POLITECNICO DI TORINO

Master's Degree Course
in Computer Engineering
(Graphics and Multimedia)

Master's Degree Thesis

Efficient Extraction of Motion Vectors
from H264 Video Streams

Supervisor:
Professor Enrico Masala

Candidate:
Cosmin Angheluta

Academic Year 2020/2021

In collaboration with Addfor S.p.A.

Abstract

Many modern applications use video analysis, and in particular motion analysis, to achieve different purposes, from motion recognition to motion prediction. In this regard, a common interest is to speed up the extraction of this type of data, in the form of motion vectors, from the most commonly used video standard: the H264 Standard.

The typical H264 decoders, in particular FFmpeg, decode the entire video to extract the motion vectors - a process that is highly expensive in terms of timings. For this reason, the thesis studies the possibility of building an enlightened version of a decoder composed only of the necessary parts for motion extraction (mainly entropy decoding).

The first part of the thesis analyzes this possibility by simplifying the available FFmpeg decoder without optimizing its performance, while the second part deals with the construction of an extractor, following the ITU-T H264 Standard, and the management of motion vectors.

The final extractor performs better than the entire decoder, reaching the expectations of an improvement that performs 29% better with the CAVLC and 20% with CABAC entropy codings.

Also, as a side effect, the memory usage is largely improved, since the data for the entire frame is not needed anymore. Here, the improvement is even bigger (91% for CAVLC and 78% for CABAC), making this algorithm perfect for simple or embedded systems.

Index

Chapter 1 - Introduction **5**

Chapter 2 - Background **8**

Section 2.1 - H264/AVC (Advanced Video Codec) 9

Section 2.1.1 - Spatial and Temporal Redundancy 9

Section 2.1.2 - Profiles and Levels 10

Section 2.1.3 - H264 Encoder/Decoder 11

Section 2.2 - Bitstream and Network Abstraction Layer 12

Section 2.3 - FFmpeg 14

Section 2.3.1 - FFmpeg Functions 14

Section 2.3.2 - Structures 15

Chapter 3 - Intermediate Project **16**

Section 3.1 - Decoder of FFmpeg 16

Section 3.2 - Chain of functions 16

Section 3.2.1 - main and read_rtsp_stream 16

Section 3.2.2 - h264_decode_frame and decode_nal_units 17

Section 3.2.3 - ff_h264_execute_decode_slices 18

Section 3.2.4 - decode_slice 18

Section 3.2.5 - ff_h264_decode_mb_cavlc and ff_h264_decode_mb_cabac 19

Section 3.5 - Adjustments of the FFmpeg library 22

Chapter 4 - Motion Vector Extraction **24**

Section 4.1 - Bitstream Reading and NAL Identification 24

Section 4.1.1 - Management functions 24

Section 4.1.2 - Reading and Skipping functions 25

Section 4.1.3 - Extracting NAL units from the bitstream 26

Section 4.1.3.1 - Annex B NAL Splitting 26

Section 4.1.3.2 - AVCC NAL Splitting 26

Section 4.2 - Environment Elements 26

Section 4.2.1 - SPS and PPS NAL Units 27

Section 4.2.2 - Slice Headers 27

Section 4.2.3 - Slice and Stream Contexts 28

Section 4.3 - Motion Vector Prediction 29

Section 4.4 - DPB, RPLR and MMCO 30

Section 4.5 - CAVLC and CABAC 31

Section 4.4.1 - CAVLC (Context-Adaptive Variable-Length Coding) 32

Section 4.4.1.1 - Exp-Golomb Entropy Coding 33

Section 4.4.1.2 - CAVLC Encoding for Residual Information 33

Section 4.4.1.3 - CAVLC Decoding 36

Section 4.4.2 - CABAC (Context Adaptive Binary Arithmetic Coding) 40

Section 4.4.2.1 - CABAC Decoding 41

Section 4.4.2.2 - Initialization 41

Section 4.4.2.3 - Binarization 42

Section 4.4.2.4 - Decoding Process 44
Section 4.4.2.5 - Example of CABAC coding process 45

Chapter 5 - Temporal & Memory Usage Analysis 47
Section 5.1 - Measurements and Descriptions 47
Section 5.2 - Difference between RAM Availability 48
Section 5.3 - Complete Decoding and MV Extraction Analysis 49
Section 5.4 - Comparison between CAVLC and CABAC 55
Section 5.5 - Difference in the achieved time 56
Section 5.6 - Memory Usage 57

Chapter 6 - Conclusion 60

Acknowledgments 62

Bibliography and Sitography 63

Chapter 1 - Introduction

An open challenge in video analysis is to efficiently extract the information about the movement in the video without performing a complete decode of the frames. This information, in the most recent video codecs, is stored in motion vectors: particular structures that indicate the movement of an object from a frame to another.

Efficiency, in this case, is the ability to obtain the motion vectors in the fastest way possible, since modern decoders, like the one in libavcodec of FFmpeg, need a complete decoding process to extract them. So the purpose of the thesis is to reduce the decoding part to the minimum, and optimize the reading of the bitstream to speed up the extraction process.

Currently, the extraction of motion vectors from a video stream is performed mostly in pixel domain, which means that the entire frame has to be decoded in order to extract the information about the motion vectors (side data). This traditional approach decodes the received stream to gain access to every single pixel within a video frame, so it needs to store the pixels in a frame buffer to access them during the analysis. In this process, every information about the data is decoded, including luminance, color, and texture information.

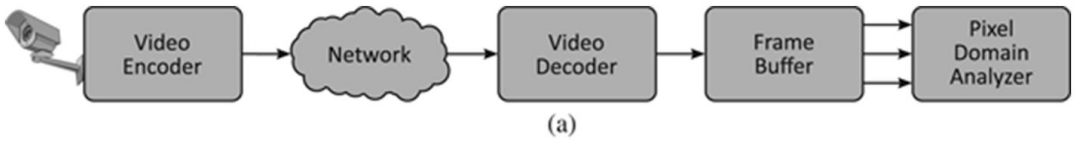

Figure 1.1: Pixel Domain Decoding Chain[1]

Still, with the growth in importance of this information, new approaches have been developed to extract motion vectors from the video stream, without having to decode the entire video.

Those new approaches are different one from another in terms of what they need to achieve, but they all have a common basis: a high awareness of the scheme used to encode the data, so as to correctly read the information it contains without misinterpreting it.

Also, since the new approaches analyze the bitstream directly, and the bitstream contains the compressed data of a video, the analysis is called in “compressed domain”.

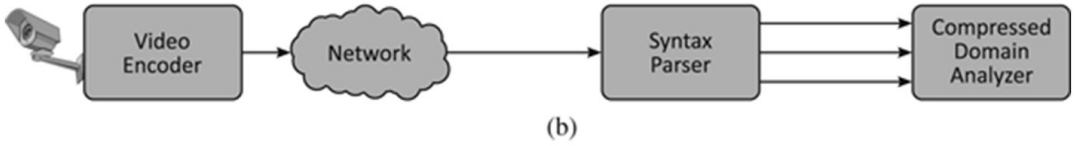

Figure 2: Compressed Domain Decoding Chain[1]

Different from the pixel domain, the compressed domain approach does not perform complex decoding operations like motion compensation, so it results being less time and memory consuming, and much faster.

However, to extract useful information from a video, another characteristic has to be considered. The movement in a frame can be generated in two different ways - one involving the movements of the camera which make the entire view translate, and one involving the movement of the objects in the frame with a fixed camera.
In the first case, little information can be extracted, since every part of the frame will be filled with motion vectors, and no real information about the real movement can be retrieved. In the second case instead, the motion vectors will be limited to the moving objects of the camera, so more data could be extracted from the bitstream considering the magnitude of the vectors and their directions. By doing that, a different type of movement could be differentiated by applying thresholds and filters on the vectors and extracting only those with a particular behavior.

The algorithm presented in this paper (Chapter 4) performs a partial decoding process in a mixed approach that uses both the pixel and the compressed domain of the H264/AVC (Advanced Video Coding) video standard, starting from the example of the well-known FFmpeg library (Chapter 3).
The scheme of the algorithm will follow the FFmpeg guideline - with a linear structure of functions that read an H264 stream from the most external package to the most internal information. Starting with the entire video stream, divided into NAL units in an RBSP connection, to the single slices, the macroblocks, and finally the motion vectors.
Also, since these actions are already performed by the FFmpeg library, a first solution will light those functions removing all the useless parts of the decoding process, while the final

solution will completely use a specifically designed library that allows using the least amount of memory and overall computational power possible.

Following this structure and profiling the FFmpeg library, the extraction of the motion vectors should improve up to 74% of the pure decoding part, since the CABAC/CAVLC decoding (first part of the process) takes only 23,64% of the pixel domain decoding process (Figure 1.1). Still, the figure does not represent the entire decoding process, which involves the storage of more information about the frames to correctly maintain the caches of the motion vectors.

In general, considering the entire decoding process, the percentages are:

- 6,40% for entropy decoding, not removable.
- 15,13% for motion estimation/compensation, which is removable.
- 8,47% for filtering functions, which are removable.

With these assumptions, the overall expected improvement should be around 23,60%.

Figure 1.1 - FFmpeg Profiling [2]

(Entropy Decoding 23,64% - Motion Compensation 51,85% - Loop Filter 22,22%)

The success of the thesis will lighten every process that needs to study the motion vectors in a video. Decoding a video file is a high-consuming activity, especially considering the new developments in the machine learning field for motion prediction and recognition. In these situations, the reduction of the extraction time could make the difference, even more if the application has a timing risk factor, like for supervising systems.

In this specific case, a supervising system generates uninterrupted and numerous streams of video (one for each camera). An approach that first decodes the video and then calculates the motion vectors needs a resourceful system, both in terms of memory and computing capacities. For this reason, a second approach that extracts the motion vectors directly from the bitstream could improve the entire process twice: one in terms of the storage memory (the videos does not need storing), and one in terms of timing (the decoding process and the following extraction will be skipped or partially skipped).

Chapter 2 - Background

The reason behind the choice of using a mixed approach algorithm involves many restrictions of the compressed domain.
First of all, there is very little information about the compressed domain extraction of motion vectors from the bitstream of H264 videos. Even if it is theoretically possible, there are no well-known algorithms that perform this extraction. Also, many partial compression schemes exist, and each of them requires specific limits on the range of information they can extract.
A compressed analysis of the bitstream, without any structural decoding involved, could extract only a few pieces of information from the video, like the global direction of the movement or the detection of changes in a static scene.

When more specific information about the video needs to be extracted, like the motion of objects in the scene, a more complex structure is needed.
In their study (Moving Object Detection in the H264 Compressed Domain, IEEE 2010), Kapotas and Skodras described a structure analyzing the compressed domain that calculates relative motion vectors between frames and filters them through a threshold. This operation allows the algorithm to distinguish between foreground and background and eliminate all kinds of micro-movements.
However, in a video, motion vectors do not represent real movements, but the offset of a macroblock between frames. This value could be the result of a moving object, the motion of the camera, or even the repetition of similar macroblocks. To prevent that and analyze only the parts in *real motion,* the videos studied by Kapotas and Skodras are all shot with a fixed-camera.

The objective of this thesis is quite different.
Instead of analyzing only the moving objects, the discussed algorithm will extract all the motion vectors and retrieve them frame by frame in dedicated structures.
This extraction will perform a complete bitstream parse, plus an entropy decoding step, removing from the equation everything that follows.

Section 2.1 - H264/AVC (Advanced Video Codec)

Section 2.1.1 - Spatial and Temporal Redundancy

The H264/AVC codec is a video standard that allows a video to be compressed and transmitted with more ease than the uncompressed version. The standard aims to achieve two different goals:

1. Allow the transmission between two systems that use different languages by providing a common language.
2. Reduce the number of bits transferred through the communication channel to relieve the throughput of the network.

In particular, this last operation is the most interesting characteristic of the standard - the higher is the compression rate, the higher is the amount of information that the network can transmit in a specific range of time.

AVC is the fourth generation of algorithms of its kind (MPEG-4), and it uses mainly spatial and temporal redundancy to compress the frames of a video.

First, the encoder splits each frame into macroblocks, which are groups of pixels of fixed dimension. Then, each macroblock is encoded entirely from scratch or by using one of the redundancies available.

The redundancy of a macroblock represents its repeatability. In spatial redundancy, the encoder finds a repetition between the neighboring blocks of the same frame. This type of prediction is called intra-prediction. Temporal redundancy, instead, involves repetitions between macroblocks of different frames. In this case, the prediction is called inter-prediction.

The inter-prediction is the focus of the thesis.

This prediction contains the motion vectors - a two-dimensional structure that provides the offset of movement for a macroblock from its reference.

Also, to keep track of the type of predictions allowed, every frame is labeled as I, P, or B, based on the macroblocks it can contain.

I-frames (or Intra) are the most expensive ones because they have a low compression ratio. They allow intra-prediction but forbid inter-prediction. They also provide a starting point for the decoding, allowing operations like skipping of a video or resynchronization in case of frame loss over the network.

Contrarily, P and B-frames allow both intra and inter-prediction, providing a better compression rate. Both of them need a reference frame from which to rebuild their information. The referenced frame can be of type I and P, or in some cases even B. In particular, P-frames make use of only one reference frame, while B-frames use two references.

Besides that, frames can also allow skip macroblocks (SI, SP-frames). Skip macroblocks are particular blocks for which the encoder does not encode anything in the bitstream, so they have to be derived completely from their neighbors or their references.

Also, a distinction between decoding and display order is necessary to understand P and B-frame. The display order (Figure 2.1) is the sequence of frames shown in the reproduction process of a video. Contrarily, the decoding order is the sequence of frames read by the decoder.

In display order, P and B frames can reference both past or future pictures, while in decoding order they can only reference the past.

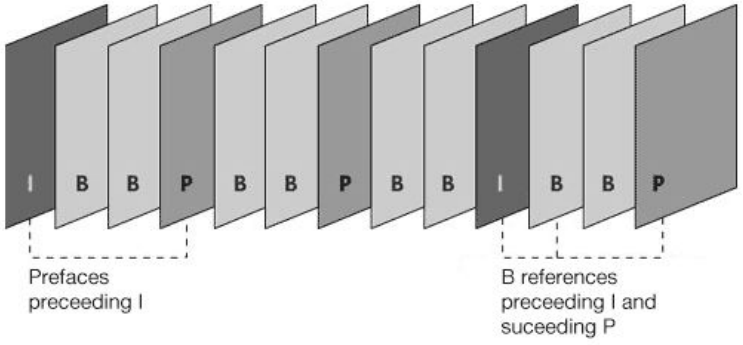

Figure 2.1 - I,P,B Frames in Display Order[3]

These are the main concepts behind compression. A process that ultimately allows obtaining performance advantages for video transmission both with low and high bandwidths.

Section 2.1.2 - Profiles and Levels

Another characteristic of the H264/AVC standard is its versatility, which allows its usage even in environments with different resources. In this regard, the standard contains many profiles and levels, each one with its characteristics and performances.

The H264 profiles specify conformance points to achieve interoperability between different applications with similar requirements. The levels, instead, set a range of constraints on the main parameters of the bitstream.

The most used profiles are currently the Baseline, Main, Extended and High profiles. All of them provide a set of functions and tools, including different entropy decoding techniques.
Context Adaptive Binary Arithmetic Coding (CABAC) and Context Adaptive Variable Length Coding (CAVLC) are allowed in some profiles and forbidden in others. The Baseline and Extended profiles, for example, only accept CAVLC entropy coding, while Main and High can use both CAVLC and CABAC.

