2018

Fotis E. Alexakos

[Evaluation of multiple slices and tiles in HEVC video encoding]

Supervisor: Dr. Maria G. Koziri, Computer Science Department, University of Thessaly, Lamia

«Υπεύθυνη Δήλωση μη λογοκλοπής και ανάληψης προσωπικής ευθύνης»

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, και γνωρίζοντας τις συνέπειες της λογοκλοπής, δηλώνω υπεύθυνα και ενυπογράφως ότι η παρούσα εργασία με τίτλο «*Evaluation of multiple slices and tiles in HEVC video encoding*» αποτελεί προϊόν αυστηρά προσωπικής εργασίας και όλες οι πηγές από τις οποίες χρησιμοποίησα δεδομένα, ιδέες, φράσεις, προτάσεις ή λέξεις, είτε επακριβώς (όπως υπάρχουν στο πρωτότυπο ή μεταφρασμένες) είτε με παράφραση, έχουν δηλωθεί κατάλληλα και ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή (κατά το πρότυπο IEEE 2006). Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

25 Ιουνίου 2018

Ο ΔΗΛΩΝ

Φώτης Ε. Αλεξάκος

“Affirmation of no plagiarism and responsibility assumption”

Having full awareness of copyright laws and plagiarism consequences, I hereby responsibly declare and sign that this present work bearing the title “*Evaluation of multiple slices and tiles in HEVC video encoding*” is strictly the fruit of my own personal labour. Also, every source of information, ideas or wording used (either as they are or with edits) is clearly referenced appropriately in the text, while a full list of the above mentioned sources is to be found at the end of this work (IEEE 2006 style). Finally, I, the undersigned, fully assume all the legal and administrative consequences that might arise in the case that this work or parts of it will be proved to infringe copyright or be a product of plagiarism, either today, or in the future.

June 25th, 2018

Sincerely,

Fotis E. Alexakos

Contents

List of figures

List of Tables

Abstract

The ever growing demand for even better and better video quality have driven the development of sophisticated video coding techniques with High Efficiency Video Coding (HEVC) being the most recent standard. HEVC (known also as H.265) provides for great compression without perceivable loss in image quality. This standard is about to replace its predecessor H.264, widely known as MPEG-4. Yet, this efficiency of H.265 comes at the cost of a (much) increased coding complexity algorithm. In other words, coding time mainly, but also decoding time are higher.

In order to deal with the above issue, H.265 standard is designed so as to promote code parallelization at a high grade. Thus, one can choose between the following two high level parallelization methods:

i. Frame segmentation in slices or/and tiles and
ii. Wavefront parallelization

Purpose of this thesis is to evaluate various segmentation ways, trying to conclude about the best segmentation in slices and tiles in order to achieve the most efficient parallelization keeping the highest coding efficiency possible.

Structure of this essay is as follows: First, we present shortly and briefly compare the most common video coding standards that have been developed until today. Next, we delve into HEVC with reference to various slicing and tiling methods used to improve its performance via parallelization strategies. We continue presenting HEVC Reference Software and the Test Model [1], [2] we used for our tests and experiments. Finally, we describe in details several "runs" of HEVC encoder whose output naturally leads to specific conclusions.

CHAPTER 1: Brief Introduction to Video Coding

Preliminaries

Traditionally, digital video has always been the most space demanding Computer Science application. This is because, a movie is in fact too much information in the form of thousands and thousands images presented as "frames". Each one of these frames needs several bytes to be described depending on its resolution (in pixels), colour bit depth and so on. Also, people's demand concerning video quality has grown significantly lately. Thus, having started with –say- CGA [3] resolutions of 320x200 pixels with 4-bit colour depth back in the '80s, we tend to use High Definition(HD) and Ultra High Definition(UHD) Video of such resolutions as 3840x2160 and 10-bits colours, or even 4K and 8K standards etc. Moreover, increased traffic caused by applications such as video-on-demand, video apps for mobile devices (smartphones, tablets) and so on, impose severe challenge on today's networks. Also, there is increased desire for higher quality and resolutions in mobile applications [4].

As a result, there was an early need for algorithms to compress video so as to reduce needed storage capacity or/and bandwidth and transmission time accordingly. Today, in the ICT business, we distinguish between two types of data compression algorithms: Lossy and lossless. According to Wikipedia, the term **lossy** or **irreversible** compression describes algorithms that encode data either discarding part of them or by using inexact approximations to convey the content [5]. On the other hand, the term **lossless compression** refers to techniques that encode data in a way that they can be decompressed precisely to their original form and size; while we would get an approximation only of the prototype data in case lossy compression were chosen. Of course this "approximation" used in lossy compression strategies leads to higher compression rates (i.e. smaller files). [6]

Run Length Encoding (RLE) and Lempel-Ziv-Welch (LZW) are *lossless* compression examples, with H.264 and H.265 being *lossy* compression algorithms. Now, Video Compression is an application of data compression and its objective is to remove redundant information from a video and to omit those parts of the video that will not be noticed by a human eye [7].

Video Coding standards evolution

Typically video compression algorithms are based on the fact that most pixels in a frame are highly correlated with others in the same frame or adjacent (previous or next) ones. Therefore, we can reduce the amount of data required to represent a video by removing any redundancy inside the frame (intra-frame) or in-between them (inter-frame) [7].

We can broadly classify redundancy in a video as follows:

- **Spatial redundancy:** Or Intra-Frame Redundancy. This term is attributed to redundant information existing within the frame. Since a video frame is simply a picture that can be independently processed, we can remove such redundancies by applying various digital image compression algorithms on each frame.
- **Temporal redundancy:** It is natural for frames that are captured within hundredths of a second to be highly correlated. In other words: They are adjacent (in time) and present extremely few differences. This is called Temporal or Inter-Frame redundancy. [7]

An encoder uses intra-prediction methods to eliminate spatial redundancy and inter- to remove temporal one. In order to optimize predictions, several settings have to be applied on the encoder. In general, we divide frames in three categories: I(Intra), P(Predicted) and B(Bidirectional). For P and B frames, both inter- and intra-predictions can be used, while we apply only intra- for I frames. (The prefix '*inter*' has the meaning of ‘*between frames*’ while ‘*intra*’ stands for ‘*in a frame*’). Also, a P frame is predicted from a reference one preceeding it, while a B frame can be either predicted from a preceding or subsequent one. For inter-prediction purposes, the encoder reorders frames as they arrive. [8].

Nowadays, any modern video compression standard uses similar basic steps to encode a video [7]:

1) Divide each frame of the video into blocks of pixels
2) Identify and remove spatial redundancies within each frame
3) Exploit temporal redundancies between the adjacent frames and remove those redundancies.
4) Identify and remove the remaining spatial redundancies using quantization, transformation and entropy encoding.

We will see more about the above techniques in the discussion about the HEVC standard [9].

We are going to discuss about various Video Coding standards here. Most of them have appeared since the early ‘90s and were developed by the **ITU-T** and **ISO** organizations about which we talk below. The following figure shows their evolution until today.

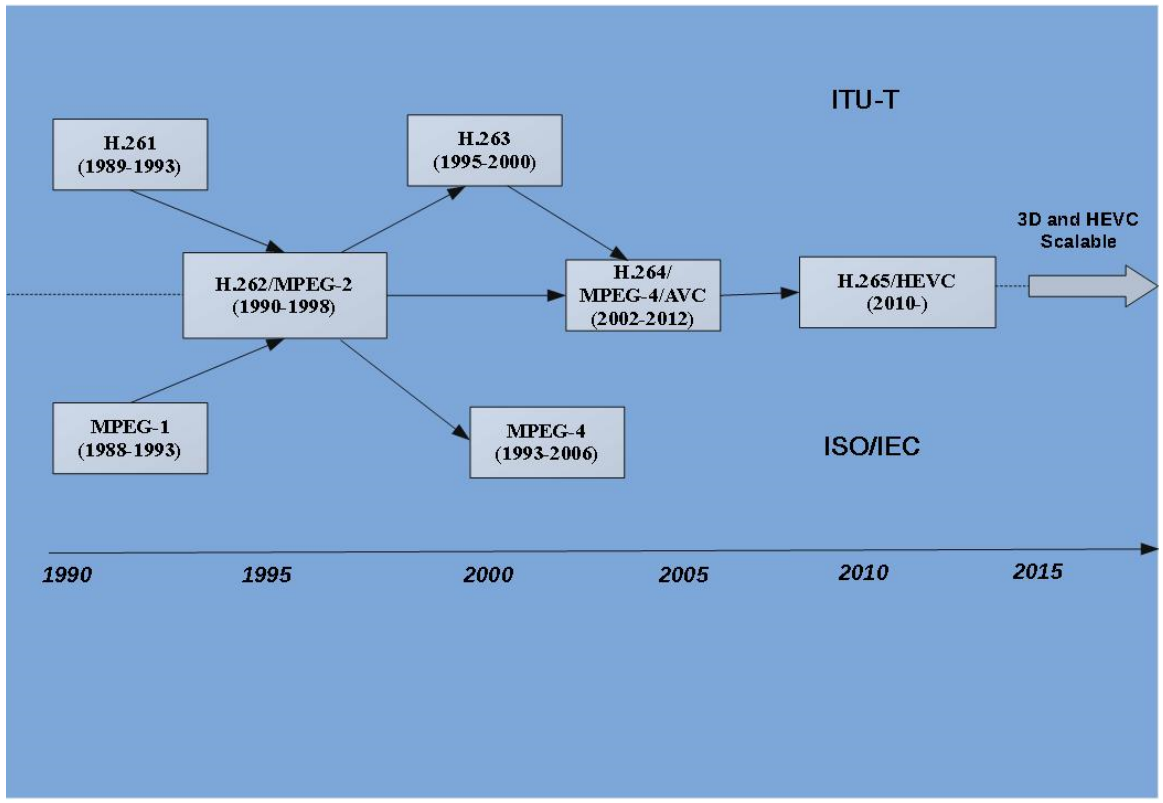


Figure 1: Evolution of Video Coding Standards

ITU-T is the Telecommunication Standardization Sector of ITU which in turn stands for International Telecommunication Union. ISO is the acronym of the

International Organization for Standardization. ITU-T is also known as Video Coding Experts Group (VCEG) while ISO can also be found as Moving Picture Experts Group (MPEG). ITU-T since its formation in 1997 aims at maintaining prior video coding standards and developing modern ones, as today there is high demand for moving pictures services, either conversational and non-conversational. On the other hand, VCEG was founded earlier (in 1988) to provide for video and audio coding standards to serve applications such as video distribution and digital storage media. In 2001, both organizations merged to the Joint Video Team (JVT) to develop a new International Standard (Recommendation) today known as the H.264 Recommendation/MPEG-4 part 10 standard [10].

Digest of a Video Coding Standards comparison

H.261

The first video coding standard developed by ITU-T immediately after its formation was the H.261 one. Actually, this came to be just the first member of a whole codecs family under the naming standard of H.26*x*. It was also the first well spread and used standard, i.e. with major support. At first it aimed to serve the need for video transmission over ISDN lines supporting one or more 64Kbps channels. In fact, the standard only describes the decoder allowing the encoder design to use any motion compensation method as long as the output could be handled properly by the decoder. In any case, a 16x16 block called *macroblock* is the main processing unit. Inter- and intra-predition is supported while the 8x8 Discrete Cosine Transformation(DCT) was introduced followed by rounding the coefficients (scalar quantization). Consequently, they are scanned in a zig-zag run and coded (variable length coded is supported) to remove redundant information. Any international video coding standard which has been introduced since then is closely based on the same mechanisms. [10]

MPEG-1

MPEG-1 was developed by the homonymous Group during 1993 in order to cater for the compression of VHS digital video and video CDs. It supports input sources with resolutions of 352x288 (PAL) or 352x240 (NTSC) processed at 1.5Mbps. At higher bit rates MPEG-1 provides better video quality than H.261. [10]

H.262/MPEG-2

The successor of the above standards were H.262, also known as MPEG-2. It was developed by ITU-T and MPEG together in 1992. It outclasses MPEG-1 as it supports interlaced video (used in older TV systems) and offers better performance at bitrates greater or equal to 3Mbps. Backwards compatibility with MPEG-1 is also supported for consistency purposes. This means that an MPEG-2 player can decode both MPEG-2 and MPEG-1 videos. [10]

H.263, H.263+ and H.263++

The descendant of H.262 standard was naturally H.263 and its enhancements (versions 2 and 3). It was developed (ver. 1) in 1995 by ITU-T. H.263 is quite efficient in conferencing as it offers double quality at any bit rate compared to its predecessors. Compared to H.261 it supports:

- DCT coefficients are coded using 3-D variable length code
- Bi-directional prediction
- Arithmetic entropy coding.
- Median motion vector prediction

H.263+ is another name for the version 2 of H.263 which was presented at early 1998. This update offered support for features such as flexible and custom video formats, error robustness and Supplemental Enhancement Information(SEI). Finally, in 2000 an H.263++ version (or version 3) was released that supported an improved compression efficiency over H.263, better picture quality, packet loss concern, even more resilience to errors and additional SEI [10].

MPEG-4

MPEG-4 standard was developed by MPEG (Moving Picture Experts Group) in late 1998. It acquired the formal International Standard Status of MPEG-4 ver. 2 at the very beginning of the millennium. This standard supports various applications which can be surveillance cameras with poor resolutions or HDTV broadcasting and DVDs. MPEG-4 Part 2 has about 21 profiles. Some of these sophisticated profiles are:

- ✓ Simple Face Animation
- ✓ Simple FBA
- ✓ Scalable Texture
- ✓ Advanced Core
- ✓ N-Bit
- ✓ Hybrid
- ✓ Advanced Coding Efficiency
- ✓ Advanced Real Time Simple [10].

H.264/MPEG-4 Part 10/AVC

H.264/MPEG-4 AVC released in 2003 is a joint project done by ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). These standards show significant improvement in intra coding and inter coding efficiency. It presents enhanced error robustness, and increased flexibility. It has efficient motion compensation and reduced bit-rate. Different block sizes are used for performing motion compensation which results in better video quality. The basic processing unit is 16x16 pixel macro blocks. The two entropy encoding methods used are CAVLC and CABAC. For all syntax elements, Context-Adaptive Variable-Length Coding (CAVLC) uses a single codeword set. RLE is used to code the transformation coefficient. In Context-Adaptive Binary Arithmetic coding, information entropy (from symbols coded in the near past) is exploited for encoding. It also uses arithmetic coding for transmission.

Some of the current applications for AVC are: Internet Video (.mp4 files), High Definition TV, Video Conference etc [7].

In comparison to prior video coding standards, H.264 saves almost half the bit rate while increasing greatly the compression percentage. This standard supports organization of coded info plus flexibility in coding and thus can increase resilience to errors. One have to notice though that the increased coding efficiency and flexibility suffers a penalty of increased complexity (i.e. execution time) compared with older standards [10]. The same seems to be valid for the most contemporary video coding standard –H.265- described in the pages that follow.

CHAPTER 2: Overview of the HEVC standard

Introduction

In January 2010 the ITU-T/VCEG and MPEG organizations jointly issued a Call for Proposals (CfP) that led to the formal launch of the HEVC project. Since a project to create another video coding standard that would offer the best compression-to-quality ratio ever was quite bold, both ITU-T and MPEG had already studied its feasibility. Today the project outcome is formally standardized as ITU-T Recommendation H.265 or MPEG-H part 2. Although, there was a first version of HEVC available from the beginning of 2013 still, the new standard was not defined officially until April 2013 [9].

Features of H.265

The figure below illustrates the architecture of the HEVC transcoder. We present briefly H.265 features in the following paragraphs, based on [4] and [11].

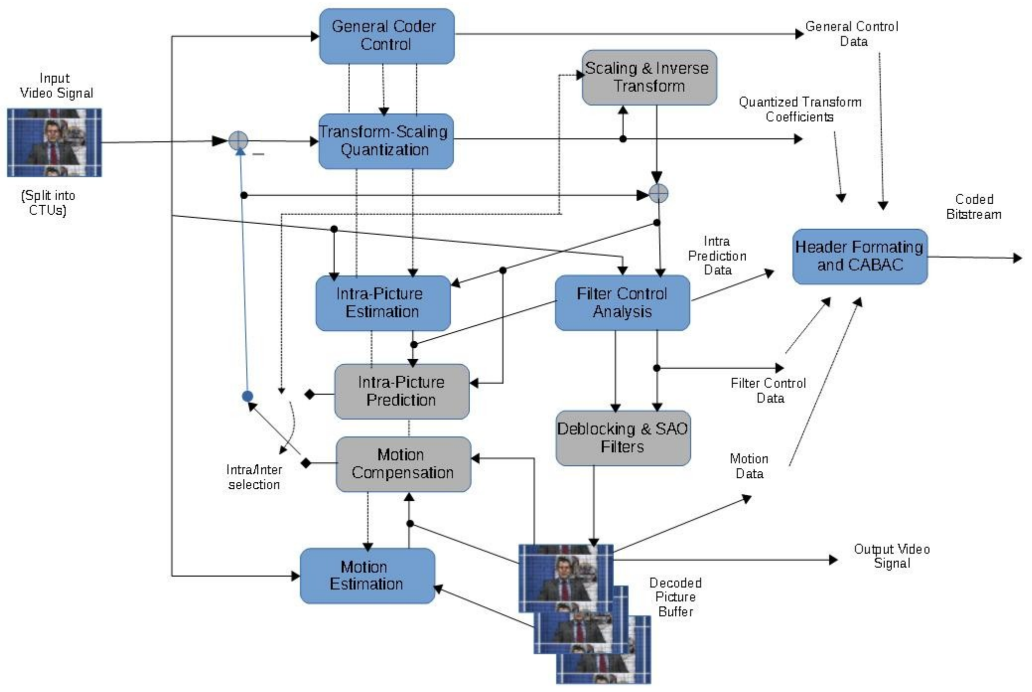


Figure 2: Block diagram of an HEVC encoder (with greyed decoder modules)

The basic block in HEVC is known as the largest coding unit (LCU) and can be recursively split into smaller coding units (CUs), which in turn can be split into small prediction units (PUs) and transform units (TU). These concepts are explained below.

1) *Coding tree units (CTUs) consisting of coding tree blocks (CTBs) :* Previous standards (such as MPEG-4), used *macroblocks* as basic coding units. Each macroblock was composed of 16x16 strictly sized blocks of luminance samples and two 8x8 blocks with the corresponding chroma samples (that is in the case of 4:2:0 color sampling, which is the most common). Its HEVC analogous is the *coding tree unit* (CTU). Its size is selected by the encoder and can be larger than 16x16. There is a -rather important- naming convention here: In texts concerning the HEVC standard, when a term ends with 'unit', a logical unit is indicated which will be eventually encoded in a bit stream. On the other hand, if a term ends with 'block', a portion of video frame buffer to be processed by a module is implied. Thus, a coding tree **unit** (CTU) is a logical unit that consists of three coding tree **blocks** (CTBs). One CTB for luminance (luma) and two for the corresponding chroma samples. Syntax elements are also included. A luma CTB can have sizes of 16x16, 32x32 or 64x64 samples. The larger the size, the better the compression. Now, each CTB can be split into smaller Coding Blocks (CBs) with multiple ways in a tree-like structure ('quadtree') to help decide the prediction type (inter- or intra-picture). (Figure 3).

