DECODING HIGH-THROUGHPUT JPEG2000 (HTJ2K) 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), also known as JPEG 2000 Part 15, is the most recent addition to the JPEG2000 suite of coding tools. The file extension JPH has been designated for compressed images employing this new part of the standard. This new part describes a "fast" block coder for the JPEG 2000 format, while retaining most other JPEG2000 features and capabilities intact. The HTJ2K block coder is amenable to parallelizable high-speed encoding and decoding implementations; moreover, it is designed to allow lossless transcoding of already compressed JPEG2000 images that employ the regular block coder. HTJ2K supports the scalability options available in the JPEG2000 format except for quality scalability, which is available only to a limited extent. This work gives a high-level overview of this new block coder; we also present preliminary performance results for a GPU implementation. We show that a low-end GPU can decode 4K 4:4:4 12-bit videos at more than 60 frames per second (fps) while a high-end GPU can decode 8K HDR videos at more than 120 fps.

Index Terms— Image compression, graphical processing unit

1. INTRODUCTION

The JPEG2000 suite of image coding standards has been successful in a wide variety of applications and markets due to its versatility and performance. The main stages of the JPEG2000 compression pipeline are color transform, wavelet transform, and entropy coding, which is the task of the block coder. The color transform stage transforms the color components of an image to a new color space that is more suitable for compression from the human visual system point of view. The wavelet transform decomposes a color component into a set of subbands, exploiting spatial correlation. These subbands are then divided into rectangular regions known as codeblocks. The block coder then encodes the samples of each codeblock independently of other codeblocks. The compressed byte-streams produced by the block coder then undergo a post-compression rate-distortion (PCRD) optimization stage whereby coded data are assigned to quality layers within the generated codestream.

The JPEG2000 block coder employs a fractional bitplane coder; that is, the block coder employs three passes to encode each bitplane of the codeblock samples, where each pass encodes only some of the samples in that bitplane. These passes are known as: the significance propagation pass (SPP), the magnitude refinement pass (MRP), and the cleanup pass (CUP) [1].

The most computationally demanding stage of the JPEG2000 image compression pipeline is the block coder; the color and wavelet transforms in JPEG2000 involve a small number of arithmetic operations and are easily parallelized.

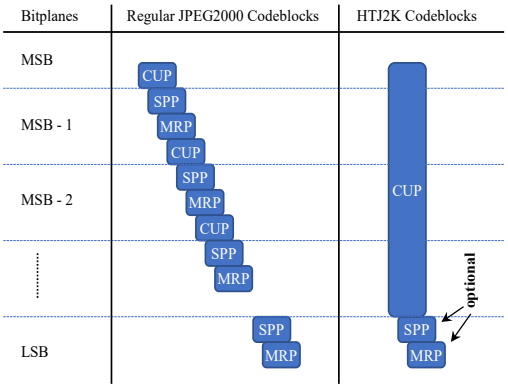

Fig. 1. Coding passes in regular JPEG2000 and HTJ2K codeblocks.

The block coding complexity can be partly attributed to the need of the block coder to visit codeblock samples multiple times as it generates the passes of the compressed codestream. But perhaps the more important source of complexity is the serial nature of the context-adaptive arithmetic coder used for entropy coding, which makes parallel processing very hard if not impossible.

To improve coding speed, the block coder can employ the "BYPASS" mode, which utilizes a simple bit stuffing for the SPP and MRP; i.e., no entropy coding is employed. In practice however, the improvement is modest, and it comes with a small reduction in coding efficiency.

A more important speedup approach has been to use multiple processing units (or cores) to code codeblocks; this is possible because each codeblock is coded independently of other codeblocks. However, for many applications, it is still desirable to have a block coder that has a lower complexity, which is what the high-throughput JPEG2000 (HTJ2K) is addressing.

On a CPU with AVX2, the speedup obtained from the HTJ2K block coder, compared to a regular JPEG2000 block coder, is anywhere between 10x faster decoding at low bitrates and 42x lossless encoding, with an average increase of 9% or less in compressed codestream bitrate [2], compared to regular JPEG2000. Moreover, on a 4-core Skylake-generation CPU, HTJ2K can encode and decode 4K 4:4:4 12-bit videos at 60fps or more [2]. This work explores the speed of HTJ2K decoding on a GPU.

