AVX2-Optimized Interpolation Filters for HEVC Inter Encoding

Alexandre Mercat, Ari Lemmetti, Joose Sainio, and Jarno Vanne
Ultra Video Group, *Tampere University*
Tampere, Finland

Abstract—**High Efficiency Video Coding (HEVC) sets the stage for economic video transmission and storage, but its inherent computational complexity calls for powerful implementations. This paper addresses the principal performance bottleneck of HEVC codecs by introducing AVX2-vectorized algorithms for HEVC interpolation filters. The proposed speed-up techniques include 1) a data permutation scheme for the horizontal interpolation stage; 2) a sliding window strategy for the vertical interpolation stage; 3) optimal usage of horizontal and vertical interpolation during fractional motion estimation; and 4) a lane-based approach to double the vector lengths from 128-bit legacy vector extensions to 256bits of AVX2. Our AVX2-optimized interpolation filters were benchmarked as part of the practical Kvazaar open-source HEVC encoder. On an Intel 8-core Xeon processor, they were shown to be 9.7 and 8.5 times as fast as scalar interpolation with the Kvazaar ultrafast and veryslow presets, respectively. In both cases, changing over from scalar to vectorized interpolation increases the coding speed of Kvazaar by more than 50%, which stresses the importance of interpolation optimizations in modern video encoders.**

Keywords— High Efficiency Video Coding (HEVC), single instruction multiple data (SIMD), Advanced Vector Extensions 2 (AVX2), interpolation filter, Kvazaar HEVC encoder

I. Introduction

A plurality of media applications has made digital video ubiquitous in our multimedia-driven society. The skyrocketing video transmission and storage needs of these applications is being met by a series of international video coding standards, of which *Advanced Video Coding* (*AVC/H.264*) [1], *High Efficiency Video Coding* (*HEVC/H.265*) [2], and *Versatile Video Coding* (*VVC/H.266*) [3] currently dominate the landscape.

This work focuses on HEVC that is one of the most widespread video formats nowadays [4]. HEVC is able to double the coding efficiency over that of AVC for the same objective visual quality, but its computational complexity [5] tends to act as a barrier for developing practical applications. Therefore, implementing high-speed video encoders calls for high-performance computing platforms and designing them in compliance with green computing also requires efficient speed-up techniques that tend to provide power savings [6].

HEVC coding gains stem mainly from the new block partitioning structure and improved *motion compensated prediction* (*MCP*). In MCP, interpolation filters are used to obtain samples between integer pixels for blocks with fractional pixel motion, which is why many encoders also use them for *fractional motion estimation* (*FME*). The accuracy of the HEVC interpolation filters was improved over that of AVC [7], but at the cost of higher complexity. For example, the interpolation was reported to account for 38% of the total encoding complexity of *HEVC test model* (*HM*) [8].

This work was supported in part by the AI for situational Awareness (AISA) project led by Nokia and funded by Business Finland, and the Academy of Finland (decision no. 349216).

In software implementations, the HEVC interpolation filters can be accelerated through multithreading and vectorization. A couple of *single instruction multiple data* (*SIMD*) optimizations [9]–[11] have been proposed in the literature. However, they were limited to older vector extensions, the decoder side of the codec, or HM that is not designed for practical coding.

This work uses *Advanced Vector Extensions 2* (*AVX2*) to accelerate the *interpolation filters* and FME of the practical open-source HEVC encoder called *Kvazaar* [12], [13]. The widespread *Intel Broadwell* microarchitecture was selected as the primary target hardware. To the best of our knowledge, the proposed solutions are the sole AVX2-optimized HEVC interpolation filters that are distributed under the permissive 3-clause BSD license [13]. Please refer to [14] for the implementation details.

The remainder of the paper is outlined as follows. Section II gives an overview of the HEVC interpolation and existing optimization techniques for it. The proposed filter optimizations are detailed in Section III and evaluated in Section IV. Finally, Section V concludes the paper.

