High Throughput JPEG 2000 (HTJ2K): Algorithm, Performance and Potential

Authors: David Taubman[1,2], Aous Naman[1,2], Reji Mathew[1], Michael Smith[3], Osamu Watanabe[4] and Pierre-Anthony Lemieux[5]

Affiliations:
1. UNSW Sydney, Australia.
2. Kakadu R&D Pty. Ltd., Australia
3. Wavelet Consulting LLC, U.S.A.
4. Takushoku University, Japan
5. Sandflow Consulting LLC, U.S.A.

Last updated: 29 May 2020

Forward

This document is expected to evolve over time to provide as much information as possible to facilitate assessment and deployment of the flexible and efficient new High Throughput JPEG 2000 standard: ITU-T Rec T.814 | IS 15444-15.

TABLE OF CONTENTS

1. OVERVIEW 3

1.1. THROUGHPUT AND KEY FEATURES OF HTJ2K 4
1.2. QUALITY SCALABILITY IN HTJ2K 4
1.3. RELEVANT STANDARDS 5
1.4. ROYALTY STATUS OF HTJ2K 5
1.5. SOFTWARE TOOLS AND RESOURCES FOR HTJ2K 6
1.6. ORGANIZATION OF THIS DOCUMENT 7

2. HTJ2K USAGE SCENARIOS 8

2.1. DISTRIBUTION, ARCHIVAL AND MANAGEMENT OF CINEMA AND VIDEO ASSETS 8
2.2. ENERGY EFFICIENT IMAGE CAPTURE AND PREVIEW 8
2.3. LIVE PERFORMANCES, VIEW FINDING AND TELEPRESENCE SYSTEMS 9
2.4. HIGH RESOLUTION VIDEO CAPTURE 10
2.5. CACHE MANAGEMENT FOR INTERACTIVE RENDERING 10
2.6. OPTIMIZED IMAGE DELIVERY IN WEBSERVERS 11

3. OVERVIEW OF THE HTJ2K CODEC 12

3.1. HIGH LEVEL OVERVIEW 12
3.2. RELATIONSHIP BETWEEN HT AND J2K-1 BLOCK CODING PASSES 12
3.3. RATE CONTROL AND COMPLEXITY CONTROL FOR HTJ2K ENCODERS 14
3.4. Q-FACTOR FOR JPEG 2000 16

4. INTRODUCTION TO THE HT BLOCK CODING ALGORITHM 20

4.1. HT CODEWORD SEGMENTS AND CONCURRENCY 20
4.2. HT REFINEMENT PASSES 20
4.3. CLEANUP MAGNITUDE EXPONENTS, EXPONENT BOUNDS AND MAGSGN VALUES 21
4.4. MAIN ELEMENTS OF THE HT CLEANUP PASS 22
4.5. FORMATION OF THE HT CLEANUP AND REFINEMENT BYTE-STREAMS 25

5. LOW COMPLEXITY AND HIGH CONCURRENCY ATTRIBUTES OF HTJ2K 27

5.1. PARALLELISM AND CONCURRENCY 27
5.2. ENCODING RESOURCES AND VECTORIZABILITY 28
5.3. DECODING RESOURCES AND VECTORIZABILITY 29

6. HTJ2K CODE-STREAM STRUCTURE AND FEATURES 31

7. CODING EFFICIENCY AND THROUGHPUT IN SOFTWARE 32

7.1. COMPRESSION OF PROFESSIONAL PHOTOGRAPHIC MATERIAL 32
7.2. 4K 4:4:4 12BIT VIDEO ENCODING AND DECODING ON AN I7 4-CORE CPU 35
7.3. 4K 4:2:2 10BIT VIDEO ENCODING AND DECODING ON AN I7 4-CORE CPU 41
7.4. COMPRESSION OF LARGE AERIAL IMAGERY 43
7.5. COMPRESSION OF ENTERTAINMENT IMAGERY 45
7.5.1. 3840X2160P 10BIT ITU-R BT.709 4:4:4 23.976P SDR 46
7.5.2. 3840X2160P 12BIT 24P DCINEMA DCDM SDR 48
7.5.3. 3840X2160 24P 16BIT HALF-FLOAT OPENEXR (SCENE-LINEAR HDR) 50

8. BRIEF DISCUSSION OF GPU AND HARDWARE DEPLOYMENTS 52

9. LOW-LATENCY COMPRESSION WITH HTJ2K 53

10. JPH COMPARISON WITH JPG 65

11. CONCLUSIONS 68

12. BIBLIOGRAPHY 69

1. Overview

JPEG 2000 provides a rich set of features that find application in many diverse fields. Some of the most important features, as found in JPEG 2000 Part-1 [1] are as follows:

- **Compression efficiency**
- **Quality scalability:** ability to extract almost any reduced quality representation from the code-stream while retaining full coding efficiency
- **Resolution scalability:** ability to extract almost any power-of-2 related resolution from the code-stream while retaining the full coding efficiency
- **Region-of-interest accessibility:** ability to reconstruct or communicate an arbitrary spatial region, while retaining high coding efficiency
- **Parallelism:** ability to perform the encoding or decoding processes in parallel across a large number of CPU cores, GPU threads or in hardware
- **Non-iterative optimal rate control**: ability to achieve a target compressed size without iterative encoding.
- The ability to target visually relevant optimization objectives
- Error resilience
- Compressed domain (i.e. very low memory) transposition, flipping and cropping operations
- Ability to re-sequence information at the code-block, precinct or J2K packet level, so as to target progressive build-up or low latency objectives.

Most of these features derive from the use of the EBCOT algorithm (Embedded Block Coding with Optimized Truncation), while use of the hierarchical Discrete Wavelet Transform (DWT) also plays an important role.

In addition to these core features, the JPEG 2000 suite of standards provide good support for the following applications:

- Efficient and responsive remote interactive browsing of imagery (including video and animations), via JPEG 2000 Part-9 [2], also known as the JPIP standard.
- Efficient on demand rendering of arbitrary regions from huge imagery sources.
- High dynamic range compression, through the use of non-linear tone curves and/or custom floating-point mappings, as defined in JPEG 2000 Part-2 [3].
- Rich metadata annotation, as also defined in JPEG 2000 Part-2.
- Efficient compression of hyper-spectral and volumetric content, as also defined in JPEG 2000 Part-2 [3].

The one significant drawback of the JPEG 2000 standards is computational complexity. JPEG 2000 provides many modes that can be used to target efficient hardware solutions. Also, software implementations are sufficiently efficient that images of just about any displayable size can be rendered nearly instantaneously on modern platforms, including mobile devices. However, for video applications and for applications that are especially power/energy conscious, compression and rendering complexity can become a problem.

This document describes a drop-in replacement for the original JPEG 2000 block coding algorithm, along with its implications for the whole JPEG 2000 family of standards and for a variety of image and video compression, archival and rendering tasks. In this document, we refer to the original block coding algorithm, defined in JPEG 2000 Part-1 as the “J2K-1” algorithm, while the new block coder is known as the “HT” (High Throughput) algorithm.

Terminology and Standards:

- **HT** = "High Throughput"
- **HTJ2K code-stream** = a coded image representation conforming to ITU-T Rec T.814 | IS 15444-15, read as "High Throughput JPEG 2000, which is based on the representation in JPEG 2000 Part-1 and compatible with the features of JPEG 2000 Part-2. HTJ2K code-streams can contain J2K-1 code-blocks, in addition to HT code-blocks.
- **JPH** = an image file format described in ITU-T Rec T.814 | IS 15444-15, that is based on the JP2 format in JPEG 2000 Part-1, with enhanced colour space and colour association capabilities, embedding a single HTJ2K code-stream.
- **FBCOT** = "Fast Block Coding with Optimized Truncation" is an algorithm that can be used by HTJ2K encoders to achieve a target compressed size without iterative encoding, similar to the well-known EBCOT algorithm that is used with J2K-1. FBCOT works with the HT block coder and may include additional complexity management.

1.1. Throughput and key features of HTJ2K

At quality levels of interest to most applications, the HT block coding algorithm is an order of magnitude faster than J2K-1 in a carefully optimized CPU-based implementation, as determined by a careful evaluation during the development of the HTJ2K standard by the ISO/IEC working group known as JPEG (formally ISO/IEC JTC1/SC29/WG1). This evaluation was conducted in accordance with the detailed procedure outlined in the HTJ2K call for proposals [4]; a comprehensive public summary of the results is available in [5].

As shown later in this document, the entire end-to-end encoding or decoding system can be almost 10 times faster for an HTJ2K code-stream than one based on the J2K-1 algorithm, at typical compressed bit-rates employed in professional applications, with much larger speedups on the order of 30x for lossless compression or decompression.

As a drop-in replacement for J2K-1, the HT block coding algorithm can be used in conjunction with pretty much any of the technologies described in the entire JPEG 2000 family of standards, to dramatically increase throughput, while retaining most existing features.

Importantly, the HT block coding algorithm allows completely lossless transcoding to/from code-streams that have been generated using the J2K-1 algorithm.

The price paid for the dramatic reduction in complexity associated with the HT block coding algorithm is a modest reduction in compression efficiency (typically 6% to 10%), plus substantial elimination of the quality scalability feature of JPEG 2000.

1.2. Quality scalability in HTJ2K

Quality scalability is important, especially for interactive browsing applications via JPIP, where it enables a large effective saving in communication bandwidth and/or increase in responsiveness of the browsing application, allowing delivery of only the information that an interactive client requires, or delivering the information in an order that makes the browsing experience as seamless as possible. Quality scalability can also be important in distribution systems; it provides an elegant solution for progressive degradation of image/video archives if required; and it also enables selective protection and/or encryption of content, all of which are important for many applications.

J2K-1 achieves quality scalability by coding each code-block through a finely embedded collection of coding passes, allowing the coded representation of each block to be truncated at any point. The HT block coder also employs coding passes but they

are not fully embedded, and at most three passes are preserved for each code-block within the final code-stream, leaving at best a very limited form of quality scalability.

However, the reversible transcoding property of the HT block coding algorithm allows quality scalability to be retrieved or temporarily surrendered at will, by selectively transcoding the block bit streams associated with images, video frames, volumes, or even individual regions within an image or volume, in order to manage performance and power consumption objectives within a media distribution and processing system.

The HTJ2K code-stream syntax supports the recording of coding pass boundaries associated with any number of JPEG 2000 quality layers, even though only some of the coding passes are actually present. This allows all quality scalability attributes from J2K-1 code-blocks to be fully preserved upon transcoding to the HT format. It also allows an encoder to insert quality scalability information into an original HTJ2K code-stream so that it can be recovered upon transcoding to J2K-1, or used to guide the transcoding from one HTJ2K code-stream to another, at a reduced bit-rate, without the involvement of any J2K-1 block bit-streams.

1.3. Relevant standards

The HTJ2K standard, corresponding to Part-15 of the JPEG 2000 family of standards [6], is a common text between the ITU and ISO/IEC. The published text is available from the ITU at https://www.itu.int/rec/T-REC-T.814-201906-I/en.

HTJ2K builds upon JPEG 2000 Part-1 [1], introducing only an alternate block coder. Accordingly, HTJ2K cannot be fully understood or implemented without reference to this standard. In particular HTJ2K depends upon a minor revision of this standard that is published by the ITU and ISO/IEC and is currently available free of charge at https://www.itu.int/rec/T-REC-T.800-201906-I/en.

HTJ2K is compatible with all of the extended coding technologies described in JPEG 2000 Part-2 [3], including its substantial Amendment 3 [7]. In particular, the combination of HTJ2K with Part-2 multi-component transforms and with Part-2 non-linear point transforms should be especially valuable for certain applications.

Part-4 of the JPEG 2000 family of standards [8] describes conformance testing procedures and provides code-streams and corresponding images for use in such testing procedures. This standard is the subject of an active amendment that reached Committee Draft stage in July 2019, the purpose of which is to augment all Part-1 code-streams and conformance testing procedures with corresponding code-streams and procedures for Part-15.

Part-5 of the JPEG 2000 family of standards [9] provides reference software for JPEG 2000 Part-1. This standard is the subject of an active amendment that also reached Committee Draft stage in July 2019, the purpose of which is to add reference software for Part-15.

Part-16 of the JPEG 2000 family of standards specifies the carriage of JPEG 2000 code-streams in ISO/IEC 23008-12, commonly referred to as HEIF. A revision is underway to support more flexible wrapping of all JPEG 2000 code-streams, including HTJ2K.

1.4. Royalty status of HTJ2K

HTJ2K builds on the tools and technologies of JPEG 2000, a royalty free standard that was produced almost two decades ago, yet remains the most scalable and versatile still image coding standard available. The HT block coding algorithm and the FBCOT

encoding framework described in Section 3 and 4 are derived from technologies originally proposed in [10], [11] and [12]. As owner of relevant intellectual property behind this technology, Kakadu R&D Pty Ltd has made royalty free declarations to the ITU and the ISO, with the objective of maintaining the royalty free status of the entire JPEG 2000 family of standards.

1.5. Software tools and resources for HTJ2K

Kakadu SDK and Demonstration Executables

The Kakadu[1] software toolkit provides a comprehensive and highly efficient implementation of JPEG 2000 Part-1 [1], most of JPEG 2000 Part-2 [3] and JPEG 2000 Part-9 [2], in addition to the motion JPEG 2000 file format of Part-3. As core contributors to the new HTJ2K standard, Kakadu Software Pty Ltd has also developed a highly efficient implementation of JPEG 2000 Part-15 [6], which has been integrated into version 8.0 of the Kakadu toolkit.

Kakadu provides a suite of powerful demonstration applications[2] that are widely used to create, read and experiment with JPEG 2000 content, all of which support the new HTJ2K standard.

Kakadu is written in C++, but provides extensive non-native interfaces to other programming environments, such as Java and C#. The Kakadu implementation supports all HTJ2K encoding options described in this document, HTJ2K decoding, region-of-interest and resolution-of-interest decoding, interactive rendering, client-server based dissemination of HTJ2K and other JPEG 2000 content, and reversible transcoding between code-streams based on J2K-1 and HT block coding algorithms, including the transcoding of HT to HT and J2K-1 to J2K-1 based code-streams. Kakadu also supports the JP2, JPX and MJ2 file formats and elementary broadcast streams, and provides extensive support for the metadata that can be maintained within such file formats.

HTJ2K Reference Software

Shigetaka Ogawa, from ICT Link, Japan, has independently developed a reference implementation of the HTJ2K standard known as "TT," written in C++, which forms the basis for the HTJ2K reference software in JPEG 2000 Part-5 [9].

Matlab-based HTJ2K Implementation

Dr. Osamu Watanabe, of Takushoku University, Japan, has developed a Matlab-based implementation of the HTJ2K standard, supporting both encoding and decoding, as described in [13].

OpenJPH: An Open Source HTJ2K Software in C++ and JavaScript

Dr. Aous Naman, from the University of NSW, Sydney Australia, has developed an implementation of the HTJ2K standard, which is available as open source software[3]. This implementation includes both encoding and decoding tools. As open source software, of course, the implementation has not been optimized nearly to the same extent as full commercial offerings such as that of Kakadu.

[1] https://kakadusoftware.com
[2] https://kakadusoftware.com/downloads
[3] https://github.com/aous72/OpenJPH

A light weight JavaScript decoder, derived from the same code base, is also available, smoothing the path to adoption of HTJ2K code-streams and JPH files in web-based applications.

1.6. Organization of this document

The remainder of this document is organized as follows. Section 2 provides further motivation and context for the HTJ2K standard by describing some important applications that can benefit substantially from the new technology. Section 3 provides an overview of the HTJ2K technology, without delving into technical details, illustrating the relationship between the HT block coding algorithm and the rest of the system, explaining the significance of HTJ2K's reversible transcoding feature, and describing the FBCOT paradigm, with rate and complexity control, that enables extremely high throughput compression with exact non-iterative rate control.

Section 4 provides some technical details of the HT block coding that will be of interest to developers as a precursor to reading the actual text of the standard, while Sections 5 and 6 provide high level descriptions of the beneficial properties and features of the HT block coding algorithm and the HTJ2K codestream syntax, respectively.

Section 7 provides experimental results based on the Kakadu implementation of HTJ2K, demonstrating both the coding efficiency and throughput attributes of HTJ2K, where throughput is assessed in a software environment.

Section 8 is reserved for a discussion of HTJ2K deployment on other platforms – primarily GPU platforms, but some discussion of hardware deployment will be added as the document evolves.

Section 9 considers low-latency encoding with HTJ2K, providing an analysis and experimental results for end-to-end latencies as low as a few tens of lines.

Finally, Section 10 compares HTJ2K with other relevant non-JPEG 2000 compression schemes, investigating both coding efficiency and throughput, where possible.

2. HTJ2K usage scenarios

We describe here just a few application areas which can benefit from the new HTJ2K standard. These can mostly be understood as exercising various paths through the system diagram in **Figure 1**.

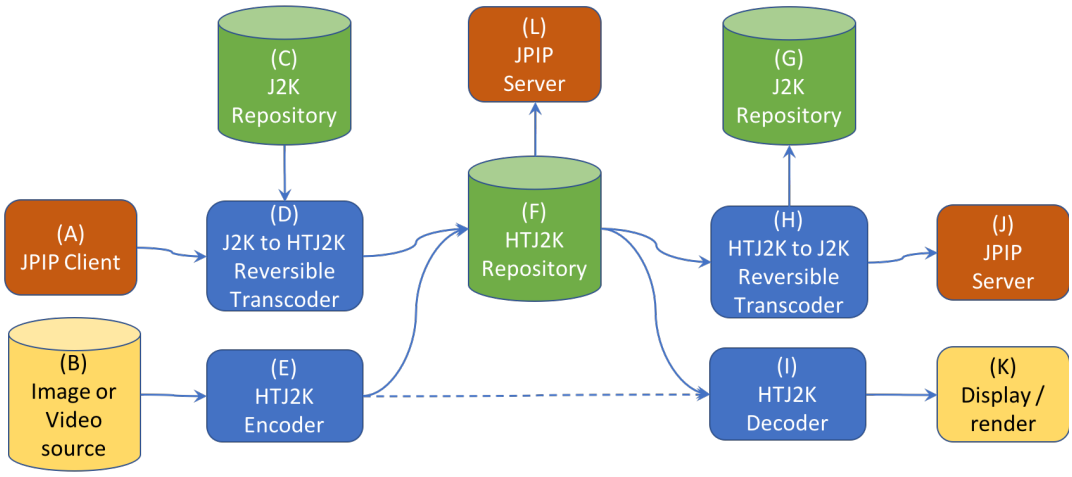


Figure 1: HTJ2K systems diagram

2.1. Distribution, archival and management of cinema and video assets

JPEG 2000 provides an excellent framework as an intermediate distribution format for video content, allowing the resolution (or even region) of interest to be extracted efficiently, close to the point of delivery, whereupon it may be transcoded into any of a number of common streaming video formats.

This is essentially what happens already with JPEG 2000, facilitated by the Interoperable Master Format (IMF) specifications defined by the Society for Motion Picture Television Engineers (SMPTE). Such applications can benefit considerably from use of the HTJ2K representation, which retains all desired features of JPEG 2000 for such applications while dramatically reducing computational load.

Following path **C-D-F-I-K** in the HTJ2K systems diagram of **Figure 1**, existing archived JPEG 2000 media can be transcoded, stored, distributed and rendered with greatly reduced computational complexity. Path **F-H-G** allows the original JPEG 2000 representation to be recovered losslessly, if required.

SMPTE has completed its work on adding support for HTJ2K to MXF by modifying the existing SMPTE ST 422 standard[4]. This should make it easy to adapt HTJ2K to D-Cinema, IMF and other MXF-based applications. While the changes to ST 422 are minimal, they are nevertheless necessary to unambiguously signal the presence of HTJ2K code-streams and the capabilities required to process them.

2.2. Energy efficient image capture and preview

HTJ2K and its associated JPH file format have many desirable properties as an image capture format for digital cameras, mobile phones and more advanced multi-sensor imaging devices. Direct encoding to HTJ2K requires very low energy in software, GPU or hardware. HTJ2K supports very high sample precisions (e.g., 12-, 16- or even 24 bits per sample), allowing content with extremely high dynamic range to be accurately

[4] https://kws.smpte.org/higherlogic/ws/public/projects/581/details

preserved in both linear and log-like representations. In the extreme, HTJ2K can be used to losslessly encode half-float or even full IEEE floating point representations of scene radiance. The JPH file format supports application defined colour associations, including multiply associated colours and offset sampling locations, allowing both raw sensor data and tone mapped RGB imagery to be preserved in the same format.

Perhaps more important are the resolution scalability and region accessibility features of HTJ2K, which allow a JPH file to be interactively previewed with extremely low energy – vastly lower than the energy required to interact with a JPEG file. This usage scenario exercises path **B-E-F-I-K** in the HTJ2K systems diagram of **Figure 1**.

Additionally, HTJ2K works seamlessly with the JPIP standard [2] for interactive remote browsing of JPEG 2000 imagery. This provides a highly efficient mechanism to support remote preview of captured content from devices in an ad-hoc network, exercising path **B-E-F-L**. For conventional JPEG 2000 content, this mechanism has proven highly effective in challenging communication environments, such as those encountered in a battlefield. The most efficient and responsive possible browsing of remote content is enabled by transcoding HT code-blocks on demand, to the fully embedded J2K-1 representation, exercising path **B-E-F-H-J** in the HTJ2K systems diagram.

2.3. Live performances, view finding and telepresence systems

Live video exercises path **B-E-I-K** in the HTJ2K systems diagram of **Error! Reference s ource not found.**, taking advantage of HTJ2K's extremely high encoding and decoding throughput, its high coding efficiency even at lower bit-rates, and its non-iterative rate control capability, with low (sub-frame) end-to-end latency. These properties allow high quality 4K content to be streamed over conventional IP or local wireless links. Heterogeneous systems, involving low cost hardware, software or GPU platforms are fully supported with tuneable latency, due to the flexible block-based structure of JPEG 2000.

JPEG 2000 in general, and HTJ2K in particular, can achieve very low end-to-end latencies, down to a few tens of frame lines if required, by appropriate choice of parameters for the wavelet transform, code-block dimensions, precinct dimensions, and packet progression order. To do this, code-stream content must be finalized and flushed regularly to the transport layer. Constant bit-rate transport can also be achieved by careful use of the non-iterative rate control strategies available in JPEG 2000 and HTJ2K, and by inserting stuffing bytes into the code-block byte-streams where necessary to avoid buffer underflow. Both the J2K-1 and HT block coding algorithms allow the insertion of stuffing bytes after encoding, without adversely affecting the decoded content.

As with all JPEG 2000 code-streams, HTJ2K code-streams are relatively robust to errors, and the code space has been carefully reserved during design to ensure that resynchronization can be achieved unambiguously at marker boundaries.

Content can be streamed to multiple devices with varying levels of computational resources, some of which may render the content at a reduced resolution or over a restricted region of interest, as the need arises, consuming less resources as they do so.

Since HTJ2K code-streams are composed of many independently coded code-blocks, decoders can trade latency for parallelism without being constrained by complex interleaving dependencies. For example, a hardware decoder may achieve high throughput with extremely low levels of latency, by taking full advantage of low level concurrency features of the HT algorithm. Meanwhile, a CPU or GPU based decoder

might achieve the same throughput but with much higher latency, by taking advantage of the macroscopic parallelism afforded by the code-block structure.

2.4. High resolution video capture

Most mobile devices do not have sufficient CPU power to directly compress the high resolution video streams that their image sensors are capable of producing, if the J2K-1 block coding algorithm is used. HTJ2K addresses this problem by allowing compressed streams to be produced in real time, exercising path **B-E-F** in the HTJ2K systems diagram of **Figure 1**. If desired, they can later be transcoded to the J2K-1 representation, exercising path **F-H-G** or **F-H-J**. There may be many reasons to do this, but it is particularly valuable if quality scalability attributes are important – e.g. for efficient interactive browsing via JPIP [2].

While transcoding is possible between any two compression formats, transcoding between JPEG 2000 code-streams that use the HT block coder and those that use the J2K-1 block coder has the following special properties:

a) Transcoding can take place in the transformed domain;
b) Transcoding is inherently lossless (i.e., reversible), even if though the compression itself is lossy; and
c) Transcoding can be done selectively, as the need arises, even on a code-block by code-block basis.

Apart from transcoding to J2K-1, the HTJ2K encoded video itself has many valuable properties:

a) HTJ2K code-streams can be decoded with very high throughput, with modest computational, GPU or hardware resources;
b) The content can be decoded at a reduced resolution or for an arbitrary region of interest, with commensurately reduced computational resources.

2.5. Cache management for interactive rendering

In many applications of JPEG 2000, content is rendered multiple times. For example, large images are viewed interactively, by resolution or region of interest, so that successive rendered views contain many code-blocks in common.

While it is possible to cache fully decoded code-blocks in memory, this is both complex and wasteful of memory. Instead, each time a region of interest is rendered at some resolution of interest, the relevant code-blocks are normally decoded on demand, being accessed dynamically, either from a local file or a client cache.

The introduction of HTJ2K provides much more energy efficient ways to handle such applications. If the source content already employs the HT block coder, repeated decoding of common code-blocks during interactive rendering becomes vastly cheaper than with the J2K-1 block coding algorithm. On the other hand, if the original media uses the J2K-1 block coder, code-blocks that are being repeatedly accessed can be transcoded to an equivalent HT representation and stored within an intelligent content cache, from which re-rendering becomes much less expensive in the future. This approach corresponds to path **A-D-F-I-K** in the HTJ2K systems block diagram of **Figure 1**. It is extremely efficient, because the most expensive element in transcoding from the J2K-1 to HT formats is the J2K-1 block decoding step, which comes for free when transcoding only those code-blocks that are already being accessed. The HT and J2K-1 representations consume similar amounts of cache memory and all information is preserved.

2.6. Optimized image delivery in webservers

Nowadays, devices that are capable of browsing the Internet come in a large variety of screen sizes and resolutions. As such, adapting image resolution, and perhaps cropping, to better match the rendering device display improves user experience, by reducing needed bandwidth and rendering time. This is motivation for the "Responsive Images" initiative.

The responsive images initiative has existed for several years now and is currently supported by all widely used browsers: e.g., Firefox since version 38; Chrome since version 38; and Edge since version 18. The existing approach is for a webpage to specify different images for different resolutions; a web browser then requests the images that best match the display device. This requires a webserver to store and serve a potentially very large number of different, yet related images, which increases the server's workload, either due to increased cache thrashing, increased hard drive access, and/or increased memory.

HTJ2K is in a unique position to better address the goals of the responsive images initiative. Taking advantage of resolution scalability and random access capabilities, all desired resolution and cropping combinations of a single image can be efficiently served from a single JPH file, reducing the web server's workload, and avoiding the proliferation of redundant representations. This could be realized, for example, through a mod to the widely deployed Apache webserver, with metadata side files employed to simplify the discovery of fragments from the full JPH file that need to be served; on the client side, the Javascript-based HT decoder mentioned in Section 1.5 can be used to make the content available to all existing web browsers.

3. Overview of the HTJ2K codec

3.1. High level overview

Figure 2 illustrates the elements of an HTJ2K encoding system, along with the corresponding decoder. HTJ2K substantially preserves the existing architecture and code-stream syntax of JPEG 2000. Imagery is first subjected to any re-quired multi-component transforms and/or non-linear point transforms, as allowed by Part-1 or Part-2 of JPEG 2000, after which transformed image components are processed by a reversible or irreversible Discrete Wavelet Transform (DWT), which decomposes each component into a hierarchy of detail sub-bands and one base (LL) sub-band.