HTJ2K supports most of the JPEG2000 features, except for limited quality scalability; however, resolution scalability is still available and can be used as a proxy for quality scalability. Moreover, HTJ2K allows lossless transcoding to and from regular JPEG2000; one usage scenario is to use HTJ2K on a compute-limited device to code images while they are being captured, and

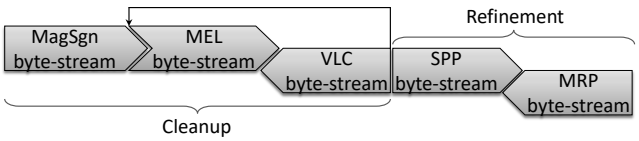

Fig. 3. The segments of a HTJ2K codeblock. The last two bytes of the cleanup pass contain a pointer to the start of the MEL segment.

then losslessly transcode them to regular JPEG2000 for archival purposes or for dissemination using JPIP [3]. Another scenario is when an interactive playback machine decodes a regular JPEG2000 codestream for display, simultaneously encoding it back into HTJ2K to save memory/computation should the end-user need to look at that frame again at some future point in time.

For fast image coding, an ISO working group, known as the JTC1/SC29/WG1, has also developed a lightweight image compression standard, known as the JPEG-XS [4], to serve 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. HTJ2K images have higher PSNR than JPEG-XS at the same data rate. In Section 4, we show that the proposed HTJ2K GPU implementation is also faster than a recently reported JPEG-XS GPU implementation [5].

The earliest implementation of regular JPEG2000 on a GPU is perhaps by [6]. Nowadays, commercial GPU implementations are available from [7] and [8]. To get more speed from a GPU, some researchers chose to modify the block coder [9], [10], and the entropy coding itself [9], breaking compatibility with JPEG2000. By contrast, HTJ2K deliberately adopts a coding pass strategy that represents exactly the same information as the original JPEG 2000 block coder.

2. THE HTJ2K CODEBLOCK CODESTREAM

The codestream of an HTJ2K codeblock usually has 1, 2, or 3 passes, starting with a CUP, followed by an optional SPP, and a yet optional MRP. The CUP of an HTJ2K stream usually contains many complete bitplanes, whereas the regular CUP contains only the cleanup pass of a bitplane. This is depicted in Fig. 1. HTJ2K adopts the BYPASS mode of regular JPEG2000 for coding the SPP and MRP; their availability facilitates transcoding. Moreover, when an HTJ2K codestream is directly generated (i.e., not transcoded), the existence of SPP and MRP helps with the PCRD optimization stage by providing the HTJ2K encoder with more possible truncation points (i.e., providing finer granularity).

For a given codeblock, the number of passes and their sizes are signaled in the header of the precinct containing the codeblock. Fig. 2 shows the byte-stream segments of a HTJ2K codeblock; the CUP is comprised of a magnitude-sign (MagSgn) segment that grows forward, a MEL segment that also grows forward, and a VLC segment that grows backward. The last two bytes in the CUP segment signal the position of the MEL segment. The SPP segment grows forward while the MRP, if it exists, grows backward. This forward-backward arrangement exposes more parallelism; for example, a decoder can choose, as done here, to decode the MEL and VLC segments of the CUP, together with the SPP, and to later on decode the MagSgn of the CUP together with the MRP.

HTJ2K processes codeblock samples in 2x2 blocks, known as "quads," in a raster order, as shown in Fig. 3. For a codeblock with width W and height H, we have $W_Q \times H_Q$ quads, where $W_Q = \lceil W/2 \rceil$ and $H_Q = \lceil H/2 \rceil$. We write Q_q for a quad, where $q = 0,1,\dots,W_Q \times H_Q - 1$. We also write $\mu_n \in \{0,1,\dots\}$ for the

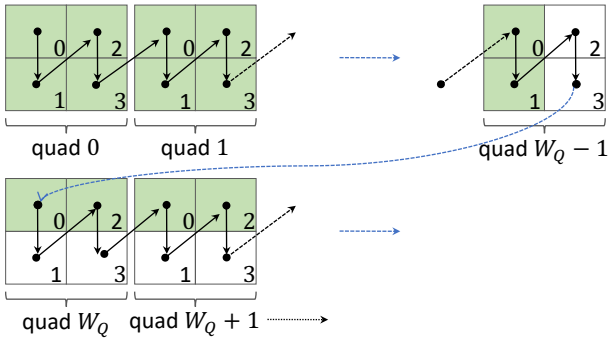

Fig. 2. HTJ2K encoder processes samples in quads. The white blocks are outside the image, therefore are assumed to be zero for this codeblock. The bottom-right corner of each sample shows the subindex j.

magnitude of the quantized codeblock sample x_n at location n, and $s_n \in \{0,1\}$ for its sign, where 0 and 1 represent positive and negative values, respectively. Here, the subscript n is equal to $4q + j$, where j is shown in in Fig. 3.

