An Evaluation of Parallelization Concepts for Baseline-Profile compliant H.264/AVC Decoders

Klaus Schöffmann, Markus Fauster, Oliver Lampl, and Laszlo Böszörmenyi

Department of Information Technology (ITEC), University of Klagenfurt
Universitätsstr. 65-67, 9020 Klagenfurt, Austria
{ks,mfauster,laszlo}@itec.uni-klu.ac.at, olampl@edu.uni-klu.ac.at

Abstract. Due to the increasing performance requirements of decoding H.264/AVC in HDTV or larger resolutions, new approaches are necessary to enable real-time processing. According to the current trend to parallel computation in all performance classes, decoding of AVC must be mapped to these architectures even though this is complicated by the increased complexity and many data dependencies in the codec. We propose and evaluate different ways of using multithreading to speed-up our .NET implemented decoder. While slice based approaches scale best, this is not a flexible approach because of the reliance on specially encoded streams. Functional partitioning and macroblock pipelining prove to be a good alternative for almost all evaluated videos.

Key words: AVC; H.264; Parallelization; Decoding; Evaluation; Multi-Threading; Pipelining

1 Introduction

In recent years, parallelization of computer programs experienced a certain renaissance due to the current architecture of processors used in the mainstream market. CPU manufacturers have changed their CPU architectures from a single-core to a multi-core design. Several independent processors are combined into one package with a shared cache and bus interface. While some current desktop and mobile computers already use a dual-core design, future systems and also entertainment devices will use many cores. In addition, today's multimedia applications require high CPU performance due to advanced video compression standards and the emerging demand for large and detailed digital movies. While current consumer computers still have problems to encode High Definition (HD) videos in real-time, the decoding process is currently fast enough on many modern computers. However, the current trend in the entertainment industry is to go beyond HD and to use Cinema-HD - with a resolution of 4096x2160 pixels - in the entire chain from production to consumption. Future encoders and decoders will have to process approximately four times more data than with current HD videos. Due to the trends mentioned above and since decoding is also part of encoding, future decoders and encoders will have to optimize their processes. They should take advantage of advanced instruction sets (such as

MMX/SSE/SSE2[1]), and possibilities for parallelization provided by the architecture of the CPU. Many multimedia applications are especially well-suited for parallelization because some steps of processing videos can be done independently. However, in order to increase the efficiency of compression, current video coding standards like H.264/AVC [2] have introduced additional dependencies into some parts of the encoding/decoding process. Those dependencies require more advanced parallelization schemes. Manually converting sequential programs into parallel ones is a difficult and error-prone task. In [3] we have already presented a slight language extension to the C# programming language which enables the automatic parallelization of loop operations on independent blocks of data. This extension has been implemented in the Mono [4] compiler and produces multithreaded code, because the run-time systems (Mono, .NET) map multiple threads to the underlying multicore architecture. Other, non thread-based schemes are subject of further study.

In this article, we summarize the results of evaluating several different parallelization schemes for an H.264/AVC baseline-profile decoder. Our decoder has been implemented with the C# programming language using a common style of object-oriented programming. Since the aim of our implementation was to create a portable baseline-profile decoder for the Common-Language-Runtime, we did not use any platform dependent optimizations. We have measured the costs of different parts in the decoding process and used the results as a basis for further parallelization strategies. Although our implementation has high memory requirements and moderate object creation time, the results of our measurements (see figures 1 and 2) show a similar distribution of the workload as in the H.264/AVC reference decoder (as presented in [5]). Although slice level parallelization is an obvious solution achieving good speed-up values (see [6], [7], [8], [9]), it has some disadvantages as well. It can only be applied if frames of a video are separated into slices, which depends on the encoder. Moreover, supporting slice-level parallelization reduces coding efficiency due to the reference limits during the encoding process. Thus, we have evaluated alternative parallelization concepts and analyzed the achievable speed-up. Simple parallelization schemes were generated automatically through our compiler extension as presented in [3]. Some more sophisticated schemes had to be adapted manually (see section 4).

