Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions

Xiaosong Zhou[†], Eric Q. Li[†], and Yen-Kuang Chen[†‡]
[†]Intel China Research Center, Intel Corporation, Beijing
[‡]Microprocessor Research Labs, Intel Corporation, Santa Clara

ABSTRACT

As emerging video coding standards, e.g. H.264, aim at high-quality video contents at low bit-rates, the encoding and decoding processes require much more computation than most existing standards do. This paper analyzes software implementation of a real-time H.264 decoder on general-purpose processors with media instructions. Specifically, we discuss how to optimize the speed of H.264 decoders on Intel Pentium 4 processors. This paper first analyzes the reference implementation to identify the time-consuming modules. Our study shows that a number of components, e.g., motion compensation and inverse integer transform, are the most time-consuming modules in the H.264 decoder. Second, we present a list of performance optimization methods using media instructions to improve the efficiency of these modules. After appropriate optimizations, the decoder speed improved by more than 3x---it can decode a 720x480 resolution video sequence at 48 frames per second on 2.4GHz Intel Pentium 4 processors compared to reference software's 12 frames per second. The optimization techniques demonstrated in this paper can also be applied to other video/image processing applications. Additionally, after presenting detailed application behavior on general-purpose processors, this paper discusses a few recommendations on how to design future efficient/powerful video/image applications/standards with given hardware implications.

Keywords: H.264, Integer Transform, Motion Compensation, MMX/SSE/SSE2 Technologies, SIMD

1. INTRODUCTION

Most modern microprocessors have multimedia instructions to facilitate multimedia applications. For example, the single-instruction-multiple-data (SIMD) execution model was introduced in Intel architectures. MMX, SSE and SSE2 Technologies [3] can execute several computations in parallel with 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, the complexity of emerging multimedia applications imposes new demands on processor performance.

H.264 [1] 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 those existing standards, such as, H.263 and MPEG-4 [2]. Furthermore, a number of new features and capabilities have been introduced in H.264 to efficiently improve the coding performance. As the standard becomes more complex, the decoding process requires much more computation powers than most existing standards.

In this paper, we will discuss the analysis and optimization of H.264 decoder on Intel Pentium 4 processors. In our analysis, we have identified some new time-consuming kernels and present corresponding optimization methods on them. Section 2 provides a brief introduction of the H.264 standard. In Section 3, a simple analysis is performed on the decoder to identify the most time-consuming kernels. Next, appropriate optimization methods are used to improve the efficiency of these kernels. Section 4 shows the overall performance of the optimized decoder, and micro-architecture characteristics have been provided. Finally, discussions, conclusion, and possible future work are presented in Sections 5 and 6.

Copyright 2003 SPIE and IS&T.
This paper will be published in Proceeding of SPIE Conference on Image and Video Communications and Processing, vol. 5022, and is made available as an electronic preprint with permission of SPIE and IS&T. One print or electronic copy may be for personal use only. Systematic or multiple reproduction, distribution to multiple locations via electronic or other means, duplication of any material in this paper for a fee or for commercial purposes, or modification of the content of the paper are prohibited.

