TECHNISCHE UNIVERSITÄT
ILMENAU

Department of Electrical Engineering and Information Technology

High Throughput Image Compression and Decompression on GPUs

Doctoral Thesis

Submitted by:	Volker Bruns
Submission Date:	21.11.2021
Defense Date:	28.6.2022
Advisor:	Prof. Dr.-Ing. Dr. rer. nat. h.c. mult. Karlheinz Brandenburg
Reviewers:	Prof. Dr.-Ing. Dr. rer. nat. h.c. mult. Karlheinz Brandenburg
	Prof. Dr.-Ing. Siegfried Fößel
	PD Dr.-Ing. Thomas Wittenberg

DOI: 10.22032/dbt.52369
URN: urn:nbn:de:gbv:ilm1-2022000235

I. Abstract

This work investigates possibilities to create a high throughput, GPU-friendly, intra-only, Wavelet-based video compression algorithm optimized for visually lossless applications. Addressing the key observation that JPEG 2000's entropy coder is a bottleneck and might be overly complex for a medium-to-high bit rate scenario, various algorithmic alterations are proposed and evaluated.

First, JPEG 2000 part-2's *Selective Arithmetic Coding* mode is realized on the GPU, but the gains in terms of an increased throughput are shown to be limited. Instead, two independent alterations not compliant to the standard are proposed, that (1) give up the concept of intra-bit plane truncation points (single-pass mode) and (2) introduce a true raw-coding mode that is fully parallelizable and does not require any context modeling. The single-pass mode alone yields speed-ups of around 1.3x for the encoder and 1.15x for the decoder at virtually no decrease in compression rate. When enabling the raw-coding mode after four (three) bit planes, the speed-up rises to 1.4x (1.5x) for the encoder and 1.4x (1.6x) for the decoder. Low-resolution sequences benefit more from the sample-parallel raw-coding mode, as their relative increase in parallelism is higher compared to 4K/UHD sequences.

Next, an alternative block coder from the literature, the *Bitplane Coder with Parallel Coefficient Processing* (BPC-PaCo), is evaluated. By giving up the context adaptiveness, it increases the parallelism in the block coder by processing a 64x64 code block concurrently with 32 arithmetic coders in a tightly synchronized fashion and interleaving their bit streams in a deterministic manner. An experiment is conducted that shows how a probability model averaged from a representative set of test sequences generalizes well and yields competitive compression efficiency. A combination of BPC-PaCo with the single-pass mode is proposed. For the demanding "VQEG crowd run" sequence, it increases the speed-up over the original *Embedded Block Coder with Optimized Truncation* (EBCOT) implementation from 2.15x (BPC-PaCo with two passes) to 2.6x (proposed BPC-PaCo with single-pass mode) and increases the PSNR penalty (w.r.t. EBCOT) from at most 0.7 dB to at most 1 dB.

A theoretical runtime model is formulated that allows for predicting the runtime of a kernel on another GPU. The model is suitable also for complex data-dependent routines. The prediction accuracy is evaluated for two such kernels with three GPUs and shows good results for devices within the same generation.

After the time-consuming context modeling and arithmetic coding routines are sped up, the *Post-Compression Rate-Distortion Optimization* (PCRD-Opt.) and packetization routines start to weigh in more significantly. An algorithm is presented and evaluated that searches for the optimal truncations points (given an available bit rate budget) and builds the entire code stream on the GPU in parallel, reducing the amount of data that has to be transferred back into host memory to a minimum.

Lastly, the first ever GPU-decoder realization of the new JPEG XS standard is presented. JPEG XS was designed from the grounds to be a low complexity codec and for the first time explicitly demanded GPU-friendliness already in the call for proposals. Starting at bit rates above 1 bpp, the decoder is around 2x faster compared to the original JPEG 2000 and 1.5x faster compared to JPEG 2000 with the fastest evaluated entropy coder (BPC-PaCo with single-pass mode). With a high-end GeForce GTX 1080, a throughput of around 200 fps is achieved for a UHD 4:4:4 sequence. On a PCI 2.0 16x connection, however, the throughput is limited at 125 fps, when transferring decoded frames back to host memory in parallel.

Finally, all coders are evaluated in terms of quality and throughput. A case study examines the use case of transmitting video over a bandwidth-limited channel, e.g. into the cloud, and it is shown that it is beneficial to be able to select from a multitude of codecs with different computation complexity trade-offs.

Keywords: JPEG 2000, EBCOT, JPEG XS, GPGPU, CUDA

II. Zusammenfassung

Diese Arbeit befasst sich mit der Entwicklung eines GPU-freundlichen, intra-only, Wavelet-basierten Videokompressionsverfahrens mit hohem Durchsatz, das für visuell verlustfreie Anwendungen optimiert ist. Ausgehend von der Beobachtung, dass der JPEG 2000 Entropie-Kodierer ein Flaschenhals ist und für Anwendungen mit mittlerer bis hoher Bitrate zu komplex erscheint, werden verschiedene algorithmische Änderungen vorgeschlagen und bewertet.

Zunächst wird der *Selective Arithmetic Coding Mode* aus JPEG 2000 Part-2 parallel auf der GPU realisiert, wobei sich die Erhöhung des Durchsatzes hierdurch als begrenzt zeigt. Stattdessen werden zwei nicht mit dem JPEG 2000 Standard kompatible Änderungen vorgeschlagen, die (1) die für hohe Bitraten weniger relevante Funktionalität aufgeben Bitströme innerhalb einer Bitebene beschneiden zu können (Single-Pass-Modus) und (2) einen echten Rohcodierungsmodus einführen, der sample-weise parallelisierbar ist und keine aufwendige Kontextmodellierung erfordert. Der Single-Pass-Modus allein führt bei einer vernachlässigbaren Verringerung der Kompressionsrate zu einer Beschleunigung von etwa 1,3x für den Encoder und 1,15x für den Decoder. Wenn der Rohcodierungsmodus nach vier (drei) Bitebenen aktiviert wird, steigt die Geschwindigkeit auf 1,4x (1,5x) für den Encoder und 1,4x (1,6x) für den Decoder. Niedrigaufgelöste Sequenzen profitieren stärker vom hochparallelen Rohcodierungsmodus, da dort die relative Steigerung der Parallelität im Vergleich zu 4K-Sequenzen höher ist.

Als nächstes wird ein alternativer Entropiekodierer aus der Literatur, der *Bitplane Coder with Parallel Coefficient Processing* (BPC-PaCo), evaluiert. Durch Aufgabe der Kontextadaptivität wird die Parallelität im Blockkodierer erhöht, indem ein 64x64-Codeblock gleichzeitig mit 32 arithmetischen Kodierern auf eng synchronisierte Weise verarbeitet wird und die Bitströme auf deterministische Weise verschachtelt werden. Es wird gezeigt und evaluiert, wie ein aus einem repräsentativen Satz von Testsequenzen ein gemitteltes statisches Wahrscheinlichkeitsmodell abgeleitet werden kann und dass dessen Kompressionseffizienz ausreichend nah an den auf die Sequenzen zugeschnittenen Modellen liegt. Es wird eine Kombination von *BPC-PaCo* mit dem *Single-Pass-Modus* vorgeschlagen. Für die anspruchsvolle „VQEG-Crowd-Run“-Sequenz erhöht dies die Geschwindigkeit gegenüber dem JPEG 2000 Entropiekodierer, *Embedded Block Coder mit Optimized Truncation* (EBCOT), von 2,15x (BPC-PaCo mit zwei Pässen) auf 2,6x (Kombination aus BPC-PaCo mit Single-Pass-Modus), wobei sich die Verminderung des Spitzen-Signal-Rausch-Verhältnis (PSNR) von 0,7 dB auf etwa 1 dB erhöht.

Es wird des Weiteren ein theoretisches Laufzeitmodell formuliert, das es ermöglicht die Laufzeit einer Routine auf einer anderen GPU vorherzusagen. Das Modell ist auch auf komplexe datenabhängigen Algorithmen anwendbar. Die Vorhersagegenauigkeit wird exemplarisch für zwei Kernels auf drei Grafikkarten untersucht mit dem Ergebnis, dass die Laufzeit genau vorhergesagt wird, aber nur innerhalb einer GPU-Serie.

Nachdem die zeitaufwendigen Routinen des Kontextmodellierers und arithmetischen Coders beschleunigt wurden, beginnen die *Post-Compression Rate-Distortion Optimization* (PCRD-Opt.) und die Paketisierungsroutinen stärker ins Gewicht zu fallen. Es wird ein Algorithmus vorgestellt, der im Fall eines limitierten Bitratenbudgets die optimalen Bitstrom-Beschneidungspunkte findet und den gesamten Code-Stream parallel auf der GPU zusammenbaut. So wird die Datenmenge, die zurück von der GPU in den Hostspeicher übertragen werden muss, auf ein Minimum reduziert.

Schließlich wird erstmals für den neuen Kompressionsstandard JPEG XS eine parallel GPU-Umsetzung vorgeschlagen und evaluiert. JPEG XS wurde von Grund auf als Low Complexity Codec konzipiert und forderte erstmals explizit GPU-Freundlichkeit bereits im *Call for Proposals*. Ab Bitraten über 1 bpp ist der Decoder etwa 2x schneller im Vergleich zu JPEG 2000 und 1,5x schneller im Vergleich zu JPEG 2000 mit dem schnellsten hier vorgestellten Entropiekodierer (BPC-PaCo mit Single-Pass-Modus). Mit einer High-End-GeForce GTX 1080 wird ein Durchsatz von rund 200 fps für eine UHD-4:4:4-Sequenz erreicht. Bei einer PCI 2.0 16x-Verbindung ist der Durchsatz jedoch auf 125 fps begrenzt, wenn dekodierte Bilder parallel zurück in den Hostspeicher übertragen werden.

Abschließend werden alle vorgestellten Verfahren hinsichtlich Qualität und Durchsatz verglichen. In einer Fallstudie wird die Übertragung von Video über einen bandbreitenbegrenzten Kanal, z.B. in die Cloud, untersucht und gezeigt, dass es von Vorteil ist aus einer Vielzahl von Codecs mit unterschiedlichen Balancen zwischen Kompressionsrate und Laufzeit auswählen zu können.

Schlüsselwörter: JPEG 2000, EBCOT, JPEG XS, GPGPU, CUDA

III. Acknowledgements

The idea and motivation for this work rose during the exciting years of work on the easyDCP software in the Fraunhofer IIS's Digital Cinema research group. easyDCP Player was the first big project in my career and became maybe the first commercial software-only solution that allowed to playback digital cinema packages on a consumer PC using commodity hardware, requiring three GPUs in the beginning. For the idea and possibility to join the Digital Cinema team, I want to thank first and foremost Dr. Heiko Sparenberg and Dr. Siegfried Fößel. I also want to thank the remaining easyDCP team, in particular, my office roommate Carsten Feldheim and fellow GPU developers Dr. Miguel Angel Martinez del Amor and Bilal Ahmed who were always up for bouncing ideas and figuring out yet another way of re-implementing the same kernel, hoping to squeeze out another millisecond.

Additionally I owe many thanks to the JPEG XS team Dr. Joachim Keinert, Dr. Thomas Richter and Wolfgang Heppner.

My thanks also go to my PhD advisor Prof. Dr. Karlheinz Brandenburg for providing invaluable feedback and input throughout the course of this work and fostering exchange among the fellow PhD students at TU Ilmenau. And I would like to thank PD Dr. Thomas Wittenberg for reviewing this work and providing key advice based on a long academic career and expert knowledge in image processing.

Lastly, carrying out a PhD thesis partly on the job, but oftentimes off the job, demands to sacrifice many hours of spare time and more importantly family time. I cannot thank my wife Stefanie enough, who has had my back during all these years, and my son Benno, who gave me the strength and motivation to go the last extra mile and finish this work.

Table of Contents

1 Introduction **8**

1.1 Motivation 8

1.2 Research Objectives 9

1.3 Publications 10

1.3.1 PCS'16 - Sample-parallel execution of EBCOT in fast mode 10

1.3.2 ICIP'17 - GPU-friendly EBCOT variant with single-pass scan order 10

1.3.3 MMSP'17 - Evaluation of GPU/CPU co-processing models for JPEG 2000 packetization 11

1.3.4 PCS'18 - Decoding JPEG XS on a GPU 11

1.3.5 SPIE'17 - Parallel efficient rate control methods for JPEG 2000 (Co-authored) 12

2 Technical Background **13**

2.1 Image Compression Theory 13

2.1.1 Error 13

2.1.2 Classes 14

2.2 Wavelet based still image compression 15

2.3 General Purpose Computing on GPUs 18

2.3.1 GPGPU Frameworks 18

2.3.2 CUDA Architecture 19

2.3.3 CUDA Programming Model 20

2.3.4 Data Transfer 23

2.4 Parallel Primitives 24

2.4.1 Scan 25

2.4.2 Reduction 25

3 State of the Art **26**

3.1 JPEG 2000 26

3.2 BPC-PaCo 27

3.3 Low Complexity Coding 27

3.3.1 Professional Video Production 27

3.3.2 Video-over-IP 28

3.3.3 Requirements on Video Codec 28

3.3.4 Codecs 29

3.4 Standardization 30

3.4.1 JPEG XS 30

3.4.2 JPEG 2000 High Throughput 31

4 JPEG 2000 .... **32**

4.1 Description of the Encoder 32

4.1.1 Multi-Component Transform 32

4.1.2 Wavelet Transform 33

4.1.3 Quantization 33

4.1.4 Block Coding 33
4.2 Exploiting Parallelism in JPEG 2000 40
4.2.1 Decoupled Context Modeling and Arithmetic Coding in EBCOT Encoder 40
4.2.2 Sample-parallel EBCOT encoding 41
4.2.3 Conclusion 43

5 JPEG 2000 Compliant Extensions 45

5.1 Motivation: Poor compression efficiency in lower bit planes 45
5.2 Selective Arithmetic Coding Bypass style 46
5.3 Sample-Parallel Execution of EBCOT in Fast Mode 47
5.4 Parallel Bit-stuffing 49
5.5 Experimental Results 51
5.6 Conclusion 54

6 Rate Control and Parallel Packetization 55

6.1 Proposed GPU/CPU Co-Processing Models 55
6.2 Parallel Post Compression Rate Control Optimization 56
6.3 Parallel Packetization 57
6.4 Experimental Results 61
6.5 Conclusion 62

7 JPEG 2000 Incompliant Extensions 64

7.1 Single-Pass Scan Order 64
7.2 Raw Bit plane Coding 66
7.3 Experimental Results 67
7.4 Conclusion 72

8 BPC-PaCo 73

8.1 Stationary Probability Model 74
8.2 Fixed-Length Coding 74
8.3 Two or three passes per bit plane 75
8.4 Proposed change: Single-Pass Scan Order 76
8.5 Look-up Table Structure 78
8.6 Probability Estimates Variance 78
8.7 Experimental Results 83
8.8 Conclusion 87

9 Runtime Model 90

9.1 CUDA Benchmark-based Runtime Model 90
9.2 Case Study A: *Context Modeling Kernel* 93
9.3 Case Study B: *Arithmetic Coding Kernel* 102
9.4 Conclusion 108

10 JPEG XS 109

10.1 Description of the Encoder 109
10.1.1 Color Transform 110
10.1.2 Wavelet Transform 110

10.1.3 Magnitude Bit Coding 111
10.1.4 Sign Bit Coding 111
10.1.5 MSB Position Coding 111
10.1.6 Truncation Position Signaling 112
10.1.7 Code stream Structure 112
10.2 Encoding JPEG XS on a GPU 113
10.3 Decoding JPEG XS on a GPU 114
10.3.1 Depacketization 114
10.3.2 Reconstruction of MSB Position 114
10.3.3 Unpacking Magnitude and Sign bits 115
10.3.4 Inverse DWT 116
10.3.5 Inverse Reversible Color Transform 116
10.4 Experimental Results 117
10.5 Conclusion 119
11 Evaluation 120
11.1 Impact of Quantization 120
11.2 Quality 123
11.3 Throughput 125
11.4 Case Study: Compressed Video Transmission 131
12 Summary 135
13 Conclusion 138
13.1 Contributions 140
13.2 Limitations and Future Work 141
13.2.1 Limitations 141
13.2.2 Future Work 141
Bibliography 143

1 Introduction

1.1 Motivation

The aim of this work is to investigate possibilities to create a GPU-friendly intra-only, Wavelet-based video image compression algorithm that is optimized for visually lossless applications. “GPU-friendly” here means that the codec can be strongly parallelized and leverage massively parallel architectures such as a graphics card that today is equipped with thousands of computing cores. GPUs use very lightweight threads that ideally process no more than a handful of samples each.

The main motivation for this work stems from the author’s previous position in the *digital cinema* research group of Fraunhofer IIS. During this time, cinema worldwide transitioned from distributing analog film reels to shipping hard drives containing the so-called *Digital Cinema Package* (DCP) format [1]. In order to guarantee a pristine image quality while still benefitting from relatively high compression efficiency (among other reasons), the creators chose the intra-frame compression ISO IEC standard JPEG 2000 [2]. Its compression efficiency comes at the cost of a high complexity that makes the codec very computationally expensive. While cinema servers contain a dedicated JPEG 2000 FPGA decoder to guarantee real-time playback, authoring and quality control tools would be realized more flexibly and more cost effectively as software solutions. DCP mastering stations require fast encoders. Real-time decoding is required for pre-release screenings as well as quality control. Having been involved in the JPEG 2000 *Digital Cinema* profile standardization, Fraunhofer IIS develops such software solutions. The author’s master thesis tackled the problem of developing a decoder capable of playing back DCPs in real-time using commodity hardware. At the time, NVIDIA released the first version of its *Compute Unified Device Architecture* (CUDA) that permitted developers to execute arbitrary programs on the *Graphics Processing Unit* (GPU) leveraging its hundreds or thousands of cores [3]. Over the next years, a CUDA-accelerated JPEG 2000 codec has been developed. While some coding steps of JPEG 2000 are perfectly suited for an execution on a massively parallel device, others, namely the entropy coder, require considerable effort. On top of that, it soon became clear that the digital cinema profile - or the entire entropy coder named *Embedded Block Coder with Optimized Truncation* (*EBCOT*) for that matter - had not been specified with massively parallel architectures in mind. The primary focus was to deliver superior quality at very low bit rates rather than lowering the complexity at medium to high bit rates.

More recently, the *Interoperable Master Format* (IMF) [4] has been created as a business-to-business master exchange format superior to the predominant *Apple ProRes* format and compression scheme [5]. Most notably, *Netflix* asks their suppliers to deliver all UHD and HDR content in this format [6]. While initially the broadcast profile had been used, the latest IMF extension specifies that the new IMF JPEG 2000 profiles be employed which permit bit rates up to 800 Mbit/s as opposed to the maximal 250 Mbit/s in the cinema profiles [7–9]. For a 90 minutes feature film this amounts to a maximum theoretical size of over 500 GB per IMF package. This underlines that compression is essential, but at the same time creatives expect to be empowered to playback movies in real-time and render them faster than real time, requiring high-throughput codecs.

When computing power is scarce, often cloud computing is a solution. However, for encoding movies the challenge is to transfer the input data, often uncompressed video, into the cloud in the first place. Chapter 11.4 will examine the relationship between available upload bandwidth, compression efficiency and encoding throughput in more detail.

Beyond digital cinema, other domains such as live TV production require *Low Complexity Coding*. In order to be able to reuse existing IP or SDI infrastructure, a lightweight compression would often suffice to accommodate for an increase in resolution or frame rate. For example, a 60 fps 4:2:2 signal with 10 bits/sample amounts to a raw bandwidth of 9.27 Gbit/s. Taking into account the protocol overhead, a compression ratio of around 1:4 would be sufficient for transmitting the signal over an existing dual SDI infrastructure with 3 Gbit/s bandwidth. Similarly, an UHD-2 (8K) signal with 37.08 Gbit/s could be sent over a 10 Gbit/s Ethernet IP infrastructure, if it were compressed with a little over 1:4. Since higher complexity typically demands chips with a larger die size and this increases the cost, a tailored codec that is as simple as possible is preferred.

1.2 Research Objectives

This work is concerned with the question how high quality intra-frame image compression schemes such as JPEG 2000 can be efficiently executed on a massively parallel device architecture. It also asks how the inherent trade-off between compression efficiency and computational complexity can be fine-tuned when targeting parallel devices. Can the throughput be increased considerably while keeping the cost of decreased compression efficiency at a minimum?

The JPEG 2000 standard already offers some optional advanced features to reduce the computational complexity. Though the use of these options is prohibited in the cinema, broadcast and IMF profiles, it is nonetheless worthwhile to examine their potential for GPU-accelerated image compression and decompression. Is there much to gain by permitting these features in the JPEG 2000 profiles employed by cinema and broadcast industry? (Chapter 5)

Extending beyond the scope of the JPEG 2000 standard, how can the compression scheme be altered when giving up compatibility? Can the throughput bottleneck be remedied by designing a drop-in replacement for the complex entropy coder? (Chapter 7)

Lastly, how does the requirement to expose as much parallelism as possible to take advantage of massively parallel architectures affect the design of a new low complexity codec? At the time this manuscript is written, in 2019, the JPEG XS standard is being standardized. It is the first JPEG codec to require explicitly in its *call for proposals* a compatibility with GPU architectures. As part of this PhD thesis, the first GPU-accelerated JPEG XS decoder has been designed and realized. (Chapter 10.3)

1.3 Publications

The achieved research results have been published in the IEEE conference papers listed below. All publications underwent a full-manuscript (not abstract-only) peer-review process and are available for download on *IEEE Xplore*.[1] A more comprehensive list of both published and unpublished contributions is given in chapter 13.1.

1.3.1 PCS'16 - Sample-parallel execution of EBCOT in fast mode

Bruns, Volker; Martinez-del-Amor, Miguel A., "Sample-parallel execution of EBCOT in fast mode", ***Proceedings 2016 Picture Coding Symposium (PCS)*****, pp. 1–5, Nuremberg, Germany, December 2016**

JPEG 2000's most computationally expensive building block is the *Embedded Block Coder with Optimized Truncation* (EBCOT). This paper evaluates how encoders targeting a parallel architecture such as a GPU can increase their throughput in use cases where very high data rates are used. The compression efficiency in the less significant bit planes is then often poor and it is beneficial to enable the *Selective Arithmetic Coding Bypass* style (fast mode) in order to trade a small loss in compression efficiency for a reduction of the computational complexity. More importantly, this style exposes a more finely grained parallelism that can be exploited to execute the raw coding passes, including bit stuffing, in a sample-parallel fashion. For a latency- or memory critical application that encodes one frame at a time, EBCOT's tier-1 is sped up between 1.1x and 2.4x compared to an optimized GPU-based implementation. When a low GPU occupancy has already been addressed by encoding multiple frames in parallel, the throughput can still be improved by 5% for high-entropy images and 27% for low-entropy images. Best results are obtained when enabling the fast mode after the fourth significant bit plane. For most of the test images, the compression rate is within 1% of the original.

1.3.2 ICIP'17 - GPU-friendly EBCOT variant with single-pass scan order

Bruns, Volker; Martinez-del-Amor, Miguel A.; Sparenberg, Heiko, "GPU-friendly EBCOT variant with single-pass scan order and raw bit plane coding", ***2017 IEEE International Conference on Image Processing (ICIP)*****, pp. 3230–3234, Beijing, China, September 2017**

A major drawback of JPEG 2000 is the computational complexity of its entropy coder. This paper proposes two alterations to the original algorithm that seek to improve the trade-off between compression efficiency and throughput. Firstly, magnitude bits within a bit plane are not prioritized by their significance anymore, but instead coded in a single pass instead of three, reducing the amount of expensive memory accesses at the cost of fewer truncation points. Secondly, low bit planes can entirely

[1] https://ieeexplore.ieee.org/search/searchresult.jsp?newsearch=true&searchWithin=%22First%20Name%22:volker&searchWithin=%22Last%20Name%22:bruns

bypass the arithmetic coder and thus do not require any context modeling. Both the encoder and decoder can process such bit planes in a sample-parallel fashion. In contrast, the standard's *Selective Arithmetic Coding Bypass* option still requires context modelling as only the symbols from two of the three passes bypass the arithmetic coder.

Experiments show average speed-ups of 1.6x (1.8x) for the encoder and 1.5x (1.9x) for the decoder, when beginning raw-coding after the fourth (third) bit plane, while the data rate increases only by 1.15x (1.3x).

1.3.3 MMSP'17 - Evaluation of GPU/CPU co-processing models for JPEG 2000 packetization

Bruns, Volker; Martinez-del-Amor, Miguel A.; Sparenberg, Heiko, "Evaluation of GPU/CPU co-processing models for JPEG 2000 packetization", ***2017 IEEE 19th International Workshop on Multimedia Signal Processing (MMSP)*****, pp. 1–6, Luton, UK, October 2017**

With the bottom-line goal of increasing the throughput of a GPU-accelerated JPEG 2000 encoder, this paper evaluates whether the post-compression rate control and packetization routines should be carried out on the CPU or on the GPU. Three co-processing models that differ in how the workload is split between the CPU and GPU are introduced. Both routines are discussed and algorithms for executing them in parallel are presented. Experimental results for compressing a detail-rich UHD sequence to 4 bits/sample indicate speed-ups of 200x for the rate control and 100x for the packetization compared to the single-threaded implementation in the commercial *Kakadu* library. These two routines executed on the CPU take four times as long as all remaining coding steps on the GPU and therefore present a bottleneck. Even if the CPU bottleneck could be avoided with multi-threading, it is still beneficial to execute all coding steps on the GPU as this minimizes the required device-to-host transfer and thereby speeds up the critical path from 17.2 fps to 19.5 fps for 4 bits/sample and to 22.4 fps for 0.16 bits/sample.

1.3.4 PCS'18 - Decoding JPEG XS on a GPU

Bruns, Volker; Richter, Thomas; Ahmed, Bilal; Keinert, Joachim; Fößel, Siegfried, „Decoding JPEG XS on a GPU", ***Proceedings 2018 Picture Coding Symposium (PCS)*****, San Francisco, USA, June 2018**

JPEG XS is an upcoming lightweight image compression standard that is especially developed to meet the requirements of compressed video-over-IP use cases. It is designed with not only CPU, FPGA or ASIC platforms in mind, but explicitly also targets GPUs. Though not yet finished, the codec is now sufficiently mature to present a first NVIDIA CUDA-based GPU decoder architecture and preliminary performance results. On a 2014 mid-range GPU with 640 cores a 12 bit UHD 4:2:2 (4:4:4) can be decoded with 54 (42) fps. The algorithm scales very well: on a 2017 high-end GPU with 2560 cores the throughput increases to 190 (150) fps. In contrast, an optimized GPU-accelerated JPEG 2000 decoder takes 2x as long for high compression ratios that yield a PSNR of 40 dB and 3x as long for lower compression ratios with a PSNR of over 50 dB.

1.3.5 SPIE’17 - Parallel efficient rate control methods for JPEG 2000 (Co-authored)

Martínez-del-Amor, Miguel Á.; Bruns, Volker; Sparenberg, Heiko, „Parallel efficient rate control methods for JPEG 2000”, Proc. SPIE 10396, Applications of Digital Image Processing XL, San Diego, USA, Sep. 2017

Since the introduction of JPEG 2000, several rate control methods have been proposed. Among them, *Post-Compression Rate-Distortion Optimization* (PCRD-Opt) is the most widely used and the one recommended by the standard. The approach followed by this method is to first compress all code blocks, and subsequently, optimally truncate the set of generated bit streams according to the target bit rate constraint. The literature proposes various strategies on how to estimate ahead of time where a block will be truncated in order to stop the execution prematurely and save time. However, none of them has been defined bearing in mind a parallel implementation.

Today, multi-core and many-core architectures are becoming popular for JPEG 2000 codecs implementations. Therefore, in this paper, we analyze how some techniques for efficient rate control can be deployed in GPUs. In order to do that, the design of our GPU-based codec is extended, allowing stopping the process at a given point. This extension also harnesses a higher level of parallelism on the GPU, leading to up to 40% of speedup with 4K test material on a Titan X. In a second step, three selected rate control methods are adapted and implemented in our parallel encoder. A comparison is then carried out, and used to select the best candidate to be deployed in a GPU encoder, which gave an extra 40% of speedup in those situations where it was really employed.

2 Technical Background

2.1 Image Compression Theory

The goal of image compression is to reduce the number of bits required to store the image. When the reconstructed image $\hat{I}$ is bit-identical to the original image *I*, the compression scheme is referred to as *lossless*. If the reconstructed image is similar, but not identical to the original image, the compression is *lossy*. The difference between the reconstructed image and the original image is called the *error* (or sometimes *noise*). The goal of image compression schemes is to reduce the required size as much as possible while keeping the error minimal.

2.1.1 Error

Various ways of computing the error can be found in the literature. A simple error measure is the *Mean Squared Error* (MSE), which corresponds to the *Euclidean Distance*. The MSE accumulates the squared differences of all corresponding pixels and then normalizes by the number of pixels. Given the width *w* and height *h* of a monochrome image, the MSE is defined as:

$$\text{MSE} = \frac{1}{wh} \sum_{x=0}^{w-1} \sum_{y}^{h-1} (I(x,y) - \hat{I}(x,y))^2$$

Often, the *Peak Signal to Noise Ratio* (PSNR) is stated instead of the MSE. Given that pixel values are stored with a dynamic range of *b* bits, the PSNR is defined as:

$$PSNR = 10 * \log_{10}(\frac{(2^b - 1)^2}{MSE}$$

Since the PSNR is a logarithmic scale, it is expressed with the unit *dB*. Higher values indicate a smaller error.

A limitation of the MSE or PSNR is that it does not take into account the human visual system (HVS) and as such must be used with care when comparing the quality of reconstructed images [10]. To a human, the subjective image quality is influenced by how the error is distributed in the image. For instance, a small pixel error would be much more noticeable in the context of a blue sky than in the context of a tree with many detailed leaves. More involved quality metrics such as the *Structured Similarity Index* (SSIM) [10] or Netflix's *Video Multi-Method Assessment Fusion* (VMAF) [11] should be used when evaluating the perceived quality of an image compression scheme. In this work, however, the codecs are operated in a domain, where the reconstructed image is required to be *visually lossless*, which implies that a human cannot tell the reconstructed image apart from the original one, even at the presence of noise. In the literature, a value of 40 db PSNR is sometimes cited as the threshold that marks the beginning of the visually lossless coding regime, though it is dependent on image content [12, 13]. The majority of the experiments reported in this work regard only compression rate and performance.

Those experiments where the impact on the image quality is evaluated, the quality is expressed in terms of dB PSNR and as such must be regarded with care as the order in which methods are ranked might change when evaluated with a more sophisticated quality measure that takes into account the human visual perception.

2.1.2 Classes

Image compression schemes can be classified among various dimensions.

Lossy vs. lossless – As described in the previous chapter, codecs can operate lossy or losslessly. A requirement for lossless coding is that the compressed input signal is reconstructed perfectly, i.e. bit-identically. This is easiest accomplished by performing all operations with integer operations in order to prevent rounding errors, though this can also be achieved with floating point arithmetic. Aside from rounding errors, information is typically discarded in the encoder's quantizer and so lossless codecs do not quantize.

Intra vs. Inter - An important attribute is whether each image in a video sequence is compressed independently (*intra-frame compression*) or whether the compressor utilizes the observation that subsequent images are often similar to one another and expresses one image in terms of another one (*inter-frame compression*). Inter-frame compression codecs cannot only exploit redundancies within a single frame (intra), but can additionally exploit temporal redundancies (inter) and therefore usually achieve a higher compression efficiency. A drawback is that transmission or storage errors can propagate along the causal chain of coded frames, which makes them much more noticeable to the viewer. Another drawback is that frame-accurate random access is more complicated, as it typically requires that, multiple other frames are decoded first, which the selected frame depends upon. For this reason, codecs used during the movie production and post-production typically do not use inter-frame compression.

High efficiency vs. low complexity – In distribution scenarios where the bandwidth is very expensive (e.g. TV satellite distribution) or where it is simply limited (video streaming) codecs are employed that compress the signal as much as possible at the cost of noticeable but tolerable quality loss. A comparatively high complexity of employed codecs is acceptable, because in these two examples, the number of worldwide receivers is by far large enough to justify the high non-recurring engineering costs required for specialized hardware realizations (e.g. video processor on GPU or ASIC) of the decoders in order to enable real-time playback in consumer TVs, computers or smartphones. In contribution scenarios, such as on-board compression in professional film cameras or video-over-IP transmission boxes, however, FPGA-based solutions are preferred. Higher complexity then equals larger die size, which equals higher cost. Given that *Gigabit Ethernet* is now readily available, a codec is desired that compresses just well enough to achieve the available bit rate at the lowest possible complexity.

Recent examples of high efficiency inter-frame codecs are *High Efficiency Video Codec* (HEVC) standardized by ISO/IEC [14] or *AOMedia Video 1* (AV1) developed by the *Alliance for Open Media* (AOMedia) [15].

This work, however, is only concerned with intra-frame, Wavelet-based contribution codecs of a comparatively low complexity. Arguably, JPEG 2000 cannot be regarded as a low-complexity codec, but the first half of this work focuses on reducing its complexity. JPEG XS has been designed from the beginning as a low-complexity codec, hence the name, which stands for extra small and extra speed.

2.2 Wavelet based still image compression

Both JPEG 2000 and JPEG XS are wavelet-based compression schemes that loosely follow the procedure outlined in the figure below. The encoder first seeks to concentrate the image's energy into as few samples as possible. To this end, the first step is typically to decorrelate the colors by transforming from the RGB color space into a luminance-chrominance color space. This step is referred to as the *Multi Component Transform* (MCT). Since the human visual system is more perceptible to changes in brightness than to changes in color, optionally the chrominance channels can be subsampled.

Table 2-1: Chroma Subsampling Schemes

Subsampling scheme	Chrominance channels
4:4:4	Not subsampled
4:2:2	Width is halved, height stays the same
4:2:0	Both width and height are halved

Next, each component (color channel) is individually wavelet-transformed with a *Discrete Wavelet Transform* (DWT). The wavelet transform poses an alternative to the *Discrete Cosine Transform* (DCT) that has been employed by many legacy codecs such as JPEG or MPEG-2 (used in DVDs). Whereas a cosine transform similar to a *Fourier* transform divides a localized window in the time-frequency plane uniformly, the wavelet transform seeks to make a trade-off between spectral and temporal localization. In the case of two-dimensional signals such as images, the time domain translates to spatial information. It achieves the desired property that low frequency image contributions are represented with higher spectral resolution (and lower spatial resolution) whereas high-frequency contributions are sampled with higher spatial resolution (and lower spectral resolution). From this follows the key difference that DWT-compressed images are less prone to block-artefacts as the image is first wavelet-transformed and then the individual subbands are divided into blocks. In lossy coding, blocks in lower-frequency bands are quantized less heavily than those in higher-frequency bands. In contrast, DCT-based codecs first split the image into blocks in the spatial-domain and then transform each block into the frequency domain. The remainder of this chapter explains this in more detail.

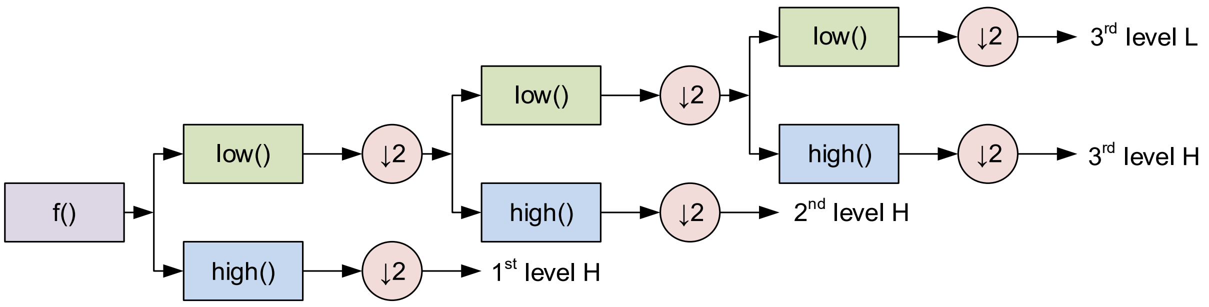


Figure 2-1 – Filter bank design of Fast Wavelet Transform

The *Fast Wavelet Transform* devised by S. Mallat formulates the DWT as a two-channel sub band coder as depicted in the above figure for the one-dimensional case [16]. While both multi-resolution coding and pyramidal algorithms were already known, Mallat built up on these approaches and introduced the use of orthogonal wavelets, devised by I. Daubechies the year before [17], as a model for decorrelating the signal at different resolution levels. It can be computed with quadrature mirror filters (QMF). A signal *f()* is divided into an *approximation* part and a *detail* part by processing it with a low-pass and high-pass filter. Both parts are subsequently downsampled in order to keep the combined length of discrete output signals equal to the input signal. For an input signal of length *N*, this process can be repeated recursively at most $log_2(N)$ times by filtering the approximation signal again and again.

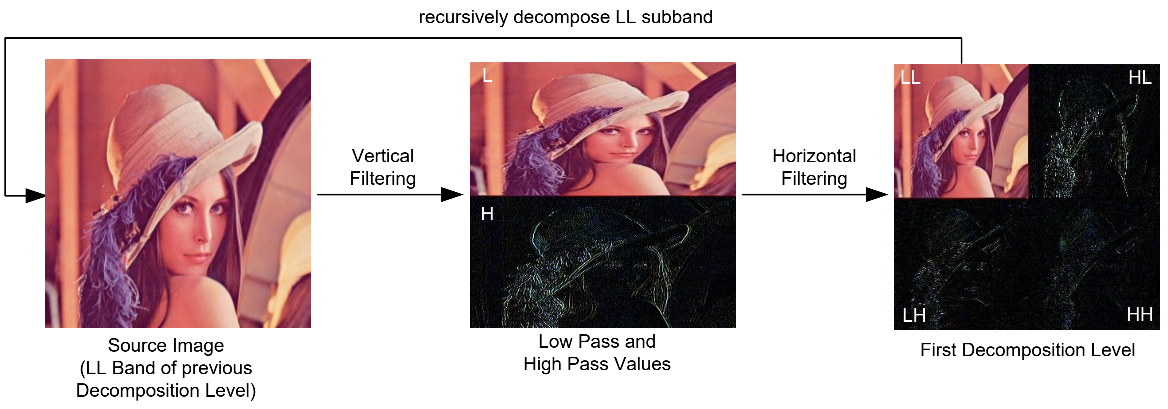

recursively decompose LL subband
Vertical Filtering
Horizontal Filtering
L
H
LL
HL
LH
HH
Source Image (LL Band of previous Decomposition Level)
Low Pass and High Pass Values
First Decomposition Level

Figure 2-2 - 2D dyadic wavelet decomposition

When extending this concept to the case of two-dimension signals, the transform's separability can be utilized. The previous figure shows how an input image is first filtered vertically yielding a subband *L* with the low-frequency *approximation* signal and a subband *H* with the high-frequency *detail* signal. The resulting image is then additionally filtered horizontally. Alternatively, the image can be filtered along the same direction twice when transposing the intermediate result in-between both operations. The resulting image contains four sub-bands. The *LL* band now contains a lower-resolution version of the original image. The *HL* and *LH* bands contain horizontal and vertical edges, respectively, and the *HH* band contains diagonal edges. The result after an additional wavelet level is depicted in Figure 2-3.

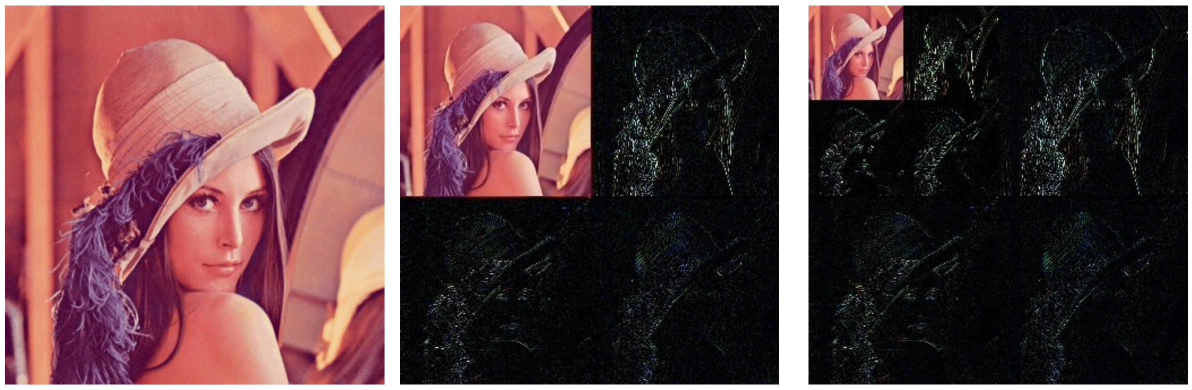


Figure 2-3 - Lenna (left) - 1-level decomposition of Lenna (center) - 2-level decomposition of Lenna (right) (H subbands are multiplied by factor 10 for illustrative purposes)

The individual wavelet subbands can then optionally be quantized. A larger quantization step size will cause coefficients to be stored with a lower accuracy. Prioritizing low frequencies over high frequencies, more bits are spent on storing coefficients from low-frequency subbands by employing smaller step sizes and increasing the step size for higher frequency bands.

The subbands can then be divided into non-overlapping blocks or precincts that are independently entropy-coded. Finally, the packetizer organizes the entropy-coded bit streams into an overall code stream comprised of individual packets and headers. The following diagram (Figure 2-4) presents a high-level overview of a wavelet-based image encoder.

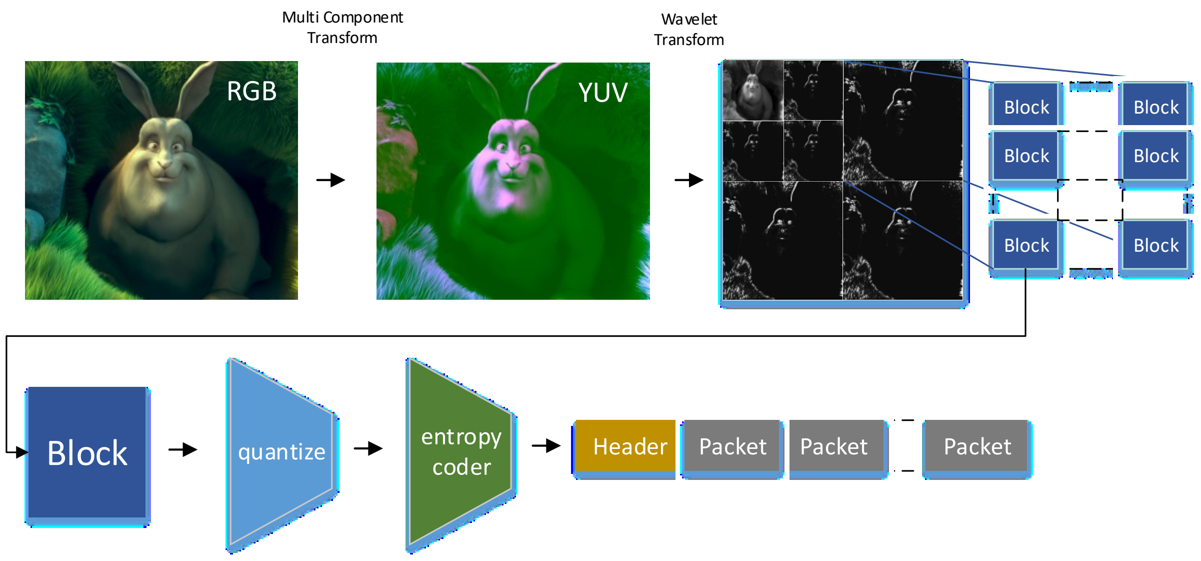


Figure 2-4 - Block Diagram of Wavelet-based Encoder

The major contrast to a DCT-based intra-frame image codec is that the image is first transformed and then divided into relatively large blocks. A DCT codec will divide the color-transformed channels into smaller blocks, e.g. of size 8x8, and then apply the DCT on each individual block. In order to leverage that adjacent blocks have DC-coefficients of similar magnitude, they are sometimes additionally coded with a differential pulse code modulation (DPCM). The AC coefficients of a single block also contain redundancy, which can be exploited by ordering them by frequency (“zig-zag order”) and applying a run-length or Huffman coding. For instance, legacy *JPEG, Apple ProRes* and *Avid DNxHD* follow this scheme. A major drawback is that such coders tend to produce block artefacts at medium to low bit rates. For a parallel decoder, an additional drawback is that DCT-blocks cannot immediately be decoded concurrently, as the DC-coefficients are usually DPCM-coded and have to be reconstructed first before the AC coefficients can be decoded.