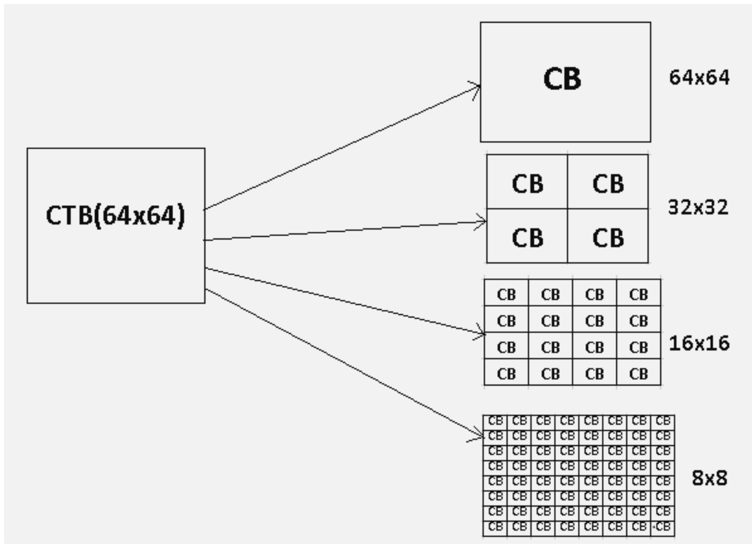


Figure 3: CTB to CB partinioning

2) *Coding units (CUs) and coding blocks (CBs):* A Coding Unit (CU) is used to code the prediction type. Each CU consists of a luma (Y) CB and two chroma ones (Cb and Cr) with the associated syntax elements. The size and positions of CBs are specified by that quadtree CTU syntax mentioned before, with the root being the CTU itself. Thus, the largest size a luma CB can have, is that of the luma CTB. We can have CUs with size up to 64x64 pixels, which is the Largest Coding Unit (LCU) size. This makes the LCU 16 times larger than the macroblock the core of the coding layer of AVC/MPEG4. CTU is concurrently split into CBs. One or many CUs form a CTB while CUs are further partitioned into Prediction Units (PUs) that form tree-like structures of Transform Units (TUs).(Figures 4 and 5).

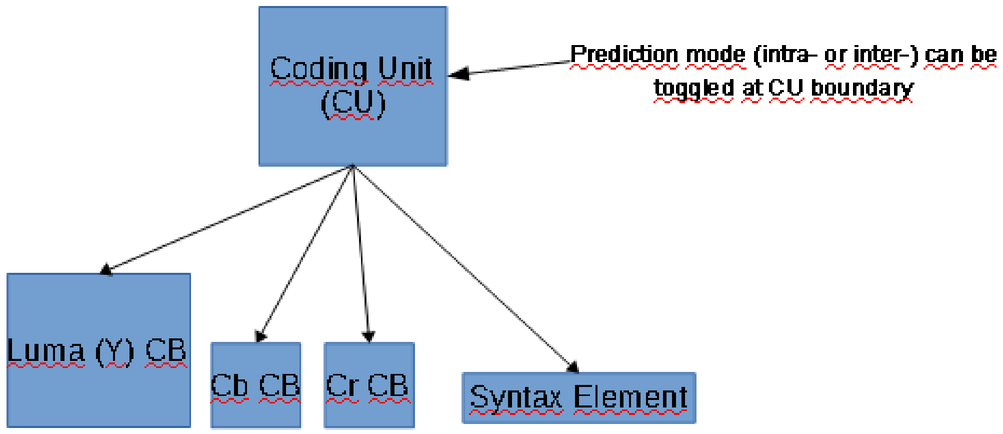

Coding Unit (CU)
Prediction mode (intra- or inter-) can be toggled at CU boundary
Luma (Y) CB
Cb CB
Cr CB
Syntax Element

Figure 4: Coding Unit partitioning

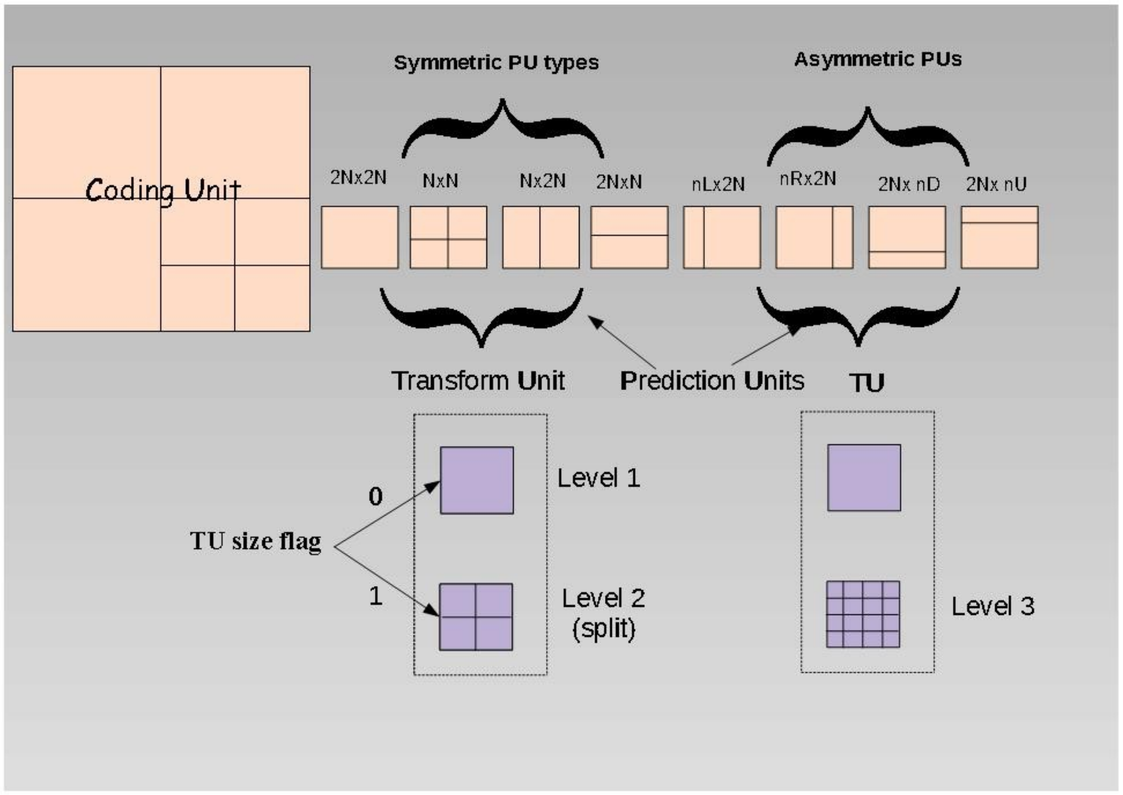


Figure 5: Splitting of CUs to PUs and then into TUs

3) *Prediction units and prediction blocks (PBs):* As mentioned above, CUs (being formed by CBs) are the decision points for the prediction type. For this to work, CBs are also partitioned to prediction blocks (PBs) according to spatial (intra-) or temporal (inter-) predictability. PB sizes may vary from 4x4 to as much as 64x64 samples. (Figure 6).

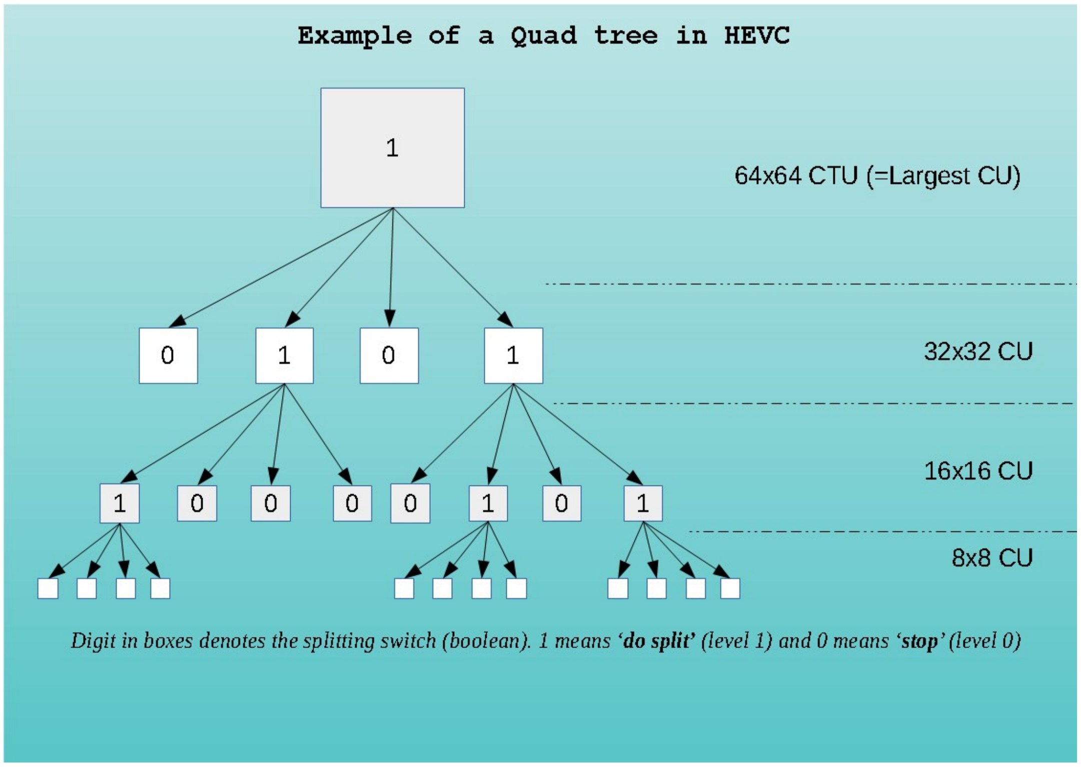

Example of a Quad tree in HEVC
1
64x64 CTU (=Largest CU)
0
1
0
1
32x32 CU
1
0
0
0
0
1
0
1
16x16 CU
8x8 CU
Digit in boxes denotes the splitting switch (boolean). 1 means 'do split' (level 1) and 0 means 'stop' (level 0)

Figure 6: Prediction Units and Prediction Blocks sizes

4) *TUs and transform blocks*: A Coding Unit forms the root of a Transform Unit (TU). Transform Blocks (TBs) are, essentially, blocks of signal samples upon which the same transform is applied. A luma or chroma CB may have the size of a single corresponding (Y, Cb or Cr) TB or may be further split to smaller TBs. Integer arithmetic transformations akin to DCT (Discrete Cosine Transform) or DST (Discrete Sine Transform) are applied to TBs with sizes of 4x4, 8x8, 16x16 or 32x32 (squares) with DST preferred for luma intra (spatial) prediction residuals. (Fig. 7).

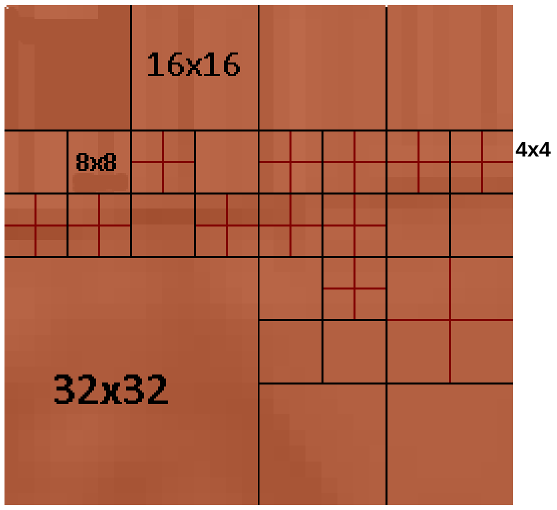


Figure 7: An example of arranging TUs in an LCU.

5) *Motion vector signaling:* By using data from adjacent Prediction Blocks (PBs) and the reference frame, the best candidates are derived. This is called Advanced Motion Vector Prediction (AMVP). HEVC also supports a merge mode for Motion Vector coding that allows the inheritance of MVs from spatially or temporally or spatially neighboring PBs.
6) *Motion compensation:* Motion Vectors use quarter–sample precision and interpolation of fractional-sample positions is achieved via 7 or 8 –tap filters. As in its predecessor standard, HEVC can use multi reference pictures and uni- or bipredictive (that "looks" back and forth) coding. This is achieved by transmitting one or two MVs respectively for each PB. Weighted prediction is supported too by applying offset and scaling operations to the prediction signals.
7) *Intra-picture prediction:* HEVC uses samples of adjacent block borders in the same picture as reference for spatial prediction requirements. This is chosen when inter prediction is not applied. Two (2) planar modes are supported (namely Intra Planar and Intra DC) and 33 angular, thus providing for 35 prediction modes. Prediction mode selection is based on those modes of adjacent PBs decoded previously.
8) *Quantization control:* The H.265 standard uses uniform reconstruction quantization (URQ) as its predecessor. Numerous scaling matrices support the available TB sizes. The quantization step (Qstep) value is determined by an integer Quantization Parameter (QP). QP is in the range [0,51] for 8-bit sequences. [12]
9) *Entropy coding:* HEVC uses an improved version of Context Adaptive Binary Arithmetic Coding (CABAC) scheme for entropy coding. Although it shares the same basic idea as the algorithm in MPEG-4 (a variant of arithmetic coding offering lossless compression) [13], CABAC is now optimized for improved utilization of parallel architectures (yielding better speed efficiency), less usage of context memory and better compression ratios.
10) *In-loop deblocking filtering:* In order to smooth several discontinuities that are often observed at the boundaries of PBs and TBs, a parallelization friendly deblocking filter is used, very much like the one in MPEG-4. Besides the enhancements concerning parallelism, its decision-making and filtering mechanism are also simpler. The filter is operated after the inverse quantization process and its output is fed to the Sample Adaptive Offset filter described below.
11) *Sample adaptive offset (SAO):* To further improve the quality of reconstructed frames and thus optimizing the decoder's output, a SAO filter is operated after the deblocking one. It aims to reduce sample distortion and that is generally achieved by mapping offsets to reconstructed samples according to their classification in several categories. The appropriate offsets are then to be added to each sample depending on its category. [14]

HEVC Computational Complexity

As we have already mentioned, the high coding efficiency achieved by H.265 standard, comes at the cost of higher computational complexity, i.e. increased encoding and decoding (mainly encoding) time. We are going to present this problem in some detail here, because it is this increased computational complexity that has motivated research in exploring ways to speed up the whole trancoding processes.

In [15] (Chapter 4) one can find an extensive study of HEVC's computational complexity in comparison to that of its predecessor H.264/AVC. Nevertheless, we conducted our own tests to compare execution time of HEVC (for specific configurations) with that of its predecessor H.264/AVC. A sample lies below, where two (2) different H.265 encoding configurations have been tested for four (4) Quantization Parameter (QP) values each. Two (2) of the usual test sequences (Kimono and Traffic) were encoded using both H.265 and H.264 reference software (HM 16.14 for H.265 and JM 19.0 for H.264: Will be further described later). The available hardware utilized was an Intel Core 2 Duo E8400 CPU running 64-bit Ubuntu Linux. Execution times are depicted in the barcharts below. As can be clearly seen the computational complexity of HEVC is quite higher than this of H.264/AVC.

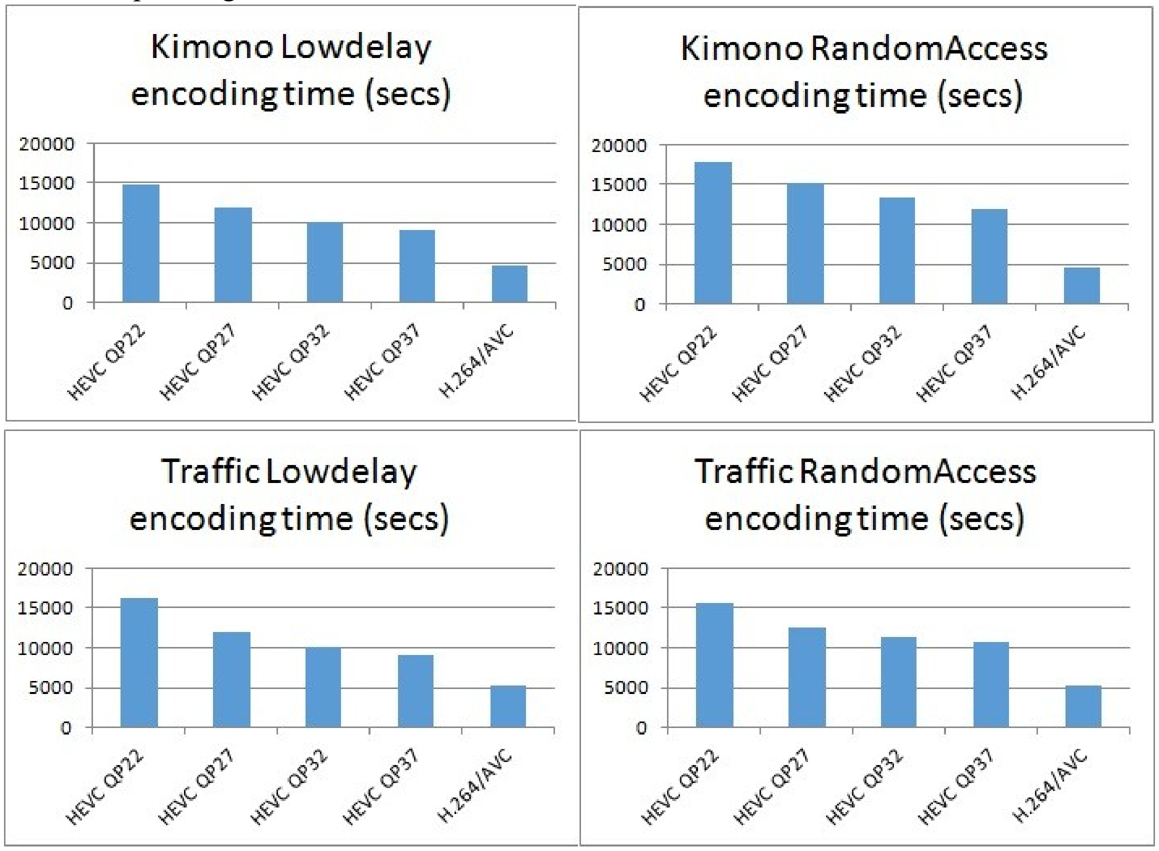


Figure 8:Encoding time of various HEVC configurations vs. MPEG4 for the Kimono and Traffic sequences

Finally, other researchers like in [16] have also measured the ratio of coding complexity increase in HEVC, always compared with that of H.264/AVC. For the purposes of the above mentioned paper, HEVC encoder and decoder were run under several configurations described as: All-intra (**AI**), random access (**RA**), low-delay *B* (**LB**) and low-delay *P* (**LP**). Those test runs proved that H.265 takes up to triple the time that H.264 needs (on average) to encode the same sequence. Especially when only intra prediction is selected, HEVC is 3.2 times slower than MPEG4.

