Low level optimizations of the Future Video Coding inverse transforms

Vincent GIRAUD[1]*

Abstract
The Video Coding Experts Group (VCEG) and the Moving Picture Experts Group (MPEG) are preparing a new video standard called Future Video Coding (FVC), to respond to the growing demands in terms of video consumption. Its goal is to provide the same subjective video quality as H.265/MPEG-4 HEVC, with a bitrate reduced by 50%. One of the new considered features is the Adaptive Multiple Transform (AMT) : it replaces the classic Discrete Cosine Transform (DCT) by a set of different transformations. For each Transform Block (TB), the transforms that will provide the best compression are chosen. This enhancement comes at the cost of a certain complexity. This paper provides low level, spatially parallel optimizations that work on all the considered transforms during video decoding.

Keywords
Optimization — Future Video Coding (FVC) — Inverse transform design — Prediction residues — Single Instruction on Multiple Data (SIMD)

[1] *Department of Electronics and Industrial Informatics, INSA Rennes, Rennes, France*
***Corresponding author**: vincent@giraud.site

Contents

Introduction 1
0.1 Context . . . 1
0.2 Contribution . . . 1
0.3 Plan of the paper . . . 2
1 Related works 2
1.1 H.265/MPEG-4 HEVC . . . 2
1.2 The Adaptive Multiple Transform (AMT) . . . 2
1.3 Single Instruction Multiple Data computing . . . 2
2 Proposed solutions 2
2.1 Exploitation of spatial parallelism . . . 2
2.2 Customized treatment for constant residues . . . 4
3 Results and discussion 5
3.1 Technical configuration . . . 5
3.2 Results . . . 5
4 Conclusion 5
References 5

Introduction

0.1 Context

During the 2000s, digital video mediums replaced the analog ones. Various video standards have succeeded one another since then, in order to provide better bitrate requirements for the same subjective quality. H.265/MPEG-4 High Efficiency Video Coding (HEVC) became the most popular one after its release in 2013. However, video consumption is changing to become more mobile, and to require greater spatial resolutions : 3840×2160 (4K) and 7680×4320 (8K) are becoming accessible to the general public. The High Dynamic Range (HDR) technology will need higher bit depths too.

In order to provide better compression, all current standards predict video frames via interpicture and intrapicture predictions. Each frame is divided into smaller blocks, which are deduced from the rest of the frame or from previous or futur pictures. As these techniques may give incorrect results, it is necessary to correct them by adding residual data, which is the difference between the input data and the result of the prediction. To avoid having significant amounts of residues in the end video files, they are being processed through mathematical transformations. In H.265/MPEG-4 HEVC, the Discrete Cosine Transform (DCT) is used[1] : it allows good compression with great energy compaction, and features numerous practical mathematical properties, like symmetry and having smaller transforms matrices contained in the bigger ones[2]. Most of the processed residual data consists of a two-dimensional vector with values concentrated in the lower frequencies.

0.2 Contribution

The Video Coding Experts Group (VCEG) and the Moving Picture Experts Group (MPEG), as the Joint Video Experts Team (JVET), are developing a video standard called Future Video Coding (FVC) to respond to the new needs in terms of video consumption. Its goal is to achieve a 50% reduction in bitrate for the same subjective quality. The DCT may not be the only transformation used, and thus it is necessary to develop optimizations other than the ones specific to it. The goal of this paper is to provide generic low level optimizations of inverse transforms. Mathematically speaking, forward and inverse transforms are applied in the

same way — meaning these techniques could be applied to any forward transform; however, only inverse transform matrices are precisely defined by video standards to avoid divergence because of rounding effects[3]. These definitions offer further opportunities that will be analyzed. Also, since H.265/MPEG-4 HEVC introduced a wide dynamic variation in the sizes of Transform Blocks (TBs) — now ranging from 4×4 to 32×32[4], these optimizations need to be potentially applicable to multiple inverse transform sizes. As videos are becoming more and more heavy, the main risk is to not be able to decode residues in real time, therefore, not being able to play a video in its original speed.

