Eetu Latja

Parallel Acceleration of H.265 Video Processing

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 12.5.2017

Thesis supervisor:

Senior university lecturer Vesa Hirvisalo

Thesis advisor:

M.Sc. Jussi Hanhirova

Aalto University
School of Science

AALTO UNIVERSITY SCHOOL OF SCIENCE	ABSTRACT OF THE MASTER'S THESIS

Author: Eetu Latja

Title: Parallel Acceleration of H.265 Video Processing

Date: 12.5.2017 Language: English Number of pages: 7+65

Department of Computer Science

Degree programme: Master's Programme in Computer, Communication and Information Sciences

Major: Software Technology Code: SCI3042

Supervisor: Senior university lecturer Vesa Hirvisalo

Advisor: M.Sc. Jussi Hanhirova

The objective of this study was to research the scalability of the parallel features in the new H.265 video compression standard, also know as High Efficiency Video Coding (HEVC). Compared to its predecessor, the H.264 standard, H.265 typically achieves around 50% bitrate reduction for the same subjective video quality. Especially videos with higher resolution (Full HD and beyond) achieve better compression ratios. Also a better utilization of parallel computing resources is provided.

H.265 introduces two novel parallelization features: Tiles and Wavefront Parallel Processing (WPP). In Tiles, each video frame is divided into areas that can be decoded without referencing to other areas in the same frame. In WPP, the relations between code blocks in a frame are encoded so that the decoding process can progress through the frame as a front using multiple threads. In this study, the reference implementation for the H.265 decoder was augmented to support both of these parallelization features. The performance of the parallel implementations was measured using three different setups.

From the measurement results it could be seen that the introduction of more CPU cores reduced the total decode time of the video frames to a certain point. When using the Tiles feature, it was observed that the encoding geometry, i.e. how each frame was divided into individually decodable areas, had a noticeable effect on the decode times with certain thread counts. When using WPP, it was observed that what was mostly synchronization overhead, sometimes had a negative effect on the decode times when using larger (4–12) amounts of threads.

Keywords: H.265, HEVC, High Efficiency Video Coding, video compression, decoder, parallelism, WPP, Wavefront Parallel Processing, Tiles

AALTO-YLIOPISTO	DIPLOMITYÖN
PERUSTIETEIDEN KORKEAKOULU	TIIVISTELMÄ

Tekijä: Eetu Latja

Työn nimi: Rinnakkainen toteutus H.265 videokoodaus standardille

Päivämäärä: 12.5.2017 Kieli: Englanti Sivumäärä: 7+65

Tietotekniikan laitos

Tutkinto-ohjelma: Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

Pääaine: Ohjelmistotekniikka Koodi: SCI3042

Valvoja: Vanhempi yliopistonlehtori Vesa Hirvisalo

Ohjaaja: DI Jussi Hanhirova

Tämän tutkimuksen aiheena oli tutkia uuden H.265 videonpakkausstandardin (tunnetaan myös nimellä HEVC (engl. *High Efficiency Video Coding*)) rinnakkaisuusominaisuuksien skaalautuvuutta. Verrattuna edeltäjäänsä, H.264 videonpakkaustandardiin, H.265 tyypillisesti saavuttaa samalla kuvanlaadulla noin 50% pienemmän pakkauskoon. Erityisesti suuren resoluution videoilla (Full HD ja suuremmat) pakkaustehokkuuden paremmuus korostuu. Huomiota on kiinnitetty myös moniydinprosessoreiden hyödyntämiseen videokoodauksessa.

H.265 tarjoaa kaksi uutta rinnakkaisuusominaisuutta: niin kutsutut Tiles- ja WPP-menetelmät (engl. *Wavefront Parallel Processing*). Tiles-menetelmässä jokainen videon kuva jaetaan alueisiin jotka voidaan purkaa viittaamatta saman kuvan muihin alueisiin. WPP-menetelmässä suhteet kuvan lohkoihin pakataan siten että purkamisprosessi pystyy etenemään kuvan läpi rintamana hyödyntäen useampia säikeitä. Tässä tutkimuksessa H.265 videodekooderin referenssitoteutusta laajennettiin tukemaan molempia näistä rinnakkaisuusominaisuuksista. Suorituskykyä mitattiin käyttäen kolmea eri mittausasetelmaa.

Mittaustuloksista ilmeni että prosessoriydinten lukumäärän kasvattaminen nopeutti videoiden purkamista tiettyyn pisteeseen asti. Tiles-menetelmää mitatessa havaittiin että alueiden geometrialla, eli kuinka kuva jaettiin riippumattomiin alueisiin, on huomattava vaikutus purkamisnopeuteen tietyillä säiemäärillä. WPP-menetelmää mitattaessa havaittiin että korkeampiin säiemääriin (4–12) siirryttäessä purkamisnopeus alkoi hidastua. Tämä johtui pääasiassa säikeiden keskinäiseen synkronointiin kuluvasta ajasta.

Avainsanat: H.265, HEVC, High Efficiency Video Coding, videonpakkaus, dekooderi, rinnakkaisuus, WPP, Wavefront Parallel Processing, Tiles

Preface

First, I want to express my gratitude to my supervisor Vesa Hirvisalo and my advisor Jussi Hanhirova for their support and guidance in this endeavour that took a little longer than originally was planned. Also other employees of the Embedded Systems Group at Aalto University, I want to thank you for the interesting conversations and for providing a pleasant working environment. Big thanks to Exove, my current employer, for giving me the time to finalize this thesis and for all the understanding and support.

I'm grateful for whatever combination of probabilities and forces of the universe that made me born in a loving family and has put me in situations that have led to this point. Too often we ignore the impact of others in our success. I want to thank the free Finnish education system for making it possible for me to study the things I'm passionate about. I'll promise to be worth the investment.

Finally, none of this would have been possible or made any sense without all the great people I have in my life. I want to thank my parents for the love and the support in the things I've gone after. Thanks to all my awesome friends for all the fun moments and experiences I've had with you. You've always been there for me. Especially, thanks to my long-time friend Petja. Rest in peace brother; you left us way too soon.

Helsinki, 11.5.2017

Eetu Latja

Contents

Chapter 1

Introduction

Digital video is present everywhere today; efficient video compression techniques and fast networks have made streaming services into everyday commodities, many countries have adopted or are currently transitioning into digital television, and optical storage media, such as DVDs and Blu-rays, are commonly used to distribute videos to consumers. None of this would be possible without video compression.

Raw video takes up a lot of storage space; it takes over 1 TB of space to store just the video data of a two-hour long Full HD movie. This would be impractical for consumers and streaming would be impossible using consumer-grade networks. Modern video compression techniques can reduce the size of the video up to 500 times of the original while retaining good subjective quality.

Currently one of the most used video compression standard is the H.264/MPEG-4 AVC standard. It is used in Blu-rays, internet streaming services, and HDTV broadcasts, just to mention a few.

In June 2013, a joint collaboration between ISO/IEC and ITU-T called JCT-VC, released the H.265 video coding standard, also known as HEVC (High Efficiency Video Coding). H.265 is intended to be a successor to the H.264 standard. Typically H.265 achieves around 50% lower bitrate for the same subjective video quality compared to H.264. Due to larger block sizes and divisible coding blocks, especially high resolution videos (Full HD and beyond) achieve better compression ratios when compressed using H.265.

H.265 introduces two novel parallelization features for block level parallelism: Tiles and Wavefront Parallel Processing (WPP). In Tiles, each frame is divided into rectangular areas which can be decoded independently. In other words, each area can be decoded using an individual thread without the need to synchronize with other threads. In WPP, the dependencies between the blocks in a frame are encoded so that the decoding can progress as a diagonal front through the frame using multiple threads.

The objective of this study was to implement and measure the scalability of parallel execution of the Tiles and WPP features. A data structure responsible for dividing the computational tasks between threads and synchronizing them, named job queue, was implemented. Using two different configurations, the job queue is able to implement parallelism for both Tiles and WPP. The performance of both features was measured using three different setups and four different videos.

In the measurements, Tiles method performed slightly better in almost all cases compared to WPP. For both methods, adding a few CPU threads introduced a noticeable speedup compared to the single thread case. Depending on the case, after 4–6 threads adding more threads had no or very little effect on the speedup.

From the Tiles measurements it was observed that the amount of tiles in the frame needed to correlate with the amount of threads to achieve maximum performance. If it does not, threads will spend more time waiting and the performance will be suboptimal.

For WPP, it was observed that the performance started to slightly decrease when going to high thread counts. This is mostly due to synchronization overhead. The same effect was not observed for Tiles as it does not require synchronization between the threads.

In chapter 2 the basic fundamentals of digital video and video compression are presented. In chapter 3 some typical environments where video codecs are used and needed are discussed. In chapter 4 the theory behind parallel computing is presented. In chapter 5 an overview of the H.265 video coding standard is provided and other similar research works are presented. In chapter 6 the parallel implementation of Tiles and WPP is explained. In chapter 7 the measurement setups are described. In chapter 8 the measured results are presented, analyzed, and compared to other similar research works. Chapter 9 concludes this thesis and summarizes the obtained results.

Chapter 2

Video compression

Compression is the act of representing some data using a fewer amount of bits than its original form. Video compression, or video coding, is the act of reducing the number of bits needed to express a digital video sequence. Raw, or uncompressed, video data has a very high bitrate and it must be compressed for practical storage and transmission. To give some perspective, just the raw video data of a two hour Full HD quality movie takes 1.22 TB (assuming 24 bits per pixel and 30 FPS), whereas one may find compressed Full HD movies compressed to a few gigabytes without a noticeable reduction in quality.

The act of compressing digital video is also called encoding and the program responsible for this is called an encoder. Respectively, the decompression is called decoding, which is done by a decoder. The encoder/decoder pair is called a codec.

Compression can be either lossless or lossy. In lossless compression, the video stream reconstructed by the decoder is identical to the original video stream. If lossy compression is used, the reconstructed stream is an approximation of the original stream. Lossy compression tries to decrease the amount of redundant data without affecting the subjective quality perceived by the viewer. The methods used in lossy compression are largely based on limitations and functioning of the human visual system. Using lossless compression, a compression ratio of only around 3–4 can be achieved, whereas lossy compression can give ratios from few hundreds up to even a thousand. Many features used in video compression are also used when compressing images, such as block coding, transforms, quantization, and prediction based on adjacent blocks [1].

In this chapter we will first discuss about the general structure of a video decoder. Then different prediction models are introduced, namely intra and inter prediction, and it is discussed why data redundancy and motion are important for video coding. Finally transformations, quantization, and entropy encoding are discussed.

The information presented in this chapter is mainly based on [2].

2.1 Codec structure

As mentioned before, a codec encodes a source video sequence into a compressed form and decodes this to produce a copy (lossless compression) or an approximation (lossy compression) of the source video. The codec represents the original video using a model, which is a data efficient representation that can be used to construct a copy or an approximation of the original video. An ideal model is such that it represents the video using as few bits as possible with as high a quality as possible. Usually, these two goals are conflicting, and video coding often is a tradeoff between quality and bitrate. Most modern video coding formats employ a so called hybrid model, where the coding scheme combines prediction and transforms for the prediction error [3].

A video encoder consists of three main functional units: a prediction model, a spatial model, and an entropy decoder. The input to prediction model is the raw video sequence, whose redundancy the model tries to reduce by using the correlation of adjacent video frames. Output of the prediction model is a residual frame, which created by subtracting the predicted frame from the original frame, and the model parameters used to achieve the prediction.

The residual frame is used as an input for the spatial model. In the spatial model, the residual samples are transformed into another domain. The transform should be chosen so that the best possible compression rate is achieved in the entropy decoder. The transform should be done into a domain where most of the information is concentrated on few samples and more insignificant samples can be dropped out. After the transformation, the samples are quantized, which is simply a decision of the precision used to code the samples into the decoded bitstream.

After the spatial model, the model parameters from the prediction model and the transformed samples from the spatial model are given as an input to the entropy decoder. Entropy decoder uses statistical compression techniques, i.e. the same ones used to compress arbitrary binary data, to reduce statistical redundancy in the data. The entropy decoder outputs compressed bitstream, which can then be stored and/or transmitted.

A video decoder does the same process as an encoder, except in an opposite order. An entropy decoder decodes the parameters and transformed samples from the bitstream, from which the spatial model reconstructs the residual frame (or an approximation of it if the coding is lossy). The decoder uses the prediction parameters and previously decoded frames to construct the predicted frame. The residual frame is then added to the predicted frame to obtain the original frame (or the approximation of it).

2.2 Redundancy in video data

Video coding usually exploits spatial and temporal redundancy to achieve high compression ratios. In the temporal domain, i.e. adjacent frames, the frames are usually highly correlated. The correlation is higher the slower the motion is and the faster the sampling rate, or frame rate, is. In the spatial domain, pixels tend to correlate with neighboring pixels. Large areas with little details have a much higher compression rate than detailed areas.

2.3 Prediction

The aim of prediction is to reduce data redundancy by forming a prediction of the current frame. This prediction is subtracted from the original frame, producing a residual frame which is then feeded to the entropy decoder. The reason for all this is that the residual frame has less entropy than the original frame, and therefore the entropy decoder can compress the residual frame more efficiently than the original frame. The more accurate the prediction is, the less entropy the residual contains. But there is a flip side to prediction: higher prediction accuracy means more prediction parameters. Therefore, one problem the encoder must solve is to find a prediction so that the entropy compression of the prediction parameters and the residual frame is as small as possible.

The prediction can be done from previously coded frames by exploiting temporal redundancy, called inter prediction or temporal prediction, or from previously coded image samples in the same frame by exploiting spatial redundancy, called intra prediction or spatial prediction. When constructing the prediction, it is essential that the encoder uses only data that is available to the decoder. Therefore, when inter prediction is used, the encoder must reconstruct the previously encoded frames like the decoder does and make the predictions and residual calculations using these reconstructed frames.

2.4 Inter prediction

In inter prediction, the current frame is predicted from one or more previously decoded frames. The decoding order is not necessarily the same as the display order; frames can also be predicted from future frames, if they precede in the decoding order. The frames used as source for the prediction are called reference frames.

The simplest way to predict is to use the previous frame as the prediction of the current frame. The residual frame is then simply the subtraction of the two consecutive frames. This prediction works for the parts of the frame that are not moving, but gives poor results for the parts that are moving. This results in high amounts of entropy in the areas of motion.

2.5 Motion

The fundamental idea of moving picture is to represent movement. Causes of motion are rigid body motion, deformable object motion, camera motion, and lightning changes. These changes cannot be predicted by using the subtraction of consecutive frames. A better prediction technique is to compensate for motion between two frames. This is called motion compensation. [4]

One can estimate the trajectories of each pixel between two frames, the process of which is called motion estimation. It produces a vector field known as an optical flow, whose elements are called motion vectors. From optical flow it is possible to form an accurate prediction for most pixels in the frame. However, there are several reasons why this method is not practical for motion compensation. First, it is a computationally intensive task to calculate the motion vectors for each pixel. Second, the motion vectors must be encoded in the bitstream. If the motion vectors for each pixel are encoded in the bitstream, this causes the bitrate to increase drastically and the benefit of the small residual is lost.

2.6 Block-based prediction

A practical method of motion compensation is to divide the frame into rectangular areas, called blocks, and compensate their movement. All modern video coding standards support this method.

An optical flow for block prediction is done like an optical flow for each pixel. For each block in the current frame, the reference frame is searched for an area that minimizes the residual. Instead of getting a motion vector for each pixel, a motion vector for each block is acquired. Depending on the used block size, the optical flow of block prediction usually requires significantly less bits to transmit.

Block-based motion compensation is relatively straightforward and computationally feasible, blocks fit well in rectangular video frames, and it is easy to combine with block-based transforms, which we will focus on later. However, there are several disadvantages in block-based motion compensation: moving objects usually do not have neat edges that match the rectangular blocks, they often move a fractional amount of pixels, and many types of motion are are hard to compensate for using block-based methods, e.g. snowfall, smoke, deformable objects, and rotation. Despite the disadvantages, the advantages are often larger and block-based methods are used in all current video coding standards. The blocks can be fixed size or can be broken down into smaller blocks [5].

2.7 Intra prediction

Intra prediction makes use of the spatial redundancy found in individual frames. Frames often contain a lot of smooth areas where the color stays the same or varies gradually. This can be used to predict values of pixels based on the values of the surrounding pixels. In intra prediction, parts of the current frame are created using the already decoded parts in the same frame. Similar techniques are used when compressing still images. [6]

Intra prediction must be used when no other frames are available for inter prediction. Intra frames are included into the video streams at small interval so that jumping to a position in the video is possible. It is also necessary in live streams to include intra frames so that new streamers are able to join and existing viewers are able to get back on track in cases when data is lost due to network issues.