CHAPTER 3: An overview of available parallelization methods in HEVC

Introduction

Before we delve into HEVC and its parallelization, we have to explain the term "*coding efficiency*" which has been and will be used many times in the present thesis. So, by the term *coding efficiency* we denote the maximum video quality we can achieve keeping the bit rate at a desirable low level. In other words, coding efficiency has to do with how much can a coder compress a sequence (i.e. decrease its bit rate) without noticeable degradation in quality, or even with no quality loss. [17] Now, in order to estimate coding efficiency, two significant metrics are employed: Bit rate (commonly written as one word:bitrate) and Peak Signal to Noise Ratio(PSNR).

Bitrate, as the word implies, is the number of bits of information that are processed per time unit (usually per second). When we deal with video, this number is normally expressed in *Kbits per second* or *Kbps*.

Peak Signal to Noise Ratio – PSNR expresses the comparison (ratio) of the maximum possible value of a signal to that of the noise that corrupts it, where the term *noise* generally denotes "unwanted signal" that disrupts the original. We use the decibel logarithmic scale to measure PSNR. When we talk about video transcoding, PSNR is a metric of the difference between the original (raw) video and that played by a decoder. Thus, low PSNR characterizes a poor quality, "noisy" signal. Another important particularity about video is that PSNR is computed for both luma and chroma samples. Yet, as the human visual system is more sensitive to luminance (brightness), Y-PSNR (or luma-PSNR) is the preferred metric. [18].

As we have already stated, while HEVC offers much better coding efficiency compared to H.264/AVC, this great improvement suffers the penalty of increased computational complexity. Unfortunately, although hardware becomes more powerful by the day, all these advances in CPU, bus, memory technology etc. cannot cope with the above stated problem. Therefore, several computer scientists all over the globe continuously work on researching the acceleration of HEVC algorithms. In fact, this great HEVC computational complexity was expected since the standard was conceived, thus leading HEVC developers to make the software easy to be parallelized from the beginning. On the other hand, parallelization in MPEG4 was only an afterthought. [19]. According to several researchers, parallel computing is the means to accelerate HEVC processing. Therefore, high-level parallelization is supported by some HEVC features like wavefront parallel processing (WPP) [20], [21], slices and tiles [19], [22], [23], [24] and some features which allow low-level parallelization (inside the encoding process), such as local parallel method which allows parallel motion estimation [24]. Exploiting CPU features like SIMD, MISD, MIMD etc. (e.g. Intel's SSE, SSSE3, AVX etc. instructions) has also been extensively studied as a way to achieve faster encodings [25], [26].

Before we focus on parallelization based on frame segmentation to slices and tiles, we shall briefly refer to some of the aforementioned other methods and tools employed to speed up the encoding process.

Exploitation of data level parallelism

Today's CPUs (at least most of them) provide media instructions. A typical example on common server platform is the Intel and AMD SSE/MMX technologies, which are based on the single-instruction-multiple-data (SIMD) methods. By exploiting the significant data-level parallelism, SIMD technologies provide a series of effective approaches for fast algorithm implementation, which brings useful guidance to optimize the computational performance [27]. Several works address performance improvement of audio/video signal processing using NEON compiler intrinsic on ARM platforms, and SSE or AVX intrinsics on Intel platforms [25].

Now, if we examine and test HEVC using some version of the reference software and also utilizing an appropriate profiler, it has been shown that the most time-consuming modules are (in descending order):

i. Motion Compensation
ii. Hadamard transform
iii. SAD & SSD calculations (Sum of Absolute Differences, Sum of Square Differences)
iv. Integer transforms
v. Rate-Distortion Optimizated Quantization (RDOQ)
vi. Memory operations

(The above are depicted in the chart below).

Therefore, we can speed up encoding by improving for example the existing MC implementation using SSSE3 instructions like PMADDUBSW to compute the required vector products [26], [27].

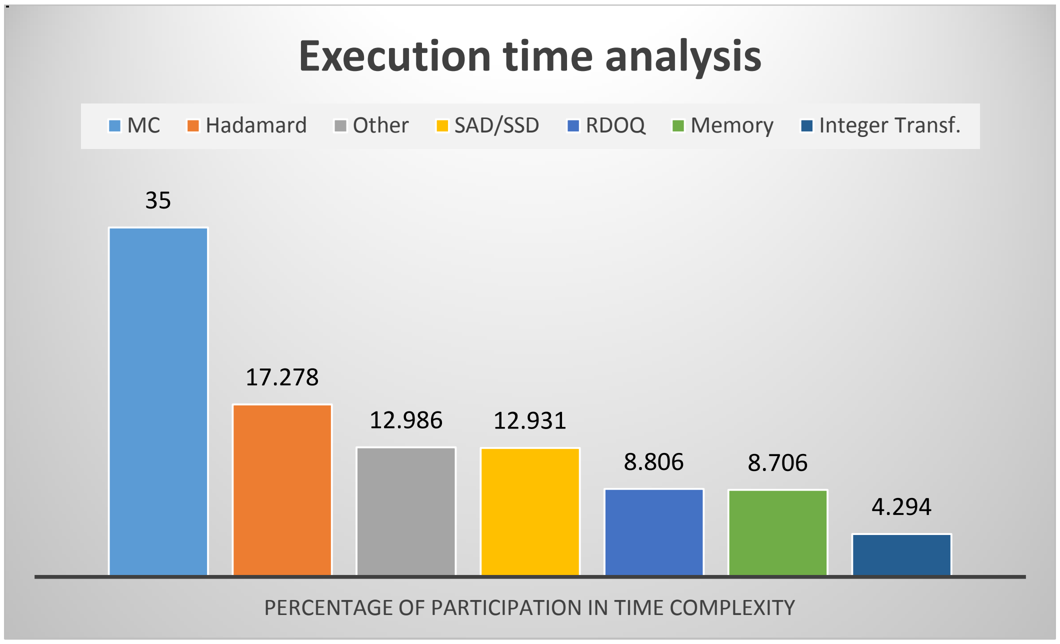


Figure 9: Execution Time Analysis of the HEVC Encoder

Wavefront Parallel Processing

Another method used to speed up the whole encoding process is Wavefront Parallel Processing (denoted as WPP from now on). The idea here is to partition a frame into ‘treeblocks’ organized as rows that can be processed each one by a different execution thread. It has been proved that in such a case we have only low coding losses. Of course, any coding dependency is to be preserved. To elaborate on this last statement: In order to freely process a treeblock, we need to have ready the top-left, left and top-right treeblocks for predictions to work as expected. So we must “interpose” two treeblocks (at least) between consecutive treeblocks rows we process in parallel. Due to this, heavy communication between CPU-cores is required (something that is not needed if we use tiling without cross border filtering). Fortunately, modern CPUs include many cores which can easily communicate with each other, thus making WPP well suited for today’s hardware, especially when the last one is supported by appropriate software libraries. Another advantage of WPP is that it can be implemented almost out-of-the-box. That is because several operations as predictive and entropy coding, or in-loop filtering can be performed in one processing cycle and WPP has no impact on the single step processing capability. We can find examples of WPP utilization in streaming HD video over broadband (e.g. fiber optic) channels and in applications where delays are unacceptable. (Though in the last case, WPP should be combined with dependent slices). [28].

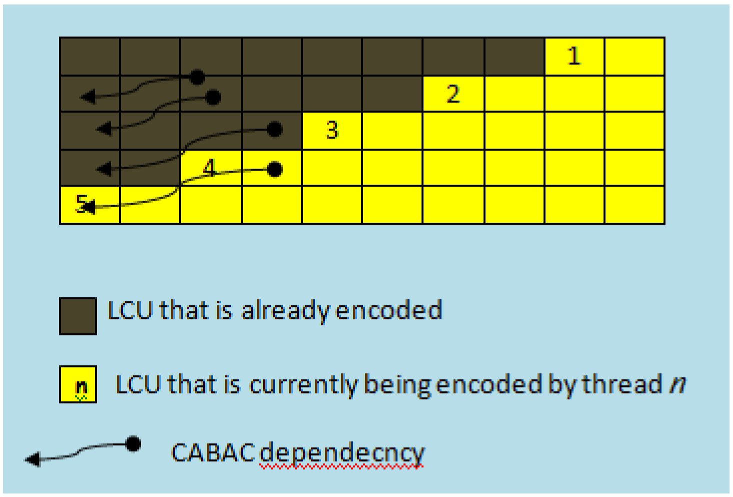

1
2
3
4
5
LCU that is already encoded
n
LCU that is currently being encoded by thread n
CABAC dependecncy

Figure 10: Demonstration of WPP using five (5) threads

Slices

Besides rows of ‘treeblocks’, *slices* are another way to partition a frame in order to parallelize its coding or decoding. By ‘slices’ we describe frame partitions that can be transcoded separately (alone- by a single CPU thread). This means that we don’t need any information from other such slices in order to process a specific one. Therefore, a slice cannot use Coding Units from neighboring ones for prediction (intra or inter). It is also obvious that the number of CUs composing a slice should be an integer one. Of course, a slice can never extend to multiple frames. We use the notation *I-slice* to refer to a slice that consists of intra-prediction CUs only, *P-slices* are slices that can contain both inter-prediction and intra-prediction CUs but in the case of inter-prediction it is unidirectional, while slices that can contain intra- and bidirectional inter-prediction are refered as B-slices (B stands for '**B**'idirectional). Thus, while Motion Estimation and Motion Compensation in a B-slice can use up to two blocks in different reference frames, in a P-slice, only one reference frame can be used by its CUs [29].

Network parameters such as the largest protocol unit that can be transferred (or Maximum Transmission Unit (MTU)) or graphic specific constraints such as the maximum number of Coding Tree Blocks that can be contained in a slice can greatly affect slice partitioning. To elaborate on this, we use Fig. 12. We can see here that partitioning occurs if we follow the raster-scan order within the picture thus yielding less spatial correlation within the frame. [22]

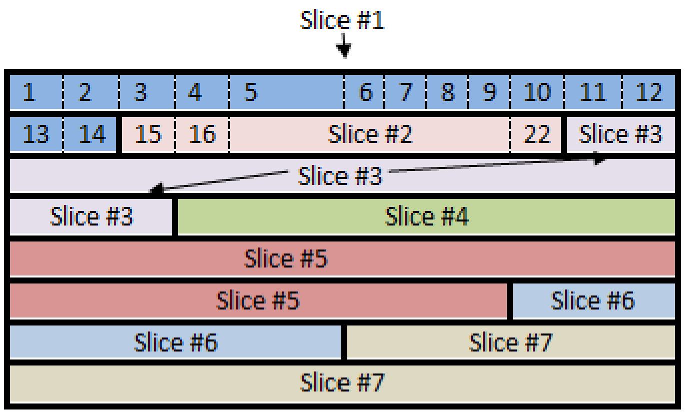


Figure 11: A slice-based partitioned frame with CTBs following a raster scan order within it

We also have to notice that every slice includes additional information in the form of a header. Of course, this increases its size and produces overhead that we cannot ignore at lower bitrates. This extra information, combined with the aforementioned reduced spatial correlation, also harms coding efficiency. To put it in a nutshell: Using many slices improves parallelism but may lead to non-negligible coding losses. On the other hand, if we choose only one or few slices per frame, decoder might not be able to perform in real-time. This is because, the number of slices is determined during encoding time, but the decoder counts on them to improve its performance [19].

Tiles

Besides slices, tiles are another structure used to facilitate parallel processing. As we have seen, each frame is partitioned into CTBs in a rows x columns manner. A tile can be thought of as the rectangular region formed by the intersection of a row and column. With tiles, there can be uniform spacing in row and column boundaries specification, or not. Generally, tile partitioning is considered to be more flexible than slicing. This is mostly because as tiles share the same boundaries with CTBs, they are more compact (spatially) than a slice containing the same number of CTBs. This yields higher pixel correlation with regard to slices. Moreover, there is no tile header info in contrary to slice headers. Although each tile can contain different number of CTBs, their number is always integer. A raster scan order is followed when processing LCUs in each tile and the same order (raster scan) is followed when processing tiles in a frame [30](Fig. 11). To make a long story short: Tile partitioning of pictures, combined with parallel processing seems to really improve coding speed. Yet, as with slices, there is a price to be paid: A degradation in coding efficiency. [31]

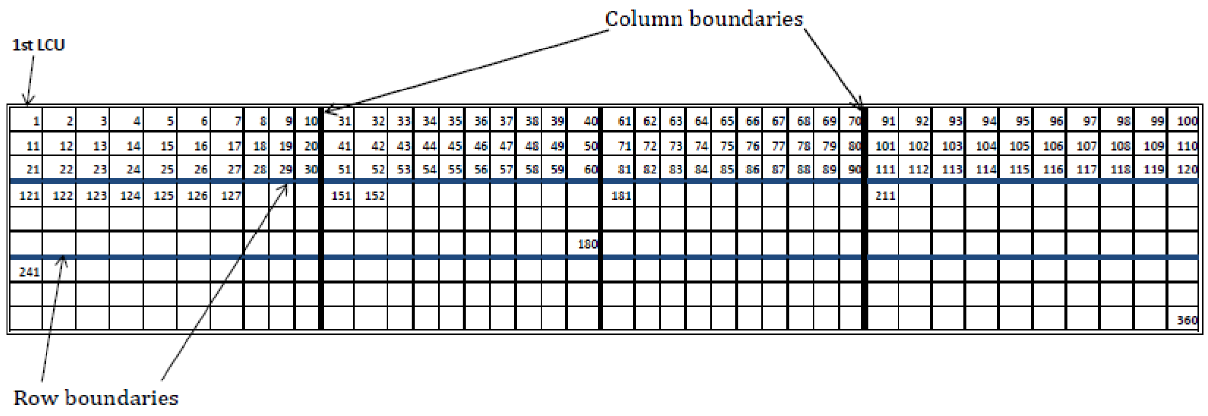


Figure 12: An example of Tiles partitioning using four columns and three rows

Slices and tiles

It is also possible that both tile and slice segments coexist in the same picture; in which case rules have to be set on the way tiles relate to slices. Specifically, no CTB in a tile can span multiple slices and no CTB in a slice can span multiple tiles. [19]. When such conditions are met, the only way that a segment of a slice or even a whole one can span multiple tiles, is its starting point to coincide with that of a tile. (Fig. 14, 15 below) [9].

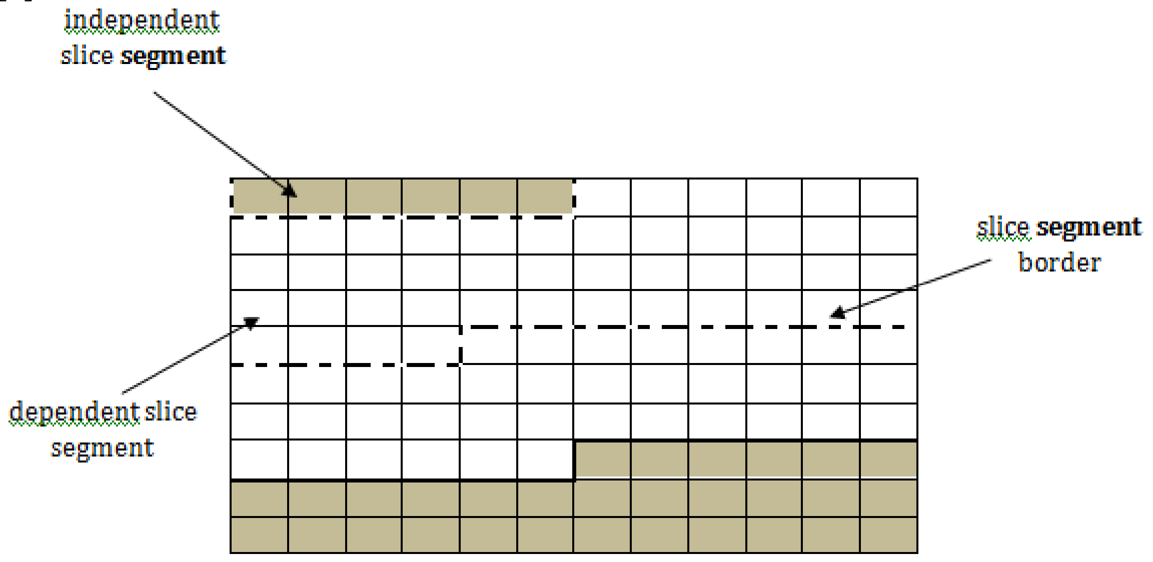


Figure 13: Example of slice segment partitioning of a frame

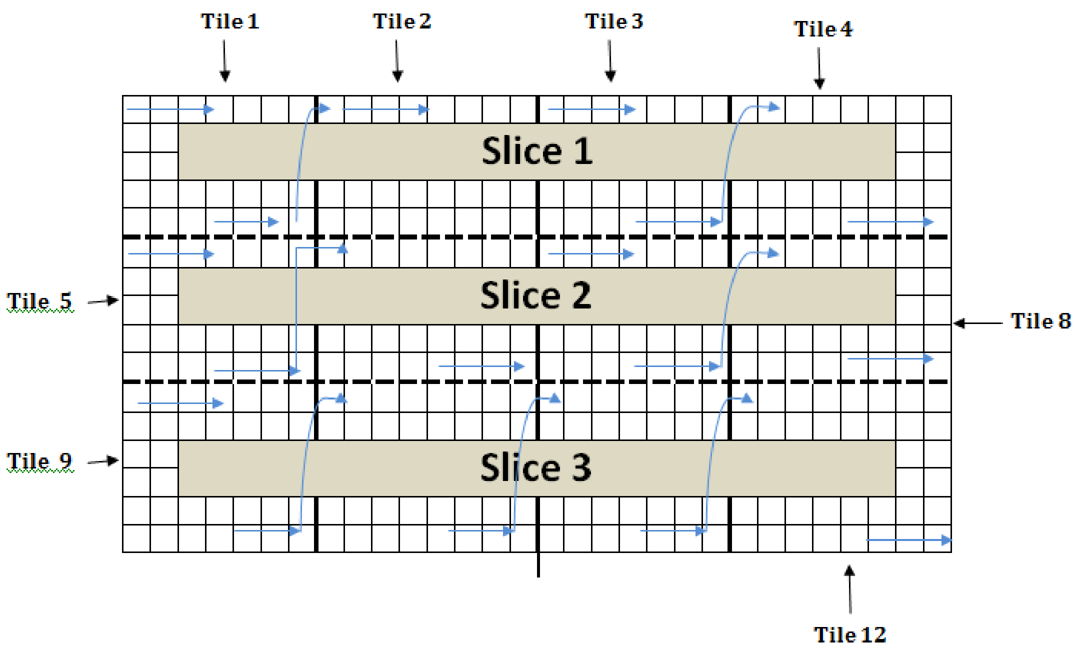


Figure 14: Example of tile segment partitioning of a frame

The benefits and drawbacks of the above approaches are examined in detail in [24]. There we can find tables that assess the degree of CPU (or CPU cores') utilization, in terms of load balancing percentage (with 100% being excellent); depending on the frame resolution and the layout of the chosen partitioning scheme (in a 'slice X tiles' pattern).

