Submitted to Special Issue on Emerging H.264/AVC Video Coding Standard,
Journal of Visual Communication and Image Representation,
by Kumar, Mandal, and Panchanathan.

Implementation of H.264 Encoder and Decoder on Personal Computers

Yen-Kuang Chen*, Eric Q. Li, Xiaosong Zhou†, and Steven Ge
Corporate Technology Group, Intel Corp

ABSTRACT

H.264 is an emerging video coding standard, which aims at compressing high-quality video contents at low bit-rates. While the new encoding and decoding processes are similar to many previous standards, the new standard includes a number of new features and thus requires much more computation than most existing standards do. The complexity of H.264 standard poses a large amount of challenges to implementing the encoder/decoder in real-time via software on personal computers. This work analyzes software implementation of H.264 encoder and decoder on general-purpose processors with media instructions and multi-threading capabilities. Specifically, we discuss how to optimize the algorithms of H.264 encoders and decoders on Intel Pentium 4 processors. We first analyze the reference implementation to identify the time-consuming modules, and present optimization methods using media instructions to improve the speed of these modules. After appropriate optimizations, the speed of the codec improves by more than 3x. Nonetheless, the H.264 encoder is still too complicated to be implemented in real-time on a single processor. Thus, we also study how to partition the H.264 encoder into multiple threads, which then can be run on systems with multiple processors or multi-threading capabilities. We analyze different multi-threading schemes that have different quality/performance, and propose a scheme with good scalability (i.e., speed) and good quality. Our encoder can obtain another 3.8x speedup on a four-processor system or 4.6x speedup on a four-processor system with Hyper-Treading Technology. This work demonstrates that hardware-specific algorithm modifications can speed up the H.264 decoder and encoder substantially. The performance improvement techniques on modern microprocessors demonstrated in this work can be applied not only to H.264, but also to other video or multimedia processing applications.

Keywords: H.264, video codec, multimedia, MMX/SSE Technologies, SIMD, Hyper-Threading Technology, multi-threading

1. INTRODUCTION

H.264 [7] is an emerging video coding standard proposed by the Joint Video Team (JVT). The new standard is aimed at high-quality coding of video contents at very low bit-rates. H.264 uses the same hybrid block-based motion compensation and transform coding model as some existing standards, such as H.263 and MPEG-4 [8]. Furthermore, a number of new features and capabilities have been introduced in H.264 to effectively improve the compression efficiency. As the standard becomes more complex, the encoding and decoding processes require much more computation power than most existing standards. For example, the reference encoder on the state-of-the-art processors run at orders of magnitude slower than real time even for CIF sized video sequences.

While the complexity of emerging video codec (and many other multimedia applications) imposes new demands on processor performance, most modern microprocessors have multimedia instructions and multi-threading capabilities to facilitate multimedia applications. First, the single-instruction-multiple-data (SIMD) execution model was introduced to execute several computations in parallel with a single instruction. Examples include MMX and SSE Technologies [9, 12] in Intel architectures. Second,

* Yen-Kuang Chen is the corresponding author. Intel Corp, SC12-303, 2200 Mission College Blvd., Santa Clara, CA 95052. Phone: (408)765-8845. Fax: (408)653-8511. Email: yen-kuang.chen@intel.com
† Xiaosong Zhou is currently a Ph.D. student at University of Southern California.

simultaneous multi-threading [20] and chip-multi-processor [5] were introduced to enable a processor to execute multiple threads simultaneously. Examples include Hyper-Threading Technology in Intel architectures. Because multimedia applications tend to exhibit large amounts of computation with parallelism, we can also exploit the parallelism in the thread level besides executing multiple data in a single instruction. These advances in personal computers in addition to higher clock frequency have provided the necessary computation power for many multimedia applications.

Nonetheless, to implement multimedia applications on personal computers requires some hardware-specific algorithm modifications. In this paper, we analyze software implementation and optimize the algorithms of H.264 encoder and decoder on general-purpose processors with media instructions and multi-threading capabilities.

The contributions of this paper include:

(1) By analyzing the decoder and encoder software implementation, we identified (i) the time consuming modules of the H.264 codec on personal computers, and (ii) the hardware-specific algorithm modifications to implement those modules efficiently on the modern microprocessors.

(2) Using the latest MMX/SSE technologies, we speeded up the H.264 decoder and encoder significantly. While previously [2, 17] demonstrated how to use MMX technology for MPEG-4 decoder and H.263 encoder, this work is on H.264 specific modules.

(3) Using the latest multi-threading architectures with Hyper-Threading Technology, we speeded up the H.264 encoder by another substantial amount. Moreover, we have done an in-depth study on different tradeoffs in video quality and parallelization.

The novelties in this paper include:

(1) We implemented a sub-pixel interpolation procedure illustrated in Figure 3 (Section 3.3) to reduce unaligned memory loads. Our proposed scheme is 3x faster than the reference implementation.

(2) We designed a chain matrix multiplication procedure depicted in Figure 8 (Section 3.4) to avoid data re-arrangement for integer-transform implementation. Our SIMD matrix multiplication scheme is 4x faster than the reference implementation using the multiplication-free algorithm.

(3) We multi-threaded the H.264 encoder in a fine granularity (using a wavefront order as shown in Figure 14 in Section 4.3) and demonstrated good speedup with limited video quality degradation. While there are some works in parallel MPEG encoders [1, 3, 4, 22, 23], all of them are based on coarser granularities (either on group-of-picture level or slice level).

To our best knowledge, we are the first one in the literature that demonstrated the above [4, 14, 25]. Previously, [13] demonstrated algorithmic-level optimization to motion estimation of H.264. And, [18, 19] analyzed the complexity of H.264 decoder and encoder on personal computers.

The rest of the paper is organized as the following: Section 2 provides a brief introduction of the uniqueness in H.264 standard. Section 3 shows our SIMD optimization methods used to improve the speed. Section 4 discusses the design consideration of multi-threading implementation of the encoder, including performance and quality tradeoff. Section 5 discusses the required hardware-specific algorithm modifications to implement multimedia applications on the modern microprocessors. Finally, conclusions are presented in Sections 6.

2. UNIQUENESS OF THE H.264 STANDARD

H.264 has similar coding schemes as H.263 and MPEG-4. The standard includes the use of block-based motion compensation, DCT-like transform coding, scalar quantization, zigzag scanning and entropy coding. However, specific new concepts and features differentiate H.264 from the other standards. Here we briefly introduce a number of the unique features included in the baseline profile of H.264 coding standard.

First, the motion compensation model used in H.264 is more flexible and efficient than those in the early standards. Multiple reference frames for prediction is supported in the standard, and more choices of

motion compensation block sizes and shapes are provided for each macroblock (e.g., 16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4). High motion vector resolution is specified, where sub-pel interpolation could provide higher spatial accuracy at fractional positions. In addition, a well-designed in-loop deblocking filter is used to reduce visual artifact. The new methods provide a more precise model for motion compensation, which can dramatically minimize the impact of the difference of predicted blocks, and yield a much better perceptual quality for the decoded video stream.

Another uniqueness of H.264 is that the conventional DCT transform is replaced by a DCT-like integer transform. The transform can be implemented with additions and shifts only (therefore, so-called 'multiplication-free'), which is believed to have a significant computational advantage compared with DCT transforms. Moreover, the precise integer transform eliminates any mismatch between the encoder and the decoder [16].

The standard also allows flexible choices for slice size---even one slice per frame. For video streaming, the slice size is usually decided by the packet size of the network protocol. For other purposes, the whole frame can be coded into a single slice for better compression efficiency. In MPEG-2, baseline MPEG-4, or H.263, a picture must be divided into slices. This breaks the dependence between macroblocks. When a macroblock in one slice cannot exploit other macroblocks in another slice for compression, the compression efficiency decreases. H.264, without compulsorily breaking a frame into multiple slices, can offer better coding efficiency.

3. IMPLEMENTATION OF H.264 DECODER USING SIMD MEDIA INSTRUCTIONS

While today's H.264 reference code can decode approximately 30 frames of CIF images per second on state-of-the-art personal computers, it is an order of magnitude slower than a well-optimized MPEG-2 decoder [6]. This is because of two factors: (1) algorithmic complexities and (2) implementation optimality. In this section, we present optimization methods using media instructions to improve the speed of the H.264 decoder.

