# HEVC Complexity and Implementation Analysis

Frank Bossen, *Member, IEEE,* Benjamin Bross, *Student Member, IEEE,* Karsten Sühring, and David Flynn

***Abstract*—Advances in video compression technology have been driven by ever-increasing processing power available in software and hardware. The emerging High Efficiency Video Coding (HEVC) standard aims to provide a doubling in coding efficiency with respect to the H.264/AVC high profile, delivering the same video quality at half the bit rate. In this paper, complexity-related aspects that were considered in the standardization process are described. Furthermore, profiling of reference software and optimized software gives an indication of where HEVC may be more complex than its predecessors and where it may be simpler. Overall, the complexity of HEVC decoders does not appear to be significantly different from that of H.264/AVC decoders; this makes HEVC decoding in software very practical on current hardware. HEVC encoders are expected to be several times more complex than H.264/AVC encoders and will be a subject of research in years to come.**

***Index Terms*—High Efficiency Video Coding (HEVC), video coding.**

## I. Introduction

THIS PAPER gives an overview of complexity and implementation issues in the context of the emerging High Efficiency Video Coding (HEVC) standard. The HEVC project is conducted by the Joint Collaborative Team on Video Coding (JCT-VC), and is a joint effort between ITU-T and ISO/IEC. Reference software, called the HEVC test model (HM), is being developed along with the draft standard. At the time of writing, the current version of HM is 8.0, which corresponds to the HEVC text specification draft 8 [1]. It is assumed that the reader has some familiarity with the draft HEVC standard, an overview of which can be found in [2].

Complexity assessment is a complex topic in itself, and one aim of this paper is to highlight some aspects of the HEVC design where some notion of complexity was considered. This is the topic of Section II.

A second aim of this paper is to provide and discuss data resulting from profiling existing software implementations of HEVC. Sections III and IV present results obtained with the HM encoder and decoder, and Section V discusses an optimized implementation of a decoder.

Manuscript received May 26, 2012; revised August 19, 2012; accepted August 24, 2012. Date of publication October 2, 2012; date of current version January 8, 2013. This paper was recommended by Associate Editor H. Gharavi.

F. Bossen is with DOCOMO Innovations, Inc., Palo Alto, CA 94304 USA (e-mail: bossen@docomoinnovations.com).

B. Bross and K. Sühring are with the Image and Video Coding Group, Fraunhofer Institute for Telecommunications–Heinrich Hertz Institute, Berlin 10587, Germany (e-mail: benjamin.bross@hhi.fraunhofer.de; karsten.suehring@hhi.fraunhofer.de).

D. Flynn is with Research In Motion, Ltd., Waterloo, ON N2L 3W8, Canada (e-mail: dflynn@iee.org).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSVT.2012.2221255

## II. Design Aspects

### A. Quadtree-Based Block Partitioning

HEVC retains the basic hybrid coding architecture of prior video coding standards, such as H.264/AVC [3]. A significant difference lies in the use of a more adaptive quadtree structure based on a coding tree unit (CTU) instead of a macroblock. In principle, the quadtree coding structure is described by means of blocks and units. A block defines an array of samples and sizes thereof, whereas a unit encapsulates one luma and corresponding chroma blocks together with syntax needed to code these. Consequently, a CTU includes coding tree blocks (CTB) and syntax specifying coding data and further subdivision. This subdivision results in coding unit (CU) leaves with coding blocks (CB). Each CU incorporates more entities for the purpose of prediction, so-called prediction units (PU), and of transform, so-called transform units (TU). Similarly, each CB is split into prediction blocks (PB) and transform blocks (TB). This variable-size, adaptive approach is particularly suited to larger resolutions, such as $4k \times 2k$, which is a target resolution for some HEVC applications. An exemplary CB and TB quadtree structure is given in Fig. 1. All partitioning modes specifying how to split a CB into PBs are depicted in Fig. 2. The decoding of the quadtree structures is not much of an additional burden because the quadtrees can be easily traversed in a depth-first fashion using in a z-scan order. Partitioning modes for inter picture coded CUs feature nonsquare PUs. Support for these nonsquare shapes requires additional logic in a decoder as multiple conversions between z-scan and raster scan orders may be required. At the encoder side, simple tree-pruning algorithms exist to estimate the optimal partitioning in a rate-distortion sense [4], [5].

Sections below describe various tools of HEVC and review complexity aspects that were considered in the development of the HEVC specification, using H.264/AVC as a reference where appropriate.

### B. Intra Picture Prediction

Intra picture prediction in HEVC is quite similar to H.264/AVC. Samples are predicted from reconstructed samples of neighboring blocks. The mode categories remain identical: DC, plane, horizontal/vertical, and directional; although the nomenclature has somewhat changed with planar and angular, respectively, corresponding to H.264/AVC's plane and directional modes. A significant change comes from the introduction of larger block sizes, where intra picture prediction

1051-8215/$31.00 © 2012 IEEE

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000001
IPR2024-00604 (Amazon.com, Inc., Amazon.com Services LLC v. Nokia Technologies Oy)

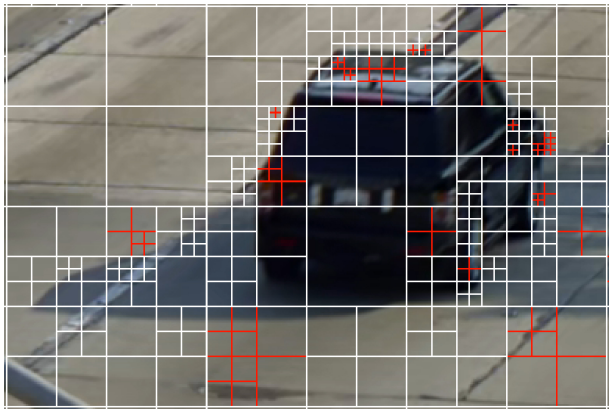


Fig. 1. Detail of 4k × 2k *Traffic* sequence showing the coding block (white) and nested transform block (red) structure resulting from recursive quadtree partitioning.

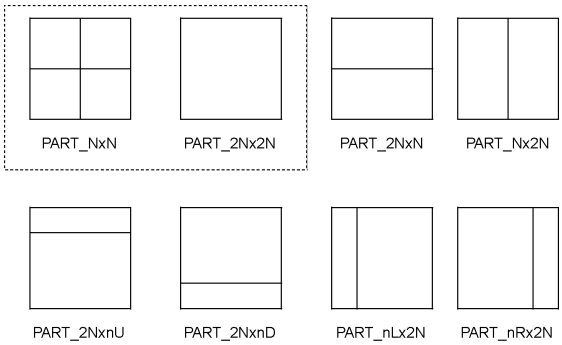



Fig. 2. All prediction block partitioning modes. Inter picture coded CUs can apply all modes, while intra picture coded CUs can apply only the first two.

using one of 35 modes may be performed for blocks of size up to $32 \times 32$ samples. The smallest block size is unchanged from H.264/AVC at $4 \times 4$ and remains a complexity bottleneck because of the serial nature of intra picture prediction.

For the DC, horizontal, and vertical modes an additional postprocess is defined in HEVC wherein a row and/or column is filtered such as to maintain continuity across block boundaries. This addition is not expected to have an impact on the worst case complexity since these three modes are the simplest to begin with.

In the case of the planar mode, consider that the generating equations are probably not adequate to determine complexity, as it is possible to easily incrementally compute predicted sample values. For the H.264/AVC plane mode it is expected that one 16-bit addition, one 16-bit shift, and one clip to the 8-bit range are required per sample. For the HEVC planar mode, this becomes three 16-bit additions and one 16-bit shift. These two modes are thus expected to have similar complexities.

The angular modes of HEVC are more complex than the directional H.264/AVC modes as multiplication is required. Each predicted sample is computed as $((32 - w) \cdot x_i + w \cdot x_{i+1} + 16) >> 5$, where $x_i$ are reference samples and $w$ is a weighting factor. The weighting factor remains constant across a predicted row or column that facilitates single-instruction multiple-data (SIMD) implementations. A single function may be used to cover all 33 prediction angles, thereby reducing the amount of code needed to implement this feature.

As in H.264/AVC, reference samples may be smoothed prior to prediction. The smoothing process is the same although it is applied more selectively, depending upon the prediction mode.

From an encoding perspective, the increased number of prediction modes (35 in HEVC versus 9 in H.264/AVC) will require good mode selection heuristics to maintain a reasonable search complexity.