II. HEVC Interpolation

In HEVC, the highest granularity of luma is 1/4-pixel (quarter-pixel, QPEL), but an encoder may limit the fractional *motion vector* (*MV*) accuracy to 1/2-pixel (half-pixel, HPEL) or integer precision. HEVC uses separable one-dimensional 8-tap (or 7-tap) filters for luma and 4-tap filters for chroma.

A. Basic Operating Principle

The interpolation is carried out by first applying the horizontal filter and then the vertical filter. For luma samples, the horizontal and vertical filtering steps are defined as

$$hor = \left(\sum_{x=-3}^{4} w_x \times ref[x_0 + x]\right) \gg (B-8), \quad (1)$$

$$ver = \left(\sum_{x=-3}^{4} w_y \times im\,[y_0 + y]\right) \gg 6, \quad (2)$$

where *hor* and *ver* denote horizontally and vertically filtered samples, respectively. The weights w_x and w_y belong to a predefined filter coefficient set. They are selected by the fractional part of the MV. A pixel row of the reference picture (*ref*) is horizontally indexed by x_0 and x that are given by the integer part of an arbitrary MV and a filter tap offset, respectively. Correspondingly, a horizontally filtered intermediate sample column (*im*) is vertically indexed by y_0 and y. B is the reference sample bit depth and the $\gg$ operator denotes a bitwise right shift.

B. Existing Optimizations

A couple of SIMD-optimized HEVC interpolation filters have been announced in the past decade, but they all suffer from functional or performance limitations. Most of them also stick to 128-bit legacy SIMD optimizations and give very little to no details about the vectorized implementation.

Chi et al. [9] presented SIMD optimizations up to AVX2 for the whole HEVC decoder and analyzed them on several

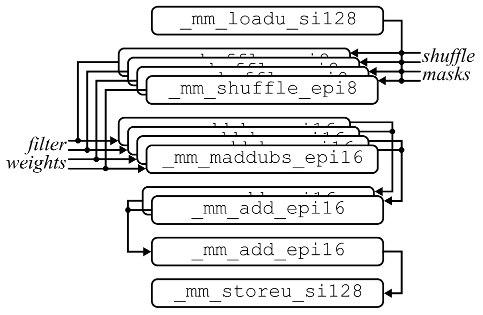


Fig. 1. Vectorized horizontal step to compute a 128-bit chunk of a row.

instruction set architectures and microarchitectures. However, as the work was limited to HEVC decoding, no encoder specific optimizations were considered.

The existing open-source HM [8] and x265 [15] encoders employ SIMD optimizations in interpolation. Nevertheless, HM settles for less intensive vectorization, whereas x265 optimizations are written in assembly for a large set of HEVC features, which results in code that is more laborious to read and maintain.

Additional SIMD optimizations were published for the HM encoder in [10], [11], of which the former interpolated fixed-sized blocks for motion compensation by using 128-bit SIMD extensions of the x86 architecture. The latter implemented a frame-level interpolation filtering scheme, where all fractional pixels were interpolated once and stored into buffers to avoid redundant operations. Parallelization with OpenMP or GPU offloading was suggested, but vectorization was not considered. It is also noteworthy that the complexity distribution of HM may differ from that of a practical encoder.

III. Proposed Vectorization Techniques

The proposed SIMD algorithms are implemented using AVX2 intrinsics, which are also used to specify the applied instructions in this paper. The focus is set on luma interpolation of 8-bit content, since chroma interpolation consumes significantly less CPU time in practical coding. Both luma and chroma *prediction blocks* (*PBs*) are filtered with similar techniques. The luma PB sizes range from 4×8 (or 8×4) to 64×64, whereas the 4:2:0 subsampled chroma PB dimensions are half of that.

A. Vectorization of Horizontal Interpolation

The proposed implementation is based on [10] and its novelty lies in an altered data permutation scheme that allows to replace the horizontal additions (`_mm_hadd_epi16`) of the original implementation with their regular counterparts (`_mm_add_epi16`). This approach reduces the number of issued micro-operations and thereby improves microarchitectural performance [16] [17]. Furthermore, the 128-bit design is extended to utilize 256-bit AVX2 operations.