Section 3.1 briefly introduces the media instructions available on Intel microprocessors. Section 3.2 identified the time-consuming modules of H.264 decoder before exploiting the benefits of media instructions. Section 3.3 and Section 3.4 illustrate the techniques of speeding up two of the most time-consuming modules---motion compensation and integer transform. Section 3.5 shows that after appropriate optimizations, the speed of the codec improves by more than 3x.

3.1 Introduction of Intel SIMD instructions

The Single Instruction, Multiple Data (SIMD) execution model was introduced in the Intel architectures, with MMX technology and streaming SIMD extensions (SSE). The SIMD technology enables multiple arithmetic or logic operations to be executed simultaneously, in order to improve the speed of the program. This is realized by introducing longer registers (64-bit MMX registers and 128-bit SSE registers) and processing multiple data units stored in these registers simultaneously. For example, a 128-bit SSE register can be used to store up to 16 units of 8-bit integers, and thus up to 16 arithmetic operations can be executed simultaneously by using two SSE registers. This results in a significant speed improvement.

Nevertheless, there are restrictions on these SIMD instructions. While loading data into these longer registers, an unaligned data access can incur significant performance penalties. This is particularly true for cache line splits. The size of a cache line is 64 bytes in the Pentium 4 processor, and is 32 bytes in Pentium III processors. On the Pentium 4 processor, an access to data that are not aligned on 64-byte boundary leads to two memory accesses and requires several micro-operations to be executed instead of one. Memory accesses that span either 16 byte (but not 64 byte) boundaries are still likely to incur a large performance penalty. This is because they are executed near instruction retirement, and can incur stalls that are on the order of the depth of the microprocessor pipeline. For best performance, we should access data on natural operand size address boundaries, such as, access 64-bit data whose base address is a multiple of eight and access 128-bit data whose base address is a multiple of sixteen. Otherwise, the memory access is unaligned.

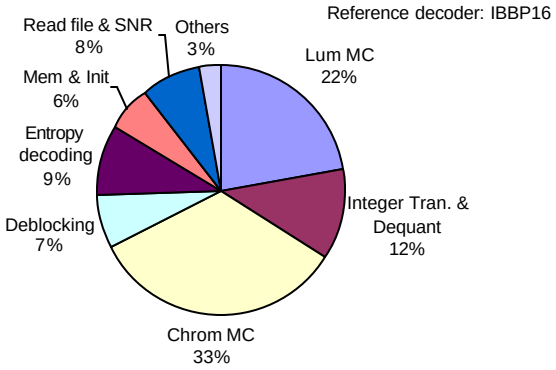


Figure 1: Time breakdown of H.264 reference decoder.

The rest of Section 3 demonstrates how to implement the H.264 decoder using the benefits of media instructions while circumventing the restrictions.

3.2 Identifying Time-Consuming Modules of the H.264 Reference Decoder

Our first step is to identify time-consuming modules in H.264 decoder. Figure 1 shows the breakdown of the execution time of the H.264 reference decoder on a 2.4GHz Pentium 4 processor with 512MByte memory. The on-chip first-level cache and second-level cache are 8K and 512Kbytes, respectively. The decoder runs under Microsoft Windows XP. In the experiment, we choose CIF-resolution foreman sequence with the IBBP structure. (The decoding rate is approximately 30.7 frames/s.)

From the diagram, it is easy to identify the following time-consuming modules in the decoder: luminance and chrominance motion compensation, inverse integer transform, entropy decoding and deblocking filtering. (Due to space limitation, the time breakdown for different test conditions is not included here. Readers can find the execution time profiles for two different bitrates from [25]. Basically, time consuming modules are roughly the same.) We are interested in implementing these most time-consuming modules using SIMD technologies in order to improve the speed of the decoder. In the following sub-sections, we will discuss implementation techniques in details.

3.3 Motion Compensation

Among many time-consuming modules in H.264, motion compensation is the most time-consuming one. As H.264 uses a ¼ pixel resolution for motion vectors and motion compensation, intensive computation is required for interpolating pixels at fractional positions in the H.264 decoder. As described before, H.264 uses seven different block sizes to perform motion estimation on each macroblock. For the interpolation process, all of the blocks are based on 4x4 block size. In this case, for SIMD implementation SSE registers could hardly be utilized fully. This is because only 4 pixels are calculated at one time. From a number of experiments, we have found that the total number of 4x4, 4x8 and 8x4 blocks used in motion estimation takes up only a small portion among different block sizes (~5% in P-frames and ~1% in B-frames). Therefore, we developed an efficient sub-pixel interpolation method for 8x8 blocks, while similar SIMD implementations can also be applied to macroblocks for 4x4 blocks. This approach makes full use of the 128-bit register, and improves the speed for pixel interpolation during both luminance and chrominance motion compensation.

3.3.1 Luminance Motion Compensation and Filter Design

In H.264 decoder, a 6-tap FIR filter is implemented for ¼ pixel interpolation. The filter coefficients are constants and pre-stored. The interpolation process conducts vector product between the coefficient

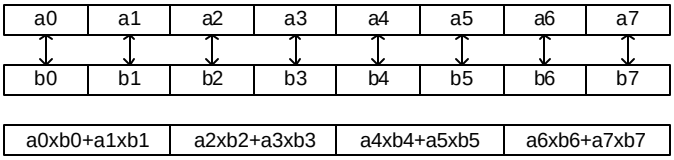


Figure 2: PMADDWD instruction.

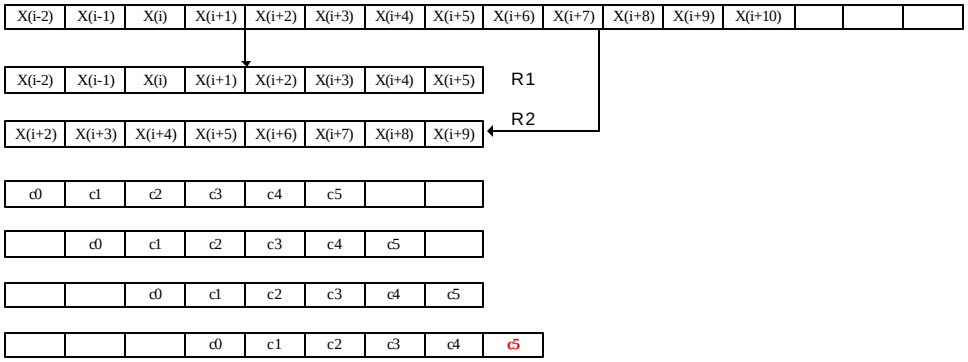


Figure 3: FIR implementation of sub-pel interpolation.

vector and the image data vector. An SSE instruction PMADDWD [10] can facilitate this implementation. Figure 2 shows how the instruction PMADDWD works: the instruction will take two 128-bit SSE registers as its operands, where each data unit must be 16-bit integers. Eight multiplications will be performed on the pairs of 16-bit integers simultaneously. And, the results in pairs are added together and stored in one SSE register as 32-bit integers. Therefore, the vector product mentioned above can be implemented by loading the image data vector and the filter coefficient vector into two SSE registers and performing PMADDWD instruction on them, then summing up the four 32-bit data units in the resulting register to get the final result.

It seems that a different image data vector must be loaded each time while calculating a new pixel. For interpolating an 8x8 block in one dimension, the vectors must be loaded 64 times. Because most of these data loading will be unaligned loads[‡], the loading penalty and overhead will be huge. Thus, it will be more desirable to reuse the previously loaded image data in order to avoid the abundant loading operations.

Here is our scheme for row interpolation. We can first load 6+8-1 =13 relative pixels into a 128-bit register (as the image data are stored in bytes). We then unpack and shift them into two SSE registers (R1 and R2) with required 16-bit length as shown in Figure 3. Since the filter coefficients must be shifted along the row, we can load 4 different copies of the filter coefficients with different shift phases, and then perform PMADDWD instruction on R1 and the four copies of the filter coefficients to get the first 4 pixels. Please note that while calculating the fourth pixel, since the coefficient c5 (equals to 1) can't fit into the fourth coefficient register, one more addition operation will be performed on x(i+6) to calculate the fourth interpolated pixel separately. By using the same approach, the next four pixels can be calculated using R2. As this method avoids penalties introduced by repetitive unaligned loads and reduces the data loading overhead, it greatly outperforms the original method.

