Proceedings of the Federated Conference on Computer Science and Information Systems pp. 607–611
DOI: 10.15439/2017F185
ISSN 2300-5963 ACSIS, Vol. 11

H.265 Inverse Transform FPGA implementation in Impulse C

Sławomir Cichoń
NOKIA
Krakow Technology Center
AGH University of Science and Technology,
Department of Automatics and Biomedical Engineering,
Email: slawomir.cichon@nokia.com

Marek Gorgoń
AGH University of Science and Technology,
Department of Automatics and Biomedical Engineering,
Mickiewicz Avenue 30,
30-059 Krakow, Poland
Email: mago@agh.edu.pl

***Abstract*—High Efficiency Video Coding (HEVC), a modern video compression standard, exceeds the predecessor H.264 in efficiency by 50%, but with cost of increased complexity. It is one of main research topics for FPGA engineers working on image compression algorithms. On the other hand high-level synthesis tools after few years of lower interest from the industry and academic research, started to gain more of it recently. This paper presents FPGA implementation of HEVC 2D Inverse DCT transform implemented on Xilinx Virtex-6 using Impulse C high level language. Achieved results exceed 1080p@30fps with relatively high FPGA clock frequency and moderate resource usage.**

I. Introduction

H.265 is the most recent video coding algorithm released by joint collaboration between ITU and ISO organizations [1], and also described in details in [2]. It is claimed that this compression is 50% better then its predecessor, H.264. Both mentioned video coding standards use finite precision approximation of Discrete Coding Transform to change from the spatial domain to frequency, however H.264 uses only transform block sizes 4x4 and 8x8. HEVC uses various, so called Transform Unit (TU) sizes, ranging from 4x4 to 32x32 pixels.

High level synthesis languages have gained focus in recent years both in academic and industry research. During last years, few such types of commercial and academic tools have been developed. Impulse C is one of languages which can be translated to HDL, and further synthesized. It allows also to partition the solution, to run it in the mixed software/hardware environment. HLS usage can significantly shorten development cycle, but with cost of FPGA resources and lower clock frequency achieved.

Most of important scientific journals published special issue editions focused entirely on H.265 implementations, both hardware and software, to mention [3] and [4]. The majority of those articles are dealing with encoding challenges. Some of them like [5] exploits Graphics Processing Units (GPUs) to accelerate the intra decoding procedure in HEVC decoder. Hardware partial implementations of H.265 in HLS are presented e.g., in [6] and [7] dealing with only part of the standard, which may imply overall challenges in implementing the entire HEVC encoding/decoding in FPGA. In general, number of published hardware implementations of HEVC decoder in FPGA (full or partial) is relatively large, but there is very small number of publications on H.265 decoders using high level languages. In this paper, authors would like to reference publication related to HEVC IDCT implementation using Xilinx Vivado HLS and compared with few other implementations [8].

This paper presents first known to authors, H.265 Inverse Discrete Cosine/Sine Transform hardware implementation in Impulse C language [9], and achieved results in terms of clock frequency, frame rate and resource usage in comparison with [10]. Solution was verified on hardware platform PICO M503 [11], equipped with Virtex-6 FPGA family. This paper consist of few sections. In the following subsection, Impulse C features have been very briefly described, and their influence on the resulting implementation performance have been discussed in later section. Next section presents basic informations about 2D IDCT. Later proposed hardware architecture is depicted, following with achieved results in comparison with other solutions. Conclusions are closing this paper.

A. Impulse C - high level language

High level synthesis is a set of tools able to translate algorithm description written in a high level language (mostly C/C++-based), to industry standard hardware description languages (HDL), like Verilog or VHDL, which then can be synthesized for the desired FPGA family. They provide also tools to analyse the parallelism of the generated code. HLS needs to also provide capabilities for the high-speed communication and synchronization between processing elements, to allow for the efficient algorithm decomposition into execution units running in parallel manner. One of the language from this group is Impulse C [9]. As the name suggests, it is ANSI C-based language, supporting almost all of its syntax, with addition of some library functions used for communication. Algorithm described in Impulse C can be decomposed into parallel processing units called processes. They can exchange data or/and synchronize between each other using few mechanisms, like streams, which allow for fast data exchange in FIFO-like manner. Signals allow to achieve synchronization between processes and pass single

