Encoding High-Throughput Jpeg2000 (Htj2k) Images on A Gpu

Author:
Naman, AT; Taubman, D

Publication details:
Proceedings - International Conference on Image Processing, ICIP
v. 2020-October
pp. 1171 - 1175
9781728163956 (ISBN)
1522-4880 (ISSN)

Event details:
2020 IEEE International Conference on Image Processing (ICIP)
ELECTR NETWORK
2020-09-25 - 2020-09-28

Publication Date:
2020-10-01

Publisher DOI:
https://doi.org/10.1109/ICIP40778.2020.9190899

ENCODING HIGH-THROUGHPUT JPEG2000 (HTJ2K) IMAGES ON A GPU

Aous Thabit Naman and David Taubman

School of Electrical Engineering and Telecommunications,
The University of New South Wales (UNSW), Sydney, Australia

ABSTRACT

High-Throughput JPEG2000 (HTJ2K) is a new addition to the JPEG2000 suite of coding tools; it has been recently approved as Part-15 of the JPEG2000 standard, and the JPH file extension has been designated for it. The HTJ2K employs a new "fast" block coder that can achieve higher encoding and decoding throughput than a conventional JPEG2000 (C-J2K) encoder. The higher throughput is achieved because the HTJ2K codec processes wavelet coefficients in a smaller number of steps than C-J2K. Moreover, the HTJ2K block coder is also more amenable to parallelizable high-speed software and hardware implementations. The HTJ2K retains most of the features and capabilities of JPEG2000, and it also supports lossless transcoding between HTJ2K and already compressed C-J2K images. Quality scalability however is more limited than C-J2K. In a recent work, we presented preliminary performance results for decoding HTJ2K images on a GPU. In this work, we present a GPU-based HTJ2K encoder; we also present early encoding results for such an encoder, showing that it is possible to encode 4K 4:4:4 HDR videos at more than 70 frames per second (fps) on a low-end card, while a high-end GPU can encode such videos at more than 400 fps.

Index Terms— JPEG 2000, JPH, Image coding, Graphical processing unit, Low complexity

1. INTRODUCTION

Due to its features, flexibility, and efficiency, the JPEG2000 image compression format has proven to be very successful for certain applications; these include digital cinema, medical imaging, remote sensing, and broadcast content mastering. To appreciate HTJ2K, we need to delve more into the design of C-J2K [1]. The C-J2K pipeline is composed of three main stages: color transform, wavelet transform, and block coding (or entropy coding). The color transform attempts to represent color images in a color space that is more amenable to compression from the human visual system point of view. Then, the wavelet transform decomposes each color component into a set of subbands, exploiting spatial redundancy. Each subband is then quantized and partitioned into a set of disjoint rectangular or square blocks, known as codeblocks. Each codeblock is then independently encoded by the block coder.

In C-J2K, the most computationally demanding stage is the block coder. The C-J2K block coder is a fractional bitplane adaptive arithmetic coder. Each bitplane undergoes three coding passes; these are known as the significance propagation pass (SPP), the magnitude refinement pass (MRP), and the cleanup pass (CUP). Details of these passes can be found in [1], [2]. These passes are shown in Fig. 1. The computational complexity arises from visiting (or processing) each wavelet coefficient in multiple passes; additionally, the arithmetic coder is a serial machine, making parallelization hard if not impossible.

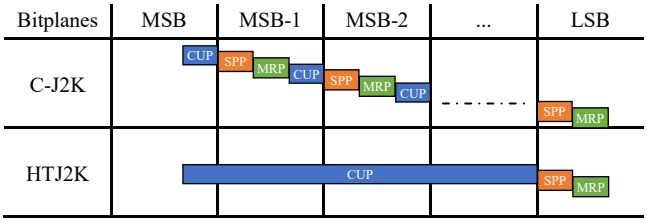


Fig. 1. Coding passes in C-J2K and HTJ2K.