2.3 General Purpose Computing on GPUs

GPUs have originally been designed to carry out solely graphics-processing related tasks. Driven by the ever more demanding computer games, GPUs have become very powerful parallel devices outnumbering the number of cores in a CPU by far (Figure 2-5). As the graphics pipeline became more customizable by means of writing shader programs researchers began to formulate general purpose computations in terms of graphics processing problems in order to be able to execute them on the GPU [18]. The concept of carrying out generic computing problems that have traditionally been executed on the CPU is referred to *General Purpose Computing on GPUs* (GPGPU). In 2007, NVIDIA released the first version of their CUDA toolkit, simplifying the task of GPGPU significantly.

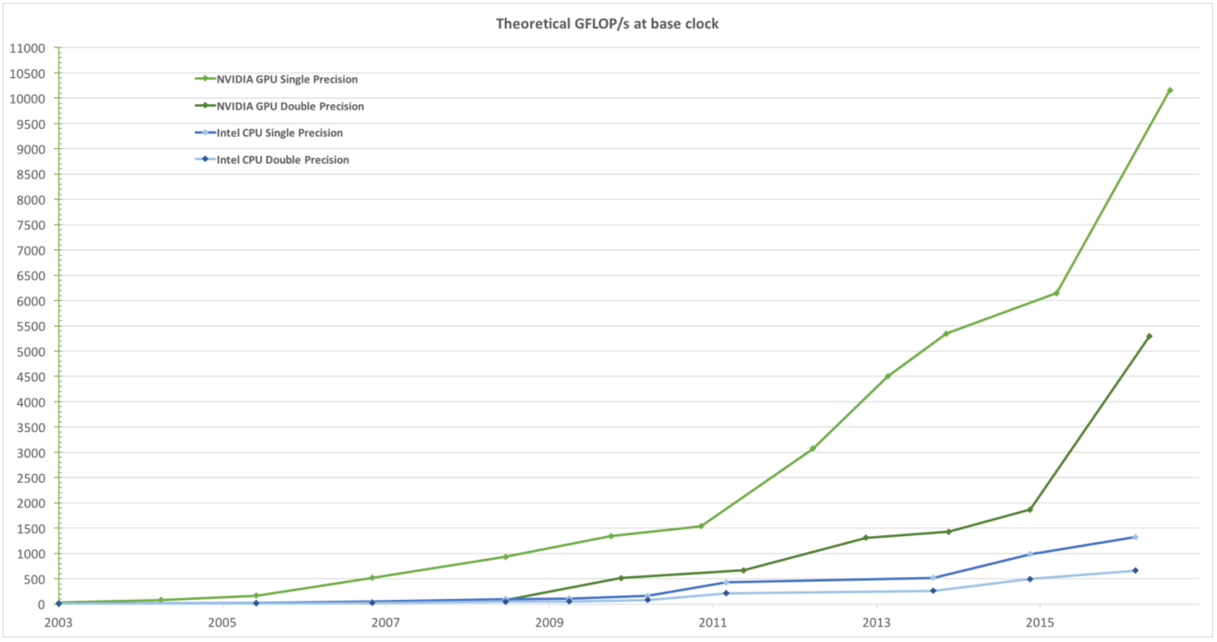


Figure 2-5 - Floating-Point Operations per Second for the CPU and GPU. Source: [19]

2.3.1 GPGPU Frameworks

CUDA is not the only compute API. Though it is free of charge, CUDA is a proprietary solution and only works with NVIDIA GPUs. This poses a limitation as most notably Apple computers cannot be easily targeted by CUDA-based software. Apple desktop computers have been using AMD cards exclusively in the past years.

The Khronos Group, most famous for having specified the graphics processing language OpenGL, has released an alternative solution called *Open Compute Language* (OpenCL) [20]. While many concepts are very similar to NVIDIA's CUDA, the single most striking difference is that device code is compiled online at runtime just as it is the case with OpenGL GLSL shaders. OpenCL devices are not limited to GPUs, but also the CPU is regarded as a parallel device. *Altera*, later acquired by *Intel*, has even released an SDK for writing FPGA code using OpenCL [21].

Other compute APIs are AMD's *Heterogeneous Compute* (HC) [22], Microsoft's *C++ AMP* [23] and Apple's *Metal* [24]. The OpenMP ("open multi-processing") API, managed by the "OpenMP architecture review board (ARB)" is known to provide a convenient way to parallelize for-loops by preceding them with the *parallel for* pragma directive. In version 4.0 it introduced the *target* keyword that allows offloading code to a particular device target such as a GPU [25].

The compute APIs use similar design concepts, but different terminology. A mapping of the different terms can be found at [22]. Various cross compilers or source-to-source translators have been developed, in particular to translate from CUDA to OpenCL [26, 27]. AMD recently released its *Heterogeneous-Computing Interface for Portability* (HIP). It is a C++ dialect that can be compiled either for AMD hardware using their HCC compiler or for NVIDIA hardware using the CUDA compiler [22]. A team at Google has developed an open source CUDA compiler [28].

The algorithms presented in this work have been realized with CUDA and compiled with NVIDIA's CUDA compiler *NVCC*. While some of the algorithms should be straightforward to translate to other compute APIs, others utilize CUDA-specific techniques such as register shuffling or SIMD group voting routines in order to achieve the highest performance. A comparison of how the proposed algorithms perform when realized in the various compute languages would be interesting and valuable, but has not been carried out as it is outside of the scope of this work.

2.3.2 CUDA Architecture

In NVIDIA terminology, a GPU is comprised of multiple Streaming Multiprocessors (SM) that each have a fixed number of cores. Figure 2-6 shows a multiprocessor of the NVIDIA *Pascal* architecture that is used in the *GeForce 10* series. It incorporates 64 single precision floating point cores partitioned into two blocks that each have their own scheduler and 32 KB register file.

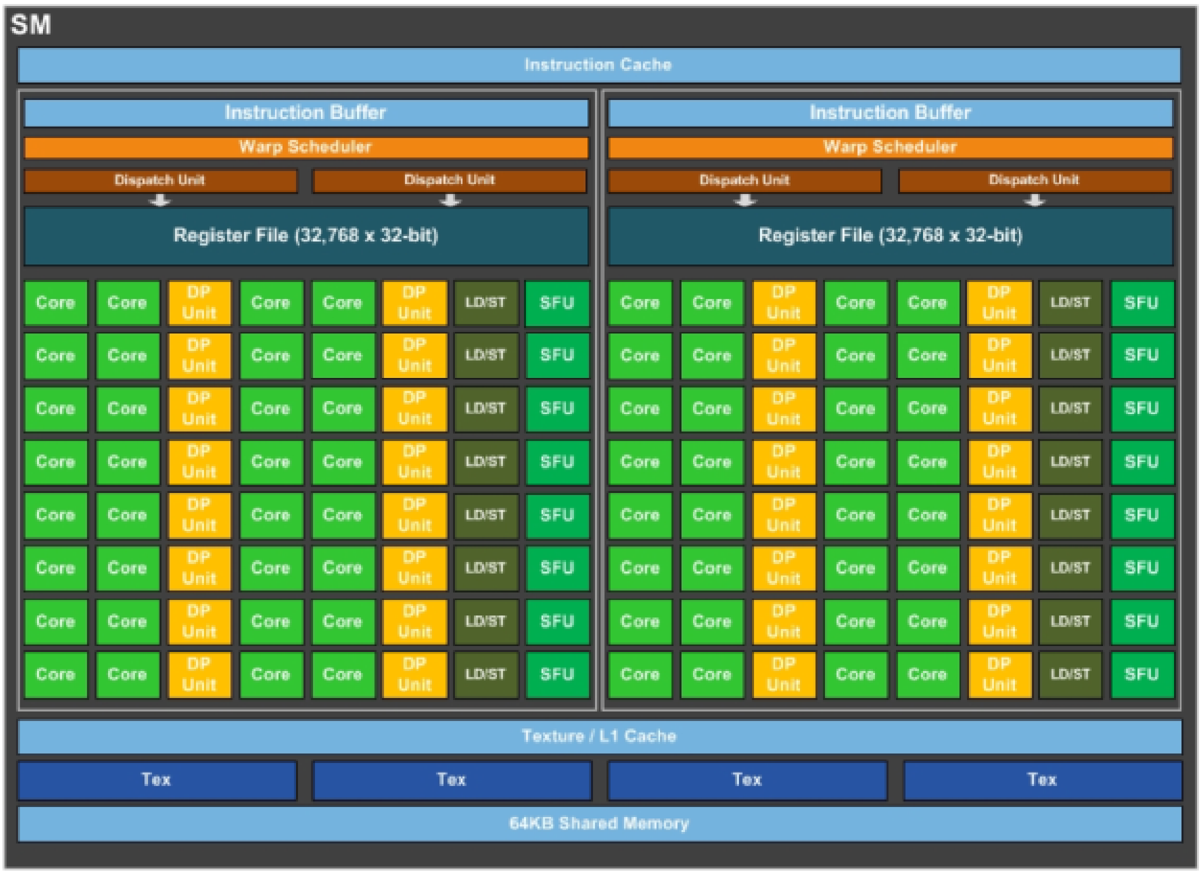


Figure 2-6 – NVIDIA Streaming Multiprocessor for *Pascal* architecture. Source: [29]

The 32 cores that share a single instruction scheduler can be regarded as a *Single Instruction Multiple Data* (SIMD) group, termed *warp* in CUDA terminology. The SIMD width has been 32 since CUDA 1.0. Fast interactions among threads in a SIMD group, but also across SIMD groups are possible by using the 64 KB shared memory.

2.3.3 CUDA Programming Model

CUDA introduces some extensions to the C language that are recognized by the NVIDIA C Compiler (NVCC). Functions that shall be executed on the GPU are termed *kernels*. GPU code is parsed offline and translated into an intermediate language. When a kernel is launched from a CPU (*host*) function, the caller assigns a threading layout to the kernel. The layout is comprised of a three-dimensional *grid* made up of three-dimensional *blocks*. Each point in the block represents a single thread and is identified by its integer x-, y- and z-coordinate (Figure 2-7).

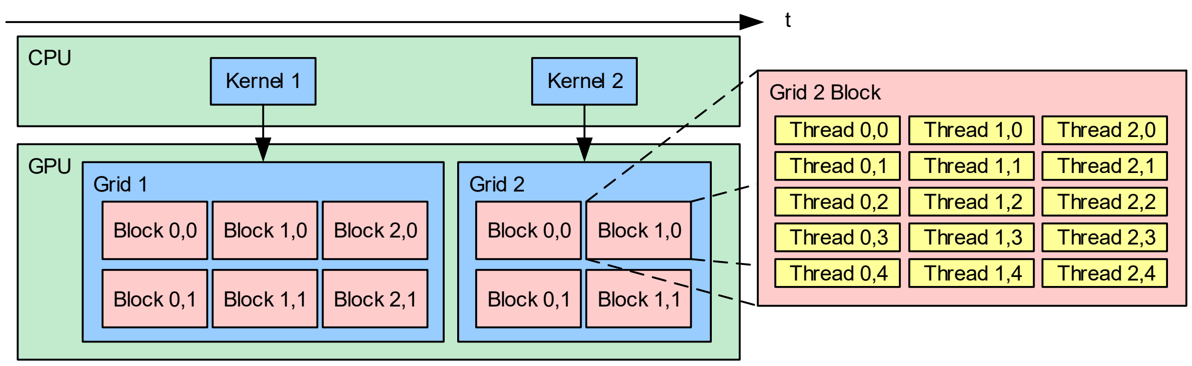


Figure 2-7 - CUDA thread batching model –comprised of grid, blocks and threads. Adapted from: [19]

The grid and block dimensions as well as the current block and thread IDs are available to the developer as built-in variables. This way the SIMD paradigm can be conveniently realized by defining a mapping from thread coordinates to input and output data inside the kernel. A block is executed by a single multiprocessor and so threads within a block can be synchronized and share data via the on-chip shared memory (Figure 2-8). Threads across blocks cannot be synchronized and no assumptions can be made as to the order in which the scheduler will swap in blocks.

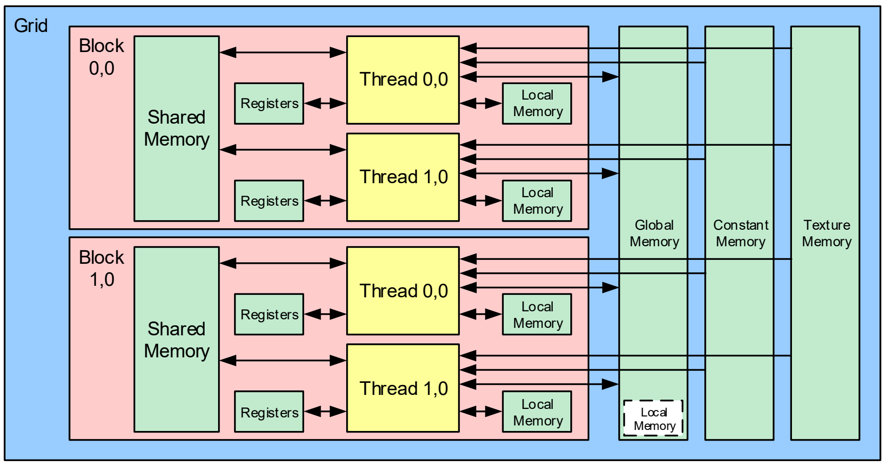


Figure 2-8 - CUDA memory model. Adapted from: [19]

In the CUDA programming model a parallel program is described in terms of thousands of lightweight threads where the programmer intentionally oversubscribes the number of physical cores in the GPU. This way the program scales well to more powerful or even future GPUs with more cores (Figure 2-9).

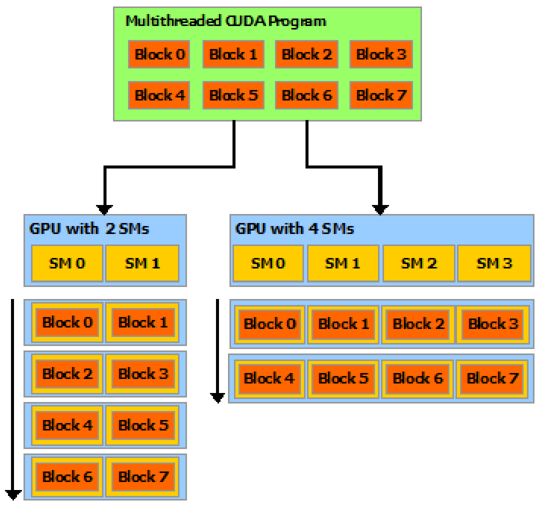


Figure 2-9 - left: Automatic Scalability. SM = Streaming Multiprocessor: Source: [19]

2.3.3.1 Asynchronous Memory Transfer

Before a kernel can carry out an image operation, pixels need to be transferred from the host's RAM into global graphics memory. Similarly, the result needs to be transferred back from device into host memory after the operation has finished. GPU operations such as memory transfers or kernel executions can be carried out asynchronously. This way, *host* (CPU) and *device* (GPU) can operate concurrently. GPUs are also capable of overlapping a memory transfer with a kernel, multiple kernels or even two memory transfers of opposite directions, provided they have two copy engines. A prerequisite for non-blocking memory transfers between host and device is that the host memory is located in non-pageable (*pinned*) memory that is guaranteed not to have been swapped out into the page file by the operating system.

2.3.3.2 Streams

Overlapping two or more kernels only takes effect when one kernel alone does not fully utilize all available GPU resources. Often, this is the case when the previous kernel is almost finished and only few thread blocks are left to be executed. Then the scheduler can start assigning the first blocks of the next kernel to multiprocessors that were otherwise idle, keeping the overall GPU load high even in between kernel launches. Causalities between operations can be modeled by scheduling operations into separate *streams*. Operations within the same stream are guaranteed to be executed in order (unless otherwise specified), but operations from separate streams can be overlapped. Inter-stream dependencies can be modeled by scheduling synchronization points into streams.

2.3.3.3 Occupancy

An important measure of how efficiently a kernel operates is its achieved occupancy. For a single multiprocessor i, it is defined as

$$Occupancy(i) = \frac{SIMD\_groups_{active}^{i}}{SIMD\_groups_{\max active}^{i}}$$

A high occupancy indicates a high instruction efficiency. The maximum number of active SIMD groups is usually computed by dividing a single multiprocessor's available resources (registers, number of threads, shared memory) by the resources required by one SIMD group since all SIMD groups have to share the resources.

Often it is advantageous to limit the resource consumption of a single thread block in order to enable the scheduler to keep two or more blocks resident in the same multiprocessor. This way, if one block is stalled, e.g. when waiting for the result of an access to slow global memory, the scheduler can hide the latency by executing another block in the meantime.

2.3.4 Data Transfer

When using the GPU as a co-processor input data has to be transferred to the GPU and output data has to be transferred back to the CPU. For the encoder, the CPU-to-GPU transfer involves more data, for the decoder the GPU-to-CPU transfer is more critical. Unless, in case of a decoder, the decoded images are directly output to a computer monitor and not transferred back to the CPU, both transfers' durations need to be included in any time measurements in order to achieve a fair comparison with CPU-based implementations.

Modern graphics cards can transfer and compute in parallel. On professional graphics cards, it is in some cases even possible to transfer in both directions in parallel. Nonetheless, computing and transferring in parallel increases the latency and when the computation kernels are heavily optimized, the parallel data transfer could become a bottleneck. The subsequent chapters throughout this work repeatedly address how a particular coding mode affects the amount of data that has to be transferred. To this end, it is important to assess whether the data transfer rate will likely remain critical in the future or whether it will become negligible due to an increasing bandwidth.

In computers, data is transferred via the *Peripheral Component Interconnect Express* (PCI Express), which is standardized by the *PCI Special Interest Group* (PCI-SIG) [30]. Figure 2-10 shows for RGB image streams with 16 bit per sample and resolutions between HD and 16K, how many images per second can be transferred when using 16 lanes. To account for protocol overhead, a netto transfer rate of 80% of the theoretically achievable rate is assumed.

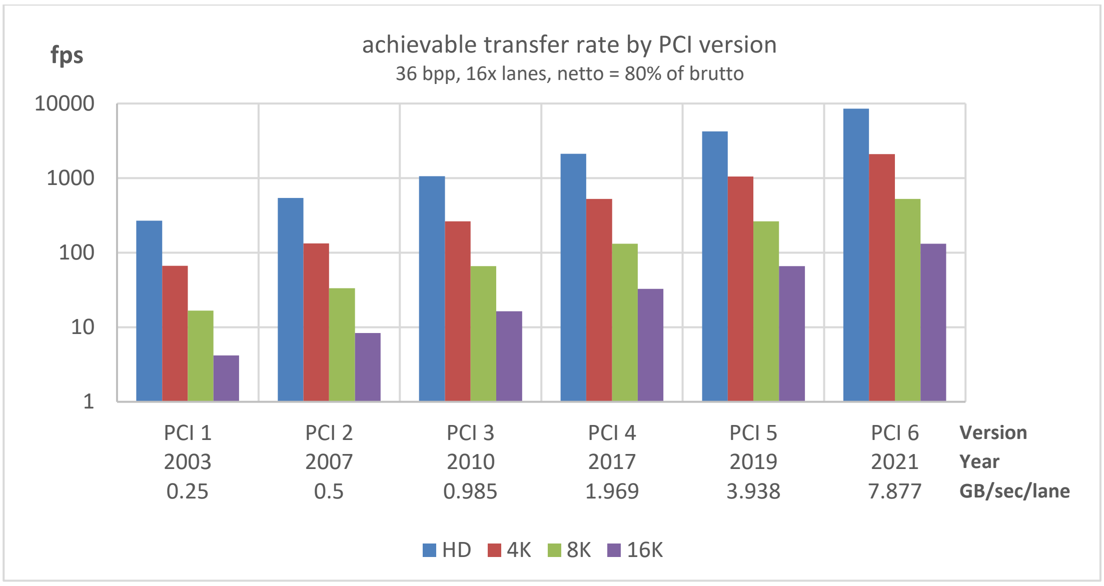


Figure 2-10 - achievable uncompressed image transfer rates over PCI express [30]

While initially it took between three and seven years for a new major PCI revision to be released (1.0 in 2003, 2.0 in 2007, 3.0 in 2010, 4.0 in 2017), beginning with version 4.0 a new major revision has been released every other year (4.0 in 2017, 5.0 in 2019, 6.0 in 2021) [30]. So far, with every major release the theoretically achievable data rate has doubled. Is the data rate bottleneck a problem of the past that will soon be solved by better hardware? Using 16 lanes, the theoretically achievable data rate has increased from 4000 MB/sec with version 1.0 in 2003 to 120.000 MB/sec with version 6.0 released in 2021. When considering a resolution of 4K, already PCI 3 achieves a transfer rate of 263 frames per second. However, when assuming a larger image resolution of 8K (8192 x 4095 pixels), PCI 4 released only in 2017 is required to achieve more than 100 fps. PCI 5 released recently in 2019 yields 263 fps. When the resolution is quadrupled again from 8K to 16K, PCI 5 achieves 66 fps and PCI 6 131 fps.

In conclusion, it appears that the achievable data transfer rate grows faster than the demand in bandwidth due to an increasing image resolution increases. However, another important aspect is to account for a significant delay of multiple years that are required for a new standard to penetrate the market and become widely available. Especially when targeting existing commodity hardware where it is not given that the latest standards are available, it is found worthwhile to consider and minimize, where possible, the amount of data transferred between CPU and GPU.

2.4 Parallel Primitives

This work proposes several application oriented algorithms that draw heavily on existing parallel primitives. Popular open source libraries that offer highly optimized implementation of such primitives are *CUDA UnBound* (CUB) [31], *Modern GPU* [32] and *thrust* [33]. A comparison of these libraries can be found at [31]. The algorithms implemented in this work utilize the CUB library. It organizes and optimizes primitives by their scopes, i.e. whether they are collectively executed

1. by a single SIMD-group (“warp-wide”)
2. by a thread block (“block-wide”) or
3. by an entire grid of threads (“device-wide”)

In the following, the scan and reduction primitives are shortly addressed, as these are the ones that are most heavily referred to in this work.

2.4.1 Scan

When the scan operation is an addition, it is also called a prefix sum. Other binary associative operations are supported by the scan algorithm, but this work uses mostly the sum operation.

inclusive prefix sum: $y[i] = \sum_{j=0}^{i} x[i]$ exclusive prefix sum: $y[i] = \sum_{j=0}^{i-1} x[i]$

One distinguishes between an *inclusive* and *external* scan. The inclusive scan adds as the current value *x[i]* to the sum, whereas an exclusive prefix sum only considers the previous elements *x[0]* through *x[i-1]*.

An extension of the continuous prefix scan is a *segmented* prefix scan. The input signal *x[i]* is accompanied by an equally sized vector of binary flags *f[i] e {0,1}*, where a 1-flag indicates that the prefix sum shall be restarted and a 0-flag indicates that it shall be continued.

Parallel implementations split the overall operation into small partial sub-tasks, e.g. in the case of a prefix sum, the addition of only two values. Recursively, the partial results are combined until the overall result is available. An excellent description of how the scan operation can be very efficiently implemented for CUDA has been composed by NVIDIA’s Mark Harris et al in the *GPU Gems 3* chapter *Parallel Prefix Sum (Scan) with CUDA* [34, 35].

2.4.2 Reduction

The goal of the data parallel reduction primitive is to reduce an array of elements to a shorter array or even a single element, based on a predefined operation. The operation is often a min-, max- or sum-operation. Another problem where a reduction is frequently used is to remove elements of a certain value (e.g. all zeros) from an input sequence.

Mark Harris of NVIDIA gave a presentation on how to best implement a reduction with CUDA in 2007 [36]. Since the shuffle instructions and fast atomic device memory access became available with the release of the NVIDIA *Kepler* architecture, it has been beneficial to implement the reduction primitive in a different way as is described in this blog post [37].

3 State of the Art

3.1 JPEG 2000

Considerable research has been devoted to re-formulating the individual JPEG 2000 coding-blocks for an execution on a GPU. Even before CUDA or OpenCL were released, Tenllado et al presented how to implement 2D DWTs efficiently on GPUs by repurposing programmable graphics shaders [18]. More recently, it was shown how to leverage CUDA's register-shuffling intrinsics in a parallel DWT implementation [38]. Whereas the default solution for exchanging data among neighboring threads is via the on-chip shared memory, the authors propose an implementation where threads from a single SIMD-group exchange registers directly using the newly introduced shuffling intrinsics.

The EBCOT algorithm poses a greater challenge as it does not inherently expose finely grained parallelism. A sample-parallel context modeling algorithm was presented in [39] and later in [40] and [41]. Whereas in the decoder, a feedback loop between the context-modeler and arithmetic decoder exists – the current decoded value is required in order to compute the new context, which is in turn required by the arithmetic decoder to decode the next value – the communication in the encoder is unidirectional. This can be utilized to decouple context modelling from arithmetic coding. The causal dependency introduced by the significance propagation is realized with an iterative loop and the voting intrinsic. Since this technique is being employed by the reference GPU-accelerated EBCOT implementation used in the work, the concept is described in more detail in chapter 4.2.2.

A CUDA-based open source encoder, titled *cuj2k*, was released by the University of Stuttgart, where color transform, DWT, quantization and EBCOT are offloaded to the GPU, but PCRD-Opt. and packetization are executed on the CPU [42]. The last update is from 2013, though. Another CUDA-based open source implementation named *GPU JPEG2K* is published by *Poznan Supercomputing and Networking Center*. The website was also last changed in 2013. It claims to be slightly faster than Stuttgart's *cuj2k* [43]. According to the published results, both encoders reach only less than 2 fps on a GeForce GTX 480 for an UHD RGB24 image. They are therefore not further regarded in the context of this work. An EBCOT variant especially tailored for GPU architectures is described in [40].

Breaking compatibility with the standard, the same paper, [40], introduces a parallelizable arithmetic coder (AC) that produces separate interleaved bit streams for each context. The entropy coder variant is compared to the reference C implementation, but unfortunately, the gain of the parallel AC over the default MQ-coder is not reported. The runtime for tier-1 of entropy coder alone for the 5 MP *bike* and *café* test images [44] is reported to be below 4 fps on a GeForce GTX 480 (vs. 4-5 secs with the C implementation).

[45] proposes the "ultrafast" mode, where EBCOT is replaced by a fixed codebook Huffman coder preceded by an optional prediction step. The work aims for faster CPU implementations and reports

speed-ups of up to 1.5x (2.9x) for the encoder (decoder). Since the Huffman coder does not operate bit plane-wise, the embeddedness property of the resulting bit streams as well as the possibility for a post-compression rate control is lost.

GPU-specific implementation optimizations for the MQ-coder were presented in [46]. Aside from research related to parallel architectures, another related field is that of alternate rate control algorithms, especially since the standard leaves some room for implementations here. The PCRD-Opt.-based rate control was first proposed by Taubman in [47] and adopted as an example into the standard. Many algorithms have the goal of estimating prior to compression, which passes will end up being truncated. An excellent overview is given by Aulí Llinás' in his PhD thesis [48]. A discussion of rate control heuristics specific to a parallel GPU-implementation can be found in [49]. Since rate-control heuristics are not discussed in the context of this work, the topic is not further addressed here. Chapter 11.1 addresses how the chosen quantization step size affects the achievable quality and impacts the quality. Similarly, a heuristic to predict a good quantization ahead of time is not addressed either.

3.2 BPC-PaCo

Developed by Aulí Llinás, Enfedeque et al at *Universitat Autonoma de Barcelona*, the *Bit Plane Coder with Parallel Coefficient processing* (BPC-PaCo) has been published between 2015 and 2017 [50–53]. It presents a drop-in replacement for JPEG 2000's EBCOT entropy coder with the objective of exposing more finely grained parallelism. Since it is so closely related to this work, a separate chapter is devoted to describing *BPC-PaCo* in more detail (chapter 8).

3.3 Low Complexity Coding

3.3.1 Professional Video Production

One predominant use case for low complexity coding is the in-house transmission in professional video production where the state of the art is uncompressed transmission over SDI. The demand for higher resolutions and higher frame rates has been addressed by improving the hardware layer. Available SDI bandwidth was increased from 1.5 Gbit/s to 3 Gbit/s, 6 Gbit/s and even 12 Gbit/s. This, however, is still not sufficient to transmit uncompressed UHDp60 without chroma subsampling as listed in the following table.

Table 3-1: uncompressed bandwidth overview. * Does not consider protocol overhead.

Resolution	Frame Rate	Subsampling	Bit Depth	Required payload bandwidth	Required Compression Ratio incl. Protocol overhead
HD	60	4:2:2	10	2.31 Gbit/s	1 GigE: 1:3-1:4 1,5 Gbit/s SDI: 1:3
HD	60	4:4:4	10	3.47 Gbit/s	1 GigE: 1:5-1:6 1,5 Gbit/s SDI: 1:3
UHD	60	4:2:2	10	9.27 Gbit/s	3 Gbit/s SDI: 1:4 6 Gbit/s SDI: 1:2
UHD	60	4:4:4	10	13.90 Gbit/s	6 Gbit/s SDI: 1:3
UHD-2 (8K)	60	4:2:2	10	37.08 Gbit/s	10 GigE: 1:5 12 Gbit/s SDI: 1:4 – 1:5

Lightweight coding is not likely going to be used for distribution over WAN or contribution, as these applications require higher compression ratios. Today, JPEG 2000 is frequently used in this context.

3.3.2 Video-over-IP

The lower maximum cable length of the high-bandwidth hardware links poses an additional downside. Instead of transmitting video over a dedicated SDI infrastructure, an alternative is to transmit it embedded in the IP protocol (video-over-IP). The main advantage of video-over-IP is that existing Ethernet infrastructure can be reused. In order to facilitate convenient transitioning between SDI and Ethernet infrastructure, initially the approach was simply to transmit entire SDI frames comprising audio, video and ancillary data. *SMPTE St 2022-6* defines how an SDI frame is encapsulated in the *Real-Time Transfer Protocol* (RTP), which operates on top of the UDP and IP layers [54, 55]. A major drawback to this approach is that the SDI protocol adds considerable overhead of up to 30% of the net data rate. Furthermore, audio and video essences cannot be routed or processed individually.

These drawbacks have been remedied by the *SMPTE St 2110* set of standards. Essences are now directly mapped to RTP and streamed separately [56]. At the time this manuscript was written in 2020, only a mapping for uncompressed images had been defined. An image compression scheme was not yet supported.

3.3.3 Requirements on Video Codec

Regardless of whether the link is SDI or IP, transmitting high resolution, high framerate video requires a light compression. In a production environment, the image quality needs to be visually lossless and in order to support multiple transitions between SDI and IP links, the codec needs to be very robust to multiple encoding-decoding cycles. The *European Broadcasting Union* (EBU) recommend that codecs employed during production should yield “quasi-transparent quality”, i.e. no visible degradation, throughout at least seven cycles [57]. Especially in live production, the latency is another important criterion. This includes the time it takes from the moment a frame was captured with the camera to the time it is displayed on the screen. For sub-frame latency codecs, the latency is measured in number of image rows. The encoder compresses the first rows and then emits them as soon as possible so that the decoder can reconstruct them in parallel. The decoder releases the first reconstructed rows even before the encoder has reached the end of the frame.

3.3.4 Codecs

This chapter briefly addresses the main representatives for lightweight image compression codecs. [58] discusses in more detail the advantages and drawbacks of the existing codecs.

SMPTE VC-2 - *SMPTE VC-2* is based on the *Dirac Pro* codec developed by the *British Broadcasting Corporation* (BBC) [59][60]. *Dirac* is a video compression scheme developed by BBC research starting from 2003 as successor for MPEG-2 and targeting HD and beyond applications. It competes with H.264/AVC. The full specifications were first released in 2008 [61]. A low-complexity, intra-frame-only version called *Dirac Pro* was standardized as *SMPTE VC-2* in 2010.

Dirac's Main Profile is a hybrid coding scheme with motion estimation and compensation utilizing long GOPs with I, P and B frames. The image is converted to the YCoCg color space and then split into overlapping blocks of variable size for motion compensation. Motion vectors can be defined per macro-block (4x4 blocks), sub-macro-block (2x2 blocks) or block with a precision up to 1/8 of a pixel and may point into at most two reference frames from the past or future. Optionally, motion vectors can be expressed relative to a global motion vector that describes a zoom, tilt or pan. The residual error signal is then wavelet-transformed with one of seven wavelet filters, including CDF 9/7, LeGall 5/3 or Haar-Transform. Sub-bands are then quantized and optionally split into code blocks of arbitrary size. Code-blocks (or in their absence entire sub-bands) and motion vector data are then entropy-coded with a context-adaptive binary arithmetic coder. Code-blocks are traversed with only one pass per bit plane and quantization indices are assigned to one of 17 contexts, depending on their state and neighborhood.

Four profiles are defined: *Main* (long GOP), *Main* (Intra, AC), *Simple* (Intra, no AC) and *Low Delay*. Only the latter three are available in *SMPTE VC-2*.

In contrast to *Dirac Core, Dirac Pro* is an intra-frame only scheme. It is constrained to a 10 bit 4:2:2 pixel format. Coefficients in the LL0 (aka DC) subband are predicted based on three neighbors (W, NW, N). The arithmetic coder is replaced by a computationally less expensive *Exp.-Golomb* coder in the *Low Delay* profile. Spatially corresponding code blocks from all sub-bands are grouped into slices and ordered as such in the output stream, enabling the decoder to independently reconstruct the pixel domain area represented by a single slice. Two applications are *Dirac Pro 1.5*, which achieves a 5:2 compression of full HDTV (1080p50/60) for transport over (1.5 Gbit/s) HD-SDI links at a delay of only six HD lines

(0.25 msecs) and *Dirac Pro 270* for the compression of 1080i50/60 signals for transport over SD-SDI links (270 Mbit/s) [60].

Apple ProRes / SMPTE RDD 36 – The *Apple ProRes* collection of codecs supports both chroma-subsampled and non-subsampled formats with and without an alpha plane (4:2:2, 4:4:4, 4:2:2:4, 4:4:4:4) and bit depths of 8, 10 or 12 bits per sample [62, 63]. Similarly to legacy JPEG and MPEG-1 or MPEG-2, the image is converted into the YCbCr color space and divided into macro-blocks (16x16 pixels) and then further into blocks (8x8 samples) that are each DCT-transformed and quantized. For 4:2:2 images, a macro-block will contain four blocks from the luminance channel (Y) and two blocks from each of the chrominance channels. Up to eight horizontally adjacent macro-blocks are in turn grouped into slices that are each independently entropy-coded. For DC coefficients, the prediction residual is coded. AC-coefficients are run-length-coded in a zig-zag scan order. The resulting symbols are coded in a combination of *Exp.-Golomb* and *Golomb-Rice* (small values with *Golomb-Rice*-code, large values with *Exp.-Golomb* code) [62].

ProRes is a variable-bit rate-coder. According to [63] the variability is small and a maximum bit rate exists that is never more than 10% above the target bit rate. The decoding speed for a 1080p 422 HQ (*4444 XQ,* no alpha) frame on a 12-core *Mac Pro* is stated to be under 2 ms (6 ms). For a 1080p24 sequence the target data rates lie between 36 Mbit/s for *422 Proxy* and 396 Mbit/s for *4444 XQ* (no alpha). For a 2160p24 (UHD) sequence, the date rate is almost exactly four times as high. The similarity to *Avid's DNxHD* family of codecs (standardized as SMPTE VC-3) is high [64].

Intopix TICO / SMPTE RDD 35 – TICO stands for *Tiny Codec* and was selected by the JPEG to become the basis for the new JPEG XS codec [65]. Images are first wavelet-transformed with an integer-based *LeGall 5/3* wavelet. Coefficients are then quantized and grouped into sets of four samples each. A simple entropy coder merely computes the highest significant bit plane index (so-called greatest coding line index – GCLI) for each group. Since neighboring GCLIs tend to be similar, their residual is computed and run-length-coded. The sign- and magnitude bits between the GCLI bit plane and the truncation bit plane (selected by the quantizer) are directly appended to the bit stream.

3.4 Standardization

3.4.1 JPEG XS

The JPEG XS call for proposal was released in March of 2016. JPEG XS is a separate family of specifications that currently comprises five parts. The goal was to devise a codec with very low latency and complexity that is well suited for operation on professional video links such as SDI or video-over-IP. The compression rate it was designed for is between 1:2 and 1:6.

The call for proposals was answered by various parties. [66] gives an overview on the contributions. In the end, *Intopix s.a.*, creators of the TICO (*Tiny Codec*) lightweight codec, and *Fraunhofer IIS* were tasked to develop the new standard.

The submission named *Fast Block Coding with Optimized Truncation* (FBCOT) by David Taubman of *University of New South Wales* was found to be too complex still [12]. It went on to become the basis of the subsequent *JPEG 2000 High Throughput* undertaking.

The core coding system standard has been published in May of 2019 as " ISO/IEC 21122-1:2019" [67]

3.4.2 JPEG 2000 High Throughput

In summer of 2017 the JPEG 2000 standardization committee released a call for proposals for a high-throughput drop-in replacement for EBCOT [68]. It was published not as a new standard, but as part 15 of the JPEG 2000 standards family [69]. The so-called *High-Throughput JPEG 2000* (HTJ2K) activity aims to increase the throughput at minimal cost in compression efficiency, while keeping compatibility with the JPEG 2000 framework as high as possible.

4 JPEG 2000

4.1 Description of the Encoder

Compression scheme specifications typically define the code stream format and how it shall be interpreted by a decoder in order to reconstruct the original data. This leaves vendors some degree of freedom in how they implement the encoder. However, this chapter describes the JPEG 2000 specifications from the encoder's point of view for better comprehensibility (Figure 4-1).

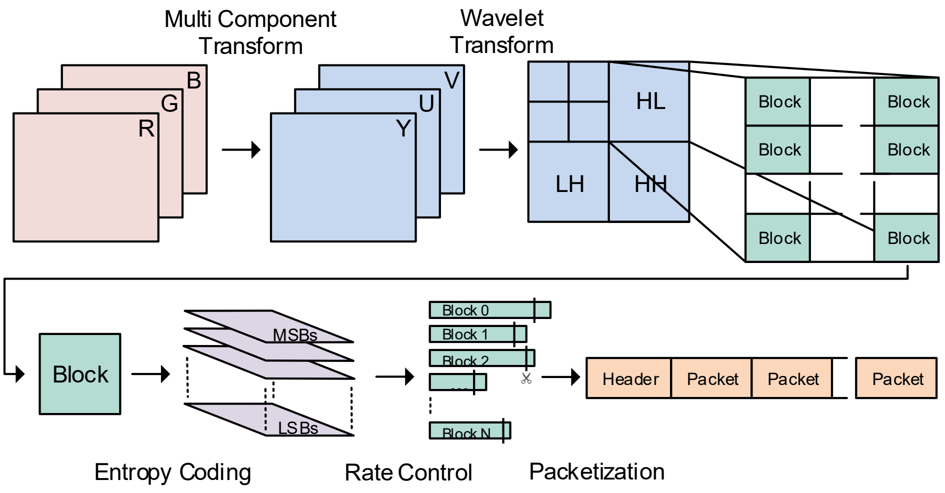


Figure 4-1 – JPEG 2000 encoder block overview

JPEG 2000 is an intra-frame compression codec. For a single image, it first seeks to compact the energy of the image into as few samples as possible. To do that, it first decorrelates luminosity ("brightness") from color information. It then processes each channel individually with an energy compaction transform, in the case of JPEG 2000 a Wavelet transform. In the lossy case, it will then quantize the coefficients. The quantization step size is typically lower for luminance coefficients than for chrominance coefficients. Likewise, it is lower for low-frequency signals and grows towards higher frequencies. The encoder will then split each wavelet subband into non-overlapping uniformly sized blocks, e.g. of size 32x32, that are each independently entropy-coded into a bit stream. The rate controller then truncates the bit streams, if necessary, to meet the maximum bit rate constraint. Lastly, the bit streams are organized into a code stream according to a predefined progression order, e.g. by quality, by resolution, or by component. The following chapters will elaborate on the individual coding steps.

4.1.1 Multi-Component Transform

The *Multiple Component Transform* (MCT) provides a generic way of transforming the input image channels into a representation better suited for compression. For the case where the inputs image comprises a red, green and blue channel, JPEG 2000 part-1 contains two predefined transforms that separate luminosity from color. The human visual system is more perceptive to changes in luminance than in chrominance and this can be exploited by compressing the latter more heavily. In lossy mode, the *Irreversible Color Transform* (ICT) is the standard point-wise linear RGB-to-YC_bCr color transform. The *Reversible Component Transform* (RCT), on the other hand, is defined as a non-linear relation between original and targeted color space and is implemented using integer arithmetic to prevent rounding errors.

4.1.2 Wavelet Transform

In JPEG 2000 part-1, bi-orthogonal wavelets are used, specifically the real *Cohen-Daubechies-Feauveau 9-tap/7-tap kernel* (CDF 9/7) for the irreversible path and the integer *LeGall 5-tap/3-tap kernel* (LeGall 5/3) for the reversible path. Part-2 allows for employing other wavelet filters as well.

The number of wavelet levels depends on the image resolution. The standard demands only that the lowest subband (LL0) must contain at least 128 samples in each dimension. In the profiles relevant for professional media production, five decompositions yielding six embedded resolution levels are demanded for HD or 2K images, which yields a LL0 size of approximately 256x256 samples. According to JPEG 2000 part-1, the number of horizontal and vertical decompositions must be equal.

4.1.3 Quantization

In the irreversible path, a *deadzone scalar quantization* is employed. In this type of quantization, the central interval around zero, referred to as the deadzone, is twice as wide as all other intervals. Step sizes may be defined per subband. They are explicitly communicated to the decoder in the code stream as subband attributes. In lossless mode, wavelet coefficients are not quantized. For further details regarding the theory on quantization and JPEG 2000 specifics, the reader is referred to [70].

