JOINT ALGORITHM/CODE-LEVEL OPTIMIZATION OF H.264 VIDEO DECODER FOR MOBILE MULTIMEDIA APPLICATIONS

Ting-Yu Huang[1], Guo-An Jian[1], Jui-Chin Chu[1], Ching-Lung Su[2, 3], and Jiun-In Guo[1]

[1]Department of Computer Science and Information Engineering, National Chung Cheng University, Chia-Yi 621, Taiwan, R.O.C.

[2]Department of Electronic Engineering, National Yunlin University of Science and Technology, Yun-lin 640, Taiwan, R.O.C.

[3]SoC Technology Center, Industrial Technology Research Institute, Hsinchu, Taiwan, R.O.C.

[1]E-mail: {htyu95m, chienka, cjc, jiguo}@cs.ccu.edu.tw, [2]E-mail: kevinsu@yuntech.edu.tw

ABSTRACT

In this paper, we propose a joint algorithm/code-level optimization scheme to make it feasible to perform real-time H.264/AVC video decoding software on ARM-based platform for mobile multimedia applications. In the algorithm-level optimization, we propose various techniques like fast interpolation scheme, zero-skipping technique for texture decoding, fast boundary strength decision for in-loop filtering, and pattern matching algorithm for CAVLD. In the code-level optimization, we propose the design techniques on minimizing memory access and branch times. The experimental result shows that we have reduced the complexity of H.264 video decoder up to 93% as compared to the reference software JM9.7. The optimized H.264 video decoder can achieve the QCIF@30Hz video decoding on an ARM9 processor when operating at 120MHz clock.

Index Terms— *Video coding, Optimization methods*

1. INTRODUCTION

H.264/AVC [1] video coding standard provides up to 50% in bit-rate savings as compared to MPEG-4 advance simple profile for the same video quality, which is resulted from the fact that H.264/AVC has exploited some advanced video coding techniques, such as variable size block motion estimation and compensation, multiple reference frames prediction, enhanced entropy coding, intra prediction, in-loop filtering, and etc. However, the outstanding performance of H.264 comes along with the overhead of extremely high algorithmic complexity, which makes it too difficult to be realized for mobile multimedia applications with low-power consumption.

Fig. 1 shows the block diagram of the H.264 baseline video decoding process. It begins to decode the compressed bitstream by the entropy decoder to produce a set of quantized coefficients. After inverse quantization (IQ) and inverse transform (IT), the quantized coefficients will be translated into a series of residual blocks. Using the header information decoded from the bitstream, the H.264 decoder creates a prediction block from motion compensation (MC) or intra-prediction. The prediction block is added to the residual block, and then the result will be filtered by the de-blocking filter to produce the decoded blocks.

Driven by the progress of science and technology, multimedia applications on mobile devices are getting more and more popular [3-6]. However, the embedded processors used in the mobile devices don't have enough computation power to realize the complex H.264 video decoder in real-time. Therefore, efficient code optimizing becomes a necessity when developing the multimedia software executed on these devices. Before doing optimization on the H.264 decoder, we analyze the complexity profiling results for H.264 to identify the computation-intensive parts. We use the ARM Developer Suite (ADS) as the profiling tool to evaluate the software performance and obtain the complexity profiling results on the H.264 reference decoder (JM) and the proposed H.264 decoder. In the profiling, we adopt the encoded QCIF, 1I14P, 256Kbps Foreman sequence with 300 frames. Table 1 shows the percentage of the complexity in some significant functions in H.264. According to Table 1, the major time-consuming modules of JM9.7 decoder [2] include interpolation, in-loop deblocking filter, and entropy decoder. This result shows us a direction in reducing the computational complexity of H.264 video decoder in this paper.

In order to overcome the design challenges of realizing the complex H.264 video decoder on embedded processors, we propose a joint algorithm/code-level optimization scheme to make it feasible to realize real-time H.264/AVC video decoding on an ARM-based platform for mobile multimedia applications. In the algorithm-level optimization, we propose various techniques like fast interpolation scheme, zero-skipping technique for texture decoding, fast boundary strength decision for in-loop filtering, and pattern matching algorithm [3] for CAVLD. In the code-level optimization, we propose the design techniques of minimizing memory access and branch times. According to the proposed techniques, we have reduced the complexity of H.264 video decoder up to 93% as compared to the reference software JM9.7. The optimized H.264 video decoder can achieve the QCIF@30Hz video decoding on an ARM9 processor when operating at 120MHz clock. We also compare our decoder with existing H.264 decoder [7] whose performance is measured by Intel VTune on Intel P4 2.0GHz CPU. According to experimental results, we have about 10%~30% improvement compared with [7].

