Tailored AVX2 Transform Kernels for Versatile Video Coding

Kari Siivonen, Joose Sainio, Alexandre Mercat, and Jarno Vanne
Ultra Video Group, Tampere University
Tampere, Finland
{kari.siivonen, joose.sainio, alexandre.mercat, jarno.vanne}@tuni.fi

***Abstract*—Transform coding tools play an integral part in video codecs due to their substantial impact on coding efficiency. The latest video coding standard, Versatile Video Coding (VVC), makes the most of these tools by introducing new DST7, DCT8, and non-square transforms alongside the conventional DCT2 transform. This paper proposes optimized AVX2 kernels for all these transforms to speed up VVC coding. Unlike existing solutions, our kernels are specially tailored for each VVC transform type and block size. Accelerating our open-source uvg266 VVC encoder with the proposed kernels yields up to a 1.1× speedup under all intra (AI) coding condition without any coding overhead. Our implementations make forward DCT2 and DST7/DCT8 transforms 4.0× and 6.7× as fast as their respective scalar implementations in the VTM reference encoder. They also outpace the AVX2 kernels of the practical VVenC encoder by factors of 3.0× and 2.8×. The respective speedups rise up to 5.3×, 11.1×, 3.4×, and 3.0× with inverse transforms.**

Keywords—Versatile Video Coding (VVC), transform, complexity reduction, Advanced Vector Extensions 2 (AVX2), practical encoder implementation

I. Introduction

The exponential growth of global video traffic is primarily fueled by the proliferation of bandwidth-intensive media applications. The *Moving Picture Experts Group* (*MPEG*) has risen to this challenge by publishing a series of video coding standards, with the latest being *Versatile Video Coding* (*VVC*) [1]. VVC provides a significant coding gain over its predecessor, *High Efficiency Video Coding* (*HEVC*) [2], but at the cost of substantially higher computational complexity [3]. Therefore, developing efficient implementations becomes imperative for widespread adoption of VVC technology.

Implementations of VVC encoders, and video encoders in general, can be broadly categorized into *hardware* (*HW*) and *software* (*SW*) solutions. HW implementations, typically in the form of dedicated ASICs or FPGAs, offer the benefits of speed and energy efficiency [4] but at the cost of flexibility. In contrast, SW implementations, while running on general-purpose processors, offer the benefits of adaptability and ease of updates, often allowing for more rapid iterations and flexibility than dedicated HW. In the domain of open-source SW VVC encoders, there are currently three noteworthy solutions: *VVC Test Model* (*VTM*) [5], VVenC [6], and uvg266 [7]. VTM is the reference encoder, which implements all the VVC coding tools but lacks optimization. VVenC is a streamlined version of VTM with many optimizations [8]. Finally, uvg266 is our practical SW encoder that is developed from our open-source Kvazaar HEVC encoder [9].

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).

Pushing the boundaries of software encoder performance hinges on maximizing processor capabilities, where vector extensions like AVX2 and AVX512 come to the fore. This work makes use of these extensions and presents AVX2 vectorized kernels for forward and inverse transforms of the uvg266 encoder. Our preference for AVX2, over the newer AVX512, stems from its wider adoption in current systems [10]. Given that transforms account for 10–30% of total encoding time [3] [11] [12], streamlining them can lead to a noticeable reduction in the overall VVC encoding complexity. Our experiments demonstrate that the proposed kernels outperform both the VTM scalar and VVenC AVX2 implementations. These kernels are incorporated into uvg266, which is publicly available on GitHub [7] under a permissive 3-clause BSD license.

The remainder of this paper is structured as follows. Section II looks into the working principles of transforms in VVC, reviews related works in the domain, and brings up their limitations. Section III details the proposed AVX2 kernels and Section IV reports the performance results for them. Finally, Section V concludes the paper.

II. Transforms in VVC

VVC encoders employ the classical hybrid video coding principle. Initially, the input frame is predicted using spatially and temporally co-located areas. The residual of this prediction is then transformed into frequency domain. Given that humans are more sensitive to changes in lower frequencies, manipulating values within frequency domain facilitates higher bitrate reduction than direct pixel domain operations [11].

A. Basic Operating Principle

In VVC, an input frame is divided into blocks of varying sizes. To find the optimal block structure and corresponding prediction mode, the encoder follows a "try-all-select-the-best" approach by comprehensively testing each potential combination. This involves generating a prediction, performing the forward transform on the residual, quantizing and dequantizing the transformed coefficient, and executing the inverse transform. A single frame might necessitate tens of millions of these evaluations, so having efficient forward and inverse transforms becomes indispensable in real-time VVC encoding.

