H.265/HEVC decoder optimization

Submitted by

Antonios Kalkanof

Advisor

Prof. Ioannis Katsavounidis

University of Thessaly
Volos, Greece
February 2014

Acknowledgements

I am grateful to my family and friends for their support over the years.

I sincerely thank my advisor, professor Ioannis Katsavounidis, whose advice and wisdom always inspired and influenced me.

Contents

Introduction

In 2013, JCT-VC released the H.265/High Efficiency Video Coding (HEVC) standard, the highly anticipated successor of H.264/MPEG4-AVC. Designed to evolve the video compression industry, HEVC supports video resolutions of up to 8k x 4k and further intends to reduce the average bit rate over H.264 (the industry's standard for the past ten years) by an additional 50% at the same video quality.

In order for the standard to be tested and evaluated by users, reference source code, known as HEVC test Model (HM), is provided online for both encoder and decoder. Both implementations are quite slow since they are coded in C++ and are not targeted for performance but merely for research and code-readability. Reference test data, such as encoded bitstreams, is also available online, covering various profiles, bitrates and resolutions.

In this thesis we optimized the HM 10.0 reference decoder on an x86 Windows 7 platform. We applied techniques such as data parallelization, using SSE2 SIMD instructions, as well as C++ specific optimizations and achieved a total speedup of up to 2.15x. Results showed that 17 out of the 20 reference bitstreams we tested with video resolution of 832x480 at various bitrates, could be decoded in real time using our optimized HM decoder.

We begin with a brief introduction to Video-encoding basics, after which we present the new video codec standard, HEVC. We continue by explaining in detail the optimizations we applied on the HM decoder and conclude our work with our results and future work.

Chapter 1. Video Encoding

Raw video

Digital video (also referred as raw video) comprises of a series of orthogonal bitmap digital images displayed in rapid succession at a constant rate. These images are called *frames* and the rate at which they are displayed is called *fps* (frames per second). Each frame consists of *pixels*.

Codecs use the YUV format (YCbCr 4:2:0) which takes into account human perception and represents data with luminance(Y) and chrominance components (Cb Cr). In specific for every 4 luminance (Y) pixels, 2 chroma (1 Cb and 1 Cr) pixels correspond, as shown below.

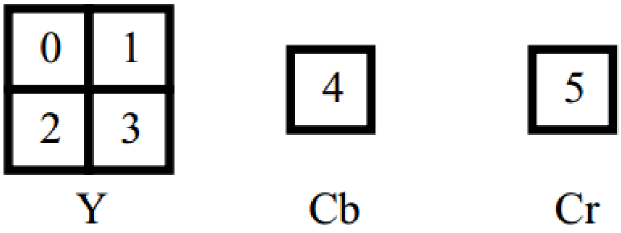


Fig. 1 YCbCr 4:2:0 format. 4 luma (Y) and 2 chroma (Cb,Cr) pixels

A pixel has a fixed value, typically 8 bits. The more bits the more subtle variations of colors can be reproduced. If it has a width of W pixels and a height of H pixels we say that the frame size is WxH.

For example, an hour long video with a frame size of 720x480 and a frame-rate of 30fps (NTSC) (or equivalently, 720x576 and 25fps for PAL systems) has 720x480 luma pixels and (360 x 240)*2 chroma pixels per frame, in total 720x480*1.5*30*3600 = 56GBytes.

In specific,

- pixels per frame = 720 * 480 * 1.5 = 518,400
- bits per frame = 518,400 * 8 = 4,147,200 = 4.14 Mbits
- bit rate = 4,147,200 * 30 = 124.4 Mbits/sec
- video size = 124.4 * 3600 = **56 GBytes!**

Today's internet resources and storage capacities cannot handle this excessive amount of data, unless it can be reduced by orders of magnitude. This is the work of a software or hardware program, called video codec.

Video Codecs

Introduction

A video codec first *encodes* the original source, achieving a high compression ratio, for example 50:1. After the encoded video is transmitted/stored, the codec *decodes* it, thus producing a lossy representation of the original. It is specifically, the decoding stage that is standardized by the different video standards that have emerged over the years (MPEG-1, MPEG-2, H.264, HEVC). Finally the reproduced result can be rendered. Video codecs in general sacrifice quality for quantity.

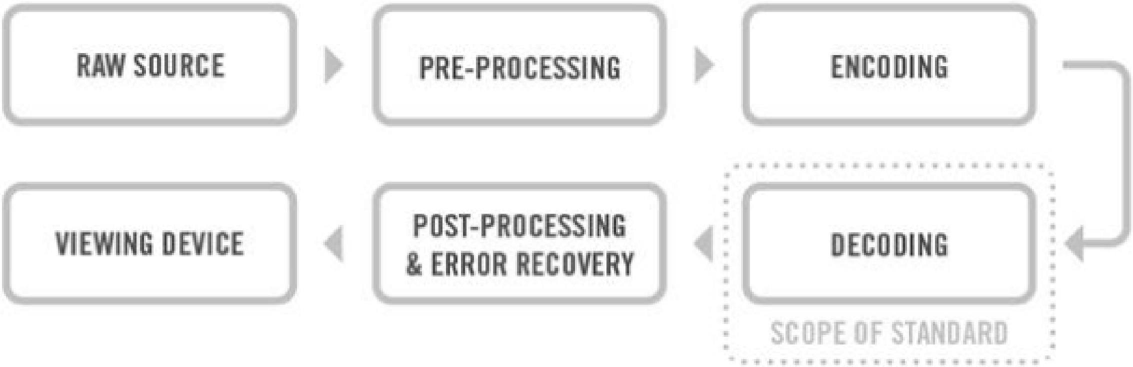


Fig. 2 Basic stages of video codecs

Basic principle

A video encoder works by using entropy coding techniques to reduce the size of raw video. As known, any signal has inherent entropy (the minimum amount of info that is necessary to represent a signal without distortion) and a corresponding rate-distortion function which shows the minimum amount of information necessary to represent a signal at a given distortion. As a result an encoder typically sends a much smaller amount of information than the original uncompressed signal. A video decoder can recover the original signal or an approximation to it, from the information sent by the encoder.

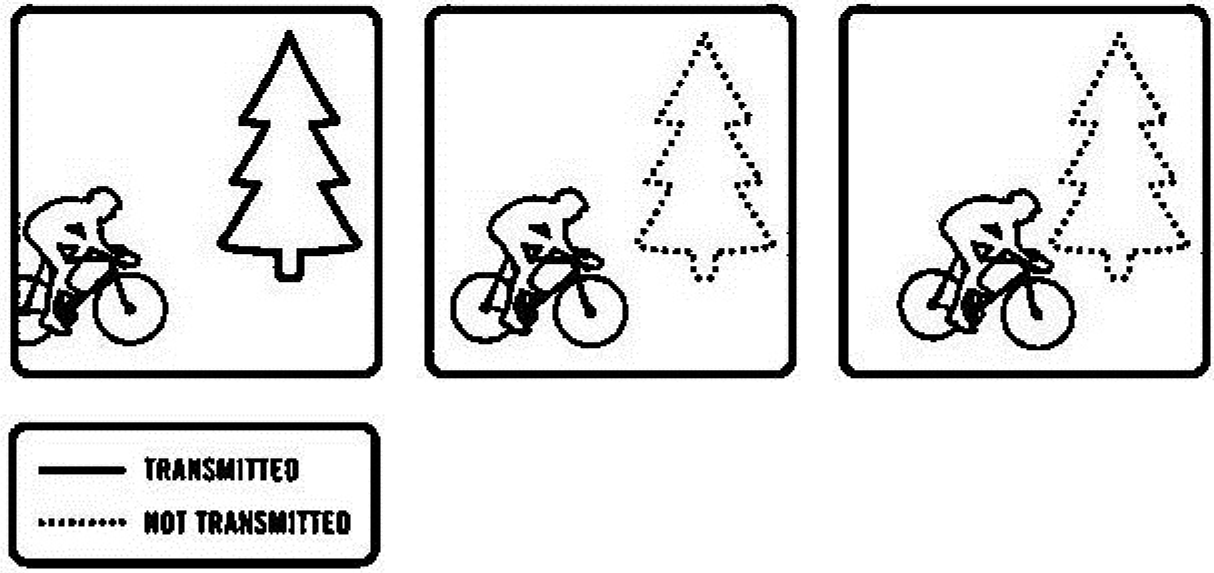


Fig. 3 Example of exploiting temporal redundancy between frames

Video encoding relies on two basic assumptions. The first is that human eye sensitivity to distortion in a picture is smaller for high frequencies. The second is that pictures share a lot of common pixels, although in different locations, even when movement occurs, between one frame and the next. Data can be reduced both by allowing distortion where it is less visible (ie high frequencies) and by sending only the difference between one picture and the next.

Pre/Post processing Stage

Usually input devices and video codecs use different color formats to represent images. For example cameras represent the captured data in the RGB field using 8 bits for each of the three colors (24 bit color depth). Codecs, on the other hand, use the YUV format, described above. Conversion from one format to the other is required before encoding and after decoding.

Encoding/Decoding Stage

Block-based processing

A video codec divides a frame into evenly sized blocks, known also as Macroblocks (MBs) or Coding Tree Blocks (CTBs). These are the basic processing units and each of these blocks passes through all of the following stages, in order for the whole frame to be encoded/decoded.

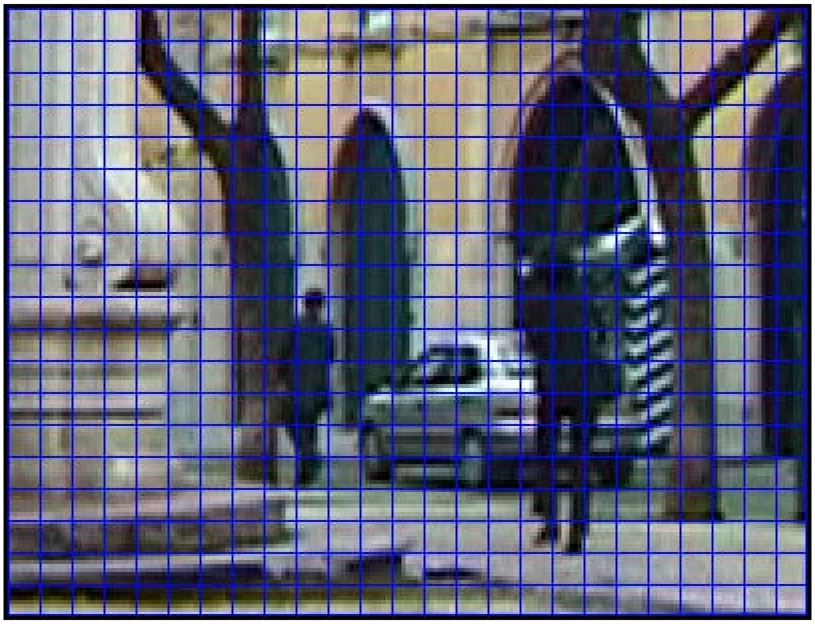


Fig. 4 A frame partitioned into blocks

Frame Types

Video frames are typically grouped into I and P/B. I (intra) are also called intra frames and are encoded independently, similar to the way JPEG pictures are coded.

P/B frames on the other hand, where P/B stands for predictive/bi-predictive are inter-coded, meaning their decoding depends on data from other decoded frames.

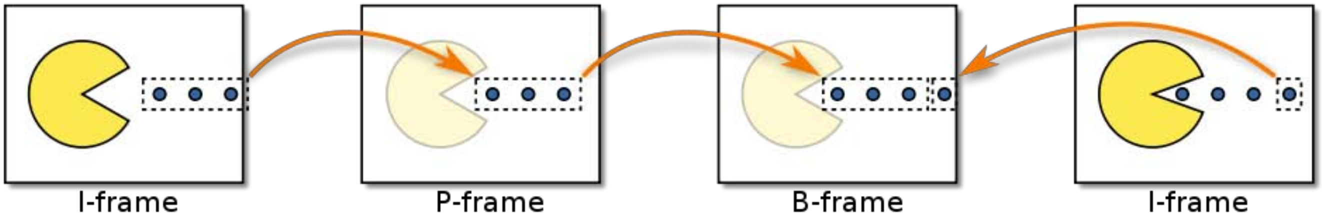


Fig. 5 Example of types. Notice how P-frames are dependent on I-frames, while B-frames depend on both

Pixel prediction

Intra-Prediction

Intra coding exploits *spatial* redundancy, which exists between regions of the same frame, by using pixels that have already been coded - and thus known to the decoder, as well - as reference for the current block. For example, horizontal intra prediction uses pixels from the border of the horizontally previous block as reference for the pixels of the current block. More details will be provided in subsequent sections.

Motion Estimation/Compensation (Inter-prediction)

Inter-coding uses motion estimation which takes advantage of the temporal redundancy that exists between regions of different frames. It is in this step that the different pixels are identified, making it the most important and the most compute intensive part of a video encoder. Instead of encoding a whole block, an encoder only calculates an offset (called a *Motion Vector*), which represents how much the current block has to move relative to a block in a reference frame. Finding the best motion vector (or equivalently the optimal reference block) out of multiple locations in different frames, is a process called *Motion Estimation* and is executed by the encoder. The reverse process, called *Motion Compensation*, is executed by the decoder and it uses the provided motion vector, in order to assemble the correct prediction.

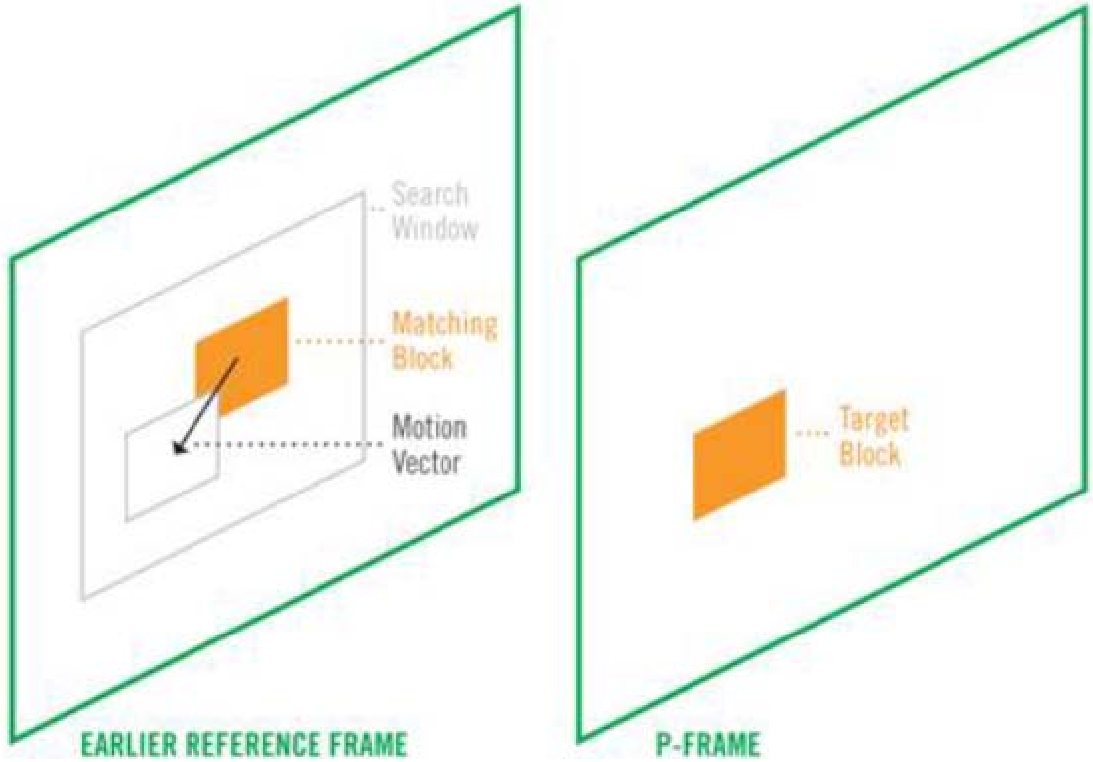


Fig. 6 Motion Estimation

Residual coding

After finding the best motion vector, the encoder calculates the difference between the current block and the reference block thus obtaining pixel differences that are further encoded.