0.3 Plan of the paper

This paper is organized as follows. Section 1 exposes current means implemented to improve residual data processing. In section 2, several ways of refinement are proposed. Section 3 presents the effects of such techniques. Finally, conclusions are given in section 4.

1. Related works

1.1 H.265/MPEG-4 HEVC

In H.265/MPEG-4 HEVC, which is VCEG and MPEG's last video standard, the DCT is the sole application exploited for doing two-dimensional transformations — with the exception of a Discrete Sine Transform (DST) used for 4×4 intra blocks[1]. Thus, many specialized optimizations were elaborated for this particular operation. Hardware implementations have been made, notably by using spare matrix decompositions[5]. Algorithms using its specific properties have been developed[2]. These improvements helped the standard grow as it pushed efficiency further.

1.2 The Adaptive Multiple Transform (AMT)

In order to improve compression, a new approach to residues processing is considered in the FVC standard. Rather than using a single transform, multiple ones could be used. Every time a residual block needs to be processed, a FVC encoder would have to select between a set of transforms; with the objective of reducing both correlation and distortion. In the case of a single fixed transform, the DCT is confirmed as the optimal choice[6]; but the exploitation of a set of different transforms has been proven to be 5% more efficient for the same objective quality using the Bjøntegaard-Delta Bit-Rate (BD-BR) method[7]. This test has been made on the Joint Exploration Model (JEM) software, proposed by the JVET to measure new video tools' relevancy, with the transforms specified in figure 1. This dynamic way of processing residual data has been called the Adaptive Multiple Transform (AMT).

For each TB, the best transform to use is chosen by doing Rate–Distortion Optimization (RDO) which can rely on measures like the Peak Signal to Noise Ratio (PSNR) — therefore on the mean squared error —, as does the BD-BR method[9]. It should be noted that these indicators are purely objective and do not capture in any way the subjective fidelity. When choosing for a transform, there is the possibility of not applying the same for both dimensions, but instead using one in the horizontal direction, and another one vertically. All these new opportunities also come with its share of complexity.

DCT-II	$T_i(j) = \omega_0 \times \sqrt{\frac{2}{N}} \times cos(\frac{\pi \times i \times (2j+1)}{2N})$ where $\omega_0 = \begin{cases} \sqrt{\frac{2}{N}} & \text{if } i = 0 \\ 1 & \text{if } i \neq 0 \end{cases}$
DCT-V	$T_i(j) = \omega_0 \times \omega_1 \times \sqrt{\frac{2}{2N-1}} \times cos(\frac{2 \times \pi \times i \times j}{2N-1})$ where $\omega_0 = \begin{cases} \sqrt{\frac{2}{N}} & \text{if } i = 0 \\ 1 & \text{if } i \neq 0 \end{cases}$ and $\omega_1 = \begin{cases} \sqrt{\frac{2}{N}} & \text{if } j = 0 \\ 1 & \text{if } j \neq 0 \end{cases}$
DCT-VIII	$T_i(j) = \sqrt{\frac{4}{2N+1}} \times cos(\frac{\pi \times t(2i+1) \times (2j+1)}{4N+2})$
DST-I	$T_i(j) = \sqrt{\frac{2}{N+1}} \times sin(\frac{\pi \times (i+1) \times (j+1)}{N+1})$
DST-VII	$T_i(j) = \sqrt{\frac{4}{2N+1}} \times sin(\frac{\pi \times (2i+1) \times (j+1)}{2N+1})$

Figure 1. The five trigonometric transforms used in the JEM software to test the relevancy of the AMT[8]. The classic DCT is defined as the DCT-II. N represents the size of the vectors.