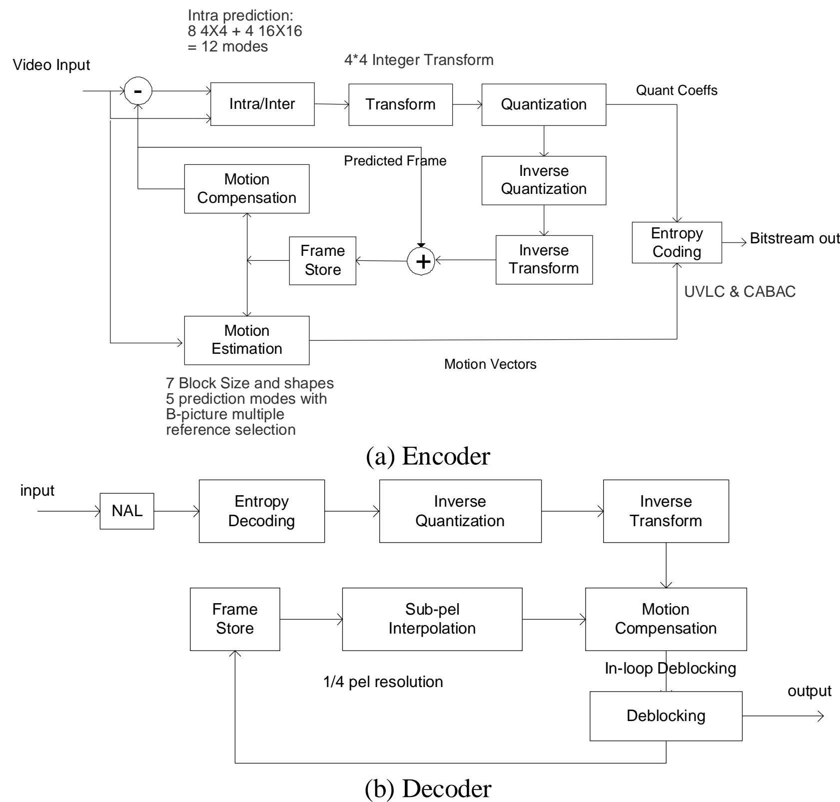
2. OVERVIEW OF THE H.264 STANDARD

H.264 has similar coding schemes as H.263 and MPEG-4. However, specific new concepts and features differentiate H.264 from the other standards. The draft of H.26L video coding standard was introduced by ITU-T Video Coding Expert Group (VCEG). In late 2001, the other leading video coding standardization group, ISO/IEC joined VCEG and formed the new working group – the Joint Video Team (JVT) to work on this emerging standard. Now the new standard is known as H.264 and also MPEG-4 Part 10, “Advanced Video Coding.” Figure 1 shows the encoding and decoding diagram of the standard. The standard includes the use of block-based motion compensation, DCT-like residual coding, scalar quantization, zigzag scanning and entropy coding. Furthermore, the standard includes some novel features to offer better compression efficiency.

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 a much larger number of different motion compensation block sizes are used for each macroblock (16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4, as illustrated in Figure 2). High motion vector resolution is specified, where sub-pel interpolation could provide higher spatial accuracy at fractional positions. In addition, deblocking filter within the motion compensation loop 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 DCT transform is replaced by a DCT-like integer transform in the H.264 standard. These transforms can be implemented with additions and shifts, therefore they are so-called ‘multiplication-free’, which is believed to have a significant computational advantage compared with DCT transforms. Also the small size helps to reduce the blocking and ringing artifacts. Moreover, the precise integer transform eliminates any mismatch between the encoder and the decoder [6].

The standard also implements a more complex and efficient entropy coding method: the context-based arithmetic coding



(a) Encoder

(b) Decoder

Figure 1: Encoder and decoder diagram of H.264

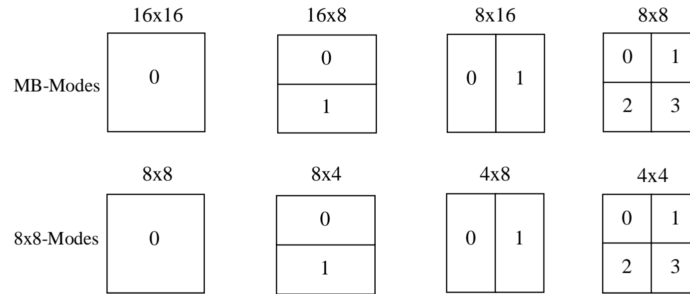


Figure 2: Block models.

(CABAC). Instead of using a universal coding table, the probability model of this coding method is adaptable to the changing statistics of incoming data; therefore it can offer better coding efficiency. The normally used universal variable length coding (UVLC) scheme is still supported in the system and serves as an alternative method.

3. IMPLEMENTATION OF H.264 DECODER USING MEDIA INSTRUCTIONS

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 efficiency 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. Intel Pentium 4 processors extend SIMD computation model with the introduction of SSE2, which operates on packed double (float)-precision data elements as well as 128-bit packed integers. The full set of SIMD capability greatly improves the performance of multimedia applications. For example, a 128-bit SSE2 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 SSE2 registers. This results in a significant performance improvement.

