INVITED PAPER

A Technical Overview of AV1

This article provides insight into the AOMedia Video 1 (AV1) video compression format. It includes a technical overview of the AV1 codec design as well as a performance evaluation that compares AV1 to its predecessor VP9.

By Jingning Han, *Senior Member IEEE*, Bohan Li, *Member IEEE*, Debargha Mukherjee, *Senior Member IEEE*, Ching-Han Chiang, Adrian Grange, Cheng Chen, Hui Su, Sarah Parker, Sai Deng, Urvang Joshi, Yue Chen, Yunqing Wang, Paul Wilkins, Yaowu Xu, *Senior Member IEEE*, and James Bankoski, *Member IEEE*

ABSTRACT | The AV1 video compression format is developed by the Alliance for Open Media consortium. It achieves more than a 30% reduction in bit rate compared to its predecessor VP9 for the same decoded video quality. This article provides a technical overview of the AV1 codec design that enables the compression performance gains with considerations for hardware feasibility.

KEYWORDS | Alliance of Open Media; AV1; video compression.

I. INTRODUCTION

The last decade has seen a steady and significant growth of web-based video applications, including video-on-demand (VoD) service, live streaming, conferencing, and virtual reality [1]. Bandwidth and storage costs have driven the need for video compression techniques with better compression efficiency. VP9 [2] and HEVC [3], both debuted in 2013, achieved in the range of 50% higher compression performance [4] than the prior codec H.264/AVC [5] and were quickly adopted by the industry.

As the demand for high-performance video compression continued to grow, the Alliance for Open Media [6] was formed in 2015 as a consortium for the development of open, royalty-free technology for multimedia delivery. Its first video compression format AV1, released in 2018, enabled about 30% compression gains over its predecessor VP9. The AV1 format is already supported by many web platforms, including Android, Chrome, Microsoft Edge, and Firefox, and multiple web-based video service providers, including YouTube, Netflix, Vimeo, and Bitmovin, have begun rolling out AV1 streaming services at scale.

Web-based video applications have seen a rapid shift from conventional desktop computers to mobile devices and TVs in recent years. For example, it is quite common to see users watch YouTube and Facebook videos on mobile phones. Meanwhile, nearly all the smart TVs after 2015 have native apps to support movie playback from YouTube, Netflix, and Amazon. Therefore, a new generation video compression format needs to ensure that it is decodable on these devices. However, to improve the compression efficiency, it is almost inevitable that a new codec will include coding techniques that are more computationally complex than its predecessors. With the slowdown in the growth of general CPU clock frequency and power constraints on mobile devices, in particular, next-generation video compression codecs are expected to rely heavily on dedicated hardware decoders. Therefore, during the AV1 development process, all the coding tools were carefully reviewed for hardware considerations (e.g., latency and silicon area), which resulted in a codec design well-balanced for compression performance and hardware feasibility.

This article provides a technical overview of the AV1 codec. Prior literature highlights some major characteristics of the codec and reports preliminary performance results [7]–[9]. A description of the available coding tools in AV1 is provided in [8]. For syntax element definition and decoder operation logic, the readers are referred to the AV1 specification [9]. Instead, this article will focus on the design theories of the compression techniques and the considerations for hardware decoder feasibility, which together define the current state of the AV1 codec. For certain coding tools that potentially demand substantial searches to realize the compression gains, it is imperative to complement them with proper encoder strategies that materialize the coding gains at a practical encoder complexity. We will further explore approaches to optimizing

Manuscript received May 13, 2020; revised November 2, 2020 and December 17, 2020; accepted February 1, 2021. Date of publication February 26, 2021; date of current version August 20, 2021. *(Corresponding author: Jingning Han.)*
The authors are with Google LLC, Mountain View, CA 94043 USA (e-mail: jingning@google.com).

Digital Object Identifier 10.1109/JPROC.2021.3058584

the tradeoff between encoder complexity and the coding performance therein. The AV1 codec includes contributions from the entire AOMedia teams [6] and the greater ecosystem around the globe. An incomplete contributor list can be found at [10].

The AV1 codec supports input video signals in the 4:0:0 (monochrome), 4:2:0, 4:2:2, and 4:4:4 formats. The allowed pixel representations are 8, 10, and 12 bit. The AV1 codec operates on pixel blocks. Each pixel block is processed in a predictive-transform coding scheme, where the prediction comes from either intraframe reference pixels, interframe motion compensation, or some combinations of the two. The residuals undergo a 2-D unitary transform to further remove the spatial correlations, and the transform coefficients are quantized. Both the prediction syntax elements and the quantized transform coefficient indexes are then entropy coded using arithmetic coding. There are three optional in-loop postprocessing filter stages to enhance the quality of the reconstructed frame for reference by subsequent coded frames. A normative film grain synthesis unit is also available to improve the perceptual quality of the displayed frames.

We will start by considering frame-level designs, before progressing on to look at coding block-level operations and the entropy coding system applied to all syntax elements. Finally, we will discuss in-loop and out-of-loop filterings. The coding performance is evaluated using libaom AV1 encoder [11], which is developed as a production codec for various services, including VoD, video conferencing, and light field, with encoder optimizations that utilize the AV1 coding tools for compression performance improvements while keeping the computational complexity in check. We note that the libaom AV1 encoder optimization is being actively developed for better compression performance and higher encoding speed. We refer to the webpage [12] for the related performance statistics update.

II. HIGH-LEVEL SYNTAX

The AV1 bitstream is packetized into open bitstream units (OBUs). An ordered sequence of OBUs is fed into the AV1 decoding process, where each OBU comprises a variable length string of bytes. An OBU contains a header and a payload. The header identifies the OBU type and specifies the payload size. Typical OBU types include the following.

1) **Sequence Header** contains information that applies to the entire sequence, e.g., sequence profile (see Section VIII) and whether to enable certain coding tools.
2) **Temporal Delimiter** indicates the frame presentation time stamp. All displayable frames following a temporal delimiter OBU will use this time stamp, until the next temporal delimiter OBU arrives. A temporal delimiter and its subsequent OBUs of the same time stamp are referred to as a temporal unit. In the context of scalable coding, the compression data associated with all representations of a frame at various spatial and fidelity resolutions will be in the same temporal unit.
3) **Frame Header** sets up the coding information for a given frame, including signaling inter or intraframe type, indicating the reference frames and signaling probability model update method.
4) **Tile Group** contains the tile data associated with a frame. Each tile can be independently decoded. The collective reconstructions form the reconstructed frame after potential loop filtering.
5) **Frame** contains the frame header and tile data. The frame OBU is largely equivalent to a frame header OBU and a tile group OBU but allows less overhead cost.
6) **Metadata** carries information, such as high dynamic range, scalability, and timecode.
7) **Tile List** contains tile data similar to a tile group OBU. However, each tile here has an additional header that indicates its reference frame index and position in the current frame. This allows the decoder to process a subset of tiles and display the corresponding part of the frame, without the need to fully decode all the tiles in the frame. Such capability is desirable for light field applications [13].

We refer to [9] for bit field definitions and more detailed consideration of high-level syntax.

III. REFERENCE FRAME SYSTEM

A. Reference Frames

The AV1 codec allows a maximum of eight frames in its decoded frame buffer. For a coding frame, it can choose any seven frames from the decoded frame buffer as its reference frames. The bitstream allows the encoder to explicitly assign each reference a unique reference frame index ranging from 1 to 7. In principle, the reference frames indices 1–4 are designated for the frames that precede the current frame in terms of display order, while indices 5–7 are for reference frames coming after the current one. For compound interprediction, two references can be combined to form the prediction (see Section V-C4). If both reference frames either precede or follow the current frame, this is considered to be the unidirectional compound prediction. This contrasts with bidirectional compound prediction where there is one previous and one future reference frame. In practice, the codec can link a reference frame index to any frame in the decoded frame buffer, which allows it to fill all the reference frame indexes when there are not enough reference frames on either side.

In estimation theory, it is commonly known that extrapolation (unidirectional compound) is usually less accurate than interpolation (bidirectional compound) prediction [14]. The allowed unidirectional reference frame combinations are hence limited to only four possible pairs, i.e., (1, 2), (1, 3), (1, 4), and (5, 7), but all the 12 combinations in the bidirectional case are supported. This limitation reduces the total number of compound reference frame combinations from 21 to 16. It follows the

assumption that if the numbers of the reference frames on both sides of the current frame in natural display order are largely balanced, the bidirectional predictions are likely to provide a better prediction. When most reference frames are on the one side of the current frame, the extrapolations that involve the nearest one are more relevant to the current frame.

When a frame coding is complete, the encoder can decide which reference frame in the decoded frame buffer to replace and explicitly signals this in the bitstream. The mechanism also allows one to bypass updating the decoded frame buffer. This is particularly useful for high motion videos where certain frames are less relevant to neighboring frames.

B. Alternate Reference Frame

The alternate reference frame (ARF) is a frame that will be coded and stored in the decoded frame buffer with the option of not being displayed. It serves as a reference frame for subsequent frames to be processed. To transmit a frame for display, the AV1 codec can either code a new frame or directly use a frame in the decoded frame buffer—this is called "show existing frame." An ARF that is later being directly displayed can be effectively used to code a future frame in a pyramid coding structure [15].

Moreover, the encoder has the option to synthesize a frame that can potentially reduce the collective prediction errors among several display frames. One example is to apply temporal filtering along the motion trajectories of consecutive original frames to build an ARF, which retains the common information [16] with the acquisition noise on each individual frame largely removed. The encoder typically uses a relatively lower quantization step size to code the common information (i.e., ARF) to optimize the overall rate-distortion performance [17]. A potential downside here is that this results in an additional frame for decoders to process, which could potentially stretch throughput capacity on some hardware. To balance the compression performance and decoder throughput, each level definition defines an upper bound on the permissible decoded sample rate, namely maximum decode rate. Since the decoded sample rate is calculated based on the total number of samples in both displayable frames and ARFs that will not be used as a "show existing frame," it effectively limits the number of allowable synthesized ARF frames.

C. Frame Scaling

The AV1 codec supports the option to scale a source frame to a lower resolution for compression and rescale the reconstructed frame to the original frame resolution. This design is particularly useful when a few frames are overly complex to compress and, hence, cannot fit in the target streaming bandwidth range. The downscaling factor is constrained to be within the range of 8/16–15/16. The reconstructed frame is first linearly upscaled to the original size, followed by a loop restoration filter as part of the postprocessing stage. Both the linear upscaling filter and the loop restoration filter operations are normatively defined. We will discuss it with more details in Section VII-D. In order to maintain a cost-effective hardware implementation where no additional expense on line buffers is required beyond the size for regular frame decoding, the rescaling process is limited to the horizontal direction. The upscaled and filtered version of the decoded frame will be available as a reference frame for coding subsequent frames.

IV. SUPERBLOCK AND TILE

A. Superblock

A superblock is the largest coding block that the AV1 codec can process. The superblock size can be either 128×128 luma samples or 64×64 luma samples, which is signaled by the sequence header. A superblock can be further partitioned into smaller coding blocks, each with its own prediction and transform modes. A superblock coding is only dependent on its above and left neighboring superblocks.

B. Tile

A tile is a rectangular array of superblocks whose spatial referencing, including intraprediction reference and the probability model update, is limited to be within the tile boundary. As a result, the tiles within a frame can be independently coded, which facilitates simple and effective multithreading for both encoder and decoder implementations. The minimum tile size is one superblock. The maximum tile width corresponds to 4096 luma samples, and the maximum tile size corresponds to 4096×2304 luma samples. A maximum of 512 tiles are allowed in a frame.

AV1 supports two ways to specify the tile size for each frame. The uniform tile size option follows the VP9 tile design and assumes all the tiles within a frame are of the same dimension, except those sitting at the bottom or right frame boundary. It allows one to identify the number of tiles vertically and horizontally in the bitstream and derives the tile dimension based on the frame size. The second option, the nonuniform tile size, assumes a lattice form of tiles. The spacing is nonuniform in both vertical and horizontal directions, and tile dimensions must be specified in the bitstream in units of superblocks. It is designed to recognize the fact that the computational complexity differs across superblocks within a frame due to the variations in video signal statistics. The nonuniform tile size option allows one to use smaller tile sizes for regions that require higher computational complexity, thereby balancing the workload among threads. This is particularly useful when one has ample computing resources in terms of multiple cores and needs to minimize the frame coding latency. An example is provided in Fig. 1 to demonstrate the two tile options.

The uniform/nonuniform tile size options and the tile sizes are decided on a frame by frame basis. It is

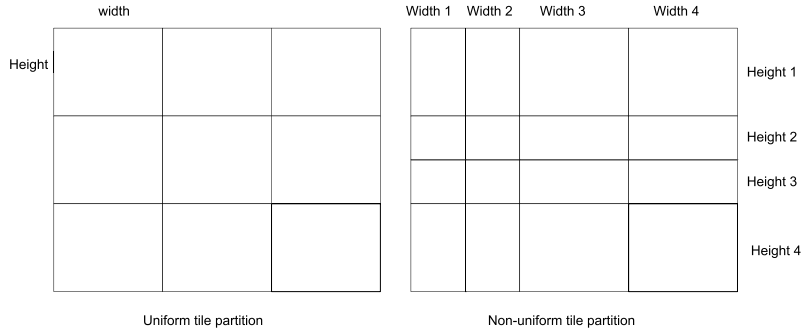


Fig. 1. ***Illustration of the uniform and nonuniform tile sizes. The uniform tile size option uses the same tile dimension across the frame. The nonuniform tile size option requires a series of width and height values to determine the lattice.***

noteworthy that the postprocessing filters are applied across the tile boundaries to avoid potential coding artifacts (e.g., blocking artifacts) along the tile edges.

V. CODING BLOCK OPERATIONS

A. Coding Block Partitioning

A superblock can be recursively partitioned into smaller block sizes for coding. AV1 inherits the recursive block partitioning design used in VP9. To reduce the overhead cost on prediction mode coding for video signals that are highly correlated, a situation typically seen in 4k videos, AV1 supports a maximum coding block size of 128×128 luma samples. The allowed partition options at each block level include ten possibilities, as shown in Fig. 2. To improve the prediction quality for complex videos, the minimum coding block size is extended to 4×4 luma samples. While such extensions provide more coding flexibility, they have implications for hardware decoders. Certain block size-dependent constraints are specifically designed to circumvent such complications.

1) Block Size-Dependent Constraints: The core computing unit in a hardware decoder is typically designed around a superblock. Increasing the superblock size from 64×64 to 128×128 would require about four times silicon area for the core computing unit. To resolve this issue, we constrain the decoding operations to be conducted in 64×64 units even for larger block sizes. For example, to decode a 128×128 block in the YUV420 format, one needs to decode the luma and chroma components corresponding to the first 64×64 block, followed by those corresponding to the next 64×64 block, and so on, in contrast to processing the luma component for the entire 128×128 block, followed by the chroma components. This constraint effectively rearranges the entropy coding order for the luma and chroma components and has no penalty on the compression performance. It allows a hardware decoder to process a 128×128 block as a series of 64×64 blocks and, hence, retain the same silicon area.

At the other end of the spectrum, the use of 4×4 coding blocks increases the worst case latency in the YUV420 format, which happens when all the coding blocks are 4×4 luma samples and are coded using intraprediction modes. To rebuild an intracoded block, one needs to wait for its above and left neighbors to be fully reconstructed because of the spatial pixel referencing. In VP9, the 4×4 luma samples within a luma 8×8 block are all coded in either inter or intramode. If it is in the intramode, the collocated 4×4 chroma components will use an intraprediction mode followed by a 4×4 transform. An unconstrained 4×4 coding block size would require each 2×2 chroma samples to go through prediction and transform coding, which creates dependence in the chroma component decoding. Note that intermodes do not have such spatial dependence issues.

AV1 adopts a constrained chroma component coding for 4×4 blocks in the YUV420 format to resolve this latency issue. If all the luma blocks within an 8×8 block are coded in the intermode, the chroma component will be predicted in 2×2 units using the motion information derived from the corresponding luma block. If any luma block is coded in an intramode, the chroma component will follow the bottom-right 4×4 luma block's coding mode and conduct the prediction in 4×4 units. The prediction residuals of chroma components then go through a 4×4 transform.

These block size-dependent constraints enable the extension of the coding block partition system with limited impact on hardware feasibility. However, an extensive rate-distortion optimization search is required to translate this increased flexibility into compression gains.

2) Two-Stage Block Partitioning Search: Observing that the key flexibility in variable coding block size is provided by the recursive partition that goes through the square coding blocks, one possibility is to employ a two-stage partition search approach. The first pass starts from the largest

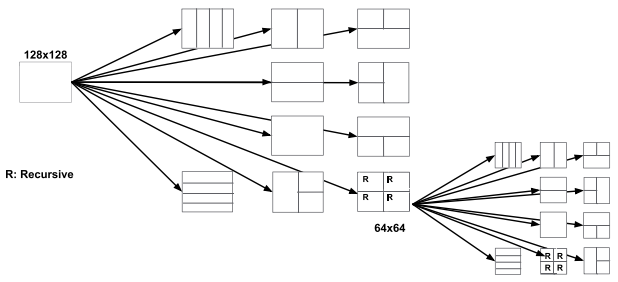


Fig. 2. ***Recursive block partition tree in AV1.***

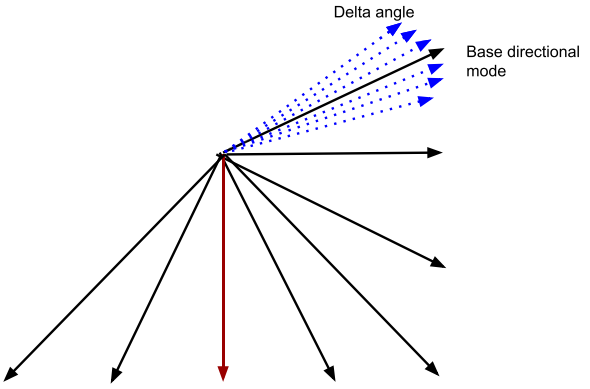


Fig. 3. *Directional intraprediction modes. The original eight directions in VP9 are used as a base. Each allows a supplementary signal to tune the prediction angle in units of 3°.*

coding block size and goes through square partitions only. For each coding block, the rate-distortion search is limited, e.g., only using the largest transform block and 2-D DCT kernel. Its partition decisions can be analyzed to determine the most likely operating range, in which the second block partition search will conduct an extensive rate-distortion optimization search for all the ten possible partitions. Changing the allowed block size search range drawn from the first pass partition results would give different tradeoffs between the compression performance and the encoding speed. We refer to [18] for more experimental results.