### C. Inter Picture Prediction

Inter picture prediction, or motion compensation, is conceptually very simple in HEVC, but comes with some overhead compared to H.264/AVC. The use of a separable 8-tap filter for luma sub-pel positions leads to an increase in memory bandwidth and in the number of multiply-accumulate operations required for motion compensation. Filter coefficients are limited to the 7-bit signed range to minimize hardware cost. In software, motion compensation of an $N \times N$ block consists of $8 + 56/N$ 8-bit multiply-accumulate operations per sample and eight 16-bit multiply-accumulate operations per sample. For chroma sub-pel positions, a separable 4-tap filter with the same limitations as for the luma filter coefficients is applied. This also increases the memory bandwidth and the number of operations compared to H.264/AVC where bilinear inter polation is used for chroma sub-pel positions.

Another area where the implementation cost is increased is the intermediate storage buffers, particularly in the bipredictive case. Indeed, two 16-bit buffers are required to hold data, whereas in H.264/AVC, one 8-bit buffer and one 16-bit buffer are sufficient. In an HEVC implementation these buffers do not necessarily need to be increased to reflect the maximum PB size of $64 \times 64$. Motion compensation of larger blocks may be decomposed into, and processed in, smaller blocks to achieve a desired trade-off between memory requirements and a number of operations.

H.264/AVC defines restrictions on motion data that are aimed at reducing memory bandwidth. For example, the number of motion vectors used in two consecutive macroblocks is limited. HEVC adopts a different approach and defines restrictions that are much simpler for an encoder to conform to: the smallest motion compensation blocks are of luma size $4 \times 8$ and $8 \times 4$, thereby prohibiting $4 \times 4$ inter picture prediction, and are constrained to make use of only the first reference picture list (i.e., no biprediction for $4 \times 8$ and $8 \times 4$ luma blocks).

HEVC introduces a so-called merge mode, which sets all motion parameters of an inter picture predicted block equal to the parameters of a merge candidate [6]. The merge mode and the motion vector prediction process optionally allow a picture to reuse motion vectors of prior pictures for motion vector coding, in essence similar to the H.264/AVC temporal direct mode. While H.264/AVC downsamples motion vectors to the $8 \times 8$ level, HEVC further reduces memory requirements by keeping a single motion vector per $16 \times 16$ block.

HEVC offers more ways to split a picture into motion-compensated partition patterns. While this does not significantly impact a decoder, it leaves an encoder with many more

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000002

choices. This additional freedom is expected to increase the complexity of encoders that fully leverage the capabilities of HEVC.

### *D. Transforms and Quantization*

H.264/AVC features 4-point and 8-point transforms that have a very low implementation cost. This low cost is achieved by relying on simple sequences of shift and add operations. This design strategy does not easily extend to larger transform sizes, such as 16- and 32-point. HEVC thus takes a different approach and simply defines transforms (of size $4 \times 4$, $8 \times 8$, $16 \times 16$, and $32 \times 32$) as straightforward fixed-point matrix multiplications. The matrix multiplications for the vertical and horizontal component of the inverse transform are shown in (1) and (2), respectively

$$\mathbf{Y} = s\left(\mathbf{C}^T \cdot \mathbf{T}\right) \tag{1}$$

$$\mathbf{R} = \mathbf{Y}^T \cdot \mathbf{T} \tag{2}$$

where $s()$ is a scaling and saturating function that guarantees that values of $\mathbf{Y}$ can be represented using 16 bit. Each factor in the transform matrix $\mathbf{T}$ is represented using signed 8-bit numbers. Operations are defined such that 16-bit signed coefficients $\mathbf{C}$ are multiplied with the factors and, hence, greater than 16-bit accumulation is required. As the transforms are integer approximations of a discrete cosine transform, they retain the symmetry properties thereof, thereby enabling a partial butterfly implementation. For the 4-point transform, an alternative transform approximating a discrete sine transform is also defined.

Although there has been some concern about the implementation complexity of the 32-point transform, data given in [7] indicates 158 cycles for an $8 \times 8$ inverse transform, 861 cycles for a $16 \times 16$ inverse transform, and 4696 cycles for a $32 \times 32$ inverse transform on an Intel processor. If normalizing these values by the associated block sizes, respectively, 2.47, 3.36, and 4.59 cycles are required per sample. The time cost per sample of a $32 \times 32$ inverse transform is thus less than twice that of an $8 \times 8$ inverse transform. Furthermore, the cycle count for larger transforms may often be reduced by taking advantage of the fact that the most high-frequency coefficients are typically zero. Determining which bounding subblock of coefficients is nonzero is facilitated by using a $4 \times 4$ coding structure for the entropy coding of transform coefficients. The bounding subblock may thus be determined at a reasonable granularity ($4 \times 4$) without having to consider the position of each nonzero coefficient.

It should also be noted that the transform order is changed with respect to H.264/AVC. HEVC defines a column–row order for the inverse transform. Due to the regular uniform structure of the matrix multiplication and partial butterfly designs, this approach may be preferred in both hardware and software. In software it is preferable to transform rows, as one entire row of coefficients may easily be held in registers (a row of thirty-two 32-bit accumulators requires eight 128-bit registers, which is implementable on several architectures without register spilling). This property is not necessarily maintained with more irregular but fully decomposed transform designs, which look attractive in terms of primitive operation counts, but require a greater number of registers and software operations to implement. As can be seen from (1), applying the transpose to the coefficients $\mathbf{C}$ allows implementations to transforms rows only. Note that the transpose can be integrated in the inverse scan without adding complexity.

### *E. Entropy Coding*

Unlike the H.264/AVC specification that features CAVLC and CABAC [8] entropy coders, HEVC defines CABAC as the single entropy coding method. CABAC incorporates three stages: binarization of syntax elements, context modeling, and binary arithmetic coding. While the acronym and the core arithmetic coding engine remain the same as in H.264/AVC, there are a number of differences in context modeling and binarization as described below.

In the development of HEVC, a substantial amount of effort has been devoted to reduce the number of contexts. While the version 1.0 of the HM featured in excess of 700 contexts, version 8.0 has only 154. This number compares favorably to H.264/AVC, where 299 contexts are used, assuming support for frame coding in the 4:2:0 color format (progressive high profile). 237 of these 299 contexts are involved in residual signal coding whereas HEVC uses 112 of the 154 for this purpose. When comparing the reduction of 53% in residual coding with the reduction of 32% for the remaining syntax elements, it becomes clear that most effort has been put into reducing the number of contexts associated with the residual syntax. This reduction in the number of contexts contributes to lowering the amount of memory required by the entropy decoder and the cost of initializing the engine. Initialization values of the states are defined with 8 bit per context, reduced from 16 in H.264/AVC, thereby further reducing memory requirements.

One widely used method for determining contexts in H.264/AVC is to use spatial neighborhood relationships. For example, using the value above and to the left to derive a context for the current block. In HEVC such spatial dependencies have been mostly avoided such as to reduce the number of line buffers.

Substantial effort has also been devoted to enable parallel context processing, where a decoder has the ability to derive multiple context indices in parallel. These techniques apply mostly to transform coefficient coding, which becomes the entropy decoding bottleneck at high bit rates. One example is the modification of the significance map coding. In H.264/AVC, two interleaved flags are used to signal whether the current coefficient has a nonzero value (significant_coeff_flag) and whether it is the last one in coding order (last_significant_coeff_flag). This makes it impossible to derive the significant_coeff_flag and last_significant_coeff_flag contexts in parallel. HEVC breaks this dependency by explicitly signaling the horizontal and vertical offset of the last significant coefficient in the current block before parsing the significant_coeff_flags [9].