2 Short Overview of H.264/AVC

In H.264/AVC the video bitstream is organized in *Network Abstraction Layer* Units. The most important NAL Unit types are *Sequence Parameter Sets* (SPS), *Picture Parameter Sets* (PPS), and *Slices*. Only slices are relevant for this article, for the other ones please see [2]. A slice consists of a header and a number of macroblocks. A macroblock (MB) represents a 16x16 pixel area of the decoded picture and contains entropy-coded coefficients. Three different types of macroblocks are defined in the baseline-profile: *I-MBs* contain intra-coded coefficients of the corresponding luma and chroma components. An intra-coded macroblock has no reference to other frames but may reference other macroblocks in

the current frame. *P-MBs* are inter-coded macroblocks which contain the motion prediction from one frame in the past and the coefficients that encode only the difference to that reference (this is called *residual*). For *Skipped-MB*s, there are no coefficients stored in the bitstream. Such macroblocks use motion-vector-prediction only (see later). A slice is a container for macroblocks and represents either an entire frame or a part of a frame. Each slice of a frame can be decoded independently from other slices in that frame. In general, there are two different types of slices used in H.264/AVC baseline-profile. *I-Slices* which may contain I-MBs and Skipped-MBs, and *P-Slices* which may contain P-MBs, I-MBs, and Skipped-MBs. A macroblock itself is composed of partitions that contain 16x16 (only one partition), 16x8, 8x16, 8x8, 4x8, 8x4 or 4x4 pixels. Since a partition is the elementary unit of motion prediction, encoders typically use large partitions for homogeneous areas and small partitions for areas with varying luma/chroma values. While most of the header fields in an SPS, PPS and a slice are coded using Exp-Golomb Codes, the coefficients of a macroblock are coded using either Context-based Adaptive Binary Arithmetic Coding (CABAC) or Context-based Adaptive Variable Length Coding (CAVLC) techniques, as specified in [2].

The general decoding process for a baseline-profile compliant H.264/AVC decoder consists of the following steps:

Coefficients Parsing: For each macroblock the decoder reads the macroblock-type followed by optional motion vectors, the scaling delta (quantization delta), and the CAVLC coded residual coefficients for luma and chroma. For I-MBs the macroblock-type specifies the chosen intra prediction mode and whether it has been applied to the entire 16x16 block or to 4x4 blocks. The motion vectors of P-MBs specify the position of the referenced data in the reference frame. The process of parsing macroblock coefficients is not parallelizable because at the begining of a slice the start positions of the macroblocks are not known due to variable length coding.

Inverse Transform: After parsing the coefficients, the inverse transformation process is invoked for each macroblock. It is performed on 4x4 blocks of coefficients and consists of two phases: In the first phase the coefficients are scanned in inverse zig-zag scan-order and scaled using a function depending on a context-adaptive scaling value, which is computed from the value of the previous macroblock by adding the scaling delta for the current macroblock. For the first macroblock in a slice the scaling value is initialized to the value specified in the slice header. In the second phase the coefficients are inversely transformed using a DCT-like inverse transformation scheme, which is mainly based on integer additions, subtractions and bit-shifting operations. The inverse transformation process is the only part in the decoding process which is really straightforward to parallelize because it references no other macroblocks. In section 5 we show our results of using parallel inverse transformation.

Inter Prediction: After inverse transformation the inter prediction process can take place for P-MBs. Inter prediction uses motion vector values to add the decoded data from referenced frames to the inverse transformed residual. Motion vectors rely on motion-vector-prediction. Thus, motion vectors of a macroblock

specify only the difference to the averaged motion vector values from the macroblock to the left, above, and right-above. Motion-vector-prediction is also applied to Skipped-MBs. The accuracy of motion vectors is 1/4 pixel for luma and 1/8 pixel for chroma. This means that the referenced data has to be interpolated, which is an expensive task in the decoding process. It is possible to parallelize the inter prediction process when operating at macroblock or slice (resp. frame) level. However, as the motion-vector-prediction uses the left/above/right-above macroblock as an input, at least this task has to be done sequentially. In section 5 we show our results of using parallel inter prediction.