1.3 Single Instruction Multiple Data computing

A Single Instruction on Multiple Data (SIMD) architecture is a design that allows for spatial parallelism in the treatment of data. In classic computing, an instruction results in one value, usually after processing one or two. With a SIMD architecture however, an instruction can compute with vectors of data. Arithmetic operations are done element to element. Vector-processing is especially efficient with repetitive procedures.

Such designs have been implemented in mainstream processors since the late 1990s, in the form of instruction set extensions. On x86-based hardware, they notably include MultiMedia eXtensions (MMX), Streaming SIMD Extensions (SSE), 3DNow!, and Advanced Vector Extensions (AVX). These do not work with the same vector's size, ranging from 64 bits to 256 bits. However, there is not a single fixed manipulated values' bit depth : the programmer can choose between a few elements' possible sizes. In order to not restrict their use to assembly writers, some extensions come with a library of C functions that provide access to the instructions; they are called intrinsics.

2. Proposed solutions

Against this background, it is important to develop optimizations which work independently from the applied transform. The proposed solutions are FVC-friendly, as they can deal with all the mathematical applications integrated in it. It should be recalled that they all are linear maps, and thus can each be represented by a matrix, and executed via a matrix multiplication.

2.1 Exploitation of spatial parallelism

As mentioned earlier, SIMD processing is beneficial for repetitive tasks. A matrix multiplication can be presumed

as one : let m be the size of two square matrices. Multiplying them requires m^3 multiplications and $m^2 \times (m-1)$ additions, for a total of $m^2 \times (2m-1)$ operations. The implementation of parallel computing for the AMT is thus considered here. The SSE is chosen because of its wide implantation — it is available on all modern processors. It provides calculation on 128-bit vectors, with the elements' size ranging from 8 to 64 bits. In the case of residual data processing, using a bit depth of 32 bits — giving vectors containing 4 variables — is justified by the need to store large intermediate values. By convention, little-endian will be used in this paper when representing a vector.

Let B be the 4×4 residual block that needs to be compressed, Dc the matrix for the column transform and Dr the matrix for the row one. Applying a two-dimensional transformation is done by computing $(DcB)Dr^T$, that is :

$$\left(\begin{bmatrix} Dc_0 & Dc_1 & Dc_2 & Dc_3 \\ Dc_4 & Dc_5 & Dc_6 & Dc_7 \\ Dc_8 & Dc_9 & Dc_{10} & Dc_{11} \\ Dc_{12} & Dc_{13} & Dc_{14} & Dc_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix} \right) \times \begin{bmatrix} Dr_0 & Dr_4 & Dr_8 & Dr_{12} \\ Dr_1 & Dr_5 & Dr_9 & Dr_{13} \\ Dr_2 & Dr_6 & Dr_{10} & Dr_{14} \\ Dr_3 & Dr_7 & Dr_{11} & Dr_{15} \end{bmatrix}$$

A first algorithm is developed where the residues are loaded into vectors of size 4, and where for each matrix multiplication, a line is computed by accumulating the products of a line from the second matrix and a broadcast of the corresponding coefficient from the first one, as shown in figure 2.

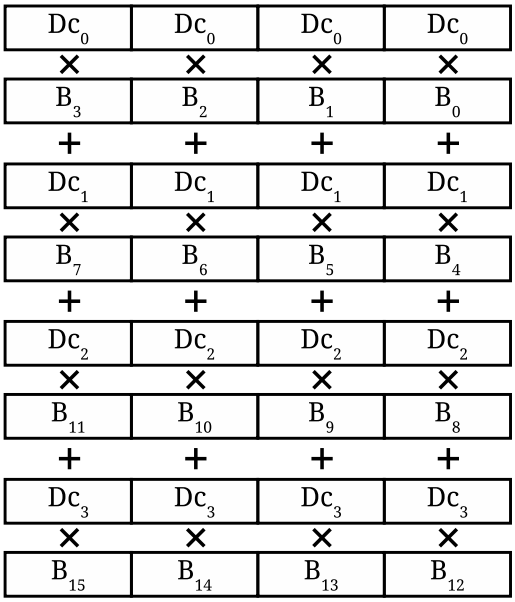

Figure 2. A first way of computing a line in a matrix multiplication.