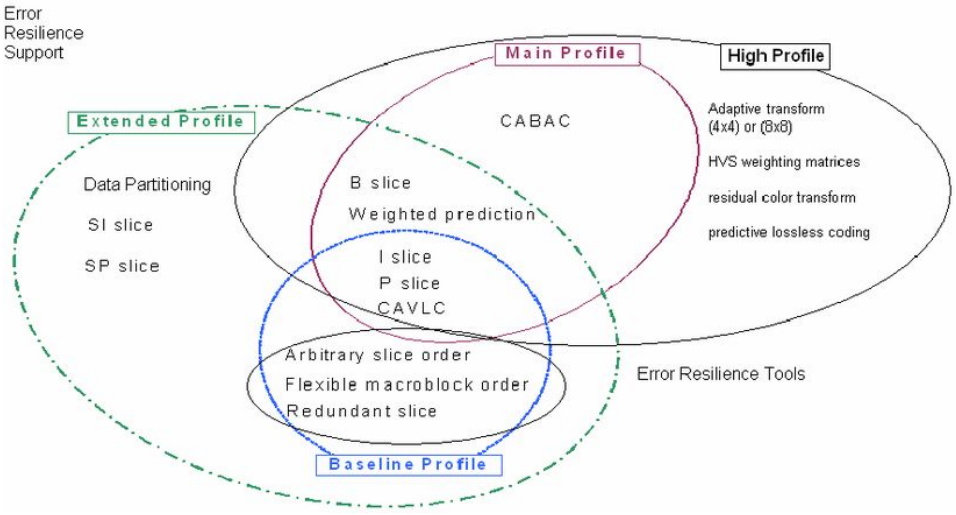

Figure 2.2 - Distribution of profiles in H264[4]

Levels are not quite important for the goal of the thesis, since they define restriction on variables like the maximum sample rate, bitrate, or compression rate. The only meaningful information for the extraction is the length of the decoded picture buffer (DPB). This structure will store the caches for each frame until the picture is de-referenced.

Section 2.1.3 - H264 Encoder/Decoder

The H264 standard exploits temporal redundancy by subdividing the frames in macroblocks of 16x16, 16x8, or 8x16 dimensions. These blocks can be subdivided even more into sub-blocks down to 4x4.
Every block is paired to a couple of motion vectors, one for each of the cardinal axes. Also, each block contains information about the luminance and the chrominance of its pixels, which represent the light intensity and the color respectively.

An H264 decoder comprises different blocks that allow the conversion from the bitstream to the syntax elements of the video.

The first step of the process is the parser, able to read and retrieve one or more bits at a time. Follows the entropy decoder, which is the core de-compression tool of the bitstream. Then, the inverse quantizer allows the evaluation of the coefficients of the Discrete Fourier Transform.

The decoder stores the motion vectors inside the bitstream, without compression, but the parser has to read every bit to maintain the synchronization.

The blocks after the inverse quantizer, included, are skipped in the extraction process. The linear transform block allows the definition of the data in the spatial domain starting from the inversely scanned coefficients. Then, depending on inter or intra-prediction, the motion compensation process builds the final frame using the referenced pictures. Also, a deblocking filter performs a smoothing operation on the borders of the blocks to smooth the edges and better the image.

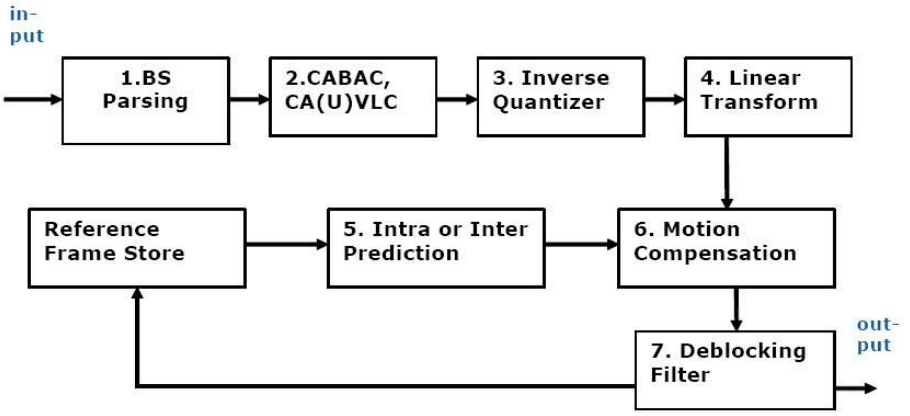

Figure 2.3 - H264 Decoder[5]

Section 2.2 - Bitstream and Network Abstraction Layer

The encoding of the video allows its transmission over the network. But, since network transmission works with packets, the video has to be divided into units smaller than frames, called slices. Each slice is enveloped inside a packet that allows its transmission, called Network Abstraction Layer Unit (NALU), and then sent through the network. Each NALU contains a header and a payload, the RBSP (Raw Byte Sequence Payload).

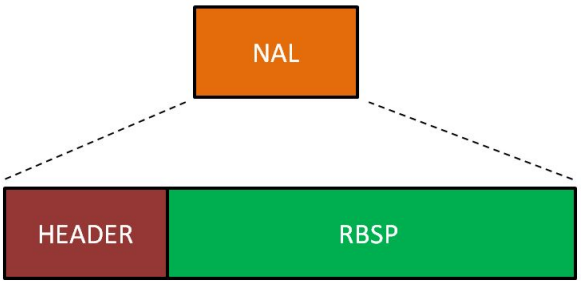

Figure 2.5 - NAL Header + RBSP[6]

The NAL header contains different pieces of information. The most important one is the NAL type, which specifies how to interpret the payload correctly.

The most common NAL units are the Sequence Parameter Set (SPS), the Picture Parameter Set (PPS), and the Slice NALs. The first two contain the necessary data to interpret the stream, while the last one is the data.

The slice NAL unit can be of two types: an IDR or an ordinary slice. The IDR defines an I-frame, which needs initially peculiar management, as explained in Section 2.1. Since it does not contain any motion vectors, its process initializes the context for the following frames.

All slices are divided into two parts, the slice header, and the slice data. The slice header contains information on the data, like the referenced frames, the entropy coding, or the frame number. The slice data, instead, contains the sequence of macroblocks

Each macroblock consists of different parts: the type, the prediction type, the coded block pattern (CPB), the quantization parameter (QP), the motion vectors (MV), and the residual data. This last part includes up to three components: the luminance (Y), and the chrominance (Cb&Cr).

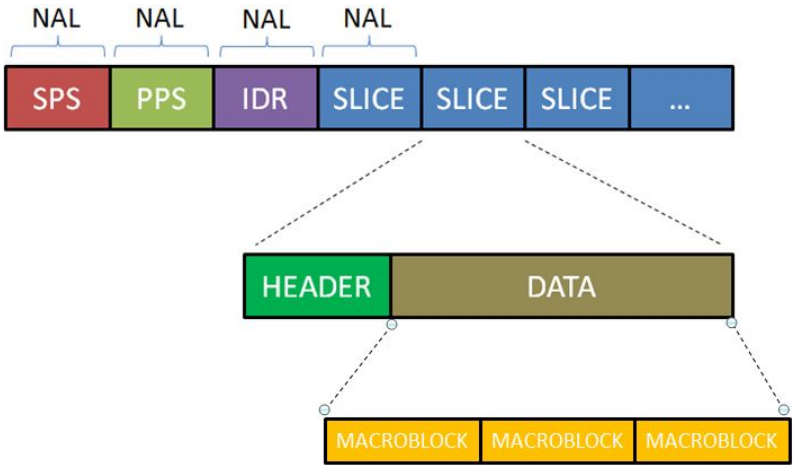

Figure 2.6 - Types of NAL[6]

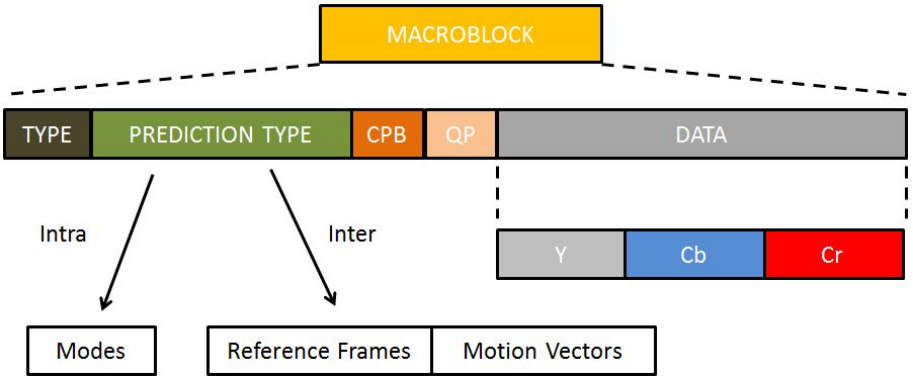

Figure 2.7 - Macroblock Subsections[6]

Between all the parameters, the CBP and the QP give information on how to use the residual data. But the important information to extract is contained in the prediction type part, which can be of two different kinds.

If the macroblock involves intra-prediction, this field will contain the prediction mode. Otherwise, if it involves inter-prediction, it will contain the motion vectors to extract.

Section 2.3 - FFmpeg

FFmpeg is a library with a wide set of functions and tools that allow the encoding, decoding, and analysis of different video standards.

The following section will provide a large view of the main functions of the library, while Chapter 3 will analyze some of them more in-depth.

Section 2.3.1 - FFmpeg Functions

The functions of the FFmpeg library can split into four main parts: the Parser, the Entropy Decoder, the Decoder, and the Loop Filter.

- The Parser is the part of the library that involves reading and decoding the bitstream directly. It contains functions like ff_h264_decode_seq_parameter_set, or ff_h264_decode_nal, which can read the bitstream and give it meaning.

  The main purpose of these functions is to read the bitstream and extract parameters and data from it.
- The Entropy Decoder part contains two main functions, ff_h264_decode_mb_cabac and ff_h264_decode_mb_cavlc. Both these functions are the main focus of the thesis since they are the key to extract the motion vectors from the bitstream. This section

also contains the functions for the reading of residual data, which will be of less interest.

- The Decoder part naturally follows the entropy decoding part. The main function of this section is ff_h264_hl_decode_mb, which decodes a macroblock in its entirety, performing the inverse DCT, the motion estimation, and the building of the final frame.
- The Loop Filter part is composed of the loop_filter function, which acts only if the deblocking filter is active. The deblocking filter is applied to improve video quality and prediction performance by smoothing the sharp edges of the macroblocks.

The first two parts of the FFmpeg library for H264 are the only ones needed for the extraction of motion vectors. The first for the reading of the parameters in the NALUs, the second for the entropy decoding.

Section 2.3.2 - Structures

In FFmpeg, the raw data of the NAL unit is stored inside an AVPacket structure. An AVPacket is the input of the decoder, while the AVFrame is its output.

Other important structures for the decoding process are.

- The H264Context, which contains the parameters of the decoding process.
- The H264SliceContext, which contains the parameters of the slice header and the caches of the slice.
- The AVCodecContext, which specifies the parameters of the decoder.
- The GetBitContext, which allows the buffered reading of the bitstream.
- H264Ref, which defines a reference from a frame to another.
- H264Picture, which contains the data and caches of a frame, AVFrame, and motion vectors included.

The following chapter will describe more in-depth how the motion vectors are calculated inside the FFmpeg library and the intermediate project of the thesis.

Chapter 3 - Intermediate Project

Section 3.1 - Decoder of FFmpeg

The FFmpeg library has many decoders based on the type of file the user provides in the input. For this reason, the decoding operations that transform a packet (AVPacket) in a frame (AVFrame) are managed by two main interface functions: av_send_packet and av_receive_frame. Also, when using the FFmpeg library to extract motion vectors from a video, the parameter +export_mvs should be passed in the av_dict_set function.
Any time a packet is sent to the decoder with the av_send_packet function, the user will receive back a frame through av_receive_frame and a structure called AVFrameSideData, where the motion vectors will be recorded.
The problem with this solution is that all the frames are entirely decoded to extract the motion vectors, so the process consumes an expensive amount of time to calculate useless information.
Despite this, an intermediate project has been developed to test if the motion vectors can be correctly extracted from the bitstream only by removing the unnecessary parts from the functions (Section 3.4).
This project still uses the complex structures of FFmpeg and therefore will result highly inefficient compared to the final project, especially in terms of memory usage.

Section 3.2 - Chain of functions

The structure allowing the decode of an AVpacket in FFmpeg is very linear. Therefore, in the following sections, the functions will be analyzed based on their queueing order.

Section 3.2.1 - main and read_rtsp_stream

The main function of the testing software checks the input - which is a string with the address of the video stream.
The read_rtsp_stream function, instead, initializes all the contexts used by the library to decode the packets. The structures of the FFmpeg contexts are:

- AVFormatContext: contains the streams in input, their duration, the references to the codecs used, and other useful information.
- AVCodecContext: contains the caches for the codec.

- AVCodec: is different for each standard and defines the API between the AVPacket and the needed methods to process it. It contains a reference for all the main functions of the codec, like receive_packet and receive_frame.
- AVStream: is the structure that contains the bitstream of a stream, plus some structures for the metadata and the side data.

In this function there are many calls to the library, like avformat_open_input, av_find_best_stream, avcodec_find_decoder, avcodec_alloc_context3, avcodec_open2, and avcodec_parameters_to_context. In particular, the av_find_best_stream function searches for the index of the correct stream, since the video can contain multiple ones (for video, audio, and even subtitles).
Once the stream is correctly open, the packets are read one by one from the AVFormatContext using the av_read_frame function. Then, h264_decode_frame is called.

Section 3.2.2 - h264_decode_frame and decode_nal_units

The h264_decode_frame function had many checks on the side data in the original form. Still, the side data has been removed, so also the function is lighter.
In its final form, the function only sets some flag values in the context. It checks if the buffer with the bitstream is empty, and if it is not, it calls the next function.

The ff_h2645_packet_split functions split the bitstream into the different NAL units. Then every NAL unit type is analyzed, and the right function is called to read the packet properly.
The types of NAL unit are the following (Table 7.1 in the ITU-T H264 Reference):

- IDR (Instantaneous Decoder Refresh): is the NAL unit containing an IDR frame, an I frame that clears the contexts of the reference picture buffer by marking all the pictures as “unused for reference”. This means that none of the frames after the IDR frame unit can reference a frame before the IDR frame.
- SPS (Sequence Parameter Set): it contains information about the profiles and levels of conformity of a coded video sequence, defined through a set of constraint flags. It also contains the parameters to read the sequence of slices, like the height and the width, the picture order method, the chroma format, and the bit depth (Section 7.4.2.1 ITU-T H264).

- PPS (Picture Parameter Set): it contains parameters to apply in the decoding process of one or more individual pictures inside a coded video sequence. For example, the prediction flag, the transform mode, or the scaling list (Section 7.4.2.2 ITU-T H264).
- SEI (Supplemental Enhanced Information): it contains additional information for the decoding process that is mostly unnecessary to decode the pictures. More on the messages in these structures is available in Annex D (Section 7.4.2.3 ITU-T H264 Reference).

It is also important to notice that for each type of NAL unit there is a particular function called to deal with it:

- IDR → If the buffers are not clear, then the IDR function is called, and then the code proceeds as a normal slice with the ff_h264_queue_decode_slice and ff_h264_execute_decode_slices functions.
- SEI → In this case the ff_h264_sei_decode function is called to read the sei messages and update the variables.
- SPS → The GetBitContext is extracted from the H2645NAL structure, and the ff_h264_decode_seq_parameter_set function is called.
- PPS → Follows the same function of the SPS, but it calls ff_h264_decode_picture_parameter_set instead.