2.8 Transformation

In the transformation stage, the residual video data from the prediction stage is transformed into a transformation domain. The purpose of the transformation is to represent the residual in a domain where its components are as decorrelated as possible. This way the most insignificant components can be discarded, thus reducing the amount of data in the output bitstream.

Transformation used in video coding fall to two categories: image-based and block-based. Block-based transformations operate on small rectangular areas. They have small memory requirements and naturally match with block-based prediction, but they tend to produce artefacts at the block boundaries. Popular examples of block-based transformations are Karhunen-Loeve Transformation (KLT), Singular Value Decomposition (SVD), and Discrete Cosine Transformation (DCT) and its approximations [7].

Image-based transformations operate on the entire frame or a larger portion of it. Image-based transformations beat block-based transformation for still image compression, but they have larger memory requirements and do not perform so well with block-based prediction methods. The most popular image transformation is the Discrete Wavelet Transformation (DWT).

2.9 Quantization

In quantization stage, a range of values is mapped to a reduced range of values. The idea behind this is that the quantized values can be represented using fewer bits than the original values. Quantization naturally reduces the quality of the signal as many different input values are mapped to the same output value. The reduction of quality depends on how many possible output values exist. Fewer output values requires less bits to represent, but results in worse quality. Quantization is the most significant cause of loss in video coding.

A scalar quantizer maps a single scalar value to a single scalar value. A simple example is rounding fractional numbers to the nearest integers. A more general example of this is the use of quantization parameter (also known as step size), which is used to divide the value before rounding. The quantization parameter can thus be used to control the amount of possible output values.

A vector quantizer maps a set of data to a single value, called codeword. When this value is decoded, it is mapped to an approximation of the original set of input data. The set of these vector values is called a codebook. The key problem in vector quantizer design is to find a codebook that is as small as possible, but can approximate all possible values without generating much error.

2.10 Entropy encoding

The final stage of video encoding is the entropy encoding. Entropy encoding is a name for general lossless data compression methods that are independent of the type of the data being compressed. The basic idea in any entropy compression technique is to find the most abundant symbols in the data and represent those using the least amount of bits.

In video compression, entoropy decoding is used to store the prediction residuals, prediction modes, motions vectors, etc. into the bitstream. Often high-level syntax elements are encoded using less complex fixed length codes. The syntax elements are interesting not only to the decoder, but can also be used by other applications to determine the features of the bitstream. Everything related to the compressed video data is usually encoded using variable length codes, which are more complex but offer better compression ratio compared to fixed length codes.

2.11 Video coding standards

The development of video coding standards has been going on since the 1980s when the first video coding standard H.120 was released in 1984. The H.264/MPEG-4 AVC, published in 2003, is currently the most used video coding standard [8].

The development of video coding standards has been driven by two application spaces: real-time video communication and distribution or broadcasting of video content. The ITU Telecommunication Standardization Sector (ITU-T) and the International Standardization Organization/International Electrotechnical Commission (ISO/IEC), have been publishing video coding standards for these purposes both individually and in collaboration. The latest video coding standard released from colloboration between ITU-T and ISO/IEC is the H.265, or HEVC, video coding standard. The video coding standards released by ITU-T and/or ISO/IEC are listed in Table 2.1.

Table 2.1: The video coding standards published by ITU-T and/or ISO/IEC [9].

ITU-T	MPEG	Also known as	Published
H.120			1984
H.261			1988
	MPEG-1 Part 2		1993
H.262	MPEG-2 Part 2	MPEG-2 Video	1995
H.263			1996
	MPEG-4 Part 2	MPEG-4 Visual	1999
H.264	MPEG-4 Part 10	Advanced Video Coding (AVC)	2003
H.265	MPEG-H Part 2	High Efficiency Video Coding (HEVC)	2013

In addition to the proprietary standard published by ITU-T and ISO/IEC, other organizations have released open and royalty free coding standards. Xiph.org Foundation has released its Theora videocoding standard in 2008 and is currently developing its successor, Daala. Daala aims to be technically superior to the H.265 standard in terms of performance. [10]

Google has released its open and royalty free video coding standards VP8 and VP9 in 2008 and 2013, respectively [8]. Today, Google has joined forces with companies like Netflix, Intel, Amazon, etc., to form Alliance for Open Media (AOMedia) [11]. The goal of the organization is to develop royalty free video compressions standards as an alternative to the proprietary, patent protected video coding standards. The organization is currently developing its AOMedia Video 1 (AV1) standard.

Chapter 3

Codec usage environments

Digital video can be used in many different situations and by various hardware devices. In this chapter we will introduce the most common use cases of digital video and codecs. First we will discuss the recording of video and storing the video as files. Next we introduce portable video storage formats and discuss DVD and Blu-ray formats in more depth. Following this streaming and broadcasting of digital video is discussed. Finally digital video in the context of video communication is discussed.

3.1 Recording video

Most consumer devices capable of video capture, such as smartphones, store the recorded videos in a lossy compression format. This is due to the large size of uncompressed or losslessly compressed video. Only the most high-grade professional devices usually store the captured video in an uncompressed or lossless format. This requires a few orders of magnitude more storage space compared to storing using lossy compression, but the picture quality is not reduced.

3.2 Video files

Probably the most simple use cases for codecs are

- encoding an uncompressed video stream (e.g. originating from a digital video camera) into a compressed video file, or in other words, creating and storing a video file,
- and decoding a compressed video file to an uncompressed video stream (e.g. to be displayed by a video player), or in other words, playing back a video file.

When recording video, the encoder should be able to either

1. encode the uncompressed video stream in real-time
2. or have a large enough buffer to hold the incoming uncompressed video data and finish the encoding after the recording has been stopped.

When playing a video, the decoder should be able to either

1. decode the compressed video stream in real-time
2. or have enough decompressed video in its buffer before the playback is started in order to view the video without a lag.

3.3 Transcoding

Transcoding is the process of taking a video stream compressed with some coding format (format A) and transforming that stream to another coding format (format B). Essentially this means

1. decoding the original video stream using a decoder capable of decoding format A to an uncompressed format
2. and encoding the uncompressed video stream using an encoder capable of encoding the uncompressed format to format B.

Transcoding can change the format of a video from one coding standard to another or inside a video format, for example, by converting to a lower quality. Transcoding between lossy formats is usually a lossy process, causing distortion known as the generation loss. [12]

3.4 Portable video storage media

Portable video storage formats (such as DVD and Blu-ray) can be used to distribute video using physical media. The video data is stored into the physical medium using some compression format. A special hardware device (such as a DVD or Blu-ray player) is required for reading the compressed video stream from the physical media and a decoder is required to uncompress the compressed video stream.

The compressed video stream is read from the medium and fed to the decoder in real time. The decoder decodes the compressed video stream and the device forwards the uncompressed stream to an output port. From the output port it can be received, for example, by a monitor.

3.4.1 DVD

DVD-Video is currently, among with Blu-ray, one of the most used formats to store video on optical discs. More specifically, DVD-Video stores video data on DVDs (digital video disc). The DVD-Video specification was created by DVD Forum which is an organization comprised of multiple software, hardware, and media companies.

The contents of a DVD-Video are stored on the disc using VOB (Video Object) container format. VOB container files can contain the video, audio, subtitles, menus and navigation contents. The video data in a VOB container can be encoded using one of two coding formats:

- H.262/MPEG-2 Part 2,
- or MPEG-1 Part 2.

DVDs can contain the video data on one or both sides of the disc. Each side can be single- or double-layered. The size of a single-sided DVD varies from 4.37 GiB (single-layer) to 7.95 GiB (double-layer) and, depending on the characteristics of the content, it can hold up to around 4 hours (double-layer) of video.

The available resolutions for DVD-Video range from 352×240 to 720×576 whereas the frame rate can be either 25 or 29.97 frames per seconds. The maximum bitrate is 9.8 Mbps (without sound), but the "typical" bitrate varies from 3.5–6 Mbps. [13]

3.4.2 Blu-ray

Blu-ray, also Blu-Ray disc (BD), is the optical disc data storage format designed to be the successor of DVD. Blu-Ray was developed by Blu-ray Disc Association (BDA), an organization consisting of multiple electronics, software, and media companies.

The contents in consumer Blu-Rays videos are stored into the disc using Blu-ray Disc Audio-Video (BDAV) MPEG-2 Transport Stream (M2TS) container format. The video data in BDAV can be encoded using one of the following coding standards:

- H.262/MPEG-2 Part 2
- H.264/MPEG-4 (AVC)
- VC-1/SMPTE 421M

Originally Blu-Ray video supported resolutions from 720×480 to 1920×1080 and frame rates from 23.976 to 59.94. For progressive scan videos with resolutions larger than 1280×720, only frame rates up to 24 frames per second were supported. The maximum video bitrate of Blu-ray Video is 40 Mbps. [14]

The original Blu-Rays can be single- or dual-layered, with storage capacity of 25 GiB or 50 GiB, respectively. In June 2010, BDA published BDXL specification which introduced triple- and quadruple-layered discs with capacity of 100 GiB or 128 GiB, respectively. BDXL discs require a device capable of reading them and are not compatible with pre-BDXL Blu-Ray devices.

In May 2015, Ultra HD Blu-ray specification was published by Blu-ray Disc Association. Using H.265/MPEG-H Part 2 (HEVC) coding standard, Ultra HD Blu-ray supports 4K UHD resolution (3840×2160) with frame rates up to 60 frames per second using progressive scan. Three disc sizes are available; 50 GiB, 66 GiB, and 100 GiB; and the maximum video bitrate is 100 Mbps. Ultra HD Blu-ray discs are not compatible with pre-existing Blu-ray devices. [15]

3.5 Progressive downloading

Before discussing video streaming, a similar yet simpler and more restrictive technique for transmitting and playing back video is progressive downloading. In progressive downloading, the user starts downloading the video like any other file, but also starts playing back the video while it is still on transit. This requires a format that can be played back before the whole download is finished.

With traditional progressive downloading, the playback can not seek to a point that has not yet been downloaded. This means that for in order to jump to a specific point in the video, the whole video until that point has to be downloaded.

3.6 Streaming

Video streaming means sending video data over a network in a manner that it can be played back without first downloading the whole video file. When streaming digital video, the data is split into small chunks which are individually transported to the receiver. The receiver decodes and plays back these blocks one after another. [16]

Depending on the bitrate, which is affected by the coding efficiency and the resolution, the bandwidth of the network must be high enough to transmit the blocks in real-time or the playback will start lagging behind.

In contrast to progressive playback, streamed videos can be seeked without downloading the entire video up to the seek point. This is usually achieved using the following techniques:

- The client makes requests to the server to transmit the video data for a given point in time to which the server sends the correct chunk. This requires that both the client and the server understand the streaming and control protocols.

- When starting the stream, the client downloads a manifest file containing metadata of all all the chunks in the video. Based on the metadata, the client requests the desired chunk. The server stores every chunk as a separate file and upon client requests, it sends these files to the client. Since the client is responsible for choosing the correct chunk, no special streaming server is required. Any server capable of sending files is sufficient.

Traditionally video streaming protocols have used specific ports, which often causes problems with firewalls. In recent years, HTTP based streaming protocols, such as HTTP Live Streaming (HSL) [17] and MPEG-DASH [18], have been introduced. Working on top of HTTP, these protocols do not require any additional firewall configuration if HTTP is already enabled.

3.7 Live streaming

Compared to streaming an already existing video, live video streaming works by transmitting the video instantly after it has been recorded, almost in real-time. This method can be used to broadcast events, such as sports, in real time. In order for the sending stream to not start lagging behind, it must be able to encode the raw video stream in real time. [19, 20]

The size of the buffers in the sending and receiving end affect the experience of live streaming. The larger the buffers are, the more stable the stream is, but a larger buffer causes more delay in the transmission. Usually some amount of delay (seconds) is acceptable when watching a live broadcast. After all, live broadcasts usually already have an intentional few second delay to prevent undesirable material from airing.

In order to transmit to people with different network bandwidths and device capabilities, multiple streams with different bitrates must be encoded.

3.8 Digital television

In many countries, analog television broadcasts have been switched off or the transition from analog to digital broadcasts is underway. In digital television, the video signal is encoded digitally and broadcast using the television network. In order to view the digital television transmissions, a device capable of understanding and decoding the digital signal is required. From the perspective of the video stream, digital television has a lot in common with streaming.

Digital Video Broadcasting (DVB) is one set of digital television standards used in Europe and Australia, among others [21]. The currently widely used DVB standard for terrestial video, DVB-T, can support video encoded using MPEG-2, MPEG-4, and H.264 standards. More advanced coding formats are available using the newer

DVB-T2 version of the standard. For example, Germany started commercial DVB-T2 broadcasts using H.265 encoding in May 2016 [22].

3.9 Video call

A video call is a call between two participants which contains both video and audio. Essentially, a video call contains two real-time video streams, one from each participant. Therefore, both participants must be able to encode their recorded video and decode the stream sent by the other participant.

Compared to real-time broadcasting, a smaller delay is required in video calls. When watching a live broadcast, a delay of multiple seconds does not hinder the quality, since no two-way interaction is required. When conversing via a video call, long delays will reduce the conversation experience, as the other recipient will appear unresponsive. Therefore, in video calls it is usually desirable to reduce the buffer size and picture quality in order to make the delay as small as possible.

Modern browsers support WebRTC which is a collection of standards, protocols, and JavaScript APIs users can use for peer-to-peer communication using web browsers. With `RTCPeerConnection` interface users can communicate using video and audio. [23]

3.10 Videoconference

Videoconferencing is the technique of allowing multiple locations to participate in a conference using two-way video and audio transmissions. Compared to video calls, videoconferencing aims to connect multiple conferences or locations, rather than individuals, together. [24, 25]

The individual participants can be combined into a single videoconference using either two ways:

- Using a Multipoint Control Unit (MCU). An MCU is a component (either software, hardware or a mixture of both) which combines multiple incoming video connections into a single outgoing video stream. The outgoing video stream is transmitted back to all participants.
- A technique called Decentralized Multipoint allows videoconferencing without a central component. Each participant transmits their stream directly to all other participants in the conference. This allows a better quality and smaller delay, since the stream does not have to be relayed through a central component, and the availability of the central component is not a concern. However, the amount of traffic in the network is increased which may cause problems.

Videoconferencing can work in two operating modes:

- In Voice-Activated Switch (VAS) mode the participant with the loudest voice level will be seen by other participants. The output stream can be the same as the incoming stream of the loudest participant so no decoding is necessarily needed. However, if adaptive bitrate is supported, transcoding is needed in order to generate streams with different bitrates.
- In Continuous Presence (CP) mode the incoming streams are combined into output streams containing multiple participants. Usually a larger area in the picture is reserved for the participant who is currently speaking. For each participant, a different stream can be generated, for example, the participant's own video might not be included. In CP mode, the incoming streams must be decoded and then combined into multiple output streams, which must be encoded.

Chapter 4

Parallel computation

Parallel computation is a form of computation where multiple computations are carried out simultaneously. Traditionally, computer programs have been executed in serial, i.e. a single processor executing instructions one after another. In parallel computation, the problem is broken down into independent parts that can be executed simultaneously, i.e. in parallel.

Traditionally, parallelism has mostly been exploited in supercomputers, clusters, and other high performance systems. In recent years, consumer level computers and even mobile phones, have started utilizing multiple CPUs. The main reason for this is that the clock frequency of a single core can not be increased anymore, which previously has been the main benefactor to performance. In addition, the power consumption is relative to the square of the frequency. This means that multiple smaller frequency cores with the same overall computational power as a single larger frequency core, will consume less power for the same task.

In this chapter we will first introduce different forms of parallelism. Next we will introduce Amdahl's law which gives a theoretical limit to how well a system may be speed up using parallelization. We also discuss how this relates to scaling of systems. Granularity, or how decomposed the parallel task is, is discussed next. The topic of concurrency is introduced next, following with mutual exclusion and synchronization. After this, cache cohorence and consistency in multicore systems is discussed. Next Flynn's taxonomy is introduced which classifies parallel systems based on their data and control flows. Finally, different parallel programming models are discussed. A good introduction to parallel computing can be found in [26], which is used extensively as a reference in this chapter.

4.1 Levels of parallelism

Parallelism can be divided into several different forms:

Bit-level parallelism

Bit-level parallelism is based on increasing the processor word size. Increasing the processor word size reduces the number of instructions required to perform operations on operands that are larger than the word size. Consider for example, addition of two 64-bit integers on a processor with 32-bit word size (two add-instructions plus handling the carry bit) vs. 64-bit word size (one add-instruction). The first computer to support bit-level parallelism was Whirlwind I, developed by MIT in 1951.