Intra Prediction: Intra prediction is applied to I-MBs after inverse transformation, depending on the macroblock-type. Intra prediction may use already predicted surrounding macroblocks (more precisely the left, above, left-above, and right-above MBs) of the current frame as a reference. Similar to P-MBs, the data gathered from the intra prediction is added to the inverse transformed residual. This sum is copied to the decoded buffer which is used as an input for the next step (deblocking filter). It is not a straightforward task to parallelize intra prediction due to the possible existence of referenced macroblocks. A schedule of operations based on references could be calculated and processed in correct order as described in [10].

Deblocking: The deblocking filter (DF) is applied to decoded samples on all 4x4 block edges except to those at frame borders. It uses the already deblocked pair of macroblocks above and to the left as an input for its operation. It is possible to parallelize the deblocking filter when following some processing rules as described in section 4.

More details about H.264/AVC can be found in [2], [5], and [11].

3 Related Work

Research on parallelization of H.264 is primarily done for encoders due to their higher performance requirements in comparison to the decoder. The authors of [6] compare the parallelization of MPEG-2, MPEG-4 and H.264 video encoders for multiprocessor architectures. The parallelization of H.264 is found to be the most difficult due to data dependencies and varying macro block sizes, therefore slice level parallelization is used. Encoding a separate slice per thread leads to a reduction of the workload of up to 96% (for 64 CPUs) but increases bitrate requirements by about 20% for 32 slices. The Intel Hyperthreading architecture is the platform used in [7], where the implementation uses slice level threading as well. The tradeoff between speed-up and compression efficiency is evaluated, leading to the conclusion that it is important to keep the number of slices as low as possible, even though a higher number enables higher parallelism. Therefore, in a later work([12]), a multi-level approach is adopted which processes two frames in a pipeline and uses multiple threads to work on their macroblocks. In [8] a hybrid approach is introduced, which combines the advantages of GoP level (high throughput) with slice level parallelization (low latency) using an encoding cluster. But due to the resulting increase in bitrate, the slice level parallelization

is only advantageous for up to 12 slice encoders per GoP. Research and practice indicate a trend to avoid encoding with multiple slices whenever possible, in order to get the best picture quality with regard to the bitrate. While slice level parallelization is an obvious approach to support multithreaded decoding, these results encouraged us to evaluate other approaches to partition the workload.

Most research on H.264 decoders focuses on CPU specific optimizations and hardware/software combinations: For example, [5] discusses optimizations of the reference decoder for media instruction sets. Execution time of the reference decoder is dominated by chrominance and luminance motion compensation and integer transformation. Optimizations achieve a speed up of 10.2x, 2.9x and 4.3x, respectively, for these operations. Entropy decoding and deblocking were barely accelerated at all, and in the optimized decoder they account for half of the decoding time altogether. The authors of [13] provide a detailed analysis of the performance of H.264 decoding when optimized for AltiVec SIMD architectures. While the distribution of time spent at different stages varies with source material and encoding options, the deblocking filter usually dominates because of the limited data level parallelism that could be exploited using AltiVec. In [10] a data-partitioning approach to multithreaded H.264 decoding is described, which does not depend on slices in the encoded video. Macroblock dependencies are resolved by employing a work schedule on the processing order of data partitions. This work targeted a specialized architecture consisting of RISC and media processors and is therefore not directly applicable to general purpose CPUs. An optimized H.264 decoder with block-level pipelining, which is implemented on an ARM platform, is described in [14]. While the implemented optimizations in the software result in a considerable speed-up compared to the reference decoder, the focus is on the integration of a dedicated coprocessor which can process motion compensation and integer IDCT for single macroblocks. Pure software parallelization on shared memory multiprocessors was evaluated for MPEG-2 in [9], where GoP level and slice level parallelism were compared. While both approaches deliver very good speed-ups, parallel processing of entire GoPs requires less synchronization and leads to higher efficiency, but increases delay on start and random access and has extreme memory requirements.