After the NAL unit and the slice header have been correctly read, the ff_h264_execute_decode_slices function is called to proceed in the decoding process.

Section 3.2.3 - ff_h264_execute_decode_slices

The ff_h264_execute_decode_slices function checks the slices stored into the slice context in the H264Context structure. If there is a single queued slice, the decode_slice function is called with the correct context. Otherwise, the function enters a for loop where an H264SliceContext is created for each context, and the decode_slice function is called multiple times until every context has been decoded.

Section 3.2.4 - decode_slice

The decode_slice function aims to understand which entropy encoding technique has been used to code the picture, to call the proper function, and then proceed into the decoding process.

If the entropy coding is CABAC (Context-Adaptive Binary Arithmetic Coding), the initialization functions for the CABAC decoding process are called (align_get_bits, ff_h264_init_cabac_decoder, ff_h264_init_cabac_states and get_cabac_terminate) and the decoding process is started by calling ff_h264_decode_mb_cabac.
If the entropy coding is CAVLC (Context-Adaptive Variable-Length Coding), only one initialization function for the CAVLC decoding process is called (ff_h264_decode_init_vlc), and the decoding process is started by calling ff_h264_decode_mb_cavlc.
Two more steps follow each entropy decoding function into the decoding process, but the partial decoder does not contain them since they are meaningless in the extraction of motion vectors:

- ff_h264_hl_decode_mb: performs the entire decoding process of the slice
- loop_function: acts as the deblocking filter.

Section 3.2.5 - ff_h264_decode_mb_cavlc and ff_h264_decode_mb_cabac

Since the two functions for the entropy decoding have the same structure, the following section will analyze only the CAVLC function.
The ff_h264_decode_mb_cavlc function follows the following path:

1. Decode the skip macroblocks separately.
2. Obtain the macroblock type (mb_type) from the bitstream.
3. Fill in the information on the reference caches (top, left, and top-right macroblocks).
4. Based on the macroblock type:
    a. Intra prediction:
        i. If the macroblock is 4x4 the macroblock is analyzed separately.
        ii. If the macroblock is 16x16 no more decoding is needed.
    b. The block is divisible into 4 sub-blocks (only 8x8 macroblock). These blocks are analyzed separately.
        i. Analysis of the number of the referring frame of the sub-macroblock.
        ii. Analysis of the motion vector of the sub-macroblock.
    c. The block is of other dimensions (16x16, 16x8, or 8x16), then each macroblock is analyzed separately:
        i. Analysis of the number of the referring frame of the macroblock.
        ii. Analysis of the motion vector.
5. Decode of the residual information.

6. Spread the information over the entire frame.

Skip Macroblocks

The first check of the ff_h264_decode_mb_cavlc function is if there are macroblocks of type skip. Skip macroblocks are recorded in sequences, so when one skip macroblock is found, also the number of following skip macroblocks is read to interpret them correctly.

If the current macroblock is of type skip, the function decode_mb_skip is called. This function consists of two parts:

1. If the slice is of type B, then the neighbors and the caches of the macroblock are filled by calling fill_decode_neighbors and fill_decode_caches. Also, the ff_h264_pred_direct motion is called, which is an interface able to choose between spatial and temporal prediction. Both of these functions update the values in the reference lists (ref_list), the reference caches (ref_cache), and the motion vector caches (mv_cache).
2. Otherwise, the function fill_decode_neighbors will fill the neighbors' caches, and the function pred_pskip_motion will calculate the motion vectors of the skip macroblock considering the left, top, and top-right reference. (The prediction process is described in Section 3.3).

Macroblock Type

If the macroblock is not of type skip, the function starts by retrieving the macroblock type (mb_type) from the bitstream with the function get_ue_golomb. (More on the bitstream reading will be discussed in Chapter 4).

A check is then performed on the type of slice, to understand if it is of type B, P, or I. In all the cases, the information of the partition count (the parts in which the block can be subdivided), and the macroblock type for the following section, are extracted from some particular structures called ff_h264_x_mb_type_info (where x can represent the type of slice, b, p or i).

Caches Filling

Once the information of the current macroblock is calculated, the neighbors and the caches fill up through fill_decode_neighbors and fill_decode_cache. Then, the real intra-prediction decoding part begins.

If the macroblock is of type intra4x4, pred_intra_mode fills the left, top, and top-right neighbors of the current macroblock. After this operation, the caches are filled with the two functions write_back_intra_pred_mode and fill_rectangle.
Otherwise, if the block is of type intra with an edge of 16 pixels (Intra16x16), no more operations are needed.

Sub-Macroblocks

In some cases, the encoder splits the macroblock into 4 sub-blocks to analyze its movement. Usually, those blocks are the ones with the most movement in the frame, so every sub-macroblock will pair with its motion vectors.
In this situation, the decoder extracts the sub-type for each sub-block and fills the caches differently, storing values on a smaller range.
Then, for each macroblock, the references are filled with values from the bitstream. The motion vectors, instead, are calculated through the pred_motion function, plus a value read from the bitstream (motion vector difference, MVD). (More on the motion vector difference will be discussed in Chapter 4, Section 3).
Lastly, the ff_h264_pred_direct_motion function functions fills the caches.

Particular Macroblocks

If the macroblock is direct but not divisible in partitions, a specific case is defined for each of the remaining sizes: 16x16, 16x8, or 8x16.
All these cases have separated but similar code. First, the value of the reference cache is read from the bitstream with the get_ue_golomb_31 function, then the motion vector difference is calculated by the pred_motion function, and the motion vector cache is filled with the fill_rectangle function.

Motion Record

At the end of the prediction, if the macroblock was inter, the write_back_motion function updates the motion vector caches. This function contains internally two steps, one for each of the two reference lists used by the macroblock. The lists are both filled only with B macroblocks, while P macroblocks only fill up one of them.

Residual Data

The last part of the ff_h264_decode_mb_cavlc function reads the residual information about the macroblock that allows the decoding process to decode the entire picture.

This part is not necessary to extract motion vectors. However, to keep the synchronization of the bitstream, the bits still need to be read. In particular, the residual information doesn't occupy a fixed number of bits, which means it is not possible to skip them without corrupting the entire process.

For the purpose of this intermediate project, this part was left unchanged, but will be improved in the final project.

Section 3.5 - Adjustments of the FFmpeg library

The adjustments applied to the FFmpeg library to simplify the code and make it more effective were of four types:

1. Elimination of the frame decoding function:

   In the decode_slice function, after each call to the ff_h264_decode_mb_calvlc/cabac functions, there is a call to the ff_h264_hl_decode_mb. This function starts the next steps of the decoding process, split between the decoding of a complex and a simple part.

   Once the extraction has been correctly performed, no more information is needed, so this function has been removed.

2. Elimination of the loop function:

   After the decoding part, with the ff_h264_hl_decode_mb function, also the loop_function is called. This part of the code has to do with the deblocking filter applied to the compressed video to better the visual quality. The filter smooths the sharp edges that developed between the macroblocks after the compression process, so it is eliminable.

3. Elimination of the hardware acceleration part:

   The FFmpeg library provides a complex structure of hardware acceleration. Since this part is strongly connected with the FFmpeg library and acts differently depending on the type of standard, it has been removed.

4. Elimination of the error resilience part:

   The FFmpeg library also provides a feature for error resilience that can recover some parts of the bitstream in case they are lost, using some default values. This feature is not necessary to extract motion vectors, which will reset to the next IDR frame.

5. Partial elimination of the residual decoding part:

   The residual decoding part, explained in detail in Section 4, is one of the most complex parts of the process, and it involves evaluating all the non-zero coefficients of the frame, splitting them into blocks from 4x4 to 2x2. One part of the decoding is unavoidable since it reads part of the bitstream, while the evaluation is skippable, and it will be in the final project.

   The evaluation is mainly an operation of multiplication of a coefficient, extracted from the bitstream, and a scale factor, the quantization factor. The decoder reads both those elements from the bitstream, but their multiplication is not necessary.

Chapter 4 - Motion Vector Extraction

The following chapter will explain the main steps of the creation of a dedicated library to extract the motion vectors from an H264/AVC bitstream. Following the decoding order, Section 1 will deal with the bitstream reader and the identification of the NAL units. Section 4.2 will deal with the context variables read from the SPS, PPS, and slice header information. Section 4.3 will explain how the evaluation of motion vectors work, and how they are stored. Finally, section 4.4 will explain the two entropy decoding algorithms for decoding syntax elements and the residual information of the blocks, CAVLC and CABAC.

Structure of the Developed Library

The developed library analyzes the bitstream following the ITU-T H264 Standard and the FFmpeg library discussed in Chapter 3.

The input of the library is the raw data of the RTSP stream.

Once the bitstream is available in packets, the software identifies the NAL units. The identification follows two paths, depending on the bitstream format - one is the Annex B specification, the other is the AVCC specification.

Then, the extractor identifies the NAL unit by its header and passes the RBSP data to the next part of the process. For slices, it reads the slice header and all the macroblocks contained inside through a loop.

For each macroblock, the extractor will follow the structure of FFmpeg. First, it reads the macroblock type, and then different paths will be followed depending on the type of macroblock, as specified in the previous section.

Finally, it reads the residual information through CABAC or CAVLC decoding.

Section 4.1 - Bitstream Reading and NAL Identification

The functions involving the bitstream are of three different types: management, reading and skipping.

Section 4.1.1 - Management functions

The management functions deal with the initialization of the bitstream, the allocation of memory, and the conformity of the structures.

The bitstream is a simple structure made by three pointers: one to the start of the data, one to its end, and one pointing to the current position in the bitstream parsing operations. Also, there is a bits_left variable, in a range [0,7], which indicates the byte alignment (the position in the current byte). This part will come into use for CABAC decoding since it works with bytes, so the bitstream has to be aligned before starting the decoding.

Other management functions allow copying a bitstream, retrieving the position, detecting an overrun or the end of the bitstream.
The copying function is used in some situations where the bitstream needs duplication, for example, when initializing the CABAC decoder.
The end of stream functions is used to check if there is more data in the bitstream, as a safety check to understand if the bitstream has more data to read or if it is finished.
Last, the overrun function is a check on the current position of the bitstream which should never pass over the end pointer. This condition will mostly end-up in a segmentation fault problem, so it needs a constant check.

Section 4.1.2 - Reading and Skipping functions

As explained in Chapter 2, the main focus of the H264 standard is to compress data. For this reason, depending on the maximum value, a different number of bits have been dedicated to each variable. In many cases, the variables are encoded by subtracting a value from the final value to use the least amount of bits possible.
Since the encoding is complex, specific functions are needed to read, skip, or observe the following bits. Some are specific, like bitstreamReadU1, bitstreamReadU8, or bitstreamSkipU1. Others are generic, like bitstreamReadU, which reads the number of bits given as input. Those types of functions are used when the number of bits to read is not fixed or depends on the precedent readings.
Finally, a function highly used in the entropy decoding process is bitstreamNextBits, which reads several bits from the bitstream without moving the parsing pointer. This prediction technique allows the software to make decisions and give significance to the values read from the bitstream depending on their prefix (further explanation in Section 4.4).

Section 4.1.3 - Extracting NAL units from the bitstream

The NAL, as explained in Section 2, is the basic unit of an RTSP video stream, and its splitting is the first step of a correct reading of the data.
Unfortunately, there are two different techniques to split the NAL units, one following the Annex B section of the H264 standard, and the other following another syntax, called AVCC.

Section 4.1.3.1 - Annex B NAL Splitting

Following the Annex B definition, the NAL units in a bitstream always start with the unique code 0x000001 (which can also have 8 more zeros, 0x00000001). Identifying different NAL units, in this case, becomes problematic since it involves analyzing twice the bitstream: the first time to find the end of the NAL, which means the beginning of the next NAL or the end of the bitstream, and the second time to read the information they contain.

([start code]NALU | [start code] NALU | ...)

This behaviour is highly expensive, both in terms of memory, but also in terms of decoding time. For this reason the second technique has become more and more used.

Section 4.1.3.2 - AVCC NAL Splitting

The second technique uses a typical structure of networking - the header. In this case, there is a header for the entire stream, called extra data, which defines the main structural factors. The level and profile of the stream, the number of SPS and PPS (and their data), and the number of bytes preceding each NAL unit (from 1 to 4). This last part contains the length of each NAL unit.

([extradata] | [NAL Length] NALU | [NAL Length] NAL | ...)

This structure is stricter in terms of the adaptability of the stream. After the first SPS and PPS units, it is impossible to transmit any more structures of this type, so the stream cannot change parameters without a start-over. On the other hand, the bitstream will be parsed only once since the NAL length will precede each NAL structure. For this reason, this technique is the most used one.

Section 4.2 - Environment Elements

The PPS and SPS NAL units contain the most important variables of the environment, together with the slice headers. Those three structures, plus two for the context variables, define the entire decoding environment.

Section 4.2.1 - SPS and PPS NAL Units

The Sequence Parameter Set (SPS) is a type of NAL unit of the H264 Standard that contains information and parameters to apply to a sequence of different frames of the stream. The most important ones are the *chroma_format_idc*, used in the reading process of the residual data, and the picture dimensions (width and height).
Also, the bitstream contains a section of parameters called Visual Usability Information (VUI). This part is useless for the extraction but, as for the residual data, it has to be read.
Moreover, SPS contains information about the referenced pictures, like the number of reference pictures, or if the references allow gaps. This information will become useful to check the references of a picture and the possibility to free it.

The Picture Parameter Set (PPS), differently from the SPS, contains information about the single pictures. Here, the most important information is stored, like the entropy_coding_flag, which defines if the used encoding was CABAC or CAVLC, or the number_of_slice_groups parameter, which specifies how many slices compose a picture.
Other important information contained in the PPS are:

- *pic_init_qp_minus16,* the initial quantization parameter for the decoding process of the residual data. For each macroblock, also a delta parameter will be read and added to this value to obtain the final value.
- *transform_8x8_mode_flag*, which indicates a particular decoding process before the deblocking filter, which is not always active.

Section 4.2.2 - Slice Headers

The slice header contains the most important information of the entire stream since it defines the parsing of each slice.
For the extraction, these are the most used parameters:

- *slice_type*: defines the types of macroblocks contained inside the slice, so I, P, or B types, but also SI and SP.
- *frame_num*: represents the number of the frame from which the slice is part. The frame number is represented on log2(maxFrameNumber) bits to limit the number of bits used in the bitstream.
- *idr_pic_id*: is a flag used to identify an IDR picture, which is an I picture used that defines an access point to the video, from which the video can start playing.

- *num_ref_idx_ln_active_minus1*: defines the maximum reference index for referincing pictures in the list 0 or 1 (l0 or l1).
- *cabac_init_idc*: this parameter is present only with CABAC encoding, and identifies the type of CABAC context that should be used. The process is explained better in the last Section 4 of this chapter.
- *slice_qp/qs_delta*: these are differential parameters to add to the variable read from the PPS structure to obtain the final quantization parameter and chroma quantization.
- *Reference Pic List Reordering / Decoding Reference Picture Marking*: when the slice is of type P or B, the information of the reference pictures need to be stored, for list L0, when the picture is of type P, and also for list L1, when it is of type B.
- *Prediction Weight Table*: contains different information about the weight of the chroma and luma of the slice.

