A Memory Efficient Fine Grain Scalability Coefficient Encoding Method for H.264/AVC Scalable Video Extension

Meng-Wei Shen, Gwo-Long Li, and Tian-Sheuan Chang
Institute of Electronics
National Chiao-Tung University
Hsinchu, Taiwan
mwshen.ee96g@nctu.edu.tw, glli.ee95g@nctu.edu.tw, tschang@twins.ee.nctu.edu.tw

Abstract—In this paper, a memory efficient Fine Grain Scalability (FGS) coefficient encoding method is proposed to reduce the external memory access requirement. In the H.264/AVC Scalable Video Extension, the FGS coefficients encoding is frame based. However, the frame based mechanism results in the difficulty of hardware implementation due to large internal memory requirements and external memory accesses. Therefore, a non-uniform memory size design which can achieve low external memory access is proposed to realize the macroblock based FGS coefficients encoding. Compared to previous work, our proposed method can save at least 38KB external memory accesses per frame in average.

I. INTRODUCTION

For bandwidth constraint networks, the available bandwidth for transmitting video data is not constant but varies over time. Traditional video coding standards optimize the video quality at a given bit-rate, but it can not deal with bandwidth variation problem effectively. As an extension of H.264/AVC, Scalable Video Coding (SVC) addresses the bandwidth variation problem in an efficient way. It provides three scalabilities: spatial scalability, temporal scalability and quality scalability [1]. Spatial scalability supports multiple display resolutions, temporal scalability supports different frame rate, and SNR scalability can be realized by FGS.

However, the FGS coefficient encoding in SVC is performed by frame based instead of MB based. This mechanism causes the FGS hardware implementation more difficult. To deal with this problem, Chen et al. [2] proposed a macroblock based FGS coefficients encoding method by adopting the bucket scan method. Nevertheless, the bucket size dominates the overall performance of FGS coefficients encoding. Therefore, the trade-off between the bucket size and external memory access becomes the main issue in designing the FGS coefficients encoder.

In this paper, we first analyze the distribution of the coefficients in FGS layer. Afterward, a non-uniform memory method is proposed to fully utilize the statistical property which conducted from analysis section for reducing external memory accesses.

This paper is organized as follows: First, the FGS coefficients coding is briefly described in Section II. Section III presents our motivation. Proposed methods are provided in Section IV. Section V shows the simulation results and conclusions are given in Section VI.

II. OVERVIEW OF FGS COEFFICIENT ENCODING

In FGS enhancement layer, the coefficients before entropy encoding are processed in two types of passes: significant pass and refinement pass [3]. Fig. 1 shows two types of passes for encoding FGS coefficients. The significant pass encodes a FGS enhancement layer block from a FGS base layer block which has zero value. The refinement pass encodes a FGS enhancement layer block from a nonzero FGS value at the base layer block.

In order to arbitrarily truncate the bitstream, FGS scans every macroblock in one frame by turns [4]. Fig. 2 shows an example of FGS scan order with only four blocks for simplicity. The table at the upper part is the coefficients in zigzag scan order and the bottom part is the scan order of this example. All blocks in the frame are scanned in every round. The scan principle is that only coefficients starting at a scan position equal to the round number are coded. If the start position is a refinement coefficient, only encodes this coefficient. If the start position is a significance coefficient, encodes all significance coefficients but skips the refinement coefficients in the path until a nonzero significance coefficient
is reached. For example, at round 0, we scan \((0 \ 0 \ 0 \ 1)\) for block 0, C for block 1, I for block 2, and \((0 \ 0 \ 0 \ 0 \ 0 \ 0)\) for block 3. At round 1, A for block 0, \((0 \ 1)\) for block 1, I for block 2, and no coefficient for block 3 are scanned. At round 2, only scan I for block 2 is needed. Repeat the scan principle until there is no coefficient needs to be coded. Finally, the coding order is \((0 \ 0 \ 0 \ 1), C, I, (0 \ 0 \ 0 \ 0 \ 0 \ 1), A, (0 \ 1), I, I, B, (0 \ 1), I, ..., I, O\).