In the VVC standard, the implementations of forward and inverse transforms are articulated as matrix multiplication using 6-bit fractional precision, based on predefined coefficient tables. These transforms are *two-dimensional* (*2D*) and involve both horizontal and vertical transforms. The horizontal transform is executed first in the forward case and the inverse transform in the vertical case.

979-8-3503-3757-0/23/$31.00 ©2023 IEEE

Compared to its predecessor HEVC, VVC newly introduces non-square transforms and *multiple transform selection* (*MTS*) search process. The main challenge posed by VVC non-square transforms is that both horizontal and vertical transforms must be aware of and account for the dimensions of the other. VVC supports 2D transforms with size of $N \times M$, where $N, M \in \{2, 4, 8, 16, 32, 64\}$ so that 64 is optional. Although MTS offers a choice between *discrete sine transform* 7 (*DST7*) and *discrete cosine transform 8* (*DCT8*) alongside the default DCT2, its implementation remains straightforward. The underlying process is the same matrix multiplication, albeit with different coefficients. However, it is important to note that the $N \times 32$ and $32 \times M$ transforms only produce maximum of $N \times 16$ and $16 \times M$ coefficients, respectively.

B. Existing Transform Optimizations

Two seminal studies on HEVC [13], [14] provide context for advancements of VVC. In HEVC, the DCT2 transform leveraged the partial butterfly algorithm to minimize the number of multiplications [13]. Furthermore, we [14] introduced AVX2 vectorized kernels for the square transforms in HEVC.

Two major trends emerge in the VVC transform optimization in the literature.

1) Pruning MTS search process: several studies have proposed simplifying the transform process by trimming down the number of transforms evaluated during the search.

- Fu *et al.* [15] and He *et al.* [16] both introduced early termination mechanisms for the MTS search based on the spatial frequency and the calculated rate-distortion cost of the transforms.
- Wang *et al*. [17] put forth a method that bypasses specific DST7 and DCT8 combinations based on which residual corner exhibits the highest intensity.
- Saldanha *et al.* [18] ventured a different path by employing decision trees to reduce the number of evaluated transforms for each block.

2) Addressing transform complexity: A distinct set of works has focused on directly mitigating the inherent complexity of the transforms.

- Zhang *et al.* [19] introduced a partial butterfly optimization specifically tailored for DST7 and DCT8 kernels. These techniques were included in VTM, in addition to the partial butterfly optimization of DCT2.
- Hao *et al.* [20] took a more HW-oriented approach and designed a reconfigurable MTS architecture for both FPGAs and ASICs.
- VVenC includes AVX2 vectorized kernels for DCT2, DST7, and DCT8 [21]. VVenC uses generic functions for both forward and inverse transforms, forcing a consistent arrangement after each vertical and horizontal transform.

C. Limitations of Existing Solutions

Pruning the MTS search process indeed reduces the overall encoding complexity, but the inherent complexity of the matrix multiplication remains at both encoding and decoding stages. HW-oriented solutions, like those designed for FPGA and ASIC, perform efficiently, but their rigidity can hinder adaptability. Slow adoption rates of VVC HW decoders and encoders sets the stage for potent SW alternatives. However, existing AVX2 kernels designed specifically for HEVC square blocks would need modifications for non-square blocks of VVC, and the implementation would still be suboptimal. Conversely, AVX2 kernels of the VVenC speed up non-square VVC transforms, but their generalized approach to 2D transforms suggests untapped performance potential.

TABLE I. PERFORMANCE OF RELEVANT INSTRINSICS [22]

Intrinsic	Latency (cycle)	Throughput (cycle)
`_mm256_madd_epi16`	5	0.50
`_mm256_mulhi_epi16`	5	0.50
`_mm256_mullo_epi16`	5	0.50
`_mm256_mullo_epi32`	10	0.66
`_mm256_add_epi32`	1	0.33
`_mm256_hadd_epi32`	3	2.00
`_mm256_shuffle_epiX`	1	1.00
`_mm256_permuteNxM_epiX`	3	1.00
`_mm256_packs_epi32`	1	1.00
`_mm256_unpacklo_epiX`	1	1.00
`_mm256_unpackhi_epiX`	1	1.00
`_mm256_set1_epi32`	3	0.92

III. PROPOSED TRANSFORM KERNELS

Our main design principle is to craft dedicated transform functions for each VVC transform and block size. All these kernels are designed for uvg266 which does not support transforms with a dimension of 64. Therefore, only transform sizes of up to 32×32 pixels are addressed in this paper.