From all this information, the prediction weight table is parsed only to advance with the pointer to the bitstream, but the information is never used for the extraction.

Section 4.2.3 - Slice and Stream Contexts

The extractor, as well as the FFmpeg library, uses two structures to keep the information updated in and between slices- the h264slice structure, and the h264stream structure.
The information contained in the stream structure is:

- Pointers to the SPS, PPS, and the current slice header.
- Parameters regarding the position of the current macroblock in the picture, and other information defining its dimensions.
- A pointer to the current picture (h264picture) that contains information about the macroblock types (used for neighboring operations).
- A structure containing all the slice types, also used for neighboring operations.
- For the residual information, structures defining the zigzag scan type are important, plus a cache for the record of the non zero counts.
- For the CABAC decoding, some more information is needed, like the CBP and Chroma Prediction Mode tables (which can be ignored in CAVLC, but needed to modify the context in CABAC), a counter for the referencing lists, and a table for direct prediction.

The information contained in the slice structure is:

- Information about the position of the current macroblock, its type (eventually a sub_type), and its prediction mode.
- Information about the skip run macroblock if there is at least one SKIP macroblock currently being decoded.
- Caches for intra prediction (for both 4x4 and 16x16 macroblocks), references, and motion vectors. These pieces of information are used for neighboring operations and are highly important in the evaluation of motion vectors.
- Information about the indices of the neighboring macroblocks for the current macroblock. This information speeds up the neighboring filling, allowing a single read from memory instead of two (without these values, the software will need to retrieve the index of the structures where the values are stored before reading them).
- Data for residual information, like the local non-zero count cache, the Qscale and, only for CABAC, the CABAC context, its state, the direct prediction cache, and the information on CBPs for the top and left block.

Section 4.3 - Motion Vector Prediction

To achieve higher compression, H264 encodes the motion vectors as its difference from the referenced frame.

$$\mathbf{MV(k) = MV(k-1)+MVD}$$

where MV(k) is the motion vector at frame k, MV(k-1) is the motion vector of the preceding frame, and MVD is the motion vector difference, encoded in the bitstream.

Motion vectors can assume high values of magnitude, which would require many bits on the bitstream to be encoded. So the encoder only codifies the MVD in the bitstream, while the other part of the motion vector should be kept in memory by the decoder.

Also, to initialize a new motion vector for future frames, the encoder will transmit a null motion vector in the frame, so the MV(k-1) value will be available at the k frame.

All motion prediction functions, like pred_motion, pred_16x8_motion, pred_8x16_motion, and pred_pskip_motion, retrieve the value of MV(k-1) from the referenced frames. In particular, they calculate this value as the median of the three neighbor macroblocks: left, top, and top-right.

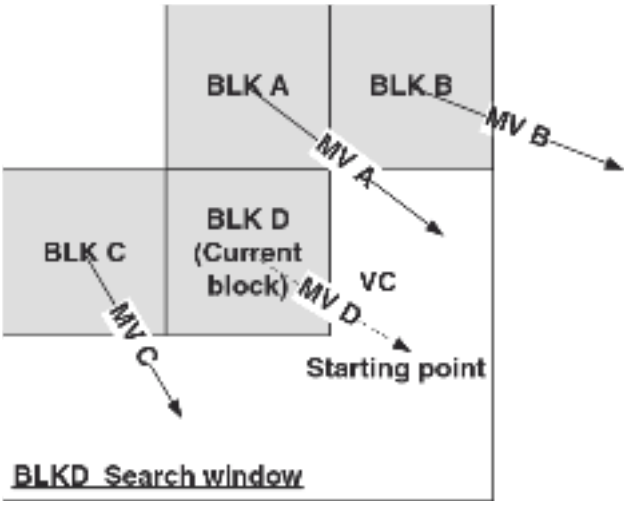

Figure 4.1 - Scheme of the motion vector prediction[7]

There are three main causes of different prediction:

1. If there are at least two values available → the motion vector is the median between all three values (with the third being zero if unavailable).
2. If there is only one value available → the motion vector is inherited from that single reference.
3. In all the other cases → the motion vector is inherited from the left reference, if available, or calculated as the median between all three values.

For the prediction of the motion for skip macroblocks, the process is a little bit more complex because the function needs to derive all the information from the neighboring caches, but the prediction part is the same.

Section 4.4 - DPB, RPLR and MMCO

While the above information is sufficient for the Baseline Profile, it is not for the Main, which uses B-type frames that are more complex to manage. In this case, some additional structures are needed - one to store short-term references, one to store long references, and one to store the referenced picture, with their values of macroblock type and motion values.
Each frame, so each slice, will have a list of references that indicates the decoder where to retrieve the information about the past frames in the DPB (Delayed Picture Buffer) and help rebuild the value of the motion vectors for the current frame.

These structures introduce a need of managing, referencing, and freeing the memory when the picture is not needed anymore - operations decoded as MMCO (Memory Management Control Operations) and defined in the header of the slice.
Each header can define the following MMCOs:

1. SHORT_TERM_UNUSED: marks a short reference as unused, so the picture can be freed.
2. LONG_TERM_UNUSED: marks a long reference as unused, so the picture can be freed.
3. SHORT_TO_LONG_TERM: marks a short reference as a long reference, and stores it in a different structure.
4. LONG_TERM_MAX_INDEX: defines the maximum index of frames a reference can influence.
5. ALL_UNUSED: resets the references and marks them all as unused.
6. CURRENT_TO_LONG_TERM: marks the current picture as a long reference.

Another important structure is the RPLR (Reference Pic List Reorder), which manages the order of the references so the correct reference is considered in the decoding process. The RPLR can transmit three values:

1. RPLR_ABS_DIFF_ADD/SUBTRACT: indicates a value that has to be added or subtracted from the frame number of the referenced picture. Then the referenced ID of the picture is set.
2. RPLR_LONG_TERM: defines the picture of the id referenced by a long term reference.
3. RPLR_END: defines the end of the reordering.

Section 4.5 - CAVLC and CABAC

The syntax elements of the bitstream, together with the residual data, are the majority of the data contained in the bitstream. These bits contain the partial values of the luma and chroma information of the video.
H264 supports two encoding techniques for the residual data:

- CABAC provides an improvement in the quality of about 7-10% but higher CPU consumption.
- CAVLC provides an improvement in CPU consumption of 10-15%, but lower quality.

Also, CABAC is only used in the Main and High Profile of H264, while CAVLC can be used by the Baseline, Main and Extended Profiles, usually on low-powered devices, like cell phones or tablets.

Section 4.4.1 - CAVLC (Context-Adaptive Variable-Length Coding)

CAVLC is an entropy coding technique that provides a lossless compression, which is a compression that doesn't cause a loss of quality. CAVLC works with residuals encoded in zig-zag order, and blocks of 4x4 (or 2x2) dimensions of coefficients, and is performed when entropy_coding_mode is set to 0.

The algorithm was structured to take advantage of many characteristics of quantized 4x4 blocks:

1. After the prediction, transformation, and quantization, the blocks contain mostly zeros (sparse blocks)
2. The number of non-zero coefficients of neighboring blocks is correlated. For this reason, the number of coefficients (totalCoeff) is encoded using a look-up table, which depends on the number of non-zero coefficients of the neighboring blocks.
3. The highest non-zero coefficients are often sequences of +/-1, so they are encoded in a compact way (trailingOnes).
4. The magnitude (level) of the non-zero coefficients tends to be higher at the start of the reordered array and lower towards high frequencies.

For this algorithm, the decoding and the encoding process involve four main variables:

1. Coefficient Token: is a variable that contains information about the number of non-zero coefficients in the block and the trailing ones.
2. Levels: represent the amplitude of the non-zero coefficients.
3. Total Zeros: is a variable that represents the number of zeros before the last non-zero coefficient of the block.
4. Run: represent the number of zeros before each non-zero coefficient.

Using these four variables, it is possible to encode or decode any block of coefficients possible, from 2x2 to 4x4.

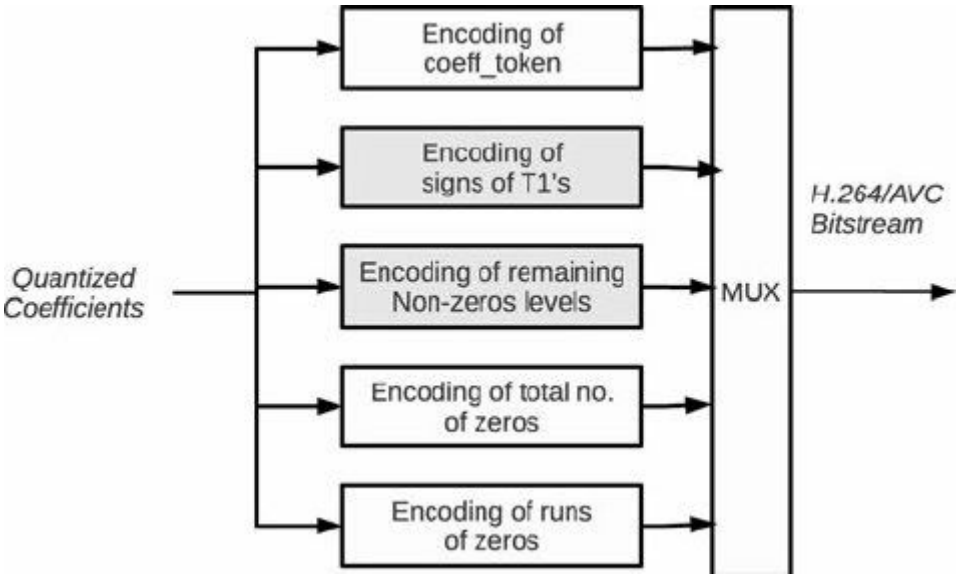

Figure 4.2 - CAVLC syntax elements[8]

Section 4.4.1.1 - Exp-Golomb Entropy Coding

When using CAVLC, not any syntax element is encoded using this technique - macroblock types, for example, are encoded through Exp-Golomb Entropy Coding. This technique is used to encode variable-length codes constructed regularly [M zeros][1][INFO], where INFO is a field of M bits carrying information.

The length of each codeword is 2M+1 bits, which the decoder will read bit by bit. Starting from M zero bits, followed by the first one bit, and then the INFO field. The value decoded will be **codeNum = 2M + INFO -1**.

Once the code number is known, the value (v) of the syntax element can be evaluated through four different mappings:

1. Direct Mapping ue(v) → **codeNum = v**
2. Signed Mapping se(v)

   → **codedNum = 2|v|** **if v < 0**

   → **codedNum = 2|v| - 1** **if v >= 0**
3. Mapped Symbols me(v) → specific mapping in **Table 9-4 of ITU-T H264 Standard**

Section 4.4.1.2 - CAVLC Encoding for Residual Information

The encoding process of the residual information starts with the rearrangements of the 4x4 data block and the computation of three key values:

- totalCoeff → represents the number of total non-zero coefficients in the block

- trailingOnes → represents the number of non-zero coefficients that have values of "1", with a maximum of 3 coefficients (the other ones are considered normal coefficients)
- nC → represents the correlation of the current blocks with their neighbors.

This value is computed as nC = (nA + nB +1) >> 1 when both left and top blocks are available, where nA is the correlation with the block on the top, nB is the correlation with the block on the left. When only one of the two blocks is available, nC=nN where N=A or N=B depending on which block is available. If none of the blocks is available, nC=0.
Also, nC has two particular values that depend on the chromaticity of the current block. In those two cases, nC=-1 when the YUV format is 4:2:0, and nC=-2 when the format is 4:2:2 (respectively chroma_format_idc == 1 OR chroma_format_idc == 2).

Based on these three values, a variable-length code, called coefficient token, is chosen from Table 9.5 in the ITU-T H264 Standard. Then, the sign of the trailing ones is encoded in the bitstream with [0,3] bits, with 0 representing a positive value, and 1 a negative one.

Once the trailing ones are encoded, the rest of the coefficients (totalCoeff - trailingOnes) have to be processed. Here, an important thing has to be considered: if the value of trailingOnes is less than three, then the first coefficient can't be +/-1.
Also, to save code bits, the amplitude is always reduced by 1, so if the level is positive it has to be increased by one, if it is negative it has to be decreased by one.
The levelCode is processed in two different ways, based on the sign of the level (amplitude):

- if the level is positive → levelCode = (level << 1) - 2
- if the level is negative → levelCode = - (level << 1) - 1

All the levels are encoded as prefix + suffix. Here, in case totalCoeffs > 10 and trailingOnes < 3, the suffixLength is initialized to 0. Then, the level prefix are calculated as:

levelPrefix = levelCode / (1 << suffixLength)

levelSuffix = levelCode % (1 << suffixLength)

From Table 9.6, considering the levelPrefix, a corresponding bitstream is encoded. Also the corresponding suffixLength is encoded.
Once a coefficient has been encoded, the suffixLength is incremented when:

- suffixLength == 0.
- levelCode > (3 << (suffixLenght - 1)) && suffixLength < 6

and the encoding proceeds until the coefficients finish.

The following part of the algorithm encodes the zeroes before each coefficient. For this reason, the variable totalZeros is important, since it indicates the sum of all the zeros before the last non-zero coefficient.
Another two important variables are zerosLeft and runBefore:

- zerosLeft represents the number of all zeros before the current non-zero coefficient, and it starts equal to the totalZeros value.
- runBefore represents the number of zeros immediately before the current coefficient

Once these values are calculated for each non-zero coefficient, starting from the highest frequency, the tables 9-7, 9-8, 9-9 (for zerosLeft), and 9-10 (for runBefore) are used to define the bitstream to encode.

Example of Encoding:
Coefficients: [0, 3, 0, 1, -1, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

1. First of all, the total non-zero coefficients are calculated, totalCoeff = 5.
2. Then, the trailing ones, trailingOnes = 3 (there are 4 ones, but the maximum value of trailingOnes is 3, the other ones are considered as normal non-zero coefficients).
3. A value of nC = 1 will be considered for the purpose of the exercise.
4. From Table 9-5 → coeff_token = codedBitstream = 0000100
5. The trailing coefficients are 1, -1, 1 (highest frequency) → codedBitstream = 011
6. Missing other 2 levels to encode, [3, 1], starting from the highest frequency:
    a. levelCode = (level << 1) - 2 = (1 << 1) - 2 = 0.
       levelPrefix = levelCode / (1 << suffixLength) = 0 / (1 << 0) = 0
       levelSuffix = levelCode % (1 << suffixLength) = 0 / (1 << 0) = 0
       From Table 9-6 → codedBitstream = 1
    b. levelCode = (level << 1) - 2 = (3 << 1) - 2 = 4
       levelPrefix = levelCode / (1 << suffixLength) = 4 / (1 << 1) = 2
       levelSuffix = levelCode % (1 << suffixLength) = 4 / (1 << 1) = 0
       levelSuffixLength = 1
       From Table 9-6 → codedBitstream = 0010

7. Now it is time to calculate the zeros. First of all, the totalZeros value is encoded from Table 9-7 considering also totalCoeff. In this case, totalZeros = 3 && totalCoeff =5 → codedBitstream = 111
8. Next, the zeros before each coefficient are encoded, starting from the non-zero coefficient with the highest frequency, which is the last 1.
    a. zerosLeft == 3 && runBefore == 1 → codedBitstream = 10
    b. zerosLeft == 2 && runBefore == 0 → codedBitstream = 1
    c. zerosLeft == 2 && runBefore == 0 → codedBistream = 1
    d. zerosLeft == 2 && runBefore == 1 → codedBitstream = 01
    e. zerosLeft == 1 && runBefore == 1 → no coded bitstream required.