\[
\begin{array}{c|cccccccc}
\text{Block} & \text{0} & \text{A} & \text{B} & \text{0} & \text{1} & \text{I} & \text{0} & \text{0} & \text{0} \\
\text{Block} & \text{1} & \text{C} & \text{0} & \text{1} & \text{0} & \text{I} & \text{0} & \text{0} & \text{0} \\
\text{Block} & \text{2} & \text{1} & \text{1} & \text{1} & \text{1} & \text{0} & \text{0} & \text{T} & \text{G} \\
\text{Block} & \text{3} & \text{0} & \text{0} & \text{0} & \text{J} & \text{0} & \text{K} & \text{0} & \text{1} \\
\end{array}
\]

Round 0 \(= \{0;0;0;1\}; 1; (C); 2; (I); 3; (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)\)
Round 1 \(= \{0;A;1;0;1;2;I\}\)
Round 2 \(= \{2;I\}\)
Round 3 \(= \{0;B;1;0;1;2;3;I\}\)
Round 4 \(= \{2;E;O;B;3;I\}\)
Round 5 \(= \{I;E;O;B;3;I\}\)
Round 6 \(= \{0;I\}\)
Round 7 \(= \{0;E;O;B;2;E\}\)
Round 8 \(= \{2;F;E;2;3;0;I\}\)
Round 9 \(= \{2;G\}\)
Round 10 \(= \{3;F;2\}\)
Round 11 \(= \{2;D;B;3;M\}\)
Round 12 \(= \{3;N\}\)
Round 13 \(= \{I;D;3;E;O;B\}\)
Round 14 \(= \{2;I\}\)
Round 15 \(= \{3;O\}\)

Fig. 2. Example of FGS coefficients coding order

To accomplish the FGS coefficients coding described above, traditional manner has to load coefficients of entire frame into internal memory. Take 4CIF (704x576) as example, internal memory needs to allocate about 400K to store whole frame. If frame size is bigger than 4CIF, such as HD 720p (1280x720) and 1080p (1920x1080), it takes more internal memory requirements. This internal memory requirement is unlikely for hardware implementation.

Generally, the hardware design of video coding is based on macroblock level. In order to make the FGS hardware design practicable, Chen et al. [2] proposed a scan bucket algorithm which moves FGS scan from frame level to macroblock level. Traditional FGS coding order requires many scans through all blocks in the whole frame. However, the proposed method by [2] only loads a macroblock one time and arranges the coefficients to be coded in advance. That is, when one block is chosen, the coefficients in this block are classified and then put them into corresponding bucket. Fig. 3 shows an example of this method. If block 0 is scanned, storing \((0 \ 0 \ 0 \ 1)\) into bucket 0, A into bucket 1, B into bucket 3, I into bucket 6, and # (end of block) into bucket 7. Afterward, other blocks are processed in turns. When the data in the bucket is full, the coefficients are stored into external memory. By using this method, FGS scan can change to macroblock level rather than frame level. It only needs to load 256 byte (1 macroblock size) to internal memory instead of loading entire frame (400K for 4CIF) at one time.

III. MOTIVATION

The scan bucket algorithm provides a new way to move FGS scan from frame level to macroblock level. Only with some extra internal memory (bucket memory), the size of internal memory is saved from frame data to macroblock data.

However, the probability of FGS coefficients in enhancement layer equal to zero may be high. This may cause internal memory waste. Fig. 4 shows an example to demonstrate this situation. After the scan bucket algorithm, the bucket distribution is like Fig. 5. It can be observed that the bucket with small index contains most of coefficients, but other buckets only have few coefficients inside. If the bucket size is set to a small value, like 1, total internal bucket memory will be 6 bytes. Although this setting saves internal memory, it increases the external memory access by 9 since there are 9 coefficients should be stored into external memory. If the bucket size is set to a large number, like 6, only the bucket 0 has to send 3 coefficients to the external memory. However, this setting wastes too much internal memory since there are 27 internal memory spaces are unused. This is a trade-off between bucket size (internal memory) and external memory access.