Instruction-level parallelism

Instruction-level parallelism means the simultaneous execution of instructions in a processor. Instruction-level parallelism can be achieved many ways. In a pipelined processor, the datapath is divided into multiple stages, each capable of handling a different instruction at a given time. This causes the execution of instructions to overlap. In superscalar processors, the functional units of the datapath are duplicated, therefore allowing simultaneous execution of multiple instructions. How the instructions are scheduled for these processors can be done either by the hardware (dynamic parallelism) or by a compiler (static parallelism).

Data parallelism

In data parallelism, the same task can be done for individual data elements without dependence from each other. Matrix addition is a simple example of data parallelism, since the same operation is done for each corresponding elements and the result is not dependent on the order of these operations.

Task parallelism

In task parallelism, the initial task is divided into smaller tasks which can be executed in parallel. These tasks can be run on same or different sets of data. Some kind of communication mechanism is needed to ensure the synchronized execution of the tasks.

4.2 Amdahl's law

Amdahl's law predicts the maximum improvement to the overall performance of a system, when a part of the system is improved [27]. In parallel programming, this is often used to predict how much performance can be increased by adding more hardware threads, when the percentage of parallel section is known. Mathematically, Amdahl's Law can be expressed as

$$S(N) = \frac{1}{B + \frac{1}{N}(1 - B)}, \tag{4.1}$$

where S is total speedup, N is the number of threads, and B is the fraction of the task that is serial. The theoretical maximum speedup can be calculated by setting

$N \to \infty$ in Eq. (4.1), which gives us

$$S(N \to \infty) = \frac{1}{B}. \tag{4.2}$$

What this result indicates, is that if a program contains a serial section, there is a theoretical limit for the maximum speedup that cannot be exceeded by adding more hardware threads. This is rather intuitive, since we cannot speed up the serial section by adding more computational units. Its execution will always take the same amount of time regardless the amount of threads executing the parallel section.

4.3 Scalability

In practice, programs often do not scale as Amdahl's Law predicts. Control mechanisms add overhead to parallel execution and the underlying hardware, e.g. too small cache size, may reduce the benefits gained by adding more hardware threads. Scalability of a parallel system is the measure of how well the speedup is proportional to the number of processors used [28, 26]. In a perfectly scalable system, the speedup depends linearly on the number of processors.

From Amdahl's Law, it follows that the scalability of any program with a serial section will at some point start to decrease. Modifying Amdahl's Law so that the fraction of serial section is not constant, but decreases in respect to input size, we can describe a model that scales infinitely as long as we grow the input size. [28]

4.4 Granularity

The quantity which measures the size and number of the serial tasks the parallel problem is decomposed into is called granularity [28, 26]. A decomposition into a large number of small tasks is called fine-grained, whereas a decomposition into few large tasks is called coarse-grained.

The problem should be decomposed so that there exists enough tasks to keep all processors busy at all times. The more fine-grained the decomposition is, the more evenly the processing is divided between different processors. However, when the number of tasks increases, so increases the parallelism overhead. The designers of parallel systems need to find the optimal granularity in order to gain maximum performance. [28]

4.5 Concurrency

Concurrency is the apparent simultaneous execution of computational tasks. In other words, the tasks can be split into smaller chunks that can be interleaved without affecting the outcome of the computation.

In a single-processor system, the apparent simultaneous execution of tasks can be achieved by splitting the execution into small time slices. These slices are scheduled in and out of the processor by the operating system. The switching of tasks is called a context switch. The lengths of the time slices are smaller than what can be perceived by humans. Therefore, for the user it seems that the tasks are running simultaneously. In this way, it is possible to create interactive user interfaces which appear to execute multiple programs simultaneously using single-processor systems.

In a multi-processor system, in addition to interleaving, the execution of the chunks can also overlap. Parallel system also display concurrency, but the computation can be truly simultaneous. [29]

4.6 Mutual exclusion

Consider a resource which can only be accessed by one process at a time and two processes both of which will want an access to the aforementioned resource. If the access to the resource, also known as critical section, is not controlled in any way, the output of the processes will depend on the interleaved execution order of the instructions which accessed the critical section. This is called a race condition.

The prevention of uncontrolled access to the critical section is called mutual exclusion. Mutual exclusion can be implemented using either hardware or software solutions. However, on the most fundamental level, mutual exclusion must always rely on some primitive mechanism provided by the hardware. [29]

4.7 Synchronization

When doing parallel computation, the executions of different threads usually need to communicate with each other. For example, the execution of a certain part of one thread might be dependant on the result of a certain part of some other thread. Even if all the threads are totally independent, they might need to signal the master thread when they have finished. This communication and cooperation is called synchronization. [29]

Practically all basic hardware synchronization primitives are built using a mechanism capable of reading and modifying a memory location. Usually user programs use an higher level synchronization primitives, such as semaphores, mutexes, locks, monitors, or message passing, built on top of the low level primitives. What is common for all synchronization primitives is that threads can utilize them to wait on and/or communicate with each other. [30]

4.8 Cache coherence and consistency

Multiple processor cores in the same chip most likely share a common physical address space. Therefore, copying memory data into the processor caches introduces new problems as the modifications of a memory location must be directed to all caches holding copies of that location.

In simple terms, cache coherency means that a read to any cache in the system will give the most recent value for a memory location. In other words, if data from some memory location exists in one or more caches, it must be updated to all these caches when a write occurs into that memory location.

When multiple writes are made to a memory location, these writes should occur in the same order in all caches. If this is not the case, some caches will be left with incoherent data. When the writes to a memory location appear in same order in all caches, the caches are said to be consistent. [30]

4.9 Classification of parallel systems

Many different parallel systems have been introduced during the last fifty years or so. In general, a parallel computer is a system consisting of multiple processing elements, that cooperatively solve large problems. The category of parallel computers is quite large: the number and complexity of the processing elements, the structure of the interconnect network between these elements, and how the work is coordinated between the elements are characteristics that greatly affect the performance and application area of the parallel system. [28]

In order to effectively compare parallel systems, classifications must be made. One such classification is given by Flynn's taxonomy, which divides parallel systems into categories based on data and control flows. Flynn's taxonomy distinguishes four categories which are: [28, 31]

Single Instruction, Single Data (SISD)
A SISD architecture consists of one processing element. At each step, an instruction and the corresponding data is loaded and executed. A conventional sequential computer falls under this category.

Multiple Instruction, Single Data (MISD)
In MISD architecture, multiple processing elements are executing different

instructions on the same data. This model is very restrictive and a commercial MISD systems has never been built [28].

Single Instruction, Multiple Data (SIMD)
A SIMD architecture consists of multiple processing elements each executing the same instructions, but with differing data. A good example of a SIMD system is a modern graphic processing unit (GPU).

Multiple Instruction, Multiple Data (MIMD)
In MIMD architecture, multiple processing elements are executing differing instructions on differing data. Multicore processors and cluster systems are MIMD systems.

Parallel systems can also be classified as manycore or multicore systems. The loose distinction between many- and multicore is the number of cores on the same chip: in multicore the number of cores is from two up to several tens, whereas in manycore the number of cores is upwards from several tens, likely hundreds and even up to thousands [32].

As the number of cores in a single chip grows, the cost of cache coherency between the private caches of the cores increases. As a result, it can be a good idea to abandon cache consistency across all cores in favor of better performance. One definition of many- and multicore is based on cache coherency: if the number of processing units is so large that the cache coherence is omitted, the chip is manycore, otherwise it is a multicore.

4.10 Programming models

Parallel programming models are an abstraction of the underlying hardware and memory architectures. Any parallel model can, at least in theory, be implemented using any kind of hardware or another programming model. It therefore allows the programmer to use the programming model without the need to understand the details of the underlying implementation.

4.10.1 Shared Memory Models

As the name implies, in shared memory model, task share a common address space (or a portion of it) that each task can access and modify asynchronously. As any portion of data can be freely read and modified by any task, some kind of synchronization and locking mechanism is often used to access shared memory.

Shared memory is an efficient way to pass data between tasks, since no copying of data is required. This saves time and memory as no redundant copies of the same data are needed.

Typically in operating systems, each process has its own address space, which is shared by the threads of the process. Creation of shared memory portion between two or more processes is done by using mechanisms, i.e. system calls, provided by the operating system.

4.10.2 Message Passing Model

In message passing models, the tasks exchange messages with each other. These messages are used to send data as well as synchronize between tasks. As opposed to shared memory model, in message passing model each task operates in its local private memory.

Usually in message passing model, each resource is owned by some task. Direct access to the resource can be done only by this task. Locking can be achieved so that other tasks request operations on the resource, which the owner then processes atomically. The most notable message passing system is the Message Passing Interface (MPI).

4.10.3 Implicit parallelism model

Implicit parallelism tries to hide the parallelism from the programmer. The compiler or interpreter implicitly tries to use parallelism without any directives or functions that control parallel execution.

As the programmer does not need to worry about the implementation of the parallelism, he or she can instead focus more on solving the actual problem. However, this reduces the amount of control the programmer has and can often reduce the efficiency of the program compared to explicit parallelism.

Chapter 5

H.265 Video Coding Standard

H.265 (also known as High Efficiency Video Coding, HEVC) is the most recent video coding standard released in collaboration by the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG), partnership known as the Joint Collaborative Team on Video Coding (JCT-VC). H.265 is a successor to the popular and currently widely used H.264/MPEG-4 AVC video coding standard.

The principal design ideas behind H.265 have been a better support for larger-than-HD quality resolutions (4K and 8K resolutions) and better parallelizability of the used algorithms. As with previous ITU-T and ISO/IEC video coding standards, only the structure of the encoded bitstream is standardized. Each decoder conforming to the standard should produce identical output for the same bitstream. This gives a maximal freedom for encoder and decoder designers to optimize their implementations.

In this chapter we present the H.265 video coding standard. First we will consider the high-level syntax and introduce the Network Abstraction Layer which wraps the encoded video for transmission and storage. Then we will consider the Video Coding Layer which contains the actual video data. In this section we will look in more detail how H.265 implements the video compression concepts introduced in chapter 2, such as sampling, inter and intra picture prediction, quantization, and entropy decoding. Also the novel block structure of H.265 is described. Next we will introduce the parallel features of H.265 which are Tiles, Wavefront Parallel Processing, and independent slice segments. Finally we will finish the chapter by discussing the performance improvements of H.265 compared to H.264 and take a look at related work done by others.

The information presented in this chapter is mainly based on [33, 9].

5.1 High-level syntax

A great deal of features from H.264 have been inherited to H.265. The improvements of H.265 are largely based on small improvements on these pre-existing features.

The encoded video data is stored in Video Coding Layer (VCL) units, also known as slices. Network Abstraction Layer (NAL) is used to wrap the slices into a bitstream suitable for transmission and storage and provides an error resilient means for transportation.

5.2 Network Abstraction Layer

Network Abstraction Layer (NAL) is an abstraction that wraps the encoded video for transportation and storage. NAL consists of segments called NAL units (NALU), which can contain either VCL or non-VCL data. A set of NALUs (both VCL and non-VCL) forming a frame is called an access unit (AU).

The NALUs consist of a two byte header and the payload, also known as raw byte sequence payload (RBSP). The header contains the type of the NALU and the temporal layer ID.

The VCL NALU contains the coded video data of one slice segment. The NALU types for VCL specify if the NALU can be random accessed or if the NALU is referenced from other pictures. The non-VCL NALUs can contain video (VPS), sequence (SPS), or picture parameters sets (PPS), which containg profile/level information, or supplemental enhancement information (SEI), which is not required by the decoding process, but may provide otherwise useful information for the decoder.

5.3 Video Coding Layer

The video coding layer in H.265 has the same hybrid approach (combining both intra-/interpicture prediction and transforms) that has been used in all video coding standards since H.261.

As today nearly all video material and displays use progressive scan, H.265 has no explicit coding features to support compression of interlaced video. However, H.265 provides a metadata syntax to indicate that the coded bitstream originates from an interlaced video and each field is encoded as a separate picture.

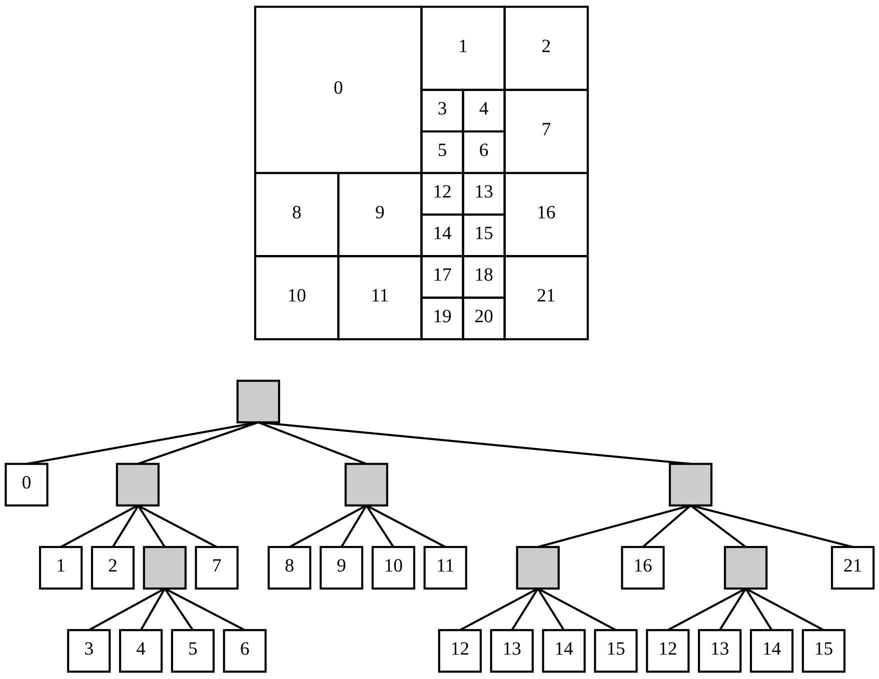

Figure 5.1: An example structure of a coding tree with the scan order shown by numbers.

5.3.1 Sampling

To represent video signals, H.265 typically uses YCbCr color space with YCbCr sampling, although other sampling formats are possible. In this sampling mode, the luma (Y) component has four times as many samples (two luma samples per one chroma sample in both horizontal and vertical directions) as both chroma components (Cb and Cr). For a progressive scan video with a picture size $W \times H$, where W is the width and H is the height of the picture in samples, the size of the luma component is $W \times H$ and the size of each chroma component is $W/2 \times H/2$. Each sample is usually represented using 8 or 10 bits, 8 bits being the more typical option.

5.3.2 Coding tree structure

In H.265, each frame is divided into Coding Tree Units (CTU), each consisting of a luma Coding Tree Block (CTB) and two chroma CTBs, which contain the sample data, and the syntax elements associated with the CTU. The CTU is the basic processing unit in the standard. The size of each luma CTB is equal and can be chosen to be 16×16, 32×32, or 64×64 samples (i.e. pixels). The sizes of the chroma CTBs are proportional to the corresponding luma CTB. When using 4:2:0 sampling mode, for a CTU with a luma CTB of size $L \times L$, the size of the corresponding chroma CTBs is $L/2 \times L/2$.

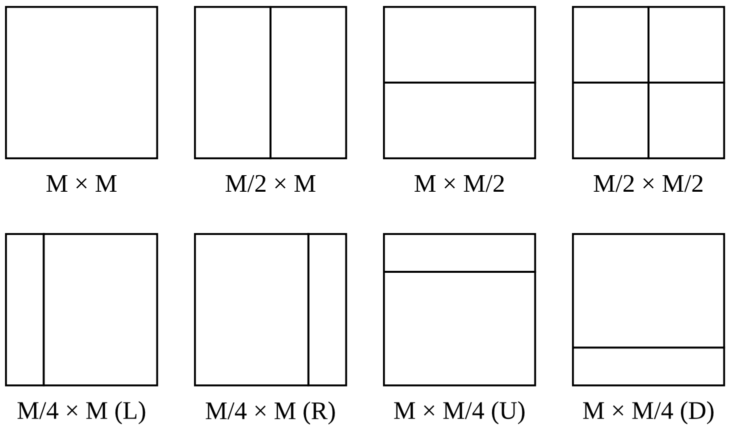

Figure 5.2: Splitting modes for splitting CBs into PBs.

Compared to the previous H.264 coding standard which had block size of 16×16 pixels, the larger block sizes are designed to give better compression rates when compressing videos that have large resolutions (Full HD and beyond). Larger block sizes require more computation when being encoded, but in contrast they reduce the time required to decode the bitstream.

Each CTU is divided into a quadtree structure consisting of Coding Units (CU) and correspondingly each CTB is divided into Coding Blocks (CB). The root of the coding tree is the CTU, which can contain one CU with size equal to the CTU or be split to four equal sized CUs. An example of a coding tree structure is shown in Fig. 5.1. Each CU contains the associated luma and chroma CBs and the syntax elements.