Final bitstream: **0000 1000 1110 0101 1110 1101**

Section 4.4.1.3 - CAVLC Decoding

The decoding process uses the maxCoeffNumber variable in input, and the block of coefficients, which can be a luma or chroma block.

The first step of the decoding process involves retrieving the two variables: totalCoeff and trailingOnes, which is performed through the look-up Table 9.5 from ITU-T H264 Standard by giving in input nC and the bitstream. The nC variable is derived from the neighboring blocks as specified in section 3.4.1.1, while the coeffToken, which is the bitstream value to consider, is unique for each column of the table.
Once totalCoeff and trailingOnes have been decoded, the trailingOne bits are read, each bit representing the sign of the respective one (0 is positive, 1 is negative).

If the difference between the total number of coefficients and the trailing ones is greater than zero, it means there are other coefficients to decode, so the following passages are repeated totalCoeff - trailingOnes times.

The decoding of each coefficient is performed by calculating prefixes and suffixes. The suffix length is initialized to 1 if the total coefficients are more than 10 and the trailing ones are less than 3, in all the other cases it starts from 0. On the other hand, the level prefix is calculated reading the bitstream until a bit set to 1 is found (Table 9-6 ITU-T H264 Standard). The algorithm used is the following one:

```
leadingZeroBits = -1
for(i=0; !i; leadingZeroBits++)
        b = read_bits(1)
levelPrefix = leadingZeroBits
```

Once the levelPrefix has been calculated, the level suffix size is set to 4 and suffixLength to 0 if the levelPrefix is 14. Otherwise, if levelPrefix is greater than 15, the level suffix size is set to levelPrefix-3. Lastly, level suffix size is equal to suffixLength.

The levelSuffix is then calculated from the levelSuffixSize. If the levelSuffixSize is greater than 0, the levelSuffix is read from the bitstream as u(v) from levelSuffixSize bits. Otherwise, the levelSuffix is set to 0.

The variable levelCode is then set to the minimum between 15 and the level prefix, shifted by suffixLength positions, with the level_suffix added.

levelCode = [min(15, levelPrefix) << levelSuffix] + levelSuffix

Next, levelCode is incremented by a variable value:

- if levelPrefix >= 15 and suffixLength == 0 → levelCode is incremented by 15.
- if levelPrefix >= 16 → levelCode is incremented by (1 << (levelPrefix - 3)) - 4096.
- if the current level is equal to the trailingOnes variable, and the trailingOnes variable is also less than 3 → levelCode is incremented also by 2.

Lastly, the final value of the non-zero coefficient is derived using the levelCode variable. If the levelCode is even, then level[i] = (levelCode + 2) >> 1, otherwise, if levelCode is odd, then level[i] = (-levelCode - 1) >> 1.

Also, suffixLength is updated to calculate the next non-zero coefficient. If suffixLength is zero, then it is set to 1. Also, if the absolute value of the current non-zero coefficient is greater than (3<<(suffixLength-1)) and suffixLength is less than 6, the suffixLength is incremented by 1.

Once all the non-zero coefficients have been encoded, also the zeroes have to be encoded. This process is based on the variable maxNumCoeff given as input.

The maxNumCoeff can assume three different values:

- maxNumCoeff == 4 if the block is of type Chroma DC 2x2 (4:2:0) → Table 9.9 (a)
- maxNumCoeff == 8 if the block is of type Chroma DC 2x4 (4:2:2) → Table 9.9 (b)

- In all the other cases the Tables 9.7 and 9.8 from the ITU-T H264 Standard are used.

From totalCoeff and maxNumCoeff, the zerosLeft value is derived by subtracting the first from the second. These are the number of zeros the decoder needs to insert in the bitstream to fully reconstruct the block. If the number of non-zero coefficients is equal to the maximum number of coefficients, then there are no zeros left, so the encoding process is finished.

From the tables defined above, the number of total zeros is derived by giving as input to the look-up tables the number of totalCoefficients and the bitstream. Then, a loop is performed to calculate how many zeros precede any of the non-zero coefficients.
For each non-zero coefficient:

1. The variable run, representing the number of zeros that precede the coefficient, is calculated from tables 9-10, taking as input the zerosLeft variable and the bitstream. A run_before value is retrieved, which is the number of zeros to put before the current coefficient. If the variable zerosLeft is equal to 0, there are no zeros before the current coefficient, which also means the decoding process is finished.
2. The number of zeros inserted before the current coefficient is subtracted from the zerosLeft variable, and the algorithm proceeds to the next coefficient.
3. If there are any zeros left when the coefficients are finished, the last value of the run is set to the number of zeros left. These are all the zeros after the last non-zero parameter.

Now that every information is available, the block is reconstructed using the list of non-zero coefficients, including the trailing ones, and a list of values representing the zeroes before each non-zero coefficient.

Example of Decoding:
Coded Bitstream: **0000 1000 1110 0101 1110 1101**

1. The decoding process starts with the extraction of the total coefficients and the trailing ones from the look-up table 9-5. Considering nC=1 → coeffToken = 0000100 so totalCoeff = 5 and trailingOnes = 3.
2. The next 3 bits are the trailing ones sign flag = 011:
    a. trailing_one_sign_flag[0] = 0 → level[0] = 1 - (2 * 0) = 1

b. trailing_one_sign_flag[1] = 1 → level[1] = 1 - (2 * 1) = -1

c. trailing_one_sign_flag[2] = 1 → level[2] = 1 - (2 * 1) = -1

→ **coeffLevel = [-1, -1, 1]**

3. The levelPrefix and the suffixLength are both set to 0.
4. Now the decoding of the rest of the non-zero coefficients start, by calculating first the levelCode = Min(15, level_prefix) << suffixLength + level_suffix. In this case, level_prefix = level_suffix = suffixLength = 0. The prefix is derived from the look-up table 9-6, with bitstring = 1 → levelCode = 0 << 0 + 0 = 0
5. Now, the evaluation of the level depends on the parity of the levelCode. Since levelCode is even, level[3] = (levelCode + 2) >> 1 = (0 + 2) >> 1 = 1.

   → **coeffLevel = [1 -1 -1 1]**
6. The suffixLength is incremented → suffixLength = 1.
7. From the look-up table 9-6, with bitstream = 001 → levelPrefix = 2 and levelSuffixSize = 1, so another bit from the bitstream is read. With these values it is possible to calculate the levelCode = Min(15, 2) << 1 + 0 = 4.
8. Since the levelCode is odd again, the same formula is used to evaluate it. level[4] = (levelCode + 2) >> 1 = (4 + 1) >> 1 = 3.

   → **coeffLevel = [3, 1, -1, -1, 1]**
9. Considering totalCoeff = 5, and the look-up table 9-7 → totalZeros = zerosLeft = 3.
10. For each coefficient, the zeros before are now derived:
    a. zerosLeft == 3 && bitstream == 10 && table 9-10 → runBefore[0] = 1
    b. zerosLeft == 2 && bitstream = 1 && table 9-10 → runBefore[1] = 0
    c. zerosLeft == 2 && bitstream = 1 && table 9-10 → runBefore[2] = 0
    d. zerosLeft == 2 && bitstream = 01 && table 9-10 → runBefore[3] = 1
    e. zerosLeft = 1 && one missing non-zero coefficient → runBefore[4] = 1

    → **coeffLevel[0, 3, 0, 1, -1, -1, 0, 1]**
11. Now, after combining the information of the runBefore and the levels, all the remaining coefficients are set to 0.

Final Block Coefficients: **[0, 3, 0, 1, -1, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]**

Section 4.4.2 - CABAC (Context Adaptive Binary Arithmetic Coding)

CABAC is an arithmetic coding system used to encode/decode some of the H264 syntax elements. The technique is used when entropy_coding_mode is set to 1, and achieves good compression performances through three different actions:

- it selects probability models for each syntax element according to the context of the element itself
- it adapts probability estimates based on the local statistics
- it used arithmetic coding

With CABAC, each symbol is coded by following four main stages:

1. **Binarization**: since CABAC is a binary arithmetic coding algorithm, it means that only binary operations are encoded, so zeros and ones. Every other non-binary symbol, like motion vectors, is binarized and converted into binary code before the arithmetic coding.
2. **Context Model Selection**: a step used to understand which context to choose to encode or decode the symbol. A context model is the probability model for one or more bins of the binarized symbol. Also, the selection is performed from many available models that depend on the statistics of recently coded/decoded symbols. The context model will store the probability of each bin to be a one or a zero.
3. **Arithmetic Encoding**: this module encodes each bin according to the context model previously selected. For each bin, the encoder will choose between a zero or a one.
4. **Probability Update**: in the last step the context model is updated based on the value that has been encoded in the previous step. In particular, if the code was 1, the frequency of 1 will increase, the same for the 0 value.

The first step is performed only once, while the steps from 2 to 4 are performed for each bit of the binarized symbol, as shown in the figure below.

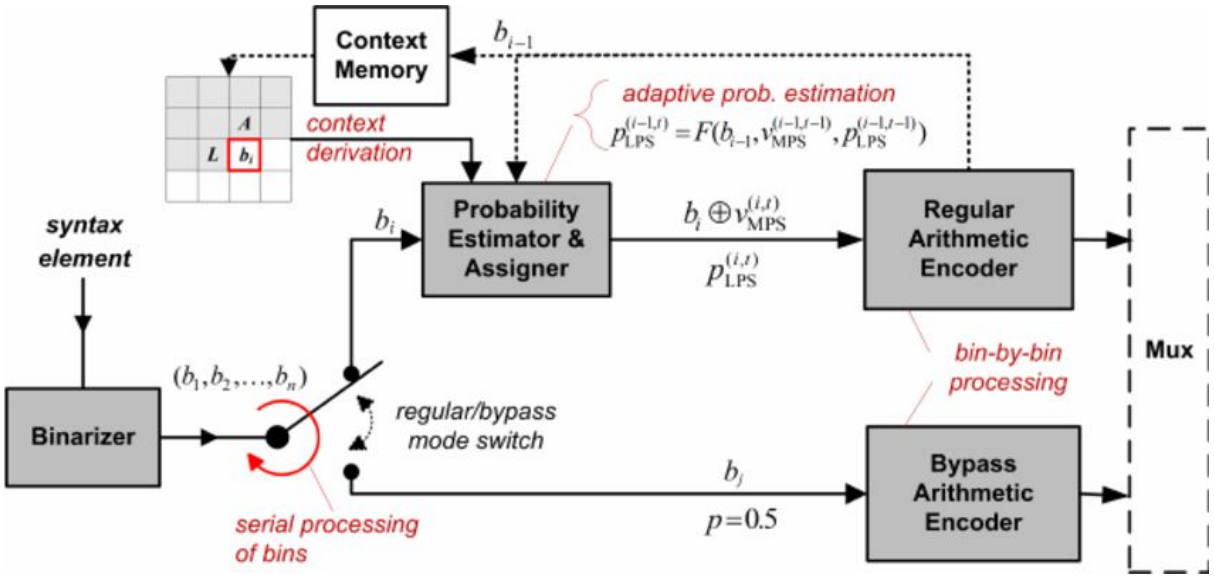

Figure 4.3 - CABAC Decoder[9]

Section 4.4.2.1 - CABAC Decoding

The CABAC decoding process starts when the entropy_coding_mode_flag is equal to 1, information read from the SPS NAL unit that precedes the slices to decode. As shown in Figure 9-1 of the ITU-T H264 Standard paper, when starting the decoding process for each slice, CABAC needs its context. Once the initialization is done, the entire slice can be decoded using the available context.

Section 4.4.2.2 - Initialization

The initialization of the CABAC context is composed of two different parts, one setting up the variables in the context, and one the decoding engine. The first part is performed when the decoding of a slice begins, while the second part recurs when for any syntax element that is a macroblock.

The initialization of the variables of the CABAC context follows the Tables 9-12 to 9-33 in the ITU-T H264 Standard paper. These tables contain two variables, m and n, which are retrieved by the decoding process using a context index (ctxIdx) for indexing.

Also, each syntax element (SE) has its table. For example, the macroblock type uses the tables 9-12 to 9-14, while the motion vectors (difference) use the table 9-15.

A list of each syntax element coded through the CABAC encoder can be found in the ITU-T H264 Standard, in Table 9-11.

For each context variable, two other variables are initialized: the probability state index (pStateIdx), and the value of the most probable symbol (valMPS). These variables are assigned through the following pseudo-code:

```
preCtxState = Clip3(1, 126, ((m * Clip3(0, 51, SliceQPy)) >> 4) + n);
if(preCtxState <= 63){
        pStateIdx = 64 - preCtxState
        valMPS = 0
}else{
        pStateIdx = preCtxState - 64
        valMPS = 1
}
```

where SliceQPy can be retrieve from the PPS NAL variable pic_init_qp_minus26 through the following equation:

SliceQPy = pic_init_qp_minus26 + 26 + slice_qp_delta

and Clip3 is a function that retrieves one of the three values given in input, depending on the relationship of the last parameter with the other two:

Clip3(x,y,z) = {x if z<x; y if z>y, z otherwise}

The values of ctxIdx that should be used to decode a particular slice are also listed in Table 9-11. Also, for P, SP, and B slices, the initialization needs another parameter, which is the cabac_init_idc and it is available in the slice header.

The second part of the initialization involves the arithmetic decoding engine - a system composed of two registers, codIRange and codIOffset, both in 16bit precision. These two values represent the status of the engine, and are set as follows:

- codIRange = 510.
- codIOffset = read_bits(9), interpreted as an unsigned integer.

Section 4.4.2.3 - Binarization

This process is called each time a syntax element needs decoding, and it returns as result three variables: maxBinIdxCtx, ctxIdxOffset, and bypassFlag.
Using Table 9-34, it is possible to understand, based on the syntax element requested in input, which value the three variables will assume.

There are four types of binarization possible, plus some specific binarization depending on the syntax element to encode/decode:

1. **Unary** (Clause 9.3.2.1) - the value of the syntax element is easily obtained by counting all the "1" read from the bitstream, until the first "0".
2. **Truncated Unary** (Clause 9.3.2.2) - the truncated binarization invokes the unary binarization giving it in input a max value. If the max value is reached, the syntax element assumes that value, otherwise the value from the unary binarization is returned.
3. **Concatenated Unary / k-th order Exp-Golomb** (Clause 9.3.2.3) - this binarization is composed of two parts, one decoding the prefix and one the suffix of the final result. The prefix is solved by calling the truncated binarization, giving in input a specific max value depending on the syntax element.

Also, the variable k is specified for each syntax element and is used for the binarization of the suffix part of the process, which doesn't have to be always performed.

1. **Fixed-Length** (Clause 9.3.2.4) - this binarization has the particularity to always occupy the same length, specified in the input. The process starts from the less significant bit, with increasing values of the binIdx, and it doesn't have to be filled to end.
2. Another specific binarization composed of the concatenation of two bin strings, prefix, and suffix, is used to binarize the macroblock type (Clause 9.3.2.5). Depending on the slice type, different processes will be used.
    a. For I and SI slices Table 9-36 will be used, with the exception that, in the second case, only a single bit of prefix will be transmitted, communicating if the single macroblock is of type SI or not.
    b. For P, SP, and B slices, Table 9-37 will be used. With the exception that, every time an I macroblock type is included in those slices, the syntax element is defined as prefix and suffix, where the suffix is taken from the table of I and SI slices. Plus, a subtraction is performed on the macroblock value, to put it in the correct range of values for the suffix binarization.
3. Also, the binarization of the coded block pattern is composed of a prefix and a suffix, only when the chroma array type is not 0 or 3. The prefix is the fixed length binarization of the CodedBlockPatternLuma variable, while the suffix part is the truncated binarization of the CodedBlockPatternChroma variable.

