H.264 encoder speed-up via joint algorithm/code-level optimization

Yu-Lun Lai[a], Yu-Yuan Tseng[a], Chia-Wen Lin[1a], Zhi Zhou[b], and Ming-Ting Sun[b]

[a]Dept. Computer Science & Information Eng., National Chung Cheng Univ., Chiayi 621, Taiwan
[b]Dept. of Electrical Engineering, Univ. of Washington, Seattle, WA 98195, USA

ABSTRACT

The outstanding coding performance of H.264 comes with the cost of significantly higher complexity, making it too complex to be applied widely. This work aims at accelerating the H.264 encoder using joint algorithm/code-level optimization techniques so as to make it feasible to perform real-time encoding on a commercial personal computer. We propose a fast inter-mode decision scheme based on spatio-temporal information of neighboring macroblocks for the algorithm-level optimization. We use a commercial profiling tool to identify most time consuming modules and then apply several code-level optimization techniques, including frame-memory rearrangement, single-instruction-multiple-data (SIMD) implementations based on the Intel MMX/SSE2 instruction sets. Search mode reordering and early termination for variable block-size motion estimation, are then applied to speed up these time-critical modules. The simulation results show that our proposed joint optimization H.264 encoder achieves a speed-up factor of up to 18 compared to the reference encoder without introducing serious quality degradation.

Keywords: H.264, encoder optimization, mode decision, single-instruction-multiple-data (SIMD)

1. INTRODUCTION

For multimedia applications, compression is one of the most crucial issues, because raw video/audio files demand a huge amount of storage space or transmission bandwidth. Several coding standards have been defined to meet various application areas. The H.264/ AVC video coding standard[1,2] was defined by the Joint Video Team (JVT), which was formed jointly by the ISO/IEC Moving Picture Expert Group (MPEG) and the ITU-T Video Coding Expert Group (VCEG). The primary goal of H.264 is to develop a brand-new video coding technology, which is highly-efficient, network-friendly, and error-resilient for the applications ranging from mobile video to HDTV. H.264 improves the rate-distortion performance by exploiting advanced video coding technologies, such as Variable Size Block Motion Estimation (VSBME), Multiple Reference Prediction, Context-Based Adaptive Binary Arithmetic Coding (CABAC). It can save up to 50% in bit-rates as compared to MPEG-4 Advanced Simple Profile (ASP)[2]. The outstanding coding performance of H.264, however, comes with the cost of significantly higher complexity, making it too complex to be applied widely[3,4].

In H.264, the motion estimation and mode decision operations can consume as much as 66% computation of the whole encoding process[4]. In order to reduce its computational complexity, there exist many fast algorithms for implementing motion estimation and mode decision which can be used to accelerate H.264 codecs[5-15]. In fast motion estimation algorithms, some use fast search strategies that reduce the average number of search-points, such as three-step search (TSS)[5], and diamond search (DS)[6], etc. Some are based on pixel-decimation in the matching criterion[7,8]. The fast search strategies often can achieve a significant improvement in coding speed, but may also cause more quality degradation than the pixel-decimated matching criterion. Most fast mode decision algorithms usually eliminate the candidate block modes to be verified[9-15].

Since multimedia applications are getting more and more popular, most modern microprocessors have been embedded with specific multimedia instructions to speed up image and video processing programs. The single-instruction-multiple-data (SIMD) model was introduced in Intel processor. Utilizing MMX/SSE/SSE2 technologies, several data-independent instructions can be executed in parallel. In video coding applications, a large number of small-size native data type operations are performed frequently, and the operations on different data are independent to others. These features make

[1] cwlin@cs.ccu.edu.tw; phone +886 5 272 0411 ext. 33120; fax +886 5 272 0859; http://www.cs.ccu.edu.tw/~cwlin

it suitable to exploit the parallelism with the Intel SIMD technologies[16-19].

The good coding performance of H.264 is achieved at the cost of complexity, making it too complex to be applied widely. This paper focuses on the computational complexity reduction for H.264 coding standard, making it feasible to perform real-time encoding on a personal computer. Before applying optimization to the encoder, complexity analyses have to be performed first during the software optimization process so as to identify the computationally critical paths. The Intel® VTune™ Performance Analyzer[20] is used in this work as the profiling tool to evaluate the software performance and obtain the complexity profile of the reference and optimized encoders. In this paper, all the codes are compiled by the Intel® Compiler 7.1[21] to fully utilize the SIMD instruction sets. Fig. 1 shows the high-level execution-time breakdown of the H.264 JM7.3 reference encoder. In this experiment, the *Foreman* sequence (150 frames, QCIF format, IPPPPP… frame structure) is encoded on an Intel Pentium-4 2.4GHz PC with 512 MB memory under the Microsoft Windows XP. The RD optimization option in the reference encoder is turned off for the experiments. According to Fig. 1, the most time-consuming modules of JM7.3 encoder are *Motion Estimation*, *Interpolation*, *SATD*, and *DCT*. Therefore, these modules are in the top priority list to optimize.