5.3.3 Prediction units and blocks

The type of prediction (either intra- or interprediction) is decided at the CU level and is stored into Prediction Units (PU) and Prediction Blocks (PB). The last level CB is always associated with a PB of similar size or splitted into one extra level of multiple PBs.

When using intraprediction mode, the CB is always associated with one similar size PB except for CBs having the smallest allowed CB size. In the latter case, the PU can be split into four smaller PUs.

When the prediction mode is signaled as inter, the CB can be split into either one, two, or four PBs. The splitting to four is similar to the case in intraprediction mode and allowed only for CBs having the minimum allowed size. When the CB is split into two PBs, six different types of splitting are possible.

The six different splitting types are shown in Fig. 5.2. The upper figures view the the cases of not splitting, splitting the CB into two equal sizes PBs, and splitting into four equal sized PBs. The partition types on the lower row are called Asymmetric Motion Partitioning (AMP), and are allowed only for CBs having luma block size of 16 or larger.

5.3.4 Transmission units and blocks

The prediction residuals which contain the actual data, are stored in Transmission Units (TU). The TUs contain the Transmission Blocks (TB) which hold the residuals for luma and chroma channels. The TU tree structure has its root at the CU level and each CU may contain one TU or be splitted down to 4×4 sized TUs. The TBs contain the prediction residuals transformed using transforms similar to discrete cosine transform (DCT) or discrete sine transform (DST). It is possible for TBs to span over PB boundaries.

To better understand how TBs and PBs work, let's go through how they function in the encoding process. In the beginning we have the raw video frame sample data. First, we create the prediction for the frame, which gives us the PBs and the prediction residual data. Next, this residual data is divided to TBs which will be subject to transforms and entropy compression. In other words, PBs contain the information how the prediction is made where as TBs contain the video (residual) data.

5.3.5 Intrapicture prediction

Intrapicture prediction operates using the previously decoded sample data from spatially neighboring samples to create the prediction. A CB predicted using intrapicture prediction can either be mapped as a PB directly or be split into four equal sized PBs. This splitting is only allowed if the size of the CB equals the minimum allowed CB size.

Intrapicture prediction in H.265 uses different modes of prediction based on which mode gives the best prediction result. The modes are as follows:

Intra angular prediction

Intra angular prediction works similarly than in H.264, except that the number of possible directions has been increased from 8 to 33. In angular prediction, the already decoded neighboring samples in the given direction are used as a reference sample. Each TB is predicted from the spatially neighboring samples that have been decoded, but not yet filtered by in-loop filters.

The actual reference sample is calculated from the two nearest samples using interpolation with $\frac{1}{32}$ sample accuracy. The density of directions is larger when near horizontal and vertical directions. The prediction method is the same for all block sizes, unlike in H.264 which uses different methods for each block size.

To better understand how intra angular prediction functions, let us first consider the vertical and horizontal directions. In vertical mode, all the samples in a given column in the current PB get the same value, which is equal to the value of the bottom sample in the same column of the upper block. Similary for horizontal mode, the samples get the value of the rightmost sample in the corresponding row in the block to the left. The modes with directions between these two directions are basically linear combinations of these modes.

Intra DC prediction

Intra DC prediction is a pretty coarse prediction method; all samples in the block are given the average value of all the surrounding decoded reference samples. This is a simple choice for large picture areas with the same color and little tone changes.

Intra planar prediction

Intra planar prediction uses four corner reference samples to calculate the predicted sample values. Two linear predictions are created using these four samples and their average is used to give values to the predicted samples. Planar prediction is intended to reduce discontinuities along the block boundaries. In H.265, planar prediction is allowed for all block sizes, whereas in H.264 it is allowed only for PBs of size 16×16.

In H.265, *reference sample smoothing* is sometimes used to smoothen the reference samples before prediction. This smoothing is applied by the encoder based on, e.g. directionality, level of discontinuity and block size. The reason behind smoothing is that since we are predicting, we do not need the details in the reference samples. When we reduce the energy in the reference samples, we get statistically less energy in the residual, and therefore achieve better compression ratio. H.265 uses three-tap filter similar to H.264 for reference sample smoothing.

To reduce the discontinuities between TBs, H.265 uses the so called *boundary value smoothing*. It is used for intra DC prediction mode and two intar angular modes (horizontal and vertical), when the block size is smaller than 32×32. In DC mode, the first row and column of samples are replaced by two-tap filtered samples of their own value and the adjacent reference sample value. In intra angular modes, half the difference of the top-left reference sample and the current adjacent reference sample is added to the first row and column of the TB.

For slice and tile boundaries, the neighboring reference samples are not available. This is also the case for constrained intra prediction, where any interpicture predicted samples in the current frame are not available for intrapicture prediction. This prevents the potentially erroneous previously decoded data from propagating to the following frames. In these cases the non-available reference samples are substituted with the closest available reference sample values.

In H.265 we have a total of 35 modes (33 angular modes, DC mode and planar mode) for intraprediction. The decoder tries to predict the used prediction mode using the prediction modes used for the neighboring blocks. H.265 considers three most probable modes (MPMs), instead of the one MPM in H.264. If the prediction uses one of the three predicted modes, then only the index of the MPM is coded into the bitstream. Otherwise the prediction mode index is coded into the bitstream.

5.3.6 Interpicture prediction

Interpicture prediction uses the data from temporally neighboring samples to create the prediction. Interpicture prediction offers more partition types than intrapicture prediction. When using intrapicture prediction, the CB can be mapped as PB, or be split to two (horizontally or vertically) or four equal sized PBs. Split to four PBs is allowed only when the CB size is equal to the minimum CB size. In addition to these symmetric splits, interpicture prediction offers a way to split the CB into two different size PBs. The split can be done horizontally or vertically so that the larger PB has three times the amount of samples compared the smaller PB. This type of split is know as asymmetric motion partition (ASP).

Interpicture prediction can use temporally preceding or following pictures that already have been decoded, as a base for prediction. The picture is identified by the reference picture index that is signaled to the decoder. Motion vector tells the decoder what area to use as a reference in the reference picture. The motion vector components are not necessarily integers. In these cases, the reference samples are calculated using interpolation. This is known as *fractional sample interpolation.* For luma samples, the accuracy for the motion vector components is a quarter sample. Eight-tap filter is used to calculate half samples and seven-tap filter is used to calculate quarter samples from neighboring samples. For chroma samples, a similar method is used with slightly different details.

Motion of blocks in a picture is often strongly correlated. It is therefore sensible to derive the motion vectors from the neighboring blocks. The motion vectors for motion compensation can be derived in two ways. The motions vectors can be predicted using temporally or spatially neighboring PBs. In this case, the reference PB and the vector residual are signaled in the bitstream. This is known as *advanced motion vector prediction* (AMVP). The motion vectors can also be inherited from temporally or spatially neighboring PBs. This is known as *merge mode*, since a merged region sharing all the data is formed.

5.3.7 Transform and quantization

H.265 uses a transform similar to discrete cosine transform (DCT) to transform the prediction residuals into a different basis. For the transform of luma interpicture prediction residuals of size 4×4, a transform similar to discrete sine transform is alternatively specified.

As in H.264, *uniform reconstruction quantization* (URQ) is used for quantization. This specifies a logarithmic quantization parameter signaled in the bitstream. Also quantization matrices can be used.

5.3.8 Entropy coding

The only possible entropy encoding method in H.265 is the *context-adaptive binary arithmetic coding* (CABAC) already supported in H.264. The core algorithm in H.265 is the same as in H.264 but with a few optimizations. H.264 also supported *context-adaptive variable-length coding* (CAVLC) entropy coding scheme, but this was dropped to reduce complexity [9]. A more detailed description of these encoding schemes can be found in [34].

5.3.9 In-loop filters

Two filters are applied to the reconstructed frames before they are inserted into the decoded picture buffer. First is the *deblocking filter* (DBF). The purpose of DBF is to smoothen the block boundaries and reduce compression artifacts than can occur when using block-based prediction. DBF is applied to samples in the block boundaries. It was already included in H.264 and is mostly similar in H.265.

H.265 introduces a new in-loop filter called *sample adaptive offset* (SAO). In contrast to DBF, SAO is applied to all samples. In SAO, offsets are encoded into the bitstream. These offsets are applied to the reconstructed samples by the decoder, reducing distortion in the reconstructed picture. [35]

5.4 Parallel Features in H.265

One of the main purposes of H.265 is to improve the parallel processing capabilities of video encoding and decoding. Next we will introduce the new parallel features added into H.265 for improved parallelism.

5.4.1 Tiles

Tiles is the partitioning of a frame into independent rectangular areas called tiles. Each tile can be processed independently without any information from other tiles in the same frame. This provides coarse-grained parallelism that can be exploited in encoding and decoding processes as no synchronization between tiles in the same frame is required. The main purpose of tiles is to increase the parallel computing capabilities. Tiles can also be used to combine video from multiple sources without having to decode and re-encode the streams, a property that is extremely useful in videoconferencing, for example [25]. In H.265, Tiles are supported for levels 3 and higher. [36]

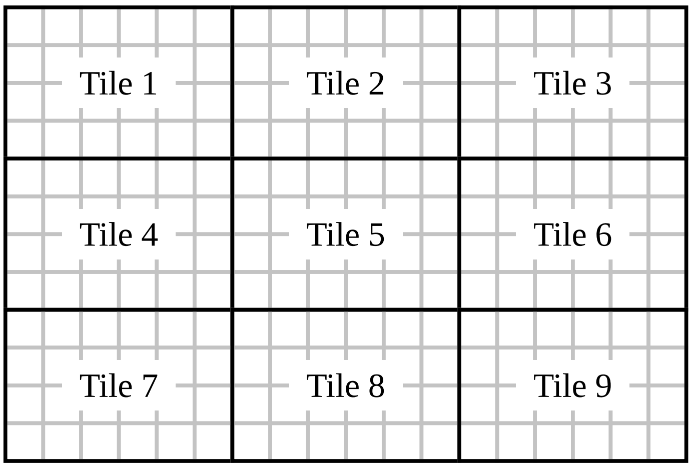

Figure 5.3: The division of a frame with 18 × 12 CTUs into 9 even-sized tiles. The black lines mark the boundaries of the tiles whereas the grey lines mark the boundaries of the CTUs.

The number and shape of the tiles is coded into the bitstream by the encoder. Therefore, the maximum level of parallelism available for the decoder is decided during the encoding process. The tile spacing can be signaled explicitly or a uniform spacing can be used. The tile boundaries can not cross the CTU boundaries. The CTUs in a tile a read in raster scan order. A division of a frame into tiles is presented in Fig. 5.3.

5.4.2 Wavefront Parallel Processing

In Wavefront Parallel Processing (WPP), the CTUs of a frame are encoded so that the encoding of a CTU can start when

- the CTU above and to the right has been decoded (or does not exist)
- and the CTU to the left has been decoded (or does not exist).

These kind of restrictions will cause the decoding of a frame to progress as a diagonal front, as presented in Fig. 5.4. The maximum level of parallelism when using WPP is determined by the number of CTU rows in the frame.

5.4.3 Independent Slice Segments

Slices are sets of CTUs in frame raster scan order. Each frame is constructed of one or more slices. They are the units in which the VCL data is included in the bitstream.

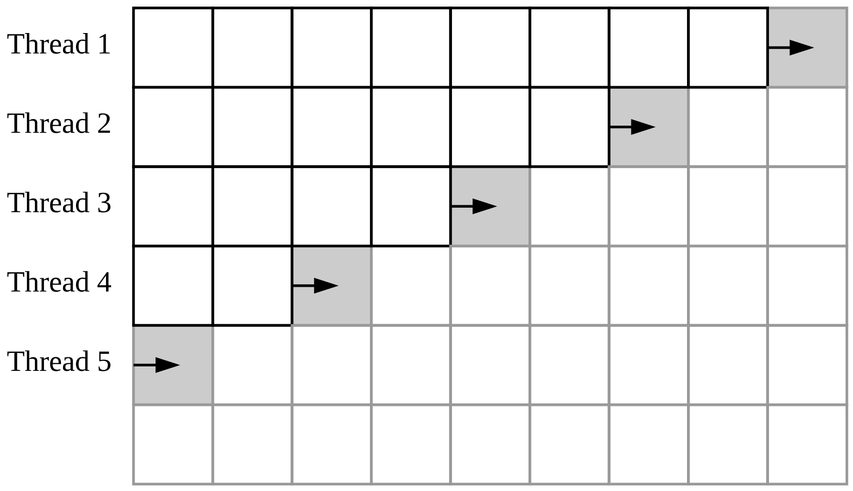

Figure 5.4: The progression of a wavefront in a frame with 9×6 CTUs. Squares with black borders are decoded CTUs, squares with grey background are CTUs whose decoding can start, and squares with grey borders are CTUs whose decoding can not yet start.

Slices are already present in H.264 and were originally included for error resilience. However, using independent slice segments, which are essentially slices that can be decoded without outside dependencies, it is possible to achieve parallelism using slices. In this thesis however, we will not be considering slice parallelism in more depth.

5.5 Performance

The improvements in compression efficiency compared to H.264 are not due to any single new aspect on H.265, but rather a lot of small improvements on components that aggregate to give significant improvements. Especially on larger resolutions, a larger LCU size and coding tree structure give smaller bitrates for the same quality. [37]

The two new parallelization schemes, Tiles and WPP, are intended to improve the picture decode time by utilizing multiple threads more efficiently. These will be our main focus in this thesis.

Traditionally, general purpose CPUs have not been efficient enough to decode, and especially not to encode, high quality video in real-time. Therefore, dedicated hardware has been needed in good quality video solutions. Previous video compression standards have included a lot of serial algorithms in them, and it has been a challenge to implement these in parallel circuits [38]. With modern consumer devices (laptops and smartphones), good quality video can be decoded using general purpose CPUs [39].

5.5.1 Encodind and decoding as computational problems

As computational problems, video encoding and decoding are quite different in terms of complexity. Decoding is a straighforward operation; the output of all standard compliant decoders will be identical for the same bitstream. Encoding however, is a complex optimization and decision making problem and different encoder implementations can resolve it with different solutions [40]. Depending on the tools available, encoders compliant to the same standard can produce a range of varying bitstreams with different bitrates for the same input. For example, in real-time communication the encoder does not have a lot of time to compress the video. Compare this to a scenario where the video is ready days or weeks before the distribution begins. In the latter case, much more time is available for the encoder to optimize the bitrate.

Most common use case for decoders is when the video is played back. In this case, the decoding needs to happen in real-time, otherwise the viewing experience will not be satisfactory. In some cases, such as transcoding videos to other formats before distribution, this restriction does not need to apply. Encoding needs to happen in real-time in video communication and live broadcasts, but if the video is not distributed in real-time, the time restrictions for encoding are not so crucial.

5.6 Related work

In this section we will look at some other work related to parallel scalability of the H.265. A good overview of HEVC parallelization strategies is provided in [41]. Also a novel approach called overlapped wavefront (OWF) is introduced. OWF is similar to WPP, except that in OWF the decoding of the next frame can be started while the final CTU rows of the previous frame are still being decoded. This sets some limitations to the lengths of the motion vectors used in inter prediction.

Scalable High Efficiency Video Coding (SHVC), an extension to H.265, was proposed in [42]. In SHVC, the parallel WPP solution is extended with frame-based parallelism. A parallelization method based on entropy slices was proposed in [43]. This results in a wavefront like decoding pattern similar to WPP.

A theoretical model for the scalability of WPP is derived in [44]. This model is tested with measurements done using TILE-Gx36 manycore processor using up to 36 cores. Since the maximum parallelism level of WPP is determined by the amount of CTU rows, larger resolution videos naturally can reach better scalabilty. In the measurements, WPP was also combined with slices to achieve better scalability.

Chapter 6

Parallel implementation of H.265

In this chapter we describe and implement a parallel implementation of a H.265 video decoder. First we describe a synchronized data structure, called a job queue, which can be used to implement parallel Tiles and Wavefront Parallel Processing (WPP) capabilities into the decoder. We extend the HEVC reference decoder implementation to support Tiles and WPP by using the POSIX Threads library as a provider of parallelism. In the last sections, we will describe how the job queue is used to implement the parallel features.

6.1 Job Queue

To implement the parallelism into the reference decoder implementation [45], we created a synchronization mechanism that we call a *job queue*. The job queue is a versatile structure which could be used to create parallel implementations for many other types of problems as well.

6.1.1 Concept

The idea behind the job queue is to divide the computational task into tasks that can be computed concurrently, called *jobs*. Each job is handled sequentially by a single *worker thread*. In our implementation, the job queue is started by the *main thread*, i.e. the thread that runs the sequential program. The main thread divides the computational task into jobs, creates and starts the worker threads, and finally starts waiting for the completion of all the worker threads.