With this algorithm, the two-dimensional transform needs 32 `pmaddwd` and 32 `paddd` instructions. The many loading instructions dedicated to the coefficient broadcasts can be disadvantageous. Also, it should be noted that the `pmaddwd` instruction is not fully exploited here : every time it is used, a multiplication and an addition are wasted; this is the case because all values need only 16 bits but are contained in 32 bits, thus the upper bits of each 32-bit elements are zeros. On the other hand, this procedure requires almost no vector reorganization : for both matrix multiplications, the coefficients from the first matrix are simply broadcasted one by one, and the values from the second one are loaded, then unpacked with `punpcklwd` and `punpckhwd` instructions to make them fit in 32-bit elements.

A second algorithm is considered. In this one, vectors are loaded only before matrix multiplications, but their elements get heavily reorganized. Indeed, in this method, the scalar product is used, thus the second matrix's vectors need to contain values along the columns; however they are loaded along the rows. To change this, four temporary vectors are generated by doing four dual unpacking with zeros — by using `punpcklwd` and `punpckhwd` —, as shown in figure 3.

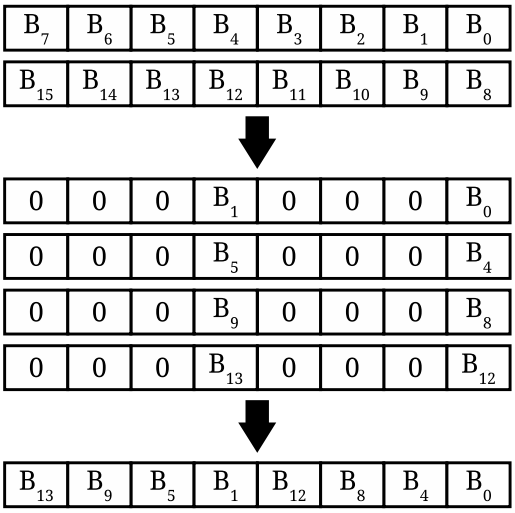

Figure 3. An illustration of a method for obtaining a vector along the columns instead of the rows. To get the second one, the exact same technique can be reapplied, but the two input vectors need to be shifted 4 bytes to the right.

The first one is obtained by executing a low then another low unpack on the first input vector. The second one by doing a high then a low unpack on the first input vector. The third one by executing a low then another low unpack on the second input vector. The fourth one by doing a high then a low unpack on the second input vector. Once this is done, the four temporary vectors have to be merged : the second, third, and fourth vectors will be shifted 2, 4, and 6 bytes to the left respectively with `pslldq` instructions, and all of them will be fused together with `paddw` instructions. This whole procedure works directly when looking for the first output vector, but in order to get the second one, a right shift of 4 bytes has to be done on both input vectors at the beginning with the `psrldq` instructions. Once the rearrangement is finished, the actual computation can start. The left matrix's coefficients are loaded in the form of registers storing two duplicates of their corresponding line — this does not need reorganization as before, because the inverse transform matrices are defined and stored in the program; thus any manipulation can be done directly in the source code. Those vectors will be multiplied and added to the recently made ones with `pmaddwd` and `phaddd` instructions, as seen in figure 4.