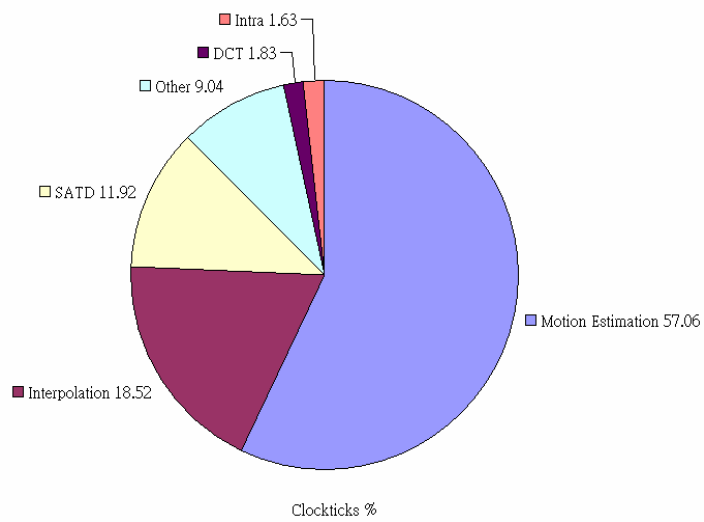


Fig. 1. High-level time breakdown of the JM7.3 encoder execution time by Intel® VTune™.

In this work, both algorithm-level optimization and code-level optimization are applied to speed up the H.264 software encoder on a commercial personal computer supporting SIMD instruction sets. We propose a fast inter mode decision scheme based on spatio-temporal information of neighboring macroblocks for the algorithm-level optimization. Besides, we also adopt a fast hexagonal-based motion estimation scheme[22-24] to reduce the computation. We then use a commercial profiling tool to identify most time consuming modules and then apply several code-level optimization techniques, including frame-memory rearrangement and SIMD implementations based on the Intel MMX/SSE/SSE2 instruction sets to further speed up these time-critical modules.

The rest of this paper is organized as follows. Section 2 addresses the proposed algorithm-level fast inter-mode decision scheme. Section 3 describes the proposed code-level optimization schemes. Experimental results and analyses are shown in Section 4. Finally the conclusion will be drawn in Section 5.

2. FAST CODING MODE DECISION USING SPATIO-TEMPORAL PREDICTIONS

To determine a block size mode used for variable-size block coding, two methods are generally used: a top-down splitting method and a bottom-up merging method. Both of the methods need to select one initial block size mode for motion prediction. For the bottom-up merging method, a smallest block size mode is chosen among available block size modes as the initial block size mode for performing the motion estimation. Conversely, for the top-down splitting method, a largest block size mode is chosen as the initial block size mode. The methods then decide whether the initial

block size mode satisfies predetermined conditions according to the motion prediction result. If so, the methods use the initial block size mode for encoding. Otherwise, the methods choose other block size modes for motion predictions and decide a best block size mode from the motion prediction results. In general, with higher bit rates, there is a better chance to use a smaller block size mode for encoding, that is, to use the bottom-up merging method. With lower bit rates, however, there is a better chance to use the top-down splitting method.

The conventional exhaustive mode decision analyzes seven possible block-size modes and selects a best one from the seven modes for encoding. The proposed method, on the other hand, need only analyze a subset of the seven modes by taking using spatio-temporal predictions from neighboring blocks such that the time for encoding can be reduced drastically. The coding modes of five neighboring blocks: the left, upper, upper left, and upper right blocks of the current block, and the block at the same location in the previous frame are used for prediction. To avoid reducing the coding performance due to improper use of spatio-temporal predictions, our method, namely, enhanced fast mode decision (EFMD), further analyzes the reliability of each predicted mode of each inter-block before using the predicted mode for encoding. If the predicted mode is reliable, the inter-block then uses the predicted mode for encoding. Otherwise, a full mode search will be performed on the whole inter-block to search for a best mode. We use the MV variance within a macroblock and the magnitude of MV difference defined in (1) and (2), respectively, to evaluate the reliability of the neighboring prediction information in order to reduce the quality degradation caused by incorrect predictions.

$$MV\_VAR_{\text{cur}} = \frac{1}{n}\sum_{m=0}^{n}\left\{\left(MVx_{\text{cur}}^{m} - \overline{MVx}_{\text{cur}}\right)^2 + \left(MVx_{\text{cur}}^{m} - \overline{MVx}_{\text{cur}}\right)^2\right\} \tag{1}$$

$$Mag\_dif_{\text{cur}} = \left|MVx_{\text{cur}} - \overline{MVx}_{\text{ref}}\right| + \left|MVy_{\text{cur}} - \overline{MVy}_{\text{ref}}\right| \tag{2}$$

$$\overline{MVx}_{\text{cur}} = (\sum_{m=1}^{n} MVx_{\text{cur}}^{m})/n,\ \overline{MVy}_{\text{cur}} = (\sum_{m=1}^{n} MVy_{\text{cur}}^{m})/n,\ \overline{MVx}_{\text{ref}} = (\sum_{m=1}^{n} MVx_{\text{ref}}^{m})/n,\ \overline{MVy}_{\text{ref}} = (\sum_{m=1}^{n} MVy_{\text{ref}}^{m})/n \tag{3}$$

where n is the number of motion vectors existed in each block, MVx and MVy are the horizontal and vertical component MVs, respectively, and MV_{cur} and MV_{ref} represent MVs of a current block and a reference block, respectively.

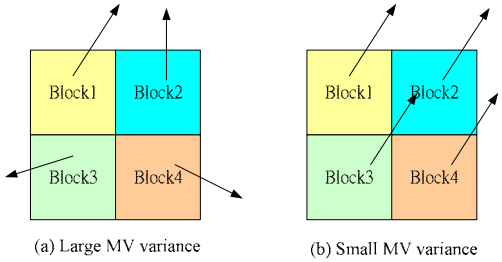


Fig. 2. Illustration of motion blocks that have a large MV variance and a small MV variance.