We will next discuss the compression techniques available at a coding block level within a partition.

B. Intraframe Prediction

For a coding block in intramode, the prediction mode for the luma component and the prediction mode for both chroma components are signaled separately in the bitstream. The luma prediction mode is entropy coded using a probability model based on its above and left coding blocks' prediction context. The entropy coding of the chroma prediction mode is conditioned on the state of the luma prediction mode. The intraprediction operates in units of transform blocks (as introduced in Section V-E) and uses previously decoded boundary pixels as a reference.

1) Directional Intraprediction: AV1 extends the directional intraprediction options in VP9 to support higher granularity. The original eight directional modes in VP9 are used as a base in AV1, with a supplementary signal to fine-tune the prediction angle. This comprises up to three steps clockwise or counterclockwise, each of 3°, as shown in Fig. 3. A two-tap bilinear filter is used to interpolate the reference pixels when a prediction points to a subpixel position. For coding block size of less than 8×8, only the eight base directional modes are allowed since the small number of pixels to be predicted does not justify the overhead cost of the additional granularity.

2) Nondirectional Smooth Intraprediction: VP9 has two nondirectional intrasmooth prediction modes: dc_PRED and TM_PRED. AV1 adds three new smooth prediction modes that estimate pixels using a distance weighted linear combination, namely SMOOTH_V_PRED, SMOOTH_H_PRED, and SMOOTH_PRED. They use the bottom-left (BL) and top-right (TR) reference pixels to fill the right-most column and bottom-row, thereby forming a closed-loop boundary condition for interpolation. We use the notations in Fig. 4 to demonstrate their computation procedures.

1) SMOOTH_H_PRED: $P_H = w(x)L + (1 - w(x))\text{TR}$.
2) SMOOTH_V_PRED: $P_V = w(y)T + (1 - w(y))\text{BL}$.
3) SMOOTH_PRED: $P = (P_H + P_V)/2$.

Here, $w(x)$ represents the weight based on distance x from the boundary, whose values are preset.

AV1 replaces the TM_PRED mode that operates as

$$P = T + L - TL$$

with a PAETH_PRED mode that follows:

$$P = \text{argmin}|x - (T + L - TL)| \quad \forall x \in \{T, L, TL\}.$$

The nonlinearity in the PAETH_PRED mode allows the prediction to steer the referencing angle to align with the direction that exhibits highest correlation.

3) Recursive Intraprediction: The interpixel correlation is modeled as a 2-D first-order Markov field. Let $X(i, j)$ denote a pixel at position (i, j). Its prediction is formed by

$$\hat{X}(i, j) = \alpha\hat{X}(i-1, j) + \beta\hat{X}(i, j-1) + \gamma\hat{X}(i-1, j-1) \quad (1)$$

where $\hat{X}$s on the right-hand side are the available reconstructed boundary pixels or the prediction of the above and left pixels. The coefficient set $\{\alpha, \beta, \gamma\}$ forms a linear predictor based on the spatial correlations. A total of five different sets of linear predictors are defined in AV1; each represents a different spatial correlation pattern.

To improve hardware throughput, instead of recursively predicting each pixel, AV1 predicts a 4×2 pixel patch from its adjacent neighbors, e.g., $p_0 - p_6$ for the blue patch $x_0 - x_7$ in Fig. 5, whose coefficients can be directly derived from

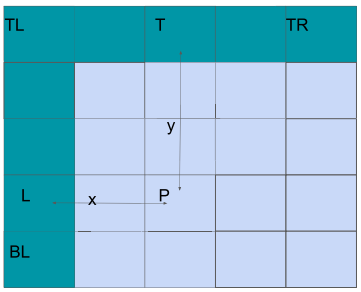


Fig. 4. *Illustration of the distance weighted smooth intraprediction. The dark green pixels are the reference, and the light blue ones are the prediction. The variables x and y are the distance from left and top boundaries, respectively.*

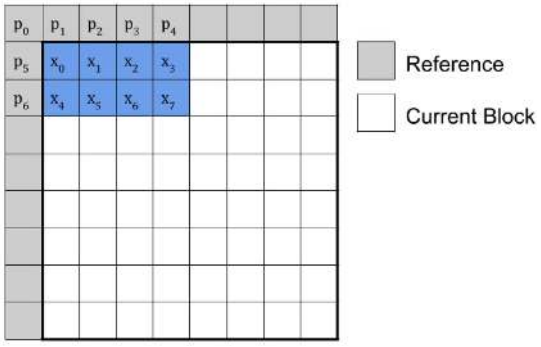


Fig. 5. *Recursive-filter-based intrapredictor. Reference pixels p_0–p_6 are used to linearly predict the 4×2 patch in blue. The predicted pixels will be used as reference for next 4×2 patch in the current block.*

$\{\alpha, \beta, \gamma\}$ by expanding the recursion

$$\begin{aligned} x_0 &= \alpha p_1 + \beta p_5 + \gamma p_0 \\ x_1 &= \alpha p_2 + \beta x_0 + \gamma p_1 \\ &= \beta\gamma p_0 + (\alpha\beta + \gamma) p_1 + \alpha p_2 + \beta^2 \, p_5 \\ x_2 &= \alpha p_3 + \beta x_1 + \gamma p_2 \\ &= \gamma\beta^2 \, p_0 + (\alpha\beta + \gamma)\beta p_1 + (\alpha\beta + \gamma) p_2 + \alpha p_3 + \beta^3 \, p_5 \\ &\vdots \end{aligned}$$

Such expansion avoids the interpixel dependence within a 4×2 patch, thereby allowing hardware decoders to process the predictions in parallel.

4) Chroma From Luma Prediction: Chroma from luma prediction models chroma pixels as a linear function of corresponding reconstructed luma pixels. As depicted in Fig. 6, the predicted chroma pixels are obtained by adding the dc prediction of the chroma block to a scaled ac contribution, which is the result of multiplying the ac component of the downsampled luma block by a scaling factor explicitly signaled in the bitstream [19].

5) Intrablock Copy: AV1 allows intraframe motion-compensated prediction, which uses the previously coded pixels within the same frame, namely Intrablock Copy (IntraBC). A motion vector at full pixel resolution is used to locate the reference block. This may imply a half-pixel accuracy motion displacement for the chroma components, in which context a bilinear filter is used to conduct sub-pixel interpolation. The IntraBC mode is only available for intracoding frames and can be turned on and off by frame header.

Typical hardware decoders pipeline the pixel reconstruction and the postprocessing filter stages such that the postprocessing filters are applied to the decoded superblocks, while later superblocks in the same frame are being decoded. Hence, an IntraBC reference block is retrieved from the pixels after postprocessing filters. In contrast, a typical encoder would process all the coding blocks within a frame and then decide the postprocessing filter parameters that minimize the reconstruction error. Therefore, the IntraBC mode most likely accesses the coded pixels prior to the postprocessing filters for rate-distortion optimization. Such discrepancy hinders the efficiency of IntraBC mode. To circumvent this issue, all the postprocessing filters are disabled if the IntraBC mode is allowed in an intra-only coded frame.

In practice, the IntraBC mode is most likely useful for images that contain a substantial amount of text content or similar repeated patterns, in which setting post-processing filters are less effective. For natural images where pixels largely form an autoregressive (AR) model, the encoder needs to be cautious regarding the use of IntraBC mode, as the absence of postprocessing filters may trigger visual artifacts at coarse quantization.

6) Color Palette: In this mode, a color palette ranging between two to eight base colors (i.e., a pixel value) is built for each luma/chroma plane, where each pixel gets assigned a color index. The number of base colors is an encoder decision that determines the tradeoff between fidelity and compactness. The base colors are predictively coded in the bitstream using those of neighboring blocks as reference. The color indexes are coded pixel-by-pixel using a probability model conditioned on previously coded color indexes. The luma and chroma channels can decide whether to use the palette mode independently. This mode is particularly suitable for a pixel block that contains limited pixel variations.

C. Interframe Prediction

AV1 supports rich toolsets to exploit the temporal correlation in video signals. These include adaptive filtering in translational motion compensation, affine motion compensation, and highly flexible compound prediction modes.

1) Translational Motion Compensation: A coding block uses a motion vector to find its prediction in a reference frame. It first maps its current position, e.g., top-left pixel position (x_0, y_0) in Fig. 7, in the reference frame. It is then displaced by the motion vector to the target reference block whose top-left pixel is located at (x_1, y_1).

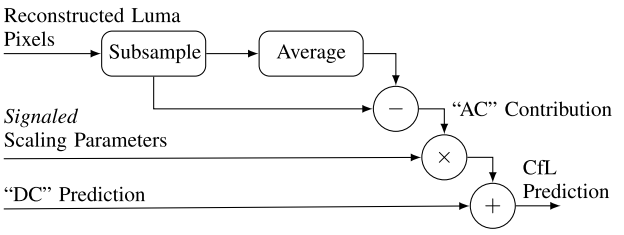


Fig. 6. *Outline of the operations required to build the CfL prediction [19].*

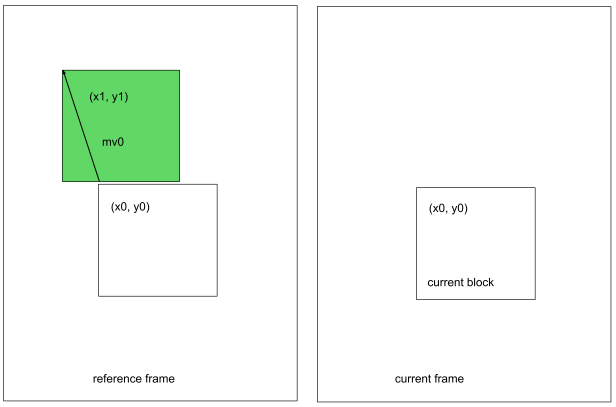


Fig. 7. ***Translational motion-compensated prediction.***

AV1 allows 1/8 pixel motion vector accuracy. A subpixel is generated through separable interpolation filters. A typical procedure is shown in Fig. 8, where one first computes the horizontal interpolation through all the related rows. The second vertical filter is applied to the resulting intermediate pixels to produce the final subpixel. It is clear that the intermediate pixels (orange) can be reused to produce multiple final subpixels (green).

Common block-based encoder motion estimations are conducted via measurements of the sum of absolute difference (SAD) or the sum of squared error (SSE) [20]–[22], which tends to favor a reference block that resembles the dc and lower ac frequency components well, whereas the high-frequency components are less reliably predicted. An interpolation filter with a high cutoff frequency would allow more high-frequency components from the reference region to form the prediction and is suitable for cases where the high-frequency components between the reference and the current block are highly correlated. Conversely, an interpolation filter with a low cutoff frequency would largely remove high-frequency components that are less relevant to the current block.

An adaptive interpolation filter scheme is used in VP9, where an intercoded block in VP9 can choose from three eight-tap interpolation filters that correspond to different cutoff frequencies in a Hamming window in the frequency domain. The selected interpolation filter is applied to both vertical and horizontal directions. AV1 inherits the interpolation filter selection design and extends it to support independent filter selection for the vertical and horizontal directions, respectively. It exploits the potential temporal statistical discrepancy between the vertical and horizontal directions for improved prediction quality. Each direction can choose from three finite impulse response (FIR) filters, namely SMOOTH, REGULAR, and SHARP in ascending order of cutoff frequencies. A heat map of the correlations between the prediction and the source signals in the transform domain is shown in Fig. 9, where the prediction and source block pairs are grouped according to their optimal 2-D interpolation filters. It is evident that the signal statistics differ in vertical and horizontal directions, and an independent filter selection in each direction captures such discrepancy well.

To reduce the decoder complexity, the SMOOTH and REGULAR filters adopt a six-tap FIR design, which appears to be sufficient for a smooth and flat baseband. The SHARP filter continues to use an eight-tap FIR design to mitigate the ripple effect near the cutoff frequency. The filter coefficients that correspond to half-pixel interpolation are

$$\begin{aligned} \text{SMOOTH} \quad & [-2, 14, 52, 52, 14, -2] \\ \text{REGULAR} \quad & [2, -14, 76, 76, -14, 2] \\ \text{SHARP} \quad & [-4, 12, -24, 80, 80, -24, 12, -4] \end{aligned}$$

whose frequency responses are shown in Fig. 10. To further reduce the worst case complexity when all coding blocks are in 4×4 luma samples, there are two additional four-tap filters that are used when the coding block has dimensions of 4 or less. The filter coefficients for half-pixel interpolation are

$$\begin{aligned} \text{SMOOTH} \quad & [12, 52, 52, 12] \\ \text{REGULAR} \quad & [-12, 76, 76, -12]. \end{aligned}$$

The SHARP filter option is not applicable due to the short filter taps.

2) Affine Model Parameters: Besides conventional translational motion compensation, AV1 also supports the affine transformation model that projects a current pixel at (x, y) to a prediction pixel at (x', y') in a reference frame through

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{2}$$

The tuple (h_{13}, h_{23}) corresponds to a conventional motion vector used in the translational model. Parameters h_{11} and

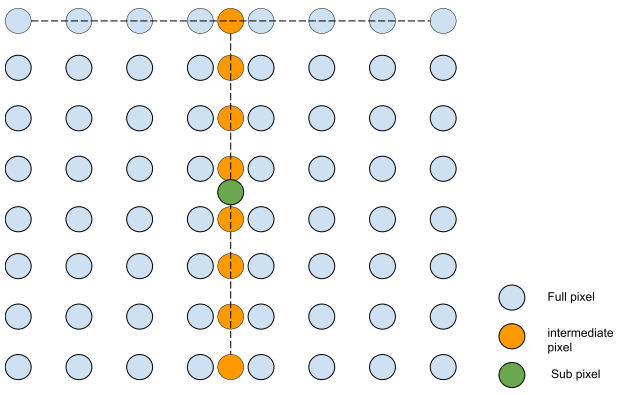


Fig. 8. ***Subpixel generation through separable interpolation filter.***

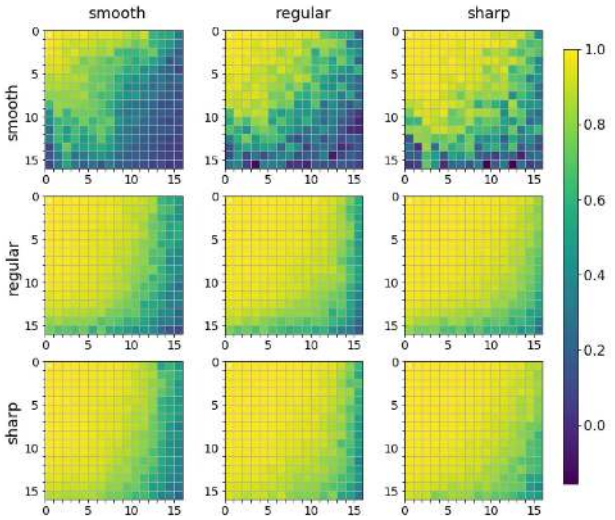


Fig. 9. ***Heat map of the correlations between the prediction and the source signals in the transform domain. The motion estimation here is in units of 16 × 16 block. The prediction and source blocks are grouped based on their optimal interpolation filters. The test clip is old_town_cross_480p. The top and left labels mark the interpolation filters used in the horizontal and vertical directions, respectively. It is evident that the groups using the SHARP filter tend to have a higher correlation in high-frequency components along the corresponding direction.***

h_{22} control the scaling factors in the vertical and horizontal axes and, in conjunction with the pair h_{12} and h_{21}, decide the rotation angle.

A global affine model is associated with each reference frame, where each of the four nontranslational parameters has 12-bit precision, and the translational motion vector is coded in 15-bit precision. A coding block can choose to use it directly provided the reference frame index. The global affine model captures the frame-level scaling and rotation and, hence, primarily focuses on the settings of rigid motion over the entire frame. In addition, a local

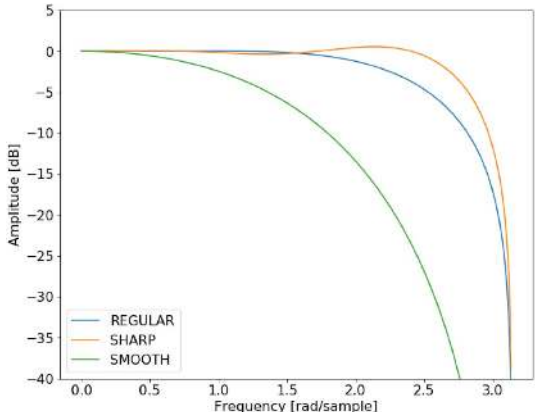


Fig. 10. ***Frequency responses of the three interpolation filters at half-pixel position.***

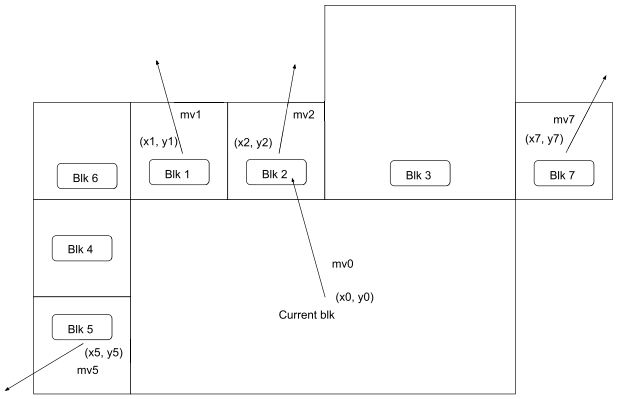


Fig. 11. ***Illustration of the local affine parameter estimation.***

affine model at the coding block level would be desirable to adaptively track the nontranslational motion activities that vary across the frame. However, the overhead cost of sending the affine model parameters per coding block also introduces additional side information [23]. As a result, various research efforts focus on the estimation of the affine model parameters without the extra overhead [24], [25]. A local affine parameter estimation scheme based on the regular translational motion vectors from spatial neighboring blocks has also been developed for AV1.