To our best knowledge, at time of writing, no similar evaluation of such a wide variety of parallelization approaches for H.264 decoding has been published.

4 Parallelization Concepts

4.1 Parallelization on Slice-Level

An obvious approach for parallelizing H.264/AVC is to use slices as units of parallelization because slices can be encoded/decoded independently from each other in single a frame. In general, with slice-level parallelism good speed-ups are obtainable; however, it has many disadvantages as well. In order to be useful for a variable number of CPUs, the frame should be partitioned into many slices rather than just a few. The number of slices for a frame depends on the chosen encoder which is usually not suggestible by the decoder. Furthermore, with each

additional slice of a frame, the required bitrate increases. As described in [6], using 32 slices increases the bitrate requirements by about 20%. Due to these disadvantages we have analyzed alternative parallelization schemes which are discussed in section 4.2. We have implemented slice-level parallelism using our language extension introduced in [3].

4.2 Parallelization on Macroblock-Level

Due to the results of our measurements of the workload in our H.264/AVC decoder which are illustrated in figures 1 and 2, we have analyzed the individual steps of the decoding process for further parallelization. As shown in the figures the most expensive task is inverse transformation, which has an average share of 42% in the overall decoding process. Another very expensive part is inter prediction with 31% average fraction, although for small video-resolutions (with smaller bitrates) this value decreases. Furthermore, the deblocking filter produces high workload as well, especially for low bitrates.

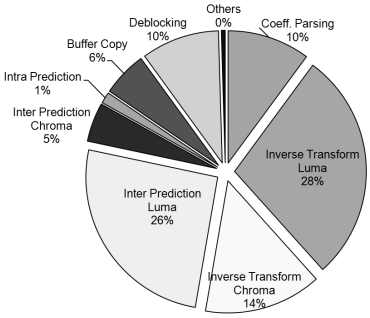

Fig. 1. Load distribution of our H.264/AVC decoder

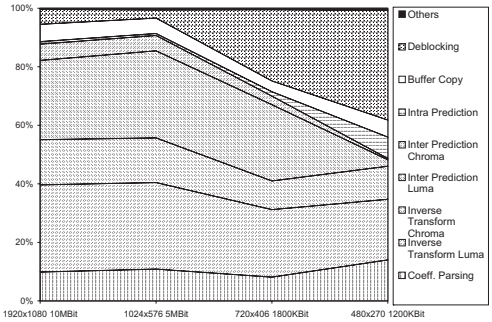

Fig. 2. Load distribution with different resolutions and bitrates

Parallelization of the Inverse Transformation: Inverse transformation can be easily parallelized because there are no dependencies between the macroblocks. However, using a high number of small processing units[1] results in very poor performance when using the default parallelization behavior of our language extension which interleaves the threads over all blocks of data. Instead, for measuring parallel inverse transformation, we used manually adapted multithreading code which processes evenly sized partitions of consecutive macroblocks. For example, when using four threads for 1000 macroblocks, each thread processes 250 consecutive macroblocks.

Parallelization of the Inter Prediction: Inter prediction can be parallelized when separating motion-vector-prediction, which has to be done sequentially, from the other steps. The remaining steps of inter prediction can be done

[1] as there are e.g. above 8000 macroblocks for a HD resolution video

simultaneously. We used the same strategy for parallelizing inter prediction as with inverse transformation (i.e. uniform areas of consecutive blocks).