A. High-Level Structure

Explicitly defining distinct VVC transform functions for each block size brings two advantages. First, adaptive ordering ensures that results from the initial transform are not tied down to a specific order, but the subsequent transform has the freedom to use results directly or adjust their order as necessary. Secondly, each transform size is individually optimized for performance. However, the improved performance comes with increased development time.

The structural blueprint for implementing each vertical and horizontal transform remains consistent. The samples are initially arranged for the first transform that does not impose any specific order on its results. Instead, the second transform ensures the final outcomes are sequenced appropriately for subsequent operations.

Each transform is implemented with a control function and two sub-functions dedicated to each vertical and horizontal transform. The control function takes care of different transform types by selecting the appropriate coefficients and setting up common parameters needed by sub-functions. It avoids direct data manipulation by delegating all computational tasks and sample rearrangements to the sub-functions.

B. Intrinsic Choices

All of the computations start from the underlying fact that uvg266 stores residual as 16-bit integers. Table I reports latencies and throughputs for intrinsics relevant to transform implementation, as observed on the Skylake architecture [22]. Latency equals the cycle count before a result of an operation becomes available, whereas throughput measures the average cycle count per produced output. For instance, a throughput of 0.5 for the `_mm256_madd_epi16` intrinsics suggests that two multiply add (`madd`) intrinsics complete every cycle.

The `madd` intrinsic emerges as the most efficient multiplication intrinsic. It facilitates sixteen multiplications and eight additions, resulting in eight 32-bit values. In contrast, intrinsics like multiply low (`mullo`) and multiply high (`mulhi`) demand two `unpack` intrinsics, and an addition

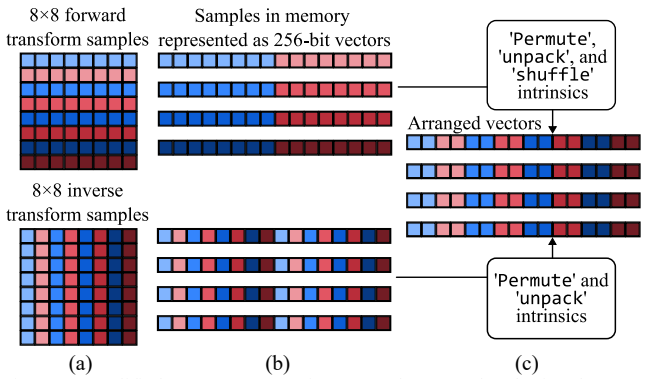

Fig. 1. Exemplified rearrangement of 8×8 samples. Samples sharing the same colour and shade undergo multiplication with a. (a) Samples represented as transform blocks. (b) Memory representation in a vectorized format. (c) Vectors arranged for calculation.

to achieve the same results, making them less efficient. Correspondingly, higher performance of regular addition (`add`) intrinsics makes them preferable over horizontal addition (`hadd`) intrinsics.

C. Pre-Processing

The effective utilization of `madd` and `add` intrinsics necessitates a strategic rearrangement of samples. Fig. 1 visualizes the rearrangement process with 8×8 samples. Fig. 1(a) presents the samples as transform blocks, Fig. 1(b) showcases their memory representation in a vectorized format, and Fig. 1(c) depicts the vectors arranged for calculation. Using `madd` intrinsics efficiently requires that values intended for addition are paired. Furhermore, the outputs of consecutive `madd` intrinsics should be organized according to Fig. 1(c) to facilitate their summation with `add` intrinsics. Given this storage method, reordering the coefficient tables is preferred over sample reordering, as coefficient tables can be freely reordered in advance, whereas samples must be reordered in real-time during the transform. Although rearrangement requires extra cycles it enables more performant `madd` and `add` instrinsics.

In addition to sample rearrangement, there is a need to rearrange transform coefficients. Two primary strategies are employed for these calculations:

1. Strategy 1: coefficient pair duplication followed by matrix calculation in row major order;
2. Strategy 2: sample pair duplication with calculations in column major order.

It is worth noting that Strategy 1 always requires the re-arrangement process. Strategy 2 however, requires the samples to be in a linear order, so the re-arrangement can be omitted in specific cases, e.g., the horizontal half of any 32×N forward transform.

For transforms with dimensions 2, 4, and 8, coefficient values are duplicated in static tables. However, minimizing the cache load becomes essential with larger coefficient tables. Therefore, larger transforms performed duplication during calculation using `_mm256_set1_epi32` intrinsic.

For the smaller dimensions, Strategy 1 proves to be more optimal. It allows calculations in a sequence that does not immediately demand the result of a multiplication, effectively countering the notable latency of the `madd` intrinsic. On the other hand, Strategy 2 is favored for transforms of a 32-dimension for two reasons. First, Strategy 1 encounters cache load issues because of the surge of intermediate values.