For column interpolation, simple packed shifts and additions can facilitate the interpolation process. The filter coefficients are (1, -5, 20, 20, 5, 1), which can be simply substituted with shifts and additions (e.g. 20=2^4+2^2). Therefore, six rows of data will be first loaded into six 128-bit registers, followed by paced shifts and additions to calculate the final result. Because the SSE registers can hold eight 16-bit data, we can calculate 8 interpolated pixels in a row. The advantage of 8x8 block size is shown in this module,

[‡] Section 3.1 describes the problem of unaligned memory accesses.

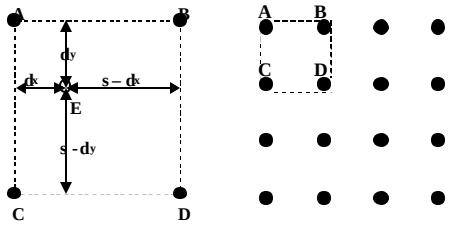


Figure 4: Chrominance motion compensation

compared with interpolation on 4x4 blocks, nearly 50% computations are saved during the column interpolation stage.

3.3.2 Chrominance Motion Compensation

Unlike many previous video coding standards, chrominance motion compensation occupies a significant percentage in the total motion compensation process of the H.264 reference decoder. As shown in Figure 4, the interpolated chrominance pixel E is derived from the 4 corresponding pixels (A, B, C, D) with the formula:

$$E = \left((s-d_x)(s-d_y)A + d_x(s-d_y)B + (s-d_x)d_yC + d_xd_yD + s^2/2\right)/s^2$$

where A, B, C and D are the integer position reference picture samples surrounding the fractional sample location, and d_x and d_y are the fractional parts of the sample position in units of one eighth samples for quarter sample interpolation. Because d_x , d_y, (s- d_x), and (s-d_y) stay the same in the same block, we can calculate the coefficients (s- d_x)(s-d_y), d_x(s-d_y), (s- d_x)d_y, and d_xd_y in advance. After that, we can simply use SIMD instructions on multiple pixels (e.g., 4x4 block) to compute the interpolated chrominance pixels with similar idea described in Section 3.3.1.

3.4 Inverse Transform and Matrix Multiplication

Instead of using DCT/IDCT, H.264 uses a 4x4 integer transform to convert spatial-domain signals into frequency-domain and vice versa. Figure 5 shows the mathematical form of the inverse integer transform in H.264 decoder, which can be calculated using matrix multiplications. Since the first and the

$$R = \begin{bmatrix} 1 & 1 & 1 & 1/2 \\ 1 & 1/2 & -1 & -1 \\ 1 & -1/2 & -1 & 1 \\ 1 & -1 & 1 & -1/2 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{10} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/2 & -1/2 & -1 \\ 1 & -1 & -1 & 1 \\ 1/2 & -1 & 1 & -1/2 \end{bmatrix}$$

X **A** **B**

Figure 5: Inverse integer transform in H.264 decoder.

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} \\ r_{10} & r_{11} & r_{12} & r_{13} \\ r_{20} & r_{21} & r_{22} & r_{23} \\ r_{30} & r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{10} & y_{11} & y_{12} & y_{13} \\ y_{20} & y_{21} & y_{22} & y_{23} \\ y_{30} & y_{31} & y_{32} & y_{33} \end{bmatrix}$$

Figure 6: 4x4 matrix multiplications.

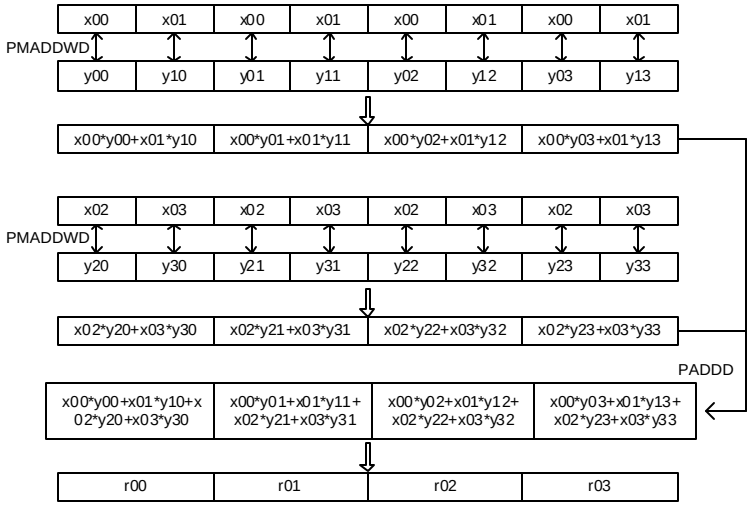


Figure 7: Efficient 4x4 matrix multiplication implementation.

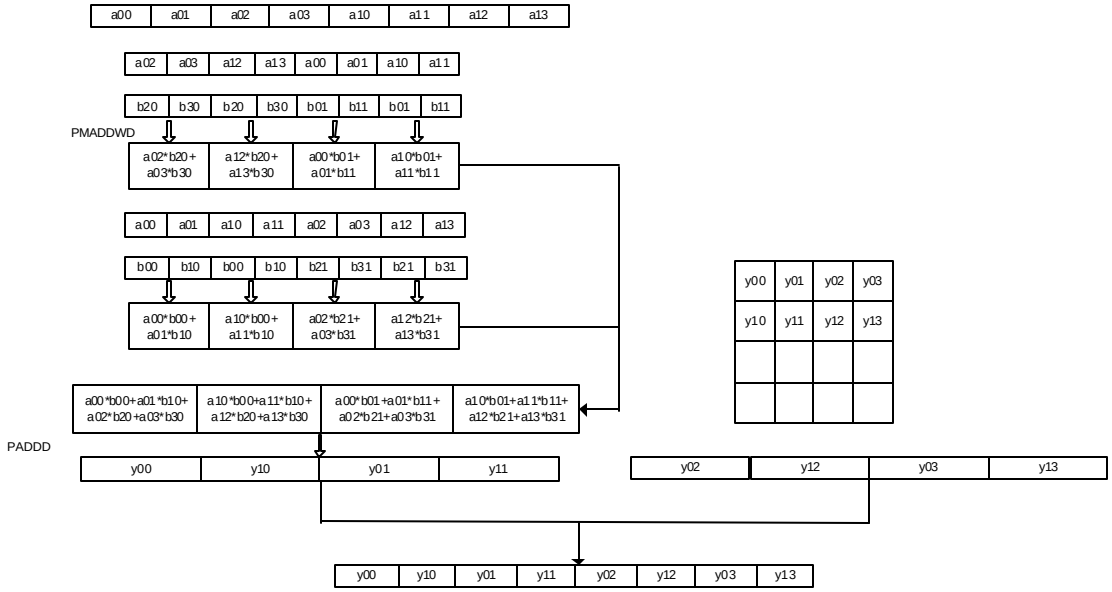


Figure 8: Chain matrix multiplications.

last matrices have only constant coefficients of ±1 and ±½, it is possible to implement the calculation by using only additions, subtractions, and shifts. This 'multiplication-free' method is quite efficient and, thus, has been implemented in the reference code. Nonetheless, it is hard to further optimize it by utilizing SIMD instructions because the operations on each operand are different. However, the original form of the integer and inverse integer transform are 4x4 matrix multiplications, which can be implemented by using SIMD instructions, hopefully yielding better performance than the 'multiplication-free' method.

Figure 6 shows a typical 4x4 matrix multiplication. As PMADDWD can be used to optimize the operation of vector products in the FIR filtering, we can store a row vector of X and a column vector of Y into two registers, and then use PMADDWD to produce the results. However, this simplest scheme is not the optimal. First, to calculate each element after PMADDWD instruction, we must shift the result,

retrieve, and add the 32-bit integers 3 times. Also, it is costly to load column vector of matrix Y into an SSE register. There will be overhead for these operations. Thus, we must utilize the structure of matrix multiplication to avoid this problem. The following shows an efficient matrix multiplication method that can be completed without shifting or abundant loading operations.