The burden of entropy decoding with context modeling grows with bit rate as more bins need to be processed. There-

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000003

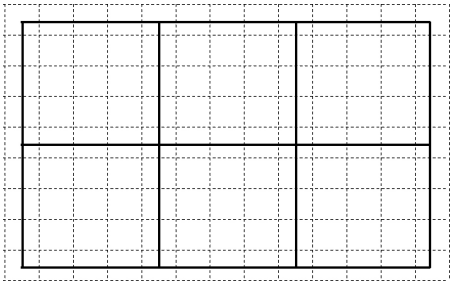


Fig. 3. Alignment of 8 × 8 blocks (dashed lines) to which the deblocking filter can be applied independently. Solid lines represent CTB boundaries.

fore, the bin strings of large syntax elements are divided into a prefix and a suffix. All prefix bins are coded in regular mode (i.e., using context modeling), whereas all suffix bins are coded in a bypass mode. The cost of decoding a bin in bypass mode is lower than in regular mode. Furthermore, the ratio of bins to bits is fixed at 1:1 for bypass mode, whereas it is generally higher for the regular mode. In H.264/AVC, motion vector differences and transform coefficient levels are binarized using this method as their values might become quite large. The boundary between prefix and suffix in H.264/AVC is quite high for the transform coefficient levels (15 bins). At the highest bit rates, level coding becomes the bottleneck as it consumes most of the bits and bins. It is thus desirable to maximize the use of bypass mode at high bit rates. Consequently, in HEVC, a new binarization scheme using Golomb-Rice codes reduces the theoretical worst case number of regular transform coefficient bins from 15 to 3 [10]. When processing large coefficients, the boundary between prefix and suffix can be lowered such that in the worst case a maximum of approximately 1.6 regular bins need to be processed per coefficient [11]. This average holds for any block of 16 transform coefficients.

### F. Deblocking Filter

The deblocking filter relies on the same principles as in H.264/AVC and shares many design aspects with it. However, it differs in ways that have a significant impact on complexity. While in H.264/AVC each edge on a 4 × 4 grid may be filtered, HEVC limits the filtering to the edges lying on an 8 × 8 grid. This immediately reduces by half the number of filter modes that need to be computed and the number of samples that may be filtered. The order in which edges are processed is also modified such as to enable parallel processing. A picture may be segmented into 8 × 8 blocks that can all be processed in parallel, as only edges internal to these blocks need to be filtered. The position of these blocks is depicted in Fig. 3. Some of these blocks overlap CTB boundaries, and slice boundaries when multiple slices are present. This feature makes it possible to filter slice boundaries in any order without affecting the reconstructed picture.

Note that vertical edges are filtered before horizontal edges. Consequently, modified samples resulting from filtering vertical edges are used in filtering horizontal edges. This allows for different parallel implementations. In one, all vertical edges are filtered in parallel, then horizontal edges are filtered in parallel. Another implementation would enable simultaneous parallel processing of vertical and horizontal edges, where the horizontal edge filtering process is delayed in a way such that the samples to be filtered have already been processed by the vertical edge filter.

However, there are also aspects of HEVC that increase the complexity of the filter, such as the addition of clipping in the strong filter mode.

### G. Sample-Adaptive Offset Filter

Compared to H.264/AVC, where only a deblocking filter is applied in the decoding loop, the current draft HEVC specification features an additional sample-adaptive offset (SAO) filter. This filter represents an additional stage, thereby increasing complexity.

The SAO filter simply adds offset values to certain sample values and it can be implemented in a fairly straightforward way, where the offset to be added to each sample may be obtained by indexing a small lookup table. The index into the lookup table may be computed according to one of the two modes being used. For one of the modes, the so-called band offset, the sample values are quantized to index the table. So all samples lying in one band of the value range are using the same offset. Edge offset, as the other mode, requires more operations since it calculates the index based on differences between the current and two neighboring samples. Although the operations are simple, SAO represents an added burden as it may require either an additional decoding pass, or an increase in line buffers. The offsets are transmitted in the bitstream and thus need to be derived by an encoder. If considering all SAO modes, the search process in the encoder can be expected to require about an order of magnitude more computation than the SAO decoding process.

### H. High-Level Parallelism

High-level parallelism refers to the ability to simultaneously process multiple regions of a single picture. Support for such parallelism may be advantageous to both encoders and decoders where multiple identical processing cores may be used in parallel. HEVC includes three concepts that enable some degree of high-level parallelism: slices, tiles, and wavefronts.

Slices follow the same concept as in H.264/AVC and allow a picture to be partitioned into groups of consecutive CTUs in raster scan order, each for transmission in a separate network adaptation layer unit that may be parsed and decoded independently, except for optional interslice filtering. Slices break prediction dependences at their boundary, which causes a loss in coding efficiency and can also create visible artifacts at these borders. The design of slices is more concerned with error resilience or maximum transmission unit size matching than a parallel coding technique, although it has undoubtedly been exploited for this purpose in the past.

Tiles can be used to split a picture horizontally and vertically into multiple rectangular regions. Like slices, tiles break prediction dependences at their boundaries. Within a picture, consecutive tiles are represented in raster scan order. The scan order of CTBs remains a raster scan, but is limited to the confines of each tile boundary. When splitting a picture

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000004

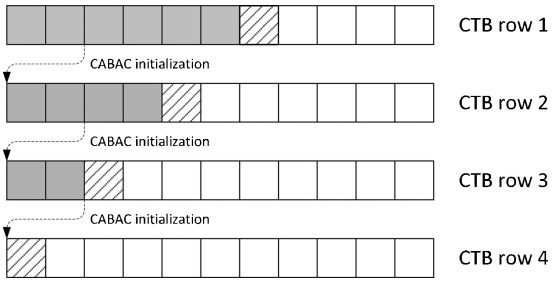



Fig. 4. Example of wavefront processing. Each CTB row can be processed in parallel. For processing the striped CTB in each row, the processing of the shaded CTBs in the row above needs to be finished.

horizontally, tiles may be used to reduce line buffer sizes in an encoder, as it operates on regions narrower than a full picture. Tiles also permit the composition of a picture from multiple rectangular sources that are encoded independently.

Wavefronts split a picture into CTU rows, where each CTU row may be processed in a different thread. Dependences between rows are maintained except for the CABAC context state, which is reinitialized at the beginning of each CTU row. To improve the compression efficiency, rather than performing a normal CABAC reinitialization, the context state is inherited from the second CTU of the previous row, permitting a simple form of 2-D adaptation. Fig. 4 illustrates this process.

To enable a decoder to exploit parallel processing of tiles and wavefronts, it must be possible to identify the position in the bitstream where each tile or slice starts. This overhead is kept to a minimum by providing a table of offsets, describing the entry point of each tile or slice. While it may seem excessive to signal every entry point without the option to omit some, in the case of tiles, their presence allows decoder designers to choose between decoding each tile individually following the per tile raster scan, or decoding CTUs in the picture raster scan order. As for wavefronts, requiring there to be as many wavefront entry points as CTU rows resolves the conflict between the optimal number of wavefronts for different encoder and decoder architectures, especially in situations where the encoder has no knowledge of the decoder.

The current draft HEVC standard does not permit the simultaneous use of tiles and wavefronts when there is more than one tile per picture. However, neither tiles nor wavefronts prohibit the use of slices.

It is interesting to examine the implementation burden of the tile and wavefront tools in the context of a single-core architecture and that of a multicore architecture. In the case of a single-core implementation for tiles, the extra overhead comes in the form of more complicated boundary condition checking, performing a CABAC reset for each tile and the need to perform the optional filtering of tile boundaries. There is also the potential for improved data-locality and cache access associated with operating on a subregion of the picture. In a wavefront implementation, additional storage is required to save the CABAC context state between CTU rows and to perform a CABAC reset at the start of each row using this saved state.

In the case of a multicore implementation, the additional overhead compared to the single-core case relates to memory-bandwidth. Since each tile is completely independent, each processing core may decode any tile with little intercore communication or synchronization. A complication is the management of performing in-loop filtering across the tile boundaries, which can either be delegated to a postprocess, or with some loose synchronization and some data exchange, may be performed on the fly. A multicore wavefront implementation will require a higher degree of communication between cores and more frequent synchronization operations than a tile-based alternative, due to the sharing of reconstructed samples and mode predictors between CTU rows. The maximum parallel improvement from a wavefront implementation is limited by the ramp-up time required for all cores to become fully utilized and a higher susceptibility to dependency related stalls between CTB rows.

All high-level parallelization tools become more useful with image sizes growing beyond HD for both encoder and decoder. At small image sizes where real-time decoding in a single-threaded manner is possible, the overhead associated with parallelization might be too high for there to be any meaningful benefit. For large image sizes it might be useful to enforce a minimum number of picture partitions to guarantee a minimum level of parallelism for the decoder. However, the current draft HEVC standard does not mandate the use of any high-level parallelism tools. As such, their use in decoders is only a benefit to architectures that can opportunistically exploit them.