Coding efficiency comes from efficiently coding the location of significant samples (i.e., those that are non-zero after quantization) and the number of bits needed to represent them; bit patterns of sample values among adjacent samples have very little redundancy. This is true for HTJ2K, and for a wide range of image coding standards, including JPEG. For HTJ2K, this information is communicated using the MEL and the VLC byte-streams. For this purpose, we associate an exponent bound U_q with each quad Q_q such that $U_q \geq E_n$, where $n \in \{4q + j \mid j = 0,1,2,3\}$ and E_n is the exponent of sample x_n. The value $E_n - 1$ is the minimum number of bits needed to represent the value $\mu_n - 1$; the exponent E_n value is equal to $\lceil \log_2 \mu_n \rceil + 1$, when $\mu_n \geq 1$, and 0 when $\mu_n = 0$. We also write σ_n for the significance state of the sample x_n at location n, where $\sigma_n = 1$ for a significant sample ($\mu_n \geq 1$), and 0 otherwise. The significance states of a quad Q_q are concatenated in one significance pattern ρ_q, given by

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

Next, we explore the HTJ2K byte-streams.

2.1. The MEL Byte-Stream

The MEL coder is an adaptive run-length coder that can efficiently code runs of 0; this helps with coding stretches of all-zero quads. It is used in JPEG-LS [11] with 32 states, where it is called MELCODE. To keep complexity low, the HTJ2K uses a MEL coder with 13 states; each state k of the MEL coder is associated with an exponent e_k, and a threshold $\tau_k = 2^{e_k}$. A "0" codeword signals a run of τ_k 0s, while a "1" codeword indicates a run of 0s terminating with 1; a "1" codeword is followed by e_k bits of data specifying the number of zeros before the terminating one. Each run, complete or terminated, adapts the state of the MEL coder. The use of the MEL byte-stream is described in Section 2.3.

2.2. The VLC Byte-Stream

For a given quad Q_q, the VLC byte-stream carries a context-adaptive Huffman codeword, and when necessary, an offset u_q that is used to evaluate U_q. For each pair of quads, these parts are interleaved; that is, the byte-stream has a Huffman codeword for the first quad, then the second, followed by the offset u_q for the first

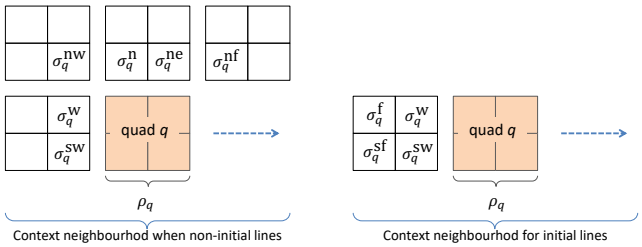

Fig. 4. The context neighborhood of a quad for initial line (first row) of quads in a codeblock, and non-initial.

quad, then the second. Decoding a Huffman codeword reveals 13 bits of data; these are: the 4-bit significance pattern ρ_q, an offset indicator bit u_q^{off}, and two 4-bit "EMB" patterns ϵ_q^{k} and ϵ_q^{1}. The meaning and use of EMB patterns ϵ_q^{k} and ϵ_q^{1} will be explained in Section 2.4.

For a quad Q_q, the Huffman codeword context c_q is 3 bits, and is composed from the significance of the samples near that quad; these nearby samples are shown in Fig. 4., for the initial line of quads (i.e., the first row of the codeblock) and non-initial quads. These 3 bits can be calculated once the significance patterns ρ_q of preceding quads are decoded. The HTJ2K uses a Huffman code with up to 7-bit codewords; to decode the HTJ2K Huffman codewords, it is convenient to use a lookup table with 1024 entries (3-bit contexts and up to 7-bit codewords).