Fig. 2(a) and (b) illustrate motion blocks that have a large MV variance and a small MV variance, respectively. As shown in the figures, a 16×16 motion block is divided into four 8×8 sub-blocks: Block1~Block4. In Fig. 2(a), each of the four 8×8 sub-blocks has a different MV so that the whole motion block has a large MV variance. On the contrary, in Fig. 2(b), each of the four 8×8 sub-blocks has a similar MV so that the MV variance of the motion blocks is small. In this case, the 16×16 macroblock itself should be used for encoding. That is, when the MV variance of smaller blocks is small, a larger block encompassing the smaller blocks is more suitable for encoding. However, if the block mode is 16×16, there is only one group of MVs after the motion estimation. Consequently, no MV variance can be calculated therefrom. Accordingly, the proposed method only considers the MV variance for determining the reliability when the predicted block modes are smaller than the 16×16 mode, in which the modes of which the MV variances are larger than a threshold value TH_{var} are determined as reliable. A statistic result that uses the MV variance to determine the reliability of a block size mode that is smaller than the 16×16 mode is shown in Table 1. From Table 1, the average accuracy is higher than 85%.

As described above, only the reliabilities of the block size modes smaller than the 16×16 mode are determined by the MV variance in our method. For the 16×16 mode, the reliability is determined from its magnitude of MV difference defined in (2). It is known that if two adjacent blocks belong to a same object or have a same motion trajectory, the chance of the

two adjacent blocks using a same block size mode for encoding will be very high. Accordingly, the MVs of the two adjacent blocks are also similar, that is, the magnitude of MV difference of the two adjacent blocks is small. On the contrary, if the MVs of the two adjacent blocks are different, it can be predicted that the two adjacent blocks have different motion trajectories, that is, we should not use a same block size mode for encoding. Based on this concept, if the magnitude of MV difference of a current block from its adjacent block is smaller than a threshold value TH_{mag}, the current block is then considered as reliable. It can be seen that the average accuracy is higher than 80% when the blocks are determined reliable.

Table 1. Statistics of accuracy of using neighboring coding modes for prediction

QP = 28	MV variance		Magnitude of MV difference	
	$P(T\|A)$	$P(F\|A)$	$P(T\|A)$	$P(F\|A)$
Foreman	81%	19%	75%	25%
Coastguard	84%	16%	70%	30%
Carphone	85%	15%	78%	22%
Container	92%	8%	93%	7%
Akiyo	87%	13%	94%	6%
Average	86%	14%	82%	18%
A： $MV\_VAR_{cur} > TH_{var}$; B： $Mag\_dif_{cur} < TH_{mag}$; T： Reference information correct; F： Reference information incorrect				

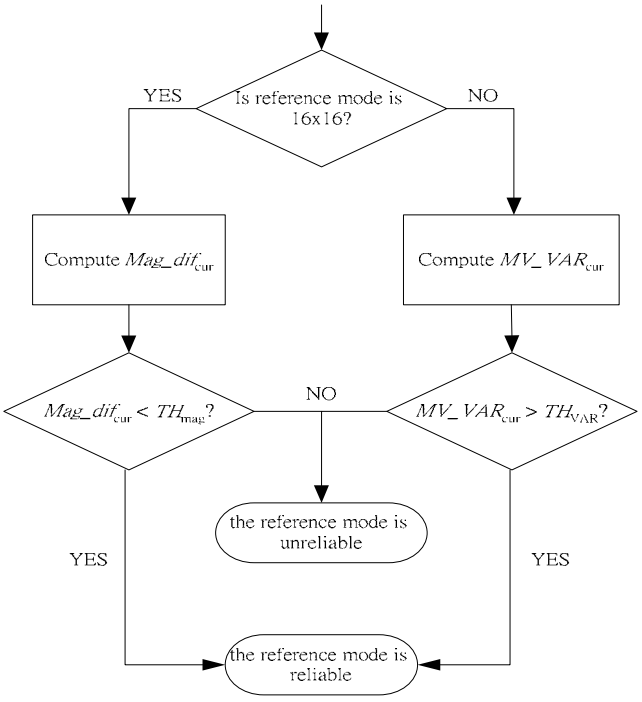


Fig. 3. Flowchart for reliability test.

The flow diagram of reliability check is shown in Fig. 3. The process first determines whether the reference mode is a 16×16 block-size mode. If so, the reliability of the reference mode is determined by its magnitude of MV difference. If the reference mode is smaller than 16×16, the reliability of the reference mode is determined by its MV variance. By analyzing the predicted information, a number of reference modes are obtained. Generally, if the prediction information for some reference blocks is the same, the block modes used for these reference blocks are also the same. Furthermore, our experiments show that if more than half of the reference blocks use a same block mode, the possibility that the current encoding block will use the same block size mode for encoding is very high. If more than half of the reference blocks use a majority block size mode, the method then determines whether this majority block mode is reliable. If it is reliable, the method then uses the majority block mode to encode the current block. If the majority block mode is unreliable, however, the method then has to perform a full-mode search to select a best block size mode. While performing the mode estimation, we use the early termination method[15] to determine whether or not to stop block splitting earlier to further reduce the computation. The two thresholds, TH_{var} and TH_{mag}, used for reliability test are determined by collecting the values of MV variance and magnitude of MV differences and from the previous frames.