Parallelization of the Deblocking Filter We have developed an approach that supports multiple threads deblocking a single frame simultaneously, while considering that deblocking of a macroblock requires read/write access to the blocks directly above and to the right. As shown in figure 3, it is necessary to synchronize all threads in a staggered pattern[2], where each worker remains at least two macroblocks left of the one above it. This way, a complete parallelization of deblocking is theoretically possible for the most part of an image, excluding 2*(numThreads-1) macroblocks at the top-left and the bottom-right of the frame[3]. This method could be extended by processing vertical ranges starting from the left at same time, but due to the need for more complex synchronization and the good speed-up of the top-down solution, this approach was abandoned.

The same solution seems to be possible for intra prediction and motion-vector-prediction, where similar speedups can be expected.

Parallelization using Macroblock Pipelining We have, furthermore, developed another parallelization scheme which we denote as *macroblock pipelining*. Thereby macroblocks of a frame are processed in parallel, each part of the decoding process working on another macroblock. Consider figure 4 to see how this pipelining works with multithreading. One thread parses the macroblock coefficients while another thread performs inverse transformation one block behind. This pipelining continues until deblocking which is performed by a fifth thread. All threads are synchronized on the basis of macroblock level. Of course, in this version pipelining has a scaling limit to five threads, however, because different parts of the pipeline take different time (e.g. coefficients parsing is much faster than inverse transform, compare to figure 2), this scheme can be extended to use several threads in the most expensive parts of the pipeline. However, we have not yet performed measurements for this extension.

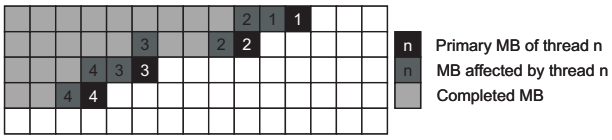

Fig. 3. Multithreaded deblocking using staggered threads

5 Performance Results and Analysis

For our measurements, we used the *Elephants Dream*[15] video which we encoded with the current release of the x264 encoder[16] using different resolutions

[2] We did not discover other decoders using this technique, but found out that [12] integrates it into the decoder.

[3] Assuming the ideal case that numThreads|imageHeightInMBs.

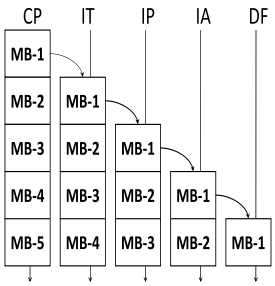

Fig. 4. Parallelization by using macroblock pipelining.

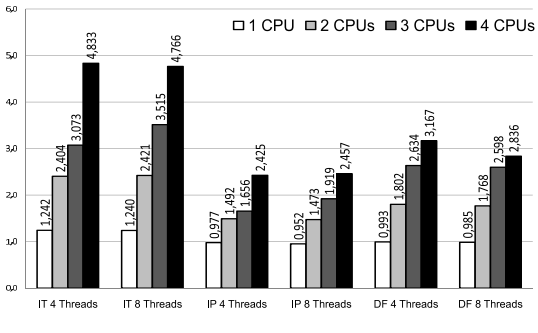

Fig. 5. Measurement results showing speed-up when parallelizing different parts of the decoding process (CP=coeff parsing, IT=inverse transformation, IP=inter prediction, IA=intra prediction, DF=deblocking).

(480x270 to 1920x1080) and bitrates (1200KBit/s to 10MBit/s). The videos encoded with x264 do contain only one slice per frame. For comparison we used encodings of QuickTime Pro [17] with default baseline-profile settings. Per default QuickTime Pro uses seven slices per frame. Our measurements have been performed on an SMP system with 4 Intel® Xeon™ CPUs, each 1.5 GHz with 256 KB Cache, and 3.5 GB RAM running Windows Server 2003® with the Microsoft .NET Runtime 2.0. For measuring less than four CPUs, we deactivated the others in the boot.ini file. We did not used the Mono Runtime for our Measurements because in its current version 1.2.2.1, it is much slower than the Microsoft .NET Runtime.

