Tampere University

Jaakko Laitinen

SCALABLE KVAZAAR

Implementing HEVC Scalability Extension in Kvazaar Open-Source Encoder

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Assoc. Prof. Jarno Vanne
Prof. Timo D. Hämäläinen
November 2020

ABSTRACT

Jaakko Laitinen: Scalable Kvazaar — Implementing HEVC Scalability Extension in Kvazaar Open-Source Encoder
Master of Science Thesis
Tampere University
Master's Programme in Information Technology
November 2020

Global internet video traffic is predicted to keep increasing year after year, making up 82% of all internet traffic by the year 2022. This increase is accelerated by the proliferation of different media devices and higher video resolutions. To address the explosive growth of video traffic, more efficient video compression methods are needed. To this end, ITU-T VCEG and ISO/IEC MPEG defined the *High Efficiency Video Coding* (HEVC) standard that improves compression efficiency by roughly 50% for the same visual quality over the previous video coding standards. The HEVC standard also includes *Scalable High Efficiency Video Coding* (SHVC) — the scalable extensions of HEVC — for creating video streams that can adapt to changing network conditions and playback devices.

SHVC extends the functionality of HEVC by creating an encoded video that can be made up of several versions of the same input with different quality and video parameters. It uses an *inter-layer reference* (ILR) mechanism to improve coding efficiency by taking advantage of the redundancy between the different video quality versions. The types of scalability discussed in this thesis are quality and spatial scalability; they can be used to provide the user with videos containing multiple quality and resolution levels, respectively.

At the time of writing, no open-source solutions are available for real-time SHVC encoding. To remedy this, the Scalable Kvazaar SHVC encoder is proposed in this work. It is based on the open-source Kvazaar HEVC encoder, and it implements quality and spatial scalability functionality in a practical encoder. This thesis proposes three main optimization approaches to accelerate Scalable Kvazaar. The first two approaches involve integrating inter-layer processing with *wavefront parallel processing* (WPP) and *overlapped wavefront* (OWF) parallelization. The third approach employs *single instruction, multiple data* (SIMD) optimizations on the picture upscaling functions. Through these new SHVC specific optimizations, Scalable Kvazaar aims to achieve real-time coding speeds.

The performance of Scalable Kvazaar was measured with three test cases. In the quality scalability case, Scalable Kvazaar is able to reduce bit rate by 16.16% on average, and it achieves a $1.20\times$ speedup over simulcast coding, where each quality version is encoded separately. The respective values for $2\times$ spatial scalability are 13.12% and $1.03\times$. Finally, when using a $1.5\times$ scaling ratio, the bit rate reduction is 23.19% and the speedup is $1.04\times$. As for absolute coding speed, Scalable Kvazaar was able to encode 1080p video at over 40 frames per second in all test cases, thereby hitting the real-time encoding target.

Keywords: HEVC, SHVC, Kvazaar, Scalable Kvazaar, Open-source

TIIVISTELMÄ

Jaakko Laitinen: Skaalautuva Kvazaar — HEVC:n skaalautuvan lisäosan toteuttaminen avoimen lähdekoodin Kvazaar-videonpakkausohjelmistoon
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Marraskuu 2020

Maailman internet-liikenne koostuu enenemässä määrin videosta, jonka ennustetaan saavuttavan 82% osuuden vuoteen 2022 mennessä. Kasvua vauhdittavat medialaitteiden videoresoluutioiden kasvu sekä video-ominaisuuksien monipuolistuminen. ITU-T VCEG ja ISO/IEC ovat vastanneet tähän videoliikenteen kasvuun määrittelemällä *High Efficiency Video Coding* (HEVC) videonpakkausstandardin, joka pienentää videon kokoa noin 50% samalla kuvanlaadulla edellisiin videonpakkausstandardeihin verrattuna. HEVC-standardi sisältää myös tuen *Scalable High Efficiency Video Coding* (SHVC) –lisäosalle, joka mahdollistaa koodattujen videoiden mukautumisen muuttuviin verkon olosuhteisiin ja jopa eri tehoisille laitteille.

SHVC-pakattu video voi sisältää eri laaduilla ja resoluutioilla pakattuja versioita samasta videosta. SHVC käyttää hyödykseen versioiden välistä samankaltaisuutta, mikä mahdollistaa paremman pakkaustehokkuuden saavuttamisen. Tässä työssä keskitytään pääasiassa laadun ja kuvakoon skaalautuvuuteen.

SHVC-pakkaukseen tarkoitettuja ohjelmistoja ei ole tällä hetkellä monia. Erityisesti reaaliaikaiseen pakkaukseen kykeneviä avoimen lähdekoodin ohjelmistoja ei kirjoitushetkellä ole olemassa. Tässä työssä esitellään SHVC-pakkaukseen kykenevä Scalable Kvazaar –niminen ohjelmisto. Se pohjautuu avoimen lähdekoodin HEVC-videokooderiin nimeltä Kvazaar ja tukee kuvanlaadun sekä kuvakoon muuttamista. Tässä työssä esitetään kolme nopeutuskeinoa Scalable Kvazaarille. Kaksi ensimmäistä keinoa liittävät skaalautuvuuden kuvansisäiseen rinnakkaiseen prosessointiin (WPP) ja kuvien väliseen rinnakkaiseen pakkaukseen (OWF). Kolmas keino optimoi kuvan skaalaukseen käytettyjä funktioita vektorikäskyjen avulla (SIMD). Näiden uusien skaalautuvuuten liittyvien optimointien avulla, Scalable Kvazaar pyrkii saavuttamaan reaaliaikaisen pakkauksen.

Scalable Kvazaarin pakkaustehokkuutta ja suorituskykyä on mitattu kolmessa eri testikokoonpanossa, joissa verrokkina käytettiin Kvazaarin pakkaamia videoita. Kun ainoastaan versioiden välistä laatua muutettiin, Scalable Kvazaar pienensi tiedostokokoa keskimäärin 16.16% ja saavutti $1.20\times$ nopeutuksen Kvazaariin verrattuna. Tarkasteltaessa kokoonpanoa, jossa versioiden resoluutioiden välinen suhde oli $2\times$, vastaavat tulokset olivat 13.12% ja $1.03\times$. Resoluutioiden suhteen ollessa $1.5\times$, tiedostokoko pieneni 23.19% ja pakkaus nopeutui $1.04\times$ kertaiseksi. Absoluuttista pakkausnopeutta tarkasteltaessa Scalable Kvazaar kykeni pakkaamaan 1080p videota yli 40 kuvaa sekunnissa kaikissa testikokoonpanoissa saavuttaen reaaliaikaisen suorituskyvyn.

Avainsanat: HEVC, SHVC, Kvazaar, Skaalautuva Kvazaar, Avoimen lähdekoodin

PREFACE

This thesis was written as part of research conducted in the Laboratory of Pervasive Computing at the late Tampere University of Technology and, more recently, in the unit of Computing Sciences at the unified Tampere University.

I would like to thank my examiner and advisor Jarno Vanne for his guidance with this thesis and for giving me the opportunity to work on this project and be a part of Ultra Video Group. In addition, I would also like to thank Marko Viitanen for helping with this thesis. Finally, I would like to thank everyone at Ultra Video Group for making it such a great place to work.

A special thank you to my parents for their support and assistance during my studies as well as for their help getting me this far.

In Tampere, 1st November 2020

Jaakko Laitinen

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

4K	designates a class of resolutions with roughly 4,000 horizontal pixels, depending on the specific standard
AMP	asymmetric motion partitioning
AMVP	advanced motion vector prediction
API	application programming interface
AVX	Advanced Vector Extensions; an x86 instruction set extension, providing CPU-level support for SIMD operations
B	number of bits used to represent a pixel value
BD-rate	Bjøntegaard delta bit rate; a metric for calculating video coding efficiency using piece-wise cubic interpolation to compare bit rate curves
BL	base layer
C	an imperative programming language with a long history dating back to the 1970s
CABAC	context-based adaptive binary arithmetic coding
CDN	content delivery network
CLI	command-line interface
C++	a successor to the C programming language that adds object-oriented and other high-level features
CPU	central processing unit; a component in modern computers for performing various computational operations
CTU	coding tree unit
CU	coding unit
DCT	discrete cosine transform
DPB	decoded picture buffer
ΔQP	delta value for QP used in SNR scalability
EL	enhancement layer
FPS	frames per second
GOP	group of pictures

H	height of a picture P
HD	high-definition
HEVC	High Efficiency Video Coding
HM	HEVC Test Model
IEC	International Electrotechnical Commission (Commission électrotechnique internationale in French)
ILR	inter-layer reference
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union, standardization sector (Union Internationale des Télécommunications in French)
LID	layer ID
LUT	lookup table
MPEG	Moving Picture Experts Group
MSE	mean squared error
NALU	network abstraction layer unit
NAVETTA	Not Another Video Encoder Tester Tool Alternative; a Python based framework for testing (scalable) video encoding software
OWF	overlapped wavefront
P	matrix representing the pixel values of an input picture of size $W \times H$
p_{xy}	value of the element in the xth column and yth row of the P matrix
POC	picture order count
P'	matrix representing the pixel values of a reconstructed picture of size $W \times H$
p'_{xy}	value of the element in the xth column and yth row of the P' matrix
PPS	picture parameter set
P'_{U}	matrix representing the pixel values of the U channel of a reconstructed picture
P'_{V}	matrix representing the pixel values of the V channel of a reconstructed picture
P'_{Y}	matrix representing the pixel values of the Y channel of a reconstructed picture
PSNR	peak signal-to-noise ratio

$\mathrm{PSNR}_{\mathrm{YUV}}$	weighted average PSNR value of all three color channels of an YCbCr picture
PU	prediction unit
P_{U}	matrix representing the pixel values of the U channel of an input picture
P_{V}	matrix representing the pixel values of the V channel of an input picture
P_{Y}	matrix representing the pixel values of the Y channel of an input picture
QP	quantization parameter
RGB	a color model that divides colors into three parts (red, green, blue) that, when added together, form the actual color
RL	reference layer
SAO	sample adaptive offset
SHM	SHVC Test Model
SHVC	Scalable High Efficiency Video Coding
SIMD	single instruction, multiple data
SMP	symmetric motion partitioning
SNR	signal-to-noise ratio
SPS	sequence parameter set
SSIM	structural similarity index measure; an objective image distortion metric designed to correlate well with subjective quality
TID	temporal ID
TMVP	temporal motion vector prediction
TU	transform unit
TV	also known as television; a device for viewing moving picture content, often transmitted over ground cables or using radio-frequencies
U	the blue difference chroma component of a YCbCr picture (i.e., Cb)
UHD	ultra-high-definition
V	the red difference chroma component of a YCbCr picture (i.e., Cr)
VCEG	Video Coding Experts Group
VOD	video-on-demand

VPS	video parameter set
VVC	Versitile Video Coding
W	width of a picture P
WPP	wavefront parallel processing
Y	luma (or luminance) component of a YCbCr picture
YCbCr	a color space that defines three color components: Y, U (also called Cb), and V (also called Cr)
YUV	a (file) format for representing YCbCr data

1 INTRODUCTION

Global video traffic on the internet is predicted to increase four-fold between the years 2017 and 2022, making up 82% of all internet traffic in 2022 [1]. In addition, the trend seems to be towards *ultra-high-definition* (UHD) [2] video, with two-thirds of all connected TVs supporting UHD content by 2023 [3]. Furthermore, various mobile devices are becoming more popular for viewing video content, and the number of these devices is predicted to grow around 6% annually.

To combat the increasing demand for high resolution video, the ITU-T VCEG and ISO/IEC MPEG standardization organizations drafted a video coding standard, for (lossy) video compression, that was first released as a twin text in 2013 [4]. This standard is called the *High Efficiency Video Coding* (HEVC) standard and is the latest mainstream video coding standard from ITU-T and ISO/IEC. It aims to improve coding efficiency by 50% when compared with previous standards [5]. Subsequent versions of the HEVC standard have added various extensions to the base standard [6]. This thesis focuses on the *Scalable High Efficiency Video Coding* (SHVC) [7] extension that was finalized in 2014. SHVC addresses the need to support various viewing devices and resolutions.

SHVC enables encoding video with multiple layers of different quality levels. These layers can have different video parameters, usually such that higher layers represent higher quality video. The lowest quality layer is referred to as the *base layer* (BL) and is decodable with a standard HEVC decoder. Higher layers, referred to as *enhancement layers* (ELs), provide better quality versions of the video. They can use lower quality ELs and the BL as an *inter-layer reference* (ILR). This improves coding efficiency over that of simulcast coding, where separate video sequences are encoded for each quality setting represented by the BL and ELs, respectively. The main forms of scalability, i.e., the video parameters that can be changed between layers in SHVC are:

Quality scalability layers are encoded with different quality settings

Spatial scalability layers have different spatial resolutions

Bit depth scalability layers use different numbers of bits to represent a pixel

Color gamut scalability layers use different color spaces

Hybrid codec scalability the BL can be a non-HEVC video sequence

In addition, temporal scalability, for changing frame rate, is a scalability related feature, but it was already included in the base HEVC standard. As an improvement over previous scalable coding standards, SHVC was designed to only require high-level changes to a single-layer HEVC encoder, so as to minimize implementation overhead [7].

The main use cases of SHVC include *video-on-demand* (VOD) [8], broadcasting [9], and teleconferencing [10] where mobile devices may not be able to decode video at the highest quality. Media-aware network elements can be used to reduce bandwidth by dropping (higher) ELs and only transmitting the BL and possibly some (lower) ELs. SHVC can also improve error resiliency by providing the BL with a reliable, but small throughput, channel and the EL with a less reliable, high throughput, channel [9].

For HEVC, many practical encoder implementations exist; noteworthy open-source HEVC encoders include Kvazaar [11], x265 [12], SVT-HEVC Encoder [13], and Turing codec [14]. SHVC, on the other hand, lacks practical open-source encoders. It does, however, have a reference encoder (*SHVC Test Model* (SHM) [15]), but it targets the best possible coding efficiency, leading to a prohibitively high computational complexity, making it only usable in research and conformance testing. Parois et al. [16] have presented a closed-source real-time SHVC encoder, based on a proprietary solution, making its wider adaptation difficult. On the other hand, an efficient open-source SHVC decoder implementation has been presented by Hamidouche et al. [17].

This thesis seeks to solve the lack of practical SHVC encoders by presenting an open-source implementation that conforms to the HEVC scalability extension; specifically, the targeted scalability features are quality and spatial scalability. The proposed SHVC encoder has been built on top of Kvazaar — an open-source HEVC encoder — and is referred to as Scalable Kvazaar [18] with the latest version of its source code and issue tracker available on github [19]. The SHVC encoding functionality in Scalable Kvazaar has been optimized with the aim of achieving real-time encoding speeds. Finally, the performance of Scalable Kvazaar has been measured using common methods and metrics from the field of video compression. Moreover, the results are validated by comparing them to the respective SHM results. As a bonus, a more thorough examination of SHVC, made possible by Scalable Kvazaar, is carried out. To help measure the performance of the aforementioned encoders, a testing framework, for scalable encoding, is additionally presented.

Chapter 2 goes over the basic principles behind HEVC and SHVC. Chapter 3 presents the research and testing methodology used throughout the rest of this thesis. In Chapter 4, Scalable Kvazaar is introduced, including an overview of Kvazaar. In order to measure the performance of scalable encoding, a testing framework is presented in Chapter 5. Chapter 6 goes over the results of the various tests performed on Scalable Kvazaar and SHM. Finally, Chapter 7 concludes the paper and discusses possible future work.

2 HIGH EFFICIENCY VIDEO CODING (HEVC)

HEVC is a video coding standard for (lossy) video compression released as a twin text by ITU-T VCEG and the ISO/IEC MPEG standardization organizations [4]. The base standard [5] has been supplemented with various extensions [6], of which, SHVC [7] is the main focus for this thesis.

This chapter is divided into three parts. In Section 2.1, the base standard and its basic operating principles are introduced. Next, Section 2.2 goes over new coding tools introduced in SHVC. Finally, Section 2.3 briefly covers the reference implementations of HEVC and SHVC as well as their reported performance figures.

2.1 HEVC Standard

The HEVC standard employs a block-based hybrid video coding[1] scheme similarly to its predecessors [20]. HEVC is mainly designed for lossy video compression[2], but a lossless coding mode is still defined.

The high-level flow of HEVC encoding is depicted in Figure 2.1. The encoding process starts by dividing the input picture — format given in Section 2.1.1 — into equal sized blocks (see Section 2.1.2). Next, each block is passed to intra-/inter-picture prediction (defined in Section 2.1.3). A residual signal is calculated by taking the difference between the input picture and the result from the intra-/inter-picture prediction step. The residual is then passed to the transform and quantization step (described in Section 2.1.4). A reconstructed picture is generated using the inverse[3] operations of the previous steps. Some additional loop filter operations (Section 2.1.5) may be applied to the reconstructed picture before passing it to the *decoded picture buffer* (DPB), that holds reconstructed pictures for later use. Finally, information from the other steps and specified encoding parameters are used to generate the HEVC bitstream (described in Section 2.1.6 and 2.1.7).

[1] A video coding method that uses intra-/inter-picture prediction combined with 2-D transformation of the prediction residuals on uniform blocks of the input video.

[2] In lossy video coding, some information is discarded to achieve better compression, thus resulting in a loss of detail/quality in the reconstructed video.

[3] Here, inverse is used in the sense of the operation a decoder would perform on the results of the initial encoding operation.

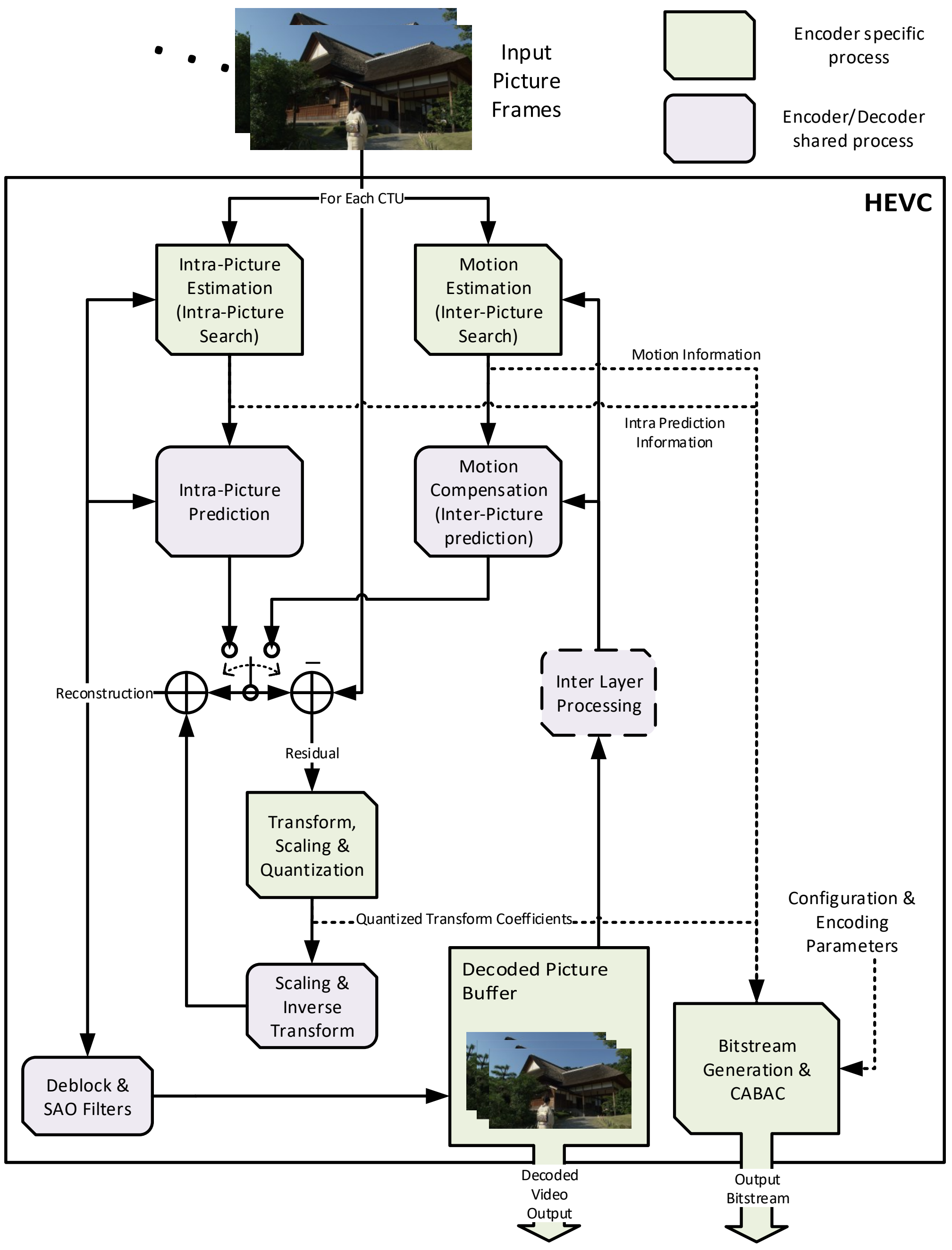

Input
Picture
Frames
Encoder specific process
Encoder/Decoder shared process
HEVC
For Each CTU
Intra-Picture Estimation (Intra-Picture Search)
Motion Estimation (Inter-Picture Search)
Motion Information
Intra Prediction Information
Intra-Picture Prediction
Motion Compensation (Inter-Picture prediction)
Reconstruction
Residual
Inter Layer Processing
Transform, Scaling & Quantization
Quantized Transform Coefficients
Configuration & Encoding Parameters
Decoded Picture Buffer
Scaling & Inverse Transform
Bitstream Generation & CABAC
Deblock & SAO Filters
Decoded Video Output
Output Bitstream

Figure 2.1. *Overview of the HEVC encoding process, including the SHVC specific inter-layer processing step.*

2.1.1 Picture Format

A picture is commonly represented by a $W \times H$ matrix of pixel values, with W representing the width of the picture and H the height. Moreover, a picture is usually made up of three color channels, each with their own matrices for the respective color's pixel values. A series of these pictures then forms a video sequence. In a picture, each pixel is represented by a fixed number of bits. This number is generally referred to as the bit depth of

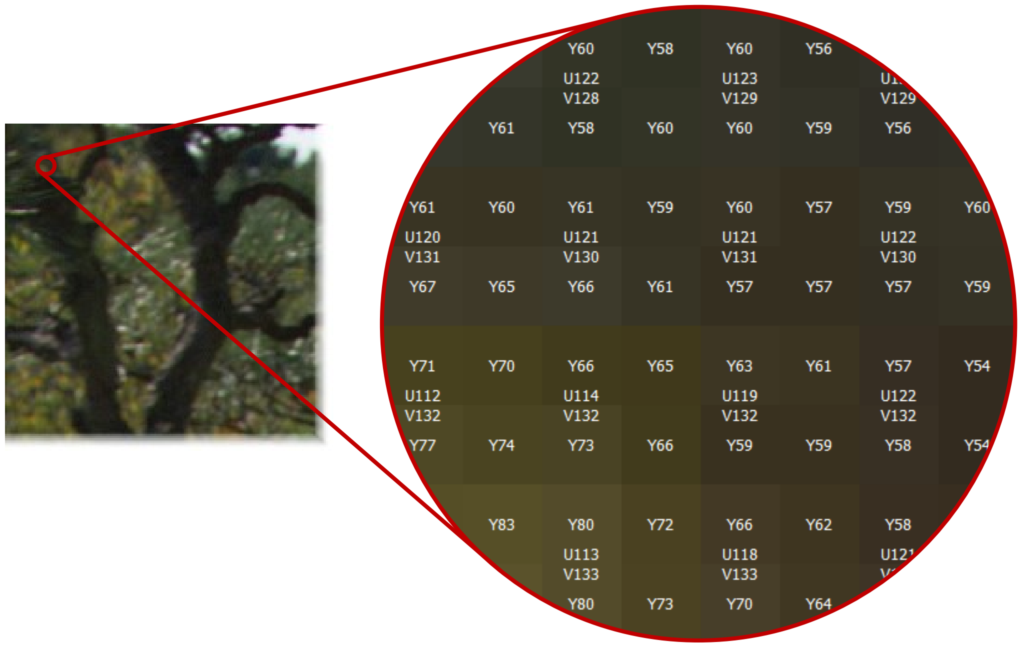

Y60 Y58 Y60 Y56
U122 V128 U123 V129 V129
Y61 Y58 Y60 Y60 Y59 Y56
Y61 Y60 Y61 Y59 Y60 Y57 Y59 Y60
U120 V131 U121 V130 U121 V131 U122 V130
Y67 Y65 Y66 Y61 Y57 Y57 Y57 Y59
Y71 Y70 Y66 Y65 Y63 Y61 Y57 Y54
U112 V132 U114 V132 U119 V132 U122 V132
Y77 Y74 Y73 Y66 Y59 Y59 Y58
Y83 Y80 Y72 Y66 Y62 Y58
U113 V133 U118 V133
Y80 Y73 Y70 Y64

Figure 2.2. *Visualization of a YUV image.*

the picture. In addition, pictures have an associated color space/gamut that determines how pixel values are mapped to the final colors.

Instead of the more common RGB based picture format, HEVC uses YCbCr — also referred to as YUV — as its input picture format [5]. YUV is made up of a luma (i.e., brightness) component (Y) and two chroma (i.e., color) components (U and V). The Y-component can be calculated from the weighted sum of the red, green, and blue color signals, whereas the U-/V-components are computed as the difference between the red/blue signal and the luminance signal [21].

As the human visual system is more sensitive to brightness than to color, HEVC opts for a 4:2:0 sampling structure by default [5]. With 4:2:0 sampling, each chroma component has one fourth of the samples compared to luma, i.e., the chroma resolution is halved in both the vertical and horizontal direction. Figure 2.2 shows a visualization of the 4:2:0 sampled YUV image.

2.1.2 Block Partitioning Structure

The basic processing unit in HEVC is the *coding tree unit* (CTU). Each input picture is divided into equal sized square blocks (i.e., CTUs) and each of the blocks are processed separately. HEVC supports CTU sizes of 16×16, 32×32, and 64×64 pixels; the CTU size is signaled in the bitstream. The CTU can recursively be divided into *coding*

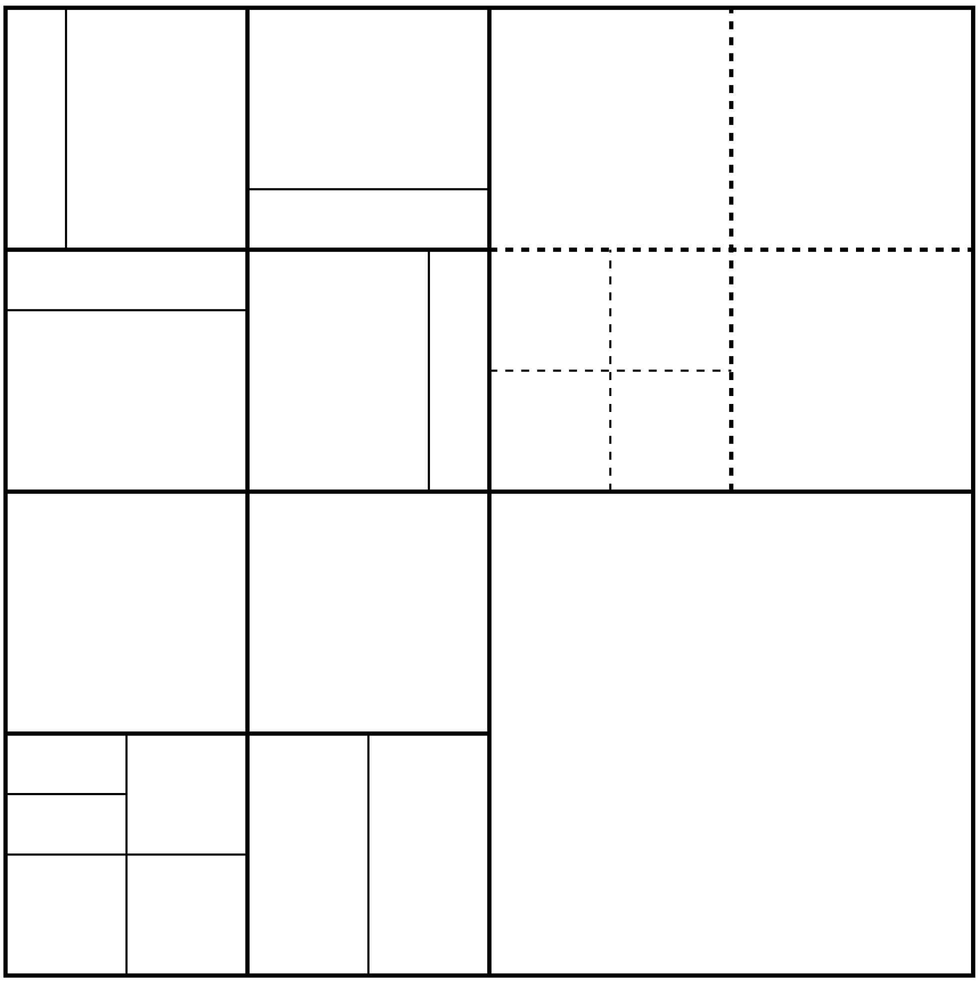


Figure 2.3. *CTU subdivision with transform splits (dashed lines) and examples of AMP (bottom left quadrant) and SMP (top left quadrant) blocks.*

units (CUs) to a minimum size of 8×8 pixels. The division into CUs follows a quadtree[4] structure. This division forms the block structure for the CTU. Each leaf CU is additionally used as the root for *prediction units* (PUs) (see Sections 2.1.3) and *transform units* (TUs) (see Section 2.1.4). An example of CTU division is shown in Figure 2.3. [5]

Moving to a higher level of abstraction, the CTUs of a coded picture can be collected into high-level structures called slices. Slices can contain a varying number of CTUs taken in raster scan[5] order. In other words, a coded picture can be made up of one or several slices. Each slice can be decoded independently from each other slice, but at the cost of coding efficiency. Nonetheless, slices can improve error resiliency, since other parts of the frame can still be decoded even if some slices are missing. Furthermore, network transmission usually imposes a maximum packet size that can be circumvented by dividing the frame into smaller slices. [5]

[4] A tree structure, where a parent node has four child nodes.