Figure 7 shows an example of computing the first row of the resulting matrix. Matrix X and Y are loaded into SSE registers in a particular form. The loading process can be realized efficiently with combinations of SIMD instructions. After that, PMADDWD is performed on these registers in pairs. Then the two resulting 128-bit data are added together in the 32-bit integer precision, hence the first row of the resulting matrix is obtained right away. For the next three rows, similar methods will be utilized, in this case there is no need to reload matrix Y, and the final results are well aligned in rows that do not need any additional operations to rearrange their orders. This method can be applied for multiple 4x4 matrix multiplications, and it can also be revised to fit for matrix multiplication of other sizes.

The realization of our H.264 integer transform extends the above general approach. We prepare the data in advance so that the result of the first matrix multiplication is in order for the second matrix multiplication to use immediately. In this case, no extra data re-arrangement is needed in the middle of the processing. Moreover, as shown in Figure 5, the first and last matrices in the equation are constant. Thus, minimal extra cost to pre-arrange data in the right format.

We first multiply the last two matrices A and B together and store the result in matrix Y. Next, we multiply the first matrix X with the matrix Y to get the final resulting matrix R. Because we have pre-arranged the data before the multiplications, the results of the multiplications and additions are in order during the chain matrix multiplication process. Figure 8 illustrates the whole inverse integer transform process. First, we load matrix A as 16-bit integer operands into an SSE register and shuffle their positions to the right order. Second, we load the first two columns in matrix B into another two registers, where the data are all constant and have been prepared in advance. After that, PMADDWD instruction will be performed on these two registers in pairs. Therefore, we get y_{00}, y_{10}, y_{01}, and y_{11} in the first SSE register. Similar operations will be utilized on the last two columns in matrix B to calculate the temporary results y_{02}, y_{12}, y_{03}, and y_{13}. In this case, we can use a pack operation on these two registers to form the final result, and their positions are shown in the matrix of Figure 5. Now, the resulting register contains the data just in the right order for the next step of matrix multiplication. Because we have considered the data position ahead of time, there will no need to further rearrange their positions during the processing. This reduces the middle-processing time compared with the original method.

Table 1 shows a comparison of three different implementations for the inverse transform. It shows the time used by 50 million runs of each of the programs. It is clear that the 'Multiplication–Free' method implemented in the reference code outperforms the original matrix multiplication implementation substantially. However, with the method described above, we achieve another 4x speedup.

Table 1: Comparison of different implementations on inverse transform.

Implementation Method	Time Used (seconds)	Speedup from original matrix multiplications	Speedup from multiplication-free
Original Matrix Multiplication	8.22	-	0.3
Multiplication-Free	2.41	3.4	-
Matrix Multiplication with SSE	0.56	14.7	4.3

Table 2: Kernel module speedup of H.264 decoder and encoder

Decoder Modules	Speedup
Luminance Motion Compensation	2.9x
Chrominance Motion Compensation	10.2x
Integer Transform	4.3x
Deblocking Filter	1.1x

Encoder Modules	Speedup
SAD calculation	3.5x
Hadamard Transform	1.6x
Sub-Pel Motion Search	1.3x
Integer Transform and Quantization	1.3x
Quarter Pel Interpolation	2.0x

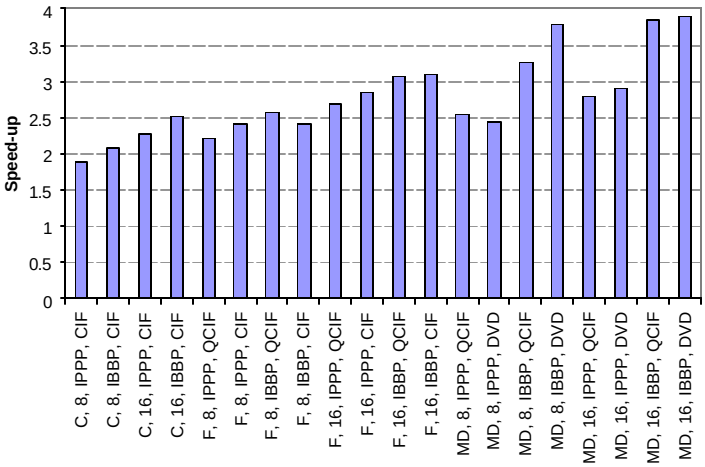


Figure 9: Decoder speedup on different video streams. The first parameter in the x-axis label is the video sequence (C: coastguard, MD: mother & daughter, F: foreman). The second parameter is the quantization step. The third parameter is the group-of-picture structure. The fourth parameter is the video resolution.

3.5 Speedup of SIMD-Optimized H.264 Decoder

As a result of our implementation described above, the optimized H.264 decoder is about 2.5~3 times faster than the reference decoder.

Table 2 shows the speedup for the kernel modules in the decoder, where chrominance motion compensation gets a remarkable improvement mainly as a result of our efficient implementation on sub-pixel interpolation. (Similar techniques can be applied to the encoders as well. Table 2 also shows the speedup for each key module residing in H.264 encoder.)

Figure 9 shows the speedup of the optimized decoder on different encoded video streams. It is clear that for video streams with higher bit-rates (such as Coastguard, and video streams with smaller quantization steps), it is harder to achieve large speedup. This is due to the fact that the entropy decoding process, which will have a much larger impact on the system speed, is one of the modules not yet optimized in our work. Video streams in IBBP modes achieve better speedup because the motion compensation module in their decoding process is more critical than video streams in IPPP mode.

4. MULTI-THREADED IMPLEMENTATION OF H.264 ENCODER

As Section 3 demonstrated that using MMX/SSE technologies can speed up the H.264 decoder by 2~4x, we applied the similar techniques to the H.264 reference encoder and achieve 2~3x speed improvement. Additionally, similar to many other works in the literature [13], we implemented fast motion estimation and mode decision algorithms on top of the SIMD-optimized reference encoder. As motion estimation and mode decision are the performance bottlenecks, we got another 2~4x speedup after algorithmic optimization.[§] While it is much faster than it was, the speed of the encoder is still too slow on a

[§] There are a number of existing techniques for algorithmic optimization, e.g., [21, 26]. We implemented an 8~15x faster motion estimation algorithm, which is a hierarchical multi-hexagon motion search scheme with predictor selection and prediction mode reordering [15]. Additionally, we implemented a 7~10x faster 4x4 mode decision algorithm, which utilizes the edge directional information of the video content to reduce the intra prediction modes. Furthermore, in rate-distortion optimization criterion, we just calculate the SATD value to decide the best intra prediction mode. Afterward, our SIMD-optimized and algorithm-optimized encoder is 2~4x faster than the SIMD-optimized encoder, and the degradation of video quality is ~0.06dB at the same bitrate.

single processor. Even with SIMD media technology and algorithmic optimization, it only encodes about 1 frame per second for CIF-resolution sequences on state-of-the-art personal computers. This means that there is a lot of room for us to continue improving the speed of the H.264 encoder. In this section, we demonstrate how to partition the H.264 encoder into multiple threads, which then can be run faster on systems with multiple processors or multi-threading capabilities.

Section 4.1 briefly introduces Intel's simultaneous multi-threading technology. Section 4.2 discusses possible parallelism available in the H.264 encoder and analyzes different multi-threading schemes that demonstrates different quality/performance. Section 4.3 depicts our proposed fine-granularity parallel implementation, which is with good scalability (i.e., speed) and good quality. Section 4.4 shows that our multi-threaded encoder has 3.8x speedup on a four-processor system or 4.6x speedup on a four-processor system with Hyper-Threading Technology.

4.1 Introduction to Hyper-Threading Technology

Recently, processors/systems that can run multiple software threads have received increasing attention as a means of boosting overall performance. This is because multimedia applications tend to exhibit large amounts of computation and parallelism. We can also exploit the parallelism in the thread level besides executing multiple data in a single instruction.