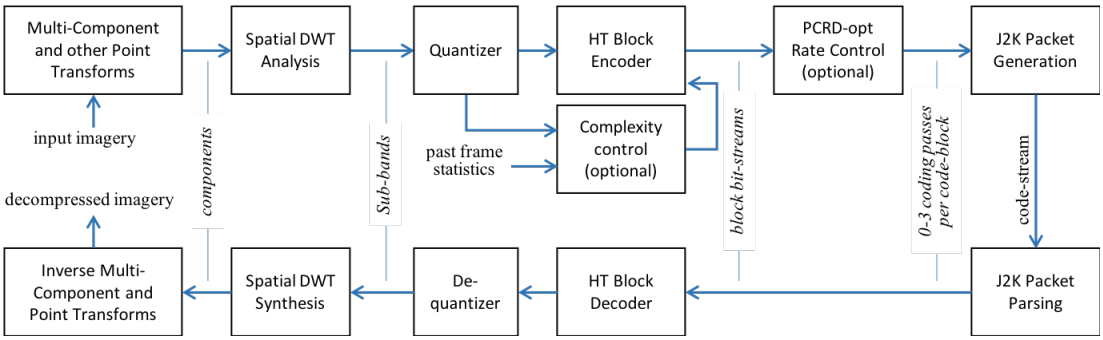


Figure 2: HTJ2K encoding system based on the FBCOT (Fast Block Coding with Optimized Truncation) paradigm.

All sub-bands are partitioned into blocks whose size is no more than 4096 samples, with typical dimensions being 64x64 or 32x32; very wide and short blocks such as 1024x4 are also important for low latency applications. Each block is individually quantized (if irreversible) and coded, producing a block bit-stream comprising zero or more coding passes.

In the encoder, an optional Post-Compression Rate-Distortion optimization (PCRD-opt) phase is used to discard generated coding passes so as to achieve a rate or distortion target, which may be global (whole code-stream) or local (small window of code-blocks).

Finally, the bits belonging to the selected coding passes from each code-block are assembled into J2K packets to form the final code-stream.

3.2. Relationship Between HT and J2K-1 Block Coding Passes

The J2K-1 block coder, as described in ITU-T Rec T.801 | IS 15444-1, processes the sub-band samples of each code-block through a sequence of coding passes. It is helpful to briefly revise this in order to explain the different, yet related coding pass structure of the HT block coding algorithm.

Let $x[\boldsymbol{n}]$ denote the samples within a code-block, indexed by location $\boldsymbol{n} = (n_1, n_2)$, where $0 \le n_1 < W$ represents horizontal position, $0 \le n_2 < H$ denotes vertical position, and W and H are the code-block's width and height, respectively. Each coding pass belongs to a bit-plane index $p \ge 0$, with respect to which the quantized magnitude of sample $x[\boldsymbol{n}]$ is given by

$$\mu_p[\boldsymbol{n}] = \left\lfloor \frac{|x[\boldsymbol{n}]|}{2^p \Delta} \right\rfloor$$

Here, Δ is the quantization step size, that is not present for reversible coding procedures.

We say that sample $x[\boldsymbol{n}]$ is "significant" with respect to bit-plane p if $\mu_p[\boldsymbol{n}] \neq 0$. The finest bit-plane (highest quality) corresponds to $p = 0$, while the coarsest quality corresponds to $p = K - 1$, where K is the maximum number of bit-planes for any code-block belonging to a given sub-band, determined by sub-band-specific parameters recorded in code-stream headers.

For each code-block a parameter μ_{start} is communicated via the relevant JPEG 2000 packet headers, which is interpreted as the number of *missing most significant bit-planes* in the code-block's representation. The J2K-1 block decoder is entitled to expect that all samples in the code-block are insignificant in bit planes $p \geq K - \mu_{\text{start}}$. Equivalently, the total number of bit-planes that may contain significant samples is given by

$$P_{lim} = K - \mu_{\text{start}}$$

J2K-1 processes sub-band samples in coding passes, known as Cleanup, Significance Propagation (SigProp) and Magnitude Refinement (MagRef). The first J2K-1 coding pass (a Cleanup pass) encodes the significance information and significant sample sign bits, for bit-plane $p = P_{lim} - 1$. For each successively finer bit-plane, three coding passes are produced. SigProp and then MagRef are refinement passes that raise the precision of only certain samples to the next bit-plane boundary, while the Cleanup pass brings the quality of all remaining samples in the code-block to the bit-plane boundary. This successive refinement process is illustrated in the upper part of **Figure 3**.

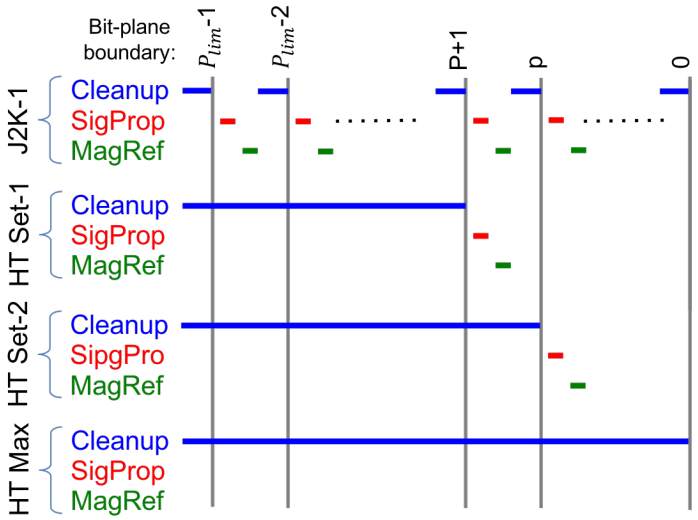


Figure 3: Bit-plane contributions of the coding passes produced by the J2K-1 and HT algorithms.

The HT block coding algorithm also adopts a coding pass structure, with Cleanup, SigProp and MagRef coding passes, defined with respect to bit-planes p. However, the Cleanup pass associated with each bit-plane p fully encodes the magnitudes $\mu_p[\boldsymbol{n}]$ and the signs of those samples $x[\boldsymbol{n}]$ for which $\mu_p[\boldsymbol{n}] \neq 0$. This information completely subsumes that associated with all previous (larger p) coding passes, so that there is no point in emitting all of them to the code-stream. The HT refinement passes, SigProp and MagRef, encode the same information as their J2K-1 counterparts, which allows a J2K-1 code-stream to be transcoded to the HTJ2K format without altering its quantized sample representation in any way. To fully recover this information, an HT block decoder must process at most one Cleanup, one SigProp and one MagRef pass, whereas a J2K-1 block decoder may need to process a large number of passes.

Figure 3 provides an illustration of the information contained within each HT coding pass, along with the information contributed by corresponding J2K-1 coding passes. The significant difference is the HT Cleanup pass, which encodes all information up to a given bit-plane boundary p. As indicated in the figure, HT coding passes can be collected into so-called "HT Sets". A complete HT Set consists of one Cleanup pass and the following SigProp and MagRef passes. An HT decoder processes the passes of only one set. The decoder would normally expect to find at most one HT Set for each code-block within the code-stream, but the HTJ2K standard does provide for the possibility of embedding multiple HT Sets.

For HT code-blocks, the μ_{start} value is derived from the parsing of JPEG 2000 packet headers, is no longer interpreted as the number of missing most significant bit-planes. $P_{lim} - 1 = K - \mu_{\text{start}} - 1$ does however retains its interpretation as the bit-plane index p of the first Cleanup pass for the code-block, whether it is a J2K-1 Cleanup pass or an HT Cleanup pass.

In both the J2K-1 and HT block coders, an encoder may drop any number of trailing coding passes from the information included in the final code-stream. Indeed, the encoder need not generate such coding passes in the first place if it can reasonably anticipate that they will be dropped. Strategies for doing this are discussed in [14].

With the HT block coder, both leading and trailing coding passes may be dropped (or never generated) by an encoder, so long as the first emitted coding pass is a Cleanup pass. As it turns out, it is usually sufficient for an HT encoder to generate just 6 coding passes, corresponding to two consecutive HT Sets, such as those identified in **Figure 3** as HT Set-1 and HT Set-2. Later, the PCRD-opt stage shown in **Figure 2** selects at most 3 passes of the generated coding passes from each code-block for inclusion in the final code-stream, where the selected passes belong to a single HT Set.

In some cases, there is no need for an encoder to generate more than a single HT Cleanup pass. This is certainly true for lossless compression, where only the Cleanup pass for $p = 0$ is of interest; this pass belongs to a degenerate HT Set, identified as the "HT Max" set in **Figure 3**, which can have no refinement passes.

During irreversible compression, the distortion associated with the HT Max set depends upon the quantization parameters that can be set to achieve a desired level of image quality, in exactly the same way that quantization is used to control compression in JPEG and most other media codecs. Section 3.4 explains the new Q-Factor for JPEG 2000, whose interpretation is designed to match that the JPEG Q-Factor that is typically used to control JPEG compression. Using this Q-Factor to drive HTJ2K compression, only one HT Cleanup pass need be generated for each code-block.

3.3. Rate control and complexity control for HTJ2K encoders

As suggested above, there are multiple ways for an HTJ2K encoder to compress an image or video source. The simplest approach is to generate just the single-pass HT Max set, managing the trade-off between image quality and compressed size by modulating quantization parameters, which is the subject of Section 3.4.

At the opposite extreme, an encoder can generate all possible HT coding passes – one HT Set for each significant magnitude bit-plane of each code-block – leaving the PCRD-opt rate control algorithm to determine an optimal point at which to truncate the quality of each code-block and then selecting the Cleanup, SigProp and MagRef passes (at most one of each). This extreme is wasteful of both computation and memory. In an optimized implementation, this approach is still computationally advantageous in comparison to the J2K-1 algorithm, by a substantial factor (e.g., 4 to 5 times faster), but the cost of temporarily buffering the coded data in memory is considerably higher than for J2K-1,

since multiple HT Cleanup passes contain redundant information. For reference, we refer to this as "HT-FULL" coding.

Between these two extremes, cost effective HTJ2K encoding strategies involve a complexity control module, as shown in **Figure 2**. The role of the complexity control module is to estimate the set of HT block coding passes that could be worth generating, considering an overall target compressed size or bit-rate. We use the term FBCOT (Fast Block Coding with Optimized Truncation) to refer to the combination of HT block coding, complexity control and PCRD-opt rate control.

Here we describe two different complexity control strategies. The first strategy was introduced in [15] as a strategy for high throughput video compression, using a precursor of the HT block coding algorithm. This strategy collects statistics from a previously encoded frame in the video sequence, keeping track of PCRD-opt truncation decisions that were made for each code-block in that frame. This information is used to estimate a range of truncation points that are likely to be of interest for the same code-block in the current frame, taking into account statistical variations and whether or not sufficient passes were encoded in the previous frame for the PCRD-opt algorithm to make good decisions. For future reference, we refer to this complexity control strategy as the "PCRD-STATS" method.

The second complexity control strategy is appropriate for both still image encoding and video compression. In this approach sub-band samples are quantized and analyzed online, as soon as they are produced by the DWT analysis machinery. These statistics are collected and used to form conservative estimates of the number of bytes that will be produced by the HT Cleanup pass associated with each potential bit-plane p. Based on these estimates, a preliminary rate control step determines generic code-block truncation points for each sub-band that will almost surely avoid exceeding the target compressed length, while respecting the visual or objective significance of each sub-band. These truncation points determine the first (coarsest) generated HT Cleanup pass for each code-block, but additional HT coding passes are generated to ensure the availability of sufficient coded content for the final PCRD-opt phase to achieve a near optimal rate allocation for the target compressed size or bit-rate. This strategy is employed, for example, in the Kakadu implementation mentioned in Section 1.5, where it is enabled via the "Cplex" parameter attribute – see Sections 7 and 9, where many of the command-lines used to obtain experimental results include "Cplex".

An important parameter to this "CPLEX" method is the delay that should be introduced between the point at which sub-band samples become available from the DWT analysis machinery and the point at which the HT block encoding of these same sub-band samples can proceed. At one extreme, all sub-band samples for the entire image or video frame are generated and buffered in memory first, before any block encoding occurs. At the opposite extreme, estimates can be formed as soon as the quantized sub-band samples for a row of code-blocks have been generated, allowing the block coding process to proceed immediately without any additional buffering. In this case the statistics of future sub-band samples, not yet produced by the DWT analysis machinery, must be forecast based on samples that have already been seen. Between these two extremes, of course, there is a range of possible delays, allowing the encoder to trade memory consumption for certainty about the statistics employed for complexity control. In practice, we find that the minimal delay, minimal memory variant of the CPLEX strategy produces excellent results, with coding efficiency usually very close to that which can be achieved using HT-FULL coding.

For ultra low latency applications, compressed data associated with small collections of code-blocks are subjected to rate allocation and emitted as soon as possible, rather than waiting until an image or frame has been completely encoded before the PCRD-opt

algorithm is used for rate allocation. In this case, since rate allocation is performed incrementally on small sets of code-blocks, the CPLEX method can determine the coding passes that block encoders should produce without the need for forecasts or significant additional memory.

The PCRD-STATS and CPLEX methods correspond to pure inter-frame and pure intra-frame complexity control strategies, respectively. For video coding, inter-frame methods can temporarily suffer in the presence of scene transitions or large motion. Intra-frame methods avoid these difficulties, but in their minimum memory configuration they can yield a non-ideal distribution of generated coding passes when the visual complexity of the coded content changes dramatically across an image or video frame. A combination of the two approaches could potentially be employed to further improve the performance of minimum memory complexity control.

Section 7 provides experimental results that demonstrate the effectiveness of the PCRD-STATS and CPLEX complexity control strategies. In both cases, there is little or no value in generating more than two HT Sets for each code-block. The resulting 6 passes have some redundancy, in that the second Cleanup pass contains all of the information found in the first 3 coding passes, but the consequent increase in memory consumption relative to J2K-1 is not all that large. Moreover, the PCRD-opt algorithm can be executed incrementally to regularly discard unwanted data and flush finalized code-stream content progressively, subject to the selection of a spatially progressive code-stream organization.

3.4. Q-Factor for JPEG 2000

As mentioned at the end of Section 3.2, the quality of JPEG 2000 compressed content depends both on the quantization applied to individual wavelet subbands and truncation that is applied at the code-block level. Quantization occurs prior to encoding, while truncation occurs after encoding. Both provide very similar operational rate-distortion properties, but with very different implications for system functionality and computational complexity.

The ability to achieve precise rate-control without iterative encoding, through post-compression rate-distortion optimization (PCRD-opt) is a well-known distinguishing feature of JPEG 2000, but it is less well-known that quality can also be controlled directly by choosing an appropriate set of quantization parameters, as is done in almost all other image and video codecs. Depending on the application, this may be beneficial, since it can avoid or limit the need to generate coding passes that are then discarded during the PCRD-opt step. For the HT block coder, in particular, the encoding workload can be reduced to the production of just one (or perhaps zero) HT Cleanup pass per code-block when quantization alone is used to drive the compressed image quality.

Until recently, there has been no well established method for choosing JPEG 2000 quantization parameters based on an intuitive “Quality Factor” (or Qfactor), as is commonly done with the traditional JPEG algorithm. This obstacle was substantially removed by a recent published study [16], which provided a recommended way to select JPEG 2000 subband quantization step sizes based on a Qfactor in the range 0-100, such that the subjective quality of the resulting imagery is roughly comparable to that of JPEG at the same Qfactor. This study considered only 8-bit 4:2:0 content and Qfactors up to 85, at which point subjective discrimination between original and compressed content becomes very difficult. Here we describe a more comprehensive approach that can be used to derive quantization step sizes for content with any sample precision, for a more general set of colour (or non-colour) representations, which works over the full range of Qfactor values, providing a sensible definition for what it means to achieve quality 100. The approach described here can be accessed via the “Qfactor” attribute of

the Kakadu implementation mentioned in Section 1.5; it is (or soon will be) also implemented by the OpenJPH software mentioned in Section 1.5.

It is well known that a near optimal trade-off between compressed bit-rate and Mean Squared Error (MSE) distortion results from the selection of quantization step sizes $\Delta_{b,c}$ for each subband b of component c, according to the rule

$$\Delta_b = \frac{\Delta_{\text{ref}}}{\sqrt{G_b G_c}},$$

where Δ_{ref} is a global quantization control parameter, $G_b = \|\boldsymbol{s}_b\|^2$ is the squared Euclidean norm of the synthesis basis vectors $\boldsymbol{s}_b$ associated with subband b of the relevant wavelet transform[5], and $G_c = \|\boldsymbol{m}_c\|^2$ is the squared Euclidean norm of the multi-component synthesis operator that represents the contribution of component c (e.g., Y, Cb or Cr) to reconstructed image samples (usually R, G and B)[6]. The quantities G_b and G_c are commonly known as energy gain factors for the subband and multi-component synthesis operations, respectively.

More generally, if the distortion objective is visually Weighted Mean Squared Error (WMSE) with weights $W_b \leq 1$ for each subband b, then the approximately optimal strategy is to assign

$$\Delta_b = \frac{\Delta_{\text{ref}}}{\sqrt{W_b G_b G_c}}$$

In this formulation, weights W_b are normally chosen to be 1 for the lowest frequency subbands of all image components, but smaller for higher frequency subbands where distortion can be harder to perceive. A well established set of visual weighting factors for 4:4:4 YCbCr content with sRGB primaries and gamma are given by **Table 1**. These are also suitable for RGB content compressed using the multi-component decorrelating transforms of JPEG 2000 Part-1. From these, one can easily derive suitable weighting factors for 4:2:0 and 4:2:2 content, as shown in **Table 2** and **Table 3**. These tables all provide parameters for 5 levels of wavelet decomposition only. If fewer levels of decomposition are employed, trailing columns in the tables are dropped, as appropriate; if more levels are used, the additional low frequency subbands should be assigned visual weights of 1.0. In all cases, the LL subband at the base of the decomposition tree should be assigned a visual weight of 1.0.

Table 1: Square roots $\sqrt{W_b}$ of the visual weighting factors for 4:4:4 YCbCr content

	HH_1	LH_1/HL_1	HH_2	LH_2/HL_2	HH_3	LH_3/HL_3	HH_4	LH_4/HL_4	HH_5	LH_5/HL_5
Y	0.0901	0.2758	0.7018	0.8378	1.0	1.0	1.0	1.0	1.0	1.0
Cb	0.0263	0.0863	0.1362	0.2564	0.3346	0.4691	0.5444	0.6523	0.7078	0.7797
Cr	0.0773	0.1835	0.2598	0.4130	0.5040	0.6464	0.7220	0.8254	0.8769	0.9424

[5] G_b can be obtained as the sum of the squares of the samples produced by synthesizing an image from exactly one non-zero sample, with magnitude 1, placed sufficiently far from the image boundaries within subband b.

[6] For the YCbCr to RGB transformation that also corresponds to the ICT in JPEG 2000 Part-1, $\sqrt{G_{\text{Y}}} = 1.7321$, $\sqrt{G_{\text{Cb}}} = 1.8051$ and $\sqrt{G_{\text{Cr}}} = 1.5734$.

Table 2: Square roots $\sqrt{W_b}$ of the visual weighting factors for 4:2:0 YCbCr content

	HH_1	LH_1/HL_1	HH_2	LH_2/HL_2	HH_3	LH_3/HL_3	HH_4	LH_4/HL_4	HH_5	LH_5/HL_5
Y	0.0901	0.2758	0.7018	0.8378	1.0	1.0	1.0	1.0	1.0	1.0
Cb	0.1362	0.2564	0.3346	0.4691	0.5444	0.6523	0.7078	0.7797	1.0	1.0
Cr	0.2598	0.4130	0.5040	0.6464	0.7220	0.8254	0.8769	0.9424	1.0	1.0

Table 3: Square roots $\sqrt{W_b}$ of the visual weighting factors for 4:2:2 YCbCr content – note that HL is horizontally high-pass and vertically low-pass, while LH is horizontally low-pass and vertically high-pass.

	HH_1	LH_1	HL_1	HH_2	LH_2	HL_2	HH_3	LH_3	HL_3	HH_4	LH_4	HL_4	HH_5	LH_5	HL_5
Y	0.0901	0.2758	0.2758	0.7018	0.8378	0.8378	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Cb	0.0863	0.0863	0.2564	0.2564	0.2564	0.4691	0.4691	0.4691	0.6523	0.6523	0.6523	0.7797	0.7797	0.7797	1.0
Cr	0.1835	0.1835	0.4130	0.4130	0.4130	0.6464	0.6464	0.6464	0.8254	0.8254	0.8254	0.9424	0.9424	0.9424	1.0

It remains to explain how the global reference step size Δ_{ref} is derived from a Qfactor Q in the range 0 to 100. To ensure that greyscale images receive the same quantization assignment as the luminance component of a colour image, we first express Δ_{ref} in terms of a quantity Δ_Q that depends only upon Q; specifically,

$$\Delta_{\text{ref}} = \Delta_Q \cdot \sqrt{G_{\text{Y}}},$$

where G_{Y} is the multi-component energy gain factor for the luminance component (usually the first one). Then Δ_Q is derived from the Qfactor Q, as follows:

$$\Delta_Q = 2^{P_0} \cdot \alpha_Q \cdot M_Q + \frac{1}{\sqrt{2}}, \text{ where}$$

$$M_Q = M(Q) \triangleq \begin{cases} \dfrac{50}{Q} & \text{if } 0 < Q < 50 \\ 2\left(1 - \dfrac{Q}{100}\right) & \text{if } 50 \le Q \le 100 \end{cases}$$

Here, P_0 is the sample data precision (bit-depth) of the original colour image samples or luminance component (usually all components have the same precision). The study of [16] recommended the selection of a constant value of 0.04 for the scaling factor α_Q, using the visual weights from **Table 2** with 4:2:0 YCbCr imagery at $P_0 = 8$ bits. Moreover, the study in [16] did not go past $Q = 85$ and did not include the offset $\frac{1}{\sqrt{2}}$ from the above expression, which is important as $Q \to 100$.

To provide sensible behaviour for large Qfactors, we recommend adjusting the visual weights W_b so that $W_b \to 1$ as $Q \to 100$. This makes a lot of sense, because distortion becomes very difficult to perceive at high Qfactors without heavily magnifying the images. Magnification invalidates the human visual system contrast sensitivity assumptions, on which the original weights were based.

It can be shown that adjusting the visual weights towards 1 also ensures that the MSE distortion converges to 50% of the natural distortion associated with digitizing continuous intensity values to the precision P_0. This is due to the constant offset of $\frac{1}{\sqrt{2}}$, which dominates the expression for Δ_Q as M_Q approaches 0. This then becomes the interpretation of lossy compression at "100% quality."

To compensate for the increase of visual weights W_b towards 1 at high Qfactors, it is also important to adjust the scaling factor α_Q. A good way to introduce all of these adjustments is through the following logarithmic model:

$$W_{b,Q} = \begin{cases} W_b & M_Q \le M_{T0} \\ W_b^{(\log M_{T1} - \log M_Q)/(\log M_{T1} - \log M_{T0})} & M_{T0} < M_Q < M_{T1} \\ 1 & M_Q \ge M_{T1} \end{cases}$$

$$\alpha_Q = \begin{cases} \alpha_{T0} & M_Q \le M_{T0} \\ \alpha_{T1} \cdot \left(\frac{\alpha_{T0}}{\alpha_{T1}}\right)^{(\log M_{T1} - \log M_Q)/(\log M_{T1} - \log M_{T0})} & M_{T0} < M_Q < M_{T1} \\ \alpha_{T1} & M_Q \ge M_{T1} \end{cases}$$

This model is parametrized by thresholds $M_{T0} < M_{T1}$ on the linear modulation parameter M_Q, over which the visual weights progress logarithmically from W_b to 1, while the quantization scaling factor progresses logarithmically from α_{T0} to α_{T1}. The following parameter choices are recommended:

$$\alpha_{T0} = 0.04, \qquad M_{T0} = M(65.0)$$
$$\alpha_{T1} = 0.10, \qquad M_{T1} = M(97.0)$$

This means that the visual weights take their nominal values at Qfactors below 65, while they complete their transition to uniform weights ($W_{b,Q}$=1) at a Qfactor of 97. One can verify that this choice results in the visual weights W_b being raised to powers of 0.86, 0.77, 0.66 and 0.49 at common quality factors of 75, 80, 85 and 90.

While the complete scheme described above has not yet been the subject of comprehensive subjective testing, informal visual inspection suggests that the visual quality associated with this approach is almost indistinguishable from the more limited approach in [16], for 8-bit 4:2:0 YCbCr content at quality factors below 85. Under these conditions, visual image quality can be expected to be broadly similar to that of traditional JPEG.

We conclude this section by pointing out that while the JPEG 2000 Qfactor can be used all by itself to control generated image quality, it can also be combined with the PCRD-opt algorithm to simultaneously ensure that the compressed image size or video bit-rate conforms to a strict bound. This can be done both with the original J2K-1 block coding algorithm or the HT algorithm of HTJ2K.

4. Introduction to the HT block coding algorithm

4.1. HT codeword segments and concurrency

Apart from the block coding algorithm itself, the HT block coder has no substantial impact on other JPEG 2000 code-stream constructs or their interpretation. Precincts, packets, tiles, transforms, and all other JPEG 2000 elements remain substantially unchanged, except in the so-called MIXED mode, where J2K-1 and HT block coding algorithms can both be used within a single tile-component.

In general, the construction of JPEG 2000 packets depends upon a set of codeword segments that are produced by the block coder. In its default mode, the J2K-1 block coder produces only one codeword segment, containing all coding passes, but there are other modes in which the coding passes are partitioned into multiple codeword segments. The significant property of a JPEG 2000 codeword segment is that the total number of bytes within the segment is explicitly encoded within packet headers.

For the HT block coder, each HT Set produces one or two codeword segments: the Cleanup pass produces its own HT Cleanup segment, while the SigProp and MagRef refinement passes, if present, contribute to an HT Refinement segment. **Figure 4** reveals the relationship between HT codeword segments and their constituent byte-streams. The Cleanup pass produces three separate byte-streams, which are packed into the Cleanup codeword segment. In addition to the segment length, L_{cup}, which is identified by JPEG 2000 packet headers, 12 bits are reserved at the end of the segment to store an interface locator word (ILW), which allows the boundary between the first and second byte-streams to be discovered by the decoder. The boundary between the last two byte-streams need not be explicitly communicated since the streams grow in opposite directions.

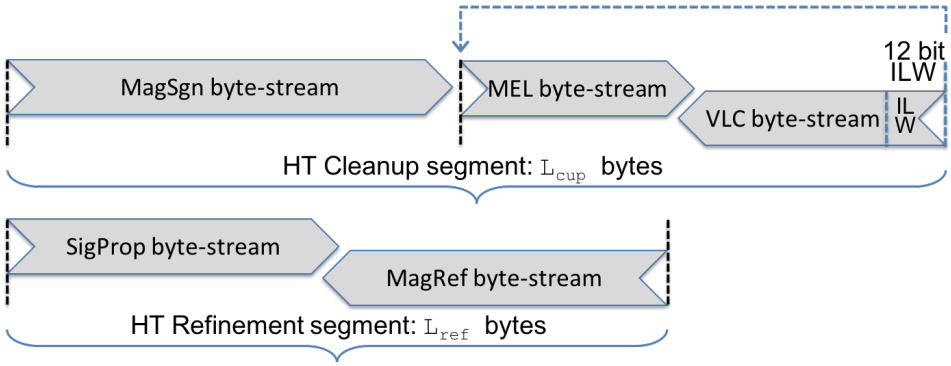


Figure 4: HT codeword segments and their constituent byte-streams.

The HT Refinement segment employs the same forward-backward strategy to avoid the need to explicitly signal the boundary between the forward growing SigProp byte-stream and the backward growing MagRef byte-stream. Based on the segment length L_{cup}, that is identified by JPEG 2000 packet headers, a decoder can recover both byte-streams, enabling concurrent processing of both the SigProp and MagRef passes. If the MagRef pass is not present, then the HT Refinement segment consists only of the forward growing SigProp byte-stream.

4.2. HT refinement passes

The HT SigProp and MagRef coding passes do not depend on each other; they depend only on the immediately preceding HT Cleanup pass. These encode the same

information and follow the same 4-line stripe-oriented scanning pattern as the corresponding J2K-1 coding passes, as shown in **Figure 5**.

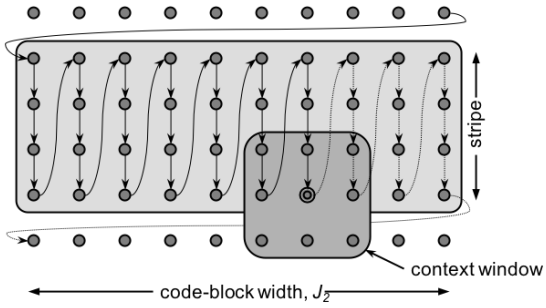


Figure 5: 4-line stripe-oriented scan followed by MagRef and SigProp coding passes.