2.3. Decoding U_q and E_n

For a quad with a context c_q equal to zero, a single decoded bit is retrieved from the MEL decoder. If this bit is zero, the exponent bound U_q is zero for that quad, and all the samples are zero in that quad. Conversely, if that bit is 1, or if the quad's context c_q is not zero, the exponent bound U_q is not zero, and must be calculated using a predict and increase strategy. To this end, a predictor κ_q is generated as explained shortly. Then, an offset indicator u_q^{off} value of zero indicates that the predictor κ_q is large enough, and U_q is equal to κ_q. If the offset indicator u_q^{off} is 1, the predictor κ_q is insufficient and U_q is equal to $\kappa_q + u_q$, where the offset u_q is obtained from the VLC byte-stream, as explained earlier. For non-initial lines, the predictor κ_q is obtained using

$$\kappa_q = \max\{1, \gamma_q \cdot (\max\{E_q^{\text{nw}}, E_q^{\text{n}}, E_q^{\text{ne}}, E_q^{\text{nf}}\} - 1)\}$$

where E_q^{nw}, E_q^{n}, E_q^{ne}, and E_q^{nf} are shown in Fig. 5., and

$$\gamma_q = \begin{cases} 0 & \text{if } \rho_q \in \{0,1,2,4,8\} \\ 1 & \text{otherwise} \end{cases}$$

Thus, the predictor κ_q depends on the exponents of the neighboring samples in the previous line if more than one of these samples is significant; otherwise the predictor is one. Initial lines do not have a previous line, and therefore κ_q is set to 1; however, to improve coding efficiency, a modified policy is used to calculate U_q. For more details, the interested reader is referred to [12]. While the design of HTJ2K could have used the samples to the left of a quad Q_q to help determine the predictor κ_q, the decoding of quad Q_q would have depended on samples that are being decoded, delaying the decoding process.

2.4. The MagSgn Byte-Stream

The EMB patterns ϵ_q^{k} and ϵ_q^{1} provide information about whether a sample exponent E_n is equal to the exponent bound U_q, but they might not carry this information for all the samples in the quad Q_q.

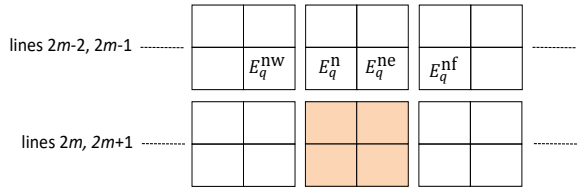

Fig. 5. Neighborhood information used to estimate the predictor κ_q for a non-initial quad Q_q.

The "known bit" pattern ϵ_q^{k} identifies for which samples this information is available, and the "known 1" pattern ϵ_q^{1} provides this information. For a significant sample in quad q, U_q bits are sufficient to communicate the sign and magnitude of that sample. However, if $\epsilon_q^{\text{k}} = 1$, then the most significant bit of the U_q magnitude bits is conveyed by the VLC codeword as the ϵ_q^{1} value. Thus, the EMB patterns can save up to 4 bits from the MagSgn byte-stream by spending fewer bits in a VLC codeword. The EMB patterns are given by

$$\epsilon_q^{\text{k}} = k_{4q} + 2 \cdot k_{4q+1} + 4 \cdot k_{4q+2} + 8 \cdot k_{4q+3}, \text{where } k_n \in \{0,1\}$$
$$\epsilon_q^{1} = i_{4q} + 2 \cdot i_{4q+1} + 4 \cdot i_{4q+2} + 8 \cdot i_{4q+3}, \text{where } i_n \in \{0,1\}$$

For a sample x_n in quad Q_q, a value y_n comprising m_n bits are read from the MagSgn byte-stream, where $m_n = \sigma_n \cdot U_q - k_n$. Then, $i_n \cdot 2^{m_n}$ is added to y_n, from which the sample is obtained.

HTJ2K is fast because it visits a sample a small number of times. Also, decoding a single codeword of the VLC byte-stream provides information for 4 samples; moreover, the interleaved decoding of pairs of quads exposes instruction level parallelism that a superscalar processor can exploit. Additionally, decoding the MagSgn byte-stream can proceed from one row of quads to the next, without any dependencies on horizontally adjacent samples.

Decoding the SPP and MRP byte-streams is not explained here, because they are closely related to the original standard [1], [13].

3. A GPU-BASED DECODER FOR HTJ2K

Only 64x64 codeblocks are explored here; a 3-color 4K image usually has around 6300 such codeblocks. The host CPU decodes precinct headers and prepares lists for byte-stream locations inside the HTJ2K codestream; the host CPU then uploads the file and the lists to the GPU. This takes a small amount of time, and can be run in parallel with GPU decoding a previous frame.