Section 4.4.2.4 - Decoding Process

Once the three variables maxBinIdxCtx, bypassFlag, and ctxIdxOffset are defined by the binarization process, the value of the syntax element is finally read from the bitstream.

The parsing is performed by adding one bit at a time to a temporary bit string and, for each addition, the bit string is compared to the bins of the binarization. If the bit string and one of the bin strings are equal, the bit string is the value of the syntax element, otherwise, a new bit is added to the bit string.

In cases in which the binarization involves a prefix and a suffix, the process is the same for the prefix but varies for the suffix, and a set of specific rules are applied depending on which of the prefix bin string resonates with the bit string.

The decoding process involves two types of sub-processes:

- if the bypassFlag is set, the DecodeBypass process will be performed
- otherwise, the parsing of each bin follows two derivation paths, the first derives the index of the context (ctxIdx), while the second uses it to decode the value of the bin. For this purpose, Tables 7-39 to 9-43 are used.

The derivation of the ctxIdx is specific for each syntax element, while the arithmetic decoding part to evaluate a single bin is the same, and is described in the following flowchart.

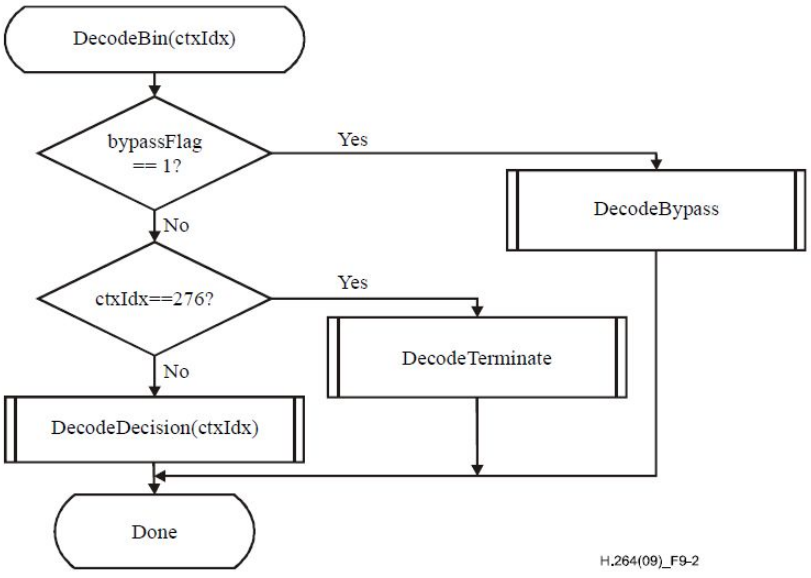

Figure 4.3 - DecodeBin Process

Depending on the two variables bypassFlag and ctxIdx, three different decode processes will be called:

1. **Decode Decision**: decodes a bin value and updates the current state of the CABAC context. Ends with a Renorm function, which normalizes the context when the codIRange >= 276 by shifting it and codIOffset by one position, and also filling the rightmost bit of codIOffset with a bit from the bitstream.
2. **Decode Terminate**: this is a particular decoding function that only applies for end_of_slice_flag, and when ctxIdx = 276. This function decrements codIRange by two, then it compares it with codIOffset. If codIOffset is bigger or equal, it predicts a binValue = 1, otherwise it predicts a binValue = 0, and performs a normalization.
3. **Decode Bypass**: is the process called only by some syntax elements. It performs the same operations of codIOffset of the normalization function. Then, it compares codIOffset with codIRange, and if it is bigger or equal, it predicts a binValue = 1 and updates codIOffset by subtracting codIRange from it. Otherwise, it predicts a binValue = 0.

Section 4.4.2.5 - Example of CABAC coding process

Since the purpose of the thesis is to extract motion vectors from the bitstream, the following section will analyze the coding process for the motion vector difference (MVD).
In both CABAC and CAVLC, the motion vector of a macroblock is encoded as the difference between the motion vector in the current macroblock, and the same motion vector of the macroblock in the precedent frame, as explained in Section 4.3.
Although, in CABAC, the motion vector difference is coded differently depending on the magnitude: if the |MVD| is lower than 9, then the following rule will be used, otherwise the previously explained Exp-Golomb process will be used.

|MVD| = x → Binarization = x times "1" followed by one "0"

Ex. |MVD| = 5 → Binarization = 111110

Once the binarization has been derived, the context model has to be chosen for each bin. Starting from bin 1, there are three models to choose from, and the choice is performed based on the norm of the two previously coded values (ek).

Ek = |MVDa| + |MVDb|

where a,b are the blocks to the left and above the current block.

- 0 <= Ek < 3 → Context model 0
- 3 <= Ek < 33 → Context model 1
- 33 <= Ek → Context model 2

This system is used because, statistically, the magnitude of the current macroblock and the ones to its left and top side are comparable, which means that if the neighbors have a large magnitude, it is highly probable the current MVD will have a large magnitude too.

For the other bins, one of the further four context models will be used (model 3 to 6), in ascending order. Bins from the 5th to the 9th position will all use context model 6.

The chosen context model will contain two probability values, one for the "1" value, and one for "0". These probabilities determine the ranges the arithmetic coder will use to encode the bin.

The last step involves updating the context model with the encoded value. If a "1" was encoded, the respective probability will increase, the same for the "0" value. This will iterate until, after different bins are encoded, the probabilities will exceed a threshold value which will reset them.

Chapter 5 - Temporal & Memory Usage Analysis

The following chapter exposes the analysis of the temporal results the developed project reached, providing also an analysis of the memory usage for the two decoding processes: complete and partial.

The analysis uses different types of videos and different systems. The videos have been encoded on two different profiles, Baseline and Main, the first one using the CAVLC entropy coding, while the second using the CABAC coding.

Also, since the decoding processes one macroblock at a time, the videos are differentiated regarding their length, expressed in terms of total macroblocks.

Video	Frame Dimension	Total Blocks	Skip Blocks	Skip %
city_overview	1080x1350	10039860	2681734	26,71
discover_japan	1280x720	21733200	8882933	40,87
discover_argentina	1280x720	22759200	7988046	35,10
river_oversee	3840x2160	23295600	9827947	42,19
low_light_city	1920x1080	29082240	18539909	63,75
discover_singapore	1280x720	29523600	14779924	50,06
dancing_flames	3840x2160	43448400	22714036	52,28
discover_miami	1920x1080	46136640	25548702	55,38
sintel	1280x546	56980000	25335456	44,46
spring	1920x804	65135160	37568714	57,68

Table 5.1 - Video Information

The systems used vary on the RAM used: Linux (Ubuntu) 64 bit with a clock frequency of 3,6Ghz and 8, 4, or 1GB of RAM.

The number of cores is irrelevant since the decoding process has been forced to run in a single t-hread and couldn’t use this characteristic.

Section 5.1 - Measurements and Descriptions

The measurements of every time variable exposed in the next paragraphs have been obtained as the mean value resulting from 20 single measurements. Also, before the calculation of the

mean value, the two highest and the two lowest measurements have been removed, so to mitigate the error introduced by outliers, in both directions.

For a more complete analysis, the content of the videos will be shortly explained:

1. *City Overview*: a time-lapse effect on a highly detailed city, with movements of different entities and speed. The camera is in slow motion, together with the background, while elements like the cars on the streets and the clouds are extremely fast due to the time-lapse effect.
2. *Discover Videos*: a series of different videos shot by drones that perform similar movements. The camera is slow and steady, while the background is movement sinuously.
3. *River Oversee*: a fast drone-video that flies over a river and raises to a bridge over a valley in a mountain landscape. The movement is extremely fast at the beginning of the video, slowing down later on.
4. *Low Light City*: a video captured with a fixed camera in a highly crowded city at night. The lights of the cars partially reveal the movements of the people on the street.
5. *Dancing Flames*: an almost-fixed camera records a character dancing with a smoke generator in hand. The video is in slow motion, and the smoke covers almost the entire frame by the end of the video.
6. *Sintel & Spring*: two short animated films developed by Blender. Both of those videos, compared to the others, contain the vastest variety of movements and changes of environments. In terms of length, one lasts half of the other.

Section 5.2 - Difference between RAM Availability

Considering the different RAM settings, the following data has been measured by decoding some videos.

RAM SETTINGS (FFMPEG)	T_8GB (s)	ST.DEV	T_4GB (s)	ST.DEV	T_1GB (s)	ST.DEV
low_light_city (CAVLC)	15,25	0,45	15,32	1,40	15,55	0,29
low_light_city (CABAC)	27,10	1,08	26,10	1,16	24,50	0,17
dancing_flames (CAVLC)	21,98	0,20	21,56	0,21	21,47	0,14
dancing_flames (CABAC)	30,21	0,85	28,27	0,32	28,03	0,34

city_overview (CAVLC)	13,81	0,23	13,48	0,21	11,56	0,18
city_overview (CABAC)	21,80	0,24	21,12	0,64	19,76	0,24
river_oversee(CAVLC)	11,76	0,42	11,34	0,12	9,15	0,07
river_oversee(CABAC)	17,48	0,49	16,91	0,44	15,20	0,10

Table 5.2 - Mean timings for FFmpeg Decoder

RAM SETTINGS (MVExtractor)	T_8GB (s)	ST.DEV	T_4GB (s)	ST.DEV	T_1GB (s)	ST.DEV
low_light_city (CAVLC)	9,11	0,10	9,35	0,23	9,24	0,15
low_light_city (CABAC)	26,62	0,48	26,12	0,41	25,72	0,26
dancing_flames (CAVLC)	14,82	0,08	15,01	0,15	14,88	0,21
dancing_flames (CABAC)	30,58	0,39	31,22	0,32	30,28	0,27
city_overview (CAVLC)	9,53	0,05	9,49	0,14	9,73	0,21
city_overview (CABAC)	23,24	0,12	24,78	0,47	23,04	0,12
river_oversee(CAVLC)	7,65	0,08	8,13	0,21	7,58	0,18
river_oversee(CABAC)	15,77	0,07	15,21	0,20	15,85	0,15

Table 5.3 - Mean timings for Motion Vector Extractor

It is possible to notice how, in both cases, the RAM available does not have a major impact on the performance of the decoders, both for CABAC and CAVLC cases. For this reason, there are cases in which a system using 1GB of RAM could perform faster than a system using 8GB.

Both decoding and extraction processes perform one macroblock at a time, using some caches for the current frame and, sometimes, additional caches for potential reference frames. In both cases, those caches are little enough to enter 1GB of RAM, so there are no concrete variations on the speed depending on the available memory of the system.

Section 5.3 - Complete Decoding and MV Extraction Analysis

The following section will analyze the difference between a complete decoding process and the simple extraction of motion vectors.

Video	Total Blocks (10^6)	Sec/Mil (Decode)	Sec/Mil (Extraction)	Improvement
city_overview	10,04	1,38	0,95	31,00
discover_japan	21,73	0,89	0,73	18,32
discover_argentina	22,75	0,93	0,74	19,88
river_oversee	23,29	0,50	0,33	34,96
low_light_city	29,08	0,52	0,31	40,26
discover_singapore	29,52	0,60	0,49	18,04
dancing_flames	43,45	0,51	0,34	32,58
discover_miami	46,13	0,50	0,38	24,40
sintel	56,98	0,89	0,73	18,27
spring	65,13	0,56	0,45	20,50

Table 5.4a - Mean Timings Comparison CAVLC

Video	Total Blocks (10^6)	Sec/Mil (Decode)	Sec/Mil (Extraction)	Improvement
city_overview	10,04	2,62	1,99	24,03
discover_japan	21,73	1,75	1,54	11,97
discover_argentina	22,75	1,70	1,49	12,43
river_oversee	23,29	1,07	0,68	36,93
low_light_city	29,08	1,23	0,92	25,74
discover_singapore	29,52	1,30	1,17	10,35
dancing_flames	43,45	1,05	0,70	32,96
discover_miami	46,13	0,95	0,74	22,24
sintel	56,98	1,85	1,63	12,08
spring	65,13	1,11	0,92	17,66

Table 5.4b - Mean Timings Comparison CABAC

Comparing the two measures, the one of the complete decoding and the one of the motion vector extraction, it is possible to highlight 3 different correlations (taking as examples the CABAC results).

Case 1 - Correlation between length (in frames) and time

As it is possible to evince from Figure 5.1, there is a strong correlation between the length of a video (in frames) and the decoding process. This is because the most time-consuming part of the decoding process is the reading of the slice header, which happens at least one time for each frame (a NAL unit can contain up to one frame).

On the contrary, the decoder only spends about 40% of the time reading the bitstream, while the other part is dedicated to the remaining part of the decoding process, mainly motion compensation, IDCT, luma, and chroma prediction.

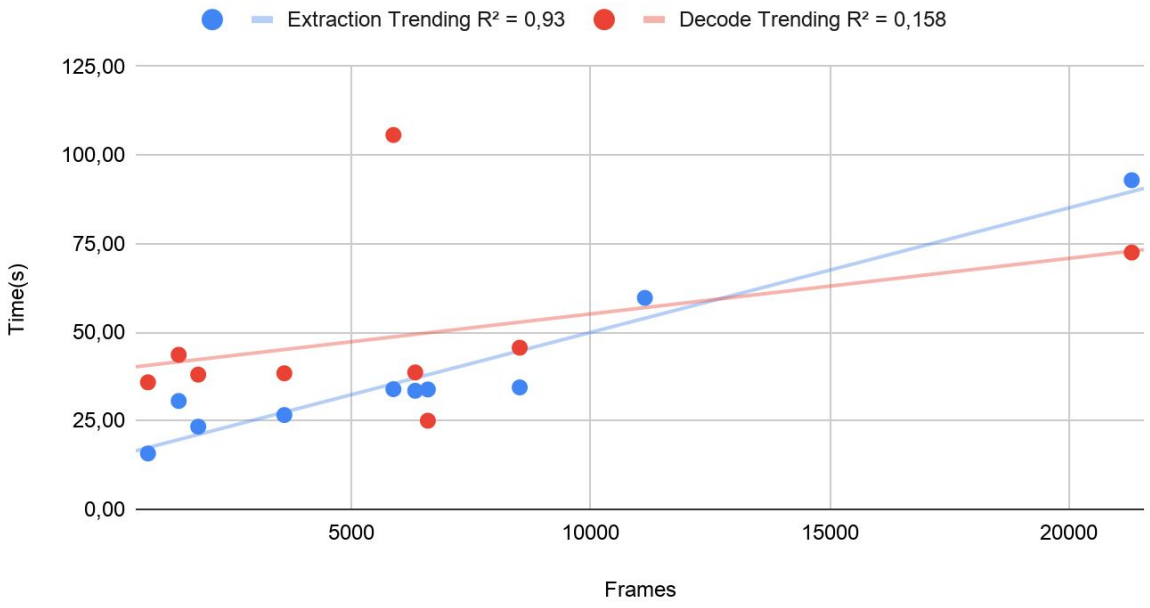

Figure 5.1 - Correlation between Decoding Time(red), Extraction Time(blue) and Number of Frames

Case 2 - Correlation between skip macroblocks and time

Another parameter that influences the decoding time of a video, and thus the extraction time, is the percentage of skip macroblocks.