Nevertheless, there are restrictions on these SIMD instructions. While loading data into these longer registers, a misaligned 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 µops to be executed instead of one. Accesses that span either 16 byte or 64 byte boundaries are likely to incur a large performance penalty, since they are executed near retirement, and can incur stalls that are on the order of the depth of the pipeline. For best performance, 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.

In this section, we demonstrate how to implement the H.264 decoder using the benefits of media instructions while circumventing the restrictions.

3.2 KERNEL IDENTIFICATION OF THE H.264 DECODER

In order to identify time-consuming kernels in H.264 decoder, we breakdown the execution time of the H.264 decoder on 2.4GHz Pentium 4 processors, as shown in Figure 3. It decodes a H.264 CIF format video bitstream with IBBP prediction mode. The decoding rate is approximately 30.7 frames/s.

Experiments were conducted on 2.4GHz Intel Pentium 4 processors with 512MByte memory. The on-chip first-level cache (L1) and second-level cache (L2) are 8K and 512Kbytes, respectively. The decoder runs under Microsoft Windows XP. In the experiment, we choose CIF format sequence foreman with IBBP group-of-picture structure as its basic configuration.

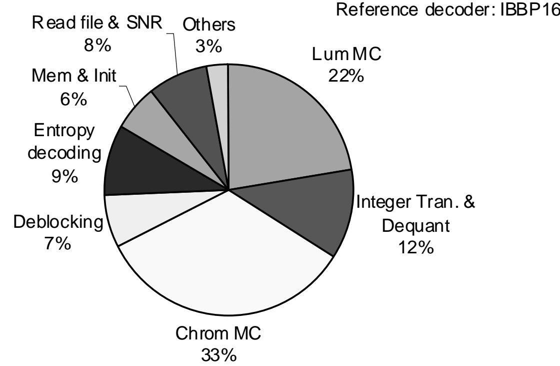


Figure 3: Time breakdown of H.264 reference decoder.

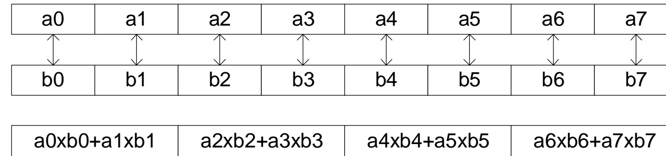


Figure 4: PMADDWD instruction.

From the diagram, it is easy to identify the following modules as the key kernels in the decoder: luminance and chrominance motion compensation, inverse integer transform, entropy decoding and deblocking filtering, of which motion compensation is the most time-consuming module. Unlike many previous video coding standards, chrominance motion compensation occupies a significant percentage in the total motion compensation process of the H.264 standard. We are interested in implementing these most time-consuming modules using SIMD technologies in order to improve the efficiency of the decoder. In the following sections, we will discuss implementation techniques in details.

3.3 MOTION COMPENSATION

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 not be fully utilized. 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 based on 8x8 block size to work on most macroblocks, and similar SIMD implementations can also be applied to macroblocks on 4x4 level. This approach fully makes use of the 128-bit register, and improves the efficiency 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 parameters are constants and pre-stored in different phases. The interpolation process conducts vector product between the parameter vector and the image data vector. An SSE-2 instruction PMADDWD [4] can facilitate this implementation. Figure 4 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 parameter 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.

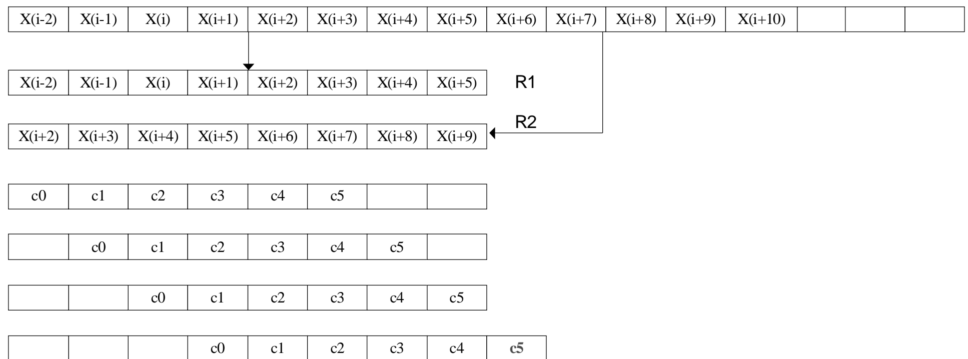


Figure 5: FIR implementation of sub-pel interpolation.