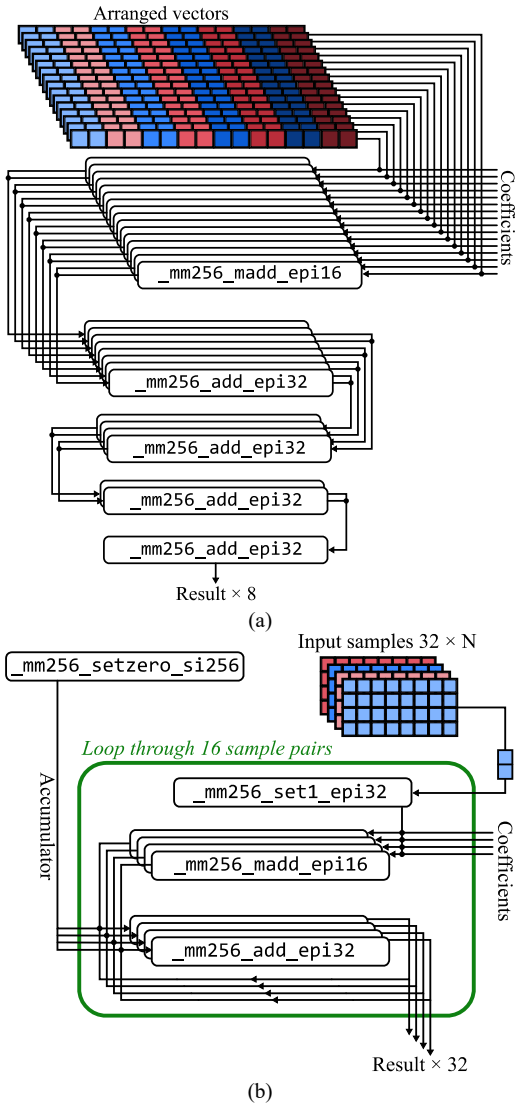

Fig. 2. Comparative implementations of 32×N transforms. (a) Coefficient duplication with intermediate values, processing eight rows of samples alongside a single row of coefficients. (b) Sample duplication, using accumulator variables, processing a single row of samples.

Secondly, the latency of `madd` operations can be mitigated by executing multiple multiplications using same sample data but varying coefficient sets.

D. Core Computation

After the sample rearrangement, the execution of both forward and inverse transforms are largely identical in both horizontal and vertical directions. The primary distinction between them arises from the choice between Strategy 1 and Strategy 2. The former duplicates coefficients and uses intermediate values, whereas the latter duplicates the samples and uses accumulators in place of intermediate variables. They are visualized with 32×N transforms in Fig. 2, where Fig. 2(a) illustrates Strategy 1 and Fig. 2(b) Strategy 2.

Using Strategy 1, all multiplications for one row of samples are first executed as `madd` intrinsics. This extends the outcomes to a 32-bit format, which are then stored as intermediate variables. In the case of 16×16 transform, the calculation is divided into two loops to alleviate cache load. Following this, the intermediate variables are summed up. Smaller transforms require only a few additions to yield the

TABLE II. DISTTRIBUTION OF ASSEMBLY OPERATIONS FOR 32×32 HORIZONTAL FORWARD TRANSFORM

Scalar	Cycle count	Share
control logic	1260	25.1%
Vectorized instructions		
`vmovaps`	6	0.1%
`vmovdqu`	816	16.3%
`vpbroadcastd`	197	3.9%
`vpmaddwd`	**1213**	**24.2%**
`vpaddd`	1231	24.5%
`vmovd`	159	3.2%
`vpsrad`	52	1.0%
`vpackssdw`	50	1.0%
`vmovups`	35	0.7%
Total	5019	100.0%

final results, whereas larger transforms necessitate more additions due to a greater number of intermediate results. For transforms where one dimension is either two or four, `hadd` intrinsics are essential because regular 32-bit additions produce eight-wide results. Generally, Strategy 1 is better suited for smaller transforms.

In Strategy 2, instead of using intermediate variables, the addition process leverages an accumulator. While this method introduces a few redundant additions on the first iteration of the accumulator, the overal performance is better due to the minimized cache load issues. Overall, Strategy 2 is more suited for larger transforms. Moreover, as explained at the end of Section II.A, for DST7 and DCT8 transforms with a dimension of 32, some calculations become redundant. In the horizontal forward transform for 32-wide transforms, there is no need to compute the right half of the values. Similarly, for vertical transforms with height of 32, only the calculations for the top half are necessary. In the case of the vertical inverse transform with width of 32, the values on the right half invariably results in zero and can be omitted.