Transform

The first step to improve coding efficiency to de-correlate the video signal, transforming it from the time to the frequency domain. In doing so, we obtain less non zero values in the Frequency domain compared to the pixel value differences in the time domain. The Discrete Cosine Transform (DCT) and Inverse DCT (IDCT) are useful for this purpose.

$$t_u = \frac{C_u}{2}\sum_{x=0}^{7} s_x \cos\frac{(2x+1)\pi u}{16}, u = 0..7$$

$$C_0 = \frac{1}{\sqrt{2}},\ C_n = 1 \text{ for n} = 1..7$$

DCT function

Quantization

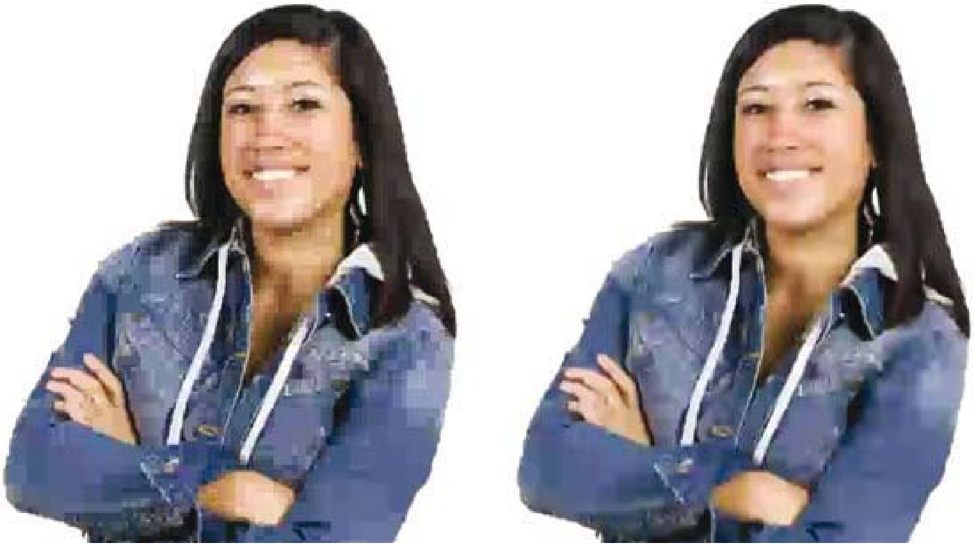
The frequency coefficients computed in the previous step are further quantized, which makes the encoding process lossy. However, the amount of distortion can be controlled by a quantization parameter (QP), taking values in the range [0,51], where QP = 0 represents no quantization (lossless quality) and QP = 51 means heavy distortion and high compression ratio. Quantization typically involves dividing frequency coefficients with an integer parameter, known as QP, and keeping the integral part.

Entropy coding

By now, the necessary data for transmission/storage (i.e motion vectors and quantized DCT coefficients) are available. However there is still redundancy in this data that can be exploited. Algorithms in this stage, such as Huffman encoding and Arithmetic coding, perform lossless compression thus determining the smallest amount of bits needed to represent this data. Thus, the final encoded bitstream is ready for storage or transmission.

Deblocking filtering

Finally because of the blocking nature of all the algorithms (block-based coding,motion estimation, transform etc) artifacts typically appear in the decoded video. Artifacts are uneven, in terms of location, regions, usually concentrated around block edges and are noticeable by the human eye. That’s why each decoded frame further undergoes one or more filtering processes that smooth out transitions between blocks.

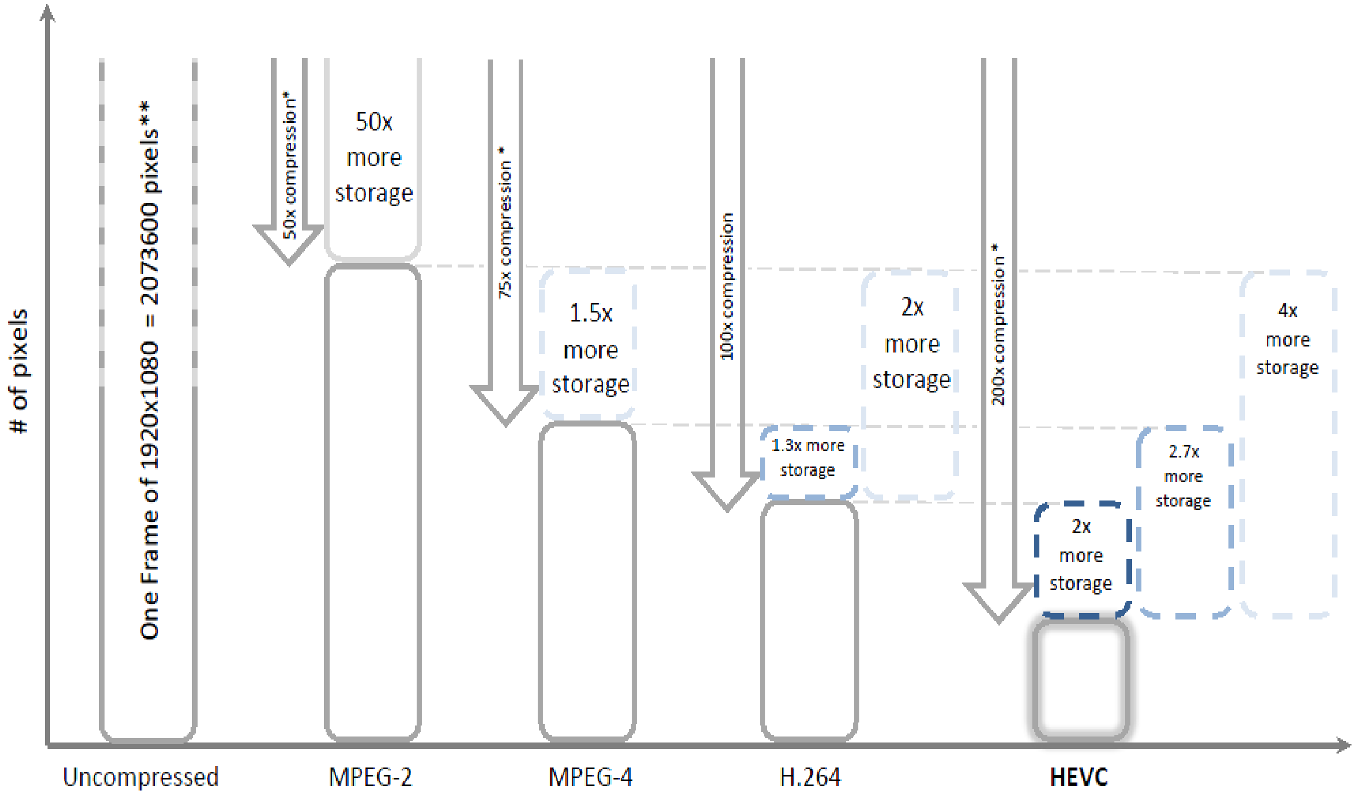


Without deblocking *With deblocking*

Fig. 7 Effects of the Deblocking filter

Brief History

Many successful and well-established standards have been developed. Starting from MPEG-1 and MPEG-2 in 1993 and 1996 respectively, which set the ground for video compression, later MPEG-4 and H.264, also known as Advanced Video Coding (AVC), emerged, the latter being the industry standard for the past 10 years. H.265, better known as HEVC, is H.264’s successor and intends to reduce data rates by approximately 50%. The next picture shows a brief history of the standards and the achieved compression ratios.



* Maximum possible compression. ** One Frame of 1920x1080 = 2073600 pixels

Fig. 8 Compression ratios

Chapter 2.

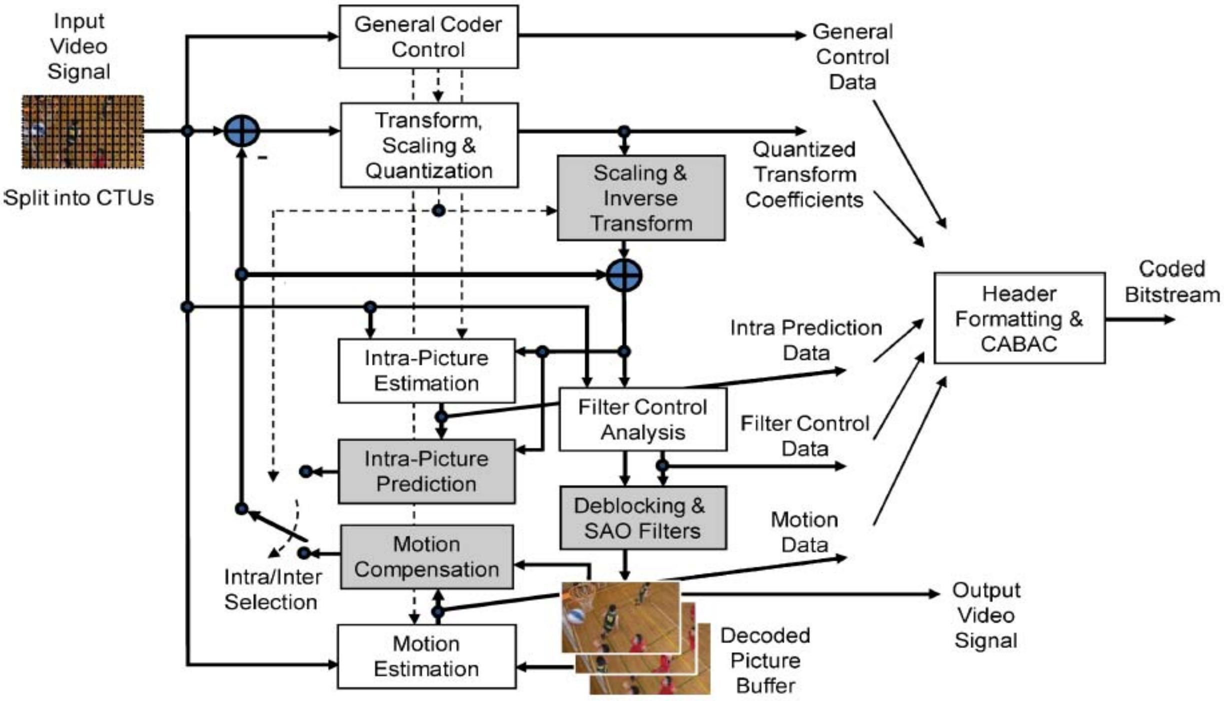
High Efficiency Video Coding (HEVC)

Introduction

HEVC was developed by the Joint Collaborative Team on Video Coding (JCT-VC), a partnership consisting of the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG).

Designed to evolve the video compression industry, HEVC addresses essentially all existing video applications but intends to reduce the average bit rate over H.264/MPEG4-AVC by an additional 50% at the same video quality, and provide substantially higher quality at the same bit rate. HEVC supports video resolutions of up to 8k x 4k and also provides tools to expose and exploit parallelism suitable for modern multicore/manycore architectures. Finally, the standard tries to remain network-friendly by targeting a wide range of devices and transport systems.

H.265 follows the well established hybrid (motion-compensated, transform + quantization) video encoder schemes, shown below.



Typical HEVC video encoder (with decoder modeling elements shaded in light gray).

Fig. 9 Hybrid HEVC encoder

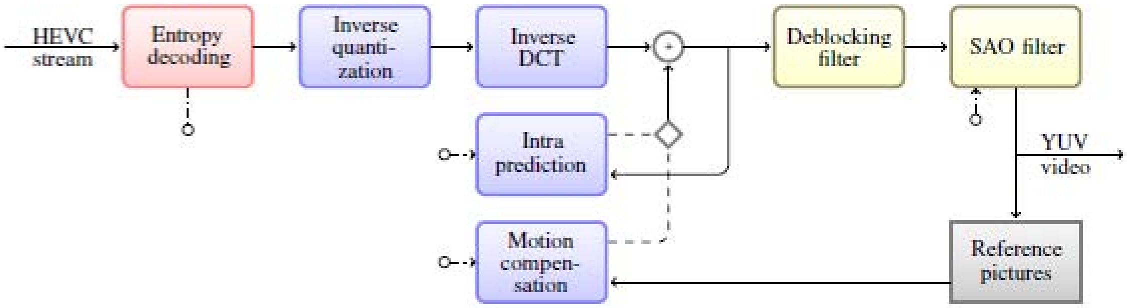


Fig. 10 HEVC decoder

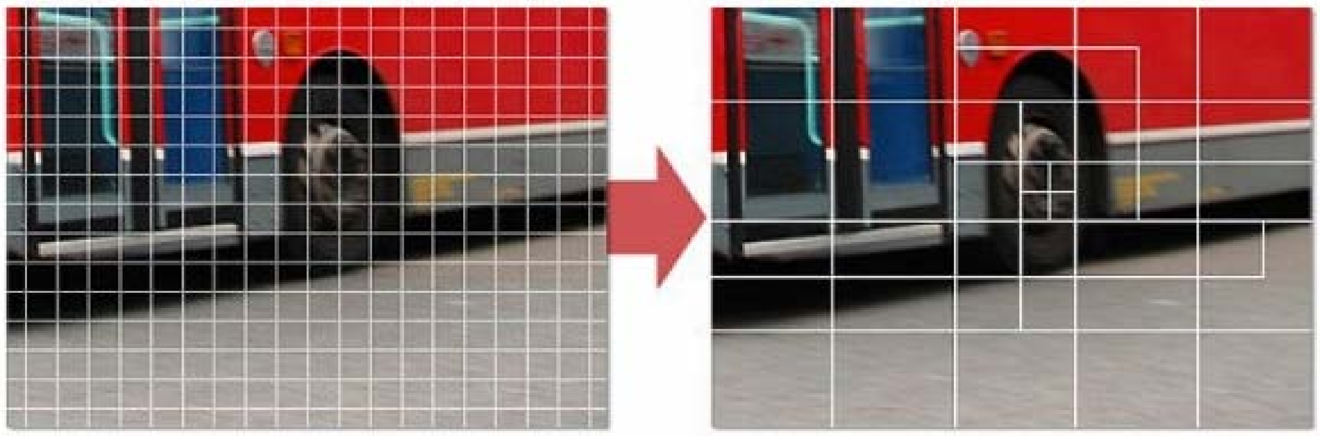
Partitioning

Basic blocks

The Basic Partitioning Unit

H.265 doesn’t use 16x16 macroblocks like its predecessors (MPEG1/2, MPEG4, H.264). We briefly mention that a macroblock in these older standards consists of 3 arrays, one of size 16x16 for the luminance (Y), and two for each one of the chrominance (Cb/Cr) components corresponding to the same physical location on a video frame. Depending on the chroma sampling, the chroma arrays can be 8x8 for the most popular consumer video format, called YUV420, 8x16 in case of YUV422 and up to 16x16 for the YUV444 color format.

In H.265 nomenclature, combining information from 3 color component arrays forms a *unit*, while each one of the components is termed a *block*. For ease in notation, we will be using the luminance component to describe H.265’s basic partitioning in the following. HEVC pictures are divided into so-called *coding tree blocks*, or CTBs for short, which appear in the picture in raster scan order. Depending on the stream parameters, they are either 64x64, 32x32 or 16x16, and this information is coded in the sequence-parameter-set (SPS) structure.



Macroblock partitioning (previous codecs) *CTB partitioning (HEVC)*

Fig.11 HEVC basic partitioning

Each CTB can be split recursively in a quadtree structure, all the way down to 8x8. So for example a 32x32 CTB can consist of three 16x16 and four 8x8 regions. As another example a 64x64 CTB can consist solely of one 64x64 region. These partitions of a CTB are called *coding blocks*, or CBs. CBs are the basic unit of prediction in HEVC. The CBs in a CTB are traversed and coded in Z-order.