All significance and associated sign information from the HT SigProp pass is emitted as raw binary digits and all magnitude refinement information from the HT MagRef pass is emitted as raw binary digits.

The HT MagRef pass is identical to that of the J2K-1 algorithm, operating in the BYPASS[7] mode, except that code bits are packed into bytes with a little-endian bit order, and the bits grow backwards within the HT Refinement segment to form the MagRef byte-stream.

The HT SigProp coding pass is also very similar to that of the J2K-1 block coder, operating in the BYPASS mode, with the two difference outlined below. As with J2K-1, the samples that are coded within the HT SigProp pass are those that have at least one significant neighbour within the context window shown in **Figure 5**, as coded within the Cleanup pass or previously within the same SigProp pass.

1. Code bits are again packed into bytes of the raw bit-stream with a little-endian bit order, whereas the J2K-1 block coder uses a big-endian bit packing order.
2. The significance bits associated with four consecutive stripe columns are emitted first, followed by the associated sign bits, before advancing to the next set of four stripe columns; by contrast, the J2K-1 SigProp pass inserts any required sign bit immediately after the same sample's magnitude bit.

These modifications together have implementation advantages over the methods of the J2K-1 algorithm, especially for software based implementations. We note that the second modification listed above is carefully crafted to allow accelerated decoding based on modest lookup tables.

4.3. Cleanup magnitude exponents, exponent bounds and MagSgn values

Each HT Set has an associated base bit-plane p, such that the HT Cleanup pass encodes magnitudes

$$\mu_p[\boldsymbol{n}] = \left\lfloor \frac{|x[\boldsymbol{n}]|}{2^p \Delta} \right\rfloor,$$

along with the signs $\chi[\boldsymbol{n}]$ of those samples for which $\mu_p[\boldsymbol{n}] \neq 0$, and the refinement passes, if present, encode the extra information required to refine $\mu_p[\boldsymbol{n}]$ to $\mu_{p-1}[\boldsymbol{n}]$ for certain samples in the code-block.

[7] More precisely, this is the arithmetic coder bypass mode of the J2K-1 algorithm, for which SigProp and MagRef symbols are emitted as raw bits, and codeword segment boundaries are inserted to separate the arithmetically coded Cleanup pass from the bypassed refinement passes.

To understand the Cleanup pass, it is helpful to introduce a binary significance flag

$$\sigma_p[\boldsymbol{n}] = \begin{cases} 0 & \text{if } \mu_p[\boldsymbol{n}] = 0 \\ 1 & \text{if } \mu_p[\boldsymbol{n}] \neq 0 \end{cases}$$

It is also helpful to introduce the concept of a magnitude exponent, which is defined as follows, noting that $\mathbb{N}$ is the set of natural numbers.

$$\epsilon_p[\boldsymbol{n}] = min\left\{\epsilon \in \mathbb{N} \left| \mu_p[\boldsymbol{n}] - \frac{1}{2} < 2^{\epsilon-1} \right.\right\}$$

Note that a sample is significant ($\sigma_p[\boldsymbol{n}] = 1$, or equivalently $\mu_p[\boldsymbol{n}] \neq 1$) if and only if $\epsilon_{\mathrm{p}}[\mathbf{n}] \neq 0$. The HT Cleanup algorithm explicitly codes significance information, after which it is only necessary to code the sign $\chi[\boldsymbol{n}]$ and the value of $\mu_p[\boldsymbol{n}] - 1$ for each significant sample. This information is combined into a so-called "MagSgn" value

$$v_p[\boldsymbol{n}] = \chi[\boldsymbol{n}] + 2(\mu_p[\boldsymbol{n}] - 1) < 2^{\epsilon_p[\boldsymbol{n}]}$$

In what follows, we drop the sub-script p, from all notation, so as to avoid excessively complicating the description.

Non-zero exponents are coded via so-called "unsigned residuals" with respect to a set of exponent predictors $\kappa[\boldsymbol{n}]$. The nature of this residual coding process is that the coded information may only provide a bound

$$U[\boldsymbol{n}] \geq \epsilon[\boldsymbol{n}]$$

U[$\boldsymbol{n}$] LSBs of the MagSgn value $v[\boldsymbol{n}]$ are then sufficient to completely describe the sign and magnitude of each significant sample.

The actual number of LSBs from each MagSgn value v[$\boldsymbol{n}$] that are emitted to the MagSgn byte-stream shown in **Figure 4** is given by

$$m[\boldsymbol{n}] = (U[\boldsymbol{n}] - t[\boldsymbol{n}]) \cdot \sigma[\boldsymbol{n}]$$

where $t[\boldsymbol{n}] \in \{0,1\}$ identifies the number of most significant (or "top") magnitude bits that are effectively communicated by context adaptive variable length coding techniques.

4.4. Main elements of the HT Cleanup pass

Figure 6 shows the main elements of the Cleanup pass encoding algorithm. Code-block samples are coded in 2x2 quads q, following a line-pair oriented scanning pattern. We use the notation $v_q[n]$ (and similar) to refer to the four samples of quad q, where $n = 0, 1, 2$ and 3 correspond to the top-left, bottom-left, top-right and bottom-right samples of the quad. A single exponent bound U_q is formed for the four samples in each quad q, so that

$$U_q \geq \epsilon_q[n], \qquad 0 \leq n \leq 3$$

The value of U_q is derived from the maximum magnitude exponent in the quad

$$\epsilon_q^{max} = \max\{\epsilon_{\mathrm{q}}[0], \epsilon_{\mathrm{q}}[1], \epsilon_{\mathrm{q}}[2], \epsilon_{\mathrm{q}}[3]\}$$

together with a (biased) exponent "predictor" κ_q that is formed from the magnitude exponents of the four samples immediately above quad q, as shown in red in **Figure 6**, along with the significance pattern $\boldsymbol{\rho}_q$. Specifically,

$$\mathrm{U_q} = \min\{\epsilon_q^{max}, \kappa_q\}$$

and $u_q = U_q - \kappa_q$ is an "unsigned residual", that can also be formed via a saturating subtraction, as shown in the figure. The value of U_q is communicated by encoding this unsigned residual u_q.

We also introduce a so-called "exponent max bound" (EMB) pattern $\boldsymbol{e}_q$, whose 4 bits $e_q[n]$ correspond to the most significant of the U_q bits for each $v_q[n]$, except where $u_q = 0$. That is,

$$e_q = \left\lfloor \frac{v_q[n]}{2^{U_q-1}} \right\rfloor \cdot \begin{cases} 0 & u_q = 0 \\ 1 & u_q > 0 \end{cases}, \qquad 0 \le n \le 3$$

You can think of $\boldsymbol{e}_q$ as the most significant magnitude bit-plane within quad q, except that it is possible (though rare) that the bound U_q is strictly larger than all four of the quad's magnitude exponents.

In addition to the EMB pattern $\boldsymbol{e}_q$, each quad q has a significance pattern $\boldsymbol{\rho}_q$, whose 4 bits are the significance flags $\sigma_q[n]$, and a 3-bit context c_q that is formed from the significance of three pairs of previously coded samples, as shown in green in the **Figure 6**.

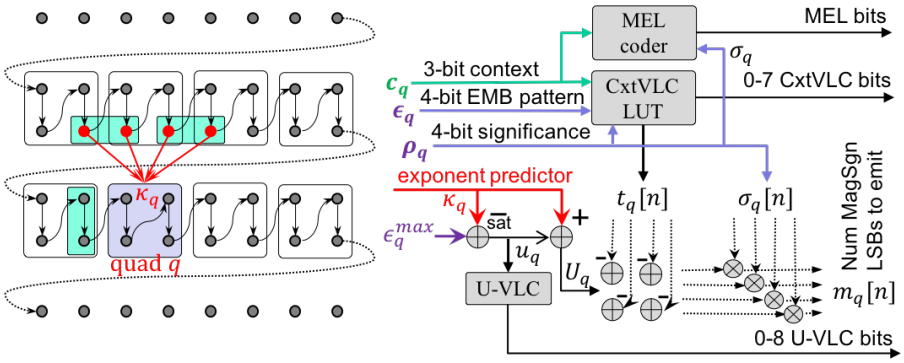


Figure 6: Main elements of the Cleanup encoding algorithm

The central element in the HT block encoder is the context-adaptive variable length coder (CxtVLC), which maps the context c_q, the EMB pattern $\boldsymbol{e}_q$ and the significance pattern $\boldsymbol{\rho}_q$, to a codeword with 0 to 7 bits that represents some or all of the information from $\boldsymbol{e}_q$ and $\boldsymbol{\rho}_q$ – that is, the HT CxtVLC represents a variable-to-variable length code. While it often happens that all EMB bits $e_q[n]$ can be represented by the emitted CxtVLC codeword for a quad, in some cases only some of the EMB bits are encoded this way, so the CxtVLC lookup table shown in **Figure 6** also emits a set of 1-bit counts $t_q[n]$ indicating the number of EMB bits from each sample that are identified by the codeword; all remaining bits from each SgnMag value $v_q[n]$ are emitted directly to the SgnMag byte-stream. There are $m_q[n] = (U_q - t_q[n]) \cdot \sigma_q[n]$ such remaining bits for each $n \in \{0,1,2,3\}$, noting that insignificant samples require no bits at all and the significance pattern $\boldsymbol{\rho}_q$ is always fully encoded by the CxtVLC codeword.

Quads whose context c_q is 0 are known as AZC (All-Zero Context) quads. For each AZC quad, a binary quad-significance symbol σ_q is encoded using an adaptive run-length coding state machine (MEL coder). If $\sigma_q = 0$, the entire quad is insignificant, which means that $\boldsymbol{\rho}_q = 0$ and $\boldsymbol{e}_q = 0$ and the CxtVLC codeword is empty (0 length). If $\sigma_q \neq 0$, the significance pattern $\boldsymbol{\rho}_q$ is known to be non-zero, so no codespace in the CxtVLC codebook for outcomes in which c_q and $\boldsymbol{\rho}_q$ are both 0. The MEL coder encodes AZC quad-significance symbols σ_q with similar coding efficiency to that of an adaptive arithmetic coder, but with only 13 states. This small number of states allows multiple symbols to be encoded concurrently using table lookup strategies in software.

In addition to the CxtVLC and MEL adaptive run-length coding tools, the unsigned residuals u_q are also subjected to variable length coding via the U-VLC shown in **Figure 6**. The U-VLC is a highly structured code that can be computed directly; alternatively, the U-VLC can be implemented using a table lookup approach. The value $u_q = 0$ yields a 0-length U-VLC codeword, since this event is always encoded as part of the CxtVLC codeword. Note that $u_q = 0$ if and only if the EMB pattern $\boldsymbol{e}_q = 0$.

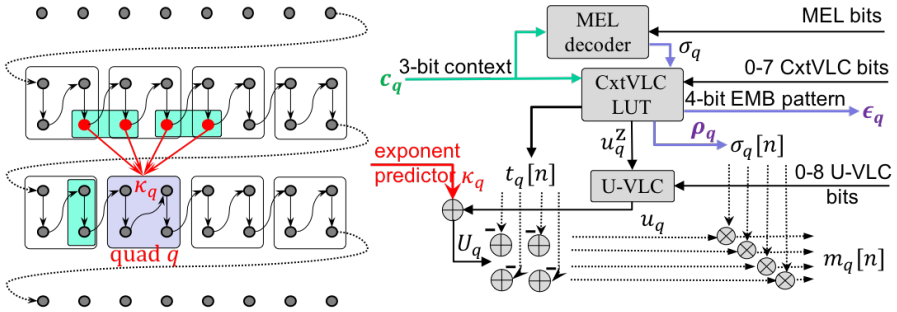


Figure 7: Main elements of the Cleanup decoding algorithm.

Figure 7 shows the main elements of the HT Cleanup pass decoder. Of course, the decoder has much in common with the encoder discussed above, but the CxtVLC decoding process deserves special attention. In the case of an AZC quad ($c_q = 0$), no CxtVLC bits are consumed unless the MEL decoder discovers that the quad is significant – i.e., $\sigma_q = 1$. The CxtVLC decoder deduces and emits a binary symbol u_q^z, which indicates whether or not $u_q = 0$. As noted above in the discussion of HT Cleanup encoding, the event $u_q = 0$ is always encoded as part of the CxtVLC codeword. If the decoded value of u_q^z is 0, then $u_q = 0$ and no bits are consumed by the U-VLC decoding process. Otherwise, u_q^z is 1 and the U-VLC decoding process consumes between 1 and 8 bits to discover the non-zero value of u_q, from which $U_q = u_q + \kappa_q$ is recovered.

It is worth reviewing the dependencies that exist between different aspects of the HT Cleanup decoding algorithm; these are illustrated in **Figure 8**. First observe that MEL decoding can proceed without any dependence upon VLC decoding or MagSgn bit unpacking processes. These other processes only affect the discovery of AZC quads, allowing the number of required MEL symbols σ_q and their associated quads q to be determined, but a MEL decoder is free to speculatively decode symbols in advance, consuming bytes in reverse from the end of the HT Cleanup segment.

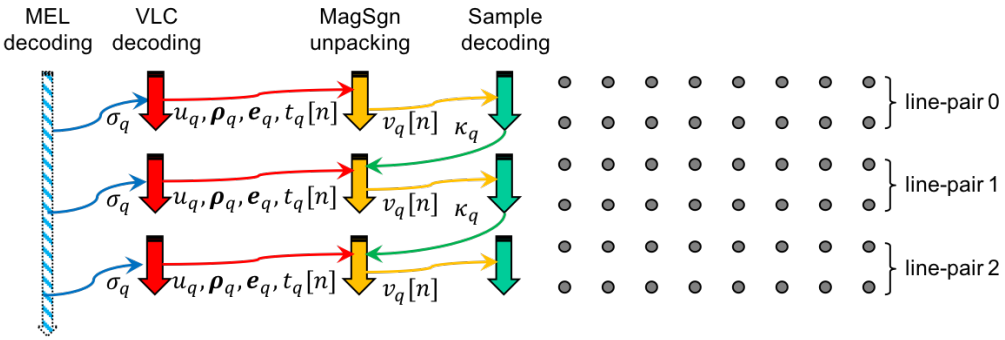


Figure 8: HT Cleanup decoding dependencies.

The VLC decoding processes (CxtVLC and U-VLC) can also be performed without any dependence upon MagSgn bit unpacking, so long as the decoded MEL symbols σ_q are available.

In order to determine the number of bits $m_q[n]$ to unpack from the MagSgn byte-stream for each sample within a given line-pair, it is necessary not only to decode all relevant MEL symbols and VLC codewords, but also to form exponents predictors κ_q, so that the exponent bounds U_q can be discovered by adding in decoded unsigned residuals u_q. The predictors κ_q cannot all be computed in advance, since they depend upon fully decoded sample values from the preceding line (the samples shown in red within **Figure 7**). However, the MagSgn unpacking process can be performed in a line-pair interleaved fashion. That is, after unpacking MagSgn bits for a given line-pair, sample values for the line-pair can be reconstructed, allowing predictors to be formed for the next line-pair and so forth.

For software and GPU deployments, this means that the recovery of decoded samples can be vectorized, exploiting vector widths up to $2J_2$, where J_2 is the nominal code-block width. In all common applications of JPEG 2000, nominal code-block widths are divisible by 32, allowing for processing in vectors with up to 64 samples each.

We finish this section by pointing out that the algorithmic descriptions provided here are not complete and not to be relied upon by implementors, who should consult the HTJ2K standard itself. In particular, we have not described the behavior of the HT Cleanup pass for the first line-pair of a code-block, where there no preceding line is available from which to form exponent "predictors" κ_q. Rather than simply taking κ_q to be 0 in the initial line-pair of a code-block, the HT Cleanup pass partially repurposes the MEL adaptive run-length coder to more efficiently communicate exponent bounds U_q. This is done in such a way as to ensure that no more than one pair of binary symbols is subjected to MEL coding for every quad-pair (octet), the same bound that applies within non-initial line-pairs. The initial line-pair also uses different contexts c_q and a different set of CxtVLC codewords.

4.5. Formation of the HT Cleanup and Refinement byte-streams

All of the HT byte-streams shown in **Figure 4** are formed using bit-stuffing procedures that insert an extra 0 bit, if required, to ensure that no pair of consecutive bytes in the coded data stream can form a 16-bit value in the range FF90 to FFFF. In addition to bit-stuffing, encoders are required to ensure that no HT codeword segment ends in an FF. These are fundamental requirements for all JPEG 2000 family files, which allows marker codes in this range to carry special meaning and also to be used for resynchronization after a transmission error.

Apart from this bit-stuffing operation, formation of the HT SigProp and MagRef Refinement byte-streams has been discussed already in Section 4.2. It is worth noting, however, that if the SigProp byte-stream becomes exhausted during decoding (i.e., the end of the HT Refinement codeword segment is reached), the decoding procedure synthesizes 0s rather than FF bytes. Unlike J2K-1, this allows a SigProp-only HT Refinement codeword segment to be truncated anywhere at all without introducing damaging visual artefacts, providing a useful mechanism for FBCOT to achieve exact rate control.

The HT MagSgn byte-stream is formed by packing $m_q[n]$ LSBs from the MagSgn values $v_q[n]$ of each quad's samples, in little-endian order, followed by bit-stuffing. Encoders discard any trailing FF from the MagSgn byte-stream, since decoders automatically append an FF to the byte-stream if it becomes exhausted.

The HT VLC byte-stream is formed by interleaving CxtVLC and U-VLC bits on a quad-pair (octet) basis. Specifically, the CxtVLC bits from both quads in a pair appear first, followed by 0-3 prefix bits from the U-VLC codewords of both quads, and then U-VLC suffix bits. This interleaving scheme allows a pair of U-VLC prefixes to be decoded

together using a modest lookup table, which is advantageous for software and GPU deployments; indeed in the vast majority of cases there are no U-VLC suffix bits at all. The interleaved VLC bit-streams are packed in little-endian order and subjected to bit-stuffing to form the VLC byte-stream.

The HT MEL byte-stream is formed by concatenating bits produced by the adaptive run-length MEL encoding state machine in big-endian order, again subject to a bit-stuffing procedure. The interface between the VLC and MEL byte-streams within an HT Cleanup segment is not explicitly signaled, so an encoder can arrange for the byte-streams to overlap, if desired, so long as this still results in correct decoding and does not introduce false marker codes in the range FF90 to FFFF.

5. Low complexity and high concurrency attributes of HTJ2K

The HT block coding algorithm has been carefully designed with software as well as hardware performance in mind. Most of the decisions taken in the development of the algorithm involved a careful software-algorithm co-design effort, in which the impact of each successive change was assessed separately with respect to CPU acceleration and coding efficiency.

From the hardware perspective, a major consideration in the development of the HT algorithm was to ensure a very high level of concurrency and parallelism. In particular, tight interleaving of coded bits has been studiously avoided, since this tends to introduce complex dependencies between algorithmic elements that may have different latencies.

In general, both hardware and software implementations of the encoding and decoding procedures can independently trade memory and latency for parallelism, without affecting inter-operability in any way.

5.1. Parallelism and concurrency

Software, GPU and hardware implementations of the HT block coder can all benefit from the macroscopic parallelism that comes from the use of independent code-blocks that is central to the JPEG 2000 framework. All implementations can trade memory and latency for parallelism using this mechanism, noting that a typical image or video frame has thousands of code-blocks, and large sources can have millions of them.

For a single code-block, all coding passes can be processed concurrently in an HT block encoder. Moreover, multiples sets of Cleanup, SigProp and MagRef passes (i.e., multiple HT Sets) can be generated concurrently by an encoder, if required.

HT block decoders can also process HT Cleanup, SigProp and MagRef passes concurrently, since their forward and backward growing byte-streams are simultaneously accessible within the HT Refinement segment shown in **Figure 4**. Note, however, that the decoding of SigProp and MagRef passes cannot commence until at least one full line-pair has been fully decoded within the Cleanup pass. If the CAUSAL mode is used, this delay is sufficient. Otherwise, two full line-pairs usually need to be decoded within the HT Cleanup pass before SigProp pass decoding can commence, due to the downward extent of the context window that drives significance propagation – see **Figure 5**. This limitation does not exist, however, for code-blocks of height 4 or less, which may be important for ultra-low latency applications.

The HT Cleanup pass itself provides considerable internal concurrency through the provision of separate MEL, VLC and MagSgn byte-streams. This allows concurrent processing of the MEL adaptive run-length decoding steps, the CxtVLC and U-VLC decoding steps, and the MagSgn bit-stream unpacking steps. As revealed by **Figure 8**, the only limitation to this concurrency, for decoders, is that MagSgn bit unpacking needs to be interleaved on a line-pair basis with the recovery of decoded sample values, due to the need to form exponent predictors for one line-pair from decoded samples in the preceding line-pair.

These low level concurrency features are extremely valuable for hardware implementation, but are also useful in software and GPU based deployments, since they allow encoders and decoders to process distinct phases of the algorithm in an order of their choosing, allowing register and execution threads to be deployed in a manner that is optimized to the type of work performed by each phase.

5.2. Encoding resources and vectorizability

A central feature of any high throughput software implementation is extensive exploitation of vector processing instructions. Modern x86 family CPUs support 256-bit vectors with up to 32 parallel byte-oriented operations through the AVX-2 instruction set. It is also desirable to be able to take advantage of the 512-bit vectors offered by AVX-512, and of course the almost ubiquitous 128-bit vector processing support offered by older x86 and current ARM processors. The HT block coding algorithm is able to take advantage of all such instruction sets.

Additionally, specialized instructions such as `popcnt` (count 1's in a register), `lzcnt` (leading zero bit count), `tzcnt` (trailing zero bit count), `pext` (parallel extraction of bits from a register based on a mask register), `pdep` (parallel deposit of bits to a register based on a mask register) are beneficial.

Almost all phases of the HT block encoder are amenable to vectorization. Broadly, the encoding steps are as follows:

1. Determination of magnitude exponents $\epsilon_q[n]$.
2. Formation of significance patterns $\boldsymbol{\rho}_q$, contexts c_q and EMB patterns $\boldsymbol{e}_q$.
3. Generation of exponent predictors κ_q, residuals u_q and exponent bounds U_q.
4. CxtVLC lookup to recover codewords and "top" bit counts $t_q[n]$.
5. U-VLC encoding, by direct computation or lookup
6. Computation of MagSgn bit counts $m_q[n]$ from U_q and $t_q[n]$.
7. Formation and concatenation of MEL symbols σ_q.
8. Formation of MagSgn values $v_q[n]$
9. Interleaving and bit-stuffing of CxtVLC and U-VLC bits to the VLC byte-stream.
10. Interleaving and bit-stuffing of MagSgn bits to the MagSgn byte-stream.
11. MEL adaptive run-length coding with bit-stuffing to the MEL byte-stream.
12. Membership propagation, packing and stuffing of significance and refinement bits to form the SigProp byte-stream.
13. Packing and stuffing of magnitude refinement bits to the MagRef byte-stream.

Remarkably, out of all these steps, the only ones that cannot be entirely vectorized are (4) and (11) – i.e., the CxtVLC lookup step and the MEL adaptive run-length coding procedure. In AVX-2 and AVX-512 architectures, the CxtVLC lookup can be vectorized using gather instructions, which are efficient but still rely upon sequential memory access at the hardware level. Also, even though the MEL coding procedure cannot really be vectorized, it can be accelerated in software by collecting symbols into runs and using lookup-based approaches that allow multiple runs to be encoded at once. Since the number of MEL runs in a code-block is usually small, the impact of MEL coding on overall throughput is very small. For hardware implementations, symbol-by-symbol implementations of the MEL encoder require very little logic and benefit from the fact that no more than two MEL symbols can be generated by any quad-pair (octet).

Apart from the memory required to buffer quantized sub-band samples produced by DWT analysis to form whole rows of code-blocks for encoding, the main additional resource is the lookup table required for CxtVLC coding. As suggested by **Figure 6**, CxtVLC encoding involves a lookup table with 2048 entries (11 index bits) formed from the quantities c_q, $\boldsymbol{\rho}_q$ and $\boldsymbol{e}_q$, with 14 bit entries: 7 codeword bits; 3 codeword length bits; and 4 "top" bit counts $t_q[n]$. This suggests the need for a 4kB lookup table with one access performed per 2x2 quad. In fact there are two such lookup tables, since a separate set of CxtVLC codewords are employed in the first line-pair of each code-block. It turns out, however, that the lookup tables are sparsely populated, with only 647 valid

entries each. This sparsity can be easily exploited in hardware implementations, and may also be exploited in GPU implementations to reduce access contention.

5.3. Decoding resources and vectorizability

Not surprisingly, decoding is harder to vectorize than encoding, since the decoder does not have access to all the sample values ahead of time. In the encoder, coding contexts, predictors and the like can be generated in parallel, while the decoder discovers the relevant quantities only during the decoding process itself. The same is true for the J2K-1 algorithm, for which decoder acceleration is known to be harder than encoder acceleration.

Considerable effort was invested in the design of the HT algorithm to facilitate high throughput decoding. Although dependencies inevitably exist between decoding steps, many things can still happen in parallel. The triple byte-stream structure of the HT Cleanup pass (see **Figure 4**) is important here, since it allows MEL symbols to be decoded ahead of time (or concurrently), whereupon VLC codewords can be decoded without any dependence on the adaptive MEL decoding process. All VLC decoding steps can be completed, if desired, before MagSgn bits are unpacked to complete the reconstruction of sample values. This is true, even though the number of bits $m_q[n]$ that appear for each sample in the MagSgn byte-stream depend upon previously unpacked MagSgn bits, because unsigned exponent residuals u_q are coded in a way that does not depend upon the exponent predictors κ_q themselves. These dependencies (or lack thereof) are conveniently illustrated in **Figure 8**.

Broadly, the decoding steps are as follows:

1. Bit unstuffing procedures.
2. Decoding of MEL symbols σ_q.
3. Decoding of CxtVLC and U-VLC codewords to recover $\boldsymbol{\rho}_q$, $\boldsymbol{e}_q$, $t_q[n]$ and u_q.
4. Formation of exponent predictors κ_q, bounds U_q and MagSgn LSB counts $m_q[n]$.
5. Reconstruction of MagSgn values $v_q[n]$.
6. Significance propagation and recovery of significance and sign bits from the SigProp byte-stream, if present.
7. Recovery of magnitude refinement bits from the MagRef byte-stream, if present.
8. Integration of Cleanup, SigProp and MagRef information to recover quantized sub-band samples $x[\boldsymbol{n}]$.

Of these, the steps that are not amenable to vectorization are (2), (3) and (6) – i.e., MEL decoding, CxtVLC and U-VLC decoding, and significance propagation. In practice, only about half of the code-blocks found in a lossy compressed code-stream usually include SigProp coding passes. Moreover refinement passes can always be discarded in a software environment if computational resources are under pressure. As already mentioned in the encoding context, MEL symbol decoding tends not to be expensive since MEL symbols can be processed in runs, lookup based approaches can be employed to decode multiple runs at once, and the number of MEL runs in a code-block tends to be small. For hardware deployments, symbol-by-symbol MEL decoding requires very little logic, and there can be no more than 2 MEL symbols per quad-pair (octet).

The main challenge for efficient implementation of the HT block decoder is on the inherently sequential CxtVLC and U-VLC decoding processes. The CxtVLC decoding step involves one table lookup per 2x2 quad, while decoding of the U-VLC prefix bits can be achieved either with a lookup table or by direct computation (simple logic in hardware). For lookup approaches, a single table lookup can decode the U-VLC prefixes for a pair of quads (octet) at once, after which the length of the full U-VLC codewords are known and the any U-VLC suffices can be extracted independently.

CxtVLC decoding involves a lookup table with 10 index bits: 3 context bits; and 7 prefix bits from the VLC byte-stream. Accordingly, a 2048 entry lookup table is required, with 16-bit entries: 3 bits for the codeword length; 4 bits for $\boldsymbol{\rho}_q$; 4 bits for $\boldsymbol{e}_q$; 4 "top" bit counts t_q[n]; and 1 bit for the u_q^z flag shown in **Figure 7**.