### *I. Miscellaneous*

The total amount of memory required for HEVC decoding can be expected to be similar to that for H.264/AVC decoding. Most of the memory is required for the decoded picture buffer that holds multiple pictures. The size of this buffer, as defined by levels, may be larger in HEVC for a given maximum picture size. Such an increase in memory requirement is not a fundamental property of the HEVC design, but comes from the desire to harmonize the size of the buffer in picture units across all levels.

HEVC may also require more cache memory due to the larger block sizes that it supports. In H.264/AVC, macroblocks of size $16 \times 16$ define the buffer size required for storing predictions and residuals. In HEVC, intra picture prediction and transforms may be of size $32 \times 32$, and the size of the associated buffers thus quadruples.

It should also be noted that HEVC lacks coding tools specific to field coding. The absence of such tools, in particular tools that enable switching between frame and field coding within a frame (such as MBAFF in H.264/AVC), considerably simplifies the design.

### *J. Summary*

While the complexity of some key modules such as transforms, intra picture prediction, and motion compensation is likely higher in HEVC than in H.264/AVC, complexity was reduced in others such as entropy coding and deblocking. Complexity differences in motion compensation, entropy coding, and in-loop filtering are expected to be the most substantial. The implementation cost of an HEVC decoder is thus

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000005

not expected to be much higher than that of an H.264/AVC decoder, even with the addition of an in-loop filter such as SAO.

From an encoder perspective, things look different: HEVC features many more mode combinations as a result of the added flexibility from the quadtree structures and the increase of intra picture prediction modes. An encoder fully exploiting the capabilities of HEVC is thus expected to be several times more complex than an H.264/AVC encoder. This added complexity does however have a substantial benefit in the expected significant improvement in rate-distortion performance.

## III. HM Encoder

### A. Overview

The purpose of the HM encoder is mainly to provide a common reference implementation of an HEVC encoder that is useful as a test bed for evaluating technologies and for independent encoder or decoder development. Written in C++, the HM is not aimed at providing a real-world example of an HEVC encoder. The HM encoder is quite slow and using it generally involves having a large computer cluster available. Although the HM encoding speed may be less than ideal in most situations, some encoding time considerations have been made during the development of the HM. While up to 100 hours were needed to encode a single 10-second test case with version 0.7 of the reference software, the time was reduced to a more manageable 20 hours a few versions later.

Among the contributing factors to the sluggishness of the HM encoder is a heavy reliance on brute-force rate-distortion optimization. Such optimization is also applied during the quantization process.

The JCT-VC common test conditions [12] define a set of encoder configurations used in experiments. These configurations include the following:

1) All intra (AI), where all pictures are encoded using I slices.
2) Random access (RA), where picture reordering is used in a pyramidal structure with a random access picture about every 1 s. This configuration emulates what may be used in a broadcasting environment.
3) Low delay with B slices (LB), where no picture reordering is used and only the first frame is encoded using I slices. This configuration emulates what may be used in a videoconferencing environment.

While most of the features of HEVC are exercised in these configurations, it should be noted that some are not, including weighted prediction and quantization scaling matrices. High-level parallelism tools are also disabled. The results presented in the following sections thus reflect what is achievable using single-threaded encoders and decoders.

Table II shows the encoding times for the class B and C sequences (see Table I) from the JCT-VC common test conditions, each of which is 10 s long. Results are limited to the two intermediate quantization parameter (QP) values defined in the test conditions. Times are recorded in tens of seconds such as to illustrate the ratio to real-time operation.

TABLE I
Test Sequences

| Class | Sequence | Resolution | Frame Rate (Hz) |
|---|---|---|---|
| B | *Kimono* | 1920 × 1080 | 24 |
| | *ParkScene* | | 24 |
| | *Cactus* | | 50 |
| | *Basketball Drive* | | 50 |
| | *BQTerrace* | | 60 |
| C | *Basketball Drill* | 832 × 480 | 50 |
| | *BQMall* | | 60 |
| | *PartyScene* | | 50 |
| | *RaceHorses* | | 30 |

TABLE II
Encoding Time of HM 8.0

| | Time (10 s) | | | | | |
|---|---|---|---|---|---|---|
| Sequence | AI27 | AI32 | RA27 | RA32 | LB27 | LB32 |
| *Kimono* | 393 | 357 | 1283 | 1123 | 2016 | 1739 |
| *ParkScene* | 462 | 395 | 1145 | 1000 | 1743 | 1501 |
| *Cactus* | 955 | 811 | 2590 | 2257 | 3635 | 3133 |
| *Basketball Drive* | 870 | 759 | 3155 | 2707 | 4417 | 3793 |
| *BQTerrace* | 1228 | 1043 | 2936 | 2485 | 4029 | 3315 |
| *Basketball Drill* | 194 | 166 | 606 | 515 | 826 | 700 |
| *BQMall* | 229 | 202 | 642 | 562 | 900 | 779 |
| *PartyScene* | 245 | 210 | 614 | 505 | 882 | 724 |
| *RaceHorses* | 120 | 104 | 481 | 396 | 686 | 570 |

AI27 is all-intra configuration with QP set to 27. RA is random access and LB is low delay using B slices.

Even for intra-only configurations, the encoding time may exceed 1000 times real time. Encoding times were obtained on a cluster containing Xeon-based servers (E5670 clocked at 2.93 GHz) and using gcc 4.4.5.

### B. Profiling Results

The HM encoder has been profiled to determine which components are the most time consuming. Table III shows the time spent in various C++ classes in an example encoding. In the all-intra configuration a significant amount of time (about a quarter of the total) is spent in the TComTrQuant class, where rate-distortion optimized quantization (RDOQ) takes place. Transforms account for 9% on top of this. Intra picture prediction further accounts for close to 16% (TComPrediction and TComPattern classes). On the entropy coding front, the amount of time spent in core CABAC operations is small: TEncBinCABAC∗ classes, which include TEncBinCABAC and TEncBinCABACCounter, account for about 2%. Note that the TEncBinCABACCounter class is used to estimate a number of bits generated by a CABAC engine without deriving the values of those bits. More time is spent on scanning and context derivation: TEncSbac class at over 8% and getSigCtxInc in TComTrQuant class at 1.7%.

In the random access configuration it is evident that motion estimation takes up a significant portion of encoding time. Computation of the sum of absolute differences (SAD) and other distortion metrics takes place in the TComRdCost class, which accounts for about 40% of encoding time. Furthermore, motion compensation filtering happens in TComInterpolationFilter, which accounts for 20% of encoding time. It thus seems

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000006

TABLE III
ENCODING TIME DISTRIBUTION BY CLASS

| Function | Time (%) AI | RA |
|---|---|---|
| TEncSearch | 11.8 | 7.4 |
| TComTrQuant | 24.4 | 10.7 |
| TComRdCost | 9.8 | 38.8 |
| TComInterpolationFilter | 0.0 | 19.8 |
| TComYUV | 0.1 | 1.7 |
| partialButterfly∗ | 8.7 | 4.0 |
| TComDataCU | 5.8 | 2.7 |
| TEncSbac | 8.4 | 3.5 |
| TEncEntropy | 1.2 | 0.6 |
| TEncBinCABAC∗ | 2.2 | 0.9 |
| TComPrediction | 10.0 | 1.1 |
| TComPattern | 6.6 | 0.4 |
| memcpy/memset | 11.0 | 7.1 |

Classes consuming 1% or more of time in the HM 8.0 encoder, contributing to more than 95% of total time. 1080p sequence: *BasketballDrive*, QP = 27.

that a significant amount of time may be spent on fractional-pel search refinement in the motion estimation process. The fractional-pel search in the HM is not optimized for speed. It consists of testing nine displacements on a half-pel grid and another nine displacements on a quarter-pel grid. Furthermore, the use of the merge mode with a large number of candidates may also contribute to the time usage of inter polation filtering, as the 2-D inter polation process has to be repeated for each merge candidate. The amount of time spent on inter polation filtering may be reduced by precomputing filtered reference pictures for each of the 16 fractional displacements on a quarter-pel grid. This would however contribute to a significant increase in memory requirements and may not be suited for all architectures.

As in the all-intra configuration, most of the time spent in TComTrQuant may be attributed to RDOQ as the actual transforms are tallied separately (partialButterfly∗ at about 4%).

It is also interesting to note that a significant amount of time is spent on setting and copying memory areas (memcpy and memset system functions). In the all-intra case, around one-third of the time spent in these system functions (3.6% of total time) comes from initializing arrays to 0 in the RDOQ function. A similar trend is observed in the random access case.