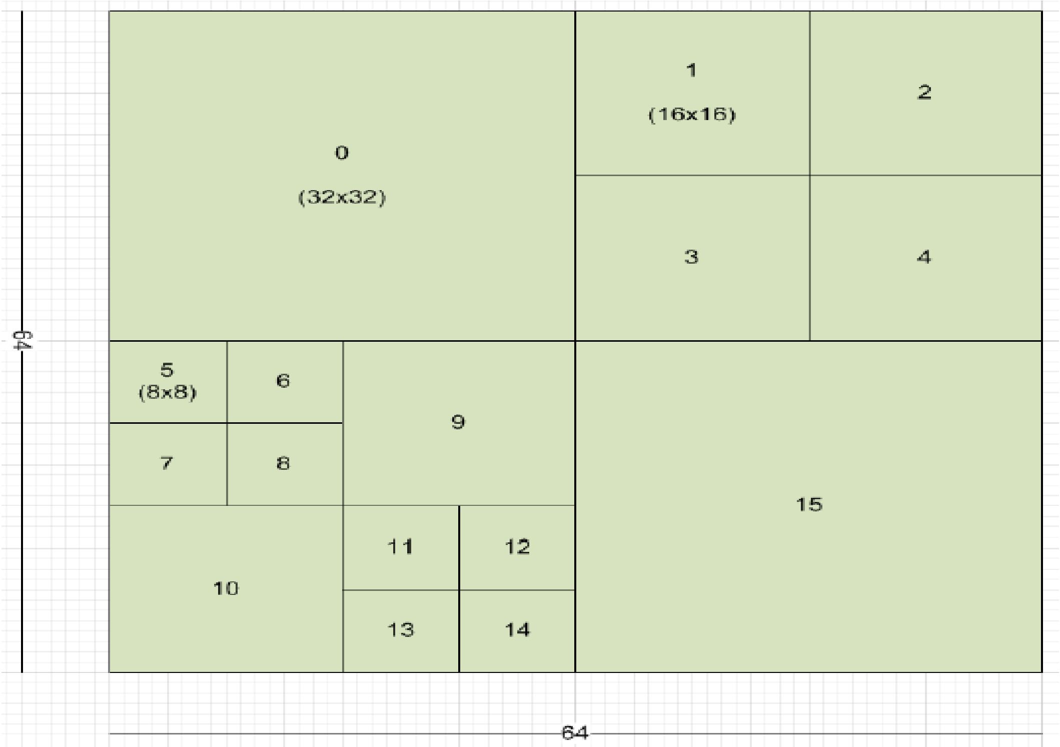


Fig. 12 Example ordering in a 64x64 CTB.
Numbers 0 through 15 illustrate the Z-order traversal

Prediction blocks

Solely for the purposes of Inter and Intra prediction, a CB can be further subdivided (not recursively, only once) into *prediction blocks*, PBs for short. This decision is made by the encoder and it's conveyed to the decoder. For intra-prediction a CB can be further split, with a smallest PB size of 4x4. For inter-prediction, alternative subdivisions are available since 4x4 (subdivision #4 in the figure below) is forbidden.

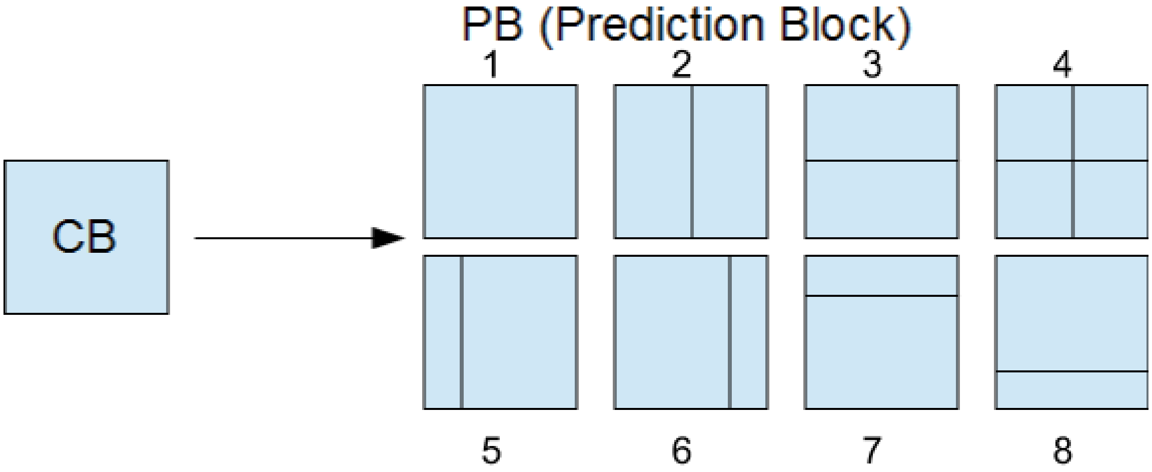


Fig. 13 PB subdivisions

Intra-prediction subdivisions: 1,4

Inter-prediction subdivisions: 1,2,3,5,6,7

Transform blocks

For residual coding, a CB can be recursively subdivided (the same way CTBs are split into CBs) into *transform blocks*, TBs, of size 4x4, 8x8, 16x16 or 32x32, as shown below on Fig.X Partitioning of TBs and PBs is performed independently, since it happens at different stages, thus PBs and TBs need not be aligned.

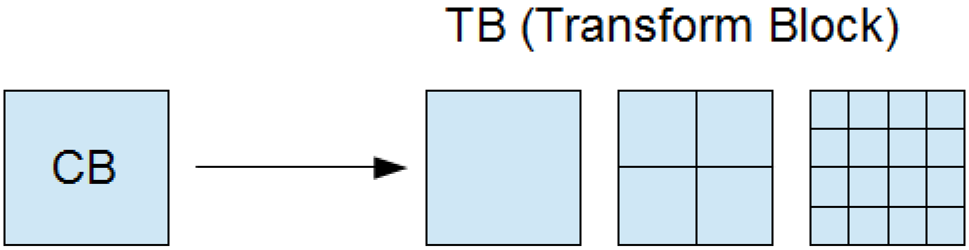


Fig. 14 TB subdivisions

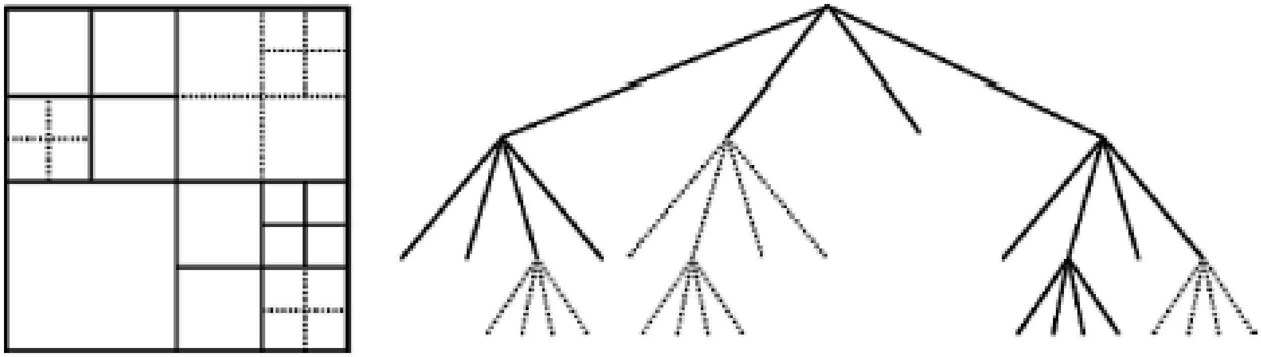


Fig. 15 Transform quadtree recursive partitioning (CBs full lines, TBs dotted lines)

HEVC supports video of resolution up to 8k x 4k, so it is necessary to offer so many partitioning options, in order to cover videos of different resolutions/contents. It provides the encoder great flexibility to use large partitions when no motion, or large object motion is present and small partitions when more detailed predictions are needed (small object motion). This leads to higher coding efficiency, since large prediction units (up to its maximum, the size of CTBs) can be cheaply coded when they fit the content. Furthermore, when some parts of a CTB need more detailed predictions, these can also be efficiently described. Quadtree representations, although not a novel idea, are applied extensively in H.265. In previous standards, they were limited to the “4MV” mode in MPEG4-part2 (partition of a 16x16 macroblock into 4, 8x8 prediction blocks) and variable prediction sizes between 4x4 and 16x16 in H.264.

Chroma sampling

HEVC supports YCbCr 4:2:0 sampling which means that each luma CTB of LxL size is followed by 2 Chroma CTBs of size L/2 x L/2. The structure which holds all this data is called CTU (coding tree *Unit*). A CTU is a *logical* unit - the actual data is contained in the corresponding CTBs, which are the ones that undergo all subsequent processing. Respectively there are CUs, PUs and TUs, corresponding to CBs, PBs and TBs. One needs to keep in mind that chrominance CTBs, CBs, PBs and TBs follow the quadtree structure of their luminance counterparts.

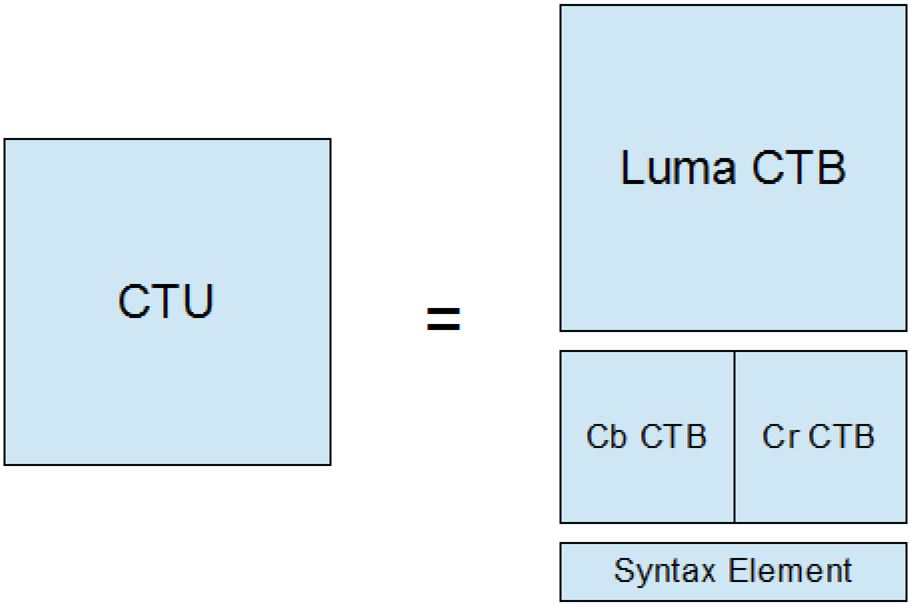


Fig. 16 CTU logical structure with YCbCr 4:2:0 sampling

Block arrangements and Parallelization tools

Slices

Slices are consecutive CTUs that are processed in raster scan order. A picture is a collection of one or more slices. The default encoder configuration is 1 slice per frame. Two slices can be correctly decoded independently, meaning that all data required for the different decoding stages (transform,quantization,motion compensation) is available in each slice header. The algorithms applied on each stage have no data dependencies between slices of the same picture, with the exception of the deblocking filter when applied on slice boundaries.

There are 3 kind of slices

1. I-slices, where only intra-prediction is allowed for every CTU.
2. P-slices, allow both intra-prediction and inter-prediction with only one frame per PB can be used as a reference (uni-prediction)
3. B-slices, allow intra-prediction and inter-prediction, with both uni-prediction and bi-prediction. In the later case, pixels from two different frames can be averaged - or weighted-averaged, in general - to form prediction for a given

PB.

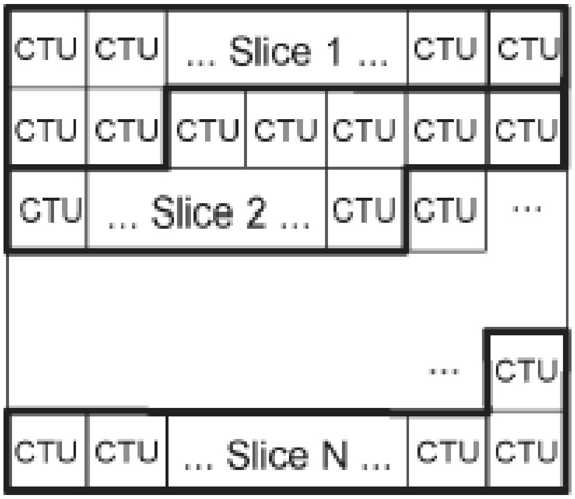


Fig. 17 Slice partitioning

Parallelization tools are used in order to provide implementability for parallel processing.

Tiles

Tiles are self-contained and independently decodable rectangular regions of the picture. Multiple tiles may share header information by being contained in the same slice. Alternatively, a single tile may contain multiple slices. A tile consists of a rectangular arrangement of CTUs.

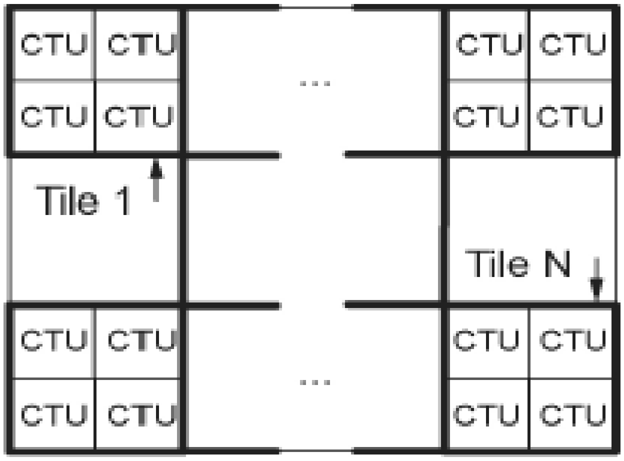


Fig. 18 Tile partitioning

Wavefront parallel processing (WPP)

Finally, with wavefront parallel processing, a slice is divided into rows of CTUs. Decoding of each row can begin as soon as a few decisions that are needed for prediction and adaptation of the entropy coder have been made in the preceding row. This supports parallel processing of rows of CTUs by using several processing threads in the encoder or decoder (or both), typically with some time-offset among them. For design simplicity, WPP is not allowed to be used in combination with tiles.

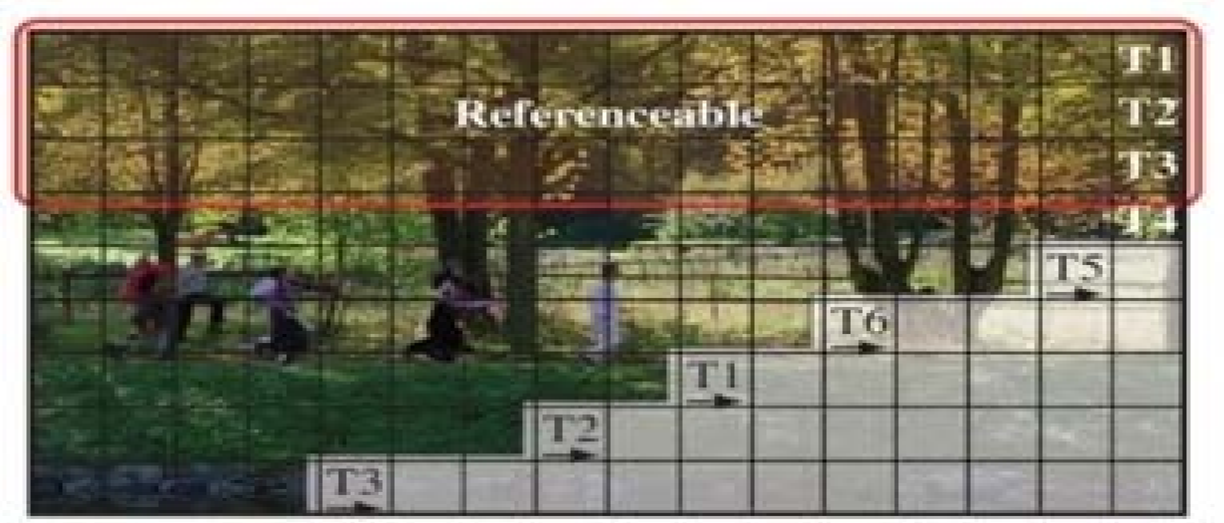
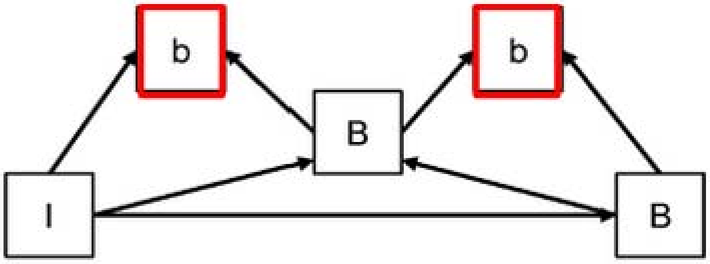


Fig. 19 WPP partitioning. Workloads of threads T1 and T2 are distanced by 2 CTUs

Temporal sub-layering

A more subtle form of parallelization can be achieved with temporal sub-layering support. Each frame belongs to a layer identified by a temporal id (0,...6). Frames on the same temporal layer with non overlapping dependencies can be decoded in parallel as shown in the example below.