In 2002, Intel introduced Hyper-Threading Technology, which runs multiple threads simultaneously, into Intel® Xeon™ processors and Pentium® 4 processors [20]. In a processor with Hyper-Threading Technology, one physical processor exposes two logical processors. Similar to a dual-core or dual-processor system, a processor with Hyper-Threading Technology appears to an application as two processors. Two applications or threads can be executed in parallel. The major difference between processors with Hyper-Threading Technology and dual-processor systems is the different amounts of duplicated resources. In processors with Hyper-Threading Technology, only a small amount of the hardware resources are duplicated, while the front-end logic, execution units, out-of-order retirement engine, and the memory hierarchy components are shared. Thus, compared to processors without Hyper-Threading Technology, the die-size is increased by less than 5% [20].

As Intel Corporation and many other companies are making future processors capable of multi-threading [5], the rest of Section 4 demonstrates how to improve the speed of H.264 encoder on systems with multi-threading capabilities.

4.2 Possible Parallelism vs. Quality

Before we determine the multi-threading scheme, we should decide the thread granularity. A video sequence typically is composed of group of pictures (GOP), which consists of frames. Frames can further decomposed into slices, which contains macroblocks. The most straightforward approach to encoding the video sequences is by GOPs. This is because each GOP acts independently from each other [1, 22]. Another approach is based on slice level. This is because slices are self-content and independent of other slices [23]. These decompositions are naturally the opportunities in parallelizing the H.264 encoder. While these places are good opportunities for parallelism, these places may not be the best places. For example, GOP-level parallelism is simple, but multi-threading at the GOP-level parallelism incurs a very long latency.

4.2.1 Slice-Level Parallelism

One possible scheme of decomposition is to divide a frame into small slices. The advantage of parallelizing among slices is that the slices in a frame are independent. Thus, we can simultaneously encode all slices in any order. However, the disadvantage is that (different from MPEG-2 standard) slices in H.264 may significantly increase the bitrate at the same video quality. Figure 10 shows the compression efficiency of the video encoder (rate-distortion) when a frame is divided into different numbers of slices. When a frame is divided into 9 slices, the bit-rate at the same quality is about 10~15% higher. This is because slices break the dependence between macroblocks. When a macroblock in one slice cannot exploit another

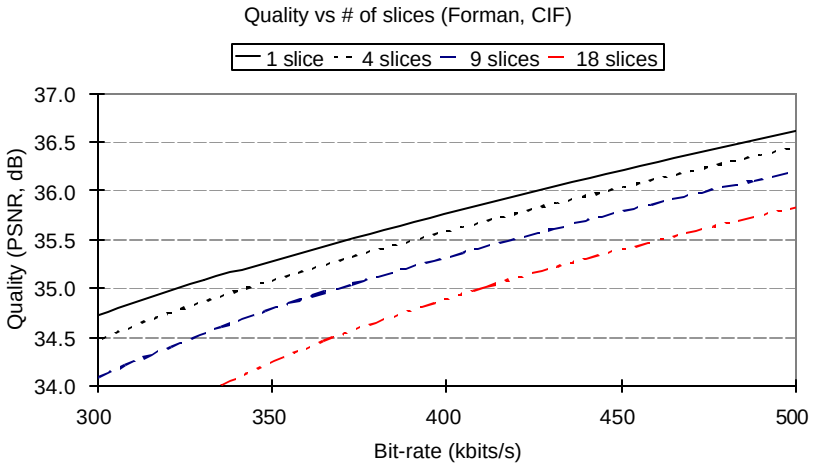


Figure 10: Encoded picture quality vs the number of slices in a picture.

macroblock in another slice for compression, the compression efficiency decreases. In order not to increase the bit-rate at the same video quality of the parallelized encoder, we should exploit other parallelism in the video encoder.

4.2.2 Frame-Level Parallelism

Another possible scheme of exploiting parallelism is to identify independent frames. Normally, we encode a sequence of frames using an IBBPBBP… structure. While some frames (e.g., P frames) are reference frames for others, some frames are not necessary. To increase parallelism, we treat B frames as non-referenced frames in our implementation of H.264 encoder. The dependence among the frames is showed in Figure 11. In this PBB encoding structure, the completion of encoding a P frame will make the subsequent one P frame and two B frames ready for encoding. The more frames encoded simultaneous, the more parallelism we can explore. Therefore, the P frame is on the critical point in the encoder. Accelerating P-frame encoding will bring more frames ready for encoding, and avoid the idle of threads. In our implementation, we will encode I or P frames first, then B frames.

Unlike dividing a frame into slices, utilizing parallelism among frames will not increase the bit rate. However, the dependence among them will limit the threads scalability. One scheme is to combine the above two approaches into one implementation. First, the parallelism among frames is explored; we can gain performance from it without bit rate increase. After we reach the upper limit of the number of threads in the frame-level parallelism, the parallel among slices is explored subsequently. As a result, processor resources is utilized as much as possible and the compression ratio is kept as high as possible (i.e., the bit-rate as low as possible).

4.2.3 Performance Analysis

Because dividing a frame into multiple slices increases not only the parallelism, but also the bit rate, we should carefully choose the number of slices per frame. Figure 12 shows the relationship of the execution time and the bit rate vs the number of slices in a frame. We change the number of slices from 1 to 18, while keeping the encoded picture quality constant. On a dual-processor system, the execution has a big speedup when the number of slices in a frame changes from 1 to 2, but nearly keeps unchanged while the number of slices changes from 2 to 18. Meanwhile, the bit-rate increase is small if the number of slices is smaller than 3, but starts to become larger after 3 slices. In this case, dividing a frame to 2 or 3 slices is a good-enough trade-off point between the bit rate and the execution time. On our 4-processor system with Hyper-Threading Technology, we need much more than 3 slices to keep 8 logical processors busy. We need 9 threads to achieve roughly the best performance. In this case, the bit-rate increases more than 10%.

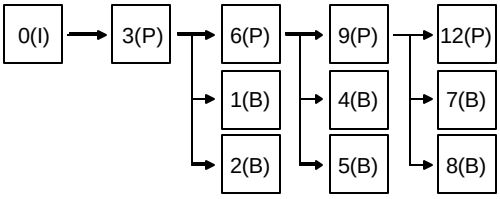
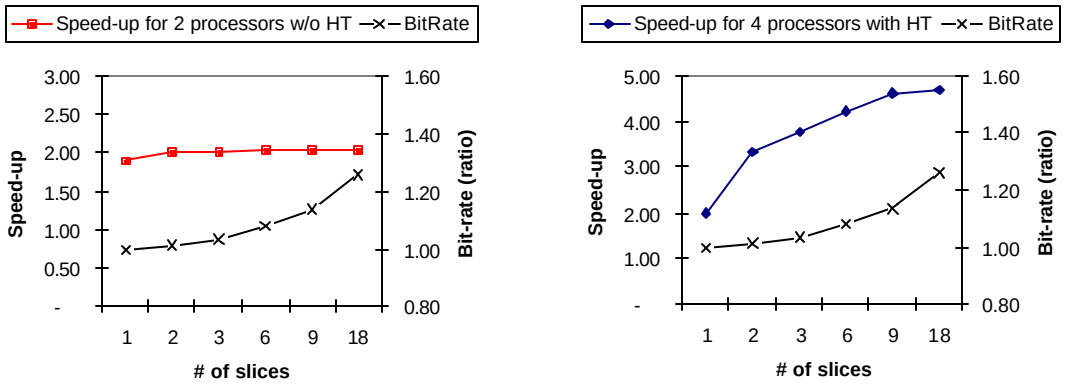


Figure 11: Data dependence among frames.



(a) (b)

Figure 12: Speedup and bit rate vs the # of slices in a frame

In order not to increase the bit-rate at the same video quality of the parallelized encoder, we should exploit other parallelism in the video encoder.

4.3 Our Fine-Granularity Parallel Implementation of H.264 Encoder

As illustrated in Section 4.2, slice-level parallelism is not an optimal choice for the multi-threading of H.264 video encoding. We should find some finer data partitioning, i.e., macroblock-level parallelism, to provide enough parallel capabilities without any video quality losses. Our work is inspired by [24], where a fine-grained parallel decoder implementation is demonstrated. Data partitions used are basically built from the macroblock level.

4.3.1 Macroblock-Level Parallelism

The current coding macroblock has dependencies on its adjacent left, upper-left, upper, and upper-right macroblocks, as shown in Figure 13.