It is best to think of decoding the HTJ2K CUP as a two-step process. The first step, handled by the KCUPS1 kernel, decodes the MEL and VLC byte-streams. Here, a single thread is used to decode each codeblock. The Huffman table needed by the VLC decoder is transferred from global memory to shared memory by the first warp of each CUDA block. For a given quad Q_q, decoding these byte-streams produces the significance pattern ρ_q, the EMB patterns ϵ_q^{k} and ϵ_q^{1}, and the offset u_q. These values are stored together in a 32-bit integer in global memory, for retrieval by the second decoding kernel. For 4K images with 64x64 codeblocks, this kernel underutilizes the GTX1080 card; better utilization can be achieved with 32x32 blocks. The KCUPS1 uses 45 registers.

The second step, handled by the KCUPS2 kernel, uses the data stored by the KCUPS1 kernel to decode the MagSgn byte-stream, and reconstruct 32-bit sign-magnitude values for each sample x_n. Each thread in a warp decodes two columns of a codeblock; thus, one warp is needed to decode a 64x64 codeblock. The KCUPS2 uses 64 registers.

Table 1. The GPU cards used in this work

Card	CUDA Cores	Boost Clock (MHz)	Mem. BW (GB/s)	Attainable Mem. BW (GB/s)	PCIe 3.0 Lanes	DMA Cntrlrs
GT1030	384	1468	48	~40	x4	2
GTX1060	1280	1785	192	~160	x16	2
GTX1080	2560	1847	320	~240	x16	2

Table 2. 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.

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

A practical decoder, can always choose to discard the SPP and MRP to reduce complexity, in exchange for some reduction in visual quality; here we consider such a decoder and we denote it by "NR" for "no refinement." We also consider a decoder, denoted by "R," that decodes these passes. This decoder employs the modified kernels KCUPS1-SPP and KCUPS2-MRP; the KCUPS1-SPP kernel decodes the SPP byte-stream together with decoding the MEL and VLC byte-streams, which is possible because the significance information needed for decoding the SPP becomes available as the MEL and VLC byte-streams are decoded. The decoded SPP is stored in global memory (2 bits/sample). This kernel uses 77 registers and 144 bytes of shared memory per thread as scratchpad memory. The KCUPS2-MRP kernel, which uses 82 registers, decodes the MagSgn and MRP byte-streams, and also utilizes the decoded SPP information to reconstruct sample values.

For wavelet synthesis WSYN, the approach proposed in [14] is used. Our implementation uses 32-bit floats, which should be sufficient for images with more than 16-bit sample values. The last step of synthesis employs a "special" kernel that performs the color transform step immediately after the wavelet synthesis step, writing back a 16-bit reconstructed image in interleaved format suitable for storage; this saves memory transactions. Codeblocks that contain zero coding passes (i.e., all their sample values are zero) are handled on the fly during the wavelet synthesis step; the synthesis kernel retrieves decoded samples only for codeblocks that have one or more passes, filling zeros for other codeblocks. To do this, the synthesis kernel receives a list for codeblocks that have zero passes. This save memory bandwidth. The special wavelet synthesis kernel uses 125 registers; although this can limit occupancy, the kernel still achieves very high memory bandwidth utilization because of the high instruction-level parallelism in processing three colors.

It is possible, although not explored here, to run the decoder in a low-delay mode, whereby decoding a frame is performed in batches; each batch decodes a preset number of rows, such as 10% of the height of a frame.

Table 3. Decoding performance of the HTJ2K decoder for the 4K 4:4:4 12bit video test sequence ARRI AlexaDrums. "1b" is for 1 bit/pixel, "4b" is for 4 bits/pixel, and "ls" is for lossless. †unknown codeblock size & bitrate. *estimated from result in [5].

	GT1030			GTX1060			GTX1080		
Kernel	1b	4b	ls	1b	4b	ls	1b	4b	ls
	Time to decode one frame (ms) w/o refinement (NR)								
KCUPS1	.560	1.77	4.43	.408	.485	1.48	.385	.420	.520
KCUPS2	.727	2.00	4.88	.212	.538	1.36	.128	.310	.729
	Time to decode one frame (ms) with refinement (R)								
KCUPS1-SPP	1.12	2.81	-	.856	1.00	-	.807	.895	-
KCUPS2-MRP	1.04	2.82	-	.300	.806	-	.172	.430	-
	Wavelet Synthesis and Color Transform (ms/frame)								
WSYN+Color	3.96	4.57	6.15	1.11	1.29	1.66	.750	.886	1.19
	Frames per second								
Proposed-NR	180	118	62	550	420	220	770	588	402
Proposed-R	160	96	-	425	317	-	560	440	-
JPEG-XS [5]		NA			NA		233*	194*	NA
JPEG2000 [7]		NA			95†			NA	

4. EXPERIMENTAL RESULTS

Experimental results are obtained using 3 different NVidia GPUs: a low-end GT 1030 with GDDR5, a mid-range GTX 1060, and an enthusiast GTX 1080 card. Table 1 lists the specifications of these cards. All code is written in CUDA with C++.

