Efficient SIMD Optimization of HEVC Encoder over X86 Processors

Keji Chen, Yizhou Duan, Leju Yan, Jun Sun* and Zongming Guo
Institute of Computer Science and Technology, Peking University, Beijing 100871, China
E-mail: {chenkeji, duanyizhou, yanleju, jsun, guozongming}@pku.edu.cn  Tel: +86-10-82529523

*is the corresponding author.

This work was supported by National Natural Science Foundation of China under contract No. 61071082, National Basic Research Program (973 Program) of China under contract No.2009CB320907 and National Key Technology R&D Program of China under Grant 2012BAH18B03.

Abstract— High Efficient Video Coding (HEVC) is the next generation video coding standard in progress. Based on the traditional hybrid coding framework, HEVC implements enhanced tools to improve compression efficiency at the cost of far more computational payload than the capacity of real-time video applications. In this paper, we focus on the fast implementation of the HEVC encoder over modern Intel x86 processors. First, we identify the most time-consuming modules of HM 6.2 encoder, represented by motion compensation, Hadamard transform, sum of difference (SAD/SSD) calculation and integer transform. Then the single-instruction-multiple-data (SIMD) methods are exploited to optimize the computational performance of these modules. Experimental results show that in these modules the optimized encoder achieves 56% - 85% time saving compared with the HM 6.2 encoder over Intel i5-750 processor.

I. INTRODUCTION

In order to effectively improve video coding efficiency, ISO-IEC/MPEG and ITU-T/VCEG has formed Joint Collaborative Team on Video Coding (JCT-VC) to develop the next generation video coding standard called High Efficient Video Coding (HEVC). With a lot of enhanced coding tools introduced, HEVC is expected to achieve 50% bit rate reductions at similar mean opinion score (MOS) compared with the previous standard H.264/AVC. However, the computational complexity of HEVC has greatly increased, making encoding speed a serious problem in the implementation of HEVC.

In recent years, most high-performance micro-processors provide media instructions. A typical example on common server platform is the Intel MMX/SSE technologies [1], which are based on the single-instruction-multiple-data (SIMD) methods. By exploiting the significant data-level parallelism, SIMD technologies provide a series of effective approaches for fast algorithm implementation, which brings useful guidance to optimize the computational performance. In this paper, we focus on the fast implementation of HEVC reference encoder of HM 6.2 [2] over Intel x86 processors based on SIMD technologies.

The rest of this paper is organized as follows. A brief overview of the new features in HEVC is given in Section 2. And the complexity analysis of HEVC encoder, the SIMD optimization algorithm design and implementation are provided in Section 3. Experimental results of the optimization are shown in Section 4. Finally, conclusions are given in Section 5.

II. OVERVIEW OF THE NEW FEATURES IN HEVC

HEVC utilizes a quadtree structure [3] to support large and flexible block sizes. The size of a coding unit (CU) can be 64x64, 32x32, 16x16 and 8x8. Each CU is split into one or more prediction units (PU) and transform units (TU). For PU, the width and height of a PU varies from 4 to 64, indicating that the blocks to be processed in motion compensation (MC) can be as large as 64x64. In motion estimation (ME), sum of absolute differences (SAD) and sum of absolute transformed differences (SATD) of different block sizes are calculated. Due to the flexible block structure, each 4x4 block will be calculated several times from 4x4 to 64x64 ME, which can be quite time-consuming. And also, the prediction method in HEVC is much more complex than that of H.264/AVC: in intra luminance prediction, up to 35 candidate modes are evaluated, while in inter luminance prediction, separable filters [4] are utilized, including a horizontal and a vertical interpolation step utilizing two 8-tap interpolation filters respectively. For TU, the size of a TU can be 32x32, 16x16, 8x8, 4x4 or even non-squared, which can be much larger than that of H.264/AVC. In addition, the transform method is no longer multiplication-free [5]. As a result, the computational complexity of transform coding in HEVC has greatly increased. Some other sophisticated methods are also introduced in high efficiency profile of HEVC, like adaptive loop-filter (ALF) [6]. However, we will not cover them in this paper.