Encoding functions related to SAO are absent from Table III, as the amount of time spent inside them are below 1% of the total encoding time.

### C. Alternative Tradeoffs

There are a number of fast encoding methods available in the HM encoder. Table IV shows an overview of the different configuration options.

FastSearch, FEN, FDM, and FastTS are already enabled in the HEVC common coding conditions. Table V shows encoding time with ASR, ECU, CFM, and ESD enabled in addition to that. All of these additional modes only influence the inter picture prediction mode decision. Thus only random access and low delay configurations are shown. All of these methods reduce the number of searched partitions based on the skip mode performance. In the rate-distortion decision this mode is more likely to be chosen for higher QP values. This explains why we see a greater speedup with higher QPs. In the QP range illustrated, the encoding time can be reduced by a factor between two and three. For QP equal to 37 a factor up to six can be observed in single cases. The Bjøntegaard Delta (BD) [19] rate values in Table V have been calculated using the four QP points of the common coding conditions. As shown, the coding efficiency penalty is below 3% BD rate.

TABLE IV
OVERVIEW OF FAST ENCODING MODES

| Encoder parameter | Description |
|---|---|
| FastSearch | TZ search similar to the corresponding search mode in the reference software for H.264/AVC scalable video coding (JSVM) [13]. |
| ASR | Automatic search range adaptation based on picture order count difference. |
| FEN | Use subsampled SAD for blocks with more than eight rows for integer motion estimation and use one iteration instead of four for bipredictive search. |
| ECU | No further split if skip mode is used [14]. |
| FDM | Stop merge search if skip mode (getQtRootCbf()) is chosen in xCheckRDCostMerge2N × 2N() [15]. |
| CFM | Terminate encoding decision if partitioning (PU) has coded block flag equal to zero [16]. |
| ESD | Check inter $2N \times 2N$ first and terminate if motion vector difference is zero and coded block flag equal to zero [17]. |
| FastTS | For luma transform skip mode is only checked if the CU size is 8 × 8, the partition mode is IntraNxN, and the TU size is 4 × 4. For 4 × 4 chroma TU transform skip mode is not checked, when the 8 × 8 luma TU is not split into four 4 × 4 TUs or when the 8 × 8 luma TU is split into four 4 × 4 TUs but none of these TUs uses the transform skip mode. [18] |

TABLE V
ENCODING TIME AND PERFORMANCE OF HM 8.0 WITH FAST OPTION

| Sequence | Time (10 s) RA27 | RA32 | BD rate | Time (10 s) LB27 | LB32 | BD rate |
|---|---|---|---|---|---|---|
| *Kimono* | 628 | 416 | 1.6% | 1251 | 845 | 1.2% |
| *ParkScene* | 478 | 306 | 2.2% | 953 | 605 | 1.6% |
| *Cactus* | 1237 | 860 | 2.8% | 2115 | 1461 | 2.1% |
| *Basketball Drive* | 1631 | 1150 | 2.2% | 2749 | 1882 | 1.1% |
| *BQTerrace* | 1180 | 617 | 2.3% | 1955 | 980 | 1.2% |
| *Basketball Drill* | 367 | 253 | 2.8% | 581 | 402 | 0.9% |
| *BQMall* | 334 | 235 | 2.3% | 553 | 389 | 1.5% |
| *PartyScene* | 371 | 243 | 1.8% | 640 | 431 | 1.2% |
| *RaceHorses* | 321 | 225 | 2.5% | 535 | 381 | 1.0% |
| Mean speedup | 2.0 × | 2.6× | | 1.6× | 2.1× | |

RA27 is random access configuration with QP set to 27, LB is low delay using B slices. BD rate is the luma difference in BD rate over four QP points.

### D. Discussion

The HM software does not feature low-level optimizations such as assembly written functions to compute block differences or inter polate a block. Doing so would permit speeding up the encoder by about two to three times. This, however, is a far cry from the factor of 5000 between the HM encoding time

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000007

TABLE VI
DECODING TIME OF HM 8.0

| | Time (s) | | | | | |
|---|---|---|---|---|---|---|
| Sequence | AI27 | AI32 | RA27 | RA32 | LB27 | LB32 |
| *Kimono* | 34.8 | 31.2 | 20.8 | 18.5 | 22.5 | 18.4 |
| *ParkScene* | 49.9 | 41.1 | 20.9 | 18.4 | 22.1 | 18.0 |
| *Cactus* | 96.8 | 83.4 | 39.8 | 34.8 | 40.4 | 32.7 |
| *Basketball Drive* | 83.9 | 73.7 | 45.0 | 39.0 | 50.7 | 41.6 |
| *BQTerrace* | 126.6 | 107.9 | 50.1 | 40.3 | 54.8 | 40.4 |
| *Basketball Drill* | 22.5 | 18.6 | 9.8 | 8.3 | 10.2 | 8.2 |
| *BQMall* | 25.5 | 22.5 | 10.7 | 9.2 | 11.2 | 9.5 |
| *PartyScene* | 28.8 | 24.7 | 10.9 | 8.9 | 12.5 | 9.5 |
| *RaceHorses* | 12.5 | 11.2 | 7.4 | 6.0 | 8.2 | 6.5 |

AI27 is all-intra configuration with QP set to 27. RA is random access and LB is low delay using B slices.

TABLE VII
DECODING TIME DISTRIBUTION BY CLASS

| | Time (%) | |
|---|---|---|
| Function | AI | RA |
| TComTrQuant | 8.7 | 4.2 |
| TComInterpolationFilter | 0.0 | 24.8 |
| TComYUV | 0.5 | 8.2 |
| partialButterfly∗ | 15.9 | 7.6 |
| TComDataCU | 7.5 | 7.1 |
| TDecSbac | 6.2 | 2.8 |
| TDecEntropy | 1.4 | 1.0 |
| TDecBinCABAC | 5.3 | 2.3 |
| TDecCU | 7.2 | 2.6 |
| TComPrediction | 5.1 | 2.3 |
| TComPattern | 9.4 | 2.6 |
| TComSampleAdaptiveOffset | 3.8 | 2.4 |
| TComLoopFilter | 12.9 | 12.4 |
| memcpy/memset | 6.2 | 10.1 |

Classes consuming 1% or more of time in the HM 8.0 decoder, contributing to more than 90% of total time. 1080p sequence: *BasketballDrive*, QP = 27.

and real-time encoding. Much more significant work will be required and it is expected that developing encoder heuristics for HEVC will become a major topic of research in the coming years. Multithreading may play a significant role in the design of fast HEVC encoders.

It should be noted that the HM encoder already features some tricks to speed it up. For example, mode search for intra picture prediction prunes the search space based on a simple distortion metric, and bit counting for CABAC uses table-based estimations rather than trial encodes.

Compared to the H.264/AVC JM reference software [20], the HM provides far fewer parameters to configure the encoder. It is not possible to switch single encoding modes like partition shapes, sizes, or intra picture prediction modes, or to completely turn off rate-distortion optimization. The HM always uses 16-bit data formats to store picture samples, even when operating in 8-bit coding mode. This requires more memory and more memory bandwidth compared to the JM, which can be configured to use only eight bits per sample. Although the fast coding modes of the JM and HM cannot be directly compared, in some tests with similar configurations the JM encoder was running at least four times faster than the HM.

## IV. HM DECODER

Similar to the HM encoder, the HM decoder is an example of implementation aimed at correctness, completeness, and readability. It runs in a single thread and no parallelization techniques are used.

Table VI shows the decoding times for class B and C sequences using the common coding conditions. The calculation of MD5 checksums and writing of decoded pictures to a file have been disabled to allow a pure decoding time measurement. Using 10-s sequences, times up to 126.6 s have been measured (*BQTerrace*, all-intra, QP 27), which would require a speedup of nearly a factor of 13 for real-time decoding in this particular case. If the QP is further lowered to 22, this factor increases to 17.

The decoding times thus depend strongly on the selected QP value. At higher bit rates (i.e., lower QP) more coefficients are coded, which requires more time for the CABAC parsing process, but also smaller block sizes will be selected, which can put an additional burden on the decoder. The difference between random access and low delay is rather small, while all-intra decoding requires up to twice the time of the random access case at the same QP.