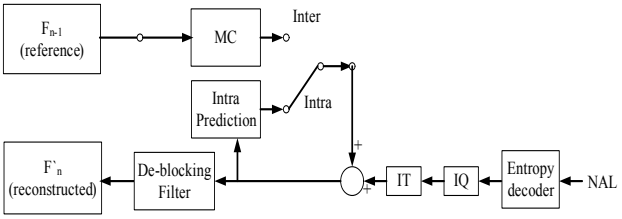

Fig. 1. Block diagram of the decoding process for H.264 baseline profile

Table 1. Complexity profiling for key functions in H.264

Function name	Occupied complexity percentage
MC (Interpolation)	37%
Deblocking Filter	20%
Entropy Decoder	18%
Inverse Transform	4%
MC (Reconstruction)	3%
Inverse Quantization	3%
Intra Prediction	3%
Others	12%

The rest of this paper is organized as follows. The proposed design techniques for optimizing H.264 video decoder will be described in Section 2. Then, the experimental results are discussed in Section 3. Finally, we conclude this paper in Section 4.

2. PROPOSED SOFTWARE OPTIMIZATION METHODOLOGY

In order to realize H.264 video decoding in real-time, we propose some optimization techniques from the aspect of joint algorithm-level/code-level optimization. In the following, we will describe the proposed techniques from the two different aspects in more details.

2.1 Code-Level Optimization

2.1.1 Memory access minimization

Memory access will cost considerable waiting cycles in executing the H.264 video decoder since the access time for memory is much longer than that for registers. Due to this reason, we have tried to minimize the frequency of memory access as most as possible. Let us take the decoding flow shown in Fig. 2(a) for example. The results of interpolation will be stored in a temporary buffer first. Then they will be added with residual data and moved into frame buffer when residual data are available. This fact suffers from additional memory access. Hence, we modify the decoding flow as shown in Fig. 2(b). The results of interpolation will directly be added with residual data and stored in frame buffer to reduce the memory access.

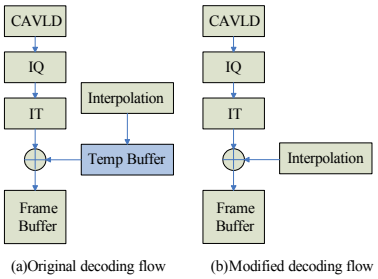

Fig. 2. Modifying H.264 decoding flow to reduce memory access

On the other hand, the reference software of H.264, i.e. JM9.7, has a bad coding style. The original C code for the function "showbits" of CAVLD is shown as below. This function only processes one bit at a time so that it needs to access external memory frequently. We modify it by partitioning it into three cases as shown in Fig. 3. Therefore, we can process several bits at one time in CAVLD, which reduces the complexity from O(n) to O(1).

```
while (numbits--){
        inf <<=1;
        inf |=   ((*(buffer)>>bitoffset--) & 0x01) ;
        if (bitoffset < 0){
            buffer++;
            bitoffset = 7;
        }
}
return inf;
```

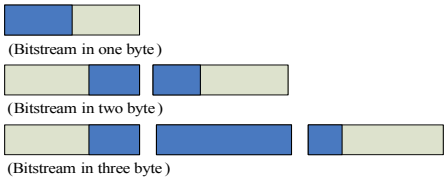

Fig. 3. Modifying the "showbits" in CAVLD for low memory access

After memory access minimization, the total computation of H.264 decoder is reduced about 10%, which shows that good software structure will improve the performance.