4.1.4 Block Coding

Each color channel's wavelet subbands are then divided into non-overlapping code blocks that are independently encoded. The dimensions are defined in the DCI-, IMF- and broadcast-profiles to be 32x32. For a 2K image with three color channels, this yields roughly 5000-6000 blocks. EBCOT encodes each of these blocks into an embedded bit stream. JPEG 2000 employs a post-compression rate-control. That means it first attempts to compress an image as well as possible, given a user-defined quantization step size. If the accumulated length of all code blocks' bit streams does not exceed the desired target, no further work is required. If it has been exceeded, however, some or all of the bit streams need to be truncated. The goal is to do this in a fashion that the error introduced by an omission of bits is minimized. In order to solve this optimization problem, the block coder computes for some points in a single bit stream how much the distortion would be affected by including the additional bits since the previous point. Since it is only permitted to truncate the bit stream at any of these points, they are referred to as *truncation points*. This way, a rate-distortion plot for each code block can be devised, which is then used to determine the most appropriate cut-off point. Without going into details at this point, it becomes visible that it is beneficial to have as many potential truncation points as possible. On the other hand, increasing the number of truncation points will also raise the amount of required computations. In order to yield good quality even at very low bit rates, EBCOT not only adds truncation points after each bit plane, but additionally allows for fractional bit planes. In the original EBCOT publication, each bit of a bit plane is coded in one of four passes based on how likely it is that the inclusion of this bit will decrease the distortion [47]. The passes are executed one after the other, starting with the one that is estimated to improve the quality the most. In the course of the standardization, however, the number of passes was changed from four to three in order to reduce the computational costs.

EBCOT consists of two tiers. In tier-1, blocks are processed bit plane by bit plane, starting with the most significant magnitude bit plane. All bits are processed in a predefined scan order, passing over each plane three times. Each bit, together with a context describing the local neighborhood, is fed to a context-adaptive binary arithmetic coder, called MQ-coder. Successively, the MQ-coder produces a bit stream. With the goal of meeting a user- or profile-defined global bit rate constraint, in tier-2 the set of bit streams, one per code block, will be truncated, if necessary. If multiple quality layers are used, tier-2 will additionally decide when to start allocating bits to the next layer.

4.1.4.1 Tier-1 – Context Modeling and Entropy Coding

EBCOT first converts the quantization indices into a sign-magnitude representation. The most significant bit plane now represents the signs, with 1-bits indicating a negative sign and 0-bits indicating a positive sign. The remaining bit planes present the samples' magnitudes as visualized in Figure 4-2.

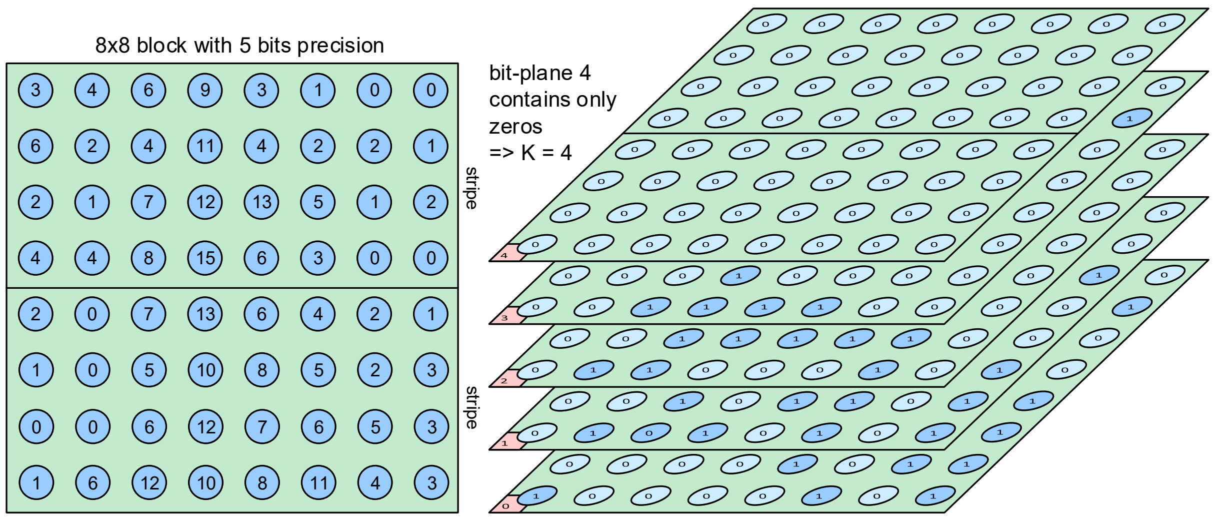


Figure 4-2 – bit plane view on 8x8 code block. Magnitudes are stated with five bits precision, the sixth plane with the sign bits is not depicted.

Each bit plane is divided into stripes with four rows each. The scan order is

1. by bit plane from MSB to LSB (skip initial MSBs that only contain 0 bits)

2. three passes per bit plane
3. in each pass by stripe from top to bottom
4. by stripe column from left to right
5. by sample from top to bottom

Figure 4-3 shows the scan order within one stripe. Initial magnitude bit planes that consist entirely of 0-bits are skipped. Instead, the number of insignificant MSB bit planes is signaled to the decoder by coding it as part of the code block header in the code stream.

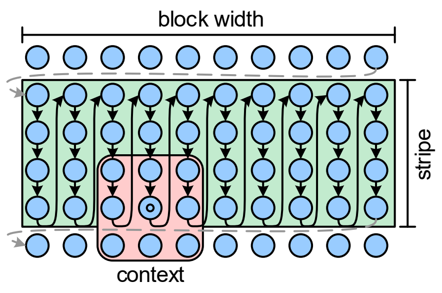


Figure 4-3 – block coding scan order and context modeling

In a given bit plane, each bit is coded in one of the three passes. The bit value *d* together with a context *cx e{0-17}* is fed into the MQ-coder. The arithmetic coder maintains separate probabilities for each of the 18 contexts. Depending on whether the incoming symbol *d* equals the most probable symbol (MPS) or not, the MQ-coder will update the current context's MPS probability according to a predefined table with 46 static transitions.

In order to be able to understand how the contexts are formed, the three pass types and four coding primitives need to be introduced. Each bit of a bit plane is coded in exactly one of these three passes using one of the four coding primitives listed in Table 4-1.

Table 4-1 - EBCOT Coding Primitives

Acronym	Name	Contexts	Meaning
ZC	Zero coding	9, computed from significance of neighbors	Code current magnitude bit, while sample is still insignificant
SC	Sign coding	5, computed based on signs of neighbors	Code sign bit
MRC	Magnitude refinement coding	3, based on first refinement or not and whether neighborhood is significant	Code current magnitude bit, after sample has already become significant
RLC	Run length coding	1: “uniform probability”	Code length of zero run within one stripe column. Used when entire stripe column neighborhood is insignificant

The membership for each pass is computed based on a context that is maintained per sample:

Table 4-2 - EBCOT state variables

Variable	Scope	Meaning
σ	across all bit planes	Sample is already significant, it is non-zero in decoder's eyes because the first *d=1* bit has been emitted
σ´	across all bit planes	Magnitude refinement coding has been applied
η	Current bit plane only	Current bit has been coded in SPP

An overview of the three pass types can now be given in terms of the employed coding primitives and context variables:

Table 4-3 - EBCOT pass types

Acronym	Name	Eligible primitives	Membership Condition	Meaning
SPP	*Significance Propagation Pass*	ZC, SC	$\sigma[x,y]=0 \wedge \sigma[x\pm1,y\pm1] \neq 0$	"Sample still insignificant, but at least one neighbor significant"
MRP	*Magnitude Refinement Pass*	MRC	$\sigma[x,y]=1 \wedge \eta[x,y]=0$	"Sample already significant, but did not only just turn significant in this bit plane's SPP"
CUP	*Clean-Up Pass*	ZC, SC, RLC	$\sigma[x,y]=0 \wedge \eta[x,y]=0$	"Sample not yet significant and has not been coded in this bit plane's SPP"

Significance Propagation Pass

The significance propagation pass seeks to code all those samples that are still insignificant, i.e. their value is zero in the decoder's eyes, but that are likely to have a value of *d=1*. This assumption is made when at least one of the samples from the 3x3 neighborhood is already significant, i.e. a *d=1* value has already been emitted to the bit stream. For those samples in the scan order past (W, NW, N, NE) the 1-bit could have been emitted in the current bit plane or earlier bit planes, for samples from the scan order future it must have been emitted in an earlier bit plane. In case *d* is indeed 1, the sign bit is interleaved immediately following the magnitude bit. It is clear why the set of bits that will turn the currently zero-valued sample (in the eyes of the decoder) into a non-zero value is considered very important and thus included before any magnitude refinement bits. A single 1-bit in a high bit plane would reduce the difference between the decoder's currently assumed value (0) and the true value immensely and thus reduce the error significantly. At the same time, this information is only valid and useful if the sign is known. By interleaving it right away, a situation is prevented where the bit stream is truncated between the most significant magnitude bit and the sign bit.

Magnitude Refinement Pass

This pass selects all those samples that have already turned significant in a previous bit plane. Even before this pass, the decoder already knew that the sample has a non-zero magnitude. The additional information from this pass merely *refines* the magnitude by informing the decoder whether this less significant magnitude bit has a true bit value of zero or one. To give an example, a true value 19 (0b10011) has already turned significant when the fifth bit plane had been coded. In the significance pass of the fifth bit plane a 1-bit had been “sent” from the encoder to the decoder, i.e. in the decoder’s eyes the value is now 0b10000 (16). In fact, since the decoder knows that the least significant four bit planes have not been coded yet this value could eventually be reconstructed to any value between 0b11111 (31) and 0b10000 (16). Instead of assuming zeros for all remaining bit planes, it will implicitly set the next lower bit in order to “guess” a value in the middle of this range: 0b11000 (24). When the fourth bit plane is coded, this value is coded in the magnitude refinement pass, since it has already turned significant (first 1-bit send) in the fifth plane. The value 19 contains a 0-bit at the fourth magnitude bit plane, and so now the decoder will correct the reconstructed value from 0b11000 (24) to 0b10100 (20). In this example, by coding the additional magnitude bit, the error between the reconstructed value (24) and the true value (19) has been reduced from 5 to 1.

Clean-Up Pass

All remaining bits of the current plane that have not been coded in the previous two passes will be coded in this pass. Namely, samples that are still insignificant (zero in the decoder’s eyes) and have a neighborhood where all samples are also still insignificant. If this is the case for all four samples in the stripe column, the run-length coding primitive will be used. A zero-run is limited to a single stripe column, i.e. a length of four. It codes one bit that indicates whether the run is interrupted or not. In case it is interrupted, it codes two more bits that together state the length of the zero-run. The remaining bits after the interrupted run are then regularly coded with the zero-coding primitive, optionally followed by the sign-coding primitive, if necessary.

The first magnitude bit plane is not separated into three passes. Instead, only the Clean-up-pass is carried out. Figure 4-4 visualizes for a high-entropy image, grouped by resolution level, when samples turn significant, respectively their sign bit is coded.

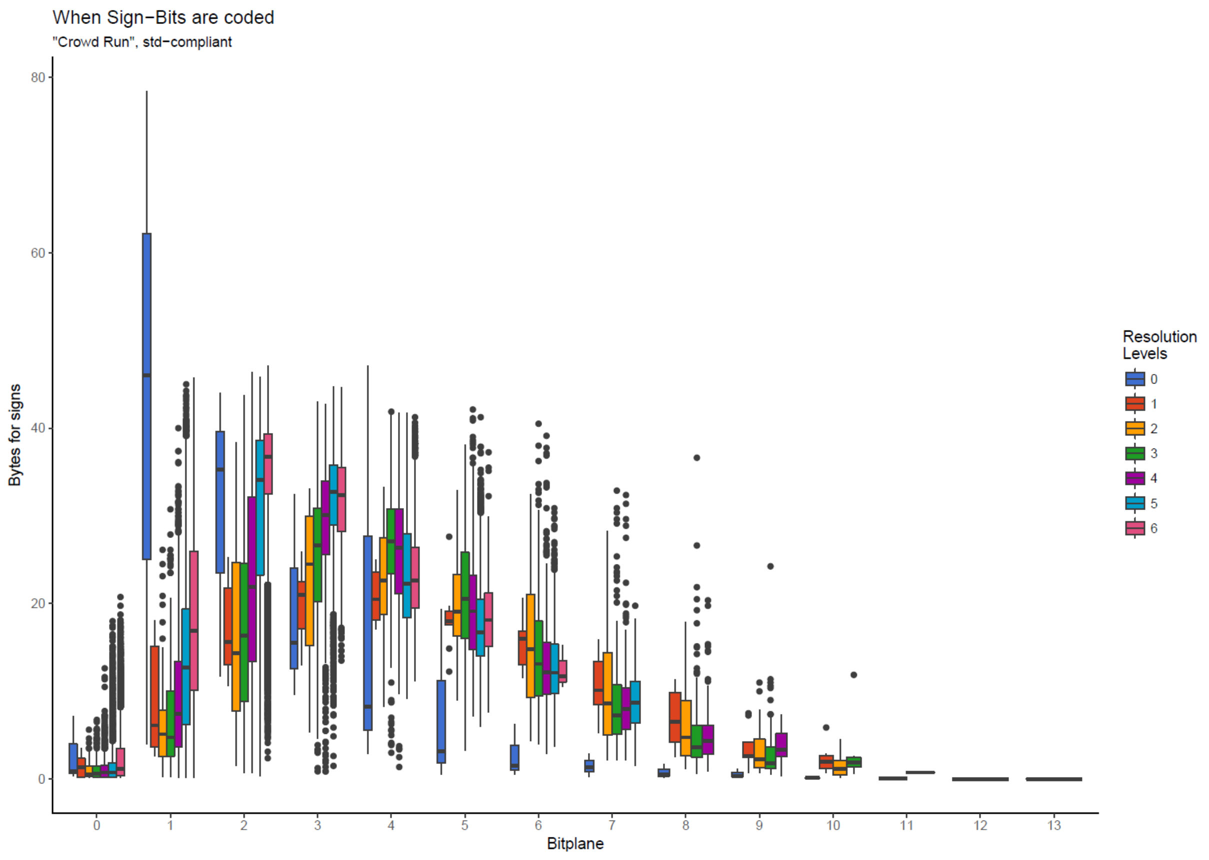


Figure 4-4 - bytes for sign bits over bit planes grouped by resolution level for standard-compliant JPEG 2000 coding

4.1.4.2 Tier-2 – Rate Control and Packetization

Rate Control

This paragraph stems from the author's published work [71].

Tier-2 is responsible for truncating the code blocks' bit streams in order to comply with a user- or profile-defined target bit rate constraint. If the accumulated size of all bit streams exceeds the available bit rate budget, the goal is to truncate the set of bit streams in a way that minimizes the error introduced by the truncations. The standard proposes the *Post-Compression-Rate Distortion-Optimization* (PCRD-Opt.) algorithm. It relies on the block encoder (tier-1) to provide as side-information an estimate of each truncation point's slope on a rate-distortion plot (Figure 4-5). It is then an optimization problem to find the set of truncation points n_j - one for every block's bit stream j - that yields the lowest overall distortion while still staying within the available data rate budget. The algorithm is explained in detail in the standard [2] and the original publication [47]. To give a brief overview here, it can be shown that only the truncation points that lie on the convex hull of their block's rate distortion plot are viable candidates and so all potential points for a single code block have strictly decreasing slopes. It follows that a set of truncation points n_j^λ can be constructed by defining a slope threshold λ and then selecting, for each code block, the truncation point that has a slope closest to, but not below this threshold. A bisection search will gradually lead to the best available slope threshold λ. It starts by probing a threshold half way between the minimum and maximum possible slopes.

The bit stream lengths and number of included passes for each code block can be computed based on the current set of truncation points. Then the packet headers, tile-part headers and main headers are simulated and their lengths summed up to give the total code stream size. As long as this computed size exceeds (or is too far below) the specified budget, the procedure is repeated with a lower (higher) slope threshold.

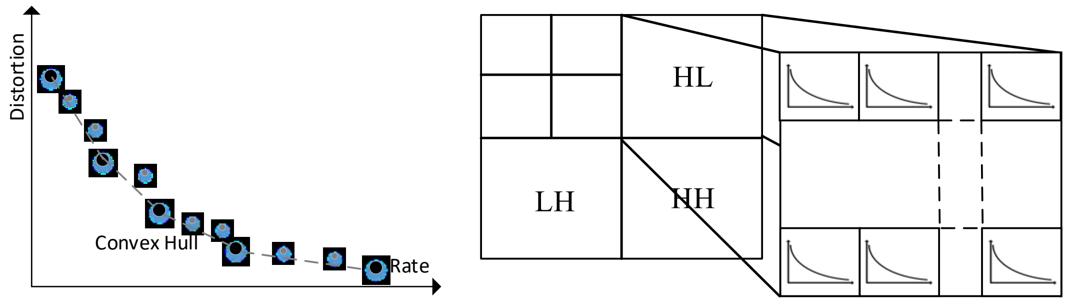


Figure 4-5 – left: Rate-distortion (RD) plot for a single code block. Right: RD-plots for all code blocks

Packetization

Structurally, a JPEG 2000 code stream consists of a set of markers that can be categorized into a main header and tile parts. A tile part is a sub-portion of the compressed image and its content depends on the progression order. The stream is organized according to four dimensions of scalability:

- **Layer**. Progression by layer causes a bit stream to be progressive by quality. All data of layer *l* packets appears before any layer *l+1* packet.
- **Resolution**. Progression by resolution guarantees that resolution level *r* is brought to full quality before any packets from level *r+1* are included.
- **Component**. Progression by component defines that all packets from component *c* appear in the code stream before any packets containing data from component *c+1*.
- **Position**. Progression by position can be regarded as a spatial progression. Each tile's packet will be ordered according to the tile's spatial position from top to bottom.

A quality progressive code stream similar to legacy JPEG can be achieved by grouping code block contributions in a predefined number of layers, where each layer contains additional information and thus further enhances the quality. The user can either manually specify bit rates for each layer or simply select the number of layers, leaving it up to the encoder to determine an optimal distribution of bit rates. Neither the cinema profiles, nor the broadcast or IMF profiles permit the use of quality layers.

Multiple of these dimensions can be supported simultaneously. In the same manner, a so-called transcoder application is able to perform various operations, like spatial transformations, change of progression order or resizing or adjustment of bit rate, without having to decode and re-encode the compressed image, by simply re-ordering, dropping or truncating selected packets.

When the major progression order is by color channel, each tile-part corresponds to a color channel. The main header comprises global properties and coding options required to decode and interpret the image. Tile-part headers contain further properties that can vary by tile-part, such as the bit depth. They are followed by the tile-part body, which is comprised of a set of packets. Each packet contains the compressed code blocks from one particular precinct in one quality layer. Since the use of quality layers is prohibited in the DCI, broadcast and IMF profiles, we assume there is only a single quality layer going forward. Each packet header contains the code blocks' number of skipped all-zero bit planes, pass count and segment lengths. Since these values are likely to be similar for the code blocks in one precinct, they are compressed with tag tree and comma codes. For improved random access, the tile-parts' lengths can be stated in the main header[2]. This is mandatory in the DCI, broadcast and IMF profiles.

4.2 Exploiting Parallelism in JPEG 2000

A parallelization of the color transform is straight forward as each pixel can be transformed independently. [38] describes how the DWT can be implemented by utilizing the CUDA register shuffling operations in order to avoid having to store intermediate results in shared memory. This chapter will instead focus on the entropy coder and discuss if and how the block coder can be executed with finely grained concurrency.

4.2.1 Decoupled Context Modeling and Arithmetic Coding in EBCOT Encoder

The block coder is by far the most time-consuming coding block of the JPEG 2000 coder [72][34]. In terms of being able to parallelize the block coder, a key difference between the encoder and decoder needs to be pointed out. On the encoder side, communication between the context modeler and arithmetic coder is uni-directional (Figure 4-6). The context modeler pushes context-decision pairs into the arithmetic encoder and modeling of subsequent pairs is not dependent upon any return value of the arithmetic coder. In the decoder, however, the communication is bi-directional. The context modeler pushes the next context in, which the arithmetic decoder requires to be able to decode the next decision bit from the bit stream.

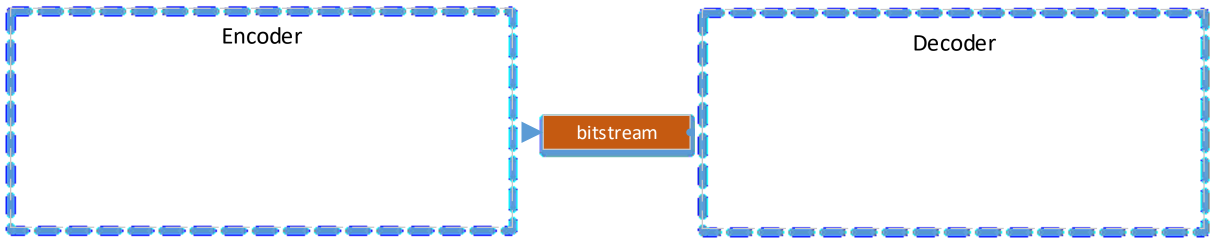


Figure 4-6 - Communication between context modeler and MQ-coder

It follows, that context modeling and arithmetic coding can be decoupled in the encoder, but not in the decoder. This makes it possible to overlap context modeling for bit plane *n+1* with MQ-coding of the

[2]in the *Tile-Part Length in Main Header* (TLM)-marker

context-decision pairs of bit plane *n*. For this reason, context modeling (CM) and MQ-coding (MQ) are implemented in separate kernels that each operate on a single bit plane (Figure 4-7). The loop over all bit planes is executed on the host [49].

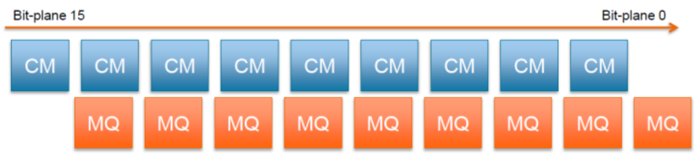


Figure 4-7 - overlapped context modeling (CM) and MQ-coder kernels in EBCOT encoder. Diagram copied without changes from [49]

4.2.2 Sample-parallel EBCOT encoding

Since the per-context probabilities are never reset in the arithmetic coder and significance states can propagate throughout a code block, a causal dependency chain exists between the very first bit coded in the most significant bit plane's first pass and the last bit coded in the least-significant plane's last pass. However, the uniform communication between context modeler and MQ-encoder can be exploited to reformulate the three pass executions within a single bit plane as a concurrent algorithm [39, 40]. Technically, it is possible to assign one thread per sample, but for implementation reasons it is advantageous to assign one thread per stripe column. This way the state variables of four samples can be coded into a single 32-bit register and additionally the run-length coding primitive is easier to realize. The worst-case amount of symbols per stripe column is stated in the following table.

Table 4-4 - Worst case number of symbols per stripe column

Pass	Symbols	Explanation
SPP	4 x (ZC+SC) = 8	Zero-coding and sign-coding for each sample
MRP	4 x MRC = 4	MRC for each of the four samples
CUP	RLC + SC + 3 x (ZC+SC) = 10	Run length-coding 1 01 signals that run is interrupted after 1 bit. It follows sign-coding for the first sample and then ZC+SC for all the remaining samples.

In order to carry out context modeling in parallel, two new state variables, ρ and τ, are introduced [39].

Table 4-5 – sample-parallel EBCOT state variables

Variable	Scope	Meaning
ρ	across all bit planes	Sample became significant in any previous bit plane
τ	current bit plane only	Sample will become significant during current bit plane's SPP

In a per-bit plane initialization routine, all threads concurrently determine their values for ρ and τ. It is straightforward to compute ρ by simply checking if any previous magnitude bit for the sample is 1.

τ is computed iteratively. The current sample will be coded during the current SPP, if it lies in a preferred neighborhood, i.e. any neighbor has already become significant in the scan order past. It will additionally become significant if the current magnitude bit is 1. Since the significance is propagated along the scan order, this routine is repeated until all threads report that their value for τ has not changed anymore compared to the previous iteration. This can be implemented with CUDA's thread voting instructions, specifically the *any* command.

Once every thread has determined their four samples' ρ and τ states, they can carry out the four coding primitives according to the following membership conditions.

Table 4-6 – sample parallel EBCOT coding primitives membership conditions for bit plane n

Acronym	Membership Condition	Pass
ZC	$\rho^{n+1} = 0$ (Previous magnitude bits all 0)	$\tau = \begin{cases} 1 \rightarrow SPP \\ 0 \rightarrow CUP \end{cases}$
SC	$\rho^{n+1} = 0 \wedge d=1$ (Current magnitude bit is the first 1-bit)	$\tau = \begin{cases} 1 \rightarrow SPP \\ 0 \rightarrow CUP \end{cases}$
MRC	$\rho^{n+1} = 1$ (any previous magnitude bit is 1)	MRP
RLC	$\rho^{n+1} = 0 \ \wedge \rho[\pm 1, \pm 1]^{n+1} = 0$ for all samples in stripe (all samples in stripe column and their neighbors insignificant)	CUP

The context-decision pairs then have to be sorted by pass type and stored to global memory (Figure 4-8) so that the subsequent MQ-encoding kernel can read and arithmetically encode them in the correct order (Figure 4-9).

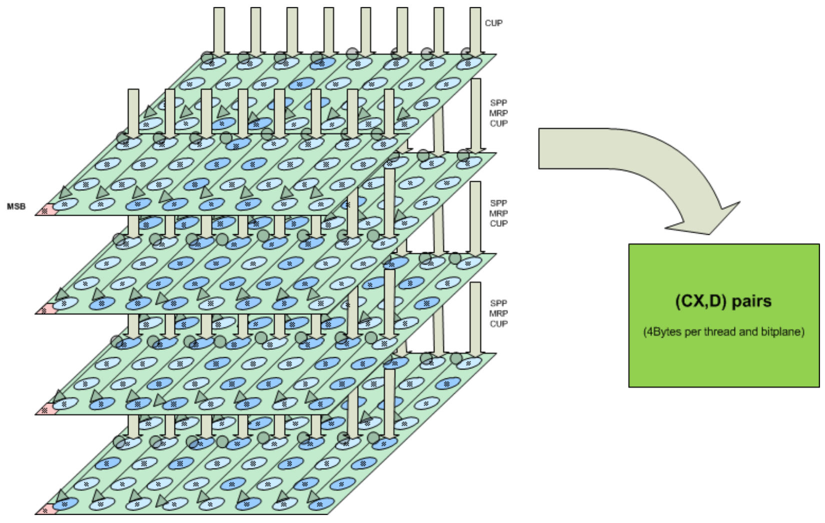


Figure 4-8 - Design of the CM kernel (note that code blocks have been simplified to 8x8 here)

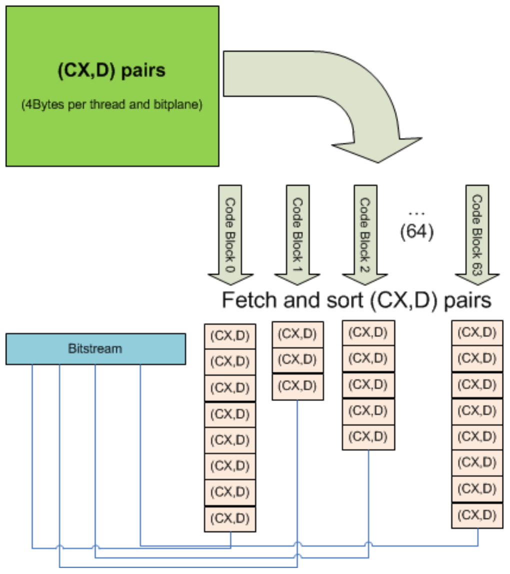


Figure 4-9 - Design of the MQ kernel

4.2.3 Conclusion

A theoretical analysis of the degree of parallelism exposed by the algorithm provides the basis for selecting a suitable kernel design and threading layout. Two key observations lead to entirely different designs in the encoder and decoder as listed in Table 4-7.

Table 4-7 - impact of degree of parallelism exposed by algorithm on implementation design

observation	exploitation
In the encoder, communication between the context modeler and arithmetic coder is uni-directional, while in the decoder it is a bi-directional feedback loop.	By decoupling context modeling and arithmetic coding into separate kernels, both kernels can run in parallel, increasing the GPU utilization
The encoder has full knowledge of all sample values ahead of time. In contrast, the decoder knows only the portion of the magnitudes that has already been decoded.	Only in the encoder, context modelling can be realized in a sample-parallel fashion by carrying out the significance propagation using collective thread voting primitives in a pre-processing step.

Due to the points listed in the table above, entropy coding in the encoder is realized in two separate kernels that are each spawned once per bit plane. The loop that iterates over all bit planes and schedules which code blocks still have pending work must be carried out on the host. In contrast, both for context modeling and arithmetic coding a code block is the smallest work unit on the decoder side and cannot be further broken down. Consequently, entropy coding can be combined into a single kernel with one thread per code block. Here, the loop over each bit plane takes place in the GPU kernel.

The GPGPU programming paradigm is designed for very lightweight threads. Having to assign an entire code block to a single thread violates the GPGPU design concept and prevents a good GPU utilization and in turn optimal throughput. This motivates to investigate possibilities to change the entropy coder design with the goal of mitigating this drawback by exposing a higher degree of parallelism.

5 JPEG 2000 Compliant Extensions

5.1 Motivation: Poor compression efficiency in lower bit planes

The discussion in this chapter is motivated by the observation that DCI-compliant code streams of 2K 24 fps DCPs very frequently do not exceed the maximum bit rate of 250 Mbit/s. Consequently, even the lower bit planes are processed with three passes each, which takes a lot of effort. The question is whether the assumptions made by the context modeler and arithmetic coder on the probability distribution in the source signal still hold even at the lowest bit planes.

The following Figure 5-1 visualizes the results of an experiment run on an image from the DCI *Standard Evaluation Material* (StEM) [73] that contains relatively many details. The stacks show for each resolution level of the luminance channels, the mean compression rate separated by pass type and bit plane.

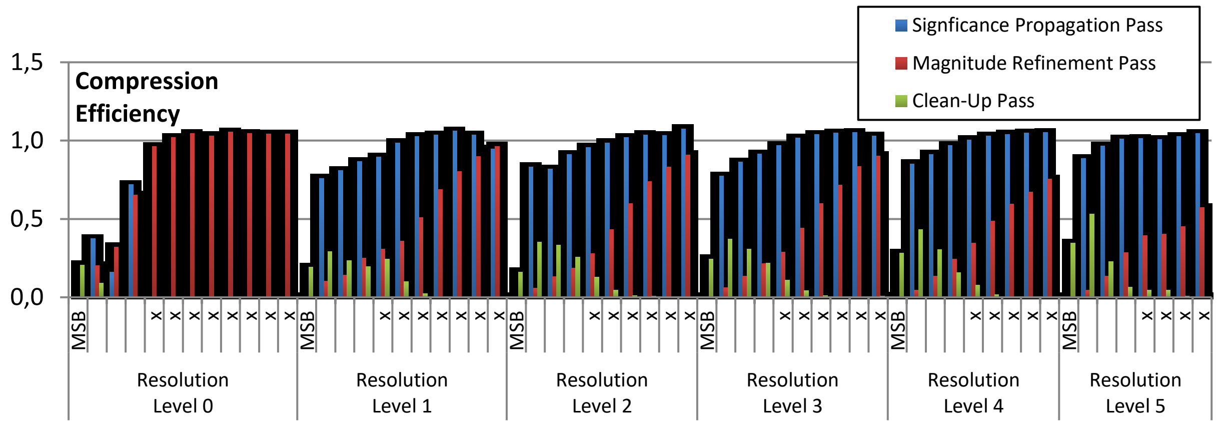


Figure 5-1 - Compression efficiency for a frame from the DCI StEM sequence ("MM_2K_XYZ_01571", Y channel, DCI 2K profile). Values below 1 indicate compression, values above 1 expansion. If fast mode starting after fourth bit plane were enabled, symbols from SPP and MRP in bit planes marked with an "x" would bypass the MQ-encoder, reaching a value of 1.

The compression efficiency of any given code-pass can be defined as the inverse ratio of the number of symbols that are fed into the MQ-encoder to the number of bits that are emitted by the coder. Ratios smaller than one indicate that the MQ-encoder has compressed the signal while ratios larger than one indicate an expansion.

It is evident that across all resolution levels, the compression efficiency is highest in the most significant bit planes and decreases towards the LSBs. In some occasions, the bits coded in magnitude refinement and significance propagation passes are even expanded, not compressed. The reason is that the Laplacian distribution assumed by the coder is not actually present anymore, but instead the signal approaches a uniform distribution [74].

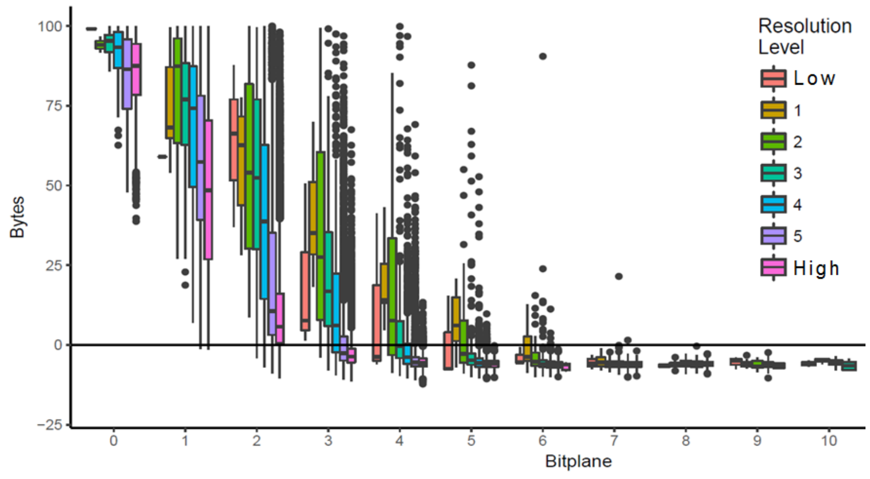


Figure 5-2- Bytes saved by entropy-coding for (*Crowd Run* sequence)

A second example (Figure 5-2) visualizes for the *Crowd Run* sequence, how many bytes are saved by entropy coding, grouped by resolution level and bit plane. Starting from the sixth bit plane, the majority of passes get expanded rather than compressed. Moreover, it is visible that in the three most significant bit planes, the lower frequencies, i.e. the subbands in the lowest (red bars) and second lowest (brownish bars) and third lowest (green bars) resolution levels, compress better than the higher frequencies.

5.2 Selective Arithmetic Coding Bypass style

This paragraph stems from the author's published work [75].

To address the issue laid out in the previous chapter, JPEG 2000 defines the *Selective Arithmetic Coding Bypass* style, sometimes also denoted simply as *fast mode* in the literature. When enabled, symbols from the significance propagation and magnitude refinement passes starting from a code block's fifth significant bit plane are directly appended to the bit stream, bypassing the MQ-encoder. Symbols from clean-up passes must still be fed into the MQ-encoder as usual, as they typically compress well. A bit stream needs to be terminated whenever switching between non-bypassed and bypassed mode so that in the end it consists of multiple *segments*. This coding style is signaled globally for either the entire image or image component – it cannot be set on a code-block-by-code-block basis. Part 2 of the standard enhances the mode by enabling the user to select whether the arithmetic coder in the significance propagation and magnitude refinement passes should already be bypassed as early as in the second, third or fourth bit plane.

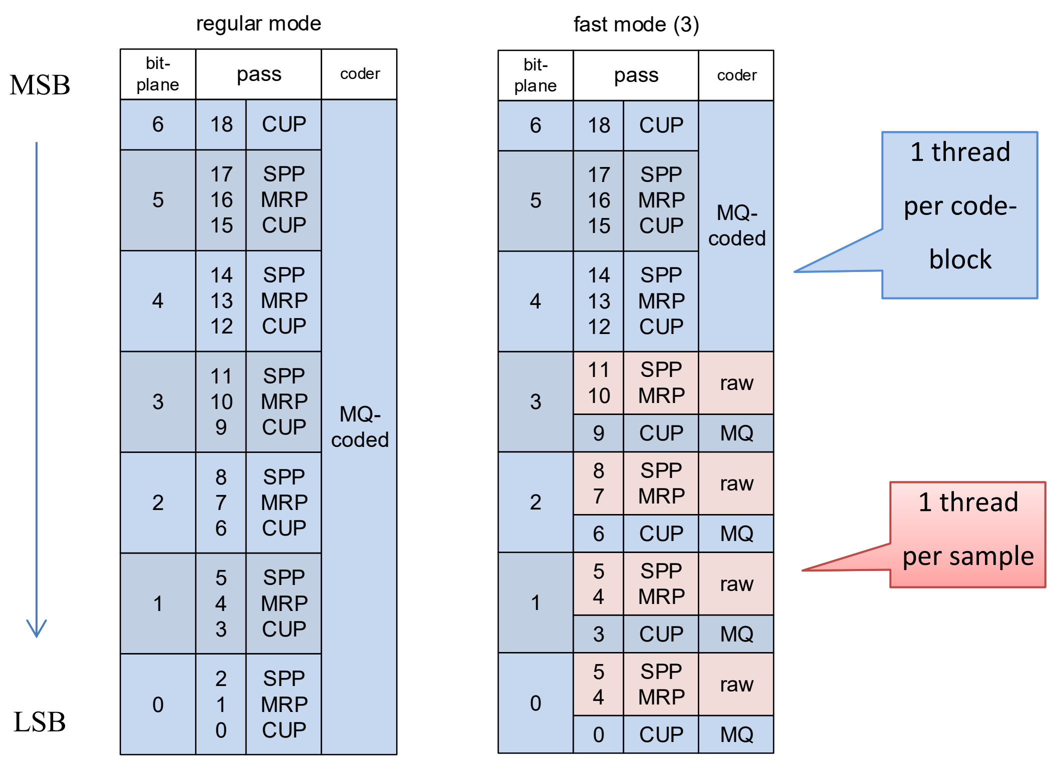


Figure 5-3 - Selective Arithmetic Bypassing Mode

5.3 Sample-Parallel Execution of EBCOT in Fast Mode

The fast mode is not permitted in any of the DCI, broadcast or IMF profiles. Nonetheless, it is helpful to find out whether a JPEG 2000 encoder running on a GPU can benefit from enabling this mode.

The contribution presented at the *Picture Coding Symposium* 2016 [75] is motivated by the observations laid out in the previous chapters and answers the following research question: *Can an implementation of EBCOT that targets parallel architectures such as GPUs benefit from the more finely grained parallelism exposed by this coding style and thereby increase its throughput*?

The contents of this chapter have been published in [75].

When operating EBCOT without fast mode, context modeling can be executed in a sample-parallel manner as previously discussed. Arithmetic encoding, however, does not expose intra-code-block parallelism. The MQ-encoding kernel is spawned with one thread per code block, which is far from ideal, as it does not meet the CUDA design concept of very lightweight threads.

With the fast mode enabled, context modeling is only required for the clean-up pass. However, in order to determine the pass membership in a concurrent manner, the general structure of the sample-parallel algorithm presented before still has to be followed. Magnitude bits from the SP and MR passes are raw-coded. They are coded in the same scanning order as before, but the coding operation merely appends the bit directly to the bit stream.

Given that context modeling and raw-coding can be done concurrently, the only coding step that is not well parallelizable is MQ-encoding of the context-decision pairs from the clean-up pass. A first question is whether this still justifies separating the coding steps into separate kernels with different respective threading layouts or whether it is faster to simply fuse all operations into a single kernel.

A high-level presentation of a parallel EBCOT"s tier-1 in fast mode, where all coding operations are realized in a single kernel, is presented in **Algorithm 5-1.**

Algorithm 5-1 - EBCOT tier-1 with bypassed SPP and MRP

one thread per stripe column

```
1.  contextModeling() // SPP+MRP+CUP

2.  numLocalSyms[] = countLocalSymbols()
3.  shared sharedMem[] // byte buffer

4.  // ---- SPP ----
5.  startIdx = exclScan(numLocalSyms[SPP])
6.  putBitsAtomic(sharedMem, startIdx,
7.  getLocalSppSyms(), numLocalSyms[SPP])

8.  // ---- MRP ----
9.  startIdx = exclScan(numLocalSyms[MRP])
10. putBitsAtomic(sharedMem, startIdx,
11. getLocalMrpSyms(), numLocalSyms[MRP])

12. // ---- SPP + MRP bitstuff & copy ----
13. totalNumSyms = scanTotalNumSyms(numLocalSyms)
14. numBytes = bitstuff(sharedMem,
15. totalNumSyms[SPP] + totalNumSyms[MRP])
16. if (threadId() == 0) // is block leader?
17.   terminateRawSegm(sharedMem)
18. if (threadId() < numBytes)
19.   bitstream[threadId()] = sharedMem[threadId()]

20. // ---- CUP ----
21. startIdx = exclScan(numLocalSyms[CUP])
22. putBits(sharedMem, startIdx,
23. getLocalCupCtxds(), numLocalSyms[CUP])

24. shared mqenc;
25. if (threadId() == 0) { // is block leader?
26.   startMqSegm(mqenc)
27.   for (i = 0; i < numLocalSyms[CUP]; i++)
28.     mqEncode(mqenc, bit stream, sharedMem[i])
29.   terminateMqSegm(mqenc, bit stream);
30. }
```

The context modeling related routines as discussed in the previous chapter are summarized in the algorithm's first line. Following the context modeling stage, the collected symbols need to be fed into the MQ- or raw-coder in the proper order: first symbols from the SPP, then those from the MRP, and finally all remaining symbols from the CUP. Symbols from the first two passes bypass the MQ-coder so that the threads can collaboratively create the bit stream segment in parallel. Each thread knows how many symbols it must code for the current pass (line 3).

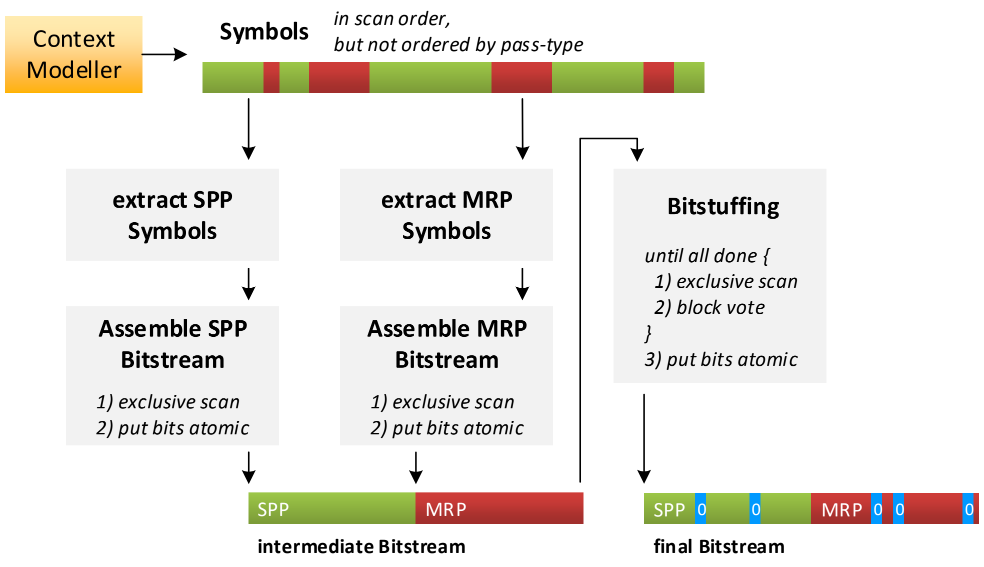


Figure 5-4 – workflow of sample-parallel EBCOT encoder in fast mode

The offset from the start of the bit stream segment where a thread needs to write the current pass's symbols is defined by the sum of all previous threads' symbols. It can be efficiently calculated by executing a block-wise exclusive prefix scan (lines 7, 12). Next, each thread copies its symbols to the bit stream. Since multiple threads will have to write to the same byte, they must use an atomic bitwise *or* operation. An implementation is described in detail in [76]. The proposed algorithm first creates the bit stream in an intermediate buffer in shared memory (see Figure 5-4). This is advantageous because bit stuffing is separated into a subsequent separate function that will have to read and write the bit stream at least one more time. A side-effect is that the implementation benefits from the native support for shared memory atomics introduced in *NVIDIA's Maxwell* architecture [77]. Next, the same steps need to be repeated for the SPP so that afterwards the entire raw bit stream segment will have been written to the intermediate buffer in shared memory.