The proposed approach is depicted in Fig. 1. It loads consecutive samples from a reference sample row to a vector register using `_mm_loadu_si128`. The samples are then replicated and shuffled into appropriate positions in four registers with `_mm_shuffle_epi8` and the corresponding shuffle control masks, `s01`–`s67`, as presented in Fig. 2. The vector of horizontally filtered 16-bit intermediate samples is obtained by first multiplying the samples with predefined filter coefficients, `w01`–`w67`, and adding the products

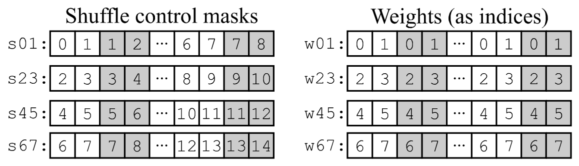


Fig. 2. Paired data arrangement for the horizontal interpolation stage.

pairwise while doubling the word size with `_mm_maddubs_epi16`. Then, `_mm_add_epi16` is used to sum up these parts in different registers until each group of eight multiplied input samples have been reduced to a single element in the 8-tap filtering. The filter weights are denoted in Fig. 2 by indices to a filter coefficient set that is specified by the fractional part of the MV.

Extending the method to 256 bits can be achieved without significant overhead by using the philosophy based on 128-bit *lanes* [16]. The extended functions can filter two rows at a time, one in both 128-bit lanes of a full 256-bit register. First, the data used in the 128-bit shuffle mask and filter weight registers is duplicated to both lanes of the register. Then, elements from the second row are loaded and inserted to the upper lane (`_mm256_inserti128_si256`). The modifications in shuffles and arithmetic operations are limited to replacing the 128-bit operations with their 256-bit counterparts. Similarly, contents of the upper lane are extracted to store the results into memory (`_mm256_extracti128_si256`).

B. Vectorization of Vertical Interpolation

This work proposes an original algorithm for the vertical interpolation stage. The main differences between the vectorized horizontal and vertical implementations are the memory access pattern and the arrangement of elements in the registers. Vertically aligned pixels from the intermediate array of horizontally filtered samples need to be transformed into a horizontal pairwise order. This way, the arithmetic operations can be performed with `_mm_madd_epi16` as in the horizontal filtering stage, but with larger element sizes due to the bit width requirements of the HEVC standard.

The vertical filtering scheme minimizes unnecessary memory operations with a sliding window stored in a set of registers. Conceptually, a block is stored in the vector registers so that a single register contains a group of adjacent samples from one row and different registers contain samples from consecutive rows. One output row can be filtered from 8 input rows for luma. The sliding window allows reusing the last 7 rows for the interpolation of the next result row, so only one new row needs to be loaded from memory to gather all input samples. Replacing the current row with the next one only requires one vector load and 7 register-to-register move instructions that are exceptionally quick to execute on Broadwell [17]. In this approach, the window traverses the data vertically one row at a time before processing the next group of columns. However, it also allows reusing the results of shuffle operations besides the previously loaded rows.

Unlike the horizontal step, the vertical step uses `_mm_unpacklo_epi16` to arrange data elements pairwise. It interleaves 16-bit elements of the low 64 bits of two 128-bit registers. The appropriate arrangement is achieved by interleaving the elements from two consecutive rows. When a result row is interpolated, the original 8 rows are transformed into 4 vector registers containing element pairs from the four adjacent columns. The window approach

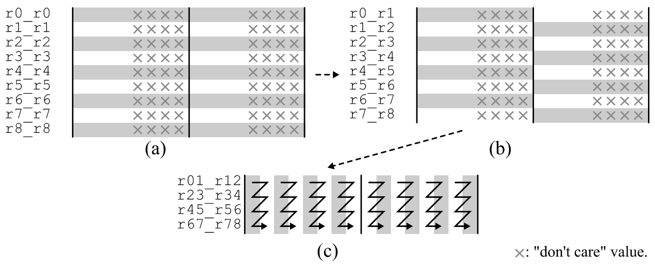


Fig. 3 Setting up the dual-lane filtering window of the vertical filtering step.

is further refined to store the samples in this modified arrangement instead of storing them by rows. The window can be shifted with register-to-register moves as before, but without the need for unpacking for the next iterations. However, four registers cannot maximize the reuse since they only contain pairs, where elements from even rows come first. That is, the first output row is filtered from input row pairs 01, 23, 45, and 67, whereas the second row requires row pairs 12, 34, 56, and 78. Thus, two rows are filtered at the same time to make even-odd and odd-even pairs available for the next iterations.