To further illustrate which decoder functions are time consuming, Table VII shows profiling results of the HM decoder by C++ class. In the all-intra configuration decoding time is dominated by partialBufferfly∗, TComDataCU, TComPattern, and TComLoopFilter. partialBufferfly∗ represents the inverse transform functions and accounts for about 15% of decoding time. While this amount of time is significant, it is also expected that it could be significantly reduced by implementing partial transform techniques. Such partial transform techniques would be particularly effective on the larger transforms where high-frequency coefficients are most likely zero. The TComPattern class deals with reference sample generation for intra picture prediction. The high amount of time spent in this class is somewhat surprising and may hint at a poor implementation that relies on copying reference samples multiple times. The TComLoopFilter class implements the deblocking filter. The TComDataCU class deals with managing most data elements within a CU. Functions for deriving the addresses of neighboring blocks are among the most time consuming in TComDataCU. This is partly due to the number of conditions that require checking (slice, CTB, entropy slice, and tile boundaries).

In the random access configuration, decoding time is dominated by TComInterpolationFilter and TComLoopFilter. TComInterpolationFilter comprises the functions for inter polation filtering in the motion compensation process.

To achieve real-time performance it is expected that all components of the HM decoder would require improvements. The next section considers a decoder where such improvements were made.

## V. OPTIMIZED SOFTWARE DECODER

### A. Introduction

This section discusses the performance of an optimized implementation of an HEVC software decoder. While this

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000008

TABLE VIII
DECODING TIME (ARM): RANDOM ACCESS CONFIGURATION

| Sequence | Frame rate (Hz) | Bit rate (Mbit/s) | Time (s) | Bit rate (Mbit/s) | Time (s) |
|---|---|---|---|---|---|
| *BasketballDrill* | 50 | 1.66 | 7.5 | 0.81 | 6.2 |
| *BQMall* | 60 | 1.70 | 8.4 | 0.85 | 7.2 |
| *PartyScene* | 50 | 3.10 | 9.4 | 1.46 | 7.6 |
| *RaceHorses* | 30 | 2.03 | 6.8 | 0.94 | 5.5 |

TABLE IX
DECODING TIME DISTRIBUTION (ARM): RANDOM ACCESS CONFIGURATION

| Sequence | MC (%) | QT (%) | PR (%) | ED (%) | DF (%) | SF (%) |
|---|---|---|---|---|---|---|
| *BasketballDrill* QP = 27 | 36 | 5 | 7 | 26 | 19 | 5 |
| *BasketballDrill* QP = 32 | 44 | 4 | 6 | 19 | 20 | 3 |
| *BQMall* QP = 27 | 46 | 4 | 5 | 23 | 16 | 3 |
| *BQMall* QP = 32 | 53 | 3 | 4 | 16 | 17 | 2 |
| *PartyScene* QP = 27 | 40 | 5 | 5 | 31 | 13 | 3 |
| *PartyScene* QP = 32 | 49 | 3 | 5 | 22 | 15 | 3 |
| *RaceHorses* QP = 27 | 35 | 4 | 7 | 30 | 16 | 5 |
| *RaceHorses* QP = 32 | 43 | 4 | 6 | 22 | 19 | 4 |
| Average | 43 | 4 | 6 | 24 | 17 | 4 |

MC: motion compensation. QT: inverse quantization and transform. PR: intra picture prediction and picture construction process. ED: entropy decoding. DF: deblocking filter. SF: sample-adaptive offset filter.

decoder is significantly faster than the HM decoder, no claims are made as to its optimality and faster decoders might be designed. It is an evolution of the decoder previously described in [7] and [21], which was written in C from scratch.

Code was optimized for two different instruction set architectures (ISA): x86 and ARM. In both cases SIMD instructions are heavily used. Intrinsics are inserted in C code to make use of these instructions. Assembly code is also used on ARM. Extensions up to SSE4.1 are used on x86, and NEON on ARM.

### B. Profiling Results (ARM)

Class C sequences ($832 \times 480$ luma samples) are used for profiling on ARM. Experiments are run on a tablet featuring a 1 GHz Cortex-A9 dual-core processor. Decoded video is displayed live using OpenGL ES 2.0 in parallel with decoding. A separate thread is used for the decoding loop. This decoding loop is not split into additional threads. To account for the variability in decoding time associated with each frame, the frame buffer used for decoding and display has 16 entries to enable smooth playback.

Table VIII shows the decoding time for the random access configuration using QP values 27 and 32. All test cases are decoded in real time. The data suggest that decoding wide 480p video at 2 Mb/s and 30 f/s is easily achievable on production tablets available at the time of writing. Although not tested, this is also likely the case on lower power devices, such as smartphones. However, the impact on power consumption was not measured.

Table IX and Fig. 5 show the distribution of decoding time across various modules. Motion compensation is the most time consuming and takes up close to half the decoding time. The loop filters (deblocking and SAO) contribute to about a fifth of decoding time and entropy decoding to about one quarter. Inverse quantization and transform contribute to only about 4%. It should be noted that this percentage is significantly lower than reported in Section IV. Two contributing factors are the exploitation of partial inverse transforms and aggressive code optimization. The introduction of large transforms does thus not appear to significantly affect software decoding times.

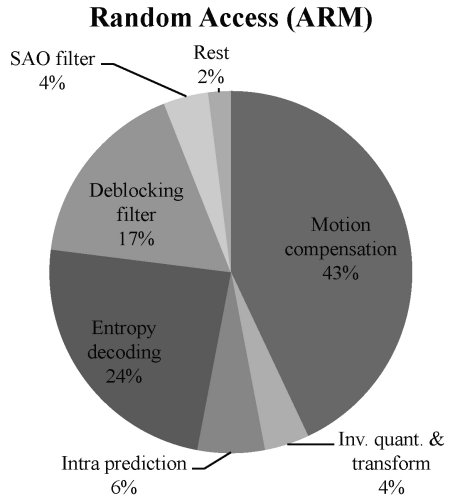



Fig. 5. Average decoding time distribution (ARM): random access configuration.

TABLE X
DECODING TIME (ARM): ALL-INTRA CONFIGURATION

| Sequence | Frame rate (Hz) | Bit rate (Mbit/s) | Time (s) | Bit rate (Mbit/s) | Time (s) |
|---|---|---|---|---|---|
| *BasketballDrill* | 50 | 11.31 | 19.1 | 6.07 | 14.7 |
| *BQMall* | 60 | 13.97 | 23.3 | 8.22 | 19.1 |
| *PartyScene* | 50 | 27.23 | 30.4 | 16.23 | 23.2 |
| *RaceHorses* | 30 | 8.99 | 12.9 | 5.11 | 10.5 |

Table X shows decoding times for all-intra configurations. Unlike for random access configurations, real-time decoding is not achieved in any tested case. One reason is the amount of processing required to perform entropy decoding. In I slices, most of the bits represent transform coefficients and parsing is thus one of the bottlenecks. This is confirmed in profiling results in Table XI and Fig. 6, where parsing of transform coefficients represents 36% of decoding time on average. In the worst case among the tested cases, it represents as much as 48%. This is however not the only bottleneck, since after the deduction of decoding time associated with it, decoding times remain larger than 10 s in several cases. The next most time-consuming module is intra picture prediction and the picture construction process which accounts for a fifth of decoding time on average.

### C. Profiling Results (x86)

Class B sequences were used for profiling on an x86 computer, where bit rates go up to about 7 Mbit/s in random

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000009

TABLE XI
DECODING TIME DISTRIBUTION (ARM): ALL-INTRA CONFIGURATION

| Sequence | CP (%) | ED (%) | DF (%) | QT (%) | PR (%) | SF (%) |
|---|---|---|---|---|---|---|
| *BasketballDrill* QP = 27 | 32 | 17 | 14 | 8 | 22 | 6 |
| *BasketballDrill* QP = 32 | 26 | 16 | 18 | 8 | 22 | 8 |
| *BQMall* QP = 27 | 35 | 16 | 13 | 9 | 20 | 6 |
| *BQMall* QP = 32 | 28 | 16 | 16 | 10 | 22 | 6 |
| *PartyScene* QP = 27 | 48 | 15 | 8 | 6 | 17 | 4 |
| *PartyScene* QP = 32 | 39 | 17 | 11 | 7 | 19 | 5 |
| *RaceHorses* QP = 27 | 43 | 12 | 11 | 10 | 16 | 5 |
| *RaceHorses* QP = 32 | 34 | 14 | 14 | 11 | 19 | 6 |
| Average | 36 | 15 | 13 | 9 | 20 | 6 |

CP: transform coefficient parsing. ED: entropy decoding, excluding transform coefficient parsing. DF: deblocking filter. QT: inverse quantization and transform. PR: intra picture prediction and picture construction
process. SF: sample-adaptive offset filter.