POC	0	1	2	3	4
Decoding order	1	4	3	5	2
RPS	–	[0,2]	[0,4]	[2,4]	[0]

Fig. 20 Example of temporal sub-layering. Red colored frames have temporal id = 2. When their dependencies are resolved, they can be decoded in parallel.

Inter-prediction

HEVC, just like H.264, uses quarter pixel precision for Inter-prediction, which means that a motion vector can point to units of one-quarter the distance between adjacent luma samples. For chroma samples the precision is determined from the chroma sampling format, which for 4:2:0 YCbCr, is one-eighth the distance of chroma samples.

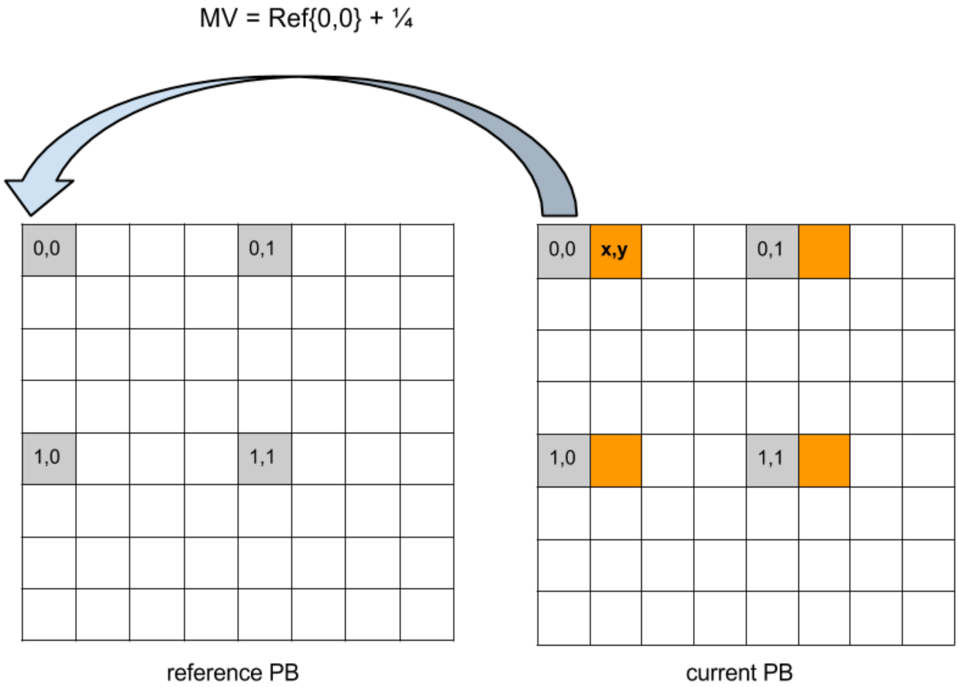


Fig. 21 Example of quarter pixel precision.
Grey squares represent integer (original) pixel sample.
Orange samples are to be generated.

Non-integer position samples have to be generated by the decoder, since frame data contain only integer pixel values. HEVC uses separable application of an 8-tap filter for the half-sample luma positions and a 7-tap filter for the quarter-sample luma positions. In H.264, there is a 6-tap filter applied separably in order to obtain half-pixel values. Quarter-pixel values are then obtained by averaging half- and integer pixel values. In H.265, rounding is only applied after the final filter pass has been completed.

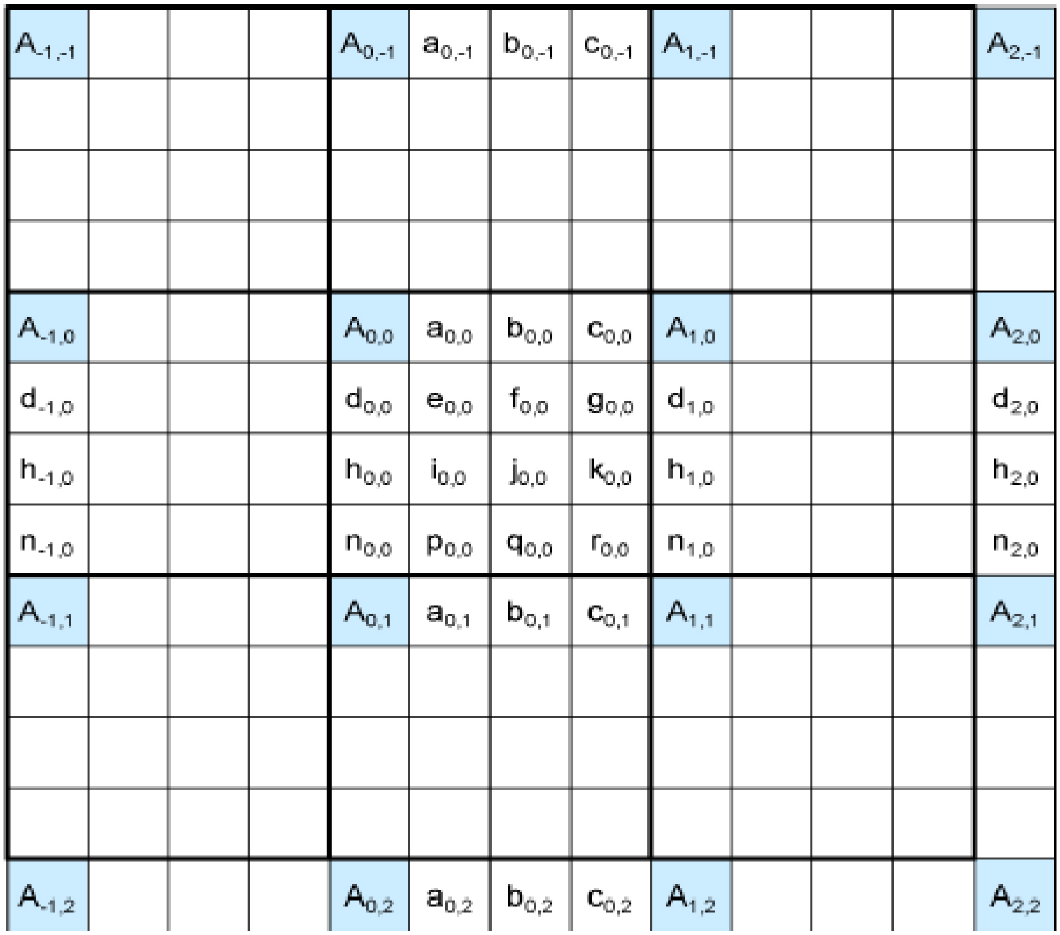
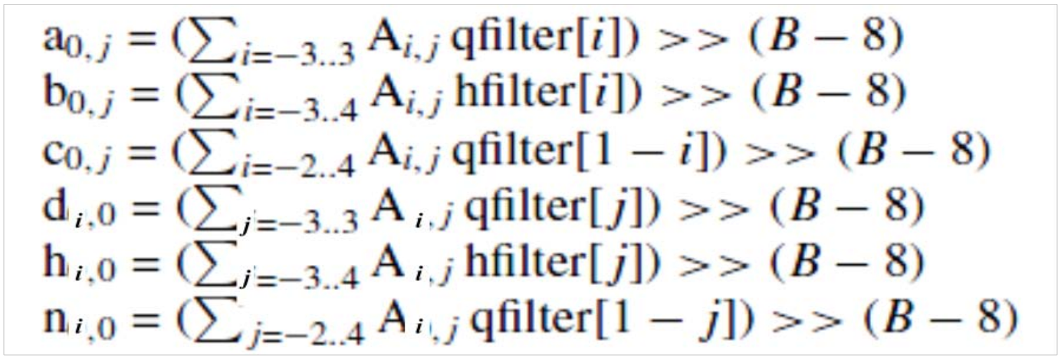


Fig. 22 Luma interpolation positions

upper case A i,j :	***available** samples on integer positions*
lower case (rest):	*samples that must be **generated**.*
a,b,c & d,h,n :	*only **one** filter application needed*
e,f,g,i,j,k,p,q,r:	***two** filter applications required.*

1-D Filter pass

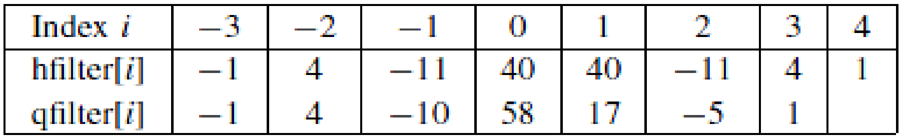
Samples labeled **a0,j, b0,j, c0,j, di,0, hi,0** and **ni,0** are derived from samples **Ai,j** by applying the 8-tap filter for half-sample positions (**b0,j** and **hi,0**) while the 7-tap filter is used for the quarter-sample positions (**a0,j, c0,j, di,0** and **ni,0**) as follows:



$$
\begin{aligned}
a_{0,j} &= \left(\textstyle\sum_{i=-3..3} A_{i,j}\,\mathrm{qfilter}[i]\right) >> (B-8)\\
b_{0,j} &= \left(\textstyle\sum_{i=-3..4} A_{i,j}\,\mathrm{hfilter}[i]\right) >> (B-8)\\
c_{0,j} &= \left(\textstyle\sum_{i=-2..4} A_{i,j}\,\mathrm{qfilter}[1-i]\right) >> (B-8)\\
d_{i,0} &= \left(\textstyle\sum_{j=-3..3} A_{i,j}\,\mathrm{qfilter}[j]\right) >> (B-8)\\
h_{i,0} &= \left(\textstyle\sum_{j=-3..4} A_{i,j}\,\mathrm{hfilter}[j]\right) >> (B-8)\\
n_{i,0} &= \left(\textstyle\sum_{j=-2..4} A_{i,j}\,\mathrm{qfilter}[1-j]\right) >> (B-8)
\end{aligned}
$$

Fig. 23 First filter pass formulas

B is the Bitdepth, in bits per luma/chroma sample, which is usually 8. This means that the rounding operation (arithmetic right shift by 0, in this case) can be omitted. The filter coefficients for Luma samples are the following - hfilter[] is the 8-tap filter applied for half-pixel positions and qfilter[] is the 7-tap filter is applied in the form given below, or reflected for the two quarter-pixel positions, ¼ and ¾, correspondingly.



Index i	−3	−2	−1	0	1	2	3	4
hfilter[i]	−1	4	−11	40	40	−11	4	1
qfilter[i]	−1	4	−10	58	17	−5	1	

Fig. 24 Luma interpolation filter coefficients

2-D Filter pass

In order to calculate the rest of the positions, the filter is applied again but now on the non-integer samples calculated in the previous step, as shown below. When B = 8, the order of the filter applications is interchangeable because there is no loss of precision.

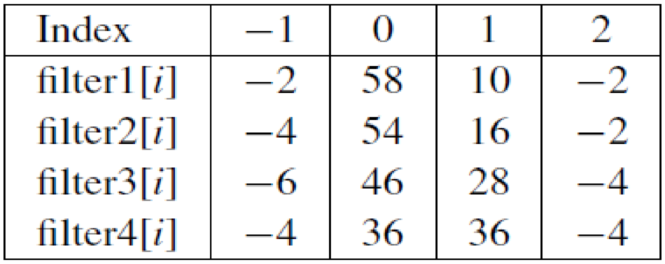
$$
\begin{aligned}
\mathrm{e}_{0,0} &= (\textstyle\sum_{v=-3..3} \mathrm{a}_{0,v}\,\mathrm{qfilter}[v]) >> 6\\
\mathrm{f}_{0,0} &= (\textstyle\sum_{v=-3..3} \mathrm{b}_{0,v}\,\mathrm{qfilter}[v]) >> 6\\
\mathrm{g}_{0,0} &= (\textstyle\sum_{v=-3..3} \mathrm{c}_{0,v}\,\mathrm{qfilter}[v]) >> 6\\
i_{0,0} &= (\textstyle\sum_{v=-3..4} \mathrm{a}_{0,v}\,\mathrm{hfilter}[v]) >> 6\\
j_{0,0} &= (\textstyle\sum_{v=-3..4} \mathrm{b}_{0,v}\,\mathrm{hfilter}[v]) >> 6\\
\mathrm{k}_{0,0} &= (\textstyle\sum_{v=-3..4} \mathrm{c}_{0,v}\,\mathrm{hfilter}[v]) >> 6\\
\mathrm{p}_{0,0} &= (\textstyle\sum_{v=-2..4} \mathrm{a}_{0,v}\,\mathrm{qfilter}[1-v]) >> 6\\
\mathrm{q}_{0,0} &= (\textstyle\sum_{v=-2..4} \mathrm{b}_{0,v}\,\mathrm{qfilter}[1-v]) >> 6\\
\mathrm{r}_{0,0} &= (\textstyle\sum_{v=-2..4} \mathrm{c}_{0,v}\,\mathrm{qfilter}[1-v]) >> 6.
\end{aligned}
$$

Fig. 25 Second filter pass formulas

After this step, *weighted prediction* is applied if necessary. While H.264 supported both implicit and explicit prediction, HEVC supports only the latter. The encoder conveys the specific values, which are scaled, offset and rounded by the decoder, using the following formula

```
src0[x][y] * w0+ src1[x][y] * w1 + (( offset0 + offset1 + 1)>> 1)
```

Chroma filtering requires only 4 adjacent samples (4-tap filter) and, as we explained earlier, the accuracy is one eighth of a pixel. Coefficients for positions ⅛, 2/8, ⅜ and 4/8 correspond to filter1[], filter2[], filter3[] and filter4[], while positions ⅝, 6/8 and ⅞ are reflected versions of filter3[], filter2[] and filter1[], correspondingly.

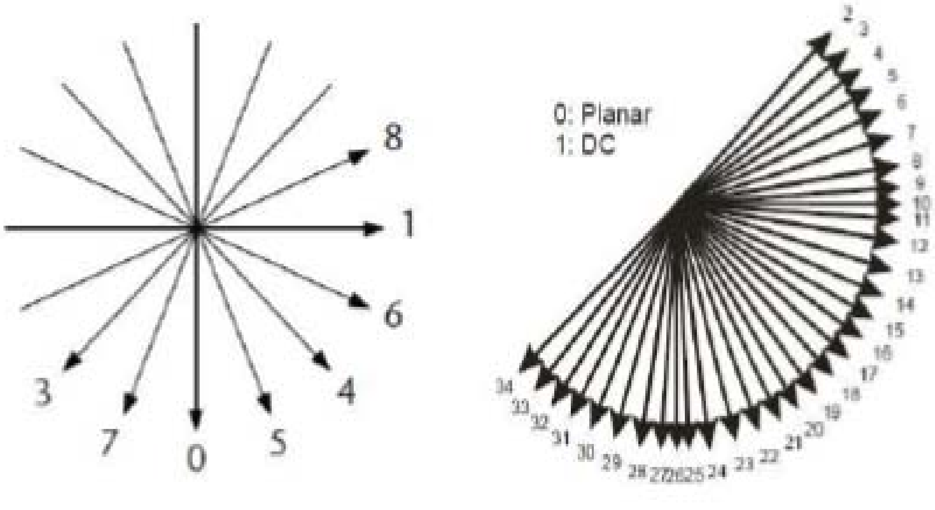


Index	−1	0	1	2
filter1[i]	−2	58	10	−2
filter2[i]	−4	54	16	−2
filter3[i]	−6	46	28	−4
filter4[i]	−4	36	36	−4

Fig. 26 Chroma interpolation filter coefficients

Intra-prediction

Intra prediction is similar to the one in H.264 but with additional modes available. In total, 35 modes are available, 33 of those are directional. The other two are the DC (use the average of all neighboring pixels as common prediction for all pixels in current block) and Planar (form a bi-linear interpolation surface with respect to neighboring pixels) modes.



H.264 VS. HEVC INTRA PREDICTION MODES

Residual Coding

Transform

Integer basis functions are integer versions of the discrete cosine transform (DCT) and are defined for square TB sizes 4x4, 8x8, 16x16, and 32x32. For the 4×4 transform of luma intra-picture prediction residuals, an integer transform based on the discrete sine transform (DST) is alternatively specified.

Quantization

As in H.264/MPEG-4 AVC, uniform reconstruction quantization (URQ) is used in

HEVC, with quantization scaling matrices supported for the various transform block sizes. Quantization parameter (QP) varies from 0 (no quantization applied) to 51 (highest compression ratio, with heaviest compression artifacts).