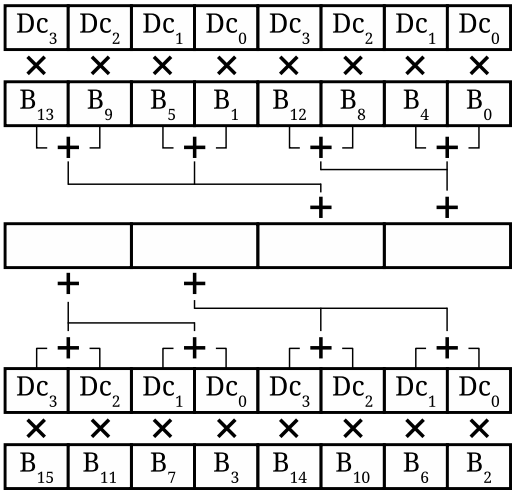

Figure 4. A second way of computing a line in a matrix multiplication. The register in the middle receives the output vector containing a line of the resulting matrix.

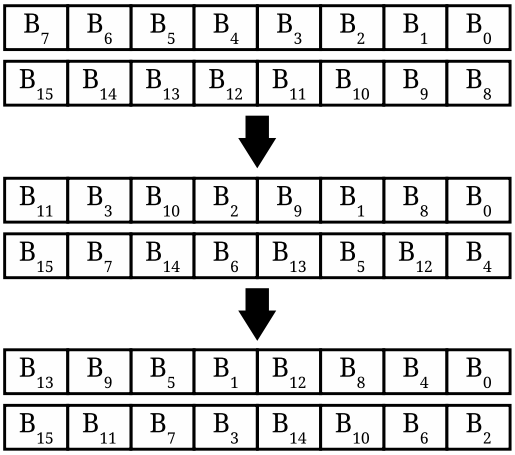

Figure 5. Illustration of another method for making vectors along the columns instead of the rows.

After this first transformation, the residual block still needs a second one. But this time it will act as the left matrix of this product; thus its vectors need a reorganization to adopt the same duplicated form that the first transform matrix had to adopt, as represented in figure 4. Each such vector is created by using a `movlhps` or a `movhlps` instruction on the two same packings — with `packssdw` — of either the two first lines or the two last lines. As an example, the second vector is made by using a `movhlps` instruction on the two same packings of the two first lines.

In total, this algorithm required only 16 `pmaddwd` and 8 `phaddd` instructions for the whole two-dimensional transform. However, this comes at the cost of heavy rearrangements of elements before matrix multiplications. Nevertheless, the `pmaddwd` instruction is fully exploited here.

Finally, a third method is proposed, really similar to the second one. The only change is the reorganization before the first matrix multiplication. As shown in figure 5, only two temporary vectors are required here. They are each obtained by executing a `punpcklwd` or a `punpckhwd` instruction on both inputs. Then, to get the output vectors, the exact same operation has to be applied on the temporary vectors. This slightly different way of proceeding might possibly accelerate the transform.

In every algorithm, between and after the two matrix multiplications, all coefficients in the residual block are shifted to the right with `psrad` instructions, in order to have them fit in the desired bit depth — by 6 and 13 bits respectively, for 8-bit residues[2]. But before doing a n-bit shift, the $n-1$ bits are always incremented by one with a `paddd` instruction : this provides rounding, as after the shift, the first bits — or the n ones — are incremented only if a rounding up is necessary. On a different matter, it should be noted that these optimizations can be used in the context of matrices bigger than 4×4 : 8×8, 16×16, and 32×32 matrix calculations will have to apply a divide and rule algorithm to obtain their product from a 4×4 multiplication. Also, regarding the instruction set extensions, it can be shown that the first presented method only need instructions from SSE and SSE2 — unlike the others, which need instructions from SSE3, too. Even if the first algorithm turns out to be less efficient, it could still be implemented as a mean to support SIMD optimizations on old systems; particularly on those equipped with processors from between 2001 and 2004.