IV. PROPOSED METHOD

In this section, we analyze the distributive characteristic of the coefficients for FGS. According to the statistical results, we propose our method in the second part.

A. Statistic

In enhancement FGS layer, coefficients are coded by 4x4 or 8x8 transform block size. Five different sequences with three spatial layers: QCIF, CIF, and 4CIF are analyzed. We set
QP=28 for FGS base layer, QP=16 for FGS enhancement layer and encode 99 frames. According to statistical results, in average, 98.84% and 1.16% of macroblocks are coded by 4x4 and 8x8 with size of QCIF, respectively. For the CIF sizes, 97.97% and 2.03% of macroblocks are respectively coded by 4x4 and 8x8. Furthermore, 94.91% and 5.09% of macroblocks are coded by 4x4 and 8x8 for 4CIF size, respectively. The details are summarized in Table I.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>QCIF (4x4) (%)</th>
<th>QCIF (8x8) (%)</th>
<th>CIF (4x4) (%)</th>
<th>CIF (8x8) (%)</th>
<th>4CIF (4x4) (%)</th>
<th>4CIF (8x8) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreman</td>
<td>95.84</td>
<td>4.16</td>
<td>93.89</td>
<td>6.11</td>
<td>91.11</td>
<td>8.89</td>
</tr>
<tr>
<td>Stefan</td>
<td>99.60</td>
<td>0.40</td>
<td>99.30</td>
<td>0.70</td>
<td>95.30</td>
<td>4.70</td>
</tr>
<tr>
<td>Football</td>
<td>99.14</td>
<td>0.86</td>
<td>97.30</td>
<td>2.70</td>
<td>89.72</td>
<td>10.28</td>
</tr>
<tr>
<td>Mobile</td>
<td>99.87</td>
<td>0.13</td>
<td>99.77</td>
<td>0.23</td>
<td>99.20</td>
<td>0.80</td>
</tr>
<tr>
<td>Weather</td>
<td>99.73</td>
<td>0.27</td>
<td>99.61</td>
<td>0.39</td>
<td>99.22</td>
<td>0.78</td>
</tr>
<tr>
<td>Average</td>
<td>98.84</td>
<td>1.16</td>
<td>97.97</td>
<td>2.03</td>
<td>94.91</td>
<td>5.09</td>
</tr>
</tbody>
</table>

From Table I, we can observe most of blocks are coded by 4x4 transform block size, which has only 16 coefficients. It means that most coefficients are stored into bucket 0 to bucket 15. Only when coefficients are coded by 8x8 transform blocks, bucket 16 to bucket 63 are used. Fig. 6 and Fig. 7 show the accumulated probabilities of each scan bucket for Foreman and Mobile. In these figures, x-axis is the bucket number, and y-axis indicates the accumulative percentage from bucket 0 to certain bucket. From these figures, we found that the first sixteen buckets contain most of coefficients. Table II. summarizes the accumulative percentage of the first sixteen buckets. On average, 95.23%, 95.81%, 90.06% of coefficients are stored in the first 16 buckets for QCIF, CIF, and 4CIF, respectively.