As for the encoder, two of these CxtVLC lookup tables are required, since the first line-pair of a code-block has a different set of codewords. In total, therefore, 8kB of lookup table resources are required for HT block decoding, with one lookup performed per 2x2 quad. In a hardware implementation, these lookup resources could be shared by multiple concurrent block decoding modules.

6. HTJ2K code-stream structure and features

An HTJ2K code-stream generally conforms to the same specifications as other code-streams from the JPEG 2000 family, building on the syntax and conventions defined in JPEG 2000 Part-1 [1]. The main differences are as follows:

1. The COD and COC marker segments that are used to signal default coding parameters and, if required, override those parameters on a tile-, component-, or tile-component basis, are extended to allow identification of the block coding algorithm as either J2K-1 or the HT algorithm.
2. It is also possible to identify MIXED mode encoding, where each code-block within a tile-component can use either the J2K-1 or HT block coding algorithm.
3. Packet header parsing procedures are updated to handle the signaling of HT codeword segments, in addition to the original J2K-1 codeword segments, while also providing the mechanism required to discover block coder type for each code-block in the MIXED mode mentioned above.
4. A CPF (Compatible Profile) marker segment is introduced to record the profile information of a compatible Part-1 code-stream, from which the HTJ2K code-stream might have been obtained by transcoding. The primary purpose of CPF is to allow profile information to be preserved during transcoding, so that it can be restored if the HTJ2K code-stream is transcoded to a Part-1 code-stream.
5. All Part-15 code-streams are required to include a CAP (Capabilities) marker segment that identifies the presence of Part-15 capabilities.
6. The Part-15 field within the CAP marker segment identifies an important global attribute of the code-stream, known as the “magnitude bound” (MAGB), which identifies the largest number of magnitude bit-planes that a decoder may need to process from any code-block in order to fully decode the content. The bound is signalled in a way that simplifies transcoding to different resolutions, for reversible and irreversible content. HTJ2K profiles for various applications in the future should specify maximum values for MAGB so as to maximize inter-operability.

Even though the HT block coder is not embedded, HTJ2K code-streams can fully retain the association between coding passes and quality layers that is foundational to quality scalable representations based on the J2K-1 algorithm. This information is retained via “HT placeholder passes” that can be signaled within JPEG 2000 packet headers. In this way, HTJ2K retains all the functionality of JPEG 2000 packets – a packet represents the contribution from a single quality layer to a given precinct, which is a spatially contiguous collection of code-blocks from a single resolution level of a tile-component.

JPEG 2000 Part-15 is compatible with the coding extensions found in JPEG 2000 Part-2 [3] and potentially other parts of the JPEG 2000 family of standards. This means that an HTJ2K code-stream can employ advanced multi-component transforms, non-linear point transforms, different numbers of horizontal and vertical levels of wavelet transformation, and many other advanced features that are useful for multispectral imaging, high dynamic range content, ultra-low latency coding, raw sensor data compression and much more.

7. Coding efficiency and throughput in software

7.1. Compression of professional photographic material

Here we compress 224 digital camera images, each having a resolution of 36.4 Megapixels, captured using a Sony A7R camera with Cannon USM F2.8 24-70mm lens. The photographs were captured using a range of aperture settings for stylistic effect, exhibiting both shallow and wide depths of field, with ISO gain settings in the range 200-800. All content was captured raw and converted to 16-bit/channel RGB TIFF images, such that the source images occupy 217MB each on disk.

In a first experiment, each photo is compressed using the Kakadu demonstration application `kdu_compress` to the JP2 format, employing the original J2K-1 block coding algorithm. The compressed images are then reversibly transcoded to JPH files, employing the HT block coding algorithm. The purpose of this exercise is to measure the coding efficiency of the HT block coder, in comparison to J2K-1. Results are presented in **Figure 9**.

As to the compression parameters, we employ 64x64 code-blocks, 8 levels of the CDF 9/7 wavelet transform (the irreversible transform defined in JPEG 2000 Part-1), and we drive the PCRD-opt rate control procedure with the visually weighted objective function that is generally preferred for JPEG 2000 content – this is the default visual weighting procedure employed by the Kakadu demonstration applications. It is worth noting that all compressed JP2 images have spatial random access enabled through the use of relative modest precinct dimensions (256x256) and inclusion of random access pointers. This, together with the 8 levels of wavelet decomposition, renders the compressed content extremely accessible for rapid preview and interactive exploration. All of these features are fully preserved by the transcoding to JPH files with embedded HTJ2K code-streams.

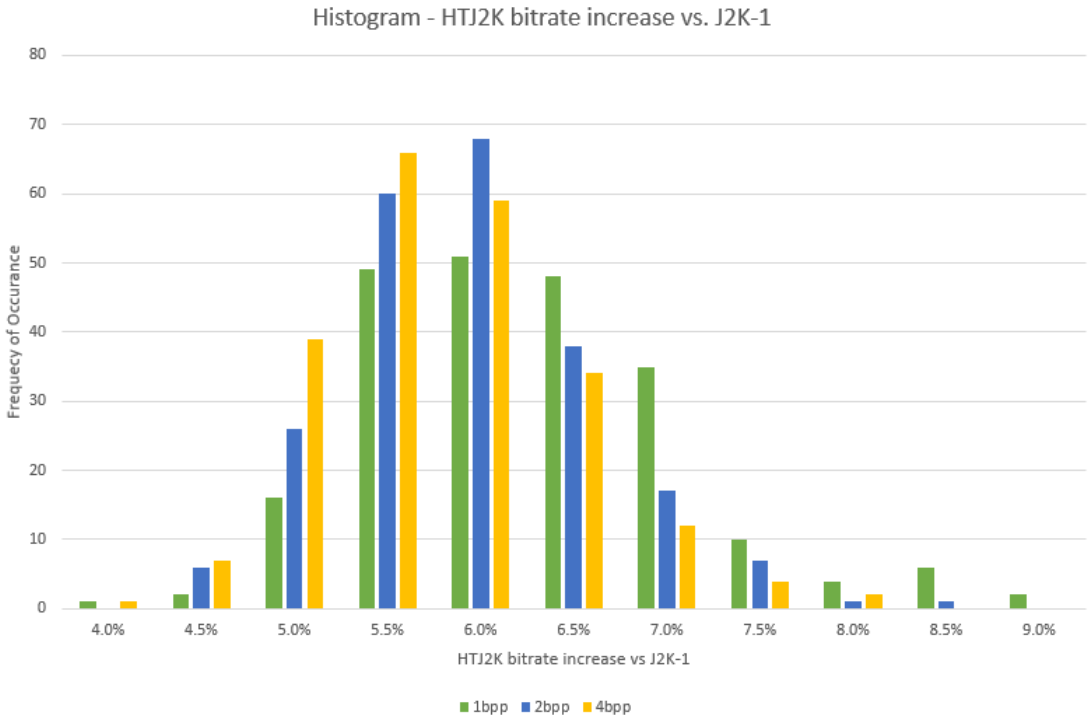


Figure 9: Relative increase in size when each of 224 x 36.4 Mpel photos is reversibly transcoded from JP2 to the JPH format, with an embedded HTJ2K code-stream that uses only HT code-blocks. The initial JP2 representation of each photo is encoded at 1 bits/pixel, 2 bits/pixel and 4 bits/pixel, resulting in size increase histograms for each case.

In a second experiment, each image is encoded directly to a JPH file whose HTJ2K code-stream uses only the HT block coding algorithm. Two encoding methods are employed, corresponding to the "HT-FULL" and "CPLEX" methods described in Section 3.3. For the CPLEX method, in this case, the block encoder is constrained to produce at most two HT-Sets (6 HT coding passes) per code-block. Moreover, a minimum delay (no extra memory) variant of the CPLEX algorithm is employed, with forecasting of the scene complexity statistics for as-yet unseen subband samples performed via spatial extrapolation with a governing set of fixed background (average) scene complexity statistics used to improve robustness. Encoding is performed using version 8.0.2 of the Kakadu demonstration application `kdu_compress`[8]. Decoding is achieved using the Kakadu demonstration application `kdu_expand`. The small set of background statistics are created once, using the `kdu_compress`[9] application.

Figure 10 through **Figure 12** show the PSNR values obtained from this experiment for each image at bit-rates of 1, 2 and 4 bits/pixel (bpp), using JPEG 2000 Part-1 (J2K-1) and HTJ2K in the HT-FULL and CPLEX configurations. Overall PSNR statistics are further summarized in **Table 4:** Average PSNR taken over all 224 x 36.4 Mpel photos, compressed using JPEG 2000 Part-1 (J2K-1) and HTJ2K, with HT-FULL and CPLEX configurations.. Observe that the impact of the CPLEX encoding complexity constraint during HTJ2K encoding is generally very small, costing less than 0.1dB on average.

Table 4: Average PSNR taken over all 224 x 36.4 Mpel photos, compressed using JPEG 2000 Part-1 (J2K-1) and HTJ2K, with HT-FULL and CPLEX configurations.

	1 bit/pixel			2 bits/pixel			4 bits/pixel		
	J2K-1	HT-FULL	CPLEX	J2K-1	HT-FULL	CPLEX	J2K-1	HT-FULL	CPLEX
PSNR (dB)	43.00	42.63	42.57	47.76	47.30	47.22	54.64	54.04	53.94

From **Figure 10** through **Figure 12**, it is possible to observe that there are a few outliers for which the CPLEX method introduces a significantly larger loss. In particular, 6 of the images exhibit a loss of 0.25dB or more. However, all of these images turn out to be extremely compressible, with PSNR values above 50dB even 1 bit/pixel, so that the impact of the CPLEX complexity control method on image quality is actually insignificant for all 224 images. The six outlier images are shown in **Figure 13** for completeness.

[8] For reference, the exact command-line used for the 2 bpp CPLEX controlled configuration is shown below. For the HT-FULL configuration, the "`-bstats`" argument is removed.

```
kdu_compress -i <im> -o out.jph Clevels=8 Qstep=0.001 -bstats bs.txt
```

[9] For reference, the commands used to collect background statistics in the "bs.txt" file are as follows:

```
kdu_compress -i <im_1> Cmodes=HT Clevels=8 Qstep=0.001 -bstats -,bs.txt
kdu_compress -i <im_2> Cmodes=HT Clevels=8 Qstep=0.001 -bstats bs.txt,bs.txt
...
kdu_compress -i <im_N> Cmodes=HT Clevels=8 Qstep=0.001 -bstats bs.txt,bs.txt
```

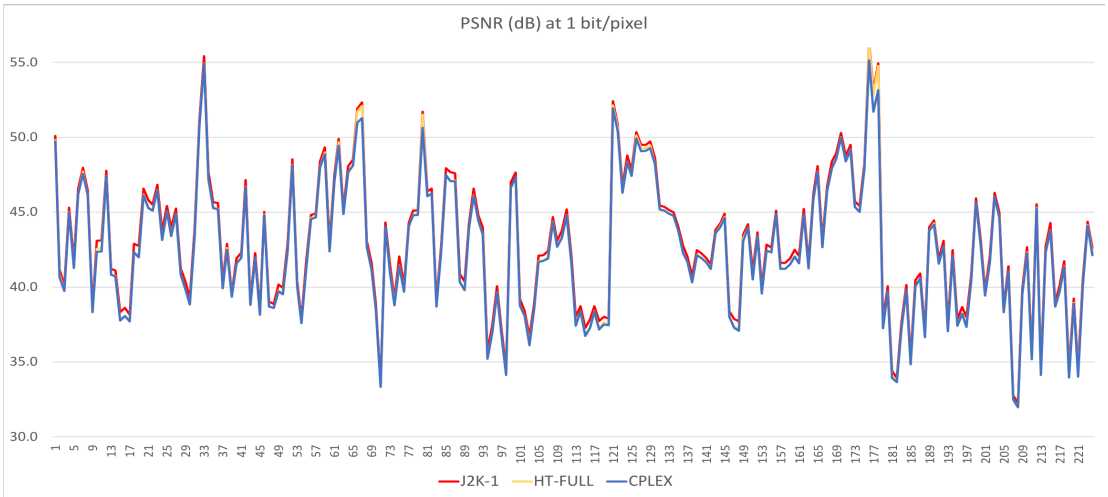


Figure 10: PSNR for each 36.4 Mpel photo after compression to JPEG 2000 Part-1 (J2K-1) and HTJ2K (HT-FULL and CPLEX methods) formats @ 1 bpp.

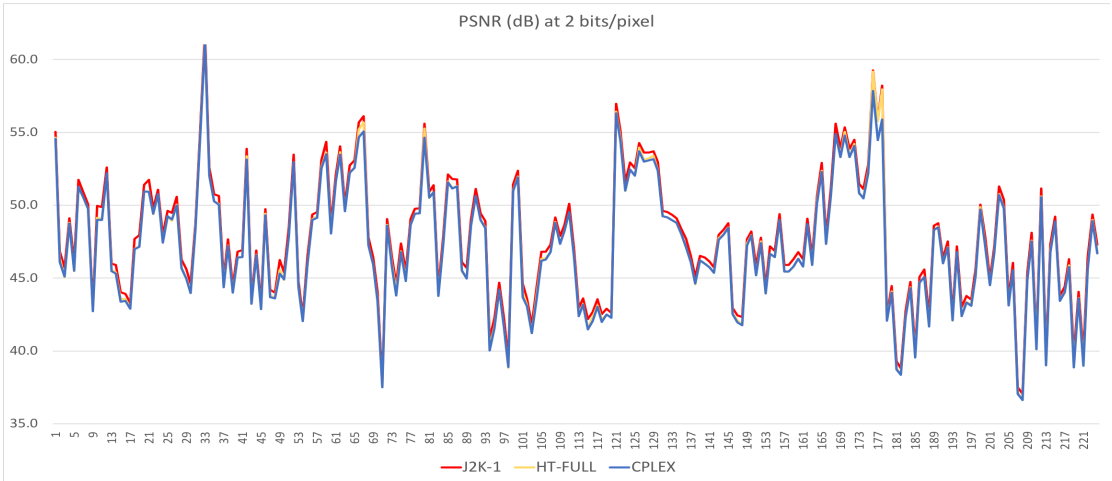


Figure 11: PSNR for each 36.4 Mpel photo after compression to JPEG 2000 Part-1 (J2K-1) and HTJ2K (HT-FULL and CPLEX methods) formats @ 2 bpp.

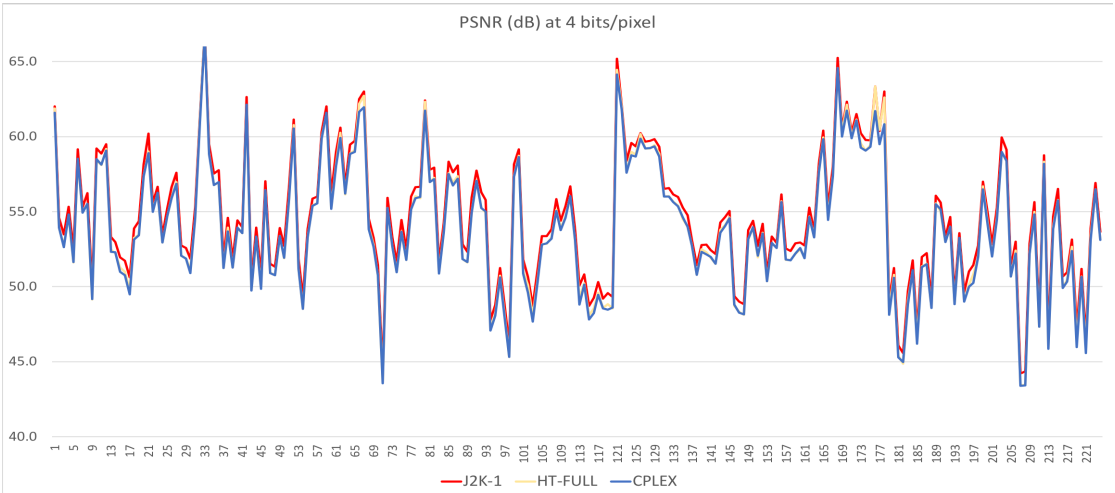


Figure 12: PSNR for each 36.4 Mpel photo after compression to JPEG 2000 Part-1 (J2K-1) and HTJ2K (HT-FULL and CPLEX methods) formats @ 4 bpp.



Figure 13: Six outlier images, for which the CPLEX complexity control algorithm that limits the number of HT-Sets to at most two per code-block results in a loss of 0.25dB or more in PSNR, relative to HT-FULL. In raster order, these six images have CPLEX losses of 0.25 dB, 0.74 dB, 0.84 dB, 0.88 dB, 1.0 dB and 1.6 dB.

7.2. 4K 4:4:4 12bit video encoding and decoding on an i7 4-core CPU

In this section, we explore the throughput of HTJ2K in the context of a video encoding application. For simplicity, in the current study we just use one of the standard JPEG test video sequences:

ARRI_AlexaDrums_3840x2160p_24_12b_P3_444

The sequence has 292 frames, at 60 frames/second, with 12 bits per RGB channel (36 bits per pixel), and a resolution of 3840 x 2160. There is no particular reason for selecting this sequence over any other, except that it is a good representative of professional digital video that has not previously been compressed. At the end of this section, however, we explore an artificial 4K video sequence that has been designed to be extremely challenging for high performance complexity constrained encoding.

We explore here the two FBCOT complexity control strategies identified in Section 3.3 as "PCRD-STATS" and "CPLEX", at three target bit-rates: 1 bit/pixel; 2 bits/pixel; and 4 bits/pixel. For reference, the 4K digital cinema standard specifies a maximum effective bit-rate of around 1.6 bits/pixel at these resolutions. We compare encoding and decoding throughputs with those achieved using the J2K-1 block coding algorithm, reporting both absolute frame rates and (in parentheses) the relative increase in throughput achieved by HTJ2K.

The results presented in **Table 5** through **Table 9** correspond to different sets of coding parameters, while the results within each table correspond to different complexity control procedures and bit-rates. All throughputs are obtained using an HP EliteDesk 800 G2 Small Form Factor PC (L1G76AV) with 2x8GB of DDR4 2133 MHz RAM, **an Intel i7-6700 4-core processor with 3.4GHz base frequency** and 4.0GHz max turbo frequency. Encoding is achieved using the Kakadu demonstration application `kdu_v_compress`[10]. Decoding is achieved using the Kakadu demonstration

[10] For reference, the exact command-line used for the 2 bpp CPLEX controlled configuration in **Table 5** is shown below. To remove disk I/O dependencies from throughput measurements, we add "`-frame_reps 64`" to these options. For PCRD-STATS complexity control, the "`Cplex`" option is replaced by "`-proc_limit 6,1,100`".

```
kdu_v_compress -i arri.vix -o arri.mj2 Corder=RPCL Clevels=6
Cblk={32,128} -rate 2 Qfix16=FREE Cmodes=HT -fastest
```

application `kdu_vex_fast`[11]. These employ a multi-threaded processing model (8 threads here) in which all threads work together on the same video frame, but spill across to the next frame as soon as all work for the current frame is exhausted.

For **Table 5** we employ a slightly unusual rectangular 128x32 code-block structure, since it helps reduce memory bandwidth by keeping most of the working data within the processor's caches. We also employ the irreversible 5/3 wavelet transform, with visual weighting employed in the rate and complexity control processes to yield what is usually considered to be a higher visual quality than unweighted MSE (equivalently PSNR) based rate control. Note that the irreversible 5/3 DWT requires the ATK (Arbitrary Transform Kernels) feature from JPEG 2000 Part-2 [3]. **Table 6** and **Table 7** are similar, except that they use 32x32 and 64x64 code-blocks, respectively.

Since **Table 5** corresponds to the fastest configuration, we also report absolute throughput in Giga-Samples/second (GS/s) there.

Table 8 is similar to **Table 7**, with 64x64 code-blocks, except that we use the more complex irreversible 9/7 DWT that is available in JPEG 2000 Part-1. In **Table 9** we explore the impact on throughput of modifying the optimization objective to unweighted MSE (PSNR). This tends to leave fewer all-zero code-blocks in the high frequency sub-bands.

Table 5: Irreversible 5/3 DWT with **visually weighted** rate allocation, with **128x32 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K-1 frames/s	DEC:J2K-1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	15.7	33.6	102 (6.5x)	185 (5.5x)	112 (7.1x) = **2.8 GS/s**	187 (5.6x) = **4.6 GS/s**
2 bpp	10.1	17.6	86 (10x)	148 (8.4x)	95 (9.4x) = **2.4 GS/s**	156 (8.9x) = **3.9 GS/s**
4 bpp	6.2	9.2	69 (11x)	112 (12x)	75 (12x) = **1.9 GS/s**	117 (13x) = **2.9 GS/s**

Table 6: 4K, 4:4:4 12b/channel video, **irreversible 5/3** DWT with **visually weighted** rate allocation, with **32x32 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K-1 frames/s	DEC:J2K-1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	12.8	32.5	71 (5.5x)	172 (5.3x)	94 (7.3x)	175 (5.4x)
2 bpp	8.8	17.1	62 (7.0x)	133 (7.8x)	76 (8.6x)	143 (8.4x)
4 bpp	5.7	8.9	53 (9.3x)	101 (11x)	59 (10x)	105 (12x)

```
-double_buffering 16 Qstep=0.001 Kkernels:I2=I5X3 Catk=2
Cplex={6,EST,0.25,0}
```

[11] The exact command-line used to measure decoding throughput is as follows:

```
kdu_vex_fast -i arri.mj2 -display -repeat 2
-double_buffering 20 -engine_threads 8 -read_ahead 4
```

Table 7: 4K, 4:4:4 12b/channel video, **irreversible 5/3** DWT with **visually weighted** rate allocation, with **64x64 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K-1 frames/s	DEC:J2K-1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	15.5	32.8	95 (6.1x)	152 (4.6x)	103 (6.6x)	148 (4.5x)
2 bpp	9.9	17.4	81 (8.2x)	139 (8.0x)	90 (9.1x)	143 (9.0x)
4 bpp	6.2	9.1	69 (11x)	109 (12x)	72 (12x)	112 (12x)

Table 8: 4K, 4:4:4 12b/channel video, **irreversible 9/7** DWT with **visually weighted** rate allocation, with **64x64 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K1 frames/s	DEC:J2K1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	15.1	31.5	86 (5.7x)	115 (3.7x)	85 (5.6x)	115 (3.7x)
2 bpp	9.6	16.9	74 (7.7x)	112(6.6x)	83 (8.6x)	115 (6.8x)
4 bpp	5.9	8.8	63 (11x)	96 (11x)	70 (12x)	102 (12x)

Table 9: 4K, 4:4:4 12b/channel video, **irreversible 9/7** DWT with **MSE(PSNR)-based** rate allocation, with **64x64 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K1 frames/s	DEC:J2K1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	13.2	31.2	81 (6.1x)	117 (3.8x)	84 (6.4x)	117 (3.8x)
2 bpp	8.5	16.7	69 (8.1x)	108 (6.5x)	76 (8.9x)	112 (6.7x)
4 bpp	5.3	8.6	57 (11x)	86 (10x)	64 (12x)	91 (11x)

Table 10 provides the results of a J2K1 to HT transcoding experiment, where the video stream is first compressed using the original J2K-1 block coding algorithm using the same parameters as in **Table 8** (irreversible 9/7 DWT with visually weighted rate allocation and 64 × 64 code-blocks), after which each frame is transcoded to use the HT block coder, and the frames are assembled into a new MJ2 file, for which the decoding throughput is measured. The first 3 columns of **Table 10** are naturally identical to those of **Table 8**, while the last two columns report the effect of transcoding on the average size of a compressed frame, and decompression throughput on the transcoded content.

Table 10: Transcoding of **4K, 4:4:4 12b**/channel ARRI_AlexaDrums_P3 from J2K-1 to HT, showing compressed size increase and throughputs, measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock. Compression parameters are the same as those in **Table 8**, so the J2K-1 columns are identical to those in that table.

bits/pel	ENC:J2K1 frames/s	DEC:J2K1 frames/s	HT Size Increase	DEC:HT frames/s
1 bpp	15.1	31.5	6.0%	118 (3.7x)
2 bpp	9.6	16.9	8.2%	115 (6.8x)
4 bpp	5.9	8.8	10.6%	96 (11x)

Table 11 provides PSNR results for the configurations whose throughput is reported in **Table 9**, so as to reveal the impact of HT encoding and complexity control on image quality. It is appropriate to compare PSNR values here, since the PCRD-opt and complexity control algorithms are configured to target MSE(PSNR) as the optimization objective. The "HT-FULL" column in the table is provided as a reference point for HTJ2K encoding without any complexity control.

Table 11: 4K, 4:4:4 12b/channel ARRI_AlexaDrums_P3, overall PSNR from mean squared error (MSE) over all channels and all 292 video frames, for the **irreversible 9/7** configurations with **MSE-based** rate control, for which throughputs are reported in **Table 9**. The headings "STATS-6p" and "CPLEX-6p" emphasize the fact that only 6 HT coding passes (two HT-Sets) are generated for each code-block in the PCRD-STATS and CPLEX complexity control strategies here.

	J2K-1	HTJ2K	HTJ2K	HTJ2K
bits/pel	J2K-1 blocks	HT-FULL	STATS-6p	CPLEX-6p
1.0 bpp	41.41 dB	40.97 dB	40.96 dB	40.92 dB
2.0 bpp	45.54 dB	44.90 dB	44.90 dB	44.83 dB
4.0 bpp	51.18 dB	50.12 dB	50.18 dB	50.10 dB

Table 11 suggest that "PCRD-STATS" is very slightly better than "CPLEX" for image quality, despite being a little slower. However, these results do not tell the full story concerning the behavior of various complexity control strategies, both for J2K-1 and HTJ2K, since the Arri Alexa video sequence employed here has scene complexity that is reasonably consistent across both space and time.

To tease out the behavior of different approaches, we construct an artificial 4K 4:4:4 RGB video sequence consisting of 48 frames with 5 scene cuts (6 segments), alternating between two different types of content. Four of the six segments are much easier to compress than the other two, with substantial amounts of overcast sky in the upper half of the picture, so that the scene complexity also varies strongly from the top to the bottom of the frame. The content is compressed to four different bit-rates, 1 bpp, 2bpp and 4bpp, with constant rate control, so that each frame has the same compressed size. PSNR traces, from the average mean squared error in each frame across the R, G and B channels, are plotted in Figure 14 through Figure 16.

The compression options for these figures are essentially identical to those used for Table 11, except that we add the configuration "J2K1-A" which means that "`-accurate`" is added to the "`kdu_v_compress`" command-line associated with the "J2K1" configuration. The "`-accurate`" option disables Kakadu's slope threshold prediction feature, where the distortion-length slope threshold for a frame is estimated from that used in previous frames, and the estimated slope is used to terminate the block encoding process early, if appropriate, rather than generating all possible coding passes [14]. Disabling this feature significantly increases the encoding complexity for J2K-1, but avoids a penalty in the transition from frames with high scene complexity to low scene complexity (low to high PSNR), as evidenced by the brief departure of the orange and dark blue traces in the figures. Apart from this, we remove "`Qfix16=FREE`" from all encoding command-lines and use "`Qstep=0.0001`".

Observe that the "STATS-6p" option ("PCRD-STATS" complexity control) exhibits significant performance loss at scene transitions, especially in transitions from low to high scene complexity (high to low PSNR). This is because it assumes that local scene complexity will not change too rapidly from frame to frame, so it can wind up skipping too many coarse HT Sets in a high complexity frame that follows a low complexity one.

By contrast, the "CPLEX-6p" option (green trace) provides uniformly high quality that is almost identical to "HT-FULL." In fact, "CPLEX-6p" is even more effective than "HT-FULL" in handling the high to low complexity transitions, since even the "HT-FULL" algorithm relies upon the slope threshold prediction strategy mentioned above. As for J2K1, this can be disabled with the "`-accurate`" option, but at the expense of significantly lower throughput.