To improve encoding or decoding speed, a C-J2K encoder can use the "BYPASS" mode, whereby data generated by the SPP and MRP passes are stored as is, without entropy coding; this approach improves speed but reduces the coding efficiency by a small amount. Another approach is to employ multiple processing cores; this is possible because codeblocks are encoded independently of each other.

The HTJ2K [3] introduces a new block coder that is an order of magnitude faster than C-J2K block coder for lossy compression, and even more for lossless coding. This improvement in speed comes with a small reduction in coding efficiency. The reader is referred to [4] for speedup and coding efficiency results. The HTJ2K block coder speedup is achieved using a newly defined CUP that encodes multiple bitplanes in one pass. Importantly, the design of the HTJ2K coder permits practical software and hardware designs that can exploit parallel processing techniques within a codeblock.

HTJ2K supports most of the features of C-J2K, except for quality scalability where limited support is provided. For many applications, resolution scalability can serve as a proxy for quality scalability. HTJ2K also supports lossless transcoding between HTJ2K and already compressed C-J2K codestreams. This enables compute constrained systems, for example, to employ HTJ2K image compression when they are capturing images; these images can then be transcoded to C-J2K for archival or dissemination over the Internet using JPIP [5].

The earliest implementation of C-J2K on a GPU is perhaps by Fürst et al. [6]. Nowadays, commercial GPU implementations are available from Comprimato [7] and Fastvideo [8]. To get more speed from a GPU, some researchers chose to modify the block

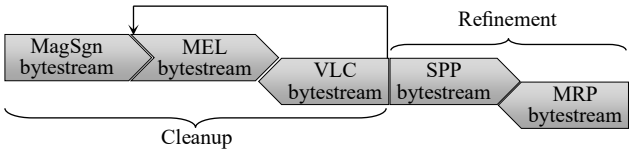


Fig. 2. The segments of a HTJ2K codestream. The last two bytes of the cleanup pass contain a pointer to the start of the MEL segment.

coder [9], [10], and the entropy coder itself [9], breaking compatibility with C-J2K. By contrast, HTJ2K deliberately adopts a coding pass strategy that enables lossless transcoding between HTJ2K and C-J2K.

For fast image coding, the ISO JTC1/SC29/WG1 working group has recently developed the JPEG-XS standard [11], to server as a mezzanine-level image coder for video transmission over managed media networks. The main target applications of JPEG-XS is hardware infrastructure in a studio. While we compared GPU decoding performance of JPEG-XS with HTJ2K in [12], encoding performance is not available.

In a previous work [12], we presented results for HTJ2K decoding on a GPU, which show that it is possible to decode hundreds of 4K 4:4:4 HDR video frames per second on the high-end GTX1080 GPU. In this work, we present a GPU-based HTJ2K encoder.

The rest of this paper is organized as follows. In Section 2, we introduce the HTJ2K codestream. In Section 3, we detail our GPU-based HTJ2K design, and in Section 4, we our present experimental results. Section 5 states our conclusions.

2. OVERVIEW OF HTJ2K CODESTREAM

2.1. HTJ2K codestream segments

The HTJ2K codestream usually has 1, 2, or 3 coding passes, as depicted in Fig. 1; these are a CUP, followed by an optional SPP, which can be followed by an optional MRP. The option to include SPP and MRP in the HTJ2K codestream facilitates transcoding between HTJ2K and C-J2K, and also provides more potential termination points in the codestream for the post-compression rate-distortion optimization (pcRDO) step.

Fig. 2 shows the segments of a HTJ2K codestream. The CUP is composed of a magnitude-sign (MagSgn) segment that grows forward, a MEL segment that grows forward, and a VLC segment that grows backward. The last two bytes in the CUP contain a pointer to the start of the MEL segment. For convenience, we refer to the SPP, which grows forward, and the MRP, which grows backward, together as the refinement. The lengths for the CUP and the refinement are communicated in their precinct's header. The structure of the HTJ2K codestream with its many identifiable entry points enables a codec to choose the order in which it processes these segments, and also provides multiple opportunities for parallelism. This work does not employ the SPP and MRP, and therefore we will pay less attention to them in the rest of this work.