Part of such a table is presented below as Table 2. (Where: *AvgCTU* is the average number of CTUs per tile/slice and *MaxCTU* is the number of CTUs in the biggest tile/slice of the frame partition).

Num. of Processors	**Layout**	**AvgCTU**	**MaxCTU**	**Load Balance (100%)**
2560x1600 (40x25 CTUs)				
4	1x4	250	280	89
	2x2	250	260	96
	4x1	250	250	100
6	1x6	166.7	200	83
	2x3	166.7	180	93
	3x2	166.7	182	92
	6x1	166.7	175	95

Table 1: Load balancing achieved with regard to frame partitioning

CHAPTER 4: Evaluation of multiple slices and tiles

Introduction

Now that we have explained both the need for code optimization in HEVC so as to speed up processing and a way to achieve this by partitioning each frame to slices and tiles; we can proceed to present in detail the main goal of this thesis. The idea –in simple words- was to experiment with various frame segmentation ways, trying to objectively measure HEVC's coding efficiency in each case. This means that we had to run the H.265 encoder several times with different configurations collecting and evaluating any output. Here, "output" consists of both the encoded bitstream (.bin files) and the various metrics and statistics produced by the encoder. Thus, we are going to present below how these experiments were conducted and some conclusions we think were drawn.

The HEVC Test Model (HM)

The reference software for HEVC is called HM (**H**EVC Test **M**odel) and, as stated in the manual that accompanies the downloadable zip file, its main purpose is to provide a platform for researchers to experiment with. (e.g. Test different coding tools and evaluate performance accordingly). It is neither optimized in any way, nor claimed to be a super efficient implementation. Also, it is not suitable for any particular use. [2]

The software is developed in C++ and it is documented in details with the utilization of the Doxygen package. HM can be downloaded from the site: http://hevc.info/ in the form of a .zip file which contains the following:

- HM software: Support for the following profiles:
  - the Main, Main 10, and Main Still Picture profiles
  - the Monochrome, Monochrome 12 and Monochrome 16 profiles
  - the Main 12 profile
  - the Main 4:2:2 10 and Main 4:2:2 12 profiles
  - the Main 4:4:4, Main 4:4:4 10, and Main 4:4:4 12 profiles
  - the Main 4:4:4 Still Picture and Main 4:4:4 16 Still Picture profiles
  - the Main Intra, Main 10 Intra, Main 12 Intra, Main 4:2:2 10 Intra, Main 4:2:2 12 Intra, Main 4:4:4 Intra, Main 4:4:4 10 Intra, Main 4:4:4 12 Intra, and Main 4:4:4 16 Intra profiles
  - the High Throughput 4:4:4 16 Intra profile
- SHM software: Support for the Scalable Main, the Scalable Main 10, Scalable Monochrome, Scalable Monochrome 12, Scalable Monochrome 16, and Scalable Main 4:4:4 profiles
- HTM software: Support for the Multiview Main and 3D Main profiles
- HM+SCC software: Support for the following profiles:

- Screen-Extended Main and Screen-Extended Main 10 profiles
- Screen-Extended Main 4:4:4 and Screen-Extended Main 4:4:4 10 profiles
- Screen-Extended High Throughput 4:4:4, Screen-Extended High Throughput 4:4:4 10, and Screen- Extended High Throughput 14 profiles

The profiles mentioned above are supported via specific configuration files that exist in the directory named *cfg/* and we are going to discuss about some of them in the paragraphs that follow. Yet, table 3 below provides a brief explanation about the aforementioned configurations [18].

Name	Configurations
main	Uses InternalBitDepth of 8
main10	Uses InternalBitDepth of 10
Intra_main, intra_main10	All frames are I frames
lowdelay_P_main, lowdelay_P_main10	Uses an I frame followed by P frames. GOP size is 4
lowdelay_main, lowdelay_main10	Uses an I frame followed by B frames. GOP size is 4
randomaccess_main, randomaccess_main10	An I frame is inserted every 32 frames. All other frames are B frames. GOP size is 8

Table 2: Explanation of HEVC encoder configuartions

Description of our testing environment

We used HEVC Test Model (HM) **16.14** for our purposes, which was downloaded from https://hevc.hhi.fraunhofer.de/ and installed on several Linux boxes (Ubuntu 16.04, 64-bits). We found out that the software was executed quite faster in Linux O/S compared to Windows 7, 10 and macOS Sierra. (Surveying the reasons is beyond the scope of this work). Three (3) machines were used to run the encoder which allowed us to execute this number of tests simultaneously. We will present here a description of the experiments and how they were set.

We worked with the contents folder named “HM-16.14”. Building both the encoder and the decoder for Linux OS is quite simple. One needs just to type ‘make’ in HM-16.14/build/linux directory. Yet, we had to alter the ‘makefiles’ in order to make use of LLVM’s clang and clang++ compilers (initially made to be used by CERN) which seem to produce executables that run faster, especially with the –O4 switch [32]. The executables built are left in HM-16.14/bin. The encoder (which we actually ran) is under the filename: TAppEncoderStatic. There are several sample configuration files for testing purposes located in HM-16.14/cfg and they are distinguished in three main groups. The ones that contain the string ‘Intra’, those with ‘LowDelay in their filename and others with filenames containing ‘RandomAccess and they are to be used for intra-prediction only, random-access and low-delay conditions respectively [33].

CHAPTER 4: Evaluation of multiple slices and tiles

The above mentioned parameter groups define different testing conditions for the encoder that can be roughly described as follows:

All Intra: Only intra prediction is used for coding. In other words only information from within the frame is exploited for ME purposes. No previous or subsequent frames are involved. This leads to extremely poor execution time, so we will avoid using this configuration and prefer inter-prediction instead.

Low Delay: Frames are coded in the same order as their transmission. This configuration applies mostly to live video with interaction support, where delays are not tolerable while random access is not compulsory. To assess 'low delay' configuration we use either B frames with bi- and uni-prediction or P ones with unidirectional prediction only.

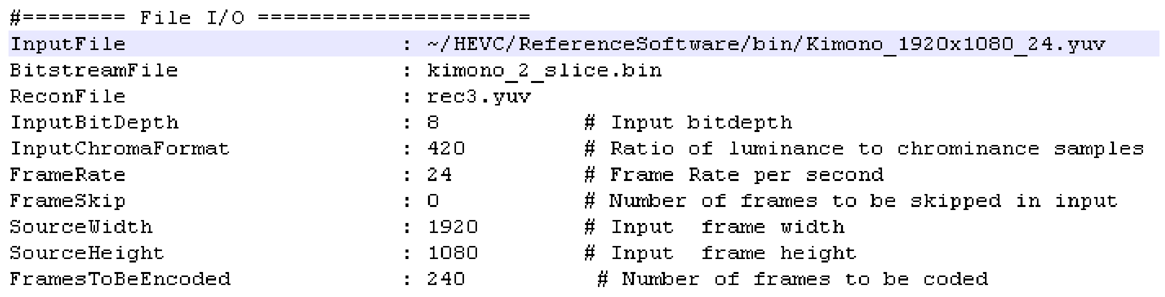
Random Access: This configuration is chosen when we want the best compression efficiency with ability to begin decoding at almost any second. In this case, pictures are not transmitted in the same order as coded, which means that a structural delay is urged. We use 'random access' in applications like video podcasting or streaming. [34].

A discrete Group of Pictures (GOP) structure is defined for each one of the above configurations. Namely, an *intra_main* .cfg file has to cater that each single frame is coded only in intra-prediction mode. A *lowdelay*_main configuration will code only the I-frame (initial frame) using intra-prediction and all the others using P or B inter- modes. For *randomaccess* configurations, sequences of either I or B frames are periodically used in a form like IBBB….BBI. [8]

Among the numerous settings one can experiment with, there are some that **must** be edited according to the properties of the video to be encoded. Thus, parameters we had to change for each test point are [33], [11]:

- *InputFile* that contains the path of the source video sequence on the system to test
- *FrameRate* which obviously defines the frame rate of the sequence to be encoded
- *SourceWidth* and *SourceHeight* are accordingly- the width and height (in pixels) of the input video
- *FramesToBeEncoded*: How many frames of the input sequence we wish to encode
- *QP* or Quantization Parameter (will be explained later)
- *InputBitDepth*: How many bits are used for color (e.g 8-bit, 10-bit high color etc)

Most of the above parameters are to be found at the header of each configuration file. Below is a sample of the ones we used.



```
#======== File I/O =====================
InputFile                       : ~/HEVC/ReferenceSoftware/bin/Kimono_1920x1080_24.yuv
BitstreamFile                   : kimono_2_slice.bin
ReconFile                       : rec3.yuv
InputBitDepth                   : 8           # Input bitdepth
InputChromaFormat               : 420         # Ratio of luminance to chrominance samples
FrameRate                       : 24          # Frame Rate per second
FrameSkip                       : 0           # Number of frames to be skipped in input
SourceWidth                     : 1920        # Input  frame width
SourceHeight                    : 1080        # Input  frame height
FramesToBeEncoded               : 240          # Number of frames to be coded
```

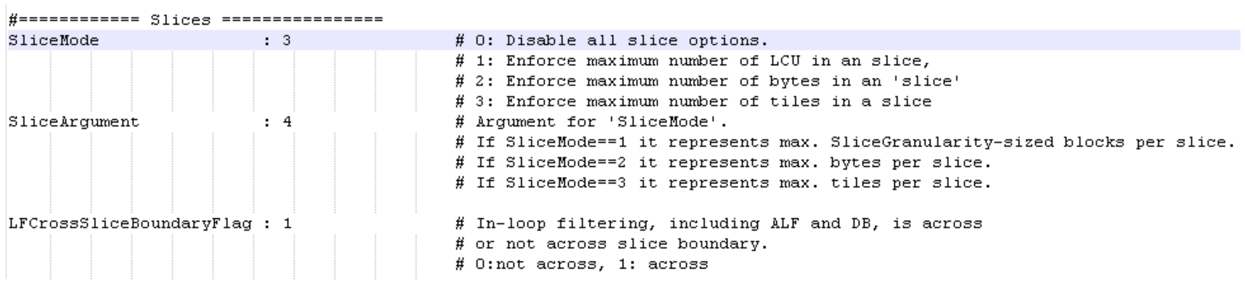
Figure 15: Sample header of an HEVC .cfg file

Besides the above, we had to tamper with some other parameters specific to the purposes of our work which will elaborate below. The rest were left with their default values.

Our experiments had to do with slicing and tiling options and also with **quantization**. For each video sequence four **quantization parameter (QP)** values were to be used: 22, 27, 32 and 37. These values define the QP values used for the I and P-frames in a sequence (configuration files further define QP values used for other frames). Yet, most of our concern had to do with slicing and tiling parameters which we explain here.

The parameters which one has to modify in slice mode, are as follows:

- **SliceMode** defines whether the input video will be partitioned into slices or not and how exactly will those slices be cut. It offers four options: 0 (no slices at all), 1 for setting a maximum number of Largest Coding Units (LCUs) in a slice, 2 for setting a maximum number of bytes in a slice and 3 to cut slices so as to ensure a maximum number of tiles in a slice. We have to assign tile partitioning parameters, in order to take into account mode 3 of SliceMode and allocate the tiles to each slice.
- **SliceArgument** is an option relative with SliceMode value. If SliceMode value is 0, nothing happens. If SliceMode value is 1, one has to edit the maximum number of blocks that each slice will contain. If SliceMode value is 2, the user has to insert the maximum number of bytes per each slice. If SliceMode value is 3, we have to provide the maximum number of tiles per slice.
- **LFCrossSliceBoundaryFlag** sets whether in-loop filters, like Adaptive Loop (ALF) and Deblocking, will be applied across or not across the slice boundary. It takes value 0 for not across and value 1 for across.

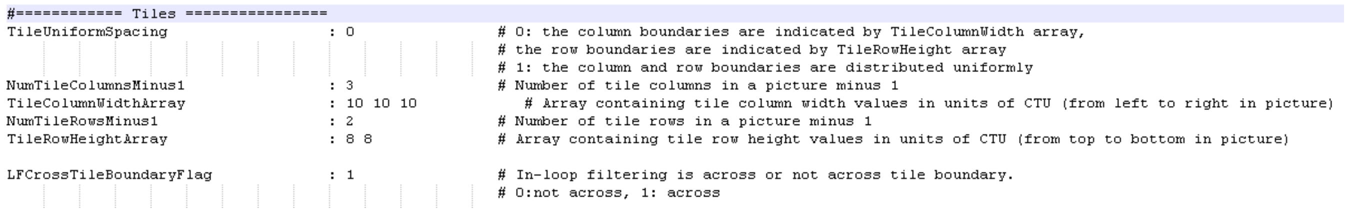


```
#============ Slices ================
SliceMode                : 3              # 0: Disable all slice options.
                                          # 1: Enforce maximum number of LCU in an slice,
                                          # 2: Enforce maximum number of bytes in an 'slice'
                                          # 3: Enforce maximum number of tiles in a slice
SliceArgument            : 4              # Argument for 'SliceMode'.
                                          # If SliceMode==1 it represents max. SliceGranularity-sized blocks per slice.
                                          # If SliceMode==2 it represents max. bytes per slice.
                                          # If SliceMode==3 it represents max. tiles per slice.

LFCrossSliceBoundaryFlag : 1              # In-loop filtering, including ALF and DB, is across
                                          # or not across slice boundary.
                                          # 0:not across, 1: across
```

Figure 16: Slicing options configuration sample

Settings concerning tiling are as follows:

- **TileUniformSpacing :** Can be 0 or 1. A value of 0 means that column boundaries are assigned by TileColumnWidthArray while row ones are assigned by TileRowHeightArray. A value of 1 implies that column and row boundaries are assigned uniformly.
- **NumTileColumnsMinus1:** If N is the number of tile columns per frame then it is set to N-1 as C array style of tile numbering (0 to N-1) is used.
- **TileColumnWidthArray** defines an array that includes tile column width values in units of CTU starting from left to right in the frame. E.g. In Kimono sequence, each frame consists of 510 CTUs, distributed in 30 columns × 17 rows. Therefore, if we want to partition the frame in 3 slices with 4 discrete tiles in each slice, we can insert values 7,8 and 7 (space separated) in TileColumnWidthArray. This will yield four tiles with 7+8+7+8=30 CTUs width respectively.
- **NumTileRowsMinus1:** Like NumTile*Columns*Minus1 mentioned above. The number of tile rows in a frame minus 1. For instance, if a frame has to be partitioned into 3 tile rows, NumTileRowsMinus1 will be 2.
- **TileRowHeightArray:** Like Tile*Column*WidthArray. Defines an array of tile row height values in units of CTU starting from top to bottom in frame. Let's take for example the Traffic sample sequence (1000 CTUs per frame in 40 columns × 25 rows). If we wish to partition each frame to 3 slices by 4 tiles each, then, one way is to set this array to contain values 8,8. Thus, three (3) tiles per column will be produced with the first two to include 8 CTUs and a third one of 9 (8+8+9=25).
- **LFCrossTileBoundaryFlag** sets whether in-loop filter is across or not across the tile boundary. It takes value 0 for not across and value 1 for across.



```
#============ Tiles ================
TileUniformSpacing                : 0          # 0: the column boundaries are indicated by TileColumnWidth array,
                                               # the row boundaries are indicated by TileRowHeight array
                                               # 1: the column and row boundaries are distributed uniformly
NumTileColumnsMinus1              : 3          # Number of tile columns in a picture minus 1
TileColumnWidthArray              : 10 10 10     # Array containing tile column width values in units of CTU (from left to right in picture)
NumTileRowsMinus1                 : 2          # Number of tile rows in a picture minus 1
TileRowHeightArray                : 8 8        # Array containing tile row height values in units of CTU (from top to bottom in picture)

LFCrossTileBoundaryFlag           : 1          # In-loop filtering is across or not across tile boundary.
                                               # 0:not across, 1: across
```

Figure 17: Tiling options configuration sample

CHAPTER 4: Evaluation of multiple slices and tiles

Let's elaborate on the above explaining the parameter values we have used. First of all, we decided to evaluate six (6) segmentation ways for two parameter groups and four (4) discrete QP values each. Specifically, we tested the following frame segmentations:

i. 2 slices – no tiles
ii. 2 slices with 6 discrete tiles each (2 x 6)
iii. 3 slices
iv. 3 slices x 4 tiles
v. 4 slices and
vi. 4 slices x 3 tiles

Each tile is inside in exactly one slice and each slice can contain only whole (not parts) tiles.

The raw videos we have dealt with are *Kimono* and *Traffic*. Their attributes are presented in the following table [33].

Sequence name	**Resolution**	**Frame count**	**Frame rate**	**Bit depth**	**Intra**	**Random access**	**Low-delay**
Traffic	2560x1600	150	30fps	8	Main/ Main10	Main/ Main10	
Kimono	1920x1080	240	24fps	8	Main/ Main10	Main/ Main10	Main/ Main10

Table 3: Two of the sequences used for HEVC testing

As we can see above, each frame in Traffic sequence consists of 2560*1600=4096000 pixels. Thus, if one chooses MaxCUWidth and MaxCUHeight of 64 pixels each (as they are by default), we have 4096000/(64*64)=1000 CTUs per picture. They are distributed at a 40x25 (width X height) pattern. In Kimono sequence, respectively, there are 1920*1080/(64*64)=30****round***(1080/64)=510 CTUs per frame as illustrated below with the image being partitioned in 10 tiles [24].