5.4 Parallel Bit-stuffing

The JPEG 2000 standard requires that whenever there is a 0xFF byte in the bit stream, the next byte must include an extra zero bit stuffed into the MSB. The original motivation behind bit-stuffing is to avoid a carry-over in the MQ-encoder, which would in theory render bit stuffing unnecessary for raw bit stream segments. However, the fact that after bit stuffing the byte following an 0xFF is guaranteed to be in the range of 0x00 to 0x7F is additionally exploited to introduce marker codes, consisting of a 0xFF byte followed by a byte in the range of 0x90 to 0xFF. For this reason, bit stuffing applies to raw bit stream segments just the same. Bit stuffing is trivial when building the bit stream serially. However, when done in parallel, it requires some effort. Algorithm 5-2 provides an overview of the basic idea and Figure 5-5 visualizes an example.

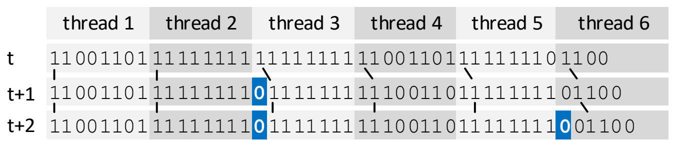


Figure 5-5 - Parallel bit stuffing example. Blue zero bits are inserted.

Each thread is in charge of writing one byte of the bit-stuffed output bit stream. In case the byte to the left is 0xFF and the byte before that is not 0xFF, a thread needs to insert a zero bit in the MSB (see thread 3 in Figure 5-5). To ensure that bits are inserted only after every other consecutive 0xFF byte, a segmented exclusive scan (see [37]) is used, where a segment represents back-to-back 0xFF bytes. The head flag, which indicates that a new segment must be started, is set whenever the previous byte is not 0xFF. Thread 4 in Figure 5-5 obtains a scan result of $n=1$, for example, because the previous $n+1=2$ bytes consist entirely of 1-bits. A thread needs to insert a zero bit only if the segmented scan result is an even number.

A consequence of inserting a bit is that all subsequent threads then need to reevaluate their decision:

a) It might be that they, too, inserted a zero bit, but that this is not actually necessary anymore given that a bit was inserted earlier on in the bit stream and all other bits were right-shifted.

b) Oppositely, it might be that they did not insert a zero bit, but now actually have to do it (see *thread 6* at $t+1$).

c) Finally, it might be that their decision still holds.

For this reason, the parallel bit stuffing routine might take multiple iterations to finish. After the initial decision, a block vote is conducted. In the first iteration, each thread votes whether or not they think they need to insert a zero bit. If none of them vote positively, no bit stuffing is necessary anywhere and the routine can finish. If at least one bit-stuff is required, the threads continue and re-evaluate their decisions, this time taking into account the previous threads' actions (after t+1, threads 4-6 know that one bit was inserted in an earlier byte). By running an exclusive scan over all threads' bit stuffing decisions, each thread can determine the amount of bit-stuffs in all previous bytes in the bit stream. Considering the bit shifts required by the zero bit inserts, they can then figure out which eight bits would eventually fall into the byte slot to their immediate left and again check, based on the result of a segmented scan, if they are required to insert a zero bit in their byte's MSB. Once again, each thread casts a vote, this time signaling if their decision of whether they need to include a zero bit or not has changed compared to the last vote. The entire process is repeated for as long as any of the threads had to correct their previous vote. In the end, each thread reads the seven or eight bits from the unstuffed bit stream that will fall into their output byte slot, insert a zero bit in the MSB or not, depending on their last vote, and finally overwrite the output byte in the bit stream.

Algorithm 5-2 - Parallel Bit-stuffing

one thread per byte

```
insert = false
numPrevInserts = 0
forever {
  lftFF = read8Bits(bit stream, (threadId()-1)*8-
    numPrevInserts) == 0xFF
  headOfSegm = !lftFF
  n = segmExclScan(lftFF, headOfSegm)
  insertNew = lftFF && isEven(n)
  myVote = insertNew != insert
  anyChanges = blockVote_any(myVote)
  if (!anyChanges)
    break
  numPrevInserts = exclScan(insertNew?1:0)
}
val = read8Bits(bit stream, threadId()*8
                            -numPrevInserts)
if (insert)
  val = (val>>1)&0x8F // insert 0-bit in MSB
bit stream[threadId()] = val
```

Since in CUDA registers are 32 bits wide, an optimization to the algorithm is for each thread to be in charge of writing four output bytes as opposed to only one byte. Special care has to be taken when participating in the scan or voting operations. Depending on the type of operation, a thread will have to recursively apply it to its four bytes and then contribute the overall result to the block-wide operation. The experimental results reported next include this optimization.

5.5 Experimental Results

Three different strategies for EBCOT's first tier in fast mode have been evaluated. They differ in how the routines for

1. context modeling
2. the two raw passes and
3. the MQ-encoded CUP

are organized into separate kernels. The starting point was the optimized, non-bypassed EBCOT implementation, which launches separate context modeling- (CM) and MQ-encoding- (MQ) kernels for every bit plane so that all code blocks are coded in parallel (Figure 4-7). Each code block starts processing at its first significant magnitude bit plane. Launching kernels individually for each bit plane is beneficial for two reasons. First, the memory required for storing the context-decision pairs collected by the CM-kernel and input to the MQ-kernel only has to be large enough to fit the worst-case amount of pairs from a single bit plane. Second, the MQ-kernel for bit plane *N* can be overlapped with the CM-kernel for bit plane *N-1*.

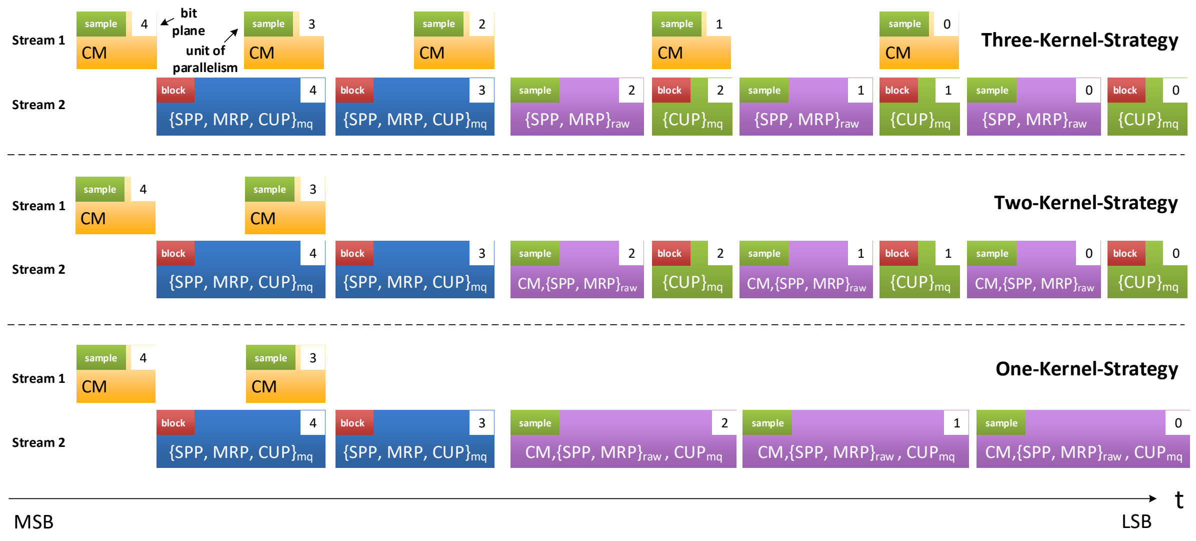


Figure 5-6 - Comparison of strategies on how to structure sample-parallel EBCOT algorithm in fast mode

Three-Kernel-Strategy – The first strategy splits the routines into three kernels: one for CM, one for coding the two raw passes and one for coding the CUP. The CUP-kernel cannot start before the raw-passes-kernel has finished, because it needs to know at which exact byte offsets to continue the code blocks' bit streams. However, the next bit plane's CM-kernel can be overlapped with the current bit plane's CUP-kernel.

Two-Kernel-Strategy - A drawback of the first approach is that the CM-kernel needs to store the modeled context-decision pairs to global memory in order to make them available to the coding kernels. Since the CM- and raw-passes-kernels both use an identical thread layout, the *Two-Kernel-Strategy* fuses them together into a single kernel. On the downside, the CUP kernel can then no longer be overlapped.

One-Kernel-Strategy - Still, the CM-kernel has to write the context-decision pairs for the CUP to global memory. The *One-Kernel-Strategy* fuses all routines into a single kernel. A positive side effect is that now the kernel can just as well loop over all bypassed bit planes internally. The drawback is that most threads stay idle while the block's first thread feeds the CUP-context-decision pairs to the MQ-encoder. However, the regular MQ-kernel with one thread per code block naturally suffers from high thread-divergence, which alleviates this drawback to some degree.

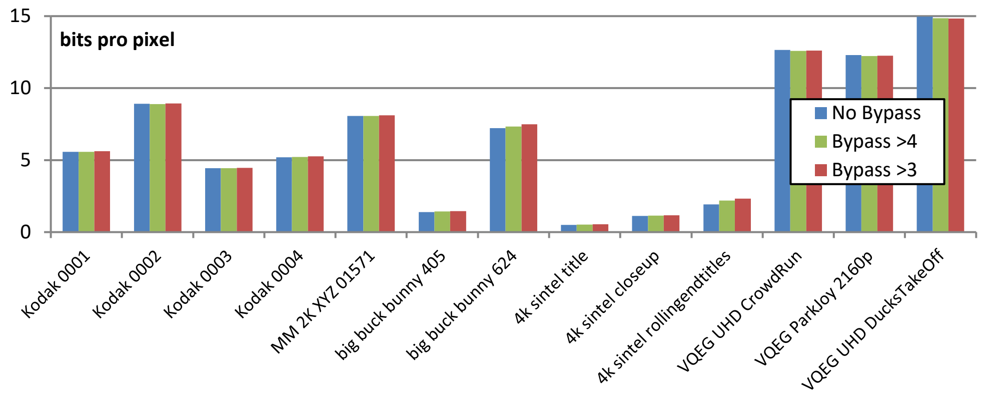


Figure 5-7 - Impact of fast mode (starting after 3 or 4 magnitude bit planes) on compression efficiency for a range of test sequences

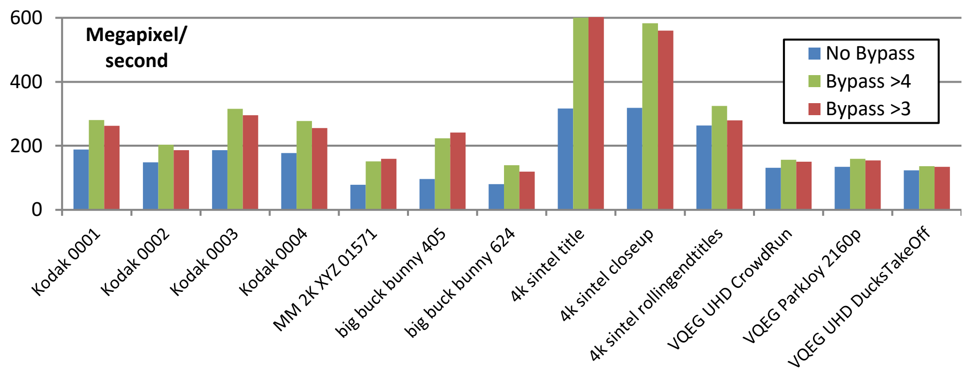


Figure 5-8 - Performance increase for EBCOT in fast mode when encoding single images (12 bit 4:4:4, no max. bit rate).

Experiments show that the *One-Kernel-Strategy* outperforms the previous two when encoding single images. It yields a speed-up of up to 2x compared to EBCOT without fast mode (Figure 5-8). However, the speed-up is not equally high for all test images. By enabling the fast mode, only the execution of the less significant bit planes (LSBs) is sped up. The number of code blocks that still have pending significant bit planes to be processed decreases with every bit plane. By the time the coder gets to the bypassed LSBs, the number of remaining code blocks has become so low, that a high-end GPU is severely under-occupied. At this point, the duration for processing the remaining LSBs without fast mode is almost constant, irrespective of the test image's resolution and entropy. The execution time for the MQ-coded bit planes, though, depends much more heavily on the resolution and entropy.

Thus, the relative time spent in the bypassed bit planes is lowest for the high-resolution, high-entropy images (*VQEG, Kodak*). The relative speed-up for only the bypassed bit planes is highest for those images where the GPU was most severely under-occupied.

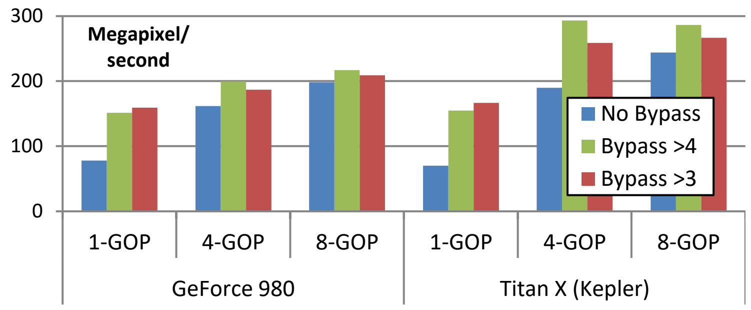


Figure 5-9 - Performance increase for EBCOT in fast mode, when encoding frames in parallel from the (12 bit, 4:4:4, no max. bit rate, 2.7 bit per sample) *SteM Confetti*.

An alternative approach to increase the amount of parallel work units is to process code blocks from multiple images in parallel. This concept is denoted as group-of-pictures (GOP)-coding here. Figure 5 compares the throughput for different GOP-sizes. The benefit of the fast mode decreases as the GOP-size, and with it the number of parallelizable code blocks, increases. However, even at a GOP-size of eight, the parallel fast mode implementation yields an increase of throughput of 5-10% for the high-entropy UHD sequences, 10-15% for the 2K sequences and up to 20% for the low-entropy 4K sequences. Since now the overall GPU occupation is higher, the *three-kernel-strategy* performs best: the timesaving gained by overlapping CM and CUP outweighs the cost of additional memory accesses.

The impact of the bit stuffing routine was only measureable for the high-entropy 4K images, where it accounted for less than 3% of the total EBCOT tier-1 runtime. The number of inserted zero bits ranges between 30 ppm for the high-entropy *Kodak_0003* test image and around 300 ppm for the low-entropy *4k_sintel_title* image. The average number of loop iterations required for the proposed parallel bit stuffing algorithm lies between two and three for all test images.

5.6 Conclusion

The research question, whether JPEG 2000's selective arithmetic coding bypass mode ("fast mode") is beneficial to parallel GPGPU implementations can be answered positively. Algorithms for a sample-parallel execution of the raw-coded passes and subsequent bit stuffing as well as a strategy on how to best organize them into GPU kernels were proposed. The increased parallelism speeds up EBCOT tier-1 by a factor of two for low-entropy images where only very few code blocks have more than four significant bit planes. For high-entropy 4K images, a moderate speed-up of between 10% and 30% was measured. The compressed size lies within 1% of that of the unmodified EBCOT, with the exception of those test images with text, especially *Sintel rolling titles* (14% larger) and *Sintel title* (6% larger), where the SPP compresses well even in the high-frequency bands and at lower bit planes (Figure 5-7).

If the increased latency and memory consumption can be tolerated, multiple frames should be encoded in parallel in order to increase the GPU-occupancy. In this case, the benefit of the fast mode is diminished significantly to speed-ups between 5% for high-entropy images and 27% for low-entropy images.

6 Rate Control and Parallel Packetization

Regardless of whether a GPU-accelerated is configured to create DCI- or IMF-compliant code streams or operates with the selective arithmetic coding bypass style (or the standard-incompliant modes presented in subsequent chapters), once all other coding steps have been optimized, EBCOT tier-2 (rate control and packetization) becomes a bottleneck. As this step is concerned with executing the rate control and assembling the truncated code block bit streams into a code stream, this is especially true for high bit rates (stated in Mbit/sec) and low frame rates, such as an IMF package with only 25 fps, when single code streams get very large.

Research question: how can the PCRD-Opt. and packetization routines be efficiently computed on a GPU? Which co-processing model is the fastest?

The contents of this chapter have been published in [71].

6.1 Proposed GPU/CPU Co-Processing Models

Figure 6-1 shows three possible co-processing models that differ in the points at which to switch the execution back from the GPU to the CPU. In the following, three models are proposed:

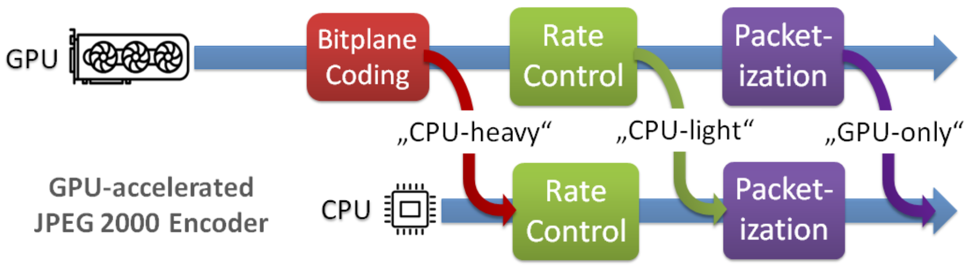


Figure 6-1 – GPU/CPU Co-Processing Models for Rate Control and Packetization

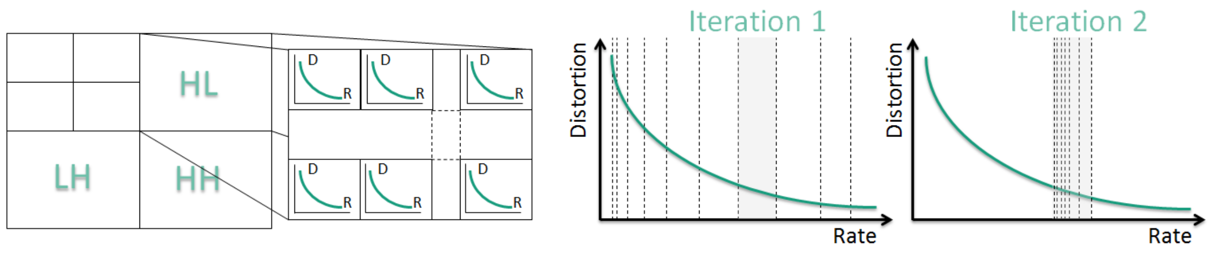
CPU-heavy - After the code blocks' bit streams have been produced on the GPU, the final *PCRD-Opt*. and packetization routines are executed by the CPU. Since the GPU and CPU can operate in parallel, the packetization of frame *N* on the CPU is overlapped with the compression of frame *N+1* on the GPU. However, a packetization on the CPU entails that the required data needs to be transferred into the host memory first. Since the bit streams' lengths are not known ahead of time, they need to be produced into dedicated fixed-size slots of a sufficiently large memory block, so that bit streams are eventually interspersed with unused gaps. Either the gaps are included in the memory transfer or the bit streams need to be compacted first. Additionally, a set of per-code-block metadata is required for the packet header construction: the number of all-zero bit planes, number of passes, pass lengths and pass slopes. Given that a UHD image without chroma-subsampling comprises around twenty thousand code blocks and assuming up to 15 magnitude bit planes per block and two bytes per metadata value, this amounts to ~3.5 MB per image, which is often more than the compressed code stream itself. While this metadata is only used internally, it nonetheless increases the PCI traffic significantly to the point where it can pose a bottleneck.

CPU-light – As part of the bisection search for the optimal slope threshold the entire code stream creation is simulated multiple times. By far the most computationally demanding sub-routine involved in this process is the packet header construction. In the *Hybrid* co-processing model, this search including the code stream simulation is executed on the GPU, and only the final packetization is left for the CPU. The pass lengths and slopes then do not have to be transferred, but instead only the total lengths of the truncated bit streams.

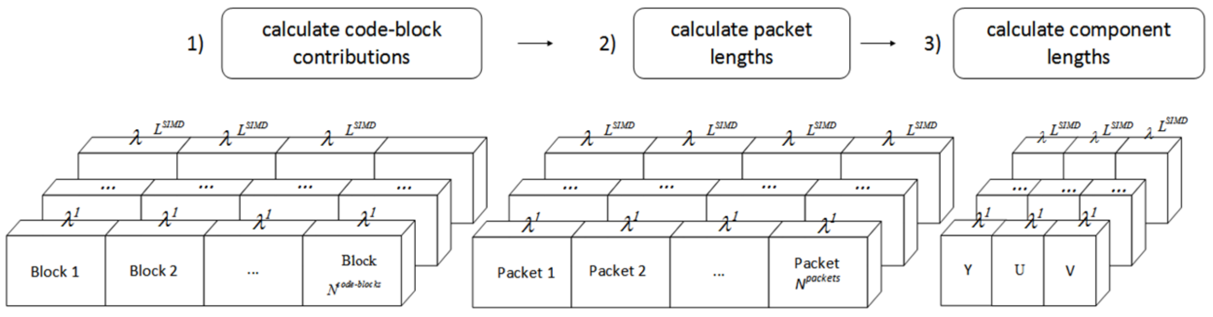
GPU-only – in this model, all processing steps are carried out on the GPU. A positive side effect is that the device-to-host transfer is decreased to the bare minimum: only the final code stream needs to be transferred, no additional per-code block metadata.

6.2 Parallel Post Compression Rate Control Optimization

In order to gain from a parallel architecture, the *PCRD-Opt*. algorithm is redefined to exploit two levels of parallelism. Figure 6-2 (a) shows the rate-distortion plot of a single code block. The truncation points are laid out on a convex curve. Instead of probing only one slope threshold at a time, multiple slopes (dashed lines) that are spaced apart equally in the current search window are probed in parallel. Iteratively, the search window is narrowed until a slope threshold is found that yields a code stream size sufficiently close to the specified maximum size. A natural choice for the number of simultaneously probed thresholds is the GPU architecture's SIMD-group size L^{SIMD}. Tests with a detail-rich UHD sequence showed that for 32 parallel probes, two iterations are usually sufficient to get within 5% of the desired code stream size. When probing only one threshold at a time, it requires nine iterations to get within 5% and sixteen iterations to get as close as possible.



(a)


1) calculate code-block contributions
2) calculate packet lengths
3) calculate component lengths
$\lambda^{L^{SIMD}}$
λ^1
...
Block 1
Block 2
Block $N^{code\text{-}blocks}$
Packet 1
Packet 2
Packet $N^{packets}$
Y
U
V

(b)

Figure 6-2 – PCRD-Opt. with two dimensions of parallelism: (a) probe multiple slope thresholds in parallel and (b) process code blocks and packets in parallel in a two-level reduction

The second level of parallelism is to process code blocks and packets in parallel. Figure 6-2 (b) shows how the overall code stream size is calculated in a two-level reduction.

1. A first kernel 1) computes per code block and probed slope threshold the number of passes and truncated bit stream length. To do that it counts the passes up to the truncation point and sums up their compressed byte lengths.
2. Subsequently, a second kernel 2) with one thread per packet and threshold computes the packet lengths by accumulating the bit stream lengths of all code blocks contained in that packet (up to 192) and adding to that the length of the simulated packet header.
3. Packet sizes are accumulated by color channel since for *DCI* profiles the maximum size per channel is also constrained. The fixed overhead for the tile-part headers also counts towards the per-channel limit and needs to be accounted for.
4. Finally, the total code stream size is the sum of all channels plus the fixed overhead of the main header.

6.3 Parallel Packetization

The code stream anatomy can be described by a table of cells, *C,* where each cell C_i^a represents a section of the final code stream that is constructed independently, *a* being the corresponding cell type abbreviation and *i* the occurrence of that cell type. Table 6-1 lists all cell types.

Table 6-1 – Code stream anatomy of cell types

Type	Abbr.	Occurrences	Length
Main Header	*MH*	1	fixed
Tile Part Length	*TLM*	1	fixed
Start of Tile and Start of Data	*SOT*	N^{TP}	fixed
Packet Header	*PH*	$N^{packets}$	dynamic
Packet Body Code-Block	*PB*	$N^{code\ blocks}$	dynamic
End of Code stream	*EOC*	1	fixed

The code stream anatomy stays constant for an entire image sequence and serves as a convenient indirection in order to hide the progression order from the GPU-kernels. The construction of each code stream then comprises three phases (as depicted in Figure 6-3):

1. Populate a table $L[C_i^a]$ with the cells' exact sizes in bytes
2. Prefix-Sum over the cell lengths $L[C_i^a]$ in order to obtain a table $\Delta[C_i^a]$ containing the cells' byte offsets in the final code stream
3. Construct the cells' contents and write them to the appropriate positions, obtaining the final code stream in GPU memory

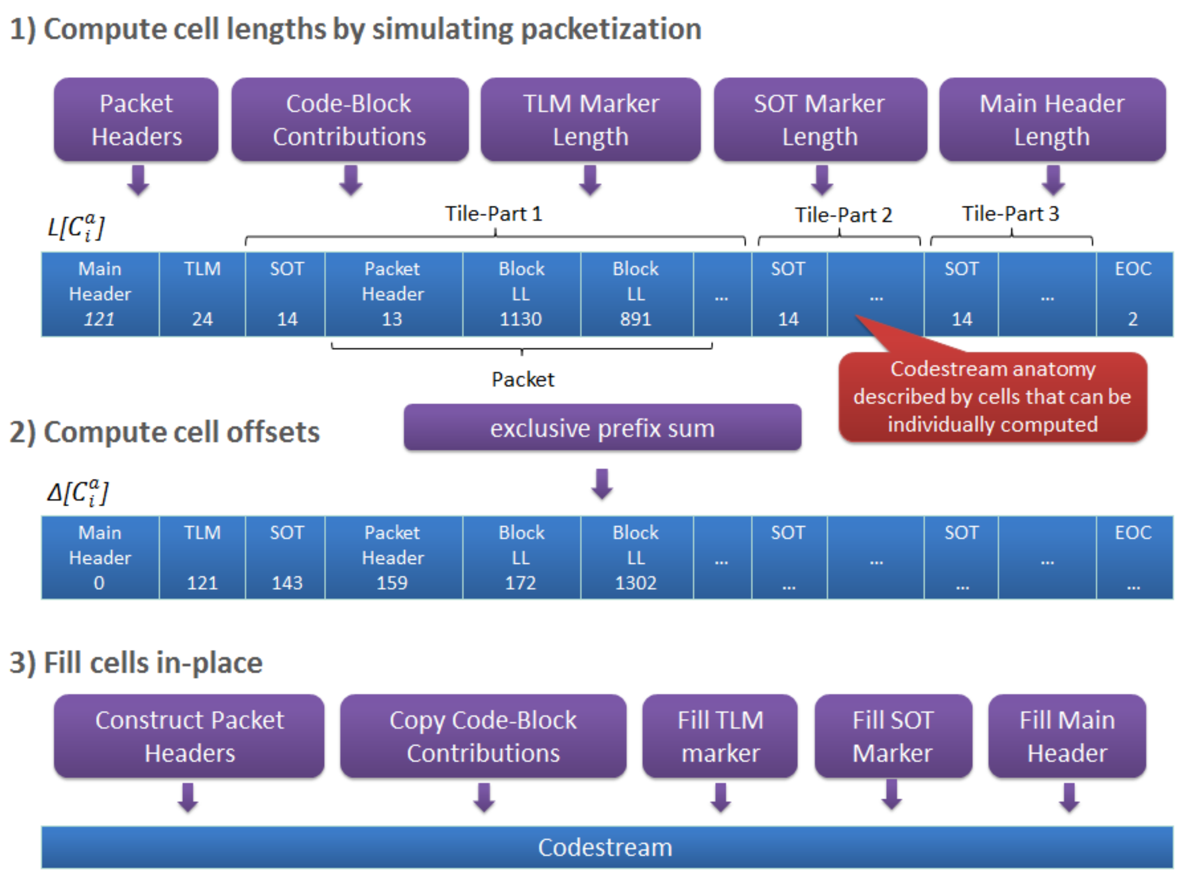


Figure 6-3 - Overview of parallel packetization

Given the number of tile-parts, N^{TP}, number of packets, $N^{packets}$, and the number of code blocks, $N^{code\ blocks}$, the number of cells, C_N, is then

$$C_N = 1(C^{MH}) + 1(C^{TLM}) + N^{TP} + N^{packets} + N^{code\ blocks} + 1(C^{EOC})$$

In the first phase, a table $L[C_i^a]$, containing the lengths in bytes of each cell, is populated. $L[C^{MH}]$ comprises the combined lengths of the main header markers[3]. Instead of transferring all required information to the GPU, the main header is constructed on the CPU. As long as the comments text marker does not change, this cell's size remains fixed for an entire image sequence. The size of $L[C^{TLM}]$ depends only on N^{TP} and the precision chosen to store the tile-parts' lengths and tile-part indices. All these variables stay constant for an entire image sequence, and thus $L[C^{TLM}]$ remains fixed. The SOT and SOD markers have a fixed size per specification.

Let b_j denote the embedded bit stream for code block B_j and b_j^z the prefix of b_j up to and including pass z. λ_j^z defines the slope for pass z of code block B_j, and L_j^z the length of b_j^z. All $L[C_p^{PB}]$ can be filled with the lengths L_j^z for all code blocks B_j that belong to precinct p. Thread coordinates can be conveniently mapped to cells C_p^{PB} via a look-up-table with $N^{code\ blocks}$ entries. The mapping of code blocks to precincts is established by a combination of three look-up tables: let $N_p^{x,y}$ denote the number of code blocks (in

[3] SOC, SIZ, COD, QCD and COM

rows *y* and columns *x*), in precinct *p* and ΔB_p the position of the precinct's first code block *x=0*, *y=0*, in a list of all code blocks sorted by packet membership $\hat{B}$. The code blocks belonging to precinct *b* are then $\hat{B}[\Delta B_p]$, …, $\hat{B}[\Delta B_p + N_p^{x,y} - 1]$.

All $L[C_p^{PH}]$ can be computed by simulating the header construction. In fact, the packet headers have already been simulated during the last *PCRD-Opt.* iteration, so an implementation can choose to store the packet header sizes and reuse the set for the selected slope threshold λ here.

Once all cells in the table $L[C_i^a]$ have been filled, the cells' offset in bytes, $\Delta[C_i^a]$, in the final code stream can be computed by executing a device-wide exclusive prefix-sum.

$$\Delta[Ci] = ExclusivePrefixSum\ (L[C_i^a])$$

Parallel implementations are available through toolboxes like *thrust* or *cub* [31, 33]. The final code stream size (excluding the length of the two-byte EOC-marker), $L^{code\ stream}$, can now be looked up in the last element of the table Δ:

$$L^{code\ stream} = \Delta[C_{N-1}^{EOC}] + L[C_{N-1}^{EOC}]$$

In the third phase, the code stream cells are filled with data. A memory block in GPU memory of size $L^{code\ stream}$ needs to be reserved, so that the individual cells can be directly written to their final locations. All cells except those of type C^{MH} and C^{EOC} are filled on the GPU. The author opted to split the tasks into four kernels – one for each of the remaining cell types. They can operate in parallel.

The C^{TLM} cell contains the lengths, L_k^{TP}, of each tile part TP_k. Similarly, C_k^{SOT} cells contain the length and index of their respective tile parts. The length of a tile part can be obtained from $\Delta[C_i^a]$:

$$L_k^{TP} = \begin{cases} \Delta[C_{k+1}^{SOT}] - \Delta[C_k^{SOT}], & 0 \leq k < N^{TP} - 1 \\ \Delta[C^{EOC}] - \Delta[C_k^{SOT}], & k = N^{TP} - 1 \end{cases}$$

The C^{PH} cells contain the side information required by the decoder to correctly interpret the compressed code blocks. The construction of tag trees and comma codes is analog to an implementation for a CPU. Since packet headers are always padded to byte boundaries, parallel threads will not write to the same byte address. Therefore, it is not necessary to write out bits with the atomic *or* intrinsic and initialize the target cell with zeros. Instead, the GPU implementation can collect emitted bits in an intermediate register in order to reduce expensive accesses to global memory. The kernel for filling C^{PH} cells is almost identical to the simulation kernel used to compute $L[C^{PH}]$. They only differ in that the latter merely counts emitted bits, while the former actually writes them to memory.

Filling in the C^{PB} cells is essentially a device-to-device copy operation as the individual bit streams b_j are already located in device memory. However, they are interspersed with gaps and yet untruncated. The task at hand is to copy each bit stream's prefix b_j^z for the previously determined slope threshold λ into the corresponding cells C_j^{PB}. A naive approach would be to have one thread per code block copy the bit stream prefix byte by byte. A faster, because more parallel, solution is to assign multiple threads to each code block and have them copy the bit stream prefix chunk wise. Again, a good choice for the amount of threads per code block is the architecture's SIMD-group size L^{SIMD}. Considering the size of a cache line $L^{cache\text{-}line}$, each thread should then copy $N = L^{cache\text{-}line} / L^{SIMD}$ bytes per iteration until L_j^z bytes have been copied by the SIMD group j. The destination address in global memory is defined by the code stream offset and therefore alignment constraints cannot be honored, but the source bit streams b_j should be placed at addresses that are a multiple of $L^{cache\text{-}line}$.

After all four kernels have finished populating their respective cells the code stream can be transferred into host memory where the remaining two cells C^{MH} and C^{EOC} can be filled.

6.4 Experimental Results

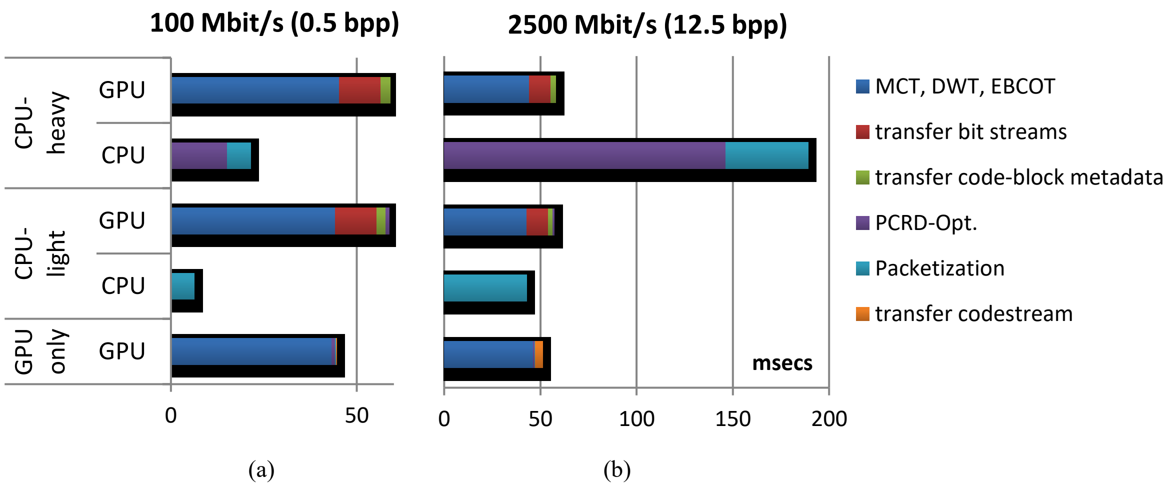


Figure 6-4 - Encoder runtimes, for VQEG Crowd Run UHD sequence, left: 0.16 bits/sample (100 Mbit/s @ 24 fps), right: 4.2 bits/sample (2500 Mbit/s @ 24 fps), DCI-compliant settings (except bit rate). GPU: GeForce GTX 1080, CPU: Kakadu v6.4 single-threaded on Intel i7 3960X

Figure 6-4 shows a comparison of the three evaluated co-processing models. Since the GPU and CPU can work on different frames in parallel, their bars are laid out side-by-side. The CPU implementation, *Kakadu v6.4*, does not employ multi-threading for either *PCRD-Opt.* or packetization since its architecture is optimized for streamlined line-by-line encoding, where the image does not have to be entirely loaded into memory at any point [78]. For this discussion, we will assume the measured CPU-runtimes can be divided by the number of CPU cores without overhead.

In the low-bit rate scenario, Figure 6-4 (a), executing both PC*RD-Opt.* and packetization on the CPU does not present a bottleneck as the GPU-portion of the encoder always takes longer. However, executing all stages on the GPU leads to an overall acceleration for concurrently processing a group of eight frames from 59 ms (16.9 fps) to 44 ms (22.4 fps). This gain stems from two reasons:

1. in order to run *PCRD-Opt.* on the CPU, the pass lengths and slopes of all code blocks have to be transferred from the GPU to the CPU, which is no longer necessary in the *GPU-only* mode

2. the device-to-host transfer of the compressed data is also faster, because the portions of the bit streams discarded by the rate control are not included anymore and the bit streams are arranged back to back without gaps.

Figure 6-4 (b) shows the runtimes for the same test sequence, when compressed less heavily. Without multi-threading, the CPU-portion takes around four times as long as the GPU portion and thus poses a bottleneck. In this scenario, a multi-threaded CPU implementation running on at least four cores would be required to remove the bottleneck. Although this was neither examined nor the goal of this work, the parallel algorithms presented in this paper should also be applicable to a CPU implementation. However, the finely grained parallelism should be grouped into coarser work units, reducing the number of threads that need to be spawned to an amount that is closer to the number of virtual CPU cores present in modern CPUs.

The GPU portion's runtime in the *CPU-light* model stays almost unchanged as the search for the best threshold takes only 0.7 ms on the GPU compared to 146 ms on the CPU. Creating the code stream entirely on the GPU takes only an extra half a millisecond. Reducing the device-to-host data transfers from 13.7 ms to 3.9 ms, the *GPU-only* model yields an overall gain from 17.2 fps to 19.5 fps, assuming the CPU-portion were sufficiently sped up by multi-threading. With respect to the single-threaded implementation evaluated here, the encoder's throughput is increased from 5.3 fps to 19.5 fps.

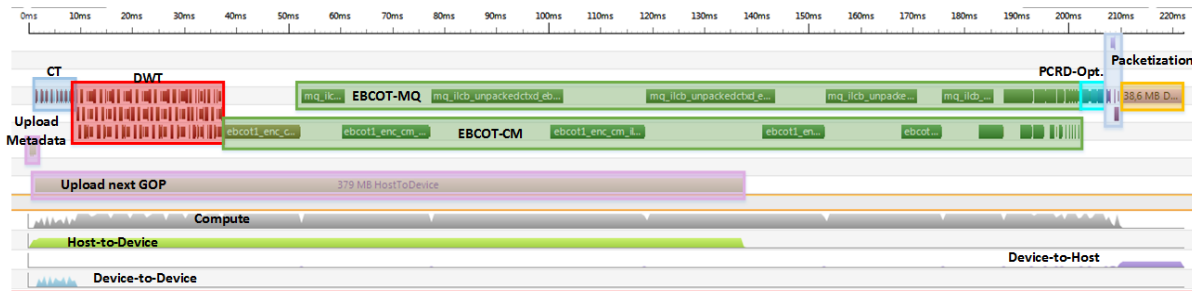


Figure 6-5 - NVIDIA Nsight profiling timeline with rate control (light blue) and packetization (dark blue) executed on the GPU.

The above figure is a screenshot from the *NVIDIA Nsight* profiler timeline. It shows how the rate control and packetization kernels are executed right after the EBCOT tier-1 context modeling (CM) and MQ-coding (MQ) kernels.

6.5 Conclusion

To answer the research questions, the *PCRD-Opt.* algorithm can be efficiently executed on a GPU, where parallelism in two dimensions can be exploited: (1) probe multiple truncation point sets in parallel and (2) process code blocks and packets in parallel. At large bit rates, *PCRD-Opt*. executed on a CPU by a single-threaded implementation was shown to be the bottleneck in an otherwise entirely GPU-optimized JPEG 2000 encoder. It runs about two orders of magnitudes faster when run on a GPU.

Furthermore, a parallel table-driven algorithm for executing the packetization on a GPU was presented. It runs 80x faster compared to the single-threaded CPU implementation, but more importantly, it leads to an overall speed-up of the encoder, because the amount of data that needs to be transferred into host memory is minimized. Table 6-2 lists a condensed summary of the findings.

Table 6-2 – Comparison of results for co-processing models

Model	Discussion
CPU-heavy	• PCRD-Opt. requires multi-threading at high bit rates or else poses bottleneck • Packetization uncritical, when done in parallel with GPU, but then leads to higher latency
CPU-light	• PCRD-Opt. 200x faster for UHD example with 12.5 bpp • Discarded portions of bit stream can be omitted from transfer, but requires that bit streams are compacted first • Per-Code-Block metadata still needs to be transferred for packetization on CPU
GPU-only	• Packetization 100x faster for UHD example with 12.5 bpp on GPU. • Transfer is minimized to bare code stream, which leads to overall speed-up. • Lower latency, because no coding steps executed on CPU anymore.

In conclusion, creating the code stream entirely on the GPU pays off. The time saved in the transfer outweighs the extra time it takes to run the rate control and packetization on the GPU. This effect weighs in more heavily when compressing high-entropy images to low bit rates, but could be alleviated by anticipating which passes will end up being discarded by the rate control and not coding them in the first place.

When compressing to high bit rates, on the other hand, it was shown that rate control and packetization would need to be multi-threaded when run on the CPU or otherwise the CPU portion will present a bottleneck. Creating the code stream entirely on the GPU has the positive side effect that the overall runtime is decreased, since then the rate-control is carried out on the GPU and the GPU-to-CPU transfer is reduced to the already truncated bit streams. If, on the other hand, the rate control is carried out on the CPU, the entire bit streams need to be transferred to the CPU, since it is not yet known if and where they will be truncated. This benefit weighs in more heavily at low bit rates when more data is discarded by the rate control. If the available bit rate budget is not exhausted, the amount of data to be transferred is identical regardless whether the rate control is computed on the GPU or CPU. Additionally, the latency is reduced, because the GPU and CPU do not need to operate on different frames in parallel anymore. Finally, and this is an important aspect in applications where video compression is only one of many parallel tasks, the CPU is left free to do other tasks.

7 JPEG 2000 Incompliant Extensions

Chapter 5.1 pointed out that at high data rates the EBCOT compression efficiency is poor for the lower bit planes that contain mainly noise. The standard addresses the high data rate scenario by defining the optional selective bypassing mode where part of the samples bypass the arithmetic coder and are raw-coded. It only imposes a minor penalty, if any, on the compression efficiency. However, the experimental results from the previous chapter have shown that the speed-up achieved in a GPU implementation is only mediocre.

This chapter elaborates on the question how the throughput can be further increased on massively parallel architectures when giving up compatibility with the JPEG 2000 standard. Two algorithmic changes are proposed:

1. Single-pass scan order
2. Raw Bit-plane coding

The research questions that arise are: How much of a speed-up can be achieved? What is the cost in terms of a diminished compression efficiency?

The contents of this chapter have been published in [79].

7.1 Single-Pass Scan Order

When targeting a massively parallel architecture such as a GPU, the goal is to break down the problem at hand into as many parallelizable work units as possible.

Encoder: For each bit plane, an encoder can model the samples' contexts in a sample-parallel fashion by using the iterative significance propagation algorithm presented in [39]. The resulting set of context-decision pairs is subsequently fed into the MQ-coder. Due to its context-adaptive nature, this arithmetic coding step does not expose any intra-block parallelism. Therefore, since both kernels (context modeling and MQ-coding) expose very different degrees of concurrency, a separation into two distinct kernels that are each invoked once per bit plane in an interleaved order is well suited. For each bit plane p, where $p=0$ is the LSB, the host launches a sample-parallel context modeling kernel ($CM\text{-}kernel_p$) followed by a block-parallel arithmetic coding kernel ($MQ\text{-}kernel_p$).