Skip macroblocks are a special type of block for which the encoder does not encode bits in the bitstream, so it is up to the decoder to derive the blocks using its neighbors. The derivation can be of two types:

- Spatial Prediction → the block is reconstructed from its neighbors of the same frame (macroblock on the left and those on the top).
- Temporal Prediction → the block is reconstructed from its neighbors in past frames (considering the decoding order of frames).

The videos analyzed in this section have the following skip composition:

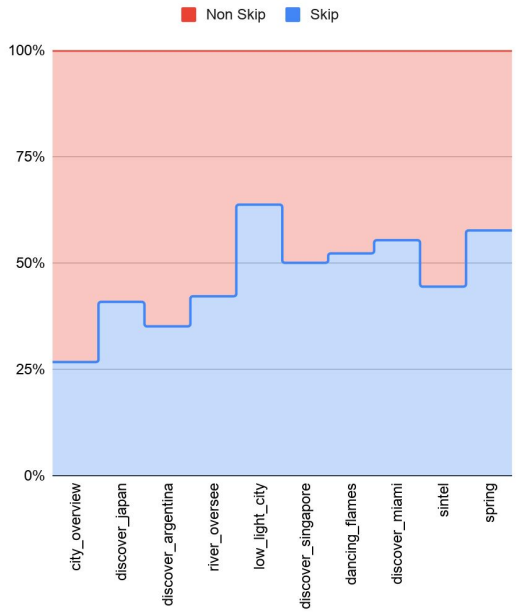

Figure 5.2 - Skip Percentages

Considering the decoding time per million of macroblocks (Figure 5.3), it is possible to observe that there is a correlation between the percentage of skip macroblocks and the decoding/extraction time.

In particular, videos with a high percentage of skip macroblocks achieve a lower score of seconds per million of macroblocks. For example, discover_miami or sintel, which have both a skip percentage around 55%, achieve a score of 0,95 and 1,1 seconds per million of macroblocks.

On the contrary, city_overview, which has the least amount of skip macroblocks (only 26,71%), has also the highest score of 2,62 seconds per million of macroblocks.

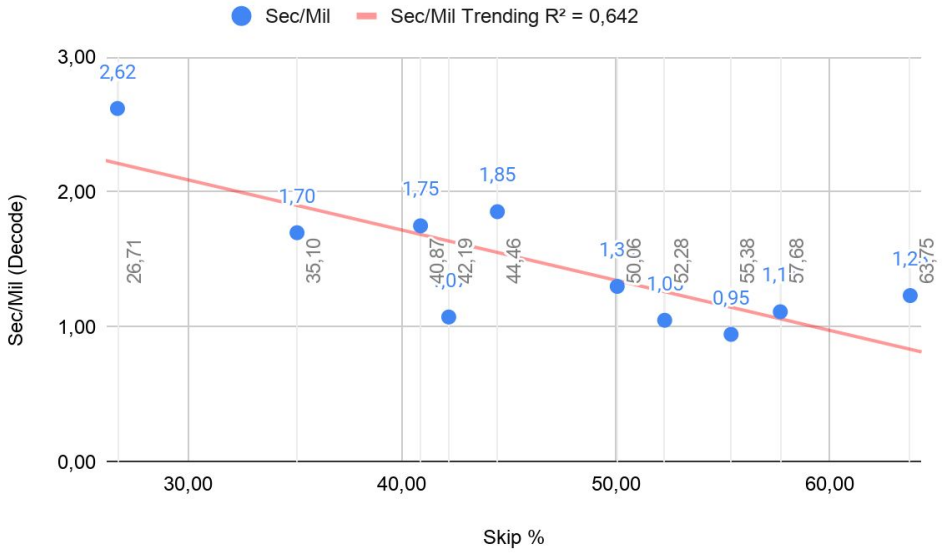

Figure 5.3.a - Correlation between Skip Macroblocks and Decoding Time

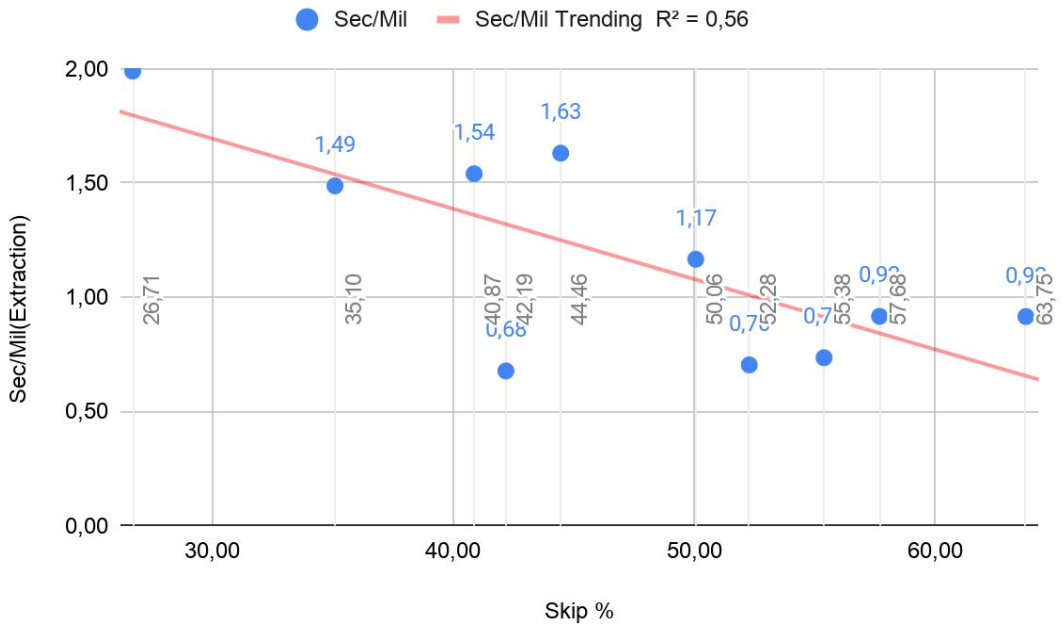

Figure 5.3.b - Correlation between Skip Macroblocks and Extraction Time

From the point of view of the extraction, the correlation with skipped macroblocks is weaker - the decoding time has a coefficient of determination of 0,64 against the 0,56 of the extraction time.

The reason behind this behavior is the absence of the quantization process and the IDCT in the decoding process. By skipping those two actions, the decoder saves more time for skipped macroblocks, while the extractor skips those steps for any macroblock, so the gain in speed is limited.

Case 3 - Correlation between Non-Zero Motion Vectors and Time

One last thing that influences the extraction time is the amount of non-zero motion vectors in the video.

The decoder and the extractor perform the same amount of operations for every motion vector, both if they are zero or non-zero, but while the writing of zero motion vectors can speed up through directives, the writing of non-zero motion vectors cannot speed up.

Analyzing the trending (Figure 5.4), it is possible to see how non-zero motion vectors are the parameter that most of all influence the extraction time, with an R2 coefficient of 0,8.

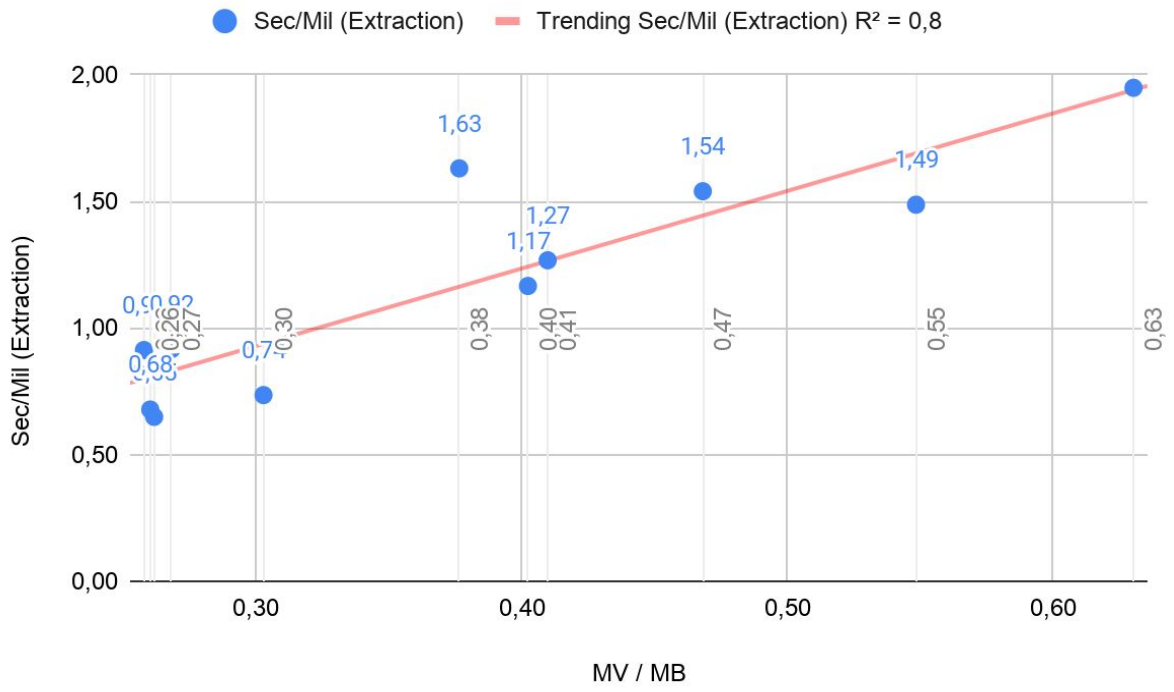

Figure 5.4 - Correlation between Non-Zero Motion Vectors and Extraction Time

Video	Total MV (10^6)	Total MV (10^6)	Non-Zero MV/MB (CAVLC)	Non-Zero MV/MB (CABAC)
city_overview	6,46	8,46	0,64	0,84
discover_japan	10,17	15,34	0,47	0,71
discover_argentina	12,48	18,56	0,55	0,82
river_oversee	6,36	12,39	0,27	0,53
low_light_city	7,52	8,54	0,26	0,29
discover_singapore	11,88	17,57	0,40	0,60
dancing_flames	11,73	25,30	0,27	0,58
discover_miami	13,97	28,87	0,30	0,63
sintel	33,27	21,45	0,58	0,38
spring	32,74	17,46	0,50	0,27

Table 5.5 - Macroblock with Non-Zero Motion Vectors

From this data, it is also possible to understand why river_oversee and spring have such strange behavior and their decoding and extraction last longer if they are longer than other videos.

In particular, river_oversee has a density of non-zero motion vectors of 0,53, against the 0,82 of discover_argentina. The same happens for spring, which has a density of 0,27 non-zero motion vectors per macroblock, while sintel has a density of 0,38. Accordingly, their extraction times are 92,92s for sintel, against 59,70s for spring.

Section 5.4 - Comparison between CAVLC and CABAC

As explained in Chapter 4, the residual decoding of the bitstream is necessary to keep the synchronization intact and continue to extract information without any errors. Still, as it is visible from Tables 5.4, CAVLC and CABAC are different in terms of complexity and speed:

- CAVLC achieves a lower decoding complexity but also a lower compression ratio, so lower extraction time.
- CABAC achieves a higher decoding complexity but also a higher compression ratio, so higher extraction time.

Figure 5.5 shows the results from the extraction process using the same videos with two entropy coding techniques.

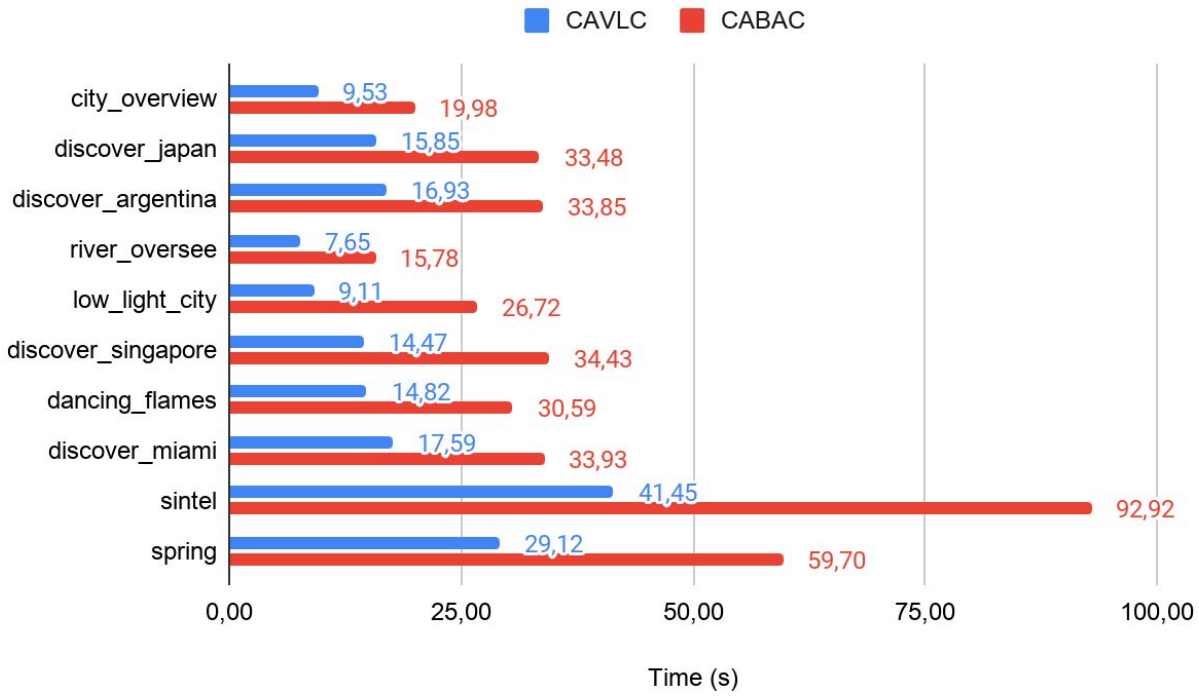

Figure 5.5 - Comparison between CAVLC/CABAC Timings

Since CABAC is more complex, and the extraction of information from the bitstream involves more instructions (Chapter 4), also the extraction time is longer. Nevertheless, as it is possible to observe from Table 5.5, CABAC videos contain a major number of non-zero motion vectors compared to the CAVLC counterpart, so more operations to perform. And since the extraction time is highly correlated with the number of non-zero motion vectors to calculate (as explained in section 5.3 Case 3 - Figure 5.4), the extraction time is higher.
In general, analyzing the data, it is possible to assume that:

$$ExtractionTime_{CABAC} \geqq 2 * ExtractionTime_{CAVLC}$$

Section 5.5 - Difference in the achieved time

In the comparison between the temporal achievements of the developed extractor (Extractor 2) and intermediate solution described in Chapter 3 (Extractor 1), there is a discrepancy in the results achieved, mostly because the extractor derived from FFmpeg still executes some operations that are impossible to eliminate without re-writing the code.

