An Efficient SIMD Implementation of the H.265 Decoder for Mobile Architecture

Massimo Bariani[1], Paolo Lambruschini[1](✉), Marco Raggio[1], and Luca Pezzoni[2]

[1] DITEN, University of Genoa, Genoa, Italy
{bariani,lambruschini,raggio}@dibe.unige.it
[2] AST, STMicroelectronics, Milan, Italy
luca.pezzoni@st.com

Abstract. This paper focuses on an efficient optimization of the H.265 video decoder on suitable architectures for mobile devices. The solutions developed to support the H.265 features, and the achieved performances are shown. The most demanding modules have been optimized with Single Instruction Multiple Data (SIMD) instructions and we keep in special account the memory handling, with the minimization of the memory transfer. The effectiveness of the proposed solutions has been demonstrated on ARM architecture. In particular, we have selected the dual-core Cortex A9 processor with NEON SIMD extension.

Keywords: H.265 · SIMD optimization · ARM NEON · Video compression · Mobile application

1 Introduction

Video compression algorithms play a fundamental role in enjoying the multimedia contents, allowing high video quality thanks to the remarkable enhancement of video resolutions. This is especially true in mobile environment where wired networks are not present and the wireless bandwidth can be reduced. Several compression standards have been developed in the past years: VC-1 also known with the name SMPTE 421M[1], MPEG-2/H.262 [2], and H.264/AVC [3][4].

The nowadays trend sees a constant improvement of screen resolutions and display capabilities of mobile devices. The last smartphones and tablets can address video in Ultra High Definition (UHD) format [5], with a resolution of 3840 × 2160 (8.3 megapixel). In the future, mobile devices will handle even higher resolutions, may be the Full Ultra High Definition (FUHD) format, with a resolution of 7680×4320 (33.2 megapixel).

The internet traffic has grown significantly in recent years, especially boosted by video content. Moreover, in next future it will rise approximately 85 percent of global consumer traffic [6]. Video will make up about 72% of the data consumed by mobile devices by 2019[7].

In this scenario, the continuous improvement of video resolutions leads to very high bandwidth occupation and this can be a serious issue especially on mobile network.

V. Murino et al. (Eds.): ICIAP 2015 Workshops, LNCS 9281, pp. 563–570, 2015.
DOI: 10.1007/978-3-319-23222-5_68

Therefore, the performance enhancement of compression algorithms plays an important role, reducing the amount of data transmitted to enjoy multimedia contents.

To face the challenge of transfer very high resolution video contents, in recent years, a new video compression standard was developed: High Efficiency Video Coding (HEVC/H.265) [8] [9]. HEVC/H.265 aim is to increase the compression efficiency by 50% if compared to the H.264/AVC, while maintaining the same level of visual quality.

In Fig. 1, the HEVC decoder block diagram is illustrated. The main difference with H.264 previous standard is the picture partitioning that is not fixed to 16x16 pixels blocks, but is flexible.

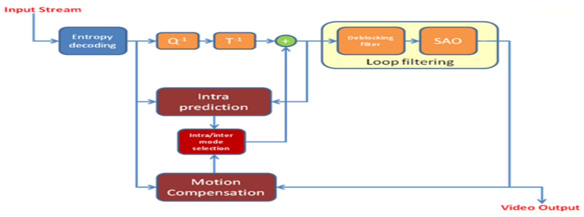

Fig. 1. HEVC decoder block diagram

Analogous to the concept of H.264 macroblock, HEVC defines the Coding Tree Unit (CTU) having maximum dimension 64x64 pixels [8][9]. The CTU is subdivided in square areas named Coding Unit (CU), with a quadtree scheme, which are the base blocks for the intra and inter coding. Their size can vary from 64x64 to 8x8 pixels.

The basic unit used in the prediction process is the Prediction Unit (PU). Each CU (NxN dimension) can contain: 1 PU (NxN), 2 PUs (NxN/2 or N/2xN), or 4 PUs (N/2xN/2), and their dimension can range from 64x64 to 4x4 pixels. The Discrete Cosine Transform (DCT) is utilized in the HEVC standard for coding the residual. Transform and quantization are applied to the Transform Unit (TU). Each CU can be spitted in several TUs having size ranging from 32x32 to 4x4 pixels.