The translational motion vector (h_{13}, h_{23}) in the local affine model is explicitly transmitted in the bitstream. To estimate the other four parameters, it hypothesizes that the local scaling and rotation factors can be reflected by the pattern of the spatial neighbors' motion activities. The codec scans through a block's nearest neighbors and finds the ones whose motion vector points toward the same reference frame. A maximum of eight candidate reference blocks is allowed. For each selected reference block, its center point will first be offset by the center location of the current block to create an original sample position. This offset version will then add the motion vector difference between the two blocks to form the destination sample position after the affine transformation. Least-squares regression is conducted over the available original and destination sample position pairs to calculate the affine model parameters.

We use Fig. 11 as an example to demonstrate the affine parameter estimation process. The nearest neighbor blocks are marked by the scan order. For Block k, its center position is denoted by (x_k, y_k), and the motion vector is denoted by mv_k. The current block is denoted by $k = 0$. Assume that, in this case, Blocks 1, 2, 5, and 7 share the same reference as the current block and are selected as the reference blocks. The original sample position is formed as

$$(a_k, b_k) = (x_k, y_k) - (x_0, y_0) \tag{3}$$

where $k \in \{1, 2, 5, 7\}$. The corresponding destination sample position is obtained by further adding the motion

vector difference

$$\left(a'_k, b'_k\right) = (a_k, b_k) + (mv_k.x, mv_k.y) - (mv_0.x, mv_0.y). \quad (4)$$

To formulate the least-squares regression, we denote the sample data as

$$P = \begin{bmatrix} a_1, b_1 \\ a_2, b_2 \\ a_5, b_5 \\ a_7, b_7 \end{bmatrix}, \quad q = \begin{bmatrix} a'_1 \\ a'_2 \\ a'_5 \\ a'_7 \end{bmatrix}, \text{ and } r = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_5 \\ b'_7 \end{bmatrix}. \quad (5)$$

The least-squares regression gives the affine parameter in (2) as

$$\begin{bmatrix} h_{11} \\ h_{12} \end{bmatrix} = (P^T P)^{-1} P^T q, \text{ and } \begin{bmatrix} h_{21} \\ h_{22} \end{bmatrix} = (P^T P)^{-1} P^T r. \quad (6)$$

In practice, one needs to ensure that the spatial neighboring block is relevant to the current block. Hence, we discard the reference block if any component of the motion vector difference is above 8 pixels in the absolute value. Furthermore, if the number of available reference blocks is less than 2, the least-squares regression problem is ill posed; hence, the local affine model is disabled.

3) Affine Motion Compensation: With the affine model established, we next discuss techniques in AV1 for efficient prediction construction [26]. The affine model is allowed for block size at 8×8 and above. A prediction block is decomposed into 8×8 units. The center pixel of each 8×8 prediction unit is first determined by the translational motion vector (h_{13}, h_{23}), as shown in Fig. 12. The rest of the pixels at position (x, y) in the green square in Fig. 12 are scaled and rotated around the center pixel at (x_1, y_1) to form the affine projection (x', y') in the dashed line following:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x - x_1 \\ y - y_1 \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}. \quad (7)$$

The affine projection allows 1/64 pixel precision. A set of eight-tap FIR filters (six-tap in certain corner cases) is designed to construct the subpixel interpolations. A conventional translational model has a uniform subpixel offset across the entire block, which allows one to effectively "reuse" most intermediate outcomes to reduce the overall computation. Typically, as introduced in Section V-C1, to interpolate an 8×8 block, a horizontal filter is first applied to generate an intermediate 15×8 array from a 15×15 reference region. The second vertical filter is then applied to the intermediate 15×8 array to produce the final 8×8 prediction block. Hence, a translational model requires $(15 \times 8) \times 8$ multiplications for the horizontal filter stage and $(8 \times 8) \times 8$ multiplications for the vertical filter stage, 1472 multiplications in total.

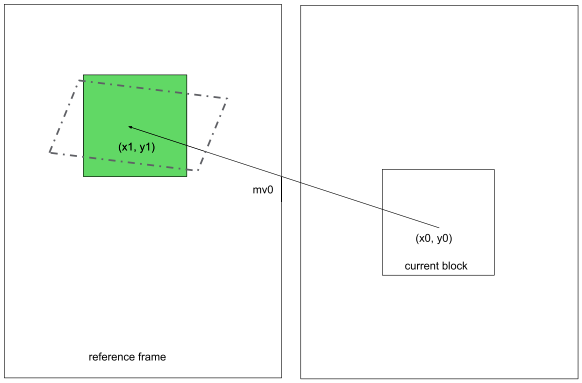


Fig. 12. ***Build the affine prediction.***

Unlike the translational model, it is reasonable to assume that each pixel in an affine model has a different subpixel offset due to the rotation and scaling effect. Directly computing each pixel would require $64 \times 8 \times 8 = 4096$ multiplications. Observe, however, that the rotation and scaling matrix in (7) can be decomposed into two shear matrices

$$\begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \gamma & 1+\delta \end{bmatrix} \begin{bmatrix} 1+\alpha & \beta \\ 0 & 1 \end{bmatrix} \quad (8)$$

where the first term on the right-hand side corresponds to a vertical interpolation and the second term corresponds to a horizontal interpolation. This translates building an affine reference block into a two-stage interpolation operation. A 15×8 intermediate array is first obtained through horizontal filtering over a 15×15 reference region, where the horizontal offsets are computed as

$$\text{horz offset} = (1+\alpha)(x - x_1) + \beta(y - y_1). \quad (9)$$

The intermediate array then undergoes vertical filtering to interpolate vertical offsets

$$\text{vert offset} = \gamma(x - x_1) + (1+\delta)(y - y_1) \quad (10)$$

and generates the 8×8 prediction block. It, thus, requires a total of 1472 multiplications, the same as the translational case. However, it is noteworthy that the actual computational cost of affine model is still higher since the filter coefficients change at each pixel, whereas the translational model uses a uniform filter in the horizontal and vertical stages, respectively.

To improve the cache performance AV1 requires the horizontal offset in (9) to be within 1 pixel away from $(x - x_1)$ and the vertical offset in (10) to be within 1 pixel away from $(y - y_1)$, which constrains the reference region within

a 15×15 pixel array. Consider the first stage that generates a 15×8 intermediate pixel array. The displacements from its center are $(x - x_1) \in [-4, 4)$ and $(y - y_1) \in [-7, 8)$. Hence, we have the constraint on the maximum horizontal offset as

$$\max\ \alpha(x - x_1) + \beta(y - y_1) = 4|\alpha| + 7|\beta| < 1. \tag{11}$$

Similarly, $(x - x_1) \in [-4, 4)$ and $(y - y_1) \in [-4, 4)$ in the second stage, which leads to

$$4|\gamma| + 4|\delta| < 1. \tag{12}$$

A valid affine model in AV1 needs to satisfy both conditions in (11) and (12).

4) Compound Predictions: The motion-compensated predictions from two reference frames (see supported reference frame pairs in Section III-A) can be linearly combined through various compound modes. The compound prediction is formulated by

$$P(x, y) = m(x, y) * R_1(x, y) + (64 - m(x, y)) * R_2(x, y)$$

where the weight $m(x, y)$ is scaled by 64 for integer computation, and $R_1(x, y)$ and $R_2(x, y)$ represent the pixels at position (x, y) in the two reference blocks. $P(x, y)$ will be scaled down by 1/64 to form the final prediction.

a) Distance weighted predictor: Let d_1 and d_2 denote the temporal distance between the current frame and its two reference frames, respectively. The weight $m(x, y)$ is determined by the relative values of d_1 and d_2. Assuming that $d_1 \le d_2$, the weight coefficient is defined by

$$m(x, y) = \begin{cases} 36, & d_2 < 1.5d_1 \\ 44, & d_2 < 2.5d_1 \\ 48, & d_2 < 3.5d_1 \\ 52, & \text{otherwise}. \end{cases} \tag{13}$$

The distribution is symmetric for the case $d_1 \ge d_2$.

b) Average predictor: A special case of the distance weighted predictor, where the two references are equally weighted, i.e., $m(x, y) = 32$.

c) Difference weighted predictor: The weighting coefficient is computed per pixel based on the difference between the two reference pixels. A binary sign is sent per coding block to decide which reference block prevails when the pixel difference is above a certain threshold

$$m(x, y) = \begin{cases} 38 + \dfrac{|R_1(x, y) - R_2(x, y)|}{16}, & \text{sign} = 0 \\ 64 - \left(38 + \dfrac{|R_1(x, y) - R_2(x, y)|}{16}\right), & \text{sign} = 1. \end{cases} \tag{14}$$

Note that $m(x, y)$ is further capped by $[0, 64]$.

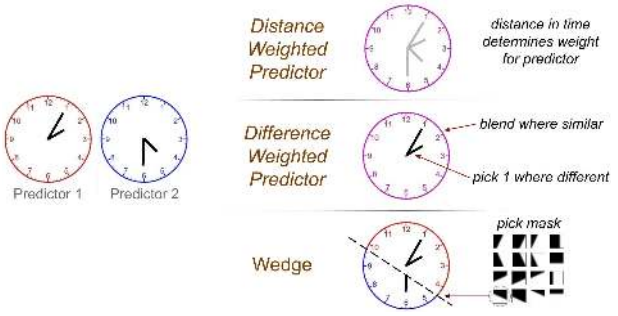


Fig. 13. *Illustration of the compound prediction modes. The distance weighted predictor uniformly combines the two reference blocks. The difference weighted predictor combines the pixels when their values are close (e.g., the dial plate and the numbers) and picks one reference when the difference is large (e.g., the clock hands). The wedge predictor uses one of the preset masks to split the block into two sections, each filled with one reference block's pixels. In the above example, it stitches the lower part predictor 1 and the higher upper part of predictor 2 as the compound prediction.*

d) Wedge mode: A set of 16 coefficient arrays has been preset for each eligible block size. They effectively split the coding block into two sections along with various oblique angles. $m(x, y)$ is mostly set to 64 in one section and 0 in the other, except near the transition edge, where there is a gradual change from 64 to 0 with 32 at the actual edge.

We use Fig. 13 to demonstrate the compound options and their effects. The numerous compound modes add substantial encoding complexity in order to realize their potential coding gains. A particular hotspot lies in the motion estimation process because each reference block is associated with its own motion vector. Simultaneously optimizing both motion vectors for a given compound mode makes the search space grow exponentially. Prior research [27] proposes a joint search approach that iteratively fixes one motion vector and searches the other motion vector until the results converge, which can significantly reduce the number of motion vector search points for a compound mode.

Other prediction modes supported by AV1 that blends multiple reference blocks include overlapped block motion compensation and a combined inter–intra-prediction mode, both of which operate on a single reference frame and allow only one motion vector.

e) Overlapped block motion compensation: The overlapped block motion compensation mode modifies the original design in [28] to account for variable block sizes [29]. It exploits the immediate spatial neighbors' motion information to improve the prediction quality for pixels near its top and left boundaries, where the true motion trajectory correlates with the motion vectors on both sides.

It first scans through the immediate neighbors above and finds up to four reference blocks that have the same reference frame as the current block. An example is shown in Fig. 14(a), where the blocks are marked according to

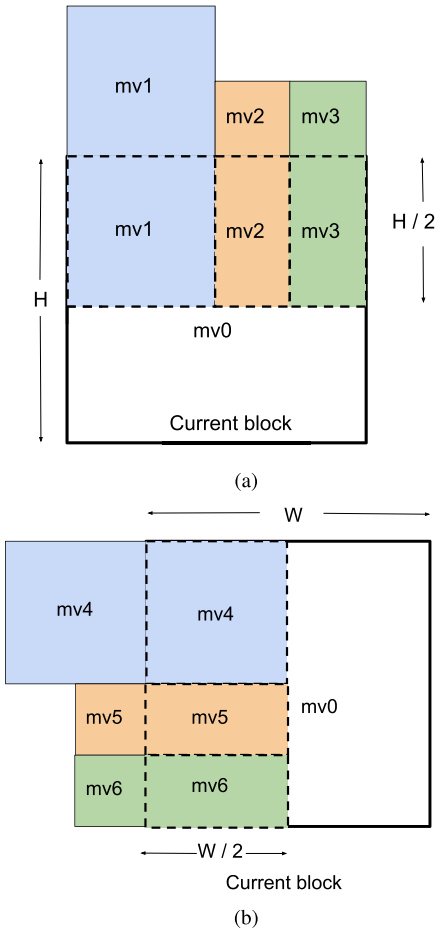


Fig. 14. ***Overlapped block motion compensation using (a) top and (b) left neighboring blocks' motion information, respectively.***

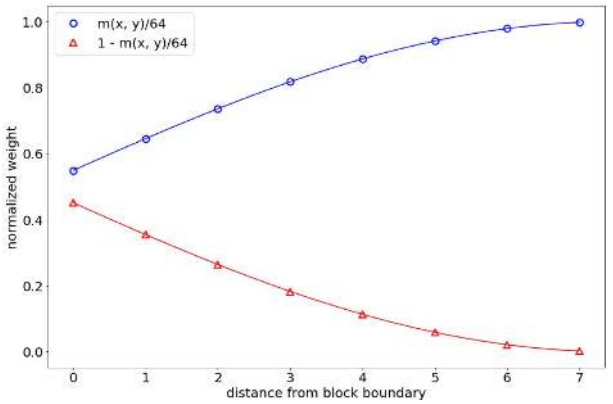


Fig. 15. ***Normalized weights for OBMC with $H = 16$ or $W = 16$.***

their scan order. The motion vector of each selected reference block is employed to generate a motion-compensated block that extends from the top boundary toward the center of the current block. Its width is the same as the reference block's width, and its height is half of the current block's height, as shown in Fig. 14(a). An intermediate blending result is formed as

$$P_{\text{int}}(x,y) = m(x,y)R_1(x,y) + (64 - m(x,y))R_{\text{above}}(x,y) \tag{15}$$

where $R_1(x,y)$ is the original motion-compensated pixel at position (x,y) using current block's motion vector mv_0, and $R_{\text{above}}(x,y)$ is the pixel from the overlapped reference block. The weight $m(x,y)$ follows a raised cosine function:

$$m(x,y) = 64 * \left(\frac{1}{2}\sin\left(\frac{\pi}{H}\left(y+\frac{1}{2}\right)\right)+\frac{1}{2}\right) \tag{16}$$

where $y = 0, 1, \ldots, H/2 - 1$ is the row index, and H is the current block height. The weight distribution for $H = 16$ is shown in Fig. 15.

The scheme next processes the immediate left neighbors to extract the available motion vectors and build overlapped reference blocks extending from the left boundary toward the center, as shown in Fig. 14(b). The final prediction is calculated by

$$P(x,y) = m(x,y)P_{\text{int}}(x,y) + (64 - m(x,y))R_{\text{left}}(x,y) \tag{17}$$

where $R_{\text{left}}(x,y)$ is the pixel from the left-hand side overlapped reference block. The weight $m(x,y)$ is a raised cosine function of the column index x

$$m(x,y) = 64 * \left(\frac{1}{2}\sin\left(\frac{\pi}{W}\left(x+\frac{1}{2}\right)\right)+\frac{1}{2}\right) \tag{18}$$

where $x = 0, 1, \ldots, W/2 - 1$ and W is the current block width.

f) Compound inter–intra-predictor: This mode combines an intraprediction block and a translational interprediction block. The intraprediction is selected among the dc, vertical, horizontal, and smooth modes (see Section V-B2). The combination can be achieved through either a wedge mask similar to the compound intercase above or a preset coefficient set that gradually reduces the intraprediction weight along its prediction direction. Examples of the preset coefficients for each intramode are shown in Fig. 16.

As discussed above, AV1 supports a large variety of compound prediction tools. Exercising each mode in the

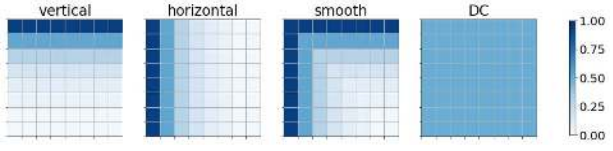


Fig. 16. ***Normalized weight masks of compound inter–intra-prediction for 8×8 blocks.***

rate-distortion optimization framework fully realizes their potential, at the cost of bringing a significant complexity load for the encoder. Efficient selection of the appropriate compound coding modes without extensive rate-distortion optimization searches remains a challenge.

D. Dynamic Motion Vector Referencing Scheme

Motion vector coding accounts for a sizable portion of the overall bit rate. Modern video codecs typically adopt predictive coding for motion vectors and code the difference using entropy coding [30], [31]. The prediction accuracy has a large impact on the coding efficiency. AV1 employs a dynamic motion vector referencing scheme that obtains candidate motion vectors from the spatial and temporal neighbors and ranks them for efficient entropy coding.

1) Spatial Motion Vector Reference: A coding block will search its spatial neighbors in the unit of 8×8 luma samples to find the ones that have the same reference frame index as the current block. For compound interprediction modes, this means the same reference frame pairs. The search region contains three 8×8 block rows above the current block and three 8×8 block columns to the left. The process is shown in Fig. 17, where the search order is shown by the index. It starts from the nearest row and column and interleaves the outer rows and columns. The TR 8×8 block is included if available. The first eight different motion vectors encountered will be recorded, along with a frequency count and whether they appear in the nearest row or column. They will then be ranked, as discussed in Section V-D4.

Note that the minimum coding block size in AV1 is 4×4. Hence, an 8×8 unit has up to four different motion vectors and reference frame indexes to search through. This would require a hardware decoder to store all the motion information at 4×4 unit precision for the three 8×8 block rows above. Hardware decoders typically use a line buffer concept, which is a dedicated buffer in the static random access memory (SRAM), a fast and expensive unit. The line buffer maintains coding information corresponding to an entire row of a frame, which will be used as context information for later coding blocks. An example of the line buffer concept is shown in Fig. 18. The line buffer size is designed for the worst case that corresponds to the maximum frame width allowed by the specification. To make the line buffer size economically feasible, AV1 adopts a design that only accesses 4×4 block motion information in the immediate above row (the green region in Fig. 17). For the rest of the rows, the codec only uses the motion information for 8×8 units. If an 8×8 block is coded using 4×4 blocks, the bottom-right 4×4 block's information will be used to represent the entire 8×8 block, as shown in Fig. 17. This halves the amount of space needed for motion data in the line buffer.