We first measured our parallelization of the elementary processing tasks independently to demonstrate the achievable speed-ups when using multithreaded implementations. The results presented in fig. 5 are calculated by comparing the duration of this elementary task between the sequential and the parallelized (n threads) versions for the input file that provided the best speed-up. Inverse transformation has no data dependencies and is therefore easily parallelizable without much synchronization overhead. It is a processor intensive operation with some data access which explains the good speed-up: Even on a single CPU a speed-up of 1.24 can be observed when using 4 threads, while 8 threads lead to a speed-up of 3.51 on three cores. Inverse transformation is obviously well suited to parallelization, as well as optimization with SIMD instructions or dedicated coprocessors (see related work).

For inter prediction, our results show that only a speed-up of 2.46 is achievable with four CPUs. Because the computational effort of inter prediction is much lower than for inverse transformation but shared memory access is higher, our measurements show a slow-down if too many threads are used (e.g. eight threads on a single CPU achieve 0.95 speed-up).

As described above, our approach to a multithreaded deblocking filter requires some synchronization between the threads, even though this is a mi-

nor influence if all threads work with a similar speed. Therefore the number of threads should be equal to the number of cores available on the system. Using two threads results in a speed-up of 1.86 on two CPUs, but with more threads the speed-up never increases beyond 1.80 on the same configuration due to synchronization overhead.

In figures 6 and 7, the results of combining parallel inverse transformation, inter prediction, and deblocking in direct comparison with macroblock pipelining and slice-level parallelism are given. For a low-resolution video (fig. 6) with only one slice per frame the combination of IT,IP, and DF achieves the best speed-up (1.86) when four threads are in use. Also macroblock pipelining (with five threads) achieves a similar result, while slice-level parallelism causes a slowdown due to the fact that only one slice is used per frame. For a high-resolution video, slice-level parallelism clearly achieves the best speed-up value of 3.2 on four CPUs, because no dependencies between slices of a frame have to be considered. However, it turns out that combining parallel inverse transformation, inter prediction, and deblocking is the best alternative (speed-up of 2.34) to slice-level parallelism.

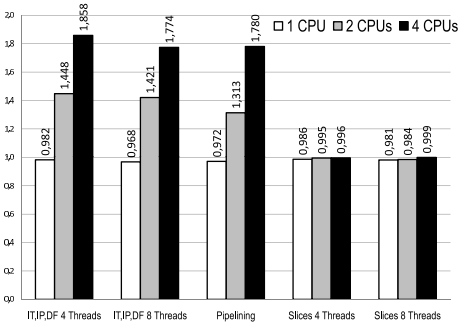

Fig. 6. Measurement results (speed-up) using the 480x270 video without slices

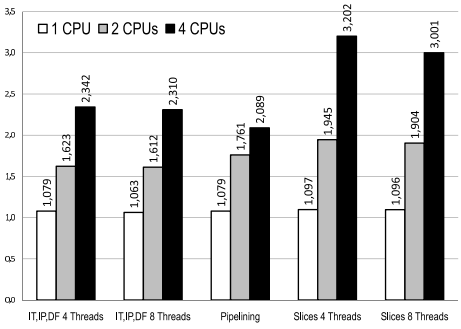

Fig. 7. Measurement results (speed-up) using the 1920x1080 video with seven slices

6 Conclusion and Further Work

Our results show that parallelizing the most expensive parts in the decoding process of an H.264/AVC baseline-profile compliant decoder results in speed-up values which are not far behind those of slice-level parallelization. From our results we can conclude, that parallelizing all three parts, inverse transformation, inter prediction, and deblocking, is the most appropriate alternative to slice-level parallelism. As further work, we continue our measurements with more than four CPUs and an extended version of macroblock-pipelining.