2.1.2 Branch minimization

In general, branch instruction will make processor stall its pipeline so that execution will be disturbed. For improving the execution efficiency, we minimize the number of branches as most as possible. For example, a 6-tap filter is adopted in luminance interpolation. Before doing filtering, the H.264 JM software checks if the reference pixels are within the memory boundaries. This operation causes a lot of redundant branches because not all interpolation operations are needed to do boundary checking. Fig. 4 shows the idea to reduce the boundary checking. We partition a macroblock into two parts. In the white part of area, all interpolation operations in this area do not need to do boundary checking. On the other hand, all interpolation operations in the black area have to check boundary first.

Using this technique has reduced about 85% and 90% branch instructions of interpolation in QCIF and CIF format videos, which amounts to totally 22% reduction in complexity of H.264 decoder.

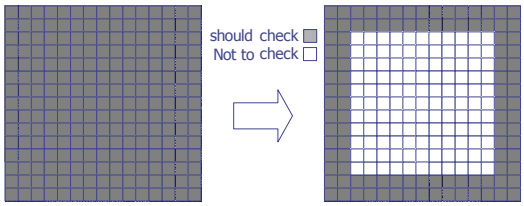

Fig. 4. Reducing boundary checking

2.2 Algorithm-Level Optimization

2.2.1 Reducing multiplications in luminance interpolation

According to the profiling result, luminance interpolation occupies a large portion, i.e. 25% of computation in H.264 video decoding. The coefficients for the 6-tap filter are defined in equation (1) and the formula of luminance interpolation is defined in equation (2).

$$\{1, -5, 20, 20, -5, 1\} \tag{1}$$

$$A - 5B + 20C + 20D - 5E + F \tag{2}$$

Fig. 5 shows the luminance interpolation for half-pixels. We observe that the 6-tap filter has symmetric coefficients so that some of multiplications can be reduced. Equation (3) shows the modified formula.

$$A + F - ((B + E) - ((C + D) << 2)) \times 5 \tag{3}$$

The original equation needs five additions and four multiplications while the modified equation (3) just needs five additions, one shift, and one multiplication. It reduces about 13% of computation in H.264 video decoder.

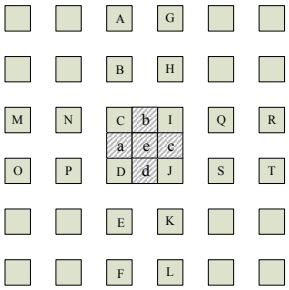

Fig. 5. Luminance interpolation with symmetry for half-pixels

2.2.2 *Simplifying chrominance interpolation*

H.264 uses a bilinear approach to do chrominance interpolation. As shown in Fig. 6, the chrominance prediction value p is calculated as weighted average of four neighboring integer samples A, B, C, and D. The formula of chrominance interpolation is defined as equation (4). Here *dx* and *dy* denote the offsets of location in fraction-sample units.

$$p = ((8-dx)(8-dy)A + dx(8-dy)B + (8-dx)dyC + dxdyD + 32 >>)6 \quad (4)$$

According to the formula, a lot of multiplications are needed in chrominance interpolation. By observing the *dx* and *dy*, we found that all the *dx* and *dy* are the same in 8×8, 8×16, 16×8 and 16 ×16 block modes. This means many redundant multiplications can be reduced. Table 2 shows the occurrence probability of the modes for Foreman video with different video resolutions.

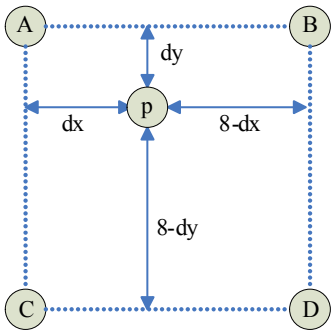

Fig. 6. The chrominance interpolation in H.264

After simplifying chrominance interpolation for the blocks larger than 8×8, we eliminate most computation in chrominance interpolation and amount to about 7% reduction of complexity in H.264.

Table 2. Probability of block type

Foreman	8×8 above	8×8 below
QCIF	87%	13%
QVGA	89%	11%
CIF	90%	10%

2.2.3 *Zero-value residual block skipping*

In the process of block reconstruction, the occurrence of zero value in the residual blocks means computation can be reduced. As shown in Table 3, we have performed some analysis for zero occurrences in inverse transform. According to the analysis, we observe the average probability for the occurrence of zero blocks is about 80%. It means about 80% of inverse transform and inverse quantization can be skipped in different sequences. Thus, we add a mechanism to detect zero blocks. As shown in Fig. 7, both inverse transform and inverse quantization can be skipped if the all-zero block is detected. Furthermore, the summation of residual blocks and predicted blocks in the final step of block reconstruction can also be skipped.