E. Partial Butterfly Consideration

The partial butterfly algorithm [13] is able to accelerate scalar implementations, but the gain does not translate to AVX2 kernels as such for several reasons. The major drawback of the partial butterfly algorithm stems from the second transform, where executing additions first can cause overflows with 16-bit numbers, preventing the efficient utilization of `madd` instrinsics. Instead, the inputs of the additions must be extended to 32 bits, necessitating the use of regular 32-bit multiplications due to the absence of a 32-bit `madd` intrinsics. The partial butterfly algorithm reduces total number of multiplications, but the instrinsics deployed for these multiplications can handle less than half as many operations per cycle compared with `madd` intrinsics, as highlighted in Table I.

Although the overflow does not occur during the first transform, the partial butterfly algorithm does not yield significant performance gain. Table II lists the cycle counts of the 32×32 forward horizontal transform, breaking down both vectorized instructions and the scalar code essential for control logic. Notably, only 24.2% of cycles are dedicated to multiplications, suggesting potential savings for the partial butterfly method. However, this advantage becomes intangible. The intermediate additions, which in the partial butterfly method replace multiplications, necessitate storing the values. This act inflates the move operations, erasing the gains achieved by cutting down multiplications, due to exacerbated cache load issues and the relatively low performance gap between `madds` and `adds`.

TABLE III. TEST SEQUENCES

Class	Format	Sequence	Frame count	Frame rate
VVC-A1	3840×2160 (2160p)	Campfire	300	30 fps
		FoodMarket4	300	60 fps
	4096×2160	Tango2	294	60 fps
VVC-A2	3840×2160 (2160p)	CatRobot	300	60 fps
		DaylightRoad2	300	60 fps
		ParkRunning3	300	50 fps
VVC-B	1920×1080 (1080p)	BasketballDrive	500	50 fps
		BQTerrace	600	60 fps
		Cactus	500	50 fps
		MarketPlace	600	60 fps
		RitualDance	600	60 fps
VVC-C	832×480 (480p)	BasketballDrill	500	50 fps
		BQMall	600	60 fps
		PartyScene	500	50 fps
		RaceHorses	300	30 fps
VVC-D	416×240 (240p)	BasketballPass	500	50 fps
		BlowingBubbles	500	50 fps
		BQSquare	600	60 fps
		RaceHorses	300	30 fps
VVC-E	1280×720 (720p)	FourPeople	600	60 fps
		Johnny	600	60 fps
		KristenAndSara	600	60 fps

TABLE IV. EXPERIMENTAL SETUP

Hardware	
CPU	Intel Xeon Processor W-2145 (Skylake)
# of Cores | Threads	8 | 16
Base Frequency	3.70GHz
RAM	64 GB / DDR4 2400 MHz
L1 | L2 | L3 cache	512 KB | 8.0 MB | 11.0 MB
Storage	SanDisk SSD X600
Software	
Encoder	*uvg266* v0.8.0
Compiler	MSVC 2022 v14.36-17.6)
Profiler	Intel VTune Profiler 2023.1.0
Test Framework	uvgVenctester
Operating System	Microsoft Windows 10 (19045.3208)

IV. EVALUATION OF THE PROPOSED KERNELS

This section delves into the assessment of the proposed transform kernels. Although the primary focus is on encoding, the inverse transforms are also applicable in decoding.

A. Experimental Setup

The experiments were carried out with *uvg266* encoder [9] under the AI condition defined by the VVC *common test conditions* (*CTC*) [23]. Even though the proposed kernels were primarily optimized for *All Intra* (*AI*) coding, they were architecturally designed to operate efficiently across all coding conditions. Table III lists the applied CTC test sequences that are categorized into six distinct classes based on varying resolutions and content types.

The kernels were benchmarked against existing solutions: the partial butterfly optimized scalar implementations of VTM and the AVX2 kernels of VVenC, hereafter referred to as **scalar-VTM** and **AVX2-VVenC**, respectively. For an accurate head-to-head assessment, transform implementations were isolated from their respective encoders and evaluated within a dedicated test bench called **standalone** testing.

In **in-encoder** testing, kernel efficiency was evaluated by comparing two versions of uvg266: one with the proposed kernels and another with scalar kernels similar to those of VTM. An in-encoder comparison of these kernels with VVenC was deemed not viable as it would introduce many extraneous factors that would be against fair comparison.