2.2. HTJ2K basic coding structure

The HTJ2K block coder processes quantized wavelet coefficients in 2x2 blocks, known as quads; these quads are visited in a raster order as shown in Fig. 3. A codeblock of width W and height H contains

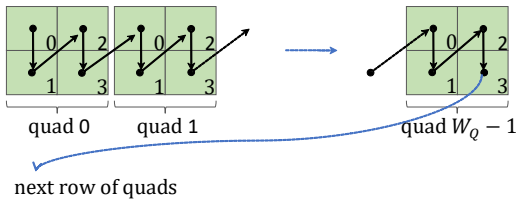


Fig. 3. HTJ2K encoder processes samples in quads. The bottom-right corner of each sample shows the subindex j.

$W_Q \times H_Q$ quads, where $W_Q = \lceil W/2 \rceil$ and $H_Q = \lceil H/2 \rceil$. We write Q_q for a quad, where $q = 0,1,\ldots, W_Q \cdot H_Q - 1$. We also write μ_n for the magnitude of a quantized wavelet coefficient x_n, and $s_n \in \{0,1\}$ for its sign. The subscript n is equal to $4q + j$, where j is a quad subindex as shown in Fig. 3.

The coding efficiency in HTJ2K comes from efficiently coding the locations of significant coefficients (i.e., those for which $\mu_n > 0$) and the number of bits needed to represent them. This information is communicated mainly by the VLC segment while the MEL segment plays a complementary role as explained in Section 2.4. The values of significant coefficients are packed in the MagSgn segment without coding.

HTJ2K codecs communicate significant coefficient locations and the number of bits needed on a per quad basis. Here, we write $\sigma_n \in \{0,1\}$ for the significance state of a coefficient x_n, and E_n for its exponent, which is equal to $\lceil \log_2 \mu_n \rceil + 1$ when $\mu_n > 0$, and 0 otherwise. The value $E_n - 1$ is the minimum number of bits needed to represent the value $\mu_n - 1$. Then, for a quad Q_q, significant coefficient locations ρ_q is given by

$$\rho_q = \sigma_{4q} + 2 \cdot \sigma_{4q+1} + 4 \cdot \sigma_{4q+2} + 8 \cdot \sigma_{4q+3}$$

and the maximum exponent is denoted by

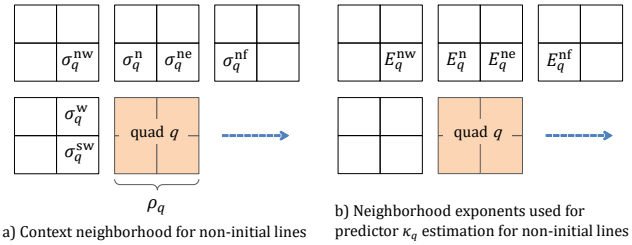
$$E_q^{max} = \max\{E_{4q}, E_{4q+1}, E_{4q+2}, E_{4q+3}\}$$

2.3. The VLC segment

The VLC segment carries context-adaptive Huffman codewords CxVLC interleaved with offsets u_q. The VLC context c_q for a quad is shown in Fig. 4a for non-initial rows of quads; the first row of quads uses a different context as detailed in [3]. Decoding a CxVLC codeword for a quad Q_q reveals the significant locations ρ_q for that quad and a bit, known as the u_q^{off} bit, that indicates the existence of u_q. Additionally, it reveals EMB patterns ϵ_q^{k} and ϵ_q^1; these patterns can signal all, some, or none of the significant coefficients that have their most significant bit (MSB) set. The existence of EMB patterns improves coding efficiency slightly by reducing the number of bits that needs to be communicated in the MagSgn segment at the expense of longer CxVLC codewords.