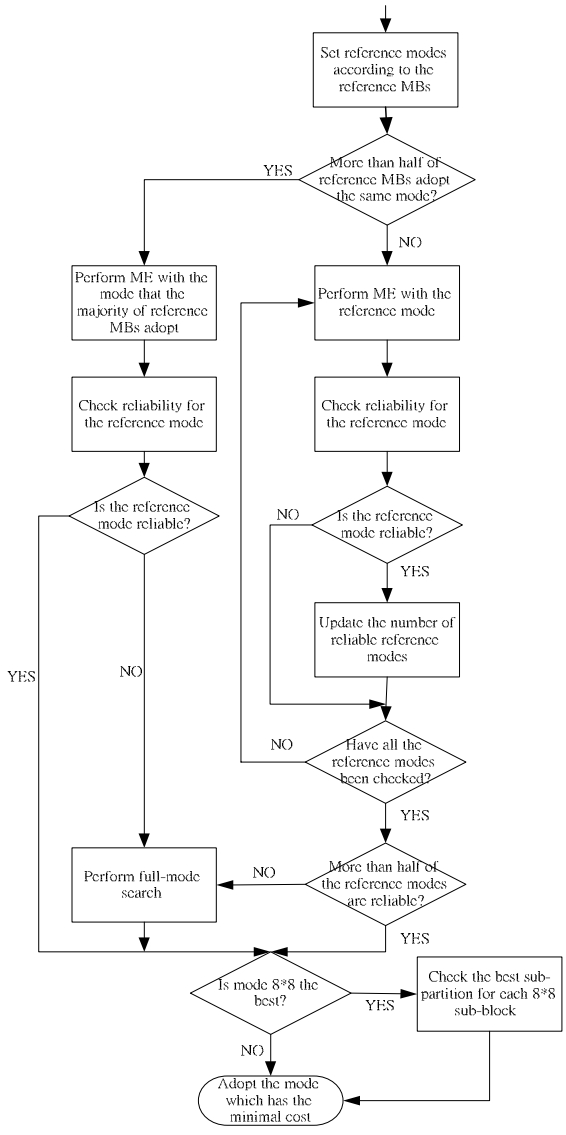


Fig. 4. Detailed flowchart for the proposed fast mode decision algorithm.

The detailed flowchart of the proposed EFMD algorithm is illustrated in Fig. 4. First, the process obtains reference block size modes according to the predicted information from the neighboring reference blocks. The method then determines whether more than half of the reference modes are the same (i.e., the majority reference block size mode.) If so (i.e., along the left branch of the flowchart), the process performs a motion estimation for the majority reference block size mode, and then checks the reliability of the majority reference mode according to the process described in Fig. 3. Should the majority reference mode be reliable, the best reference mode is used as a basis for determining the final coding mode; otherwise, a full-mode search over the 16×16, 16×8, 8×16, and 8×8 modes is performed to find out a best mode.

Furthermore, when not more than half of the reference modes are the same (i.e., along the right branch of the flowchart), i.e., not more than half of the reference motion blocks adopt the same mode, the process performs the motion estimation for all of the reference block size modes. All of the reference modes are then checked, for their reliabilities according to the process of Figure 3. After checking the reliabilities of all of the reference modes, the reference modes that are considered reliable are recorded. The process then checks whether more than half of the reference modes are reliable. If not, the process then performs a full-mode search on the current motion block to find out a best block size mode.

After finishing either of the above two branch testes, the process checks whether a best reference mode is an 8×8 mode. If so, the process goes on checking a best sub-partition for each 8×8 sub-block. As each 8×8 sub-block can be further divided into 8×4, 4×6, and 4×4 sub-blocks, it should be noted that, when the 8×8 sub-block is not the best mode, there is no need to analyze the sub-blocks smaller than the 8×8 sub-blocks because the chance that the smaller sub-blocks are the best mode is very small. As such, the encoding time can be greatly reduced. Finally, if an 8×8 block size mode is not the best, the process then adopts a mode that has the minimum cost for encoding the current motion block.

After the algorithm-level optimization for the proposed mode decision and the fast hexagonal fast motion estimation adopted from 23, several code-level optimization techniques are proposed to pursue for further speed-up. Our proposed code-level optimization techniques include memory rearrangement, and optimization with the SIMD technology for fractional-pixel interpolation, (inverse) integer transform, Hadamard transform, and SAD process.

3. CODE-LEVEL OPTIMIZATION

3.1 Optimization of interpolation

3.2.1. Memory rearrangement

In the reference software, all types of fractional pixels are stored altogether with integer pixels in an up-sampled frame memory as shown in Fig. 5(a), in which the blocks labeled with '*G*' indicate the integer-pixel locations; the blocks labeled with '*b*,' '*h*,' and '*j*' stand for the half-pixel locations; the rest are quarter-pixel locations. As a result, pixels of the same type are stored in noncontiguous locations of the memory, making it inefficient in memory access. For example, some computation-intensive modules, such as motion estimation, motion compensation, transform and inverse transforms, require a 4×4 block of luma-pixels of the same type for each operation (e.g., for block matching). Thus, 16 memory accesses are required for each operation since the 16 pixels in a 4×4 block are stored in 16 noncontiguous locations. The way the pixels are stored in the reference software lowers down the cache or memory access efficiency and does a significant negative impact on runtime performance.

In order to avoid the inefficient memory access due to noncontiguous memory access, the way the fractional pixels are stored is rearranged in the proposed encoder as shown in Fig. 5(b). Sixteen image memories, which are of the same size as the source video picture, are allocated for the storage of each type of fractional pixels. In this way, 16 pixels of a 4×4 block are now distributed contiguously in memory. Besides the improvement of the cache or memory access efficiency, loading 16 pixels of a 4x4 block now takes only 4 memory load operation if the SIMD *parallel-load* instruction is exploited, whereas this instruction cannot be utilized in the original storage arrangement.