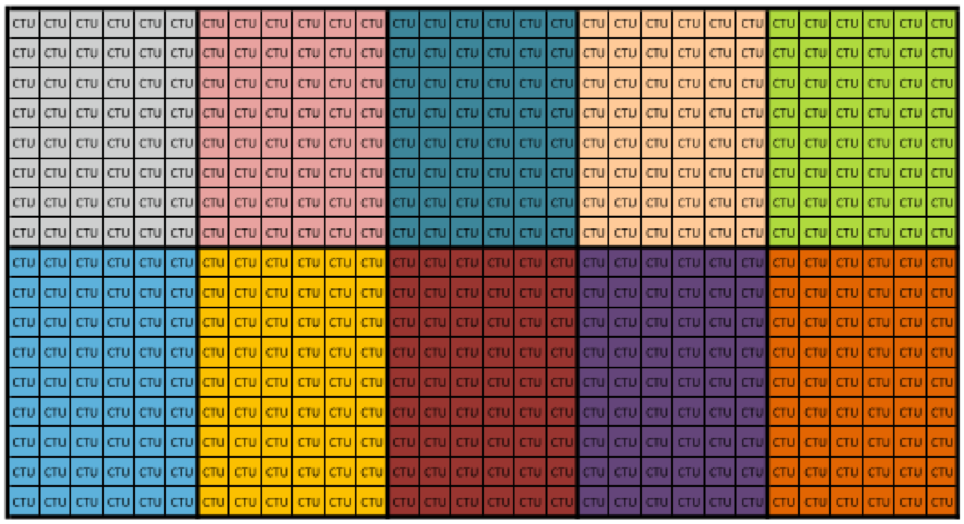


Figure 18: Division of a full-HD frame (1920x1080 pixels) into 10 tiles (5 columns with a width of 6 CTUs; 2 rows with a height of 8 and 9 CTUs each)

So, if we want to partition Traffic in 3 slices containing 4 tiles each, we can define the corresponding configuration parameters as follows:

- SliceMode=3 (enforce maximum tiles in a slice)
- SliceArgument=4 (4 tiles per slice at most)
- TileUniformSpacing=0 (TileColumnWidth indicates column boundaries and TileRowHeightArray indicates row boundaries)
- NumTileColumnsMinus1=3 (3+1=4 tile columns per picture)
- TileColumnWidthArray=[10 10 10] (which yields 4 tiles per slice (40 CTUs div 4) with a width of 10 CTUs each)
- NumTileRowsMinus1=2 (2+1=3 rows of tiles per picture)
- TileRowHeightArray=[8 8] (which yields 3 tiles (8+8+9=25 CTUs) by height).

Using software like StreamEye, we can verify that the above configuration will produce encoded BitStreams with frames partitioned in slices containing tiles as the one that can be seen below.

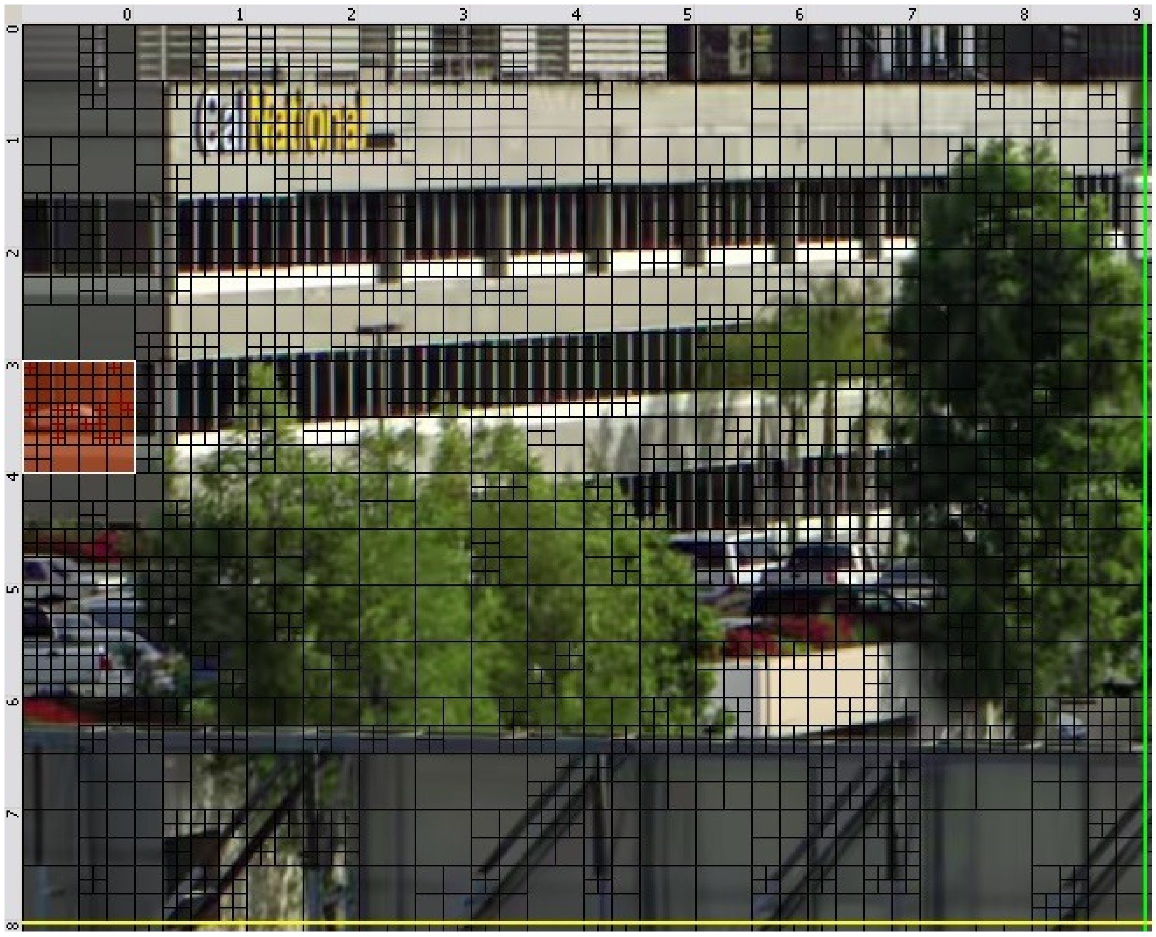


Figure 19: A tile with "dimensions" of 10 x 9 CTUs (green line separates tiles while yellow demarcates slices)

Now, let's consider another example. We are going to setup the encoder, so as to partition every Kimono frame to 4 slices with 3 tiles each. As we have seen, every Kimono frame consists of 30 CTUs in width X 17 CTUs in height=510 CTUs. Thus, we can use the following parameter set:

- SliceMode=3 (enforce maximum tiles in a slice)
- SliceArgument=3 (3 tiles per slice at most)
- TileUniformSpacing=0
- NumTileColumnsMinus1=2 (2+1=3 tile columns per picture)
- TileColumnWidthArray=[10 10] (which yields 3 tiles per slice (30 CTUs div 3) with a width of 10 CTUs each)
- NumTileRowsMinus1=2 (2+1=3 rows of tiles per picture)
- TileRowHeightArray=[5 4 4] (which yields 4 tiles (5+4+4+4=17 CTUs) by height).

A sample tile in a frame produced using configuration files like the above, looks like the one in Fig. 18.

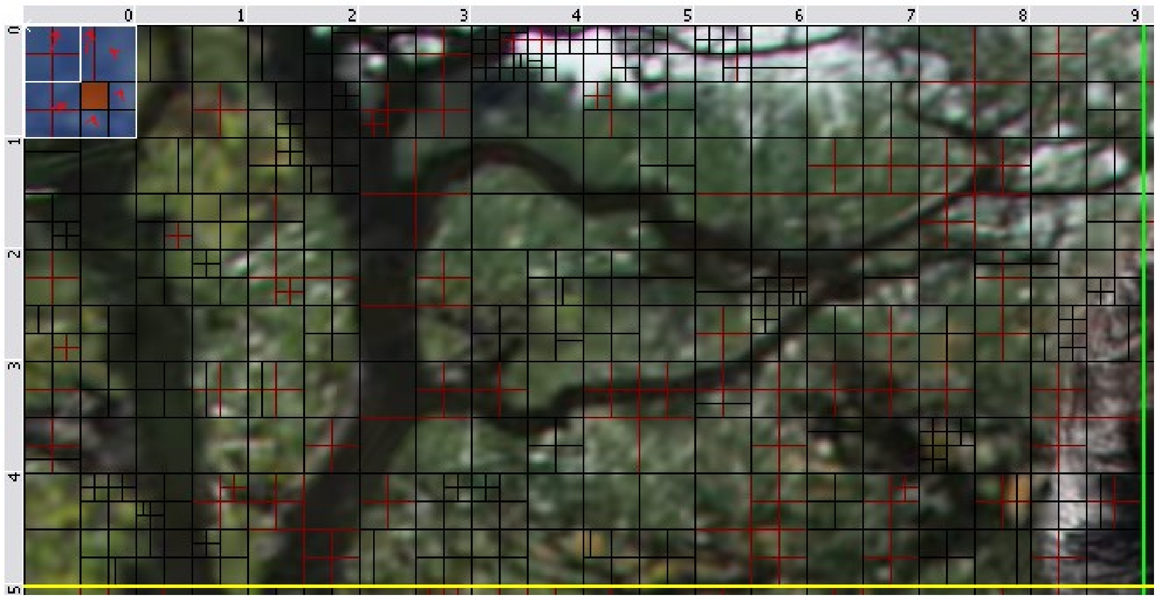


Figure 20: A tile with "dimensions" of 10 x 5 CTUs

The full list of values for the `SliceArgument`, `NumTileColumnsMinus1`, `TileColumnWidthArray`, `NumTileRowsMinus1` and `TileRowHeightArray` parameters we had to use for the chosen six segmentations for each one of the two sequences is cited in Table 5 below.

Kimono sequence (510 CTUs per frame)					
Segmentation	*SliceArgument*	*NumTileColumnsMinus1*	*TileColumnWidthArray*	*NumTileRowsMinus1*	*TileRowHeightArray*
2 slices	255	0	*irrelevant*	0	0
2x6	6	3	7 8 7	2	6 5
3 slices	170	0	*irrelevant*	0	0
3x4	4	3	7 8 7	2	6 5
4 slices	128	0	*irrelevant*	0	0
4x3	3	3	7 8 7	2	6 5
Traffic sequence (1000 CTUs per frame)					
Segmentation	*SliceArgument*	*NumTileColumnsMinus1*	*TileColumnWidthArray*	*NumTileRowsMinus1*	*TileRowHeightArray*
2 slices	500	0	*irrelevant*	0	0
2x6	6	3	10 10 10	2	8 8
3 slices	334	0	*irrelevant*	0	0
3x4	4	3	10 10 10	2	8 8
4 slices	250	0	*irrelevant*	0	0
4x3	3	3	10 10 10	2	8 8

Table 4: Partitioning parameters

Now that we have explained how frame partitioning is achieved tampering with the .cfg files, we can proceed to elaborate more on our test runs.

The encoder would be executed for four (4) QP values of 22, 27, 32 and 37 accordingly. Each experiment should be conducted twice: One time for the *main low delay P* and one time for the *random access* parameter sets. Thus we had to run the encoder 6*4*2=48 times for each sequence.

As we experimented with two (2) sequences, we had to perform 48*2=96 runs and collect this number of output files. In order to do so, twelve (12) configuration files had to be created for each sequence: Six to define the appropriate seqmentation for the low-delay parameter set and six for the random-access conditions. Below is the directory structure of our testing environment.

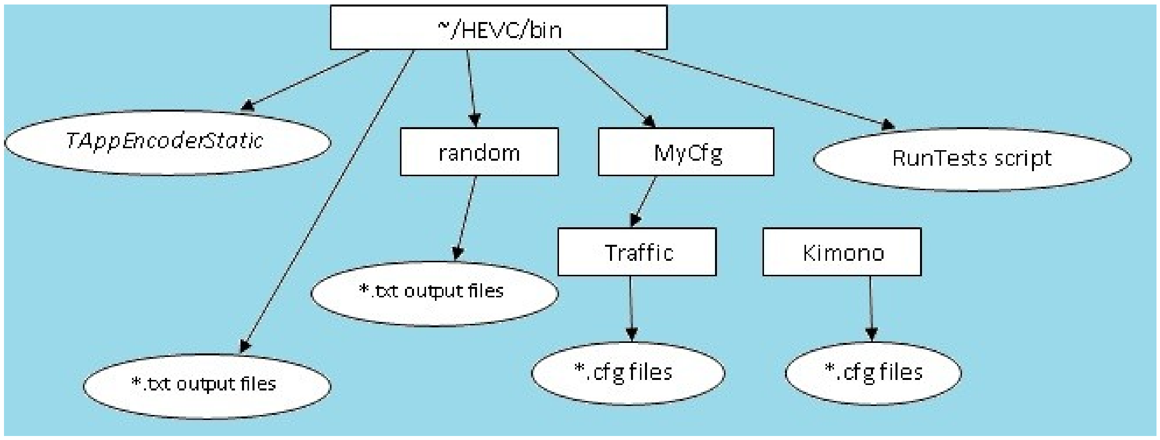


Figure 21: Directory structure of our testing environment

To explain the above figure, we have to say that each dozen of the appropriate configuration files is located under MyCfg/ directory. Obviously, config files for, say, Kimono sequence, are inside the homonymous folder. The directory named *random/* contains the output of executions with random-access conditions. Low-delay executions are left in the same directory with the encoder executable as *.txt files.

Now, in order to perform all or some of the required tests, we wrote a Bash [35] script to do the job. It is located in the same path with *TAppEncoderStatic* (the encoder executable) under the name of *RunTests*. We used the *taskset* Linux command [36] to distribute each encoder process to a different CPU core <u>when possible</u>. Part of the script, is depicted below.

```
echo "Kimono 4x3 Lowdelay"
taskset -c 0,1,2,3 ./TAppEncoderStatic -c MyCfg/Kimono/lowdelay_P_main_4_slice_3_tile.cfg > kim4x3.txt &
echo "Kimono 2 slice RandomAccess"
taskset -c 0,1,2,3 ./TAppEncoderStatic -c MyCfg/Kimono/randomaccess_main_2_slice.cfg > random/kim2sl.txt &
echo "Kimono 3 slice RandomAccess"
taskset -c 0,1,2,3 ./TAppEncoderStatic -c MyCfg/Kimono/randomaccess_main_3_slice.cfg > random/kim3sl.txt
echo "Kimono 4 slice RandomAccess"
taskset -c 0,1,2,3 ./TAppEncoderStatic -c MyCfg/Kimono/randomaccess_main_4_slice.cfg > random/kim4sl.txt &
```

Figure 22: Bash script to initiate test runs

Each time we completed a full set of tests (24 runs, 12 for each sequence), we copied the results for the specific QP value, changed the Quantization Parameter in each .cfg file and the tests started all over. To our benefit, Linux offers a very handy stream editor (`sed`) that allowed for changing QP values in all 24 configuration files with a single command like this [37]:

```
sed 's /QP : 22/QP : 27/g' *.cfg
```

As stated above, we also had to use **Elecard's StreamEye** program to check if the chosen frame segmentation was indeed applied on the encoded bitstream. We present below a frame of the Traffic sequence (rendered by StreamEye) after a 4x3 segmentation encoding. First picture exhibits slicing, the second one shows tiling and the last exhibits both (yellow gridlines demarcate slices while green lines are tile boundaries).

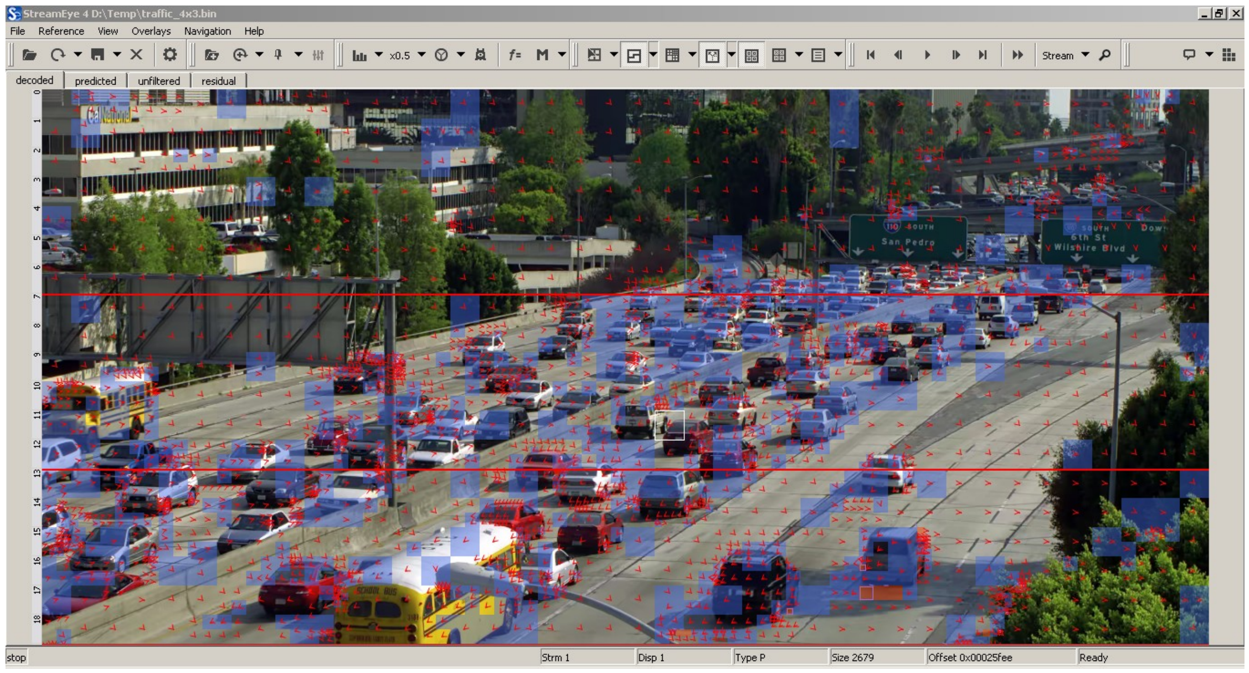

StreamEye 4 D:\Temp\traffic_4x3.bin
File Reference View Overlays Navigation Help
decoded predicted unfiltered residual
Stream
stop
Strm 1
Disp 1
Type P
Size 2679
Offset 0x00025fee
Ready