Context-adaptive-binary-arithmetic-coding (CABAC)

HEVC specifies only one entropy coding method, CABAC, rather than two (Huffman coding/arithmetic coding) as in H.264/MPEG-4 AVC. The core algorithm of CABAC is unchanged, using different contexts for the various syntax elements (motion vectors, transform coefficients), converting each syntax element into a binary string first and using binary arithmetic coding, with adaptive probability estimates for each bin (0/1).

In loop filters

After decoding of a frame has been completed, two post-processing filters are applied before a frame can be stored in the Decoded Picture Buffer and thus used for rendering and subsequent motion compensation of later frames.

Deblocking Filter

The Deblocking Filter (*DBF*), which is applied first, intends to reduce artifacts introduced by block-based processing. Unlike H.264, which deblocked 4x4 blocks, HEVC operates on an 8x8 grid, independently of the PB or TB partitioning applied on previous decoding stages. Similar to H.264, frame boundaries are not processed by the deblocking filter. Slice boundaries can also be omitted, depending on proper encoder flag in the slice header.

Deblocking Filter processes an 8x8 block as follows. For each of the 8 four-pixel long block boundaries (4 vertical and 4 horizontal), it makes a decision based on its neighboring pixels, whether to apply strong, weak or no filtering at all. Each block boundary has a corresponding area of 4x8 pixels, on which filtering is applied. All vertical edges in the picture are deblocked first, followed by horizontal edges. Because of the 8-pixel separation between edges, edges do not depend on each other, thus enabling a highly parallelizable implementation.

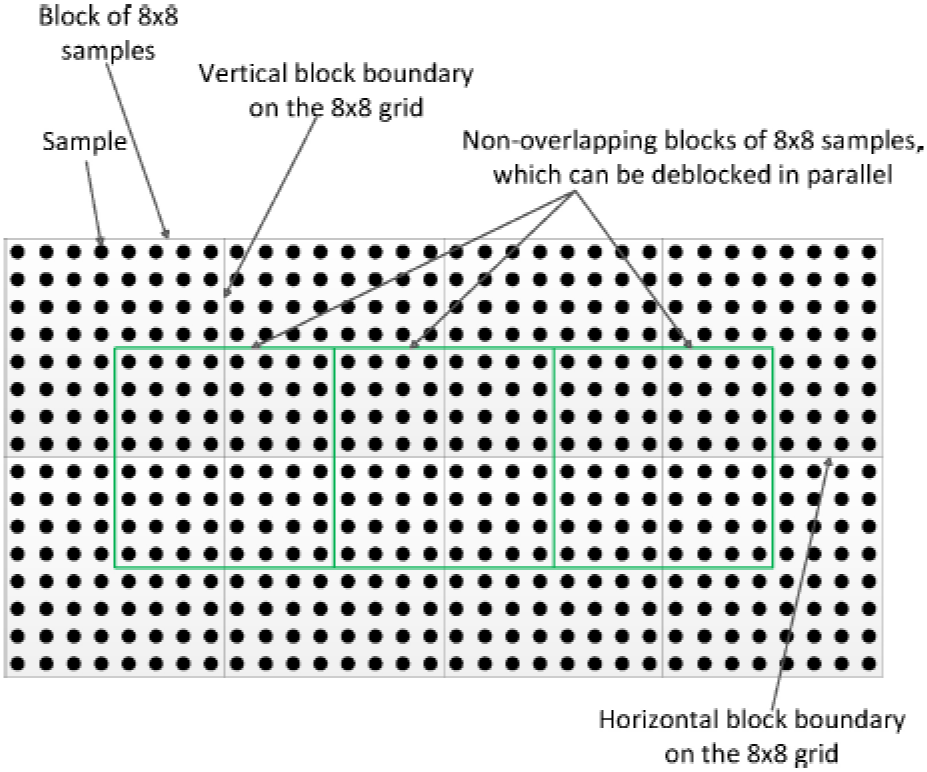

Block of 8x8 samples
Vertical block boundary on the 8x8 grid
Sample
Non-overlapping blocks of 8x8 samples, which can be deblocked in parallel
Horizontal block boundary on the 8x8 grid

Fig. 27 The deblocking filter grid

Sample Adaptive Offset (SAO)

After deblocking is performed, a second filter optionally processes the picture. This filter is called Sample Adaptive Offset, or SAO. This relatively simple process is done on a per CTB basis, and operates once on each pixel. SAO is a process that modifies the decoded samples by conditionally adding an offset value to each sample after the application of the deblocking filter, based on values in look-up tables transmitted by the encoder. For each CTB, the bitstream codes a filter type and four offset values, which range from -7 to 7 (in 8 bit precision video).There are two types of filters: Band and Edge.

HM 10.0 reference decoder

In order for the HEVC standard to be tested and evaluated by users, reference software source code, referred to as HM (HEVC test Model) is provided online for both the encoder and the decoder. The code is written in C++, but it is rather slow since it targets code-readability and experimentation, rather than performance. Alongside the reference code, reference test data such as encoded bitstreams, with their encoding configuration parameters and cross-check time results, are also provided.

Building

Since this work focused on HEVC decoding performance, we built the 32-bit HM 10.0 decoder using Microsoft Visual Studio 2010 on a Windows 7 64-bit platform with an Intel Xeon E5520 Quad-core @ 2.27GHz processor. 20 official reference encoded bitstreams were used for testing, all of which in YUV420 color format, at 832 x 480 resolution and frame rates in the range 30, 50 or 60 fps, at various quantization parameters (22,27,32,37) corresponding to compressed bitrates between 1Mbps - 6Mbps. The default configuration (.cfg file) used for the encoding of these bitstreams doesn’t use slices,tiles, or WPP nor uses temporal sub-layering and the Group-of-Pictures (GOP) structure is open. Such configuration allows no exploitation of standard parallelization tools and relies solely on an optimized reference decoder to achieve real-time decoding. The official cross-check times, which match the ones we measured, show that only a few of these bitstreams can be decoded in real time on the target platform.

Profiling

To obtain profiling results we used Intel® VTune™ Amplifier XE 2013. In order to avoid additional CPU usage, other tasks not essential to the decoding process, such as MD5 frame checksum comparisons and storing of the decoded data to an output yuv file, were disabled, after verifying the validity of the decoded results. Furthermore, all console output was redirected to a text file.

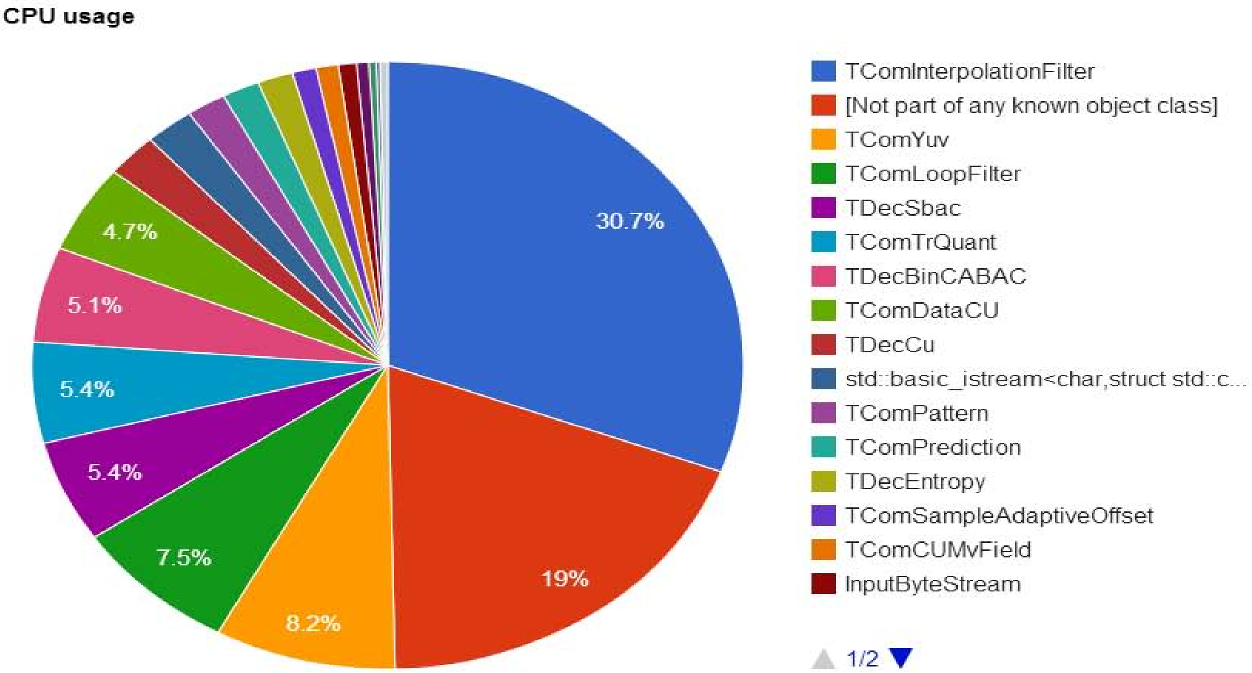


Fig. 28 Profiling results of the reference HM 10.0 decoder

TComInterpolation is the most compute-intensive class and is the first one we targeted for optimization. This class is responsible for interpolation filtering, which is applied during inter-prediction. The second group, shown as *"[not part of any known object class]"* in the profiling results because they are stray C-like functions that don't belong to any specific C++ class, contains miscellaneous functions, some used for memory allocation (such as *memfree()* and *memcopy()*), and others, like one called *partialButterfly(),* used in the Transform phase. *TComYuv* follows, which manages the yuv picture buffers and after that is *TComLoopFilter*, which handles the Deblocking and SAO filters. It is worth mentioning that these top-4 CPU consumers account for about ⅔ of total decoder complexity.

Chapter 3. Decoder optimizations

Optimized Interpolation Filter

Introduction

As a quick reminder, fractional sample interpolation for luma samples in HEVC applies an 8-tap filter on eight neighbouring pixels for the half-sample positions and a 7-tap filter on seven neighbouring pixels for the quarter-sample positions, while chroma filtering uses 4 pixels per output sample. In our implementation we exploited the nature of this algorithm using SSE2 SIMD instructions and we achieved a speedup for the entire class of 3.41x - 5.64x, depending on the test stream.

We begin this section by describing the parallelization techniques used on the HEVC interpolation filter, after which we present some code-specific optimizations we applied. Finally we compare the profiling results of the optimized versus the original version.

Algorithmic Optimizations

We need to remember that Intel's IA32 and x64 architectures offer a variety of single-instruction-multiple-data (SIMD) registers and assembly instructions that can process 64-bit (MMX registers) or 128-bit quantities (XMM registers) by treating them as arrays of bytes, short-integer (16-bit) or integer (32-bit) elements.

Optimized Vertical Luma Filter

The optimized Vertical filtering algorithm, as shown below, is quite straightforward and it is based on data-parallelization. If we multiply each PB line with its respective filter coefficient, and then add the 8 intermediate results we acquire 8 interpolated samples. The reference software (HM) uses 16-bit data to store pixel values, in order to cover all possible video signals, which means a 128-bit SSE2 register can contain up to 8 pixels. Since the most commonly used PB size is 8x8, a throughput of 8 samples per round of calculations can be achieved.

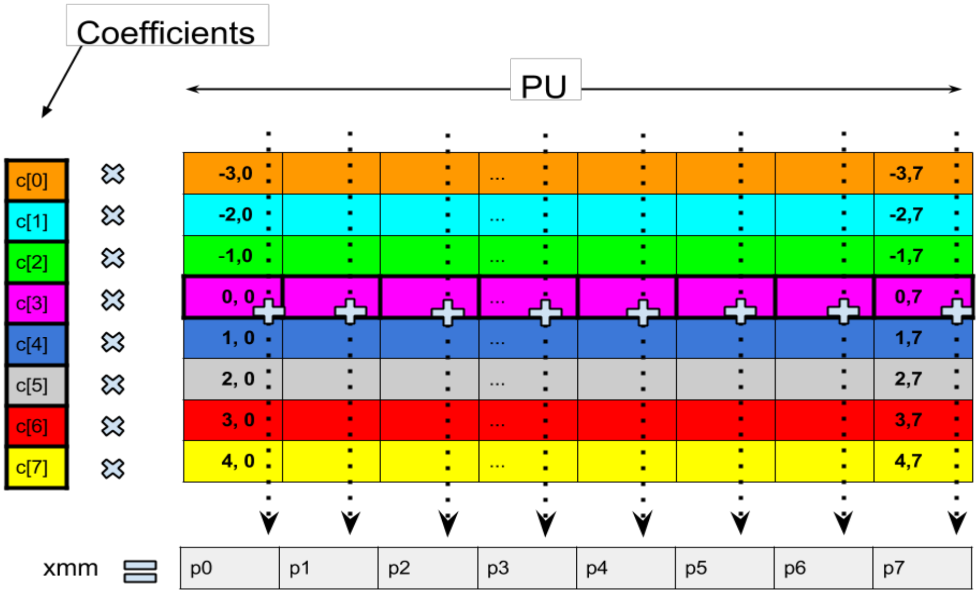


Fig.29 Vertical parallelization

Optimized Horizontal Luma Filter

First version

In our first attempt to optimize horizontal filtering, we used dot-product between a line of a PB (up to 8 pixels, to be precise) and the filter coefficient values, since SIMD instructions allow for a limited application of the Multiply-and-Add (MADD) operation. However, the challenge was to achieve throughput of 8 results per 8 MADD operations, given that intermediate multiplication results needed to be stored in 32-bit accuracy, which limited throughput to 4 samples. In our initial approach, we calculated 4 dot products which we stored on one SSE register and then stored to memory, as illustrated below. This had a throughput of 4 samples per round of calculations.

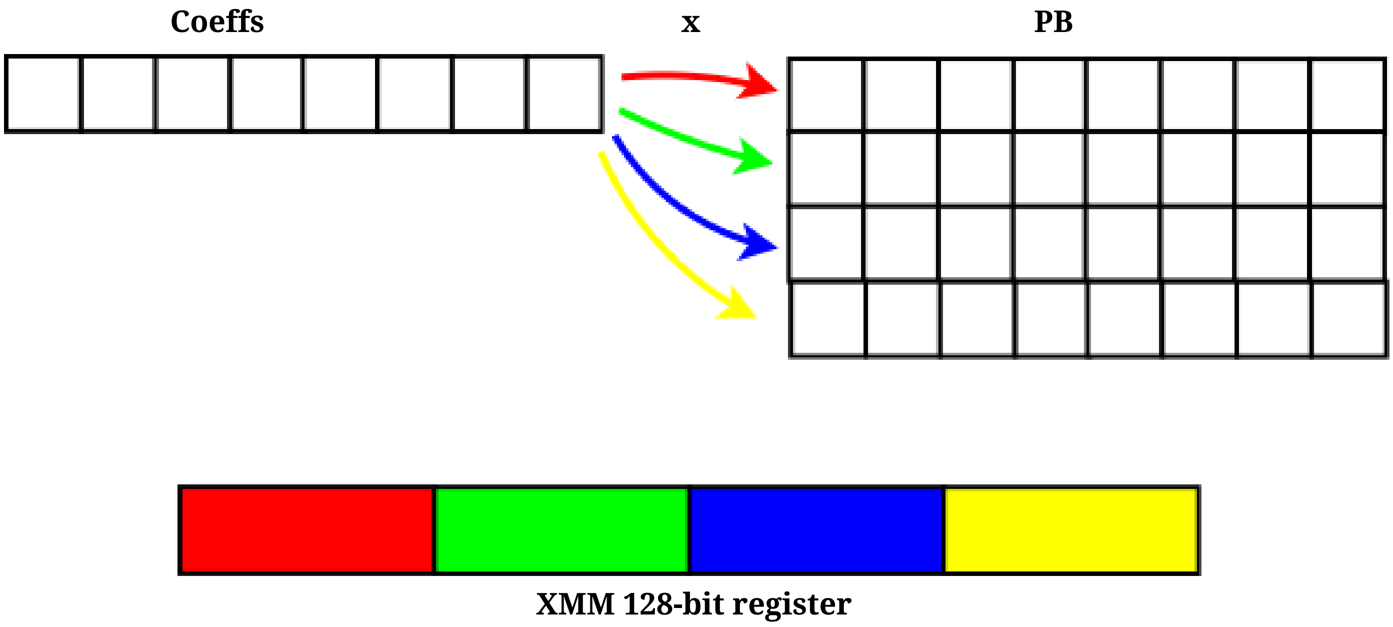


Fig. 30 Horizontal parallelization. Each colored line represents a 32-bit dot- product which is stored on an XMM register accordingly.