The intra prediction keeps the same structure of the H.264 algorithm, but the number of intra modes increases from 9 to 35.

Moreover, one additional module is involved in the loop filtering process: the Sample Adaptive Offset (SAO). The basic concept of SAO filter is to classify reconstructed pixels in different categories using intensity or edge properties. An offset is added to the pixels in each category to reduce distortion. SAO is applied after the deblocking filter, as shown in Fig. 1. Further details of HEVC algorithm can be found in literature [8] or in the ITU-T formal publication [9].

In this paper, we will focus on an optimized software implementation of the HEVC/H.265 decoder, exploiting data-level parallelism. In the description of the proposed approach, special regards is dedicated to the new H.265 features, highlighting the issues in exploiting data-level parallelism and proposing our solutions. Starting from our previous work on HEVC [15] [16], we have decided to further optimize the decoder in order to increase the achieved performance. New strategy to exploit the ARM architecture will be shown on this paper.

2 SIMD Implementation

Starting from results on performance achieved with test performed in our previous work [15] [16], we have decided to focus the optimization on the most computation demand modules of H.265.

The analysis of the H.265 decoder profiles shows that even after our past optimization [15] [16] the most onerous module is the Motion Compensation (MC), similarly to what happened before. Besides MC, inverse transform and deblocking filter are still considerably time-consuming.

Our approach in previous software development was to use the Single-Instruction-Multiple-Data (SIMD) instructions to exploit the data level parallelism during the execution. We have decided to push more deeply this SIMD optimization with a better memory handling. One of the major limits of the NEON intrinsic [14] is the impossibility to specify the alignment in the operations of load/store. We have developed specific assembler optimization to bypass this limit. The architecture utilized to validate the effectiveness of the presented work is ARM Cortex A9 with NEON extension [14].

In order to take advantage of SIMD instructions, usually software developers have to strongly modify the original source code. Main changes regard the data manipulation: SIMD instructions often require having data ordered in specific ways to fully exploit their parallelism. The amount of control-flow code is another factor that can substantially impact the performance of a SIMD implementation. In order to effectively increment the performance, the code should be linear with a well-defined flow of operations. All these changes are necessary for a generic SIMD implementation, regardless of the particular instruction-set.

In the following, the details about the optimization of H.265 modules are provided.

2.1 Deblocking Filter

Blocking artifacts are well known, since they are one of the most visible distortions due to block based video compression algorithm. For this reason, H.265 filters data before the visualization. The filtering module shown in the decoder scheme of Fig. 1 is performed in loop (loop filtering). This means that the filtered data are used inside the process loop for the motion compensation of the next frames. In order to improve the video quality, H.265 exploits a H.264-like deblocking filter, followed by the newly added SAO filter.

The deblocking filter [8] [9] processes the edges among all the possible block partitions (CUs, PUs, TUs), except for 4x4 block that are not filtered. Each edge can be processed using either a strong or a weak filter.

The first step of the optimization was to exploit SIMD instruction for minimizing the number of memory accesses. In the ANSI-C implementation several instructions for loading and storing pixels were executed. Code reorganization has been performed in order to efficiently group load/store operations at the beginning/end of each vertical and horizontal phase. Each SIMD instruction loads from memory four elements in the horizontal phase and eight elements in the vertical phase. The memory access have been optimized exploiting prefetch from cache and aligned load/store from memory. Aligned load are not always possible, but when are possible the performance increase can be significant. An example of code developed for prefetch is shown in Fig. 2.

```
#define __pldw(x) asm volatile ( " pld [%[addr]]\n" :: [addr] "r" (x));
#define __pld_2i(x, o) asm volatile ( " pld [%[addr], %[offset]]\n" :: [addr] "r" (x), [offset] "I" (o));

#define __pld_2r(x, o) asm volatile ( " pld [%[addr], %[offset]]\n" :: [addr] "r" (x), [offset] "r" (o));
```

Fig. 2. Source code for prefetch

An example of code developed for aligned load/store is shown in Fig. 3.

```
#define _vld1u8_alig(y,x,a) asm("vld1.8 %h[out], [%[addr], :%c[alig]]\n" : [out]"=w"(y) : [addr]"r"(x), [alig]"I"(a))

#define _vst1u8_alig(x,y,a) asm ("vst1.8 %h[in], [%[addr], :%c[alig]]\n" : : [addr]"r"(x), [in]"w"(y), [alig]"I"(a))
```