The storage of the coding context to the left, on the other hand, depends on the superblock size and is agnostic to the frame size. It has far less impact on the SRAM space. However, we keep its design symmetric to the above context to avoid the motion vector ranking system described in Section V-D4 favoring either side.

2) Motion Field Motion Vector Reference: Common practice extracts the temporal motion vector by referring to the collocated blocks in the reference frames [31] [32]. Its efficacy, however, is largely limited to capture motion trajectories at low velocities. To reliably track the motion trajectory for efficient motion vector prediction, AV1 uses a motion field approach [33].

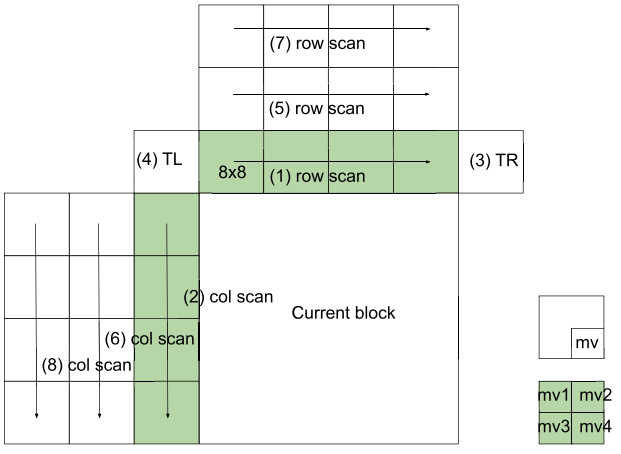


Fig. 17. *Spatial reference motion vector search pattern. The index ahead of each operation represents the processing order. TL stands for the top-left 8 × 8 block. TR stands for the top-right 8 × 8 block.*

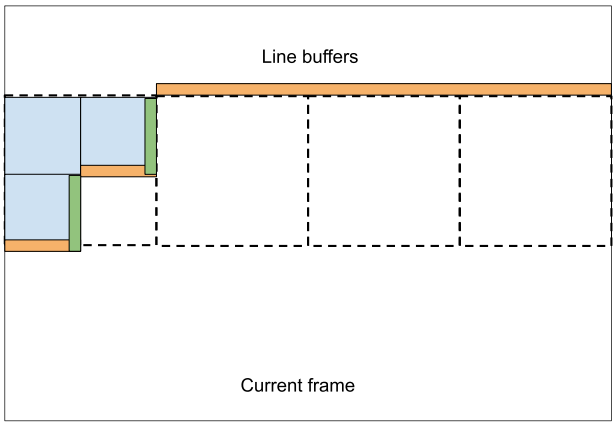


Fig. 18. *Line buffer, shown in orange, stores the coding information associated with an entire row of a frame. The dashed line shows superblocks. The information in the line buffer will be used as above context by later coding blocks across superblock boundaries. The line buffer is updated as new coding blocks (in blue) are processed. In contrast, the green shows coding information to be used as left context by later blocks, the length of which corresponds to the size of a superblock.*

A motion field is created for each reference frame ahead of processing the current frame. First, we build motion trajectories between the current frame and the previously coded frames by exploiting motion vectors from previously coded frames through either linear interpolation or extrapolation. The motion trajectories are associated with 8×8 blocks in the current frame. Next, the motion field between the current frame and a given reference frame can be formed by extending the motion trajectories from the current frame toward the reference frame.

Interpolation: The motion vector pointing from a reference frame to a prior frame crosses the current frame. An example is shown in Fig. 19. The frames are drawn in display order. The motion vector ref_mv at block (ref_blk_row, ref_blk_col) in the reference frame (shown in orange) goes through the current frame. The distance that ref_mv spans is denoted by d_1. The distance between the current frame and the reference frame where ref_mv originates is denoted by d_3. The intersection is located at the block position

$$\begin{bmatrix} \text{blk\_row} \\ \text{blk\_col} \end{bmatrix} = \begin{bmatrix} \text{ref\_blk\_row} \\ \text{ref\_blk\_col} \end{bmatrix} + \begin{bmatrix} \text{ref\_mv.row} \\ \text{ref\_mv.col} \end{bmatrix} \cdot \frac{d_3}{d_1}. \quad (19)$$

The motion field motion vector that extends from block (blk_row, blk_col) in the current frame toward a reference frame along the motion trajectory, e.g., mf_mv in blue, is calculated as

$$\begin{bmatrix} \text{mf\_mv.row} \\ \text{mf\_mv.col} \end{bmatrix} = - \begin{bmatrix} \text{ref\_mv.row} \\ \text{ref\_mv.col} \end{bmatrix} \cdot \frac{d_2}{d_1} \quad (20)$$

where d_2 is the distance between the current frame and the target reference frame that the motion field is built for.

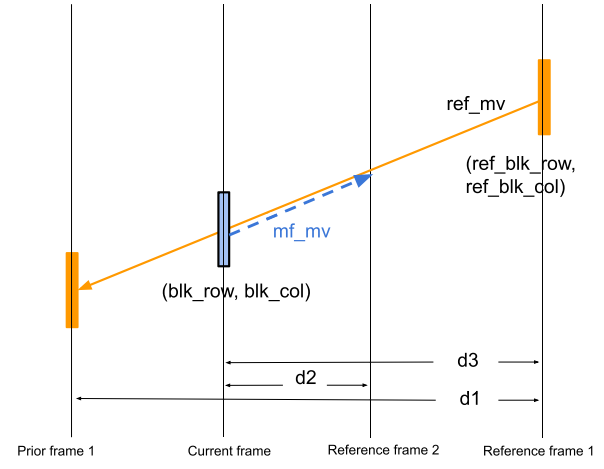


Fig. 19. ***Building motion trajectory through motion vector interpolation.***

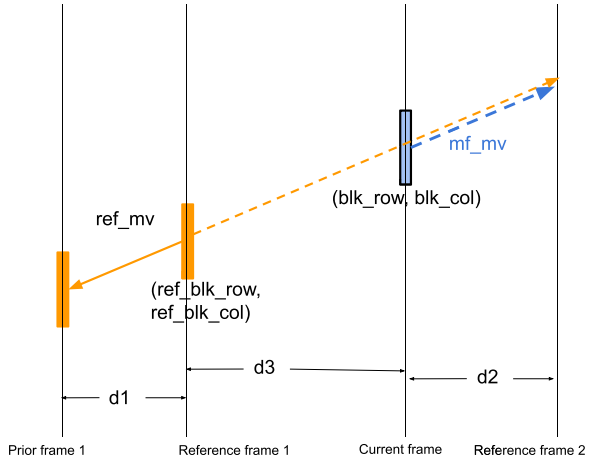


Fig. 20. ***Building motion trajectory through motion vector extrapolation.***

Extrapolation: The motion vector from a reference does not cross the current frame. An example is shown in Fig. 20. The motion vector ref_mv (in orange) points from reference frame 1 to a prior frame 1. It is extended toward the current frame, and they meet at block position

$$\begin{bmatrix} \text{blk\_row} \\ \text{blk\_col} \end{bmatrix} = \begin{bmatrix} \text{ref\_blk\_row} \\ \text{ref\_blk\_col} \end{bmatrix} - \begin{bmatrix} \text{ref\_mv.row} \\ \text{ref\_mv.col} \end{bmatrix} \cdot \frac{d_3}{d_1}. \quad (21)$$

Its motion field motion vector toward reference frame 2, mf_mv (in blue), is given by

$$\begin{bmatrix} \text{mf\_mv.row} \\ \text{mf\_mv.col} \end{bmatrix} = - \begin{bmatrix} \text{ref\_mv.row} \\ \text{ref\_mv.col} \end{bmatrix} \cdot \frac{d_2}{d_1} \quad (22)$$

where d_2 is the distance between the current frame and reference frame 2 in Fig. 20. Note that the signs in both (20) and (22) depend on whether the two reference frames are on the same side of the current frame.

Typically, interpolation provides better estimation accuracy than extrapolation. Therefore, when a block has possible motion trajectories originated from both, the extrapolated one will be discarded. A coding block uses the motion field of all its 8×8 subblocks as its temporal motion vector reference.

3) Hardware Constraints: The motion information, including the motion vector and the reference frame index, needs to be stored for later frames to build their motion fields. To reduce the memory footprint, the motion information is stored in units of 8×8 blocks. If a coding block is using compound modes, only the first motion vector is saved. The reference frame motion information is commonly stored in the dynamic random access memory (DRAM), a relatively cheaper and slower unit as compared to SRAM, in hardware decoders. It needs, however, to be transferred to SRAM for computing purposes. The bus between DRAM and SRAM is typically 32 bits wide.

To facilitate efficient data transfer, a number of data format constraints are employed. We limit the codec to use motion information from up to four reference frames (out of seven available frames) to build the motion field. Therefore, only 2 bits are needed for the reference frame index. Furthermore, a motion vector with any component magnitude above 2^{12} will be discarded. As a result, the motion vector and reference frame index together can be represented by a 32-bit unit.

As mentioned in Section V-A1, hardware decoders process frames in 64×64 block units, which makes the hardware cost invariant to the frame size. In contrast, the above motion field construction can potentially involve any motion vector in the reference frame to build the motion field for a 64×64 block, which makes the hardware cost grow as the frame resolution scales up.

To solve this problem, we constrain the maximum displacement between (ref_blk_row, ref_blk_col) and (blk_row, blk_col) during the motion vector projection. Let (base_row, base_col) denote the top-left block position of the 64×64 block that contains (ref_blk_row, ref_blk_col)

$$\text{base\_row} = (\text{ref\_blk\_row} \gg 3) \ll 3 \tag{23}$$

$$\text{base\_col} = (\text{ref\_blk\_col} \gg 3) \ll 3. \tag{24}$$

The maximum displacement constraints are

$$\text{blk\_row} \in [\text{base\_row}, \text{base\_row} + 8) \tag{25}$$

$$\text{blk\_col} \in [\text{base\_col} - 8, \text{base\_col} + 16). \tag{26}$$

Note that all the indexes here are in 8×8 luma sample block units. Any projection in (19) or (21) that goes beyond this limit will be discarded. This design localizes the reference region in the reference frame used to produce the motion field for a 64×64 pixel block to be a $64 \times (64 + 2 \times 64)$ block, as shown in Fig. 21. It allows the codec to load the necessary reference motion vectors per 64×64 block from DRAM to SRAM and process the linear projection ahead of decoding each 64×64 block. Note that we allow the width value to be larger than the height since the shaded portion of the reference motion vector array can be readily reused for decoding the next 64×64.

4) Dynamic Motion Vector Reference List: Having established the spatial and temporal reference motion vectors, we will next discuss the scheme to use them for efficient motion vector coding. The spatial and temporal reference motion vectors are classified into two categories based on where they appear: the nearest spatial neighbors and the rest. Statistically, the motion vectors from immediate above, left, and TR blocks tend to have a higher correlation with the current block than the rest and, hence, are considered with higher priority. Within each category, the motion vectors are ranked in descending order of their appearance counts within the spatial and temporal search range. A motion vector candidate with a higher appearance count is considered to be "popular" in the local region, i.e., a higher prior probability. The two categories are concatenated to form a ranked list.

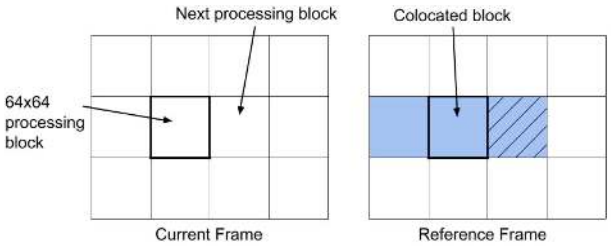


Fig. 21. *Constrained projection localizes the referencing region needed to produce the motion field for a 64 × 64 block. The colocated block in the reference frame is at the same location as the processing block in the current frame. The blue region is the extended block whose motion vectors are used to estimate the motion field for the current 64 × 64 block.*

The first four motion vectors in this ranked list will be used as candidate motion vector predictors. The encoder will pick the one that is closest to the desired motion vector and send its index to the decoder. It is not uncommon for coding blocks to have fewer than four candidate motion vectors, due to either the high flexibility in the reference frame selection, or a highly consistent motion activity in the local region. In such a context, the candidate motion vector list will be shorter than 4, which allows the codec to save bits spent on identifying the selected index. The dynamic candidate motion vector list is in contrast to the design in VP9, where one always constructs two candidate motion vectors. If not enough candidates are found, the VP9 codec will fill the list with zero vectors. AV1 also supports a special intermode that makes the interpredictor use the frame-level affine model, as discussed in Section V-C2.

The motion vector difference will be entropy coded. Since a significant portion of the coding blocks will find a zero motion vector difference, the probability model is designed to account for such bias. AV1 allows a coding block to use 1 bit to indicate whether to directly use the selected motion vector predictor as its final motion vector or to additionally code the difference. The probability model for this entropy-coded bit is conditioned on two factors: whether its spatial neighbors have a nonzero motion vector difference and whether a sufficient number of motion vector predictors are found. For compound modes, where two motion vectors need to be specified, this extends to four cases that cover where either block, both, or neither one have a zero difference motion vector. The nonzero difference motion vector coding is consistent in all cases.

E. Transform Coding

Transform coding is applied to the prediction residual to remove the potential spatial correlations. VP9 uses a uniform transform block size design, where all the transform

blocks within a coding block share the same transform size. Four square transform sizes are supported by VP9, 4×4, 8×8, 16×16, and 32×32. A set of separable 2-D transform types, constructed by combinations of 1-D discrete cosine transform (DCT) and asymmetric discrete sine transform (ADST) kernels [34], [35], is selected based on the prediction mode. AV1 inherits the transform coding scheme in VP9 and extends its flexibility in terms of both the transform block sizes and the kernels.

1) Transform Block Size: AV1 extends the maximum transform block size to 64×64. The minimum transform block size remains 4×4. In addition, rectangular transform block sizes at $N \times N/2$, $N/2 \times N$, $N \times N/4$, and $N/4 \times N$ are supported to complement the rectangular coding block sizes in Section V-A.

A recursive transform block partition approach is adopted in AV1 for all the intercoded blocks to capture localized stationary regions for transform coding efficiency. The initial transform block size matches the coding block size, unless the coding block size is above 64×64; in that case, the 64×64 transform block size is used. For the luma component, up to two levels of transform block partitioning are allowed. The recursive partition rules for $N \times N$, $N \times N/2$, and $N \times N/4$ coding blocks are shown in Fig. 22.

The intracoded block inherits the uniform transform block size approach, i.e., all transform blocks have the same size. Similar to the interblock case, the maximum transform block size matches the coding block size and can go up to two levels down for the luma component. The available options for square and rectangular coding block sizes are shown in Fig. 23.

The chroma components tend to have much fewer variations in their statistics. Therefore, the transform block is set to use the largest available size.

2) Transform Kernels: Unlike VP9 where each coding block has only one transform type, AV1 allows each transform block to choose its own transform kernel independently. The 2-D separable transform kernels are extended to combinations of four 1-D kernels: DCT, ADST, flipped ADST (FLIPADST), and identity transform (IDTX), resulting in a total of 16 2-D transform kernels. The FLIPADST is a reverse of the ADST kernel. The kernels are selected based on statistics and to accommodate various boundary conditions. The DCT kernel is widely used in signal compression and is known to approximate the optimal linear transform, the Karhunen–Loeve transform (KLT), for consistently correlated data. The ADST, on the other hand, approximates the KLT where one-sided smoothness is assumed and, therefore, is naturally suitable for coding some intraprediction residuals. Similarly, the FLIPADST captures one-sided smoothness from the opposite end. The IDTX is further included to accommodate situations where sharp transitions are contained in the block and neither DCT nor ADST is effective. Also, the IDTX, combined with other 1-D transforms, provides the 1-D transforms themselves, therefore allowing for better compression of horizontal and vertical patterns in the residual [36]. The waveforms corresponding to the four 1-D transform kernels are presented in Fig. 24 for dimension $N = 8$.

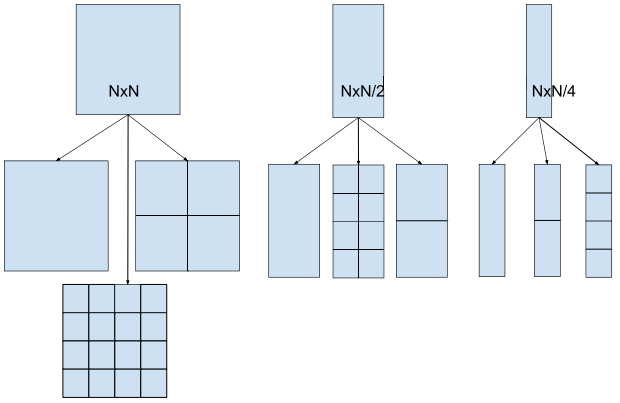


Fig. 23. *Transform block size options for square and rectangular intrablocks.*

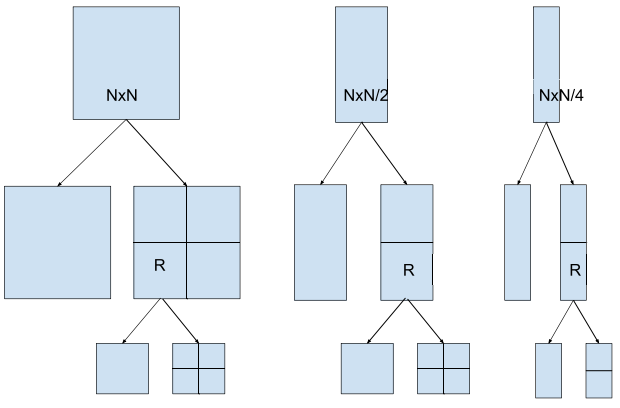


Fig. 22. *Transform block partition for square and rectangular interblocks. R denotes the recursive partition point. Each coding block allows a maximum two-level recursive partition.*

Even with modern single-instruction–multiple-data (SIMD) architectures, the inverse transform accounts for a significant portion of the decoder computational cost. The butterfly structure [37] allows a substantial reduction in multiplication operations over plain matrix multiplication, i.e., a reduction from $O(N^2)$ to $O(N\log N)$, where N is the transform dimension. Hence, it is highly desirable for large transform block sizes. Note that since the original ADST derived in [35] cannot be decomposed for the butterfly structure, a variant of it, as introduced in [38] and also as shown in Fig. 24, is adopted by AV1 for transform block sizes of 8×8 and above.