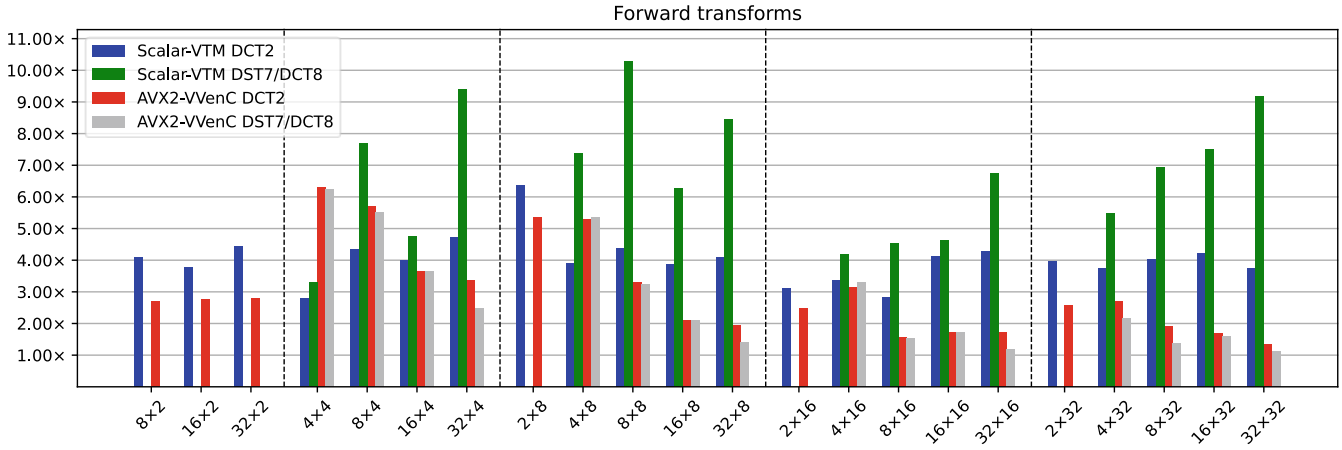

Fig 3. Standalone speedup of the proposed forward transform kernels over scalar-VTM and AVX2-VVenC implementations.

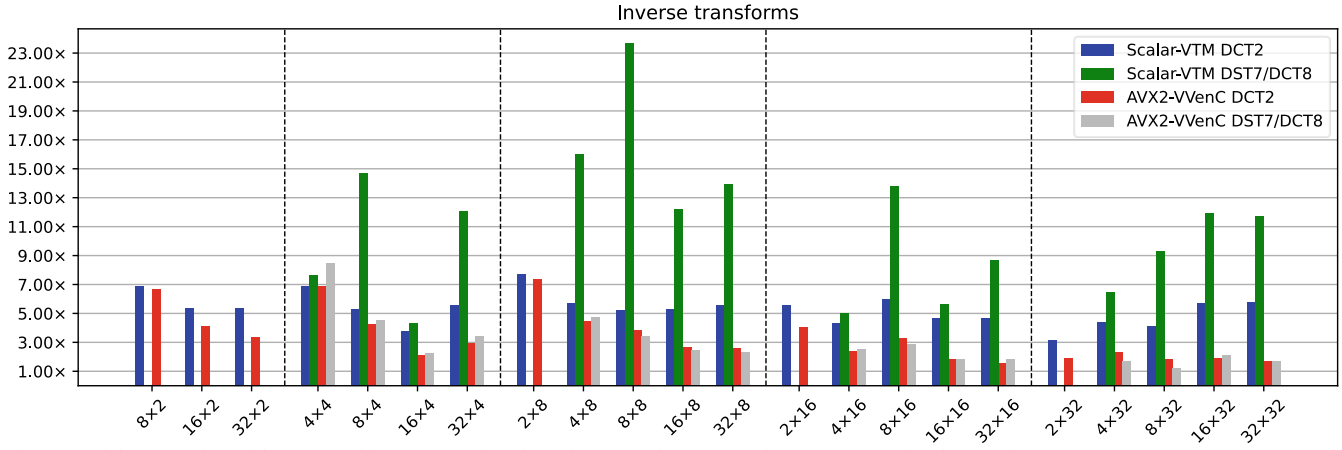

Fig 4. Standalone speedup of the proposed inverse transform kernels over scalar-VTM and AVX2-VVenC implementations.

The platform used for testing is detailed in Table IV. Standalone transform results were obtained using Intel VTune [24], ensuring minimal interference from auxiliary code. Conversely, in-encoding results were derived from uvgVenctester [25], a tool designed for measuring encoder performance. Given that both original transforms and proposed kernels yielded identical output bitstreams, only encoding speed results are provided.