Before entering the loop for shifting the window and filtering rows, a separate initialization stage is used to ensure the completeness of the window on the first iteration. Each iteration completes two rows and shifts the window.

There are two versions of the vertical filtering algorithm, called *filter backends*: one that keeps *high precision* (*hi*) for blending in bidirectional prediction, and another that rounds the filtered samples back into *pixel precision* (*px*). The px backend of the vertical step performs the bit-shift and add operations to round the samples according to the HEVC standard before finishing. On top of regular vector additions, this is achieved with `_mm_srai_epi32`, `_mm_packs_epi32`, and `_mm_packus_epi16`.

Similar to the horizontal step, the 256-bit dual-lane implementation does not require many changes, except that the 128-bit algorithm is extended to 256 bits that filters the even output rows in the low lanes and the odd output rows in the upper lanes simultaneously. Fig. 3 visualizes the steps needed to set up the sliding window in both lanes. As illustrated in Fig. 3(a), `_mm256_set1_epi64x` loads data from memory and broadcasts it into the whole vector register by filling it with the same 64-bit elements (made up of four 16-bit samples in this case). After broadcasting, the upper lane is deferred by one row with `_mm256_blend_epi32` so that the lower lane is selected from the first register and the upper lane from the second, as shown in Fig. 3(b). After this step, data is already in the correct arrangement for the interleaving which is performed using `_mm256_unpacklo_epi16`. In Fig. 3(c), jagged arrows visualize the eight consecutive samples of the same column. Before interleaving, the upper 64 bits of both lanes contain the same data as the low 64 bits. Thus, they are considered “don’t care” values, as visualized by crosses in Fig. 3. In the end, `_mm256_extracti128_si256` is used to extract the data of the upper lane and the filtered rows are stored separately. The dual-lane approach requires slight changes to the window moving and initialization. For example, a single register-to-register move shifts the window by two rows at a time in both lanes.

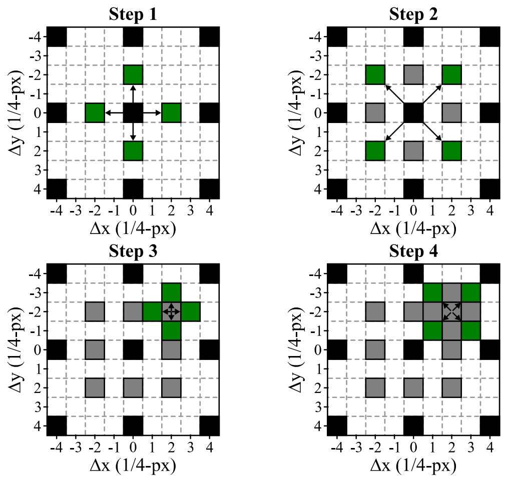


Fig. 4. FME search pattern and steps in *Kvazaar* at sample level.

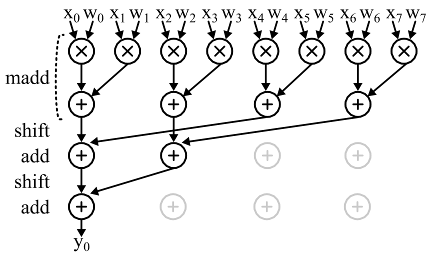
C. Vectorized Interpolation for FME

To improve accuracy and coding efficiency after integer motion estimation, *Kvazaar* performs FME in several steps as illustrated in Fig. 4. Each step searches for four new fractional MVs. The first half of the search looks for the optimal HPEL-precision MV. The second half repeats the process by searching for the optimal QPEL-precision MV around the HPEL position. Both stages are further split into two parts to enable more fine-grained control over coding efficiency and complexity. PBs given by the MVs are illustrated in green at sample level in Fig. 4.