III. IMPLEMENTATION OF HEVC ENCODER WITH SIMD OPTIMIZATION

A. Complexity Analysis of HEVC Encoder

In order to identify the most time-consuming modules in the HEVC encoder, we analyze the execution time of the HM 6.2 encoder over Intel i5-750 2.67GHz CPU with 4G Byte memory under the Windows 7 operating system. We use sequences of 720p (1280x720) resolution in random-access situation. The Intra period is set to 32 and the GOP size is 8. We run 100 frames for each sequence under the QP value of 32.
The proportion of the average execution time of HM 6.2 encoder major modules is shown in Fig. 1, from which we can find that the most time-consuming modules are MC, Hadamard transform, SAD and SSD calculations, integer transform, memory operations and rate-distortion optimization quantization (RDOQ). The computational performance of the first four modules can be effectively improved by using SIMD methods.

### Intel SSE instructions

Streaming SIMD Extensions (SSE) is the SIMD instruction set extension over the x86 architecture. It is further enhanced to SSE2, SSE3, SSSE3 and SSE4 subsequently. SSE contains eight 128-bit registers originally, known as XMM0 through XMM7. The number of register is extended to sixteen in AMD64. Each 128-bit register can be divided into two 64-bit integers, four 32-bit integers, eight 16-bit short integers or sixteen 8-bit bytes. With SSE series instructions, several XMM registers can be operated at the same time, indicating considerable data-level parallelism. In Fig. 1, the most time-consuming modules are all integer algorithm modules. Thus we only use integer SIMD instructions.

### Motion Compensation

MC is based on 1/4 pixel accuracy reconstructed pictures in HEVC. 8-tap luminance interpolation filters and 4-tap chrominance interpolation filters are utilized to obtain sub-pixels at fractional positions. The 8-tap luminance interpolation and 4-tap chrominance methods are shown in Fig. 2 and Fig. 3.

#### 8-tap luminance interpolation

\[
\begin{align*}
a &= c_0 \cdot A + c_1 \cdot B + c_2 \cdot C + c_3 \cdot D + c_4 \cdot E \\
&+ c_5 \cdot F + c_6 \cdot G + c_7 \cdot H,
\end{align*}
\]

(1)

and

\[
\begin{align*}
e &= c_0' \cdot I + c_1' \cdot J + c_2' \cdot K + c_3' \cdot L,
\end{align*}
\]

(2)

where \(c_0-c_7\) and \(c_0'-c_3'\) are interpolation coefficients. This is a typical vector product algorithm. We propose a fast implementation of MC by optimizing row and column sub-pixel interpolation for luminance and chrominance components based on Intel SIMD instructions.

The vector product in MC is somewhat tricky. One vector contains continuously aligned pixels, which are unsigned bytes. And the other contains interpolation coefficients, which are signed bytes. The SSSE3 instruction PMADDUBSW can be used to compute the vector product for row luminance sub-pixel interpolation. The PMADDUBSW instruction takes two 128-bit SSE registers as operands, with the first one containing sixteen unsigned 8-bit integers, and the second one containing sixteen signed 8-bit integers. With this instruction, we only need to sum the values in the result register to get the final results.

We consider 8-pixel horizontal luminance interpolation first. To calculate eight pixels, we need fifteen pixels in total, which can be loaded into a single register. Then we get all the necessary vectors by four PSHUFB instructions. With four PMADDUBSW instructions, vector product is finished. Finally, we use three PHADDW to summarize the result into one register. The register contains the eight aligned necessary results then. The implementation is shown in Fig. 4.

#### Luminance Horizontal Interpolation

It can be seen that, with four PSHUFB instructions, we get the wanted vectors and make the most use of four registers. The solution for the 4-pixel horizontal luminance interpolation is almost the same. We can use the first two PSHUFB instructions to process it. Then only two PHADDW instructions are needed to summarize.