Following the profiling process described in chapter 2, we tested the available bitstreams on different QP levels. Our initial profiling results (shown below) show that there is room for improvement.

	Optimized code	Serial code	Speedup
BQMall 832x480	3.21s	6.54s	**(2.03x)**
BasketBallDrillText 832x480	1.86s	3.89s	**(2.09x)**
BasketBallDrill 832x480	1.86s	3.88s	**(2.09x)**
PartyScene 832x480	3.3s	7.1s	**(2.15x)**

Fig. 31 Results of first attempt in horizontal filtering optimization

Second (and final) version

We took a better look on the horizontal filtering algorithm and redesigned it to follow to same scheme as the Vertical filter. We no longer use dot products to calculate immediately each fractional sample but rather a pipeline where each loaded *pixel stride* (8 pixels for luma, 4 for chroma) is multiplied with its respective coefficient. The first loaded pixel stride begins from position **i**, the second from **i**+**1**, … and the eighth from **i**+7 as shown on Fig.32. All the intermediate results are added at the end, thus forming 8 results per round of calculations.

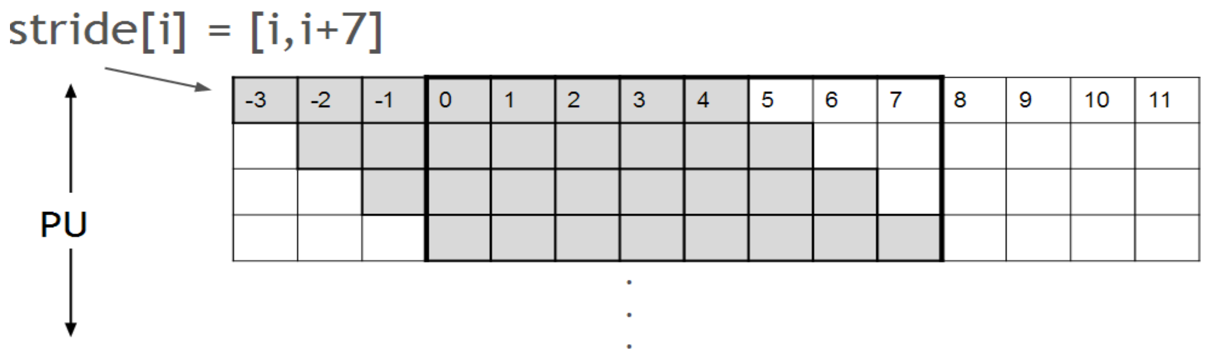


Fig. 32 Example of loaded pixel strides for Horizontal interpolation filtering.

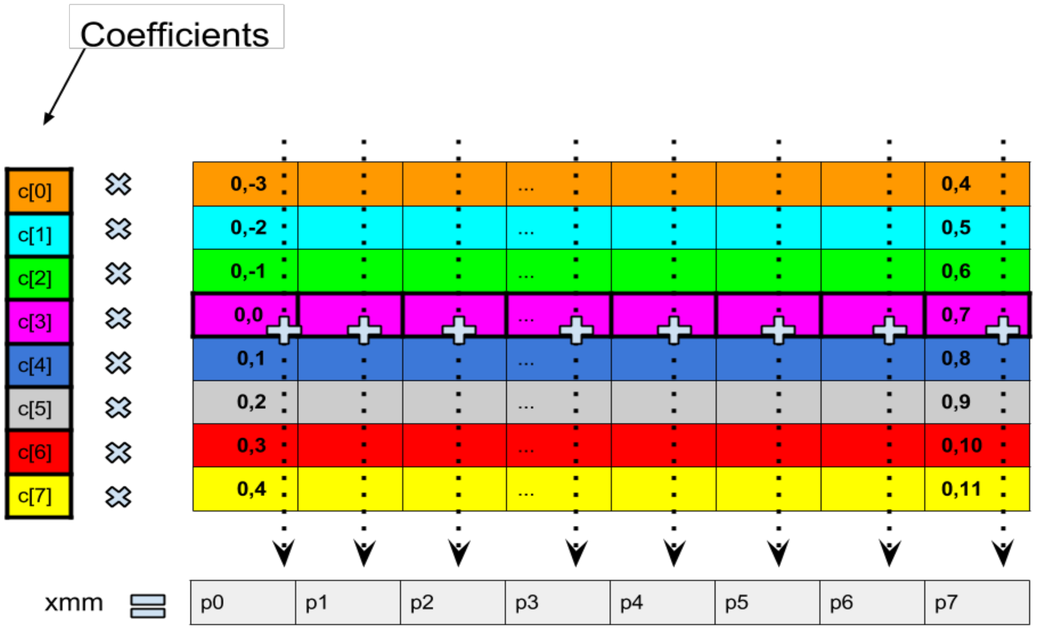


Fig. 33 Horizontal parallelization

Optimized Chroma Filtering

The same algorithms were applied on the Chroma functions of the class, with slight modifications that are explained below.

PB Sizes

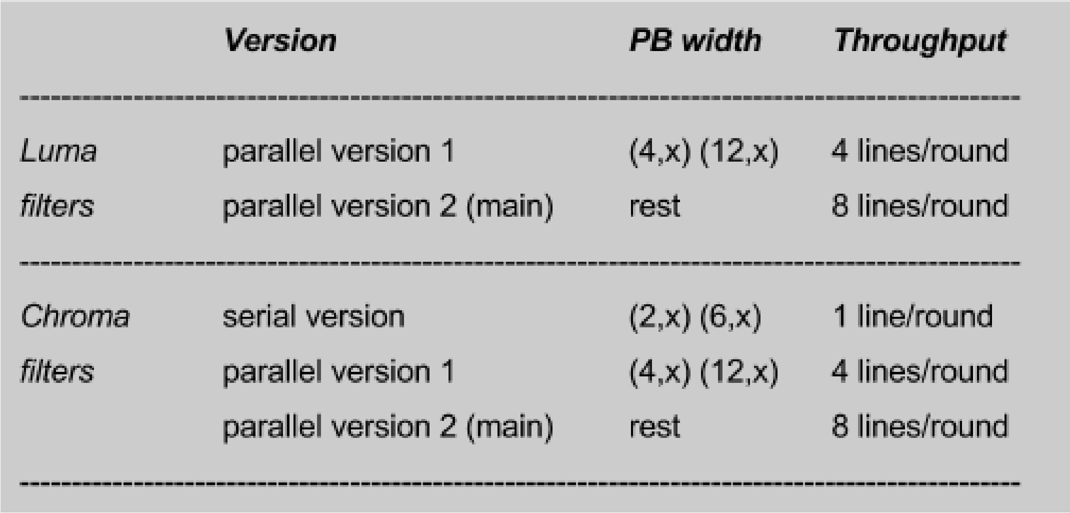
A PU consists of one MxM Luma PB, followed by 2 Chroma PBs of size M/2 x M/2 (for the case of YUV420 format). Since a CU’s minimum size is 8x8, luma PB width varies from 4 to 64 while chroma PB widths are from 2 to 32.

(**4**, 8), (4, 16),
(**8**, 4), (8, 8), (8,16), (8, 32),
(**12**, 16),
(**16**, 4), (16, 8), (16, 12), (16, 16), (16,32), (16, 64),
(**24**, 32),
(**32**, 8), (32, 16), (32, 24), (32, 32), (32,64),
(**48**, 64)
(**64**, 16), (64, 32), (64, 48), (64, 64)

PU sizes (width,height)

We are only interested in the width because it determines the throughput of the parallelized code, while height - corresponding to video lines - are processed in a for-loop and thus dictates the iterations of the algorithm. The optimized filters presented earlier can process 8 lines, thus providing 8 results per round, which can cover all PB sizes except (4,x) and (12,x). Separate versions of the functions, that handle these cases were created. Specifically for the (2,x) and (6,x) PB chroma sizes, which can be encountered rarely, we let the original standard-C version handle them. Finally, it should be noted that the most frequently used PU size is 8x8, which translates into 8x8 luma and 4x4 chroma PB sizes. These sizes were the basis for designing these algorithms.

To sum up, these are all the functions so far



	Version	PB width	Throughput
Luma filters	parallel version 1	(4,x) (12,x)	4 lines/round
	parallel version 2 (main)	rest	8 lines/round
Chroma filters	serial version	(2,x) (6,x)	1 line/round
	parallel version 1	(4,x) (12,x)	4 lines/round
	parallel version 2 (main)	rest	8 lines/round

Fig. 34 Different versions according to PB width - serial version refers to the original HM code

Code Optimizations

In this section we focus on optimizations that are not of data parallelization nature, but rather C++ and architecture specific.

1) Function Pointers

The HM reference code has a lot of control logic that uses if-else statements which aids code readability but impairs performance. Further if-statements were added for the selection of the different versions of the algorithms, we mentioned above. We decided to remove all if-statements by using function pointers and some simple hashing, in order to implement the function calling logic. This change alone brought a 1.28x speedup.

2) Input data analysis

HEVC supports profiles of both 8-bit and 10-bit pixel depth. For this reason, the input as well as the output data for every function of the reference code is 16-bit shorts. However, the *actual* data is not always 16-bit. When the first filter pass is applied, in order to calculate horizontal positions *a,b,* or *c*, or vertical positions *d,h,* or *n*, the actual input data is *8-bit* pixels. Only when a second filter pass is required (positions *e,f,g,i,j,k,p,q,r*) the actual input data is also 16-bit.

Two separate versions were created in order to take advantage of this fact. The first one, which is called during first filter pass, operates on 8-bit data and has a higher throughput than the second one, which operates on 16-bit data and is called during second filter pass. Horizontal filtering needs only the first version since it's always applied first, while Vertical filtering needs both.

3) Coefficient data analysis

i. Our motivation behind this analysis is that multiplications are costly, so special care has to be taken in order to avoid them.

i	*-3*	*-2*	*-1*	*0*	*1*	*2*	*3*	*4*
(¼) q[i]	**-1**	**4**	**-10**	**58**	**17**	**-5**	**1**	**0**
(½) h[i]	**-1**	**4**	**-11**	**40**	**40**	**-11**	**4**	**-1**
(¾) c[i]	**0**	**1**	**-5**	**17**	**58**	**-10**	**4**	**-1**

Fig. 35 Coefficient vectors

By observing the coefficient vectors for each position we can simplify the code and save clock cycles.

For example the following line, which requires 2 muls + 1 add

```
     q[-3] * line[-3] + q[-2] * line[-2]
```

can be transformed into

```
     (line[-2]<<2) - line[-3]
```

which requires 1 shift and 1 sub and thus saving clock cycles.

The quarter pixel position calculation can be rearranged like this:

```
res = ((row[-2] - row[2])<<2)  - row[2]  + row[3] - row[-3]
-10*row[-1]  + 58*row[0]  + 17*row[1]
```

(q[4] is 0 which means we don't need to load the last line)

For the half pixel position we have the following simplification:

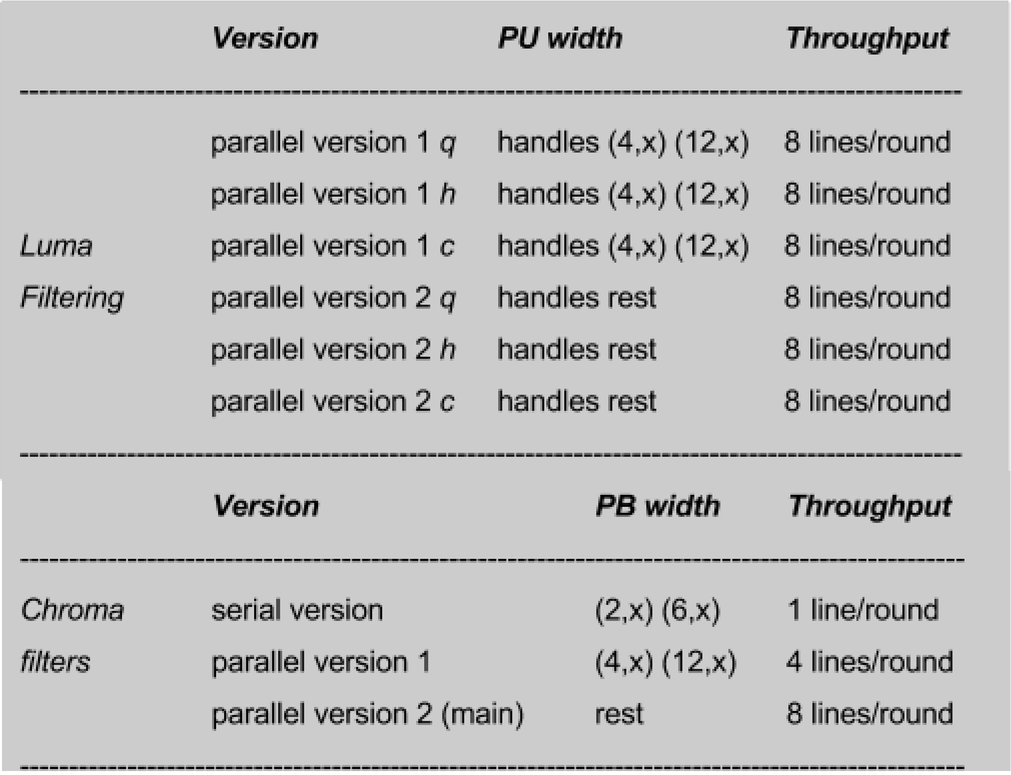
```
res = ((row[-2] + row[3])<<2) - (row[-3] + row[4])
      - 11*(row[-1] + row[2]) + 40*(row[0] + row[1])
```

The ¾ pixel position vector is the ¼ vector reflected.

ii. Next, we noticed that despite the fact that coefficient data is stored in 16 bit shorts, the actual data doesn't need more than 8bits. For example, the largest filter coefficient is +58 and can be represented with only 7 bits. The sum of all the coefficients is always +64 which needs 8 bits. This means that a total of 16 bits are needed to represent the final result and although SSE2 instructions operate on 16-bit input data, the output can also be 16-bit. Thus, we maintain the throughput that a 128-bit SSE2 register can provide for 16-bit data, which is 8.

iii. Finally, we eliminated function parameters such as coefficient arrays and booleans, which were passed between functions in an unnecessary fashion. We created global structures, that contained the coefficient data, and further stored them in aligned memory to accelerate loading from memory to registers.