The maximum exponent E_q^{max} is not communicated directly, but rather, a predict and increase strategy is employed to obtain an upper bound U_q for it. Here, a predictor κ_q is generated. For a quad Q_q, this predictor is obtained from coefficient exponents E_n in the previous row of coefficients, as shown in Fig. 4b, while a value of 1 is used for the initial (or first) row of quads. Then, the upper bound $U_q = \max\{E_q^{max}, \kappa_q\}$ is obtained using

$$U_q = \begin{cases} u_q + \kappa_q, & u_q^{\text{off}} = 1 \\ \kappa_q, & u_q^{\text{off}} = 0 \end{cases}$$



a) Context neighborhood for non-initial lines

b) Neighborhood exponents used for predictor κ_q estimation for non-initial lines

Fig. 4. Left: Context neighborhood for non-initial row of quads. Right: Neighborhood exponents used for estimating predictor κ_q for non-initial line of quads.

Table 1. PCI-Express achievable download speed expressed in frames per second; the PCIe protocol has around 25% overhead, whereby x16 PCI 3.0 can only achieve around 12GB/s of transfer bandwidth. Images with 12 bits/sample use 2 bytes/sample. PCIe interface is bidirectional; these numbers are for each direction.

Resolution	Previous Gen. x16 PCIe 2.0	Current Gen. x16 PCIe 3.0	This Year's Gen. x16 PCIe 4.0
4K 4:2:2 8b	361.7 fps	723.4 fps	1446.8 fps
4K 4:4:4 8b	241.1 fps	482.3 fps	964.5 fps
4K 4:2:2 12b	180.8 fps	361.7 fps	723.4 fps
4K 4:4:4 12b	120.6 fps	241.1 fps	482.3 fps
8K 4:4:4 12b	30.1 fps	60.3 fps	120.6 fps

The number of bits that are communicated in the MagSgn segment for a coefficient x_n within quad Q_q is U_q, except when the EMB patterns reduce this number by one. Not using coefficient exponents E_n of coefficients to the left of a quad for evaluating U_q enables parallel processing of complete rows of quads.

2.4. The MEL segment

The MEL coder is an adaptive run length coder that can efficiently code runs of 0 events. The MEL coder in JPEG-LS [13], which produces the MELCODE, has 32 states while in the HTJ2K it has 13 states, which helps reduce complexity. Each state k of the MEL coder is associated with an exponent e_k, and a threshold $\tau_k = 2^{e_k}$. A codeword of "0" signals a run of τ_k 0 events, while a codeword of "1" indicates a run of 0 events terminated with a 1 event; the "1" codeword is followed by e_k bits of data specifying the number of 0 events before the terminating 1 event. Each run, complete or terminated, adapts the state of the MEL coder.

The MEL code facilitates efficient coding of runs of all-zero quads, which occur in codeblocks in which most coefficients are quantized to zero. When the VLC context c_q of quad Q_q is zero (all the neighborhood coefficients shown in Fig. 4a are zero), a 0 event in the MEL code indicates that quad Q_q is a quad with no significant coefficients; alternatively, a 1 event indicates that it has significant coefficients and the CxVLC needs to be decoded.

2.5. The MagSgn segment

For each significant coefficient, the MagSgn segment stores that coefficient's value μ_n followed by its sign s_n; however, EMB patterns can carry the MSB for some of these coefficients and therefore there is no need to store them in this segment. All segments undergo a bit stuffing step that prevents byte values larger than hexadecimal 8F from appearing after a byte value of FF.

3. GPU-BASED HTJ2K ENCODER

3.1. HTJ2K encoder setup

Only color images of 3840x2160 pixel resolution are explored in this work. For such an image, we expect to have 6321 codeblocks of 64x64 pixels when 5 levels of decomposition are employed. The host (or CPU) is responsible of creating lists of codeblocks that needs to be encoded; these lists include the address of wavelet coefficients, strides, block dimensions, … etc.