Figure 23: Exhibiting slices in a P frame of 'Traffic' sequence

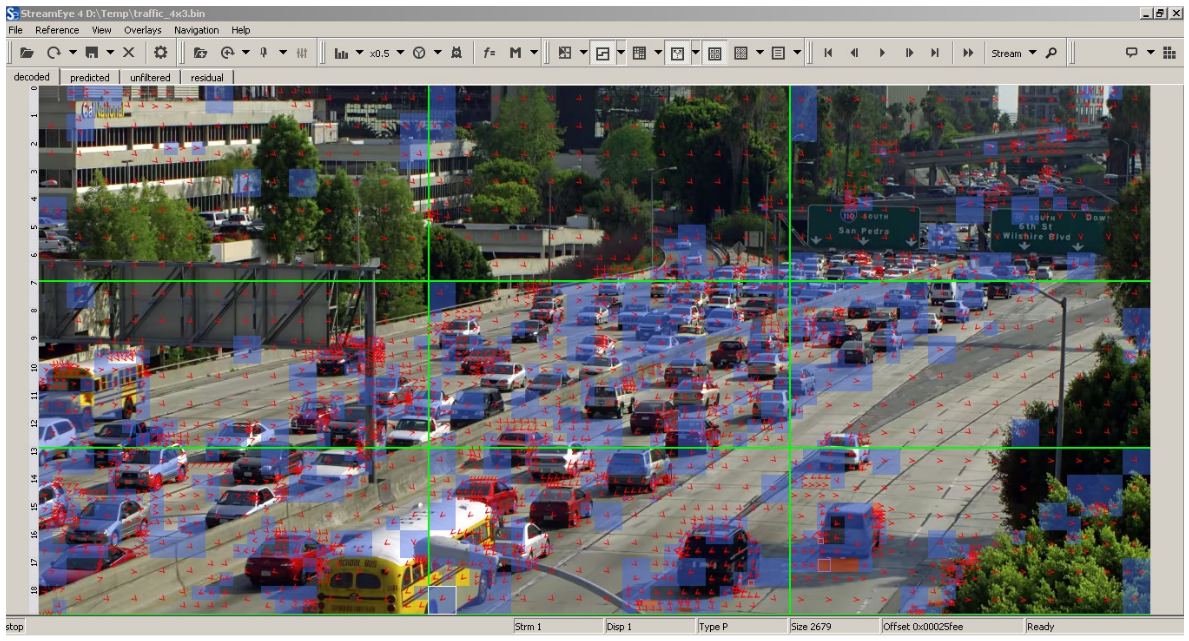

StreamEye 4 D:\Temp\traffic_4x3.bin
File Reference View Overlays Navigation Help
decoded predicted unfiltered residual
stop Strm 1 Disp 1 Type P Size 2679 Offset 0x00025fee Ready

Figure 24: Exhibiting tiles in 'Traffic' frame

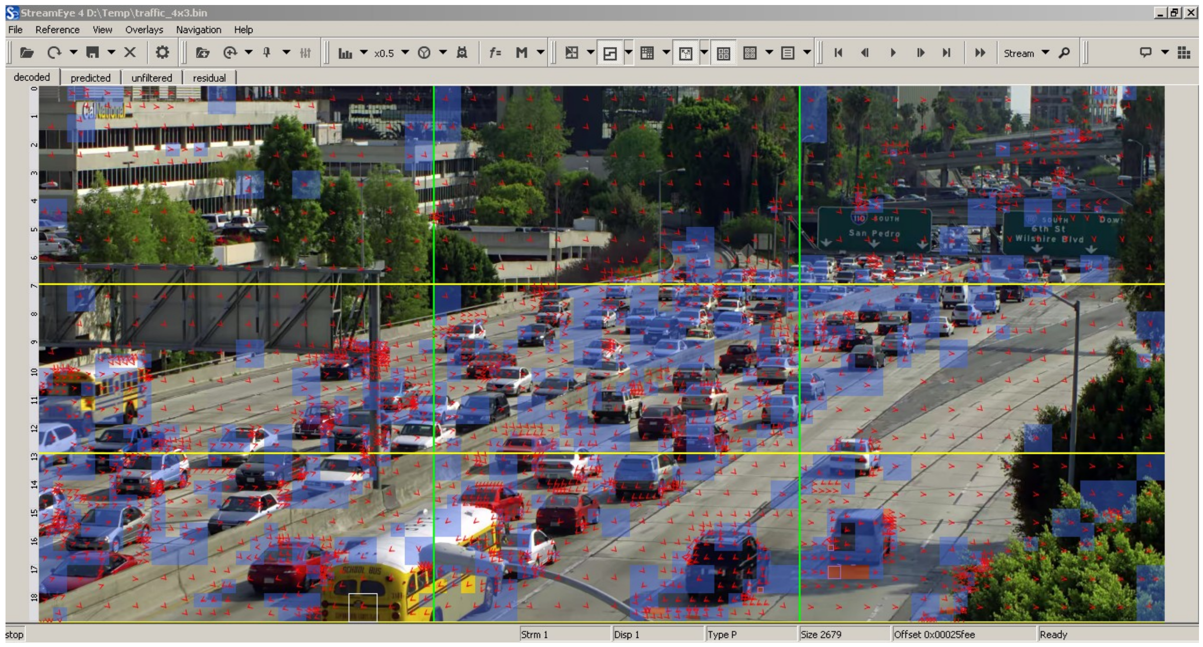

StreamEye 4 D:\Temp\traffic_4x3.bin
File Reference View Overlays Navigation Help
decoded predicted unfiltered residual
stop Strm 1 Disp 1 Type P Size 2679 Offset 0x00025fee Ready

Figure 25: Slicing AND tiling in 'Traffic'

Moreover, as can be seen in the script we wrote, the encoder was executed using Bash shell with commands and output redirections like:

```
./TAppEncoderStatic -c MyCfg/Traffic/lowdelay_P_main_2_slice_6_tile.cfg >
traf2x6lowdelay.txt
```

Thus, the output was saved in a text file each time in order to be reclaimed and studied later. The results produced by the encoder (besides the coded bitstream of course) consist of rich information about the configuration used to run it and a great deal of details concerning the coding of each frame. As a summary, the encoder provides invaluable data like the number of I, P and B frames, the average BitRate, luma PSNR, chroma PSNR, YUV-PSNR, total duration of the encoding process in seconds etc. The frames that follow illustrate some parts of one of the many outputs we had to process.

```
 HM software: Encoder Version [16.17] (including RExt)[Linux][GCC
                          5.4.0][32 bit]
Input          File                      : Kimono_1920x1080_24.yuv
Bitstream      File                      : kimono_2x6.bin
Reconstruction File                      : rec.yuv
Real      Format                         : 1920x1080 24Hz
Internal Format                          : 1920x1080 24Hz
Sequence PSNR output                     : Linear average only
Sequence MSE output                      : Disabled
Frame MSE output                         : Disabled
MS-SSIM output                           : Disabled
Cabac-zero-word-padding                  : Enabled
Frame/Field                              : Frame based coding
Frame index                              : 0 - 99 (100 frames)
Profile                                  : main
CU size / depth / total-depth            : 64 / 4 / 4
RQT trans. size (min / max)              : 4 / 32
Max RQT depth inter                      : 3
Max RQT depth intra                      : 3
Min PCM size                             : 8
Motion search range                      : 64
Intra period                             : -1
Decoding refresh type                    : 0
QP                                       : 32
Max dQP signaling depth                  : 0
Cb QP Offset                             : 0
Cr QP Offset                             : 0
QP adaptation                            : 0 (range=0)
GOP size                                 : 4
Input bit depth                          : (Y:8, C:8)
MSB-extended bit depth                   : (Y:8, C:8)
Internal bit depth                       : (Y:8, C:8)
PCM sample bit depth                     : (Y:8, C:8)
Intra reference smoothing                : Enabled
………………………
Cost function:                           : Lossy coding (default)
RateControl                              : 0
WPMethod                                 : 0
```

Table 5: Some of the first lines produced by the encoder with info about test conditions used

In the **above** table, we deliberately choose to show an example where the total time value is of no sense (negative). This is due to an overflow of the variable responsible for the timing. The overflow itself happened because of the extreme execution duration in the cases when the hardware used was not powerful enough. Allow us to notice that there had been cases when the encoder had needed more than 12 hours to process all the frames of a sequence. On the other hand, the process of encoding all 240 Kimono frames, took (only ?) 621.360 sec when an Intel core-i5 CPU was utilized.

```
       SUMMARY ----------------------------------------------------------
      Total Frames |   Bitrate     Y-PSNR    U-PSNR    V-PSNR    YUV-PSNR
         100    a    1595.2877   37.2787   40.1721   41.7759   38.0669
       I Slices----------------------------------------------------------
      Total Frames |   Bitrate     Y-PSNR    U-PSNR    V-PSNR    YUV-PSNR
          1    i    7088.2560   40.4183   41.4720   42.6766   40.8927
       P Slices----------------------------------------------------------
      Total Frames |   Bitrate     Y-PSNR    U-PSNR    V-PSNR    YUV-PSNR
         99    p    1539.8032   37.2470   40.1590   41.7668   38.0460
       B Slices----------------------------------------------------------
      Total Frames |   Bitrate     Y-PSNR    U-PSNR    V-PSNR    YUV-PSNR
          0    b        -nan      -nan      -nan      -nan      -nan
                                  RVM: 0.000
Bytes written to file: 836579 (1606.232 kbps)
Total Time:    -1559.360 sec.
```

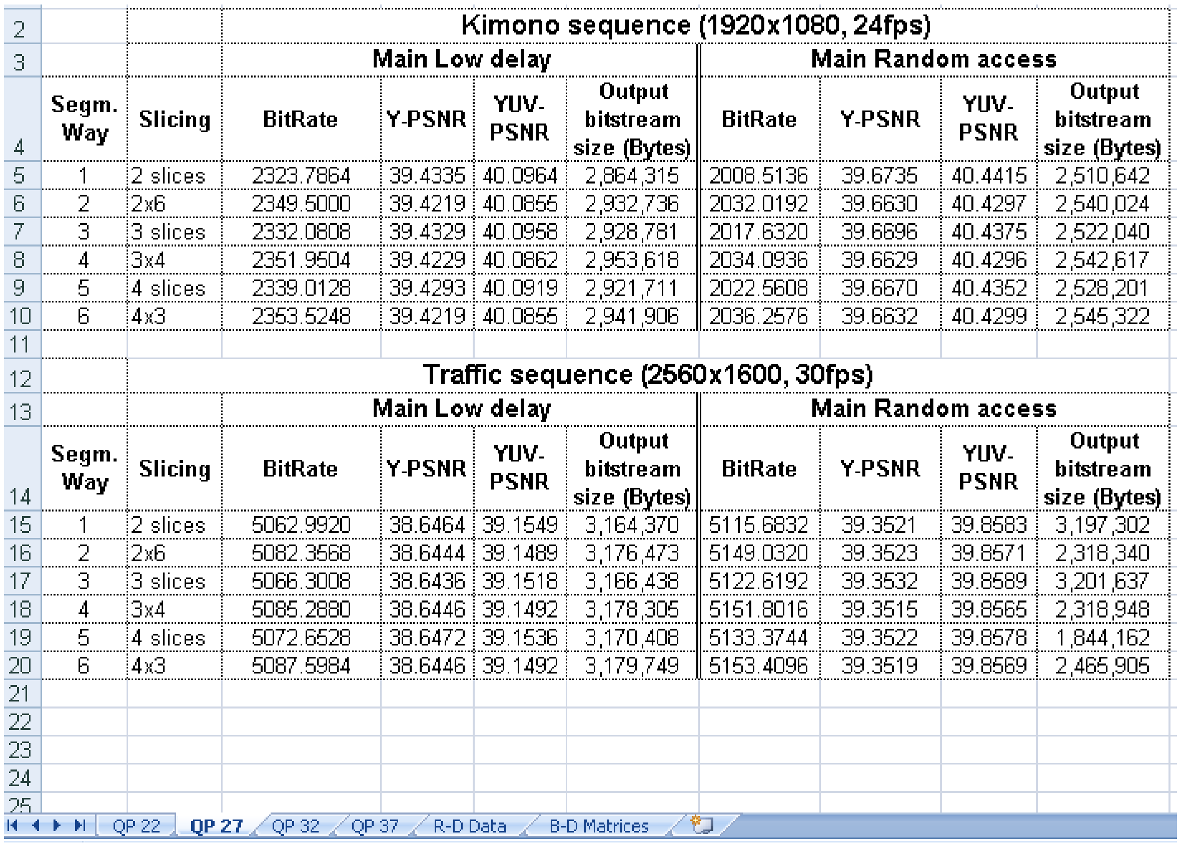
Summary of the encoding process

```
       SUMMARY ----------------------------------------------------------

      Total Frames |   Bitrate     Y-PSNR    U-PSNR    V-PSNR    YUV-PSNR

         240    a    5380.2400   41.5382   43.2229   44.7398   42.1568

Bytes written to file: 6725300 (5380.240 kbps)

Total Time:      621.360 sec.
```

Execution of the encoder on more powerful hardware

Result collection and analysis

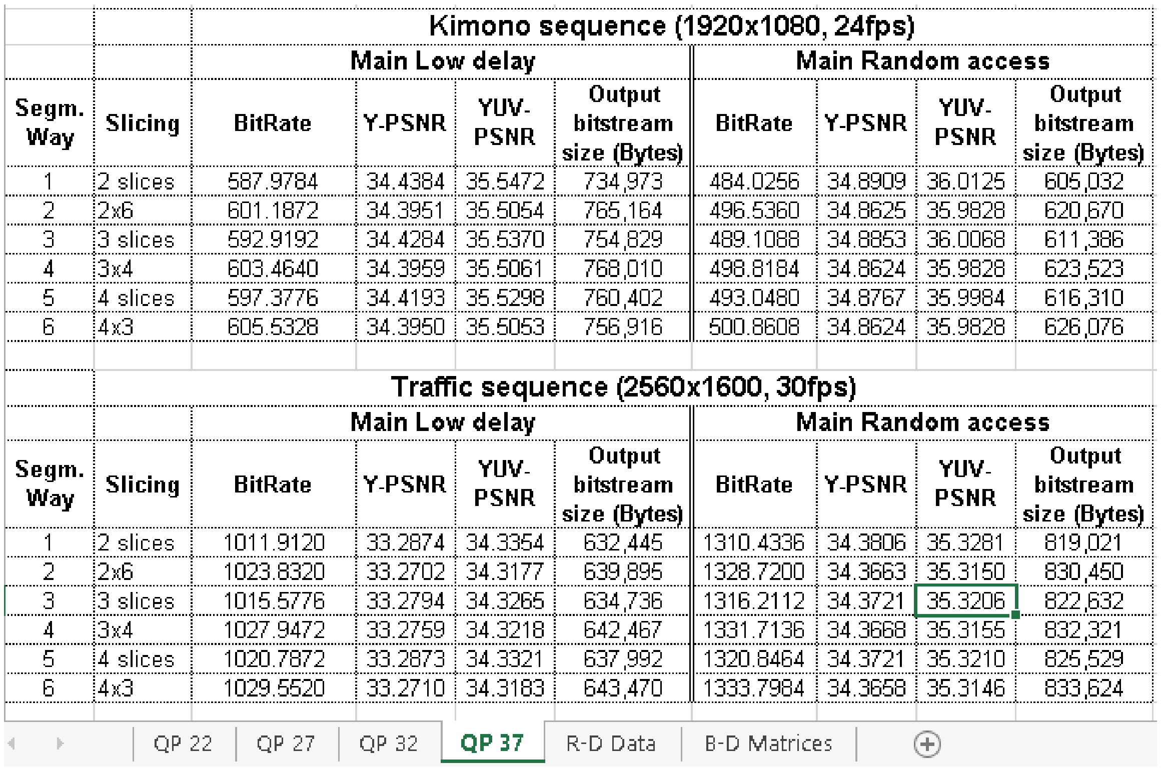
Thankfully, as we have seen, HEVC reference software does compute itself metrics as PSNR and BitRate during the encoding process. Thus, upon completion of all 96 runs, we gathered the results the encoder produced as output and created a spreadsheet like these below.



		Kimono sequence (1920x1080, 24fps)							
		Main Low delay				Main Random access			
Segm. Way	Slicing	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)
1	2 slices	2323.7864	39.4335	40.0964	2,864,315	2008.5136	39.6735	40.4415	2,510,642
2	2x6	2349.5000	39.4219	40.0855	2,932,736	2032.0192	39.6630	40.4297	2,540,024
3	3 slices	2332.0808	39.4329	40.0958	2,928,781	2017.6320	39.6696	40.4375	2,522,040
4	3x4	2351.9504	39.4229	40.0862	2,953,618	2034.0936	39.6629	40.4296	2,542,617
5	4 slices	2339.0128	39.4293	40.0919	2,921,711	2022.5608	39.6670	40.4352	2,528,201
6	4x3	2353.5248	39.4219	40.0855	2,941,906	2036.2576	39.6632	40.4299	2,545,322

		Traffic sequence (2560x1600, 30fps)							
		Main Low delay				Main Random access			
Segm. Way	Slicing	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)
1	2 slices	5062.9920	38.6464	39.1549	3,164,370	5115.6832	39.3521	39.8583	3,197,302
2	2x6	5082.3568	38.6444	39.1489	3,176,473	5149.0320	39.3523	39.8571	2,318,340
3	3 slices	5066.3008	38.6436	39.1518	3,166,438	5122.6192	39.3532	39.8589	3,201,637
4	3x4	5085.2880	38.6446	39.1492	3,178,305	5151.8016	39.3515	39.8565	2,318,948
5	4 slices	5072.6528	38.6472	39.1536	3,170,408	5133.3744	39.3522	39.8578	1,844,162
6	4x3	5087.5984	38.6446	39.1492	3,179,749	5153.4096	39.3519	39.8569	2,465,905

*Figure 26: Output data collection for **QP=27***


QP 22
QP 27
QP 32
QP 37
R-D Data
B-D Matrices

		Kimono sequence (1920x1080, 24fps)							
		Main Low delay				Main Random access			
Segm. Way	Slicing	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)
1	2 slices	587.9784	34.4384	35.5472	734,973	484.0256	34.8909	36.0125	605,032
2	2x6	601.1872	34.3951	35.5054	765,164	496.5360	34.8625	35.9828	620,670
3	3 slices	592.9192	34.4284	35.5370	754,829	489.1088	34.8853	36.0068	611,386
4	3x4	603.4640	34.3959	35.5061	768,010	498.8184	34.8624	35.9828	623,523
5	4 slices	597.3776	34.4193	35.5298	760,402	493.0480	34.8767	35.9984	616,310
6	4x3	605.5328	34.3950	35.5053	756,916	500.8608	34.8624	35.9828	626,076

		Traffic sequence (2560x1600, 30fps)							
		Main Low delay				Main Random access			
Segm. Way	Slicing	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)	BitRate	Y-PSNR	YUV-PSNR	Output bitstream size (Bytes)
1	2 slices	1011.9120	33.2874	34.3354	632,445	1310.4336	34.3806	35.3281	819,021
2	2x6	1023.8320	33.2702	34.3177	639,895	1328.7200	34.3663	35.3150	830,450
3	3 slices	1015.5776	33.2794	34.3265	634,736	1316.2112	34.3721	35.3206	822,632
4	3x4	1027.9472	33.2759	34.3218	642,467	1331.7136	34.3668	35.3155	832,321
5	4 slices	1020.7872	33.2873	34.3321	637,992	1320.8464	34.3721	35.3210	825,529
6	4x3	1029.5520	33.2710	34.3183	643,470	1333.7984	34.3658	35.3146	833,624

*Figure 27: Output data for **QP=37***

One of the first things to be noticed is the significant decline in output bitstream size. This is a clear example of the impact that QP value has on achieved compression rate (and therefore to coding efficiency) as cited in [18] and depicted in the graphs that follow.