Decoder: In contrast to the encoder, the EBCOT decoding routine does not expose any intra-block parallelism since the significance propagation for a bit plane cannot be precomputed. Context modeling and MQ-decoding are tightly coupled in a feedback loop:

The context of any sample $X[n]_p$, where n is the location (x,y) with $0 \leq x <$ block width and $0 \leq y <$ block height, cannot be computed without first decoding sample $X[n-1]_p$.

A GPU implementation can therefore combine both tasks in a single kernel that does not have to be launched individually for each bit plane. Instead, the kernel itself loops over all bit planes with three passes each.

On a modern GPU with thousands of cores, the block-wise MQ-kernel tends to under-occupy the GPU due to its coarse parallelism. This effect worsens as the number of code blocks with bit planes pending to be processed decreases with every processed bit plane. Implementations can counter-act by processing blocks from a group-of-pictures (GOP) simultaneously at the cost of an increased latency. With current GPUs, the implementation at hand saturates at a GOP-size of eight for 2K 4:4:4 images with approximately 5000 blocks per image.

The CM- and MQ-kernels are memory bound, which means that the execution of instructions is often stalled by pending memory requests. Accordingly, the throughput can be increased by reducing required read- or write accesses to memory. The encoder's context modeling-kernel can be launched with a layout as finely grained as one thread per sample. After the initial iterative significance propagation, each thread determines their sample's context by examining the neighbors. Due to the three-pass-scan order, the context-decision pairs need to be sorted by pass type before they can be handed over to the MQ-kernel via global memory (Figure 7-1 left). Each thread needs to find out at which offset to write their SPP-, MRP- or CUP-context-decision pairs. This can be achieved collaboratively by running three prefix-sum scans [35].

The proposed single-pass mode renders this additional burden unnecessary by giving up the separation into three passes altogether (Figure 7-1 right). The encoder's context modeling-kernel then requires only a single prefix-sum scan in order for each thread to determine where to write their context-decision pairs. Context labeling is not affected by the new scan order and can remain unchanged. The decoder benefits as well in that it does then only need to scan through all samples once per bit plane. Additionally, the sample state representation is simplified by one bit, as it is no longer necessary to remember for insignificant samples

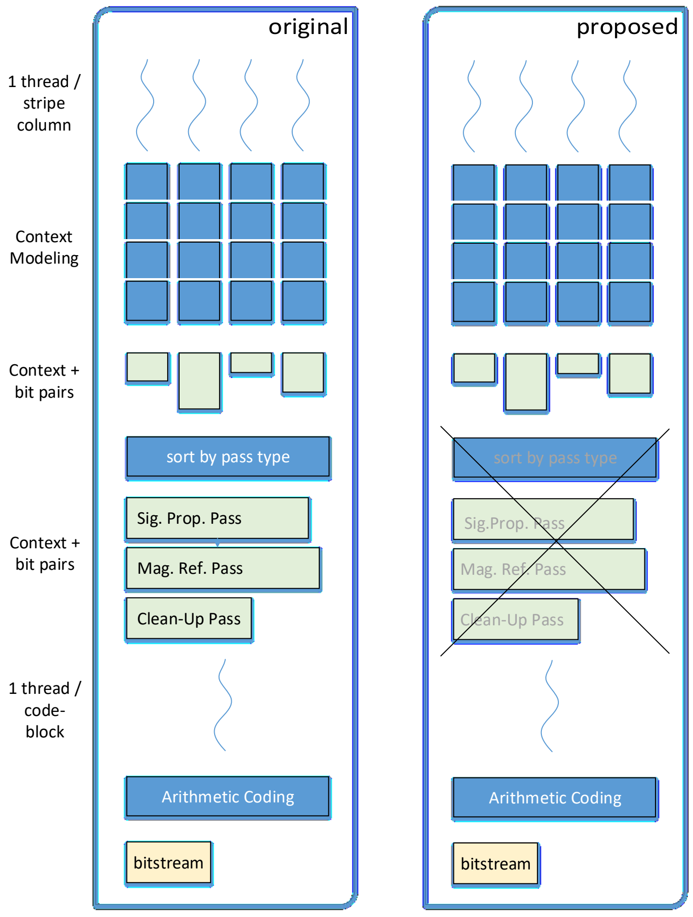


Figure 7-1- Overview Single-Pass mode

whether they had already been coded in the SPP in order to determine their membership to the CUP. The altered coding procedure is laid out in **Algorithm 7-1.**

Algorithm 7-1 - single-pass mode

```
foreach bit plane
  foreach sample in stripe-oriented scan order
    if (sample is not significant and
        any neighbor is significant)
      zeroCoding()
    if (magnitude bit is 1)
      markSampleAsSignificant()
      signCoding()
    else if (sample is significant)
      magnitudeRefinementCoding()
    else if (first row in stripe and
      all stripe column neighbors insignificant)
      runLengthCoding()
      jumpToEndOfRun()
    else
      zeroCoding()
      if (magnitude bit is 1)
        markSampleAsSignificant()
        signCoding()
```

There are two downsides to giving up the pass separation.

- Firstly, the number of eligible truncation points is decreased to a third, leaving the *Post-Compression-Rate-Distortion-Optimizer* (PCRD-Opt.) less flexibility when the maximum data rate constraint is exceeded. It can be argued that this limitation is not severe when compressing to high data rates as is the case in the use-case examined in this work. The literature shows that the coding performance achieved by various scanning order strategies is virtually identical at the end of each bit plane [51].
- Secondly, the neighborhood taken into consideration when computing the context for any of the four types of coding operation needs to be constrained to the scan order past. Whereas in the original mode, operations carried out during the MRP or CUP can already leverage the information whether or not any of the neighbors, past or future, has turned significant during the current bit plane's SPP, this is not possible anymore in the proposed variant. This might slightly impact the compression efficiency.

7.2 Raw Bit plane Coding

Part-1 of the JPEG 2000 standard defines the selective bypassing mode (fast mode). When enabled, all symbols coded during the SP- and MR-passes starting from the fifth coded magnitude bit plane are not arithmetically coded anymore, but instead appended directly to the bit stream. The underlying assumption is that the compression efficiency for these two pass types is poor at low bit planes, while for the CUP it is still reasonable to invoke the MQ-coder. The bit stream is then terminated at every switch between the raw and coded mode. A parallel implementation can benefit from this mode by separating the raw coding phases into a separate kernel that can be executed in a sample-parallel fashion. However, even the partially bypassed bit planes still necessitate the extra tasks of evaluating and updating each sample's state, including the significance propagation, in order to be able to collect all symbols and order them by pass type.

In line with the single-pass mode, the proposed raw-coding mode is also processed bit-plane-wise without a pass separation. Magnitude bits are directly concatenated to the bit stream. A block width of 32, which is used in DCI and IMF profiles, conveniently matches the SIMD group size and register size for NVIDIA CUDA devices. The encoder can collect the magnitude bits from a single row into a 32-bit register by using the *ballot* SIMD-group intrinsic and then write it out to memory. All rows of all blocks can be written in parallel. For a block size of 32x32 the magnitude bits amount to a total of 128 bytes per bit plane.

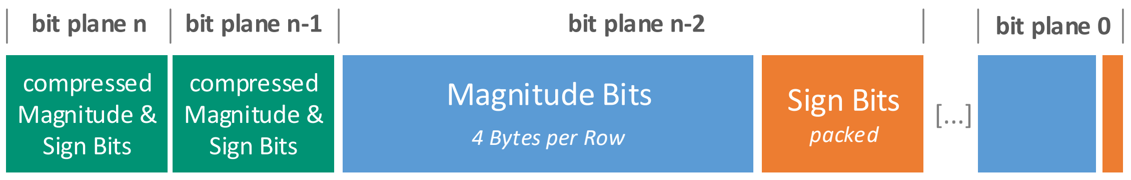


Figure 7-2 - Code stream structure when raw coding is entered after the 2nd significant bit plane. Code-block size: 32x32

Since it is not guaranteed that every sample has already turned significant by the point the first bypassed bit plane is processed, not all sign bits have necessarily been coded yet. With the goal of maintaining the embeddedness-property of the created code stream, a sample's sign bit should be coded immediately following its first significant magnitude bit. However, due to the lack of sub-bit plane truncation points, magnitude and sign bits from the same bit plane do not need to be interleaved. Instead, sign bits for newly significant samples are stored after the magnitude bits. This enables a GPU implementation to exploit that magnitude bits are laid out in a regular grid and can be accessed in parallel. Figure 7-2 shows the resulting code stream structure.

The proposed implementation strategy is to collect first the sign bits row-wise by again using the *ballot* SIMD-group intrinsic. A first ballot collects which positions in a row have turned significant; a second ballot collects the actual sign bits. One thread per row then writes the rows' new sign bits tightly packed to the bit stream. Since multiple threads might simultaneously need to write to the same byte, bits are stored using a bitwise atomic *or* operation [76]. The offset beyond the end of the magnitude bits can be computed by scanning the prefix-sum across the numbers of sign bits in each row.

7.3 Experimental Results

At first, the impact of the single-pass scan order is evaluated by examining the benchmarking results. The single-pass scan order removes some of the dominant stall reasons identified by profiling the context modeling kernel. The below diagrams compare the runtime (Figure 7-3 a, d), workload (Figure 7-3 b, e) and instruction efficiency (Figure 7-3 c, f) between the original mode and the single-pass scan order. It can be seen how the runtime improves especially in the MSBs, because the workload decreases and the efficiency slightly increases.

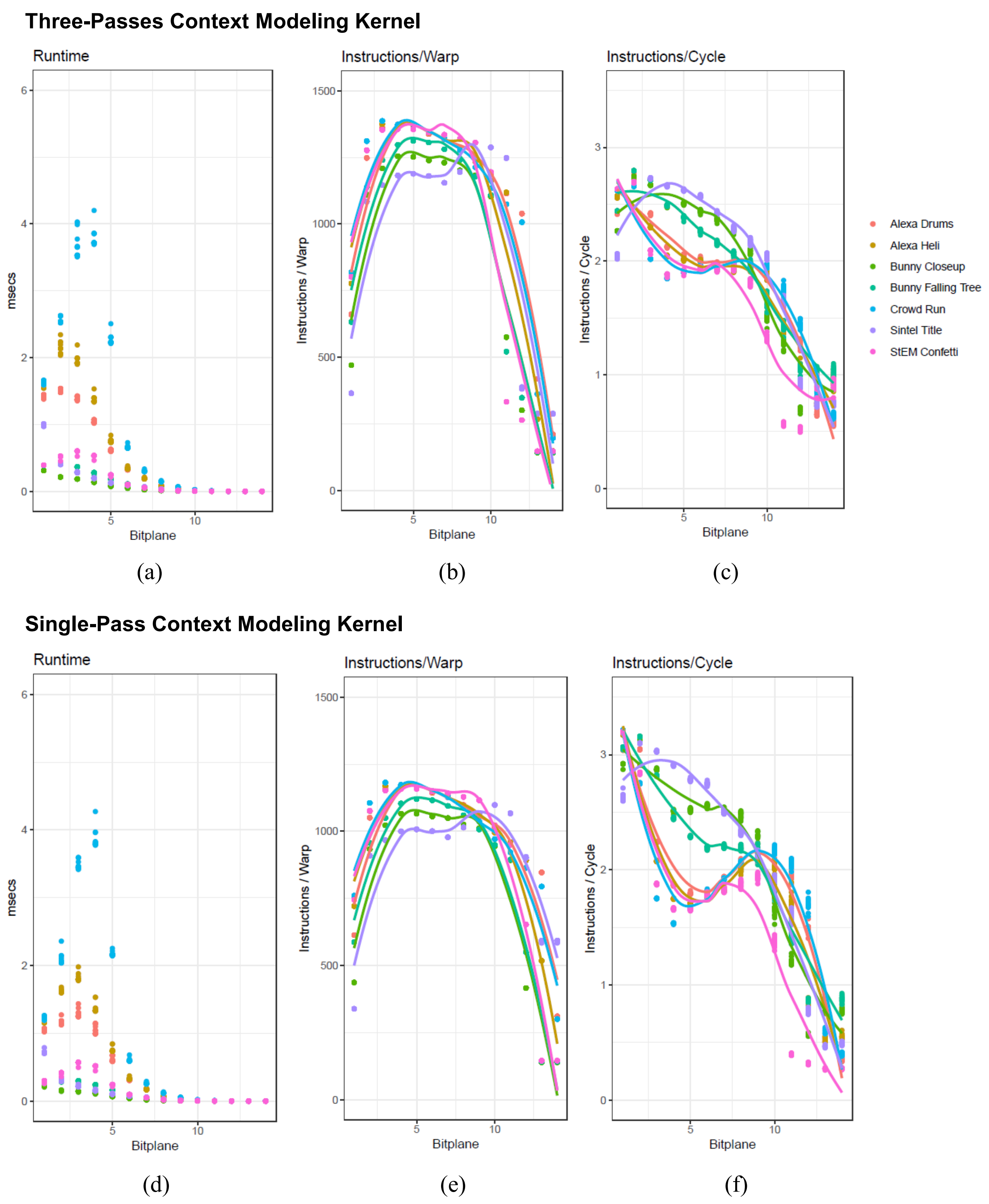


Figure 7-3 – Comparison of runtime (**a**, **d**) and benchmarking metrics instructions/warp (**b**, **e**) and instructions/cycle (**c**, **f**) for context modeling kernels with three passes (**top row**) vs. single pass (**bottom row**).

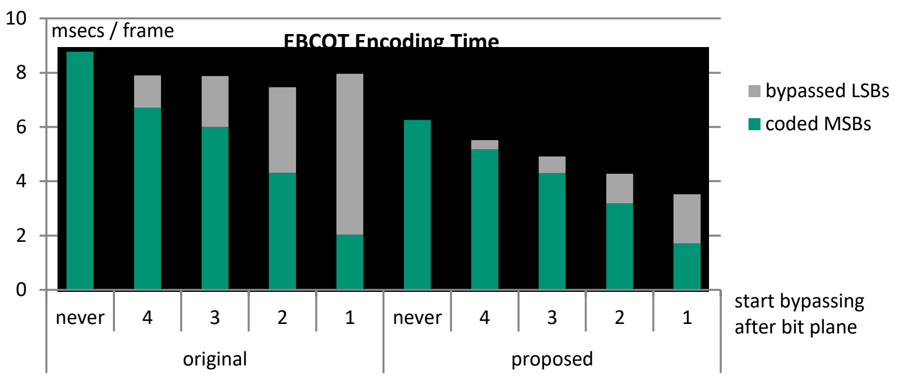


Figure 7-4 – EBCOT encoding time, separated into entropy- and raw-coded bit planes. Source: *Tears of Steel*, frames 1290-1547, 4096x1714

Figure 7-4 compares the EBCOT encoding runtime of the original three-pass mode with and without selective arithmetic coding with the new proposed single-pass and raw-coding modes. All experiments have been carried out on an *NVIDIA GeForce GTX 1080*. The speed-up due solely to the proposed single-pass mode is approximately 1.4x and can be seen by comparing the two single-colored bars, where bit planes *never* bypass the arithmetic coder. The standard-compliant bypassing mode brings only little benefit for the throughput for a GPU implementation. The proposed raw coding of the lower bit planes, on the other hand, yields an additional speed-up that grows as more bit planes are bypassed.

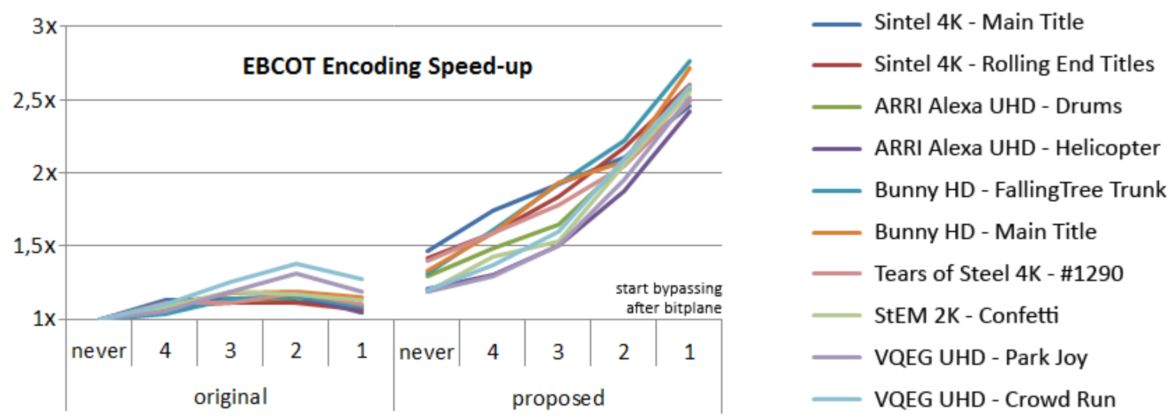


Figure 7-5 - Encoding speed-ups per test sequence for single-pass and raw-coding modes

Figure 7-5 presents an overview of the achieved speed-ups for a range of high-resolution 48 bit RGB 4:4:4 image sequences. These sequences were chosen instead of the standard test still images, since processing blocks from multiple images in parallel increases the throughput significantly. Duplicating a still image would produce unrealistic results since then the execution divergence due to different image contents is not taken into account.

The gain is highest for scenes with only few details, since then most code blocks contain only few non-zero bit planes. In this case, the GPU is most heavily under-occupied without the proposed new coding modes, as only few code blocks remain active in the lower bit planes. The average speed-up for the single-pass mode is 1.3x with a factor of 1.2x on the low end for the detail-rich *VQEG* or *StEM* sequences and almost 1.5x on the high end for the low entropy images from the *Sintel* or *Big Buck Bunny* sequences. When enabling the raw-coding mode after the fourth (third) significant bit plane, the average speed-up is increased to 1.6x (1.8x).

The impact of the single-pass mode on the decoder is lower, but still yields an average speed-up of 1.16x. The proposed raw-coding mode works better in the decoder, though, with speed-ups of 1.5x and 1.9x when enabling it after the fourth or third plane, respectively.

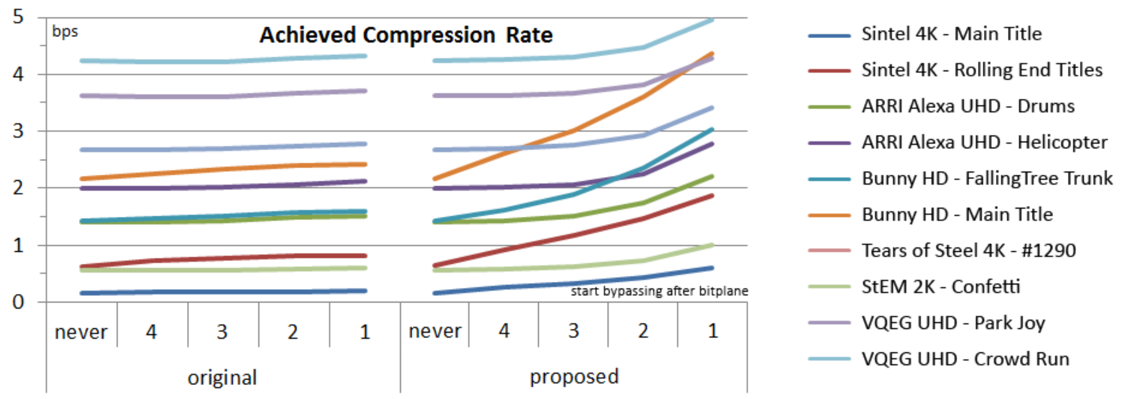


Figure 7-6 - impact per test-sequence of single-pass and raw-coding mode on compression rate

The down side to the proposed modes is a diminished compression efficiency. Figure 7-6 plots how the achieved compression rate decreases as more bit planes are subjected to raw coding. The compression rate decrease due to the single-pass mode is only about 0.1%. As expected, the data rate grows significantly, as more bit planes are coded raw.

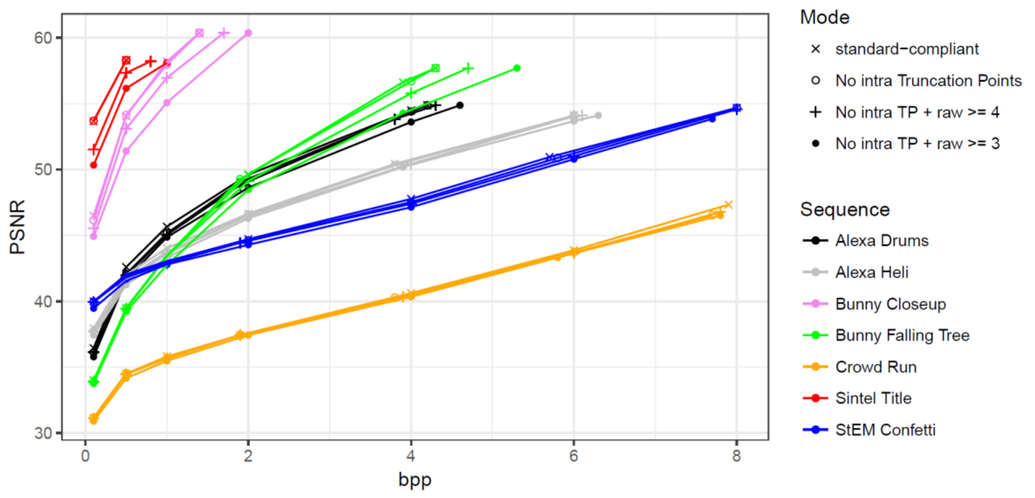


Figure 7-7 - Rate-Distortion Plot for single-pass mode and raw-coding mode vs. the default EBCOT mode.

The quality loss caused by the lower compression efficiency stays mostly within 0.5 dB PSNR as shown in Figure 7-7. The delta between the original and single-pass mode is virtually invisible in the RD-plot. The quality loss due to raw-coding is highest for the low-entropy sequences, where the PSNR was high to begin with and a slight loss in PSNR is tolerable as it remains in the "visually lossless" domain.

All sequences have been compressed with a sufficiently high maximum data rate so that no bit streams had to be truncated. It has yet to be examined how the quality compares when compressing to a fixed target rate, given that the single-pass mode yields fewer truncation points.

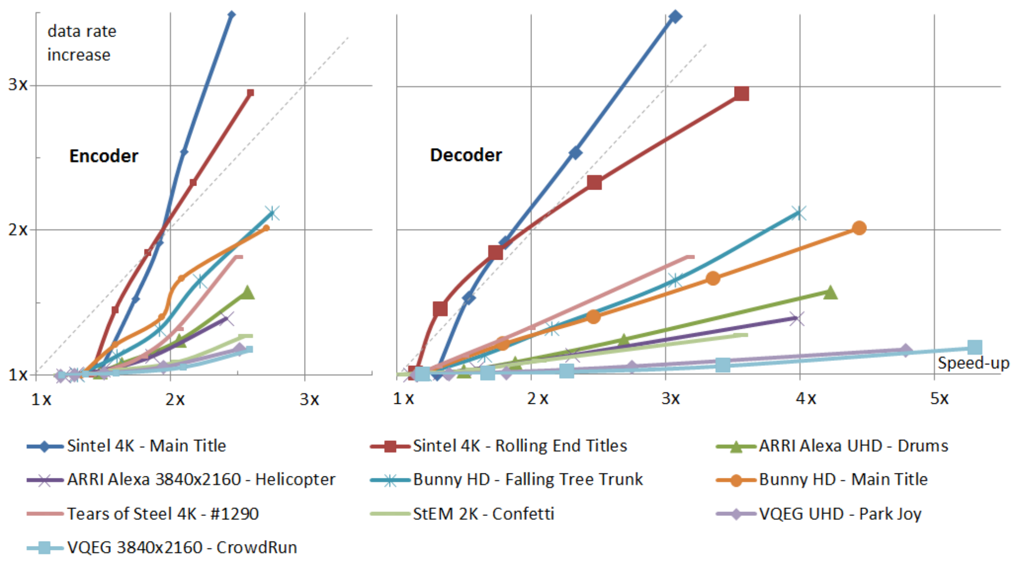


Figure 7-8 - speed-up vs. data rate trade-off per test-sequence for single-pass and raw-coding mode. Points below diagonal line present a positive trade-off.

Ultimately, an assessment of whether the gain in the form of increased encoding or decoding throughput outweighs the cost in terms of an increased data rate depends on the use case. An attempt is made here by plotting the speed-up vs. the relative increase in data rate (Figure 7-5). Every curve connects five points, which represent the result for no bypassing, bypassing after 4, 3, 2 or 1 significant bit planes. For those points that lie below the diagonal dashed line, the speed-up is higher than the relative increase in data rate.

All except two sequences stay entirely below the dashed line. The two exceptions correspond to low entropy sequences that had a high compression efficiency and throughput to begin with and could therefore be regarded as uncritical.

An attempt to condense these findings further is made in the following table.

Table 7-1 - Speed-up for single-pass and raw-coding mode. GeForce GTX 1080, 6 bpp, 4:4:4, 36 bit per pixel

	No intra TP, no raw-coding	**No intra TP raw-coding ≥ 4**	**No intra TP raw-coding ≥ 3**
Encoder	up to 1.3x	up to 1.4x	up to 1.5x
Decoder	up to 1.15x	up to 1.4x	up to 1.6x

7.4 Conclusion

Based on an examination of the bottlenecks of a JPEG 2000 implementation for GPUs, two new modes for its entropy coder EBCOT are proposed that lead to a higher throughput. The variants are standard-incompliant but compatible with the JPEG 2000 framework and can be losslessly transcoded into standard-compliant profiles. The resulting code stream is still embedded so that decoders can apply the technique of code-pass skipping in order to increase their throughput further when it is not a strict requirement to fully decode the compressed images, e.g. when previewing 36 bit DCPs on a 24 bit computer monitor [80].

The single-pass mode alone yields speed-ups between 1.2x and 1.5x for the encoder and 1.05x to 1.3x for the decoder at virtually no decrease in compression rate.

The throughput can be further increased by enabling the raw-coding mode on top of the single-pass mode. In contrast to the selective-arithmetic coding bypass mode defined in the JPEG 2000 standard, this proposed mode can be leveraged by a GPU implementation to increase the throughput significantly. This comes at the cost of an increased data rate, but the relative speed-up outweighs the relative increase in data rate for all tested detail-rich sequences. It presents an opportunity to flexibly configure the trade-off between speed and compression rate.

8 BPC-PaCo

Recently, F. Auli-Llinas from the *Universitat Autònoma de Barcelona* has also proposed an alternative GPU-friendly entropy coding scheme, named *Bitplane-Coding with Parallel Coefficient Processing* (BPC-PaCo), that can be dropped into the JPEG 2000 framework, replacing the time-consuming EBCOT algorithm [52]. In short, it is laid out for a code block size of 64x64 and employs a scan order where each parallel SIMD-thread independently processes in a zig-zag order a vertical stripe of two sample columns and feeds symbols into their own local arithmetic coder [51].

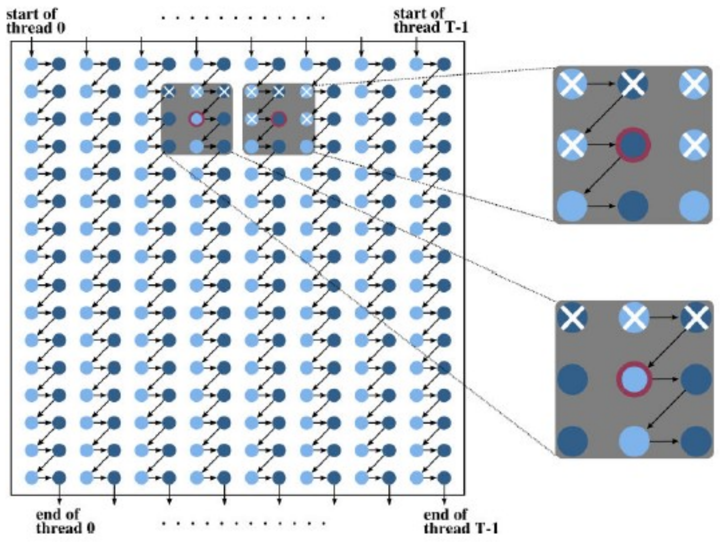


Figure 8-1 – BPC-PaCo scanning order, where a MxN code block is processed by a group of M/2 threads in lock-step parallelism. Each thread scans through two consecutive columns in a zig-zag scan order. The figure is copied without changes from [50]

The above figure shows how a group of threads walks in a tightly synchronized “lock-step” fashion through the rows from top to bottom.

Compared to EBCOT with 32x32 code blocks, the number of samples per arithmetic coder is reduced by eight from 1024 to 128 samples, which would lead to the problem of context dilution. However, this problem is avoided by making the trade-off of giving up context adaptiveness and instead use a stationary probability model that requires a-priori knowledge about the image characteristics [81–83].

The research questions that arise are: How does BPC-PaCo compare to the other coding modes presented in this work? What are the drawbacks and advantages in terms of compression efficiency and throughput? How strongly are the results affected by the choice of the static probability map created offline? Can the coding mode be further optimized for the use case of visually lossless image compression?

8.1 Stationary Probability Model

The stationary probability model for the no longer context-adaptive binary arithmetic coder is derived from the authors' insight that the MPS' probability for any given context does not change significantly within a bit plane as shown in the following figure. They propose to give up the context adaption and instead devise a static probability map with a probability per context, bit plane, subband and color component. For the example of 3 channels, 16 bands per channel (at 5 DWT decompositions), 18 context and 14 bit planes, this amounts to ~12.000 probabilities that are stored as 16 bit fixed point values. Certainly, the amount could be reduced by choosing a variable amount of bit planes, since 14 bits are not required for high frequency bands.

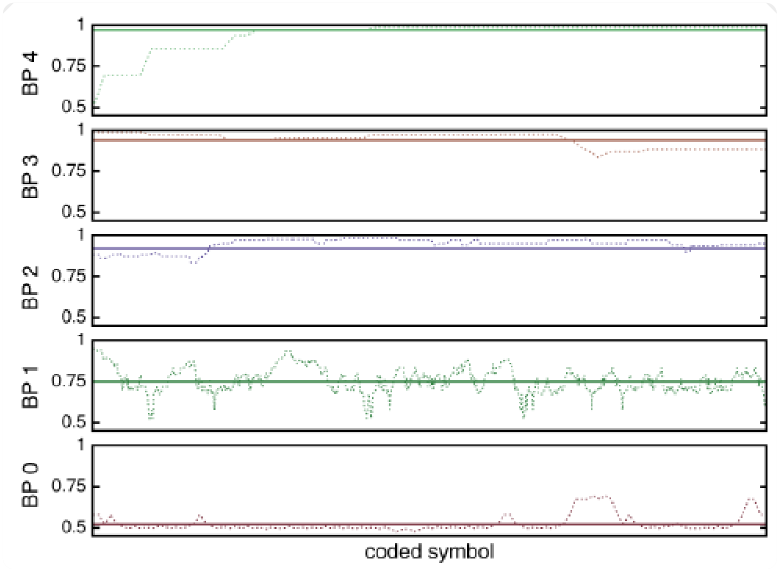


Figure 8-2 – MPS probabilities (dashed line) for one of the significance coding contexts. It is visible that the probability is centered on the bit plane's average probability. The figure is taken without changes from [82] (Figure 3)

Since the change is ultimately motivated by the goal to devise a faster coder, it is not an option to build the probability map at runtime based on the image characteristics due to the time this would take. Instead, the authors propose to pre-define models for various classes of images offline and use those. The proposed classes are *natural, aerial, AVIRIS* and *XRAY*. Results indicate that the proposed scheme reaches the same quality as EBCOT for code block sizes of 32x32 or 64x64.

8.2 Fixed-Length Coding

In [84], the authors propose to replace the variable-to-variable length MQ-coder used in JPEG 2000 with a new variable-to-fixed length coder, named FLW for "fixed-length code words".

The MQ-coder creates a single large code word, whose length cannot be predicted beforehand. In order to be compatible with registers of limited precision, it needs to renormalize frequently. In contrast, the FLW-coder emits a short code word as soon as a predefined bit length is obtained. By design, it does not need to renormalize, which reduces its complexity and leads to a speed gain of 10-30% compared to MQ-coder.

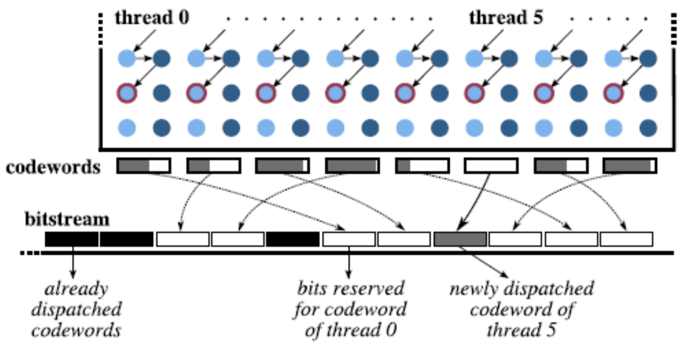

thread 0
thread 5
codewords
bitstream
already dispatched codewords
bits reserved for codeword of thread 0
newly dispatched codeword of thread 5

Figure 8-3 – fixed-length code words from all parallel threads are interleaved in a deterministic way. The figure is copied without changes from [50]

The code words' fixed length is then utilized to derive a deterministic scheme on how to interleave the threads' individual entropy coders' outputs. Whenever a thread is about to start a new code word, it first reserves a slot in the output buffer where the code word will be written. The slot is assigned by querying each round from all participating threads whether they need a new slot or not using intra-SIMD-group voting primitives.

8.3 Two or three passes per bit plane

The authors proposes to operate BPC-PaCo with either two or three passes per bit plane. Since both the context modeling and arithmetic coding phases are executed with finely grained parallelism, there is no need to split these operations into separate kernels. Instead, the entire block coding stage is carried out within a single GPU kernel that follows the algorithm laid out below.

Algorithm 8-1 – Original BPC-PaCo encoding kernel

one thread per two columns of 64x64 code block

```
1.  dwtCoeffs2x64[] = Load2x64DwtCoeffs()
2.  MSB = findMsb(dwtCoeffs2x64)
3.  if (use3passes)
4.    runPass_CUP(MSB-1, dwtCoeffs2x64)
5.  for (b in MSB-1:0) { // for each bit plane b
6.    runPass_SPP(b, dwtCoeffs2x64)
7.    runPass_MRP(b, dwtCoeffs2x64)
8.    if (use3passes)
9.      runPass_CUP(b, dwtCoeffs2x64)
10. }
```

Each thread initially loads their two columns of 64 samples each into thread local memory (line 1). Identical to the regular EBCOT context modeling kernel, all threads of a thread block then jointly find the most-significant bit plane that contains at least one 1-bit (line 2). Then they iterate over the bit planes and execute the two (or three) pass types (lines 5-9).

The next algorithm shows the significance propagation (SPP) pass. The magnitude refinement and clean up passes are structured analogously and will not be repeated here. The threads jointly iterate through each row from top to bottom (line 2) and then code their left and right samples (line 3). The threads operate in a tightly synchronized fashion, i.e. all left samples of a row are processed concurrently, followed by all right samples. In order to determine the sample's context from the 3x3 neighborhood, the samples need to be loaded (line 4). Since each thread only loaded their own 2x64 samples into local memory, the three samples from the left and right thread are accesses via the register-shuffling instructions that allow exchanging registers within the threads of a single SIMD-group.

Algorithm 8-2 - Original BPC-PaCo SPP pass

one thread per two columns of 64x64 code block

```
1.  runPass_SPP(b, dwtCoeffs2x64, color, band) {
2.    for (row r in 0:63) {
3.      for (col c in 0:1) { // left and right
4.        nbh3x3[] = load3x3Nbh(r, c)
5.        if !significance(nbh3x3[4]) { // not yet significant
6.          // "zc" zero-coding
7.          ctx = determineZcCtx(nbh3x3)
8.          prob = LUT(color,band,zc,b,ctx)
9.          d = (nbh3x3[4]>>b)&1
10.         acEnc(prob, d)
11.         // "sc" sign-coding
12.         if (d) {
13.           markSignificant(nbh3x3[4])
14.           ctx = determineScCtx(nbh3x3)
15.           prob = LUT(color,band,sc,b,ctx)
16.           d = sign(nbh3x3[4])
17.           acEnc(prob, d)
18.         }
19.       } // significant?
20.     } // left and right
21.   } // for each row
22. }
```

One bit of each local sample is used to store whether the sample has already turned significant (line 5). The coding rules of the passes have not changed with respect to the regular EBCOT algorithm. When a sample is not yet significant, it is coded with the zero-coding primitive. A context is inferred from the neighbor's significance state (line 7). In contrast to EBCOT, which uses a context-adaptive arithmetic coder, the probability for the coding operation and context is now looked up from the static look-up table (line 8). The decision bit (line 9) and probability are then passed to the fixed-length arithmetic coder (line 10). If the decision bit is one (line 12), the sample turns significant and the sign will be coded next following the same structure (line 13-17).

8.4 Proposed change: Single-Pass Scan Order

Can the coding mode be further optimized for the use case of visually lossless image compression?

Whereas the previous chapters on BPC-PaCo merely summarized the details published by Auli-Linas et al, this chapter discusses original unpublished contributions proposed by the author of this work. Analogous to the EBCOT mode proposed in chapter 7.1 "Single-Pass Scan Order", the original BPC-PaCo algorithm is modified to remove the concept of code passes. The overhead of having to pass over the samples in each bit plane more than once is given up by sacrificing intra-bit-plane truncation points. This will cost quality, especially at lower bit rates, but improve the throughput.

Algorithm 8-3 – Modified "1pass" BPC-PaCo encoding kernel

one thread per two columns of 64x64 code block

```
1.  dwtCoeffs2x64[] = Load2x64DwtCoeffs()
2.  MSB = findMsb(dwtCoeffs2x64)
3.  for (b in MSB-1:0) // for each bit plane b
4.    uint64_t sigFlagsPerRow[3][2] // sliding window registers:
5.                                  //   top, center, bottom row
6.                                  //   left and right 32 cols
7.    for (r in 0:63) { // for each row r
8.      loadStateForNewRow(sigFlagsPerRow, r,
9.                         dwtCoeffs2x64)
10.     for (c in 0:1) { // left and right column c
11.       if sigFlagsPerRow[1][c]&threadIdx.x { //significant?
12.         // "mrc" magnitude-refinement-coding
13.         ctx = determineMrcCtx(sigFlagsPerRow,r,c)
14.         prob = LUT(color,band,mrc,b,ctx)
15.         d = (dwtCoeffs2x64[r][c]>>b)&1
16.         acEnc(prob, d)
17.       }
18.       else { // not yet significant
19.         // "zc" zero-coding
20.         ctx = determineZcCtx(sigFlagsPerRow,r,c)
21.         prob = LUT(color,band,zc,b,ctx)
22.         d = (nbh3x3[4]>>b)&1
23.         acEnc(prob, d)
24.         // "sc" sign-coding
25.         if (d) {
26.           markSignificant(sigFlagsPerRow,r,c)
27.           ctx = determineScCtx(sigFlagsPerRow,r,c)
28.           prob = LUT(color,band,sc,b,ctx)
29.           d = sign(dwtCoeffs2x64[r][c])
30.           acEnc(prob, d)
31.         }
32.       } // significant?
33.     } // left and right
34.     commitState(dwtCoeffs2x64, sigFlagsPerRow, r)
35.   }// for each row
36. }
```

A significant change compared to the original BPC-PaCo algorithm is that the threads scan only once per bit plane over all samples. This also enables the way for accessing the significance states with a stenciling technique. When iterating over the 64 rows, the significance bits for a sliding window of three rows are maintained in thread-local registers (3x64 bits) (line 7 above, line 1-12 below). After each row's coding operations have been carried out, the sliding window registers are updated with the changes from the other threads (line 34 above, lines 16-19 below) and the new significance state is stored to the thread local persistent memory (line 22-25 below).

Algorithm 8-4 – Modified "1pass" BPC-PaCo encoding kernel

one thread per two columns of 64x64 code block

```
1.  loadStateForNewRow(sigFlagsPerRow[3][2], r, dwtCoeffs2x64) {
2.    if (r==0)
3.      sigFlagsPerRow = {0}
4.    else {
5.     sigFlagsPerRow[top] = sigFlagsPerRow[center]
6.     sigFlagsPerRow[center] = sigFlagsPerRow[bottom]
7.     sigFlagsPerRow[bottom][left] =
8.       ballot(significant(dwtCoeffs2x64[r+1][left]))
9.     sigFlagsPerRow[bottom][right]= // right
10.      ballot(significant(dwtCoeffs2x64[r+1][right]))
11.   }
12. }
13.
14. commitState(dwtCoeffs2x64, sigFlagsPerRow[3][2], r) {
15.   // update local registers
16.   sigFlagsPerRow[center][left] =
17.     ballot(sigFlagsPerRow[center][left] & threadIdx.x)
18.   sigFlagsPerRow[center][right] =
19.     ballot(sigFlagsPerRow[center][right] & threadIdx.x)
20.
21.   // update persistent state
```

```
  if (sigFlagsPerRow[center][left] & threadIdx.x)
    significant(dwtCoeffs2x64[r][left]) = 1
  if (sigFlagsPerRow[center][right] & threadIdx.x)
    significant(dwtCoeffs2x64[r][ right]) = 1t
}
```

8.5 Look-up Table Structure

The look-up tables are stored in a CSV format with the following columns:

- keys
    1. Color channel index: {1-3}
    2. Coding Operation: {sign-coding “sc”, zero-coding “zc”, magnitude-refinement coding “mrc”}
    3. Resolution level: {4K: 1-7, 2K: 1-6}
    4. Subband: {resolution level 1: LL, resolution level > 1: LH, HL, HH}
    5. Bit plane: {1 (MSB) – 14 (LSB)}
    6. Context: {sz: 4, mrc:1, zc:9}
- values
    1. total number of coded symbols: *N*
    2. number of coded symbols with value 0: N_0
    3. probability p e{1-127} that symbol is 0: $p = \frac{\mathrm{N0}}{N}$

The rows are sorted in the order the keys are listed above. Given this predefined order, only the third value column with the probabilities needs to be made available to the GPU kernels by uploading it into the constant read-only memory during the initialization of the GPU encoder or decoder.

The BPC-PaCo code published along with the original paper did not include functionality to include new LUTs. Instead, the encoder was modified to include a mode where it collects and stores the symbols along with the keys listed above. Since both the total number of symbols and number of zero-valued symbols per key is known, combining two or more sequence-specific LUTs into an average LUT is straight forward and will not be discussed here further.

8.6 Probability Estimates Variance

BPC-PaCo relies on probability estimates that are computed offline a-priori for a representative corpus of images. In many real-world scenarios, such knowledge about the image characteristics will not be available and so a generic default probability map will have to be used. It is thus important to know the quality variance between probability estimates inferred from various image corpora. This chapter will describe an experiment where probability lookup tables (LUTs) have been specifically computed for each of a set of image sequences. Additionally, an average LUT is computed from these specialized LUTs. The quality will be reported for each combination of input sequence and LUT.

Research question: how strongly does the image corpus used to compute a probability map influence the compression efficiency of BPC-PaCo?