[5] A 2-D array or structure is processed by first iterating over all elements of a horizontal row before moving to the next row.

On top of slices, HEVC adds two special high-level structures — tiles and wavefront rows (associated with *wavefront parallel processing* (WPP) [22]) — that provide a slightly different kind of encapsulation, compared to slices. Tiles and WPP are directed towards parallel processing and generally result in worse compression. Tiles allow dividing the CTUs into larger rectangular pieces that can be encoded independently. WPP, on the other hand, allows encoding each CTU row separately, with the restriction that the top CTU and the top right diagonal CTU of the preceding CTU row need to be encoded before encoding can start on the new row. [5]

2.1.3 Intra and Inter Prediction

HEVC employs picture prediction to exploit the spatial and temporal redundancy of the input pictures. Picture prediction can be divided into intra-picture prediction[6] and inter-picture prediction[7]. In the CTU block structure, a leaf CU is signaled to use either intra or inter prediction, depending on the final coding mode decisions [5].

Intra prediction uses edge samples from already processed and decoded neighboring CUs. The standard defines a total of 35 modes for intra prediction. Out of the 35 intra modes, 33 are angular modes that allow copying the edge samples, from a specific direction (i.e., angle), to the block being predicted. The two remaining modes, on the other hand, allow assigning the average value of the edge samples to the whole block (DC mode) or the average of two surfaces, linearly interpolated using edge samples and a corner value (planar mode). Additionally, some smoothing operations and filters are applied in special cases to remove large discontinuities between blocks. [5]

Inter prediction allows copying pixel data from reconstructed pictures in the DPB. Each picture has two designated reference lists, containing reference pictures it may use for inter prediction; one list is generally used for pictures that come before the current one, in viewing order, and the other for pictures that come after it. Each inter predicted block has a motion vector, a reference picture index, and a reference list index associated with it [5]. The motion vector represents an offset from the current block's location to the block of pixels that should be copied from the reference picture. HEVC support fractional motion vectors, for increased accuracy, when trying to match blocks, where fractional samples, down to quarter sample positions, are generated using interpolation filters. The method for deriving motion information is left open for the encoder to decide, but usually some sort of search algorithm is used.

To minimize redundant information in motion vector signaling, HEVC introduces the merge mode and *advanced motion vector prediction* (AMVP). Merge mode allows deriving a block's motion information from neighboring blocks; the neighboring merge block is se-

[6] May also be referred to as intra-picture estimation and intra-picture prediction for the inverse process.
[7] May also be referred to as motion estimation and motion compensation for the inverse process.

lected from a set list of merge candidate, with five possible spatial candidates and two temporal candidates. If merge mode is not used, a motion vector predictor is chosen from the same set as the merge candidates. With AMVP, only the difference between the current block's motion vector and the motion vector predictor's motion vector is coded to the bitstream. [5]

The temporal candidates in merge mode or AMVP are a part of *temporal motion vector prediction* (TMVP) [7]. It may be used by enabling TMVP in the bitstream and setting one of the reference frames as the colocated TMVP reference picture that is then used to derive the temporal candidates. Motion vectors from the colocated TMVP reference picture may additionally need to be scaled, to account for the temporal differences, when used as the merge candidate or motion vector predictor. TMVP uses the motion information[8] of the reference picture in the candidate derivation process.

The chosen prediction mode is specified using a PU in the CTU block structure with leaf CUs used as roots. The size of an intra predicted PU should match the root CU size. However, if the size of the CU is 8×8 pixels, the PU may be split into four 4×4 blocks. For inter predicted blocks, several split modes are available; the root CU may be split into two PUs, either symmetrically (*symmetric motion partitioning* (SMP)) or asymmetrically (*asymmetric motion partitioning* (AMP)), as seen in the bottom left and top left quadrants of Figure 2.3. In addition to the SMP and AMP split modes, inter predicted blocks support the same split modes as intra predicted blocks. [5]

2.1.4 Transform and Quantization

The transform step takes in the residual, i.e., the error/difference, between the input picture and the resulting prediction step reconstruction and applies a 2-D transform on it. The transform uses approximations of scaled *discrete cosine transform* (DCT) [23] basis functions. Additionally, the transform matrix coefficients are specified as integer values [5]. To compress the transform results, quantization[9] is applied. The strength of the quantization is controlled using a *quantization parameter* (QP), that is allowed to range from 0 to 51 [5]. A higher QP generally results in better compression, as there are less distinct values to encode, but at the cost of quality, since detail is lost in the quantization.

The transform and quantization step can be performed on block sizes ranging from 32×32 to 4×4 pixels [5]. The block structure element used to represent transform and quantization is the TU. It uses leaf CU as the root in the CTU quadtree and may be further split until the minimum TU size of 4×4 pixels is reached. Figure 2.3 shows an

[8] Motion vectors, reference picture indices, and other relevant information, derived by inter prediction for each CU; can be referred to as the motion field.

[9] Quantization refers to the process of limiting the input values to an output domain, that is usually smaller than the input domain. Each input value is mapped to the closest value in the output domain.

example TU subdivision represented by the dashed line in the bottom right quadrant.

2.1.5 Loop Filtering

The HEVC standard defines two loop filters that can be applied to the final reconstructed picture before it is stored in the DPB. The usage of the filters is signalled in the bitstream and may be skipped entirely.

The first filter is the deblocking filter [24]. It aims to reduce the artefacts caused by the block-based coding approach, especially visible at block boundaries. Deblocking only needs to be applied on block borders, limiting the number of operations performed.

The second filter is *sample adaptive offset* (SAO) [25]. SAO is always applied after deblocking (if used) and, unlike deblocking, is done for all pixels. SAO further helps reconstruct the original signals using statistical data, calculated by the encoder and transmitted in the bitstream.

2.1.6 HEVC Bitstream

In the context of HEVC, bitstream is taken to mean the stream of data — output by the encoder — that represents the coded input video and contains all necessary information for a decoder to construct an approximation of the original input video. For the bitstream to be decodable, it needs to conform to the specification defined in the HEVC standard [4].

The basic building block of the bitstream is the *network abstraction layer unit* (NALU). All other bitstream syntax elements are wrapped in a NALU. The usage of NALUs enables bitstream manipulation, such as splicing and network transmission, without the need to decode the whole bitstream; only the NALUs need to be parsed. A NALU is made up of a NALU header and the actual payload in raw bytes, with some emulation prevention bytes interspersed, as necessary, to avoid miss-parsing the NALU. The header contains all relevant information about the payload data, such as the type of the payload. [4, 5]

Some of the more relevant syntax structure elements, that can be included in the NALU payload, are:

***Video parameter set* (VPS)** Includes information about the whole video encoding process, in general. The information is mostly metadata in nature and lays a foundation for including information used by various extensions [5, 6].

***Sequence parameter set* (SPS)** Includes basic information about the video sequence being encoded, such as the size and bit depth of the video.

***Picture parameter set* (PPS)** Includes information and encoding tool parameters for a single picture of a video sequence.

Slice segment layer Contains the actual encoding information (block structure, transform coefficients, etc.) for a given slice.

Other types of payloads are mostly for utility purposes or for including optional information.

One additional way HEVC tries to achieve better compression is *context-based adaptive binary arithmetic coding* (CABAC). It is used to compress various syntax elements in the bitstream. Through arithmetic coding[10] [26, 27], CABAC removes redundancies in the bit patterns of the bitstream, yielding a reduction in the final bitstream size. The standard defines various contexts, to be used by CABAC, for different situations to allow it to approximate the true distribution of the bits being encoded. The new contexts and careful selection of how syntax elements are signaled and encoded by CABAC, result in a better coding efficiency than the previous standard, despite still using the same core CABAC algorithm [5, 28].

2.1.7 Profiles, Tiers, and Levels

HEVC can be used in a large variety of applications and use cases. To provide various conformance points for encoders and decoders to target, the standard introduces the concept of profiles, tiers, and levels [6]. These concepts improve interoperability between applications, as decoders targeting certain capabilities are able to determine, based on the profile, tier, and level, if it can decode a given bitstream. The profile, tier, and level of a given HEVC bitstream is signaled in the VPS.

The profile defines a set of encoding tools that should be supported by a decoder conforming to the specified profile. The *'Main'* profile is the basic profile for 8 bit pixel depth and 4:2:0 chroma sampling, but profiles for other depths, chroma samplings, and use cases are also defined in the specification [4]. The tier is used to distinguish between normal and demanding applications; *'Main Tier'* for the former and *'High Tier'* for the latter. Finally, the level gives a fine grained selection, together with the tier, of certain bitstream properties. Table 2.1 shows the level and tier limits for the *'Main'* profile. The table shows the maximum supported picture sizes for each level and the bit rate limits etc. [6]

2.2 Scalable High Efficiency Video Coding (SHVC)

SHVC was added to the ITU-T/ISO/IEC recommendation [4] in 2015. It introduced several features to the base standard, including *signal-to-noise ratio* (SNR), spatial, color

[10] A technique for compressing data using the underlying propability distribution of the data, effectively allowing fractional bits to achieve a near optimal compression, i.e., aproaching the limit set by entropy of the data.

Table 2.1. *Conformance limits for the HEVC* 'Main' *profile [4, 6].*

	Max Luma Picture	Max Luma Sample	Bit Rate (kbits/s)		Min Comp. Ratio	
Level	Size (samples)	Rate (samples/s)	Main Tier	High Tier	Main Tier	High Tier
1	36864	552960	128	-	2	2
2	122880	3686400	1500	-	2	2
2.1	245760	7372800	3000	-	2	2
3	552960	16588800	6000	-	2	2
3.1	983040	33177600	10000	-	2	2
4	2228224	66846720	12000	30000	4	4
4.1	2228224	133693440	20000	50000	4	4
5	8912896	267386880	25000	100000	6	4
5.1	8912896	534773760	40000	160000	8	4
5.2	8912896	1069547520	60000	240000	8	4
6	35651584	1069547520	60000	240000	8	4
6.1	35651584	2139095040	120000	480000	8	4
6.2	35651584	4278190080	240000	800000	6	4

gamut, bit depth, and hybrid codec scalability. An additional scalability feature, temporal scalability, was already included in the base standard but is covered in more detail in Section 2.2.5. The main idea behind SHVC is creating adaptable bitstreams that incorporate several different quality levels. Efficient coding is achieved by exploiting inter-layer redundancy. The different scalability features define the video parameters that can be changed between layers. The lowest quality layer is usually referred to as the BL, and higher layers are called ELs.

Section 2.2.1 first provides some motivation and use cases for SHVC. Section 2.2.2 introduces new concepts and functionality, introduced in SHVC, that are shared by the different types of scalability. Next, Section 2.2.3 goes over SNR scalability and Section 2.2.4 goes over spatial scalability — the two main focuses in this thesis. The other scalability features are described in Section 2.2.5. Finally, Section 2.2.6 goes over the main changes to the bitstream, brought on by SHVC.

2.2.1 Motivation for SHVC Coding

Cisco's annual reports [1] indicate that video traffic will be 82% of all internet traffic by 2022, with an associated four-fold increase in the total video traffic between 2017 and 2022. Furthermore, the diversity and amount of mobile devices is predicted to grow steadily at a rate of 6% per year [3]. Finally, the viewing resolutions of different devices will keep increasing; in the case of connected TVs, two-thirds are estimated to support UHD by 2023 [3]. Cisco's reports suggest that the popularity of video services and conferencing will keep increasing, but they will need to support a variety of devices and viewing resolutions. SHVC is one tool that can be used, in various applications, to efficiently provide adaptability, as described below.

Ye et al. [8] have discussed the usage of SHVC in the context of video streaming and VOD services. The popular method for providing adaptive video streams is to divide each video into small segments with a separate copy of each segment for each desired resolution and bit rate; the requested quality copy is then sent to the client. Using SHVC, instead, would allow storing video streams, of different qualities, much more efficiently. Furthermore, SHVC yields itself well to streaming services that use *content delivery networks* (CDNs) or edge servers with hierarchical caching. For example, only the BL can be stored in the first-level edge server (i.e., closest to the client), allowing for quick retrieval when starting the stream. Higher quality ELs can be stored deeper in the cache hierarchy, and thanks to bit rate reductions from SHVC, they require less bandwidth to transfer to the client than the complete high-quality video would.

Nightingale et al. [10] have proposed an SHVC-based video stream adaptation scheme. In their work, they have demonstrated that SHVC-based video streams are able to reduce their bandwidth, when necessary, more and at a lower quality penalty than HEVC streams in wireless network conditions. This is very promising for streaming SHVC in changing network conditions and may even provide better error resiliency, since the lower quality layers are still playable even if the higher bandwidth layers fail to transmit.

Ronan et al. [9] have demonstrated, how SHVC could be used in a broadcasting scenario. The SHVC encoded bitstream is broadcast and received by different devices, that can then decode the desired layers, depending on their capabilities. Even devices, without SHVC support, can at least decode the BL, since it is HEVC compliant. SHVC could even be used to include personalized content through the EL. Moreover, Lee et al. [29] have gone into more detail on using SHVC in a broadcasting scenario and have performed rigorous testing in various test conditions. They conclude that SHVC provides better results in all tested scenarios when compared with HEVC in a simulcast configuration, thus demonstrating the feasibility of SHVC in a broadcasting scenario.

Xu et al. [30] have suggested that scalable coding could be used to reduce overhead, in video conferencing, when different versions of the video stream are needed. If transcoding is not performed on the server, used to relay video to participants, the sender needs to generate the different versions and send them to the server. In this case, the upload bandwidth can be reduced using scalable coding, instead of separately encoding and sending the different video streams.

2.2.2 Scalability

The scalability extension was designed to require only high-level syntax changes so minimal modifications are required in the encoding flow [7]. The main addition to the process, described in Section 2.1, is an inter-layer processing step as seen in Figure 2.1 (dashed line box).

SHVC bitstreams contain multiple layers; one for each (different quality) video stream. Each layer has an associated ID (*layer ID* (LID)), ordered based on increasing quality. The BL is always associated with a LID of zero and should be decodable by non-SHVC decoders. Subsequent higher layers, with non-zero LIDs, are ignored by HEVC decoders, making SHVC streams at least partly decodable by non-SHVC decoders [4].

The main method SHVC achieves its coding gain is by deriving extra ILR pictures for use in inter prediction. ELs can use lower LID layers as *reference layers* (RLs), but not higher ones [7]; each layer needs to be decodable without higher LID layers to allow selectively transmitting only some lower layers [4]. For a given EL, RLs are then used to derive the ILR pictures, i.e., pictures with a matching *picture order count* (POC), but a smaller LID. Additionally, it should be noted that ILR pictures are defined as long-term reference pictures, affecting how they are treated in some cases [4].

Two types of inter prediction tools are specified for ILR pictures in the scalability extension: texture and motion prediction [7]. Texture prediction is equivalent to normal inter prediction, but motion search is omitted and a zero motion vector is used, as specified in the standard [4]. Motion prediction, on the other hand, uses motion information from the RL for TMVP, when enabled, and an ILR picture is set as the collocated TMVP reference picture.

2.2.3 SNR Scalability

With SNR scalability, each layer represents a different video quality, usually achieved by using different QPs for each layer. Using rate-control could be another way for controlling quality between layers, especially in a video streaming setting, where constant bit rates for different bandwidth requirements are usually desired.

SNR scalability generally does not require extra inter-layer processing, since it only involves changing the QP between layers. In other words, both the reconstruction and the motion field of the RL can be directly used for inter prediction. SNR scalability can, however, be combined with other types of scalability, in which case inter-layer processing may be necessary.

2.2.4 Spatial Scalability

Spatial scalability uses different resolutions for different layers, thus making it impossible to directly use texture information from the RL; texture resampling needs to be performed in the inter-layer processing step. The resampling process in SHVC is defined for arbitrary resolution ratios, whereas the previous standard only supported a set number of ratios. [7]

SHVC uses 8-tap (4-taps for chroma) interpolation filters to generate the upscaled picture used for ILR. The colocated sample position in the RL is calculated using sub-pixel precision, allowing for a good upsampling quality even with irregular scaling ratios. On top of the filters for upsampling, SHM provides 16-tap interpolation filters for downsampling purposes. The full table of interpolation filters for the resampling process can be found in Appendix A.

Algorithm 2.1 gives the outline of texture resampling. It can be divided into a horizontal step (lines 1 through 11) and a vertical step (lines 12 through 27). The horizontal step calculates filtered values into a temporary picture buffer for each pixel in a destination-width-by-source-height area. Filtering is done for each source row and a collocated column is used to determine the sample positions in the source picture buffer row. The high-precision reference position (i.e., collocated column) is calculated on line 3. It is then used to determine the phase (line 4), that is used when choosing the correct filter coefficients, and the actual collocated full-pixel position (line 3). Next, on lines 6 through 9, the algorithm applies a filter on the source picture samples, iterating over each coefficient and accumulating the values into a temporary picture buffer (line 8). Line 7 shows how the source picture sample position is calculated for each filter coefficient, using the collocated column, from line 5, as the middle point.

The vertical step is similar to the horizontal step, but using values from the temporary buffer, it calculates the final pixel values for a destination-width-by-destination-height area. Moreover, instead of iterating over picture rows and calculating collocated column positions, the vertical step iterates over picture columns and calculates the collocated rows. Additionally, a normalization step is added, shown on lines 21 through 25, to scale and clip the filtered values to the correct 8-bit pixel range. The normalization step is defined for two filter sizes; one for upsampling and the other for downsampling filters.

For motion information, as with texture information, additional motion field resampling needs to be performed when spatial scalability is used. However, RL motion information is only used if TMVP is enabled and the ILR picture is set as the colocated picture, thus allowing motion field resampling to be skipped if TMVP is not used. To save on motion field memory requirements, motion information is only stored for 16×16 blocks; the motion field resampling is done with the same granularity [7]. Once the colocated block in the RL motion field is determined, the motion information is copied to the EL motion field. Furthermore, the motion vector needs to be scaled based on the spatial ratios of the RL and EL.

Algorithm 2.2 details the procedure for motion field resampling. As mentioned above, it is only necessary to generate the upsampled motion field for CU blocks of size 16×16.

Algorithm 2.1: Basic algorithm for texture resampling.

Data: Source texture data in *src*; Filter coefficients in *filter*; The vertical and horizontal scaling ratios, shifted left by 16 bits, in *scale_x* and *scale_y*; The rounding values in *add_x* and *add_y*; The sampling phase offsets in *delta_x* and *delta_y*

Result: Resampled texture data in *dst*

```
// Horizontal step. Loop over each row in src, calculating collocated positions for each dst column
1  for $y \leftarrow 0$ to Height(src)$-1$ do
2      for $x \leftarrow 0$ to Width(dst)$-1$ do
           // Calculate the filter phase and the collocated position
3          ref_pos_16 $\leftarrow$ ShiftRight($x \cdot$ scale_x $+$ add_x, $12$)$-$delta_x;
4          phase $\leftarrow$ BitwiseAnd(ref_pos_16, $15$);
5          ref_pos $\leftarrow$ ShiftRight(ref_pos_16, $4$);
           // Apply filter to samples from src
6          for $i \leftarrow 0$ to Height(filter)$-1$ do
               // Calculate the sample position in src for the respectife filter coefficient
7              $x' \leftarrow$ Clip(ref_pos $+ i -$ShiftRight(Height(filter), $1$)$+1$, $0$, Width(src)$-1$);
8              tmp$[x, y] \leftarrow$ tmp$[x, y]+$filter[phase, $i$]$\cdot$src$[x', y]$;
9          end
10     end
11 end
// Vertical step. Loop over each column in dst (tmp), calculating collocated positions for each src (tmp) row
12 for $x \leftarrow 0$ to Width(dst)$-1$ do
13     for $y \leftarrow 0$ to Height(dst)$-1$ do
           // Calculate the filter phase and the collocated position
14         ref_pos_16 $\leftarrow$ ShiftRight($y \cdot$ scale_y $+$ add_y, $12$)$-$delta_y;
15         phase $\leftarrow$ BitwiseAnd(ref_pos_16, $15$);
16         ref_pos $\leftarrow$ ShiftRight(ref_pos_16, $4$);
17         for $i \leftarrow 0$ to Height(filter)$-1$ do
               // Calculate the sample position in src (tmp) for the respectife filter coefficient
18             $y' \leftarrow$ Clip(ref_pos $+ i -$ShiftRight(Height(filter), $1$)$+1$, $0$, Height(src)$-1$);
19             dst$[x, y] \leftarrow$ tmp$[x, y]+$filter[phase, $i$] $\cdot$ tmp$[x, y']$;
20         end
           // Normalize and clip final values to 8-bit pixel range depending on the number of filter coefficients
21         if Height(filter) $\leq 8$ then If upsampling filter is used
22             dst$[x, y]\leftarrow$ Clip(ShiftRight(dst$[x, y]+2048$, $12$), $0$, $255$);
23         else If downsampling filter is used
24             dst$[x, y]\leftarrow$ Clip(ShiftRight(dst$[x, y]+8192$, $14$), $0$, $255$);
25         end
26     end
27 end
```

Algorithm 2.2: Basic algorithm for motion field resampling.

Data: Source and destination motion fields in *src_mf* and *dst_mf*; The vertical and horizontal scaling ratios, shifted left by 16 bits, in *pos_scales*; The vertical and horizontal scaling ratios for MV scaling, shifted left by 8 bits, in *mv_scales*
Result: Resampled motion data in *dst_mf*

```
   // Loop over each $16 \times 16$ block in each dst_mf CTU
1  foreach CTU block ctu in dst_mf do
2      foreach $16 \times 16$ block cu in ctu do
           // Calculate the collocated CU in src_mf, based on the middle
              point of cu
3          col_x $\leftarrow$ ShiftLeft(ShiftRight(ScalePos(cu.pos_x +
            ShiftRight(cu.width, 1), pos_scales[0]) + 4, 4), 4);
4          col_y $\leftarrow$ ShiftLeft(ShiftRight(ScalePos(cu.pos_y +
            ShiftRight(cu.height, 1), pos_scales[1]) + 4, 4), 4);
5          col_cu $\leftarrow$ GetCuAt(src_mf, col_x, col_y);     // Return CU if inside
            frame
6          if col_cu and col_cu.type = CU_INTER then
               // Collocated CU is inside the frame and is an inter CU
               // Scale MVs from collocated CU and copy motion data to
                  dst_mf
7              cu.MVs $\leftarrow$ ScaleMVs(col_cu.MVs, mv_scales);
8              cu.MV_ref_ind $\leftarrow$ col_cu.MV_ref_ind;
9              cu.MV_ref_list $\leftarrow$ col_cu.MV_ref_list;
10         else
               // Collocated CU is outside of the frame or is an intra
                  CU
11             cu.type $\leftarrow$ CU_INTRA;
12         end
13         cu.part_size $\leftarrow$ $2N \times 2N$;
14     end
15 end
```

As a result, upsampling is performed by iterating over each EL CTU separately. Each CTU is further divided into to the aforementioned block size, and the upsampled motion data is calculated for the block. As shown on lines 3 and 4, the collocated position in the RL motion field is calculated from the center point of the current EL block, based on the given scaling ratio. Once the collocated CU has been found, the algorithm checks that it is located inside the RL picture and that its prediction type is inter (line 6); if not, the current block is marked as an intra block. When a valid collocated CU has been found, motion vectors are scaled, and other information is copied to the current block.

2.2.5 Other Scalability Features

Color gamut scalability helps fit together videos with different color spaces. A color mapping step is used to improve the coding efficiency and is always applied to the RL first, before other inter-layer processing steps. The color mapping process uses a 3D *lookup table* (LUT) based approach. Moreover, it is shown to improve coding efficiency by around 10.5% when compared with an encoding that is not using color mapping. [7]

Bit depth scalability allows layers to have different bit depths [7]. Depending on the encoders/decoders internal bit depth, no action needs to be taken, as long as the output bit depths are set correctly. When using spatial scalability, on the other hand, the bit depth difference is taken care of by the final normalization step of the resampling process [7].

Hybrid codec scalability enables using an externally provided non-HEVC BL to maintain backwards compatibility with older content without needing to re-encode. The externally provided bitstream may even be HEVC compliant if an existing encoding needs to be supplemented. With non-HEVC bitstreams, an external decoder is needed for the BL and coding information is limited to the reconstructed pictures provided by the external decoder. [7]

Temporal scalability allows varying the frame rate by only decoding lower temporal layers and dropping the higher temporal layers. The standard prohibits frames from referencing higher temporal layers so that a temporal layer is decodable without any higher temporal layers [7]. For the bitstream to have a valid temporal structure, the reference structure (i.e., *group of pictures* (GOP)) needs to be carefully chosen.

2.2.6 SHVC Bitstream

To signal the layer a NALU belongs to, a LID field is added to the NALU header [7]. In addition to LIDs, the NALU header contains a field for a *temporal ID* (TID) that is for temporal scalability. This allows easily dropping higher layer NALUs if only a lower layer is needed. The NALU LID field was already specified for HEVC so even a non-SHVC decoder can decode SHVC bitstreams; NALUs with LIDs other than zero are simply ignored [4].

The VPS includes an SHVC extension field that can be used to specify information about all layers. It contains the number of layers, their dependencies, as well as the size, depth, and chroma format of each layer. Additionally, the VPS is used to specify the type of scalability being used. The PPS, on the other hand, can be used to specify inter-layer prediction related information such as the color mapping tables used by color gamut scalability.

The extension field in the VPS is used to define the profile, tier, and level for each EL. The tier and level specifications remain largely unchanged for SHVC, but new profiles are defined to signify SHVC capabilities. The new profile, equivalent to the BL *'Main'* profile, is the *'Scalable Main'* profile.

2.3 HEVC and SHVC Test Models

At the time of writing, no practical open-source SHVC encoders existed, bar Scalable Kvazaar [18]. Parois et al. [16] have presented a SHVC encoder based on a proprietary HEVC encoder that is not available to the public. Both HEVC and SHVC have a reference implementation associated with them; the *HEVC Test Model* (HM) [31] and SHM [15], respectively. They implement all the features defined in the standards, for both the encoder and decoder side. Moreover, they represent the best the standards can offer, in terms of coding efficiency, and can be used as the baseline for testing new features and algorithms, as well as for research and conformance testing. However, the coding efficiency comes at the cost of encoding time, making them unusable for practical applications.

Ohm et al. [32] have presented a comparison between the previous H.260/MPEG family video coding standards and HEVC. The reported objective bit rate savings of HM over the previous standard (h.264/MPEG-4) were between 35.4% and 40.3% depending on the type of video content. Vanne et al. [33] have reported similar results, with a low-delay P configuration — used later in this thesis for performance measurements — providing a 35% improvement. Furthermore, Tan et al. [34] have shown, through human subjective testing, that the subjective coding efficiency improvements are as high as 64%.

Boyce et al. [7] have compared SHM coding efficiency to the equivalent simulcast HM coding. In SNR low-delay scalability coding, they reported SHM to reduce the bit rate of a two-layer video stream by 12.5% over an equivalent simulcast coding case. The respective reduction is 19.5% when only the EL is compared with the simulcast equivalent. For the spatial scalability case, bit rate is reduced by 10.3% and 21.5% for the spatial ratios of $2\times$ and $1.5\times$ respectively with a reduction of 16.8% and 39.3% when only comparing the ELs. On the other hand, moving from a single high quality HEVC stream to a respective two-layer SHVC stream is shown to increase bit rate by around 25.8% with SHM.