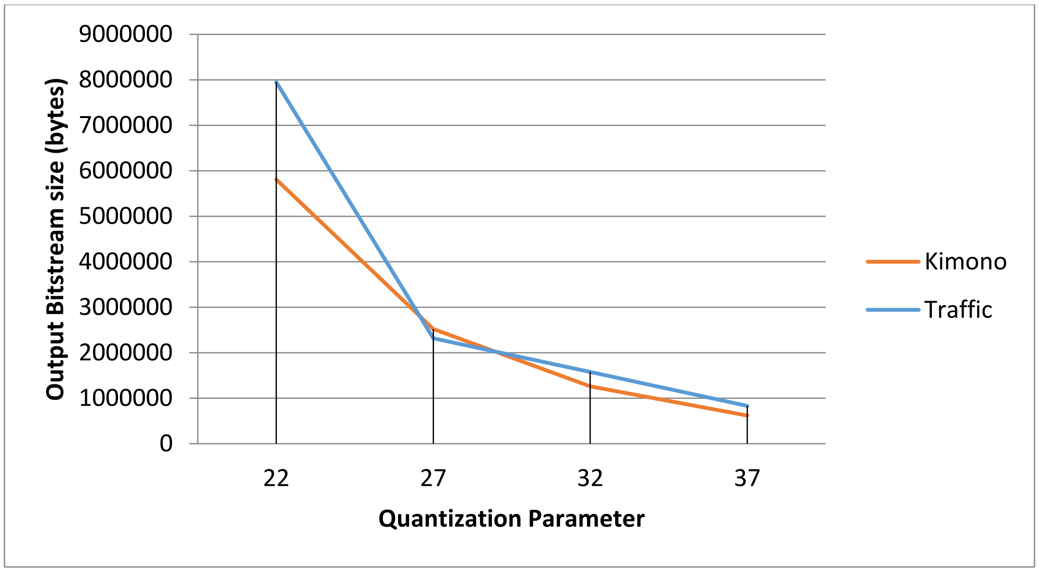
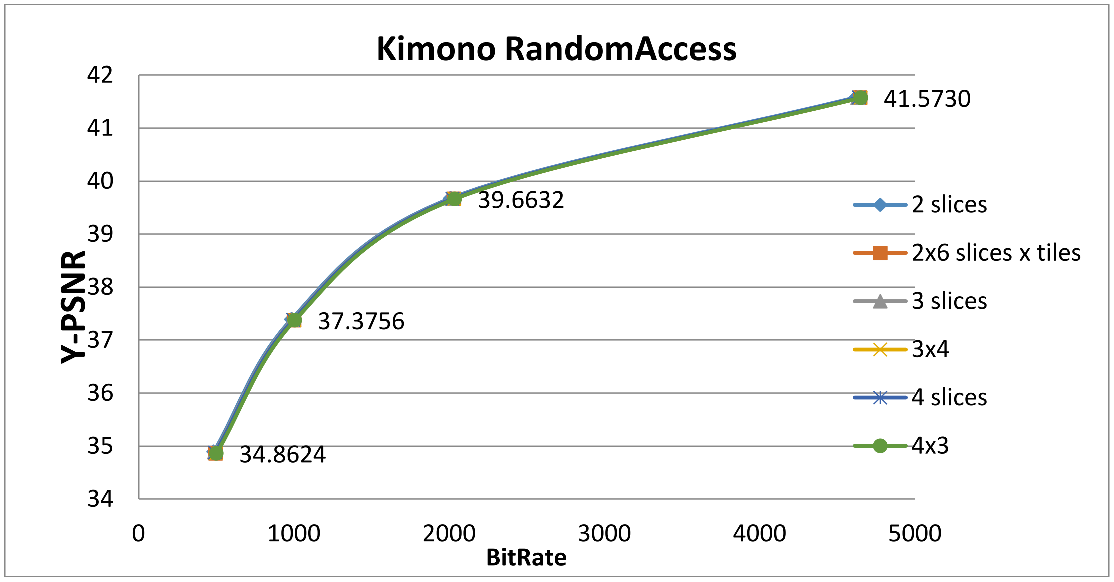
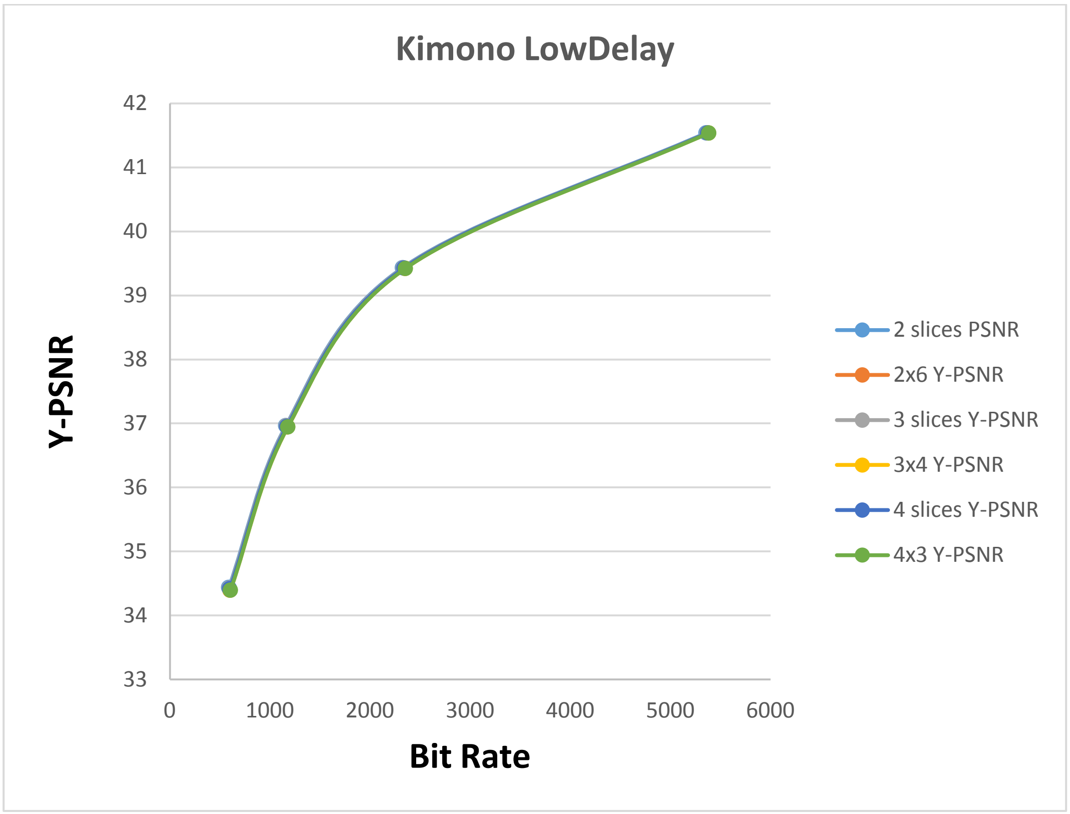
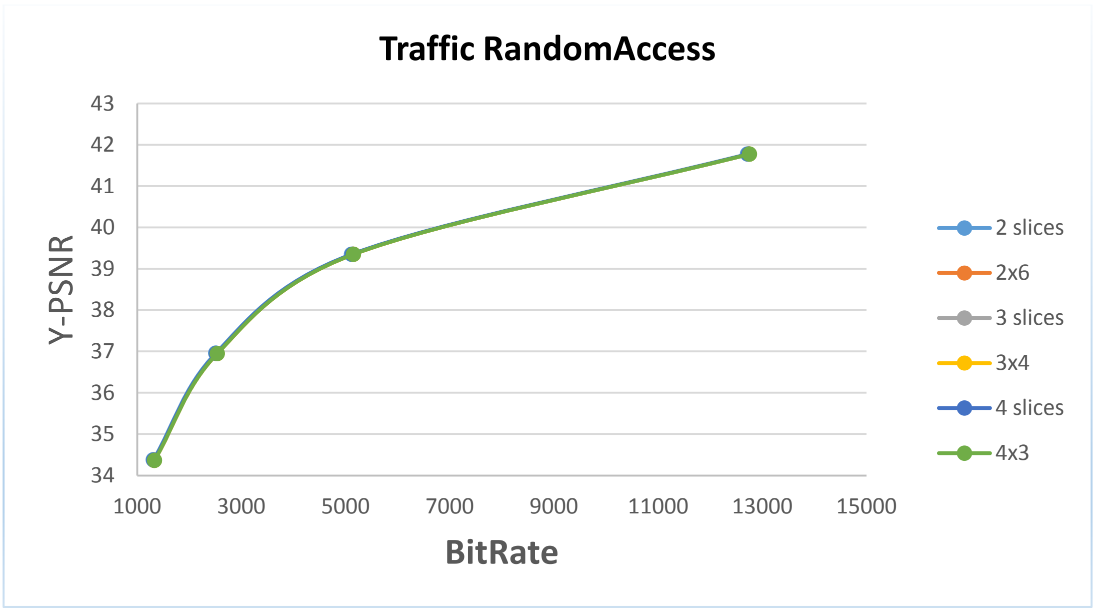


Figure 28: Impact of QP on coding efficiency

Using numeric data like those of the above spreadsheets (figures 27 & 28), we got Rate-Distortion graphs [38] as those depicted in the figures that follow.







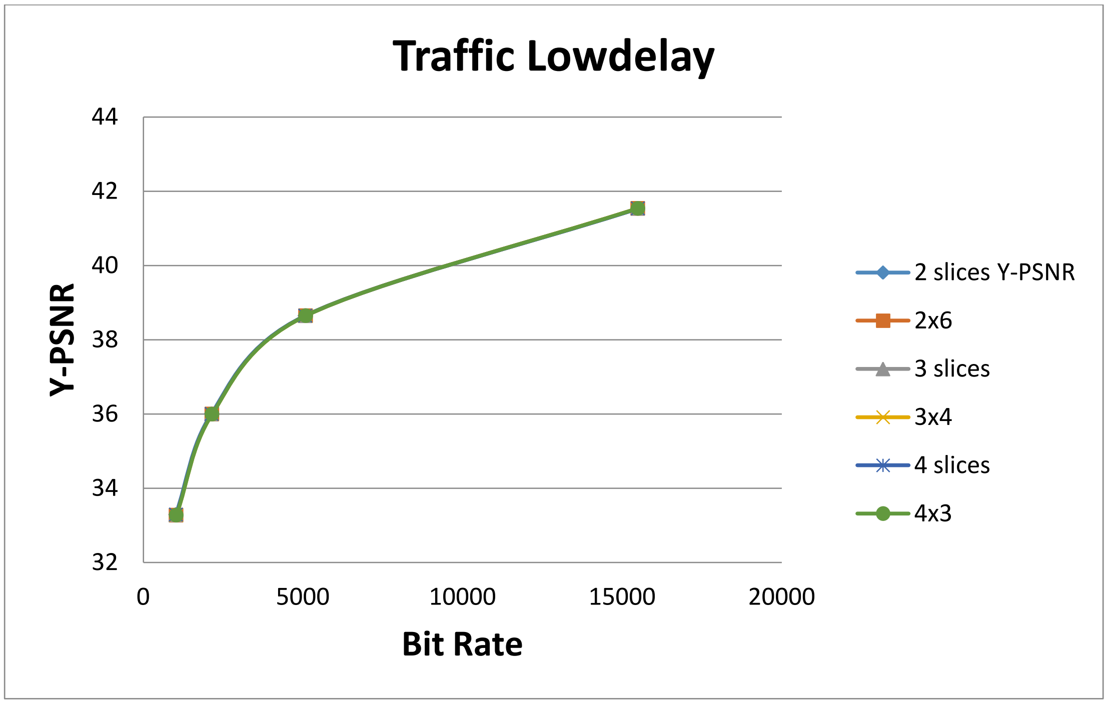


Figure 29: R-D curves demonstrating coding efficiency

Although we are supposed to read PSNR and bitrate differences between two simulation conditions in these RD plots, it is obvious that the existing differences are very hard to be distinguished. Therefore, we have to use a metric introduced in 2001 by Gisle Bjøntegaard, known as BD-PSNR [39]. Using this method, one can find the average difference between curves such as the above. Specifically, ***small BD-metric values indicate little context breaks and thus better quality. In other words the more the picture is partitioned, more contexts are broken and greater is the BD-rate increase*** [**31**].

In order to calculate BD-PSNR for our data, we used MatLab code developed by Giuseppe Valenzise in 2010, improved by Serge Matyunin in 2013 and it is freely available via GitHub [40]. We had simply to repeatedly call function `bjontegaard2()` with our results as input for both modes (‘*dsnr’* and ‘*rate’*). We have to explain here that this function’s last parameter (‘*mode’*) offers the option to calculate either the differences in Y-axis (PSNR), or in BitRate (X-axis). It is a string that can have two values:

'dsnr' - average PSNR difference or
'rate' - percentage of bitrate saving between data set 1 and data set 2
Figure 32 below demonstrates the use of the parameter.

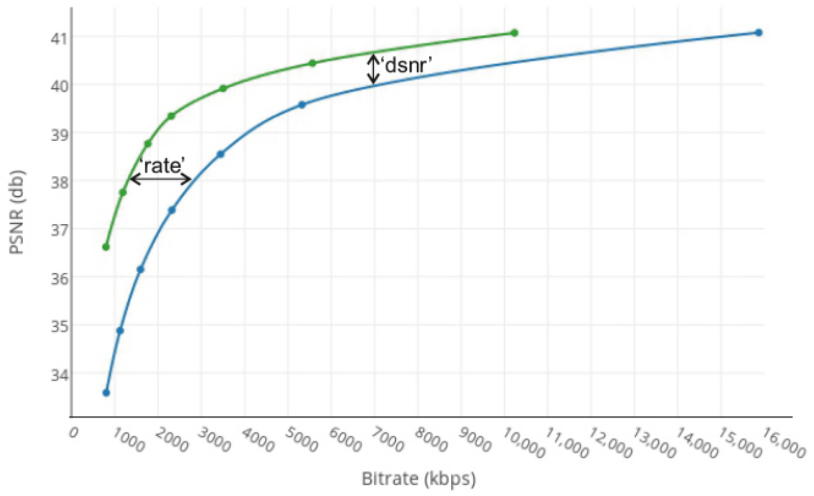
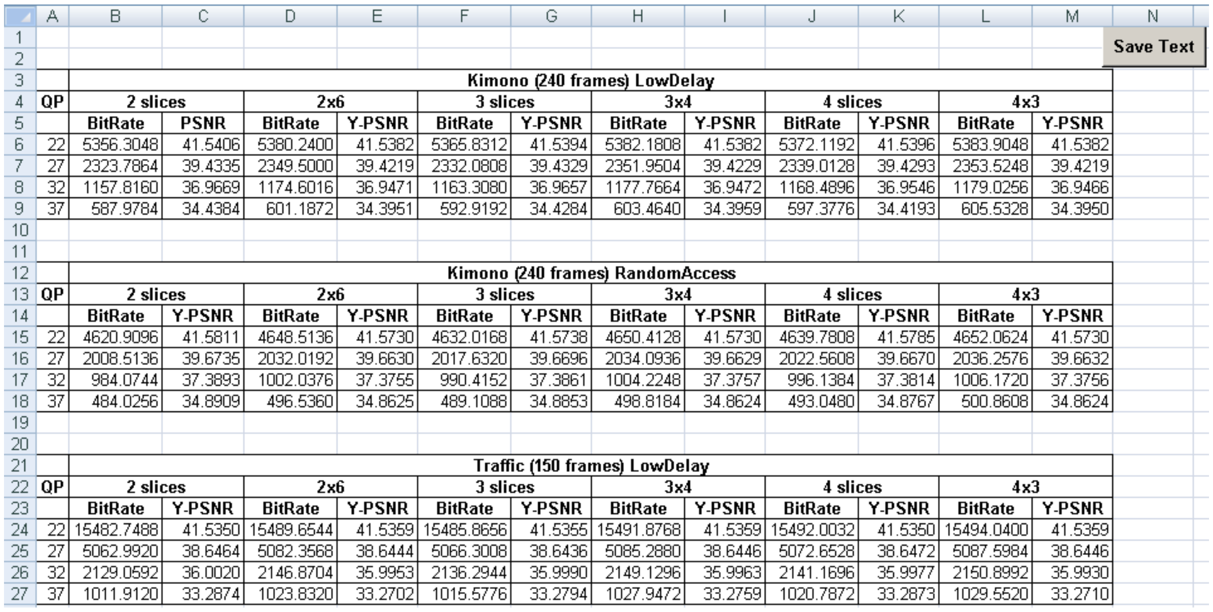


Figure 30: Sample R-D curve that shows the use of 'mode' parameter in bjontegaard2() function

We used GNU Octave [41] which interpretes and runs MatLab code to call `bjontegaard2()`. First we had to collect PSNRs and BitRates per segmentation way and QP value in a separate worksheet. Part of this worksheet is depicted below.