When the transform block size is large, the boundary effects are less pronounced, in which setting the transform coding gains of all sinusoidal transforms largely converge [35]. Therefore, only the DCT and IDTX are

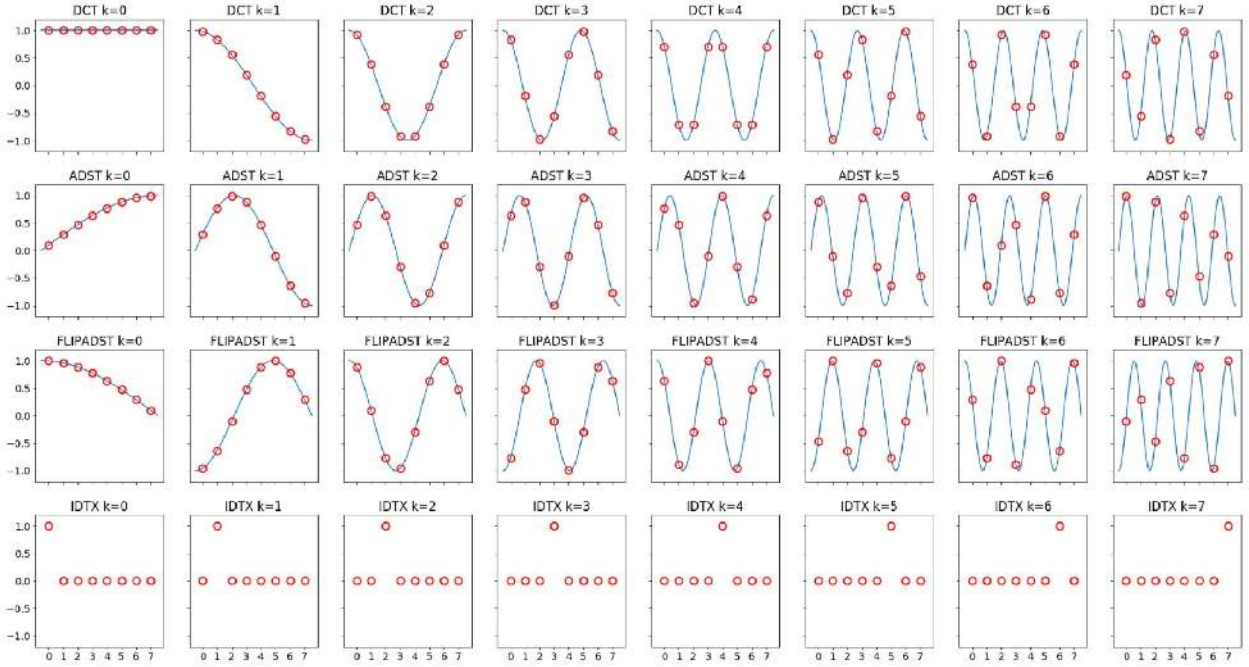


Fig. 24. *Transform kernels of DCT, ADST, FLIPADST, and IDTX for dimension $N = 8$. The discrete basis values are displayed as red circles, with blue lines indicating the associated sinusoidal function. The bases of DCT and ADST (a variant with a fast butterfly structured implementation) take the form of $\cos(((2n+1)k\pi)/2N)$ and $\sin(((2n+1)(2k+1)\pi)/4n)$, respectively, where n and k denote time index and the frequency index, taking values from $\{0, 1, \ldots, N-1\}$. FLIPADST utilizes the reversed ADST bases, and IDTX denotes the identity transformation.*

employed for transform blocks at dimension 32×32 and above.

F. Quantization

The transform coefficients are quantized, and the quantization indexes are entropy coded. The quantization parameter (QP) in AV1 ranges between 0 and 255.

1) Quantization Step Size: At a given QP, the quantization step size for the dc coefficient is smaller than that for the ac coefficient. The mapping from QP to quantization step size for both dc and ac coefficients is drawn in Fig. 25. The lossless coding mode is achieved when QP is 0. By default, all the ac coefficients will use the same quantization step size. Since the human visual system tends to have different tolerance to distortions at various frequencies, AV1 also supports 15 sets of predefined quantization weighting matrices, where the quantization step size for each individual frequency component is further scaled differently. Each frame can optionally select a quantization weighting matrix set for luma and chroma planes, respectively.

2) Quantization Parameter Modulation: AV1 assigns a base QP for a coded frame, denoted by QP_{base}. The QP values for the dc and ac coefficients in both luma and chroma components are shown in Table 1. $\Delta QP_{p,b}$ are additional offset values transmitted in the frame header,

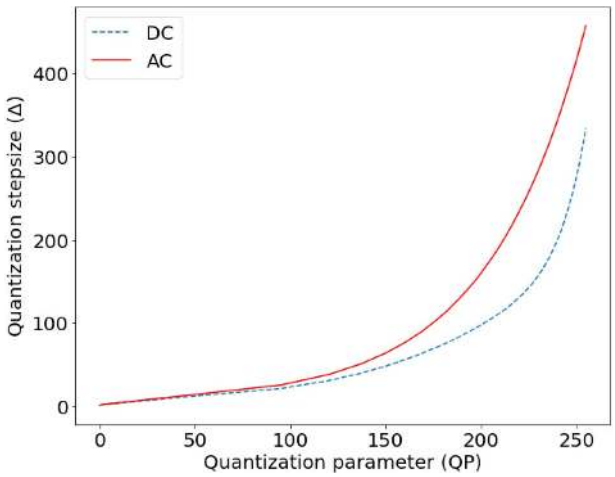


Fig. 25. *QP and quantization step size maps for dc and ac coefficients.*

where $p \in \{Y, U, V\}$ denotes the plane and $b \in \{dc, ac\}$ denotes the dc or the ac transform coefficients.

Recognizing that the coding blocks within a frame may have different rate-distortion tradeoffs, AV1 further allows QP offset at both superblock and coding block levels. The resolution of QP offset at superblock level is signaled by the frame header. The available options are 1, 2, 4, and 8. The coding block-level QP offset can be achieved through

Table 1 Frame-Level QP Values (QP_{frame}) for Y/U/V Planes

	AC	DC
Y	QP_{base}	QP_{base} + $\Delta QP_{Y,DC}$
U	QP_{base} + $\Delta QP_{U,AC}$	QP_{base} + $\Delta QP_{U,DC}$
V	QP_{base} + $\Delta QP_{V,AC}$	QP_{base} + $\Delta QP_{V,DC}$

segmentation. AV1 allows a frame to classify its coding blocks into up to eight segments; each has its own QP offset decided by the frame header. The segment index associated with each coding block is sent through the bitstream to the decoder.

Therefore, the effective QP for ac coefficients in a coding block, QP_{cb}, is given by

$$\text{QP}_{cb} = \text{clip}(\text{QP}_{\text{frame}} + \Delta\text{QP}_{sb} + \Delta\text{QP}_{\text{seg}}, 1, 255) \tag{27}$$

where ΔQP_{sb} and ΔQP_{seg} are the QP offsets from the superblock and the segment, respectively. The clip function ensures it stays within a valid range. The QP is not allowed to change from a nonzero value to zero since zero is reserved for lossless coding.

The decoder rebuilds the quantized samples using a uniform quantizer. Given the quantization step size Δ and the quantization index k, the reconstructed sample is $k\Delta$.

VI. ENTROPY CODING SYSTEM

AV1 employs an M-ary symbol arithmetic coding method that was originally developed for the Daala video codec [39] to compress the syntax elements, where integer $M \in [2,\ 14]$. The probability model is updated per symbol coding.

A. Probability Model

Consider an M-ary random variable whose probability mass function (PMF) at time n is defined as

$$\bar{P}_n = \Big[p_1(n), p_2(n), \ldots, p_M(n)\Big] \tag{28}$$

and the cumulative distribution function (CDF) given by

$$\bar{C}_n = \Big[c_1(n), c_2(n), \ldots, c_{M-1}(n), 1\Big] \tag{29}$$

where $c_k(n) = \sum_{i=1}^{k} p_i(n)$. When the symbol is coded, a new outcome $k \in \{1, 2, \ldots, M\}$ is observed. The probability model is then updated as

$$\bar{P}_n = \bar{P}_{n-1}(1-\alpha) + \alpha\bar{e}_k \tag{30}$$

where $\bar{e}_k$ is an indicator vector whose kth element is 1 and the rest are 0, and α is the update rate.

To update the CDF, we first consider $c_m(n)$ where $m < k$

$$\begin{aligned} c_m(n) = \sum_{i=1}^{m} p_i(n) &= \sum_{i=1}^{m} p_i(n-1)\cdot(1-\alpha) \\ &= c_m(n-1)\cdot(1-\alpha). \end{aligned}$$

For $m \geq k$ cases, we have

$$\begin{aligned} 1 - c_m(n) = \sum_{i=m+1}^{M} p_i(n) &= \sum_{i=m+1}^{M} p_i(n-1)\cdot(1-\alpha) \\ &= (1 - c_m(n-1))\cdot(1-\alpha) \end{aligned}$$

where the second equation follows (30) and $m+1 > k$. Rearranging the terms, we have

$$c_m(n) = c_m(n-1) + \alpha\cdot(1 - c_m(n-1)). \tag{31}$$

In summary, the CDF is updated as

$$c_m(n) = \begin{cases} c_m(n-1)\cdot(1-\alpha), & m < k \\ c_m(n-1) + \alpha\cdot(1 - c_m(n-1)), & m \geq k. \end{cases} \tag{32}$$

AV1 stores M-ary symbol probabilities in the form of CDFs. The elements in (29) are scaled by 2^{15} for integer precision. The arithmetic coding directly uses the CDFs to compress symbols [40].

The probability update rate associated with a symbol adapts based on the count of this symbol's appearance within a frame

$$\alpha = \frac{1}{2^{3+I(\text{count}>15)+I(\text{count}>32)+\min(\log_2(M),2)}} \tag{33}$$

where $I(\text{event})$ is 1 if the event is true, and 0 otherwise. It allows higher adaptation rate at the beginning of each frame. The probability models are inherited from one of the reference frames whose index is signaled in the bitstream.

B. Arithmetic Coding

The M-ary symbol arithmetic coding largely follows [40] with all the floating-point data scaled by 2^{15} and represented by 15-bit unsigned integers. To improve hardware throughput, AV1 adopts a dual model approach to make the involved multiplications fit in 16 bits. The probability model CDF is updated and maintained a 15-bit precision, but, when it is used for entropy coding, only the most significant 9 bits are fed into the arithmetic coder, as shown in Fig. 26.

Let R denote the arithmetic coder's current interval length, and Value denotes the code string value. The decoding processing is depicted in Algorithm 1. Note that the interval length R is scaled down by 1/256 prior to the

Algorithm 1 Modified Arithmetic Decoder Operations

$low \leftarrow R$
for $k = 1;\ Value < low;\ k = k + 1$ **do**
  $up \leftarrow low$
  $f \leftarrow 2^9 - (c_k >> 6)$
  $low \leftarrow ((R >> 8) \times f) >> 1$
end for
$R \leftarrow up - low$
$Value \leftarrow Value - low$

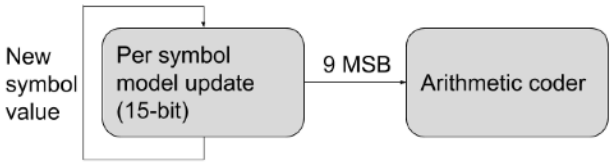


Fig. 26. *Probability model is updated and maintained in 15-bit precision, while only the most significant 9 bits are used by the arithmetic coder.*

multiplication, which makes the product $(R >> 8) \times f)$ fits into 16 bits.

C. Level Map Transform Coefficient Coding System

The transform coefficient entropy coding system is an intricate and performance-critical component in video codecs. We discuss its design in AV1 that decomposes it into a series of symbol codings.

1) Scan Order: A 2-D quantized transform coefficient matrix is first mapped into a 1-D array for sequential processing. The scan order depends on the transform kernel (see Section V-E2). A column scan is used for 1-D vertical transform, and a row scan is used for 1-D horizontal transform. In both settings, we consider that the use of 1-D transform indicates a strong correlation along the selected direction and weak correlation along the perpendicular direction. A zig-zag scan is used for both 2-D transform and identity matrix (IDTX), as shown in Fig. 27.

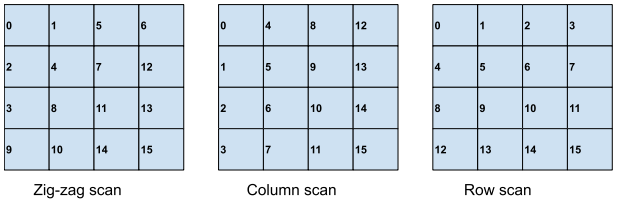


Fig. 27. *Scan order is decided by the transform kernel. An example is drawn for 4 × 4 transform blocks. The index represents the scan order. Left: zig-zag scan for 2-D transform block. Middle: column scan for 1-D vertical transform. Right: row scan for 1-D horizontal transform.*

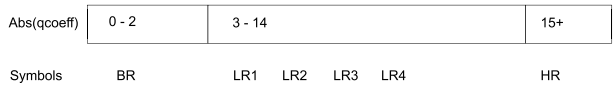


Fig. 28. *Absolute value of a quantized transform coefficient V is decomposed into BR, LR, and HR symbols.*

2) Symbols and Contexts: The index of the last nonzero coefficient in the scan order is first coded. The coefficients are then processed in reverse scan order. The range of a quantized transform coefficient is $[-2^{15}, 2^{15})$. In practice, the majority of quantized transform coefficients are concentrated close to the origin. Hence, AV1 decomposes quantized transform coefficients into four symbols.

1) *Sign bit:* When it is 1, the transform coefficient is negative; otherwise, it is positive.
2) *Base Range (BR):* The symbol contains four possible outcomes $\{0, 1, 2, > 2\}$, which are the absolute values of the quantized transform coefficient. An exception is for the last nonzero coefficient, where BR$\in \{1, 2, > 2\}$, since 0 has been ruled out.
3) *Low Range (LR):* It contains four possible outcomes $\{0, 1, 2, > 2\}$ that correspond to the residual value over the previous symbols' upper limit.
4) *High Range (HR):* The symbol has a range of $[0, 2^{15})$ and corresponds to the residual value over the previous symbols' upper limit.

To code a quantized transform coefficient V, one first processes its absolute value. As shown in Fig. 28, if $|V| \in [0, 2]$, the BR symbol is sufficient to signal it, and the coding of $|V|$ is terminated. Otherwise, the outcome of the BR symbol will be ">2"; in that case, an LR symbol is used to signal $|V|$. If $V \in [3, 5]$, this LR symbol will be able to cover its value and complete the coding. If not, the second LR is used to further code $|V|$. This is repeated up to four times, which effectively covers the range $[3, 14]$. If $|V| > 14$, an additional HR symbol is coded to signal $(|V| - 14)$.

The probability model of symbol BR is conditioned on the previously coded coefficients in the same transform block. Since a transform coefficient can have correlations with multiple neighboring samples [41], we extend the reference samples from two spatially nearest neighbors in VP9 to a region that depends on the transform kernel, as shown in Fig. 29. For 1-D transform kernels, it uses three coefficients after the current sample along the transform direction. For 2-D transform kernels, up to five neighboring coefficients in the immediate right-bottom region are used. In both cases, the absolute values of the reference coefficients are added, and the sum is considered as the context for the probability model of BR.

Similarly, the probability model of symbol LR is designed, as shown in Fig. 30, where the reference region for 2-D transform kernels is reduced to the nearest three coefficients. The symbol HR is coded using Exp-Golomb code [42].

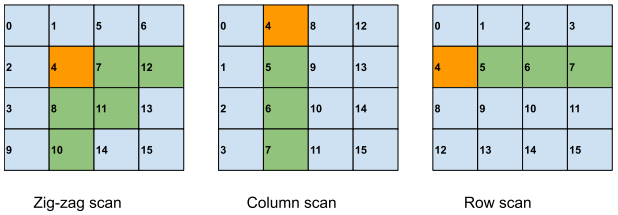


Fig. 29. *Reference region for symbol BR. Left: coefficient (in orange) in a 2-D transform block uses five previously processed coefficients (in green) to build the context for its conditional probability model. Middle and Right: coefficient (in orange) in a 1-D transform block uses three previously processed coefficients (in green) along the transform direction to build the context for its conditional probability model.*

The sign bit is only needed for nonzero quantized transform coefficients. Since the sign bits of ac coefficients are largely uncorrelated, they are coded in raw bits. To improve hardware throughput, all the sign bits of ac coefficients within a transform block are packed together for transmission in the bitstream, which allows a chunk of data to bypass the entropy coding route in hardware decoders. The sign bit of the dc coefficient, on the other hand, is entropy coded using a probability model conditioned on the sign bits of the dc coefficients in the above and left transform blocks.

VII. POSTPROCESSING FILTERS

AV1 allows three optional in-loop filter stages: a deblocking filter, a constrained directional enhancement filter (CDEF), and a loop restoration filter, as illustrated in Fig. 31. The filtered output frame is used as a reference frame for later frames. A normative film grain synthesis stage can be optionally applied prior to display. Unlike the in-loop filter stages, the results of the film grain synthesis stage do not influence the prediction for subsequent frames. It is, hence, referred to as an out-of-loop filter.