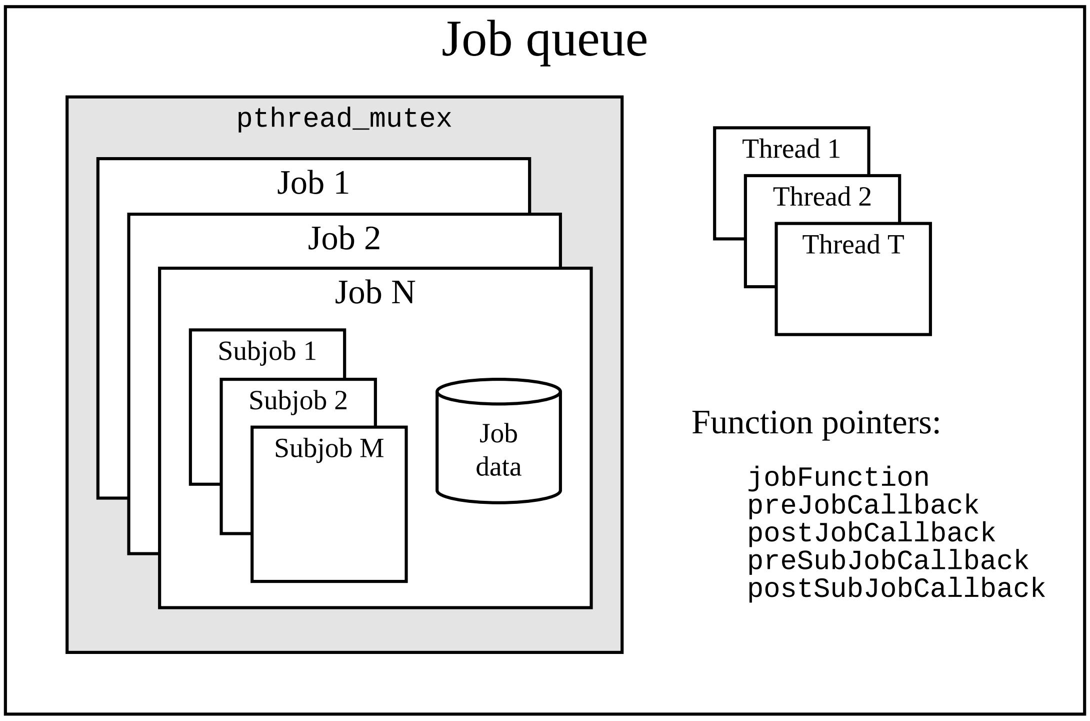

Figure 6.1: The structure of the job queue.

6.1.2 Structure

The structure of the job queue is presented in Fig. 6.1. The jobs are inserted into a FIFO queue and access to which is synchronized by a `pthread` mutex. Only one thread is allowed to push or pop data at any given time.

The function handling the jobs is also contained in the job queue data structure, as are all the callback function. This means that the same functions are called for each job. This is enough for our purposes, but could be easily be modified to support different callback functions for each job.

In order to allow fine-grained parallelism and synchronization inside the jobs, the jobs are divided into multiple *subjobs*. Subjobs are stored into a FIFO queue in the job data structure. The thread processing the job handles the subjobs in a sequential order. This is an important thing to notice, since by definition, the processing order of the subjobs does matter and they are usually dependent on each others results.

In addition to the subjob queue, the job structure contains a pointer to the job data. The sub job is not any predefined data structure, but rather a pointer of type `void*` which is given to the job function along with the job and thread data.

The number of worker threads is given at creation time and the threads are created and started when the job queue itself is started. After all the worker threads have been created, the master thread starts to wait for them to finish by joining each of them. It is also possible to assign arbitrary data for each worker thread, which is then given to the job function when it is called.

Each worker thread begins by trying to pop a job from the queue. If there was a job in the queue, it is processed by calling the job function for each of its subjobs. After the job is processed, another job is popped from the queue. This is repeated until the queue is empty, after which the thread dies.

The job queue is considered finished when each worker thread has died and the control is returned to the main thread. While it is in theory possible to add jobs to the job queue while jobs are being processed, this is highly discouraged. There is no guarantee that any of the worker threads are alive by the time the job gets added to the queue. With the current implementation, the job queue is considered finished when the job FIFO is empty. Since in our applications all the jobs are known before the queue is started, this is enough for our purposes.

The job queue could easily be modified to overcome the aforementioned problems by providing a synchronization barrier before the worker threads die. An explicit signaling would then be required for the worker threads to die. If this were to be accompanied with the possibility to signal threads waiting at this barrier when new jobs are inserted to the queue, the this concept would work also when job are generated dynamically by the job queue itself.

In order to provide synchronization between and inside the jobs, callback functions can be assigned for jobs and subjobs. Pre-job callback function is called before the worker thread starts to process any subjobs and post-job callback function is called after all the subjobs have been processed. Respectively, pre- and post-subjob callback functions are called before and after processing each subjob.

6.2 Tiles

The parallel implementation for Tiles is pretty straightforward. As the tiles in a frame contain no dependencies and do not require any synchronization, each tile is intuitively mapped to a job.

The CUs in each tile must be processed in a predifined order (the tile raster scan order), since dependencies exist between the CUs inside a tile. In our implementation subjobs are processed in this manner, so intuitively each coding unit in the tile is mapped to a subjob.

The mapping to jobs and subjobs is presented in Fig. 6.2. From the slice header we can acquire the positions where each tile resides in the bitstream. Substreams are generated beginning at these positions and given to corresponding jobs as data. Subjob data is simply the CU numbers, which are pushed to the subjob queue of each job in correct order. As each tile is independent from each other and processed

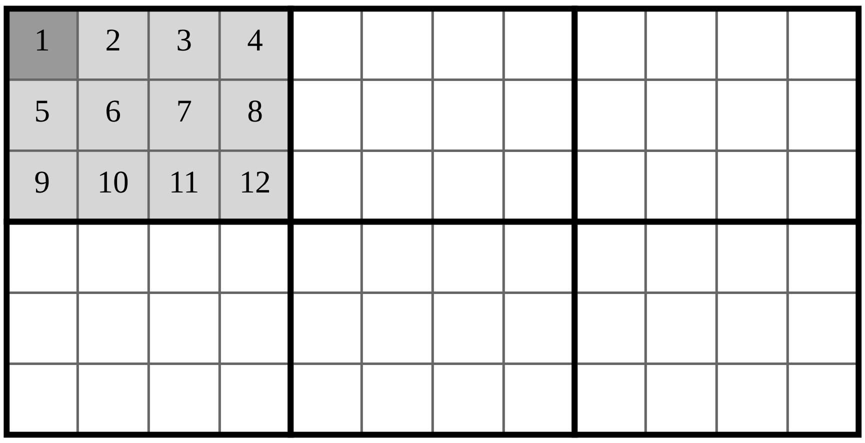

Figure 6.2: Mapping of CTUs in a frame encoded using Tiles to jobs. One job is highlighted (light grey) as well as one subjob (dark grey). Tiles are mapped to jobs and CUs are mapped to subjobs. The order of the subjobs inside a job is also shown.

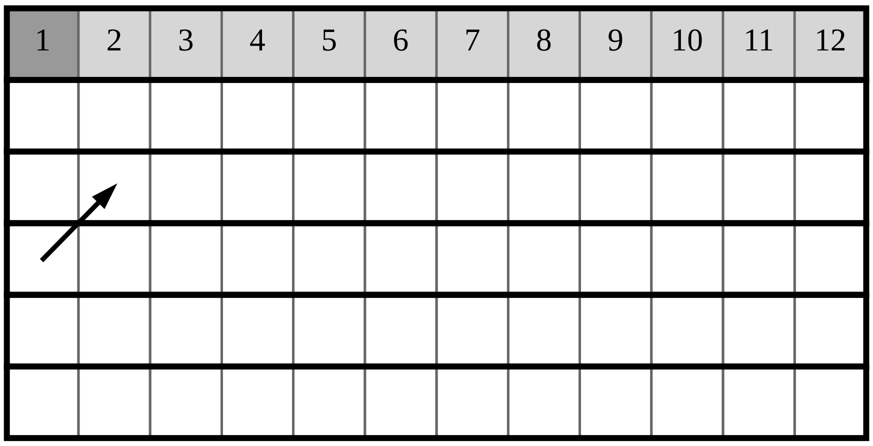

Figure 6.3: Mapping of CTUs in a frame encoded using WPP to jobs. One job is highlighted (light grey) as well as one subjob (dark grey). Tiles are mapped to jobs and CUs are mapped to subjobs. The order of the subjobs inside a job is also shown. The arrow shows the synchronization between jobs. The source CTU of the arrow must wait that the destination CTU has been decoded. The same condition holds for all CTUs in the frame, except for those where the destination CTU of the arrow does not exist.

sequentially by one thread, we can use the same picture data structure (which contains the compressed bitstream, the CU, PU and TU structure, and decompressed data for a single frame of the video) for each job without any synchronization problems.

6.3 Wavefront parallel processing

In WPP, each row of CUs can be decoded in parallel as long as the decoding progress of the row above is at least two CUs ahead. Therefore, rows in the frame are mapped to job queue jobs in our parallel WPP implementation. Similar to the parallel implementation of Tiles, CUs are mapped to subjobs. This mapping is shown in Fig. 6.3.

The above setting however, is not enough. Synchronization between the jobs is needed, since the rows are not independent of each other. To achieve this, we created a structure that stores the decoding progress of each row and handles the synchronization between the rows.

This wavefront structure consists of progress counters for each row, mutexes controlling the access to these counters, and condition variables used to signal the progression of row decoding to other threads. It has two public synchronized functions: the other is used to increment the progress of the current thread and the other to wait for the row above.

Chapter 7

Measurements

In this chapter we will introduce the measurements made to test the scalability of the parallel implementation. First we introduce the three setups used. Following this, the measured quantities are specified. Next the properties of the four videos used as input to the decoder are listed. Finally, we describe how the videos were configured for each setup for Tiles and WPP, since only one parallel feature can be enabled into the encoded video.

7.1 Measurement setups

The performances of the parallel implementations were tested using three different measurement setups. The configurations of the setups are specified in Table 7.1.

7.2 Measuring performance

The performance of the parallel implementation of the HEVC reference decoder was observed by measuring the decoding times of individual frames. The measured times were the total frame decode time, time spent in the parallel section and the time taken by the loop filters. These times were printed into standard output as they were measured. In all cases the frame decoding times were over 20 ms so the overhead of printing directly to standard output is insignificant.

7.3 Measuring hardware events

The amount of hardware events occurring was measured using the `perf stat` tool for Linux. `perf stat` can be used to read the performance counters of a CPU. Performance counters are CPU registers that are used to count the amount of hardware events in the CPU, such as the amount of cycles and instructions executed, cache misses and branch mispredictions. [46, 47]

The output of `perf stat` was directed to standard error. The hardware events measured for each setup are shown in 7.2.

Table 7.1: Measurement setups.

	Setup 1	Setup 2	Setup 3
CPU	Intel® Core™2 Quad Q9550	Intel® Core™ i7-3770	Intel® Core™ i7-970
Clock speed	2.83 GHz	3.4 GHz	3.2 GHz
Number of cores	4	4	6
Number of threads	4	8	12
L1 data cache	32 KB per core, 8-way set associative	32 KB per core	32 KB per core
L1 instruction cache	32 KB per core, 8-way set associative	32 KB per core	32 KB per core
L2 cache	6 MB per 2 cores, 24-way set associative	256 KB per core	256 KB per core
L3 cache	-	8 MB shared	12 MB shared
CPU Mark score [48]	4013 [49]	9317 [50]	8452 [51]
RAM	4 GB DDR2 800 MHz	8 GB	12 GB 1066 MHz
Operating system	Ubuntu 12.10	Ubuntu 12.04	Ubuntu 12.04
Kernel	3.5.0-45-generic	3.5.0-44-generic	3.2.0-32-generic

Table 7.2: Measured hardware events for each setup.

Event	Event name	Setup 1	Setup 2	Setup 3
L1 data cache misses	`L1-dcache-misses`	✓	✓	✓
Data TLB misses	`dTLB-misses`	✓	✓	✓
LLC cache misses	`LLC-misses`	✓	✓	✓
Page faults	`page-faults`			✓
Context switches	`context-switches`			✓
CPU migrations	`cpu-migrations`			✓

7.4 Sample videos

Both Tiles and WPP implementations were measured using four different sample videos. The properties for each sample video are listed in Table 7.3.

In the performance measurements, the average decoding time is measured, so the frame rate does not affect this. The total number of frames might affect the performance, e.g. because of caching.

Table 7.3: Properties of the sample videos. The videos are part of HEVC test sequences [52].

	Video 1	Video 2	Video 3	Video 4
Name	BQMall	ChinaSpeed	ParkScene	SlideEditing
Frame rate	60	30	24	30
Number of Frames	600	500	240	300
Width (pixels)	832	1024	1920	1280
Height (pixels)	480	768	1080	720

7.5 Tiles implementation measurements

When doing the measurements for the parallel implementation of Tiles, the number of tiles was matched with the number of threads in the setup. In other words, for each of the 4 sample videos, 3 different video streams were encoded, totaling 12 encoded streams. The Tiles configuration for each setup can be seen in Table 7.4.

Table 7.4: Tiles configurations for the measurement setups.

Number of	Setup 1	Setup 2	Setup 3
Cores	4	4	6
Threads	4	8	12
Vertical tiles	2	2	3
Horizontal tiles	2	4	4
Total tiles	4	8	12

7.6 WPP implementation measurements

Unlike for Tiles, where the number of Tiles (and therefore the possible level of parallelism) is configured before the encoding process, for WPP the maximum level of parallelism is the number of CTU rows in the encoded video stream. For each sample video, one video stream was encoded, totaling 4 encoded streams.

The videos were encoded using 64 pixels as the height and the width of the LCU. Thus the amount of CTU rows can be calculated using

$$R = \left\lceil \frac{H_{frame}}{H_{LCU}} \right\rceil, \tag{7.1}$$

where R is the number of CTU rows, H_{frame} is the height of the frame in pixels, and H_{LCU} is the height of the LCU in pixels. The amount of CTUs for each encoded sample video can be seen in Table 7.5.

Table 7.5: The amount of CTUs in the encoded sample videos.

	Video 1	Video 2	Video 3	Video 4
CTU columns	13	16	30	20
CTU rows	8	12	17	12
Total CTUs	104	192	510	240

Chapter 8

Results

In this chapter we present the measurement results for our parallelized decoder measured using the setups presented in chapter 7. We start by presenting he measurement results for each setup and video for both Tiles and WPP are presented. Next we look at the performance in general and compare Tiles and WPP measurement results. Following this, the measurement results for Tiles are looked in more details. A theoretical model explaining the measured results is presented. After this, we analyze the WPP results in more detail and present thoughts on why the WPP implementation did not scale as well as the Tiles implementation. Finally, we take a look at how our results compare to similar work done by others and present some ideas on how the measurements could be improved and what could be done in the future.

8.1 Measurement results

The measurement results for Tiles are presented in Figs. 8.1 – 8.3 and the results for WPP in Figs. 8.4 – 8.6. Note that in all cases the number of threads goes from one to the number of CPU cores in the setup, except for Setup 3, Video 1 where the number of threads goes up to 8 (despite Setup 3 having 12 CPUs). This is due to the fact that Video 1 only has 8 CTU rows and the maximum possible level of parallelism for WPP is the number of CTU rows in the frame.

8.2 Performance

For each setup, both Tiles and WPP implementation measurements are in the same magnitude. In most cases, Tiles implementation is somewhat faster, at least for larger thread counts.