(1) Intra prediction: Pixels in the current macroblock may be predicted from pixels of its left, upper-left, upper, and upper-right macroblocks.
(2) Motion vector prediction: Motion vectors are predicted from those of its left, upper-left, upper, and upper-right macroblocks.
(3) Deblocking filter: filtering is performed on the top 4 rows of pixels and leftmost 4 columns of pixel of the current macroblock, with the bottom 4 rows of the upper macroblock and the rightmost 4 columns of the left macroblock.

Due to the data dependencies described above, macroblocks can only be encoded after the adjacent partitions are already encoded.

The spatial dependencies within one frame imply a specific processing-order restriction. For example, the top-left macroblock should be processed at first, and other macroblocks must wait because they have data dependencies on it. On the other hand, after the second macroblock has been encoded, there are possibilities to select either the top-right or bottom-left macroblock for encoding.

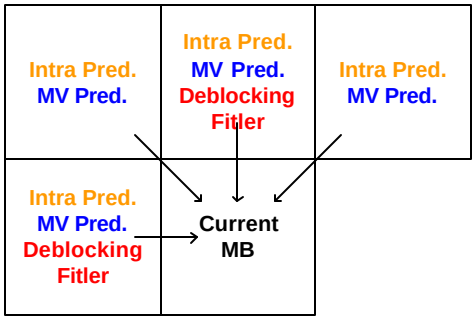


Figure 13: Possible spatial data dependencies for a macroblock

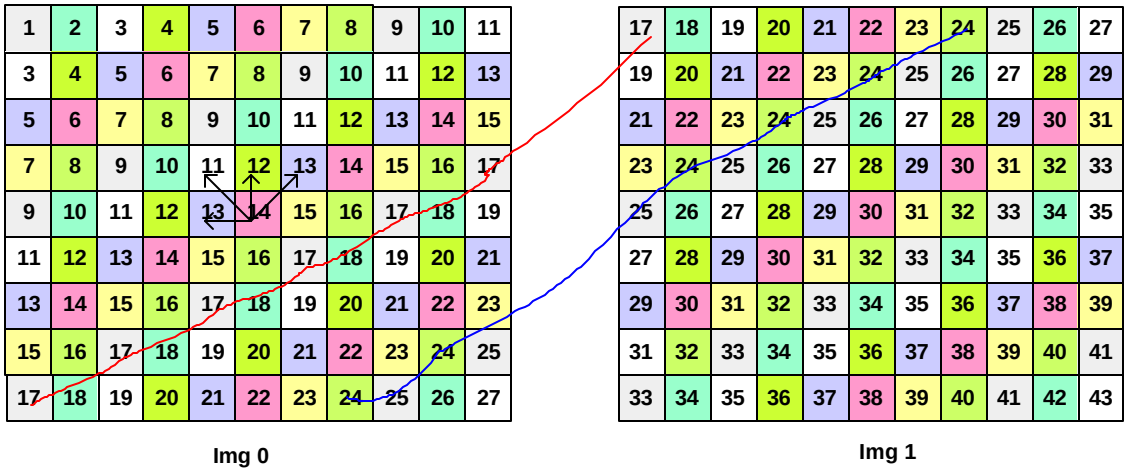


Figure 14: Macroblock-level dependency scheme and coding model

Figure 14 shows our macroblock data-partition scheme. Let the horizontal and vertical numbers of macroblocks in a frame be *w* and *h*, and *MB(i)* denote the i-th macroblock in the frame (in the raster scan order). According to the restriction of data dependencies in a frame, *MB(i)* and *MB(i+w-2)* can be encoded simultaneously and thus they have the same time stamp. *MB(w+2)* in the second row requires the information from *MB(1)*, *MB(2)*, *MB(3)*, and *MB(w+1)*. At the same time, *MB(4)* can be encoded as well. In general, the scheduling of MB encoding should be like this: {*MB(1)*}, {*MB(2)*}, {*MB(3)*, *MB(w+1)*}, {*MB(4)*, *MB(w+2)*}, {*MB(5)*, *MB(w+3)*, *MB (2w+1)*}, …, {*MB((h-1)*w)*, *MB(h*w-2)*}, {*MB(h*w-1)*}, and {*MB(h*w)*}. Numbers in Figure 14 indicate the time stamp.

4.3.2 Inter-Frame Parallelism

While the maximum number of simultaneous threads in this scheme is *(w+1)/2* (e.g., 11, 23, and 60 threads for CIF, SD and HD video), there are only limited amount of speedup if we only exploit the concurrency within one frame. This is because parallelism is dramatically limited at the beginning and at the end of encoding a frame, where only one or two macroblocks can be simultaneously encoded. In this case, it is hard for the encoder to gain good performance when the number of available hardware threads is larger. For instance, for CIF images, the speed of 8-thread encoders over that of 4-thread encoders is very small (31/27 = 1.15).

In order to improve the whole paralleling performance of video encoding, temporal dependencies should also be considered. As shown in the right-hand of Figure 14, the top-left most macroblock of Img1 can be predicted from the quarter-pel reconstructed Img0. However, it cannot be immediately encoded when the first row of Img0 have been processed. This is because of the range of motion estimation/compensation is normally larger than a couple of macroblocks. To accurately

estimate/compensate the motion from the predicted frames, we must wait until the encoded macroblocks are reconstructed after deblocking filter and quarter-pixel interpolation. If the motion search range is twice as large as macroblock size, the first top-left macroblock in Img1 cannot be encoded until the third row of Img0 is coded. After that, the macroblocks in the first row of the Img1 can be encoded simultaneously together with some macroblocks in Img0. The line with time stamp 17 in Figure 14 demonstrates that the first macroblock in Img1 can be concurrently encoded with the other six macroblocks in Img0. By adding inter-frame parallelism, our final scheme provides more parallelism in H.264 encoder.

4.3.3 Our Multi-threading Implementation of H.264 Encoder

Our multi-threading scheme has N+1 simultaneous threads when there are N processors available. A main thread handles the pre- and post-image processing tasks. Other N threads act as the working threads to encode the macroblocks between these two adjacent images alternatively. As shown in right-hand side of Figure 15, the image pre-processing module reads in the raw image data and assigns the parameters accordingly. The image post-processing module checks the encoding status, generates the output bitstream and fulfills quarter-pel image interpolation and reconstruction works. Image pre-processing and image post-processing procedures must be handled sequentially to guarantee the correctness. Hence, we assigned one main thread to handle these. The main thread takes 2%~5% of the CPU time. The majority of encoding time is spent on macroblock encoding process, dealing with the motion estimation and mode decisions functions to select the best coding mode for each macroblock. Thus, there are N threads to handle the left-

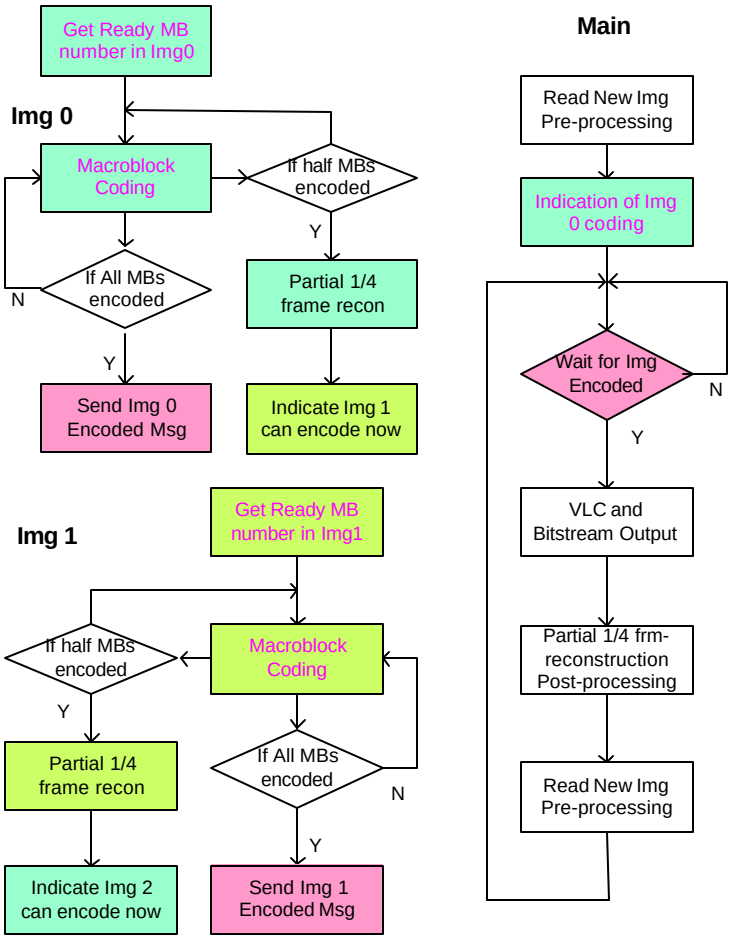


Figure 15: Outline of the proposed multi-threading scheme

hand side of Figure 15.