According to the simulation result, adopting the zero-value residual block skipping can totally reduce 6% of complexity.

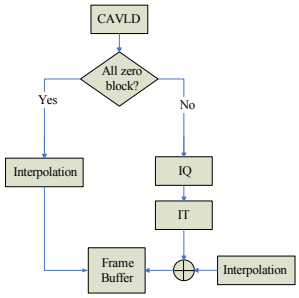

Fig. 7. Zero-value residual block skipping

Table 3. Analysis for the occurrence of all-zero blocks

Test pattern	Probability
Foreman	86%
Akiyo	80%
Coastguard	68%
Stephen	80%

2.2.4 *Deblocking stripe skipping*

In H.264, deblocking function is used to remove the block effect especially for the video encoded in low bit-rate. This process can be partitioned into two parts. The first one is getting BS (boundary strength) and the second one is to do filtering on block boundaries. In getting BS, we found that the strength of four pixels in one stripe within a 4×4 block is the same. We compute the BS at the first pixel instead of entire stripe in a block, as shown in Fig. 8.

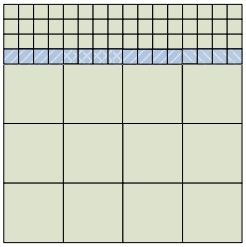
Fig. 8. The BS distribution of block

Since the BS of four pixels in one stripe will be the same, we don't have to apply deblocking to all pixels in the stripe if the BS for the first pixel is zero. Table 4 shows the analysis of BS distribution. We find the probability is about 80% when BS is zero. After deblocking stripe skipping, we totally reduce about 19% of complexity in H.264 decoder.

Table 4. The distribution of BS

Test pattern (QCIF)	BS				
	0	1	2	3	4
Foreman	77%	6.1%	8.3%	6.4%	2.2%
Akiyo	90%	1.2%	2.2%	5.1%	1.5%
Stephen	68%	8.4%	15.6%	5.8%	2.2%
Coastguard	78%	5%	9%	6%	2%

2.2.5 *Building dynamic and static table for some computation through table look-up*

In many cases, the results of computation will be the same for all frames. So we just use a look-up table to replace on-line computation, such as the computation for the position of macroblock and saturation.

The position of each macroblock will be calculated once whenever the decoder begins to decode a new frame. However, any two macroblocks in different frames that have identical number

will have the same positions. So we compute them once and build a table using table look-up to replace the computation for that position of macroblock.

Saturation is adopted in interpolation and deblocking process. Its functionality is to guarantee the pixel value will lie in between 0 and 255, as shown in equation (5). Here p and p' denote the pixel value before applying saturation and the pixel value after applying saturation, respectively.

$$p' = \max(0, \min(255, p)) \quad (5)$$

Such an operation will cost a lot of execution time due to branch instructions. Therefore, we make a statistic analysis and get a static table. Then we just use this look-up table and no longer adopt any branch instructions in doing saturation. After reducing computation by using look-up tables, we totally reduce about 13% of complexity in H.264 decoder.

3. EXPERIMENTAL RESULTS

In this section we discuss the simulation environment and experimental results by adopting the proposed design techniques. Here we use ARM926EJS simulator to evaluate performance of the proposed design. The simulator is able to report CPU core cycles and total instructions. The CPU target frequency is set at 120 MHz. Fig. 9 shows the summary of the proposed optimization method improvement and Table 5 shows the frame per second (fps) between the original version and the optimized version of H.264 decoder.

We also compare our decoder with existing H.264 decoder [7] whose performance is measured by Intel VTune on Intel P4 2.0GHz CPU. Table 6 shows that we have about 10% to 30% improvement in CIF resolution compared with [7].

In addition to the simulation results reported by the ADS, we also realize the proposed H.264 decoder on Faraday's development board called FIE8100 [8], as shown in Fig. 10. In this platform the processing core is named FA526 that is compatible with the ARM9 processing. Table 7 shows the real decoding frame rates on FIE8100.