Reconstructed frames can either be displayed or downloaded to the host system for storage. If download is desired, the PCI-Express (PCIe) interface between GPU and the host system can limit the number of frames that can be downloaded per second; Table 2 lists download speeds in fps for three PCIe generations. Experimental results confirm that these frame rates are indeed achievable. Display-only applications are not affected by this limit. All tested cards support PCIe 3.0; they also have 2 DMA controllers that enable them to upload and download data concurrently, utilizing the bi-directional bandwidth of the PCIe interface.

The performance achieved and kernel times are tabulated in Table 3 for the 4K 4:4:4 12bit test sequence ARRI AlexaDrums when 64x64 codeblocks, irreversible CDF97 wavelet, and 5 levels of decomposition are used. No overlap in frame decoding is employed, but compressed image upload is run in parallel with decoding. Frame decode rates are obtained decoding 1000 HTJ2K frames. For decoding losslessly compressed images, it is rather pointless to present results for the refinement decoder (R), because these images do not have SPP nor MRP. The table also lists results for JPEG-XS [5] and JPEG2000 [7] on GPU. For the sequence tested here, the rate-distortion performance (BD-rate, BD-PSNR) of HTJ2K compared to regular JPEG2000, both using their optimal setting, is $(9.6\%, -0.7\text{dB})$ [2]. For CPU performance, the HTJ2K decode rate, achieved on the 4-core i7-6700 CPU with a base clock of 3.4GHz, can be as high as 148 fps or more, albeit with different test conditions; more results can be found in [2]. Thus, a GPU can be a relatively low-cost way to have higher decode rates than what a 4-core CPU can deliver. It is worth noting that the computational complexity of decoding a 4K video at 480 fps is similar to that of decoding 8K at 120 fps; an 8K video however exposes more parallelism, and can therefore better utilize a GPU.

5. CONCLUSIONS

We have presented an overview of the new HTJ2K. HTJ2K is built upon and can utilize most of the JPEG2000 format features. HTJ2K codestreams can also be transcoded to and from regular JPEG2000 codestreams. Encoding and decoding of HTJ2K codestreams is many folds faster than regular JPEG2000 on CPUs. For GPUs, we have shown, based on the preliminary results presented here, that decoding HTJ2K is also many folds faster than regular JPEG2000, and we believe the same is true for encoding. A low-end GPU is sufficient for 4K HDR video playback at more than 60fps, while a high-end GPU can play an 8K HDR video at 120fps.

6. REFERENCE

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

[2] D. S. Taubman, A. T. Naman, and R. Mathew, "High Throughput Block Coding in the HTJ2K Compression Standard," in *International Conference on Image Processing (ICIP)*, Taiwan, 2019.

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

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

[5] V. Bruns, T. Richter, B. Ahmed, J. Keinert, and S. Föel, "Decoding JPEG XS on a GPU," in *2018 Picture Coding Symposium (PCS)*, 2018, pp. 111–115.

[6] N. Fürst, A. Weiß, M. Heide, S. Papandreou, and A. Balevic, "CUJ2K: A JPEG2000 Encoder on CUDA," 2009. [Online]. Available: http://cuj2k.sourceforge.net/. [Accessed: 27-Jan-2019].

[7] Comprimato, "UltraJ2K™ - JPEG2000 SDK." [Online]. Available: https://comprimato.com/products/comprimato-jpeg2000/. [Accessed: 27-Jan-2019].

[8] Fastvideo LLC, "JPEG2000 codec on GPU," *Fast Video GPU Image Processing*. [Online]. Available: https://www.fastcompression.com/products/jpeg2000/gpu-jpeg2000.htm. [Accessed: 27-Jan-2019].

[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.

[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*, 2012, pp. 1051–1054.

[11] 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*, 1996, pp. 140–149.

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

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

[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.