3 RESEARCH AND TEST METHODOLOGIES

The three main questions this thesis seeks to answer are: 1) will scalability features provide notable bit rate improvements in a practical encoder; 2) what kind of an impact does scalability have on encoding complexity; and 3) is it possible to perform SHVC encoding in real-time? To this end, this chapter presents the methods used to answer these questions. Section 3.1 goes over the conventions and principles followed during the development of Scalable Kvazaar. Section 3.2 goes into more detail about the test methods and arrangements used to evaluate the performance of Scalable Kvazaar.

3.1 Research Methods

Research, conducted for this thesis, follows the design science methodology. The creation of a practical SHVC encoder is guided by the aforementioned research questions and other objectives set for the resulting encoder. The development process consists of a series of iterative design and implementation steps, carried out in a structured manner, while guided by a rational decision-making process. Finally, the created encoder is continuously evaluated and benchmarked, as detailed below.

The development of Scalable Kvazaar seeks to first establish a functioning product that conforms to the SHVC standard. To confirm this, Scalable Kvazaar is validated against the SHM decoder, ensuring that the generated bitstream is correct. Moreover, continuous integration test are added to track the correctness of the program itself. Next, the encoding speed of Scalable Kvazaar is evaluated and analyzed using well-established software tools. Based on the analysis, the encoding process is optimized until the designated speed goal is reached. Finally, the performance of Scalable Kvazaar is measured using commonly used video coding metrics (described in Section 3.2.1) to demonstrate that it reaches the real-time encoding goal, while maintaining an acceptable level of improvement to the bit rate.

Scalable Kvazaar is developed to address the lack of practical open-source SHVC encoders. Since SHVC allows re-using most of the functionality from HEVC coding, an existing HEVC encoder — Kvazaar — was chosen as the base for Scalable Kvazaar. Furthermore, the design of Scalable Kvazaar aims to make minimal modifications to the core Kvazaar functionality and to minimize the impact of any modifications to the encoding speed of the HEVC side of the encoder. Minimizing the modifications to the core Kvazaar makes maintaining Scalable Kvazaar easier, since any future changes to Kvazaar are less likely to conflict with Scalable Kvazaar modifications, saving time and effort when integrating new improvements from the main Kvazaar branch.

3.2 Test Methods

This section presents common methods for evaluating objective video encoding performance. In addition, the test platform, test parameters, and test sequences are listed. Section 3.2.1 goes over the common performance metrics. Section 3.2.2 presents both the hardware platform and the software framework that are used for running performance tests on the encoders under evaluation. Finally, Section 3.2.3 introduces the test sequences, used as the test set for performance evaluation, in addition to giving the encoding parameters for Scalable Kvazaar and SHM.

3.2.1 Performance Metrics

The main trade-offs in video coding are between bit rate[1], coding quality (i.e, distortion), and coding complexity (i.e., time). Using more bits tends to result in better quality, but at the cost of producing larger bitstreams, whereas spending more time usually results in better compression. Coding efficiency evaluation can usually be divided into objective and subjective quality evaluations. Subjective quality evaluations rely on human testing and are more tedious to perform; thus, this thesis focuses on objective quality measures.

A widely used objective metric for measuring the distortion in video coding is *peak signal-to-noise ratio* (PSNR). For a given picture, PSNR is calculated using the *mean squared error* (MSE) between the input picture and its reconstruction, as seen in (3.1) and (3.2).

$$\mathrm{MSE}\left(P, P'\right) = \frac{1}{H \cdot W} \sum_{y=0}^{H-1} \sum_{x=0}^{W-1} \left(p_{xy} - p'_{xy}\right)^2 \tag{3.1}$$

$$\mathrm{PSNR}\left(P, P'\right) = 10 \cdot \log_{10} \left(\frac{2^B - 1}{\mathrm{MSE}\left(P, P'\right)} \right) \tag{3.2}$$

The input to PSNR is the matrix representation of a picture's pixel values; P is the original picture and P' is the respective reconstructed picture. The size of the pictures are given by W for width and H for height. The individual pixel values, in position (x, y), are given by p_{xy} and p'_{xy}, respectively. B is the bit depth of the picture and is used to calculate the maximum possible pixel value.

PSNR is calculated separately for each color component (i.e, Y, U, and V). Generally, a

[1] Amount of bits per unit of time or per frame etc.

weighted average is used to calculate the combined PSNR [32] as

$$\mathrm{PSNR_{YUV}}\left(P, P'\right) = \frac{6 \cdot \mathrm{PSNR}\left(P_{\mathrm{Y}}, P'_{\mathrm{Y}}\right) + \mathrm{PSNR}\left(P_{\mathrm{U}}, P'_{\mathrm{U}}\right) + \mathrm{PSNR}\left(P_{\mathrm{V}}, P'_{\mathrm{V}}\right)}{8}.$$

$\mathrm{PSNR_{YUV}}$ gives the weighted average of the three components, where luma channels are given by P_{Y} and P'_{Y}. Similarly, chroma channels are given by P_{U}, P'_{U}, P_{V}, and P'_{V}.

Comparing the coding efficiency of two encodings, numerically, is not a trivial task, as there are two quantities (bit rate and distorition) that vary. To address this difficulty, Bjøntegaard [35] proposes a method for calculating a BD-rate for quantifying the coding efficiency difference between two encodings. The BD-rate calculation involves calculating a number of data points[2] with varying bit rate or quality. A curve is fit through the data points and, to calculate the BD-rate, the difference (i.e., area) between the two curves is calculated using integration. However, in some cases, the aforementioned method may produce misleading results. This can be mitigated by using piece-wise cubic interpolation [36]. Generally, a negative BD-rate value can be thought of as an equal reduction in bit rate, with similar quality, when compared with the anchor.

PSNR is still the de-facto standard for distortion metrics in video encoding, but it does not necessarily correlate well with visual quality [37, 38, 39]. Many alternative metrics have been proposed [38, 40, 41], but the most prominent and widely spread one seems to be SSIM [42]. It takes advantage of structural information in an image to predict perceived quality, instead of the more direct signal error approach used by PSNR. The major downside to using SSIM is the added complexity of the calculation, making it harder to use as an optimization metric, e.g, in real-time encoding. In this work, PSNR is used together with BD-rate to evaluate coding efficiency.

When talking about coding performance, usually only the BD-rate is of interest. However, in some applications, such as real-time encoding, the coding complexity is also relevant. It can be measured by simply recording the time it takes to encode a given input sequence. Moreover, if the amount of frames, that were encoded, is known, the frame rate in *frames per second* (FPS) can be calculated by dividing the number of frames by the encoding time. This can help fine-tune the encoding process to meet potential real-time constraints.

3.2.2 Test Platform

The test platform, used for running the performance tests, was an 8-core Intel® Xeon® W-2145 CPU with 32 GB of RAM (DDR4 2666 ECC), running a 64-bit Microsoft Windows 10. The CPU has eight physical cores, supporting sixteen logical cores that can be used by parallel processing tools. The CPU also supports the AVX2 instruction set extension, allowing the use of the optimization detailed in Section 4.2.

[2] Four data points are generally used.

Table 3.1. *Test sequence information.*

Sequence name	Frame count	FPS	Resolution			
			BL $2\times$	BL $1.5\times$	BL SNR	EL
Traffic	150	30	1280x800	-	2560x1600	2560x1600
PeopleOnStreet	150	30	1280x800	-	2560x1600	2560x1600
Kimono	240	24	960x540	1280x720	1920x1080	1920x1080
ParkScene	240	24	960x540	1280x720	1920x1080	1920x1080
Cactus	500	50	960x540	1280x720	1920x1080	1920x1080
BasketballDrive	500	50	960x540	1280x720	1920x1080	1920x1080
BQTerrace	600	60	960x540	1280x720	1920x1080	1920x1080

All tests were run using the NAVETTA test framework described in Chapter 5. The complete test script (`DI_tests.py`) can be found in the NAVETTA github repository [43]b under the `tests`-folder. It contains code to generate both the Scalable Kvazaar and SHM tests and summary definitions that were used to derive the final result tables.

3.2.3 Test Parameters

Tests were performed for three different scenarios, using two-layer scalable encoding, with varying scalability types and parameters. The scenarios are SNR scalability, $2\times$ spatial scalability, and $1.5\times$ spatial scalability. Additionally, each scenario uses two different ΔQP values, as detailed below, to cover a wider range of test cases, all in line with the common SHM test conditions [44].

Altogether, seven full-length 8-bit YUV420 video sequences were taken from the common SHM test conditions. Table 3.1 tabulates the used sequences, the number of frames, and their frame rate (i.e, FPS). Additionally, the resolutions used for the BL and EL are shown. The first two resolution columns show the BL resolutions for the spatial scalability ratios of $2\times$ and $1.5\times$, respectively. The next column shows the BL resolution for SNR scalability, and the last column shows the EL resolutions used in all tests. In the spatial scalability comparisons, the input sequences were pre-scaled to the correct resolution according to the used scaling ratio ($2\times$ or $1.5\times$). Finally, it should be noted that the 1600p sequences are omitted from the $1.5\times$ spatial ratio tests, since the resulting BL resolutions would not be valid input resolutions in HEVC.

The QP ranges, used to calculate the BD-rate, are shown in Table 3.2. The BL QP column shows the base QP used in the BL. The EL column shows the tested ΔQP values, applied to the base QPs, when calculating the respective EL QP. The QP ranges and ΔQP values match those defined in the common SHM test conditions.

The coding efficiency and speed of Scalable Kvazaar v1.0.1 were compared with Kvazaar

Table 3.2. *QP ranges and ΔQP values used in tests.*

Scalability type	BL QP	EL ΔQP
SNR	26, 30, 34, 38	-6, -4
Spatial ($2\times$ and $1.5\times$)	22, 26, 30, 34	0, 2

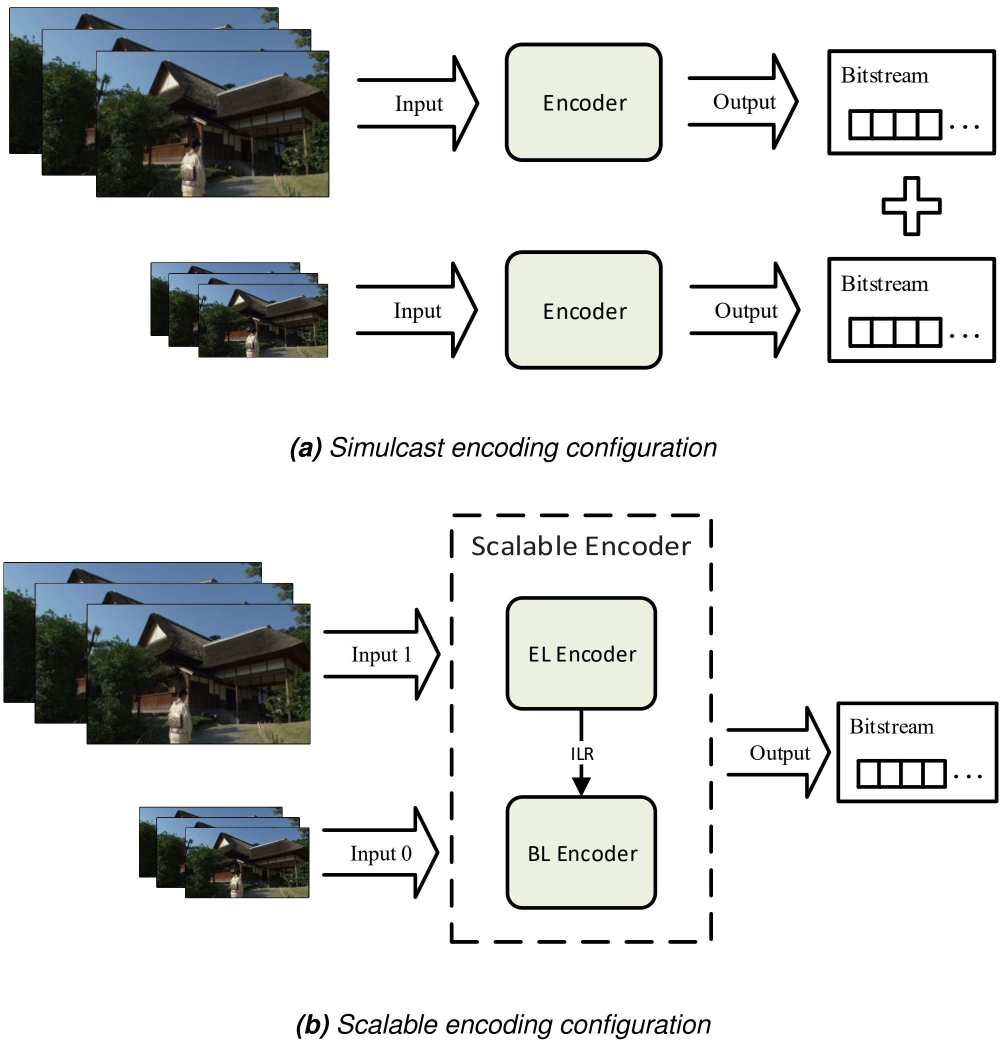


Figure 3.1. *Examples of a high-level simulcast encoding configuration and a scalable encoding configuration when using spatial scalability.*

in a simulcast configuration by encoding two-layer (BL + EL) test videos under a low-delay P coding configuration with an intra frame period of 64. The respective BL and EL encoding results of Kvazaar simulcast coding were aggregated to attain a fair comparison with the scalable encoding case. Figure 3.1 shows the difference between scalable encoding and simulcast coding.

Table 3.3 lists the parameters that were used to run the simulcast coding (both the BL and EL), spatial scalability, and SNR scalability tests. Moreover, Table 3.4 shows the most relevant coding tools and parameters that are used by the ultrafast preset of (Scalable)

Table 3.3. *Test command-line parameters used for simulcast and Scalable Kvazaar.*

Encoding type	Parameters
Simulcast	`--input=<layer sequence> --preset=ultrafast --threads=15 --owf=2 -q <layer QP>`
Scalable	`--input=<BL sequence> --preset=ultrafast -q <BL QP> --layer --input=<EL sequence> --preset=ultrafast --threads=15 --owf=2 -q <EL QP>`

Table 3.4. *Parameters of the ultrafast preset.*

Parameter	Value
Coding unit sizes	64×64, 32×32, 16×16, 8×8
Intra prediction unit sizes	16×16, 8×8
Intra prediction modes	35 (DC, planar, 33 angular)
Inter prediction unit sizes	16×16, 8×8
Motion estimation algorithm	HEXBS
Transform unit sizes	32×32, 16×16, 8×8
Mode decision metrics	SAD, SATD, SSD, CABAC
GOP structure	low-delay P
Intra frame period	64
Reference pictures	1
Temporal MV prediction	Enabled
Fractional motion estimation	Enabled
Loop filters	Deblocking
Parallelization	WPP, OWF

Kvazaar. The sizes of prediction units are limited, and advanced coding tools are disabled to attain real-time encoding speeds.

The respective tests were also conducted with SHM12.1 and HM16.10 for the sake of validating the Scalable Kvazaar tests. For SHM and HM, low-delay P and sequence specific configuration files were used, as defined in the common SHM test conditions, with QP values set manually on the command-line.

4 DESIGN AND IMPLEMENTATION OF SCALABLE KVAZAAR ENCODER

Kvazaar is an open-source HEVC encoder [45] developed by Ultra Video Group [46] — a research group at Tampere University. Scalable Kvazaar [19], as the name implies, is an SHVC encoder, implemented on top of the Kvazaar encoder.

Section 4.1 first introduces the basic structure and operation of Kvazaar relating to the implementation of the scalability extension. Section 4.2 details the modifications made to the base encoder, as well as optimizations that are used to achieve real-time encoding speeds.

4.1 Kvazaar HEVC Encoder

Kvazaar is mainly written in C[1] and incorporates many parallelism tools and optimizations to achieve practical encoding speeds [47, 48, 49, 50, 51]. Moreover, it is able to meet the requirements of real-time encoding, even when using 4K resolutions [52]. The high-level architecture of Kvazaar is depicted in Figure 4.1 and the simplified encoding flow in Figure 4.2.

First, Section 4.1.1 presents the basic architecture of the Kvazaar encoder. Next, Section 4.1.2 goes over the simplified encoding flow, as it relates to the modifications made by Scalable Kvazaar that are presented later. Finally, Section 4.1.3 introduces the parallelism tools that are used later to help accelerate Scalable Kvazaar.

4.1.1 Overall Architecture

Kvazaar and its parts are shown in Figure 4.1. They are grouped into categories roughly based on their function. Most of the functionality is located in the Kvazaar library, that can be included by external applications. Additionally, a *command-line interface* (CLI) is provided for stand-alone encoding. Kvazaar CLI takes care of parsing command-line options, reading the input frames, and writing the output to file.

The Kvazaar library is used through the *application programming interface* (API) defined

[1] An imperative programming language.

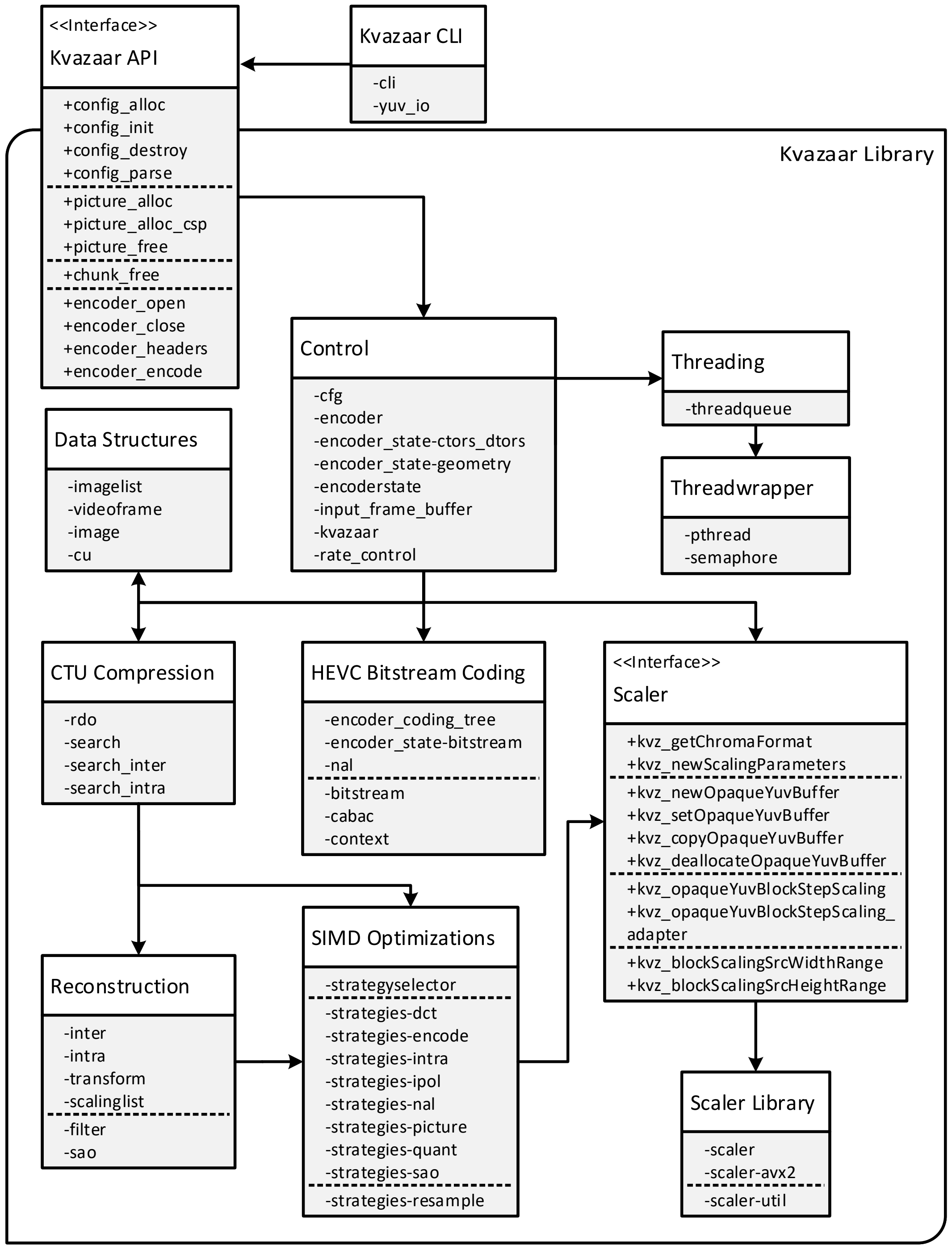


Figure 4.1. *Shared Kvazaar and Scalable Kvazaar architecture.*

in `kvazaar`. The API defines data structures for configuring the encoding process, storing picture and bitstream data, and the necessary functions for controlling the encoding process. The API can roughly be divided into functions for generating the Kvazaar configuration, managing picture data, freeing bitstream data chunks, managing the encoder, and driving the encoding process.

Most of the functionality that manages the encoding process is collected under *'Control'*. The high-level encoding flow is defined in `kvazaar`. For storing information about the whole encoding process, `encoder` defines the `encoder_control`-structure. On the other hand, the `encoder_state`-structure contains the state of a single picture being encoded.

The other important data structures are defined under *'Data Structures'*, including image and CU data types. Of special interest to scalable coding, as discussed later, is the `imagelist`-structure used for managing the reference frames in inter prediction.

The bulk of the encoding is done in *'CTU Compression'* and *'Reconstruction'*, with the former being in charge of finding the optimal coding decisions and the latter used for creating the reconstructions for coding cost evaluations. Moreover, the two loop filters (i.e., deblocking and SAO) are implemented in *'Reconstruction'*.

The main functionality for bitstream creation is collected under *'HEVC Bitstream Coding'*. The NALU header itself is defined in `nal`, whereas parameter set (i.e., VPS, SPS, and PPS) writing is defined in `encoder_state-bitstream`. The final coding decision and structure writing process is defined in `encoder_coding_tree`. The underlying CABAC implementation and bitstream generation related functionality are defined separately in `cabac`, `context`, and `bitstream`.

For performing parallel tasks, Kvazaar uses the `threadqueue`-structure. It is in charge of scheduling tasks based on the defined dependencies and executing those tasks in separate threads. Previously, the underlying implementation for threading was handled by `pthread` — an external library. However, in order to remove the external dependency, *'Threadwrapper'* was added and provides the same functionality as the `pthread` library but uses native C++[2] threading.

Kvazaar employs various *single instruction, multiple data* (SIMD) optimizations, collected under *'SIMD Optimizations'*, to accelerate data-intensive operations and functions. The availability of special SIMD instruction sets, used for the optimizations, is dependent on the underlying hardware, thus requiring a flexible way of selecting the fastest implementations based on the available special instruction sets. In Kvazaar, `strategyselector` is used for this selection process to choose the optimal functions, at run-time, from the available optimized implementations and the generic implementation.

4.1.2 Coding Scheme

Figure 4.2 depicts the simplified encoding flow of Kvazaar. The encoding process is roughly grouped into *'Control'*, *'High-Level'*, *'Mid-Level'*, and *'Low-Level'*. Additionally, the Kvazaar CLI, that uses the library through the Kvazaar API, is shown.

[2] An imperative programming language with object-oriented features.

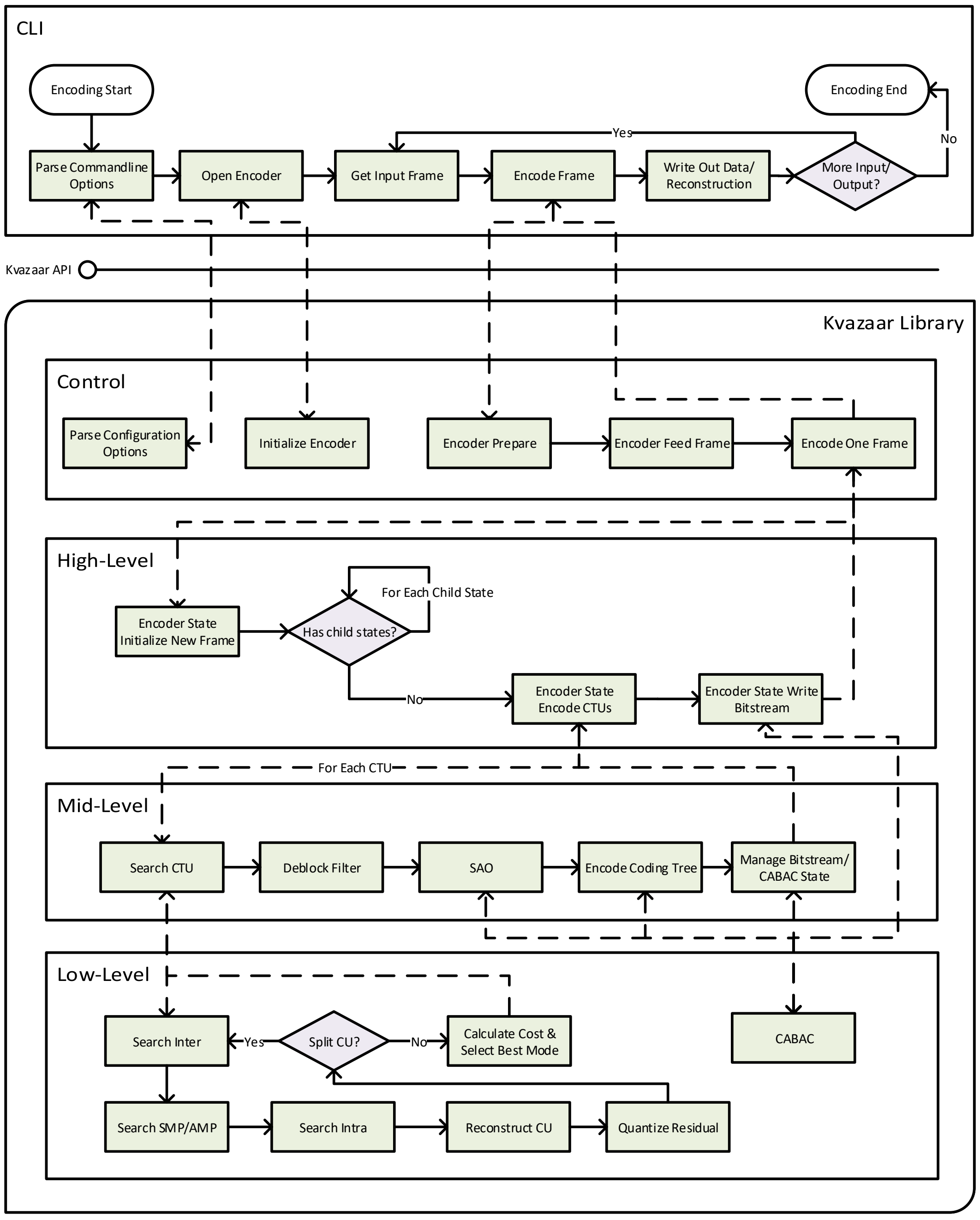


Figure 4.2. *Simplified Kvazaar encoding flow.*

The operation of the CLI is fairly straightforward. First, the encoder configuration is generated from the given command-line parameters. The configuration is then used to initialize the encoder to get it ready for encoding. Next, in the main loop of the CLI, an input picture is read and passed to the encoder. At the same time, an already encoded picture and related bitstream data are returned and written to the designated output streams. This loop is repeated until no more input pictures remain and all the output has been written out. Finally, when the encoding is finished, all allocated resources are freed and the encoder is closed.

Inside the Kvazaar library, *'Control'* handles the main encoding loop. The first step is to prepare the encoder state that, among other things, updates the reference frame information. If a hierarchical GOP is used, the encoding order differs from the viewing order (i.e., input order). To this end, Kvazaar uses an intermediate picture buffer to re-order the input pictures to the encoding order. After getting the correct picture, the actual coding can begin. If a picture has been encoded, bitstream data and the reconstructed picture are returned.

In the *'High-Level'* encoding layer, Kvazaar uses the picture passed from the *'Control'* layer to initialize the encoder state structure. The encoding process of a picture is controlled by the hierarchical encoder state that contains, in some cases, several layers of child encoder states. First, the leaf child states need to be found; leaf child states usually determine if tiles or WPP are used and are mostly related to parallelism (see Section 4.1.3). Moreover, the necessary dependencies for parallel processing are set at this point, if applicable.

After finding the leaf states, encoding proceeds by processing each CTU — a block of 64×64 luma pixels in Kvazaar — separately. When the CTU encoding is done, parameter sets and slice headers (see Section 2.1.6) are written to the bitstream.