Here are some details of how it actually works when the encoded GOP structure is IPPP. If bi-directional frames are utilized, we can have even more parallelism.

(1) The raw data of Img0 is read in and pre-processed in the main thread. After that, Img0 is ready to be encoded.
(2) Once the working threads get the ready message, they select candidate macroblocks and encode them subsequently. The macroblocks are encoded in the wavefront order as described in Section 4.3.1.
(3) While Img0 is being encoded, reconstructed images should be prepared in advance for temporal predictions for Img1. If half of Img0 have been encoded, these encoded macroblocks are used to perform partial deblocking filtering, partial quarter-pel interpolation, and partial frame reconstruction.
(4) When these preparations are finished, Img1 and Img0 can be concurrently encoded.
(5) The working threads encode these two adjacent images according to the following sequential order: If there is no more ready macroblock to be encoded in Img0, the ready macroblock in Img1 will be encoded.
(6) When all the macroblocks in Img0 have been encoded, the main thread will take over the macroblock encoding process and start the post-encoding procedure---generating the final VLC & output bitstream and finishing the residual deblocking filtering, quarter-pel interpolation, & frame reconstructions.
(7) After (6), Img0 is released. Another new image is read and pre-processed.

The above procedure continues until all candidate images are encoded.

4.6 Speedup of Multi-Threaded H.264 Encoder

This section summarizes speed improvements of our multi-threaded H.264 encoder. We compare the encoding execution time of our encoder on a first uni-processor system and the first uni-processor system with Hyper-Treading Technology as well as a second system with one processor, the second system with four processors, and the quad-processor system with Hyper-Threading Technology.

Here is our experimental set up. Our first single-processor system has a 3.06GHz Intel Pentium 4 processor with a 512KB second-level cache. Additionally, the second system has four 2.8GHz Intel Xeon processors, each of which has a 256KB second-level cache and 2MB third-level cache. In this case, we can have as many as 8 logical processors. To measure single-thread performance on the quad-processor system, we disable the other three physical processors and run a single-thread version of the application. In our experiments, we use a variety of test sequences, covering from CIF to HD resolution. Additionally, we choose high-motion as well as low-motion sequences to make the experiments more comprehensive. The simulations presented here are performed with IPPP GOP-structure, i.e., no B frames. Search range has been set to two macroblock size. Rate-distortion-based mode decision is utilized in the experiments.

Figure 16 summarizes the average speedup. Nearly optimal linear speedup has been achieved for the four-processor platform. With Hyper-Threading Technology, an average of 1.18x speedup has been obtained, which is more than the 5% extra cost in chip area to implement Hyper-Threading Technology. However, when comparing the performance of four-processor platform without and with Hyper-Threading Technology, we observe that CIF format sequences only get slightly lower than the normal Hyper-Threading speedup. According to the detailed time profile for each thread, the problem comes from the synchronization overhead and load imbalance, especially (1) during the starting up or ending encoding process, and (2) for high-motion sequences. This is because (1) the size of CIF format image is comparatively smaller for eight simultaneous working threads, and (2) the computational loads have higher variance in high-motion sequences. Therefore, it is very crucial to efficiently allocate and balance the computation loads for small-resolution and high-motion sequences.

In short, our multi-threaded encoder has 3.8x speedup on a four-processor system or 4.6x speedup on a four-processor system with Hyper-Threading Technology.

5. DISCUSSIONS

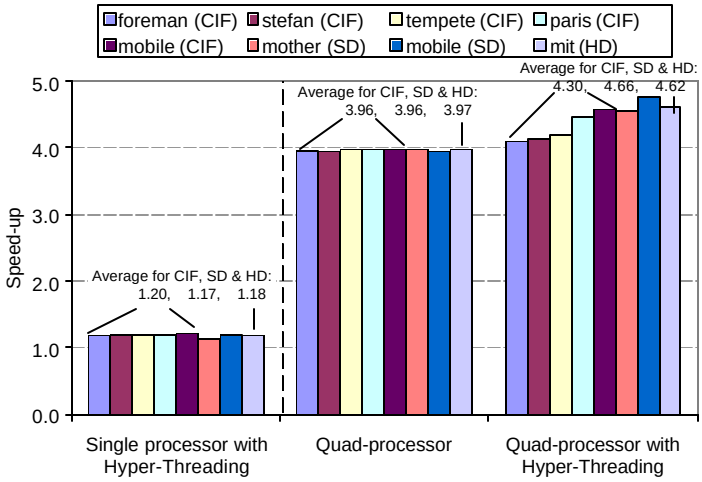


Figure 16: Speedup on different platforms

In the process of optimizing H.264 decoder and encoder, we discovered a few interesting points regarding efficient video-processing application design, with consideration of the characteristics of general-purpose processor architectures with media instructions and with multi-threading capabilities. The performance improvement techniques demonstrated in this work can be applied not only to H.264, but also to other video, image, or multimedia processing applications.

5.1 Algorithm Simplification and Optimization

In many literatures, the way to estimate the computational complexity of a given algorithm is to count the number of different types of operations needed. Algorithms were designed in a way to minimize the number of required operations. Under the newly developed processor architectures, this criterion remains critical but not as accurate. Algorithms with larger number of operations but simpler program flows may have a better performance.

For instance, in the H.264 decoder, the new integer transform were designed such that it can be implemented by using only additions, subtractions and shifts. The number of multiplications, which was considered one of the most time-consuming operations, was reduced to zero. However, as we described in Section 3.4, this implementation makes it difficult for further SIMD optimization. On the contrary, by implementing the simple traditional matrix multiplication with SIMD instructions and utilizing special implementation techniques, the integer transform actually runs faster.

As the current and future processors can process multiple data units simultaneously, computational complexity should be counted as the number of operations on the sets of these data units. This will cause a major impact on algorithm design, because different algorithm settings may have the same computational complexity if their difference does not exceed the size of these data sets.

Here is an example: The PMADDWD instruction can be used to perform FIR filtering efficiently, just like what we did in Section 3.3. As each SSE register can store up to eight 16-bit integers, as long as the filter length does not exceed eight, we can use one SSE register to store the filter coefficients, and another one to store the data to be filtered. Then eight multiplications can be processed simultaneously and the program efficiency is greatly improved. Therefore, an FIR filter of length 4 and that of length 8 will have the same computational complexity since they are processed exactly the same way in this approach. However, the two filters may provide quite different effects because their lengths are different. (Normally, filter of longer length provides better results. As a filter of length 4 and a filter of length 8 have the same complexity, using the filter of length 8 should be a better choice, right?) Same rule applies when we have

longer filters, where two or more SSE registers each will be used to store the coefficients and the data. For example filters of length 9 to 16 will have the same computational complexity.

As these newly developed processors become standardized, in order to obtain the best possible performance with given timing and cost constraints, algorithm designers must consider the new features in the processors and tailor algorithms accordingly.

