ADVANCED H.264/AVC ENCODER OPTIMIZATIONS ON A TMS320DM642 DIGITAL SIGNAL PROCESSOR

Dorian Schneider[1,3], Marco Jeub[2], Zhou Jun[3] and Song Li[3]

[1]Department of Electrical Engineering, Berlin University of Technology, Germany
[2]Institute of Communication Systems and Data Processing, RWTH Aachen University, Germany
[3]Institute of Image Communication & Information Processing, Shanghai Jiao Tong University, China

schneider@europe.com, jeub@ind.rwth-aachen.de, zhoujun@sjtu.edu.cn, song_li@sjtu.edu.cn

ABSTRACT

This paper discusses the optimization of the H.264/AVC video encoder in the context of a modified software implementation on a Texas Instruments TMS320DM642 digital signal processor. Several algorithmic optimizations are proposed to improve time critical parts of the codec like the quantization step and the pixel interpolation. The algorithms proposed in this paper invoke the Enhanced Direct Memory Access (EDMA) Controller, intrinsics and look-up tables to accelerate the encoding and do not affect the image Peak Signal-to-Noise Ratio (PSNR) or compression performance. The computational acceleration gain of these algorithms are the foundation of our real time 30 CIF frames/second baseline implementation.

Index Terms— H.264, DSP, Real-time, Optimization, EDMA

1. INTRODUCTION

H.264 is a state of the art video encoder that combines strong compression ratios with good image quality. These advantages are payed with a considerably higher need of computational power. Therefore, H.264 real-time implementations on embedded systems with limited hardware resources become a challenge. However, digital signal processors (DSP) are a predestinated platform for video encoder implementations due to their beneficial performance-cost tradeoff.

The *TMS320DM642* DSP is part of the Texas Instruments C64x series. The CPU runs at a clock rate of 600 MHz. The memory of the DM642 is organized in a 2-level Cache: 32 kByte serve as level 1 (L1) cache and operate at CPU frequency, 256 kByte serve as level 2 (L2) cache. L2 cache (also internal memory) can be configured to work as RAM or as cache. It operates at half CPU frequency. The external SDRAM is 32 MByte large and operates at 133 Mhz. The VelociTI *advanced very long instruction word* (VLIW) architecture is a feature of all *C64x* DSPs that parallelizes code execution by running up to 8 execution units at the same time. Further, the *Enhanced Direct Memory Access* (EDMA) controller is able to manage memory transfers independently from the CPU. In that way, no additional overhead is caused when large data blocks are moved between internal and external memory.

Many researches are done to improve the H.264 encoding speed on embedded systems. In most cases the encoder core functionalities are varied [1, 2], resulting in a degradation of the encoder compression and image Peak signal-to-noise ratio (PSNR). Other work focuses on improvements of the data flow among the encoder based on complicated encoder structure changes [3]. This paper presents algorithms that fully take advantage of the *TMS320DM642* hardware to accelerate the H.264 encoding of YUV video files, without altering the compression performance or the image quality of the encoder. The EDMA controller, look-up tables and intrinsics are the tools we invoked to achieve this goal.

We start by introducing a method that simplifies the quantization step and discuss an algorithm that uses the EDMA controller to accelerate the time consuming half-pixel interpolation process. Further, a method is presented that improves the quarter-pixel interpolation part using intrinsics. Finally, we discuss how the EDMA controller can be used to easily accelerate several time consuming parts of the encoder.

2. THE QUANTIZATION STEP

H.264 directly combines the integer discrete cosine transformation (integer DCT) with the quantization step. The H.264 integer DCT forward transformation is based on a modified DCT that has been optimized to avoid multiplications and can be entirely carried out using integer arithmetic. It is defined by

$$\mathbf{Y} = \mathbf{W} \otimes \mathbf{E}, \tag{1}$$

where $\mathbf{W}$ is the core transformation [4]

$$\mathbf{W} = \sum_{i=0}^{3} \sum_{j=0}^{3} \mathbf{C}_{i,j} \mathbf{X}_{i,j} \mathbf{C}_{i,j}^{T} \tag{2}$$

and $\mathbf{E}$ is a scaling matrix.

$$\mathbf{E} = \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix}, a = \frac{1}{2}, b = \sqrt{\frac{2}{5}}, c = \frac{1}{2}. \tag{3}$$