A. Deblocking Filter

The deblocking filter is applied across the transform block boundaries to remove block artifacts caused by the

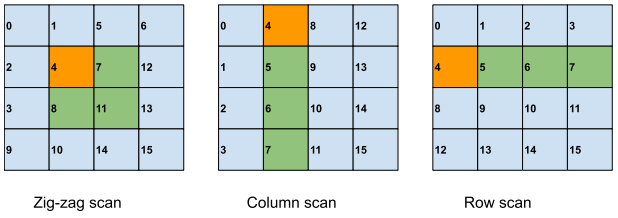


Fig. 30. *Reference region for symbol LR. Left: coefficient (in orange) in a 2-D transform block uses three previously processed coefficients (in green) to build the context for its conditional probability model. Middle and Right: coefficient (in orange) in 1-D transform block uses three previously processed coefficients (in green) along the transform direction to build the context for its conditional probability model.*

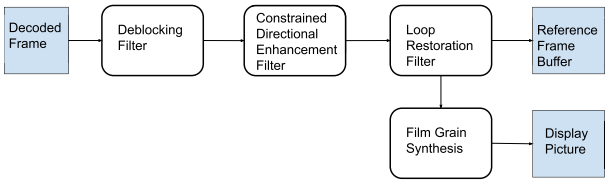


Fig. 31. *AV1 allows three optional in-loop filter stages including a deblocking filter, a CDEF, and a loop restoration filter. A normative film grain synthesis stage is supported for the displayed picture.*

quantization error. The logic for the vertical and horizontal edges is fairly similar. We use the vertical edge case to present the design principles.

1) Filter Length: AV1 supports four-, eight-, and 14-tap FIR filters for the luma components and four- and six-tap FIR filters for chroma components. All the filter coefficients are preset in the codec. The filter length is decided by the minimum transform block sizes on both sides. For example, in Fig. 32, the length of filter1 is given by $min(tx\_width1, tx\_width2)$, whereas the length of filter2 is given by $min(tx\_width1, tx\_width3)$. If the transform block dimension is 16 or above on both sides, the filter length is set to be 14.

Note that this selected filter length is the maximum filter length allowed for a given transform block boundary. The final filter further depends on a flatness metric discussed next.

2) Boundary Conditions: The FIR filters used by the deblocking stage are low-pass filters. To avoid blurring an actual edge in the original image, edge detection is conducted to disable the deblocking filter at transitions that contain a high variance signal. We use notations shown in Fig. 33, where the dashed line shows the pixels near the transform block boundary. Denote the pixels on the two sides p_0–p_6 and q_0–q_6. We consider the transition along the lines p_6–q_6 high variance and hence disable the deblocking filter if any of the following conditions is true.

1) $|p_1 - p_0| > T_0$.
2) $|q_1 - q_0| > T_0$.
3) $2|p_0 - q_0| + ((|p_1 - q_1|)/2) > T_1$.

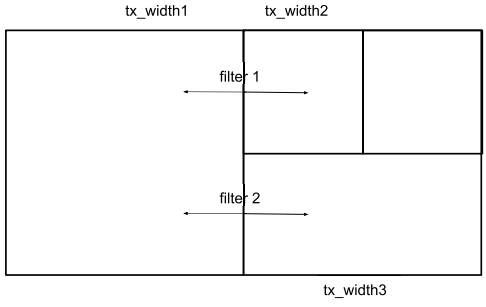


Fig. 32. *Filter length is decided by the minimum transform block sizes on both sides.*

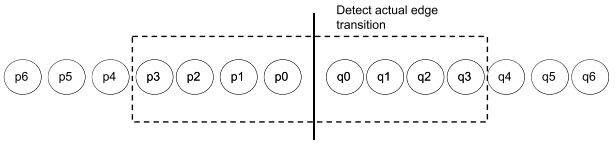


Fig. 33. *Pixels at a transform block boundary. The dashed line shows the pixels near the transform block boundary. p_0–p_6 and q_0–q_6 are the pixels on the two sides.*

If the filter length is 8 or 14, two additional samples are checked to determine if the transition contains a high variance signal.

1) $|p_3 - p_2| > T_0$.
2) $|q_3 - q_2| > T_0$.

The thresholds T_0 and T_1 can be decided on a superblock by superblock basis. A higher threshold allows more transform block boundaries to be filtered. In AV1, these thresholds can be independently set in the bitstream for the vertical and horizontal edges in the luma component and for each chroma plane.

To avoid the ringing artifacts, AV1 further requires that a long filter is only used when both sides are "flat." For the eight-tap filter, this requires $|q_k - q_0| \leq 1$ and $|p_k - p_0| \leq 1$, where $k \in \{1, 2, 3\}$. For the 14-tap filter, the condition extends to $k \in \{1, 2, \ldots, 6\}$. If any flatness condition is false, the codec reverts to a shorter filter for that boundary.

B. Constrained Directional Enhancement Filter

The CDEF allows the codec to apply a nonlinear deringing filter along certain (potentially oblique) directions [43]. It operates in 8×8 units. As presented in Fig. 34, eight preset directions are defined by rotating and reflecting the three shown templates. The decoder uses the reconstructed pixels to select the prevalent direction index by minimizing

$$E_d^2 = \sum_k \sum_{p \in P_{d,k}} (x_p - \mu_{d,k})^2 \tag{34}$$

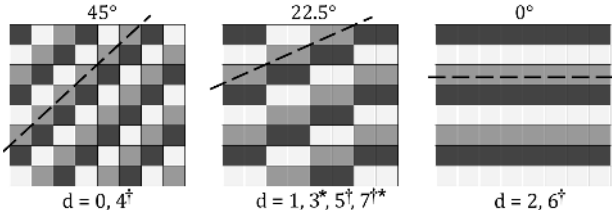


Fig. 34. *Templates of preset directions and their associated directions. The templates correspond to directions of 45°, 22.5°, and 0°, as shown by the dash lines. Each preset direction $d \in \{0, \ldots, 7\}$ can be obtained by using the template directly, rotating the template by 90° clockwise (marked by †) or reflecting the template along the horizontal axis (marked by ⋆).*

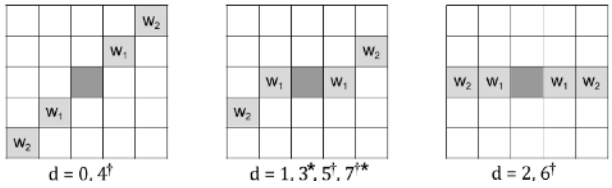


Fig. 35. *Primary filter templates associated with direction $d \in \{0, \ldots, 7\}$ (subject to rotation and reflection), where $w_1 = 4/16$ and $w_2 = 2/16$ for even strength indexes, and $w_1 = w_2 = 3/16$ for odd strength indexes.*

where x_p is the value of pixel p, $P_{d,k}$ are the pixels in line k following direction d, and $\mu_{d,k}$ is the mean value of $P_{d,k}$:

$$\mu_{d,k} = \frac{1}{|P_{d,k}|} \sum_{p \in P_{d,k}} x_p. \tag{35}$$

A primary filter is applied along the selected direction, while a secondary filter is applied along the direction oriented 45° off the primary direction. The filter operation for pixel $p(x, y)$ is formulated by

$$\begin{aligned}\hat{p}(x,y) = p(x,y) &+ \sum_{m,n} w^p_{d,m,n} f(p(m,n) - p(x,y), S^p, D) \\ &+ \sum_{m,n} w^s_{d,m,n} f(p(m,n) - p(x,y), S^s, D)\end{aligned}$$

where $w^p_{d,m,n}$ and $w^s_{d,m,n}$ are the filter coefficients associated with the primary and secondary filters, respectively, as shown in Figs. 35 and 36. S_p and S_s are the strength indexes for the primary and secondary filters, and D is the damping factor. The $f()$ is a piecewise linear function

$$f(\text{diff}, S, D) = \begin{cases} \min\left(\text{diff}, \max\left(0, S - \lfloor \frac{\text{diff}}{2^{D - \lfloor \log_2 S \rfloor}} \rfloor\right)\right), & \text{if diff} > 0 \\ \max\left(\text{diff}, \min\left(0, \lceil \frac{\text{diff}}{2^{D - \lceil \log_2 S \rceil}} \rceil\right)\right) - S, & \text{otherwise} \end{cases}$$

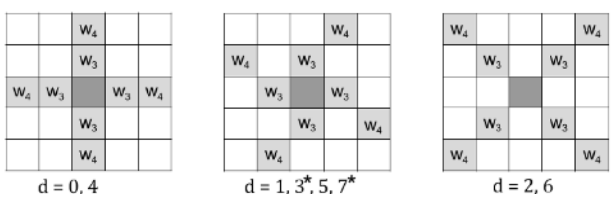


Fig. 36. *Secondary filter templates associated with each direction (subject to reflection), where $w_3 = 2/16$ and $w_4 = 1/16$. The secondary filter is applied along the direction 45° off the corresponding primary direction d.*

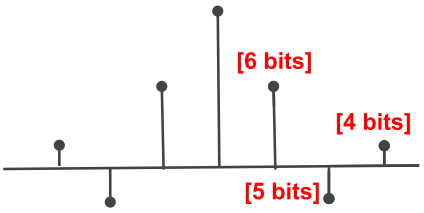


Fig. 37. *Bit precision for Wiener filter parameters.*

that rules out reference pixels whose values are far away from $p(x, y)$. Note that the reference pixels $p(m, n)$ are the reconstructed pixels after the deblocking filter is applied, but before application of the CDEF filter.

Up to eight groups of filter parameters, which includes the primary and secondary filter strength indexes of luma and chroma components, are signaled in the frame header. Each 64×64 block selects one group from the presets to control its filter operations.

C. Loop Restoration Filter

The loop restoration filter is applied to units of either 64×64, 128×128, or 256×256 pixel blocks, named loop restoration units (LRUs). Each unit can independently select either to bypass filtering, to use a Wiener filter, or to use a self-guided filter [44]. It is applied to the reconstructed pixels after any prior in-loop filtering stages.

1) Wiener Filter: A 7×7 separable Wiener filter is applied through the LRU. The filter parameters for the vertical and horizontal filters are decided by the encoder and signaled in the bitstream. Due to symmetric and normalization constraints, only three coefficients need to be sent for each filter. Also, note that the Wiener filters are expected to have a higher weight magnitude toward the origin, so the codec reduces the number of bits spent on higher tap coefficients, as shown in Fig. 37.

2) Self-Guided Filter: The scheme applies simple filters to the reconstructed pixels, X, to generate two denoised versions, X_1 and X_2, which largely preserves the edge transition. Their differences from the reconstructed pixels, $(X_1 - X)$ and $(X_2 - X)$, are used to span a subspace, upon which we project the differences between the reconstructed pixels and the original pixels, $(X_s - X)$, as shown in Fig. 38. The least-squares regression parameters obtained by the encoder are signaled to the decoder, which are used to build a linear approximation of $(X_s - X)$ based on the known bases $(X_1 - X)$ and $(X_2 - X)$.

In particular, a radius r and a noise variance e are used to generate the denoised versions of the LRU as follows.

1) Obtain the mean μ and variance σ^2 of pixels in a $(2r + 1) \times (2r + 1)$ window around every pixel x.
2) Compute the denoised pixel as

$$\hat{x} = \frac{\sigma^2}{\sigma^2 + e} x + \frac{e}{\sigma^2 + e} \mu. \tag{36}$$

The pair (r, e) effectively controls the denoising filter strength. Two sets of denoised pixels, denoted in the vector form X_1 and X_2, are generated using (r_1, e_1) and (r_2, e_2), which are selected by the encoder and are signaled in the bitstream. Let X denote the vector formed by the reconstructed pixels and X_s the vector of source pixels. The self-guided filter is formulated by

$$X_r = X + \alpha(X_1 - X) + \beta(X_2 - X). \tag{37}$$

The parameters (α, β) are obtained by the encoder using least-squares regression

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = (A^T A)^{-1} A^T b \tag{38}$$

where

$$A = \begin{bmatrix} X_1 - X \\ X_2 - X \end{bmatrix} \text{ and } b = X_s - X.$$

The parameters (α, β) are sent to the decoder to formulate (37).

D. Frame Super-Resolution

When the source input is downscaled from the original video signal, a frame super-resolution is natively supported as part of the postprocessing filtering that converts the reconstructed frame to the original dimension. As shown in Fig. 39, the frame super-resolution consists of an upsampling stage and a loop restoration filter [45].

The upsampling stage is applied to the reconstructed pixels after the CDEF filter. As mentioned in Section III-C, the downsampling and upsampling operations only apply to the horizontal direction. The upsampling process for a row of pixels in a frame is shown in Fig. 40. Let B denote the analog frame width. The downsampled frame contains D pixels in a row, and the upscaled frame contains W pixels in a row. Their sampling positions are denoted by P_k and Q_m, respectively, where $k \in \{0, 1, \ldots, D - 1\}$ and $m \in \{0, 1, \ldots, W - 1\}$. Note that P_0 and Q_0 are located at $(B/2D)$ and $(B/2W)$, respectively. After normalizing the

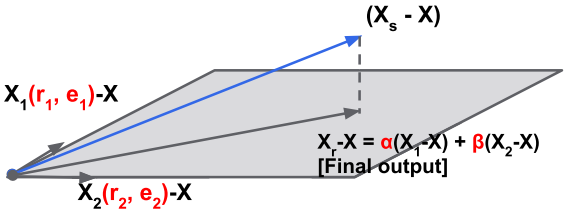


Fig. 38. *Project the gap between the source pixels X_s and reconstructed pixels X on to a subspace spanned by simple denoising results, $X_1 - X$ and $X_2 - X$. The parameters in red are the ones configurable through bitstream syntax.*

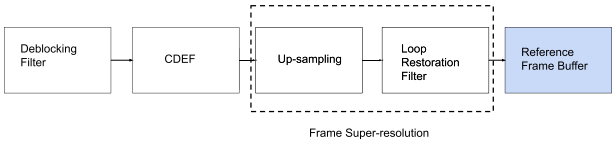


Fig. 39. *Frame super-resolution upsamples the reconstructed frame to the original dimension. It comprises a linear upsampling and a loop restoration filter.*

relative distance by (B/D), which corresponds to one full-pixel offset in the downsampled frame, it is straightforward to show that the offset of Q_m from P_0 is

$$Q_m - P_0 = \frac{D-W}{2W} + m\Delta_Q \tag{39}$$

where $\Delta_Q = (D/W)$.

In practice, these offsets are calculated at (1/16 384) pixel precision. They are rounded to the nearest (1/16)-pixel position for interpolation filter. An eight-tap FIR filter is used to generate the subpixel interpolation. Note that the rounding error

$$e = \text{round}(\Delta_Q) - \Delta_Q \tag{40}$$

is built up in the offset for Q_m, i.e., $((D-W)/2W) + m(\Delta_Q + e)$, as m increases from 0 to $W-1$. Here, the function round() maps a variable to the nearest sample in (1/16 384) resolution. This would make the leftmost pixel in a row have minimum rounding error in the offset calculation, whereas the right-most pixel has the maximum rounding error. To resolve such spatial bias, the initial offset for Q_0 is further adjusted by $-(eW/2)$, which makes the left- and right-most pixels have equal magnitude of rounding error and the middle pixel $Q_{W/2}$ close to zero rounding error. In summary, the adjusted offset of Q_m from P_0 is

$$Q_m \text{ offset} = \frac{D-W}{2W} - \frac{eW}{2} + m \text{ round}(\Delta_Q)c. \tag{41}$$

The loop restoration filter in Section VII-C is then applied to the upsampled frame to further recover the high-frequency components. It is experimentally shown in [45] that the loop restoration filter whose parameters are optimized by the encoder can substantially improve the objective quality of the upsampling frame.

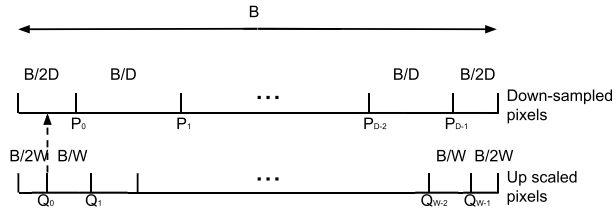


Fig. 40. *Frame super-resolution sampling positions. The analog frame width is denoted by B. The downsampled frame contains D pixels in a row, which are used to interpolate W pixels for a row in the upscaled frame.*

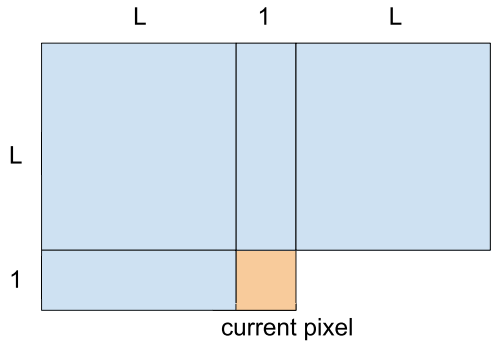


Fig. 41. *Reference region (in blue) is used by the AR model to generate the grain at a current sample (in orange). The reference region includes a $(2L+1)\times L$ block above and an $L\times 1$ block to the left. The total number of reference samples is $2L(L+1)$.*

Table 2 Capability Comparisons of AV1 Profiles

Profile	Bit-depth			Chroma sampling			
	8	10	12	4:0:0	4:2:0	4:2:2	4:4:4
Main	✓	✓		✓	✓		
High	✓	✓		✓	✓		✓
Professional	✓	✓	✓	✓	✓	✓	✓

E. Film Grain Synthesis

The film grain is widely present in creative content, such as movie and TV materials. Due to its random nature, the film grain is very difficult to compress using conventional coding tools that exploit signal correlations. AV1 provides a film grain synthesis option that builds a synthetic grain and adds it to the decoded picture prior to its display. This allows one to remove the film grain from the source video signal prior to compression. A set of model parameters is sent to the decoder to create a synthetic grain that mimics the original film grain.

AV1 adopts an AR model to build the grain signal [46]. The grain samples are generated in raster scan order. A grain sample in the luma plane is generated using a $(2L+1)\times L$ block above and an $L\times 1$ block to the left, as shown in Fig. 41, which involves $2L(L+1)$ reference samples, where $L\in\{0,1,2,3\}$. The AR model is given by

$$G(x,y) = \sum_{m,n\in S_{\text{ref}}} a_{m,n}G(x-m,y-n) + z \tag{42}$$

where S_{ref} is the reference region and z is a pseudorandom variable that is drawn from a zero-mean unit-variance Gaussian distribution. The grain samples for chroma components are generated similar to (42) with one additional input from the collocated grain sample in the luma plane.