Moving to the next layer, *'Mid-Level'* handles CTU related operations in the encoding process. The first step is to perform a recursive depth-first search, on the CTU, to find the optimal[3] CU partitioning and coding modes, as described in Sections 2.1.2 and 2.1.3. The search returns the chosen block partitions and mode decisions with related information and the reconstructed CTU. The next step is to apply loop filters (i.e., deblocking and SAO) to the reconstructed CTU, making it ready for use in inter coding. The final steps are to write the CTU coding tree and mode decisions to the bitstream and to update the CABAC state.

The first step in the CTU search, assigned to `'Low-Level'`, is to perform inter search, described in more detail by Lemmetti et al. [51], starting from the 64×64 CU block. Inter search goes over all available reference frames and chooses the one with the lowest coding cost. Motion estimation, used to find optimal motion vectors, is performed using one of several motion estimation algorithms implemented in Kvazaar and specified in the configuration. For fast motion estimation, a hexagon-based search pattern is used [53]. Additionally, inter search checks possible merge candidates for merge or skip mode and performs bi-prediction, if enabled. After inter search has been performed for the current CU, possible AMP and SMP partitions are also searched.

After inter search has been completed, intra mode search is performed, as detailed by Viitanen et al. [47]. Once the cost of the intra and inter modes have been determined, the

[3] In practice, finding the optimal solution may be too computationally expensive, and it is necessary to settle for a near-optimal solution.

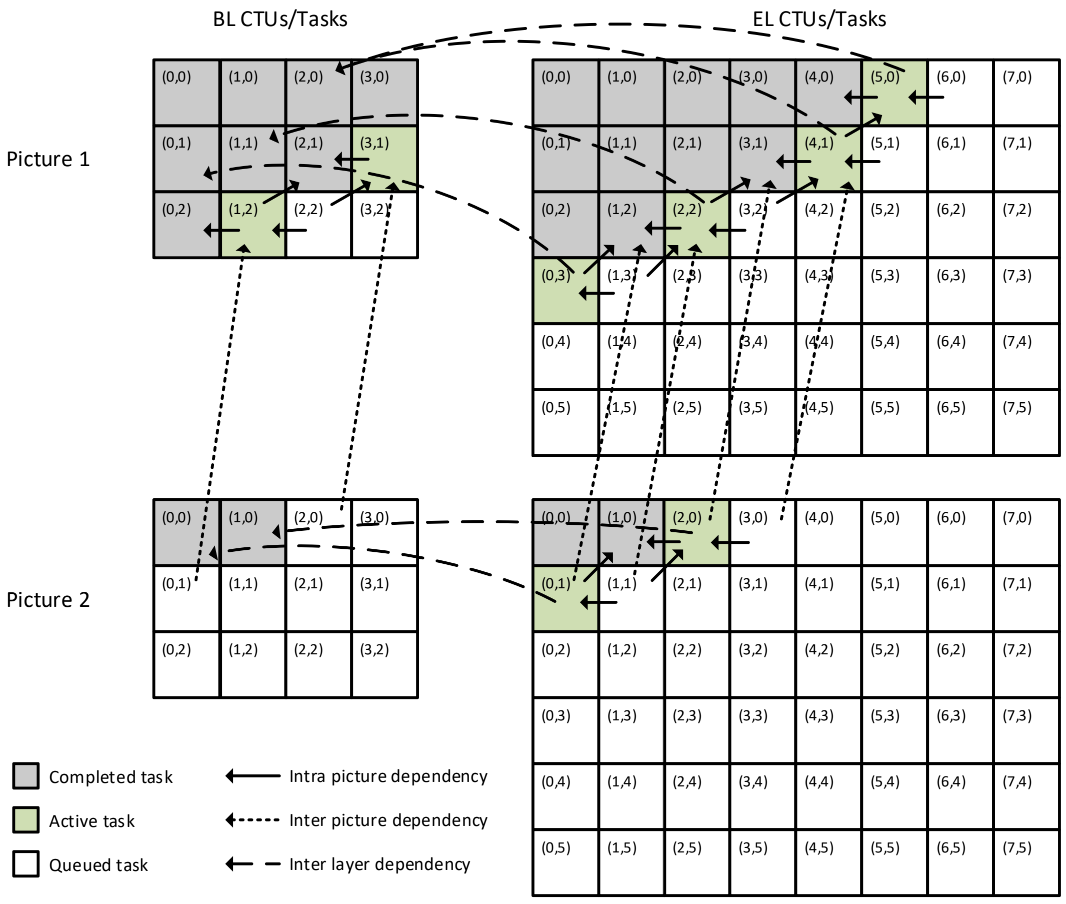


Figure 4.3. *Example Scalable Kvazaar WPP and OWF dependency diagram, with dependencies for the currently active tasks.*

one with the lowest cost is chosen. Based on the chosen prediction mode, the current CU is reconstructed to calculate the residual for the block (see Section 2.1.4). Next, Kvazaar attempts to split the current CU, if allowed, and recursively calculates the costs of the sub-CUs. After the recursion has finished, the best modes and coding tree structure are selected and propagated to the caller, along with the total cost for the CTU.

4.1.3 Parallelization

The main tools for parallel encoding in Kvazaar are tiles, WPP, and *overlapped wavefront* (OWF) [54] parallelization. OWF can be used in conjunction with WPP and tiles. On the other hand, WPP and tiles should be used separately[4]. [48]

As mentioned in Section 2.1.2, WPP allows encoding rows of CTUs almost independently [22]. In Kvazaar, each CTU row is a separate (sub-)`encoder_state`; a task is added to the `threadqueue` for each CTU in a row. Moreover, dependencies need to be added to

[4] Kvazaar supports using WPP and tiles together, but this is not allowed in the HEVC standard [4].

each task to guarantee a correct processing order and CABAC context. Figure 4.3 shows the necessary dependencies to the left CTU and the top-right CTU of the above CTU row.

Tiles allow rectangular parts of the picture to be encoded independently. As a result, Kvazaar is able to encode each tile in a separate task. The amount of parallelism depends on the tile size, but smaller tiles tend to result in worse coding efficiency.

The picture-level parallel processing tool OWF allows Kvazaar to encode multiple pictures at the same time, resulting in even more tasks being available at any given time. Especially with WPP, only a few tasks can be executed at the start and the end of a picture, so having several pictures helps keep the processor busy, leading to a more efficient pipeline. However, some additional dependencies need to be added, when using OWF, to make sure all reference frames are available for inter prediction. In Kvazaar, when using WPP with OWF, the inter picture dependency is set to the reference frame's collocated CTU's bottom-right neighbor in the lower CTU row, as seen in Figure 4.3.

4.2 SHVC Implementation

SHVC was designed from the start to only require high-level syntax changes to allow reusing a standard single-layer HEVC encoder as much as possible [7]. As a result, Scalable Kvazaar only contains a few low-level changes to the coding framework of single-layer Kvazaar. The largest differences are in the parameter set syntax (additional extension fields) and in the high-level structure of Kvazaar to accommodate multiple layers. Scalable Kvazaar strives to maintain compatibility with existing applications by not changing the Kvazaar API. Furthermore, the overhead to the single layer encoding is kept to a minimum.

Section 4.2.1 gives an overview of Scalable Kvazaar and its features. Section 4.2.2 goes over the main architecture changes and additions introduced by the scalability extension. Section 4.2.3 presents the modified encoding flow of Scalable Kvazaar. Section 4.2.5 introduces the scaler library, used for performing picture resampling and scaling, as well as the relevant functionality required by Scalable Kvazaar. Finally, Section 4.2.4 details the optimizations used to accelerate scalability extension related operations.

4.2.1 Overview

Figure 4.4 depicts a simplified block diagram of spatial scalability in Scalable Kvazaar. Separate instances of the Kvazaar encoder are prepared for the BL and EL, respectively. Each input picture is passed to both layers if only one input stream is provided. The source pictures are downscaled for the BL if the desired BL resolution differs from the input resolution (i.e., in spatial scalability). Alternatively, two input video streams can be specified, with the correct resolutions, to avoid downscaling.

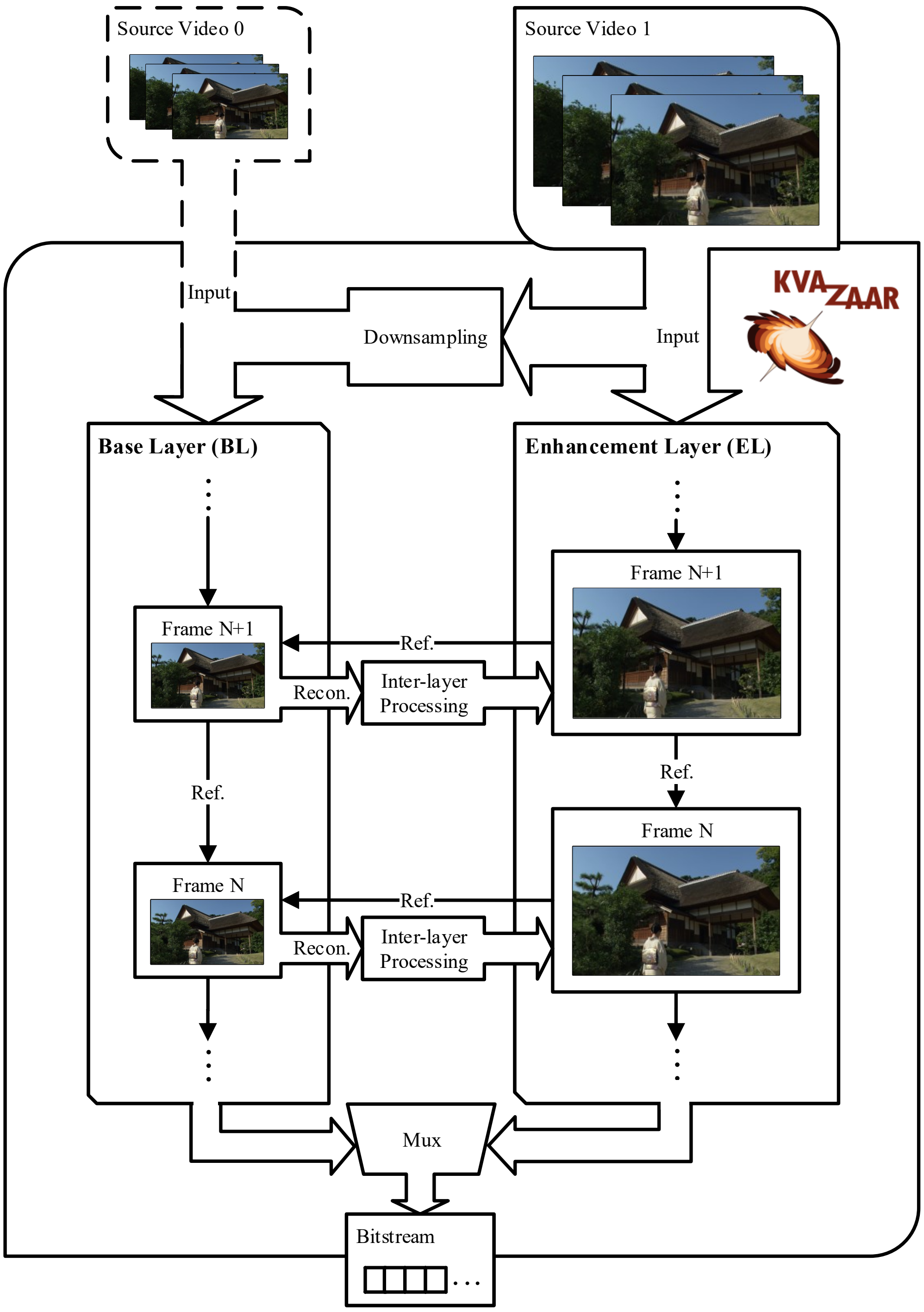


Figure 4.4. *The high-level flow of Scalable Kvazaar when using spatial scalability.*

In Scalable Kvazaar, SNR scalability (see Section 2.2.3) is achieved by assigning a smaller QP to the EL encoder, though other coding tools may also be used to achieve the same effect. The encoding flow of SNR scalability mostly follows that of spatial scalability; however, down- and upscaling is skipped if both layers have the same resolution.

The BL and EL encoders both have their respective layer parameters that control the encoding process. The BL encoder only uses BL pictures as references in inter prediction, whereas the EL encoder utilizes BL pictures (i.e., ILR pictures) on top of EL pictures, as mentioned in Section 2.2.2. ILRs are added when the reference picture list in the EL encoder is updated. Additionally, when TMVP is enabled in the EL encoder, by default, the ILR picture is set as the collocated TMVP reference picture. This means that motion information from the BL also needs to be copied when updating the reference picture list.

When only using SNR scalability, there is a one-to-one correspondence between the spatial resolutions of the BL and EL, allowing the EL encoder to directly use the reconstruction and motion information from the BL. With spatial scalability, however, an additional inter-layer processing step is performed. It involves upscaling the reconstruction and motion field of the ILR picture. As detailed in Section 2.2.4, the upsampling process for texture data is carried out using interpolation filters specified in the SHVC standard [4]. Moreover, when TMVP is enabled, motion field upsampling also needs to be performed.

The encoding process is concluded by concatenating the BL and EL bitstreams to form the final output. As mentioned in Section 2.2.6, NALUs from the different layer encoders will be distinguishable by their LID; zero for the BL and one for the EL.

Temporal scaling was already included in HEVC [5], but Kvazaar lacked full support for it. Scalable Kvazaar implements temporal scaling for hierarchical GOP structures, as this allows for a reference structure that is compatible with temporal scaling. The temporal layer of a given NALU is set in the header, allowing the decoder to skip NALUs with a TID larger than the target temporal layer.

4.2.2 Architectural Design

The high-level architecture of Scalable Kvazaar remains largely unchanged, and Figure 4.1.1 still holds true. Most changes only modify or extend the existing architecture but leave the high-level structure intact. The only addition is the *'Scaler Library'*, described in Section 4.2.5.

The configuration and CLI part of Kvazaar was modified to accommodate the extra layers and input/output pictures. The API does not need to be modified because the picture data structure can be repurposed to chain multiple pictures together by using its existing picture pointer field. However, since the picture data structure is used to store reference picture list information, such as POCs, additional reference frame information is added, including the LID, TID, and long term reference flag.

In the configuration structure, fields were added for tracking layers, ILR references, and a special data structure that handles cross-layer shared parameters. Additionally, a pointer to the next layer's configuration is added to allow passing multiple configurations through the API — one for each layer.

The `encoder_control`-structure has been supplemented with layer related information — LID, input layer, and various bitstream generation related values. Furthermore, a separate `encoder_control`-structure is created for each layer, connected by a pointer pointing to the next layer's encoder control structure. Additionally, the up- and down-scaling parameter structures — populated during encoder initialization — are located in the `encoder_control`-structure and are used for controlling the scaling process in the inter-layer processing and downscaling steps.

Modifications to the `encoder_state`-structure are largely related to the parallel task management, described in Section 4.2.4. The list of upscaling tasks is located in the `encoder_state`-structure along with other parameters such as the upscaling area and relevant buffers for each task. Moreover, EL encoder states are directly connected — using pointers — to the RL encoder states, allowing inter-layer dependencies to be set easily.

The `imagelist`-structure is used to manage references. However, reference pictures are identified by their POCs, resulting in ambiguity when adding ILR pictures. To this end, extra information needs to be added to identify the (temporal) layer as well as the long term reference picture status. The function for adding reference pictures is modified accordingly to take in the new information.

Much of the high-level picture scaling related functionality has been added under `image`. This includes the Kvazaar side picture scaling functions and the definition for a picture scaling parameter structure. When scaling a sub-block of a picture, the scaling parameter structure includes: the position and size of the block; the input and output Kvazaar picture buffers; and *'Scaler'* library buffers for the input and scaled output as well as a temporary buffer for storing the results from the initial horizontal scaling pass.

Motion field scaling functionality has been added under `cu`. In Kvazaar, motion information is contained in the `cu_array`-structure. Similarly to picture scaling, `cu` contains parameters for controlling the `cu_array` upsampling process. The parameters contain the original and the upsampled `cu_array`-structure, along with the target CTU and the scaling constants used for calculating the collocated CU and scaling the motion vectors.

The high-level search process, employed by Kvazaar, requires no changes for scalability features to work. The ILR pictures are mostly handled like any other reference picture, as was mentioned in Section 4.2.1. Only `search_inter` has been modified to skip the search procedure, since ILRs are required, in the standard [4], to use a zero motion vector, making inter search unnecessary.

SHVC brings some changes to the bitstream syntax, as described in Section 2.2.6. The NALU writing functionality in `nal` has been modified to include the LID and TID. Additionally, VPS extension syntax elements have been included in `encoder_state-bitstream`.

The extension is used to define the types of scalability being used and other parameters, for scalable encoding, such as the resolutions of layers. In addition, the extension is used to define the reference structure between layers (i.e., RLs). Lastly, profile, tier, and level information is written, for each layer, in the VPS extension. The *'Scalable Main'* profile is used for the EL, as is required when using scalability features [4].

The SPS remains largely unchanged, whereas the PPS includes an extension syntax structure similarly to the VPS. The PPS extension allows setting RL location and phase related information as well as the color mapping flag for when layers use different color spaces.

A new strategy has been added to *'SIMD Optimizations'* — `strategies-resample`. This lets Scalable Kvazaar switch between the un-optimized and AVX2 optimized resampling implementations, depending on the underlying hardware's capabilities, as described in Section 4.1.1. The implementation, chosen by `strategyselector`, is passed as a function pointer to the scaling functions when they are called.

4.2.3 Encoding Flow

Kvazaar is designed for encoding one video stream at a time, requiring a fair amount of changes in the encoding flow to enable scalable coding. The modified flow is depicted in Figure 4.5. When comparing the flow of Scalable Kvazaar to the original, in Figure 4.2, it can be seen that most changes are limited to *'High-Level'* and up.

The *'CLI'* side of Kvazaar needs to be modified to accommodate the multi-layer nature of scalable encoding. The process of parsing the command-line options and opening the encoder are mostly unchanged. The first changes are in the input picture reading; each layer may have a separate input, so an input picture needs to be read from each input stream. Input pictures are still passed and output data is read with one function call, and as a result, the encode frame step remains the same.

The output bitstream is already concatenated inside the Kvazaar library, requiring no changes to out data writing. However, if the reconstructed output pictures need to be written out to files, it is necessary to loop over the output layers. When encoding is finished, the encoder is closed and resources de-allocated, as before.

The *'Control'* portion of the Kvazaar encoding flow has gone through various changes to account for the intricacies of scalable encoding. Firstly, as mentioned in the previous subsections, Scalable Kvazaar needs to generate a configuration and initialize an encoder instance for each layer being encoded. The configurations and encoder instances are chained together using pointers to allow passing them through the Kvazaar API without changing the API itself. Secondly, in the actual encoding flow, care needs to be taken with the order in which certain functions are called to guarantee correct execution in all cases.

Encoder preparation proceeds as before but with a scalability preparation step added after the picture re-order step; this is repeated for each layer. Moreover, after scalability preparation, the frame initialization step is called separately from the one frame encode step. The scalability prepare step handles adding ILRs to the reference list, copying TMVP related information between states, and preparing upscaling and ILR picture buffers, as necessary. After all preparations are complete, encoding is started for each layer.

Special handling is needed in the *'Control'* flow when TMVP is used with SNR scalability and several frames are processed at the same time (i.e., OWF, see Section 4.1.3). Since SNR scalability allows directly using picture buffers from the RL, data races[5] may occur when copying TMVP related information; a delay is used to resolve the issue, as seen in Figure 4.5. When the delay is enabled, EL preparation and encoding is skipped until preparations for the next BL picture are done. To keep the output in sync, BL results are delayed until the matching EL picture has been encoded, i.e., until the next evocation of frame encoding.

In Figure 4.2, *'High-Level'* has been broken up and expanded to better show the new Scalable Kvazaar flow. Encoder state frame initialization has been removed from the main *'High-Level'* path, due to the changes to *'Control'*. Moreover, the CTU encoding step has been expanded into a loop — over all CTUs — to show the added inter-layer processing step. This new step is in charge of necessary preparations (i.e., scaling and dependencies) that make it possible for an EL to use ILRs.

The CTU encoding process, described in *'Mid-Level'*, is not affected by changes made for scalable encoding. The only additions to this layer are ILR processing related tasks consisting of starting block (i.e., CTU or tile) upscaling and CU array upscaling, when necessary. Moreover, parallel processing dependencies, for scaling tasks, are set in inter-layer processing, described in Section 4.2.4.

The main loop in *'Low-Level'* remains unchanged from single-layer Kvazaar. The only additions are the YUV and CU array scaling steps. In the YUV scaling step, parameters passed from *'Mid-Level'* are used to invoke block scaling. The target block is determined in *'Mid-Level'* and YUV scaling only calculates the correct areas for each scaling step and calls the scaling function with the correct buffers and parameters. YUV scaling uses *'Scaler Library'* (see Section 4.2.5) to perform the actual block scaling. The CU array scaling step performs motion field upsampling on the CTU specified in the upsampling parameters that are set in *'Mid-Level'*. Motion field upsampling follows the process described in Section 2.2.4.

[5] When concurrent processes access the same spot in memory, nondeterministic behavior may ensue, caused by the chancing order of read and write accesses.

4.2.4 Optimizations

As discussed in Section 4.1.3, Kvazaar has several parallelism tools for accelerating the encoding process. Furthermore, various SIMD optimizations are employed for computationally expensive operations [49]. Scalable Kvazaar is able to take advantage of the existing SIMD optimizations because they target block-level operations, allowing ELs to use them without any modifications. On the other hand, the parallelism tools require some additional modifications to get them working well together with the scaling process, when inter-layer processing is needed (i.e., with spatial scalability).

WPP allows encoding the CTUs of a picture almost independently; the upscaling process needs to be divided in a similar fashion to allow the EL WPP pipeline to proceed smoothly. To this end, upscaling is divided into parallel tasks that process roughly one BL CTU at a time; a dependency needs to be added between the BL WPP tasks and the upscaling task to guarantee that the BL texture and motion information is available before the corresponding EL CTU upscaling and encoding tasks begin. Moreover, the EL WPP tasks need to depend on the respective upscaling tasks to make sure the upscaled reference picture texture and motion information is available before starting to encode a CTU. Figure 4.3 shows the inter-layer dependencies between EL CTUs and BL CTUs.

With tiles, the upscaling process is divided into separate tasks, similarly to WPP, but with upscaling done at the tile level. Moreover, the dependencies between tasks are the same as with WPP but using tile tasks instead.

Loop filters add some extra considerations into the parallel processing pipeline because CTU edge pixels can only be filtered when the neighboring CTU edge pixels are also available. This means that the CTU and tile upscaling task dependencies need to be adjusted accordingly. The final texture data of the dependent BL CTU or tile will only be available after the neighboring CTUs or tiles have been encoded.

The picture-level parallelism tool OWF adds some extra dependencies to the parallel processing pipeline, as seen in Figure 4.3. Most of these dependencies are sufficiently handled by the existing implementation of the parallel processing tools. However, when using TMVP, the motion information upscaling task dependency of a WPP task needs to be adjusted to allow enough motion data to become available from the RL picture.

If spatial scaling is not used (i.e., SNR only), upscaling jobs are not used and WPP/tile dependencies can be set directly between the tasks of the different layers. In this case, layer resolutions match, making the dependency matching task as simple as using the collocated task of the RL as the dependency. However, if the loop filter settings of the layers differ, the dependency needs to be adjusted to account for edge pixel filtering.

On top of the high-level parallel processing tools used to accelerate Scalable Kvazaar, the low-level resampling (i.e., filtering) operation of the upscaling process has also been optimized. The texture resampling operation has been implemented using AVX2 intrinsic functions to take advantage of SIMD processing for maximal throughput. An overview of the optimized upsampling process is shown in Figure 4.6.

The opaque buffer, used internally by the scaler library (see Section 4.2.5), allows several source-destination bit depth combinations. The optimized resampling function supports all possible combinations. However, the optimal combination, and the one used in Scalable Kvazaar, is an 8-bit to 16-bit horizontal resampling step and a 16-bit to 8-bit vertical resampling step. Resampling is done in two parts and the bit depths have been chosen to maximize the throughput while making sure the intermediate resampling values do not overflow[6].

In the first horizontal filter pass, 8-bit input values, from the given texture block, and 8-bit filter coefficients are loaded into 256-bit registers. Using `_mm256_maddubs_epi16` and addition intrinsic functions, sixteen pixels can be processed with 8-tap filters in one batch. The resulting 16-bit intermediate values are stored in the destination buffer for the next pass. The vertical pass uses the 16-bit intermediate values and 16-bit filter coefficients along with `_mm256_madd_epi16` and addition intrinsic functions to calculate the values for eight pixels at a time. The values are then scaled back to the 8-bit value range. The scaling can be done to 8 pixels at a time and the results from four iterations can be accumulated for the final clipping and storing operation, making the most of the 256-bit registers.

4.2.5 Scaler Library

Scaler library is used for resampling (i.e., scaling) pictures, as needed by inter-layer processing. It provides functionality, for generating the scaling parameters and necessary picture buffers, as well as an extensive interface for controlling the granularity of the resampling process. Figure 4.1 shows the main functions Scalable Kvazaar uses, under the *'Scaler'*-interface, and the parts that make up the library, under *'Scaler Library'*.

The majority of the *'Scaler Library'* functionality is implemented in `scaler`. It contains the high-level picture scaling flow, functions for manipulating the buffers and data structures used in the scaling process, as well as the generic implementation of the resampling algorithm. The AVX2 optimized resampling algorithm — detailed in Section 4.2.4 — is defined in `scaler-avx2`. The resampling filter coefficients[7] and other shared helper functions are defined in `scaler-util`.

The scaler library internally uses an opaque picture buffer for manipulating picture data.

[6] The 16-bit intermediate depth is only sufficient for 8-tap upsampling filters, but not for 12-tap downsampling filters.

[7] See Appendix A for a listing of the filter coefficients.

Opaque buffers allow passing external (i.e., Kvazaar) picture buffers to the scaler library without the need for addition memory copies. Furthermore, opaque buffers allow using different pixel bit depths between scaling steps, which is useful for SIMD optimizations. Opaque picture buffers can be created using the `kvz_newOpaqueYuvBuffer`-function. It creates buffers for both the luma and chroma components at the same time. The function allows passing pointers to already existing picture data or, alternatively, allocating the required amount of memory for the specified picture size and pixel bit depth. The external buffers may also be set after creating an opaque buffer by calling the `kvz_setOpaqueYuvBuffer` function on previously created opaque buffers. Memory deallocation is handled by calling `kvz_deallocateOpaqueYuvBuffer` that optionally frees the allocated memory if no external dependencies remain for the picture data.

The scaling process is invoked by calling either `kvz_opaqueYuvBlockStepScaling` or `kvz_opaqueYuvBlockStepScaling_adapter`. The difference between the two functions is that `kvz_opaqueYuvBlockStepScaling_adapter` allows passing a function pointer to the resampling function, that should be used in the scaling process, letting the strategy selector — introduced in Section 4.1.1 — switch between the generic and AVX2 optimized version of the resampling function based on the underlying hardware capabilities. The `kvz_opaqueYuvBlockStepScaling`-function scales the source picture into the destination buffer based on the given scaling parameters. Additionally, a sub-block of the source picture may be specified if only a portion of the input picture should be resampled. With a single invocation, the `kvz_opaqueYuvBlockStepScaling` function only performs either the vertical or the horizontal scaling step; the function needs to be called two times in order to complete the scaling process.

Other useful functions, for miscellaneous tasks, include: `kvz_newScalingParameters`, `kvz_blockScalingSrcWidthRange`, and `kvz_blockScalingSrcHeightRange`. Of the three, the first one can be used to generate the scaling parameters based on the source and target picture resolutions. The two latter functions are used for calculating the area, from the source picture, needed to generate the specified scaled block of the target picture; this is useful for calculating dependencies when using WPP (see Section 4.2.4).