Each optimized *luma* filter now has 3 separate versions, one for each position (quarter pixel, half pixel, ¾ pixel). No alteration was needed for the Chroma filters. In total we have: (both Horizontal and Vertical filters have the same version-structure)



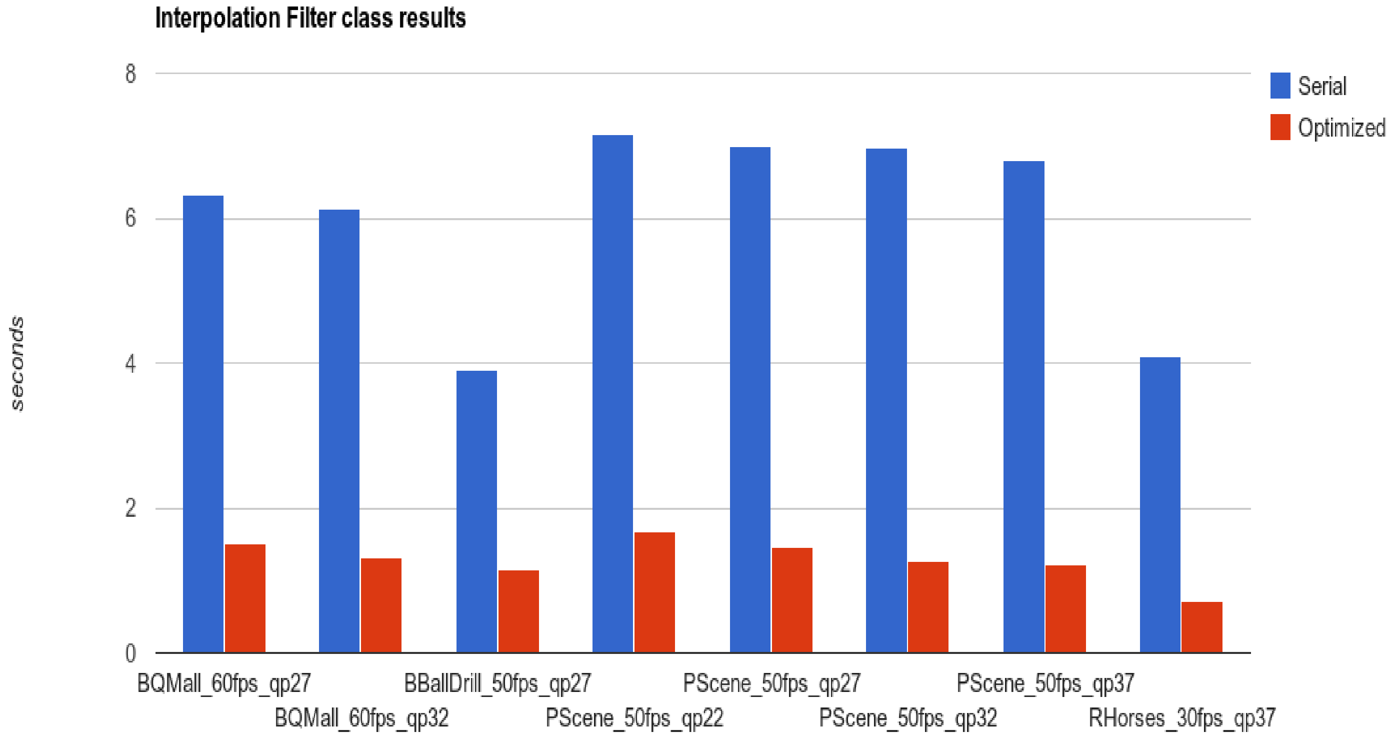
	Version	PU width	Throughput
Luma Filtering	parallel version 1 *q*	handles (4,x) (12,x)	8 lines/round
	parallel version 1 *h*	handles (4,x) (12,x)	8 lines/round
	parallel version 1 *c*	handles (4,x) (12,x)	8 lines/round
	parallel version 2 *q*	handles rest	8 lines/round
	parallel version 2 *h*	handles rest	8 lines/round
	parallel version 2 *c*	handles rest	8 lines/round

	Version	PB width	Throughput
Chroma filters	serial version	(2,x) (6,x)	1 line/round
	parallel version 1	(4,x) (12,x)	4 lines/round
	parallel version 2 (main)	rest	8 lines/round

Fig. 36 Different versions of the final optimized code according to PB width and sample position. For example parallel version 1q calculates ¼ positions for PB widths of 4 and 12. Serial refers to the original version.

Profiling Results

We profiled the available test streams and the result showed speedups from 3.5x up to 5.6x. Detailed results follow.



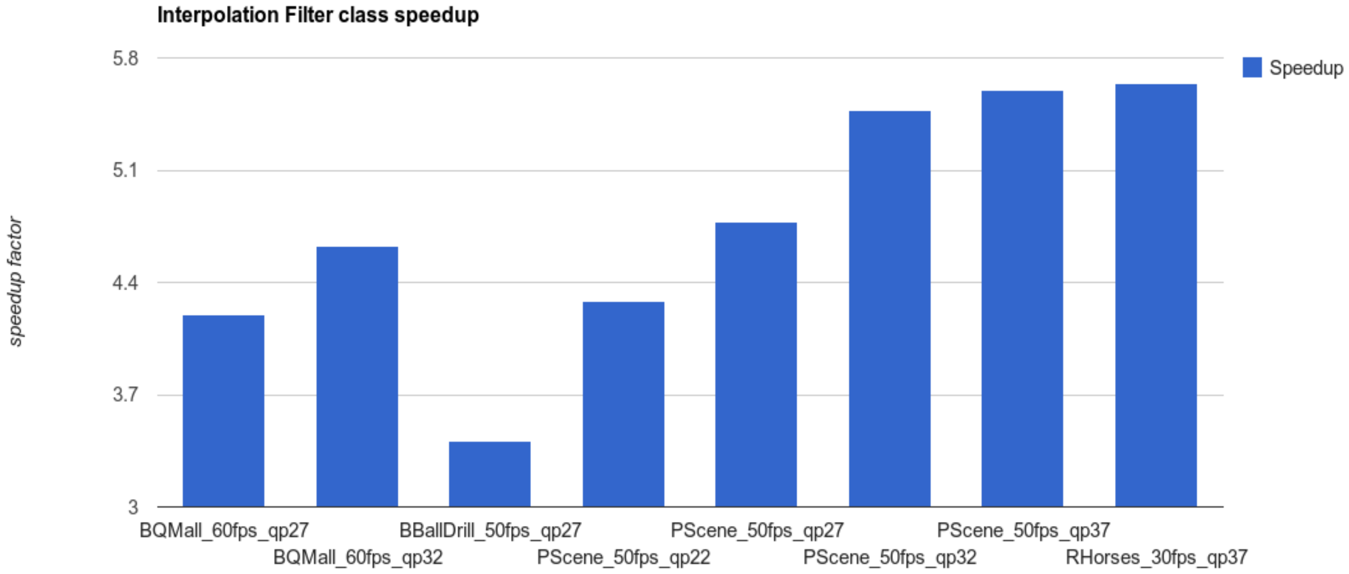
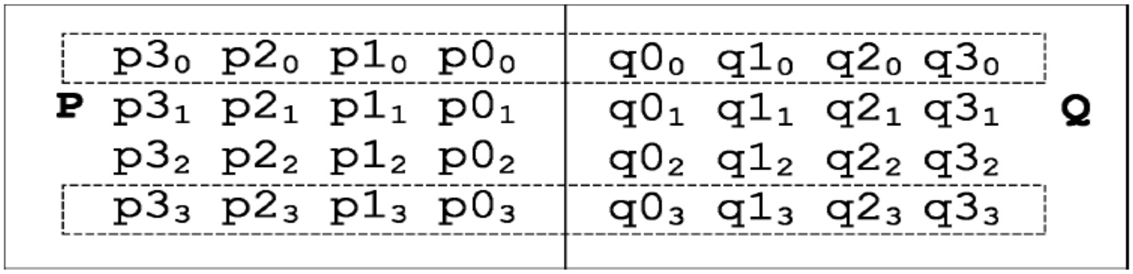


Fig. 37 Profiling results for the Interpolation filter class

Optimized Deblocking filter (DBF)

Introduction

The DBF consists of two steps and operates only on 8x8 blocks as follows. For each four-pixel long boundary (there are 8 such boundaries in a 8x8 block) a decision must be made about the strength of the applied filter (fig. 38), making this the decision-making step. After that, filtering is applied to a 4x8 area (4 rows of 4 pixels left and right from the boundary) for vertical edges or to an 8x4 area (4 columns of 4 pixels above and below the boundary) for horizontal edges. Depending on the chosen strength, different amount of pixels are altered (fig. 39). Zero filter strength means no filtering is required. A filter strength of 1 means the filter alters two of the 8 pixels involved but if further conditions are met, the total amount of pixels affected can be 3,4 or 6 (strong filtering).



	$p3_0$	$p2_0$	$p1_0$	$p0_0$	$q0_0$	$q1_0$	$q2_0$	$q3_0$	
P	$p3_1$	$p2_1$	$p1_1$	$p0_1$	$q0_1$	$q1_1$	$q2_1$	$q3_1$	Q
	$p3_2$	$p2_2$	$p1_2$	$p0_2$	$q0_2$	$q1_2$	$q2_2$	$q3_2$	
	$p3_3$	$p2_3$	$p1_3$	$p0_3$	$q0_3$	$q1_3$	$q2_3$	$q3_3$	

Fig 38. Vertical edge filter strength decision making - only the dashed lines are used for decision making. Filtering is applied to all four lines.

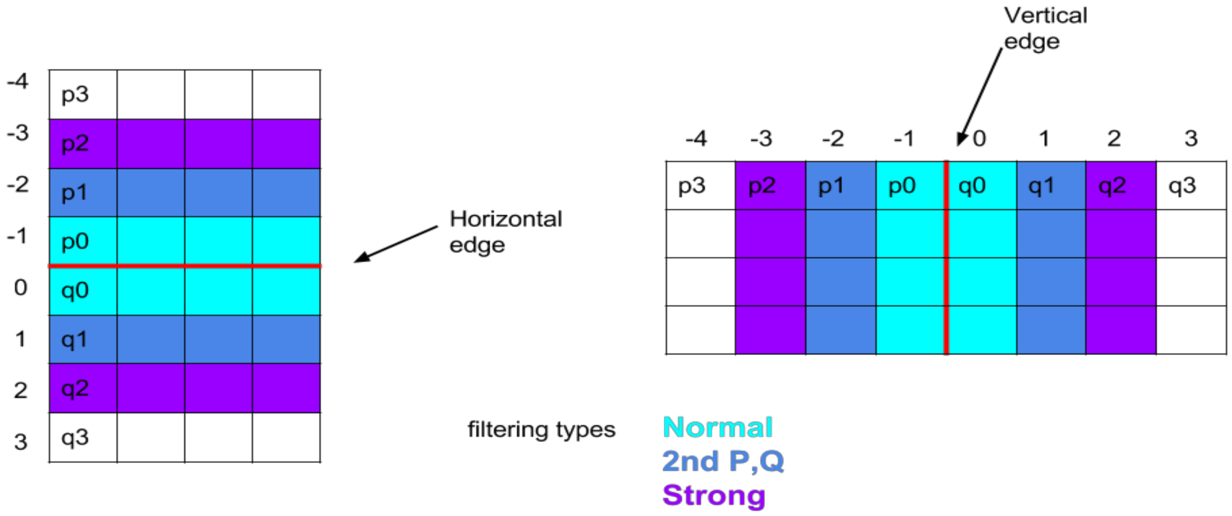


Fig 39. Types of applied filtering (pixels of the same color are processed in parallel)

Normal:	*2 pixels (p0,q0) per line*
2ndP:	*3 pixels (p0,q0,p1) per line*
2ndQ :	*3 pixels (p0,q0,q1) per line*
Strong :	*6 pixels (p2,p1,p0,q0,q1,q2) per line*

Optimizations

The DBF decision-making part is heavily control-based so it's hard to parallelize for multiple boundaries. On the other hand the filter application step is more suitable, thus we concentrated on that.

Horizontal Edge Filtering

Horizontal edge filtering resembles Vertical interpolation filtering from the previous section, which means we can use the same approach, modifying it for 4 lines accordingly. In order to minimize the heavily control logic code we created a separate function for each filter strength, combined with function pointers to call the appropriate function as needed. Fig. 39 above shows how pixels of the same color can be processed in parallel.

Vertical Edge Filtering

On the other hand, the Vertical Edge filter, is not parallel-friendly because the calculations are specific for each line. To deal with this we transpose each 8x4 block to a 4x8 one, apply Horizontal edge filtering, and then transpose it back.

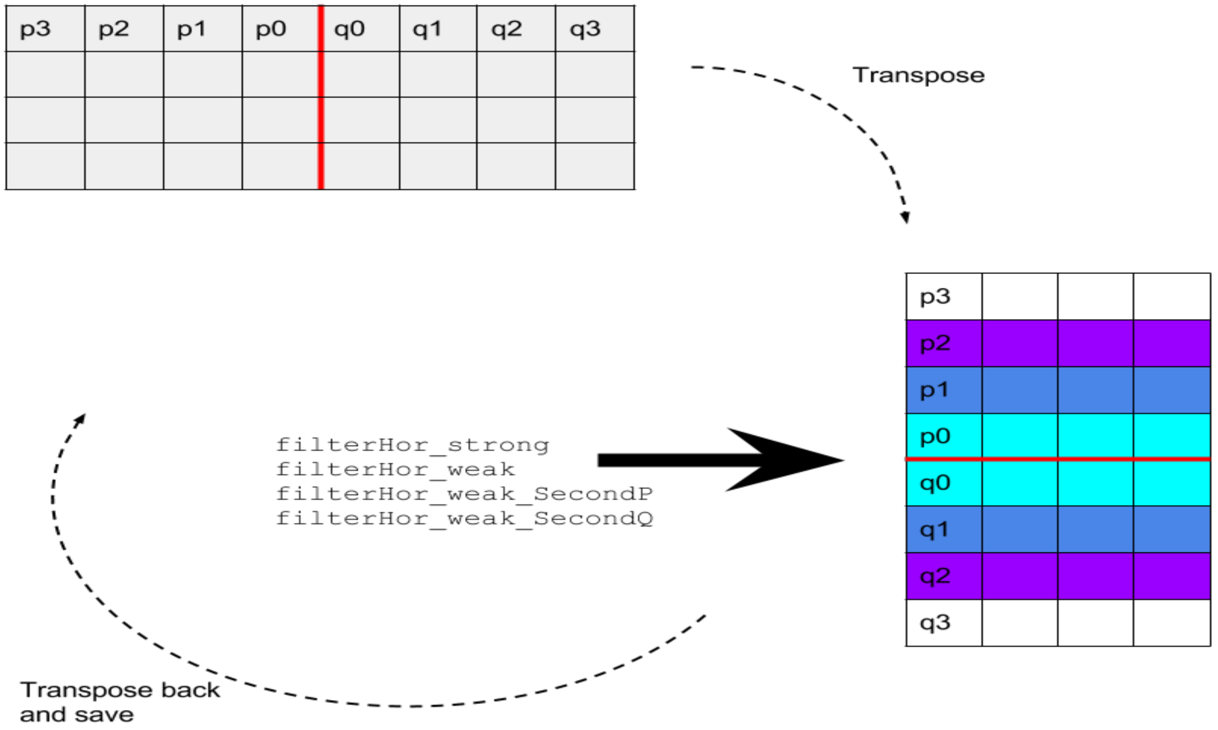


Fig. 40 Optimized Vertical Edge filtering

Finally, the 4x8 area of the filter application hinders the potential processing of 8 samples per round of calculations, that can be achieved with SSE2 instructions. We tackle this by applying the filter on a combined 8x8 block whenever two subsequent boundaries require the same filter strength.

Results

Profiling shows a small overall speedup, emphasizing that heavy control logic in the code is the actual bottleneck.

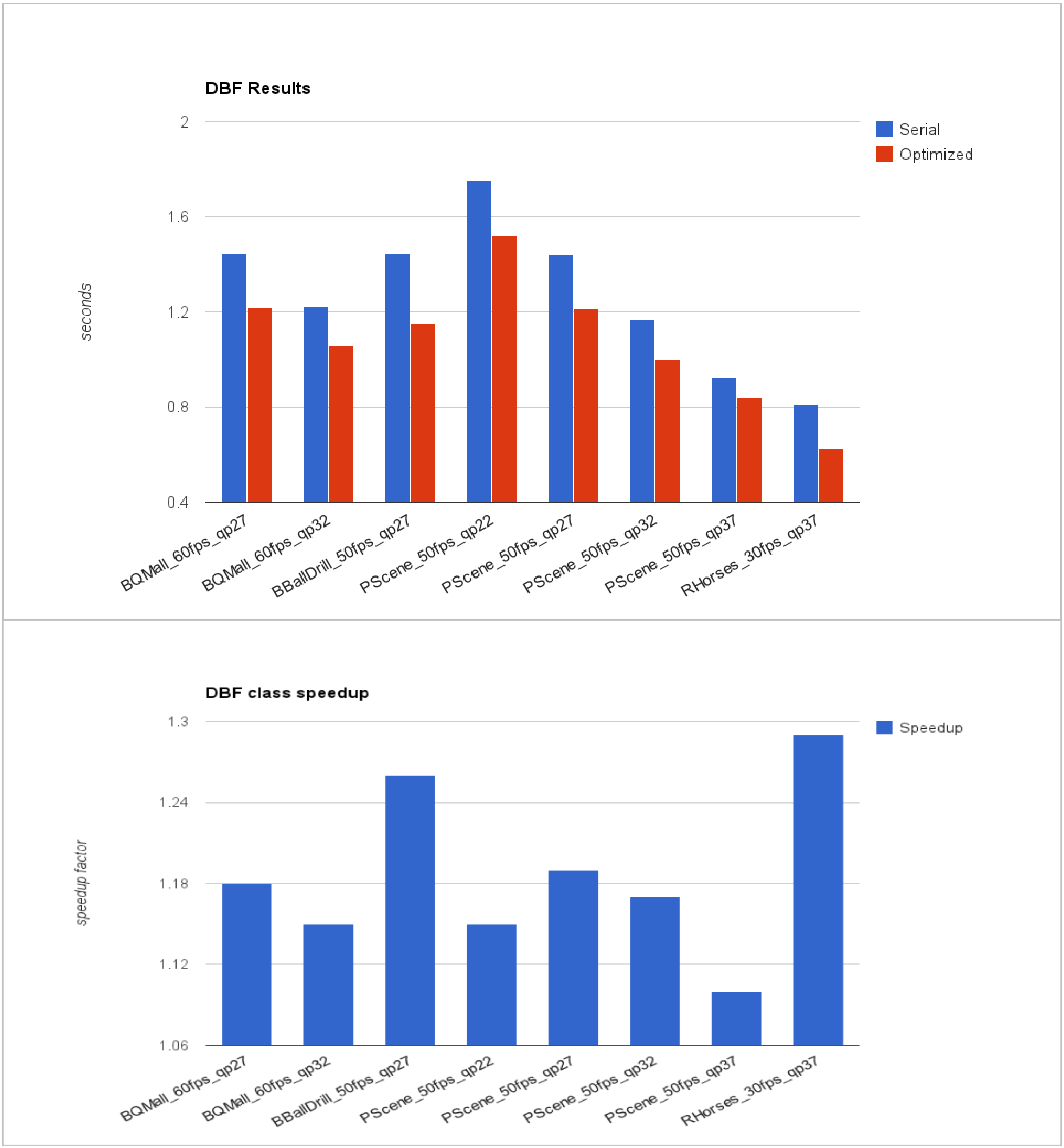
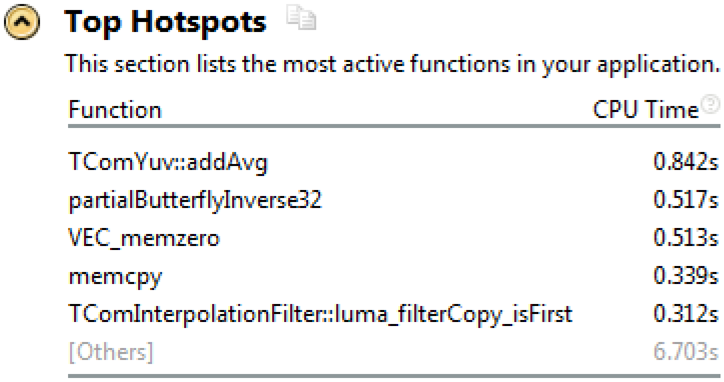


Fig. 41 Profiling results for the Deblocking filter class

Further optimizations

Misc functions

Profiling the HM decoder again, as optimized with the Interpolation and Deblocking results presented earlier, revealed new hotspots: functions that deal with embarrassingly parallel workload and take a small, but considerable nonetheless, piece of the CPU computational pie.