2.2 Customized treatment for constant residues

During video coding, a significant number of TBs end up being constant, that is, all their values are equals. These blocks only contain a direct signal that cannot be defined by frequencies. When encoded using a classic DCT for example, the result is a matrix with only one non-zero coefficient : the top left one, which represent the DC signal. During the decoding, the TB will regain its initial, constant form. If no special matrix product is used, many multiplications with zeros will be done. The goal here is to avoid this, and instead generate an output block by only processing the DC value. The procedure is illustrated in figure 6. The only non-zero coefficient of the TB is broadcasted in the four 32-bit elements of a register. The latter is then multiplied with the first column of the column transform matrix, and the first row of the row transform matrix. It should be recalled that since the Dr matrix is transposed in the two-dimensional transformation formula, the coefficients contained in the first row are not Dr_0, Dr_1, Dr_2 and Dr_3; but Dr_0, Dr_4, Dr_8 and Dr_{12}.

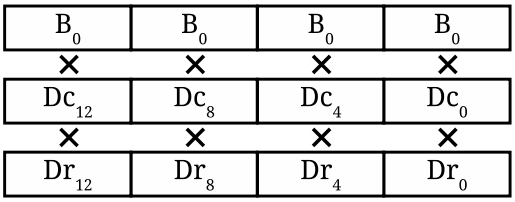

Figure 6. An optimized way for applying an inverse transform on a constant TB.

The resulting vector will contain the same four values. They have to be shifted to the right by 19 bits if the video's bit depth is 8, for fitting purposes as explained earlier; while also taking care of the rounding. All that then remains

is to pack this vector with itself by using the `packssdw` instruction : this will convert the 32-bit elements register to a 16-bit elements one, and duplicate the values in it. In the end, this method does not need reorganization, and uses only 2 `pmulld` instructions, which is a lot less than with the whole two-dimensional transform process.

3. Results and discussion

3.1 Technical configuration

To test the algorithms, a computer running GNU/Linux will be used. The kernel version is 4.9.92-1. The system's processor is an Intel Core i5-2410M working at 2,30 GHz. The testing software is not multithreaded, it runs on a single core. To measure the time taken to process, the `time` command is used, and the registered value is the user time. The software has been compiled with GNU Compiler Collection (GCC), using the `-O0` parameter to avoid unwanted behaviors, and `-msse`, `-msse2`, `-msse3`, `-msse4`, `-msse4.1`, `-msse4.2` to enable SSE instructions.

3.2 Results

Each method is executed on a different number of inverse transformations. To better put in context, an algorithm with absolutely no optimizations is also tested. It computes matrix multiplications by following the classic triple-loop pattern : one loop to scan the lines, another loop for selecting a value inside the selected line, and a third loop for executing the scalar product. It therefore does not benefit from spatial parallelism. Figure 7 presents the results of the experience.