32-bit data, similar to *rendez-vous* mechanism in real-time systems programming. Remaining synchronization methods are semaphores and shared memory. Impulse C compiler analyses the data dependency, and splits the processing into stages. In each stage all instructions without data dependency are scheduled to execute. This implies state machine implementation in generated HDL. Programmer has some influence on parallelization, using specific keywords, called pragmas, in the source code, that can, e.g. pipeline the execution of the loop, or unroll all instructions inside the loop, under some conditions. In Impulse C, developer can partition the project to split the execution between software and hardware. For some of FPGA development boards, Impulse C IDE provides also libraries and drivers, called Platform Support Package (PSP), which allow to build and run complete project in real hardware/software co-environment. Libraries provide communication mechanism, especially stream data flow, between software and hardware. In this way programmable devices can be used as a coprocessor, also this approach fits in the idea of FPGA-as-a-Service (FaaS), and latest Amazon AWS EC2 cloud [12] solution.

B. HEVC 2D transform description

HEVC defines finite precision approximation of 2-dimensional discrete cosine transform for Transform Unit sizes from 4x4 to 32x32 pixels [1] [2]. In addition, it specifies 4x4 Discrete Sine Transform approximation for use in intra-frame solutions. Similar approach was introduced in H.264. In earlier video coding standards, mathematical formula for calculating cosine transform was used, leading to different implementations, which resulted in mismatch between different codecs. Because of this reason, in newer coding standards, like H.264, VC-1, HEVC, a core, integer transform has been defined, suitable for fixed-point and hardware implementations. Scaling and inverse transform processes are specified in the normalization document, while the TU size and quantization factor are chosen by the encoder. The main purpose of the transform is to de-correlate input data, which in most cases are residual data calculated based on prediction. Inverse DCT coefficients were carefully investigated and analysed by working group defining the H.265 standard. It was decided to represent each matrix coefficient with 8-bit. To perform integer transform, scaling factor is used at the end of the process, which is a power of 2, to easily implement it as a right shift. DCT has several properties very useful in terms of usage in video coding algorithms, particularly:

- Orthogonality, which allows transform coefficient to be uncorrelated,
- Good energy compaction,
- Smaller IDCT size matrix, is a sub-sample of the higher TU size,
- All rows have equal norm.

Formula for calculating 2D IDCT is as follows:

$$Y = AXA^T \tag{1}$$

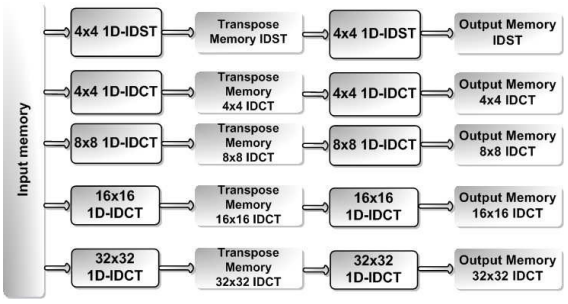

Fig. 1. HEVC 2D-IDCT proposed architecture.

It is known that 2D transform can be calculated in two 1D steps, with intermediate transpose memory. This decomposition is not standardized, but it is widely adopted in both software and hardware implementations to optimize calculations. This part of the decoding process is one of the most computationally expensive. Because of that, it is beneficial to realize it in the dedicated and optimized co-processor. Today's FPGAs are well positioned to serve such role.

II. Implementation

A. HEVC 2D IDCT Impulse C hardware implementation

In the presented Impulse C implementation of HEVC 2D IDCT/IDST, transform split into two 1D calculations with transpose memory was adopted, to pipeline the entire architecture. The implementation is based on HEVC reference software version HM-16.14 [13]. Figure 1 depicts proposed architecture. There is one single input memory for all transform sizes. All blocks performing 1D-IDCT/IDST reads from the same memory. Transpose memory has been split into 5 different memory blocks with size appropriate for the TU size. Split has been used to minimize critical path length. The same approach has been used for the output memory of the second stage of transform.