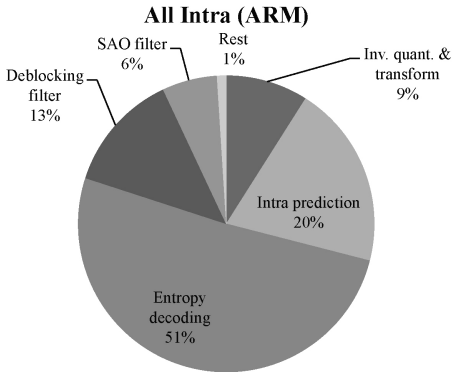



Fig. 6. Average decoding time distribution (ARM): all-intra configuration.

TABLE XII
DECODING TIME (X86): RANDOM ACCESS CONFIGURATION

| Sequence | Frame rate (Hz) | Bit rate (Mbit/s) | Time (s) | Bit rate (Mbit/s) | Time (s) |
|---|---|---|---|---|---|
| *BasketballDrive* | 50 | 6.01 | 4.9 | 2.80 | 3.8 |
| *BQTerrace* | 60 | 7.31 | 5.6 | 2.26 | 3.8 |
| *Cactus* | 50 | 5.72 | 4.0 | 2.67 | 3.0 |
| *Kimono* | 24 | 2.18 | 2.0 | 1.07 | 1.7 |
| *ParkScene* | 24 | 3.33 | 2.4 | 1.54 | 1.8 |

Decoding as a standalone process.

access configurations. Unlike in the ARM case, measurements are made using a command-line decoder. The gcc compiler version 4.7.1 was used. For random access configurations, real-time decoding is achieved on a 2012-model laptop using a single core and single thread of an Intel Core i7-3720QM processor clocked at 2.6 GHz (turbo up to 3.6 GHz), as shown in Table XII. This real-time performance is also achieved by a wide margin at 60 fps (*BQTerrace* sequence): less than 6 s are needed to decode a 10-s sequence. When comparing these results with those of Table VI, a speed-up of about 10 × is observed. However, different environments were used to obtain the results. When compensating for this difference, the speed-up is about 6 × .

Table XIII and Fig. 7 further illustrate the distribution of decoding time across different modules. As observed previously, motion compensation remains the most time-consuming module, accounting for half of the time on average.

TABLE XIII
DECODING TIME DISTRIBUTION (X86): RANDOM ACCESS CONFIGURATION

| Sequence | MC (%) | QT (%) | PR (%) | ED (%) | DF (%) | SF (%) |
|---|---|---|---|---|---|---|
| *BasketballDrive* QP = 27 | 42 | 6 | 5 | 24 | 14 | 6 |
| *BasketballDrive* QP = 32 | 52 | 5 | 4 | 17 | 15 | 3 |
| *BQTerrace* QP = 27 | 46 | 3 | 3 | 27 | 13 | 6 |
| *BQTerrace* QP = 32 | 61 | 2 | 2 | 15 | 13 | 4 |
| *Cactus* QP = 27 | 37 | 5 | 5 | 28 | 16 | 7 |
| *Cactus* QP = 32 | 45 | 5 | 4 | 20 | 17 | 5 |
| *Kimono* QP = 27 | 51 | 5 | 3 | 19 | 14 | 4 |
| *Kimono* QP = 32 | 58 | 4 | 2 | 14 | 15 | 2 |
| *ParkScene* QP = 27 | 46 | 3 | 4 | 26 | 13 | 4 |
| *ParkScene* QP = 32 | 55 | 2 | 4 | 18 | 14 | 3 |
| Average | 49 | 4 | 4 | 21 | 14 | 4 |

MC: motion compensation. QT: inverse quantization and transform. PR: intra picture prediction and picture construction process. ED: entropy decoding. DF: deblocking filter. SF: sample-adaptive offset filter.

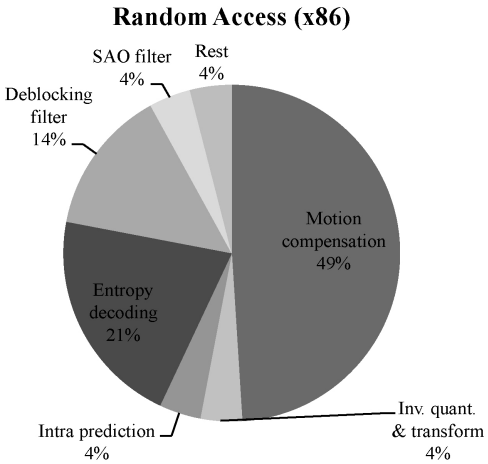



Fig. 7. Average decoding time distribution (x86): random access configuration.

For all-intra configurations, real-time decoding is not always achieved on a single core, as shown in Table XIV. In the worst case (*BQTerrace* QP = 27) 19.1 s are required to decode 10 s of video. Profiling of this case indicates that 60% of time is spent on entropy decoding, 13% on intra picture prediction and residual addition, 9% on deblocking, 7% on inverse quantization and transform, and 9% on SAO filtering. Entropy decoding thus represents a significant bottleneck, which is not surprising given that the bit rate is about 80 Mbit/s.

When considering the case studied for the HM decoder (*BasketballDrive* QP = 27, Table VII), the time proportions remain roughly similar.

### D. Discussion

The profiling results obtained on ARM and x86 are quite similar even though obtained with different sets of video sequences. In both cases the decoding time proportion associated with each module is similar, as summarized below. Motion compensation takes up about half of the decoding time. In-loop filters (deblocking and SAO) take about one fifth and

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000010

TABLE XIV
DECODING TIME (X86): ALL-INTRA

| Sequence | Frame rate [Hz] | Bit rate (Mbit/s) | Time (s) | Bit rate (Mbit/s) | Time (s) |
|---|---|---|---|---|---|
| *BasketballDrive* | 50 | 29.1 | 10.2 | 15.2 | 7.2 |
| *BQTerrace* | 60 | 79.3 | 19.1 | 40.3 | 12.8 |
| *Cactus* | 50 | 48.7 | 13.6 | 26.3 | 9.6 |
| *Kimono* | 24 | 12.1 | 3.5 | 6.8 | 2.6 |
| *ParkScene* | 24 | 28.6 | 7.3 | 14.8 | 4.8 |

Decoding as a standalone process.

entropy decoding about another fifth. The remaining 10% goes to inverse transform, intra picture prediction, and so on.

These proportions are also similar to those reported on the basis of other independent implementations [22], [23]. Some differences may be observed for motion compensation and entropy coding. The time percentages reported herein for motion compensation tend to be higher (about 50% compared to 35%–40%), and those for entropy decoding lower (about 20% compared to 25%–30%).

One limiting factor in the motion compensation process is memory bandwidth and cache misses. In the tested decoder, the memory layout of reference frames uses a raster scan with interleaved chroma components, and limited explicit prefetching is used. The interleaving of the chroma components reduces the amount of memory fetch operations and also guarantees a higher minimum width for inter polation processing. Using a different memory layout or better prefetching may help reduce the decoding time associated with motion compensation. Another limiting factor in the motion compensation process is the number of multiply–accumulate operations. In the tested decoder, generic 8-tap filter functions are used for luma, where filter coefficients are stored in a table. Tuned implementations, for example taking advantage of the knowledge of certain filter coefficients equal to 0 or 1, may also reduce the decoding time associated with motion compensation.

Entropy decoding becomes a bottleneck at high bit rates, and in I slices in particular.

Studies on H.264/AVC decoding complexity have mostly focused on the Baseline profile [24], [25]. The percentage time spent in motion compensation in the Baseline profile tends to be lower than in HEVC for the simple reason that the Baseline profile does not support B slices. On the other hand, the percentage time in the deblocking filter tended to be high in H.264/AVC. This percentage tends to be much lower in HEVC since the number of filtered edges is lower.

## VI. CONCLUSION

In conclusion, the complexity cost of HEVC to achieve superior compression performance is not obvious. While certain aspects of the design require more processing than in H.264/AVC, other aspects have been simplified.

Real-time software decoding of HEVC bitstreams is very feasible on current generation devices: 1080p60 decoding on laptops or desktops, and 480p30 decoding on mobile devices (both within reasonable bit rate ranges). Such performance is achievable without having to rely on multiple cores in the decoding process. This is important as it can provide a software path for rapid and wide adoption of HEVC.