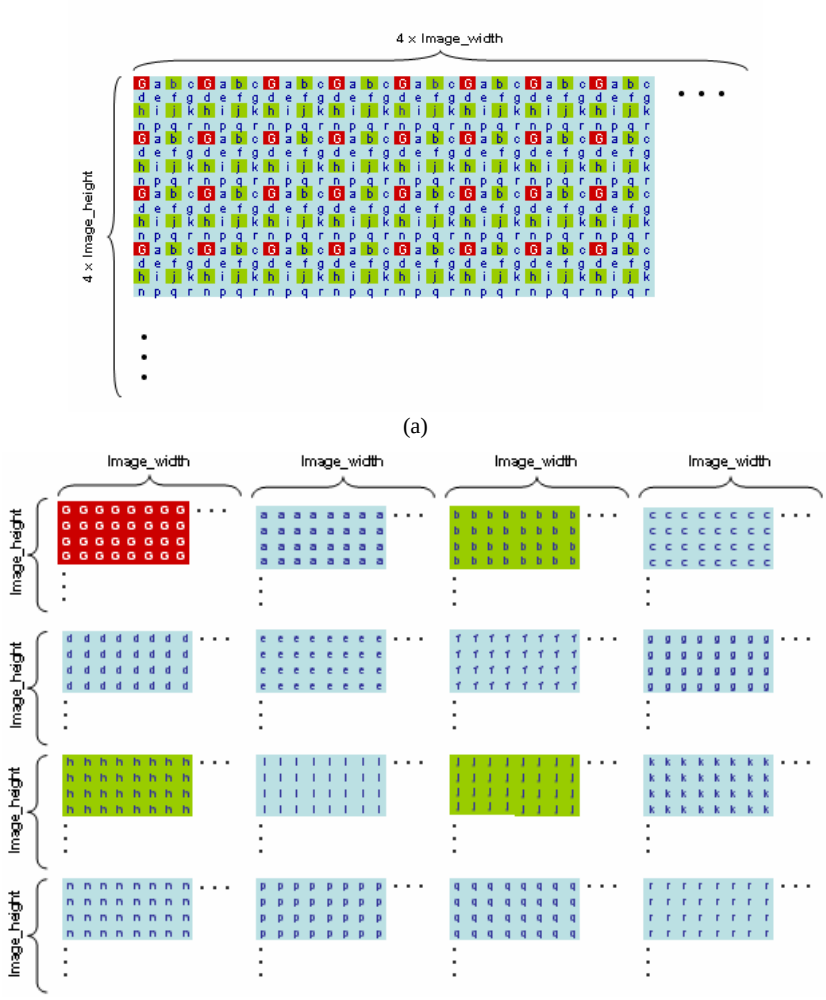


Fig. 5. (a) Frame memory arrangement in the JM7.3 reference encoder; (b) rearranged frame storage.

3.2.2 Interpolation with SIMD instructions

After rearranging the array locations of the interpolated image, we modify the interpolation function ("UnifiedOneForthPix") corresponding to the location rearrangement so as to apply the Intel SIMD technology for optimization. In the reference encoder, the fractional-pixel interpolation process can be divided into three steps:

Step 1. Those half-pixels labeled with '***b***' in a frame are interpolated first. The interpolated pixels and the integer samples are then stored into a temporary image together, which is as large as two times of the original image size.

Step 2. Those half-pixels labeled with '*h*' and '*j*' in a frame are then interpolated by applying the 6-tap FIR filter vertically to the temporary image which is created in Step 1. All integer and half pixels are then stored into the up-sampled image.

Step 3. The quarter-pixels are subsequently interpolated by applying the bilinear filter and stored in the up-sampled image.

In Step 1 of the proposed SIMD implementation, eight half-pixels (namely, half-pixel labeled with '*b*' in Fig. 5) will be obtained in parallel, except that the image-boundary case are still implemented by the non-parallel C function since there

are fewer parallelism to be exploited around the boundary. Eight integer-pixels are loaded into the SSE2 registers before being packed into 16-bit short words. Then the FIR filter is applied with only *shift* and *add*/*subtract* operations on the integer-pixels (e.g., 20x*A* = *A*<<4 + *A*<<2). It is done by loading six rows of eight integer-sample pixels into six SSE2 registers. After applying the FIR filter to each one utilizing *shift* and *add* operations, intermediate values are obtained by summing up these data in the registers in parallel. The final half-pixel values are obtained by performing parallel *shift* on the intermediate values and are then stored into a temporary memory together with the integer-samples, which is the same as the original JM encoder does.

In Step 2, the eight pixels, including four half pixels labeled with '*h*' and four half-pixels '*j*', are interpolated in parallel which is very similar to Step 1 but the FIR filter is applied vertically. After the parallel interpolation is done, all the half-pixels and integer-pixels are stored in the proposed rearranged array. Note that the rearranged array makes fractional pixels of the same type contiguous in memory storage, facilitating the implementation of the interpolation process with the SIMD instructions (contiguous pixels can be loaded into the SSE2 registers with one instruction rather than multiple memory accesses).

In Step 3, the *parallel-average (PAVG)* instruction is utilized for quarter-pixel interpolation.

The computational complexity of the "*UnifiedOneForthPix*" interpolation function is consequently reduced by about 97% with the proposed memory rearrangement and the proposed SIMD implementation of interpolation.

3.2 Optimization for SATD computation

The SATD function is used in the intra prediction and the fractional-pixel motion estimation process. When the fractional-pixel motion estimation is performed, the sum of absolute Hadamard transform coefficients is adopted as a block matching criterion. The 4×4 SATD values are obtained by performing a 4×4 Hadamard transform on the 16 sub-block DC values of an intra-coded MB or on the prediction residues of each 4×4 sub-block in an inter-coded macroblock.