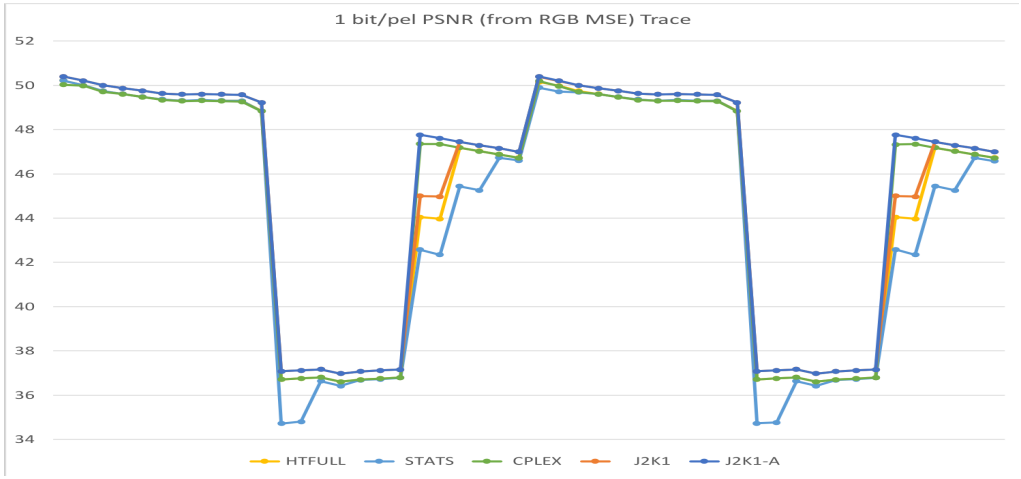


Figure 14: Exploration of complexity control strategies @ 1bpp, on video with abrupt changes in scene complexity, composed of 6 segments, 4 of which have very low complexity in the upper half of the frame.

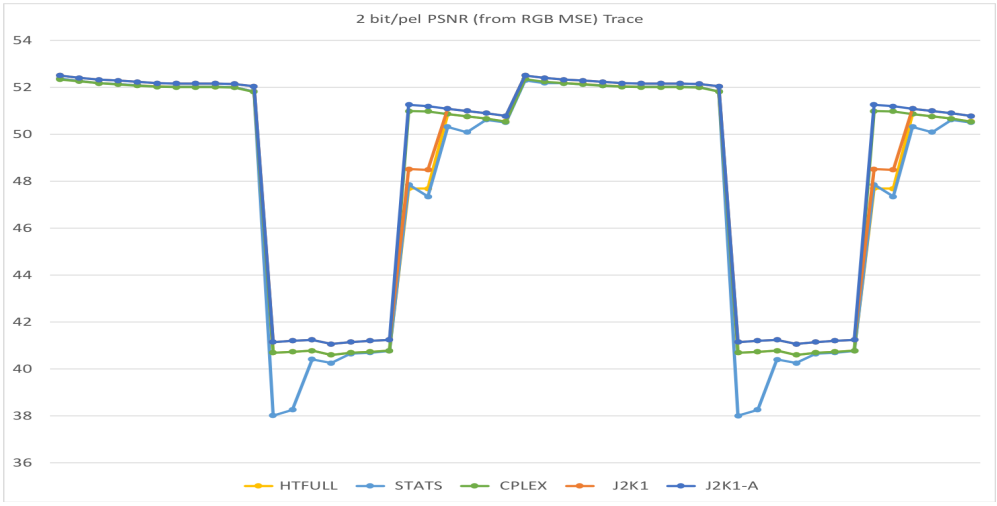


Figure 15: Exploration of complexity control strategies @ 2bpp, on video with abrupt changes in scene complexity, composed of 6 segments, 4 of which have very low complexity in the upper half of the frame.

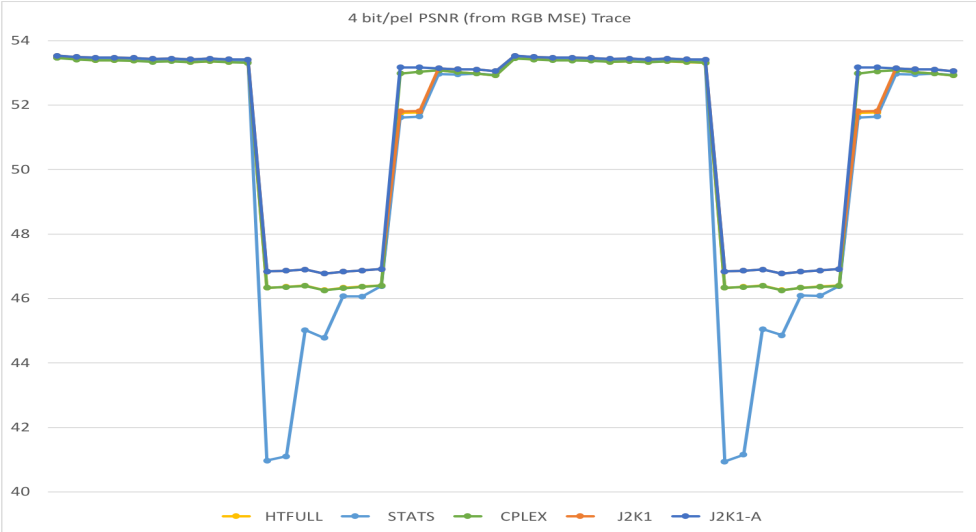


Figure 16: Exploration of complexity control strategies @ 4bpp, on video with abrupt changes in scene complexity, composed of 6 segments, 4 of which have very low complexity in the upper half of the frame.

In summary, **the "CPLEX" complexity control strategy provides excellent robustness to temporal and spatial variations in scene complexity for video encoding**, while exhibiting high coding efficiency **and the highest throughput** of all methods explored. Moreover, it appears to be quite sufficient to generate at most 6 coding passes (2 HT Sets) for each code-block.

Table 12 provides additional throughput results for special cases in which PCRD-opt rate control is not required. One such case is truly lossless compression; the other is irreversible (i.e., lossy) compression in which the quantized sub-band samples are fully encoded. As explained in Section 3.2, this corresponds to the generation of just one HT-Set, the so-called HT-Max set, for which there is just one HT coding pass, the $p = 0$ Cleanup pass – see **Figure 3**. The table reports absolute throughputs (frames per second), the HT speedup relative to J2K-1, and lossless coding efficiency (expressed as bits/sample).

Table 12: 4K, 4:4:4 12b/channel ARRI_AlexaDrums_P3, with quantizer-driven rate allocation and lossless (**one pass HT-Max set**). Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

Conditions	ENC:J2K1 frames/s	DEC:J2K1 frames/s	ENC:HT frames/s	DEC:HT frames/s
Irreversible 5/3 128x32 code-blocks @ 2 bpp	15.8	17.4	123 (7.8x) = **3.1 GS/s**	126 (7.2x) = **3.1 GS/s**
Irreversible 9/7 64x64 code-blocks @ 2 bpp	15.2	16.7	90 (5.9x) = **2.2 GS/s**	111 (6.6x) = **2.8 GS/s**
Reversible 5/3 128x32 code-blocks @ lossless	2.11 (6.1 bits/sample)	2.10 (6.1 bits/sample)	78 (37x) = **1.9 GS/s** (6.6 bits/sample)	66 (31x) = **1.6 GS/s** (6.6 bits/sample)
Reversible 5/3 64x64 code-blocks @ lossless	2.11 (6.1 bits/sample)	2.10 (6.1 bits/sample)	71 (34x) = **1.8 GS/s** (6.5 bits/sample)	66 (31x) = **1.6 GS/s** (6.5 bits/sample)

7.3. 4K 4:2:2 10bit video encoding and decoding on an i7 4-core CPU

In this section, we perform a similar investigation to that of Section 0, except that we work for 4:2:2 video content with 10 bit/channel precision. Since the total number of samples is less by a factor of 1.5x, we expect to achieve higher frame rates with HTJ2K, whose throughput depends primarily on the number of samples that need to be processed. Additionally, content with 10 bit precision can be processed with a lower internal working precision, especially with more complex transforms such as the CDF9/7, which reduces memory bandwidth pressure.

For simplicity, in the current study we just use one standard JPEG test video sequences:

EBU_PendulusWide_3840x2160p_50_10b_bt709_422

The sequence has 748 frames, at 60 frames/second, with 10 bits per channel (20 bits/pixel with 4:2:2 sampling) and a resolution of 3840 x 2160.

We repeat the same set of experiments as for the 4:4:4 12bit/channel content in Section 0, using the same Kakadu encoding commands for all configurations. For decoding, we use the following command in all cases. Results are reported in **Table 13** through **Table 19**.

```
kdu_v_expand -i ebu.mj2 -fastest -double_buffering 20
```

Not surprisingly, the speedups for HTJ2K relative to J2K-1 are all larger for 4:2:2 content, because the effective number of coded bits per original sample is higher than in the 4:4:4 case.

Table 13: 4K, 4:2:2 10b/channel video, **irreversible 5/3** DWT with **visually weighted** rate allocation, with **128x32 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K-1 frames/s	DEC:J2K-1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	16.3	35.1	132 (8.1x)	241 (6.9x)	132 (8.1x) = 2.2 **GS/s**	245 (7.0x) = 4.1 **GS/s**
2 bpp	10.6	18.5	110 (10x)	176 (9.5x)	110 (10x) = 1.8 **GS/s**	192 (10x) = 3.2 **GS/s**
4 bpp	6.7	9.3	92 (14x)	129 (14x)	91 (14x) = 1.5 **GS/s**	137 (15x) = **2.3 GS/s**

Table 14: 4K, 4:2:2 10b/channel video, **irreversible 5/3** DWT with **visually weighted** rate allocation, with **32x32 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K-1 frames/s	DEC:J2K-1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	13.8	34.3	91 (6.6x)	218 (6.4x)	111 (8.0x)	229 (6.7x)
2 bpp	9.5	18.2	77 (8.1x)	155 (8.5x)	86 (9.1x)	176 (9.7x)
4 bpp	6.2	9.2	66 (11x)	113 (12x)	70 (11x)	123 (13x)

Table 15: 4K, 4:2:2 10b/channel video, **irreversible 5/3** DWT with **visually weighted** rate allocation, with **64x64 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K-1 frames/s	DEC:J2K-1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	16.1	34.6	127 (7.9x)	209 (6.0x)	130 (8.1x)	214 (6.2x)
2 bpp	10.4	18.5	107 (10x)	166 (9.0x)	109 (10x)	181 (9.8x)
4 bpp	6.6	9.3	88 (13x)	123 (13x)	89 (13x)	131 (14x)

Table 16: 4K, 4:2:2 10b/channel video, **irreversible 9/7** DWT with **visually weighted** rate allocation, with **64x64 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K1 frames/s	DEC:J2K1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	15.8	33.1	107 (6.8x)	155 (4.7x)	103 (6.5x)	157 (4.7x)
2 bpp	10.3	18.2	98 (9.5x)	141 (7.7x)	101 (9.8x)	147 (8.1x)
4 bpp	6.5	9.3	83 (13x)	111 (12x)	94 (14x)	118 (13x)

Table 17: 4K, 4:2:2 10b/channel video, **irreversible 9/7** DWT with **MSE(PSNR)-based** rate allocation, with **64x64 code-blocks**. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

bits/pel	ENC:J2K1 frames/s	DEC:J2K1 frames/s	ENC:STATS frames/s	DEC:STATS frames/s	ENC:CPLEX frames/s	DEC:CPLEX frames/s
1 bpp	13.5	34.2	105 (7.8x)	147 (4.3x)	99 (7.3x)	148 (4.3x)
2 bpp	9.6	18.4	88 (9.2x)	135 (7.3x)	98 (10x)	137 (7.4x)
4 bpp	6.2	9.2	78 (13x)	102 (11x)	87 (14x)	109 (12x)

Table 18: Transcoding of **4K, 4:2:2 10b**/channel EBU Pendulus Wide from J2K-1 to HT, showing compressed size increase and throughputs, measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock. Compression parameters are the same as those in **Table 16**, so the J2K-1 columns are identical to those in that table.

bits/pel	ENC:J2K1 frames/s	DEC:J2K1 frames/s	HT Size Increase	DEC:HT frames/s
1 bpp	15.8	33.1	6%	156 (4.7x)
2 bpp	10.3	18.2	6%	146 (8.0x)
4 bpp	6.5	9.3	6%	112 (12x)

Table 19: 4K, 4:2:2 10b/channel EBU_PendulusWide, overall PSNR from weighted mean squared error (MSE) over all channels and all 748 video frames, for the **irreversible 9/7** configurations with **MSE-based** rate control, for which throughputs are reported in **Table 17**. Relative MSE weightings are **1:0.5:0.5** for the luminance and chrominance channels, corresponding their sampling densities in the **4:2:2** format considered here. The headings "STATS-6p" and "CPLEX-6p" emphasize the fact that only 6 HT coding passes (two HT-Sets) are generated for each code-block in the PCRD-STATS and CPLEX complexity control strategies here.

bits/pel	J2K-1 J2K-1 blocks	HTJ2K HT-FULL	HTJ2K STATS-6p	HTJ2K CPLEX-6p
1.0 bpp	41.25 dB	41.01 dB	41.02 dB	41.02 dB
2.0 bpp	44.77 dB	44.18 dB	44.22 dB	44.17 dB
4.0 bpp	52.20 dB	51.50 dB	51.50 dB	51.44 dB

Table 20 provides additional throughput results for truly lossless compression. The table reports absolute throughputs (frames per second), the HT speedup relative to J2K-1, and lossless coding efficiency (expressed as bits/sample).

Table 20: 4K, 4:2:2 10b/channel EBU_PendulusWide lossless compression (**one pass HT-Max set**). Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

Conditions	ENC:J2K1 frames/s	DEC:J2K1 frames/s	ENC:HT frames/s	DEC:HT frames/s
Reversible 5/3 128x32 code-blocks @ lossless	3.75 (5.0 bits/sample)	3.76 (5.0 bits/sample)	125 (33x) **= 2.1 GS/s** (5.4 bits/sample)	107 (28x) **= 1.8 GS/s** (5.4 bits/sample)
Reversible 5/3 64x64 code-blocks @ lossless	3.76 (5.0 bits/sample)	3.76 (5.0 bits/sample)	121 (32x) **= 2.0 GS/s** (5.4 bits/sample)	108 (28x) **= 1.8 GS/s** (5.4 bits/sample)

7.4. Compression of large aerial imagery

Here we consider the compression of a single image that is much larger than the photographic images investigated in Section 7.1. The image in question is an aerial photograph that measures 13333x13333, having RGB pixels with 24 bits each (8 bits/sample), occupying 533MB on disk.

We compress the image to various bit-rates, using MSE(PSNR) as the optimization objective for the PCRD-opt procedure. Code-block dimensions are 64x64 and the CDF9/7 wavelet transform is employed, along with the usual irreversible decorrelating colour transform (RGB to YCbCr in this case). We choose PSNR as the optimization objective so that the various compression strategies can be easily compared in terms of PSNR. In particular, we compare the performance of JP2 compression, based on the original J2K-1 block coding algorithm, with that of JPH compression, where the embedded HTJ2K code-streams uses the HT block coding algorithm exclusively. All

compression and decompression are performed using the Kakadu demonstration applications `kdu_compress`[12] and `kdu_expand`.

We compare the performance associated with "Full" HT encoding against that obtained with CPLEX-based complexity-control, as explained in Section 3.3, with minimum delay (no extra memory) and various numbers of HT coding passes. PSNR results for these different approaches are reported in **Table 21**. Evidently, for this type of content at least, it is (more than) sufficient for the HTJ2K encoder to produce 2 HT-Sets (6 coding passes) per code-block.

Table 21: Compression of a large (0.5 GB) RGB aerial image to the JP2 and JPH formats, with various levels of HT encoder complexity control, corresponding to 3, 4 and 6 coding passes per code-block, followed by unweighted (PSNR-based) PCRD-opt rate control.

bits/pel	JP2 J2K-1 blocks	JPH HT-FULL	JPH CPLEX-3p	JPH CPLEX-4p	JPH CPLEX-6p
0.5 bpp	32.47 dB	32.29 dB	32.00 dB	32.29 dB	32.29 dB
1.0 bpp	34.92 dB	34.62 dB	34.50 dB	34.62 dB	34.62 dB
2.0 bpp	38.90 dB	38.48 dB	38.30 dB	38.48 dB	38.48 dB
3.0 bpp	42.67 dB	42.20 dB	41.93 dB	42.16 dB	42.16 dB
4.0 bpp	46.09 dB	45.54 dB	45.35 dB	45.53 dB	45.53 dB

These different encoding strategies have very different throughputs. Decode throughput is obtained using the `kdu_buffered_expand` demonstration application, applied directly to the content produced by JP2 and JPH HT-FULL encodings. To evaluate the true throughput of the different encoding strategies carefully, it is important to separate out the cost of disk access. To do this, we use the `kdu_buffered_compress` demonstration application with 8-fold in-memory vertical replication of the image. Additionally, we employ an incremental code-stream flushing strategy so that the single-threaded PCRD-opt stage of the encoding pipeline can run in the background, rather than being deferred until the very end. This approach is more consistent with video compression, where the PCRD-opt and code-stream flushing stages for a previous code-stream run to completion in the background while the main processing operations for a current frame proceed in the foreground. Using this strategy[13], we obtain the encoding throughputs reported in **Table 22**. The platform used for this evaluation is the Intel i7-6700 PC described in Section 0.

[12] For reference, the exact command-line used for JPH encoding at 2 bpp is shown below. For JP2 encoding, we simply change the output file suffix. To generate the last 3 columns of **Table 21**, we add "`Cplex={3,EST,0.0,0}`", "`Cplex={4,EST,0.25,0}`" and "`Cplex={6,EST,0.25,0}`", respectively.

```
kdu_compress -i phoenix.ppm -o tmp.jph -rate 2 -tolerance 0
Qstep=0.001 -quiet -no_weights
```

[13] For reference, the exact encoding command-line used for JPH HT-Full encoding at 2 bpp is shown below. For JP2 encoding, we remove "`Cmodes=HT`". For CPLEX-constrained throughput we add "`Cplex={6,EST,0.25,0}`".

```
kdu_buffered_compress -i phoenix.ppm -vrep 8 -flush_period
1024 -rate 2 -cpu Qstep=0.001 Corder=PCRL Cmodes=HT
Cprecincts={256,1024},{256,1024},{128,1024},{64,1024},{32,1024}
```

Table 22: JP2 and JPH throughput results for a large (0.5 GB) RGB aerial image, with and without CPLEX-based complexity control. Throughputs are measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock and expressed in Mega-Samples/second (MS/s).

bits/pel	DEC:JP2	DEC:JPH	ENC:JP2	ENC:JPH HT-FULL	ENC:JPH CPLEX-6p	ENC:JPH CPLEX-3p
0.5 bpp	1028 MS/s	1804 MS/s	428 MS/s	1778 MS/s	1960 MS/s	1910 MS/s
1.0 bpp	724 MS/s	1782 MS/s	318 MS/s	1594 MS/s	1958 MS/s	1920 MS/s
2.0 bpp	424 MS/s	1756 MS/s	205 MS/s	1260 MS/s	1815 MS/s	1989 MS/s
3.0 bpp	288 MS/s	1724 MS/s	163 MS/s	1068 MS/s	1712 MS/s	1972 MS/s
4.0 bpp	217 MS/s	1631 MS/s	131 MS/s	931 MS/s	1612 MS/s	1978 MS/s

7.5. Compression of Entertainment Imagery

A short called Meridian produced using a modern ACES production workflow with high-quality visual effects and post-production techniques was released by Netflix under the "Creative Commons" license. Small thumbnails from every 1000-frames of the full 17345-frame sequence are shown in **Table 23**. Different forms of the Meridian short were used for the compression tests reported in this section.

Table 23: Thumbnails of Meridian short

 frame 87000	 frame 88000	 frame 89000
 frame 90000	 frame 91000	 frame 92000
 frame 93000	 frame 94000	 frame 95000

<table>
<tr><td>
frame 96000</td><td>
frame 97000</td><td>
frame 98000</td></tr>
<tr><td>
frame 99000</td><td>
frame 100000</td><td>

frame 101000</td></tr>
<tr><td>

frame 102000</td><td>

frame 103000</td><td></td></tr>
</table>

7.5.1. 3840x2160p 10bit ITU-R BT.709 4:4:4 23.976p SDR

The 17345 frames of the Meridian short were compressed at 400 Mbs and 800 Mbs Lossy rate using JPEG 2000 Part-1 as well as using Lossless JPEG 2000 Part-1. The compressed results were then losslessly transcoded to HTJ2K (using Kakadu's transcoding demo application `kdu_transcode`), ensuring that the same exact image would be decoded in the corresponding formats. The size of each JPEG 2000 Part-1 and Part-15 equivalent are show in Figure 17 and Figure 18. Table 24 summarizes the average increase in data rate for the equivalent Part-15 version.

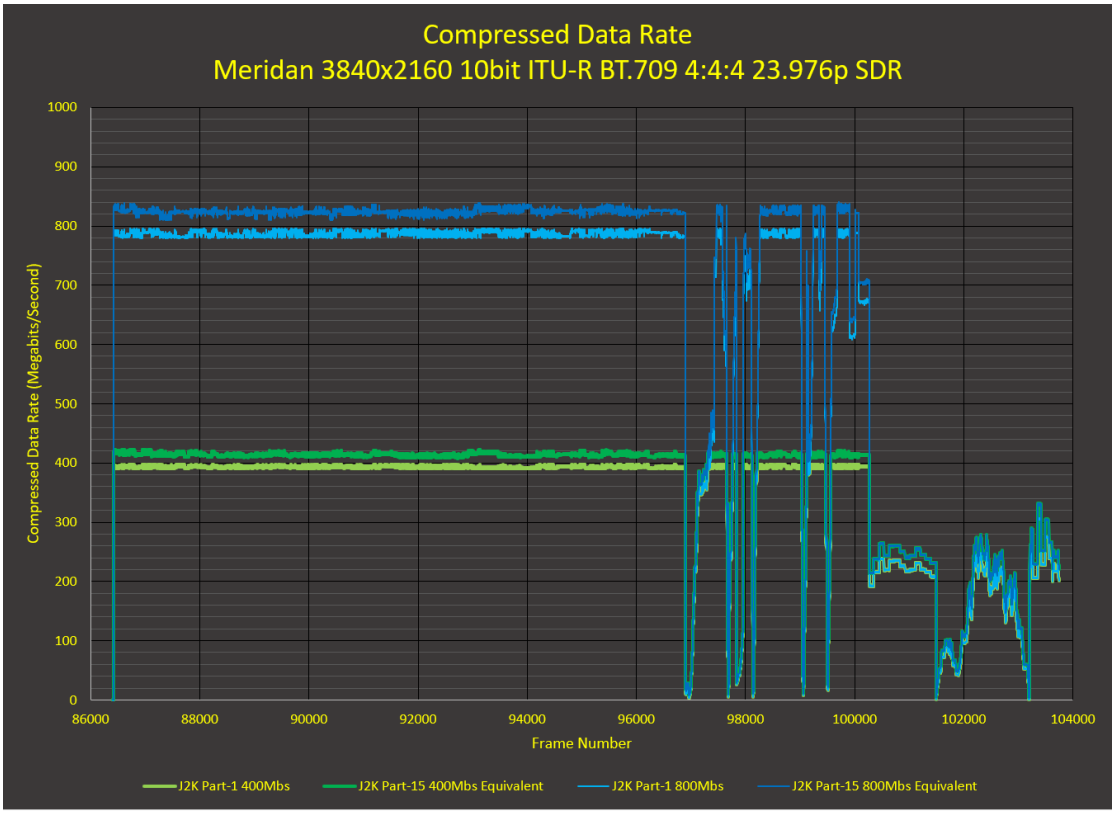


Figure 17: Meridian UHD SDR BT.709 Lossy Compressed Data Rate

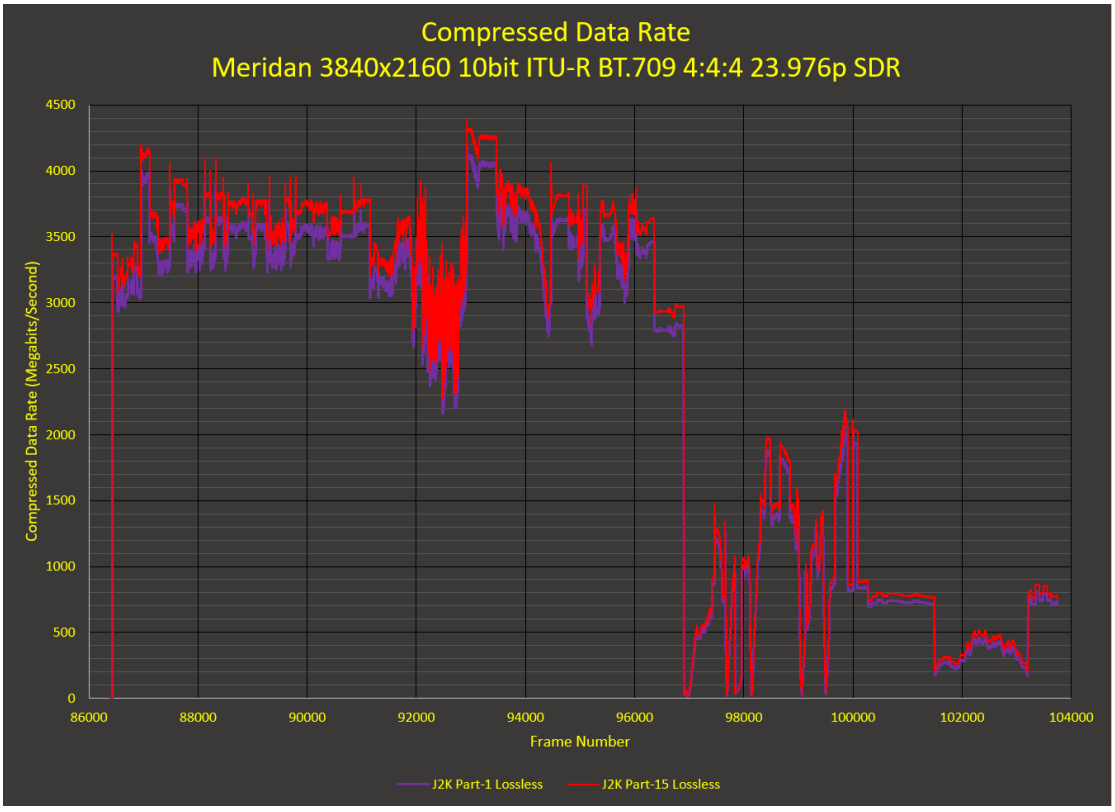


Figure 18: Meridian UHD SDR BT.709 Lossless Compressed Data Rate

Table 24: Meridian UD SDR BT.709 Average % Increase in Size over JPEG 2000 Part-1

	J2K Part-15 400Mbs Equivalent	J2K Part-15 800Mbs Equivalent	J2K Part-15 Lossless
Average % Increase in Size over JPEG 2000 Part-1	6.00%	5.16%	5.41%

A 10-second portion of the Meridian short (frames 90000-90239) was used to perform Decoding Throughput tests on a MacBook Pro 15-inch 2018 2.9GHz Intel Core i9 (6-core) 16GB RAM 1TB SSD. Kakadu's high-throughput decoding demo application `kdu_vex_fast` was used for the decoding of both Part-1 and Part-15 content, the results are summarized in Table 25.

Table 25: Meridian UHD SDR BT.709 Decoding Throughput

Test Format	Decoding Frame Rate	Effective Storage Read Rate (Megabytes/Second)	HTJ2K decoding speedup factor
Part-1 Lossy 400Mbs	24	47	
Part-15 Lossy 400Mbs Equiv.	111	228	5x
Part-1 Lossy 800Mbs	11	44	
Part-15 Lossy 800Mbs Equiv.	91	374	9x
Part-1 Lossless	2	43	
Part-15 Lossless	70	1303	30x

Note that the HTJ2K Lossless effective storage read rate was 1303 Megabytes/Second, which is approaching the test platform's state-of-the-art consumer NVMe SSD theoretical performance limit of approximately 2500 Megabytes/Second.

7.5.2. 3840x2160p 12bit 24p DCinema DCDM SDR

The BT.709 SDR version was converted to 12bit and 16bit Digital Cinema Distribution Master (DCDM) format (X'Y'Z' 2.6 Gamma) and compressed losslessly using both JPEG 2000 Part-1 and Part-15. The size of each JPEG 2000 Part-1 and Part-15 equivalent are show in Figure 19.