Fig. 3. Source code for aligned load

We have decided to write assembler code for prefetch and aligned load/store because NEON extension does not handle this feature. Even load/store with increment are exploited, in order to perform load/store and pointer updating with one instruction. All optimization with aligned load/store are possible only when data are aligned in memory. We have aligned the frame at 64 byte, but not all the operation in the decoder access to aligned data. Sometimes pixels involved in the process are unaligned. In the deblocking filter aligned load/store can be well exploited in the horizontal phase, while in the vertical the gain is negligible.

2.2 SAO Filter

The Sample Adaptive Offset (SAO) process [8] [9] is applied to the reconstruction signal after the deblocking filter. It utilizes a set of offset values given in the slice header. SAO divides a picture into CTU-aligned regions to obtain local statistical information. The aim of SAO is to reduce the distortion by adding an offset to pixels of each category in these regions [15] [16].

There are four patterns selecting the type of processing for each region (see Fig. 4). The selected pattern is sent in the bit-stream.

For every SAO type four categories can be selected, the category selection must be also computed at the decoder side. Categories are calculated comparing the value of the current pixel (denoted as "C" in Fig. 4) with the value of the two neighbouring pixels. Each category selects an offset value to be added to the original pixel.

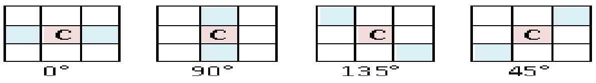

Fig. 4. 3-pixel patterns for the pixel classification in EO

One of the main issues of SAO implementation is that the category must always be computed referring to the original pixels, but the pixels are modified by the process itself. For this reason usual implementations works on two buffers, making a copy of input data before starting the process. In our implementation, we directly work on the decoded frame buffer in order to avoid copies, but we always keep in memory one or two CTU lines depending on the EO type. For example, in the vertical edge filter (90° pattern), we always load a line ahead the current line, we keep both current and next line in two temporal buffers while filtering the current line and writing the results to the decoded frame buffer. After the whole line has been completed, we switch next and current buffers and load a new line. For supporting this mechanism for all the edge offset types, we have to additionally store one frame line and one CTU column in order to manage the CTU border filtering.

Even though the operations slightly differ between the four EO types, the SIMD implementation structure is similar. The main difference is the way data is loaded into vector registers. We always load CTU-aligned vectors and misalign them to obtain the needed input pixels. For example, in the horizontal edge filter, after we load the current line, we shift the vector register one element to obtain the vector of pixels on the right.

In the SAO module we have exploited prefetch and aligned load/store like previously shown for debloking filter. For the SAO module, to exploit aligned load/store we transfer more data than what strictly necessary.

2.3 Motion Compensation

The motion compensation (MC) is the most expensive task in H.265 decoder. It has the purpose to create the temporal predictor that will be added to the decoded prediction error for creating the reconstructed block. The temporal predictor is created from the previously decoded images called reference frame.

The MC could be unidirectional when it uses only a predictor coming from one reference frames, or could be bidirectional. In this last case, it takes two different predictors coming from two different reference frames, and it merges them for creating the final predictor (see Fig. 5).

The MC could be applied to each PU, so in the worst case we could have up to two different predictors for each 4x4 smallest PU.

For creating the temporal predictors, the MC also needs to make a pixel interpolation because the movement between the actual images and the reference frames could be fractionally.

In H.265 the luma interpolation has a ¼ of pixel precision and it uses a separable horizontal and vertical FIR filter at 8 taps.

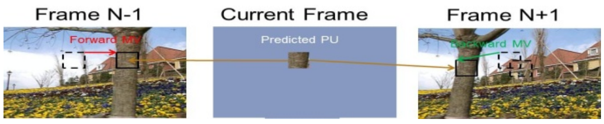

Fig. 5. Uni/Bidirectional MC.

For the Chroma Interpolation the precision is up to 1/8 pixels and it uses a separable 4 taps FIR filter.