In the SATD function, only *add* and *subtract* instructions are involved. We can use the SSE2 instructions (e.g., *parallel-add*, *parallel-subtract*) to optimize the SATD function. Computing the SATD values with SSE2 is described as follows.

Step 1. Load the block of pixels in the source and reference frames into the SSE2 registers in parallel.

Step 2. Perform block difference operation in parallel. Eight operations can be executed per-instruction.

Step 3. Use *unpack* or *shuffle* instructions to make operands which will be applied to the same operations together.

Step 4. Perform *add* and *subtract* instructions to apply the Hadamard transform.

Step 5. Calculate the absolute values of the coefficients without any branches in parallel.

Step 6. Sum up all the intermediate values in the SSE2 registers to obtain the *SATD*

Simulation results show that, the SATD functions can be sped up by about 4.2 times with the proposed SIMD implementation.

3.3 Optimization of integer transform and inverse integer transform

In H.264, a new 4x4 integer transform pair has been introduced. Since only integer instructions are used in the integer transform pair, there is no floating-point arithmetic operations involved. As depicted in Fig. 6, the integer transform pair requires only *shift* and *add* operations on 16-bit operands, making it well suited for SIMD implementations.

For clarity but without loss of generality, only the inverse transform is presented as an example in the following, since the optimization for the forward transform is very similar to that for the inverse transform. The inverse transform is implemented by performing four 1-D four-point row-wise inverse transforms followed by four 1-D four-point column-wise inverse transforms. Details of 1-D four-point inverse integer transform are shown below:

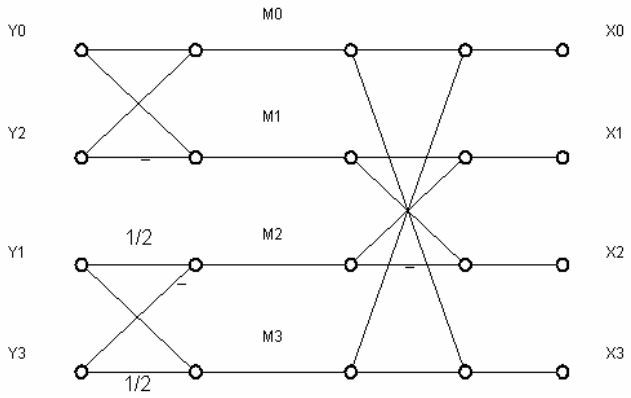


Fig. 6. A butterfly structure of 1-D 4-point inverse integer transform.

In terms of parallel implementation, other than applying the 2-D transform directly, computing the 1-D inverse transform twice is simpler to implement in parallel. The four 1-D column transforms can be done in parallel without any overhead. However, the Intel SIMD technology does not support operations on different packed parts in one register. That is, we cannot perform the row-wise inverse transform directly for all of the four coefficients in one row, since the four coefficients are kept in one packed word in one register. Therefore, the 4x4 matrix is transposed and is then followed by the row-wise transform. The inverse integer transform can be described as follows.

Step 1. Load the 16x16 coefficients into four MMX registers in transpose order (16 bits/coef.)

Step 2. Perform the column-wise 1-D inverse transform. (16-bit Word precision is required)

Step 3. Compute four 1-D inverse transform in parallel

Step 4. Transpose the matrix that produced in above step

Step 5. Perform the row-wise 1-D inverse transforms

Step 6. Add the above result (residue) to the prediction block by ((mpr<<dq_bit) + residue + dq_round) >> dq_bit

Simulation result shows that, the integer transform function can be sped up by 1.5 times with the proposed SIMD implementation.

3.4 Optimization of SAD computation

In performing the integer-pixel motion estimation, the sum of absolute difference (SAD) between the source block and the reference block is used as the block matching criterion. The operations of computing SAD on individual pairs of pixels is independent, making it suitable for SIMD implementation. The PSAD instruction for MMX and SSE2 are utilized according to the size of the matching block. The optimized algorithm for the SAD calculations is described as follows.

Case 1: If the width of the matching block is 4, perform SAD calculation in C.

Case 2: If the width of the matching block is 8-bit,

A. Load the 8 pixels of current block into the MMX1 register and the 8 pixels of reference block into the MMX2 register. (8 bits/coef.)

B. Perform the SAD instruction on MMX1 and MMX2.

Case 3: If the width of block is 16-bit,

A. Load the 16 pixels of current block into XMM1 register and the 16 pixels coefficients of reference block into XMM2 register. (8 bits/coef.)

B. Perform the SAD instruction on XMM1 and XMM2. And sum up the two intermediate SAD values in high/low word of the destination register.

Simulation results show that the integer-pixel motion estimation can be sped up by 1.5 times with the proposed SAD optimization.

4. EXPERIMENTAL RESULTS

The experiments are performed on a Pentium-4 2.4GHz personal computer equipped with 512 MB main memory and the Windows XP OS. All codes are compiled by the Intel® compiler[21]. The run-time complexities are profiled using the Intel® VTune 20 performance analyzer. Five QCIF (176×144) sequences of 150 frames, including *Foreman*, *Coastguard*, *Carphone*, *Container*, and *Akiyo*, are tested. The motion search range is set to 16, and the number of reference frames used for motion estimation is set to one. The sequences are coded in IPPPP…, with a frame rate of 30 fps. Besides, CAVLC is used as the entropy coder. The RD optimization option in the reference encoder is turned off for the experiments.