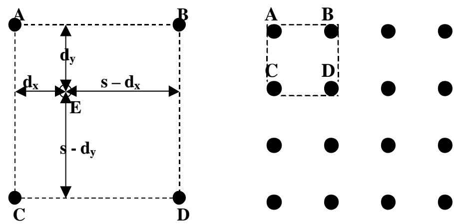


Figure 6: Chrominance motion compensation

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[1], 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.

For example, at row interpolation, as the image data are stored in bytes, we can first load 6+8-1 =13 relative pixels into a 128-bit register, then unpack and shift them into two SSE registers (R1 and R2) with required 16-bit length as shown in Figure 5. Since the filter parameters must be shifted along the row, we can load 4 different copies of the parameters with different shift phases, and then perform PMADDWD instruction on R1 and the four copies of parameters 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 parameter 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 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, compared with interpolation on 4x4 blocks, nearly 50% computations are saved during the column interpolation stage.

3.3.2 CHROMINANCE MOTION COMPENSATION

For chrominance motion compensation, the H.264 decoder's 2x2 block size prevents a direct application of SIMD instructions. As motion compensation block sizes smaller than 8x8 are rarely used, we are interested in implementing chrominance interpolation based on 4x4 block size. As shown in Figure 6, the final interpolated chrominance pixel E is derived from the 4 corresponding pixels (A, B, C, D) with the formula:

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

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

X **A** **B**

Figure 7: 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 8: 4x4 matrix multiplications.

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

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 the 16 pixels to compute the final 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 7 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 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 8 shows a typical 4x4 matrix multiplication. As with the FIR filtering optimization, PMADDWD can be used to optimize the operation of vector products. Furthermore, if we simply store the row vector of X and column vector of Y into two registers and use PMADDWD, we must shift the result, retrieve, and add the 32-bit integers 3 times to calculate each element after PMADDWD instruction. Also, it is costly to load column vector of matrix Y into an SSE2 register. There will be too much overhead for these operations. We 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 9 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.

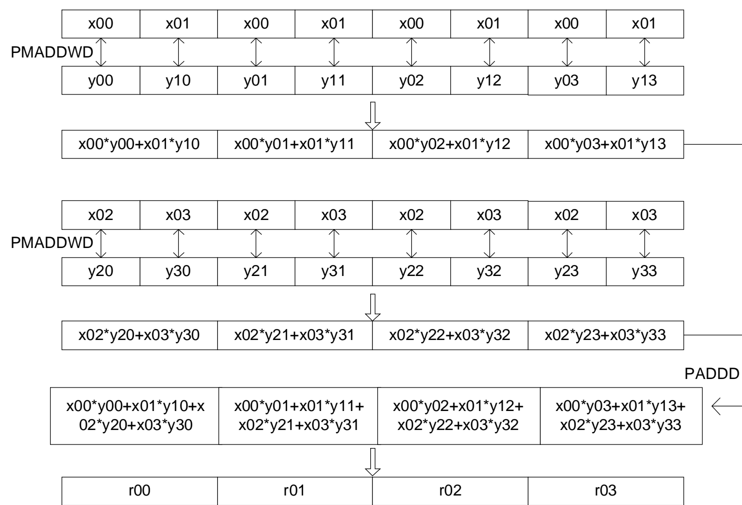


Figure 9: Efficient 4x4 matrix multiplication implementation.