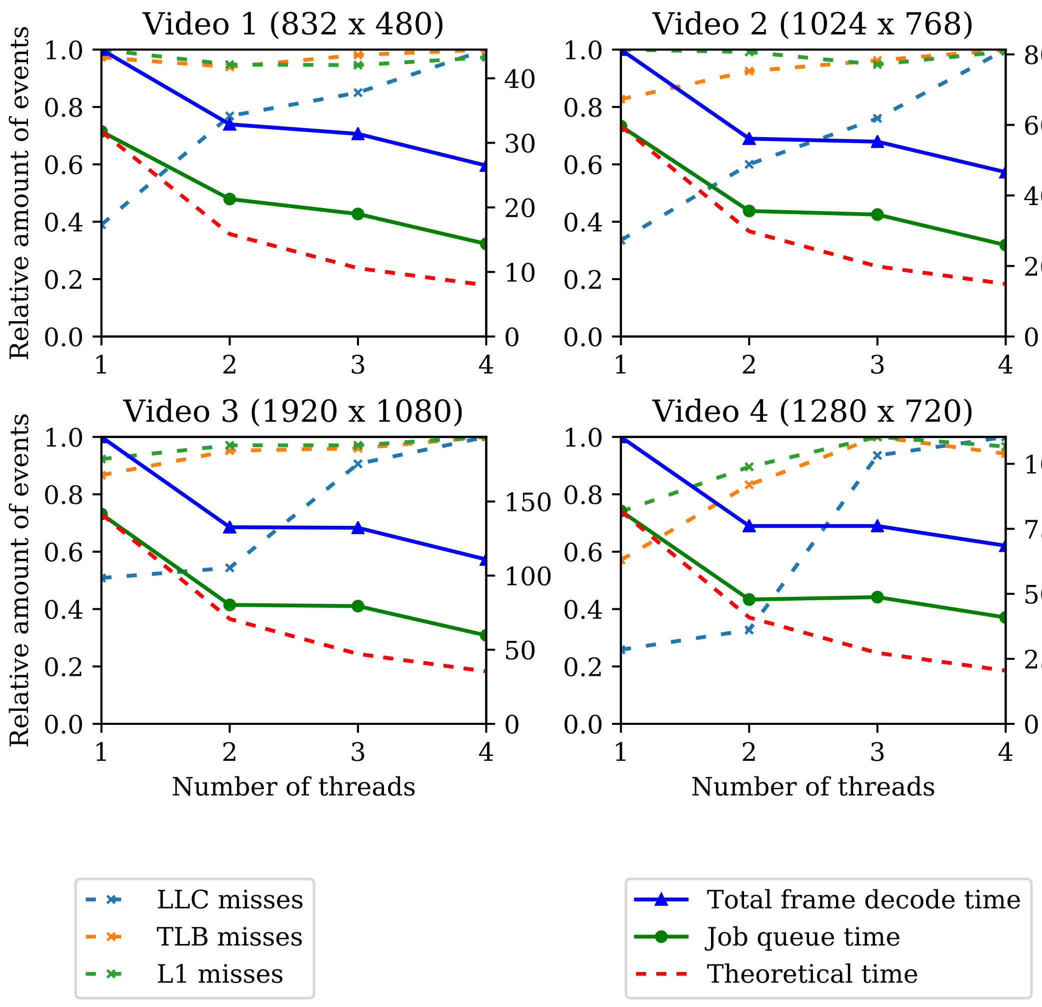

Figure 8.1: Tiles measurements for setup 1.

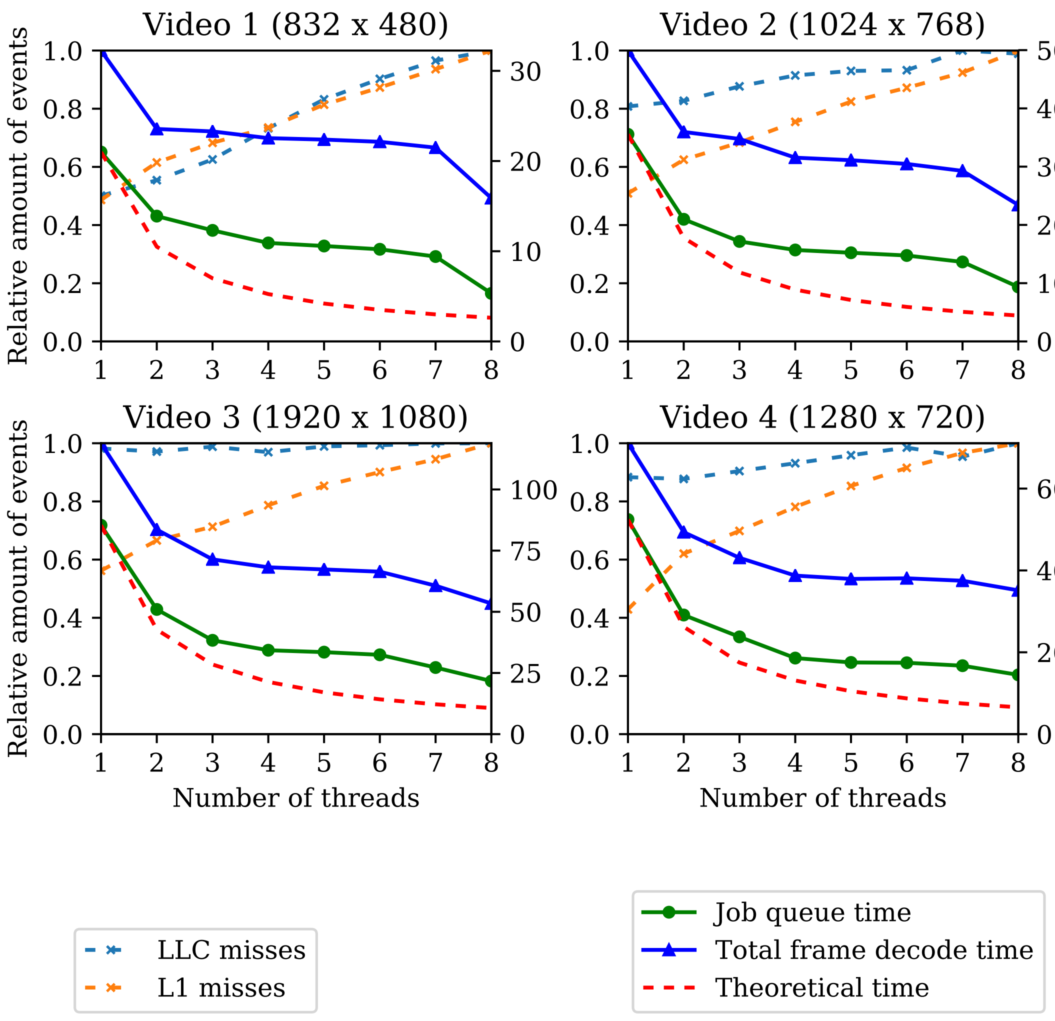

Figure 8.2: Tiles measurements for setup 2.

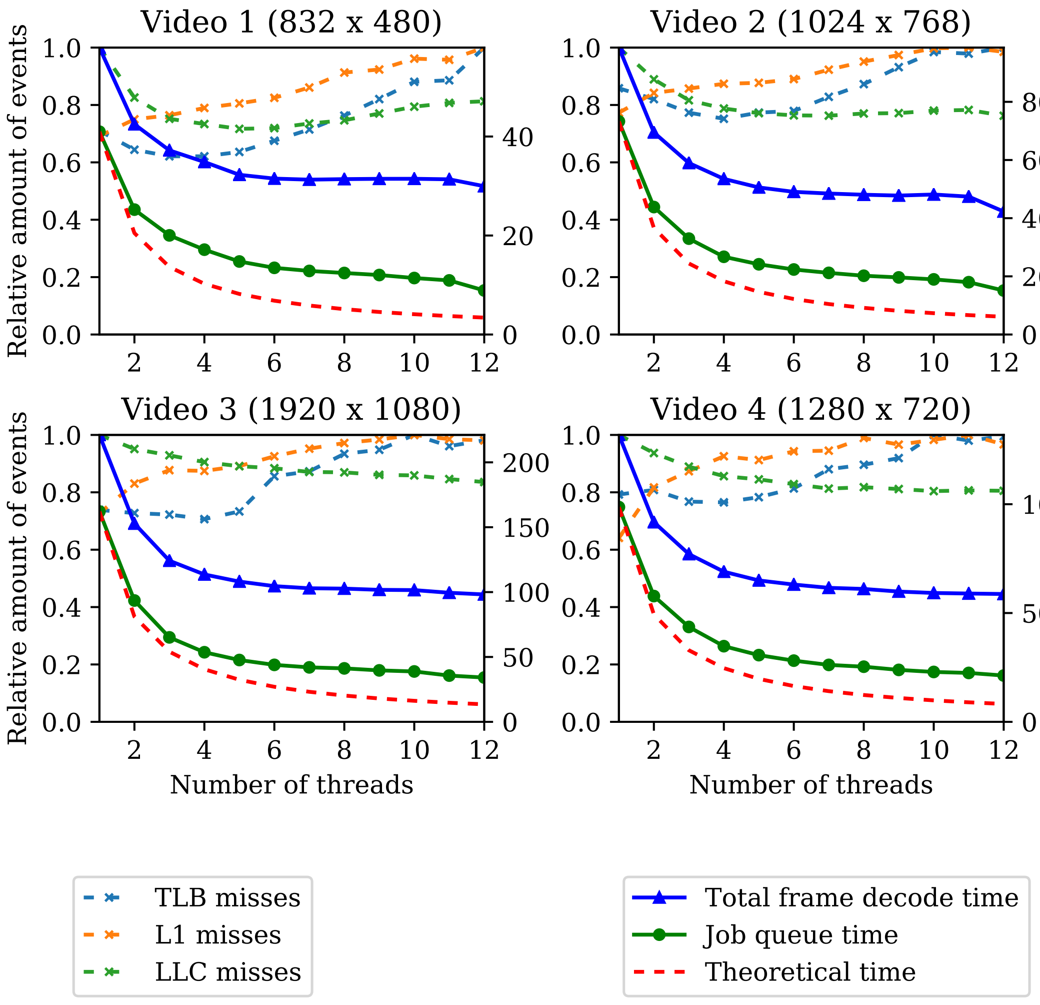

Figure 8.3: Tiles measurements for setup 3.

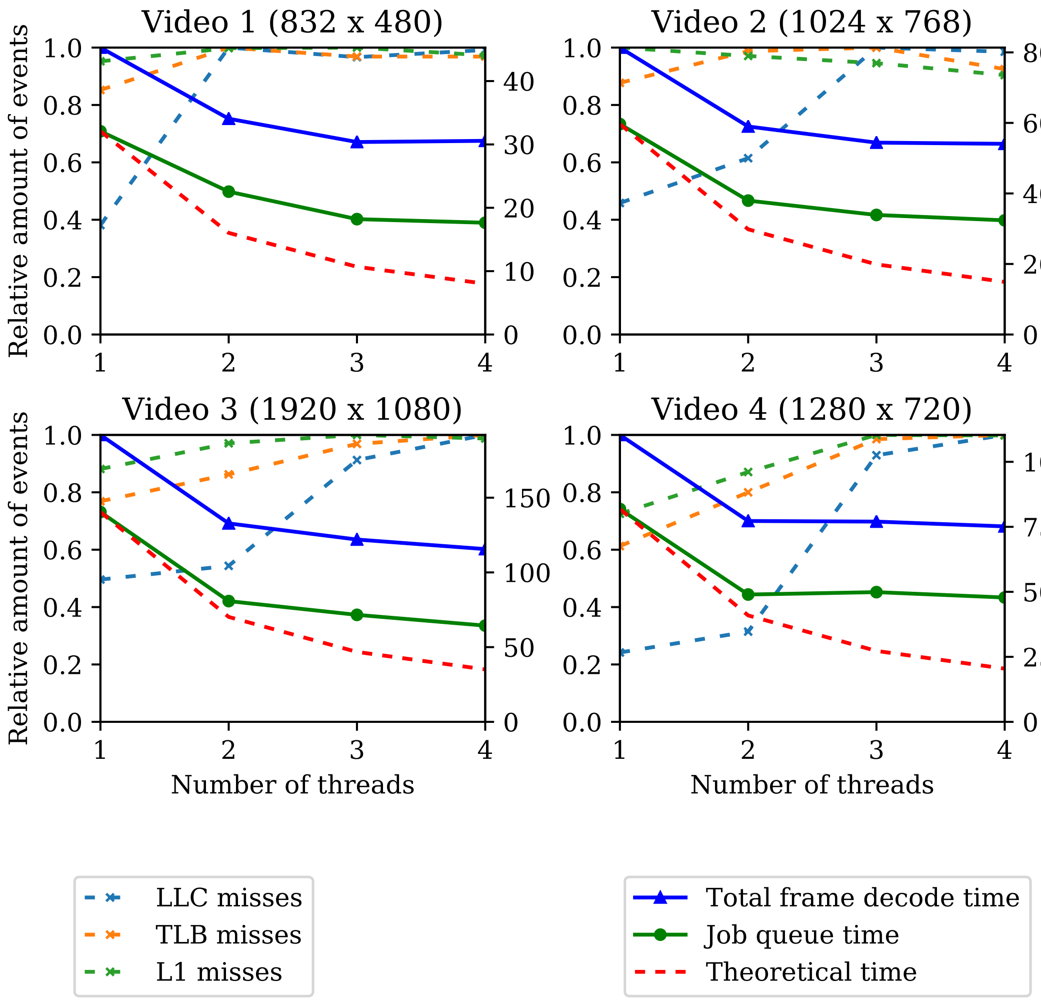

Figure 8.4: WPP measurements for setup 1.

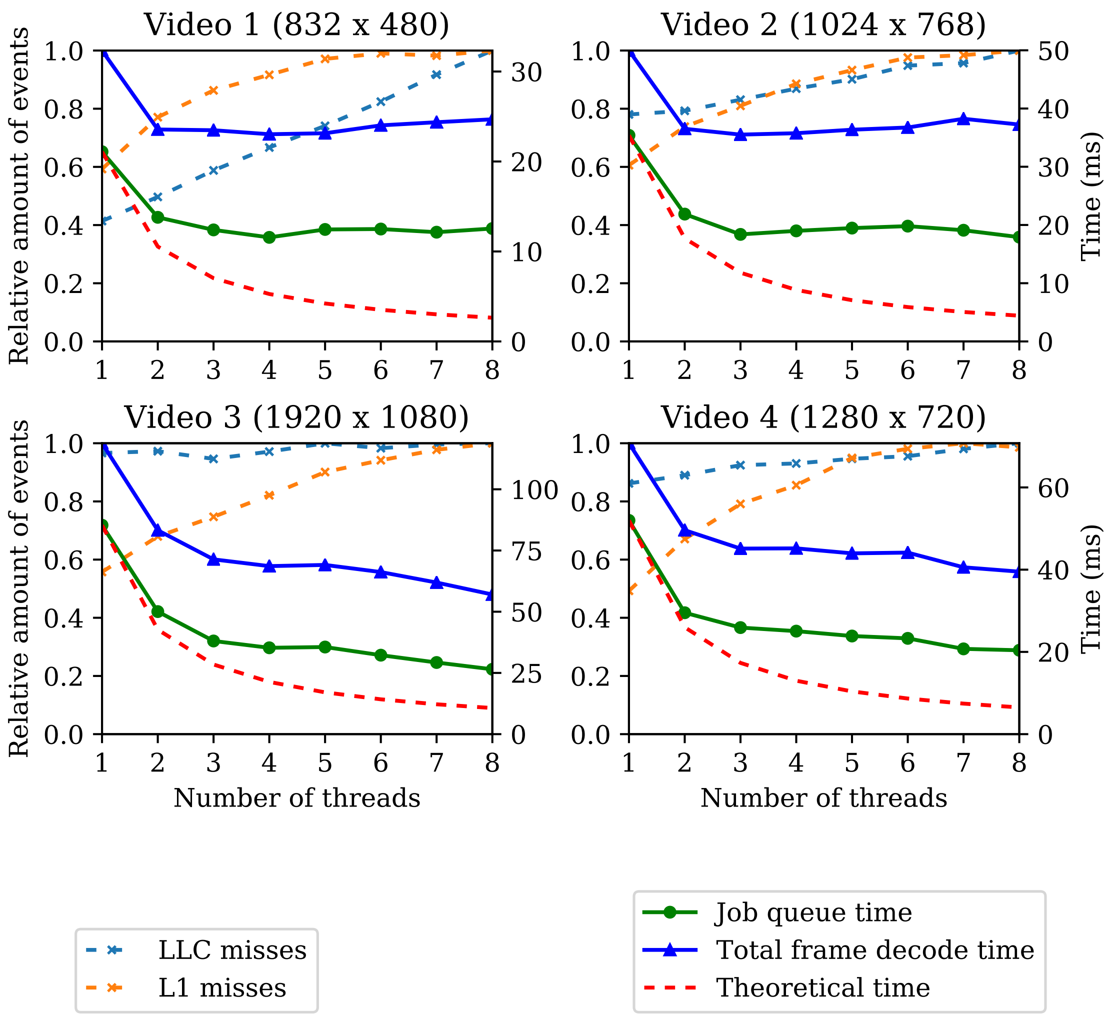

Figure 8.5: WPP measurements for setup 2.