Table 2 shows the runtime comparison for different levels of optimization. According to the simulation results, the encoding speed of the JM 7.3 is only about 4 QCIF or 1 CIF fps. With the help of hexagonal-search fast motion estimation, the encoder achieves a speed-up factor of 1.9, leading to the speed of about 8 QCIF or 2 CIF fps. With the proposed joint optimization methods, the encoder is further enhanced to achieve up to 48 QCIF and 12 CIF fps if all available block modes are searched and up to 76 QCIF and 19 CIF fps if only 3 inter modes are searched. That is, the optimized encoder can achieve a speed-up factor of about 18.

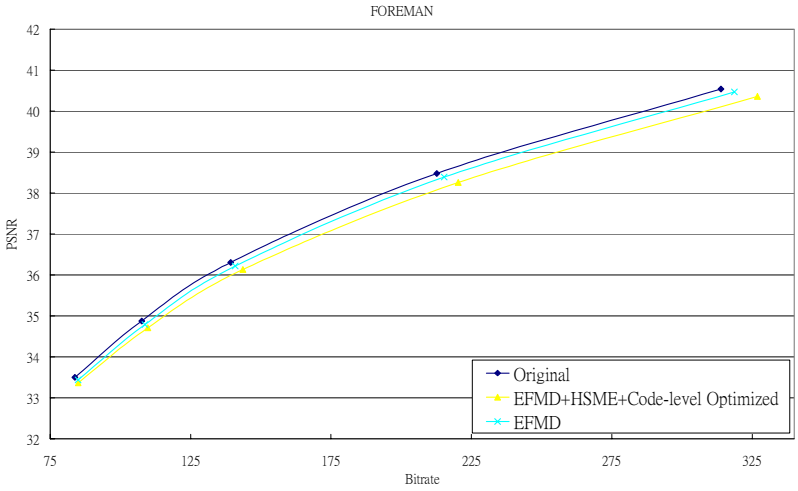
Table 2. Runtime comparison for different-levels of optimization for five test sequence (150 frames)

QP = 28	JM7.3	EFMD	Code Opt.	Joint Opt.
Foreman	47.30 s	30.74 s	21.39 s	2.83 s
Coastguard	56.30 s	31.41 s	27.57 s	3.03 s
Carphone	42.01 s	25.12 s	18.92 s	2.47 s
Container	40.63 s	19.13 s	17.62 s	2.07 s
Akiyo	34.68 s	17.12 s	13.27 s	2.04 s

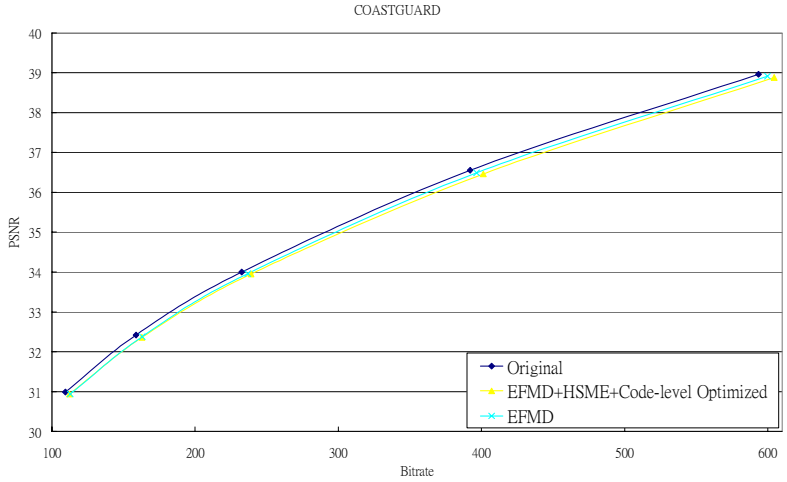
The fast mode decision and fast motion estimation will reduce the coding efficiency a little bit. The RD performances of the joint optimization scheme which integrates the proposed EFMD, a state-of-the-art fast ME algorithm[23], and code-level optimization schemes, is compared with the original JM 7.3 reference software in Table 3 and Fig. 7. For sequences without intensive motions (e.g., *Akiyo*, *Container*, and *Coastguard*), the optimized encoder yields very close coding performance as compared to the non-optimized JM7.3 coder. For sequences with relatively large motions (e.g., *Foreman*), the optimized encoder (with fast ME, mode decision and SIMD optimization) introduces about 0.6 dB degradation for the same bit rate. From the simulation results, our proposed joint optimization H.264 encoder can achieve a significant speed-up without introducing serious quality degradation.

Table 3. Average R-D performance (PSNR and bitrate) comparison with a fixed quantization step-size for five test sequence

QP = 28	PSNR Difference		Bitrate Difference	
	EFMD	Joint Opt.	EFMD	Joint Opt.
Foreman	-0.09 dB	-0.18 dB	1.16 %	3.09 %
Coastguard	-0.04 dB	-0.05 dB	1.81 %	2.87 %
Carphone	-0.13 dB	-0.22 dB	1.38 %	2.81 %
Container	-0.04 dB	-0.03 dB	2.98 %	5.41 %
Akiyo	-0.04 dB	-0.07 dB	1.42 %	2.50 %



(a)



(b)

Fig. 7. Average PSNR performance comparison using the JM7.3 software, the algorithm-level optimization, and the algorithm/code-level joint optimization: (a) *Foreman* and (b) *Coastguard*.