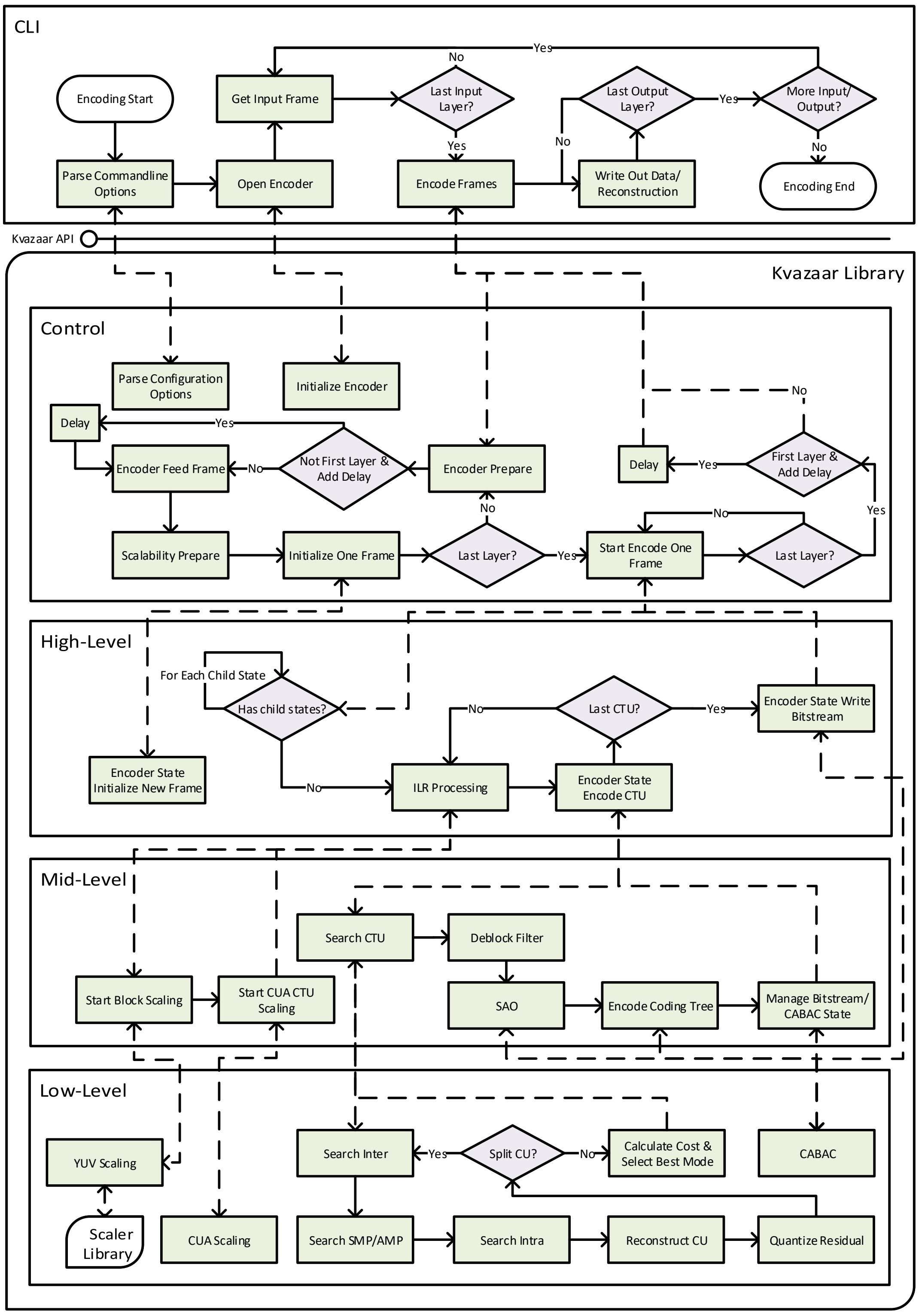


Figure 4.5. *Simplified Scalable Kvazaar encoding flow.*

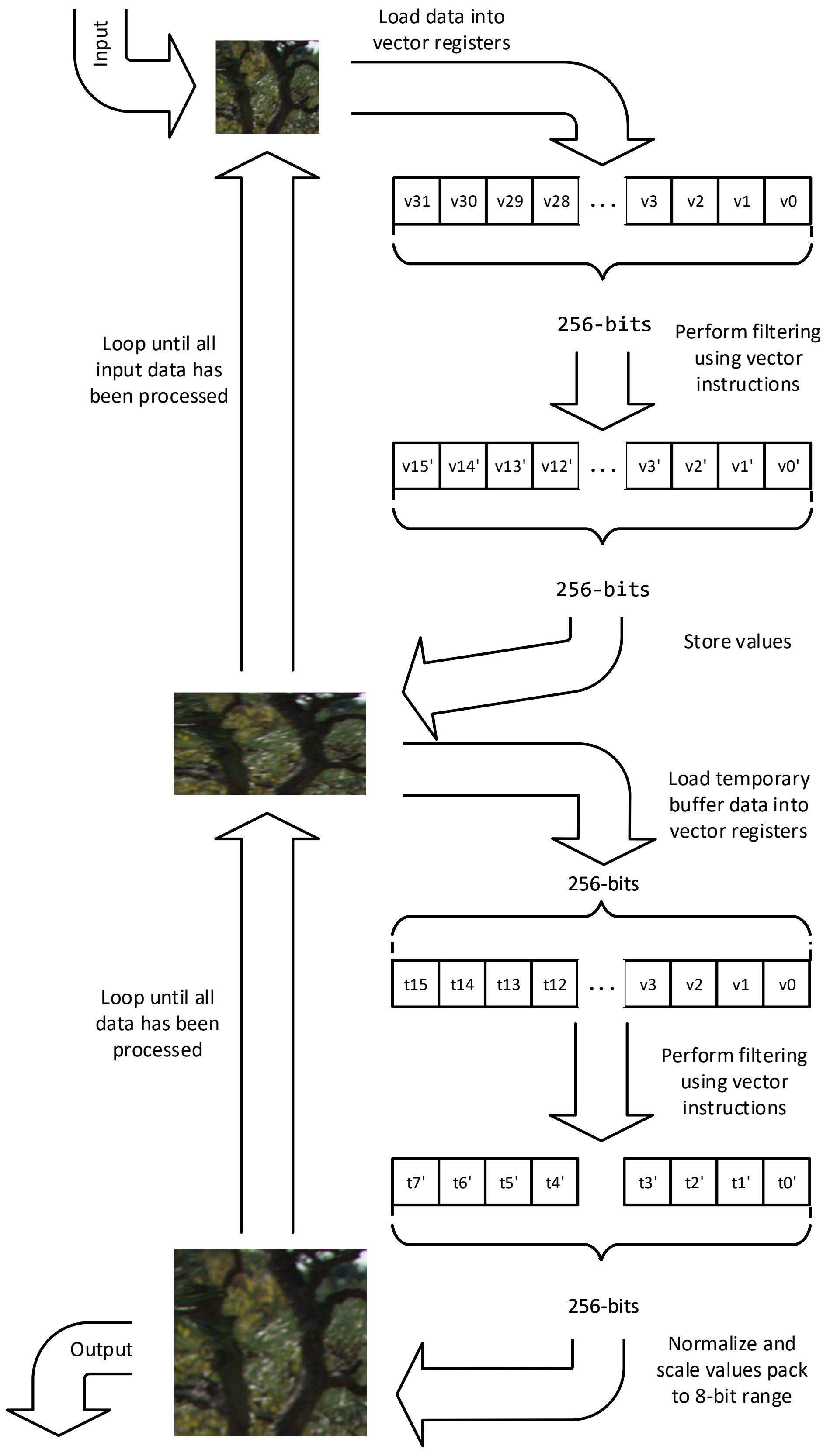

Input
Load data into vector registers
v31 v30 v29 v28 ... v3 v2 v1 v0
256-bits
Perform filtering using vector instructions
v15' v14' v13' v12' ... v3' v2' v1' v0'
256-bits
Loop until all input data has been processed
Store values
Load temporary buffer data into vector registers
256-bits
t15 t14 t13 t12 ... v3 v2 v1 v0
Loop until all data has been processed
Perform filtering using vector instructions
t7' t6' t5' t4' t3' t2' t1' t0'
256-bits
Normalize and scale values pack to 8-bit range
Output

Figure 4.6. *An overview of the AVX2 optimized upsampling process.*

5 SCALABILITY TESTING FRAMEWORK

This section introduces the testing framework NAVETTA that can be used to automate the testing of scalable encoders. NAVETTA is written as a Python[1] package with the full package structure depicted in Figure 5.1. The source code for this framework can be found on github [43].

NAVETTA can be imported as a package by other Python script-files or it can be executed directly. It provides a simple user interface, where a list of test scripts can be given as command-line parameters. Alternatively, an interactive command-line prompt can be used, allowing the user to input the test script names that should be run. The given test script names can either be those already in the `test`-package or a path to any test script. The top level module `cfg` collects most of the necessary configuration parameters that are used when running test scripts.

Section 5.1 describes the structure of a test instances. Section 5.2, on the other hand, goes over the `TestSuite`-package used to execute test instances. Section 5.3 briefly introduces the `tests`-package that is intended to contain user-created test scripts.

5.1 `TestInstances`-package

The `TestInstances` package, as the name implies, defines test instance classes that contain test parameters. Once instantiated and initialized, a test instance–object is used to run tests with the desired parameters. In Addition, it will contain the results of the test runs that can then be accessed through the test instance's API. The similarly named `TestInstance`-module, of the `TestInstance`-package, is an abstract class that defines the test instance API through which all other inherited test instance classes can be used. This allows for a unified processing workflow for heterogeneous payloads and minimizes modifications to other packages if a new inherited test instance class is added.

The main API functions, defined in the base class `TestInstance`, are `__init__`, `run`, and `getResults`. The base class directly provides a `run`-function implementation that checks if results already exist (i.e., are saved to file), for the current parameters, and if not, runs the tests and saves the results to file. The two other aforementioned interface

[1] Object-oriented general-purpose programming language.

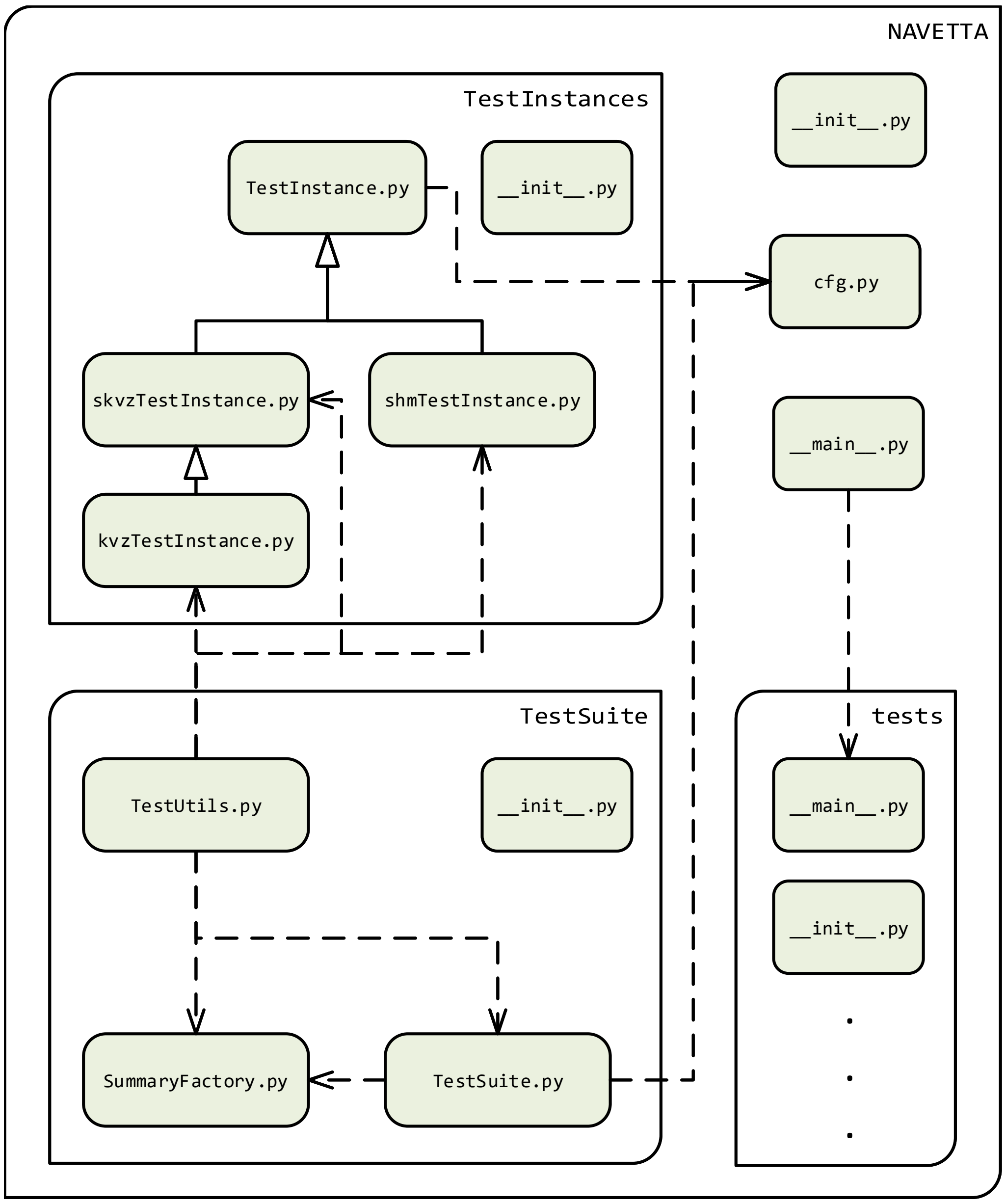


Figure 5.1. *The package structure of NAVETTA testing framework.*

functions are defined as abstract methods, thus requiring them to be implemented by the inherited class. The `__init__`-function is used to set test parameters that will be used when running the test instance. It defines common parameters that are likely to be needed by all of the different inherited test instances. They are as follow:

test_name Unique name given to the test instance, which can be used to refer to the test instance and distinguish it from other tests instances. It is also the name shown in the generated result file.

inputs List of input files (file paths inside of a tuple), representing the sequences that are used in the test. If separate input files are specified for each layer in scalable encoding, each item in the list should be a tuple, where the index, matching a given LID, contains the input file for the specified layer.

input_sizes (optional) Specify the resolution of each input file given in **inputs** (should match its structure). Does not need to be given if the resolution can be inferred from the input filenames.

input_names (optional) Specify more human readable names for the input files (i.e., no file path). Should match the **inputs** structure. If specified, these names are shown in the results instead of the input file paths if specified.

layer_args (optional) Specify additional encoding parameters that are passed to the underlying encoder. If doing scalable encoding with multiple layers, the given layer argument object should be indexable by the respective LID and return the layer specific parameters for the given layer.

layer_sizes (optional) Specify a resolution for each layer, when doing scalable encoding, if the layer resolution differs from the input resolution (e.g., spatial scalability). Gets overwritten by **input_layer_scales**.

input_layer_scales (optional) Specify a scaling ratio for each layer, when doing scalable encoding, if the layer resolution differs from the input resolution (e.g., spatial scalability). Each layer's resolution is derived by multiplying the input resolution by the respective input layer scale.

qps (optional) Specify the QP values used for BD-rate calculations. Takes four QP points that are shared by each layer if layer specific values are not given. Alternatively, if a list of tuples are given, each layer is assigned QP values from the tuple indexed by the LID (e.g., in SNR scalability).

out_name (optional) Name/path of the output files, where the results from the test encodings are saved until they can be processed.

bin_name (optional) Filename of the executable that should be used for encoding.

version (optional) Specify the version of the executable. Can be used to force re-running tests with the same parameters when the executable has changed but uses the same name.

****misc (optional)** Used to catch inherited test instance specific parameters.

As for the `getResults`-function, it takes in, as a parameter, a function that can be used to collect the results in a format specified by the caller. The Function should take in the name of the sequence, the QP, the LID, the size of the encoded file, the size of the encoded file per unit of time, the time it took to encode the file, and the PSNR. The other (private) abstract functions that need to be defined, by the inherited class, are `_get_gname_hash`, for generating a unique identifier for a test parameter set, and `_run_tests` that implements the actual test execution procedure.

The `skvzTestInstance`-class implements a test instance for Scalable Kvazaar. It handles calling Scalable Kvazaar, with the given test parameters, and collects all the pertinent information from the encoding. On top of the initialization parameters listed above, `skvzTestInstance` adds the optional parameters **validate** and **retries**. They are used to control the validation of an encoded sequence, by attempting to decode it with SHM, and setting how many times the encoding process is repeated if the decoding fails.

To support unmodified Kvazaar encoding, a `kvzTestInstance` is inherited from the aforementioned `skvzTestInstance`. Due to the similarity of the normal Kvazaar and the scalable version, only the internal result parsing method needs to be re-defined. For clarity, the results folder definition is also changed.

The `shmTestInstance` class implements a test instance for SHM encoding. Because SHM does not provide built-in tools for parallelism, `shmTestInstance` runs several SHM encodings in parallel. This is achieved by creating a worker function that is in charge of a single encoding. An asynchronous Python pool is used to manage the worker instances. This can considerably shorten testing times for the slow SHM when many parallel cores are available. Since SHM uses special configuration files to set the encoding process parameters, the `shmTestInstance` initialization definition differs slightly from the `TestInstance` initialization. A new **configs** parameter is added that is roughly equivalent to the **inputs** parameter. Since a configuration file may define the input file that should be used, the **configs** parameter can be used, instead of the **inputs** parameter, to define the test sequences.

5.2 `TestSuite`-package

The `TestsSuite`-package is in charge of running test instances and processing the results (`TestSuite`-module). The results can then be collected and formed into different kinds of summaries as the user desires (`SummaryFactory`-module). The package also provides utility functions/classes for creating test instances and generating summary definitions (`TestUtils`-module).

5.2.1 `TestSuite`-module

The `TestSuite`-module provides the `runTests`-function that takes, as a parameter, a list of test instances and a name for the output results file. Additionally, the function may be given any number of summary definitions and a list of (layer) combinations. The supported summary types are defined in the `SummaryFactory`-module. Helper functions, for creating these definitions, are found in the `TestUtils`-module. As for the (layer) combinations, they allow the user to combine results from separate tests — given as a list of test instance name tuples — as if they were one test. Regular combinations simply add the results together, but layer combinations treat the results as if each test was a specific layer; the layer is determined by the index of the test instance in the layer combination definition.

The `runTests`-function handles calling the `run`-method for each test instance and parses the results using the `getResults`-method. Next, any (layer) combinations, given by the user, are generated. Finally, the function proceeds to write the results into an Excel file. All of this can be done with heterogeneous test instance classes, without test instance type specific logic, since class inheritance was used.

5.2.2 `SummaryFactory`-module

The `SummaryFactory`-module contains definitions for different summary types. Three types are defined: **BDBRMatrix**, **Anchor_list**, and **Curve_chart**. The base summary definition, used as a container for other summary definitions, is a Python dictionary with fields for the summary type, a name, and the summary types definition. The name is used as the Excel sheet name and should be unique among the other summary definitions included in the results file. Helper functions for creating the summary definitions are included for each summary type and the base definition. In addition, there are separate functions for writing each summary type, based on the given definition, and a function that can be given several (base) definitions, and it will call the respective write function for each definition. The full summary definition hierarchy is shown in Figure 5.2.

The **BDBRMatrix** summary type allows creating an $N \times N$ summary matrix of tests, where N is the number of tests included in the results file. Each value in position (x, y), in the matrix, contains the comparison of the yth vertical test with the xth horizontal test. In addition to BD-rate comparisons, this summary type supports bit (e.g., encoded file size), PSNR, and encoding time comparisons. The **BDBRMatrix** definition contains fields for toggling the inclusion of each aforementioned comparison type and a **layers** field for selecting the target layers that should be included from a given tests. The layering field should contain a Python dictionary with test names as keys and a list of LIDs, that are to be included, as the values. If a test is not included in the **layers** field, only the total-layer (i.e., aggregation of all other layers) is included from that test.

The **Anchor_list** summary type allows creating lists of test comparisons with varying anchors. Each specified test is compared with an associated anchor that may differ between each test. The supported comparison types are the same as with **BDBRMatrix**. Furthermore, each comparison type can have different test and anchor inputs from each other. To facilitate this, a sub-definition is used. The **Anchor_list** definition contains fields for each type of comparison that should have, as values, either an anchor list sub-definition or **None** if the respective comparison type should not be included. The sub-definition should be a Python dictionary with keys for each test that is to be included and the values are tuples of anchor test names or **None** if absolute values (i.e., no comparison) are to

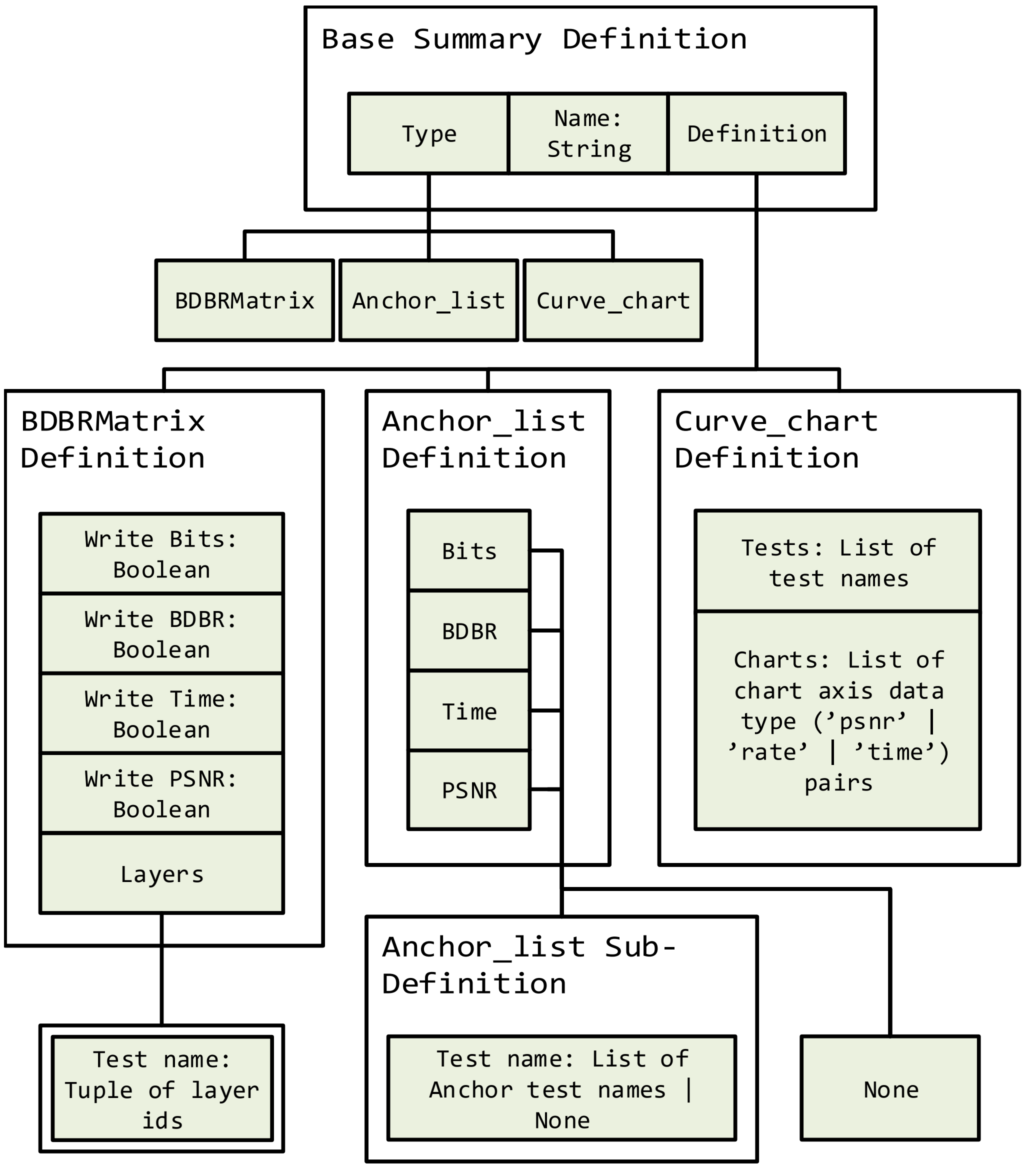


Figure 5.2. *Summary type definition hierarchy.*

be used (not valid for BD-rate).

The **Curve_chart** summary type allows creating per-sequence plots based on the values of different data-points; here the data-points are the different QP values used for the tests. This can be used to visualize the bit rate curves that underline the BD-rate calculation. The **Curve_chart** definition simply takes a list of test names, for which a curve is added in each Excel chart, and a list of chart definitions in the form of datatype pairs. The first value in the pair is used as the x-axis datatype and the second as the y-axis datatype. The supported datatypes are 'psnr' (i.e., quality), 'rate' (i.e., bit rate), and 'time' (i.e., encoding time).

5.2.3 `TestUtils`-module

The `TestUtils`-module provides helper functions for creating test scripts. It contains functions for generating summary definitions as well as a helper class definition for defining test parameter groups that can be used to generate test instances.

The `TestParameterGroup`-class can be useful for creating test instances when all combinations of several parameter values need to be tested. The class calculates a Cartesian product of the specified parameter values and creates a test instance for each combination. Additionally, these test instances can be filtered, using a filter function, to remove unwanted parameter combinations. The names of the parameters, added to a parameter group, should match the test instance construction parameters as they are passed directly to the constructor when creating test instances. User defined helper parameters can also be used because un-recognized parameters are discarded.

New parameters can be added to a `TestParameterGroup`-object with the class methods `add_parm_set` and `add_const_param`. The former adds a parameter with multiple possible values and the latter adds a parameter with one value shared by all generated test instances. For modifying the final parameter sets, passed to the test instance constructors, the `filter_parameter_group`- and `set_param_group_transformer`-method are provided. The first function takes in a filter function that is applied on top of the other filters that have been given; a parameter set needs to pass all applied filters to be included in test instance creation. Filter functions should be such that they take in all parameters, given to the parameter group instance, and return true if the given parameter value combination is acceptable.

The `set_param_group_transformer`-method, on the other hand, lets the user set a transformer function that transforms parameter sets, based on, e.g., user defined helper parameters, to derive the final parameter sets. This can be useful if, for example, the test instance name parameter needs to be derived from other parameter values. The transformer function should take in all parameters, given to the parameter group instance, and return a Python dictionary of the transformed parameter group. For getting the final parameter groups, `TestParameterGroup`-class provides the `dump`-function that returns a list of all parameter sets as dictionaries. Alternatively, the class also provides convenience functions — `to_skvz_test_instance` and `to_shm_test_instance` — that return a list of test instances, constructed with the parameter groups' parameters .

The `TestUtils`-module also provides a helper function for creating transformer functions that can be assigned to `TestParameterGroup`-objects. The `transformerFactory`-function allows the user to pass named parameters to it with a callable object (i.e., a function or lambda) as the value. These name-function pairs are used to modify the matching parameter in the parameter group by invoking the associated function. If the name does not match a parameter in the parameter group, a new parameter is added under the new name. Functions, given to the `transformerFactory`, are invoked by passing them all parameters from a parameter group; the return value should then be the new value assigned to the parameter under the name given to the `transformerFactory`-function. The helper function, thus, allows the user to easily assign parameter specific transformer functions that can still depend on other parameter's values. This is useful, for example, when creating test names based on other parameter values.

To help with generating (layer) combination definitions, the `TestUtils`-module provides the `generate_combi`-function. It takes in `TestParameterGroup`-objects, a combination condition function, an optional name function, and an optional transformation function. The combination condition function is used to decide if two parameter groups should be combined. To this end, it takes in two dictionaries representing two different parameter groups. The return value of the combination condition function should be a boolean or an integer value for enforcing an ordering of the parameter groups. The ordering of the parameter groups is used to decide the layer of each parameter group being combined in the combination definition. Additionally, a helper function (`combiFactory`), for combination function generation, is also provided. It allows the user to specify functions that take two parameter group dictionaries and per-parameter functions as parameters. The given functions are combined into a single function that aggregates the results from the other functions.

As for the name function, it can be used to specify how the names for the tests are generated; it should takes in the parameters of a parameter group and return a string. Finally, the transformation function allows specifying an additional stage that generates sub-sets from the sets generated by the combination condition function. Some other relevant functions, for combination generation, are: the `get_combi_names`-function that returns a list of combined test names based on the given combination definition; and the `get_test_names`-function that returns the names of the given test instances.

The `TestUtils`-module includes functions for creating summary type definitions. The `make_BDBRMatrix_definition`-function is used for creating **BDBRMatrix** definitions. It takes in a list of test names, a layering function, an optional filter function, and boolean values, for each comparison type, in order to select the inclusion of the respective comparison types in the summary. The layering function should take a test name and return a tuple of LIDs, representing the layers that are to be included. As before, the filter function can be used to leave out some of the given tests.

For **Anhor_List** summary definitions, two functions are included with slightly different functionality. The `make_AnchorList_singleAnchor_definition`-function allows creating anchor lists with a single anchor test used for all specified tests. For creating more complex anchor lists, the `make_AnchorList_multiAnchor_definition`-function is provided, allowing the creation of anchor lists with different anchors between tests.