In Equation (1), $\otimes$ denotes an element wise multiplication rather than a matrix multiplication. This scaling matrix $\mathbf{E}$ has directly been incorporated into the quantization process:

$$\mathbf{Z} = \sum_{i=0}^{3} \sum_{j=0}^{3} round\left(W_{i,j} \frac{E_{i,j}}{Q_{step}}\right). \tag{4}$$

Here, Q_{step} is the quantizer step size. H.264 supports 52 different values for Q_{step}, each indexed by the quantization parameter QP. $\mathbf{Z}$ is the quantized output matrix and $W_{i,j}, E_{i,j}$ refer to Equation (2)

and Equation (3) respectively. In order to fully avoid divisions, (4) has been modified to

$$\mathbf{Z} = \sum_{i=0}^{3} \sum_{j=0}^{3} round \left(W_{i,j} \frac{MF}{2^{qbits}} \right), \tag{5}$$

with

$$qbits = 15 + floor \left(\frac{QP}{6} \right) \tag{6}$$

and

$$\frac{MF}{2^{qbits}} = \frac{E_{i,j}}{\mathbf{Q}_{step}}. \tag{7}$$

Using Equation (7), the values for MF can now be calculated for given matrix indices i and j and a given index quantization parameter QP. Finally, Equation (4) can be expressed in integer arithmetic by

$$|Z_{i,j}| = (|W_{i,j}| \cdot MF + f) >> qbits, \tag{8}$$

with

$$f = \frac{2^{qbits}}{3}; f = \frac{2^{qbits}}{6} \tag{9}$$

for intra macroblocks and inter macroblocks respectively.

The H.264 reference software performs calculations for Equations (6), (7), (8) and (9) for every macroblock, which results in a high time consumption for the quantization process. Knowing that the values for *f*, *MF* and *qbits* only depend on the index parameter QP, we precalculated all possible values for *f*, *MF* and *qbits* for given *QPs* and stored the results in look-up arrays. The encoder parts that were responsible for the calculations of Equations (6), (7) and (9) have then been replaced with array accesses, indexed by the quantization parameter QP. Hence, the global encoding time of our DSP implementation could be accelerated by 6% without altering the compression performance of the encoder.

3. HALF-PIXEL INTERPOLATION

The AVC/H.264 codec uses an 6-tap FIR filter that interpolates half-pixels from adjacent integer pixel locations. This text differentiates between three different interpolation schemes: *horizontal, vertical* and *intermediate* interpolation. The horizontal interpolation describes the process of interpolating the left and right adjacent integer pixels to generate half-pixels. The pixels *aa, bb, cc, dd, ee, ff* in Figure 3 are generated in this way. On the other hand, the pixels *gg, hh, ii, jj, kk, ll* are generated by the upper and lower adjacent integer pixels, hence the name vertical interpolation. The intermediate interpolation generates the remaining half-pixels (i.e. *mm*). The process uses the already generated horizontal or vertical half-pixels as filter input.

Since the amount of data for a normal video frame is larger than the capacity of the fast internal memory of the DSP hardware, all read- and store-operations have to be done on data placed in external memory (see Figure 3). This causes a tremendous amount of overhead and cache misses. Without optimization, the half-pixel interpolation process consumes more than 45 % of the entire encoding time. Therefore, we developed an optimized algorithm that invokes the EDMA controller to preload pixel information into internal RAM. In fact, the horizontal and vertical pixel interpolation operate on the same input data, which allows us to 'EDMA transfer' a macroblock from external memory into an internal memory buffer while the CPU is filtering. For that purpose, the algorithm uses an active input buffer I1 and an passive input buffer I2. Additionally, three internal output buffers O1-O3 are used.
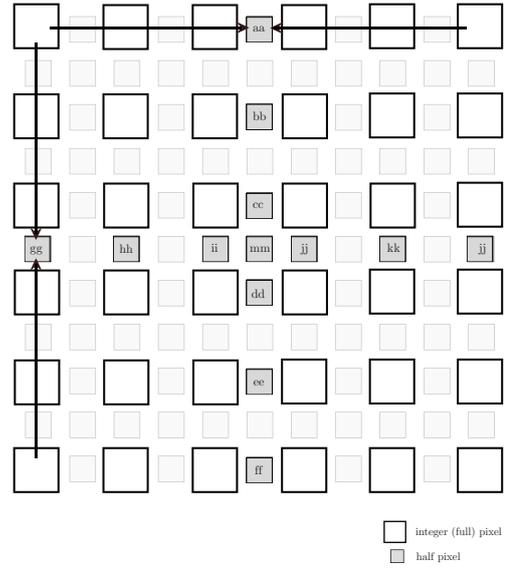

Fig. 1. Half- and quarter-pixel interpolation in AVC/H.264.