The GPU-based HTJ2K encoder can be used in many scenarios. In one scenario, the host uploads original, uncompressed images to the device (or GPU), which generates encoded codeblocks and makes them available to the host. The host downloads these codeblocks, packages them into proper HTJ2K files and save them to disk. In another scenario, the host uploads compressed HTJ2K to the GPU; the GPU would then decode these images [12], apply any desired processing to them, and then encode back to HTJ2K. The host would then package them into proper HTJ2K and save them back to disk.

For the first scenario, the bottleneck for a high-end GPU is currently the bandwidth of the PCIe interface that connects the GPU to the motherboard of the host; this is the most common way of interfacing a GPU to a host. Table 1 lists the maximum number of frames per second that can be transferred to a GPU. The PCIe interface is bidirectional, and the stated numbers are for each direction. The most commonly available PCIe interface is version 3.0. Latest offerings from AMD (CPUs and motherboards) support version 4.0, but nVidia cards are yet to support this interface. We expect to see some support for PCIe version 5.0 near the end of 2021, which doubles bandwidth yet again. From this, it can be seen that the PCIe bottleneck issue should be alleviated in a couple of years' time. For the second scenario, this is not an issue.

3.2. Color transform and wavelet analysis kernels

While it is possible to write a kernel to perform color transform only, such a kernel would waste the GPU's memory bandwidth. In this work, as in [12], color transformation is performed by the first-level DWT analysis kernel, which uses 113 registers when the irreversible CDF97 wavelet is employed. Subsequent DWT decompositions do not need a color transform, and therefore employ a simpler kernel which uses 56 registers. We refer to the color and the DWT analysis kernels collectively by KCT+DWT. These kernels also quantize the wavelet coefficients and put them in the sign-magnitude format.

For the wavelet analysis, we employ the approach proposed in [14]; each thread in a warp processes two columns, and the number of rows processed by a kernel invocation is user configurable. Here, each invocation produces 64 rows of wavelet coefficients.

The KCT+DWT kernels are memory-bound when wavelet coefficients are represented as 32-bit numbers; in this work, we use 32-bit floats for coefficients awaiting transformation and 32-bit integers for quantized coefficients. The use of 32-bit representation is more than what is needed for 12-bit image data; in fact, with 5 levels of decomposition, 16 bits should be enough.

3.3. Cleanup pass kernels

Due to the flexible design of the HTJ2K cleanup pass, a practical

Table 2. The GPU cards used in this work. †compute capability.

Card	CUDA Cores	Boost Clock (MHz)	Mem. BW (GB/s)	Attainable Mem. BW (GB/s)	PCIe 3.0 Lanes	CC†
GT1030	384	1468	48	~40	x4	6.1
GTX1660Ti	1536	1845	288	~240	x16	7.5
GTX1080	2560	1847	320	~240	x16	6.1

encoder can choose among many possible approaches. Here, we choose to implement the cleanup pass in 4 kernels; we refer to these kernels as KMagSgn, KVLC, KMEL, KVCPY.

The KMagSgn kernel reads quantized wavelet coefficients and processes them. Each thread in a warp processes 2 columns. If the nominal width of a codeblock is smaller than 64 columns, then a single warp handles multiple codeblocks; although this is not ideal because the optimal program flow for different codeblocks can be different, this achieves better utilization of hardware. This kernel uses 64 registers and around one byte of shared memory per 2 columns for the context information shown in Fig. 4.

The KMagSgn kernel generates the bitstuffed MagSgn segment together with its length and store them in the GPU's global memory. It also generates and stores state information needed by the subsequent KVLC kernel; these include CxVLC Huffman codewords, offsets u_q, and which quads have no significant coefficients.

The KVLC kernel reads the state information generated by the KMagSgn kernel, and generates the bitstuffed VLC segment, storing it in a separate global GPU memory buffer, together with its length. This kernel also packs MEL events into a contiguous stream of bits, storing them in global memory as well. This kernel uses 40 registers.