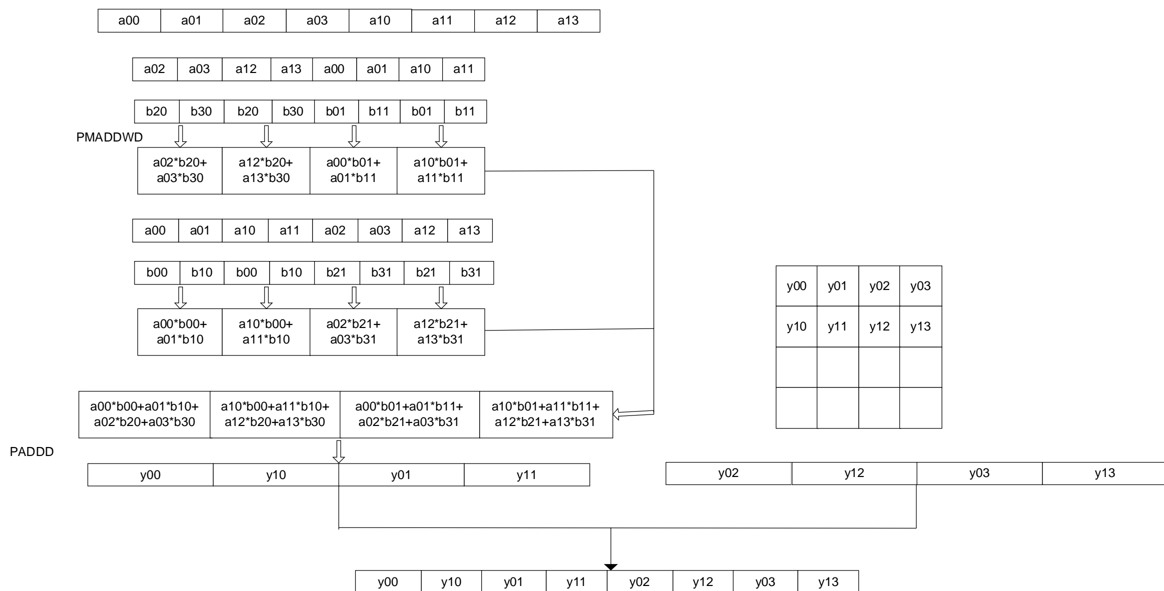


Figure 10: Chain matrix multiplications.

The realization of H.264 integer transform extends the above general approach. As shown in Figure 7, the first and last matrices in the equation are constant. According to this information, we can 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.

We first multiply the last two matrices A and B together and store the temporal result in matrix Y. Next, we multiply the first matrix X with the temporal 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 10 illustrates the whole inverse integer transform process. First, we load matrix A as 16-bit integer operands into an SSE-2 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-2 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 7. 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, another 4x speedup is achieved.

4. EXPERIMENTAL RESULTS

4.1 HIGH LEVEL BREAKDOWN OF 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.

Figure 11 shows the time breakdown of the optimized decoder at different bit-rates. Compared with Figure 3, the percentage of the motion compensation and inverse integer transform modules are dramatically minimized. At the same time, entropy decoding and deblocking filter now have a much larger impact in the H.264 decoder (from 16% up to 44%). Table 2 shows the speed-up 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. Because the deblocking filter and entropy decoding are not yet optimized, their performance is relatively lower than the other two modules.

We also measured the performance of the optimized decoder on different encoded video streams, and the results are shown in Figure 12. 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 speedups. 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 speedups because the motion compensation module in their decoding process is more critical than video streams in IPPP mode.

4.2 MICRO-ARCHITECTURE CHARACTERISTICS OF THE OPTIMIZED DECODER

We further studied the micro-architecture behaviors of the four key modules of the optimized decoder. A performance analysis tool VTune [5] is used to collect following detailed characteristics of the H.264 decoder.

Several kernel micro-architecture metrics are shown in Table 3. There are a lot of L1 and L2 cache misses in motion compensation because the reference frames could not be totally loaded into the L2 cache. For entropy decoding, branch mis-prediction is a major problem that yields lower IPC.

