are estimated to have run_before values of zero, and there are 4 used bits and 4 computed symbols. This case is shown as block 1 of the estimation module in Fig. 5, and the detailed architecture of block 1 is shown in Fig. 6.

Case 2: The middle column of the flowchart in Fig. 4 shows the case where the zeroes left value is from 3 to 6. The estimation starts at bit-14. According to the run_before decoding table in Fig. 3, the number of used bits is 2 only when the zeroes left value is
from 3 to 6. If input bits are ‘11’, the run_before value is zero, the estimation result is skip, and the estimation of the next symbol continues. If input bit is other than ‘11’, then the estimation stops, and the symbol is immediately computed by the run_before core module.

In the flowchart in Fig. 4, the estimation of multiple symbols is shown as a sequential loop. When the estimation result is skip and the count is less than MAX, it goes back to the next input bit comparison. Note that the number of used bits is 2 in this case. For parallel implementation, using the estimation of 4 symbols as an example, we check to see if the input bits are either ‘11’, ‘1111’, ‘111111’, or ‘11111111’ at the same time. When the comparison result is false for the first time, the number of used input bits and the number of computed symbols of the position are returned. This case is shown as block 2 in Fig. 5. The detailed architecture of block 2 is similar to that of block 1.
Case 3: This case shows the drastic improvement of the proposed algorithm compared to the previous one. In the previous method, we needed to check the 9 potential bit positions, and therefore 9 estimation blocks were needed. We also showed that this could be avoided by checking the 000 value at first. However, the proposed algorithm in this paper does not require checking the 9 potential bit positions, because it does not have the preceding run_before core module, as shown in Fig. 5. It only depends on the zeroes left value from the total zeroes step. Case 3 is similar to the other two cases now. The rightmost column of the flowchart in Fig. 4 shows the case where the zeroes left value is greater than 6. The estimation starts at bit-14. According to the run_before decoding table in Table 1, the number of used bits is 3 only when the zeroes left value is greater than 6. If input bits are ‘111’, the run_before value is zero, the estimation result is skip, and the estimation of the next symbol continues. If input bit is other than ‘111’, then the estimation stops, and the symbol is immediately computed by the run_before core module.

In the flowchart in Fig. 4, the estimation of multiple symbols is shown as a sequential loop. When the estimation result is skip and the count is less than MAX, it goes back to the next input bit comparison. Note that the number of used bits is 3 in this case. For parallel implementation, using the estimation of 4 symbols as an example, we check to see if the input bits are either ‘111’, ‘111111’, ‘11111111’, or ‘11111111111’ at the same time. When the comparison result is false for the first time, the number of used input bits and the number of computed symbols of the position are returned. This case is shown as block 3 in Fig. 5. The detailed architecture of block 3 is similar to those of block 1 and block 2. Unlike [6], we do not check the case of input bits equal to 000 for the run_before core module. Since the run_before core module is located in front of the estimation module, we used to have all the complication for the zeroes left value greater than 6. Now, the run_before core module is not at the front, and therefore, we do not have to worry about the case starting with bits 000. By skipping estimation when the input bits of the run_before core module are 000, we lose performance to a degree. In the proposed algorithm, however, we are able to improve performance with a simpler architecture without skipping the 000.

4. EXPERIMENTAL RESULTS

To assess the performance of the proposed algorithm and VLSI architecture, we implemented it in Verilog HDL and synthesized it in TSMC LVT90 process. The result was compared with four previous results: a sequential architecture, a parallel architecture, estimation architecture-1, and estimation architecture-2. The performance and size are the major comparison factors.

A sequential architecture is a direct implementation of the official run_before specification in [1]. It is the architecture of run_before core only; therefore, the size is very small. We use the size and performance of the sequential design as the reference of comparison.