Table 3 Compression Performance Comparison—Mid-Resolution. The Average Ratios Between AV1 and VP9 Instruction Counts for Encoding and Decoding Are Shown in Column "Enc Ins. Count" and "Dec Ins. Count," Respectively

Clip	Overall PSNR	SSIM	Enc Ins. Count	Dec Ins. Count
BQMall_832x480	-34.	-36.7	32.2	2.8
BalloonFestival_854x480	-35.3	-39.4	68.9	2.5
BasketballText_832x480	-35.8	-38.0	30.2	3.0
Basketballl_832x480	-36.1	-40.1	31.4	2.9
Campfire_854x480	-30.0	-38.3	81.9	3.0
CatRobot_854x480	-37.7	-38.0	31.9	2.8
DaylightRoad2_854x480	-29.4	-28.9	36.2	3.0
Drums_854x480	-36.8	-38.6	28.5	2.5
Flowervase_832x480	-35.2	-37.0	38.2	3.2
Keiba_832x480	-30.2	-32.7	27.2	2.9
Market3Clipr2_854x480	-36.8	-38.1	32.4	2.8
Mobisode2_832x480	-39.8	-39.4	73.3	3.0
NetflixNarrator_850x480	-35.3	-38.1	31.4	2.4
PartyScene_832x480	-33.4	-37.3	30.6	2.6
RaceHorses_832x480	-28.7	-32.2	29.9	2.9
ShowGirl2_854x480	-27.7	-30.1	33.1	2.8
Tango_854x480	-32.0	-32.1	30.9	3.1
ToddlerFountain854x480	-17.8	-21.0	41.0	3.0
TrafficFlow_854x480	-38.6	-39.8	29.2	2.6
aspen_480p	-31.0	-31.3	29.7	2.9
blue_sky_480	-39.8	-43.7	27.1	3.5
city_4cif	-33.4	-33.8	29.2	2.8
controlled_burn_480p	-36.7	-35.4	28.5	2.3
crew_4cif	-24.7	-25.2	35.3	3.5
crowd_run_480p	-24.9	-30.9	30.5	2.6
ducks_take_off_480p	-36.3	-38.7	36.4	2.6
harbour_4cif	-29.0	-32.1	33.5	2.8
ice_4cif	-27.0	-31.8	30.5	3.2
into_tree_480p	-35.5	-33.6	47.9	2.7
netflix_aerial	-47.3	-47.5	29.8	2.7
netflix_barscene	-37.6	-37.2	30.7	2.4
netflix_driving	-32.9	-33.0	34.6	2.6
netflix_foodmarket	-29.2	-29.6	33.4	2.8
netflix_ritualdance	-27.0	-32.5	49.3	3.1
netflix_rollercoaster	-33.7	-34.1	31.8	3.4
netflix_square	-32.7	-32.3	30.5	2.5
netflix_tunnelflag	-37.2	-45.6	22.7	2.7
old_town_480p	-34.7	-34.9	41.8	2.4
park_joy_480p	-27.1	-30.0	31.2	2.5
red_kayak_480p	-14.0	-14.4	36.6	3.4
rush_field_480p	-30.1	-28.9	34.2	3.0
shields_640x360	-37.2	-37.2	30.3	2.7
sintel_shot_854x480	-30.6	-33.9	37.8	3.6
snow_mnt_480p	-33.9	-30.2	60.1	2.3
soccer_4cif	-34.7	-40.7	27.6	2.8
speed_bag_480p	-44.7	-49.0	28.6	3.7
station2_480p	-57.6	-56.7	32.4	2.9
tears_of_steel_480p	-27.5	-30.5	34.0	3.2
touchdown_pass_480p	-25.3	-30.2	34.9	3.0
west_wind_480p	-23.6	-20.8	34.2	3.3
OVERALL	-33.0	-34.8	34.6	2.9

Table 4 Compression Performance Comparison—High Resolution. The Average Ratios Between AV1 and VP9 Instruction Counts for Encoding and Decoding Are Shown in Column "Enc Ins. Count" and "Dec Ins. Count," Respectively

Clip	Overall PSNR	SSIM	Enc Ins. Count	Dec Ins. Count
BalloonFestival_1280x720	-36.2	-43.0	88.2	2.5
CSGO_1080p	-29.2	-30.7	36.9	3.9
Campfire_1280x720	-31.8	-40.7	102.7	3.2
CatRobot_1280x720	-39.5	-39.4	37.3	3.0
DaylightRoad2_1280x720	-30.1	-29.8	42.5	3.1
Drums_1280x720	-37.0	-38.9	34.3	2.6
Market3Clipr2_1280x720	-38.4	-40.6	37.6	2.6
Netflix_Aerial_2048x1080	-31.4	-33.0	36.8	2.8
Netflix_Boat_2048x1080	-37.6	-38.6	33.2	2.4
Netflix_Crosswalk_2048x1080	-33.2	-35.3	37.7	2.5
Netflix_Dancers_1280x720	-38.2	-41.8	44.6	3.2
Netflix_DrivingPOV_2048x1080	-29.8	-30.1	43.2	2.9
Netflix_FoodMarket2_1280x720	-33.4	-35.0	35.8	2.5
Netflix_FoodMarket_2048x1080	-30.3	-33.0	43.2	3.1
Netflix_PierSeaside_2048x1080	-46.4	-42.1	38.3	2.9
Netflix_Square_2048x1080	-34.0	-34.8	29.6	2.8
Netflix_TunnelFlag_2048x1080	-39.3	-42.4	25.5	3.0
RollerCoaster_1280x720	-32.4	-34.9	35.8	3.2
ShowGirl2_1280x720	-28.6	-32.1	41.9	3.0
Tango_1280x720	-32.7	-32.7	36.8	3.3
ToddlerFountain1280x720	-17.9	-20.1	47.5	3.2
TrafficFlow_1280x720	-39.7	-40.5	36.7	2.7
aspen_1080p	-41.5	-45.0	32.9	3.3
basketballdrive_1080p	-32.5	-36.2	41.4	3.4
cactus_1080p	-38.2	-38.3	43.8	3.0
city_720p	-38.0	-37.6	38.4	2.6
controlled_burn_1080p	-39.3	-42.6	41.2	2.7
crew_720p	-25.4	-28.2	44.3	3.7
crowd_run_1080p	-27.0	-30.8	31.4	2.8
dinner_1080p30	-37.5	-40.2	34.0	3.4
ducks_take_off_1080p	-33.1	-35.2	40.8	2.8
factory_1080p	-33.6	-40.6	29.6	3.2
in_to_tree_1080p	-26.7	-22.8	56.2	2.9
johnny_720p	-42.9	-43.8	43.9	2.4
kristenandsara_720p	-40.6	-38.4	44.0	2.8
night_720p	-34.8	-36.6	41.1	3.0
old_town_cross_720p	-38.4	-37.6	55.5	2.5
parkjoy_1080p	-27.2	-34.0	31.9	2.7
ped_1080p	-32.9	-38.2	35.8	3.3
red_kayak_1080p	-16.1	-14.7	37.5	3.5
riverbed_1080p	-17.4	-16.9	35.5	3.6
rush_field_1080p	-27.2	-30.0	34.5	3.2
rush_hour_1080p	-26.2	-34.5	47.9	3.7
shields_720p	-45.5	-44.5	44.8	2.8
station2_1080p	-56.4	-56.4	45.6	3.9
sunflower_720p	-45.5	-49.6	26.6	3.2
tennis_1080p	-33.4	-37.9	32.8	3.3
touchdown_1080p	-29.6	-35.8	37.8	3.4
tractor_1080p	-34.9	-39.6	31.6	2.9
vidyo4_720p	-38.8	-42.1	41.7	2.9
OVERALL	-34.2	-36.4	39.6	3.1

The model parameters associated with each plane are transmitted through the bitstream to formulate the desired grain patterns.

The AR process is used to generate a template of grain samples corresponding to a 64×64 pixel block. Patches whose dimensions correspond to a 32×32 pixel block are drawn at pseudorandom positions within this template and are applied to the reconstructed video signal.

The final luma pixel at position (x, y) is given by

$$\hat{P}(x, y) = P(x, y) + f(P(x, y))G(x, y) \tag{43}$$

Table 5 Intraframe Compression Performance Comparison—Mid-Resolution. The Average Ratios Between AV1 and VP9 Instruction Counts for Encoding and Decoding Are Shown in Column "Enc Ins. Count" and "Dec Ins. Count," Respectively

Clip	Overall PSNR	SSIM	Enc Ins. Count	Dec Ins. Count
BQMall_832x480	-21.5	-21.6	9.6	1.9
BalloonFestival_854x480	-21.5	-23.4	14.1	2.0
BasketballText_832x480	-30.5	-26.9	9.6	2.0
Basketball_832x480	-31.6	-28.1	9.7	1.8
Campfire_854x480	-31.6	-28.1	12.6	1.9
CatRobot_854x480	-23.5	-23.6	10.3	1.9
DaylightRoad2_854x480	-23.2	-23.2	10.1	1.9
Drums_854x480	-22.7	-21.6	10.5	1.9
Flowervase_832x480	-22.1	-22.8	9.0	2.3
Keiba_832x480	-24.2	-21.5	8.9	2.0
Market3r2_854x480	-21.9	-20.8	10.2	1.8
Mobisode2_832x480	-32.8	-32.4	9.3	2.5
NetflixNarrator_850x480	-21.9	-23.0	9.9	2.1
PartyScene_832x480	-15.5	-14.7	11.9	1.7
RaceHorses_832x480	-16.5	-13.5	13.4	1.7
ShowGirl2_854x480	-20.2	-20.7	10.0	2.0
Tango_854x480	-24.5	-24.2	10.4	2.0
ToddlerFountain854x480	-16.6	-12.8	11.5	1.7
TrafficFlow_854x480	-31.4	-32.0	11.7	1.9
aspen_480p	-19.5	-18.9	12.6	2.0
blue_sky_480p	-19.5	-21.3	14.0	2.0
city_4cif	-19.8	-18.1	11.1	1.9
controlled_burn_480p	-15.1	-12.5	12.5	1.8
crew_4cif	-22.9	-21.3	9.9	1.9
crowd_run_480p	-13.3	-9.9	13.6	1.7
ducks_480p	-23.9	-27.4	12.9	1.6
harbour_4cif	-19.6	-17.8	11.4	2.1
ice_4cif	-27.2	-26.4	8.5	2.4
into_tree_480p	-24.1	-19.7	9.3	1.7
netflix_aerial	-15.5	-11.5	12.3	1.6
netflix_barscene	-22.3	-21.0	9.0	2.1
netflix_driving	-20.0	-20.3	10.5	1.8
netflix_foodmarket	-20.2	-17.1	10.5	1.9
netflix_ritualdance	-22.5	-20.2	11.6	1.9
netflix_rollercoaster	-24.2	-24.8	9.7	2.0
netflix_square	-16.0	-15.0	12.8	1.8
netflix_tunnelflag	-31.1	-39.1	10.9	2.0
old_town_480p	-19.1	-18.5	10.7	1.7
park_joy_480p	-15.4	-13.6	13.9	1.6
red_kayak_480p	-16.6	-15.5	11.4	2.0
rush_field_480p	-15.4	-14.1	12.8	1.8
shields_640x360	-19.9	-21.5	14.1	1.7
sintel_shot_854x480	-29.9	-31.2	10.8	3.1
snow_mnt_480p	-12.4	-8.9	18.9	2.0
soccer_4cif	-21.5	-17.9	10.5	1.9
speed_bag_480p	-28.2	-33.8	9.2	2.9
station2_480p	-24.3	-21.7	10.2	1.8
tears_of_steel_480p	-25.6	-24.4	10.4	1.9
touchdown_pass_480p	-18.8	-13.7	10.2	2.2
west_wind_480p	-20.2	-16.8	11.8	2.1
OVERALL	-22.0	-21.0	11.1	2.0

Table 6 Intraframe Compression Performance Comparison—HD Resolution. The Average Ratios Between AV1 and VP9 Instruction Counts for Encoding and Decoding Are Shown in Column "Enc Ins. Count" and "Dec Ins. Count," Respectively

Clip	Overall PSNR	SSIM	Enc Ins. Count	Dec Ins. Count
BalloonFestival_1280x720	-23.4	-25.3	12.2	2.1
CSGO_1080p	-23.4	-19.7	9.0	2.3
Campfire_1280x720	-32.3	-28.6	11.6	1.9
CatRobot_1280x720	-25.6	-26.0	8.9	2.0
DaylightRoad2_1280x720	-24.2	-24.1	8.5	1.9
Drums_1280x720	-23.4	-21.7	8.8	1.9
Market3r2_1280x720	-21.8	-19.5	8.8	1.8
Netflix_Aerial_2048x1080	-20.2	-17.5	11.1	2.0
Netflix_Boat_2048x1080	-19.6	-19.9	12.8	2.0
NetflixCrosswalk_2048x1080	-24.8	-25.7	9.0	2.6
Netflix_Dancers_1280x720	-31.4	-30.5	7.7	3.1
NetflixDrivingPOV2048x1080	-22.4	-20.4	9.1	2.0
NetflixMarket2_1280x720	-20.1	-18.9	10.7	1.8
NetflixMarket_2048x1080	-21.1	-17.9	9.5	2.0
NetflixPierSeaside2048x1080	-24.7	-21.8	9.1	2.0
Netflix_Square_2048x1080	-19.7	-18.5	10.5	2.0
NetflixTunnelFlag_2048x1080	-34.4	-35.8	8.7	2.2
RollerCoaster_1280x720	-24.9	-23.9	9.2	2.0
ShowGirl2_1280x720	-21.7	-22.1	8.6	2.2
Tango_1280x720	-26.0	-25.4	8.9	2.2
ToddlerFountain1280x720	-17.1	-13.2	9.9	1.8
TrafficFlow_1280x720	-32.6	-32.7	9.8	2.0
aspen_1080p	-22.4	-21.5	10.1	2.8
basketballdrive_1080p	-30.9	-27.8	7.6	1.9
cactus_1080p	-24.6	-22.0	8.9	1.7
city_720p	-19.7	-17.8	9.8	2.2
controlled_burn_1080p	-15.9	-14.0	12.2	2.2
crew_720p	-24.7	-22.9	8.3	2.2
crowd_run_1080p	-17.0	-12.9	11.1	1.7
dinner_1080p	-29.6	-28.9	8.7	2.4
ducks_1080p	-25.9	-27.1	9.6	1.6
factory_1080p	-23.8	-20.3	11.8	2.2
in_to_tree_1080p	-25.0	-18.3	8.8	1.6
johnny_720p	-28.5	-26.7	8.1	2.3
kristenandsara_720p	-26.6	-25.6	8.3	2.4
night_720p	-20.5	-19.3	9.6	2.0
old_town_720p	-20.2	-18.2	9.6	1.6
parkjoy_1080p	-17.7	-14.9	12.4	1.7
ped_1080p	-29.7	-29.2	7.5	2.1
red_kayak_1080p	-18.4	-16.4	11.4	2.4
riverbed_1080p	-18.1	-15.9	10.6	2.2
rush_field_1080p	-15.4	-14.3	10.7	2.2
rush_hour_1080p	-26.8	-29.7	8.3	2.5
shields_720p	-20.2	-19.5	10.3	1.7
station2_1080p	-24.8	-22.3	9.2	2.1
sunflower_720p	-29.2	-32.2	10.3	2.6
tennis_1080p	-27.5	-27.0	7.7	2.1
touchdown_1080p	-19.0	-15.7	9.2	2.5
tractor_1080p	-28.5	-28.9	9.8	2.0
vidyo4_720p	-25.1	-24.4	8.3	2.3
OVERALL	-23.8	-22.5	9.5	2.1

where $P(x,y)$ is the decoded pixel value and $f(P(x,y))$ scales the grain sample according to the collocated pixel intensity. The $f()$ is a piecewise linear function and is configured by the parameters sent through the bitstream.

The grain samples applied to the chroma components are scaled based on the chroma pixel value and the collocated luma pixel values. A chroma pixel is given by

$$\hat{P}_u(x,y) = P_u(x,y) + f(t)G_u(x,y) \tag{44}$$

$$t = b_u P_u(x,y) + d_u \bar{P}(x,y) + h_u \tag{45}$$

where $\bar{P}(x, y)$ denotes the average of the collocated luma pixels. The parameters b_u, d_u, and h_u are signaled in the bitstream for each chroma plane.

The film grain synthesis model parameters are decided on a frame by frame basis and are signaled in the frame header. AV1 also allows a frame to reuse the previous frame's model parameter set and bypass sending a new set in the frame header.

VIII. PROFILE AND LEVEL DEFINITION

AV1 defines profiles and levels to specify the decoder capability. Three profiles define support for various bit-depth and chroma sampling formats, namely *Main*, *High*, and *Professional*. The capability required for each profile is listed in Table 2.

Levels are defined to quantify the decoder performance in terms of maximum bit rate, maximum samples per picture, and other characteristics, as shown in [9]. A decoder that supports a given level should be capable of processing all bitstreams that conform to the specifications provided by the level definition. To account for various coding structure and rate allocation strategies that might be used to create a bitstream, a decoder model that describes the smoothing buffer, decoding process, frame buffering, and display process is provided to verify that a bitstream meets the level definitions [9], [47].

IX. PERFORMANCE EVALUATION

We compared the compression performance of libvpx VP9 [48] and libaom AV1 [11]. The source code of libvpx VP9 can be accessed at [48]. The experiment used the version with git hash ebac57ce. The source code of libaom AV1 can be found at [11]. The experiment used the version with git hash ac2c30ba.

Both codecs used the default two-pass encoding mode and variable bit-rate control and ran at the highest compression performance mode, i.e., cpu-used=0. To achieve the compression performance, both VP9 and AV1 encoder allowed adaptive GOP size, where the decisions were made based on the first pass encoding statistics. The QP offsets between different frames within a GOP were also adaptively optimized based on the first pass coding statistics. The test sets included video resolutions ranging from 480p to 1080p. All the clips were coded using their first 150 frames. The BD-rate reductions in overall PSNR and SSIM are shown in Tables 3 and 4.