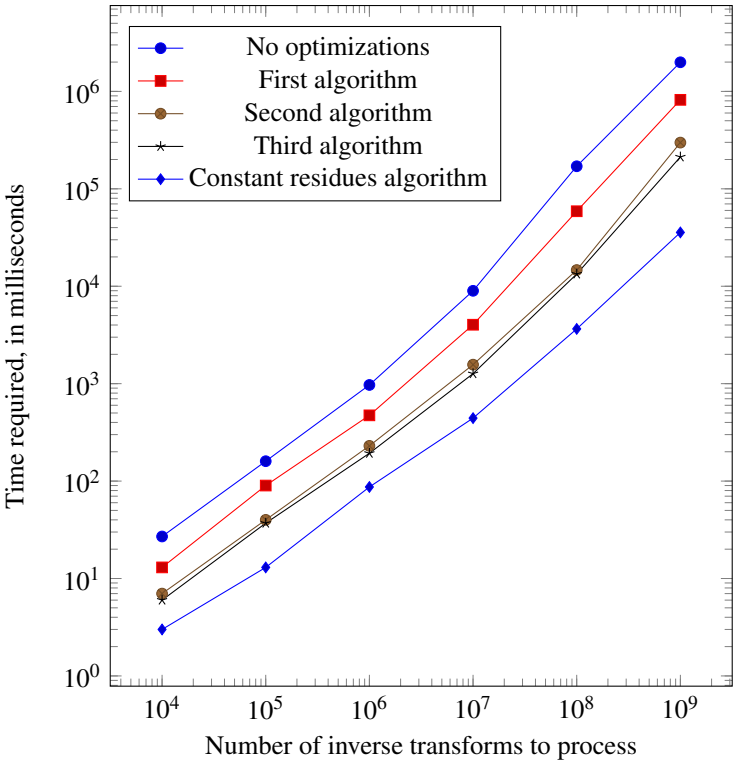

Figure 7. Times required for multiple 4×4 inverse transforms, depending on different optimizations. Both axis are logarithmic.

The non-optimized method is always and by far the slowest : this indicates that the use of SIMD instructions is effective. While the third algorithm is slightly faster than the second, they both are much more efficient than the first. This difference may be explained by the incomplete usage of `pmaddwd` instructions by the first method. Also, the third way might be a a little more advanced than the second one thanks to its more compact reorganization of vectors. The performance of the constant residues algorithm proves that it is definitely the method to choose when dealing with a constant TB. Figure 8 shows the average gain for each way of computing.

First algorithm	224 %
Second algorithm	601 %
Third algorithm	717 %
Constant residues algorithm	2588 %

Figure 8. Average gains in speed in comparison to the non-optimized method. These percentages are ratios : 100 % means it performed with neither improvement nor loss.

4. Conclusion

In order to achieve the goals set by the JVET for the FVC standard, it is important to make the most out of every opportunity in terms of compression. The 5 % gain provided by the AMT is precious, but it implies a more complex video coding or decoding. To address this problem, general optimizations have been provided in this paper. They allow for a faster processing of residues in videos. By relying on SSE, these improvements can be applied on a large number of systems, as this instruction set extension is widely installed in common computers. Spatial parallelism has proven to be much more efficient than normal computing, especially for constant residual data. The use of AVX instructions can be suggested to push efficiency even further, even though the considered transforms for the AMT will, without a doubt, offer many opportunities.

References

[1] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, Dec 2012.

[2] M. Budagavi, A. Fuldseth, G. Bjøntegaard, V. Sze, and M. Sadafale. Core transform design in the high efficiency video coding (hevc) standard. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):1029–1041, Dec 2013.

[3] F. Loras and J. Fournier. H.264/mpeg-4 avc, un nouveau standard de compression vidéo. Technical report, CORESA and France Télecom R&D, 2003.

[4] T. Nguyen, P. Helle, M. Winken, B. Bross, D. Marpe, H. Schwarz, and T. Wiegand. Transform coding techniques in hevc. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):978–989, Dec 2013.

[5] Chia-Wei Chang, Hao-Fan Hsu, Chih-Peng Fan, Chung-Bin Wu, and Robert Chen-Hao Chang. A fast algorithm-based cost-effective and hardware-efficient unified architecture design of 4×4, 8×8, 16×16, and 32×32 inverse core transforms for hevc. *Journal of Signal Processing Systems*, 82(1):69–89, Jan 2016.

[6] Pierrick Philippe, Thibaud Biatek, and Victorien Lorcy. Improvement of hevc inter-coding mode using multiple transforms, Aug 2017.

[7] Naty Sidaty, Wassim Hamidouche, Olivier Déforges, and Pierrick Philippe. Compression efficiency of the emerging video coding tools, Sep 2017.

[8] Ahmed Kammoun, Wassim Hamidouche, Fatma Belghith, Jean-François Nezan, and Nouri Masmoudi. A unified 2d hardware architecture of the future video coding adaptive multiple transforms on soc platform. *IEEE Transactions on Consumer Electronics*, 2018.

[9] Saurabh Puri, Sebastien Lasserre, and Patrick Le Callet. Cnn-based transform index prediction in multiple transforms framework to assist entropy coding, Aug 2017.