The computational complexity of FME is reduced by taking advantage of separable interpolation filters and the significant overlap of the blocks instead of interpolating the four blocks separately. For example, the PBs with vertical HPEL displacements share all but one row with the other, so only the bottom row needs to be filtered for the bottom PB and the rest can be reused from the top PB. Essentially, interpolating these two $N \times M$ blocks equals interpolating a single $N \times (M+1)$ block instead. The same applies for horizontally displaced blocks and reusing columns of filtered samples. Furthermore, the input samples for the vertical filtering stage can be reused in a similar way, since horizontally aligned but vertically displaced fractional-position PBs share most of the horizontally filtered samples.

Joint interpolation of four blocks complicates the efficient vectorization of FME with the aforementioned functions because one additional column of reference samples needs to be interpolated for PBs that have negative horizontal fractional displacement. However, the AVX2 implementation interpolates at least four columns since the luma PBs of HEVC have widths that are multiples of four. This simplifies vectorization, since the vector registers contain a power of two number of elements. However, after introducing the extra column, the input and intermediate data have an odd number of samples. To that end, the extra columns are filtered and written in contiguous memory separately from the other intermediate blocks after the horizontal filtering stage. This enables reusing the AVX2 interpolation functions for most samples, optimal memory alignment for the intermediate blocks, and efficient memory accesses to the columns with vector memory operations.

Since the column sample count is relatively low, optimizing the column filtering has less significance. Therefore, the column filtering follows the approach presented in this paper, but it utilizes 128-bit registers partially



x_0 w_0 x_1 w_1 x_2 w_2 x_3 w_3 x_4 w_4 x_5 w_5 x_6 w_6 x_7 w_7
madd
shift
add
shift
add
y_0

Fig. 5. Filtering a single sample with partial vector utilization.

TABLE I. EXPERIMENTAL SETUP

Hardware	
CPU	Intel Xeon Processor E5-2620 v4
# of Cores | Threads	8 | 16
Base Frequency	2.10GHz
RAM	32 GB / DDR4 2133 MHz
L1 | L2 | L3 cache	512 KB | 2.0 MB | 20.0 MB
Storage	SanDisk SSD X400 (540 MB/s read)
Software	
Encoder	*Kvazaar* 2.0 (c36d423)
Compiler	MSVC 2017 (19.16.27025.1)
Profiler	Intel VTune Profiler 2020 (Update 2)
Test Framework	uvgVenctester
Operating System	Microsoft Windows 10 (18363.1440)
Encoder configuration	
Config.	Parameters
Ultrafast	--preset ultrafast --threads 16 -q{QP} --[no-]cpuid --subme 2
Veryslow	--preset veryslow --threads 16 -q{QP} --[no-]cpuid

to interpolate a single sample at a time. Even though the full capacity is not utilized, many operations, especially the multiplications, are computed in parallel using the MapReduce pattern akin to Fig. 5, where the results of the operations in grey are ignored and the final value is obtained after the last step. The contiguous data layout also enables vectorizing vertical filtering of the extra columns efficiently with the same "horizontal" scheme. How the blocks are composed after the vertical filtering, depends on the MVs.

IV. PERFORMANCE RESULTS AND ANALYSIS

Table I tabulates the hardware and software setups as well as *Kvazaar* encoder configurations [13] used in our experiments. The *ultrafast* configuration was specified to perform two steps of FME with the `--subme` command-line option, whereas the *veryslow* preset implicitly includes four steps. The test set was composed of 24 HEVC common test sequences grouped into classes A–F according to resolution, content, and bit depth [18]. Encoding speeds were measured with *uvgVenctester* [19] and the relative CPU time of functions with the *Hotspots* analysis of *VTune* [20].

Table II presents the CPU times and speedups of individual functions and logical function groups when *Kvazaar* was run with and without SIMD extensions. The results are hierarchically tabulated by grouping the entries of the same abstraction level with indentation and shading.