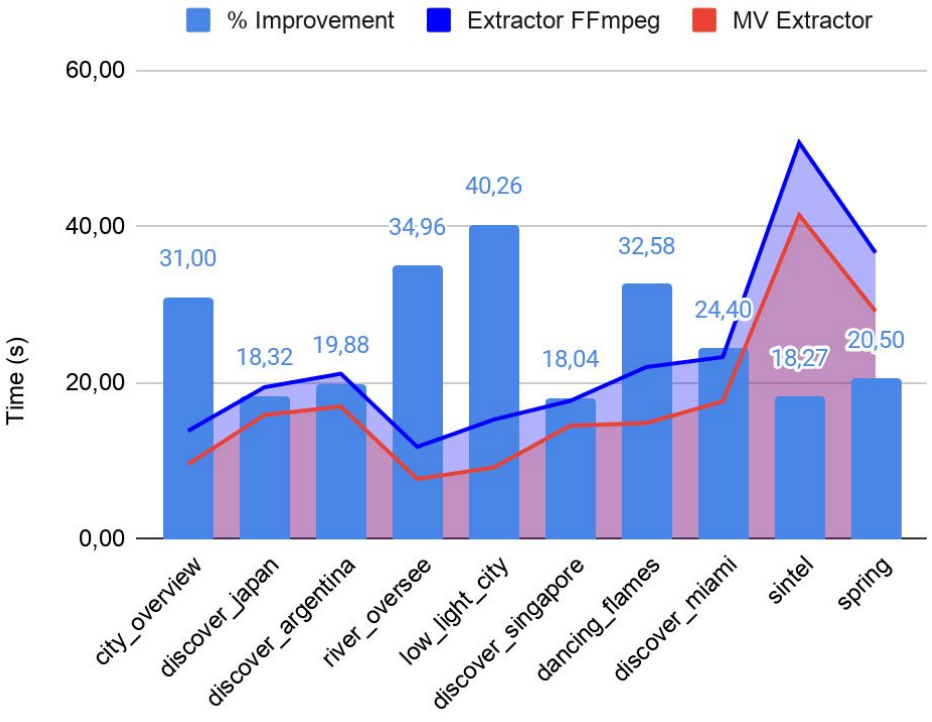

Figure 5.6 - Difference between Motion Vector Extractors and the Improvement (CAVLC)

Figure 5.6 shows how the developed extractor performs always better than the derived extractor since operations like the creation of the AVFrame, its allocation, and its management, so as the quantization process, are removed from the process.
Also, in some cases, the improvement is significantly better, mainly due to the amount of non-zero motion vectors to extract.
The improvements are calculated with the following formula:

$$Improvement = -(V_2 - V_1)/V_1$$

where V1 is the initial value, and V2 is the improved value.

Section 5.6 - Memory Usage

Video files require a high amount of memory to be reproduced, because of the need to store the information about an entire frame, or sometimes even multiple frames. In the decoding process, every information is calculated, and it needs to be stored, but in the motion vector extraction process, only the information about the motion vectors, their caches, and the caches of the referenced frames need to be tracked.
The most important structures used in FFMpeg, in terms of used memory, are the following:

1. **H264Context**: is the context of the decoding process, and contains all the data that supports the process itself. In the extraction process, some information like the Error Context, the Chroma Context, the QPel Context and, most of all, the Prediction Context are unnecessary.
2. **H264SliceContext** is the context of the decoding process for a single slice, and it is reused for every slice in both the extractor and the decoder. It contains information about the Error Context and the context of the picture in which it is contained, like the borders, the dc components or the luma and chroma caches
3. **H264Picture**: contains all the information about a particular picture, including the frame to be displayed. A picture, differently from the frame, contains a larger amount of information, from the motion vector caches, to the picture order counter, to the frame itself.

   For the extraction of the motion vectors, the frame is an unnecessary structure to keep in memory, especially in situations where pictures are out of display order and need to be delayed. Other structures unnecessary are the Thread Context, because the

extractor is a single thread, and the Qscale cache since the quantization is removed from the process.

All those structures considered, the amount of memory used by the extractor is highly limited in comparison to the decoder. The two figures below show the amount of megabytes used by the CAVLC (Figure 5.7) and CABAC videos (Figure 5.8).

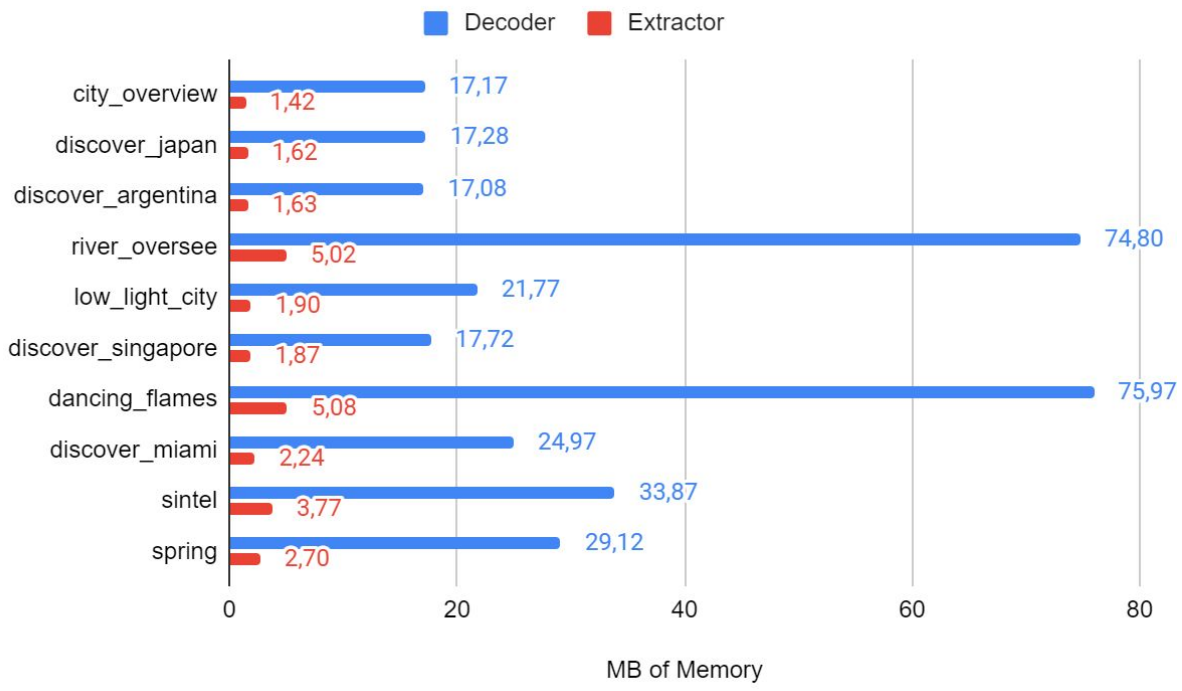

Figure 5.8 - Megabytes of Memory used by CAVLC Videos

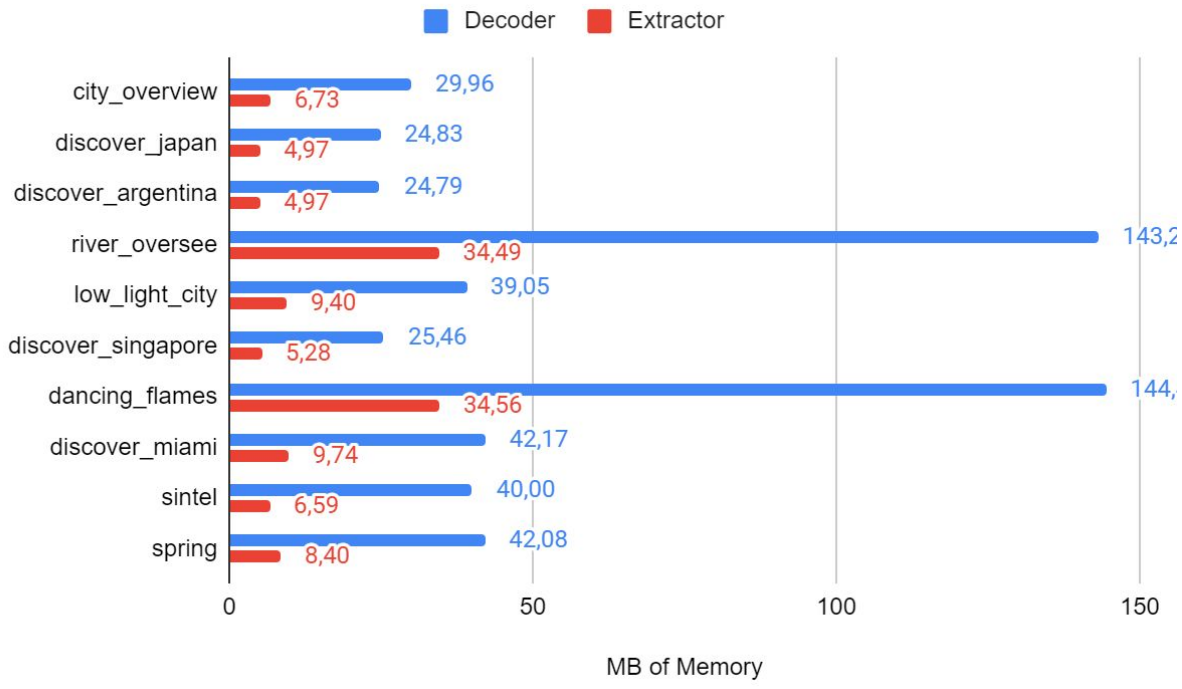

Figure 5.9 - Megabytes of Memory used by CABAC Videos

Using the formula in Section 5.5, an average improvement of 91,08% is obtained for CALVC videos, while CABAC videos can reach only an average improvement of 78,51%. The difference is mainly caused by the higher complexity CABAC has to deal with.

In particular, since the videos used for CAVLC analysis are all encoded in the Baseline Profile, while those using CABAC are encoded in Main Profile, only CABAC will have to deal with B-Frames, so it needs to keep more referenced frames than CAVLC in memory.

This result is mostly a limit of the software used to convert the videos between different profiles, which only allows using CAVLC in the Baseline Profile of H264, and CABAC for the Main Profile.

Chapter 6 - Conclusion

The objective of the thesis was to efficiently extract the motion vectors from the bitstream trying to improve the efficiency in terms of time and memory, taking as reference the decoding process of the FFmpeg library.

The study reached a positive outcome. Still, there are some differences between the improvements of the extractor, depending on different factors.

In general, there is a significant gap between CABAC and CAVLC entropy decodings. Due to its simplicity, the extraction of CAVLC reaches an average improvement of 29,31% (from the analyzed videos). On the other hand, since CABAC uses a more complex structure and reads the bitstream through more operations, its improvement is capped. In particular, it only reaches a mean value of 20,64%.

These kinds of enhancements have been reached by skimming the decoding process and removing the useless parts for the decoding process (e.g. quantization, IDCT, and motion compensation).

As it is possible to notice, the expected improvement estimated in Figure 1 (Chapter 1) has been reached. Also, since CAVLC can skip some of the quantization part of the residual decoding (less memory writes), it's improvement is higher. Contrarily, CABAC needs those informations to keep its internal context synchronized, so its improvement reaches a result close to the expected one.

In this case, the goal of H264 to obtain the highest compression rate possible conflicts with the aim of the thesis. The extraction of motion needs a fixed amount of residual data to skip from the bitstream to reach the theoretical improvement, while H264 encodes the residual data in a variable amount of bits (the least possible) to achieve a high compression ratio. Also, some other information needs to be managed to retrieve the motion vectors in the correct order (delayed picture buffer and references), which adds up to the total extraction time.

In conclusion, this approach doesn't allow a higher improvement using the H264 standard.

Even if the improvement does not reach the maximum expected value in terms of speed, it enhances another video analysis feature: memory usage. In particular, the extractor reached heights of improvement of 91% and 78%, respectively for CAVLC and CABAC entropy codings. For this reason, it would perform easily on systems with limited memory available, for example, embedded systems dedicated to the analysis of movement in a video.

The purpose of the thesis was to extract the motion from a video in the fastest way possible, to allow its analysis in real-time, without any delay effects (i.e. the extraction time should never surpass the display time).

The success of the extraction opened up the field of motion analysis to an improvement in many operations. In particular, it will be possible to apply a threshold on the extracted motion vectors to evidence fast or slow movement in a frame and distinguish them. Also, using a fixed camera, it would be easier to spot movements in a frame and localize them (e.g. in security applications).

Furthermore, the usage of machine learning techniques on motion vectors will be encouraged. Developing a model based on these algorithms already requires a great amount of memory and time to perform, so improving both those expenses could become advantageous. In this field, it would be possible to develop many ideas, from the individuation of meaningful movement in the video, to a more complex algorithm able to predict the direction and position of an object based on its past movement.

Acknowledgments

I want to take a moment to thank my professor, Enrico Masala, who accepted to be my supervisor and allowed me to engage and finish the hardest project I have ever worked on, hopefully until today. His help was crucial to the writing of this thesis, and he guided me until I reached the expected results, especially when I thought I had failed.
Also, I want to thank the entire Addofor group for offering me this thesis and in particular Daniele Trainini, who was always available to help me solve any problem and segmentation fault I found throughout my path.

Yet, the most warm-hearted thanks are for my family, and in particular to my parents, not only for the financial sacrifices they endured but also for the emotional support they provided me. From them, I learned what a united family is, and I hope one day to build at least one-tenth of what you two were capable of. Also thanks to my sister, Alexandra, for being a constant support in my life, no matter the entity of the problems.

A special thank you to my beloved girlfriend, Gaia, for encouraging me to follow all my dreams, even when they were complete failures. You made me a better person.
And last, but not least, I want to thank all my friends.
My two towers: my co-host Davide, and my almost-mentor Alessio, both of you, in your ways and for different reasons, helped me reach this achievement. My team of numerous projects, Diego, Gianpiero, and Simone: we developed some awesome apps and games together. And all the rest of you, who took part in my path until this point, and hopefully from now on.

I dedicate this achievement to all of you.
Cosmin.

Bibliography and Sitography

1. Recommendation ITU-T H264 (02/2016) - Advanced video coding for generic audiovisual services (10th Edition)
2. FFmpeg Reference (version 4.0): http://www.ffmpeg.org/doxygen/4.0/index.html
3. Moving Object Detection in the H264 Compressed Domain - Spyridon K. Kapotas and Athanassios N. Skodras - Digital Systems and Media Computing Laboratory, School of Science and Technology, Hellenic Open University.
4. Gentle Logic - Bitstream Format (Example Images) - 15th February 2021 http://gentlelogic.blogspot.com/2011/11/exploring-h264-part-2-h264-bitstream.html
5. Specifics for Entropy Decoding in FFmpeg 1 - 15th February 2021 https://blog.csdn.net/leixiaohua1020/article/details/45114453
6. Specifics for Entropy Decoding in FFmpeg 2 - 15th February 2021 http://www.voidcn.com/article/p-vunryipt-bdq.html

Schemes:

[1] → Moving object detection in the H.264/AVC compressed domain - Marcus Laumer, Peter Amon, Andreas Hutter and André Kaup - November 2016

[2] → FlameGraph - https://johnysswlab.com/wp-content/uploads/ffmpeg-regular.svg

[3] → Using H.264 video compression in IP video surveillance systems - https://www.networkwebcams.co.uk

[4] → Performance Analysis and Comparison of the Dirac Video Codec with H.264/MPEG-4 Part 10 AVC - Aruna Ravi, Kamisetty Rao - July 2011

[5] → H264 Decoder Functions - http://scc.ustc.edu.cn

[6] → Exploring H264 Bitstream Format - www.gentlelogic.blogspot.com

[7] → Serial and Parallel FPGA-based Variable Block Size Motion Estimation Processors - Brian Li & Philip Leong - 2008

[8] → Smart Selective Encryption of H.264/AVC Videos using Confidentiality Metrics - Dubois, Puech & Blanc-Talon - December 2014

[9] → Entropy Coding in HEVC - Sze Vivienne & Marpe Detlev - 2014