The single-anchor version takes in: the name of the anchor test or a name-LID pair; the tests compared with the anchor; a filter function for filtering out unwanted test-anchor pairs; a layer function for selecting the target layers (i.e., LIDs) of the tests; and a unique name for the summary definition. Both the anchor input parameter and test input parameter have a global version, that is used for all comparison types, and a comparison type specific version for specifying different tests to different comparison types. The test filter function is given an anchor test name and an input test name, and it should return true if the anchor-test pair should be included in the summary. The layer function takes in input test names and should return an iterable object, containing either name-LID pairs or just the input name.

To help with creating more complex layer functions, the `layerFuncFactory`-function is provided. It allow the user to specify several LID lists with a matching condition function. For a given test name passed to the condition function, the list of LIDs is used as the layers for the given test if the condition function returns true.

The multi-anchor function takes in similar parameters to the single-anchor variant but with a few changes. Firstly, the anchor parameters are changed to take in a function instead of a test name. Secondly, the filter and the layer function parameters are extended to have comparison specific versions. Lastly, all input tests are given in a single parameter. The anchor function is the only new kind of parameter. It should take in either the name of an input test or a name-LID pair. The return value — an iterable object containing valid anchor values — is then used as the anchor for the respective test.

For simplifying the anchor function creation, the `anchorFuncFactory_match_layer`-function is provided. It takes in a function that only accepts test names and returns the anchor information; the anchor function factory wraps the provided function, making it a proper anchor function that allows name-LID pair inputs as well. The function, that is created, sets an input anchor test's layer to match the input test's layer, if the input test's layer is specified. This allows creating anchor functions without having separate cases for each name-LID pair.

The `make_CurveChart_definition`-function can be used to create **Curve_chart** definitions. It takes in a list of test names and a layering function similarly to the **Anchor_list** definition generation functions. Additionally, the input test names can be filtered with a filter function. The helper function supports two types of charts: bit rate versus quality curves (**br_curve**) and bit rate versus time complexity curves (**time_curve**). These two types of charts are the most relevant and likely to provide logical results because of the way the data-points are generated by varying QP.

5.3 `tests`-package

The `tests`-package collects user-created test scripts. The user may run and save test scripts anywhere, but the `tests`-package provides a simple user interface for selecting and running tests using just their filename (i.e., no need to give exact file paths).

When the `__main__` module of the `tests`-package is executed, it goes through all modules currently in the `tests` folder, adding them to a list. This means any test scripts that are added to the folder get automatically included, requiring no modification of the `tests`-package when adding new test scripts. The scripts, that are found, can then be run by giving the script name as a command-line parameter when executing the `tests`-package. Alternatively, an interactive prompt can be used to specify test names when no command-line parameters are given.

6 PERFORMANCE EVALUATION

This chapter presents a quantitative evaluation of scalable encoding. Section 6.1 compares the coding efficiency and complexity of Scalable Kvazaar and SHM [15], whereas Section 6.2 provides a wider inspection of the effects of the scaling ratio and ΔQP on the performance of SHVC.

6.1 Performance Results

This section presents the performance results from tests described in Section 3.2. Results are calculated for two ΔQP values and seven sequences, as described in Section 3.2.3. Coding efficiency results are discussed in Section 6.1.1, whereas complexity is analyzed in Section 6.1.2.

6.1.1 Coding Performance Analysis

Table 6.1 tabulates the SNR BD-rate (see Section 3.2.1) results, for Scalable Kvazaar, where Kvazaar simulcast coding is used as the anchor for the BD-rate calculation. Between the two ΔQP values, Scalable Kvazaar is able to achieve an average bit rate improvement of $(12.44\% + 19.88\%)/2 = 16.16\%$. When only comparing the ELs, the average improvement is 23.02%. Table 6.2 shows the equivalent results for SHM compared with HM simulcast coding. SHM provides an average bit rate improvement of 14.21% and 22.56% for EL-only, giving similar results as Scalable Kvazaar.

When examining the individual results of the two ΔQP tests more closely, it can be seen that a smaller absolute ΔQP value results in a higher bit rate improvement. The change in BD-rate between the different ΔQP values is due to the fact that, with a large QP gap, it becomes harder to find low coding cost blocks from the BL because a higher QP results in larger distortion, which degrades the quality. With a smaller QP difference, however, the qualities of the BL and EL converge, allowing the EL to use blocks from the BL more efficiently, thereby obtaining higher BD-rate savings.

Similarly to the SNR results, Tables 6.3 and 6.4 tabulate the spatial scalability results for Scalable Kvazaar and SHM, respectively. Results are given for the scaling ratios of $2\times$ and $1.5\times$. With a $2\times$ spatial scaling ratio (Table 6.3a), Scalable Kvazaar gives an

Table 6.1. *Coding gain and speedup of Scalable Kvazaar over Kvazaar simulcast coding when using SNR scalability.*

	$\Delta QP = -6$				$\Delta QP = -4$			
	BD-rate		ΔSpeed	Speed	BD-rate		ΔSpeed	Speed
Sequence	*BL+EL*	*EL*			*BL+EL*	*EL*		
Traffic	-4.93%	-3.83%	1.11×	29 fps	-11.55%	-16.51%	1.16×	31 fps
PeopleOnStreet	-17.52%	-24.09%	1.13×	19 fps	-27.89%	-45.62%	1.18×	21 fps
Kimono	-14.90%	-20.52%	1.19×	50 fps	-23.47%	-37.73%	1.26×	55 fps
ParkScene	-10.90%	-14.12%	1.17×	48 fps	-17.33%	-26.17%	1.22×	52 fps
Cactus	-12.95%	-15.30%	1.22×	48 fps	-19.46%	-28.63%	1.26×	54 fps
BasketballDrive	-16.26%	-20.44%	1.22×	45 fps	-25.09%	-38.91%	1.28×	49 fps
BQTerrace	-9.62%	-10.82%	1.17×	43 fps	-14.39%	-19.50%	1.20×	48 fps
Average	-12.44%	-15.59%	1.17×		-19.88%	-30.44%	1.22×	

Table 6.2. *Coding gain and speedup of SHM over HM simulcast coding when using SNR scalability.*

	$\Delta QP = -6$			$\Delta QP = -4$		
	BD-rate		ΔSpeed	BD-rate		ΔSpeed
Sequence	*BL+EL*	*EL*		*BL+EL*	*EL*	
Traffic	-8.33%	-15.34%	0.96×	-12.67%	-25.52%	0.97×
PeopleOnStreet	-15.39%	-22.22%	0.99×	-25.23%	-41.31%	1.02×
Kimono	-12.22%	-18.68%	0.98×	-20.96%	-35.43%	1.00×
ParkScene	-8.42%	-14.53%	0.96×	-14.06%	-25.98%	0.97×
Cactus	-9.73%	-13.51%	0.97×	-16.88%	-27.26%	0.98×
BasketballDrive	-12.73%	-16.92%	0.98×	-21.91%	-33.75%	1.00×
BQTerrace	-7.18%	-7.97%	0.96×	-13.17%	-17.41%	0.97×
Average	-10.57%	-15.59%	0.97×	-17.84%	-29.52%	0.99×

average BD-rate decrease of 13.12%, whereas the $1.5\times$ ratio (Table 6.3b) gives a 23.19% decrease. The respective results for EL-only are 16.85% for the $2\times$ ratio and 37.02% for the $1.5\times$ ratio. Again, these results are inline with those of the SHM with the full and EL-only BD-rates being -13.58% and -21.66% for the $2\times$ ratio (Table 6.4a). Similarly, for the $1.5\times$ ratio (Table 6.4b), the respective values are -24.20% and -42.97%.

Even with the loss of detail from upsampling an already downsampled image, spatial scalability is able to reach a notable level of bit rate reduction when compared with simulcast, albeit not quite as high as with just SNR scalability. Moreover, as can be expected, with a $1.5\times$ scaling ratio, the BD-rate improvement is higher than with a $2\times$ ratio. This is due to the $1.5\times$ ratio's upsampling having comparatively more information to work with, resulting in a more accurate reference for the EL. Another thing that can be noted from the results is that increasing the QP on the EL results in a larger BD-rate improvement

Table 6.3. *Coding gain and speedup of Scalable Kvazaar over Kvazaar simulcast coding when using spatial scalability.*

(a) *Scaling ratio of* $2\times$

	$\Delta QP = 0$				$\Delta QP = 2$			
	BD-rate		ΔSpeed	Speed	BD-rate		ΔSpeed	Speed
Sequence	*BL+EL*	*EL*			*BL+EL*	*EL*		
Traffic	-3.75%	-1.49%	0.91$\times$	35 fps	-7.16%	-8.40%	0.92$\times$	36 fps
PeopleOnStreet	-15.36%	-21.27%	1.00$\times$	24 fps	-23.54%	-39.56%	1.06$\times$	27 fps
Kimono	-17.10%	-21.88%	1.06$\times$	63 fps	-23.46%	-34.56%	1.10$\times$	68 fps
ParkScene	-9.05%	-10.42%	1.00$\times$	59 fps	-13.90%	-18.92%	1.02$\times$	63 fps
Cactus	-10.37%	-9.17%	1.02$\times$	60 fps	-15.09%	-18.50%	1.05$\times$	65 fps
BasketballDrive	-13.58%	-14.42%	1.07$\times$	57 fps	-19.97%	-26.66%	1.12$\times$	62 fps
BQTerrace	-4.86%	-4.05%	0.99$\times$	54 fps	-6.49%	-6.55%	1.00$\times$	60 fps
Average	-10.58%	-11.81%	1.01$\times$		-15.66%	-21.88%	1.04$\times$	

(b) *Scaling ratio of* $1.5\times$

	$\Delta QP = 0$				$\Delta QP = 2$			
	BD-rate		ΔSpeed	Speed	BD-rate		ΔSpeed	Speed
Sequence	*BL+EL*	*EL*			*BL+EL*	*EL*		
Kimono	-26.50%	-40.90%	1.08$\times$	58 fps	-34.47%	-59.90%	1.15$\times$	63 fps
ParkScene	-16.79%	-24.59%	1.02$\times$	54 fps	-25.56%	-44.19%	1.07$\times$	58 fps
Cactus	-20.36%	-28.29%	1.07$\times$	56 fps	-28.58%	-49.42%	1.12$\times$	61 fps
BasketballDrive	-23.09%	-32.83%	1.11$\times$	53 fps	-30.86%	-52.44%	1.17$\times$	57 fps
BQTerrace	-10.14%	-12.61%	1.01$\times$	49 fps	-15.51%	-25.03%	1.03$\times$	54 fps
Average	-19.38%	-27.84%	1.06$\times$		-27.00%	-46.19%	1.11$\times$	

than with equal QPs between the layers. The higher QP, in a sense, compensates for the distortion, introduced in the upsampling process, by lowering the quality requirements of the EL, allowing the ILR to be used more readily.

Appendix B contains figures visualizing the individual QP results — used to calculate the BD-rate values — for all tests described above. These figures provide a more detailed picture of the coding efficiency but are less useful for gauging the quantitative coding efficiency.

6.1.2 Coding Complexity Analysis

The encoding speed results give a multiplier, relative to the anchor, and an absolute FPS-value. The relative coding speed multiplier has been calculated by taking the average encoding time — over the QP range used for BD-rate calculations — and comparing it to the average encoding time of the anchor; a smaller encoding time means faster speed and results in a multiplier that is greater than one. This method may not give the most accurate results, since the bit rate and quality are not the same between the two encodings being

Table 6.4. *Coding gain and speedup of SHM over HM simulcast coding when using spatial scalability.*

(a) *Scaling ratio of* $2\times$

	$\Delta QP = 0$			$\Delta QP = 2$		
	BD-rate		ΔSpeed	BD-rate		ΔSpeed
Sequence	*BL+EL*	*EL*		*BL+EL*	*EL*	
Traffic	-8.02%	-15.03%	0.94×	-10.40%	-21.94%	0.95×
PeopleOnStreet	-15.14%	-22.39%	0.98×	-24.72%	-41.06%	1.04×
Kimono	-13.97%	-21.05%	0.98×	-24.90%	-41.99%	1.03×
ParkScene	-7.64%	-12.87%	0.94×	-11.46%	-20.83%	0.96×
Cactus	-9.76%	-14.18%	0.95×	-16.05%	-25.70%	0.98×
BasketballDrive	-11.27%	-14.45%	0.97×	-19.67%	-28.99%	1.01×
BQTerrace	-6.77%	-7.85%	0.93×	-10.34%	-14.93%	0.93×
Average	-10.37%	-15.40%	0.96×	-16.79%	-27.92%	0.99×

(b) *Scaling ratio of* $1.5\times$

	$\Delta QP = 0$			$\Delta QP = 2$		
	BD-rate		ΔSpeed	BD-rate		ΔSpeed
Sequence	*BL+EL*	*EL*		*BL+EL*	*EL*	
Kimono	-25.97%	-44.04%	1.03×	-41.17%	-80.75%	1.13×
ParkScene	-14.25%	-25.94%	0.97×	-25.59%	-51.91%	1.02×
Cactus	-18.51%	-29.78%	0.99×	-31.02%	-58.72%	1.05×
BasketballDrive	-22.41%	-33.57%	1.01×	-34.29%	-61.50%	1.07×
BQTerrace	-10.48%	-13.27%	0.95×	-18.28%	-30.20%	0.96×
Average	-18.32%	-29.32%	0.99×	-30.07%	-56.62%	1.04×

compared, but it should still give a general idea of the relative coding speed. Similarly to the relative encoding speed, the absolute encoding speeds have been calculated as an average over the QP test range. Additionally, Appendix B provides figures for all the per-QP encoding times of (Scalable) Kvazaar.

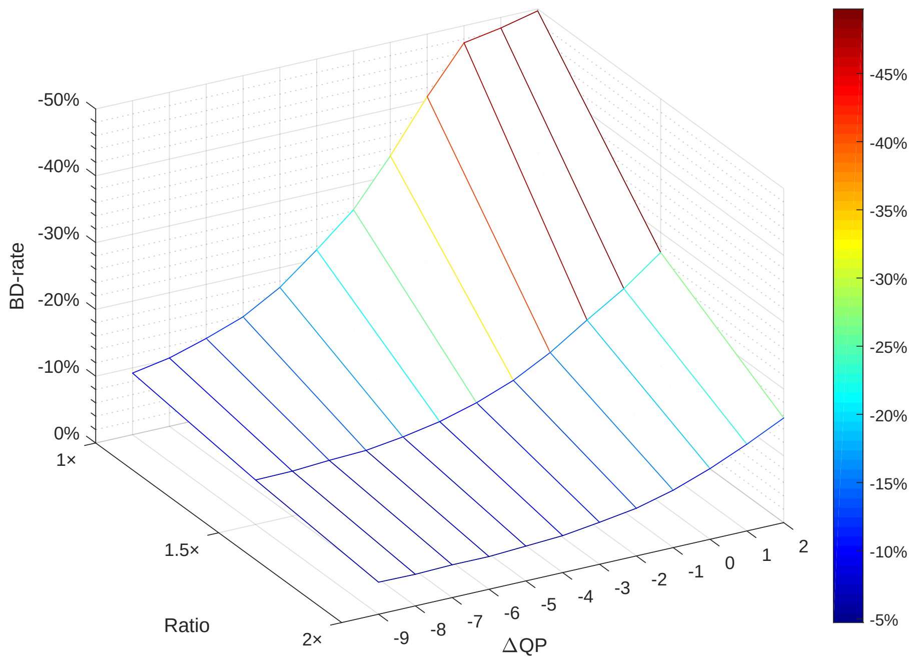
The relative encoding speed results for Scalable Kvazaar, using SNR scalability, in Table 6.1 show a significant speedup when compared with simulcast coding speeds. An average speedup of $(1.17\times + 1.22\times)/2 = 1.20\times$ is achieved over the two ΔQPs. Scalable Kvazaar is able to efficiently pipeline the BL and EL encoders; since the layer resolutions match, their processing times are well balanced and neither layer's encoder dominates the encoding workload. Moreover, encoding both layers at the same time mitigates the inherent limitations of WPP that would normally limit the number of tasks available at the beginning and the end of a frame. Scalable encoding more or less doubles the available number of tasks at any given time. SHM, on the other hand, lacks optimization and succumbs to the overhead introduced by scalable coding, resulting in an average ΔSpeed of $0.98\times$ (Table 6.2). As for absolute encoding speed, using the optimizations detailed in Section 4.2.4, Scalable Kvazaar reaches real-time encoding speeds, in most sequences, and over 40 FPS in all 1080p sequences. With SHM, however, the absolute encoding speeds have been omitted, since they are in the order of minutes-per-frame.

When using spatial scalability, the speedup of Scalable Kvazaar is more modest than with SNR scalability, but it is still able to maintain similar speeds as simulcast coding, despite the added inter-layer processing. As seen in Table 6.3a, the average speedup of Scalable Kvazaar with a $2\times$ scaling ratio is $1.03\times$ and, from Table 6.3b, the average speedup with a $1.5\times$ scaling ratio is $1.09\times$. With spatial scalability, the layer resolutions differ, causing the processing times of the BL and EL encoders to be unbalanced. This hinders the encoding pipeline, decreasing efficiency, since the BL encoder is stuck waiting for the EL encoder. With the scaling ratio of $1.5\times$, the speedup ends up being slightly higher, when compared with the $2\times$ scaling ratio results, because it is not quite as unbalanced as the $2\times$ case. With spatial scalability, it should be noted, based on the per-QP results shown in Appendix B, that Kvazaar simulcast coding seems to be faster in the higher end of the QP range, but Scalable Kvazaar is faster in the lower end. Similarly to the SNR case, SHM suffers an overall slowdown of $0.98\times$, when using $2\times$ spatial scalability, but with a $1.5\times$ ratio, manages a slight speedup of $1.02\times$. With spatial scalability, Scalable Kvazaar is mostly able to encode at above 30 FPS, and on 1080p sequences, it reaches encoding speeds of over 50 FPS in all but one sequence.

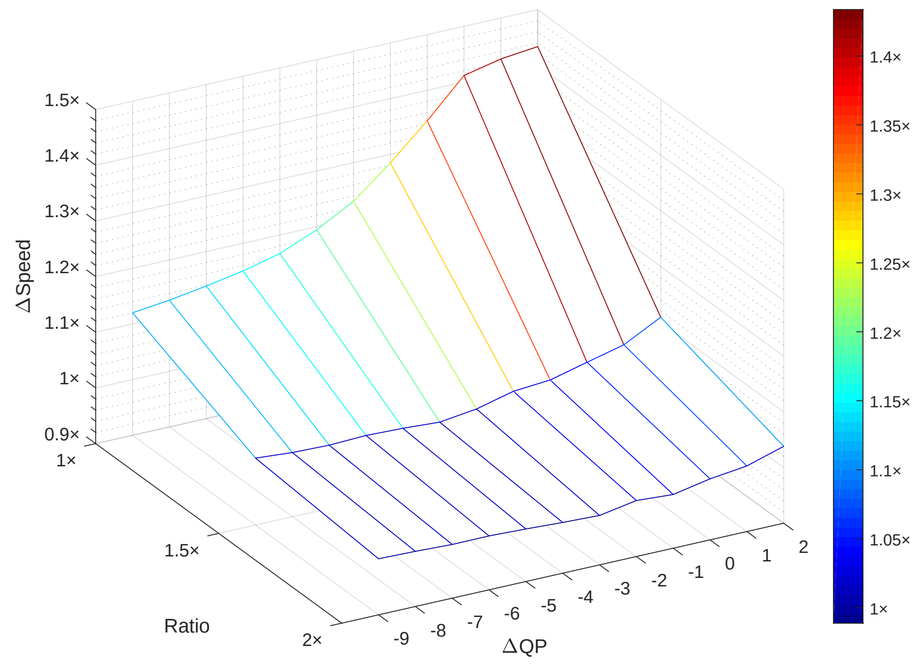
6.2 SHVC Parameter-Space Exploration

To get a better idea of the effects of ΔQP and the scaling ratio on the coding performance of SHVC, a more thorough investigation was performed. The measurements were carried out using Scalable Kvazaar as its fast encoding speeds allow running extensive tests in a reasonable time-frame. In total, twelve ΔQP values, ranging from -9 to 2, were tested in combination with three scaling ratios ($1\times$, $1.5\times$, and $2\times$). Here, the BL QPs, used for all tests, were the spatial BL QP values from Table 3.2. This is to make the qp ranges consistent between SNR and spatial scalability tests. Bar the different BL QPs, the parameter-space exploration tests were performed as described in Section 3.2. The averages from these tests are collected in Figure 6.1 that shows BD-rate improvements and the relative encoding speedup.

Figure 6.1a contains the BD-rate results for Scalable Kvazaar versus Kvazaar simulcast coding. It can be seen that the BD-rate improvements start increasing quickly as ΔQP increases. This is especially true for SNR scalability, since the BL and EL qualities start converging, as was mentioned in Section 6.1.1. When spatial scalability is used, the BD-



***(a)** BD-rate improvements over different ΔQP and scaling ratios*



***(b)** Delta speed over different ΔQP and scaling ratios*

***Figure 6.1.** Parameter-space exploration of scalable encoding.*

rate improvements are somewhat diminished, especially with a $2\times$ scaling ratio, as is to be expected due to the distortion introduced by upsampling. With the negative ΔQP values spatial scalability BD-rate improvements are almost constant. On the other hand, spatial scalability sees a notable benefit from positive ΔQP values, since the decreased quality for the EL compensates for the upsampling distortion.

Figure 6.1b contains the relative encoding speed results for Scalable Kvazaar versus Kvazaar simulcast coding. For SNR scalability, the speedup increases almost exponentially as the ΔQP increases, similarly to the respective BD-rate results. For spatial scalability, due to the added inter-layer processing, the relative speed stays fairly close to one throughout the ΔQP range but starts picking up with the positive ΔQP values. It seems that, at least in the case of Scalable Kvazaar using the ultrafast preset, SNR scalability can achieve considerable speedups when BD-rate increases. However, spatial scalability is limited by inter-layer processing even when BD-rate gains are increased, resulting in the relative speedup staying almost constant when ΔQP is changed.

7 CONCLUSION

This thesis presented an overview of scalable video coding as well as Scalable Kvazaar — an open-source SHVC encoder capable of real-time encoding. First, the basic concepts of video coding, specifically HEVC, were briefly covered, and scalability features, included in SHVC, were introduced. Some common techniques, for analyzing said video technologies, were also described. Next, this thesis gave a technical overview of Kvazaar followed by the implementation details of Scalable Kvazaar. Then, to measure the performance of Scalable Kvazaar, a scalability testing framework was described. Finally, the results from tests, performed on Scalable Kvazaar and SHM, were presented and analyzed.

Section 7.1 further presents the main contributions of this work and summarizes the main results. Section 7.2 goes over future work and possible development directions for Scalable Kvazaar.

7.1 Main Contributions

The main contributions of this work involved implementing quality and spatial scalability features in a practical encoder, resulting in Scalable Kvazaar. Moreover, the encoding process of Scalable Kvazaar was optimized using WPP, OWF, and SIMD parallelization techniques, making it possible to reach real-time encoding speeds. At the time of writing, Scalable Kvazaar is the only practical open-source SHVC encoder in existence.

Table 7.1 collects the average BD-rate results from all conducted tests. As can be seen, when comparing scalable coding to simulcast coding, the average bit rate savings of Scalable Kvazaar range from 10.58% to 27.00%. The respective results for SHM are similar in terms of magnitude, validating the Scalable Kvazaar results.

Kvazaar is capable of real-time encoding speeds. Similarly, through the aforementioned optimizations, Scalable Kvazaar is able to reach real-time encoding speeds despite the added inter-layer processing required by scalable coding. Table 7.1, likewise, summarizes the relative encoding speed results of scalable coding versus simulcast coding. Scalable Kvazaar can be seen to provide notable speedups, ranging from $1.01\times$ to $1.22\times$. The un-optimized SHM, however, suffers from scalable encoding overhead and is, on average,

Table 7.1. *Average BD-rate and relative speed results for Scalable Kvazaar and SHM.*

		Spatial				Quality	
		$\Delta QP = 0$		$\Delta QP = 2$		$\Delta QP = -6$	$\Delta QP = -4$
		$2\times$	$1.5\times$	$2\times$	$1.5\times$		
Scalable Kvazaar	BD-Rate	-10.58%	-19.38%	-15.66%	-27.00%	-12.44%	-19.88%
	ΔSpeed	1.01$\times$	1.06$\times$	1.04$\times$	1.11$\times$	1.17$\times$	1.22$\times$
SHM	BD-Rate	-10.37%	-18.32%	-16.79%	-30.07%	-10.57%	-17.84%
	ΔSpeed	0.96$\times$	0.99$\times$	0.99$\times$	1.04$\times$	0.97$\times$	0.99$\times$

Table 7.2. *Absolute encoding speed results for Scalable Kvazaar.*

	Spatial				Quality	
	$\Delta QP = 0$		$\Delta QP = 2$		$\Delta QP = -6$	$\Delta QP = -4$
Sequence	$2\times$	$1.5\times$	$2\times$	$1.5\times$		
Traffic	35 fps	-	36 fps	-	29 fps	31 fps
PeopleOnStreet	24 fps	-	27 fps	-	19 fps	21 fps
Kimono	63 fps	58 fps	68 fps	63 fps	50 fps	55 fps
ParkScene	59 fps	54 fps	63 fps	58 fps	48 fps	52 fps
Cactus	60 fps	56 fps	65 fps	61 fps	48 fps	54 fps
BasketballDrive	57 fps	53 fps	62 fps	57 fps	45 fps	49 fps
BQTerrace	54 fps	49 fps	60 fps	54 fps	43 fps	48 fps

slower than simulcast coding in most cases.

As seen from Table 7.2, Scalable Kvazaar reaches real-time encoding speeds on 1080p video sequences in all test cases. This means Scalable Kvazaar can be used in real-time application, such as video conferencing, if scalable coding is going to be used.

7.2 Future Work

The current focus of Scalable Kvazaar is on real-time encoding and, as such, has not been extensively tested with all coding tools in Kvazaar. More work would likely be needed to fully utilize all of the encoding tools. Presently, Scalable Kvazaar only fully supports one EL, so adding support for multiple ELs would be another possibility for future work. Moreover, Scalable Kvazaar does not implement all scalability types, introduced by SHVC, such as bit depth and color gamut scalability, opening another avenue for further development. However, more support for variable bit depths and color spaces would be required from the Kvazaar side.

Another path for future development could be to implement and investigate speedup techniques proposed in various papers [55, 56, 57, 58, 59]. Bailleul et al. [55] present simple early termination methods for skipping rarely used modes etc. in the EL. On the other hand, Wang et al. [56, 57] use more complicated content adaptive statistical modeling to predict the CU modes and depths for the EL when using SNR scalability. Shen and Feng [59] expand the statistical method to spatial scalabitily, as well. Additionally, Shen et al. [58] present a neural network–based solution for predicting coding information. These methods are able to cut SHM encoding speed in halve — some even more than halve. Regardless, it remains unclear how well they would work with a practical encoder.

Since Scalable Kvazaar provides a practical SHVC encoder, it would be possible to integrate it into various applications that could benefit from scalability. In the field of teleconferencing, one such candidate would be Kvazzup [60]. It is an open-source video call software that uses Kvazaar for video encoding tasks. Since Kvazzup already has Kvazaar integration, moving to Scalable Kvazaar would be trivial, once scalability support is added on the Kvazzup side.

Finally, moving beyond HEVC, the scalable encoding techniques, presented in this thesis, could be applied to other, future, codecs. One such promising video coding standard is *Versitile Video Coding* (VVC) [61] — the successor to HEVC. In VVC, the quality and spatial scalability features of SHVC have already been carried over, meaning that the techniques, and even implementation, of Scalable Kvazaar could be directly transferred to a practical VVC encoder.

REFERENCES

[1] Cisco. *Cisco Visual Networking Index: Forecast and Trends, 2017-2022*. Dec. 2018.

[2] Parameter Values for UHDTV Systems for Production and International Program Exchange. *document ITU-R Rec. BT.2020-2*. ITU-T, Oct. 2015.

[3] Cisco. *Cisco Annual Internet Report (2018-2020)*. Mar. 2020.

[4] High Efficiency Video Coding. *document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC)*. ITU-T and ISO/IEC, Nov. 2019.

[5] Sullivan, G. J., Ohm, J., Han, W. and Wiegand, T. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (Nov. 2012), 1649–1668. URL: https://ieeexplore.ieee.org/document/6316136.

[6] Sullivan, G. J., Boyce, J. M., Chen, Y., Ohm, J., Segall, C. A. and Vetro, A. Standardized Extensions of High Efficiency Video Coding (HEVC). *IEEE Journal of Selected Topics in Signal Processing* 7.6 (Nov. 2013), 1001–1016. URL: https://ieeexplore.ieee.org/document/6630053.