In all-scalar *Kvazaar*, the functions of interest to this work account for 81.5% and 75.0% of the total encoding time with the *ultrafast* and *veryslow* presets, respectively. Altogether, the proposed techniques yield roughly a 7-fold speedup with both configurations and reduce the share of these functions down to 46.4% and 33.8% of the total encoding time.

The luma sample interpolation for PBs gains up to a 14-fold speedup over the scalar equivalents, whereas the filtering for FME accelerates the operation by a factor of five at HPEL

TABLE II. COMPARISON OF SCALAR AND VECTORIZED CODING TOOLS

	Ultrafast			*Veryslow*		
	CPU time		Speedup	CPU time		Speedup
Coding tool	**Scalar**	**SIMD**		**Scalar**	**SIMD**	
Vectorized functions	81.5%	46.4%	7.1×	75.0%	33.8%	6.8×
– Interpolation	34.8%	14.5%	9.7×	23.3%	8.0%	8.5×
— Interpolation, PB	27.4%	8.6%	12.9×	8.0%	2.1%	11.5×
---- Luma, px backend	3.1%	1.0%	12.4×	1.2%	0.4%	10.4×
---- Luma, hi backend	22.5%	6.5%	14.1×	6.1%	1.4%	13.6×
---- Chroma, px backend	0.3%	0.2%	6.0×	0.2%	0.2%	3.5×
---- Chroma, hi backend	1.4%	0.8%	6.9×	0.5%	0.2%	7.3×
— Interpolation, FME	7.4%	5.9%	5.1×	15.3%	6.3%	7.5×
---- Luma, 1/2-px	7.4%	5.9%	5.1×	4.0%	2.6%	4.7×
---- Luma, 1/4-px	-	-	-	11.3%	3.7%	9.4×

TABLE III. SPEEDUPS OF LUMA FILTER AND ENCODER VECTORIZATION

		Speedup *ultrafast*	**Speedup *veryslow***
Filter	Luma, horizontal	16.4×	15.2×
	Luma, vertical, px	10.6×	9.2×
	Luma, vertical, hi	11.7×	11.4×
***Kvazaar* encoder**		**1.67×**	**1.58×**

precision. Even if almost the same low-level interpolation functions are utilized in FME and PB generation, the interpolation of extra column inflates the proportion of scalar and less intensively optimized vector code in FME. Furthermore, the scalar implementation of HPEL block interpolation exploits algorithm-level optimizations, like specializations for horizontal- or vertical-only filtering, that are not available in the vectorized implementation. Similarly, the speedups of chroma filtering with the px backend are more moderate. For example, more than one third of chroma interpolation with the px backend is spent by the scalar functions in the *veryslow* case, but its infrequent usage reduces the need for optimizations.

Table III highlights the speedups of the vectorized luma interpolation stages over the corresponding scalar implementations. Similar speedups are obtained with both configurations and the minor deviation between them stems from different block sizes and shapes. The 16-fold speedup achieved at the horizontal filtering stage can be considered satisfactory, since the vectorization makes it possible to compute 16 or 32 elements in parallel when the 256-bit registers are filled with 16-bit or 8-bit elements, respectively. The vertical stages are not more than 10 times as fast after vectorization, because they involve extra data arrangement. Altogether, the proposed filter vectorizations accelerate *Kvazaar ultrafast* configuration by 1.67× and *veryslow* configuration by 1.58× over the all-scalar anchor configuration of Kvazaar.

V. CONCLUSION

This work presented AVX2-vectorized interpolation filters for HEVC and implemented them into *Kvazaar* software encoder. The proposed optimizations included a novel data permutation scheme for the horizontal interpolation filters and a sliding window strategy for the vertical interpolation filters. In addition, it was shown that they can be efficiently utilized in FME. All these optimizations also support a lane-based approach for doubling the vector lengths from 128-bit legacy vector extensions to full 256-bit AVX2 instructions. The entire interpolation was accelerated by 9.7 and 8.5 times with high-speed and high-quality coding settings of Kvazaar. The proposed techniques are virtually agnostic to coding block sizes and shapes, so they can be used as is or with minimal changes to boost practical VVC encoders as well. Additionally, similar approach could be used with AVX512, by processing four rows at a time instead of two.