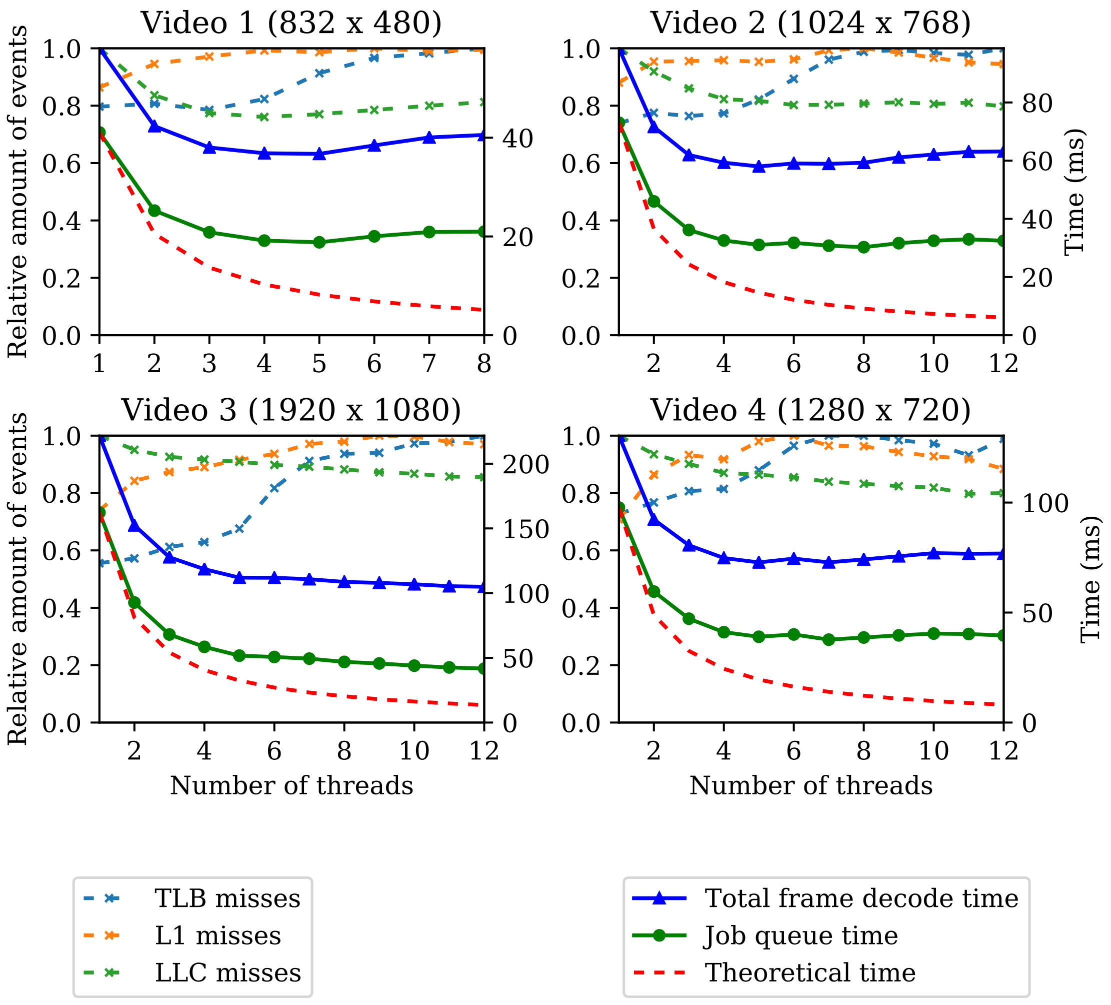

Figure 8.6: WPP measurements for setup 3. Unlike other videos, Video 1 has been measured using only up to 8 threads, unlike other videos that have been measured up to 12 threads. This is because Video 1 only has 8 CTU rows (Table 7.5) and WPP can only parallelize up to the amount of CTU rows in the video.

Setup 2 performs better than Setup 1, especially for videos with larger resolution. This is expected, since Setup 2 contains a newer CPU with much better benchmark score (see Table 7.1).

What is unexpected, is that Setup 3 performs worse than Setup 1 for Videos 1 and 2. For Videos 3 and 4, Setup 3 performs just barely batter than Setup 1. This is unexpected since Setup 3 has much better benchmark score, has more physical CPU cores, and higher clock speed compared to Setup 1 (see Table 7.1).

8.3 Tiles

The measurement results for Tiles parallel implementation have similar form for all videos on all setups (presented in Figs. 8.1–8.3). The speedup when moving from one CPU core to two cores closely follows the theoretical maximum speed up specified by Eq. (4.1). When adding more cores, speedup occurs, but the speedup is not as close to the theoretical limit. In many cases, the addition of the final core introduces a relatively large speedup.

8.3.1 Theoretical analysis of decode time

Assume that each tile takes the same amount of time to decode and there is no parallelization overhead when scheduling tiles to threads. In Fig. 8.7 is shown the total frame decode times and wasted computation times in a case where frames are decoded with 4 tiles and thread count ranges from 1 to 4.

When the number of tiles is divisible by the number of threads, no computation time is wasted. In Fig. 8.7, this is the case when thread count equals 1, 2, or 4.

When the number of tiles is not divisible by the number of threads, some threads are force to wait that the decoding of the last tiles of the current frame is complete. Only after all tiles are decoded, can the decoding of a new frame start. In Fig. 8.7, this is the case when thread count equals 3. Equal amount of time is taken to decode the frame when thread count equals 2 or 3, but with 3 threads, two threads spend one time unit waiting.

The total amount of time taken to decode a frame is given by

$$t_{total} = \left\lceil \frac{N}{T} \right\rceil, \tag{8.1}$$

where N is the amount of tiles and T is the used thread count.

The total amount of computational time, i.e. the time the threads are either decoding a tile or waiting, is given by

$$t_{comp} = t_{total}T = \left\lceil \frac{N}{T} \right\rceil T, \tag{8.2}$$

Since the total computation time required to decode the frame is N, the amount of idle computation time is given by

$$t_{idle} = t_{comp} - N = \left\lceil \frac{N}{T} \right\rceil T - N, \tag{8.3}$$

The total amount of time taken to decode a frame and the idle computational time as a function of thread count for tile counts 4, 8, and 12 are shown in Tables 8.1, 8.2, and 8.3, respectively. These are also plotted in Fig. 8.8 along with the measured times.

For thread count 4 (Setup 1), the similarity of the figures is clearly visible. Especially worth noting is that the measured total decode time stays the same when moving from 2 to 3 threads. A speedup similar to the theoretical model is observed when going from 3 to 4 threads.

For thread count 8 (Setup 2), the similarity of the figures is still visible, but no as clearly as when thread count is 4. A noticeable speedup is observed when going from 1 to 4 threads, after which no significant speedup can be observed. A speedup can be again observed when thread count is increased from 6 to 8.

For thread count 12 (Setup 3), the averaged measured times do not reflect the bumps in the theoretical times. It should however noted be that the speedup when going from thread count 11 to 12, can be seen for Video 1 and Video 2 measurements in Fig. 8.3.

Table 8.1: The total amount of time taken t_{total} and the idle computation time t_{idle} for tile count $N = 4$ and thread count $1 \leq T \leq 4$.

T	1	2	3	4
t_{total}	4	2	2	1
t_{idle}	0	0	2	0

Table 8.2: The total amount of time taken t_{total} and the idle computation time t_{idle} for tile count $N = 8$ and thread count $1 \leq T \leq 8$.

T	1	2	3	4	5	6	7	8
t_{total}	8	4	3	2	2	2	2	1
t_{idle}	0	0	1	0	2	4	6	0

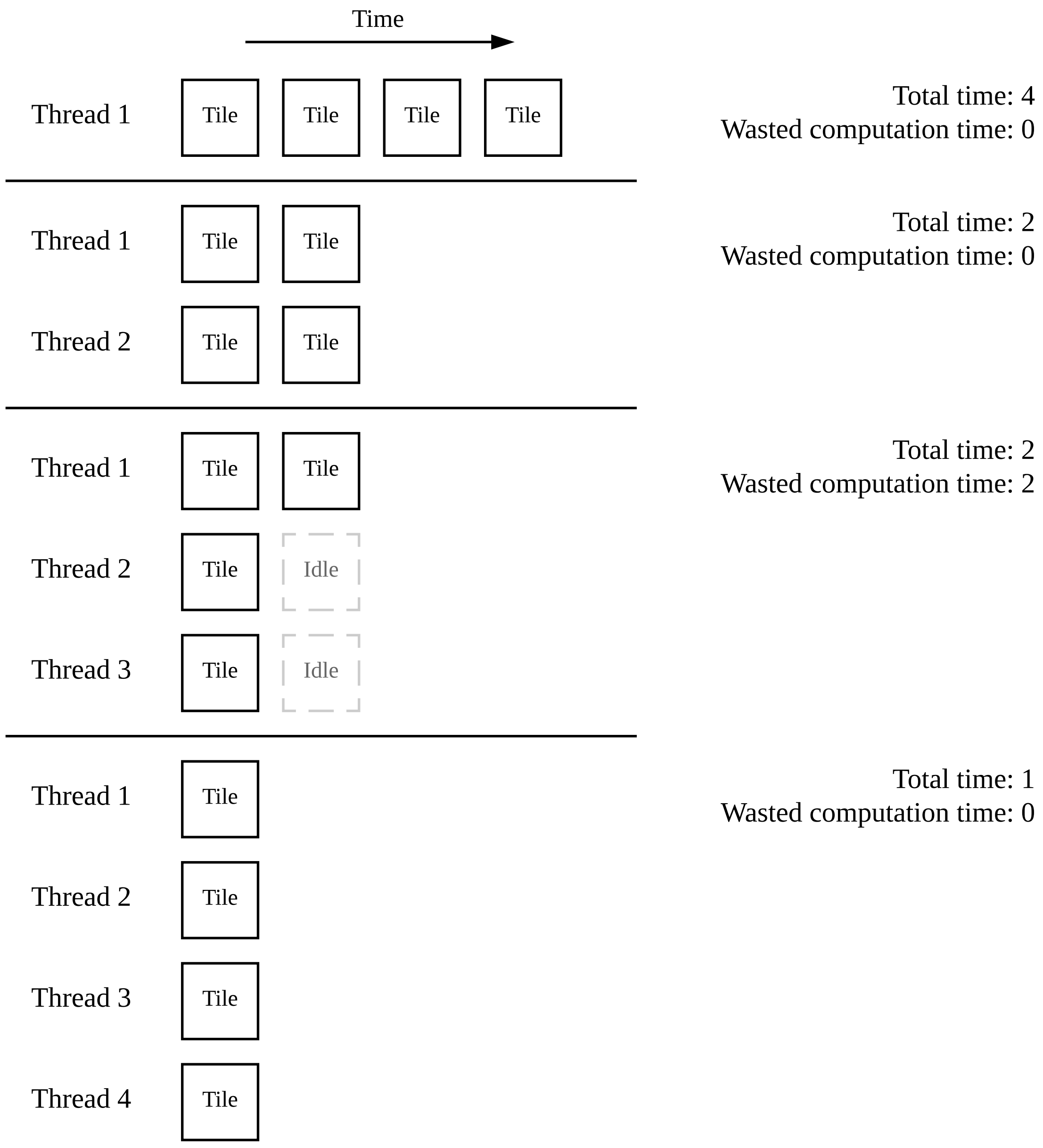

Figure 8.7: A diagram showing the total parallel section decode time of a frame for Tiles in an ideal case (all tiles take the same amount of time to decode and no parallelization overhead exists). In this case we have 4 tiles and thread count ranging from 1 to 4.

Table 8.3: The total amount of time taken t_{total} and the idle computation time t_{idle} for tile count $N = 12$ and thread count $1 \leq T \leq 12$.

T	1	2	3	4	5	6	7	8	9	10	11	12
t_{total}	12	6	4	3	3	2	2	2	2	2	2	1
t_{idle}	0	0	0	0	3	0	2	4	6	8	10	0

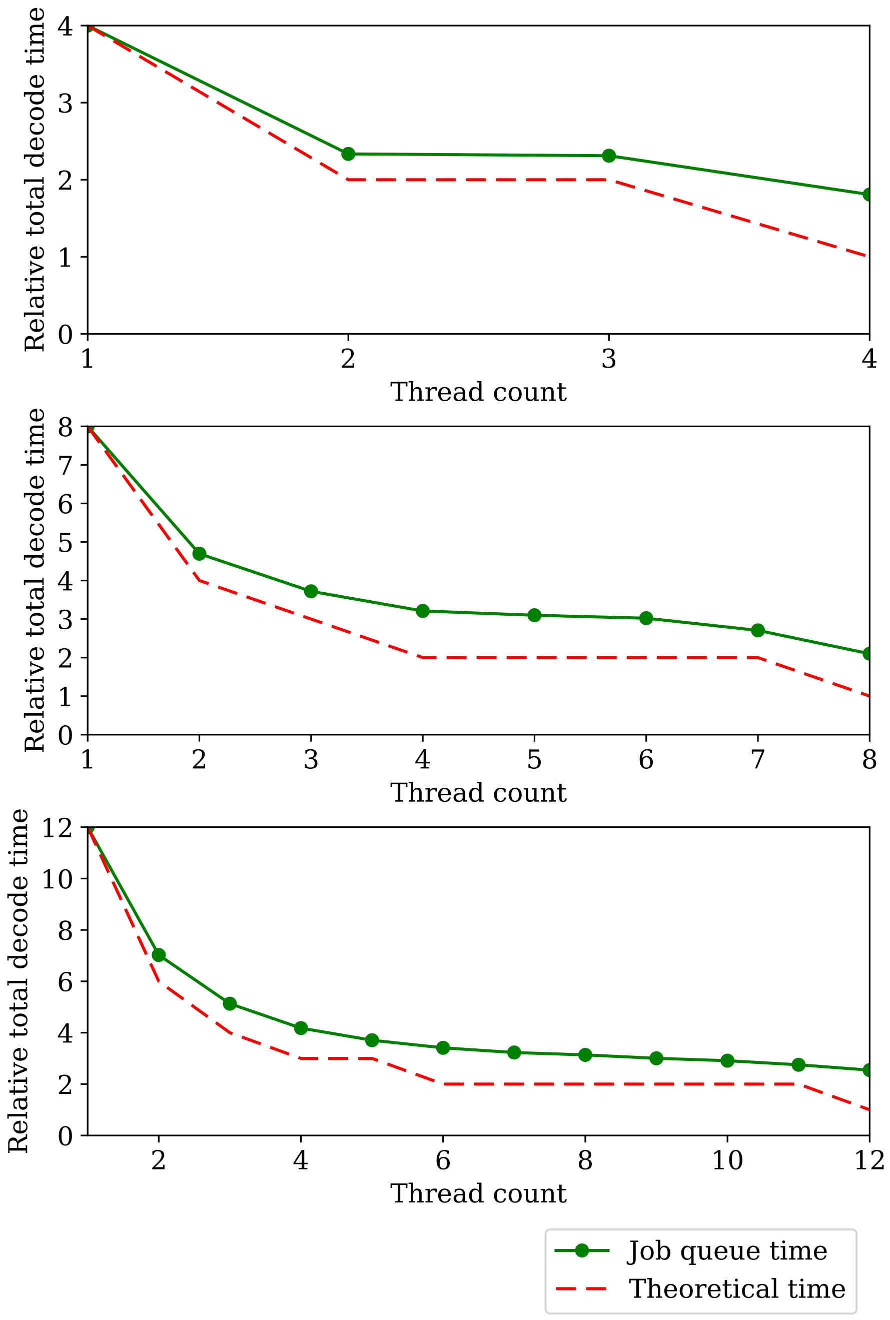

Figure 8.8: The theoretical total frame decode time for equal decode time tiles compared to the measured frame decode times. The measured times are shown are averages from the measurements of setups 1, 2, and 3, respectively. The times are scaled so that the measured times are equal to the theoretical times when thread count is 1.

In cases where the tile count is not divisible by the thread count, the wasted computation time could be avoided if the decoding of two frames could overlap. In this case, the idle threads would start decoding the next frame while some threads are still decoding the final tiles of the current frame. However, this would insert some requirements to the dependencies between tiles in consecutive frames and different filters are applied to the decoded image after this step, thus making the case more complex.

8.4 WPP

The measurement results for WPP parallel implementation are show in Figs. 8.4–8.6. Similar to the case with Tiles parallel implementation (section 8.3), the speedup when moving from one CPU core to two cores closely follows the theoretical maximum speed up specified by Eq. (4.1). However, when adding more threads, the results measured for WPP differ from the results for Tiles.

In Setup 1 (Fig. 8.4), the speedup either stops or is very minimal with the addition of the 3rd and 4th thread.

In Setup 2 (Fig. 8.5), the speedup in Videos 1 and 2 stops and even goes slightly negative when adding 3rd thread and going beyond that. Videos 3 and 4 behave somewhat similarly for WPP as they do for Tiles (Fig. 8.2); the speedup slows down after the 2nd thread, practically stops for few threads, and finally speedup is observed again after 5 or more threads.

In Setup 3 (Fig. 8.6), the speedup in Videos 1, 2, and 4 stops after the 4th thread and starts to go slightly negative. Video 3 is the only one in Setup 3 that does not present negative speedup when thread count increases.