We did a deep function specialization in order to separate luma from chroma motion compensation, monodirectional from bidirectional prediction, and no interpolation, only horizontal interpolation, only vertical interpolation or diagonal one. This process results in having 24 different functions specialized for each case.

Another modification that we have done is to work always with 8x8 PUs, also when we have a 4x4 PU. In this way we could always use all the 128-bit NEON registers in an efficient way. In this way prefetch and aligned load/store can be well exploited and time for memory transfer is minimized.

3 Results

In order to evaluate the achieved performance after the optimization process, several tests have been performed on a set of test sequences, addressing 720p resolution and different coding features. Some of the results are shown in the following tables. The utilized architecture is the ARM Cortex A9 @1.2GHz.

The following tables show results for different quantization values (QP) and different configurations: low-delay using B slices (LB), low-delay using P slices (LP), and random-access (RA). The decoder performance is measured in frames per second.

Table 1. Decoder Performance on Standard Sequences

Sequences	Number of frames	QP	Optimized decoder [fps]			JCT-VC decoder [fps]			Speed-up		
			LB	LP	RA	LB	LP	RA	LB	LP	RA
FourPeople	600	22	30.06	35.31	27.49	4.71	4.94	4.41	6.38	7.15	6.23
		27	38.43	49.01	31.96	5.86	6.16	5.05	6.56	7.96	6.33
		32	44.34	57.44	35.49	6.63	6.87	5.39	6.69	8.36	6.58
		37	50.09	65.06	38.10	7.34	7.45	5.70	6.82	8.73	6.68
Johnny	600	22	29.80	36.33	28.96	4.69	5.01	4.37	6.35	7.25	6.63
		27	39.10	52.14	36.65	5.82	6.41	5.12	6.72	8.13	7.16
		32	45.13	60.21	39.26	6.54	7.07	5.45	6.90	8.52	7.20
		37	49.86	66.86	42.25	7.28	7.55	5.74	6.85	8.86	7.36
KristenAndSara	600	22	27.17	32.88	26.12	4.16	4.58	4.00	6.53	7.18	6.53
		27	34.52	43.69	31.75	5.02	5.56	4.57	6.88	7.86	6.95
		32	40.46	51.96	35.47	5.69	6.20	4.94	7.11	8.38	7.18
		37	45.70	58.36	38.64	6.39	6.74	5.29	7.15	8.66	7.30
SlideEditing	300	22	48.21	64.23	39.93	8.38	8.31	5.98	5.75	7.73	6.68
		27	51.61	67.43	40.56	8.48	8.40	6.01	6.09	8.03	6.75
		32	52.56	71.62	38.77	8.54	8.45	5.96	6.15	8.48	6.50
		37	55.08	73.67	39.35	8.71	8.61	6.05	6.32	8.56	6.50
SlideShow	500	22	37.02	45.42	31.25	5.90	5.96	4.90	6.27	7.62	6.38
		27	40.83	50.40	33.38	6.19	6.22	5.08	6.60	8.10	6.57
		32	46.22	56.51	34.96	6.48	6.53	5.25	7.13	8.65	6.66
		37	49.12	60.55	37.02	6.80	6.83	5.36	7.22	8.86	6.91

In Table 1, the decoding performance for a set of standard sequences is shown. The execution time is strongly influenced by the input stream configuration, but the optimized code is able to streams at 30 frames per seconds (fps). The random-access configuration shows the lower results, whereas using low-delay with only P slices lead to better performance.

In the same table, we compare our results to the performance obtained with the reference decoder of the Joint Collaborative Team on Video Coding (JCT-VC), in order to put the here presented work in perspective. In particular, we refer to the H.265 decoder from the official software repository [18]. In the last three columns of the tables, we show the speed-up of our optimized decoder vs. the JCT-VC decoder. As can be noticed, the achieved performances are increased respect to our previous work.

4 Conclusions and Future Work

Video compression algorithms are computationally heavy and each new video standard introduces novelties that increase the rate of compression, but also increase the complexity of the algorithms [19]. The major issue is to conciliate the real-time requirement with the execution time achievable with the architectures available in mobile environment with a computational power usually limited.

The work presented in this paper illustrates our solutions, implemented and evaluated on a widely spread processor in mobile panorama, the ARM Cortex-A9. The experimental results show that the resulting H.265 decoder is able to achieve real-time performance for 720p video sequences.