When initializing, I1 is filled with data using the EDMA controller. This is the only moment where the CPU has to wait for a data transfer to finish. Subsequently, the horizontal interpolation starts processing and stores its output to O1. While the vertical interpolation starts filtering using I1 as input, the EDMA controller is instructed to fill I2 with subsequent macroblock information. The vertical output is stored to O2. The intermediate interpolation uses O1 or O2 as input and stores its output to O3. These operations take place in an inner loop that iterates until a row has entirely been processed. Once a macroblock has been filtered, the buffers I1 and I2 are switched and the loop starts processing the adjacent macroblock. Once a row of macroblocks has been filtered, the inner loop ends and the outer loop is triggered. Here, the EDMA controller is instructed to transfer the output buffers O1-O3 to the corresponding locations in external memory. At the same time, the inner loop restarts and processes the next raw. The flow graph for the algorithm is illustrated in Figure 3.

Due to the fact that the FIR filter only operates on data stored in fast internal memory, this pixel preload strategy reduces the necessary instruction cycles for the half-pixel interpolation by 54 % without affecting the encoder performance in any way.

4. QUARTER-PIXEL INTERPOLATION

The quarter-pixel interpolation in H.264 uses an 2-tap mean filter with half-pixels as input. Contrary to half-pixels, quarter-pixels are not generated entirely for every frame. An analyzer decides if the generation of quarter-pixels is beneficial for the motion compensation and if so, the quarter-pixels are interpolated locally. Hence, the process is far less time consuming than the half-pixel interpolation. On the other hand, this local generation prevents that the pixel preload strategy discussed in Section 3 can be applied to the quarter-pixel process because of two reasons: The exact input locations are not known and EDMA transfers become inefficient with too little amounts of data.
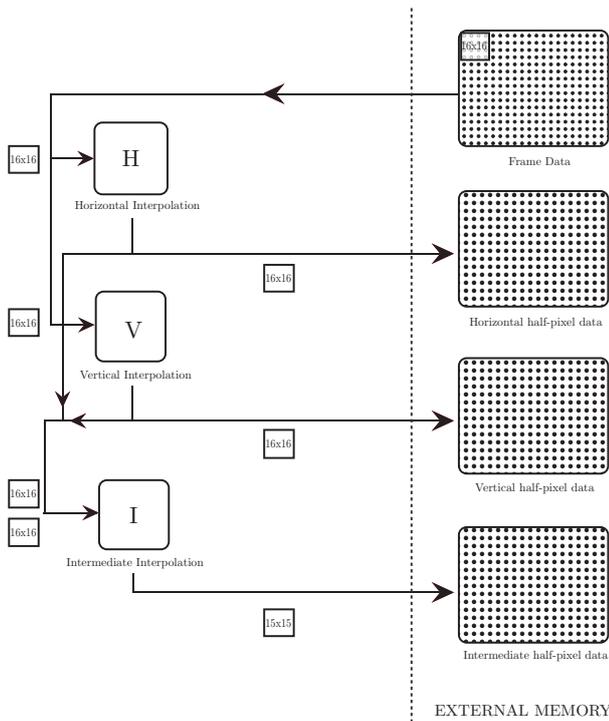
Fig. 2. JVT/AVC H.264 half-pixel interpolation flow graph for a signal processor implementation.

In any case, the quarter-pixel interpolation can greatly benefit from the usage of intrinsics. The C64x series come with a set of assembly coded functions that enable the programmer to use highly optimized hardware functions within the C-Code. The *AVGU4* intrinsic performs an averaging operation on 4 byte of packed data [5]. In that way, the 2-tap mean filter can be replaced with the intrinsic that can filter 4 bytes simultaneously with a single instruction. Theoretically, the quarter-pixel interpolation could be four times faster in that way, but in practice the input half-pixels are not always aligned to a byte boundary. Hence, the usage of a second intrinsic becomes necessary: The *MEM4* intrinsic [6] allows unaligned loads and stores of 4 bytes of data to memory with a single instruction. The *MEM4* intrinsic must be used to load the parameters for the *AVGU4* call, otherwise the output of the mean filter may not be correct .

Altogether, the usage of intrinsics accelerates the quarter-pixel interpolation by a factor of about 2, without altering the image quality or the compression ratio.