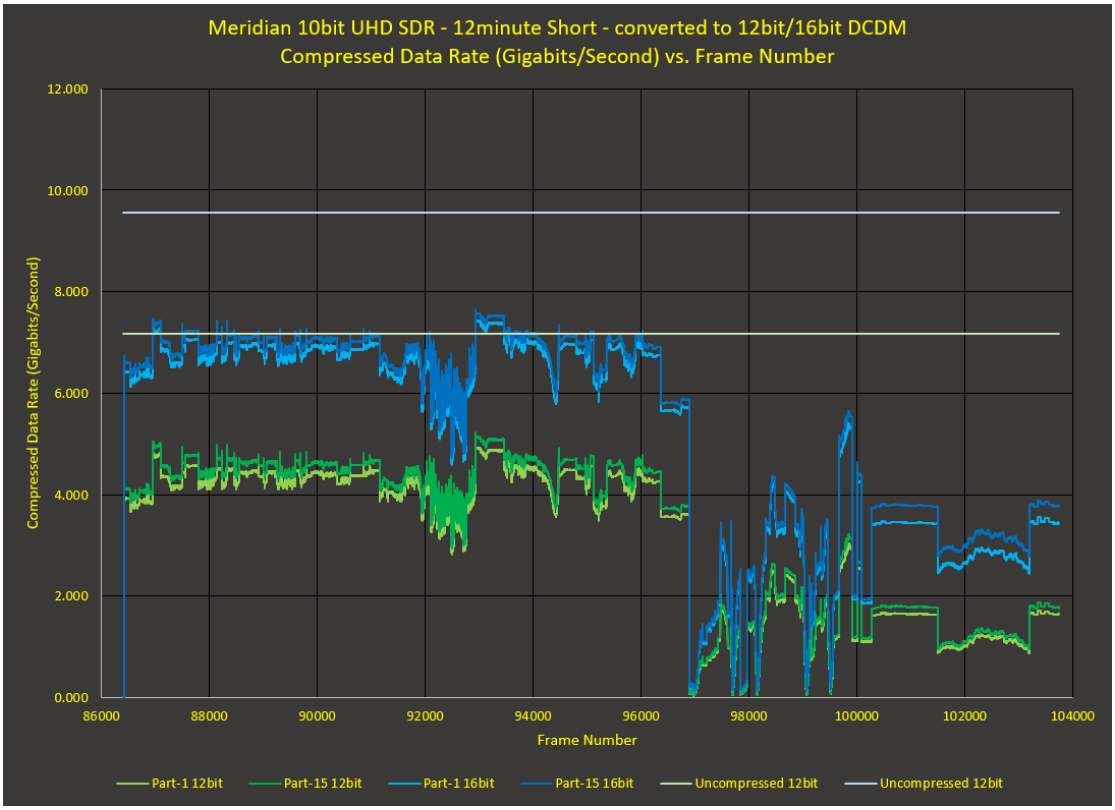


Figure 19: Meridian UHD DCDM Lossless Compressed Data Rate

A 10-second portion of the Meridian short (frames 90000-90239) was used to perform Decoding Throughput tests on a MacBook Pro 15-inch 2018 2.9GHz Intel Core i9 (6-core) 16GB RAM 1TB SSD. Kakadu's high-throughput decoding demo application `kdu_vex_fast` was used for the decoding of both Part-1 and Part-15 content, the results are summarized in Table 26. Additionally, Table 26 summarizes the average increase in data rate of the Part-15 version compared to the Part-1 version.

Table 26: Meridian UHD DCDM Decoding Throughput and HTJ2K bitrate increase

Test Format	**Decoding Frame Rate**	**HTJ2K Decoding Speedup Factor**	**HTJ2K bitrate increase**	**Average Compression Ratio**
12bit Part-1 Lossless	2			2.3:1
12bit Part-15 Lossless	67	34x	5%	2.2:1
16bit Part-1 Lossless	1.5			1.9:
16bit Part-15 Lossless	40	26x	4%	1.8:1

7.5.3. 3840x2160 24p 16bit half-float OpenEXR (Scene-Linear HDR)

The Graded Archive Master (GAM) of the SDR version of Meridian was also used for additional compression and throughput. The native format of the GAM was ACES OpenEXR uncompressed, which is a 16-bit half-float format, with 5 exponent bits and 11 mantissa bits. The range of values that are representable in this data format have a real-value range of [+65504.0, -65504.0], in addition to the non-real numbers +Inf, -Inf and NaN (Not A Number). Part-1 and Part-15 Lossless compression were used in addition to the Lossless compression formats offered by the OpenEXR v2.40 library called PIZ, ZIP and ZIPs. The resulting lossless compression ratio of each frame was computed and is shown in Figure 20.

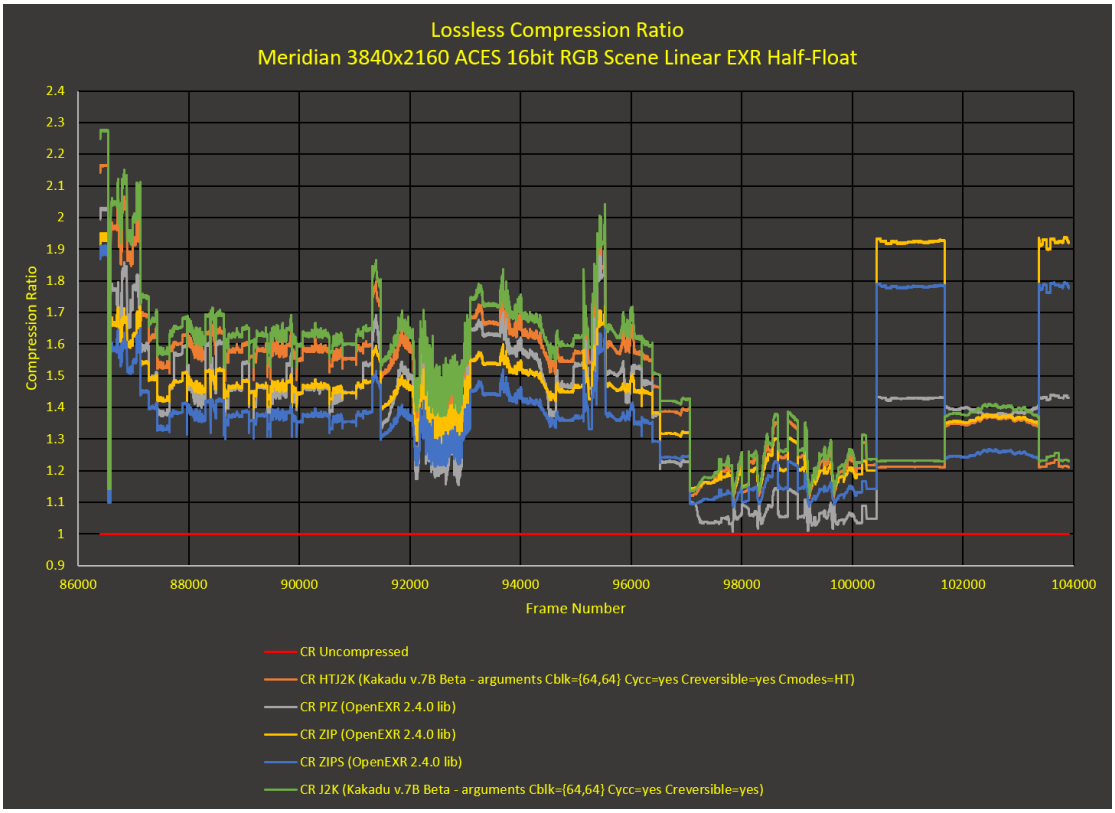


Figure 20: Meridian UHD ACES OpenEXR 16-bit half-float Lossless Compressed Data Rate

The average and median compression ratios for the various formats are shown in Table 27. Table 27 also shows the decoding throughput of a 10-second clip (frames 90000-90239) using the MacBook Pro 15-inch 2018 2.9GHz Intel Core i9 (6-core) 16GB RAM 1TB SSD test platform. Kakadu's high-throughput decoding demo application `kdu_vex_fast` was used to perform the Part-1 and Part-15 decoding.

Table 27: Meridian UHD ACES OpenEXR 16bit half-float Decoding Throughput and Compression Ratio

Compression Type	**Average Compression Ratio**	**Median Compression Ratio**	**Decoding Frame Rate**	**HTJ2K decoding speedup over J2K Part-1**
Uncompressed	1.0	1.0	9	N/A
HTJ2K Part-15	1.462	1.529	38	27x

OpenEXR PIZ	1.390	1.429	21	N/A
OpenEXR ZIP	1.454	1.450	20	N/A
OpenEXR ZIPS	1.364	1.359	12	N/A
J2K Part-1	1.501	1.573	1.4	N/A

A simple C++ program was written to access the OpenEXR v2.40 library to perform the OpenEXR decoding, with the key library access functions reproduced in Table 28.

Table 28: C++ code used to access the OpenEXR v2.40 library

C++ code used to access the OpenEXR v2.40 library (with args.number_of_threads=12)

```
Array2D<Rgba> pixels;
// set specified number of threads for internal use in OpenEXR library
setGlobalThreadCount(args.number_of_threads);
// Read the pixel data into the frame buffer
RgbaInputFile exr_file(input_filename);
Box2i dw = exr_file.dataWindow();
int width = dw.max.x - dw.min.x + 1;
int height = dw.max.y - dw.min.y + 1;
pixels.resizeErase(height, width);
exr_file.setFrameBuffer(&pixels[0][0] - dw.min.x - dw.min.y * width, 1, width);
exr_file.readPixels(dw.min.y, dw.max.y);
```

8. Brief discussion of GPU and hardware deployments

HTJ2K is highly suited to deployment on modern GPU platforms. While the status of GPU implementations for the new standard is currently much less mature than CPU implementations, some very compelling decoding results are reported in [17]. A summary of these results is presented in **Table 29**. The conditions here are the same as those explored in Section 0. Code-block dimensions are 64x64 and 5 levels of CDF9/7 wavelet transform are employed, together with the irreversible multi-component transform of JPEG 2000 Part-1. Source content is the 4K 4:4:4 video sequence "ARRI_AlexaDrums_3840x2160p_24_12b_P3_444" that was studied earlier.

Table 29: GPU-based decoding throughput for 4K 4:4:4 video at 12 bits/channel.

bits/pel	GT1030 frames/s	GT1060 frames/s	GT1080 frames/s
1.0 bpp	160	425	560
4.0 bpp	96	317	440
lossless	62	220	402

In addition to these decoding results, preliminary encoding throughputs have been established for the same content, documented in a submission by Naman and Taubman to the 2020 *IEEE International Conference on Image Processing*. This preliminary work demonstrates GPU encoding of 4K 4:4:4 video at 12 bits/channel on the NVIDIA GT1080, with a frame rate of 455 frames/second at 1.0 bpp and 435 frames/second at lossless. Updated versions of these results will be detailed more carefully in a future revision of this whitepaper.

9. Low-latency compression with HTJ2K

JPEG 2000 was designed from the outset to support low latency compression. In fact, it is possible to achieve extremely low end-to-end latencies, down to a few tens of lines of an image or video frame, without partitioning an image or video frame into tiles or multiple codestreams. Untiled low latency compression is achieved by using small "precincts" with a spatially progressive sequencing order for the JPEG 2000 compressed data packets. JPEG 2000 precincts are spatial partitions of each resolution level within each image component (or colour plane), while JPEG 2000 data packets correspond to one quality layer from all code-blocks belonging to a precinct. In low latency applications there is normally only one quality layer, so each data packet represents all compressed data contributed by code-blocks belonging to the corresponding precinct.

To achieve very low latencies, one can employ highly rectangular precincts that span the width of the image or video frame, but only cover small number of image lines – e.g., 8 or 16 lines. The precincts can be arranged to decrease in size with each successively lower resolution level, so that all resolutions of all components contribute the same number of precincts. With spatially progressive sequencing, the compressed data from precinct s of all resolutions in all image components appears (or is flushed out) together, followed by the compressed data from precinct $s + 1$ of all resolutions in all image components. We can think of the data associated with all the precincts s as constituting a "flush-set" s for the codestream. To achieve the lowest latencies, all of the code-blocks associated with flush-set s must be encoded and subjected to the PCRD-opt rate control procedure as soon as possible after the corresponding subbands can be produced by the wavelet transform.

Note that precincts are not the same as tiles. The spatial discrete wavelet transform (DWT) applied to each image component involves a continuous filtering and sub-sampling process without artificial tile boundaries in the image domain. The code-blocks that comprise each precinct serve to partition the spatial subbands produced by the DWT, but disjoint coding in the transform domain does not result in the appearance of boundary artifacts in the image domain. This is a fundamental advantage of wavelet based schemes over block-based transform schemes such as the DCT used in JPEG, and it is important to preserve this advantage even for low latency compression.

End-to-end latency refers to the time that elapses between the point at which any line of an image or video frame is pushed into the spatial wavelet transform of the compressor and the time that the corresponding line can appear after inverse spatial wavelet transform at the decompressor. It is convenient to measure latency in lines, assuming that lines arrive at a constant rate at the compressor and that a constant bit-rate communication channel is employed to pass encoded data to the decoder. The contributions to end-to-end latency are as follows:

1. DWT analysis latency corresponds to the number of extra lines that must appear beyond the precinct boundaries of a flush-set, in order to generate subband samples for all code-blocks in the flush-set; this depends on the number of vertical wavelet decomposition levels and the extent of the subband analysis filters that implement the DWT. Longer filters such as those of the Part-1 default CDF 9x7 irreversible DWT yield larger latencies than shorter filters such as those of the LeGall 5x3 DWT, whose reversible variant is defined in Part-1 – the irreversible 5x3 DWT is readily accessed using the extensions in JPEG 2000 Part-2.
2. Encoding computation represents the time taken for all of the code-blocks of a flush-set to be encoded, once their subband samples have become available.

For our analysis, we use a simple model in which encoding does not start for any code-block until all code-blocks of the flush-set have been generated by DWT analysis, so a good model for encoding computation delay is exactly L lines, where L is the flush-set height.

3. Communication latency represents the time taken for all compressed data associated with a flush-set to be communicated to the decoder. For our analysis, we use a simple model in which decoding of any of a flush-set's code-blocks cannot start until all of the flush-set's compressed data has arrived. This necessarily adds at least another L lines of latency, but instead we allow $B > L$ lines for the communication latency, where B represents the size of a "leaky bucket" that is filled with L lines worth of compressed data after the encoding of each flush-set – i.e., at intervals separated by exactly L source image lines – and drains at a constant rate, corresponding to the constant bit-rate of the communication channel. The leaky bucket can neither overflow nor underflow, so arranging for B to be larger than L allows the PCRD-opt rate control algorithm some margin to adjust the size of each coded flush-set to one that best fits the available code-block truncation points.
4. Decoding computation represents the time taken for all of the code-blocks of a flush-set to be decoded, once their compressed data has arrived. As noted, we adopt a conservative model in which decoding does not start until all data for the flush-set has arrived, even though the lower resolution code-block data which experiences highest latency in DWT synthesis is certain to arrive first. One might think that decoding computation should add L lines to the latency, and indeed this would be reasonable for a generic JPEG 2000 codestream. However, an HT block decoder can emit decoded subband samples with a delay of only 2 subband lines, even for code-blocks consisting of Cleanup, SigProp and MagRef coding passes. Noting that the highest resolution block decoders must be able to sustain a throughput of 1 subband line for every 2 image lines, there is no reason why all code-blocks cannot be processed with this same throughput, resulting in a decoding contribution of only 4 image lines to the end-to-end latency.
5. DWT synthesis latency corresponds to the number of image lines preceding the precinct boundaries of a flush-set that cannot be fully synthesized until that flush-set becomes available. This is the counterpart to DWT analysis latency in the compressor; again, it depends on the number of vertical levels of wavelet decomposition and the extent of the subband synthesis filters that are used to invert the spatial wavelet transform of each image component.

It is worth noting that only items (1), (3) and (5) in the above list are fundamental contributions to end-to-end latency that cannot be avoided by any means. Items (2) and (4) correspond to computational delay, which could be reduced by an aggressive design that is capable of encoding or decoding code-blocks much faster than is required on average – however, such designs are usually more expensive, since the block coding modules would sit idle for quite a bit of the time.

Before providing experimental results, it is worth explaining why the HT block coding algorithm is much more appropriate for low latency applications than the original J2K-1 block coding algorithm. In order to achieve high throughputs with J2K-1, it is necessary to extensively exploit the parallelism afforded by the use of independent block coding. Since J2K-1 itself is at least ten times slower than the HT block coder, this means that at least ten times more parallel block encoder or decoder instances are required to achieve the same throughput, even in a hardware implementation that is not complexity constrained. At very low latencies, the number of code-blocks in a single flush-set is relatively small, so high throughputs require multiple flush-sets to be encoded or decoded in parallel to achieve high throughputs, and this increases latency.

It is not hard to see, then, that for a given flush-set size, the maximum throughput achievable with HTJ2K should be at least ten times that achievable using J2K-1, irrespective of the amount of hardware one is prepared to dedicate to the task. In fact, one should expect an even larger gain in throughput than this for HTJ2K, because HT block coding and decoding can be performed in constant time, processing all relevant coding passes concurrently, whereas this is not generally possible for J2K-1, where code-blocks with more coding passes inevitably take longer to process than those with fewer coding passes. For a low latency application, the worst case encoding or decoding time is the one that matters most, rather than the average, and this can be much more than 10 times longer than the worst case (usually) constant) HT encoding or decoding time.

We now present experimental results obtained using a software implementation of the HTJ2K compression and decompression system that respects the flush-set model described above, using the PCRD-opt rate control procedure to assign compressed bytes to each flush-set in such a way as to avoid overflow or underflow of the leaky-bucket mentioned above. We quote fundamental and practical end-to-end latencies for different configurations of wavelet transform and precinct dimensions, where the fundamental latency excludes computation time. The practical latencies correspond to what one could expect from a good hardware implementation, rather than the software implementation itself.

Nevertheless, the Kakadu software implementation employed here is itself capable of low latencies in practice. Codestream content for each flush-set is encoded as soon as possible, subjected to PCRD-opt rate control as soon as possible, and flushed out of the compression system as soon as possible, subject to the same fundamental constraints on latency described above; similarly, compressed data is consumed on-demand, parsed and decoded as soon as possible and DWT synthesis also operates as soon as possible after the relevant subband samples have been decoded. The additional latencies one can expect from a software implementation arise from unavoidable thread context switching delays in a pre-emptive multi-tasking operating system, and from the insertion of additional buffering between elements of the processing pipeline to avoid the risk that threads get temporarily starved of work, since unnecessary idling of processing threads imposes large overheads on the efficiency of a multi-threaded software system.

The configurations explored are described via **Table 30**, with the actual command-lines used in our experiments included via footnotes. In all cases, the CPLEX complexity-constraint method described in Section 3.3 is employed to limit the amount of work performed by HT block encoders to at most 2 HT-Sets. The study here involves only individual images, but exactly the same performance should be expected from a video compressor, since there is no benefit to exploiting inter-frame statistics when the CPLEX method is employed for low latency compression.

Figure 21 through **Figure 34** present rate-distortion results separately for each of a large collection of standard test images and video frames, corresponding to different types of content. In particular, observe that the screen-content examples "RICHTER_ScreenContent" (**Figure 24**) and "APPLE_BasketBallScreen" (**Figure 25**) are unique in that higher coding efficiency is observed using the irreversible LeGall 5x3 DWT, while in all other cases the higher latency CDF 9x7 DWT yields significantly higher coding efficiency. The presented results also compare the performance of HTJ2K against that of the recent JPEG-XS coding standard, which particularly targets low latency, at three complexity points corresponding to Main, High and Light profiles. These all use a virtually irreversible form of the LeGall 5x3 DWT with 2 or 3 levels of vertical decomposition and 5 levels of horizontal decomposition, where XS(Main) and XS(Light) have 2 vertical levels and XS(High) has 3. In general HTJ2K has higher coding efficiency

than JPEG-XS, especially at lower bit-rates, but the main differentiators for HTJ2K are its extensive set of features such as spatial random access, its high degree of parallelism, and its flexibility.

A summary covering all RGB test images is shown in **Figure 35**, restricted to just a few of the latency points for clarity.

Table 30: Configurations explored for low-latency HTJ2K compression study.

Label	Flush-set height, L	Bucket lines, *B*	DWT	DWT Levels		End-to-end Latency	
				Vertical	Horizontal	Fundamental	Practical
53L24[14]	8	10	irv5x3	2	5	24 lines	36 lines
53L48[15]	16	18	irv5x3	3	5	48 lines	68 lines
53L80[16]	32	34	irv5x3	3	5	80 lines	116 lines
97L36[17]	8	10	irv9x7	2	5	36 lines	48 lines
97L76[18]	16	18	irv9x7	3	5	76 lines	96 lines
97L108[19]	32	34	irv9x7	3	5	108 lines	144 lines
97L176[20]	64	66	irv9x7	3	5	176 lines	240 lines
HTfull[21]	whole frame	whole frame	irv9x7	6	6	~2 frames	~3 frames

[14] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={4,1024} Qstep=0.0001 Clevels=5 Cdecomp=B(-:-:-),B(-:-:-),H(-) Cprecincts={8,8192},{4,8192},{2,8192} Scbr={1,10} Catk=2 Kkernels:I2=I5X3 Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

[15] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={8,512} Qstep=0.0001 Clevels=5 Cdecomp=B(-:-:-),B(-:-:-),B(-:-:-),H(-) Cprecincts={16,8192},{8,8192},{4,8192},{2,8192} Scbr={1,18} Catk=2 Kkernels:I2=I5X3 Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

[16] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={16,256} Qstep=0.0001 Clevels=5 Cdecomp=B(-:-:-),B(-:-:-),B(-:-:-),H(-) Cprecincts={32,8192},{16,8192},{8,8192},{4,8192} Scbr={1,34} Catk=2 Kkernels:I2=I5X3 Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

[17] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={4,1024} Qstep=0.0001 Clevels=5 Cdecomp=B(-:-:-),B(-:-:-),H(-) Cprecincts={8,8192},{4,8192},{2,8192} Scbr={1,10} Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

[18] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={8,512} Qstep=0.0001 Clevels=5 Cdecomp=B(-:-:-),B(-:-:-),B(-:-:-),H(-) Cprecincts={16,8192},{8,8192},{4,8192},{2,8192} Scbr={1,18} Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

[19] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={16,256} Qstep=0.0001 Clevels=5 Cdecomp=B(-:-:-),B(-:-:-),B(-:-:-),H(-) Cprecincts={32,8192},{16,8192},{8,8192},{4,8192} Scbr={1,34} Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

[20] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={16,256} Qstep=0.0001 Clevels=5 Cdecomp=B(-:-:-),B(-:-:-),B(-:-:-),H(-) Cprecincts={64,8192},{32,8192},{16,8192},{8,8192} Scbr={1,34} Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

[21] kdu_compress/kdu_v_compress parameters: `Corder=PCRL Cblk={32,128} Qstep=0.0001 Clevels=6 Cprecincts={256,256} Cmodes=HT Cplex={6,EST,0.25,-1} -no_weights`

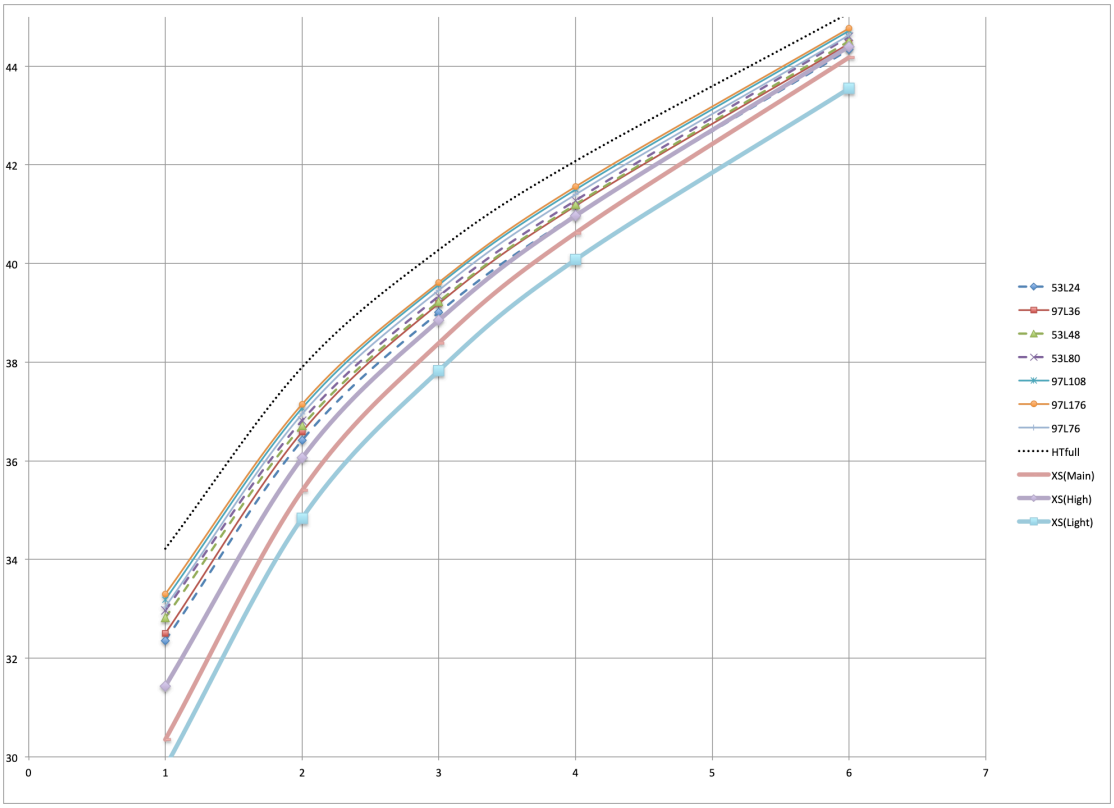


Figure 21: Coding efficiency (PSNR vs bits/pixel) for RGB JPEG test image BIKE_2048x2560_8b_RGB, with various latencies, showing various HTJ2K and JPEG-XS configurations. HTJ2K configuration labels are described in **Table 30**. JPEG-XS results are obtained using the reference software contained in ISO/IEC 21122-5 with arguments "-p 3/4/6 –o psnr" for Main/High/Light profiles with PSNR-based encoder optimization.

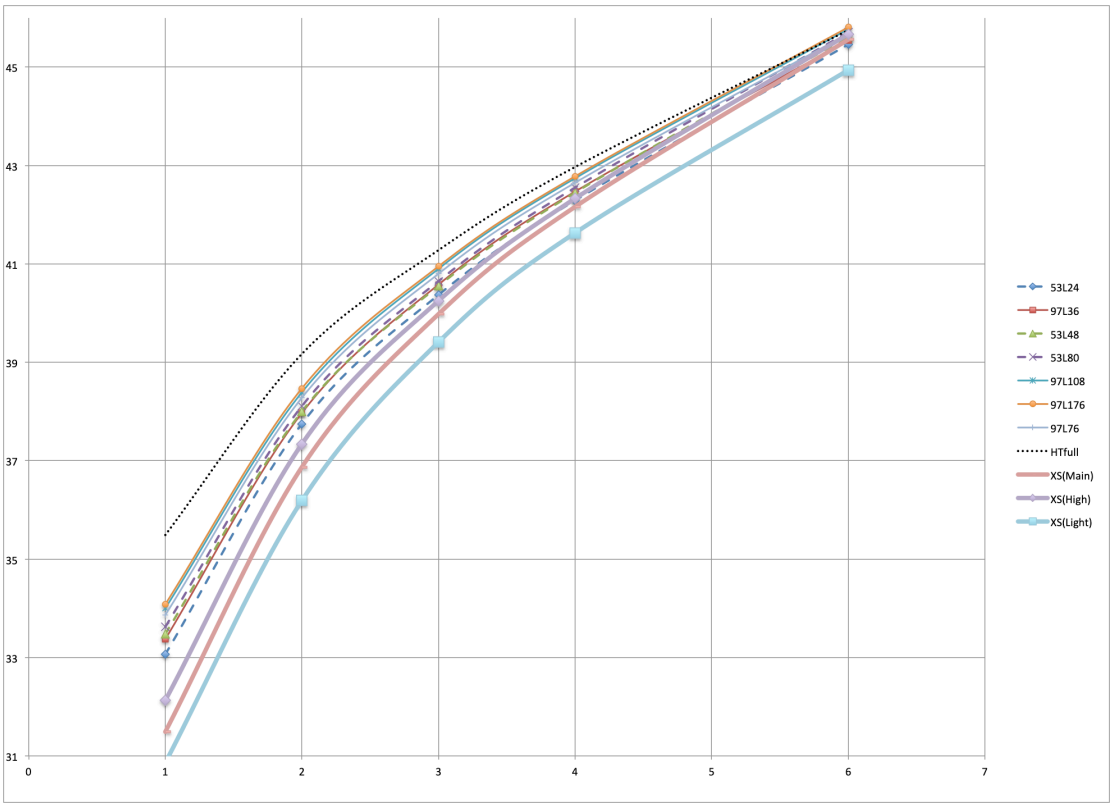


Figure 22: As in **Figure 21**, but for RGB JPEG test image WOMAN_2048x2560_8b_RGB.