### Table II. Accumulative Percentage of Coefficients in the First 16 Buckets

<table>
<thead>
<tr>
<th>Sequence</th>
<th>QCIF (%)</th>
<th>CIF (%)</th>
<th>4CIF (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreman</td>
<td>88.65</td>
<td>92.36</td>
<td>90.33</td>
</tr>
<tr>
<td>Stefan</td>
<td>96.88</td>
<td>97.52</td>
<td>88.18</td>
</tr>
<tr>
<td>Football</td>
<td>94.93</td>
<td>93.66</td>
<td>83.98</td>
</tr>
<tr>
<td>Mobile</td>
<td>98.84</td>
<td>98.31</td>
<td>94.29</td>
</tr>
<tr>
<td>Weather</td>
<td>96.84</td>
<td>97.22</td>
<td>93.51</td>
</tr>
<tr>
<td>Average</td>
<td>95.23</td>
<td>95.81</td>
<td>90.06</td>
</tr>
</tbody>
</table>

### B. Proposed Method

According to the statistical results mentioned above, we observed that most of coefficients are placed in the first sixteen buckets. Therefore, we propose a non-uniform internal memory method to fully utilize this property and thus saving the external memory requirement. The main concept of our proposed method is assign larger size for bucket 0 to bucket 15, and smaller size for bucket 16 to bucket 63. Fig. 8 shows the concept of our non-uniform memory method. The required internal memory size can be calculated by the following formula

\[ A = A_1 + A_2 = \alpha \times 16 + \beta \times 48, \]

where \( A \) is the total internal memory size, \( A_1 \) refers to the internal memory size of bucket 0 to bucket 15, \( A_2 \) denotes the internal memory size of bucket 16 to bucket 63, \( \alpha \) is the size of each bucket for bucket 0 to bucket 15, and \( \beta \) is the size of each bucket for bucket 16 to bucket 63.

![Fig. 8. Proposed non-uniform memory method](image)

By using our proposed non-uniform memory design, most of coefficients can be stored in the internal memory and thus reducing the external memory access requirement.

In order to demonstrate the benefit of the proposed non-uniform memory method, we show an example for two types of scan buckets in Fig. 9 with source data shown in Fig. 3. Fig. 9(a) is the bucket usage of [2] and Fig. 9(b) shows the bucket usage of our proposal. The size of both buckets is set to the same for fair comparison. For the bucket of Fig. 9(a), if there are more than 3 coefficients for single bucket, the exceeding coefficients have to be moved out to external memory. In this case, 9, 1 and 2 coefficients corresponding to bucket 0, bucket 1, and bucket 3 are moved to external memory, respectively. Totally, this kind of method has to place 12 coefficients to external memory. However, if we use the proposed scan bucket in Fig. 9(b), there are 6 coefficients for bucket 0 and 1
coefficient for bucket 8 need to be stored into external memory. As a result, the proposed method only needs to store 7 coefficients into external memory.

Fig. 9. Illustration for two types of scan bucket

V. SIMULATION RESULTS

In this section, we perform some simulations to verify the efficiency of our proposal. Three spatial layers with size of QCIF, CIF, and 4CIF are tested. Fig. 10 shows the simulation results of five sequences (Foreman, Stefan, Football, Mobile, and Weather) with FGS base layer QP=28 and FGS enhancement layer QP=16. In these figures, x-axis is the internal memory size and y-axis indicates the amount of external memory accesses for one frame. In the case of small internal memory size such as 25K (\(\alpha =1500, \beta =30\)), our proposed non-uniform memory method can save 38KB external memory accesses per frame in average compared to previous work. In addition, for the large internal memory size, such as 68K (\(\alpha =4000, \beta =80\)), 77K of external memory accesses per frame in average can be saved. With the increase of internal memory size, the proposed method can save extremely amount of external memory accesses whatever the content of video sequence.

Fig. 10. Simulation results of five sequences

VI. CONCLUSION

In this paper, the memory problem for implementing FGS hardware is addressed and a non-uniform memory method is proposed to achieve the best trade-off between the internal memory and external memory access. Simulation results show that our proposed method can respectively save 38KB and 77KB external memory accesses per frame in average for 25KB and 68KB internal memory size when compared to previous work. In addition, with the increase of internal memory size, the significant external memory access can be saved by our proposal.

REFERENCES