QP	Kimono (240 frames) LowDelay											
	2 slices		2x6		3 slices		3x4		4 slices		4x3	
	BitRate	PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR
22	5356.3048	41.5406	5380.2400	41.5382	5365.8312	41.5394	5382.1808	41.5382	5372.1192	41.5396	5383.9048	41.5382
27	2323.7864	39.4335	2349.5000	39.4219	2332.0808	39.4329	2351.9504	39.4229	2339.0128	39.4293	2353.5248	39.4219
32	1157.8160	36.9669	1174.6016	36.9471	1163.3080	36.9657	1177.7664	36.9472	1168.4896	36.9546	1179.0256	36.9466
37	587.9784	34.4384	601.1872	34.3951	592.9192	34.4284	603.4640	34.3959	597.3776	34.4193	605.5328	34.3950

QP	Kimono (240 frames) RandomAccess											
	2 slices		2x6		3 slices		3x4		4 slices		4x3	
	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR
22	4620.9096	41.5811	4648.5136	41.5730	4632.0168	41.5738	4650.4128	41.5730	4639.7808	41.5785	4652.0624	41.5730
27	2008.5136	39.6735	2032.0192	39.6630	2017.6320	39.6696	2034.0936	39.6629	2022.5608	39.6670	2036.2576	39.6632
32	984.0744	37.3893	1002.0376	37.3755	990.4152	37.3861	1004.2248	37.3757	996.1384	37.3814	1006.1720	37.3756
37	484.0256	34.8909	496.5360	34.8625	489.1088	34.8853	498.8184	34.8624	493.0480	34.8767	500.8608	34.8624

QP	Traffic (150 frames) LowDelay											
	2 slices		2x6		3 slices		3x4		4 slices		4x3	
	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR	BitRate	Y-PSNR
22	15482.7488	41.5350	15489.6544	41.5359	15485.8656	41.5355	15491.8768	41.5359	15492.0032	41.5350	15494.0400	41.5359
27	5062.9920	38.6464	5082.3568	38.6444	5066.3008	38.6436	5085.2880	38.6446	5072.6528	38.6472	5087.5984	38.6446
32	2129.0592	36.0020	2146.8704	35.9953	2136.2944	35.9990	2149.1296	35.9963	2141.1696	35.9977	2150.8992	35.9930
37	1011.9120	33.2874	1023.8320	33.2702	1015.5776	33.2794	1027.9472	33.2759	1020.7872	33.2873	1029.5520	33.2710

Figure 31: Summary of encoder output data

Then, we wrote Visual Basic macros that read BitRate and Y-PSNR vectors from the worksheet and produce a text file with the appropriate calls to bjontegaard2() (in MatLab syntax of course). These macros were called when clicking on button 'Save Text' (visible in the above figure). Thus, several text files were produced with calls like these below.

```
res1=bjontegaard2([ 12717.8192 , 5115.6832 , 2509.5408 , 1310.4336 ],[
41.7775 , 39.3521 , 36.9606 , 34.3806 ],[ 12750.0992 , 5149.032 ,
2533.56 , 1328.72 ],[ 41.776 , 39.3523 , 36.9497 , 34.3663 ],'dsnr');

res2=bjontegaard2([ 12717.8192 , 5115.6832 , 2509.5408 , 1310.4336 ],[
41.7775 , 39.3521 , 36.9606 , 34.3806 ],[ 12728.0832 , 5122.6192 ,
2515.9184 , 1316.2112 ],[ 41.7767 , 39.3532 , 36.9571 , 34.3721
],'dsnr');

res3=bjontegaard2([ 12717.8192 , 5115.6832 , 2509.5408 , 1310.4336 ],[
41.7775 , 39.3521 , 36.9606 , 34.3806 ],[ 12752.496 , 5151.8016 ,
2536.1888 , 1331.7136 ],[ 41.776 , 39.3515 , 36.9496 , 34.3668
],'dsnr');

res4=bjontegaard2([ 12717.8192 , 5115.6832 , 2509.5408 , 1310.4336 ],[
41.7775 , 39.3521 , 36.9606 , 34.3806 ],[ 12738.1888 , 5133.3744 ,
2522.6256 , 1320.8464 ],[ 41.7762 , 39.3522 , 36.9534 , 34.3721
],'dsnr');

res5=bjontegaard2([ 12717.8192 , 5115.6832 , 2509.5408 , 1310.4336 ],[
41.7775 , 39.3521 , 36.9606 , 34.3806 ],[ 12754.9024 , 5153.4096 ,
2539.096 , 1333.7984 ],[ 41.776 , 39.3519 , 36.9503 , 34.3658 ],'dsnr');
```

Those calls were pasted into Octave and eventually we had BD-PSNR metrics for every possible QP value and frame partinioning scheme. Figure 13 shows the data computed, with 2 slice partitioning being the "yardstick" in the first row, 2x6 is the reference for the second row etc. It is obvious that values at symmetric matrix positions are of opposite sign and thus there was no need to calculate again the average difference between, say, RD-curves of 2x6 vs. 2-sliced runs when we had done so for 2-sliced vs. 2x6 segmentation.

Bjontegaard Metric

Kimono LowDelay

Segm. Way	2 slices	2x6	3 slices	3x4	4 slices	4x3
2 slices	0	0.05966	0.01659	0.06540	0.03548	0.06987
2x6	-0.05966	0	-0.04318	0.00584	-0.02426	0.01040
3 slices	-0.01659	0.04318	0	0.04896	0.01896	0.05346
3x4	-0.06540	-0.00584	-0.04896	0	-0.03007	0.00457
4 slices	-0.03548	0.02426	-0.01896	0.03007	0	0.03459
4x3	-0.06987	-0.01040	-0.05346	-0.00457	-0.03459	0

Kimono RandomAccess

Segm. Way	2 slices	2x6	3 slices	3x4	4 slices	4x3
2 slices	0	0.060035	0.022022	0.065745	0.038253	0.070920
2x6	-0.060035	0	-0.038115	0.005816	-0.021886	0.011084
3 slices	-0.022022	0.038115	0	0.043861	0.016285	0.049067
3x4	-0.065745	-0.005816	-0.043861	0	-0.027670	0.0052872
4 slices	-0.038253	0.021886	-0.016285	0.027670	0	0.032911
4x3	-0.070920	-0.011084	-0.049067	-0.0052872	-0.032911	0

Traffic LowDelay

Segm. Way	2 slices	2x6	3 slices	3x4	4 slices	4x3
2 slices	0	0.022439	0.0084849	0.025030	0.012306	0.028514
2x6	-0.022439	0	-0.013994	0.0026518	-0.010156	0.006159
3 slices	-0.0084849	0.013994	0	0.016606	0.003847	0.016275
3x4	-0.025030	-0.0026518	-0.016606	0	-0.012779	0.0035112
4 slices	-0.012306	0.010156	-0.003847	0.012779	0	0.016275
4x3	-0.028514	-0.006159	-0.016275	-0.0035112	-0.016275	0

Bjontegaard Metric

Traffic RandomAccess

Segm. Way	2 slices	2x6	3 slices	3x4	4 slices	4x3
2 slices	0	0.031384	0.00854029	0.0347364	0.0177372	0.0369156
2x6	-0.031384	0	-0.022915	0.0033908	-0.013703	0.005597
3 slices	-0.0085403	0.022915	0	0.026285	0.009223	0.028476
3x4	-0.034736	-0.0033908	-0.026285	0	-0.017078	0.0022102
4 slices	-0.017737	0.013703	-0.009223	0.017078	0	0.019273
4x3	-0.036916	-0.005597	-0.028476	-0.0022102	-0.019273	0

Figure 32: All the calculated BD values

The above data (BD-metric values) are much more readable in barchart format. So, we decided to keep the 2 slices partitioning as a reference and produce the following graphs.

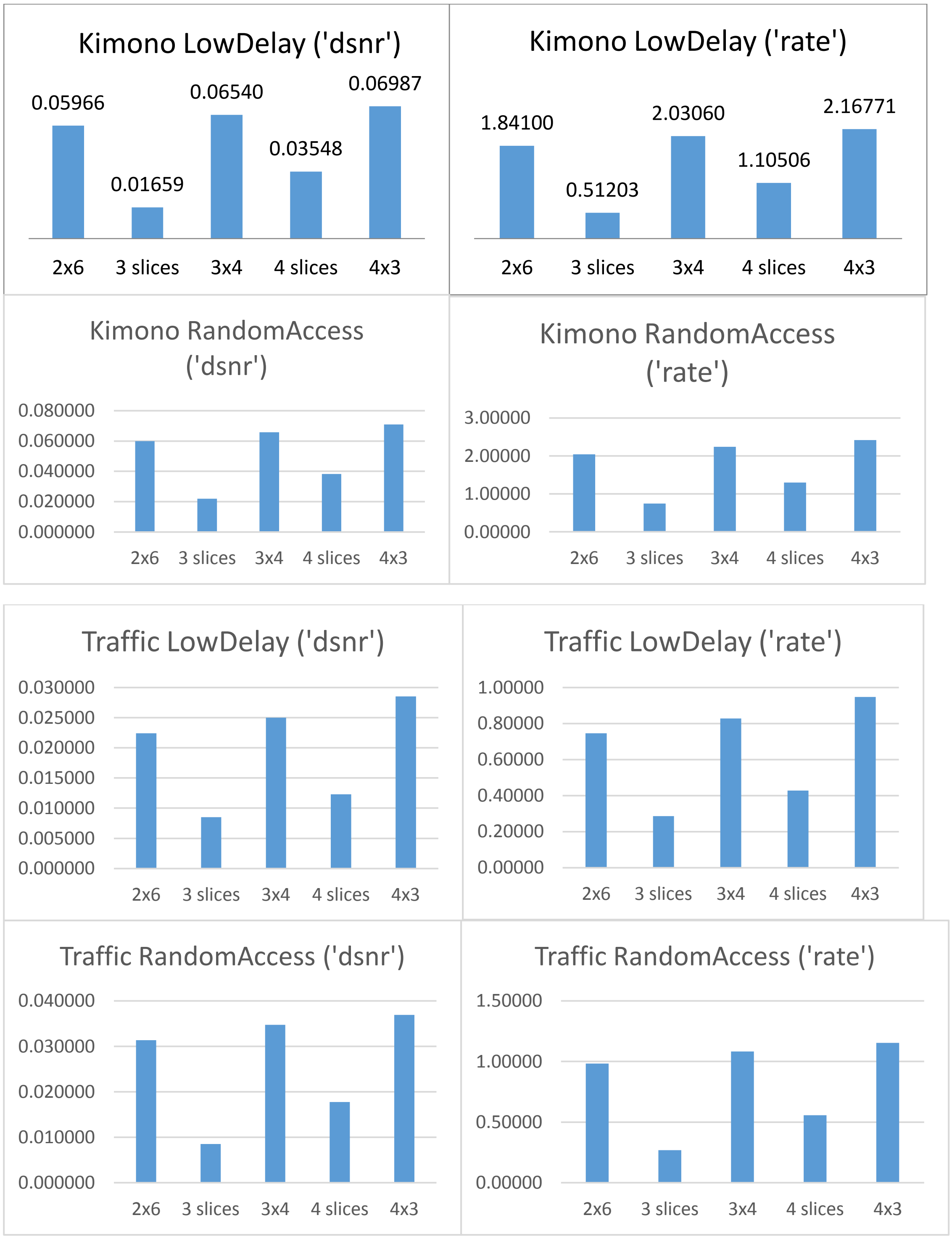


Figure 33: Bjontegaard metric barcharts

Conclusion and ideas for further research

A glance at the previously displayed barcharts shows that best coding efficiency is achieved with either no partinioning at all, or by using only slicing (no tiles at all). Yet, no partitioning is out of the question, since slicing and/or tiling is absolutely required to achieve efficient code parallelization and thus best temporal performance. Moreover, if we want to maximize parallelization, we also have to include tiles in the chosen partitioning scheme [31], [19], [23]. Thus, it seems that 2x6 (slices X tiles) is our best choice. In fact, load balancing is maximized if we use a hardware configuration of two (2) CPUs with six (6) cores each (e.g. Two Intel Xeon W-2133 CPUs [42]).

As we have seen in past works like [24], several partitionings have already been evaluated from the scope of load balancing and resource utilization in general besides the above we chose. So, it seems attractive, as a future project to further investigate CPU utilization achieved by our six (6) frame segmentations. In other words, a parallel scalability analysis like the one presented in [19] would **not** be meaningless.

References

[1] " Fraunhofer Institute for Telecommunications, Heinrich Hertz Institute, HHI," 2017. [Online]. Available: https://hevc.hhi.fraunhofer.de/.

[2] Bossen Fr., Flynn D., Sharman K., Sühring K., "JCTVC HM Software Manual," 21 1 2018. [Online]. Available: https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/trunk/doc/software-manual.pdf. [Accessed 3 2018].

[3] "Wikipedia (CGA)," [Online]. Available: https://en.wikipedia.org/wiki/Color_Graphics_Adapter.

[4] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, Thomas Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on circuits and systems for video technology,* vol. 22, no. 12, 2012.

[5] "Wikipedia Lossy compression," [Online]. Available: https://en.wikipedia.org/wiki/Lossy_compression.

[6] "Wikipedia Lossless compression," [Online]. Available: https://en.wikipedia.org/wiki/Lossless_compression.

[7] Sruthi S., Dr. Shreelekshmi R., "Video Compression - from Fundamentals to H.264 and H.265 Standards," *Int. Journal of Engineering and Computer Science ISSN:2319-7242, Vol. 4, Issue 7,* pp. 13468-13473, July 2015.

[8] R. I. Chernyak, "Analysis of the Intra Predictions in H.265/HEVC," *Applied Mathematical Sciences,* vol. 8, no. 148, pp. 7389-7408, 2014.

[9] M. B. Vivienne Sze, "Design and Implementation of Next Generation Video Coding Systems (H.265/HEVC Tutorial)," *ISCAS Tutorial,* 2014.

[10] K. D. H. J. Rao K. R., High Efficiency Video Coding(HEVC), Dordrecht: Springer, 2014.

[11] G. Dimopoulos, "IMPLEMENTATION OF HEVC (H.265) VIDEO ANALYSIS TOOL," Lamia, 2017.

[12] Wiegand Th., Sullivan G.J., Bjontegaard G., Luthra A., "Overview of the H.264/AVC Video Coding Standard," *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY,* July 2003.

[13] I. E. Richardson, The H.264 Advanced Video Compression Standard, 2nd ed., Wiley and Sons, 2010.

[14] Fu Chih-Ming, Alshina Elena, Alshin Alexander, Huang Yu-Wen, Chen Ching-Yeh, and Chia-

Yang Tsai, Chih-Wei Hsu, Lei Shaw-Min, Park Jeong-Hoon, Han Woo-Jin, "Sample Adaptive Offset in the HEVC Standard," *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY,* vol. 22, no. 12, Dec. 2012.

[15] Corrêa G., Assunção P., Agostini L., da Silva Cruz L.A., Performance and Computational Complexity Assessment of HEVC. In: Complexity-Aware High Efficiency Video Coding, Springer, Cham, 2016.

[16] M. V. T. D. H. A. H. Jarno Vanne, "Comparative Rate-Distortion-Complexity Analysis of HEVC and AVC Video Codecs," vol. 22, no. 12, Dec. 2012.

[17] Ohm Jens-Rainer, Sullivan G. J., Schwarz H., Thiow Keng Tan Th. K., Wiegand Th., "Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC)," *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY,* vol. 22, no. 12, pp. 1669-1684, Dec. 2012.

[18] B. Benny, Next-Generation Video Coding and Streaming, John Wiley & Sons, 2015.

[19] Chi Ching Chi, Mauricio Alvarez-Mesa, Ben Juurlink, Gordon Clare, "Parallel Scalability and Efficiency of HEVC Parallelization Approaches," vol. 22, no. 12, 2012.

[20] P. S. Henry F., "Wavefront Parallel Processing," in *Joint Collaborative Team on Video Coding (JCT-VC) - JCTVC-E196*, Geneva, 2011.

[21] Heng Tse Kai, Asano W., Itoh Tak., Tanizawa Ak., Yamaguchi Jun, Matsuo Tak., Kodama Tom., "A HIGHLY PARALLELIZED H.265/HEVC REAL-TIME UHD SOFTWARE ENCODER," in *IEEE International Conference on Image Processing (ICIP)*, Paris, 2014.

[22] K. Misra, A. Segall, M. Horowitz, S. Xu, A. Fuldseth and M. Zhou, "An overview of tiles in HEVC," *IEEE Journal of Selected Topics in Signal Processing,* vol. 7, no. 6, pp. 969-977, December 2013.

[23] Maria Koziri, P. Papadopoulos P, N. Tziritas, A. N. Dadaliaris, Thanasis . Loukopoulos, S. U. Khan and C. Z. Xu, "Adaptive Tile Parallelization for Fast Video Encoding in HEVC," in *12th Int. Conf. on Green Computing and Communications (GreenCom 2016)*, Kos, Greece, 2016.

[24] Migallón H., Piñol P., López-Granado O., Galiano I., Malumbres M.P., "Performance analysis of frame partitioning in parallel HEVC encoders," *Journal of Supercomputing,* 10 Jan. 2017.

[25] Mitra G., Johnston B., Rendell A.P., McCreath E., Zhou Jun, "Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms," in *IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*, 2013.

[26] Yang Lu, Qi Zhang, Bin Wei, "Real-Time CPU Based H.265/HEVC Encoding with x86 Platform Technology," in *International Conference on Computing, Networking and Communications*

(ICNC), Workshop on Computing, Networking and Communications (CNC), 2015.

[27] Chen Keji, Duan Yizhou, Yan Leju , Sun Jun, "Efficient SIMD Optimization of HEVC Encoder over X86 Processors," Beijing 100871, China.

[28] Fraunhofer Institute for Telecommunications, Heinrich Hertz Institute, HHI, "Wavefronts for HEVC Parallelism," [Online]. Available: https://www.hhi.fraunhofer.de/index.php?id=543&L=1. [Accessed March 2018].

[29] P. Piñol, H. M. Gomis, O. M. L. Granado, M. P. Malumbres, "Slice-based parallel approach for HEVC encoder," *Journal of Supercomputing,* vol. 71, no. 5, pp. 1882-1892, 2015.

[30] M. Z. Sze and Madhukar Budagavi, *Parallel tools in HEVC for high-throughput processing,* vol. 8499, 2012, pp. 8499 - 8499 - 13.

[31] Malossi G., Palomino D., Diniz Cl., Susin A., Bampi S., "Adjusting Video Tiling to Available Resources in a per-frame Basis in High Efficiency Video Coding," in *New Circuits and Systems Conference (NEWCAS), 2016 14th IEEE International*, Vancouver, BC, Canada, 2016.

[32] C. Lattner, "Introduction to the LLVM Compiler System," 4 11 2008. [Online]. Available: https://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf. [Accessed 2018].

[33] F. Bossen, "“Common Test Conditions and Software Reference Configurations”," *document JCTVC-H1100, JCT-VC,* Feb. 2012.

[34] M. Wien, High Efficiency Video Coding. Coding Tools and Specification, HeidelBerg: Springer-Verlag, 2015.

[35] "Wikipedia (BASH shell)," [Online]. Available: https://en.wikipedia.org/wiki/Bash_(Unix_shell).

[36] "Linux manpages (taskset command)," [Online]. Available: https://linux.die.net/man/1/taskset.

[37] Free Software Foundation - GNU project, "sed, a stream editor," GNU project, [Online]. Available: https://www.gnu.org/software/sed/manual/sed.html.

[38] S. Akramullah, Digital Video Concepts, Methods and Metrics: Quality, Compression, Performance, and Power Trade-off Analysis, Apress, 2014.

[39] G. Bjontegaard, "Calculation of average PSNR differences between RD-Curves. Proceedings of the ITU-T Video Coding Experts Group (VCEG)," in *VCEG-33*, 2001.

[40] M. S. Giuseppe Valenzise G., "GitHub (bjontegaard2)," 2013. [Online]. Available: https://github.com/serge-m/bjontegaard2. [Accessed March 2018].

[41] Free Software Foundation - GNU project, "GNU Octave," GNU project, 2017. [Online].

Available: https://www.gnu.org/software/octave/.

[42] Intel Corporation, "Product Specifications," Intel, [Online]. Available: https://goo.gl/wpzFGp.

[43] "Wikipedia (DCM)," [Online]. Available: https://en.wikipedia.org/wiki/Discrete_cosine_transform.

[44] "Wikipedia (DSM)," [Online]. Available: https://en.wikipedia.org/wiki/Discrete_sine_transform.

[45] J. Martínez, P. Cuenca, F. Delicado and F. Quiles, "Objective video quality metrics: A performance analysis," 3 2018.

[46] Cebrián-Márquez G., Hernández-Losada J. L., Martínez J.L., Cuenca P., Tang M., Wen J., "Accelerating HEVC using heterogeneous platforms," *Journal of Supercomputing,* no. 71, 2015.