Table 4 shows the frequency scaling result for key modules in the H.264 decoder and the overall performance. As shown in the table, for inverse integer transform and entropy decoding, the performance scales about the same with frequency. And, both of these modules have good cache hit behavior. In this case, we can conclude that the integer transform and entropy decoding modules are computation-bound; while motion compensation and deblocking filter modules are co-impacted by computation and memory system

In order to further clarify the impact of cache misses to the H.264 decoder, different cache size has been evaluated in the experiment. We choose two Pentium 4 processors with different cache size (256kB ~ 512kB), and some metrics related with cache are listed in Table 5. From the table, we can see that the L2 cache size affect the overall performance of the decoder. The run-time on a Pentium 4 processor with 512KB L2 cache is about 5% shorter than that on a Pentium 4 processor with 256KB L2 cache.

Table 1: Comparison of different implementations on inverse transform.

Implementation Method	Time Used (seconds)	Speedups from original matrix multiplications	Speedups from multiplication-free
Original Matrix Multiplication	8.22	-	0.3
Multiplication-Free	2.41	3.4	-
Matrix Multiplication with SSE2	0.56	14.7	4.3

Table 2: Kernel module speed-up.

Modules	Speed-up
Luminance Motion Compensation	2.9x
Chrominance Motion Compensation	10.2x
Integer Transform	4.3x
Deblocking Filter	1.1x

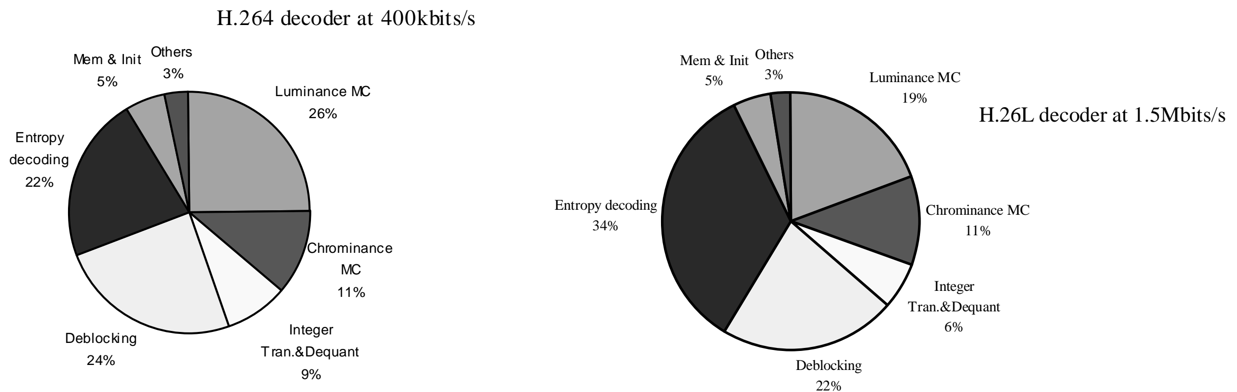


Figure 11: Time breakdown of optimized H.264 decoder.

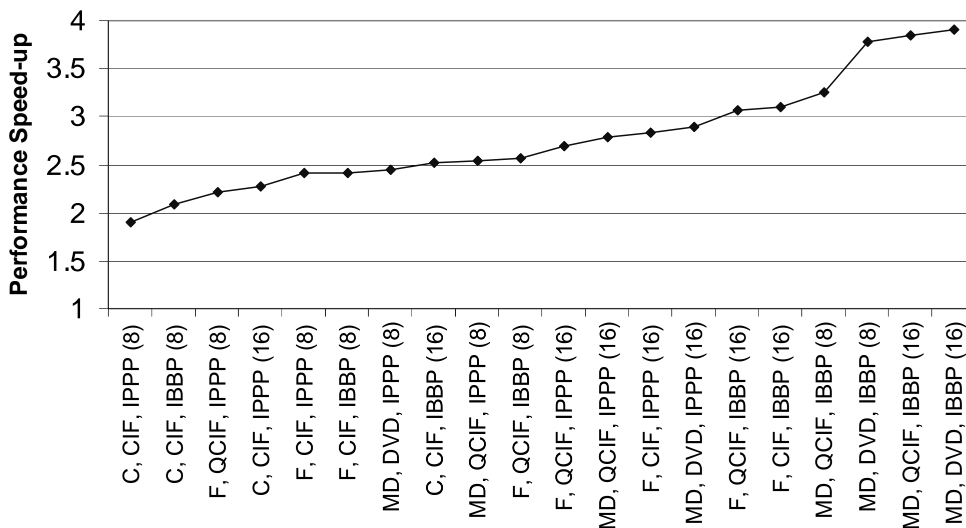


Figure 12: S-curve of decoder speedups on different video streams (C: coastguard, MD: mother & daughter, F: foreman)