5. EDMA ENHANCEMENTS

Frequently, large data blocks need to be copied from memory locations during the encoding of a video frame. In general the C function *memcpy* is invoked to handle these data transfers. When using the *memcpy* function the CPU needs to monitor the transfer such that no other operations can be performed meanwhile. As mentioned earlier, the EDMA controller is a handy tool to carry out these transfers without invoking the CPU. Therefore, we now introduce three mechanisms that can greatly benefit from the usage of the EDMA controller. One should be aware that the usage of the EDMA needs the engineer to take care of cache coherencies by himself. For de-
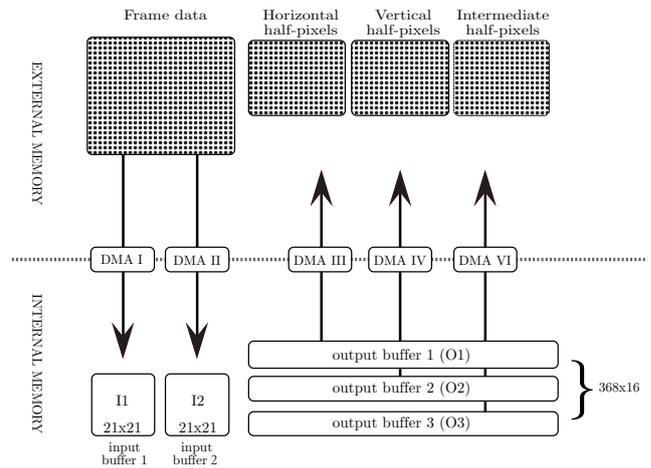

Fig. 3. EDMA transfer denotations.

tailed information about the EDMA device consider [7] and [8].

At the beginning of the encoding process the video frame needs to be copied so that data can be manipulated and restored. For that purpose the Y, U and V samples are transferred in a 2D fashion using two cascaded loops and the *memcpy* function. The C6x chip support library (CSL) provides the EDMA transfer function *DAT_copy2d* [8] that is able to move 2D data blocks with a single instruction. The function takes 6 arguments: The transfer type, the source location, the destination location and the width, height and stride of the data block. Using three separate function calls, the frame copy process could be completely handed over to the EDMA device, resulting in a considerable speed gain of the encoding process.

Before the half-pixel interpolation starts, the video frame is expanded with a several pixel wide frame that surrounds the image on the upper, lower, left and right sides. The upper and lower bands are generated by copying the first and last row of the frame consecutively into the border. Here, the CSL function *DAT_copy* [8] can be used to handle the data transfer. To generate the left and right border, a trickier handling is needed: The first (respectively last) column of the video frame needs to be copied into an array, so that the vertical alignment of the data can be transformed to an horizontal alignment. Again, the next step invokes the *DAT_copy2d* function: A 1D to 2D transfer type must be used as a function parameter and the destination stride needs to be set to two. In that way, the horizontal array is copied into the frame border in a vertical fashion.