Figure 2 presents proposed architecture, while fig. 3 the actual source code for the 4x4 1D-IDCT module. Input memory has been split into 32 separate BRAM memories, each representing single row of 2D coefficient, to read the entire column simultaneously, and then copied into several sets of register arrays for multiple read in the same cycle later on. Similar approach has been applied to the transpose memory. Transform results are clipped to the range <-32767, 32768>, before written to the either Transpose Memory or Output Memory. All complex mathematical operations have been decomposed into simpler ones, to minimize critical path in resulting implementation. Presented solution has been split into software processes and hardware processing elements. Three software processes have been defined: *Producer*, which reads the input data from the file and sends it to the FPGA over PCIe bus. *Consumer* receives the result of inverse transform from the PICO M503 board, and stores the received data in the file for further verification/processing. *Stats* process receives data with processing duration (in clock cycles) of important hardware modules for every Transform Unit to

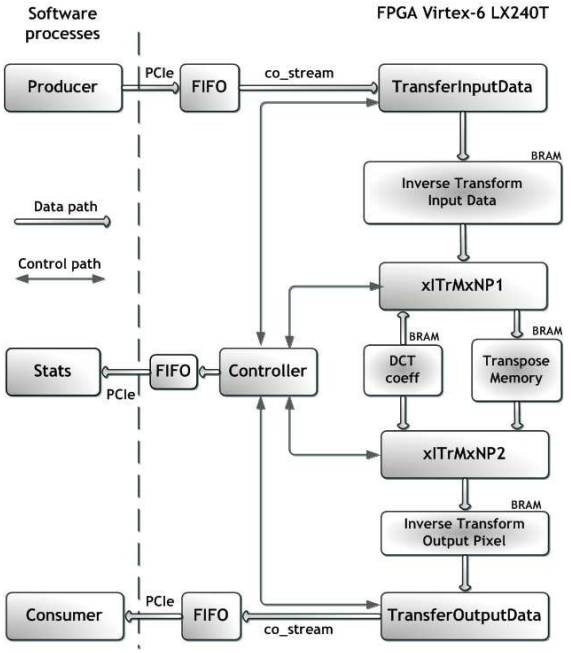

Fig. 2. HEVC Inverse Transform hardware/software architecture.

prove real-time performance of the implemented solution. On the FPGA side, five important processing elements (PEs) can be seen. xITrMxNP1 and xITrMxNP2 perform the actual 1D inverse transform in the pipeline manner. There are also modules to receive/send the data from/to the PCIe bus. Data between hardware processes are exchanged using BRAM memory. In n-th iteration one process (e.g. xITrMxNP1) writes the data to the one half of the memory, while the other (e.g. xITrMxNP2) reads from the other half. Controller process is responsible for all PEs synchronization, as well as collecting duration data and sending it over PCIe to the *Stats* software process. Presented solution is hardware/software co-design, but in FPGA only inverse transform is calculated. Software processes are responsible for data transfers.

B. Implementation improvement techniques

Regular C++ H.265 reference software [13] inverse transform implementation, can not be directly compiled using Impulse C, also achieving 30fps frame rate requirement is not guaranteed. In order to meet real-time requirements for the video decoder, several changes and improvements have been introduced into the code. They include:

- Code changes - pointer arithmetic, dynamic allocations, and C++ specific features removal,
- Impulse C specific pragmas,
- Code refactoring - assignment simplification, arrays split or/and duplication, loop unrolling and pipelining, additional function(s) and process(es) extraction, explicit clock boundaries.