REFERENCES

[1] 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, Aug. 2003, pp. 560–576.

[2] 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, Dec. 2012, pp. 1649–1668.

[3] ITU, "New 'Versatile Video Coding' standard to enable next-generation video compression," Sep. 2020, [Online]. Available: https://www.itu.int/en/mediacentre/Pages/pr13-2020-New-Versatile-Video-coding-standard-video-compression.aspx.

[4] Bitmovin, "Bitmovin Video Developer Report 2019," 2019, [Online]. Available: https://cdn2.hubspot.net/hubfs/3411032/Bitmovin%20Magazine/Video%20Developer%20Report%202019/bitmovin-video-developer-report-2019.pdf.

[5] J. Vanne, M. Viitanen, T. D. Hämäläinen, and A. Hallapuro, "Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs," *IEEE Trans. Circuits Syst. Video Technol.,* vol. 22, no. 12, Dec. 2012, pp. 1885–1898.

[6] G. Ramasubbu, A. Kaup, and C. Herglotz, "Modeling the HEVC encoding energy using the encoder processing time," in *Proc Int. Conf. on Image Process.*, Bordeaux, France, Oct 2022, pp. 3241-3245.

[7] K. Ugur et al., "Motion compensated prediction and interpolation filter design in H. 265/HEVC," *IEEE J. Sel. Topics Signal Process.,* vol. 7, no. 6, Dec. 2013, pp. 946–956.

[8] "HEVC Reference Software Version 16.20," [Online]. Available: https://vcgit.hhi.fraunhofer.de/jct-vc/HM/-/tags/HM-16.20.

[9] C. C. Chi, M. Alvarez-Mesa, B. Bross, B. Juurlink, and T. Schierl, "SIMD acceleration for HEVC decoding," *IEEE Trans. Circuits Syst. Video Technol.,* vol. 25, no. 5, May 2014, pp. 841–855.

[10] K. Chen, Y. Duan, L. Yan, J. Sun, and Z. Guo, "Efficient SIMD optimization of HEVC encoder over X86 processors," in *Proc. Asia Pacific Signal Inf. Process. Assoc. Annu. Summit Conf.*, Hollywood, California, USA, Dec. 2012, pp. 1–4.

[11] Y.-J. Ahn, T.-J. Hwang, D.-G. Sim, and W.-J. Han, "Implementation of fast HEVC encoder based on SIMD and data-level parallelism," *EURASIP J. Image Video Process.,* vol. 16, Mar. 2014, pp. 1–19.

[12] A. Lemmetti, M. Viitanen, A. Mercat, and J. Vanne, "Kvazaar 2.0: fast and efficient open-source HEVC inter encoder," in *Proc. ACM Multimedia Syst. Conf.*, New York, New York, USA, May 2020, pp. 237–242.

[13] Ultra Video Group, "Kvazaar open-source HEVC encoder," [Online]. Available: https://github.com/ultravideo/kvazaar.

[14] Ultra Video Group, "Kvazaar interpolation filter AVX2 optimization," [Online]. Available: https://github.com/ultravideo/kvazaar/blob/c36d423a8c022c2e34c88bbe7e32e05b1fe73217/src/strategies/avx2/ipol-avx2.c.

[15] MulticoreWare, Inc., "x265 HEVC encoder / H.265 video codec," [Online]. Available: https://bitbucket.org/multicoreware/x265/downloads.

[16] Intel Corporation, "Intel 64 and IA-32 architectures optimization reference manual," Feb. 2022, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[17] A. Fog, "Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," Jun. 2022, [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf.

[18] F. Bossen, "Common test conditions and software reference configurations," *document JCTVC-L1100*, Geneva, Switzerland, Jan. 2013.

[19] J. Sainio, A. Mercat, and J. Vanne, "uvgVenctester: open-source test automation framework for comprehensive video encoder benchmarking," in *Proc. ACM Multimedia Syst. Conf.*, Istanbul, Turkey, Jun. 2021, pp. 255–260.

[20] Intel Corporation, Inc., "Intel VTune performance analyzer," [Online]. Available: https://software.intel.com/content/www/us/en/develop/home.html.