Then we consider 8-pixel horizontal chrominance interpolation. To interpolate eight pixels, we need eleven pixels, which can be loaded into a single register. Then we get all the necessary vectors by two PSHUFB instructions. With two PMADDUBSW instructions, vector product is finished. Finally, we use one PHADDW to summarize the result into one register. The implementation is shown in Fig. 5.

#### Chrominance Horizontal Interpolation

The solution for 4-pixel horizontal chrominance interpolation is similar. With only one PSHUFB, one PMADDUBSW and one PHADDW instructions, we can finish this task.

For vertical interpolation, we should apply vector products on 16-bit intermediates. We use a simple algorithm in this
step. If the block width is 4, we load four intermediates with one MOVQ instruction, expand each 16-bit intermediate into 32-bit with one PMOVSXW instruction, then multiply it with one PMULLUD instruction and accumulate it with one PADDD instruction. If the block width is 8 or larger, we load eight intermediates each time with one MOVDQU instruction, multiply it with one PMULLW and one PMULHW instructions, then shuffle the two result registers with one PUNPCKLWD and one PUNPCKHWD instructions, and finally accumulate them with two PADDD instructions.

D. Hadamard Transform

Hadamard transform is utilized to find the best intra prediction mode and the best sub-pixel motion vector. Encoder calculates the difference between the predicted and the original blocks, and transforms the difference with Hadamard matrix. In HEVC, most blocks are larger than 8x8, and they are divided into several 8x8 blocks. So we optimize 8x8 Hadamard transform first.

Before 8x8 Hadamard transform, the difference of the predicted and the original blocks is loaded in eight SIMD registers. Each register contains eight 16-bit differences in a line. We apply horizontal transform first and then vertical transform. In horizontal transform, we use the algorithm in Fig. 6.

Before the transform, we should use MOVQ or MOVDQU instructions to load pixels into XMM registers. In 8x8 Hadamard transform, only eight pixels are in one line. So we can only use MOVQ instructions. But in most cases, we do 8x8 Hadamard transform several times to process a large block. If we load these pixels with MOVQ, cycles are saved then.

For the vertical transform, we use the same algorithm as horizontal transform but implement it in a different way. First, we use one PHADDW and one PHSUBW instructions for each pair, as is shown in Fig 8. Then we use eight PADDW and eight PSUBW instructions to calculate the final results, as is shown in Fig. 9.

There are 48 instructions used in transform part, making the most use of the registers and instructions. Then we need eight PABSW instructions to get the absolute values and several PADDW and PADDD instructions to accumulate the results.

Before the transform, we should use MOVQ or MOVDQU instructions to load pixels into XMM registers. In 8x8 Hadamard transform, only eight pixels are in one line. So we can only use MOVQ instructions. But in most cases, we do 8x8 Hadamard transform several times to process a large block. If we load these pixels with MOVQ, cycles are saved then. So we implement a 16x8 Hadamard transform. It works in the same way as two 8x8 Hadamard transform, and it runs faster.

For the vertical transform, we use the same algorithm as horizontal transform but implement it in a different way. First, we use one PHADDW and one PHSUBW instructions for each pair, as is shown in Fig 8. Then we use eight PADDW and eight PSUBW instructions to calculate the final results, as is shown in Fig. 9.

There are 48 instructions used in transform part, making the most use of the registers and instructions. Then we need eight PABSW instructions to get the absolute values and several PADDW and PADDD instructions to accumulate the results.

Before the transform, we should use MOVQ or MOVDQU instructions to load pixels into XMM registers. In 8x8 Hadamard transform, only eight pixels are in one line. So we can only use MOVQ instructions. But in most cases, we do 8x8 Hadamard transform several times to process a large block. If we load these pixels with MOVQ, cycles are saved then. So we implement a 16x8 Hadamard transform. It works in the same way as two 8x8 Hadamard transform, and it runs faster.