B. Experimental Results

Fig. 3 presents the **standalone** speedup results for the proposed forward transform kernels when benchmarked against scalar-VTM and AVX2-VVenC. Despite minor variances, the proposed kernels consistently outpace the scalar-VTM implementations. On average, our kernels are roughly four times as fast as the partial butterfly DCT2 implementations of scalar-VTM. The speedup is even higher with DST7/DCT8 because the partial butterfly optimizations do not harness the potential to omit certain portions of the transforms, particularly those with a dimension of 32. Comparing DCT2 with AVX2-VVenC highlights that the VVenC implementation is less optimized for smaller transforms. Moreover, for larger transforms, the proposed kernels consistently outperformed AVX2-VVenC implementations. When considering DST7/DCT8 transforms, especially those with a dimension of 32, the speedup might not be as significant as with DCT2 because the efficacy of the proposed kernels diminishes slightly as the computational load decreases. Nevertheless, the proposed kernels are consistently faster than those of VVenC.

Fig. 4 depicts the respective **standalone** speedup results for our inverse transform kernels, whose performance gains are even more substantial than those of the forward transforms in nearly all cases. The advantage over scalar-VTM can be primarily attributed to two factors. 1) The lack of partial butterfly optimization for dimensions four and eight in DST7/DCT8 transforms. 2) The column-major access pattern adopted for the inverse transforms is notably inefficient. This inefficiency stems from a worsened cache coherence and the fact that each value or coefficient access requires a multiplication to determine the correct index, thereby tripling the multiplications workload. The better speedup over AVX2-VVenC kernels in the inverse case can be explained by the VVenC approach of clamping the residual values in a distinct step. Conversely, the proposed kernels streamline this by converting values to 16 bits using the `pack` intrinsic, which simultaneously performs the clamping.

Table V presents the overall **in-encoder** speedups of the proposed kernels, averaged per QP across sequence classes. The speedup grows with higher QPs because of the increased share of transform coding tools. Though the complexity of transforms remains consistent across QPs, this is not the case with many other tools. Additionally, sequences of higher resolution attain more significant speed improvements due to the relatively smaller overhead of control logic. Excessive encoding times of larger resolutions stress the significance of our optimizations.

V. Conclusion

In this paper, we introduced novel AVX2 vectorized kernels tailored for VVC transform coding. The main novelty

TABLE V. IN-ENCODER SPEEDUP OF PROPOSED KERNELS

QP	22	27	32	37	Average
VVC-A1	1.05×	1.08×	1.12×	1.16×	**1.10×**
VVC-A2	1.04×	1.06×	1.10×	1.12×	**1.08×**
VVC-B	1.04×	1.06×	1.08×	1.11×	**1.07×**
VVC-C	1.04×	1.05×	1.06×	1.08×	**1.06×**
VVC-D	1.03×	1.04×	1.03×	1.07×	**1.04×**
VVC-E	1.05×	1.07×	1.07×	1.10×	**1.07×**
Average	**1.04×**	**1.06×**	**1.07×**	**1.10×**	**1.07×**

lies in specializing each transform according to its size, which also sets our approach apart from prevailing solutions. Experimental results show that the proposed forward transform kernels outpace the scalar implementation of VTM and the AVX2 implementation of VVenC by factors of 5.3× and 2.9×, respectively. Moreover, the inverse transforms achieve respective speedups of 8.2× and 3.2×. As a whole, using the proposed kernels accelerate the overall encoding speed of uvg266 by around 1.1×.

The future work could explore several areas for improvement. As AVX512 gains traction, there is potential to optimize our kernels for it too. Furthermore, implementing transforms with a dimension of 64 into uvg266 will trigger a need for their optimization. Finally, while the attempt to design vectorized kernels integrating the partial butterfly optimization was not fruitful, exploring alternative methodologies to reduce the operation count is still warranted.

REFERENCES

[1] 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 (accessed Aug. 28, 2023).

[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, pp. 1649–1668, Dec. 2012.

[3] A. Mercat *et al.*, "Comparative rate-distortion-complexity analysis of VVC and HEVC video codecs," *IEEE Access,* vol. 9, pp. 67813–67828, May 2021.

[4] P. Sjövall, A. Mercat, and J. Vanne, "FPGA-accelerated HEVC encoder for energy-efficient multi-access edge computing," in *Proc. Int. Conf. Image Process.* Kuala Lumpur, Malaysia, Oct. 2023.