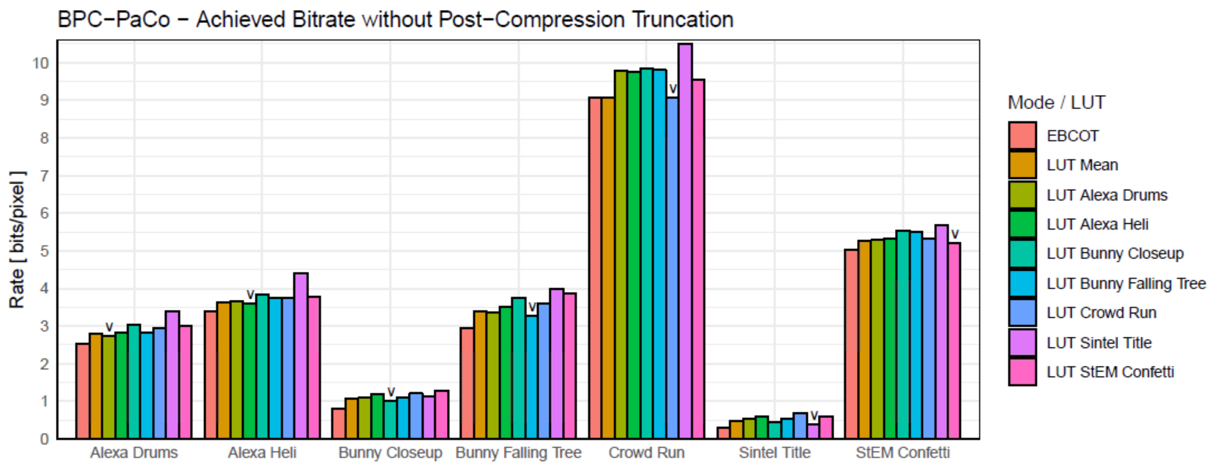


Figure 8-4 – Overview of achieved bit rate after compression (without any post-compression truncation) for every combination of input sequence and probability look-up table (LUT) for BPC-PaCo with two passes using a base quantization step size of 128^{-1}. The first two bars for each sequence show the results for EBCOT and the special LUT created by averaging all other LUTs. The sequence-specific LUT is marked with a small arrow.

Before comparing the resulting quality at a given target bit rate it is insightful to check the achievable compression rate without providing any limit on the bit rate budget. Figure 8-4 shows the results for the combination of all seven test sequences with the probability estimates computed individually for each of those sequences as well as the mean probability map over all sequences. Additionally, the results for EBCOT are included to serve as a reference. Figure 8-5 visualizes the same data by plotting the delta from the BPC-PaCo runs to EBCOT.

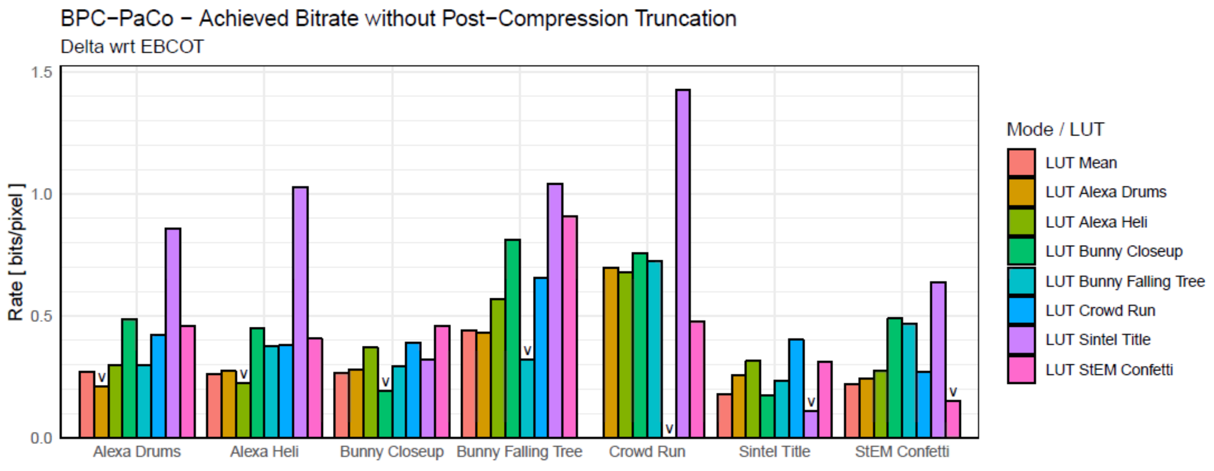


Figure 8-5 – Difference of achieved bit rate after compression (without any post-compression truncation) for every combination of input sequence and probability look-up table (LUT) from BPC-PaCo with two passes to EBCOT. The values show how many more bpp BPC-PaCo requires. The first bars for each sequence show the results for the special LUT created by averaging all other LUTs. The sequence-specific LUT is marked with a small arrow. The base quantization step size is 128^{-1}.

EBCOT yields the lowest bit rate across all test sequences, followed by the probability LUT created specifically for the respective input sequence. The mean LUT also performs well, scoring the third place in most cases.

Noticeably, for the *Crowd Run* sequence, the sequence-specific LUT and average LUT both perform equally well as EBCOT. *Crowd Run* is the most demanding sequence where the likelihood for the most probably symbol, 0 or 1, approaches 0.5 early on, resulting in a poor compression rate of any arithmetic coder. The probability LUT originally created for the *Sintel title* sequence yields by far the worst performance for *Crowd Run* with 1.4 bpp above the best result of 9 bpp, which equals an extra 15%.

The LUT created from *Sintel Title*, performs worst for all other sequences. Oppositely, the LUT designed for *Crowd Run* yields the worst result for the *Sintel Title* sequence. The LUT averaged over all sequences performs well in general with a penalty below 0.3 bits/pixel in all cases.

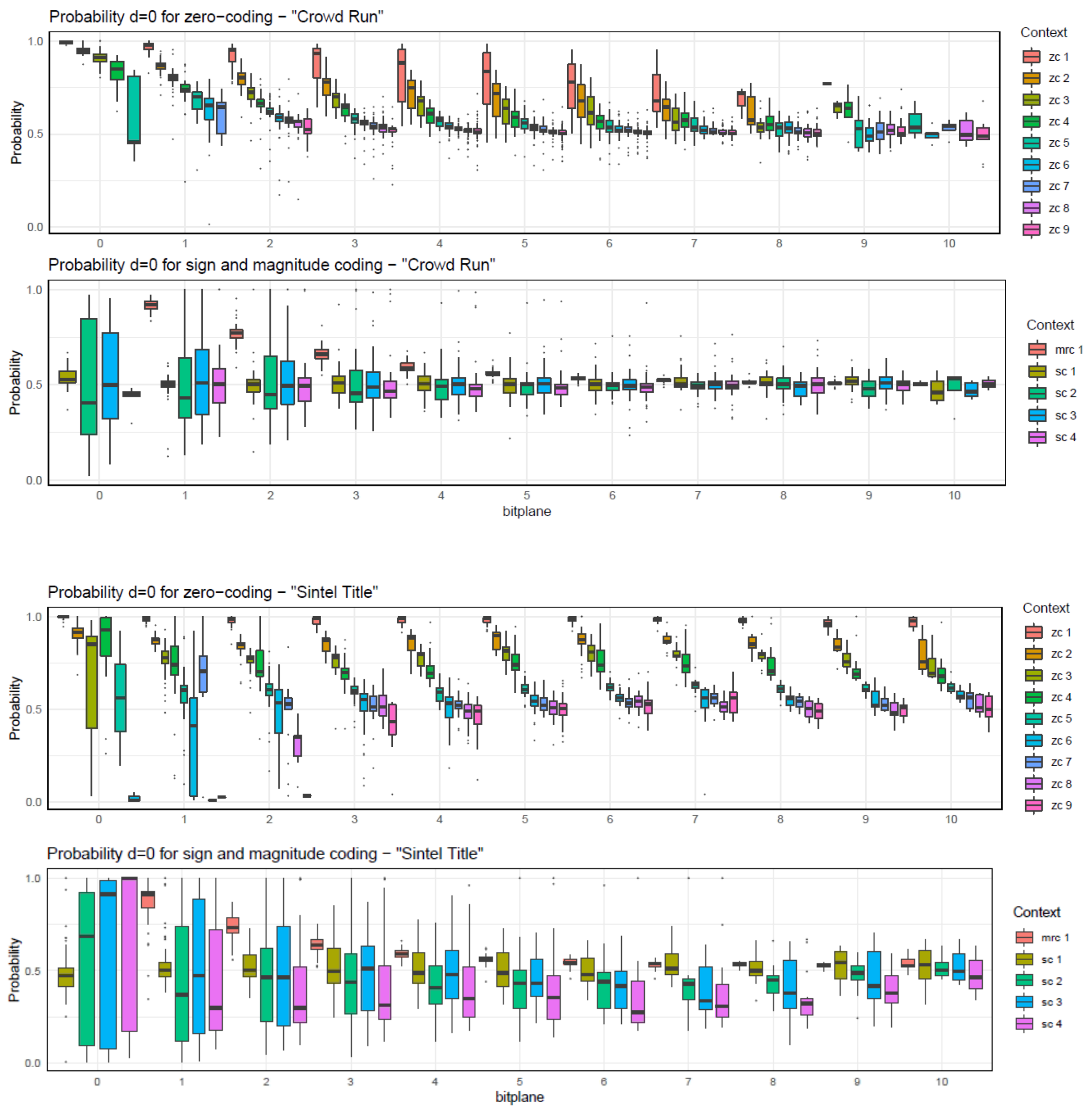


Figure 8-6 – Comparison of static probabilities, grouped by bit plane, context and coding operation, between the high entropy *Crowd Run* sequence (top two rows) and the low entropy *Sintel Title* sequence (bottom two rows). Intersections of bit plane, coding operation and context with less than 20 total symbols are omitted.

Figure 8-6 allows for comparing the probabilities of the two sequences, plotted over the bit planes from MSB to LSB and grouped by coding operation and context. The nine zero-coding contexts are grouped in one plot and the single magnitude-refinement-coding context is grouped with the four sign-coding contexts in a second plot. Each box represents the set of probabilities over all subbands in all channels. Comparing the zero-coding probabilities between *Crowd Run* and *Sintel Title*, it is visible that for the former, all the contexts approach a uniform probability towards the lower bit planes, whereas for the latter, the lower zero-coding contexts that indicate the presence of only few significant samples in the neighborhood remain skewed towards a probability of d=0. Another visible difference is that the context modeling routine for *Sintel Title* produces some zero-coding contexts where a symbol value of d=1 is more probable, especially for “zc 6” in the second bit plane and “zc 8” in the third bit plane. This is never the case for the *Crowd Run* sequence’s zero-coding contexts. The magnitude-refinement-coding probability distribution looks very similar between the two sequences. The sign-coding contexts, however, expose significantly more heavily skewed probabilities for the *Sintel* sequence.

More important than comparing the achieved bit rates given an unlimited budget is to examine the rate-distortion relation. *Given a predefined bit rate budget, how does the quality, in terms of PSNR, compare amongst the different probability tables*?

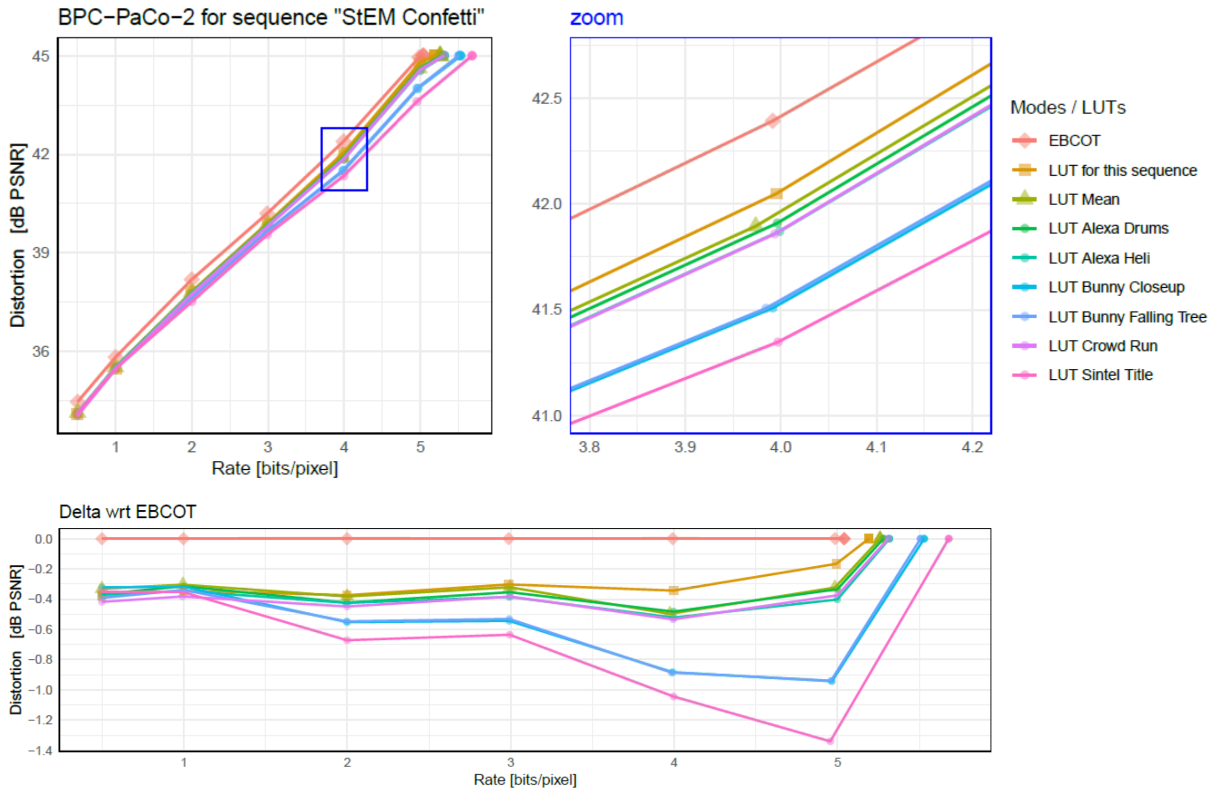


Figure 8-7 – Rate-distortion plot for BPC-PaCo with two passes per bit plane for every combination of input sequence and probability look-up table (LUT). The top right plot zooms in on the sample values at 4 bpp. The bottom plot shows the difference with respect to EBCOT. The base quantization step size is 128^{-1}.

Figure 8-7 shows a rate-distortion plot for the *StEM Confetti* sequence. EBCOT performs best across the entire range of sample bit rates from 0.5 to 6 bpp, followed by the probability map specifically tailored to the tested sequence. The variance grows with the bit rate. At low bit rates below 1 bpp it hardly matters which probability map is employed. At 2 bpp and 3 bpp the gap between the best and worst LUT stays reasonable at 0.4 dB. At 5 bpp, the gap has grown to over 1 dB.

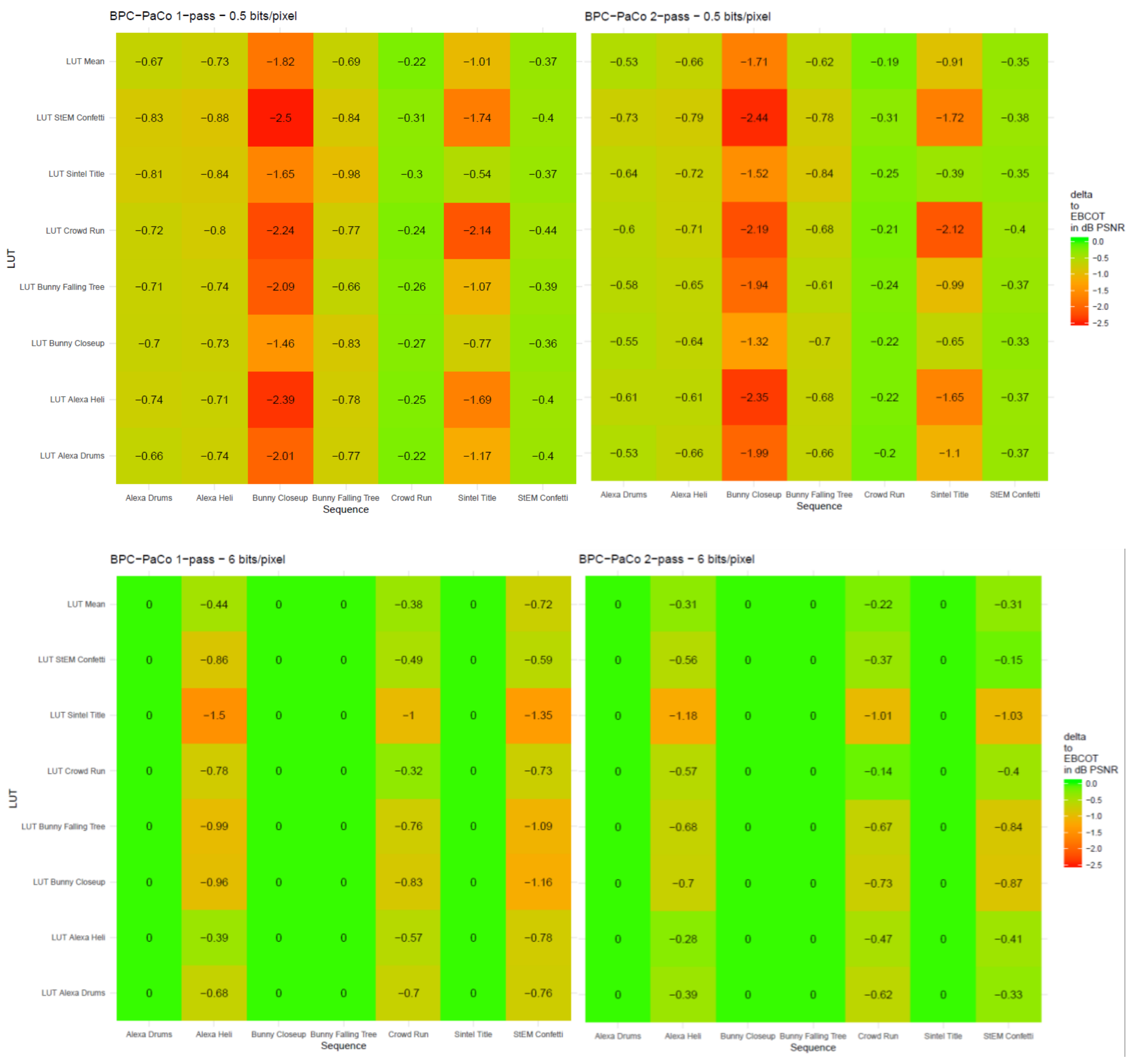


Figure 8-8 – Matrix with distortions (difference to EBCOT) for each combination of sequence (x-axis) and probability LUT (y-axis). **Top**: 0.5 bpp. **Bottom**: 6 bpp. **Left**: BPC-PaCo 1-pass mode. **Right**: BPC-PaCo 2-pass mode. The base quantization step size is 256^{-1}.

Figure 8-8 shows for the lowest (0.5 bpp, top) and highest (6 bpp, bottom) bit rate the PSNR differences to EBCOT for each combination of sequence (x-axis) and probability LUT (y-axis). At 0.5 dB, the difference with respect to EBCOT is considerable, especially for the low entropy sequences *Bunny Closeup* and *Sintel Title*. At 6 bpp, the changes remain visible only for the high entropy sequences. Using a probability LUT optimized for the *Sintel title* sequence does not work well for the *Alexa Heli*, *Crowd Run* or *StEM Confetti* sequences. The mean LUT constructed by averaging over all test sequences performs well across all four scenarios. The difference between the results of BPC PaCo 2-pass and 1-pass is marginal and lies within 0.1 dB for the average probability LUT, which lies well below the just noticeable difference (JND) according to the author's own subjective assessment.

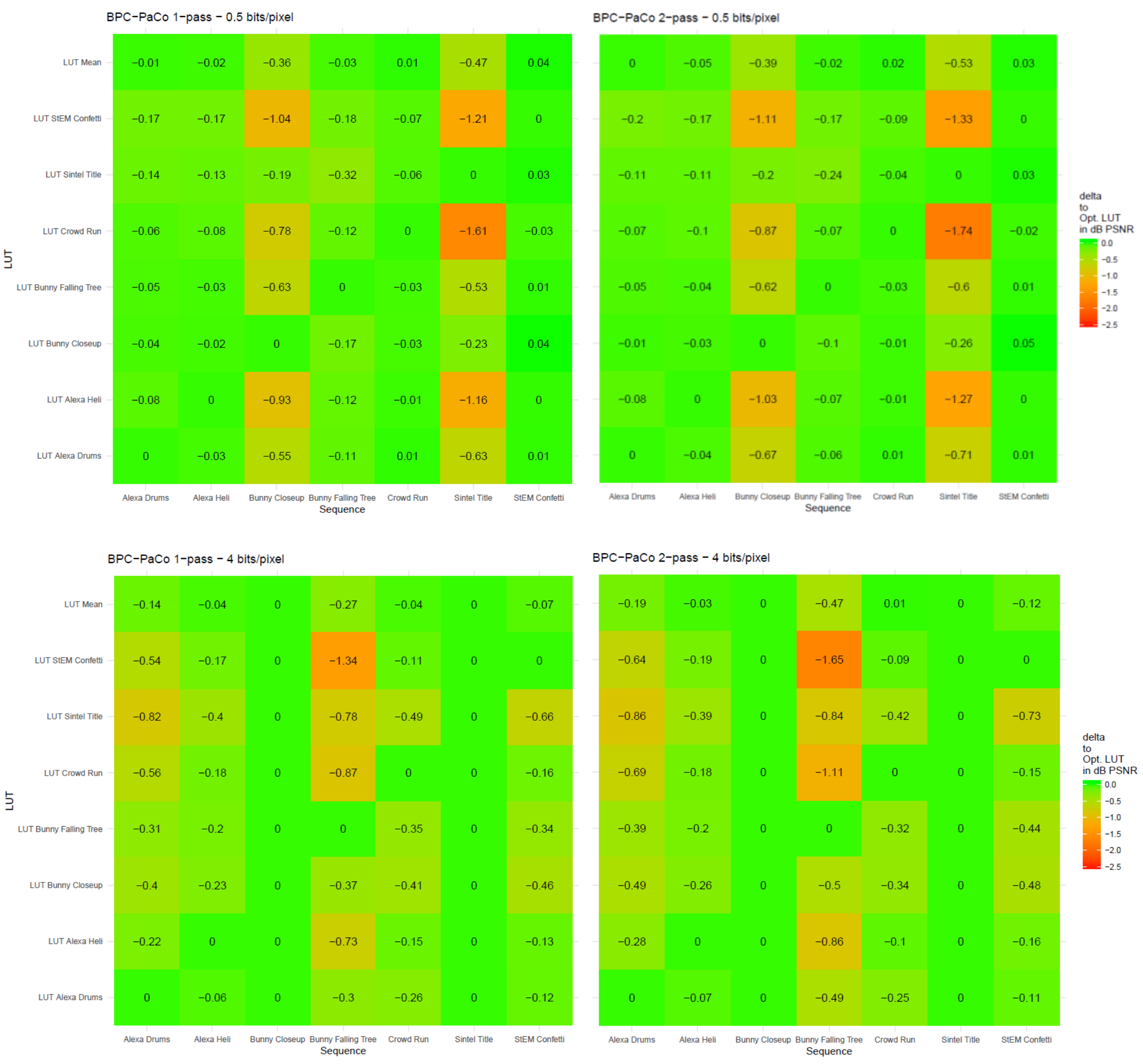


Figure 8-9 – Matrix with distortions (difference to optimal probability LUT created for the sequence) for each combination of sequence (x-axis) and probability LUT (y-axis). **Top**: 0.5 bpp. **Bottom**: 4 bpp. **Left**: BPC-PaCo 1-pass mode. **Right**: BPC-PaCo 2-pass mode. The base quantization step size is 256^{-1}.

Figure 8-9 shows again a matrix of PSNR differences, this time for 0.5 bpp and 4 bpp and with respect to the optimal probability LUT that was created from the test sequence itself. In this visualization, the intra-LUT variance can be read more conveniently.

8.7 Experimental Results

The most recent of a sequence of publications about BPC-PaCo is [53], where the authors publish results from a GPU implementation. They experiment with scan orders with two or three passes per bit plane and compare the resulting quality to standard JPEG 2000 and the runtime performance to the open source *cuj2k* [42] and *GPU JPEG2K* implementations [43].

Here, the scan order with two passes per bit plane will be compared against default JPEG 2000, denoted as “EBCOT”, since only the block coder is different, as well as the proposed BPC-PaCo flavor with a single-pass scan order described in chapter 8.4. Even though only the block coders are exchanged, all reported throughputs relate to the entire JPEG 2000 encoding and decoding routine, including memory transfers (unless otherwise specified). All block coding kernels operate collectively on the set of code blocks from a group of eight pictures, since a single image does not contain a sufficient amount of code blocks to fully load a high-end GPU. EBCOT uses code blocks of size 32x32, whereas the BPC-PaCo kernels use sizes of 64x64. Going forward, the probability map created by averaging over all test sequences is employed for experiments.

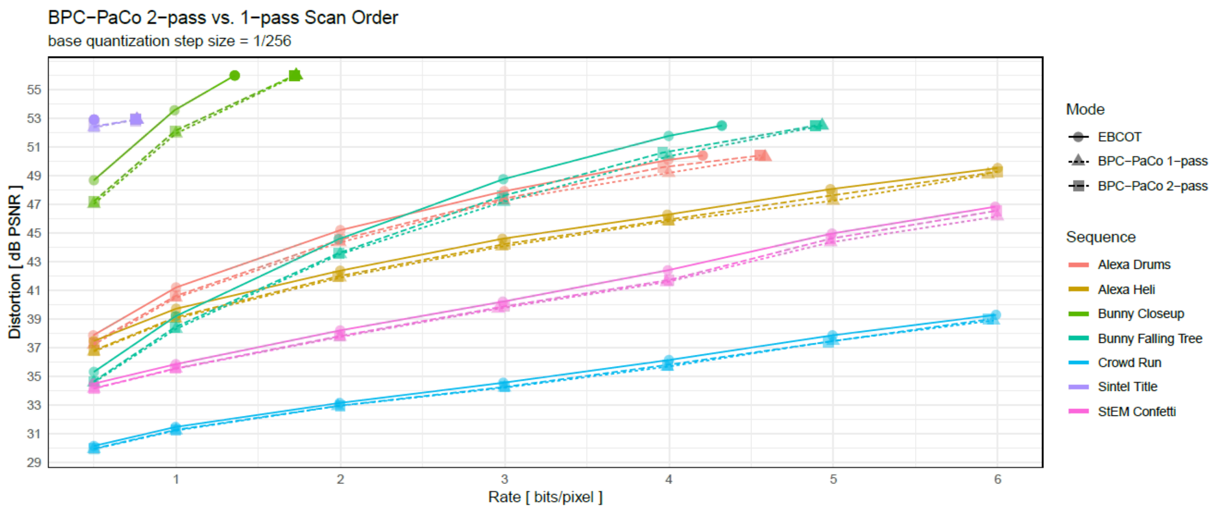


Figure 8-10 – Rate-Distortion plot for BPC-PaCo and EBCOT over all test sequences. The mean LUT created by averaging probabilities over all test sequences is used.

Since the proposed changes to BPC-PaCo trade intra-bit plane truncation points for higher speed, the first check is to compare the distortion. In the rate-distortion plot in Figure 8-10, the difference between the two BPC-PaCo flavors is hardly visible. The proposed changes cost no more than 0.3 dB compared to the original algorithm. The gap to EBCOT is slightly larger, but mainly stays below 1 dB. This is in line with the measurements reported by the original authors in [53]. The *Bunny Closeup* sequence poses an exception. The fact that the curves saturate already at a bit rate of ~1.5 bpp shows that the Full HD sequence has only few details and is very easy to compress. The gap between the original BPC-PaCo mode and EBCOT is 1.5 dB. The other scene from the *Big Buck Bunny* movie denoted as *Bunny Falling Tree* also shows differences of over 1 dB.

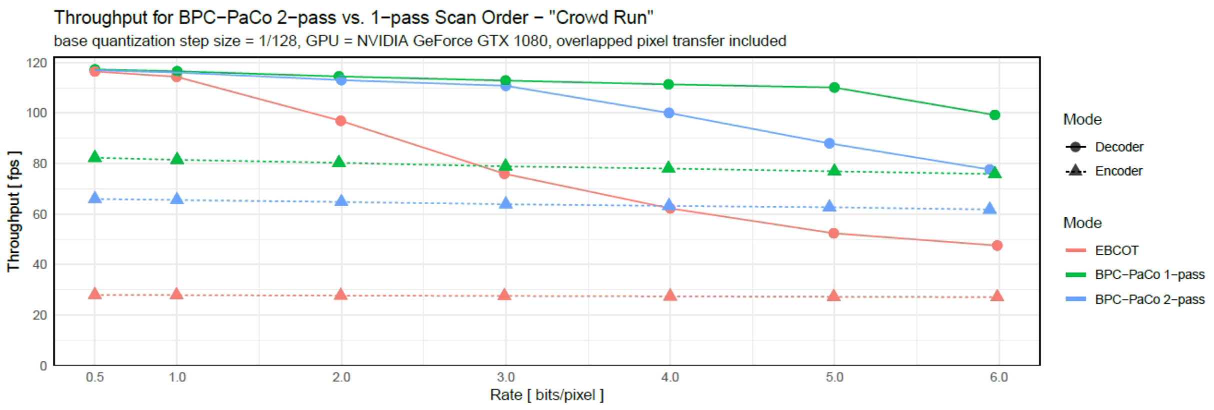


Figure 8-11 – Encoding and Decoding throughput for the *Crowd Run* test sequence, measured on a *NVIDIA GeForce GTX 1080* graphics card. The base quantization step size is 128^{-1}. *Crowd Run* sequence: 3840x2160, RGB 4:4:4, 48 bit per pixel, 8.3 MPixel, 24.9 MSamples.

The other key metric beside the quality is the achieved throughput. Figure 8-11 plots for the *Crowd Run* 4K sequence how fast the encoder and decoder operate on a *NVIDIA GeForce GTX 1080* GPU (released in 2016). The encoder throughputs are represented by the dashed lines, the decoder throughputs by solid lines. The order of the three modes is consistent between the encoder and decoder.

For the encoder the throughput is hardly affected by the target bit rate as indicated by the flat dashed lines. This is expected. For both the encoder and decoder, the component and wavelet transform coding stages are not affected by the bit rate in any case. The most time consuming block coding stage is only affected by the image content and quantization on the encoding side. The target bit rate is only met by truncating the bit streams for every code block after the block coding stage has finished. Here, no heuristic is employed to anticipate preemptively which passes or bit planes will end up getting truncated anyway and can already be skipped by the block encoder. The base quantization step size chosen for this experiment is 128^{-1}. EBCOT is slowest with 30 fps. The original BPC-PaCo with two passes per bit plane increases the encoding throughput by a factor of 2.15x to ~65 fps. The modified BPC-PaCo without any intra-bit plane truncation points increases the throughput by another 15 fps to ~80 fps in total, which equals a factor of 1.23x.

The decoding throughput, on the other hand, is heavily influenced by the bit rate as this now directly relates to the workload of the time consuming block decoding stage. All three curves are capped at close to 120 fps, because downloading the decoded images from the GPU poses a bottleneck. The transfer of the previous group of pictures is overlapped with the next group's compute kernels, but in this case downloads take longer than the decoding kernels. As discussed in chapter 2.3.4, this limitation will be mitigated when using newer hardware and based on this projection. Figure 8-12 shows a simulation that assumes a sufficiently high transfer bandwidth. In the above scenario, the GPU is connected on a PCI-Express 2.0 (standard released in 2007) bus with 16 lanes. This yields a theoretic transfer rate of 8000 MB/sec without protocol overhead. The measurements show a true transfer rate (including protocol overhead) from device memory into non-pageable host memory of ~6400 MB/sec. The transfer of a single frame of 48 MB size (3840 x 2160 samples x 3 channels x 2 Bytes per sample) takes ~7.5 ms

(133 fps). Given that the *GeForce* GPU does not overlap two transfers of opposite directions, the ~1 ms it takes to upload the compressed code stream has to be added to the 7.5 ms/frame download time, yielding the observed maximum throughput of a little under 120 fps. In scenarios where it is not required to decode the full bit depth of 16 bit per sample, but instead only RGB24 pixels with 8 bit per sample need to be reconstructed, the download size halves and the bottleneck posed by the transfer rises to 230 fps.

The decoding throughput for EBCOT lies under the transfer bottleneck only at the lowest bit rate of 0.5 bpp. Immediately, the curve falls gradually, indicating that it is then bound by the computation speed. For BPC PaCo with two passes per bit plane, the throughput starts to get computation-bound at 3 bpp, for the single-pass extension at 5 bpp. At 5 bpp, the encoding throughput for the three modes lies at 60 fps, 88 fps and 110 fps. This corresponds to a speed-up of 1.46x for BPC-PaCo 2-pass over EBCOT and another 1.25x of BPC-PaCo 1-pass over BPC-PaCo 2-pass. The following figure shows the same plot with the limiting data transfer in the decoder excluded. The benefits of the BPC-PaCo modes in terms of increased throughputs stay approximately constant throughout the entire sampled bit rate range.

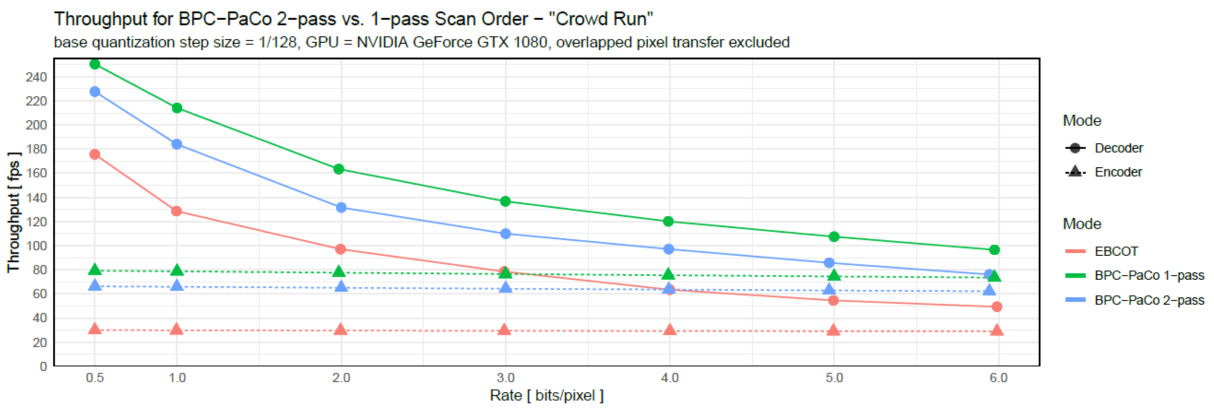


Figure 8-12 – Encoding and Decoding throughput for the *Crowd Run* test sequence. Same as Figure 8-11, except that in the decoder, the time for downloading the reconstructed images is not included.

The following figure shows the analog throughput measurements for the remaining sequences. The encoder and decoder throughputs have now been separated into individual plots and the three coding modes are distinguished by their line types in addition to the shapes.

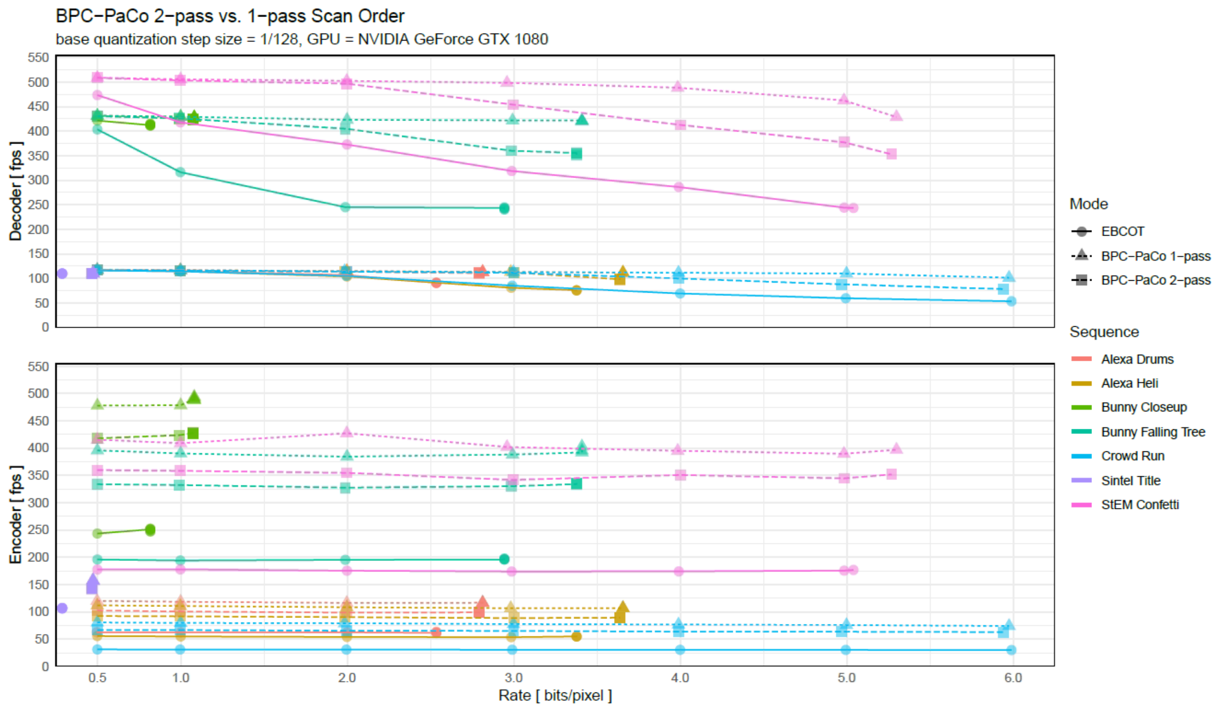


Figure 8-13 – Encoding (bottom) and decoding (top) throughput for all test sequences coded with (1) EBCOT, (2) the original BPC-PaCo with two passes per bit plane and (3) a modified BPC-PaCo with a single-pass scan order. Throughput is measured with a *NVIDIA GeForce GTX 1080*. The base quantization step size is 128^{-1}.

The trends observed and discussed above for the *Crowd Run* sequence apply to all other sequences as well (Figure 8-13). For the animated *Bunny Closeup* sequence with Full HD resolution, the encoding throughput is raised from 250 fps to ~425 fps and ~475 fps using the BPC-PaCo 2-pass and 1-pass modes, respectively. The throughput for the more complex *Bunny Falling Tree* scene from the same movie lies ~50 fps lower for EBCOT and ~75fps lower for the BPC-PaCo modes, confirming again that the complexity of the image content has a strong influence on the encoding speed.

8.8 Conclusion

How does BPC-PaCo compare to the other coding modes presented in this work? What are the drawbacks and advantages in terms of compression efficiency and throughput? The drop-in replacement for EBCOT proposed by F. Auli-Llinas et al. is compared against the author's implementation of standard JPEG 2000. Additionally, an alternative single-pass scan order, titled BPC-PaCo 1-pass, is proposed. It sacrifices intra-bit plane truncation points for increased throughput by reducing the memory accesses and allowing for employing a stenciling technique for computing the context from a sample neighborhood's significance states.

How strongly is the quality of BPC-PaCo mode affected by the quality of the static probability model? To this end, an experiment was conducted where custom-tailored probability look-up tables for each of the seven test sequences were created and applied to each test sequence. Figure 8-9 visualizes the results for each combination of test sequence and probability map. The maximum difference is 1.7 dB PSNR, when applying the LUT created from the detail-rich *Crowd Run* sequence to the low-detail *Sintel Title* sequence. The probability table averaged over all test sequences performs reasonably well, though, with a maximum difference of 0.53 dB (for *Sintel title*).

The proposed alternative scan order that sacrifices intra-bit plane truncation point lies constantly within 0.3 dB of the original mode for all test sequences over all sampled bit rates.

While the "cost" in terms of a diminished image quality was shown to be limited, both for BPC-PaCo over EBCOT and for BPC-PaCo 1-pass over BPC PaCo 2-pass, the next research question was: *What are advantages in terms of an increased throughput?*

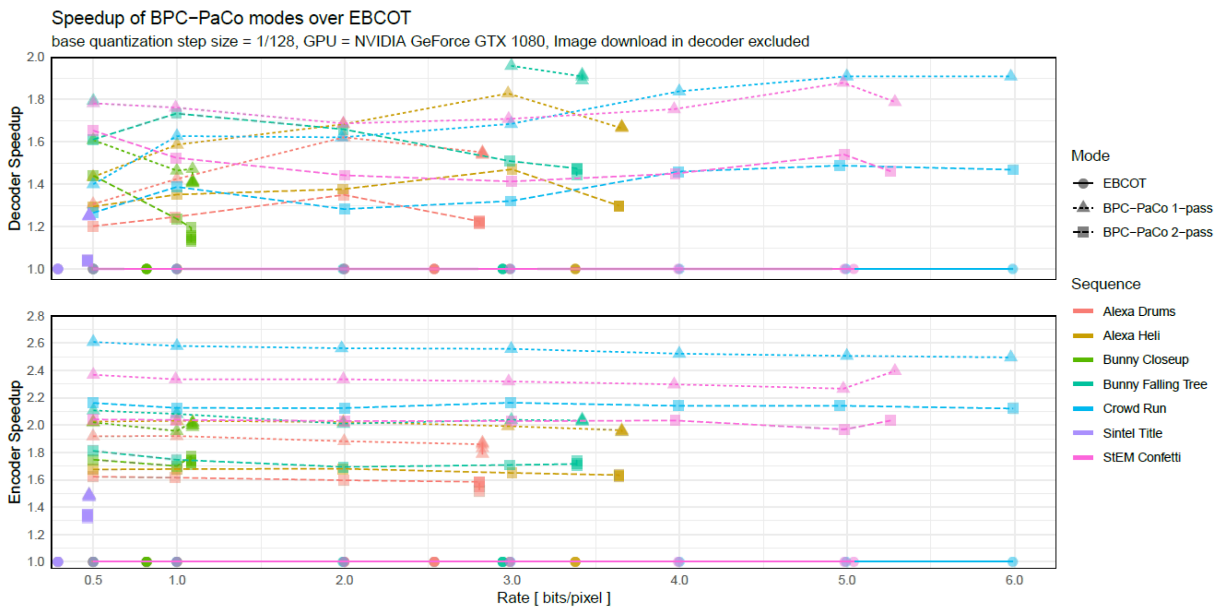


Figure 8-14 – Speedup for encoding (bottom) and decoding (top) throughput of BPC-PaCo modes over EBCOT for all test sequences. Throughput is measured with a *NVIDIA GeForce GTX 1080*. The base quantization step size is 128^{-1}. A speedup of 1x implies identical throughputs.

Figure 8-14 shows the achieved relative speed-ups over JPEG 2000 with the EBCOT block coder. For the encoder, the speed-ups remain constant regardless of the bit rate. They are only affected by the image sequence, more precisely by their complexity rather than their resolution. For the *Crowd Run* 4K sequence, the BPC-PaCo modes' speed-ups over EBCOT are 2.15x and 2.6x, while for the *StEM Confetti* 2K sequence with 2.35:1 aspect ratio they are 2x and 2.3x, respectively. For the remaining sequences, except *Sintel Title*, they lie between 1.6x-1.8x for the original two-pass mode and 1.9x-2.1x for the proposed single-pass mode.

On the decoding side, the impact on the throughput rises slightly in most cases towards the higher bit rates. For *Crowd Run*, the speedups are 1.25x and 1.4x at 0.5 bpp and they rise to 1.5x and 1.9x at 5 bpp. The reason is that the average number of passes per code block increases as the bit rate budget is increased. Since the regular EBCOT decoder is implemented with a single kernel with one thread per code block (due to the feedback loop between context modeler and arithmetic coder, the two routines cannot be decoupled), increasingly more threads of each thread group stay inactive towards the lower bit planes as they finished processing their code block’s passes. BPC-PaCo does not suffer from this problem as each thread-group collectively decodes a single code block.