In this paper some of Impulse C features, like loop unrolling and pipelining used in the presented implementation, will be

```
void partialButterflyInverse4Phase1(co_uint1 aMemoryIndex, co_int8 shift)
{
#pragma CO PRIMITIVE

    (… variable declarations …)

    {
    #pragma CO FLATTEN
        add = (shift > 0) ? (1<<(shift-1)) : 0;
        MemoryIndexToWrite = UADD1(aMemoryIndex,1);
        MemoryIndex = aMemoryIndex;
        shiftReg = shift;

        tempDCTCoeff[0] = g_aiT4_Row1[0];
        tempDCTCoeff[1] = g_aiT4_Row1_[1];

        tempDCTCoeff[2] = g_aiT4_Row3[0];
        tempDCTCoeff[3] = g_aiT4_Row3_[1];
    }

    for (j=0; j<FOUR; j++)
    {
    #pragma CO PIPELINE
    #pragma CO SET stageDelay 23
        tempCoeff[0] = g_TransferInverseTransformCoeffRow00[MemoryIndex][j];
        tempCoeff[1] = g_TransferInverseTransformCoeffRow01[MemoryIndex][j];
        tempCoeff[2] = g_TransferInverseTransformCoeffRow02[MemoryIndex][j];
        tempCoeff[3] = g_TransferInverseTransformCoeffRow03[MemoryIndex][j];
        co_par_break();
        for (unrollIter = 0; unrollIter < FOUR; unrollIter++)
        {
        #pragma CO UNROLL
            tempCoeffArray[unrollIter] = tempCoeff[unrollIter];
            tempCoeffArray2[unrollIter] = tempCoeff[unrollIter];
            tempCoeffArray3[unrollIter] = tempCoeff[unrollIter];
            tempCoeffArray4[unrollIter] = tempCoeff[unrollIter];
            tempCoeffArray5[unrollIter] = tempCoeff[unrollIter];
            tempCoeffArray6[unrollIter] = tempCoeff[unrollIter];
            tempCoeffArray7[unrollIter] = tempCoeff[unrollIter];
            tempCoeffArray8[unrollIter] = tempCoeff[unrollIter];
        }

        /* Utilizing symmetry properties to the maximum to minimize
           the number of multiplications */
        tempOk[0] = tempDCTCoeff[0] * tempCoeffArray[1];
        tempOk[1] = tempDCTCoeff[2] * tempCoeffArray2[3];
        tempOk[2] = tempDCTCoeff[1] * tempCoeffArray3[1];
        tempOk[3] = tempDCTCoeff[3] * tempCoeffArray4[3];

        tempOk[4] = 64      * tempCoeffArray5[0];
        tempOk[5] = 64      * tempCoeffArray6[2];
        tempOk[6] = 64      * tempCoeffArray7[0];
        tempOk[7] = 0 - (64 * tempCoeffArray8[2]);

        O[0] = tempOk[0] + tempOk[1];
        O[1] = tempOk[2] + tempOk[3];

        E[0] = tempOk[4] + tempOk[5];
        E[1] = tempOk[6] + tempOk[7];

        /* Combining even and odd terms at each hierarchy level
           to calculate the final spatial domain vector */
        tempPixelSum[0] = E[0] + O[0] + add;
        tempPixelSum[1] = E[1] + O[1] + add;
        tempPixelSum[2] = E[1] - O[1] + add;
        tempPixelSum[3] = E[0] - O[0] + add;

        tempPixel[0] = tempPixelSum[0] >> shiftReg;
        tempPixel[1] = tempPixelSum[1] >> shiftReg;
        tempPixel[2] = tempPixelSum[2] >> shiftReg;
        tempPixel[3] = tempPixelSum[3] >> shiftReg;

        tempPixelClipped[0] = Clip3(tempPixel[0]);
        tempPixelClipped[1] = Clip3(tempPixel[1]);
        tempPixelClipped[2] = Clip3(tempPixel[2]);
        tempPixelClipped[3] = Clip3(tempPixel[3]);

        switch (j)
        {
        case 0:
            tempResult2D[0][0] = tempPixelClipped[0];
            tempResult2D[0][1] = tempPixelClipped[1];
            tempResult2D[0][2] = tempPixelClipped[2];
            tempResult2D[0][3] = tempPixelClipped[3];
            break;
        case 1:
            tempResult2D[1][0] = tempPixelClipped[0];
            tempResult2D[1][1] = tempPixelClipped[1];
            tempResult2D[1][2] = tempPixelClipped[2];
            tempResult2D[1][3] = tempPixelClipped[3];
            break;
        case 2:
            tempResult2D[2][0] = tempPixelClipped[0];
            tempResult2D[2][1] = tempPixelClipped[1];
            tempResult2D[2][2] = tempPixelClipped[2];
            tempResult2D[2][3] = tempPixelClipped[3];
            break;
        case 3:
            tempResult2D[3][0] = tempPixelClipped[0];
            tempResult2D[3][1] = tempPixelClipped[1];
            tempResult2D[3][2] = tempPixelClipped[2];
            tempResult2D[3][3] = tempPixelClipped[3];
            break;
        default:
            break;
        }
    }

    g_TransposeMemory4Row00[MemoryIndexToWrite][0] = tempResult2D[0][0];
    g_TransposeMemory4Row00[MemoryIndexToWrite][1] = tempResult2D[0][1];
    g_TransposeMemory4Row00[MemoryIndexToWrite][2] = tempResult2D[0][2];
    g_TransposeMemory4Row00[MemoryIndexToWrite][3] = tempResult2D[0][3];

    g_TransposeMemory4Row01[MemoryIndexToWrite][0] = tempResult2D[1][0];
    g_TransposeMemory4Row01[MemoryIndexToWrite][1] = tempResult2D[1][1];
    g_TransposeMemory4Row01[MemoryIndexToWrite][2] = tempResult2D[1][2];
    g_TransposeMemory4Row01[MemoryIndexToWrite][3] = tempResult2D[1][3];

    g_TransposeMemory4Row02[MemoryIndexToWrite][0] = tempResult2D[2][0];
    g_TransposeMemory4Row02[MemoryIndexToWrite][1] = tempResult2D[2][1];
    g_TransposeMemory4Row02[MemoryIndexToWrite][2] = tempResult2D[2][2];
    g_TransposeMemory4Row02[MemoryIndexToWrite][3] = tempResult2D[2][3];

    g_TransposeMemory4Row03[MemoryIndexToWrite][0] = tempResult2D[3][0];
    g_TransposeMemory4Row03[MemoryIndexToWrite][1] = tempResult2D[3][1];
    g_TransposeMemory4Row03[MemoryIndexToWrite][2] = tempResult2D[3][2];
    g_TransposeMemory4Row03[MemoryIndexToWrite][3] = tempResult2D[3][3];
}
```