The 4x4 Hadamard transform is similar to 8x8 size. It is utilized only in 8x4, 4x8 and 4x4 blocks. It can be optimized by SIMD methods in a similar way as is introduced above.

E. SAD/SSD Calculation

The SSE instruction set uses the PSADDBW instruction to calculate SAD. This instruction gives the SAD result of two operands. Therefore, we can load operand into registers, use PSADDBW instruction, and then use PADDW to sum all the results. It should be noted that for 4x4 blocks the operations are different, in which case we set the upper 96 bits of each operand to zero, since PSADDBW can only give the SAD of 8-byte operands.

In SSD calculation, we use the PMADDWD instruction to do the multiplication. The PMADDWD instruction has two
operands, each containing eight 16-bit values. With this instruction, we can do eight multiplications each time, as is shown in Fig. 10. We calculate all the lines with PMADDWD, sum the result with PADDD, and then get final results by using PHADD.

![Image](PMADDWD Instruction)

**Fig. 10** PMADDWD Instruction

### F. Integer Transform

The integer transform module of HEVC is far more complex than that of H.264/AVC. It has more complex transform coefficient matrix and its size varies from 4x4 to 32x32. Due to the fact that the transform coefficients do not only consist of $2^n$ now, traditional multiplication-free method is no longer applicable. Thus we implement the ordinary matrix multiplication algorithm, which is expressed as

$$Y = A^T X A,$$

where $X$ is coefficient matrix and $A$ is transform coefficient matrix. In order to implement this algorithm using SIMD methods, we change the formula as

$$Y = ((X A)^T A)^T.$$

Then we implement SIMD optimization in two steps: horizontal transform and transpose. In transform step, we extend the chain matrix multiplication scheme in [7] up to blocks with size of 32x32. And we implement an 8x8 matrix multiplication kernel with 64 PMADDWD instructions to reach this approach. In transpose step, we divide the block into several 8x8 blocks and transpose it with 24 unpack instructions. Fig. 11 shows how to transpose a 8x8 block with unpack instructions.

![Image](8x8 Matrix Transpose)

**Fig. 11** 8x8 Matrix Transpose

The dashed lines in Fig. 11 represent the PUNPCKHW instruction and the solid lines represent the PUNPCKLWD instruction. After the transpose operation, we write the transposed block into the corresponding position. And then, blocks larger than 8x8 are transposed.

The difference between transform and inverse transform is their transform coefficient matrix, and their algorithm is the same. So we implement a same algorithm in inverse transform.

### IV. EXPERIMENTAL RESULTS

We have tested all the modules we optimized above. The experiments are carried out under the same environment described in Section 3.1. With the optimization methods above, we save 56% - 85% time in the most time-consuming modules. Table I shows the detailed time saving of each module.

<table>
<thead>
<tr>
<th>Module</th>
<th>Origin Time (ms)</th>
<th>Optimized Time (ms)</th>
<th>Time Saving (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motion Compensation</td>
<td>9579</td>
<td>2141</td>
<td>77.6</td>
</tr>
<tr>
<td>Hadamard Transform</td>
<td>3081</td>
<td>974</td>
<td>68.4</td>
</tr>
<tr>
<td>SAD/SSD Calculation</td>
<td>3929</td>
<td>587</td>
<td>85.1</td>
</tr>
<tr>
<td>Integer Transform</td>
<td>2501</td>
<td>1091</td>
<td>56.4</td>
</tr>
</tbody>
</table>

### V. CONCLUSIONS

HEVC is the next generation coding standard, and it will be widely used in the future. In this paper, we analyze the most time-consuming modules of the HEVC encoding process of HM 6.2 and present the optimization of these modules with SIMD technologies. Experiments on these modules show that the proposed method achieves 56% - 85% time saving compared with the original method in HM 6.2.

### REFERENCES