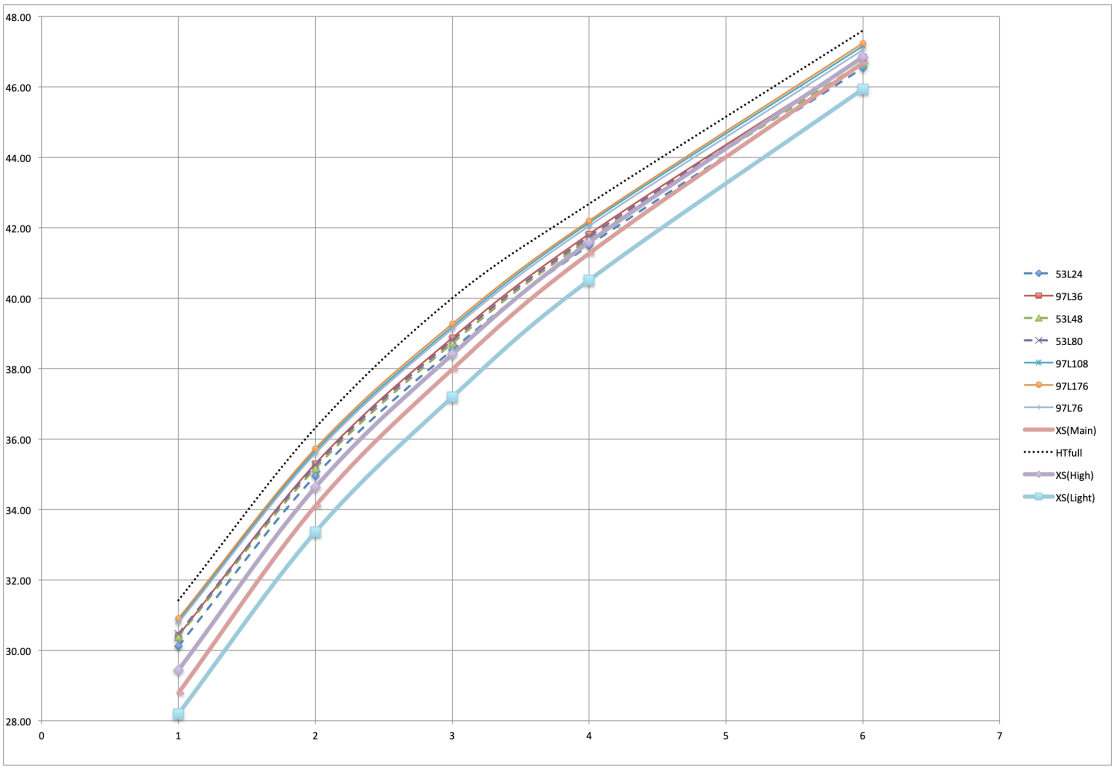


Figure 23: As in **Figure 21**, but for RGB JPEG test image BLENDER_Sintel2_4096x1744p_24_10b_sRGB_444_00004606.

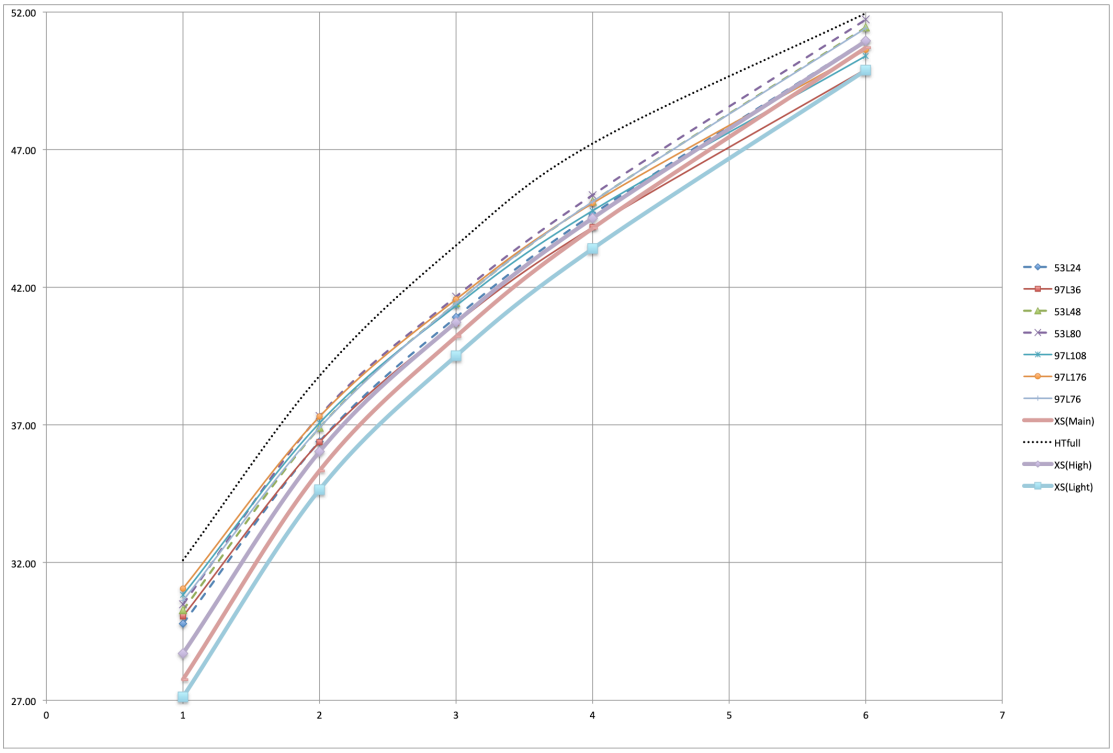


Figure 24: As in **Figure 21**, but for RGB JPEG test image RICHTER_ScreenContent_4096x2160p_15_8b_sRGB_444_0001

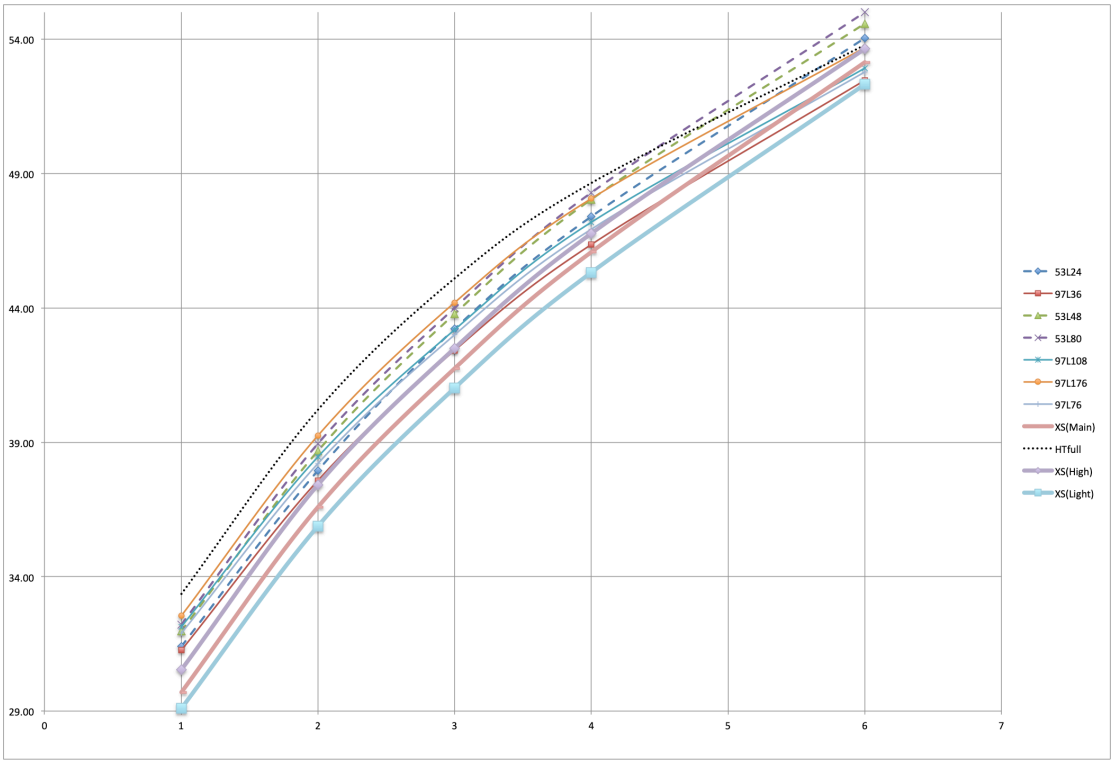


Figure 25: As in **Figure 21**, but for RGB JPEG test image APPLE_BasketBallScreen_2560x1440p_60_8b_sRGB_444_000

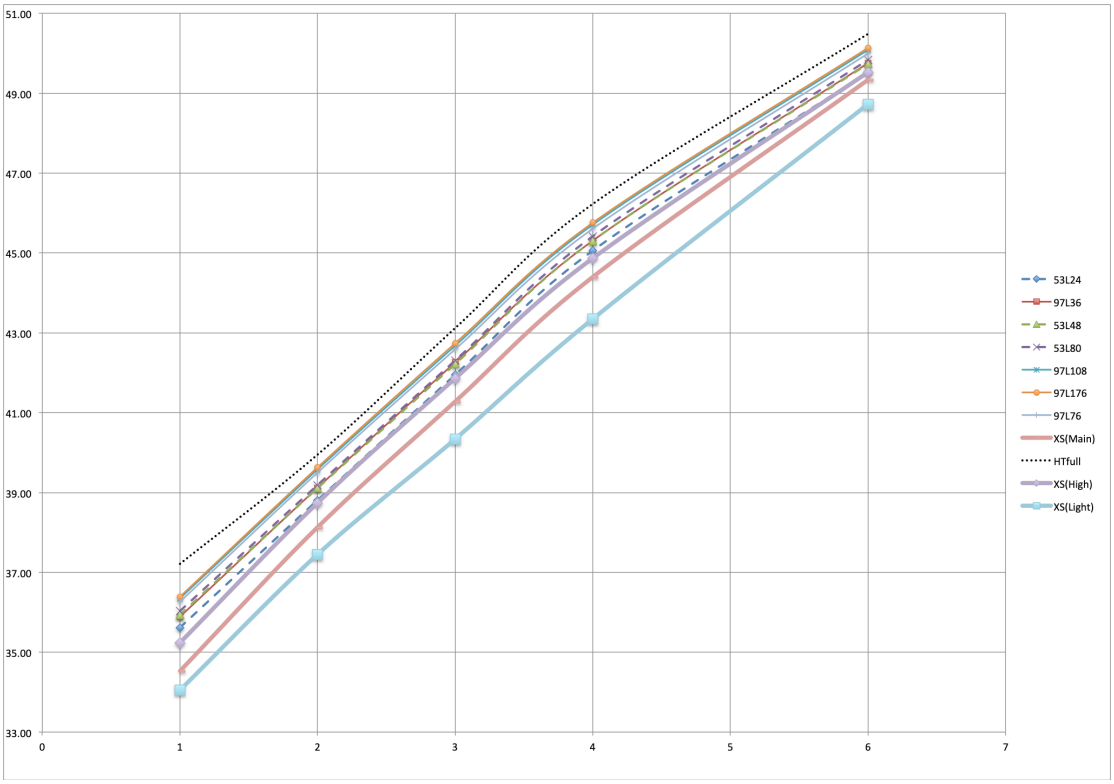


Figure 26: As in **Figure 21**, but for RGB JPEG test image EBU_PendulusWide_3840x2160p_50_10b_bt709_444_0001

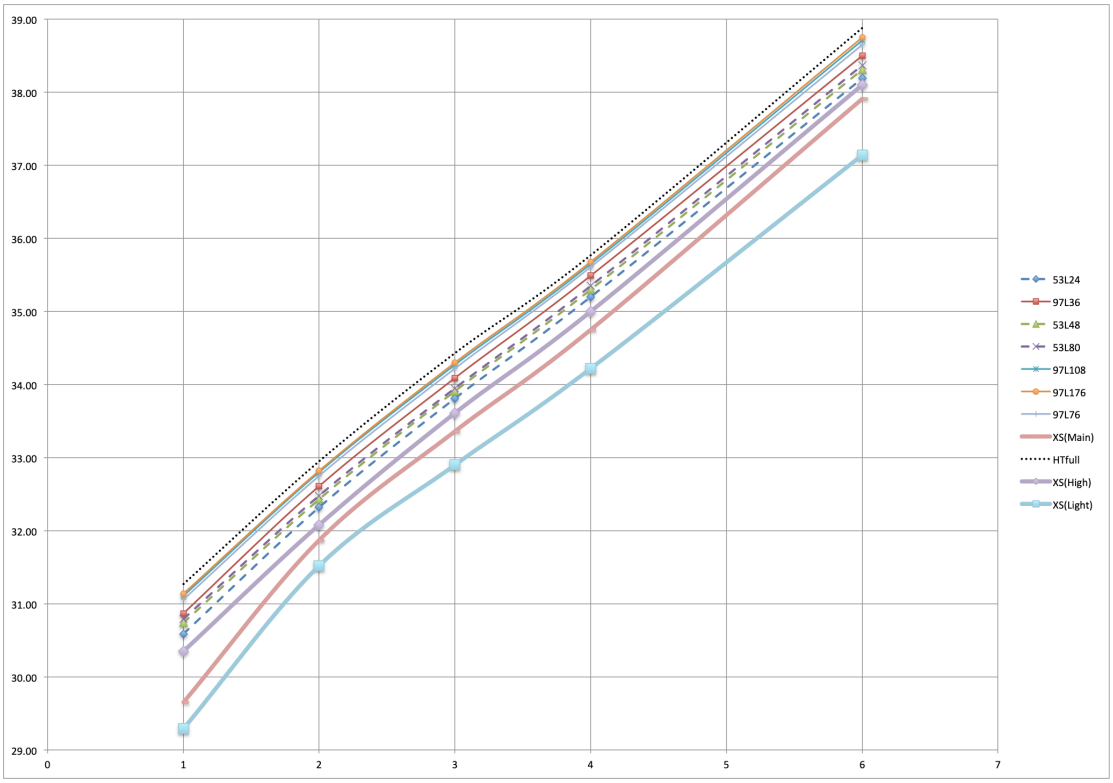


Figure 27: As in **Figure 21**, but for RGB JPEG test image VQEG_CrowdRun_3840x2160p_50_8b_bt709_444_07111

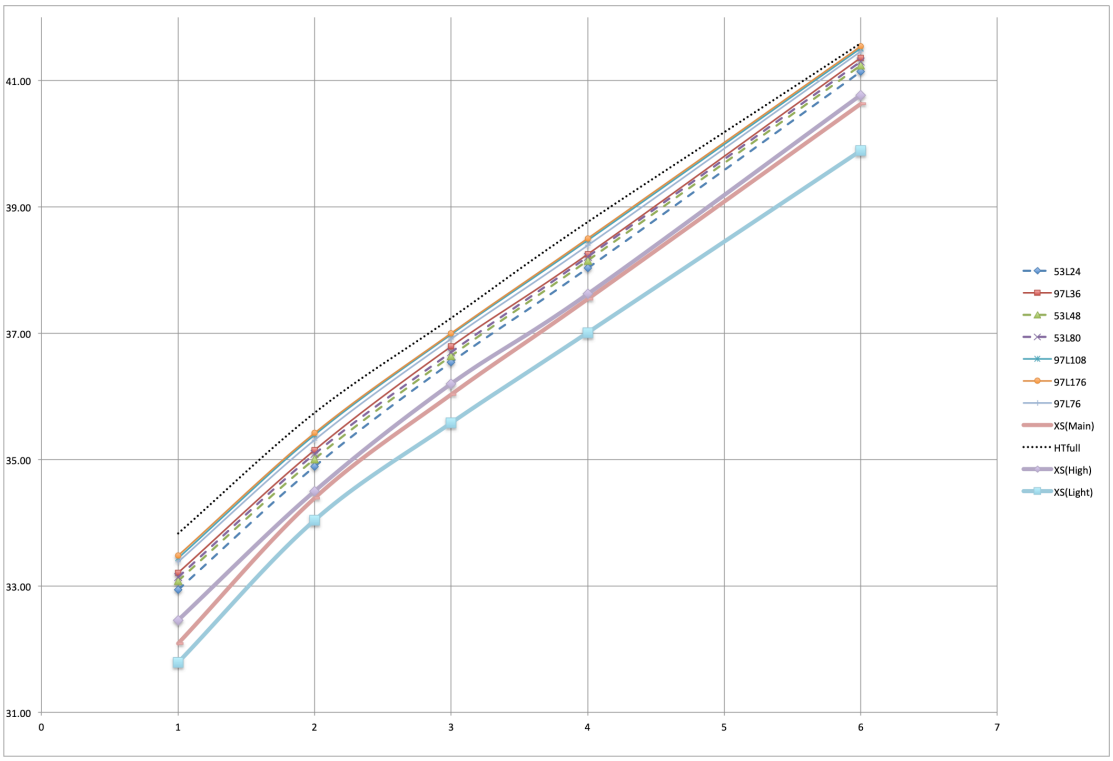


Figure 28: As in **Figure 21**, but for RGB JPEG test image VQEG_ParkJoy_3840x2160p_50_8b_bt709_444_15523

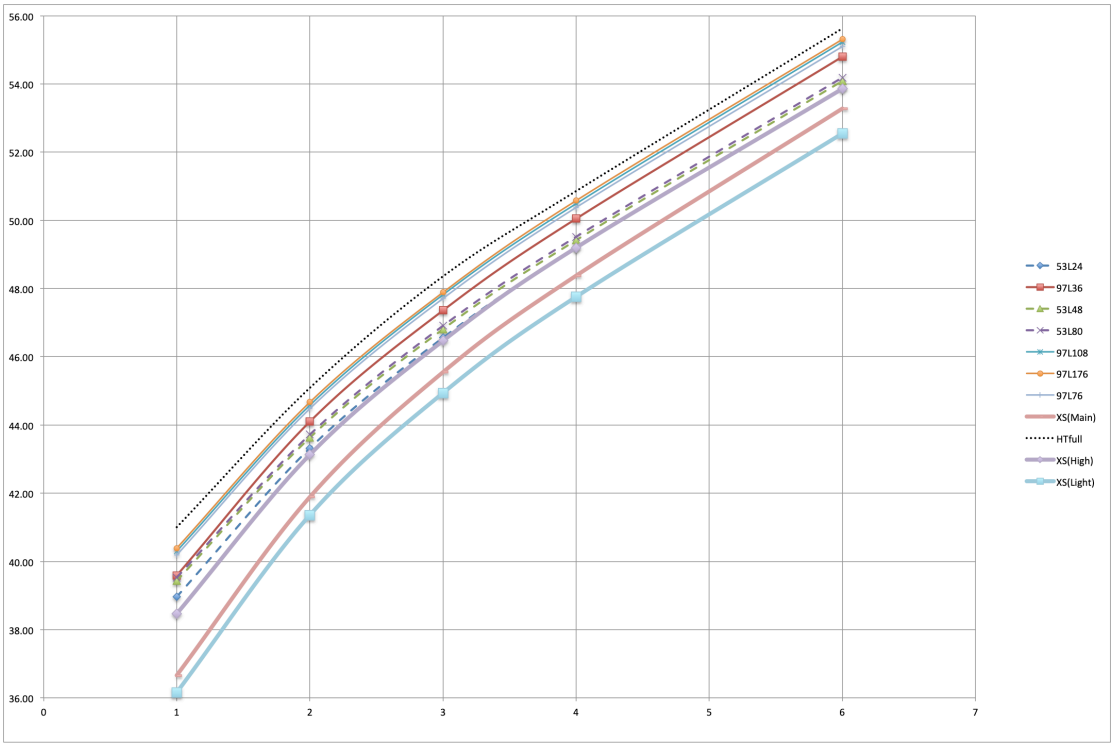


Figure 29: As in **Figure 21**, but for RGB JPEG test image ARRI_AlexaDrums_3840x2160p_24_12b_P3_444_00000

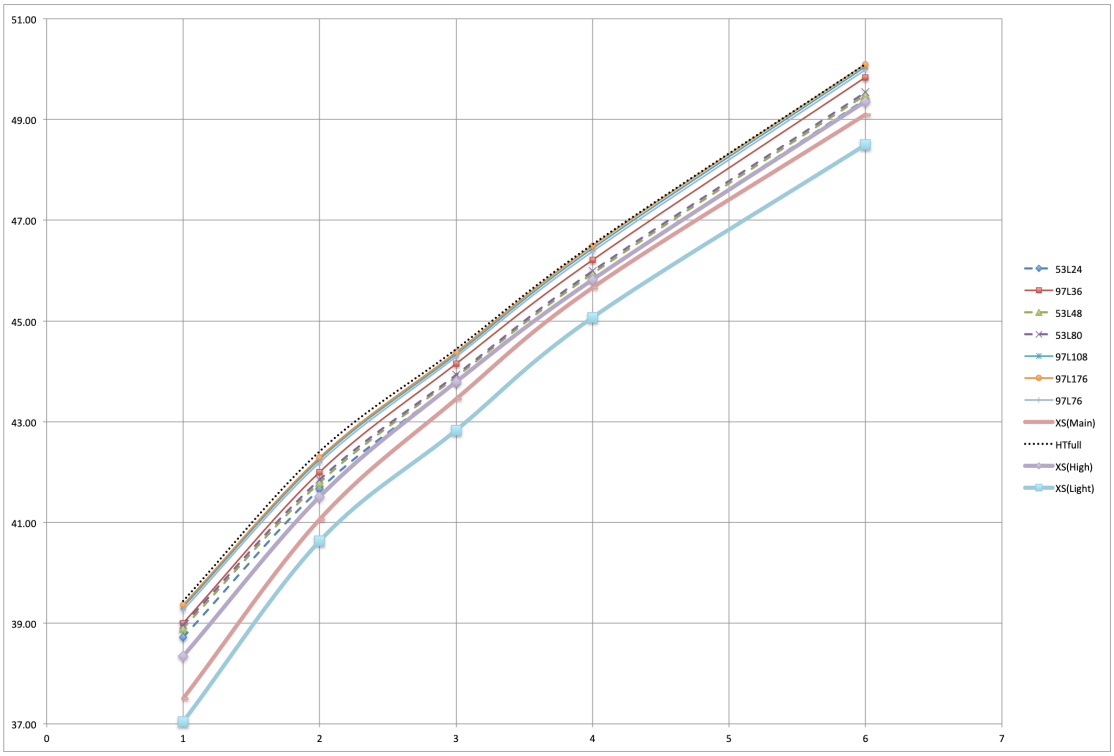


Figure 30: As in **Figure 21**, but for RGB JPEG test image ARRI_AlexaHelicopterView_3840x2160p_24_12b_P3_444_00000

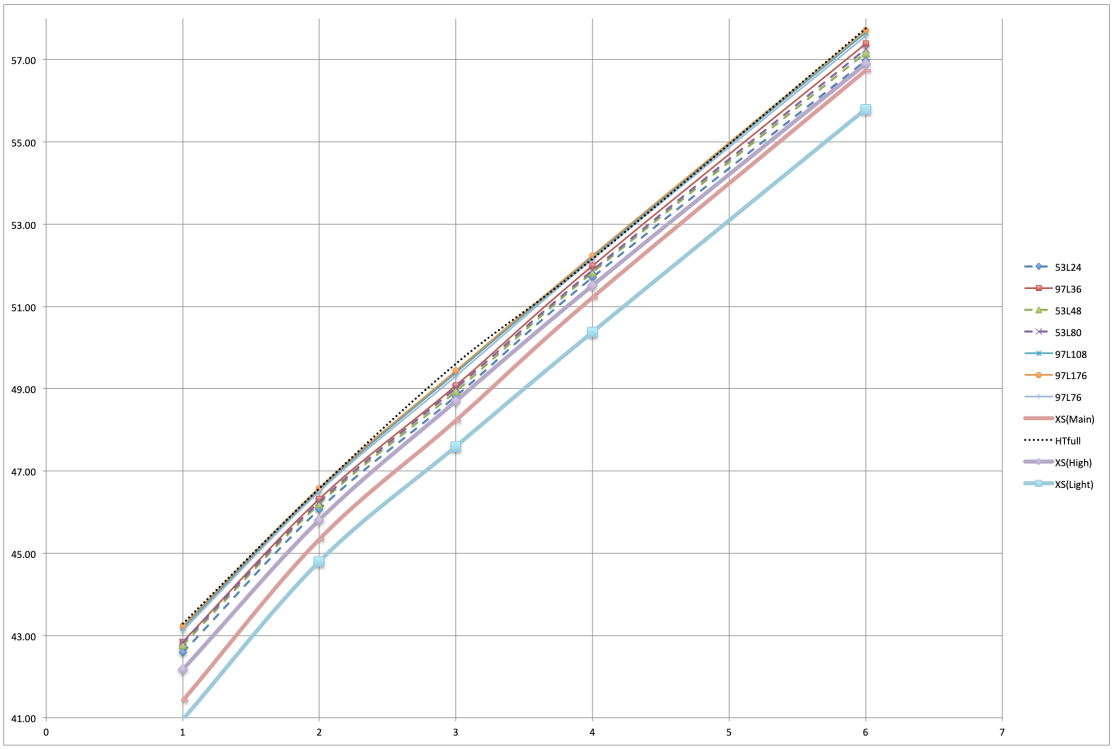


Figure 31: As in **Figure 21**, but for 4:2:2 JPEG test image ARRI_AlexaDrums_3840x2160p_24_12b_P3_422_00000

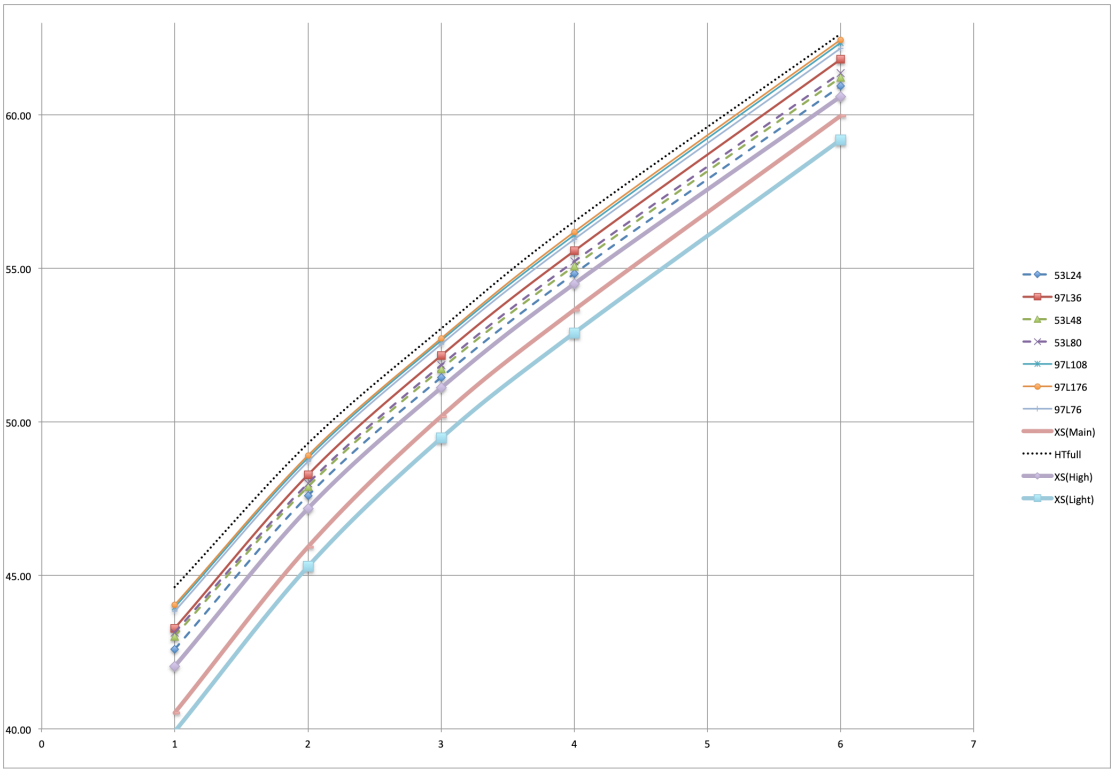


Figure 32: As in **Figure 21**, but for 4:2:2 JPEG test image ARRI_AlexaHelicopterView_3840x2160p_24_12b_P3_422_00000