5.2 Data Structure and Memory Operation

Data structures and alignments are critical for SIMD instructions. On Intel architecture, data units to be loaded must be 16-byte aligned or there will be penalties. Therefore, the programmer or system designer must be more careful while allocating memory spaces in order to avoid unaligned loads. While Intel's SSE3 technology introduces a new instruction, LDDQU, that significantly improve 128-bit unaligned memory accesses [12], it is generally not recommended unless we cannot avoid such unaligned memory accesses in the algorithm. In [11], it is shown that the performance of the block-matching motion estimation algorithm is heavily penalized because of its large number of unaligned memory accesses. Those unaligned memory accesses can hardly be avoided because of the nature of the algorithm. Because the new instruction in Intel's SSE3 technology can improve the performance of such unaligned memory access, the block-matching implementation with the new instruction is 1.37x faster than the implementation without it. However, it is noted in [12] that there are situations where the new instruction may be slower than the existing instructions. That is, the instruction should be used with care. Hence, in general, we recommend transforming the algorithms to avoid unaligned memory accesses (for example, Section 3.3.1) first. If the algorithm cannot avoid such unaligned memory accesses, we use the new instruction.

As for video processing, normally image data are stored in large matrixes. Since video decoding will process the data in blocks with the size of multiples of 4, if we align the first element of the matrix to a 16-byte aligned address, and store the whole matrix consecutively, a large portion of the operations on the matrix will benefit because their data accesses will already be properly aligned. Also if SIMD implementation is desired, the length of data units to be processed simultaneously must be identical.

5.3 Data Dependence and Parallelism

Complicated data dependence makes it harder to parallelize the decoder/encoder. On one hand, because modern processors are supporting multiple threads simultaneously, one way to making the application run faster is to exploit the parallelism. On the other hand, the application may have complicated data dependence, which is required for higher quality. Either we have to trade off between parallelism and quality, or we have to design a much more complicated parallel scheme. For example, because breaking a frame into slices in H.264 degrades the compression efficiency substantially, the scheme in [4] trades the quality for speed. For another instance, in Section 4.3, we proposed the macroblock-level decomposition, which provides good execution speedup without losing video quality. Nonetheless, our scheme needs 10~30x more synchronizations per second than the slice-based scheme. For future algorithm designs, we should consider potential parallelism available in the microprocessors, and avoid introducing complicated data dependences.

6. CONCLUSIONS

While state-of-the-art codecs are getting more complicated and demand more computational capabilities than before, personal computers are getting faster. In this paper, we have studied and optimized the performance of H.264 encoder and decoder on Intel Pentium 4 processors. This work demonstrates that, after appropriate performance optimizations with SIMD instructions and with multi-threading technology, the encoder/decoder gained significant performance improvements. Thus, it is easy to implement high-performance multimedia applications on personal computers without installing any additional special hardware.

While we demonstrated the optimization strategies for H.264 encoder/decoder on Intel architecture, the methodology can be extended in two directions. First, the performance improvement techniques can

also be applied to other video or multimedia processing applications. Second, many of the optimization strategies can also be applied to other platforms.

For best quality and performance, it is important to understand the implications from the underlying microprocessor architecture and modify the algorithm of the application accordingly. For example, we must exploit both data and program parallelism in the applications to get the best performance. Although an algorithm may require more computation (for example, matrix multiplication vs. multiplication free), the algorithm may run faster if it can be implemented in SIMD media instructions or in multiple threads (as shown in Table 1). At the same time, we should be very careful about any quality degradation when we want to increase the parallelism (as shown in Figure 12).

REFERENCES:

1. D. M. Barbosa, J. P. Kitajima, and W. Meira Jr., "Real-Time MPEG Encoding in Shared-Memory Multiprocessors," Int'l Conf. on Parallel Computing Systems, 1999.
2. F. Casalino, G. D. Cagno, and R. Luca, "MPEG-4 Video Decoder Optimization," Int'l Conf. on Multimedia Computing and Systems, vol. 1, pp.363-368, 1999.
3. Y.-K. Chen, M. Holliman, E. Debes, S. Zheltov, A. Knyazev, S. Bratanov, R. Belenov, and I. Santos, "Media Applications on Hyper-Threading Technology," Intel Technology Journal, pp. 47-57, Feb. 2002.
4. S. Ge, X. Tian, and Y.-K. Chen, "Efficient Multi-threading Implementation of H.264 Encoder on Intel Hyper-Threading Architectures," IEEE Pacific-Rim Conf. on Multimedia, pp. 469-473, Dec 2003.
5. W. W. Gibbs, "A Split at the Core", Scientific American, pp. 96-101, Nov. 2004.
6. M. Holliman, E. Q. Li, and Y.-K. Chen, "MPEG Decoding Workload Characterization," Workshop on Computer Architecture Evaluation using Commercial Workloads, pp. 23-34, Feb. 2003.
7. International Standard Organization, "Information Technology-Coding of Audio-Visual Objects, Part10--- Advanced Video Coding", ISO/IEC 14496-10.
8. International Standard Organization, "Information Technology-Coding of Audio-Visual Objects, Part2---Visual", ISO/IEC 14496-2.
9. Intel Corp., Intel® Pentium® 4 Optimization Reference Manual, Order number: 248966.
10. Intel Corp., Intel® IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference, Order number: 245471.
11. Intel Corp., "Motion Estimation Algorithms Using Streaming SIMD Extensions 3," available on-line: http://www.intel.com/cd/ids/developer/asmo-na/eng/66775.htm, Dec. 2003.
12. Intel Corporation, "Next Generation Intel® Processor: Software Developers Guide," Order number: 252490, available on-line: http://www.intel.com/cd/ids/developer/asmo-na/eng/66756.htm, Jan. 2004.
13. V. Iverson, J. McVeigh, B. Reese, "Real-Time H.264/AVC Codec on Intel Architectures," Int'l Conf. on Image Processing, pp. 1541-1544, Oct. 2004.
14. E. Q. Li and Y.-K. Chen, "Implementation of H.264 Encoder on General-Purpose Processors with Hyper-Threading Technology," SPIE Conf. on Visual Communications and Image Processing, vol. 5308, pp. 384-395, Jan. 2004.
15. X. Li, E. Q. Li, and Y.-K. Chen, "Fast Multi-Frame Motion Estimation Algorithm with Adaptive Search Strategies in H.264," Int'l Conf. on Acoustics, Speech, and Signal Processing, vol. III, pp. 369-372, May 2004.
16. H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-Complexity Transform and Quantization with 16-Bit Arithmetic for H.26L," Int'l Conf. on Image Processing, vol. 2, pp. 489-492, Oct. 2002.
17. V. Lappalainen, "Performance Analysis of Intel MMX technology for an H.263 Video Encoder," Proceeding of ACM Multimedia, pp. 309-314, 1998.
18. V. Lappalainen, A. Hallapuro, T. D. Hämäläinen, "Complexity of optimized H.26L video decoder implementation," IEEE Trans. on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 717-725, July 2003.

19. V. Lappalainen, A. Hallapuro, T. D. Hämäläinen, "Performance of H.26L Video Encoder on General-Purpose Processor," Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology, vol. 34, no. 3, pp. 239-249, July 2003.
20. D. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Microarchitecture and Performance," Intel Technology Journal, pp. 4-15, Q1 2002.
21. B. Meng, and O. C. Au, "Fast Intra-Prediction Mode Selection for 4x4 Blocks in H.264," in Int'l Conf. on Acoustics, Speech, and Signal processing, vol. 3, pp. 389-392, Apr. 2003.
22. K. Shen, L. Rowe, and E. Delp, "A Parallel Implementation of an MPEG-1 Encoder: Faster than Real-Time," SPIE Conf. on Digital Video Compression: Algorithms and Techniques, 1995.
23. H. H. Taylor, et al., "An MPEG Encoder Implementation on the Princeton Engine Video Supercomputer," Data Compression Conference, pp. 420-429, 1993.
24. E. B. van der Tol, E. G. T. Jaspers, and R. H. Gelderblom, "Mapping of H.264 Decoding on a Multiprocessor Architecture", SPIE Conf. on Image and Video Communications and Processing, vol. 5022, pp. 707-718, Jan. 2003.
25. X. Zhou, E. Q. Li, and Y.-K. Chen, "Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions," SPIE Conf. on Image and Video Communications and Processing, vol. 5022, pp. 224-235, Jan. 2003.
26. S. Zhu, and K.-K. Ma, "A New Diamond Search Algorithm for Fast Block Matching," IEEE Trans. on Circuits and Systems on Video Technology, vol. 9, no. 2, Feb. 2000, pp287-290.