Fig. 3. HEVC 4x4 1D-IDCT Impulse C implementation - code snippet.

TABLE I
RESULTS COMPARISON WITH OTHER IMPLEMENTATIONS

Solution	LUT	DFF	Slices	BRAM	Freq	FullHD fps
Proposed	22457	31591	11985	31	200	39 (*)
[10] Vivado	50566	34955	14944	13	208	54
[8] Verilog	38790	11762	11343	32	150	48

TABLE II
PROCESSING DURATION FOR EACH TU SIZE

	Duration [clock cycles]	FullHD frame rate [fps]
2D-IDST	47	32.8
4x4 2D-IDCT	47	32.8
8x8 2D-IDCT	197	31.3
16x16 2D-IDCT	812	30.4
32x32 2D-IDCT	2541	38.6

described in more detail.

1) Loop unrolling: This technique is commonly used in FPGA development. It results in speedup of the loop calculation in cost of area. In Impulse C it can be forced using dedicated pragma (*#pragma CO UNROLL*). In order to benefit from it, data array must be scalarizable, which is possible under few conditions:

- Array scalarization option is enabled in the compiler,
- Array cannot be initialized where declared,
- Array elements are accessed with constant indexes,
- Array elements cannot be read and written in single C statement,
- Loop index must by of type *int*.

In the provided code snippet in fig.3, for 4x4 1D-IDCT butterfly calculation, loop unrolling was applied. This allows to save at least 16 clock cycles per each TU (4 iterations of the inner loop * 4 iterations of the outer loop) in comparison with the implementation without it. The purpose of the outer loop is to duplicate input data, in order to access them in parallel for the transform calculation, and minimize the fanout from the *tempCoeff* array in the resulting netlist.

2) Loop pipelining: Pipelined architecture is often very effective, however not all types of algorithms can be executed in such way efficiently. Inverse transform definition fits into this architecture. In impulse C loop pipelining must be called explicitly with the special C pragma (*#pragma CO PIPELINE*). Once compiled, it can be verified with Stage Master Explorer, what is the rate and the latency of the pipeline. *Rate* equals number of cycles required to complete single loop iteration, also determines how often pipeline can consume input data. So the goal is to reach rate equal 1. *Latency* is the number of cycles required for an input data to reach its output, it is also the pipeline length. The goal here is to have it as smallest as possible, especially for loops with small number of iterations. Pragma *CO SET stageDelay*, defines maximum number of combinatorial gate delays for single stage, and it is roughly equal to the gate delay in the target hardware. To achieve rate optimal value, input data array for 1D-IDCT has been split into arrays representing each row. This allows to access the entire column of TU simultaneously. Also intermediate data arrays have been implemented and used in a way to sclarize them by the compiler, as described previously. Additional intermediate arrays have also been defined to break the critical path, however with cost of higher latency. Also each call to Clip3 method inferred separate logic in order to make those calculations simultaneous.

C. Results and comparison with other implementations

Results of the proposed implementation written in Impulse C have been compared with results presented in [10], especially for Vivado HLS implementation which seems to be the most comparable. Presented solution uses multipliers realized in DSP blocks with exception to multiplication by 64, which is replaced by left shift operation. Table I contains comparison results.