To evaluate the relative encoding and decoding complexity, we gathered the instruction counts for both the encoding and decoding processes on a single thread at each operating point. The average ratios between AV1 and VP9 are shown in column "Enc Ins. Count" and "Dec Ins. Count" in Tables 3 and 4 to reflect the relative encoding and decoding complexity, respectively. The average AV1 encoding complexity is roughly 34.6–39.6 times the VP9 encoding complexity, both at their high compression performance. The average AV1 decoding complexity, on average, is about three times of VP9 decoding complexity. Note that we use the instruction count to evaluate the codec complexity since it closely tracks the actual runtime on the same computer and is largely invariant in a cloud computing environment.

We next evaluated the intracoding performance, where all the 150 frames were coded as intraframes with the same QP. The same QP set was used for both all intra and video coding modes. The compression efficiency in overall PSNR and SSIM BD-rate reductions are shown in Tables 5 and 6. Similarly, the relative encoding and decoding complexity between AV1 and VP9 in intracoding mode are provided in column "Enc Ins. Count" and "Dec Ins. Count" in Tables 5 and 6, respectively.

Note that the results are intended for reference on the scale of the relative coding gains and complexity. Different encoder implementations might have different compression and complexity tradeoff considerations and different performance results. An extensive codec performance evaluation under various encoder constraints is beyond the scope of this article. Readers are referred to [8] for more comparison results under encoder constraints. Also, note that a dedicated decoder implementation might be further optimized for decoding complexity reduction.

X. CONCLUSION

This article provides a technical overview of the AV1 codec. It outlines the design theories of the compression techniques and the considerations for hardware feasibility, which together define the AV1 codec. ■

Acknowledgment

The AV1 codec includes contributions from the entire AOMedia teams [6] and the greater ecosystem around the globe. An incomplete contributor list can be found at [10]. The authors sincerely appreciate the constructive feedback from the anonymous reviewers and the associated editors, which substantially improves the quality of this manuscript.

REFERENCES

[1] *Cisco Annual Internet Report (2018–2023) White Paper*. Accessed: 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] D. Mukherjee *et al.*, "A technical overview of VP9—The latest open-source video codec," *SMPTE Motion Imag. J.*, vol. 124, no. 1, pp. 44–54, 2015.

[3] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[4] D. Vatolin, D. Kulikov, M. Erofeev, A. Antsiferova, S. Zvezdakov, and D. Kondranin. (2018). *Thirteen MSU Video Codecs Comparison*. [Online]. Available: http://compression.ru/video/codec_comparison/hevc_2018/#hq_report

[5] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[6] *Alliance for Open Media*. Accessed: 2021. [Online]. Available: https://aomedia.org/

[7] Y. Chen *et al.*, "An overview of core coding tools in the AV1 video codec," in *Proc. Picture Coding Symp. (PCS)*, Jun. 2018, pp. 41–45.

[8] Y. Chen *et al.*, "An overview of coding tools in AV1: The first video codec from the alliance for open media," *APSIPA Trans. Signal Inf. Process.*, vol. 9,

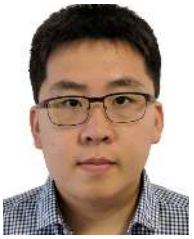
no. 6, pp. 1–15, 2020.
[9] P. de Rivaz and J. Haughton. *AV1 Bitstream & Decoding Process Specification*. Accessed: 2021. [Online]. Available: https://aomediacodec.github.io/av1-spec/av1-spec.pdf
[10] *Libaom Contributor List*. Accessed: 2021. [Online]. Available: https://aomedia.googlesource.com/aom/+/refs/heads/master/AUTHORS
[11] *Libaom AV1 Repository*. Accessed: 2021. [Online]. Available: https://aomedia.googlesource.com/aom/
[12] *Libaom Performance Tracker*. Accessed: 2021. [Online]. Available: https://datastudio.google.com/reporting/a84c7736-99c3-4ff5-a9df-92deae923294
[13] R. S. Overbeck, D. Erickson, D. Evangelakos, M. Pharr, and P. Debevec, "A system for acquiring, processing, and rendering panoramic light field stills for virtual reality," *ACM Trans. Graph.*, vol. 37, no. 6, pp. 1–15, Jan. 2019.
[14] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*, vol. 1, no. 10. New York, NY, USA: Springer, 2001.
[15] H. Schwarz, D. Marpe, and T. Wiegand, "Analysis of hierarchical B pictures and MCTF," in *Proc. IEEE Int. Conf. Multimedia Expo*, Jul. 2006, pp. 1929–1932.
[16] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Hoboken, NJ, USA: Wiley, 2012.
[17] C. Chen, J. Han, and Y. Xu, "A non-local mean temporal filter for video compression," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2020, pp. 1142–1146.
[18] C.-H. Chiang, J. Han, and Y. Xu, "A multi-pass coding mode search framework for AV1 encoder optimization," in *Proc. Data Compress. Conf. (DCC)*, Mar. 2019, pp. 458–467.
[19] L. Trudeau, N. Egge, and D. Barr, "Predicting chroma from Luma in AV1," in *Proc. Data Compress. Conf. (DCC)*, Mar. 2018, pp. 374–382.
[20] M. Jakubowski and G. Pastuszak, "Block-based motion estimation algorithms—A survey," *Opto-Electron. Rev.*, vol. 21, no. 1, pp. 86–102, 2013.
[21] I. Patras, E. A. Hendriks, and R. L. Lagendijk, "Probabilistic confidence measures for block matching motion estimation," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, no. 8, pp. 988–995, Aug. 2007.
[22] J. Vanne, E. Aho, T. D. Hamalainen, and K. Kuusilinna, "A high-performance sum of absolute difference implementation for motion estimation," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 7, pp. 876–883, Jul. 2006.
[23] H. Jozawa, K. Kamikura, A. Sagata, H. Kotera, and H. Watanabe, "Two-stage motion compensation using adaptive global MC and local affine MC," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 7, no. 1, pp. 75–85, Feb. 1997.
[24] H.-K. Cheung and W.-C. Siu, "Local affine motion prediction for H.264 without extra overhead," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2010, pp. 1555–1558.
[25] R. C. Kordasiewicz, M. D. Gallant, and S. Shirani, "Affine motion prediction based on translational motion vectors," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, no. 10, pp. 1388–1394, Oct. 2007.
[26] S. Parker, Y. Chen, D. Barker, P. de Rivaz, and D. Mukherjee, "Global and locally adaptive warped motion compensation in video compression," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2017, pp. 275–279.
[27] S.-W. Wu and A. Gersho, "Joint estimation of forward and backward motion vectors for interpolative prediction of video," *IEEE Trans. Image Process.*, vol. 3, no. 5, pp. 684–687, Sep. 1994.
[28] M. T. Orchard and G. J. Sullivan, "Overlapped block motion compensation: An estimation-theoretic approach," *IEEE Trans. Image Process.*, vol. 3, no. 5, pp. 693–699, Sep. 1994.
[29] Y. Chen and D. Mukherjee, "Variable block-size overlapped block motion compensation in the next generation open-source video codec," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2017, pp. 938–942.
[30] G. Laroche, J. Jung, and B. Pesquet-Popescu, "RD optimized coding for motion vector predictor selection," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 9, pp. 1247–1257, Sep. 2008.
[31] J.-L. Lin, Y.-W. Chen, Y.-W. Huang, and S.-M. Lei, "Motion vector coding in the HEVC standard," *IEEE J. Sel. Topics Signal Process.*, vol. 7, no. 6, pp. 957–968, Dec. 2013.
[32] A. M. Tourapis, F. Wu, and S. Li, "Direct mode coding for bipredictive slices in the H.264 standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 1, pp. 119–126, Jan. 2005.
[33] J. Han, J. Feng, Y. Teng, Y. Xu, and J. Bankoski, "A motion vector entropy coding scheme based on motion field referencing for video compression," in *Proc. 25th IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2018, pp. 3618–3622.
[34] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE Trans. Comput.*, vol. 100, no. 1, pp. 90–93, Jan. 1974.
[35] J. Han, A. Saxena, V. Melkote, and K. Rose, "Jointly optimized spatial prediction and block transform for video and image coding," *IEEE Trans. Image Process.*, vol. 21, no. 4, pp. 1874–1884, Apr. 2012.
[36] F. Kamisli and J. S. Lim, "1-D transforms for the motion compensation residual," *IEEE Trans. Image Process.*, vol. 20, no. 4, pp. 1036–1046, Apr. 2011.
[37] W.-H. Chen, C. Smith, and S. Fralick, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Commun.*, vol. 25, no. 9, pp. 1004–1009, Sep. 1977.
[38] J. Han, Y. Xu, and D. Mukherjee, "A butterfly structured design of the hybrid transform coding scheme," in *Proc. Picture Coding Symp. (PCS)*, Dec. 2013, pp. 17–20.
[39] J.-M. Valin *et al.*, "Daala: Building a next-generation video codec from unconventional technology," in *Proc. IEEE 18th Int. Workshop Multimedia Signal Process. (MMSP)*, Sep. 2016, pp. 1–6.
[40] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, 1987.
[41] J. Han, C.-H. Chiang, and Y. Xu, "A level-map approach to transform coefficient coding," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2017, pp. 3245–3249.
[42] S. W. Golomb, "Run-length encodings," *IEEE Trans. Inf. Theory*, vol. 12, no. 3, pp. 399–401, Sep. 1966.
[43] S. Midtskogen and J.-M. Valin, "The Av1 constrained directional enhancement filter (Cdef)," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2018, pp. 1193–1197.
[44] D. Mukherjee, S. Li, Y. Chen, A. Anis, S. Parker, and J. Bankoski, "A switchable loop-restoration with side-information framework for the emerging AV1 video codec," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2017, pp. 265–269.
[45] U. Joshi, D. Mukherjee, Y. Chen, S. Parker, and A. Grange, "In-loop frame super-resolution in AV1," in *Proc. Picture Coding Symp. (PCS)*, Nov. 2019, pp. 1–5.
[46] A. Norkin and N. Birkbeck, "Film grain synthesis for AV1 video codec," in *Proc. Data Compress. Conf. (DCC)*, Mar. 2018, pp. 3–12.
[47] A. Norkin. (2020). *AV1 Decoder Model*. [Online]. Available: https://norkin.org/research/av1_decoder_model/index.html
[48] *Libvpx VP9 Repository*. Accessed: 2021. [Online]. Available: https://chromium.googlesource.com/webm/libvpx

ABOUT THE AUTHORS

Jingning Han (Senior Member, IEEE) received the B.S. degree in electrical engineering from Tsinghua University, Beijing, China, in 2007, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 2008 and 2012, respectively.

He joined the WebM Codec Team, Google, Mountain View, CA, USA, in 2012, where he is the Main Architect of the VP9 and AV1 codecs. He has published more than 60 research articles. He holds more than 50 U.S. patents in the field of video coding. His research interests include video coding and computer architecture.

Dr. Han received the Dissertation Fellowship in 2012 from the Department of Electrical and Engineering, University of California at Santa Barbara. He was a recipient of the Best Student Paper Award at the IEEE International Conference on Multimedia and Expo in 2012. He also received the IEEE Signal Processing Society Best Young Author Paper Award in 2015.



Bohan Li (Member, IEEE) received the B.S. degree in electronics engineering from Tsinghua University, Beijing, China, in 2014, and the M.S. and Ph.D. degrees in electrical engineering from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 2016 and 2019, respectively.

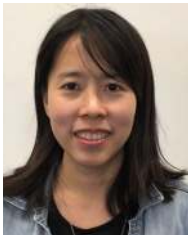
He is currently a Software Engineer with the WebM Team, Google, Mountain View, CA, USA. His main focus is on signal compression and video coding techniques.

Dr. Li is a member of the IEEE Signal Processing Society. He was a recipient of the Best Paper Award at the IEEE International Conference on Image Processing in 2017.

Debargha Mukherjee (Senior Member, IEEE) received the M.S./Ph.D. degree in electrical and computer engineering (ECE) from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 1999.

In 2009, he was with Hewlett Packard Labs, Palo Alto, CA, USA, conducting research on video/image coding and processing. Since 2010, he has been with Google Inc., Mountain View, CA, USA, where he is currently a Principal Engineer leading video codec research and development efforts. Prior to that, he was responsible for video quality control and 2-D–3-D conversion on YouTube. He has authored/coauthored more than 100 articles on various signal processing topics. He holds more than 90 U.S. patents, with many more pending. He has been delivering many workshops and talks on Google's royalty-free line of codecs since 2012, and more recently on the AV1 video codec from the Alliance for Open Media (AOM).

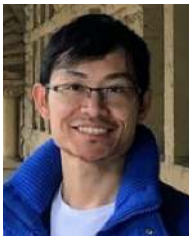
Dr. Mukherjee is a member of the IEEE Image, Video, and Multidimensional Signal Processing Technical Committee (IVMSP TC). He has served as an Associate Editor of IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY and IEEE TRANSACTIONS ON IMAGE PROCESSING.



Ching-Han Chiang received the B.S. degree in electrical engineering from National Sun Yat-sen University, Kaohsiung, Taiwan, in 2007, the M.S. degree in electrical engineering from National Tsing Hua University, Hsinchu, Taiwan, in 2010, and the M.S. degree in computer science from New York University, New York, NY, USA, in 2015.

She was a Software Engineer with MStar Semiconductor, Hsinchu. She was a Research Assistant with Academia Sinica, Taipei, Taiwan, conducting research on computer vision. She is currently a Software Engineer with Google, Mountain View, CA, USA, where she has been working on VP9 and AV1 for more than five years. Her research interests include video compression and computer vision.

Adrian Grange is currently a member of the Chrome Media Team, Google, Mountain View, CA, USA, where he was an active technical contributor during the development of the VPx range of video codecs. After helping to found the Alliance for Open Media, he chaired the Codec Working Group through the AV1 development and also co-chairs the work being undertaken to design new coding tools for a next-generation codec. He also runs the Google Chrome University Research Program.



Cheng Chen graduated from Tsinghua University, Beijing, China, in 2011. He received the Ph.D. degree from The University of Iowa, Iowa City, IA, USA, in 2016.

He is currently a Senior Software Engineer with the Core Video Compression Team, Google, Mountain View, CA, USA. His work involves research and development of open-source video coding standards, VP9 and AV1, and the encoder and decoder optimizations.

Hui Su received the B.S. degree in electrical engineering from the University of Science and Technology of China, Hefei, China, in 2009, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Maryland, College Park, MD, USA, in 2012 and 2014, respectively.

He is currently a Staff Software Engineer with the Video Compression Team, Google, Mountain View, CA, USA. He has been working on open video codecs, such as VP9 and AV1, for more than six years. His research interests include image/video compression, signal processing, and machine learning.

Sarah Parker received the B.Sc. degree in neuroscience from Brown University, Providence, RI, USA, in 2015.

She has been an Engineer with Google, Mountain View, CA, USA, working on the video, since 2015. She has contributed to the AV1 development and release and continues to work on new coding experiments to produce AV1's successor.

Sai Deng received the master's degree from The University of Utah, Salt Lake City, UT, USA, and Peking University, Beijing, China, in 2015.

He is currently a Software Engineer with the WebM Codec Team, Google, Mountain View, CA, USA. He works on VP9/AV1 video codec development and participated in multiple augmented reality and virtual reality projects, including light fields' video encoding and decoding.

Urvang Joshi received the M.E. degree in computer engineering from the Indian Institute of Science, Bengaluru, India, in 2009.

He worked on the Bing Relevance Team, Microsoft, Hyderabad, and face/object detection problems at Yahoo Labs, Bengaluru. He is currently a Senior Software Engineer with Google, Mountain View, CA, USA, where has been contributing new compression techniques for open-source video codec AV1. He also worked on developing the open-source image format WebP. His research interests include video compression and machine learning.

Yue Chen received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2011, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 2013 and 2016, respectively.

She is currently with the Video Compression Team, Google, Mountain View, CA, USA. She is also an active contributor to open-source video coding technology, including AV1 and VP9. She holds many patents in the field of video compression. Her research interests include video/image compression and processing.

Yunqing Wang received the B.S. and Ph.D. degrees in precision instrument from Tsinghua University, Beijing, China, in 1991 and 1996, respectively, and the M.S. degree in computer science from the University of Massachusetts at Amherst, Amherst, MA, USA, in 2002.

She is currently a Senior Staff Software Engineer with Google, Mountain View, CA, USA, where she has been working on AV1 and VPx series video codec development since 2007. Her research interests include video compression and optimization.

Paul Wilkins graduated from the University of Cambridge, Cambridge, U.K., in 1986.

He has been working in the field of video compression since the early 1990s. His company and video I.P. were acquired by On2 Technologies Inc., New York, NY, USA, in 1999, where he served as the S.V.P. of Research and Development and later as co-CTO. Since 2010, he has been a Technical Lead with the Codec Development Team, Google.

Yaowu Xu (Senior Member, IEEE) received the Ph.D. degree in nuclear engineering from Tsinghua University, Beijing, China, in 1997, and the Ph.D. degree in electrical and computer engineering from the University of Rochester, Rochester, NY, USA, in 2003.

He is currently a Principal Software Engineer with the Leading Google's Media Compression Teams, Google, Mountain View, CA, USA. His group is responsible for developing the core technology that enables the delivery of video, image, audio, and 3-D reality over the Internet, powering a broad range of products and services at Google, such as YouTube, Meet/Duo, Google TV, Stadia, Photos, Image Search, Ads, and AR Shopping. World-class experts in his group have been the driving force of many open-source media compression projects, such as VP9, AV1, WebP, AVIF, and Draco. Besides leading engineering teams and projects, he is the Executive Sponsor of Chrome and Photos' mentoring program, a Faculty Member of Google's Manager Development Team that trains Google's people managers, and a Volunteer Mentor to Google's top talents at various career stages. He has been granted more than 180 patents.

James Bankoski (Member, IEEE) is currently a Distinguished Engineer with Google, Mountain View, CA, USA, leading the team responsible for Google's video, audio, and image compression efforts. The team was founded when On2 Technologies Inc., New York, NY, USA, was acquired back in 2011. He was the CTO of On2 Technologies Inc. He has contributed to the design of all of On2/Google's video codecs from Tm2x through VP9, including video codecs widely used in Flash, Skype, and now WebM. His team also works on hardware implementations, and VP9 is now adopted as a hardware component by most of the major TV manufacturers. He is currently leading Google's Alliance for Open Media Codec Efforts at Google, the major new open-source codec under development. The AOM organization now comprises more than 40 companies.