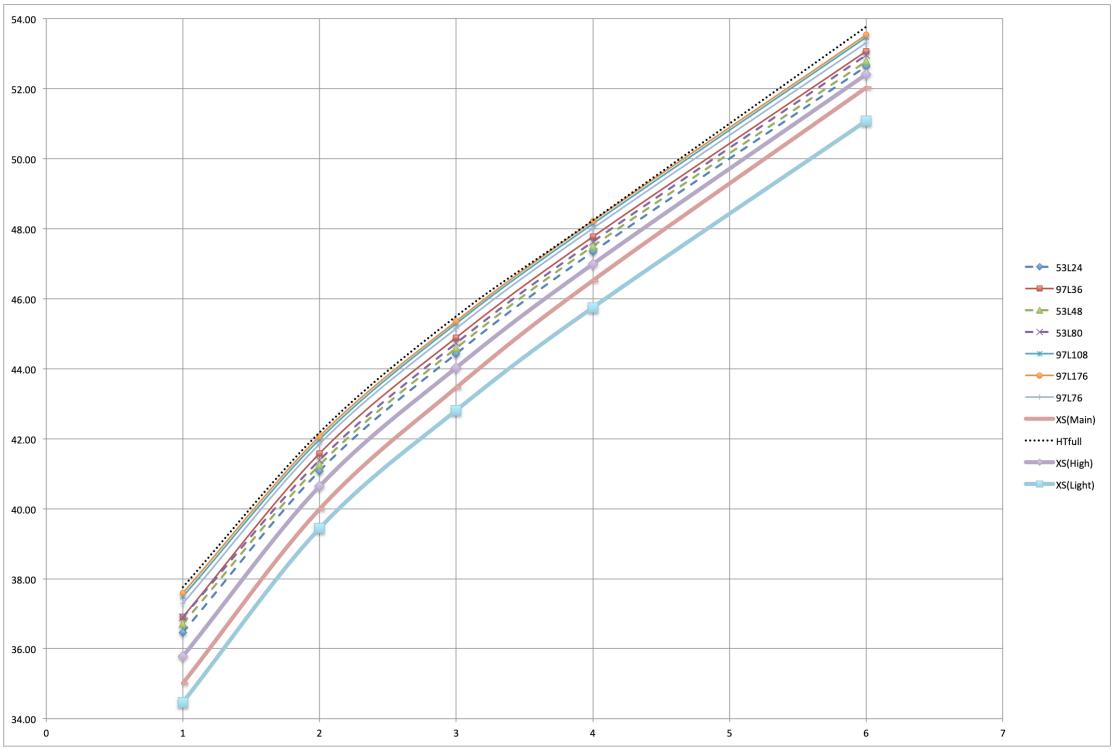


Figure 33: As in **Figure 21**, but for 4:2:2 JPEG test image ARRI_Lake2_2880x1620p_24_10b_bt709_422_0000

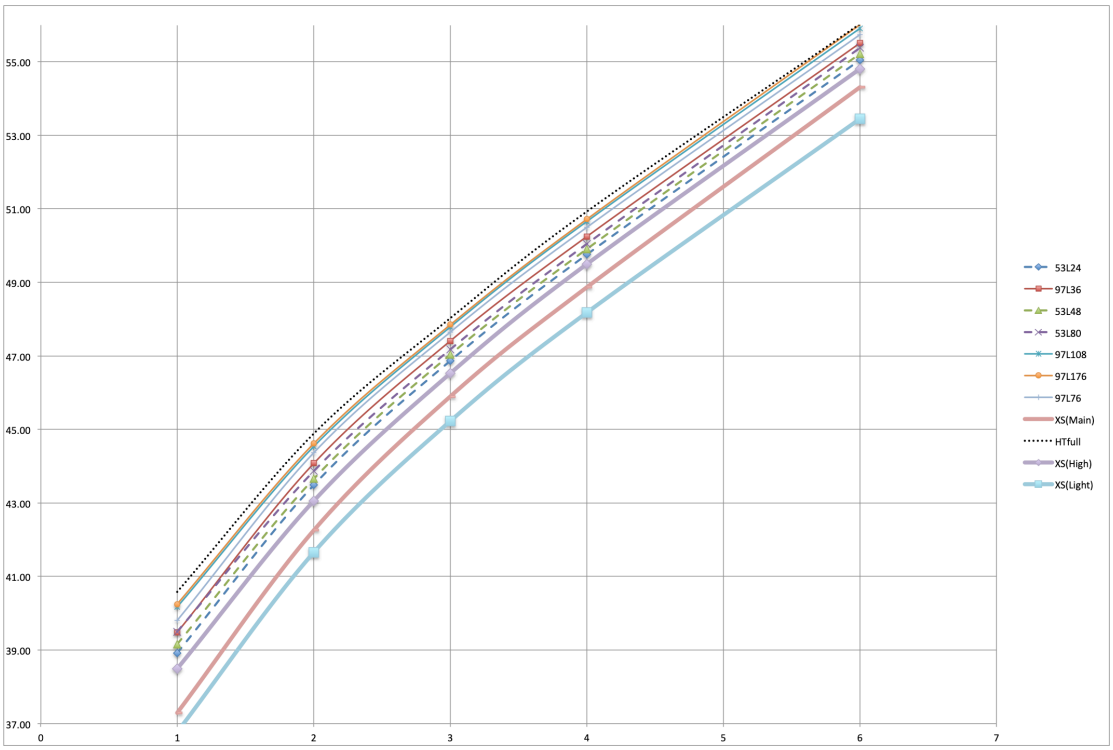


Figure 34: As in **Figure 21**, but for 4:2:2 JPEG test image ARRI_PublicUniversity_2880x1620p_24_10b_bt709_422_0000

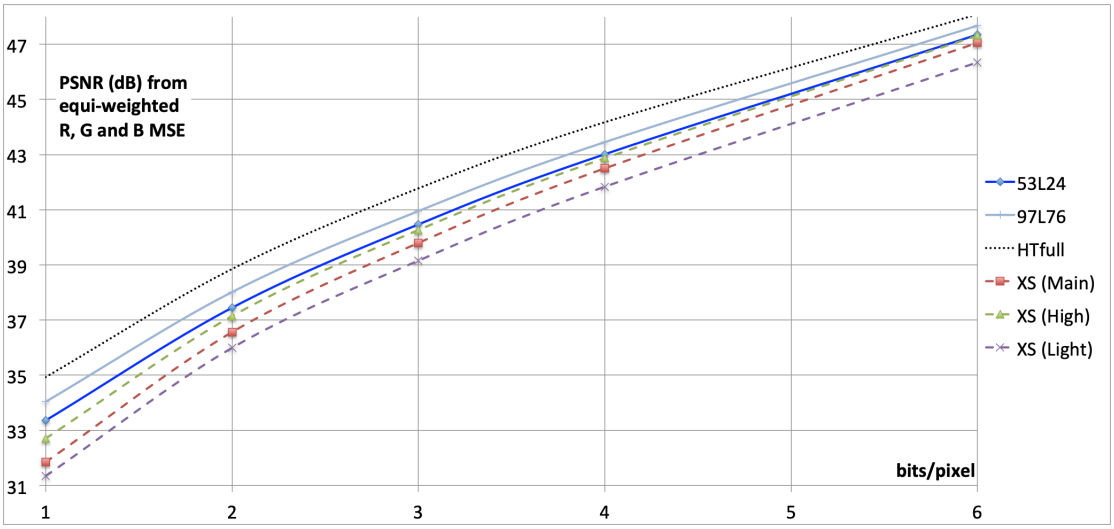


Figure 35: Average rate-distortion performance over the RGB test results presented in **Figure 21** through **Figure 30**, showing average PSNR values for a selection of the latency configurations from **Table 30**.

10. JPH comparison with JPG

In this section we provide a preliminary comparison between HTJ2K and the original JPEG standard. We can think of this as the comparison between the JPH and JPG file formats, where JPH embeds an HTJ2K code-stream and JPG embeds a JPEG code-stream. The main aspects considered in this comparison are coding efficiency, encoding throughput and decoding throughput, but it should be noted that HTJ2K offers a much larger set of extremely useful features than JPEG, including:

- HTJ2K/JPH offers multi-resolution access for free, while this is expensive and inefficient if included as an option with JPEG – progressive JPEG.
- HTJ2K/JPH offers region-of-interest accessibility, whereas JPEG files provide only a single monolithic representation of the source.
- HTJ2K/JPH offers non-iterative (one-pass) rate control, where a target compressed size can be achieved without iterative encoding, while JPEG does not offer this at all.
- HTJ2K/JPH offers very high precision, supporting virtually any type of HDR representation of interest, while JPEG is almost always limited to 8-bit precision – the 12-bit option is rarely used and slower.
- JPH offers a modern set of colour space descriptions, including HDR spaces.
- HTJ2K/JPH offers extremely high throughput lossless coding if required.
- Both HTJ2K/JPH and JPEG are intended to be royalty free standards.

For the purpose of numerical comparisons between HTJ2K and JPEG, here we use the same 4K video content as in Section 0, i.e., ARRI AlexaDrums P3, but we reduce the source sample precision to 8 bits first, for the benefit of JPG. The JPEG compression is performed using "libjpeg-turbo," which is heavily optimized for the target i7 Skylake processor (same platform used in Section 0), taking full advantage of AVX2. JPEG throughput is evaluated using TurboJPEG's "tjbench" tool, which runs the codec many times on the same image so as to amortize processing overheads not strictly related to the encoding or decoding tasks. While parallel encoding, and to some extent parallel decoding of JPEG images is possible, this is much more difficult with JPEG than it is with HTJ2K. Only single-threaded results are available for TurboJPEG, so we provide single-threaded throughput results also for HTJ2K, along with multi-threaded results that fully utilize all 4 cores (8 hardware threads) of the CPU.

Figure 36 and **Figure 37** show encoding and decoding throughputs, respectively. Evidently, HTJ2K can be faster than the TurboJPEG implementation of JPEG, both for encoding and decoding, although HTJ2K encoding with precise rate control is a little slower than TurboJPEG encoding controlled by a single Q-factor; on the other hand, quantization-based encoding of HTJ2K, which is essentially the same as Q-factor control, appears to be faster than JPEG, even in single-threaded deployment. From the perspective of throughput, the most obvious advantage of HTJ2K is parallelism – throughput essentially scales with the number of processor cores deployed[22], even when all deployed on the same image, as they are here.

Figure 38 and **Figure 39** suggest that HTJ2K has significantly higher coding efficiency than JPG. This is not surprising, since the efficiency of HTJ2K is similar to that of legacy JPEG 2000, which is well known to be substantially higher than that of JPEG in disciplined tests, especially at lower bit-rates.

[22] Of course, this can only be true for a limited number of cores, before other elements in the system limit throughput.

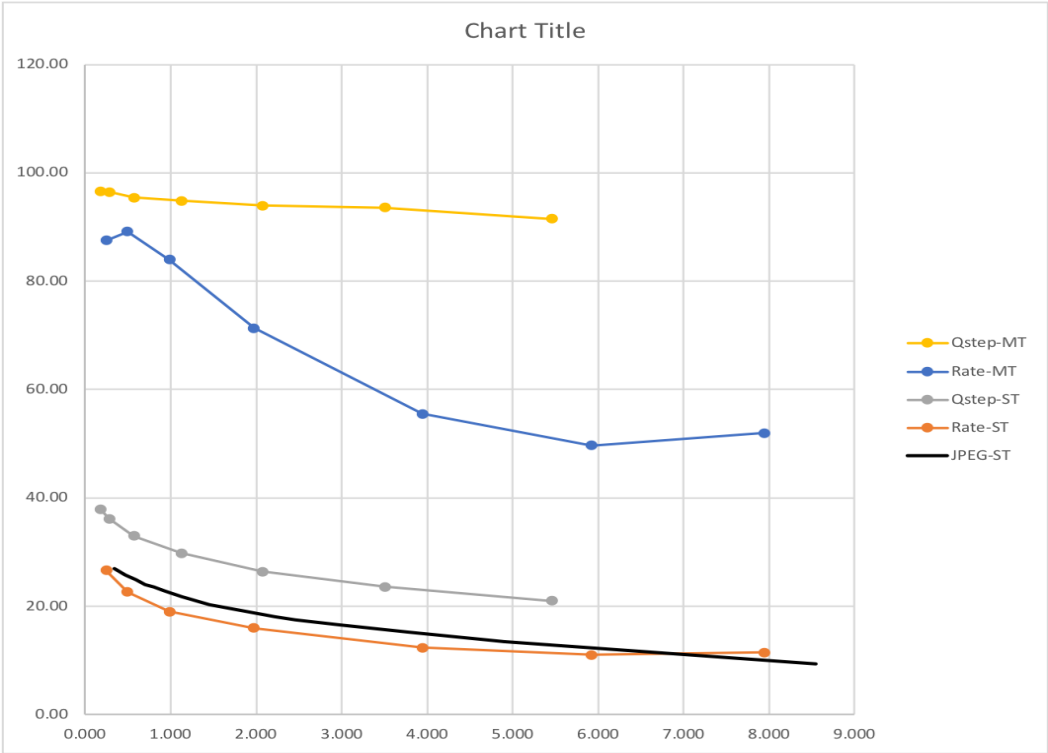


Figure 36: Encoding throughput (frames/second) for 4K 4:4:4 8-bit/channel content. Q-factor based TurboJPEG single-threaded encoding(black curve) compared against single- and multi-threaded (8 threads) HTJ2K encoding, based on quantization (Qstep = Qfactor) (grey and yellow curves) and CPLEX-based precise rate control (red and blue curves). Measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

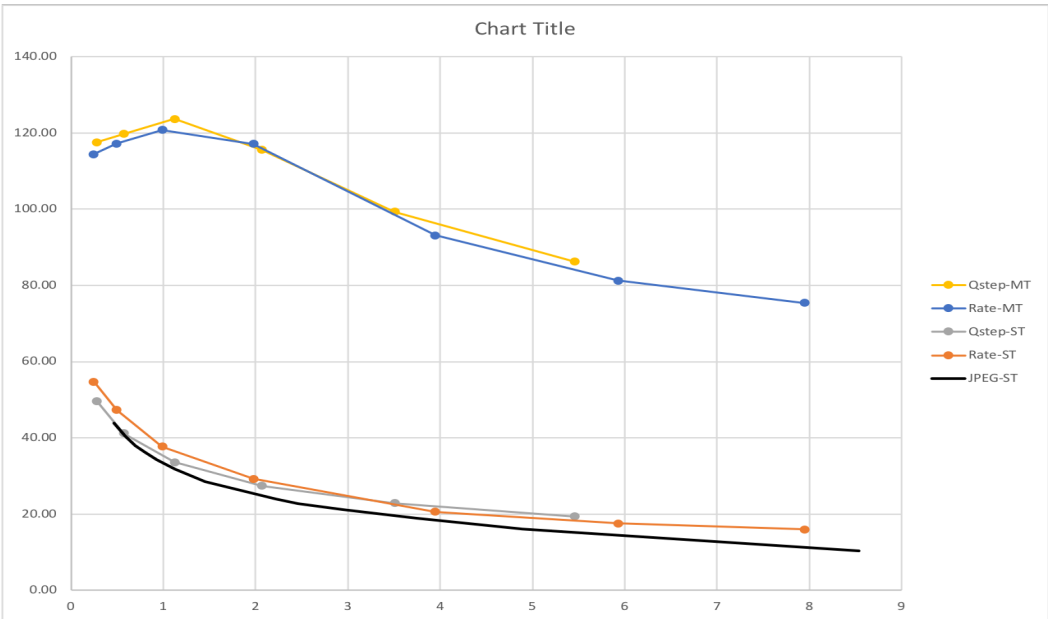


Figure 37: Decoding throughput (frames/second) for 4K 4:4:4 8-bit/channel content. The 5 curves here are obtained by decoding the corresponding content whose encoding is reported in **Figure 36**. Single-threaded TurboJPEG decoding is shown in black; single-threaded JPH decoding is shown by the red and grey curves; multi-threaded HTJ2K decoding is shown by the yellow and blue curves. Measured on a 4-core Intel i7-6700 CPU with 3.4GHz base clock.

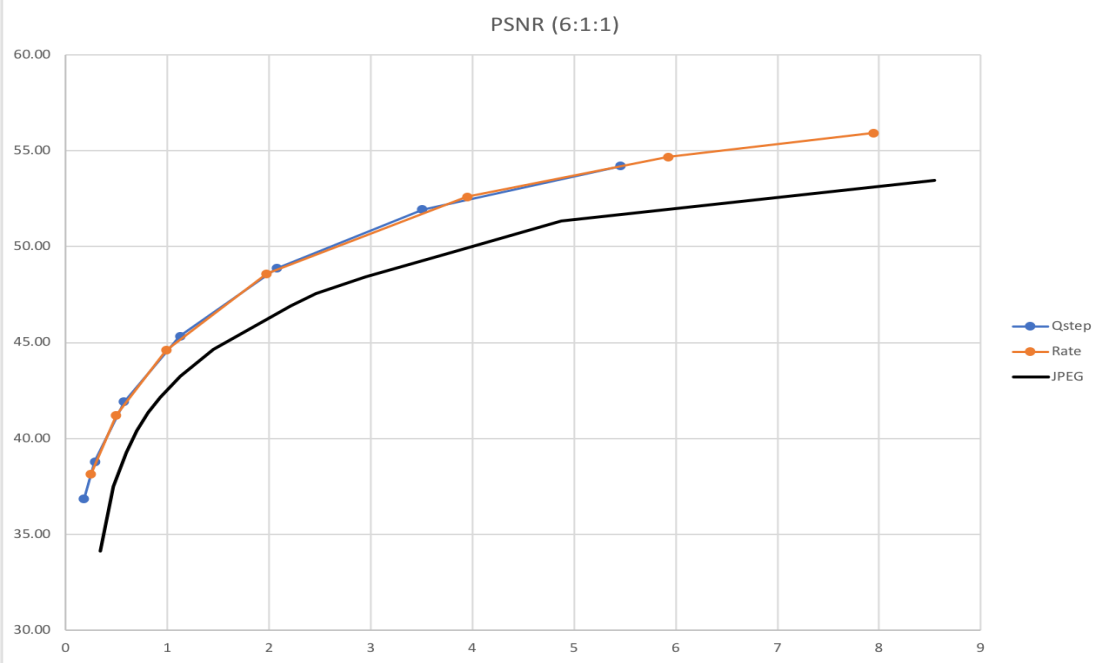


Figure 38: Peak Signal to Noise Ratio (PSNR) comparison between HTJ2K and JPG, measured in dB, based on luminance and chrominance Mean Squared Error values using a conventional 6:1:1 weighting (i.e., luminance errors are weighted higher). JPG PSNR (black curve) is compared with HTJ2K PSNR using quantization-based (red curve) control and CPLEX-based (blue curve) non-iterative precise rate control.

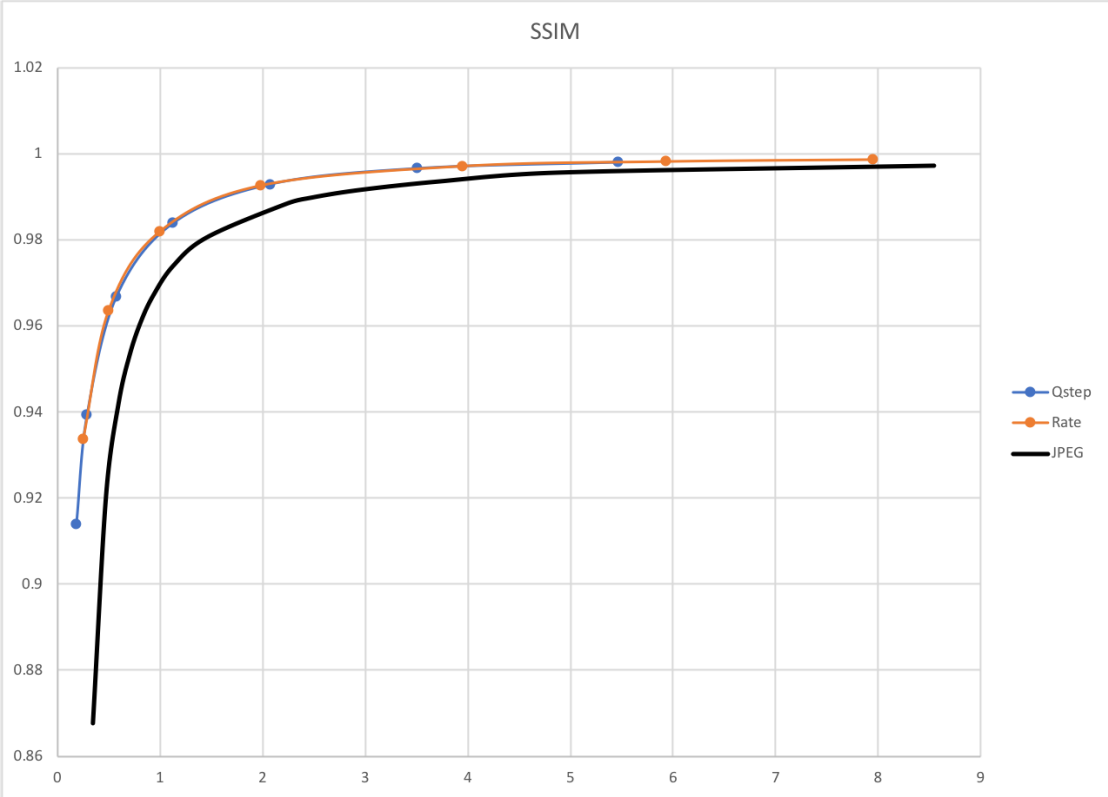


Figure 39: Structural Similarity Index Metric (SSIM) comparison between HTJ2K and JPG. JPG PSNR (black curve) is compared with HTJ2K PSNR using quantization-based (red curve) control and CPLEX-based (blue curve) non-iterative precise rate control.

For reference and reproducibility, the results presented above have been obtained using the Kakadu version 8.0 demonstration executables "kdu_v_compress" and "kdu_vex_fast," as shown below.

Quantization-based HTJ2K encoding:

```
kdu_v_compress -i arri.vix -o arri.mj2 Corder=RPCL Clevels=6
Cblk={64,64} Cmodes=HT -fastest -double_buffering 16 -no_weights
Cweight:C0=1 Cweight:C1=0.4082 Cweight:C2=0.4082 Qweights=yes Qstep=%f
```

CPLEX-based HTJ2K encoding with precise rate control:

```
kdu_v_compress -i arri.vix -o arri.mj2 Corder=RPCL Clevels=6
Cblk={64,64} Cmodes=HT -fastest -double_buffering 16 Qstep=0.001 -
no_weights Cweight:C0=1 Cweight:C1=0.4082 Cweight:C2=0.4082
Cplex={6,EST,0.25,0} -rate %f
```

HTJ2K decoding:

```
kdu_vex_fast -i arri.mj2 -double_buffering 20 -engine_threads 8 -
read_ahead 4 -repeat 4
```

11. Conclusions

The new HTJ2K compression standard provides an alternative block coding algorithm for the JPEG 2000 family of standards. While only a relatively small change to the complete suite of technology, the new block coder provides sweeping improvements to the JPEG 2000 story.

With similar or higher throughput to a heavily optimized implementation of the original JPEG algorithm (even single threaded), HTJ2K provides higher coding efficiency, resolution scalability, region-of-interest accessibility and much more parallelism. Image quality can be controlled using a recently established Qfactor, if desired, which substantially mimics the properties of the JPEG Qfactor, extending them to a much wider range of sample precisions and colour spaces. However, precise non-iterative rate control is also available (a well-known strength of JPEG 2000) and can even be combined with the Qfactor approach, to address a wide range of application needs, all while maintaining extremely high throughput.

HTJ2K preserves almost all of the rich feature set of JPEG 2000, with the exception of quality scalability, offering an order of magnitude increase in throughput (i.e., vastly lower computational complexity) with approximately 5-10% reducing in coding efficiency.

Moreover, the HT and original J2K-1 representations are fully interchangeable, allowing reversible transcoding to be incorporated at any point within a capture, distribution, archiving, caching or rendering system, to obtain the best features of both algorithms without sacrificing the integrity of the data. HTJ2K can even preserve all quality layering, profiles and other aspects of a non-HTJ2K code-stream during transcoding.

12. Bibliography

[1] "ITU-T Rec T.800 | IS 15444-1 -- JPEG 2000 image coding system: Core coding system," 2001.

[2] "ITU-T Rec T.808 | IS 15444-9 -- JPEG 2000 image coding system: Interactivity tools, APIs and protocols," 2005.

[3] "ITU-T Rec T.801 -- Information technology -- JPEG 2000 image coding system: Extensions," 2002.

[4] JPEG Committee, "High throughput JPEG 2000 (HTJ2K): call for proposals," 2017.

[5] JPEG Committee, "ISO/IEC JTC1/SC29/WG1n81025: HTJ2K test results for SMPTE liaison," November 2018. [Online]. Available: https://isotc.iso.org/livelink/livelink?func=ll&objId=16471911&objAction=browse&viewType=1.

[6] "ITU-T Rec T.814 | IS 15444-15 -- JPEG 2000 image coding system: High-throughput JPEG 2000," 2019.

[7] "ITU-T Rec T.801 | IS 15444-2 Amendment 3 -- JPEG 2000 image coding system: Extensions," 2013.

[8] "ITU-T Rec T.803 | IS 15444-4 -- JPEG 2000 image coding system: Conformance testing," 2009.

[9] "ITU-T Rec T.804 | IS 15444-5 -- JPEG 2000 image coding system: Reference software," 2015.

[10] D. Taubman, A. Naman, R. Mathew, S.-Y. Chien and T.-H. Tsai, "FBCOT: a fast block coding option for JPEG 2000," ISO/IEC JTC1/SC29/WG1 m73021, October 2016.

[11] D. Taubman, "FBCOT: Response to JPEG call for proposals on high throughput JPEG 2000," ISO/IEC JTC1/SC29/WG1 n79043, April, 2018.

[12] Taubman, D., "HTJ2K CE1.3 Contribution: CxtVLC codeword extensions," ISO/IEC JTC1/SC29/WG1 n80018, July 2018.

[13] O. Watanabe and D. Taubman, "A MATLAB implementation of the emerging HTJ2K standard," in *IEEE Global Conference on Consumer Electronics (GCCE)*, 2019.

[14] D. Taubman, "Software architectures for JPEG2000," in *Proc. IEEE Int. Conf. DSP, Santorini, Greece*, 2002.

[15] D. Taubman, A. Naman and R. Mathew, "FBCOT: a fast block coding option for JPEG2000," in *SPIE Optics and Photonics: Applications of Digital Imaging*, San Diego, 2017.

[16] S. Ahar, S. Mahmoudpour, O. Watanabe, D. Taubman and P. Schelkens, "Parameterization of the quality factor for the High Throughput JPEG 2000," in *Proc. SPIE Photonics Europe, vol.11353, Photonics and Digital Technologies for Imaging Applications VI*, April 2020.

[17] A. Naman and D. Taubman, "Decoding high-throughput JPEG2000 (HTJ2K) on a GPU," in *IEEE International Conference on Image Processing*, 2019.