5. DISCUSSIONS

In the process of optimizing H.264 decoder, we discovered a few interesting points regarding efficient video processing program design, with consideration of the characteristics of general-purpose processor architectures with media instructions.

5.1 SIMPLIFYING ALGORITHMS

The original 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 will remain 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, this implementation makes it impossible for further SIMD optimization. On the contrary, by implementing the simple traditional matrix multiplication with SIMD instructions and utilizing special implementation techniques, the efficiency of the program becomes actually better.

Table 3: Kernel characterization on 2.4GHz Pentium 4 processors.

Micro-Arch. \ Kernel	Motion Compensation	Integer Transform	Deblocking filter	VLD
IPC	0.618	1.163	0.900	0.648
UPC	0.960	1.724	1.142	0.845
SSE2/Instr.	0.282	0.268	0.000	0.000
Branches/Instr.	0.051	0.047	0.089	0.108
Branch Prediction Rate	0.964	0.996	0.932	0.903
L1 Hit-rate	0.930	0.944	0.972	0.956
L2 Hit-rate	0.871	0.999	0.808	0.950
FSB Activity	0.084	0.033	0.117	0.096
Loads/Instr.	0.394	0.294	0.326	0.296

Table 4: Frequency scaling of key modules in H.264.

CPU Frequency	Frequency scaling	Overall		Modules			
		Frames per Second	Scaling	Motion compensation	Integer Transform and Dequantation	Deblocking filter	Entropy decoding
1.2GHz	1	61.1	1	1	1	1	1
1.6GHz	1.33	78.8	1.29	1.31	1.30	1.28	1.33
2.0GHz	1.67	96.8	1.58	1.57	1.62	1.62	1.70
2.4GHz	2.00	110.0	1.80	1.83	1.92	1.80	1.98

Table 5: Performance impact of different cache size.

	Pentium 4 with 512KB L2 cache				Pentium 4 with 256KB L2 cache			
Kernel	IPC	L1 Hit-rate	L2 Hit-rate	Loads/Instructions	IPC	L1 Hit-rate	L2 Hit-rate	Loads/Instructions
Motion compensation	0.618	0.930	0.871	0.394	0.58	0.932	0.834	0.410
Integer Tran. &Dequant	1.163	0.944	0.999	0.294	1.20	0.931	0.999	0.280
Deblocking filter	0.900	0.972	0.808	0.326	0.83	0.972	0.769	0.322
Entropy decoding	0.648	0.956	0.950	0.296	0.60	0.946	0.92	0.272

5.2 CUSTOMIZING ALGORITHM SETTINGS

As the current and future processors possess the capability of processing 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 8 16-bit integers, as long as the filter length does not exceed 8, we can use one SSE register to store the filter coefficients, and another one to store the data to be filtered. Then 8 multiplications can be processed simultaneously and the program efficiency is greatly improved. Therefore an FIR filter of length 4 and 8 will have the same computational complexity since they are processed exactly the same way in this approach. But the two filters may provide quite different effects because their lengths are different. 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.

5.3 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 has to be more careful while allocating memory spaces in order to avoid unaligned loads. Also if SIMD implementation is desired, the length of data units to be processed simultaneously must be identical.

Sometimes there will be intense memory duplication operations. In this case, memory spaces should be allocated consecutively so that large scale data movement, such as data movement utilizing 128-bit registers, will be possible. The entire efficiency will thereby be improved.

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.

6. CONCLUSION

In this paper, we have optimized and studied the performance of H.264 decoder on Intel Pentium 4 processors. Performance optimizations with SIMD instructions and related discussions for the decoder of H.264 have been presented. The work demonstrates that, after appropriate performance optimizations, the decoder gained significant performance improvements.