References

1. Zhou, X., Li, E.Q., Chen, Y.K.: Implementation of H.264 Decoder on General-Purpose Processors with Media Instructions. (January 2003) in SPIE Conf. on Image and Video Communications and Processing.
2. ISO/IEC JTC 1/SC 29/WG 11: ISO/IEC FDIS 14496-10: Information Technology - Coding of audio-visual objects - Part 10: Advanced Video Coding. (March 2003)
3. Lampl, O., Stellnberger, E., Böszörményi, L.: Programming language concepts for multimedia application development. In David E. Lightfoot, C.A.S.E., ed.: LNCS 4228 Modular Programming Languages. Volume 1. (May 2006)
4. Mono: Open source .net development framework (2007) www.mono-project.com.
5. Zhou, X., Li, E.Q., Chen, Y.K.: Implementation of H.264 decoder on general-purpose processors with media instructions. In Vasudev, B., Hsing, T.R., Tescher, A.G., Ebrahimi, T., eds.: Image and Video Communications and Processing 2003. Edited by Vasudev, Bhaskaran; Hsing, T. Russell; Tescher, Andrew G.; Ebrahimi, Touradj. Proceedings of the SPIE, Volume 5022, pp. 224-235. (May 2003) 224–235
6. Jacobs, T., Chouliaras, V., Mulvaney, D.: Thread-parallel MPEG-2, MPEG-4 and H.264 video encoders for SoC multi-processor architectures. In: IEEE Transactions on Consumer Electronics. Volume 52., IEEE (Feb 2006) 269–275
7. Chen, Y.K., Tian, X., Ge, S., Girkar, M.: Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures . In: 18th International Parallel and Distributed Processing Symposium. Volume 1. (April 2004) 63–72
8. Rodriguez, A., Gonzalez, A., Malumbres, M.P.: Hierarchical Parallelization of an H.264/AVC Video Encoder. In: International Symposium on Parallel Computing in Electrical Engineering, 2006. . (Sept 2006) 363–368
9. Bilas, A., Fritts, J., Singh, J.P.: Real-time parallel MPEG-2 decoding in software. Technical Report TR-516-96 (1996)
10. van der Tol, E.B., Jaspers, E.G., Gelderblom, R.H.: Mapping of H.264 decoding on a multiprocessor architecture. In Vasudev, B., Hsing, T.R., Tescher, A.G., Ebrahimi, T., eds.: Image and Video Communications and Processing 2003. Edited by Vasudev, Bhaskaran; Hsing, T. Russell; Tescher, Andrew G.; Ebrahimi, Touradj. Proceedings of the SPIE, Volume 5022, pp. 707-718 (2003). (May 2003) 707–718
11. Richardson, I.: H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia. John Wiley & Sons Ltd., The Atrium. Southern Gate, Chichester, West Sussex PO19 8SQ, England (2003)
12. Chen, Y.K., Li, E.Q., Zhou, X., Ge, S.L.: Implementation of H.264 Encoder and Decoder on Personal Computers. In: Journal of Visual Communications and Image Representations. (2005) 509–532
13. Alvarez, M., Salami, E., Ramirez, A., Valero, M.: A performance characterization of high definition digital video decoding using H.264/AVC. In: Proceedings of the IEEE International Workload Characterization Symposium, IEEE (Oct 2005)
14. Wang, S.H., Peng, W.H., He, Y., Lin, G.Y., Lin, C.Y., Chang, S.C., Wang, C.N., Chiang, T.: A platform-based MPEG-4 advanced video coding (AVC) decoder with block level pipelining. In: Proc. of the 2003 Joint Conference of the Fourth International Conference on Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Volume 1. (Dec 2003)
15. Orange Open Movie Project: elephants dream (2007) http://orange.blender.org/.
16. Aimar, L., Merritt, L., et al.: x264 - a free h264/avc encoder (2007) http://www.videolan.org/developers/x264.html.
17. Apple Inc.: QuickTime Pro (2007) www.apple.com/de/quicktime/pro/.