Finally, the AVC/H.264 standard works with caches for pixel data that are used during the frame encoding and copied back when finished processing. The problematic with these transfers is that the source stride and the destination stride vary from time to time. It may occur that the source is a 8x8 block with a stride of 16 and the destination is a 8x8 block with a stride of 32. Since all EDMA functions only support transfers where source and destination stride are identical, one can use temporary buffers to overcome the issue. First, the source frame is copied into a buffer using the EDMA. When all buffers are filled, a second transfer moved the data from the buffer to the destination. In order to work properly, it is important that the buffer width is as wide as the maximum stride of source and destination.
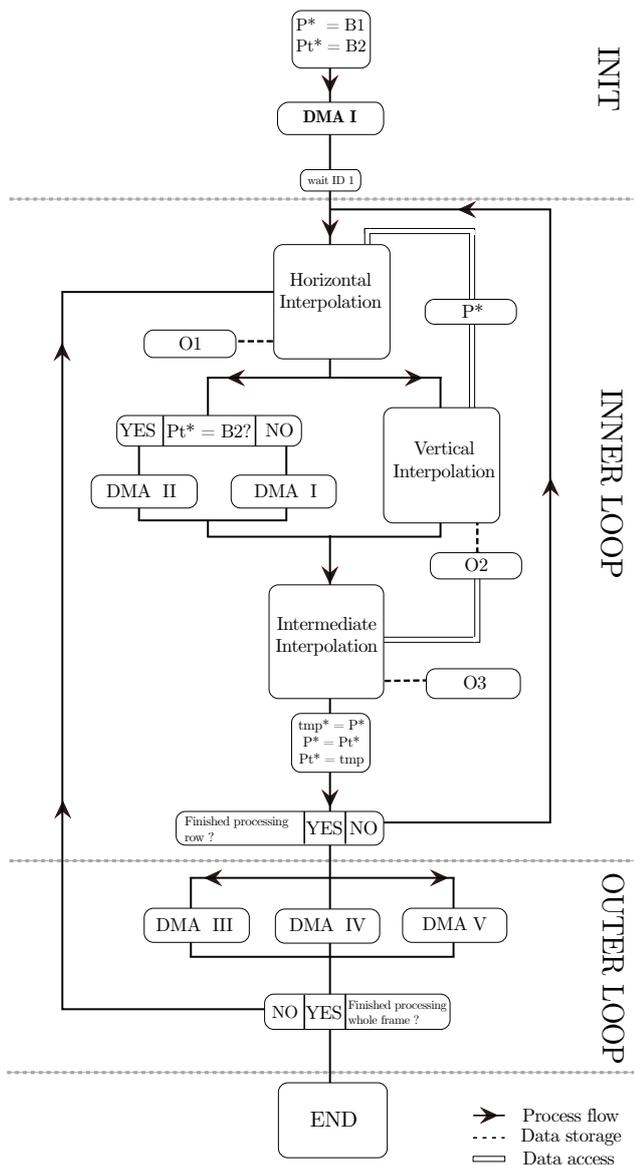
Fig. 4. Improved half-pixel interpolation flow graph. See Figure 3 for the DMA transfer denotations.

6. CONCLUSIONS

This paper proposes advanced optimization strategies to improve the computational speed of a H.264 DSP video encoder implementation. The discussed algorithms and techniques accelerate the video processing by utilizing C64x DSP appropriate characteristics and tools. The output of the encoder is not altered in any way. Altogether, the proposed methods allow a global speed gain of about 55%. Hereby, the most important acceleration factor is the improved half-pixel interpolation scheme with about 31% global speed gain.

Table 1 summarizes the optimized results by listing the acceleration gain for every proposed algorithm. The speed gain in percent indicates the speed improvement between our H.264 baseline encoder version incorporating the indicated algorithm and the same encoder version without the algorithm. The measurements were done using a Texas Instruments *TMS320DM642* DSP board, 600 MHz CPU clock rate and a 224 kB / 32 kB RAM/Cache configuration for the internal memory. The foreman CIF video stream [9] has been used as test sequence.

Algorithm	speed gain $[cycles.s^{-1}]$	speed gain $[\%]$
Quantization look-up tables	$2.64e6$	6
Half-pixel interpolation	$12.8e7$	31
Quarter-pixel interpolation	$6.42e6$	15
EDMA transfers	$1.30e6$	3

Table 1. Encoder acceleration for various optimized H.264 algorithms.

7. REFERENCES

[1] Hong-Jun Wang, Yan yan Hou, and Hua Li, "H.264/AVC video encoder algorithm optimization based on TI TMS320DM642," in *Proc. Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing IIHMSP 2007*, 26–28 Nov. 2007, vol. 1, pp. 529–532.

[2] Wen-Nung Lie, Han-Ching Yeh, T. C. I. Lin, and Chien-Fa Chen, "Hardware-efficient computing architecture for motion compensation interpolation in H.264 video coding," in *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2005*, 23–26 May 2005, pp. 2136–2139.

[3] Li Zhuo, Qiang Wang, D. D. Feng, and Lansun Shen, "Optimization and implementation of H.264 encoder on DSP platform," in *Proc. IEEE International Conference on Multimedia and Expo*, 2–5 July 2007, pp. 232–235.

[4] Iain Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*, John Wiley & Sons Ltd, Chichester, UK, 2003.

[5] Texas Instruments, *TMS320C6000 Instruction Set Simulator and Technical Reference Manual*, 2008, SPRU732G.

[6] Texas Instruments, *TMS320C6000 Optimizing Compiler Users Guide*, 2005, SPRU187N.

[7] Texas Instruments, *TMS320C6000 DSP Enhanced Direct Memory Access (EDMA) Controller Reference Guide*, 2005, SPRU234B.

[8] Texas Instruments, *TMS320C6000 Chip Support Library API Reference Guide*, 2004, SPRU401J.

[9] Arizona State University, *Repository of video test sequences of the Video Traces research group*, 2009, http://trace.eas.asu.edu/yuv/index.html.