The MEL kernel reads packed MEL events and generates a bitstuffed MEL segment. Since MEL coding is a serial operation, each thread in a warp operates on MEL events from one codeblock. The MEL kernel stores the generated MEL segment at the end of the MagSgn segment. This kernel uses 30 registers.

The KVCPY kernel copies the VLC segment to the end of the MEL segment, potentially overlapping the terminating bytes of these two segments [3]. It also generates and store the pointer at the end of the VLC segment that points to the start of the MEL segment, which is shown in Fig. 2 and explained in Section 2.1. This kernel uses 26 registers.

Due to time constraint, as of this writing, we do not have a mechanism to detect codeblocks with no significant coefficients (zero codeblocks); such a mechanism can reduce execution time for all kernels. Despite this, Section 4 shows that the design can still encode 100s of frames a second.

4. EXPERIMENTAL RESULTS

Experimental results are obtained using the nVidia cards listed in Table 2; these are the low-end GT1030 GDDR5, the mid-range GTX1660Ti, and the last-generation three-year old enthusiast GTX1080 card. All code is written in C++ to use the CUDA framework. All results are obtained with compute capability 6.1.

Encoding results are obtained using a quantization-based encoder; i.e., the encoder has no rate-control mechanism. This is reasonable since the encoder is free to choose any method that produce legible codestream. PSNR results of such a method should be close to what a full pcRDO step produces, especially at mid and high bitrates. In a future work, we will explore a method, which has been tested on CPU successfully, that can achieve rate-control with a small increase in computational complexity.

Table 3. Encoding performance of the HTJ2K decoder for the 4K 4:4:4 12bit video test sequence ARRI AlexaDrums. "1b" is for 1 bit/pixel and "ls" is for lossless. †for [7], results at an unknown bitrate and codeblock size; they are obtained on nVidia Quadro P5000, which is equivalent to GTX1080.

Kernel	GT1030		GTX1660Ti		GTX1080	
	1b	ls	1b	ls	1b	ls
	KCT+DWT time to decompose one frame (ms)					
KCT+DWT	6.233	6.233	1.410	1.410	1.304	1.304
	Time to encode one frame (ms) using 64x64 codeblocks					
KMagSgn	3.243	4.338	0.698	1.089	0.551	0.647
KVLC	1.105	1.432	0.307	0.381	0.195	0.224
KMEL	0.275	0.303	0.092	0.026	0.102	0.026
KVCPY	0.115	0.096	0.028	0.079	0.022	0.076
	Frames per second					
	90	80	391	332	455	435
	Time to encode one frame (ms) using 32x32 codeblocks					
KMagSgn	3.263	4.350	0.794	2.013	0.576	0.815
KVLC	1.434	1.530	0.377	0.630	0.374	0.463
KMEL	0.496	0.366	0.107	0.093	0.125	0.100
KVCPY	0.370	0.568	0.077	0.129	0.064	0.126
	Frames per second					
	84	76	358	230	405	353
	Frames per second					
JPEG2K [7]	NA		NA		40†	

The performance results tabulated in Table 3 are obtained encoding the 4K 4:4:4 12bit test sequence ARRI AlexaDrums. 64x64 and 32x32 codeblocks are used with the irreversible CDF97 wavelet and 5 levels of wavelet decomposition. No overlap in frame encoding is employed. Results are obtained encoding 1000 frames. Lossless results are simulated using KCT+DWT time and the time needed to encode codeblocks that are produced by a lossless HTJ2K encoder. The table also lists results for JPEG2000 encoding [7] on the nVidia Quadro P5000, which is equivalent to the GTX1080, but at unknown bitrate and codeblock size. For CPU encoding performance, a 4-core i7-6700 CPU with a base clock of 3.4GHz, can encode around 93 frames per second at 1 bits/pixel. For the sequence tested here, the rate-distortion performance (BD-rate, BD-PSNR) of HTJ2K compared to JPEG2000, both using their optimal setting, is $(9.6\%, -0.7\text{dB})$.