On the encoder side, substantial additional work is required to make a real-time encoder that delivers compression efficiency comparable to the HM encoder. This is expected to be an active area of research in years to come.

## APPENDIX
## DOWNLOADABLE RESOURCES RELATED TO THIS PAPER

All the JCT-VC documents can be found in the JCT-VC document management system at http://phenix.int-evry.fr/jct/. All cited VCEG and JVT documents are also publicly available and can be downloaded at http://wftp3.itu.int/av-arch in the video-site and jvt-site folders, respectively.

## REFERENCES


[1] B. Bross, W.-J. Han, J.-R. Ohm, G. J. Sullivan, and T. Wiegand, *High Efficiency Video Coding (HEVC) Text Specification Draft 8*, document JCTVC-J1003 of JCT-VC, Stockholm, Sweden, Jul. 2012.

[2] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1648–1667, Dec. 2012.

[3] ITU-T and ISO/IEC JTC 1, *Advanced Video Coding for Generic Audiovisual Services*, ITU-T Rec. H.264 and ISO/IEC 14496-10 (MPEG-4 AVC), 2011.

[4] P. Chou, T. Lookabaugh, and R. Gray, "Optimal pruning with applications to tree-structured source coding and modeling," *IEEE Trans. Inform. Theory*, vol. 35, no. 2, pp. 299–315, Mar. 1989.

[5] G. J. Sullivan and R. L. Baker, "Efficient quadtree coding of images and video," *IEEE Trans. Image Process.*, vol. 3, no. 3, pp. 327–331, Jan. 1994.

[6] B. Bross, S. Oudin, P. Helle, D. Marpe, and T. Wiegand, "Block merging for quadtree-based partitioning in HEVC," in *Proc. 35th SPIE Appl. Digit. Image Process.*, vol. 8499. Aug. 2012, paper 8499-0R.

[7] F. Bossen, *On Software Complexity*, document JCTVC-G757, JCT-VC, Geneva, Switzerland, Nov. 2011.

[8] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 620–636, Jul. 2003.

[9] J. Sole, R. Joshi, and M. Karczewicz, *CE11: Parallel Context Processing for the Significance Map in High Coding Eefficiency*, document JCTVC-E338, JCT-VC, Geneva, Switzerland, Mar. 2011.

[10] T. Nguyen, D. Marpe, H. Schwarz, and T. Wiegand, "Reduced-complexity entropy coding of transform coefficient levels using truncated Golomb-Rice codes in video compression," in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2011, pp. 753–756.

[11] J. Chen, W.-J. Chien, R. Joshi, J. Sole, and M. Karczewicz, *Non-CE1: Throughput Improvement on CABAC Coefficients Level Coding*, document JCTVC-H0554, JCT-VC, San Jose, CA, Feb. 2012.

[12] F. Bossen, *Common Test Conditions and Software Reference Configurations*, document JCTVC-H1100, JCT-VC, San Jose, CA, Feb. 2012.

[13] *ITU-T and ISO/IEC JTC 1, Reference Software for Scalable Video Coding*, ITU-T Rec. H.264.2 (reference software for H.264 advanced video coding) and ISO/IEC 14496-5:Amd 19 (reference software for scalable video coding), 2009–2012.

[14] K. Choi, S.-H. Park, and E. S. Jang, *Coding Tree Pruning Based CU Early Termination*, document JCTVC-F092, JCT-VC, Torino, Italy, Jul. 2011.

[15] G. Laroche, T. Poirier, and P. Onno, *Encoder Speed-up for the Motion Vector Predictor Cost Estimation*, document JCTVC-H0178, JCT-VC, San Jose, CA, Feb. 2012.

[16] R. H. Gweon and Y.-L. Lee, *Early Termination of CU Encoding to Reduce HEVC Complexity*, document JCTVC-F045, JCT-VC, Torino, Italy, Jul. 2011.


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000011

[17] J. Yang, J. Kim, K. Won, H. Lee, and B. Jeon, *Early SKIP Detection for HEVC*, document JCTVC-G543, JCT-VC, Geneva, Switzerland, Nov. 2011.

[18] P. Onno and E. Francois, *Combination of J0171 and J0389 for the Nonnormative Encoder Selection of the Intra Transform Skip*, document JCTVC-J0572, JCT-VC, Stockholm, Sweden, Jul. 2012.

[19] G. Bjøntegaard, *Calculation of Average PSNR Differences between RD Curves*, document VCEG-M33, ITU-T Q6/16, Austin, TX, Apr. 2001.

[20] JVT. (2012). *H.264/AVC Reference Software* [Online]. Available: http://iphome.hhi.de/suehring/tml/

[21] F. Bossen, *On Software Complexity: Decoding 720p Content on a Tablet*, document JCTVC-J0128, JCT-VC, Stockholm, Sweden, Jul. 2012.

[22] K. McCann, J.-Y. Choi, K. Pachauri, K. P. Choi, C. Kim, Y. Park, Y. J. Kwak, S. Jun, M. Choi, H. Yang, and J. Park, *HEVC Software Player Demonstration on Mobile Devices*, document JCTVC-G988, JCT-VC, Geneva, Switzerland, Nov. 2011.

[23] K. Veera, R. Ganguly, B. Zhou, N. Kamath, S. Chowdary, J. Du, I.-S. Chong, and M. Coban, *A Real-Time ARM HEVC Decoder Implementation*, document JCTVC-H0693, JCT-VC, San Jose, CA, Feb. 2012.

[24] V. Iverson and J. McVeigh, "Real-time H.264-AVC codec on Intel architectures," in *Proc. IEEE Int. Conf. Image Process.*, vol. 2. Oct. 2004, pp. 757–760.

[25] M. Horowitz, A. Joch, F. Kossentini, S. Member, and A. Hallapuro, "H.264/AVC baseline profile decoder complexity analysis," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 704–716, Jul. 2003.

**Frank Bossen** (M'04) received the M.Sc. degree in computer science and the Ph.D. degree in communication systems from the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, in 1996 and 1999, respectively.

Since 1995, he has been active in video coding standardization and has held a number of positions with IBM, Yorktown Heights, NY; Sony, Tokyo, Japan; GE, Ecublens, Switzerland; and NTT DOCOMO, San Jose, CA. He is currently a Research Fellow with DOCOMO Innovations, Palo Alto, CA.

**Benjamin Bross** (S'11) received the Dipl.-Ing. degree in electrical engineering from Rheinisch-Westfaelische Technische Hochschule University, Aachen, Germany, in 2008.

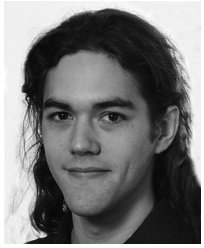
During his studies, he worked on 3-D image registration in medical imaging and on decoder side motion vector derivation in H.264/AVC. He is currently with the Image Processing Group, Fraunhofer Institute for Telecommunications–Heinrich Hertz Institute, Berlin, Germany. His current research interests include motion estimation or prediction, residual coding, and contributions to the evolving high efficiency video coding (HEVC) standard.

Mr. Bross has been coordinating core experiments for the development of HEVC and has been the Co-Chair of the editing *ad hoc* group since 2010. He was appointed Editor of the HEVC standard in July 2011.

**Karsten Sühring** received the Dipl.-Inf. degree in applied computer science from the University of Applied Sciences, Berlin, Germany, in 2001.

He is currently with the Image and Video Coding Group, Fraunhofer Institute for Telecommunication–Heinrich Hertz Institute, Berlin, where he has been engaged in video coding standardization activities and has been an editor of the reference software of H.264/MPEG-4 AVC. His current research interests include coding and transmission of video and audio content, as well as software design and optimization.

Mr. Sühring has been the Vice Chair of the JCT-VC *ad hoc* group on software development since June 2011 and is one of the coordinators for the HM reference software for HEVC.



**David Flynn** received the B.Eng. degree in computer systems engineering from the University of Warwick, Warwick, U.K., in 2005.

He was with the Research and Development Department, British Broadcasting Corporation, London, U.K. He is currently with Research In Motion, Ltd., Waterloo, ON, Canada. He has been engaged in activities related to video compression, including the standardization of high-efficiency video coding and Dirac/VC-2 video codecs.

Mr. Flynn is a member of the Institute of Engineering and Technology.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1020, Page 000012