The parallel architecture of [3] is shown in Fig. 7. It has multiple run_before cores, proportional to the degree of parallelism. Due to various starting points, multiple LUTs are used for each run_before core. The performance of this method is five times faster than the sequential one for the degree of parallelism 4. However, the resulting size is
very large (3640% larger than the sequential one).

The estimation architecture-1 of [6] is our previous estimation architecture with third estimation module to estimate all the possible 9 starting points in block 3, as shown in Fig. 8. The performance of this method is 2.34 times faster than the sequential one for degree of parallelism 4. It is much smaller than the complete parallel architecture, but still larger than the estimation architecture-2 (53% larger than the sequential one).

![Fig. 7. Parallel run_before architecture [3].](image1)

![Fig. 8. Estimation-2 architecture for run_before [6].](image2)

Estimation architecture-2 of [6] is the architecture shown in Fig. 2. It is slightly (9.8%) larger than the proposed architecture, because of the more complicated estimation logic. The performance is worse than the proposed one by 39.3%, because the preceding
run_before core consumes one cycle before the estimation module all the time. In addition, the estimation is skipped when the run_before core has an input starting with 000.

The proposed architecture is the smallest, except for the sequential one, and yields the best performance after the completely parallel architecture. Considering the slow performance of the sequential architecture and the extremely large size of the complete parallel architecture, the proposed architecture is the best choice in terms of both performance and size (134% faster and 12% larger than the sequential one for the degree of parallelism 4).

Table 1 summarizes the comparison of the five architectures in terms of size and performance relative to those of a sequential method. The degree of parallelism employed for the estimation module is 4, which can be changed easily.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Size</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential [1]</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Parallel [3]</td>
<td>37.4</td>
<td>5</td>
</tr>
<tr>
<td>Estimation-1 [6]</td>
<td>1.53</td>
<td>2.34</td>
</tr>
<tr>
<td>Estimation-2 [6]</td>
<td>1.23</td>
<td>1.68</td>
</tr>
<tr>
<td>Proposed</td>
<td>1.12</td>
<td>2.34</td>
</tr>
</tbody>
</table>

5. CONCLUSION

In this paper, we presented a high-performance algorithm and a VLSI architecture to improve the performance of run_before computation for a H.264/AVC CAVLD module. The proposed method employs the parallel estimation of skipping symbols until the skipping fails; then, the real run value is computed. Using this method, we improved the performance by 134%, and the area of run_before increased by 12% by adding the estimation module compared with the sequential implementation for the case of the degree of parallelism 4. Various methods were compared with the proposed one, and the experimental results showed that the proposed method was the best choice, considering the performance and size. The proposed method can be used for a number of modern consumer electronics applications, such as D-TVs and mobile devices that require H.264/AVC of very high performance.

REFERENCES


Jongwoo Bae received a B.S. degree in Control and Instrumentation from Seoul National University, Korea, in 1988, and M.S. degree and Ph.D. in Computer Engineering from the University of Southern California in 1989 and 1996, respectively. Dr. Bae worked as a hardware and software engineer for Actel, Avanti, and Pulsent Co. in San Jose, CA, from 1996 to 2002. He worked as a principal engineer in the Samsung Electronics System LSI Division from 2003 to 2007. He is currently an Associate Professor in the Department of Information and Communications Engineering, College of Engineering at Myongji University, Korea. His research interests include VLSI design, image/video processing, digital TV, and processors.

Jinsoo Cho received a B.S. degree in Electronic Engineering from Inha University, Korea in 1994, an M.S. degree in Electrical Engineering from Columbia University in 1998, and a Ph.D. in Electrical and Computer Engineering from the Georgia Institute of Technology in 2003. From 2004 to 2006, Dr. Cho worked as a senior engineer on the D-TV Development Team of the System LSI Division at Samsung Electronics in Korea. He is currently an Associate Professor in the Department of Computer Engineering, College of Information Technology at Gachon University in Korea. His research interests include image/video processing, digital TV, image/video enhancement, and image/video compression.