[7] Boyce, J. M., Ye, Y., Chen, J. and Ramasubramonian, A. K. Overview of SHVC: Scalable Extensions of the High Efficiency Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology* 26.1 (Jan. 2016), 20–34. URL: https://ieeexplore.ieee.org/document/7172510.

[8] Ye, Y., He, Y. and Xiu, X. Manipulating Ultra-High Definition Video Traffic. *IEEE MultiMedia* 22.3 (July 2015), 73–81. URL: https://ieeexplore.ieee.org/document/7021856.

[9] Ronan, P., Eric, T. and Mickaël, R. Hybrid Broadband/Broadcast ATSC 3.0 SHVC Distribution Chain. *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. Valencia, Spain, June 2018, 1–5. URL: https://ieeexplore.ieee.org/document/8436752.

[10] Nightingale, J., Wang, Q. and Grecos, C. Scalable HEVC (SHVC)-Based video stream adaptation in wireless networks. *2013 IEEE 24th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. London, UK, July 2013, 3573–3577. URL: https://ieeexplore.ieee.org/document/6666769.

[11] Ultra Video Group. *Kvazaar open-source HEVC Encoder*. URL: https://github.com/ultravideo/kvazaar/.

[12] MulticoreWare Inc. *x265 HEVC Encoder / H.265 Video Codec*. URL: https://bitbucket.org/multicoreware/x265_git/downloads/.

[13] 01.org. *Scalable Video Technology for HEVC Encoder (SVT-HEVC Encoder)*. URL: https://github.com/OpenVisualCloud/SVT-HEVC/.

[14] Turingcodec.org. *Turing codec*. URL: http://turingcodec.org.

[15] JCT-VC. *SHVC Reference Software, ver. SHM 12.1*. URL: http://hevc.hhi.fraunhofer.de/shvc.

[16] Parois, R., Hamidouche, W., Cabarat, P.-L., Raulet, M., Sidaty, N. and Deforges, O. 4K Real Time Software Solution of Scalable HEVC for Broadcast Video Application. *IEEE Access* 7 (2019), 46748–46762. URL: https://ieeexplore.ieee.org/document/8664002.

[17] Hamidouche, W., Raulet, M. and Deforges, O. 4K Real-Time and Parallel Software Video Decoder for Multilayer HEVC Extensions. *IEEE Transactions on Circuits and Systems for Video Technology* 26.1 (Jan. 2016), 169–180. URL: https://ieeexplore.ieee.org/document/7273890.

[18] Laitinen, J., Lemmetti, A. and Vanne, J. Real-Time Implementation Of Scalable Hevc Encoder. *2020 IEEE International Conference on Image Processing (ICIP)*. Abu Dhabi, UAE, Oct. 2020, 1166–1170. URL: https://ieeexplore.ieee.org/document/9191135.

[19] Ultra Video Group. *Scalable Kvazaar*. URL: https://github.com/ultravideo/scalable-kvazaar/.

[20] Erol, B., Kossentini, F., Joch, A., Sullivan, G. J. and Winger, L. CHAPTER 10 - MPEG-4 Visual and H.264/AVC: Standards for Modern Digital Video. *The Essential Guide to Video Processing*. Ed. by A. Bovik. Boston: Academic Press, 2009, 295–330. URL: http://www.sciencedirect.com/science/article/pii/B9780123744562000153.

[21] Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-screen 16:9 Aspect Ratios. *document ITU-R Rec. BT.601-6*. ITU-R, Mar. 2011.

[22] Clare, G., Henry, F. and Pateux, S. Wavefront parallel processing for HEVC encoding and decoding. *document JCTVC-F274*. Torino, Italy, July 2011.

[23] Rao, K. R. and Yip, P. *Discrete cosine transform: algorithms, advantages, applications*. Academic press, 2014.

[24] Norkin, A., Bjøntegaard, G., Fuldseth, A., Narroschke, M., Ikeda, M., Andersson, K., Zhou, M. and Van der Auwera, G. HEVC Deblocking Filter. *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (Dec. 2012), 1746–1754. URL: https://ieeexplore.ieee.org/document/6324414.

[25] Fu, C., Chen, C., Huang, Y. and Lei, S. Sample adaptive offset for HEVC. *2011 IEEE 13th International Workshop on Multimedia Signal Processing*. Hangzhou, China, Oct. 2011, 1–5. URL: https://ieeexplore.ieee.org/document/6093807.

[26] Langdon, G. G. An Introduction to Arithmetic Coding. *IBM Journal of Research and Development* 28.2 (Mar. 1984), 135–149.

[27] Rissanen, J. and Langdon, G. G. Arithmetic Coding. *IBM Journal of Research and Development* 23.2 (Mar. 1979), 149–162.

[28] Marpe, D., Schwarz, H. and Wiegand, T. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (July 2003), 620–636. URL: `https://ieeexplore.ieee.org/document/1218195`.

[29] Lee, J.-y., Park, S.-I., Kwon, S., Lim, B.-M., Kim, H. M., Hur, N., Pesin, A., Chevet, J.-C., Llach, J., Stein, A. J., Jeon, S. and Wu, Y. Efficient Transmission of Multiple Broadcasting Services Using LDM and SHVC. *IEEE Transactions on Broadcasting* 64.2 (June 2018), 177–187. URL: `https://ieeexplore.ieee.org/document/8063347`.

[30] Xu, Y., Yu, C., Li, J. and Liu, Y. Video Telephony for End-Consumers: Measurement Study of Google+, IChat, and Skype. *Proceedings of the 2012 Internet Measurement Conference*. IMC '12. Boston, Massachusetts, USA, 2012, 371–384. URL: `https://doi.org/10.1145/2398776.2398816`.

[31] JCT-VC. *HEVC Reference Software, ver. HM 16.10*. URL: `http://hevc.hhi.fraunhofer.de/`.

[32] Ohm, J., Sullivan, G. J., Schwarz, H., Tan, T. K. and Wiegand, T. *Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC)*. Dec. 2012. URL: `https://ieeexplore.ieee.org/document/6317156`.

[33] Vanne, J., Viitanen, M., Hämäläinen, T. D. and Hallapuro, A. Comparative Rate-Distortion-Complexity Analysis of HEVC and AVC Video Codecs. *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (Nov. 2012), 1885–1898. URL: `https://ieeexplore.ieee.org/document/6324420`.

[34] Tan, T. K., Mrak, M., Baroncini, V. and Ramzan, N. Report on HEVC compression performance verification testing. *document JCTVC-Q1011*. Valencia, Spain, May 2014.

[35] Bjøntegaard, G. Calculation of average PSNR differences between RD-curves. *document VCEG-M33*. Austin, Texas, USA, Apr. 2001.

[36] J. Wang, X. Y. and He, D. On BD-rate calculation. *document JCTV-F270*. Torino, Italy, July 2011.

[37] Eskicioglu, A. M. and Fisher, P. S. Image quality measures and their performance. *IEEE Transactions on Communications* 43.12 (Dec. 1995), 2959–2965. URL: `http://ieeexplore.ieee.org/document/477498`.

[38] Eckert, M. P. and Bradley, A. P. Perceptual quality metrics applied to still image compression. *Signal Processing* 70.3 (1998), 177–200. URL: `https://doi.org/10.1016/S0165-1684(98)00124-8`.

[39] Zhou Wang and Bovik, A. C. A universal image quality index. *IEEE Signal Processing Letters* 9.3 (Mar. 2002), 81–84. URL: https://ieeexplore.ieee.org/document/995823.

[40] Mannos, J. and Sakrison, D. The effects of a visual fidelity criterion of the encoding of images. *IEEE Transactions on Information Theory* 20.4 (July 1974), 525–536. URL: https://ieeexplore.ieee.org/document/1055250.

[41] Pappas, T. N., Safranek, R. J. and Chen, J. 8.2 - Perceptual Criteria for Image Quality Evaluation. *Handbook of Image and Video Processing (Second Edition)*. Ed. by A. BOVIK. Second Edition. Communications, Networking and Multimedia. Burlington: Academic Press, 2005, 939–959. URL: http://www.sciencedirect.com/science/article/pii/B9780121197926501182.

[42] Zhou Wang, Bovik, A. C., Sheikh, H. R. and Simoncelli, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13.4 (Apr. 2004), 600–612. URL: https://ieeexplore.ieee.org/document/1284395.

[43] Laitinen, J. *NAVETTA*. URL: https://github.com/MrAsura/NAVETTA.

[44] Seregin, V. and He, Y. Common SHM test conditions and software reference configurations. *document JCTVC-Q1009*. Valencia, Spain, May 2014.

[45] Viitanen, M., Koivula, A., Lemmetti, A., Ylä-Outinen, A., Vanne, J. and Hämäläinen, T. Kvazaar: Open-Source HEVC/H.265 Encoder. *Proceedings of the 24th ACM International Conference on Multimedia*. MM '16. Amsterdam, The Netherlands, Oct. 2016, 1179–1182. URL: http://dl.acm.org/citation.cfm?id=2973796.

[46] *Ultra Video Group*. URL: http://ultravideo.fi/.

[47] Viitanen, M., Koivula, A., Lemmetti, A., Vanne, J. and Hämäläinen, T. D. Kvazaar HEVC encoder for efficient intra coding. *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. Lisbon, Portugal, May 2015, 1662–1665. URL: http://ieeexplore.ieee.org/document/7168970.

[48] Koivula, A., Viitanen, M., Vanne, J., Hämäläinen, T. D. and Fasnacht, L. Parallelization of Kvazaar HEVC intra encoder for multi-core processors. *2015 IEEE Workshop on Signal Processing Systems (SiPS)*. Hangzhou, China, Oct. 2015, 1–6. URL: https://ieeexplore.ieee.org/document/7345015.

[49] Lemmetti, A., Koivula, A., Viitanen, M., Vanne, J. and Hämäläinen, T. D. AVX2-optimized Kvazaar HEVC intra encoder. *2016 IEEE International Conference on Image Processing (ICIP)*. Phoenix, Arizona, USA, Sept. 2016, 549–553. URL: https://ieeexplore.ieee.org/document/7532417.

[50] Lemmetti, A., Kallio, E., Viitanen, M., Vanne, J. and Hämäläinen, T. Rate-Distortion-Complexity Optimized Coding Scheme for Kvazaar HEVC Intra Encoder. *2018 Data Compression Conference*. Snowbird, Utah, USA, Mar. 2018, 419. URL: https://ieeexplore.ieee.org/document/8416636.

[51] Lemmetti, A., Viitanen, M., Mercat, A. and Vanne, J. Kvazaar 2.0: Fast and Efficient Open-Source HEVC Inter Encoder. *Proceedings of the 11th ACM Multimedia Systems Conference*. MMSys '20. Istanbul, Turkey, May 2020, 237–242. URL: https://doi.org/10.1145/3339825.3394927.

[52] Ylä-Outinen, A., Lemmetti, A., Viitanen, M., Vanne, J. and Hämäläinen, T. Kvazaar: HEVC/H.265 4K30p Intra Encoder. *2017 IEEE International Symposium on Multimedia (ISM)*. Taichung, Taiwan, Dec. 2017, 362–363. URL: https://ieeexplore.ieee.org/document/8241634.

[53] Ce Zhu, Xiao Lin and Lap-Pui Chau. Hexagon-based search pattern for fast block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology* 12.5 (May 2002), 349–355. URL: https://ieeexplore.ieee.org/document/1003474.

[54] Chi, C. C., Alvarez-Mesa, M., Juurlink, B., George, V. and Schierl, T. Improving the parallelization efficiency of HEVC decoding. *2012 19th IEEE International Conference on Image Processing*. Orlando, Florida, USA, Sept. 2012, 213–216. URL: https://ieeexplore.ieee.org/document/6466833.

[55] Bailleul, R., De Cock, J. and Van De Walle, R. Fast mode decision for SNR scalability in SHVC digest of technical papers. *2014 IEEE International Conference on Consumer Electronics (ICCE)*. Las Vegas, Nevada, USA, Jan. 2014, 193–194. URL: https://ieeexplore.ieee.org/document/6775968.

[56] Wang, D., Sun, Y., Zhu, C., Li, W. and Dufaux, F. Fast Depth and Inter Mode Prediction for Quality Scalable High Efficiency Video Coding. *IEEE Transactions on Multimedia* 22.4 (Apr. 2020), 833–845. URL: https://ieeexplore.ieee.org/document/8811621.

[57] Wang, D., Sun, Y., Zhu, C., Li, W., Dufaux, F. and Luo, J. Fast Depth and Mode Decision in Intra Prediction for Quality SHVC. *IEEE Transactions on Image Processing* 29 (Apr. 2020), 6136–6150. URL: https://ieeexplore.ieee.org/document/9075415.

[58] Shen, L., Feng, G. and An, P. SHVC CU Processing Aided by a Feedforward Neural Network. *IEEE Transactions on Industrial Informatics* 15.11 (Nov. 2019), 5803–5815. URL: https://ieeexplore.ieee.org/document/8693569.

[59] Shen, L. and Feng, G. Content-Based Adaptive SHVC Mode Decision Algorithm. *IEEE Transactions on Multimedia* 21.11 (Nov. 2019), 2714–2725. URL: https://ieeexplore.ieee.org/document/8684302.

[60] Räsänen, J., Viitanen, M., Vanne, J. and Hämäläinen, T. D. Kvazzup: Open Software for HEVC Video Calls. *2017 IEEE International Symposium on Multimedia (ISM)*. Taichung, Taiwan, Dec. 2017, 549–552. URL: https://ieeexplore.ieee.org/document/8241673.

[61] Versitile Video Coding. *document ITU-T Rec. H.266 and ISO/IEC 23090-3 (VVC)*. ITU-T and ISO/IEC, Aug. 2020.

A RESAMPLING FILTER COEFFICIENTS

Table A.1a contains the luma interpolation filter coefficients used in the texture upsampling process of SHVC. Filters are defined for sixteen phases, i.e., sub-pixel positions, and luma filtering uses eight samples/coefficients do derive one filtered value (8-tap filter). Table A.1b contains the respective filter coefficients for chroma, which uses 4-tap filters.

Table A.2 contains interpolation filter coefficients for downsacling videos, defined in SHM. The same 12-tap filters are used for both luma and chroma. Additionally, separate downsampling filters are defined for eight different scaling ratio ranges.

Table A.1. *Interpolation filter coefficients for upsampling.*

(a) *Luma*

Phase	Interp. filt. coef.							
0	0	0	0	64	0	0	0	0
1/16	0	1	-3	63	4	-2	1	0
2/16	-1	2	-5	62	8	-3	1	0
3/16	-1	3	-8	60	13	-4	1	0
4/16	-1	4	-10	58	17	-5	1	0
5/16	-1	4	-11	52	26	-8	3	-1
6/16	-1	3	-9	47	31	-10	4	-1
7/16	-1	4	-11	45	34	-10	4	-1
8/16	-1	4	-11	40	40	-11	4	-1
9/16	-1	4	-10	34	45	-11	4	-1
10/16	-1	4	-10	31	47	-9	3	-1
11/16	-1	3	-8	26	52	-11	4	-1
12/16	0	1	-5	17	58	-10	4	-1
13/16	0	1	-4	13	60	-8	3	-1
14/16	0	1	-3	8	62	-5	2	-1
15/16	0	1	-2	4	63	-3	1	0

(b) *Chroma*

Phase	Interp. filt. coef.			
0	0	64	0	0
1/16	-2	62	4	0
2/16	-2	58	10	-2
3/16	-4	56	14	-2
4/16	-4	54	16	-2
5/16	-6	52	20	-2
6/16	-6	46	28	-4
7/16	-4	42	30	-4
8/16	-4	36	36	-4
9/16	-4	30	42	-4
10/16	-4	28	46	-6
11/16	-2	20	52	-6
12/16	-2	16	54	-4
13/16	-2	14	56	-4
14/16	-2	10	58	-2
15/16	0	4	62	-2

Table A.2. *Interpolation filter coefficients for downsampling.*

(a) *Ratio* $\leq \frac{20}{19}$

Phase	Interp. filt. coef.											
0	0	0	0	0	0	128	0	0	0	0	0	0
1/16	0	0	0	2	-6	127	7	-2	0	0	0	0
2/16	0	0	0	3	-12	125	16	-5	1	0	0	0
3/16	0	0	0	4	-16	120	26	-7	1	0	0	0
4/16	0	0	0	5	-18	114	36	-10	1	0	0	0
5/16	0	0	0	5	-20	107	46	-12	2	0	0	0
6/16	0	0	0	5	-21	99	57	-15	3	0	0	0
7/16	0	0	0	5	-20	89	68	-18	4	0	0	0
8/16	0	0	0	4	-19	79	79	-19	4	0	0	0
9/16	0	0	0	4	-18	68	89	-20	5	0	0	0
10/16	0	0	0	3	-15	57	99	-21	5	0	0	0
11/16	0	0	0	2	-12	46	107	-20	5	0	0	0
12/16	0	0	0	1	-10	36	114	-18	5	0	0	0
13/16	0	0	0	1	-7	26	120	-16	4	0	0	0
14/16	0	0	0	1	-5	16	125	-12	3	0	0	0
15/16	0	0	0	0	-2	7	127	-6	2	0	0	0

(b) $\frac{20}{19} <$ *ratio* $\leq \frac{5}{4}$

Phase	Interp. filt. coef.											
0	0	2	0	-14	33	86	33	-14	0	2	0	0
1/16	0	1	1	-14	29	85	38	-13	-1	2	0	0
2/16	0	1	2	-14	24	84	43	-12	-2	2	0	0
3/16	0	1	2	-13	19	83	48	-11	-3	2	0	0
4/16	0	0	3	-13	15	81	53	-10	-4	3	0	0
5/16	0	0	3	-12	11	79	57	-8	-5	3	0	0
6/16	0	0	3	-11	7	76	62	-5	-7	3	0	0
7/16	0	0	3	-10	3	73	65	-2	-7	3	0	0
8/16	0	0	3	-9	0	70	70	0	-9	3	0	0
9/16	0	0	3	-7	-2	65	73	3	-10	3	0	0
10/16	0	0	3	-7	-5	62	76	7	-11	3	0	0
11/16	0	0	3	-5	-8	57	79	11	-12	3	0	0
12/16	0	0	3	-4	-10	53	81	15	-13	3	0	0
13/16	0	0	2	-3	-11	48	83	19	-13	2	1	0
14/16	0	0	2	-2	-12	43	84	24	-14	2	1	0
15/16	0	0	2	-1	-13	38	85	29	-14	1	1	0

(c) $\frac{5}{4} <$ *ratio* $\leq \frac{5}{3}$

Phase	Interp. filt. coef.											
0	0	5	-6	-10	37	76	37	-10	-6	5	0	0
1/16	0	5	-4	-11	33	76	40	-9	-7	5	0	0
2/16	-1	5	-3	-12	29	75	45	-7	-8	5	0	0
3/16	-1	4	-2	-13	25	75	48	-5	-9	5	1	0
4/16	-1	4	-1	-13	22	73	52	-3	-10	4	1	0
5/16	-1	4	0	-13	18	72	55	-1	-11	4	2	-1
6/16	-1	4	1	-13	14	70	59	2	-12	3	2	-1
7/16	-1	3	1	-13	11	68	62	5	-12	3	2	-1
8/16	-1	3	2	-13	8	65	65	8	-13	2	3	-1
9/16	-1	2	3	-12	5	62	68	11	-13	1	3	-1
10/16	-1	2	3	-12	2	59	70	14	-13	1	4	-1
11/16	-1	2	4	-11	-1	55	72	18	-13	0	4	-1
12/16	0	1	4	-10	-3	52	73	22	-13	-1	4	-1
13/16	0	1	5	-9	-5	48	75	25	-13	-2	4	-1
14/16	0	0	5	-8	-7	45	75	29	-12	-3	5	-1
15/16	0	0	5	-7	-9	40	76	33	-11	-4	5	0

(d) $\frac{5}{3} <$ *ratio* ≤ 2

Phase	Interp. filt. coef.											
0	2	-3	-9	6	39	58	39	6	-9	-3	2	0
1/16	2	-3	-9	4	38	58	43	7	-9	-4	1	0
2/16	2	-2	-9	2	35	58	44	9	-8	-4	1	0
3/16	1	-2	-9	1	34	58	46	11	-8	-5	1	0
4/16	1	-1	-8	-1	31	57	47	13	-7	-5	1	0
5/16	1	-1	-8	-2	29	56	49	15	-7	-6	1	1
6/16	1	0	-8	-3	26	55	51	17	-7	-6	1	1
7/16	1	0	-7	-4	24	54	52	19	-6	-7	1	1
8/16	1	0	-7	-5	22	53	53	22	-5	-7	0	1
9/16	1	1	-7	-6	19	52	54	24	-4	-7	0	1
10/16	1	1	-6	-7	17	51	55	26	-3	-8	0	1
11/16	1	1	-6	-7	15	49	56	29	-2	-8	-1	1
12/16	0	1	-5	-7	13	47	57	31	-1	-8	-1	1
13/16	0	1	-5	-8	11	46	58	34	1	-9	-2	1
14/16	0	1	-4	-8	9	44	58	35	2	-9	-2	2
15/16	0	1	-4	-9	7	43	58	38	4	-9	-3	2

(e) $2 <$ *ratio* $\leq \frac{5}{2}$

Phase	Interp. filt. coef.											
0	-2	-7	0	17	35	43	35	17	0	-7	-5	2
1/16	-2	-7	-1	16	34	43	36	18	1	-7	-5	2
2/16	-1	-7	-1	14	33	43	36	19	1	-6	-5	2
3/16	-1	-7	-2	13	32	42	37	20	3	-6	-5	2
4/16	0	-7	-3	12	31	42	38	21	3	-6	-5	2
5/16	0	-7	-3	11	30	42	39	23	4	-6	-6	1
6/16	0	-7	-4	10	29	42	40	24	5	-6	-6	1
7/16	1	-7	-4	9	27	41	40	25	6	-5	-6	1
8/16	1	-6	-5	7	26	41	41	26	7	-5	-6	1
9/16	1	-6	-5	6	25	40	41	27	9	-4	-7	1
10/16	1	-6	-6	5	24	40	42	29	10	-4	-7	0
11/16	1	-6	-6	4	23	39	42	30	11	-3	-7	0
12/16	2	-5	-6	3	21	38	42	31	12	-3	-7	0
13/16	2	-5	-6	3	20	37	42	32	13	-2	-7	-1
14/16	2	-5	-6	1	19	36	43	33	14	-1	-7	-1
15/16	2	-5	-7	1	18	36	43	34	16	-1	-7	-2

(f) $\frac{5}{2} <$ *ratio* $\leq \frac{20}{7}$

Phase	Interp. filt. coef.											
0	-6	-3	5	19	31	36	31	19	5	-3	-6	0
1/16	-6	-4	4	18	31	37	32	20	6	-3	-6	-1
2/16	-6	-4	4	17	30	36	33	21	7	-3	-6	-1
3/16	-5	-5	3	16	30	36	33	22	8	-2	-6	-2
4/16	-5	-5	2	15	29	36	34	23	9	-2	-6	-2
5/16	-5	-5	2	15	28	36	34	24	10	-2	-6	-3
6/16	-4	-5	1	14	27	36	35	24	10	-1	-6	-3
7/16	-4	-5	0	13	26	35	35	25	11	0	-5	-3
8/16	-4	-6	0	12	26	36	36	26	12	0	-6	-4
9/16	-3	-5	0	11	25	35	35	26	13	0	-5	-4
10/16	-3	-6	-1	10	24	35	36	27	14	1	-5	-4
11/16	-3	-6	-2	10	24	34	36	28	15	2	-5	-5
12/16	-2	-6	-2	9	23	34	36	29	15	2	-5	-5
13/16	-2	-6	-2	8	22	33	36	30	16	3	-5	-5
14/16	-1	-6	-3	7	21	33	36	30	17	4	-4	-6
15/16	-1	-6	-3	6	20	32	37	31	18	4	-4	-6

(g) $\frac{20}{7} < ratio \leq \frac{15}{4}$

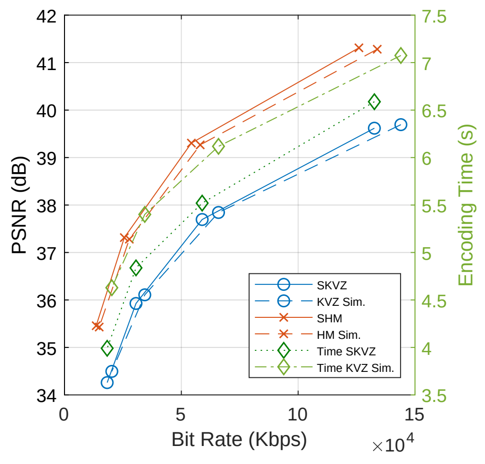
Phase	Interp. filt. coef.											
0	-9	0	9	20	28	32	28	20	9	0	-9	0
1/16	-9	0	8	19	28	32	29	20	10	0	-4	-5
2/16	-9	-1	8	18	28	32	29	21	10	1	-4	-5
3/16	-9	-1	7	18	27	32	30	22	11	1	-4	-6
4/16	-8	-2	6	17	27	32	30	22	12	2	-4	-6
5/16	-8	-2	6	16	26	32	31	23	12	2	-4	-6
6/16	-8	-2	5	16	26	31	31	23	13	3	-3	-7
7/16	-8	-3	5	15	25	31	31	24	14	4	-3	-7
8/16	-7	-3	4	14	25	31	31	25	14	4	-3	-7
9/16	-7	-3	4	14	24	31	31	25	15	5	-3	-8
10/16	-7	-3	3	13	23	31	31	26	16	5	-2	-8
11/16	-6	-4	2	12	23	31	32	26	16	6	-2	-8
12/16	-6	-4	2	12	22	30	32	27	17	6	-2	-8
13/16	-6	-4	1	11	22	30	32	27	18	7	-1	-9
14/16	-5	-4	1	10	21	29	32	28	18	8	-1	-9
15/16	-5	-4	0	10	20	29	32	28	19	8	0	-9

(h) *Ratio* $> \frac{15}{4}$

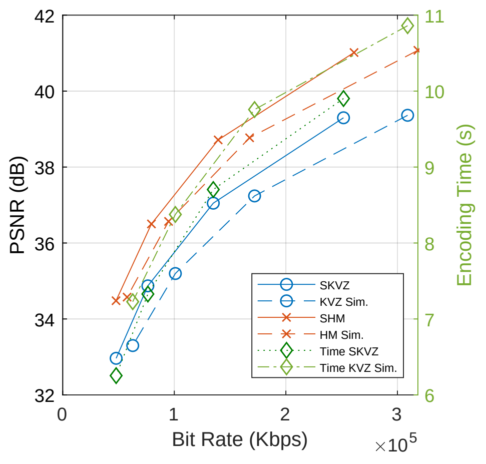
Phase	Interp. filt. coef.											
0	-8	7	13	18	22	24	22	18	13	7	2	-10
1/16	-8	7	13	18	22	23	22	19	13	7	2	-10
2/16	-8	6	12	18	22	23	22	19	14	8	2	-10
3/16	-9	6	12	17	22	23	23	19	14	8	3	-10
4/16	-9	6	12	17	21	23	23	19	14	9	3	-10
5/16	-9	5	11	17	21	23	23	20	15	9	3	-10
6/16	-9	5	11	16	21	23	23	20	15	9	4	-10
7/16	-9	5	10	16	21	23	23	20	15	10	4	-10
8/16	-10	5	10	16	20	23	23	20	16	10	5	-10
9/16	-10	4	10	15	20	23	23	21	16	10	5	-9
10/16	-10	4	9	15	20	23	23	21	16	11	5	-9
11/16	-10	3	9	15	20	23	23	21	17	11	5	-9
12/16	-10	3	9	14	19	23	23	21	17	12	6	-9
13/16	-10	3	8	14	19	23	23	22	17	12	6	-9
14/16	-10	2	8	14	19	22	23	22	18	12	6	-8
15/16	-10	2	7	13	19	22	23	22	18	13	7	-8

B BD-RATE CURVES

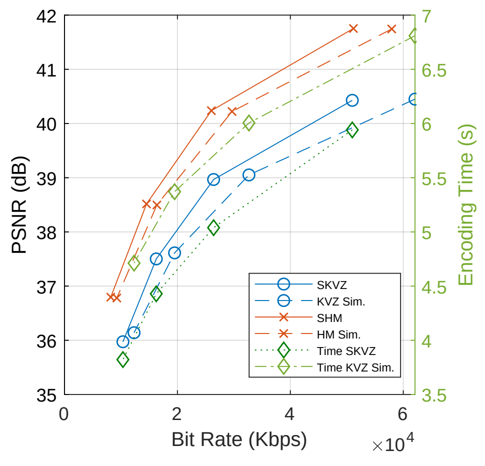
This appendix provides BD-rate curves for the results presented in Chapter 6. Additionally, encoding times for Scalable Kvazaar and Kvazaar simulcast are provided. SNR results, with $\Delta QP = -6$, are shown in Figure B.1, and $\Delta QP = -4$ results are in Figure B.2. Spatial scalability curves for the $2\times$ ratio are shown in Figures B.3 and B.4 for the two ΔQP values of 0 and 2. Finally, the $1.5\times$ ratio curves are provided in Figures B.5 and B.6, respectively.