When looking at the results per video, Video 1 shows the most negative speedup in Setups 2 and 3. Video 1 is the one with the smallest resolution and therefore the one with the least CTUs.

Video 2 also shows negative speed up in both Setups 2 and 3. It is the video with the second smallest resolution.

Video 3 is the only video that does not present any negative speedup in any in setup. It is the only video where the results between Tiles and WPP are very similar. Video 3 is the one with the largest resolution and therefore the one with the most CTUs.

Video 4 shows a small positive speedup for higher thread counts in Setup 2 but a slight negative speedup in Setup 3. It is the video with the second largest resolution.

Based on the above observations we can do a quantitative analysis. Most negative speedup among all videos is observed in Setup 3, some negative speedup is observed in Setup 2, and in Setup 1 no negative speedup is observed. It would seem that the more threads we have in the our setups, the more negative speedup we start to observe with higher thread counts.

The resolution of the video also seems to be related to the negative speedup. The smaller the video, the earlier, and more, it starts to present a negative speedup.

One possible reason for the negative speedup would be synchronization overhead. The more there are threads, the more signaling and waiting between the threads is needed. This synchronization overhead does not show up in the Tiles measurements since the worker threads on Tiles do not need to synchronize with each other; worker threads only need to synchronize with the master thread when they are ready.

8.5 Comparison to related work

In this section, we will compare our measurement results to those of the related works done by others as presented in section 5.6.

In our measurements, the Tiles implementation was slightly faster than the WPP implementation. This can also be seen from the measurements done in [41]. In their measurements, the scalability of the implementation started to reduce after 6 threads for 1080p resolution video. The same can be noticed from our measurements.

The synchronization overhead visible in our measurements was not visible in other works. This is probably due to better synchronization methods and larger video resolutions.

The video resolutions ranged from 832×480 to 1080p in our measurements. In other works, mostly videos with resolutions starting from Full HD were used. The speedup in [41, 42, 43] scaled better when measuring videos with resolution larger than 1080p (video resolutions up to 2160p were measured).

8.6 Possible improvements and future ideas

In this section we present some ideas for improvements and future work. First we discuss the measurements in general, then Tiles measurements, and finally WPP.

In the current measurement setup only the average frame decode time is measured. It would also be interesting to measure the distribution of the decode times. Another interesting quantity to measure would be the time the threads spend on waiting.

Like mentioned in section 8.5, the scalability in related works increased as the resolution increased. The largest video resolution in our measurements was 1080p (Full

HD). It would be interesting to extend the measurements to 2160p (4K) and even 4320p (8K).

8.6.1 Tiles

In the current measurement setup the tile count was always equal to the number of cores in the machine of each setup. I.e. Setup 1 only decoded frames with 4 tiles, Setup 2 only decoded frames with 8 tiles, etc. We could have, for example, ran Setup 1 with 8 and 12 tiles.

All encoded streams had an even number of tiles. We could have, for example, encoded a stream with $3 \times 3 = 9$ tiles and measure the performance when decoded using an even number of cores.

The decode times of individual tiles were not measured. These would have been interesting when comparing the measured speedup to the theoretical maximum speedup. Using Tiles parallelization, when thread count is equal to the number of tiles, the maximum speedup could only be achieved if the decode time for all tiles is equal. In practice, this naturally is not the case. The decoding of the frame is complete, and the processing can move forward, only after all threads have completed. Therefore, all other threads, except the slowest thread, will have to spend some time waiting for the slowest thread to complete. Essentially this means that in the case where the thread count equals the tile count, the frame decode time (for the parallel section) will be the decode time of the slowest tile. The greater the variance between the slowest and an average tile decode time, the less speedup will be achieved when increasing the thread count.

Generally, Tiles setup scaled well. For the 1080p resolution video, the measurement results were mostly similar to results achieved in related works mentioned in section 5.6. The step behavior explained in section 8.3.1 was not observed in any related works. This is probably because in the related work only videos with resolutions 1080p and up were measured and the step behavior was only observed with smaller resolutions.

8.6.2 WPP

In the measured video streams encoded using WPP, the LCU size was 64 for all encoded videos. It would also have been interesting to see how changing the LCU size affects the performance.

The decode times for for individual CTUs were not measured. The distribution of the CTU decode times could have been compared to the frame decode times to see if some correlation would exists.

The progressions of the wavefront could have been measured. This would require measuring the start and finish times of each CTU decoding relative to the beginning of the whole frame decoding. This way it could be seen how smooth the decoding of the whole frame was and if there exists some locations in the frame that cause problems with the parallelization.

In the current implementation, the threads always process one CTU row from the beginning to the end. In other words, the threads can not switch rows until they have completed the one they are currently processing. In some scenarios this can cause threads on lower rows to spend time waiting for threads on upper rows that are decoding more computationally intensive CTUs, even though these waiting threads could already start decoding rows lower than they are currently decoding. This is a probable cause for the synchronization overhead we observed. An improvement to the job queue to reduce the synchronization overhead could be that threads could put jobs on hold and start working on others jobs. In many related works, Ring-Line synchronization has been used in line decoders to obtain good parallelization results [43, 53, 54].

Chapter 9

Conclusions

As the demand for high resolution videos and good quality video communication grows, the need for more efficient compression techniques emerges. Since most hardware devices today have more than one CPU core, it is crucial to make the best use of the available computing resources to more efficiently compress and decompress video. This is the goal behind the H.265 video coding standard: to achieve better compression ratios for high resolution videos and take better usage of parallel computation resources.

The objective of this study was to research the scalability of parallel execution of H.265. In this study we modified the H.265 reference decoder to support two novel parallelization strategies: Tiles and Wavefront Parallel Processing (WPP). A special synchronized data structure called job queue was implemented. It was used as a framework for implementing the parallel features into the decocer. The performance of both methods was also measured using different setups and test videos.

From the Tiles measurements it was observed that the amount of tiles in the frame needed to correlate with the amount of threads to achieve maximum performance. If the amount of tiles in the frame is not divisible with the number of threads used, some threads will use time waiting. However, with larger tile counts this effect is not so noticeable as the variance of tile decoding times presumably grows. This should be kept in mind when encoding video using the Tiles method; the level of parallelism is decided when encoding the video. The effect caused by non-matching thread and tile count was not observed in measurements in related works done by others. In these works, generally videos with resolutions from 1080p (Full HD) and up were used with large tile counts (18 and up). As mentioned before, in our measurements this effect was only visible in lower resolution videos with low tile counts. Therefore, it would not even be expected to see this effect in these works.

The WPP implementation performed and scaled somewhat worse compared to the Tiles implementation in the measurements. This was presumably due to a synchronization overhead which occurred when large numbers of threads were used. In our implementation, some scenarios can cause threads to wait for other threads, even though they could switch to process other tasks in the meanwhile. It is therefore crucial that an efficient synchronization implementation is used with WPP. For example, in many related works, a method called ring-line synchronization was used with results indicating good scalability.

In our work we noticed that H.265 does not scale well when decoding videos with smaller resolutions (smaller than Full HD) compared to larger resolution videos. However, the main purpose of H.265 is to target large resolution videos. Smaller videos are easily decodable with modern devices without the need for a high level of parallelism. Combining our findings with other related work, it seems that the H.265 has met its design targets; efficient and scalable processing of high resolution videos.

In the future, it would be interesting to test the scalability of the implementation using large resolution videos, even up to 4320p (8K). In addition, the synchronization in the WPP implementation could relatively easily be improved.

References

[1] Michael W. Marcellin et al. “An overview of JPEG-2000”. In: *Data Compression Conference. Proceedings.* (2000).

[2] Iain E. Richardson. *H.264 Advanced Video Compression Standard*. 2nd Ed. Chichester, United Kingdom: Wiley, 2010. ISBN: 978-0-470-51692-8.

[3] M. Ezhilarasan and P. Thambidurai. “A Hybrid Transform Coding for Video Codec”. In: *9th International Conference on Information Technology, ICIT* (2006).

[4] Shigang Wang and Hexin Chen. “An improve algorithm of motion compensation MPEG video compression”. In: *Proceedings of the IEEE International Vehicle Electronics Conference IVEC* (1999).

[5] Mathias Wien. “Variable block-size transforms for H.264/AVC”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 13 (7 2003), pp. 604–613.

[6] Mohammed Golam Sarwer and Q. M. Jonathan Wu. *Effective Video Coding for Multimedia Applications*. Rijeka, Croatia: InTech, 2011. ISBN: 978-953-307-177-0.

[7] M. Uenohara and T. Kanade. “Optimal approximation of uniformly rotated images: relationship between Karhunen-Loeve expansion and discrete cosine transform”. In: *IEEE Transactions on Image Processing* 7 (1 1998).

[8] Juraj Bienik et al. “Performance of H.264, H.265, VP8 and VP9 Compression Standards for High Resolutions”. In: *19th International Conference on Network-Based Information Systems (NBiS)* (2016).

[9] Mathias Wien. *High Efficiency Video Coding - Coding Tools and Specification*. Springer, 2015.

[10] *Xiph.org Foundation*. URL: https://xiph.org (visited on 05/10/2017).

[11] *Alliance for Open Media (AOMedia)*. URL: http://aomedia.org (visited on 05/10/2017).

[12] Enrique de la Torre, Rafael Rodriguez-Sanchez, and Jose Luis Martínez. "Fast video transcoding from HEVC to VP9". In: *IEEE Transactions on Consumer Electronics* 61 (3 2015), pp. 336–343.

[13] James Taylor. *DVD Demystified.* 2nd Ed. McGraw-Hill, 2001. ISBN: 978-0-071-42396-0.

[14] Blu-ray Disc Association. *White Paper - Blu-ray Disc^{TM} Format.* 4th Ed. 2015.

[15] Blu-ray Disc Association. *White Paper - Blu-ray Disc^{TM} Read-Only Format (Ultra HD Blu-ray^{TM}) - Audio Visual Application Format Specifications for BD-ROM.* Version 3.1. 2016.

[16] Philippe de Cuetos and Keith W. Ross. "Unified framework for optimal video streaming". In: *23rd AnnualJoint Conference of the IEEE Computer and Communications Societies. Proceedings* (2004).

[17] Prasenjit Chakraborty, Sachin Dev, and Rajaram Hanumantacharya Naganur. "Dynamic HTTP Live Streaming Method for Live Feeds". In: *International Conference onComputational Intelligence and Communication Networks (CICN)* (2015).

[18] Cyril Concolato et al. "Adaptive Streaming of HEVC Tiled Videos using MPEG-DASH". In: *IEEE Transactions on Circuits and Systems for Video Technology* (2017).

[19] Mikko Uitto. "Energy consumption evaluation of H.264 and HEVC video encoders in high-resolution live streaming". In: *IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob). Proceedings* (2016).

[20] Xiangbo Li, Mohsen Amini Salehi, and Magdy Bayoumi. "VLSC: Video Live Streaming Using Cloud Services". In: *IEEE International Conferences on Big Data and Cloud Computing, Social Computing and Networking, Sustainable Computing and Communications (BDCloud-SocialCom-SustainCom)* (2016).

[21] Irini S. Reljin and Aleksandar N. Sugaris. "DVB standards development". In: *9th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services. Proceedings* (2009), pp. 263–272.

[22] Krešimir Šakić, Vedran Bubalo, and Sonja Grgić. "Choice of video coding standard for future DVB-T2 networks in Croatia". In: *58th International Symposium ELMAR* (2016).

[23] Ilya Grigorik. *High Performance Browser Networking.* O'Reilly Media, 2013.

[24] Eleftheriadis Alexandros, Civanlar M. Reha, and Shapiro Ofer. "Multipoint videoconferencing with scalable video coding". In: *Journal of Zhejiang University SCIENCE A* 7 (5 2006), pp. 696–705.

[25] Peter Amon, Madhurani Sapre, and Andreas Hutter. "Compressed domain stitching of HEVC streams for video conferencing applications". In: *Packet Video Workshop (PV), 19th International* (2012), pp. 36–40.

[26] Ananth Grama et al. *Introduction to Parallel Computing.* 2nd Ed. Harlow, United Kingdom: Pearson, 2003. ISBN: 0-201-64865-2.

[27] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". In: *AFIPS Conference Proceedings* 30 (1967), pp. 483–485.

[28] Gudula Rünger and Thomas Rauber. *Parallel Programming: For Multicore and Cluster Systems.* Berlin, Germany: Springer, 2010. ISBN: 978-3-642-04817-3.

[29] William Stallings. *Operating Systems: Internals and Design Principles.* 7th Ed. United Kingdom: Pearson Education Limited, 2012. ISBN: 978-0-273-75150-2.

[30] David Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* 4th Ed. Burlington, Massachusetts, USA: Morgan Kaufman, 2009. ISBN: 978-0-12-374493-7.

[31] Michael J. Flynn. "Some Computer Organizations and their Effectiveness". In: *IEEE Transactions on Computers* 21 (9 1972), pp. 948–960.

[32] András Vajda. *Programming Many-core Chips.* US: Springer, 2011. ISBN: 978-1-4419-9739-5.

[33] Gary J. Sullivan et al. "Overview of the High Efficiency Video Coding (HEVC) Standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22 (12 2012), pp. 1649–1668.

[34] Vivienne Sze and Madhukar Budagavi. "High Throughput CABAC Entropy Coding in HEVC". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22 (12 2012), pp. 1778–1791.

[35] Chih-Ming Fu et al. "Sample Adaptive Offset in the HEVC Standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22 (12 2012), pp. 1755–1764.

[36] Kiran Misra et al. "An Overview of Tiles in HEVC". In: *IEEE Journal of Selected Topics in Signal Processing* 7 (6 2013), pp. 969–977.

[37] Jens-Rainer Ohm et al. "Comparison of the Coding Efficiency of Video Coding Standards–Including High Efficiency Video Coding (HEVC)". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22 (12 2012), pp. 1669–1684.

[38] Ji-Yong Shin et al. "Hardware architecture design of an H.264/AVC video codec". In: *Proceedings of the 2006 Asia and South Pacific Design Automation Conference* (2000), pp. 750–757.

[39] Frank Bossen et al. "HEVC Complexity and Implementation Analysis". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22 (12 2012), pp. 1685–1696.

[40] Liquan Shen et al. "An Effective CU Size Decision Method for HEVC Encoders". In: *IEEE Transactions on Multimedia* 15 (2 2013), pp. 465–470.

[41] Chi Ching Chi et al. "Parallel Scalability and Efficiency of HEVC Parallelization Approaches". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22 (12 2012), pp. 1827–1838.

[42] Wassim Hamidouche, Michael Raulet, and Olivier Deforges. "Parallel SHVC decoder: Implementation and analysis". In: *IEEE International Conference on Multimedia and Expo (ICME)* (2014).

[43] Mauricio Alvarez-Mesa et al. "Parallel video decoding in the emerging HEVC standard". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2012).

[44] Shaobo Zhang, Xiaoyun Zhang, and Zhiyong Gao. "Implementation and improvement of Wavefront Parallel Processing for HEVC encoding on many-core platform". In: *IEEE International Conference on Multimedia and Expo Workshops (ICMEW)* (2014).

[45] JCT-VC. *HEVC reference software (HM)*. Version 10.1. URL: https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware.

[46] *Linux `perf` wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 02/12/2014).

[47] David Levinthal. *Performance Analysis Guide for Intel© Core^{TM} i7 Processor and Intel© Xeon^{TM} 5500 processors*. Version 1.0. 2014. URL: http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[48] *CPU Benchmarks - CPU Test Information*. URL: https://www.cpubenchmark.net/cpu_test_info.html (visited on 04/11/2017).

[49] *CPU Benchmarks - Intel© Core^{TM}2 Quad Q9550 @ 2.83GHz*. URL: http://www.cpubenchmark.net/cpu.php?cpu=Intel+Core2+Quad+Q9550+@+2.83GHz (visited on 04/11/2017).

[50] *CPU Benchmarks - Intel© Core^{TM} i7-3770 @ 3.40GHz*. URL: https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i7-3770+@+3.40GHz (visited on 04/11/2017).

[51] *CPU Benchmarks - Intel© Core^{TM} i7-970 @ 3.20GHz*. URL: https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i7-970+@+3.20GHz (visited on 04/11/2017).

[52] JCT-VC. *HEVC test sequences*.

[53] Chi Ching Chi and Ben Juurlink. “A QHD-capable parallel H.264 decoder”. In: *Proceedings of the international conference on Supercomputing (ICS)* (2011), pp. 317–326.

[54] Chi Ching Chi, Ben Juurlink, and Cor Meenderinck. “Evaluation of parallel H.264 decoding strategies for the Cell Broadband Engine”. In: *Proceedings of the 24th ACM International Conference on Supercomputing (ICS)* (2010), pp. 105–114.