Top Hotspots

This section lists the most active functions in your application.

Function	CPU Time
TComYuv::addAvg	0.842s
partialButterflyInverse32	0.517s
VEC_memzero	0.513s
memcpy	0.339s
TComInterpolationFilter::luma_filterCopy_isFirst	0.312s
[Others]	6.703s

Fig 42: Profile result after interpolation and deblocking optimizations

TComYuv::addClipCluma, TComYuv::addClipChroma, are used for clipping pixel values in their legal values (usually [0,255] for 8-bit video) after all pixel processing is concluded. TComYuv::addAvg computes the average of two pictures (`Y[x] = (srcA[x] + srcB[x])/2`) while partialButterflyInverse32 is used for the inverse Transform stage. This function basically implements matrix multiplication.

Microsoft Visual Studio

Finally we tweaked some Visual Studio configuration parameters which provided a further decrease in execution time. First, we switched from 32-bit architecture to 64-bit. This is known to produce better results, since it allows for double the number of available registers to the compiler. We also configured the linker to eliminate data or functions that are never referenced from the final executable, mainly because the HM code contained inline functions. The graph below shows the added gain from these optimizations.

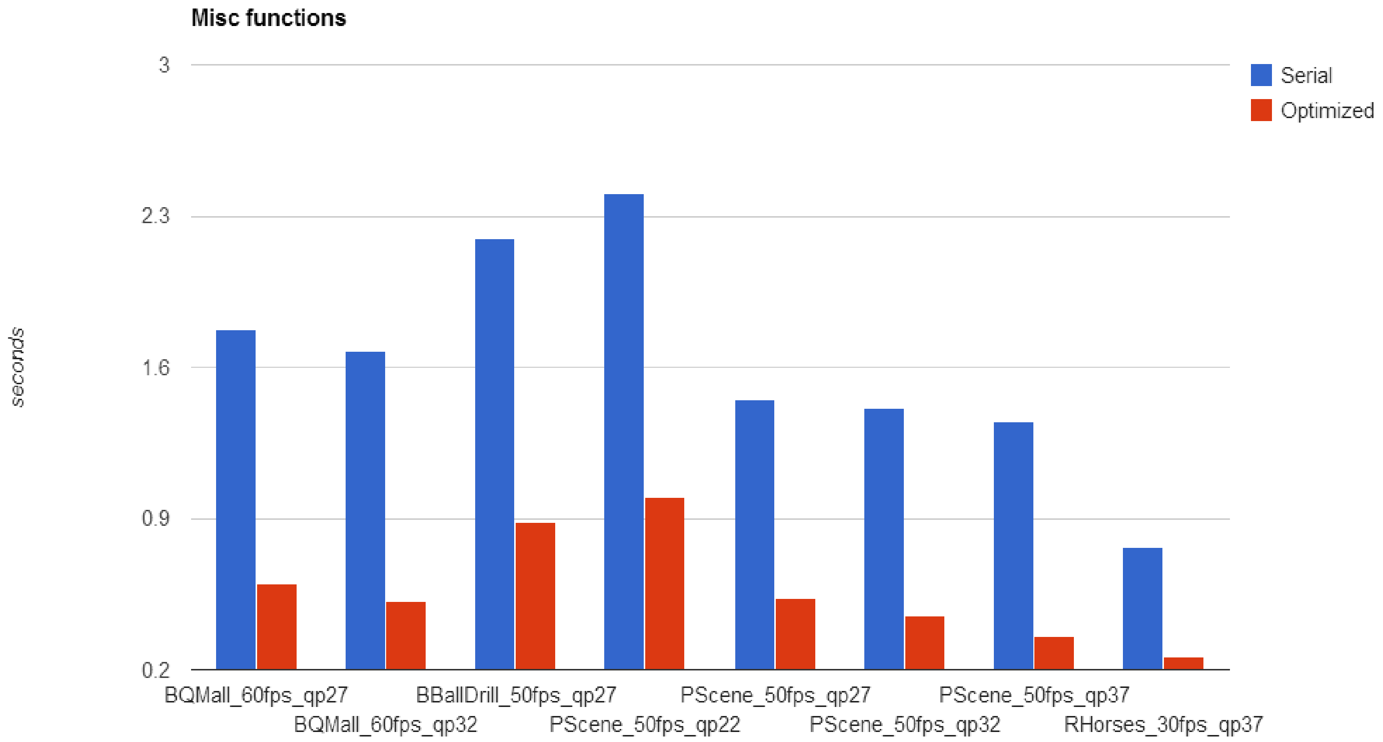


Fig 43: Result with new Visual Studio parameters after optimizing functions TComYuv::addClipCluma, TComYuv::addClipChroma, TComYuv::addAvg partialButterflyInverse32

Final Results

We compare the optimized HM 10.0 decoder with the original one and show a total speedup of up to 2.15x, thus enabling most of the reference bitstreams to be decoded in real time. While the original decoder achieved RT decoding for 7, our optimized version achieves RT for 17 out of the 20 official bitstreams.

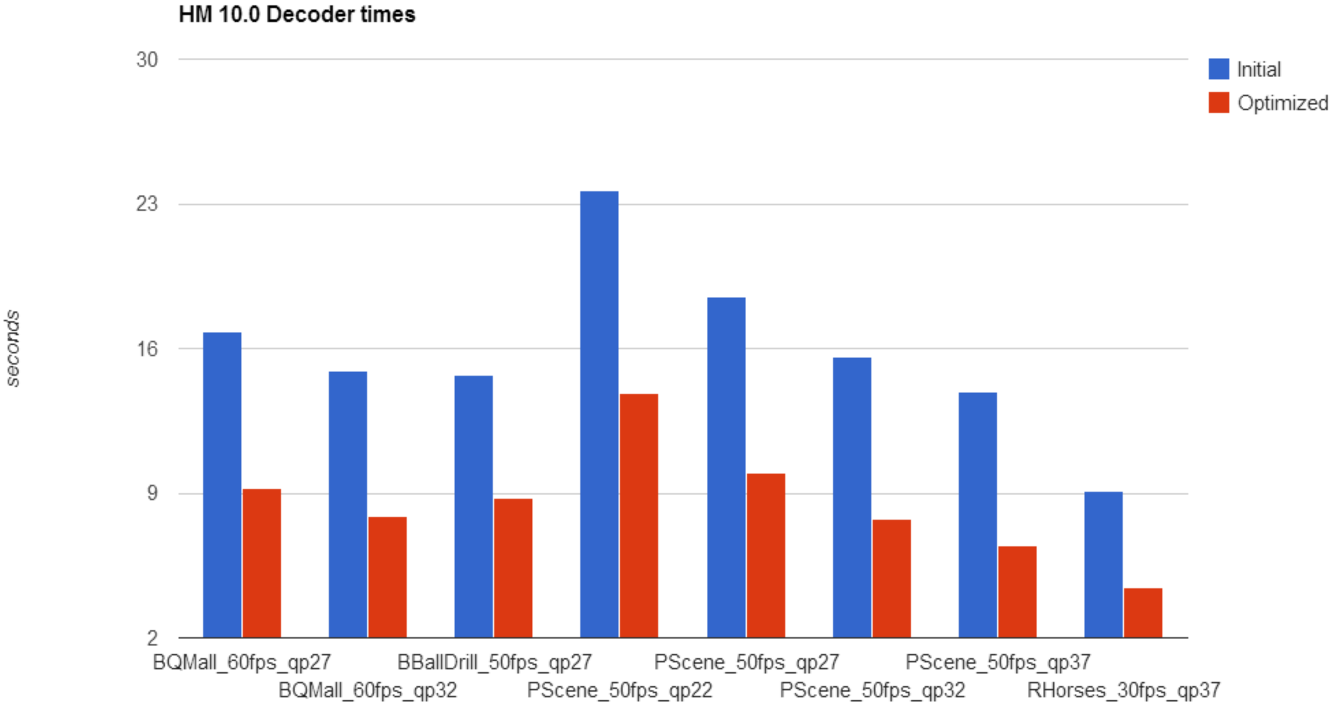


Fig. 44 Profiling of the optimized HM 10.0 decoder

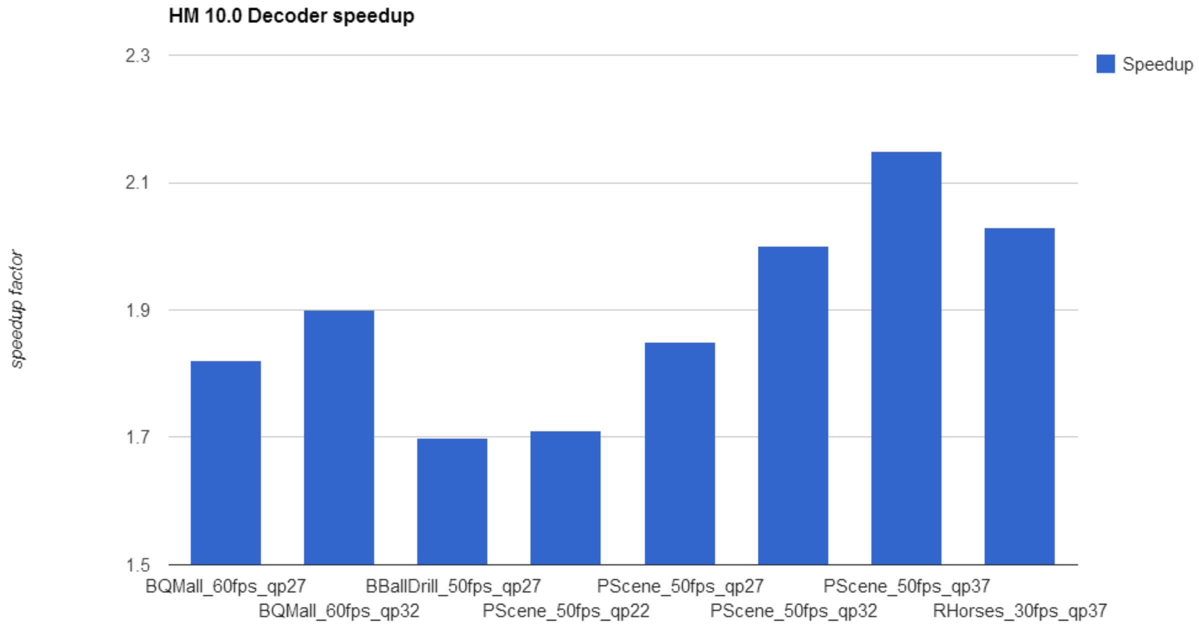


Fig. 45 Speedup gained from the optimized HM 10.0 decoder

Future work

832x480 is an ideal resolution for mobile devices. We plan to further test the optimized decoder on Android platforms and compare the gained speedups. To achieve that we plan to replace SSE2 instructions with their NEON counterparts, make key parts of the code multi-threaded and finally use a renderer.

The ground has been set as we cross-compiled our optimized code to the Android x86 platform using the Android NDK standalone toolchain with SSE2 support. We also cross-compiled the original code to Android with ARM architecture.

Summary - Conclusions

Use of codecs in the industry is major, as explained by a recent study, which states that 53% of the Internet's traffic is video. Available HM reference source codes (HM versions 1.0 through 13.0) allow studying and optimizing a complex software engineering product - a video codec- , which is interesting for two reasons: First, in an academic way, because it combines elements of different fields such as Signals & Systems, Computer architecture, High performance computing (HPC) and finally C programming. Secondly because it's considered a hot topic both industry and research-wise and results show that HEVC is here to stay.

However, understanding the HM reference code, although well written, requires a solid background in video encoding and C/C++ programming. The HM's developers used C++ which aids code-readability since mapping different video components is more intuitive with object oriented programming. Furthermore, C programmers without any C++ experience can easily understand the code but will need deeper C++ knowledge, should optimizations need to be made. Documentation is sufficient.

Coping with such a complex and large project is impossible without the use of IDE tools, debugger and profiler. Tools such as Microsoft Visual Studio and Intel Vtune Amplifier besides offering increased productivity, aided (if not allowed) debugging and optimizing the source code.

In terms of optimization techniques for the HM 10.0 decoder, or a video codec in general, data parallelism should be the method of choice, since algorithms benefit from it greatly and both hardware and software support for SIMD instructions exists in any platform. Specifically, Intel's C++ intrinsics, which provide wrappers for all versions (1,2,3 and 4) of SSE assembly instructions, are a great tool for software developers. Multi-tasking and C/C++ -specific code modifications should be the follow up approaches for codec optimization. All these techniques combined guarantee a decrease in execution time.

References

[1] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the High Efficiency Video Coding (HEVC) Standard, *IEEE Transactions on Circuits and Systems for Video Technology*, December 2012

[2] Frank Bossen, Member, Benjamin Bross, Karsten Suhring and David Flynn. HEVC Complexity and Implementation Analysis, *IEEE Transactions on Circuits and Systems for Video Technology*, December 2012

[3] Hao Lv, Ronggang Wang, Jie Wan, Huizhu Jia, Xiaodong Xie and Wen Gao
An Efficient NEON-based Quarter-pel Interpolation Method for HEVC

[4] Andrey Norkin, Gisle Bjøntegaard, Arild Fuldseth, Matthias Narroschke, Masaru Ikeda, Kenneth Andersson, Minhua Zhou, and Geert Van der Auwera. HEVC Deblocking Filter, *IEEE Transactions on Circuits and Systems for Video Technology*, December 2012

[5] Leju Yan, Yizhou Duan, Jun Sun, Zongming Guo. Implementation of the HEVC decoder on x86 processors with SIMD optimization.

[6] Chi Ching Chi · Mauricio Alvarez-Mesa · Jan Lucas · Ben Juurlink · Thomas Schierl. Parallel HEVC Decoding on Multi- and Many-core Architectures A Power and Performance Analysis

[7] ITU-T H.265 TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU (04/2013) SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services – Coding of moving video

[8] http://forum.doom9.org/showthread.php?t=167081 Hardware codec genius

[9] Intel® C++ Intrinsic Reference Document Number: 312482-003US

[10] Athanasios Leontaris, Alexis M. Tourapis. Weighted prediction methods for improved motion compensation