Full HD fps has been approximated based on input data containing 58k TUs, calculated by the reference encoder [13], for the 3840x2160 frame resolution. So for the purpose of results comparison, estimated number of Transform Units in Full HD frame equals 14.5k. Based on processing time gathered in runtime, average inverse transform duration for the first 14.5k TUs equals Tavg = 357 [clock cycles]. Achieved results are comparable to the Vivado HLS solution in terms of clock frequency, number of Slices and flip-flops used. Proposed solution uses 50% LUTs than HLS implementation presented in [10]. However the frame rate is significantly lower, but still exceeds 1080p@30fps resolution. In the currently discussed architecture, mechanism to gather duration data of all important PEs has been implemented. Complete data contains Table II. It can be seen that the most time consuming is 32x32 TU size, which is intuitive. On the other hand frame rate achieved with only this type of TUs is the highest one, as the number of such units within the video frame is smaller.

III. CONCLUSIONS

In this paper, first known to authors, 2D-IDCT HEVC hardware implementation using Impulse C has been presented, with additional software processes for data transfer and profiling. Achieved results are compared with Vivado HLS solution proposed in [10], and are better in terms of resources used, but worse in terms of frame rate. Using HLS tools can greatly speedup implementation process, minimizing number of errors, as the same C testbench can be used later in HDL simulation and hardware functional verification. Future work can include critical path minimization or/and area optimization of the implementation to achieve better frame rate, even 4K real-time requirements, however this may require newer FPGA family, e.g. Xilinx UltraScale/UltraScale+ with higher speed grade. The other direction could be to include all intra-decoder parts as either software or hardware processes, and gradually move them to the FPGA.

ACKNOWLEDGMENT

Authors would like to thank Impulse Accelerated Technologies (http://www.impulseaccelerated.com) for providing software license for the CoDeveloper Integrated Development Environment.

The work was supported by the AGH-UST grant 11.11.120.612.

REFERENCES

[1] High Efficiency Video Coding ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC), ITU-T and ISO/IEC, Apr. 2013.

[2] Sze V., Budagavi M., Sullivan G.J., *High Efficiency Video Coding (HEVC) - Algorithms and Architectures*, Springer, Switzerland; 2014, https://dx.doi.org/10.1007/978-3-319-06895-4.

[3] Sousa L., Roma N., "Special Issue on Real-time Energy-aware Circuits and Systems for HEVC and for Its 3D and SVC Extensions," Journal of Real-Time Image Processing, vol. 13, Mar. 2017, https://doi.org/10.1007/s11554-017-0675-6.

[4] Kim, B., Psannis, K. and Jun, D., "Special Issue on Architectures and Algorithms of High-efficiency Video Coding (HEVC) Standard for Real-time Video Applications," Journal of Real-Time Image Processing, vol. 12, Aug. 2016, http://dx.doi.org/10.1007/s11554-016-0595-x.

[5] de Souza, D.F., Ilic, A., Roma, N. et al., 'GPU-assisted HEVC Intra Decoder," Journal of Real-Time Image Processing, vol. 12, Aug. 2016, http://dx.doi.org/10.1007/s11554-015-0519-1.

[6] Sjövall P., Virtanen J., Vanne J., Hämäläinen T. D., "High-Level Synthesis Design Flow for HEVC Intra Encoder on SoC-FPGA," *2015 Euromicro Conference on Digital System Design,* 2015, http://dx.doi.org/10.1109/DSD.2015.67

[7] Kalali E., Hamzaoglu I., "FPGA Implementation of HEVC Intra Prediction Using High-Level Synthesis," *IEEE International Conference on Consumer Electronics ICCE, Berlin,* 2016, https://doi.org/10.1109/ICCE-Berlin.2016.7684745

[8] Kalali E., Ozcan E., Yalcinkaya O. M., Hamzaoglu I., "A low energy HEVC inverse transform hardware," IEEE Transactions on Consumer Electronics, vol. 60, no.4, pp. 754-761, Nov. 2014, https://doi.org/10.1109/TCE.2014.7027352.

[9] Pellerin D., Thibault S., *Practical FPGA Programming in C*, Prentice Hall; 2005.

[10] Kalali E., Hamzaoglu I., "FPGA Implementations of HEVC Inverse DCT Using High-Level Synthesis," *Conference on Design and Architectures for Signal and Image Processing (DASIP),* 2015, https://doi.org/10.1109/DASIP.2015.7367262.

[11] PICO M503 webpage: http://picocomputing.com/products/hpc-modules/m-503/

[12] Amazon EC2 F1 Instances: https://aws.amazon.com/ec2/instance-types/f1/

[13] HM Software Repository: https://hevc.hhi.fraunhofer.de/