[5] "VVC Reference Software Version 10.0," [Online]. Available: https://vcgit.hhi.fraunhofer.de/jvet/VVCSoftware_VTM/-/tree/VTM-10.0 (accessed Aug. 28, 2023).

[6] "Fraunhofer Versatile Video Encoder (VVenC)," [Online]. Available: https://github.com/fraunhoferhhi/vvenc (accessed Aug. 28, 2023).

[7] "uvg266," Ultra Video Group, [Online]. Available: https://github.com/ultravideo/uvg266 (accessed Aug. 28, 2023).

[8] J. Brandenburg, A. Wieckowski, T. Hinz, and B. Bross, "VVenC fraunhofer versatile video encoder v1.0.0," May 2021, [Online]. Available: https://www.hhi.fraunhofer.de/fileadmin/Departments/VCA/MC/VVC/vvenc-v1.0.0-v1.pdf (accessed Aug. 28, 2023).

[9] M. Viitanen, J. Sainio, A. Mercat, A. Lemmetti, and J. Vanne, "From HEVC to VVC: the first development steps of a practical intra video encoder," *IEEE Trans. Consum. Electron.,* vol. 68, no. 2, pp. 139–148, May 2022.

[10] "Steam Hardware & Software Survey: June 2023," Valve Inc., [Online]. Available: https://store.steampowered.com/hwsurvey (accessed Aug. 28, 2023).

[11] X. Zhao et al., "Transform coding in the VVC standard," *IEEE Trans. Circuits Syst. Video Technol.,* vol. 31, no. 10, pp. 3878–3890, Oct. 2021.

[12] J. Sainio, A. Mercat, and J. Vanne, "Design space exploration of practical VVC encoding for emerging media applications," *IEEE Trans. Consum. Electron.,* vol. 68, no. 4, pp. 387–400, Jul. 2022.

[13] M. Budagavi, A. Fuldseth, G. B. V. Sze, and M. Sadafale, "Core transform design in the High Efficiency Video Coding (HEVC) standard," *J. Sel. Topics Signal Process.,* vol. 7, no. 6, pp. 1029–1041, Dec. 2013.

[14] A. Lemmetti, A. Koivula, M. Viitanen, J. Vanne, and T. D. Hämäläinen, "AVX2-optimized Kvazaar HEVC intra encoder," in *Proc. Int. Conf. Image Process.*, Phoenix, AZ, USA, Sep. 2016.

[15] T. Fu, H. Zhang, F. Mu, and H. Chen, "Two-stage fast multiple transform selection algorithm for VVC intra coding," in *Proc. Int. Conf. Multimedia Expo*, Shanghai, China, Jul. 2019.

[16] L. He, S. Xiong, R. Yang, X. He, and H. Chen, "Low-complexity multiple transform selection combining multi-type tree partition algorithm for Versatile Video Coding," *Sensors,* vol. 22, no. 15, pp. 5523–5535, Jul. 2022.

[17] Z. Wang *et al.*, "A fast transform algorithm for VVC intra coding," in *Proc. Int. Conf. Commun., Circuits Syst.*, Singapore, Singapore, Jul. 2022.

[18] M. Saldanha, G. Sanchez, C. Marcon, and L. Agostini, "Fast transform decision scheme for VVC intra-frame prediction using decision trees," in *Proc. Int. Symp. Circuits Syst.*, Austin, TX, USA, May 2022.

[19] Z. Zhang *et al.*, "Fast DST-VII/DCT-VIII with dual implementation support for Versatile Video Coding," *Trans. Circuits Syst. Video Techn.,* vol. 31, no. 1, pp. 355–371, Jan. 2021.

[20] Z. Hao *et al.*, "A reconfigurable multiple transform selection architecture for VVC," *Trans. Very Large Scale Integ. (VLSI) Syst.,* vol. 31, no. 5, pp. 658–669, May 2023.

[21] J. Brandenburg, A. Wieckowski, A. Henkel, B. Bross, and D. Marpe, "Pareto-optimized coding configurations for VVenC, a fast and efficient VVC encoder," in *Proc. Int. Workshop Multimedia Signal Process.*, Tampere, Finland, Oct. 2021.

[22] "Intel Intrinsics Guide," Intel Corporation Inc., [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (accessed Aug. 28, 2023).

[23] F. Bossen, J. Boyce, K. Suehring, X. Li, and V. Seregin, "VTM common test conditions and software reference configurations for SDR video," *document JVET-T2010*, Teleconference, Oct. 2020.

[24] "Intel VTune performance analyzer," Intel Corporation, Inc., [Online]. Available: https://software.intel.com/content/www/us/en/develop/home.html (accessed Aug. 28, 2023).

[25] 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.