Ongoing work is addressing multi-core implementation based on previous research on VC1 decoder [20][21]. The main issue in exploiting multi-core in video decoders is the data dependency. Usually, little portions of code can be concurrently executed without having to wait for data outgoing from other modules. Moreover, most of the available parallelism resides at block level, but this fine-grain parallelism has the drawback of data communication between cores. A promising solution seems to be the subdivision of the processing into two stages: the first module for parsing and entropy decoding, and the second for the rest of the decoding process. Referring to Fig. 1, the first core will execute the entropy decoding on frame N+1, while the second core will decode the frame N. In this way, it is possible subdivide the computational weight in two cores limiting the amount of data communication.

References

1. VC-1 Compressed Video Bitstream Format and Decoding Process. SMPTE 421M-2006, SMPTE Standard (2006)
2. International Telecommunication Union: ITU-T Recommendation H.262 (11/94): generic coding of moving pictures and associated audio information – part 2: video (1994)
3. Advanced Video Coding, ITU-T Rec. H.264 and ISO/IEC 14496-10:2009, March 2010
4. Wiegand, T., Sullivan, G.J., Bjøntegaard, G., Luthra, A.: Overview of the H.264/AVC video coding standard. IEEE Trans. Circuits Syst. Video Tech. **13**(7), 560–576 (2003)
5. Nakasu, E.: Super Hi-Vision on the Horizon: A Future TV System That Conveys an Enhanced Sense of Reality and Presence. IEEE Consumer Electronics Magazine **1**(2), 36–42 (2012)
6. Cisco Corporation: Cisco Visual Networking Index: Forecast and Methodology, 2013-2018, June 2014. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html

7. Cisco Corporation: Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014–2019, February 2015. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html
8. Sullivan, G.J., Ohm, J.R., Han, W.J., Wiegand, T.: Overview of the High Efficiency Video Coding (HEVC) Standard. Circuits and Systems for Video Technology. IEEE Trans. on Circuits and Systems for Video Technology **22**(2), 1649–1668 (2012)
9. H.265: High efficiency video coding, ITU-T Rec.H.265 and ISO/IEC 23008-2 MPEG-H Part 2, November 2013
10. Guo, Z., Zhou, D., Goto, S.: An optimized MC interpolation architecture for HEVC. In: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), March 2012
11. High efficiency video coding, Recommendation ITU-T H.265 / ISO/IEC 23008-2:2013, April 2013
12. Bossen, F., Bross, B., Suhring, K., Flynn, D.: HEVC Complexity and Implementation Analysis. IEEE Trans. On Circuits and Systems for Video Technology, December 2012
13. Alvarez-Mesa, M., Chi, C.C., Juurlink, B., George, V., Schierl, T.: Parallel video decoding in the emerging HEVC standard. In: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), March 2012
14. ARM White Paper: The ARM Cortex-A9 Processors, September 2009
15. Bariani, M., Lambruschini, P., Raggio, M.: An optimized SIMD implementation of the HEVC/H.265 video decoder. In: Wireless Telecommunications Symposium (WTS) (2014). doi:10.1109/WTS.2014.6835018
16. Bariani, M., Lambruschini, P., Raggio, M.: An optimized software implementation of the HEVC/H.265 video decoder. In: 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC). doi:10.1109/CCNC.2014.7056307
17. Bariani, M., Lambruschini, P., Raggio, M.: An Efficient Multi-Core SIMD Implementation for H.264/AVC Encoder. VLSI Design **2012**, 14 (2012). Article ID 413747, doi:10.1155/2012/413747
18. https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/
19. Ohm, J.R., Sullivan, G.J., Schwarz, H., Tan, T.K., Wiegand, T.: Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC). IEEE Trans. on Circuits and Systems for Video Technology **22**(12), 1669–1684 (2012)
20. Bariani, M., Lambruschini, P., Raggio, M.: VC-1 decoder on STMicroelectronics P2012 architecture. In: Proc. of 8th Annual Intl. Workshop 'STreaming Day', Univ. of Udine, Udine, IT, September 2010. http://stday2010.uniud.it/stday2010/stday_2010.html
21. Paulin, P.: Programming challenges & solutions for multi-processor SoCs: an industrial perspective. In: 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)