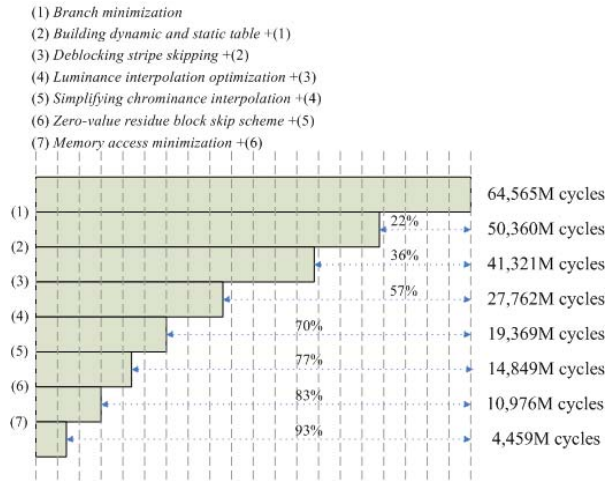

Fig.9. Summary of the improvement in the proposed algorithm

4. CONCLUSION

In this paper, we have proposed a joint algorithm/code-level optimization scheme to perform real-time H.264/AVC video decoding software on ARM-based platform for mobile multimedia applications. The proposed techniques in this paper can also be applied to other video coding standards like MPEG1/2/4, VC1, or AVS for improving its performance when realized in embedded processors. According to the proposed techniques, we have obtained more than 11 times acceleration in performance by way of optimizing the H.264 decoder in C-code level without using any hand-written assembly codes.

Table 5. Performance on ARM926EJS simulator at 120MHz

Resolution	Bitrate (kbps)	Average fps **before** optimization	Average fps **after** optimization
QCIF	256	2.66	30.50
CIF	512	0.56	8.21

Table 6. Comparison of H.264 decoder implementation

Sequence	**[7]**	**Ours**
Foreman	63 fps	75 fps
Akiyo	75 fps	100 fps
Stefan	64 fps	71 fps

Table 7. Performance on Faraday FIE8100 board at 200MHz

Test Patterns	**Bit-rate(kbps)**	**Frames per second**
Foreman(QCIF)	128	27
Akiyo(QCIF)	128	33
Stefan(QCIF)	128	25
Cars(QCIF)	128	27

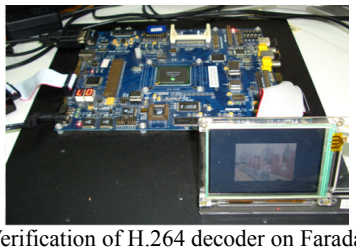

Fig.10. Verification of H.264 decoder on Faraday FIE8100 development board

REFERENCES

[1] ITU-T Recommendation H.264 & ISO/IEC 14496-10, "Advanced Video Coding for Generic Audiovisual Services", Version 4, 2005.

[2] Available via http://iphome.hhi.de/suehring/tml/

[3] S. Y. Tseng, and T. W. Hsieh, "A Pattern-search Method for H.264/AVC CAVLC Decoding", IEEE International Conference on Multimedia and Expo, pp. 1073 – 1076, Jul. 2006.

[4] V. Ramadurai, S. Jinturkar, M. Moudgill, and J. Glossner, "Implementation of H.264 decoder on Sandblaster DSP", IEEE International Conference on Multimedia and Expo, Jul. 2005.

[5] M. O. Khan, U. Khan, S. A. Rahim, S. I. Ali, "Optimization of Motion Compensation for H.264 Decoder by Pre-Calculation", 8th International Multitopic Conference, pp. 55 – 60, Dec. 2004.

[6] J. Lou, A. Jagmohan, D. He, L. Lu, and M. T. Sun, "Statistical Analysis Based H.264 High Profile Deblocking Speedup", IEEE International Symposium on Circuits and Systems, pp. 3143 – 3146, May 2007.

[7] Q. Xe, J. Liu, S. Wang, and J. Zhao," H.264/AVC baseline profile decoder optimization on independent platform", 2005 International Conference on Wireless Communications, Networking and Mobile Computing, vol. 2, pp. 1253 – 1256, Sep. 2005.

[8] Faraday Technology Corporation, FIE8100 User Guide, February 2005.