The measurements carried out were, in case of the decoder, limited by the available memory bandwidth. However, as shown in chapter 2.3.4, the available bandwidth rises fast as newer versions of the PCI express protocol are released. This coding mode will continue to benefit from faster computation speed provided by future GPU hardware improvements and not be limited by slow transfer rates.

9 Runtime Model

This chapter seeks to analyze and model the performance of a kernel. Predicting the runtime of a kernel is very challenging due to the complexity of the hierarchical memory and concurrency architecture of a modern GPU. The authors of [85] examined computation-bound kernels with the goal of reverse-engineering the behavior of the CUDA scheduler in certain scenarios when using more than one CUDA stream. [86] and [87] each summarize multiple existing GPU models for predicting the runtime of a kernel in order to combine them. They predict the runtime of sample kernels. The use cases involve relatively small kernels such as a matrix multiplication or a histogram, where it is possible to count the compute and memory instructions in order to calculate the number of instructions.

Since much of this work is concerned with the JPEG 2000 context modeling and arithmetic coding kernels, these are used as examples. In contrast to the small use cases examined in the literature listed above, these kernels' number of instructions cannot be predicted ahead of time as they depend heavily on the statistics of the input images. An image with a high entropy will produce more symbols to be encoded than a low-entropy image. On top of that, the kernels are very long which renders an instruction count impracticable. Instead of predicting the runtime merely based on the source code, an approach is taken where the kernels are executed and profiled on one GPU in order to be able to gain an understanding of the performance limitations and predict the runtime on another GPU.

9.1 CUDA Benchmark-based Runtime Model

This paragraph discusses how the runtime t^{krnl} for a given kernel on a given device can be calculated based on a set of contributing metrics. One such key metric is the activity level of the multiprocessors. The multiprocessor activity is the percentage of the kernel execution duration a multiprocessor is active [88]. Ideally, the multiprocessor activity lies close to 100%. It is lower, for instance, when the workload is balanced unevenly across all SMs or when the problem size is too low.

A single multiprocessor executes multiple thread blocks concurrently. However, blocks resident on the same multiprocessor have to share the multiprocessor's physical resources, and so the number of blocks that can be resident on a single multiprocessor depends on a thread block's resource consumption. The following design choices are made either explicitly or implicitly:

1. Number of SIMD-groups per thread block
2. Number of registers per thread
3. Shared memory usage per thread block

At least one of the above will be the limiting factor. The number of thread blocks resident per multiprocessor weighted by the multiprocessor activity yields the amount of active thread blocks per multiprocessor:

$$\frac{ThreadBlocks^{active}}{Multiprocessor} = \frac{ThreadBlocks}{Multiprocessor} * MultiprocessorActivity \tag{1}$$

The average number of active SIMD-groups per multiprocessor follows:

$$\frac{SIMDGroups^{active}}{Multiprocessor} = \frac{ThreadBlocks^{active}}{Multiprocessor} * \frac{SIMDGroups}{ThreadBlock} \tag{2}$$

The average number of cycles per SIMD-group is calculated from the workload (instructions per SIMD-group) and the efficiency at which the instructions could be executed (instructions per cycle). Both metrics are obtained by running the instruction efficiency benchmark on the profiled kernel. The amount of required cycles per SIMD-group already includes any latencies introduced by stalls.

$$\frac{Cycles}{SIMDGroup} = \frac{\frac{Instructions}{SIMDGroup}}{\frac{Instructions}{Cycle}} \tag{3}$$

Once the number of cycles per SIMD-group is known, the required number of back-to-back thread-block executions *R* can be calculated:

$$R = \frac{ThreadBlocks}{\frac{ThreadBlocks^{active}}{Multiprocessor} * Multiprocessors} \tag{4}$$

To give an example, if the problem were divided into 100 thread blocks, each multiprocessor could execute two thread blocks at a time and the GPU comprised five multiprocessors, then each multiprocessor would repeat ten rounds (*R=10*).

The total amount of cycles for running the entire kernels computes as follows:

$$\frac{Cycles}{Kernel} = \frac{Cycles}{SIMDGroup} * \frac{SIMDGroups^{active}}{Multiprocessors} * R \tag{5}$$

Finally, the kernel’s runtime can be calculated from the required number of cycles and the GPU’s clock rate:

$$t^{krnl} = \frac{\frac{Cycles}{Kernel}}{ClockRate} \tag{6}$$

The following table repeats an overview of the parameters used in the above formulas and groups them into GPU-, kernel- and benchmark-specific parameters.

Table 9-1 - Parameters required for runtime model

Parameter	Source	Description
$\frac{ThreadBlocks}{Multiprocessor}$	Combination of kernel design and GPU hardware	Calculated by dividing the resources (registers, shared memory) a multiprocessor offers by the amount of resources a single thread block requires.
$\frac{SIMDGroups}{ThreadBlock}$	Kernel design	The developer chooses the number of SIMD-groups (CUDA: 32 threads each) per thread block
$\#\,ThreadBlocks$	Problem size	The developer chooses how to split the problem into thread blocks
$\#\,Multiprocessor$	GPU hardware	High-end GPUs have more multiprocessors than low-end models
$Clock\ Rate$	GPU hardware	Often the same for all GPU models in a series
$Multiprocessor\ Activity$	Benchmark	Percentage a multiprocessor is active during the kernel runtime
$\frac{Instructions}{SIMDGroup}$	Benchmark	The number of instructions per SIMD-group indicates the workload. Ideally it is constant across all multiprocessors or else the workload is misbalanced
$\frac{Instructions}{Cycle}$	Benchmark	The number of instructions per clock indicates the efficiency. This parameter includes latencies induced by memory-accesses as well as execution dependencies and barrier-induced stalls.

9.2 Case Study A: *Context Modeling Kernel*

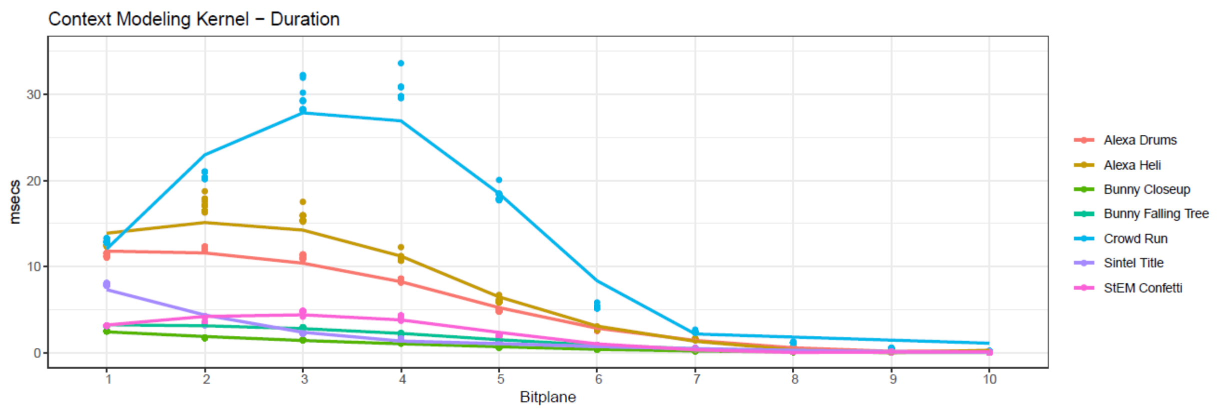


Figure 9-1 – Duration of context modeling kernel per bit plane (8 frames per kernel)

The runtime in milliseconds initially grows and then falls again with a peak at the 2nd to 4th bit plane. One key parameter that affects the runtime is the number of code blocks with passes pending to be computed. As a reminder, the number of passes differs even among all code blocks due to the subband-specific bit depths used to store quantization indices. Additionally, initial insignificant magnitude bit planes[4] are not processed and instead signaled as side-information (in the packet header) to the decoder. Therefore, many code blocks, especially from high-frequency subbands, will only contain two or three bit planes that need to be encoded while code blocks from lower frequencies might have many more passes to be computed. With every bit plane, the number of code blocks with remaining work shrinks.

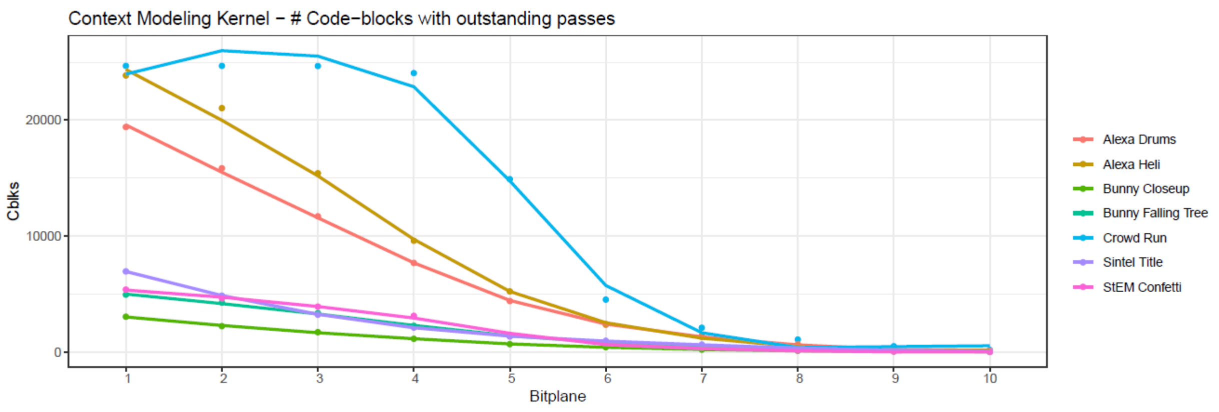


Figure 9-2 – Number of code blocks with outstanding passes, grouped by bit plane and sequence

The next plot shows the throughput over each bit plane normalized by the amount of code blocks. The first bit plane needs to be examined separately as here a thread is spawned for all code blocks, even for those that have no single non-zero magnitude bit plane. The reason is that the number of all-zero bit planes is not calculated beforehand, but as part of the first iteration of this kernel. Hence, there might be a difference between the amount of code blocks for which threads were spawned and the amount of code blocks that "participate" in this bit plane, i.e. have outstanding passes. This plot uses the amount of code

[4] all magnitude bits in the bit plane are zero

blocks with outstanding work for the first bit plane. In all subsequent bit planes, both values are always equal.

From the 2nd to the 10th bit plane, a trend is visible. The low-entropy 4K *Sintel* sequence has the highest throughput (10 MCblks/s) and the high-entropy 2K *StEM* and UHD *Crowd Run* sequences have the lowest throughput (6 MCblks/s). Clearly, another factor besides the number of active code blocks has an impact on the speed or else all sequences would have roughly the same throughput. The values beyond the 10th bit plane are no longer accurate, because the absolute run times are already very low – below 0.1 ms - at that point.

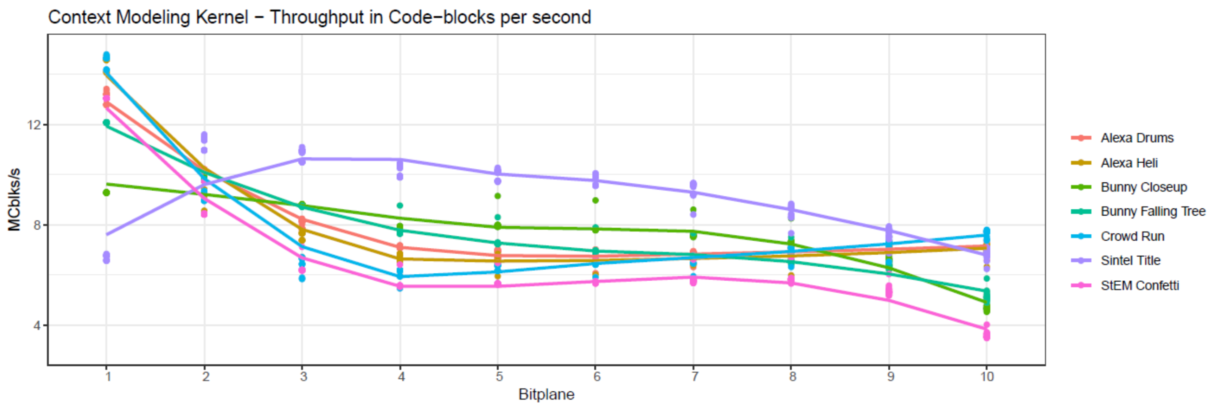


Figure 9-3 – Throughput of context modeling kernel normalized by the amount of code blocks with outstanding passes

One way to examine the workload per code block is to plot the amount of context-decisions pairs (symbols) that are coded on average per code block. For a code block size of 32x32 there are 1024 samples per code block. When a sample's first 1-bit is coded, the sign bit is interleaved as well, increasing the number of symbols per bit plane. More importantly, when a stripe column of four vertically adjacent samples as well as their neighborhood are all still insignificant (zero magnitude in decoder's view), they are run-length-coded, usually[5] decreasing the number of symbols per bit plane.

The number of symbols per code block starts low and then rises steeply throughout the first three or four bit planes. The explanation is that for the majority of the code blocks, run-length coding can be used frequently in the first bit planes. The high-entropy sequences exceed the threshold of one symbol per sample at the third or fourth bit plane. This is the point where many samples turn significant and their sign bits are interleaved.

[5] When the run is interrupted already at the 1st position and all symbols are 1 so that their signs are interleaved as well, a single run can produce as many as 12 symbols (worst case)

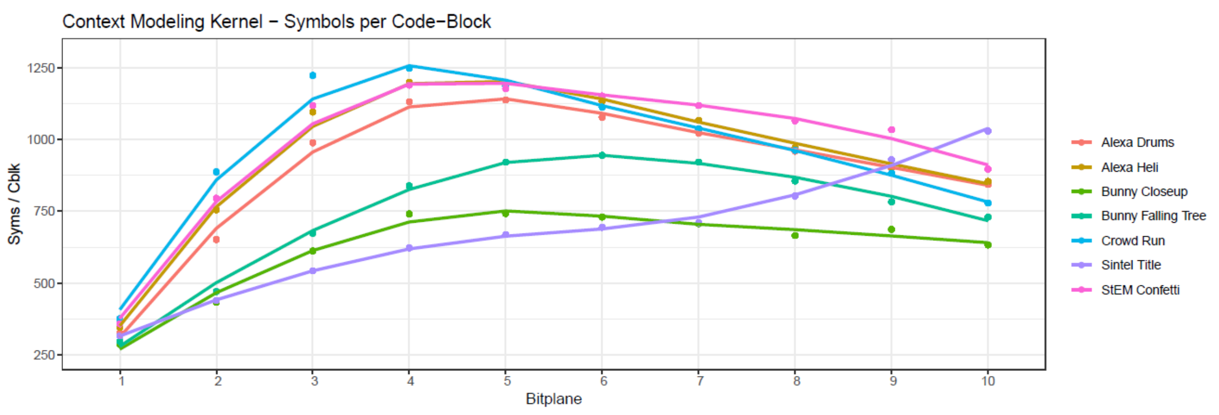


Figure 9-4 – Amount of produced symbols per code block, grouped by bit plane and sequence

Another way to examine the workload per code block is to plot the global memory accesses. The next plot shows only the read-accesses, normalized by the number of code blocks with unfinished passes.

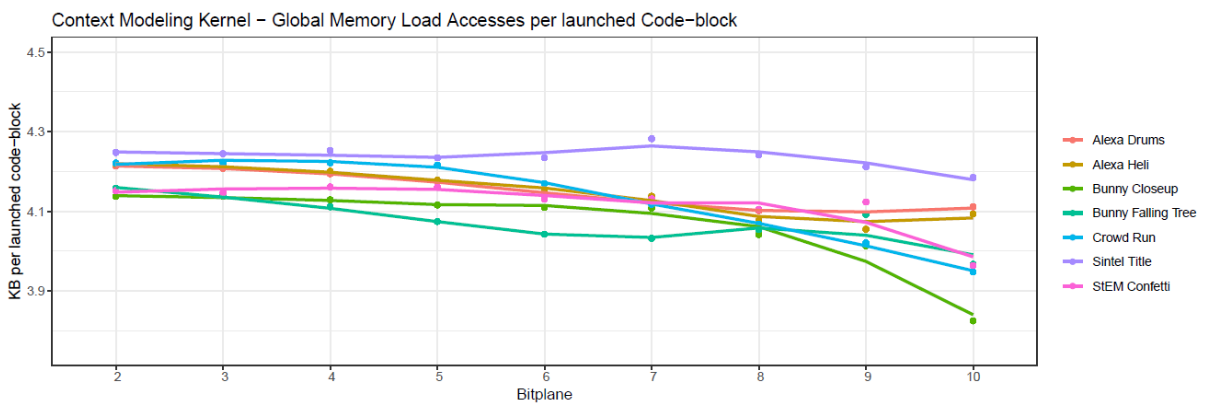


Figure 9-5 – Amount of memory loaded from global memory per code block, grouped by bit plane and sequence

The amount of loaded global memory normalized per code blocks is roughly identical for all sequences. The kernel needs to load a bit more than 4 KB per code block, which equals 4 byte per sample, on average. The state and magnitude of four vertically adjacent samples are stored in a 16 bit mask. However, since a code block is processed in parallel by 1024/4=256 threads (one thread per four samples), each thread needs to load additionally some metadata values from global memory as well. In any case, since the values are constant, the global memory read-accesses do not explain why some sequences are processed faster than others are. The number of context-decision pairs per code block was already shown to be an indicator and since they are written to global memory so that the MQ-kernel can subsequently read them from there, the global memory writes should not be constant:

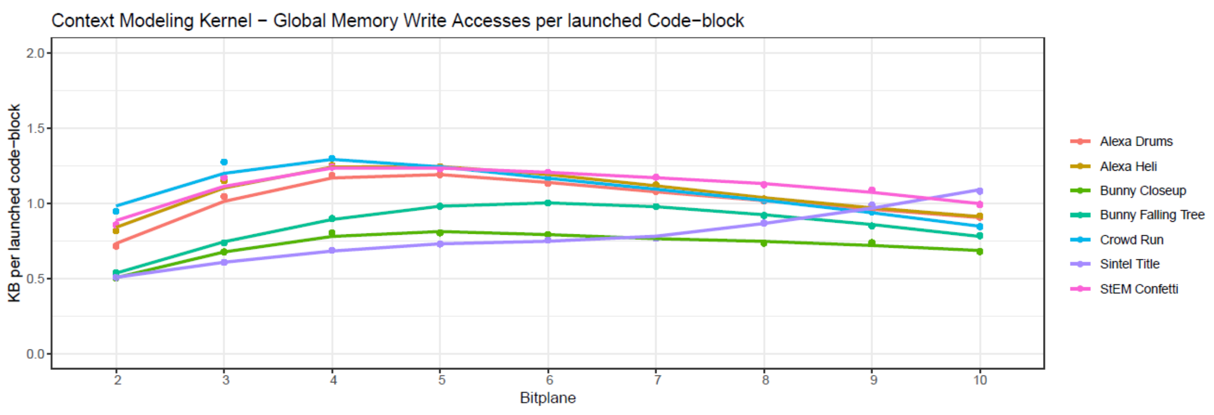


Figure 9-6 – Amount of memory stored to global memory per code block, grouped by bit plane and sequence

Indeed, the amount of stores is higher for the high-entropy sequences (*Alexa Drums, Alexa Heli, StEM Confetti, Crowd Run*) than for the low-entropy sequences (*Bunny Closeup, Bunny Falling Tree, Sintel Title*). The first bit plane again needs to be examined separately as it executes extra work. The data that is written in addition to the context-decision pairs (1 byte per pair) is

- (Fist bit plane only) number of initial all-zero bit planes and the number of passes (2 x 1 byte)
- Per code block, the number of symbols per pass (3 x 4 bytes)
- Whether the code block has no more outstanding passes (1 byte)
- The distortion change that is later required by the post-compression rate-distortion optimization routine. In the current implementation, each thread writes any change directly to global memory. The distortion is only updated when a 1-bit is coded. Distortion values are stored as 32 bit floating point values per pass (variable size).

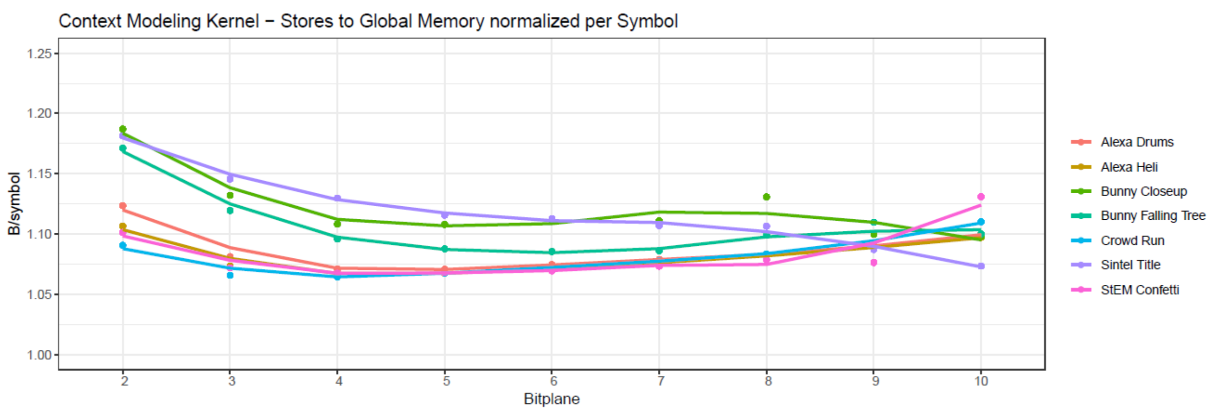


Figure 9-7 – Amount of memory stored to global memory per symbol, distinguished by bit plane and sequence

A metric more closely related to the measured runtime is the absolute amount of instructions, averaged over the amount of launched code blocks. The profiler does not report the amount of instructions directly, but only the averaged amount of instructions per SIMD-group, which can then be multiplied by the number of SIMD-groups:

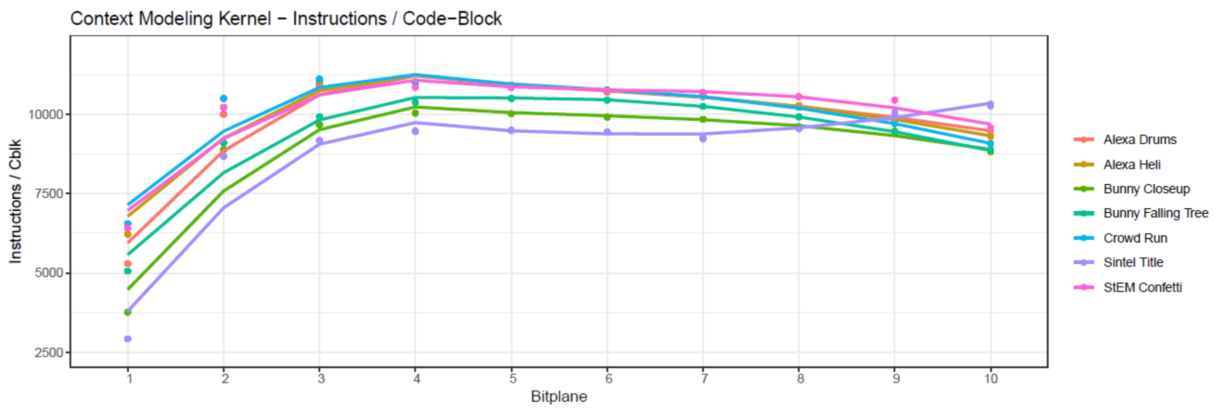


Figure 9-8 – Number of instructions per code block, grouped by bit plane and sequence

Indeed, the sequences where the number of instructions is lowest (*Sintel Title*) show the highest throughput in terms of processed code blocks/second (Figure 9-3). However, when plotting the instructions per cycle, it can be seen that the number of instructions alone is not sufficient to explain the run time.

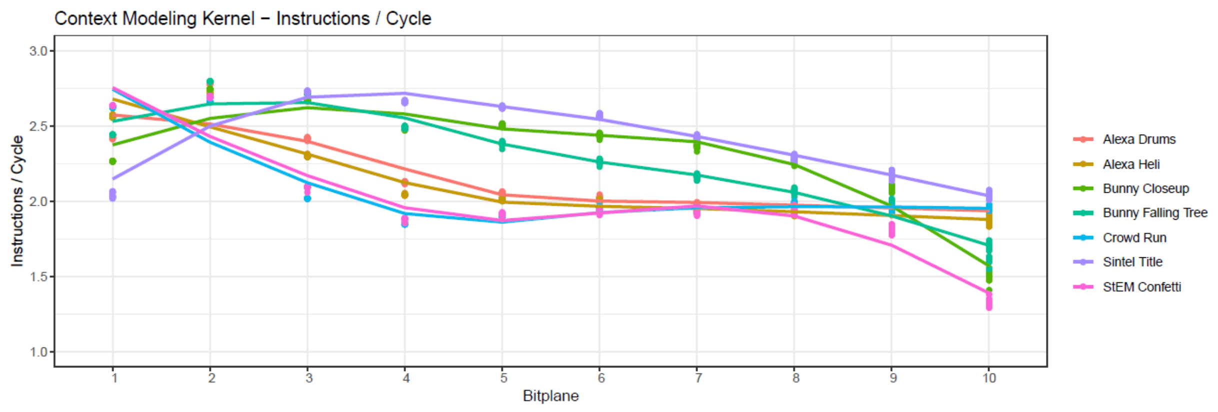


Figure 9-9 – Number of instructions per cycle, grouped by bit plane and sequence

Reasons for a low instructions-per-cycle metric are when the latencies introduced by memory accesses or execution inter-dependencies cannot be hidden. The next plot shows the percentage of clock cycles when one or more SIMD-groups were eligible for execution:

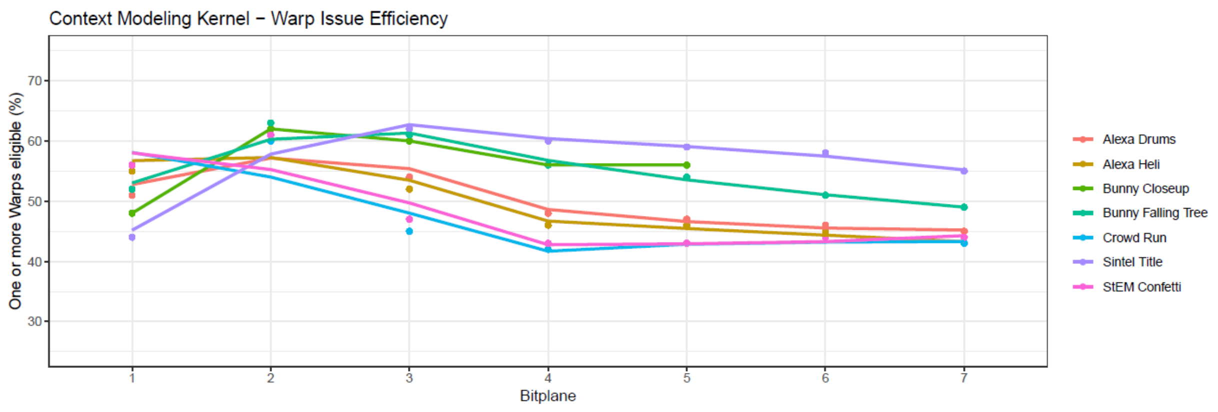


Figure 9-10 – percentage of warps eligible for execution at a given bit plane

From the third bit plane on, the efficiency is highest for sequences with fewer details and lowest for the detail-rich *Crowd Run* and *StEM Confetti* sequences.

Out of the most prominent stall reasons *Instruction Fetch, Execution Dependency, Memory Throttle, Memory Dependency, Texture, Synchronization, Constant Miss* and *Pipe Busy*, the three highest-ranking ones are plotted next:

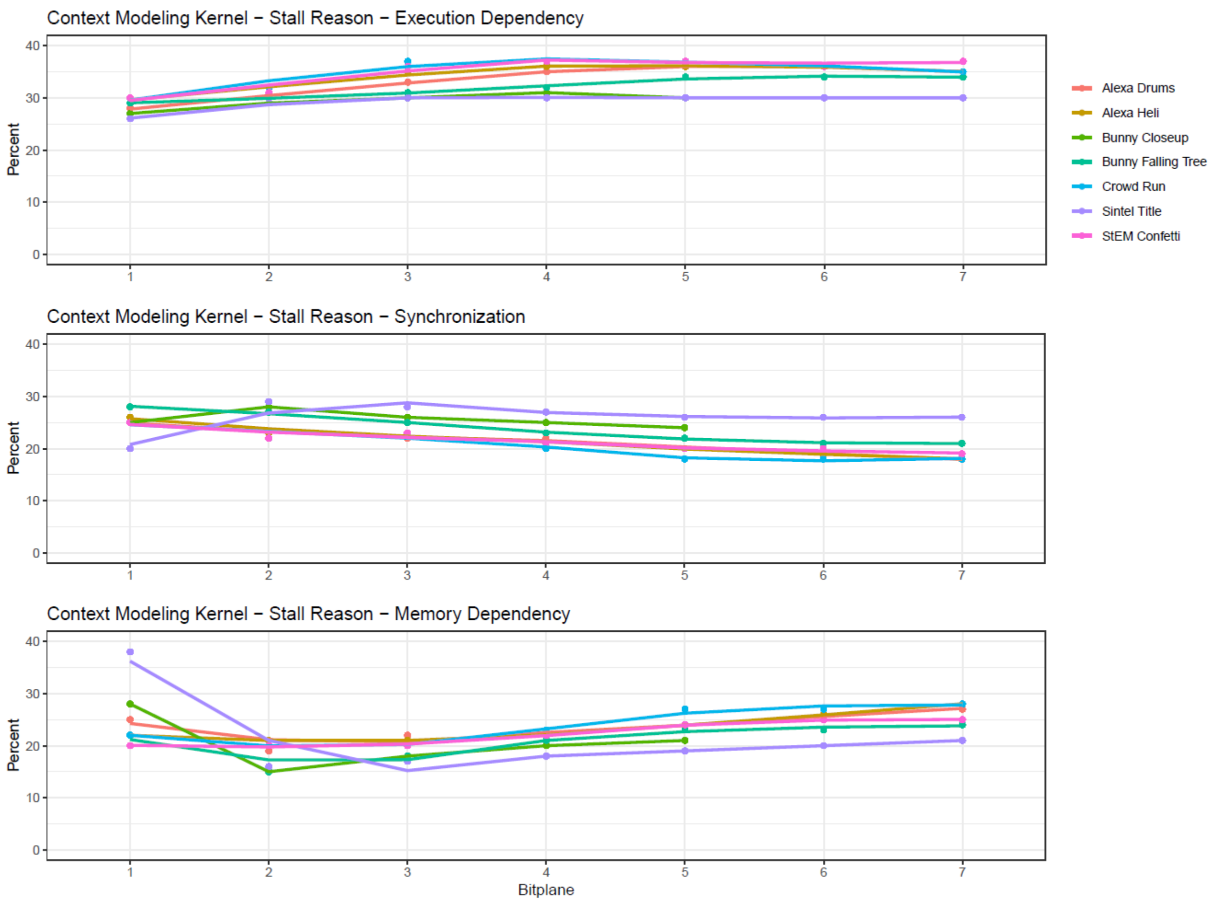


Figure 9-11 – Top 3 stall reasons in context-modeling-kernel

1. An *Execution Dependency* stall occurs when the input required by an instruction is not yet available.
2. A *Synchronization* stall occurs when a warp is blocked at a thread block barrier (__syncthreads()).
3. A *Memory Dependency* stall occurs when the program has to wait for a load or store operation to finish. Especially accesses to the global memory take very long.

For the high-entropy sequences, where the execution efficiency was measured to be lowest, the most frequent stall reason is an *Execution Dependency*. For the *Crowd Run* sequence, in the third bit plane, execution dependency stalls occur in the following spots

1. (~25%) Before significance propagation, where significance states, signs and the current magnitude bits from one code block stripe are rotated into three 32 bit integers (*preprocessBitplane()*)
2. (~25%) sign-coding
3. (~25%) in the section, where the context-decision pairs from a stripe column (up to ten) are sorted by pass type
4. (~20%) during magnitude-refinement coding and sign-coding when the distortion estimate is updated with an *atomicAdd()* call

The runtime model presented in the beginning of this chapter can now be fed with

- the benchmarking results
    - on the workload (instructions per SIMD-group)
    - and the kernel's efficiency (instructions per cycle)
- the GPU specifics (here stated for a *GeForce GTX 1080*)
    - 20 multiprocessors with 64 cores each
    - clock rate of 1.8 GHz
- and the kernel configuration
    - context modeling
        - 256 thread per thread-block = 8 SIMD-groups
        - 1 thread block per EBCOT block
        - At most 4 thread blocks fit into a multiprocessor (limited by registers)
    - MQ-encoding
        - 64 threads per thread-block = 2 SIMD-groups
        - 1 thread per EBCOT block
        - At most 12 thread blocks fit into a multiprocessor (limited by registers)
    - Operating on all code blocks from 8 consecutive frames at once

A limitation of this model is that the memory accesses are not explicitly modeled. As the scheduler attempts to hide memory access latencies by overlapping them with memory independent computations, it is very challenging to formulate the memory accesses explicitly and would require inside knowledge on the scheduling policies. Instead, the memory access performance is implicitly included in the efficiency variable stated in terms of achieved instructions / cycle. This explains why only the GPU's number of processors and their clock rate, but not its memory speed or bus widths can be considered.

The predicted kernel runtime is compared with the actually measured runtime in Figure 9-12 and shows a good accuracy.

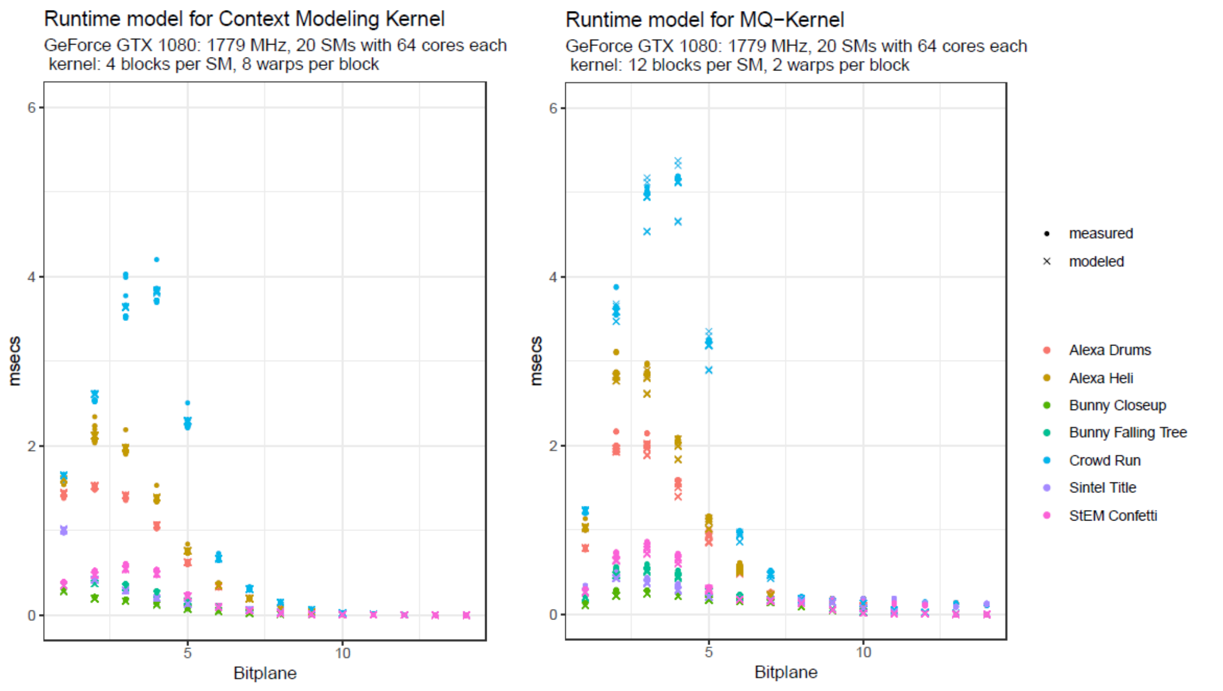


Figure 9-12 – predicted (‚x‘) vs. measured (‘*’) runtime of the context modeling (**left**) and arithmetic coding (**right**) kernels

The following charts show the results for three different GPUs. The *GeForce GTX 1080* and *GeForce GTX 1060* both stem from the *GeForce 10* series and as such, both use the *Pascal* CUDA architecture. The *GeForce GTX 1060* (6 GB) contains 1280 cores on 10 multiprocessors and the *GeForce GTX 1080* 2560 cores on 20 multiprocessors. With 1780 MHz, the 1080’s clock rate is about 6 % higher. Moreover, the *1080* has a memory bus width of 256 bit compared to 192 bit in the *1060*. The older *GeForce GTX 750 Ti* is from Q1 2014 and uses the *Maxwell* CUDA architecture. It contains 640 cores distributed across five multiprocessors, runs with a clock rate of 1135 MHz and has a bus width of 128 bit.

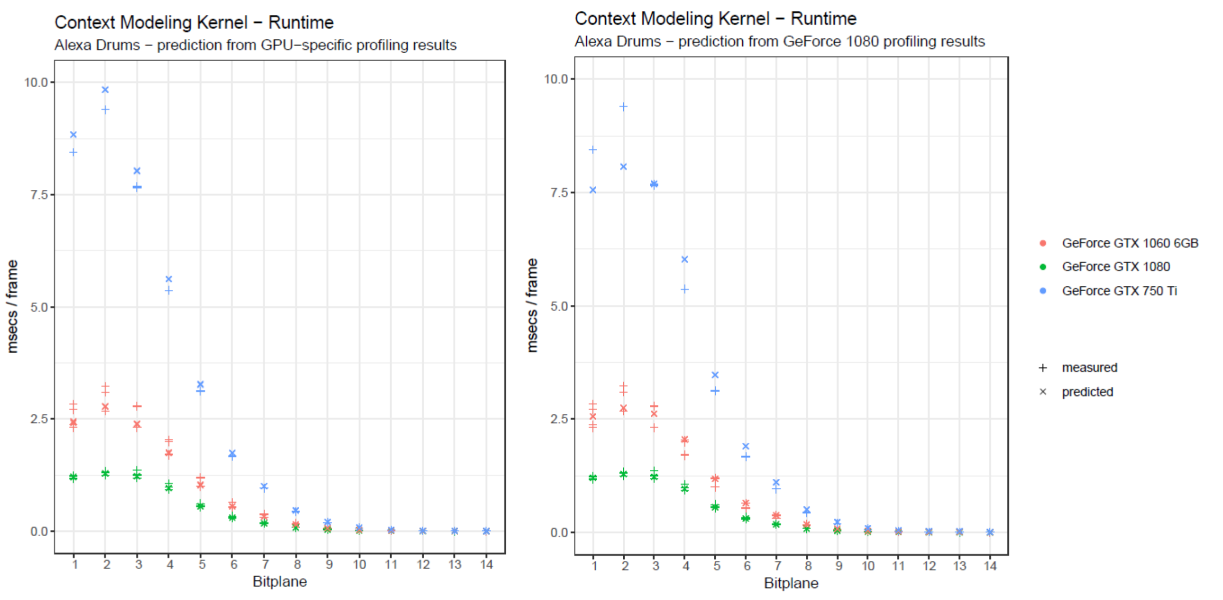


Figure 9-13 – predicted (‚x‘) vs. measured (‘*’) runtime of the context modeling-kernel for the Alexa Drums sequence on three different GPUs. **Left**: Runtimes are predicted using the respective GPU’s own profiling results. **Right**: runtimes are predicted based on the profiling results for the GeForce GTX 1080.

The *GeForce GTX 1080* deviates at most 0.1-0.2 ms and the other two devices at most 0.5 ms. The difference is highest in the most significant bit planes. In relative terms, the *GeForce GTX 1060* is a little less than 15% slower than predicted, the *GeForce GTX 750 Ti* is around 5% faster than predicted and the *GeForce GTX 1080* is up to 10% faster than predicted.

When using the benchmarking results from the *GeForce GTX 1080* experiments as a basis for predicting the runtimes on the other devices, the accuracy becomes worse, especially for the *GeForce GTX 750 Ti*. The first two bit planes are slower than predicted and the fourth and fifth bit planes are a little faster than predicted. The reason is that the instructions/cycle metrics yield different results for the devices (see Figure 9-14).

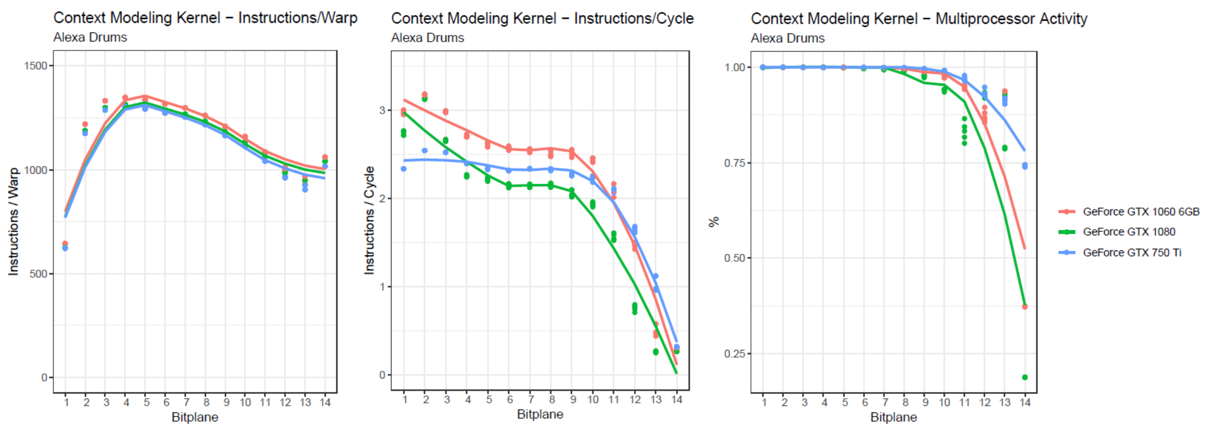


Figure 9-14 – Profiling metrics “Instructions/Warp” (**left**), “Instructions/Cycle” (**center**) and “Multiprocessor Activity” (**right**) for three GPUs

The table below shows the results for an entire frame when accumulating the measurements and predictions across all bit planes. The first three rows correspond to the results when feeding the benchmarking results from each particular GPU into the model. The bottom two rows show the results when using the metrics measured on one device (*GeForce GTX 1080*) and predicting the runtime on another device.

Table 9-2 – predicted runtime of context modeling kernel on three GPUs

Sequence	**Measured**		**Predicted**		**Error**
	GPU	msec	from GPU	msec	msec
Alexa Drums	GTX 1080	5.96	GTX 1080	5.86	-0.10
	GTX 1060 6GB	12.54	GTX 1060 6GB	11.52	-1.02
	GTX 750 Ti	37.40	GTX 750 Ti	39.15	+1,75
	GTX 1060 6GB	12.53	GTX 1080	12.50	-0.03
	GTX 750 Ti	37.40	GTX 1080	36.76	-0,64

Overall, the accuracy seems sufficiently accurate in order to make a close estimate on the forecasted runtime of the kernel on another device.