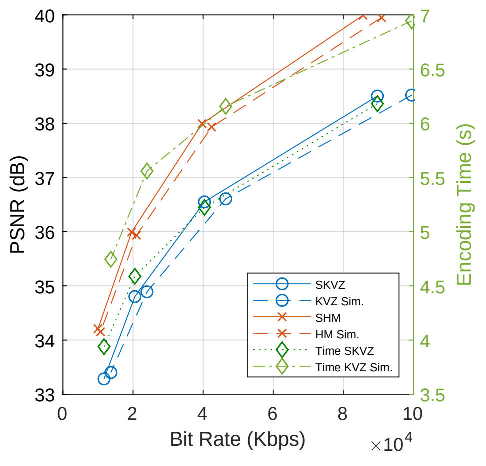
***(a)** Traffic*



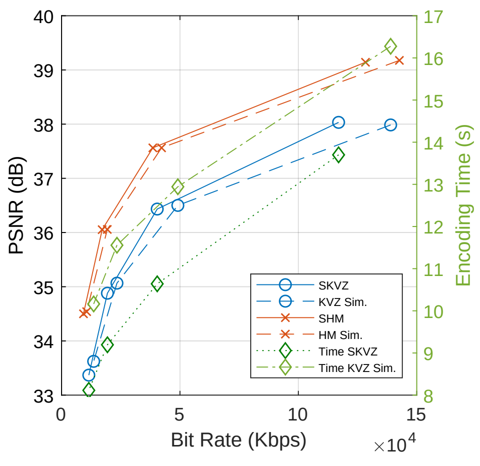
***(b)** PeopleOnStreet*



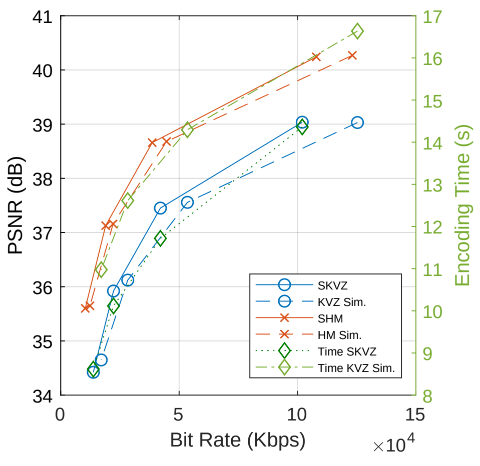
***(c)** Kimono*



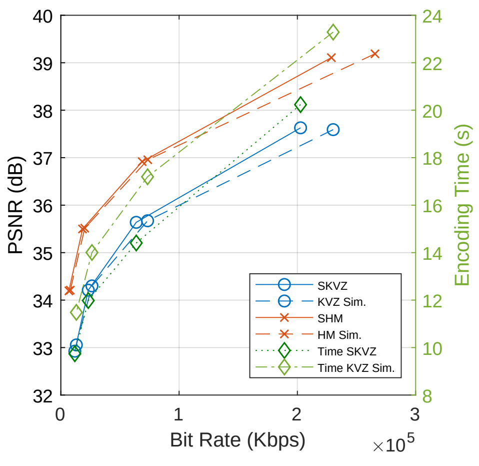
***(d)** ParkScene*



(e) *Cactus*

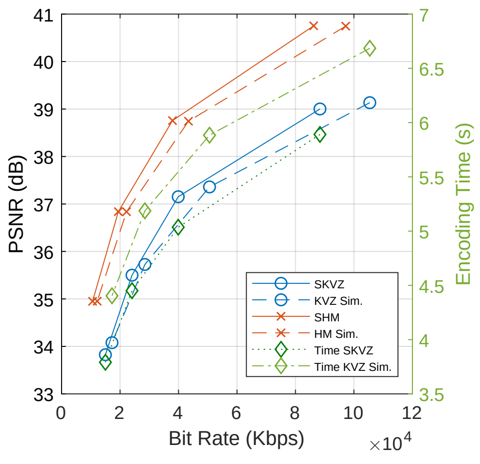


(f) *BasketballDrive*

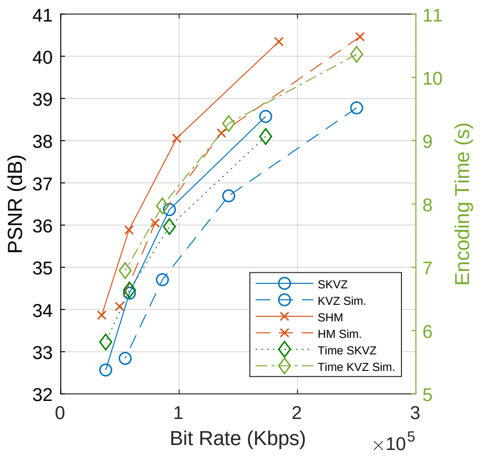


(g) *BQTerrace*

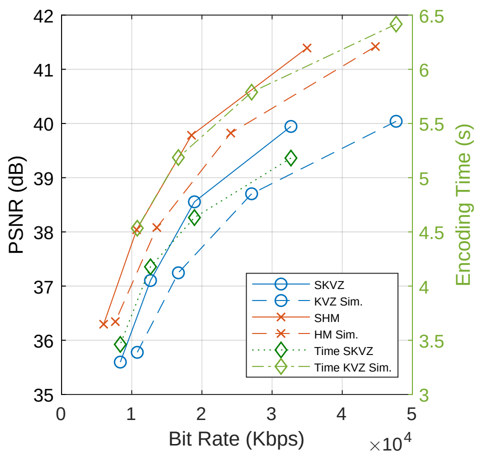
Figure B.1. *BD-rate and encoding time curves for SNR scalability with* $\Delta QP = -6$.



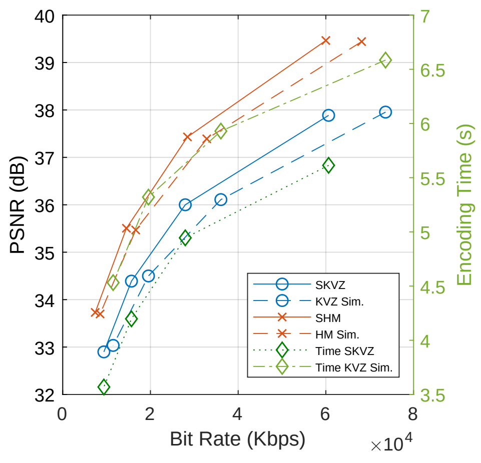
(a) *Traffic*



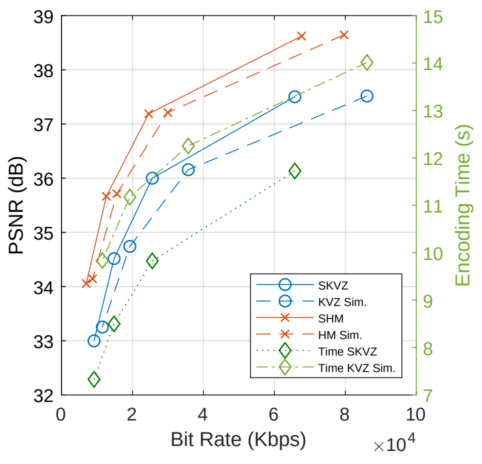
(b) *PeopleOnStreet*



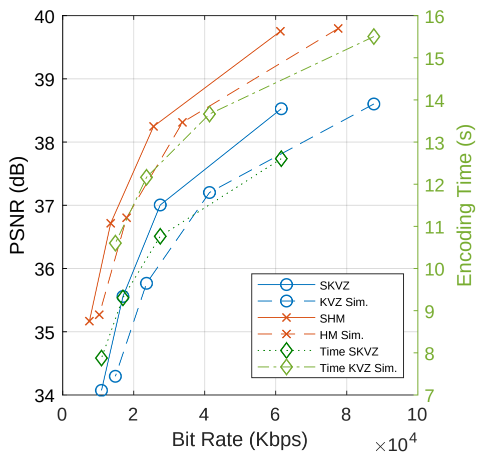
***(c)** Kimono*



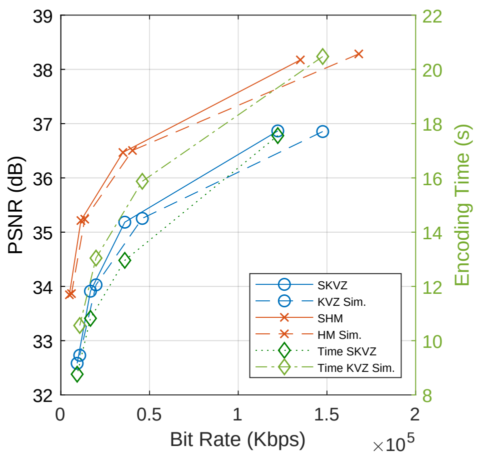
***(d)** ParkScene*



***(e)** Cactus*

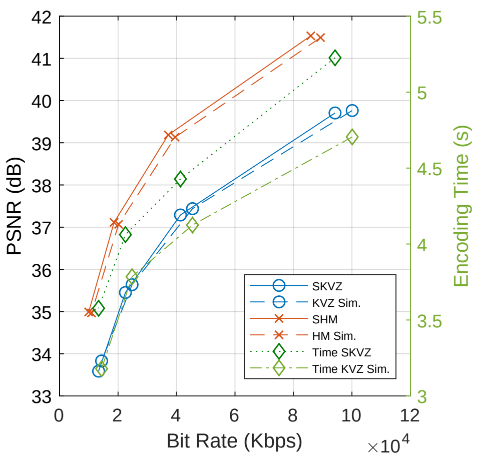


***(f)** BasketballDrive*

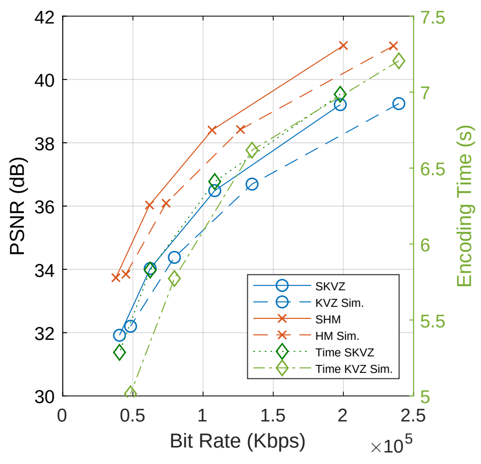


***(g)** BQTerrace*

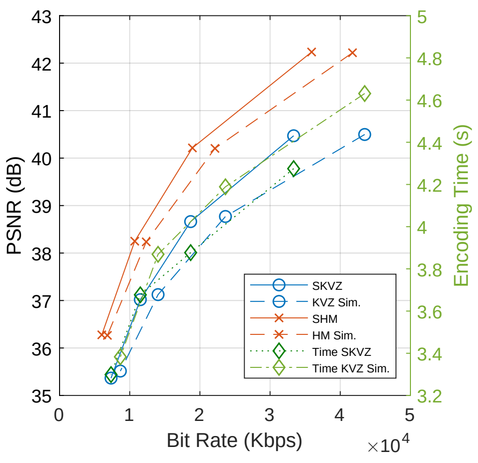
***Figure B.2.** BD-rate and encoding time curves for SNR scalability with $\Delta QP = -4$.*



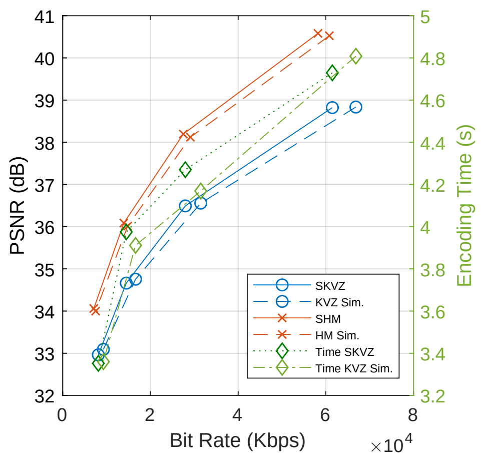
***(a)** Traffic*



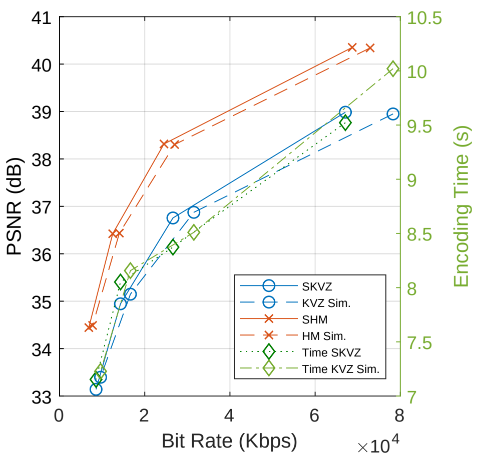
***(b)** PeopleOnStreet*



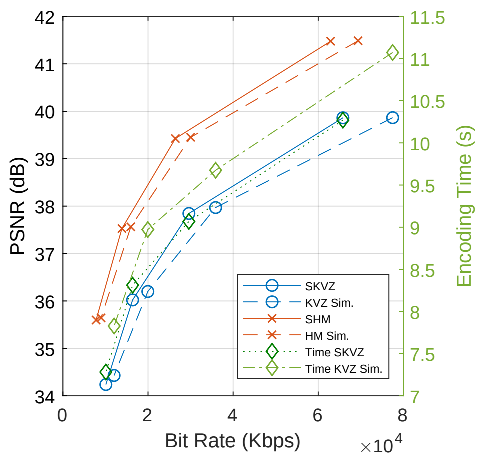
***(c)** Kimono*



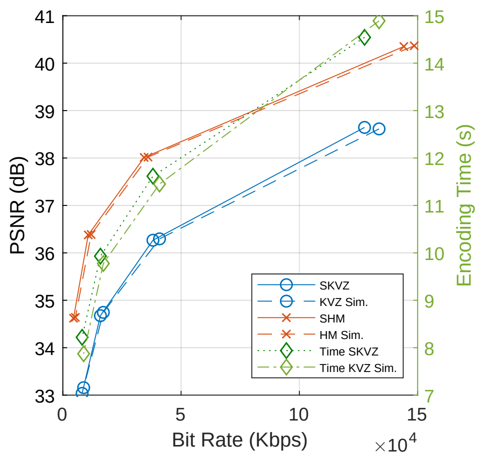
***(d)** ParkScene*



***(e)** Cactus*

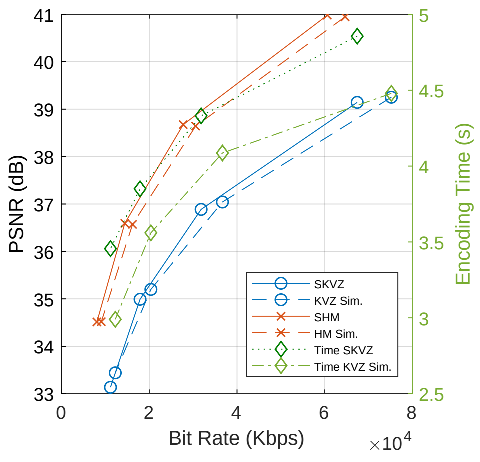


***(f)** BasketballDrive*

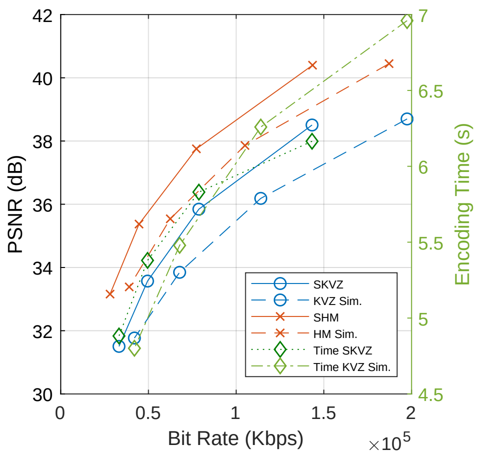


(g) *BQTerrace*

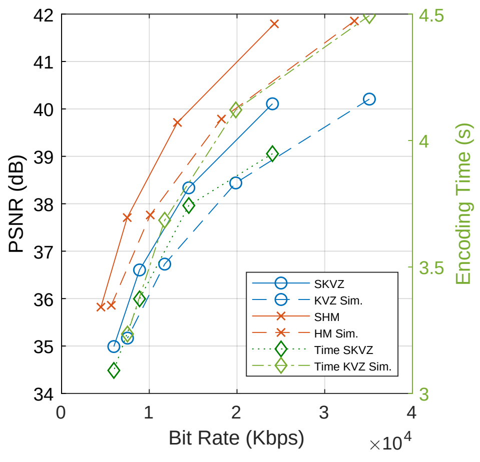
Figure B.3. *BD-rate and encoding time curves for* $2\times$ *spatial scalability with* $\Delta QP = 0$.



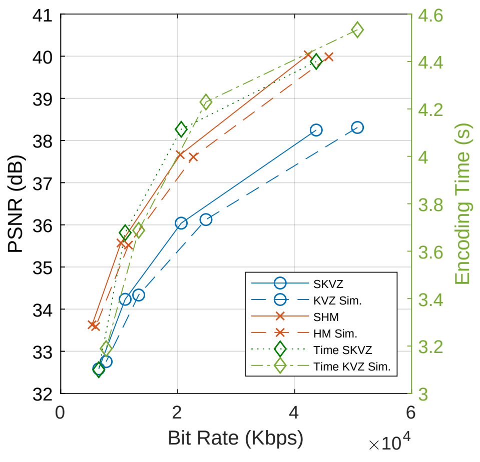
(a) *Traffic*



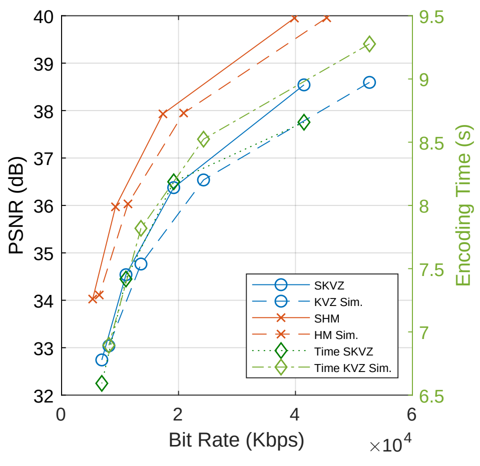
(b) *PeopleOnStreet*



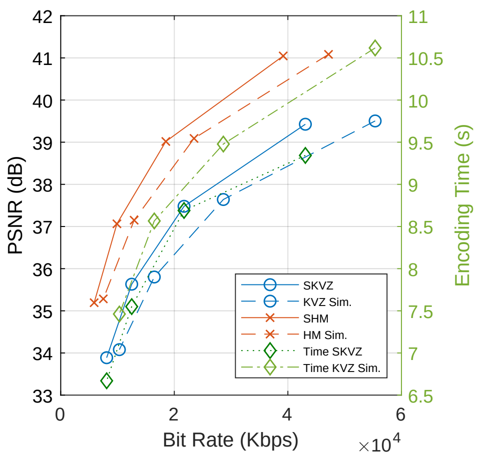
(c) *Kimono*



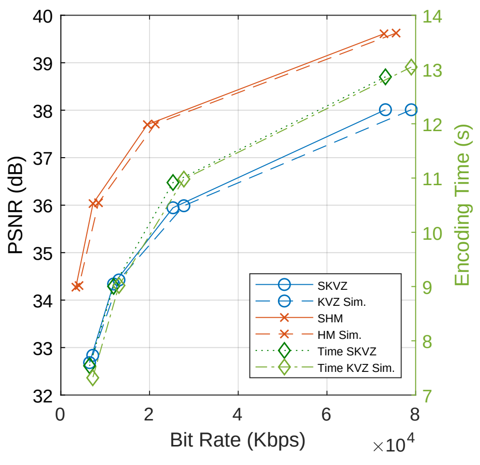
(d) *ParkScene*



***(e)** Cactus*

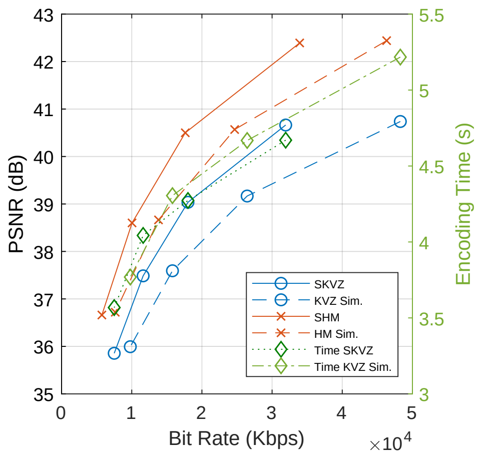


***(f)** BasketballDrive*

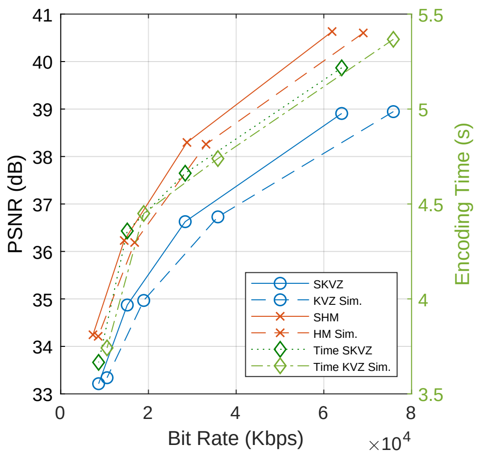


***(g)** BQTerrace*

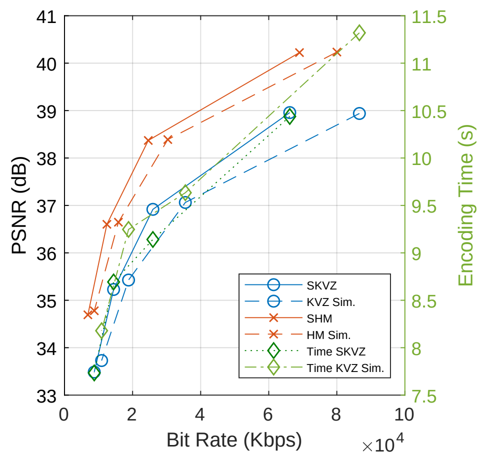
***Figure B.4.** BD-rate and encoding time curves for $2\times$ spatial scalability with $\Delta QP = 2$.*



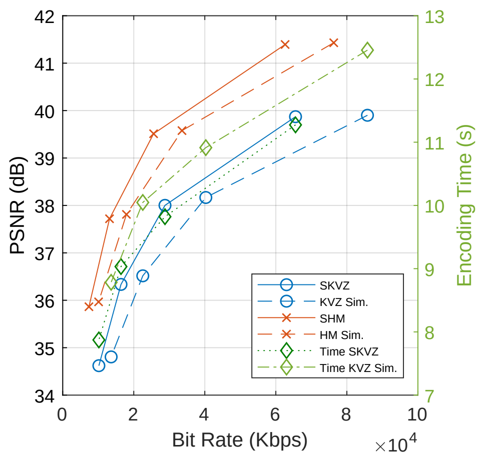
***(a)** Kimono*



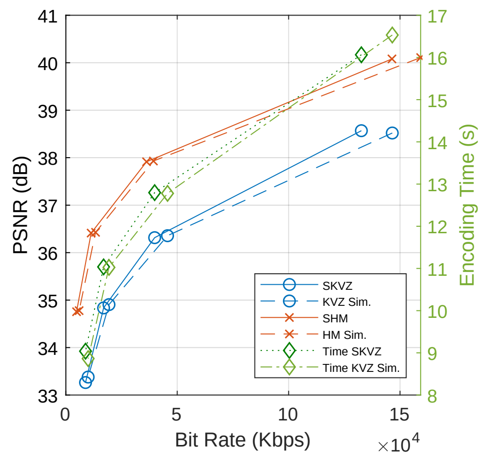
***(b)** ParkScene*



***(c)** Cactus*

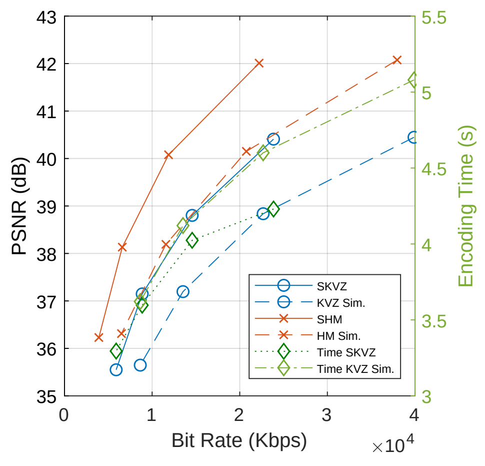


***(d)** BasketballDrive*

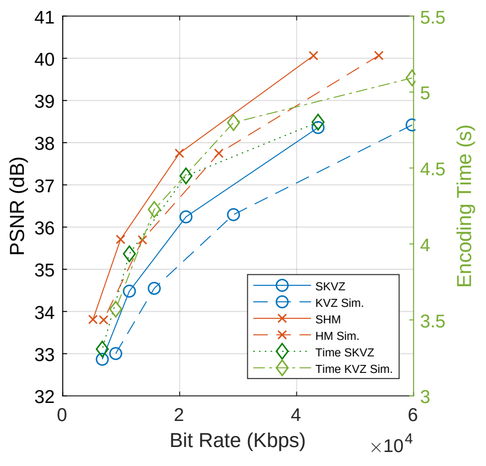


***(e)** BQTerrace*

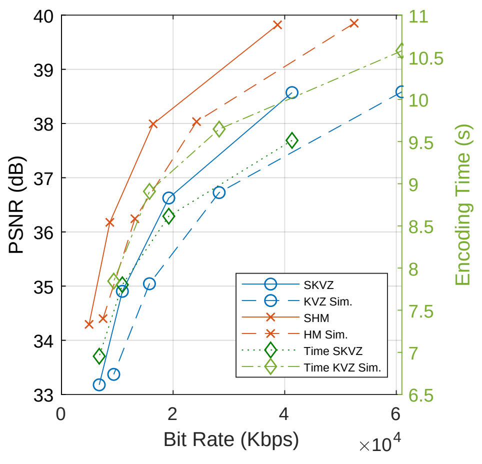
***Figure B.5.** BD-rate and encoding time curves for $1.5\times$ spatial scalability with $\Delta QP = 0$.*



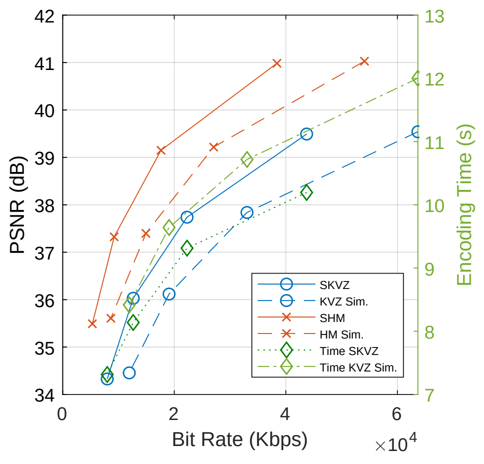
***(a)** Kimono*



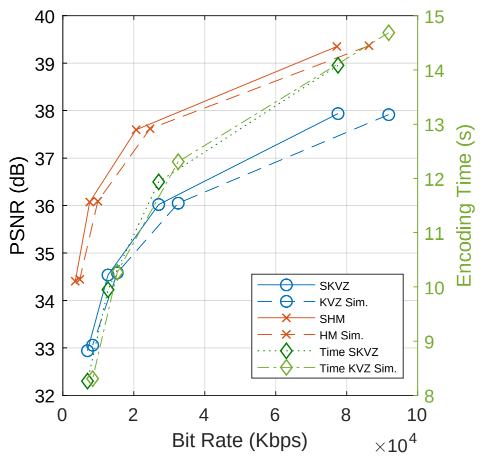
***(b)** ParkScene*



(c) *Cactus*



(d) *BasketballDrive*



(e) *BQTerrace*

Figure B.6. *BD-rate and encoding time curves for* $1.5\times$ *spatial scalability with* $\Delta QP = 2$.