5. CONCLUSIONS

In this work, we have presented a brief overview of the HTJ2K standard. HTJ2K is built on JPEG2000 and provides most of its features; it also supports lossless transcoding with JPEG2000. Importantly, it provides many fold increase in decode and encode speed compared to the original JPEG2000 on CPU and GPU. We have also presented a GPU-based implementation of an HTJ2K encoder, with preliminary performance results. We have shown that a low-end card can encode 4K 4:4:4 HDR video frames at more than 70 fps, while a high-end card can encode such frames at more than 400 fps.

6. REFERENCE

[1] ISO/IEC, "15444-1:2016 Information technology -- JPEG 2000 image coding system: Core coding system." 2016.

[2] D. S. Taubman and M. W. Marcellin, *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Norwell, MA, USA: Kluwer Academic Publishers, 2001.

[3] ISO/IEC, "15444-15:2019 Information technology -- JPEG 2000 image coding system -- Part 15: High-Throughput JPEG 2000." 2019.

[4] D. Taubman, A. Naman, and R. Mathew, "High Throughput Block Coding in the HTJ2K Compression Standard," in *2019 IEEE International Conference on Image Processing (ICIP)*, Sep. 2019, pp. 1079–1083, doi: 10.1109/ICIP.2019.8803774.

[5] ISO/IEC, "15444-9:2005 Information technology -- JPEG 2000 image coding system: Interactivity tools, APIs and protocols." 2005.

[6] N. Fürst, A. Weiß, M. Heide, S. Papandreou, and A. Balevic, "CUJ2K: A JPEG2000 Encoder on CUDA," 2009. http://cuj2k.sourceforge.net/ (accessed Feb. 04, 2020).

[7] Comprimato, "UltraJ2K™ - JPEG2000 SDK." [Online]. Available at https://comprimato.com/products/comprimato-jpeg2000/ (accessed Feb. 04, 2020).

[8] Fastvideo LLC, "JPEG2000 codec on GPU," *Fast Video GPU Image Processing*. [Online]. Available at https://www.fastcompression.com/products/gpu-jpeg2000.htm (accessed Feb. 04, 2020).

[9] P. Enfedaque, F. Aulí-Llinàs, and J. C. Moure, "GPU Implementation of Bitplane Coding with Parallel Coefficient Processing for High Performance Image Compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2272–2284, Aug. 2017, doi: 10.1109/TPDS.2017.2657506.

[10] F. Wei, Q. Cui, and Y. Li, "Fine-Granular Parallel EBCOT and Optimization with CUDA for Digital Cinema Image Compression," in *2012 IEEE International Conference on Multimedia and Expo*, Jul. 2012, pp. 1051–1054, doi: 10.1109/ICME.2012.115.

[11] ISO/IEC, "21122-1:2019 Information technology -- Low-latency lightweight image coding system -- Part 1: Core coding system." 2019.

[12] A. T. Naman and D. Taubman, "Decoding High-Throughput JPEG2000 (HTJ2K) On A GPU," in *2019 IEEE International Conference on Image Processing (ICIP)*, Sep. 2019, pp. 1084–1088, doi: 10.1109/ICIP.2019.8803729.

[13] M. J. Weinberger, G. Seroussi, and G. Sapiro, "LOCO-I: a low complexity, context-based, lossless image compression algorithm," in *Proceedings of Data Compression Conference - DCC '96*, Mar. 1996, pp. 140–149, doi: 10.1109/DCC.1996.488319.

[14] P. Enfedaque, F. Aulí-Llinàs, and J. C. Moure, "Implementation of the DWT in a GPU through a Register-based Strategy," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 12, pp. 3394–3406, Dec. 2015, doi: 10.1109/TPDS.2014.2384047.