5. CONCLUSION

In this paper, we proposed a fast mode decision algorithm. We also proposed efficient code-level optimization techniques, such as the frame memory rearrangement, the early termination for variable block size motion estimation, and several optimizations with SIMD technology, etc. We have implemented a highly optimized coder by integrating a state-of-the-art fast motion estimation and the proposed fast mode decision and code-level optimization schemes. Experimental results show that the optimized encoder can achieves a speed up factor of up to 18 compared to the reference encoder (JM 7.3) without introducing serious quality degradation, making it more feasible for real-time applications.

REFERENCES

1. Joint Video Team of ITU-T and ISO/IEC JTC 1, *Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC)*, Doc. JVT-G050, Mar. 2003.

2. T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, “Overview of the H.264/AVC video coding standard,” *IEEE Trans. Circuits Syst. Video Technol.*, vol.13, no. 7, pp. 560-576, July 2003.

3. M. Ravasi, M. Mattavelli, and C. Clerc, “A computational complexity comparison of MPEG4 and JVT codecs,” Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, Doc. JVT-D153r1-L, July 2002.

4. X. Zhou, E. Q. Li and Y. K. Chen, “Implementing H.26L decoder on general-purpose processors with media instructions”, *SPIE Conf. on Image and Video Communications and Processing*, San Diego, USA, Jan. 2003.

5. T. Koga et al, “Motion-compensated interframe coding for video conferencing,” *Proc. Nat. Telecommun. Conf.*, New Orleans, LA, pp. G.5.3.1-G.5.3.5, Dec. 1981.

6. S. Zhu and K.K. Ma, “A new diamond search algorithm for fast block-matching motion estimation,” *IEEE Trans. Image Processing*, vol. 9, pp. 287-290, Feb. 2000.

7. M. Bierling, “Displacement estimation by hierarchical block matching,” *Proc. SPIE Conf., Visual Commun. And Image Processing’88*, vol. 1001, part 2, pp. 942-951, 1988.

8. B. Liu and A. Zaccarin, “New fast algorithms for the estimation of block motion vectors,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 3, no. 2, pp. 148-157, Apr. 1993.

9. Y.-K. Tu, J.-F. Yang, Y.-N. Shen, and M.-T. Sun, “Fast variable-size block motion estimation using merging procedure with an adaptive threshold,” in *Proc. IEEE Conf. Multimedia & Expo*, vol. 2, pp. 789-792, July 2003.

10. T.-Y. Kuo and C.-H. Chan, “Fast macroblock partition prediction for H.264/AVC,” in *Proc. IEEE Conf. Multimedia & Expo*, June 2004, Taipei, Taiwan.

11. A. Ahmad, N. Khan, S. Masud, and M.A. Maud, “Selection of variable block sizes in H.264,” in *Proc. IEEE Conf. Acoustic, Speech, and Signal Processing*, May 2004, Montreal, Canada.

12. A. C. Yu, “Efficient block-size selection algorithm for inter-frame coding in H.264/MPEG-4 AVC,” in *Proc. IEEE Conf. Acoustic, Speech, and Signal Processing*, May 2004, Montreal, Canada.

13. D. Wu, S. Wu, K. P. Lim, F. Pan, Z. G. Li, and X. Lin, “Block inter mode decision for fast encoding of H.264,” in *Proc. IEEE Conf. Acoustic, Speech, and Signal Processing*, May 2004, Montreal, Canada.

14. Z. Zhou, M.-T. Sun, and S. Hsu, “Fast variable block-size motion estimation algorithms based on merge and split procedures for H.264/MPEG-4 AVC,” in *Proc. IEEE Symp. Circuits and Systems*, May 2004, Vancouver, Canada.

15. Z. Zhou and M.-T. Sun, “Fast macroblock inter mode decision and motion estimation for H.264/MPEG-4 AVC,” in *Proc. IEEE Int. Conf. Image Processing*, Oct. 2004, Singapore.

16. Y.-S. Tung, C.-C. Ho, and J.-L. Wu, “MMX-based DCT and MC algorithms for real-time pure software MPEG decoding,” in *Proc. IEEE Int. Conf. Multimedia Computing and Systems*, vol. 1, pp. 357-362, June 7-11, 1999.

17. P. Hsu and R. Liu, “Software optimization of video codecs on Pentium processor with MMX technology,” *EURASIP J. Applied Signal Processing*, vol. 2001, no. 2, pp.100-109, June 2001.

18. S. M. Akramullah, I. Ahmad, and M.-L. Liou, “Optimization of H.263 video encoding using a single processor computer: performance tradeoffs and benchmarking,” *IEEE Trans. Circuits Syst. Video Technol.*, vol.11, no. 8, pp. 901-915, Aug. 2001.

19. P. Hsu and R. Liu, “Software optimization of H.263 video encoder on Pentium processor with MMX technology,” in *Proc. IEEE Int. Conf. Multimedia and Expo*, vol. 1, pp. 103-106, July 2000.

20. Intel Corp., “Intel® VTune™ Performance Analyzer.”

21. Intel Corp., “Intel ® Compilers.”

22. C. Zhu, X. Lin, and L.-P. Chau, “Hexagon-based search pattern for fast block motion estimation,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, no. 5, pp. 349–355, May 2002.

23. J.-N. Zhang, Y.-W. He, S.-Q. Yang, and Y.-Z. Zhong, "Performance and complexity joint optimization for H.264 video coding," in *Proc. IEEE Int. Conf. Circuits and Systems*, vol. 2, pp. 888-891, May 25-28, 2003, Bangkok, Thailand.

24. Z. Chen, P. Zhou, and Y. He, “Fast integer pel and fractional pel motion estimation in for JVT”, JVT-F017r1.doc, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, 6th meeting, Dec. 2002.