We have also studied the performance of a well-optimized H.264 decoder, describing some key characteristics of the workload. For application performance, we can conclude that entropy decoding and integer transform are entirely computation-bound, scaling directly with frequency on Pentium 4 processors; motion compensation and deblocking filter module are co-impacted by computation and memory system.

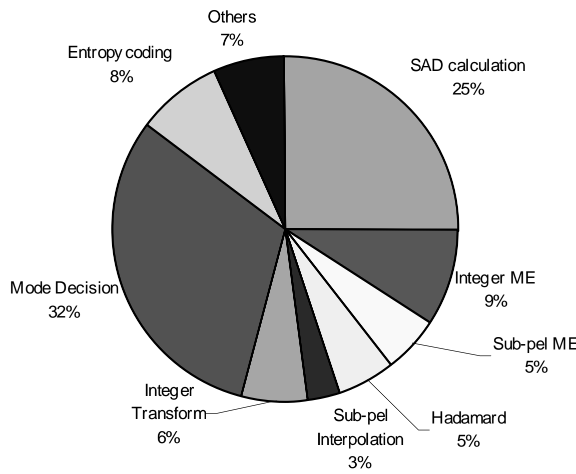
While we have dramatically improved the performance of H.264 decoder, further studies will be made on modules such as entropy decoding and deblocking filtering. Better implementations need to be achieved for these modules.

From a micro-architectural view, we can see that a number of conditional branches in H.264 decoding occur during entropy decoding and deblocking filter. Some of these branches are inherited to the algorithm itself and it is hard to eliminate them on the programming level. Moreover, these branches are data-dependent and inherently difficult to

Table 6: Speedups of the key modules in H.264 encoder.

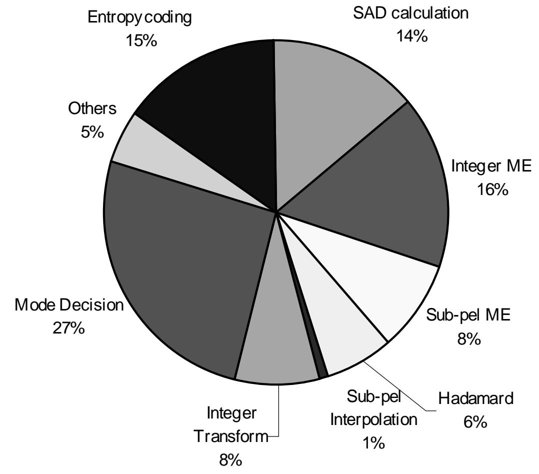
Module	Speedup
SAD Calculation	3.5x
Hadamard Transform	1.6x
Sub-Pel Search	1.3x
Integer Transform and Quantization	1.3x
¼ Pel Interpolation	2.0x

Breakdown of Reference Encoder



0.49 frames per second

Breakdown of Optimized Encoder



0.81 frames per second

Figure 13: Time breakdown of H.264 encoder.

predict. As newly developed processors have more pipeline stages, branch mis-prediction penalties degrade the performance of H.264 in these modules significantly. Therefore, it is important to avoid or reduce branches while designing the algorithms.

The related techniques involved in this paper can be applied to other video/image processing applications. For example, we have started to work on the implementation of H.264 encoder using similar techniques. Table 6 shows the speedups for each key module residing in H.264 encoder, and Figure 13 shows the time breakdown diagram of the reference encoder and the presently optimized encoder. There are still a lot of rooms for us to continue improving the performance of the H.264 encoder.

REFERENCE

1. ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC, Document JVT-D157, 4th Meeting: Klagenfurt, Austria, July 2002.
2. International Standard Organization, "Information Technology-Coding of Audio-Visual Objects, Part2---Visual", ISO/IEC 14496-2.
3. Intel Corp., Intel® Pentium®4 Optimization Reference Manual, Order number: 248966.
4. Intel Corp., Intel® IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference, Order number: 245471.
5. Intel Corp., Intel® Vtune™ Performance Analyzer.
6. 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.