9.3 Case Study B: *Arithmetic Coding Kernel*

The arithmetic coding kernel's thread grid consists of blocks with 64 threads each, where each thread is assigned to a single code block. It has a much coarser parallelism than the context modeling kernel, where a thread is in charge of only four samples and an entire thread-block is assigned to a single code block.

Each thread loads a variable amount of context-decision pairs from global memory, where they were previously written to by the preceding context modeling-kernel. Each context-decision pair is stored in one byte where the LSB is the decision bit and the seven MSBs contain the context. The output is a compressed bit stream per code block, i.e. per thread.

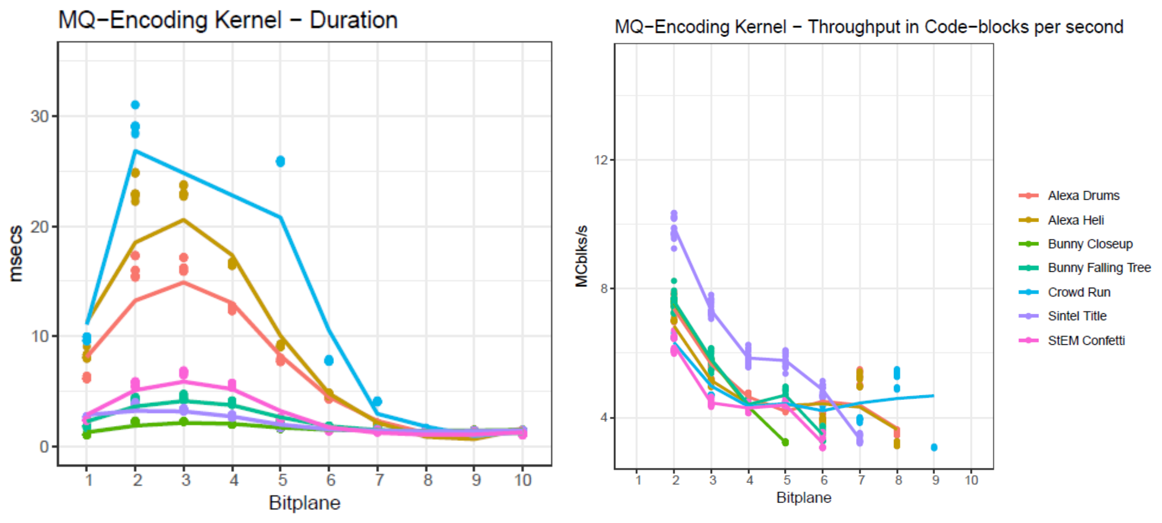


Figure 9-15 – Duration (**left**) and throughput in terms of code block / second (**right**) for the arithmetic encoding kernel operating on a group of 8 consecutive frames

The above two plots show the absolute runtime in milliseconds (left) and the throughput in terms of code blocks per second (right). A problem that becomes visible immediately is that the runtime at some point (bit plane 7-8) no longer decreases with the amount of active code blocks. Regardless how few code blocks remain, the kernel always take at least 2 to 3 milliseconds.

Since this kernel, in contrast to the context modeling kernel, suffers from low parallelism, an explanation could be that the device is under-occupied when only few code blocks, and therefore only few threads, are left. Indeed, a plot of the average activity of all multiprocessors supports this hypothesis:

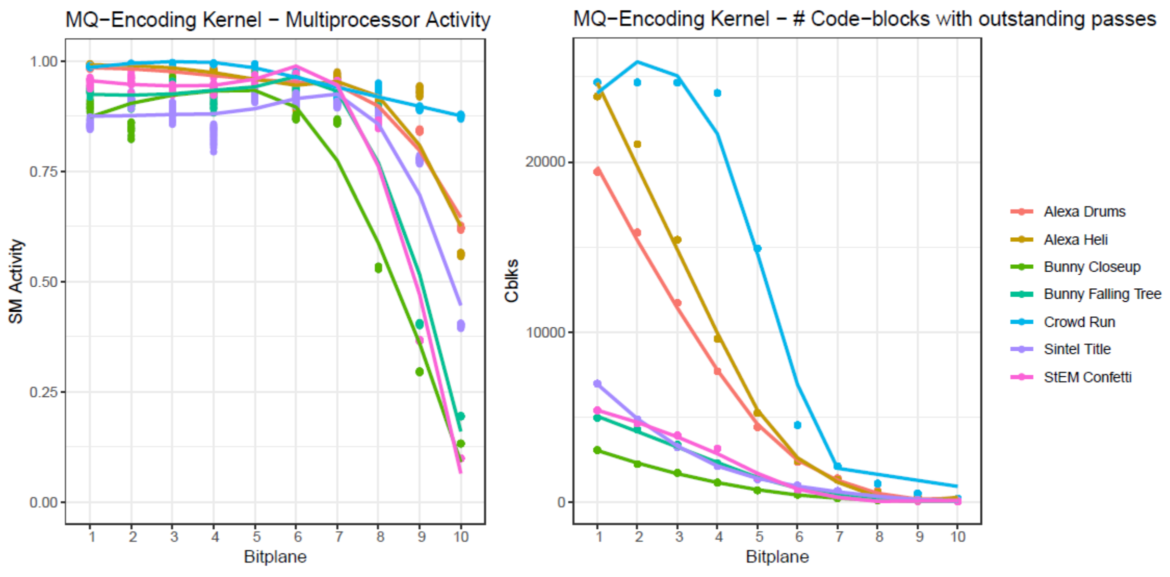
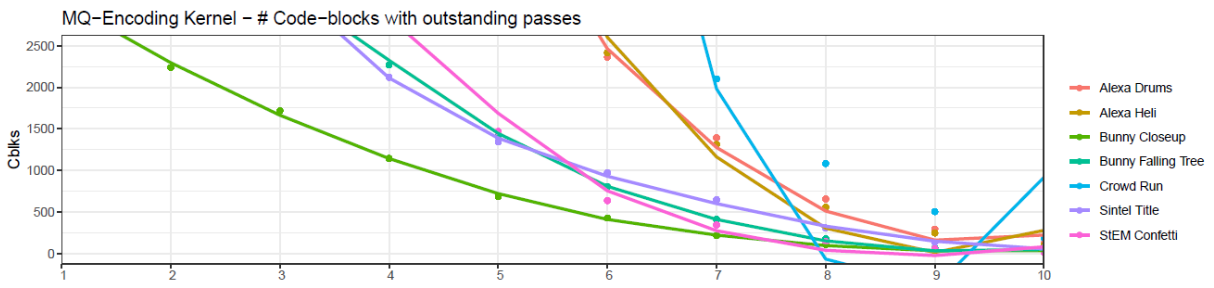


Figure 9-16 – Multiprocessor Activity (**left**) and amount of code blocks with outstanding passes (**right**) for the arithmetic encoding kernel operating on a group of 8 consecutive frames

Initially the multiprocessors stay occupied between 85% and 100% until at some point between the 7th and 10th bit plane the occupancy starts dropping significantly. For HD and 2K images, this point arrives earlier than for the UHD and 4K images, and for low-entropy sequences, it is earlier then for high-entropy sequences.

When zooming in on the critical ranges and arranging the plots so that they can be compared more easily, one can see that for this particular GPU (*GeForce GTX 1080*) the minimally required number of code blocks seems to be at around 500-1000 code blocks. The multiprocessor is considered active when at least one SIMD group is currently assigned for execution. Considering that the *GeForce GTX 1080* has 20 multiprocessors, at least 20x32 = 640 code blocks are required to be able to assign at least one SIMD-group to each multiprocessor. This is also the reason that there is a benefit to processing a group-of-pictures (GOP) in parallel as was done in these experiments. With the code blocks from only a single frame, the point where there are not enough active code blocks left to occupy the GPU would have arrived even earlier.



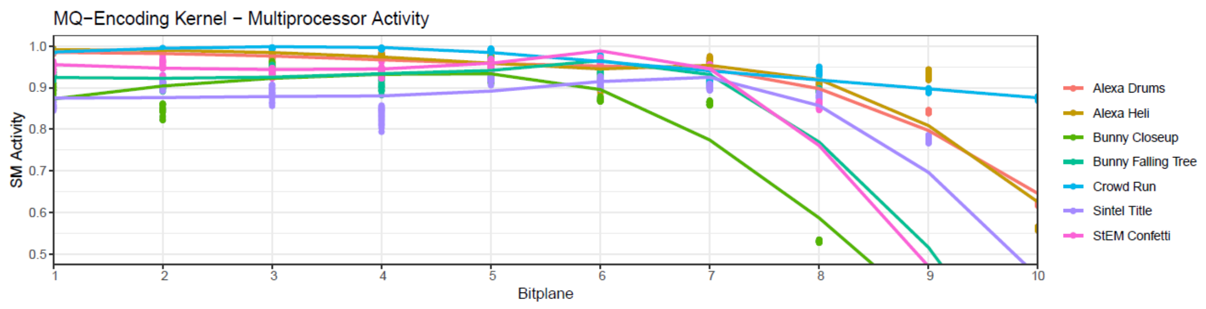


Figure 9-17 – Aligned top-and-bottom view of the amount of code blocks with outstanding passes (**top**) and the multiprocessor activity (**bottom**).

As indicated by the decreasing streaming multiprocessor activity shown in Figure 9-17 at higher x-axis values, the GPU is under-occupied when the lower bit planes are processed. Possible counter measures are:

1. One strategy could be to decrease the thread-block size so that more thread-blocks are created.
2. It might also be beneficial to use only a subset of the threads in one SIMD-group and leaving the other threads idle (thread under-subscription). This would alleviate the problem of thread divergence at the same time.
3. Thirdly, the workload could be more evenly distributed across the number of kernel invocations (load balancing).

Nevertheless, the majority of the time is spent in the first seven or more bit planes and the question remains why some sequences have a higher throughput than others do. The highest throughput in terms of code blocks per second is observed for the *Sintel Title* sequence, which is a 4K sequence with few details. On the contrary, the *Stem Confetti* sequence has the lowest throughput and it is a 2K sequence with many details.

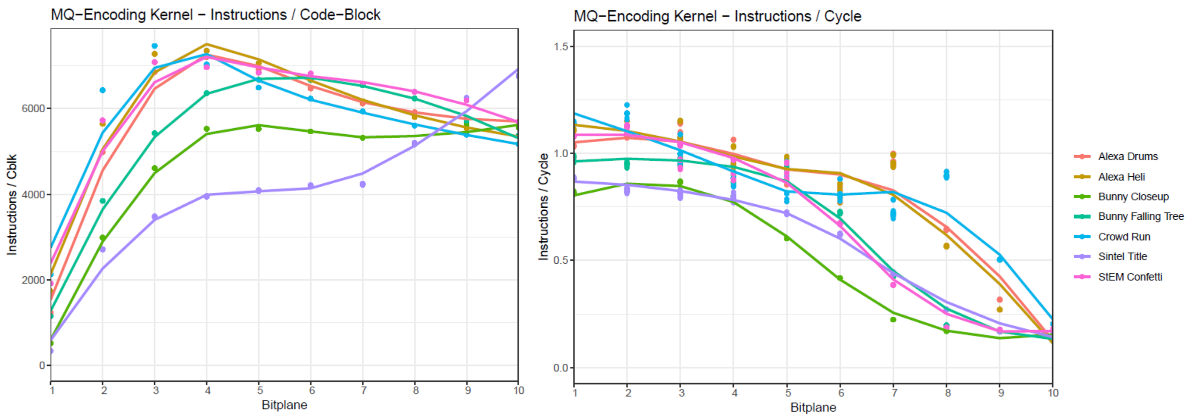


Figure 9-18 – The MQ-encoding kernel's workload (**left**) and execution efficiency (**right**).

The above two charts give information on the workload per code block (left) and execution efficiency (right). The workload seems to be considerably lower for the *Sintel Title* sequence, but also for the *Bunny*

Close-up sequence, both of which have comparatively low entropy. The execution efficiency starts at 0.75 - 1.25 instructions per cycle, which is less than half compared to the context modeling kernel that started at 2.25 – 2.75. Then it decreases for some sequences earlier than for others. As fewer active code blocks remain left, the scheduler has less flexibility to hide latencies by swapping in other thread-blocks.

As for the context modeling-kernel, the most important reason for the varying workloads is the amount of context-decision pairs per code block:

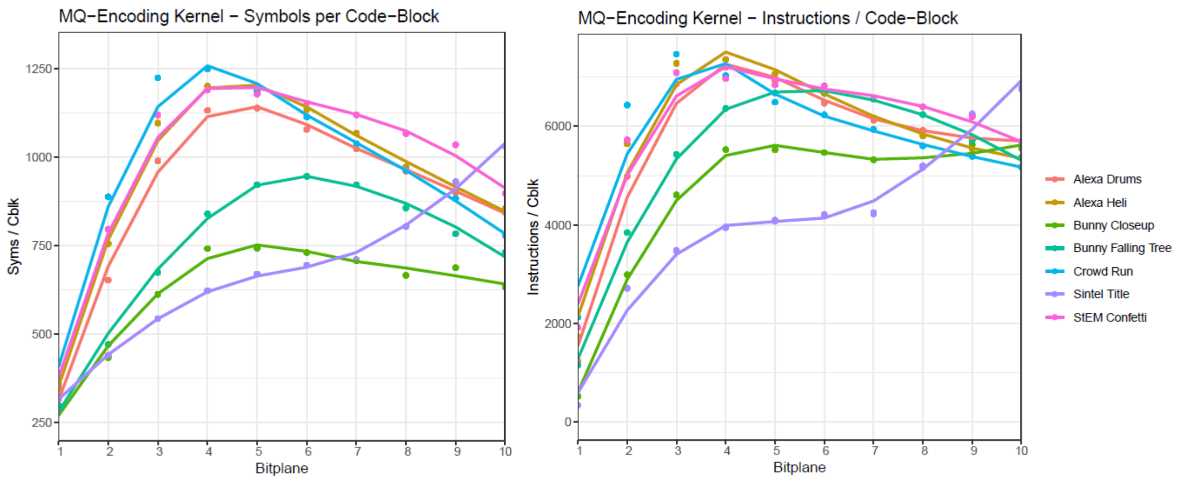


Figure 9-19 – the number of context-decision pairs to be processed by the MQ-encoding kernel (**left**) affects the instructions per code block (**right**).

The global memory accesses per context-decision pair are roughly the same for all sequences:

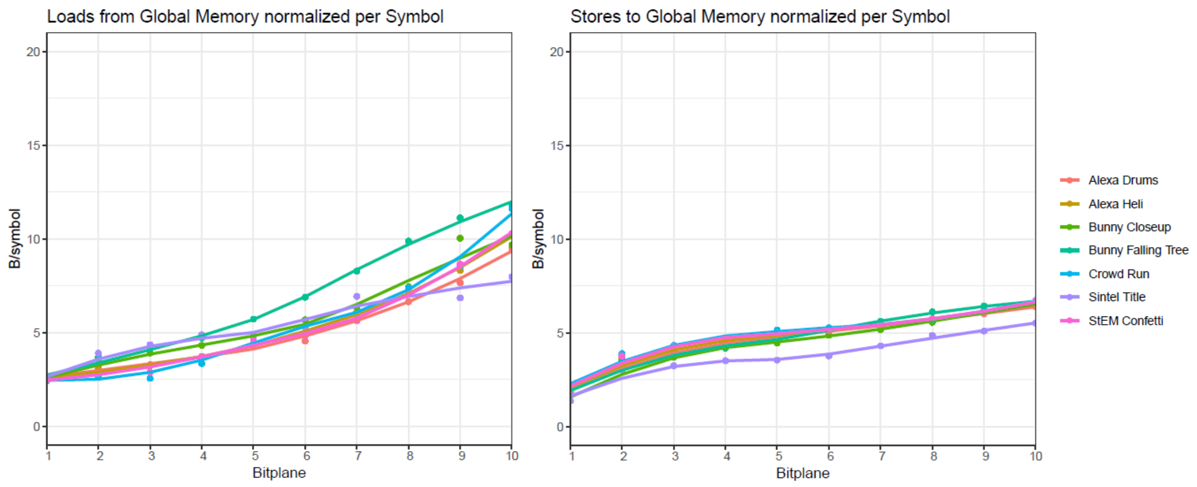


Figure 9-20 – the number of Bytes loaded (**left**) and stored (**right**) per symbol processed by the arithmetic encoding kernel.

The loads per symbol do not stay constant, because each thread additionally needs to load other inputs from global memory, such as the MQ-encoder’s state and code block metadata. Given that the average number of context-decision pairs per code block decreases towards the LSB (with exception of *Sintel Title*) the impact of the overhead becomes higher and skews the curves. The same logic also applies for explaining why the number of memory write accesses increases towards the MSB. A problem is that the amount of bytes stored is not significantly lower than the amount of data loaded, given that the kernel should compress the data. An explanation is that the kernel stores side information in addition to the output bit stream. Due to the problem that a single thread is in charge of an entire code block, each thread manages their individual MQ-encoder instance. With 64 threads per thread-block the amount of registers available to a thread block do not suffice to store all 64 MQ-coders and so they spill to global memory. The register file size has stayed constant at 64 KB in all CUDA architectures since version 3.0 (with one exception in compute capability 3.7 used only in the Tesla K80) and so this limitation will likely not go away.

Figure 9-19 (a) shows that the ratio of cycles when at least one SIMD-group is eligible for execution is only around 20%, which is significantly lower than for the context modeling-kernel, where it is between 40% and 60%.

The three predominant reasons why a SIMD-group is not eligible for execution in a cycle are *Memory Dependency*, *Execution Dependency* and *Instruction Fetch* (Figure 9-19 b-d).

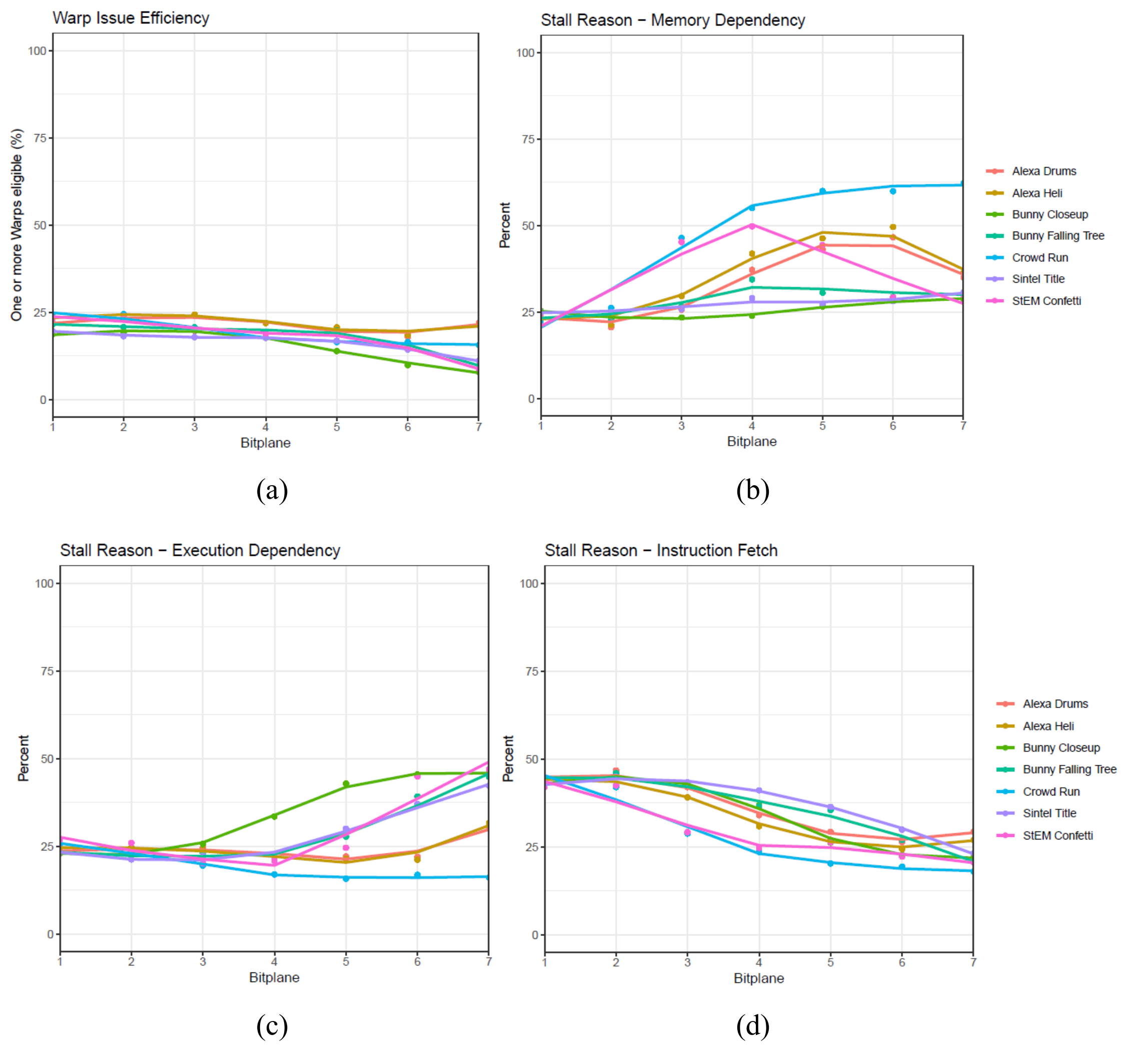


Figure 9-21 – Warp issue efficiency (**a**) and the top three stall reasons: memory dependency (**b**), execution dependency (**c**) and instruction fetch (**d**) for arithmetic encoding kernel.

In the most significant bit planes, *instruction fetch* is the predominant stall reason. It is a form of latency and occurs for instance when the body of a loop is too large to fit into a multiprocessor's instruction cache. Heavy branching is another reason that causes instruction cache misses. Beginning with the fourth bit plane, execution and memory dependency become the more common reasons.

Below, the predicted runtimes are plotted. Compared to the context modeling-kernel, the prediction error introduced by employing the benchmarking metrics measured for the *GeForce GTX 1080* for predicting the runtimes on a *GeForce GTX 750 Ti* are larger (see Figure 9-22 right).

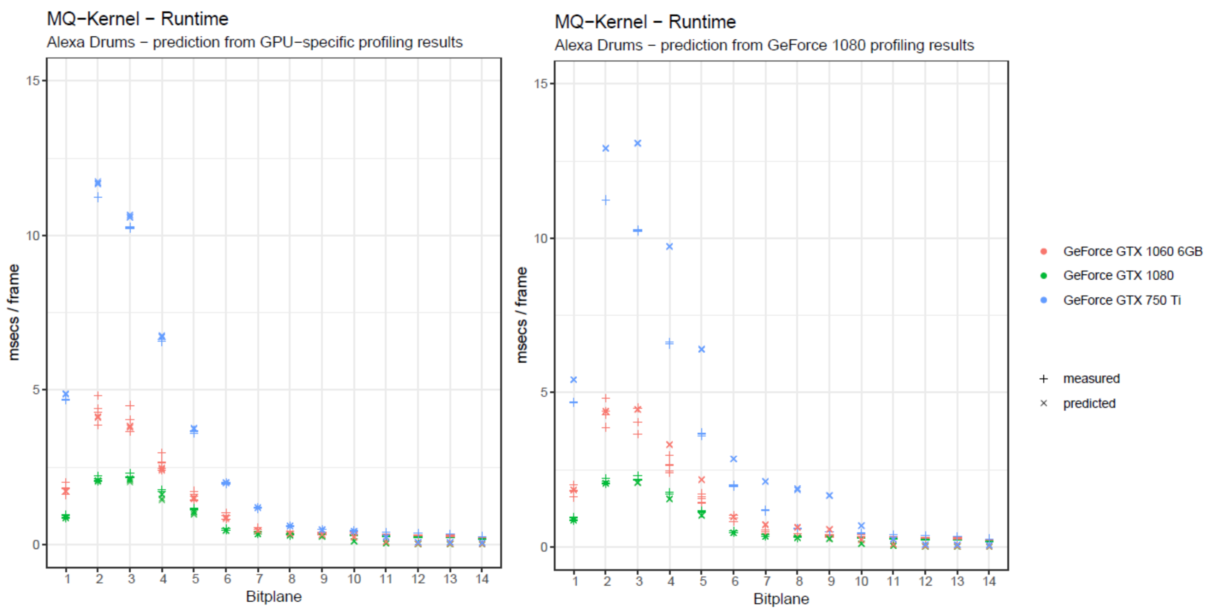


Figure 9-22 – predicted (‚x‘) vs. measured (‘*’) runtime of the arithmetic coding kernel for the Alexa Drums sequence on three different GPUs. **Left**: Runtimes are predicted using the respective GPU’s own profiling results. **Right**: runtimes are predicted based on the profiling results for the GeForce GTX 1080.

The reason is that the difference in the efficiency metric (instructions/cycle) between the two devices is larger (see Figure 9-23 center). The workload (instructions/warp) and utilization (multiprocessor activity) are largely the same. The order in which the processor usage starts dropping corresponds to the order of the number of cores: the biggest GPU starts dropping earliest.

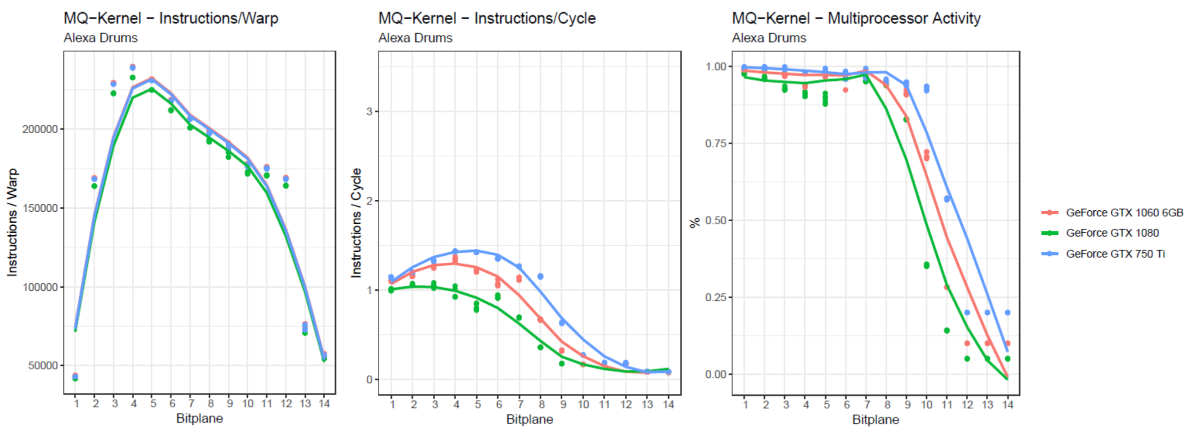


Figure 9-23 – Profiling metrics “Instructions/Warp” (**left**), “Instructions/Cycle” (**center**) and “Multiprocessor Activity” (**right**) for three GPUs

Table 9-3 – predicted runtime of arithmetic-coding kernel on three GPUs

Sequence	Measured		Predicted		Error
	GPU	msec	from GPU	msec	msec
Alexa Drums	GTX 1080	10,82	GTX 1080	9,13	-1.69
	GTX 1060 6GB	17,88	GTX 1060 6GB	15,99	-1.89
	GTX 750 Ti	42,45	GTX 750 Ti	42,83	+0.38
	GTX 1060 6GB	17,88	GTX 1080	19,48	+1.6
	GTX 750 Ti	42,45	GTX 1080	57,23	+14.78

In comparison to the context modeling kernel, the prediction accuracy is significantly worse, especially when forecasting the runtime on a *GeForce GTX 750 Ti* using the benchmarking results measured on a *GeForce GTX 1080*. For this kernel, a forecast is only sensible for GPUs of the same architecture.

9.4 Conclusion

This chapter presents a runtime model that permits to predict, based on benchmarking metrics measured for a particular kernel on one GPU, the runtime on another GPU. In two case studies, the original JPEG 2000 compliant implementations of the context modeling and arithmetic-coding kernels are evaluated. The workload for both kernels depends very strongly on the signal statistics of the source image and so a prediction that is not dependent on the benchmark results is infeasible. The context modeling kernel's runtime can be predicted quite accurately from the high-end *GeForce GTX 1080* also to the mid-range *GeForce GTX 1060 (6 GB)* and the older mid-range GPU *GeForce GTX 750 Ti*. The arithmetic-coding-kernel yields higher differences in the measured kernel efficiency (achieved instructions per clock) and so a runtime prediction is only possible with good accuracy between the *GeForce 10* GPUs from the same architecture. While both kernels achieve a full GPU utilization in the significant bit planes, the benchmarking results confirm that the coarse threading layout of the arithmetic coding kernel where a single thread processes all context-decision pairs from an entire 32x32 code block's bit plane is problematic. The workload of a single SIMD-group (instructions per warp) is considerably higher and the efficiency (instructions / block) lies at only about one third of that of the more finely parallelized context modeling kernel. A strategy for increasing the overall throughput must be to break or reduce the long causal dependency chains that occur when encoding a long bit stream with a context-adaptive entropy coder.

10 JPEG XS

JPEG XS is a recent addition to the JPEG set of still image codecs. The call for proposal was released in March of 2016 and the core coding system standard has been published in May of 2019 as " ISO/IEC 21122-1:2019" [67] and as such it has been developed during the course of this work. The call for proposals was answered by various parties [66]. *Intopix s.a.*, creators of the TICO (*Tiny Codec*) lightweight codec, and *Fraunhofer IIS* were tasked to develop the new standard. During this time, the first GPU-based JPEG XS decoder implementation has been realized. It has been originally published in mid-2018 [89] and so before the standard's first version had been finalized. Subsequently, it became part of the commercially available Fraunhofer IIS JPEG XS SDK [90].

This chapter will first describe the JPEG XS codec itself. The algorithm will be described from the encoder's point of view as it is found that the concepts are easier to understand this way. A GPU encoder implementation has not been carried out as part of this work. The GPU realization of a decoder will be described in depth in the second half of this chapter, followed by a report of experimental results and concluding remarks. The evaluation in this chapter will be limited to a comparison with standard JPEG 2000. An in-depth comparison of all coding modes investigated and proposed in this work, including JPEG 2000 and JPEG XS, is presented in the next chapter.

10.1 Description of the Encoder

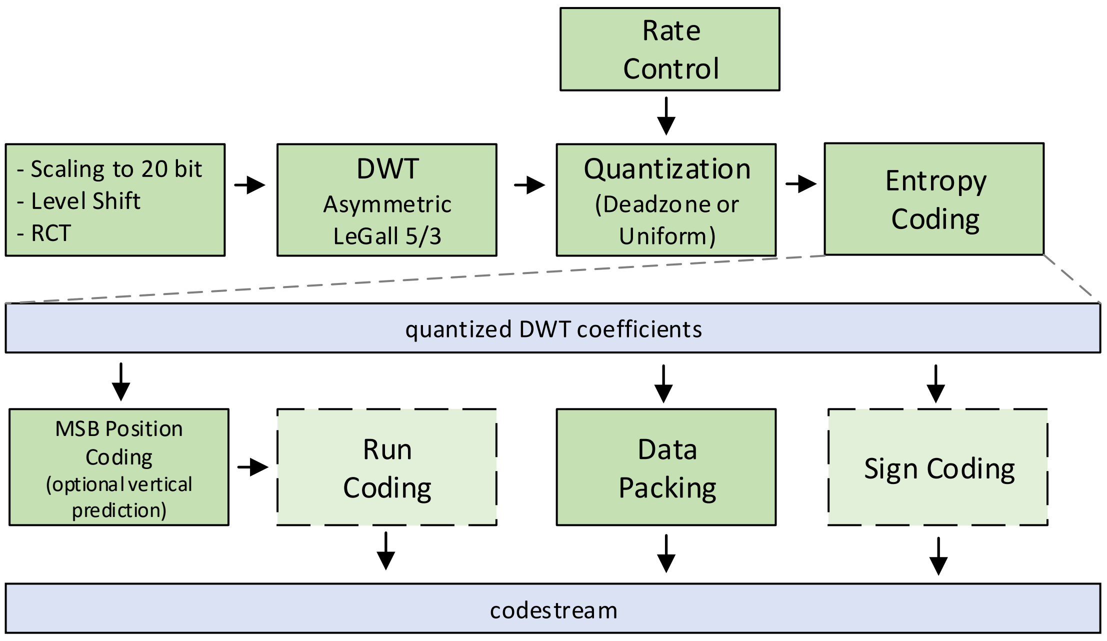

Rate Control
- Scaling to 20 bit
- Level Shift
- RCT
DWT Asymmetric LeGall 5/3
Quantization (Deadzone or Uniform)
Entropy Coding
quantized DWT coefficients
MSB Position Coding (optional vertical prediction)
Run Coding
Data Packing
Sign Coding
codestream

Figure 10-1- JPEG XS Encoder Block Diagram

The above figure (Figure 10-1) shows the major coding blocks of a JPEG XS encoder [67]. Except for the uniform quantizer, all coding operations can be realized with integer arithmetic. When selecting the deadzone quantizer, no floating-point operations are required at all. This is not only beneficial for reducing the complexity and thereby increasing the throughput, but also for ensuring a very good multi-generation-coding robustness as rounding errors can be avoided.

10.1.1 Color Transform

The color channels are first decorrelated into luminance and chrominance using an integer-based RGB to YCbCr transform similar to that of JPEG 2000 operating in lossless mode.

10.1.2 Wavelet Transform

JPEG XS employs an asymmetrical *LeGall 5/3* DWT. It is identical to the kernel used in JPEG 2000 when operated in lossless mode, except for the fact that the wavelet structure is asymmetrical. While in JPEG 2000 part-1, the number of horizontal and vertical wavelet decompositions is always equal, this is not the case in JPEG XS. In order to limit the minimum number of wavelet domain rows the decoder requires to reconstruct a single row in spatial domain (which directly affects the achievable latency), the number of iterations the image is filtered two-dimensionally, i.e. both horizontally and vertically, is limited to a minimum, e.g. only once or twice. Beyond that, the LL band is decomposed further only in the horizontal direction.

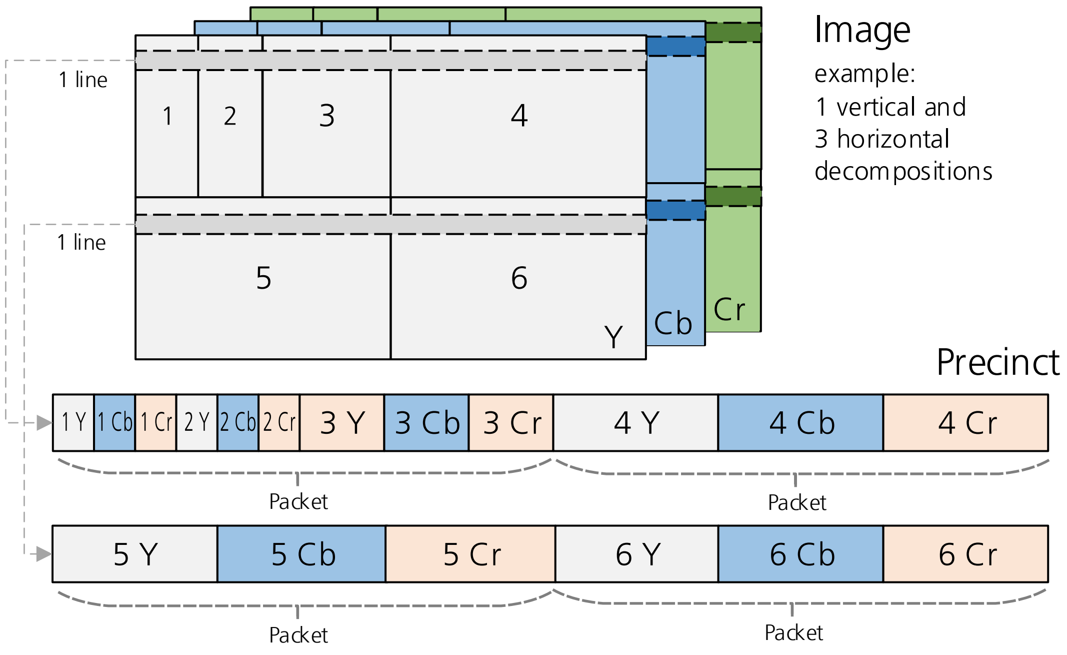


Figure 10-2 – Structure of a Precinct for a DWT with 1 vertical and 3 horizontal decompositions. The four outermost bands are each represented by a single packet, so that a decoder can reconstruct a proxy resolution simply by skipping over the packets with subbands 4, 5 and 6

The above figure (Figure 10-2) shows how an image is decomposed once in both horizontal and vertical directions and subsequently twice more in the horizontal direction only. The set of all spatially corresponding lines is denoted a precinct. Given the number of vertical decompositions v, the number of lines per precinct equals 2^{v}. The combination of a wavelet domain line from a particular subband of a particular channel is denoted a *level*.

10.1.3 Magnitude Bit Coding

The encoder processes and emits the precincts from top to bottom. Each precinct is quantized and entropy-coded. To do that, the quantized coefficients in each subband are split up into *coding groups* of four coefficients each. Looking at the signed values in sign-magnitude representation, the encoder then determines for each such coding group the smallest common *MSB Position* of the four magnitudes, which is the highest bit plane that contains not only zero bits (see Figure 10-3). Furthermore, the rate control computes per subband a *Truncation Position*. The magnitude bit planes between the *MSB Position* and the *Truncation Position* are emitted directly to the code stream and not further coded.

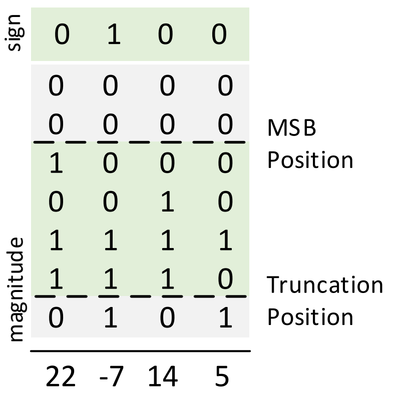


Figure 10-3 - Coding Group. Only green bit planes are included in code stream

10.1.4 Sign Bit Coding

Sign bits are processed according to one of two options. If the encoder chooses to employ *sign packing*, the sign bits of only those coefficients that have a non-zero magnitude after truncation are coded. In this mode, sign bits will be located in a code stream segment separate from the magnitude bits. The alternative option is to interleave the sign bits with the data bits. In this case, the coding group's four sign bits are simply emitted upfront. Sign packing requires a slightly higher computational complexity, but yields superior coding efficiency, since sign bits of zero-magnitude coefficients are omitted.

10.1.5 MSB Position Coding

The encoder needs to signal both the MSB position (per coding group) and the truncation position (per subband) to the decoder. MSB positions can be coded in one of three ways:

1. In the simplest case, they are coded raw with four bits per value.
2. Exploiting that MSB positions for adjacent coding groups are often similar, the encoder can choose instead to predict a MSB position from the corresponding value in the line above. The current MSB position is then coded with a variable length unary code, denoted *bounded code*, where the alphabet is constructed in a fashion that the code word length grows with the magnitude of the prediction residual: the residuals 0,-1,+1,-2,+2, etc. are mapped to 0, 10, 110, 1110, 11110, etc. [91].

3. The third alternative is for the encoder not to employ vertical prediction, in which case the alphabet's code words consist of a 1-bit-sequence with a length equal to the coded MSB position value, followed by a 0-bit serving as a comma.

On top of that, the encoder may optionally omit zero-runs from the code stream. To that end, the concept of *significance-groups* that each comprise eight MSB position values is introduced. One bit per significance-group signals whether all values in the group were zero, in which case the group is omitted from the code stream. Whether the run refers to residuals or absolute values is indicated globally in the picture header. The prediction mode (raw coding, vertical prediction on/off, significance coding on/off) can be chosen individually for every subband in a precinct. Precincts are additionally grouped into *slices*. Vertical prediction is prohibited in the first precinct of each slice, facilitating parallel processing of slices.

10.1.6 Truncation Position Signaling

The truncation position for band *b* in precinct *p*, i.e. number of LSB bit planes the encoder cut off in order to meet a bit rate constraint, is determined by the rate control and defined by means of a *quantization value Q[p]* and *refinement value R[p]* stated in the precinct header. *Q[p]* states an upper anchor on how many planes were truncated, which is decreased more for bands with important frequencies as specified by a table in the global *weights table header*. Since the encoder might not have exhausted its budget after this coarse bit rate distribution, the number of truncated bit planes can be further decreased by one for a subset of the precinct's bands. The subset consists of the first *R[p]* entries in the *subband priority list P[b],* also located in the global weights table header.

10.1.7 Code stream Structure

Table 10-1 gives an overview of the hierarchal units a JPEG XS code stream is composed of. The code stream comprises multiple global headers (most importantly *picture header, components table* and *weights table*), followed by all the slices. Each slice, in turn, contains a slice header, a payload that consists of all the precincts in that slice, and a footer.

Table 10-1 - JPEG XS code stream Hierarchy

Hierarchy Level	Description
Picture Code Stream	A single intra-coded image or video frame
Slices	Each slice can be decoded independently
Precincts	Collection of subbands that contribute to a given spatial region
Packets	Segment that contains information from one or multiple subbands
Subpackets	Packet subsegment containing a single data kind

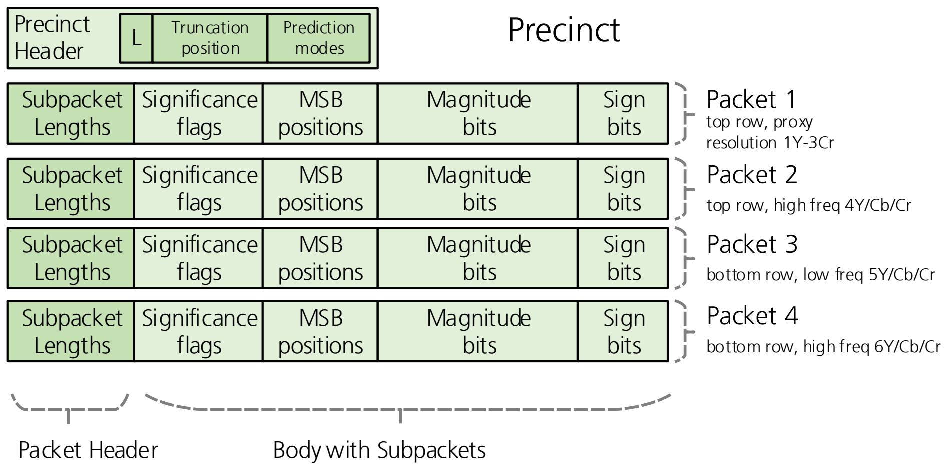


Figure 10-4 - Structure of a precinct (one vertical decomposition)

The structure of a single precinct is depicted in Figure 10-4. In the case of one vertical wavelet decomposition, a precinct comprises two rows split into four packets, each containing the information of one of the four outermost high-frequency subbands. This way, the decoder can choose to reconstruct only a proxy resolution simply by skipping over the packets 2-4, which contain the HL, LH and HH bands. In the case of two vertical decompositions, a precinct contains two additional packets for each of the extra three subbands, yielding a total of ten packets. In this scenario, a packet is still restricted to a (possibly partial) single row in wavelet domain, but the six packets from the new outermost HL, LH and HH bands contain twice as many samples as the original four packets. Each packet comprises a header that advertises the bit lengths of the subpackets in the packet body. These subpackets are:

- the significance flags
- coded MSB positions (bit plane count in the magnitude bits subpacket)
- raw magnitude bits
- and the sign bits.

When sign-packing is not used, the magnitude and sign bits are interleaved in a single data-subpacket.

10.2 Encoding JPEG XS on a GPU

A GPU-accelerated encoder has not been implemented as part of this work.

10.3 Decoding JPEG XS on a GPU

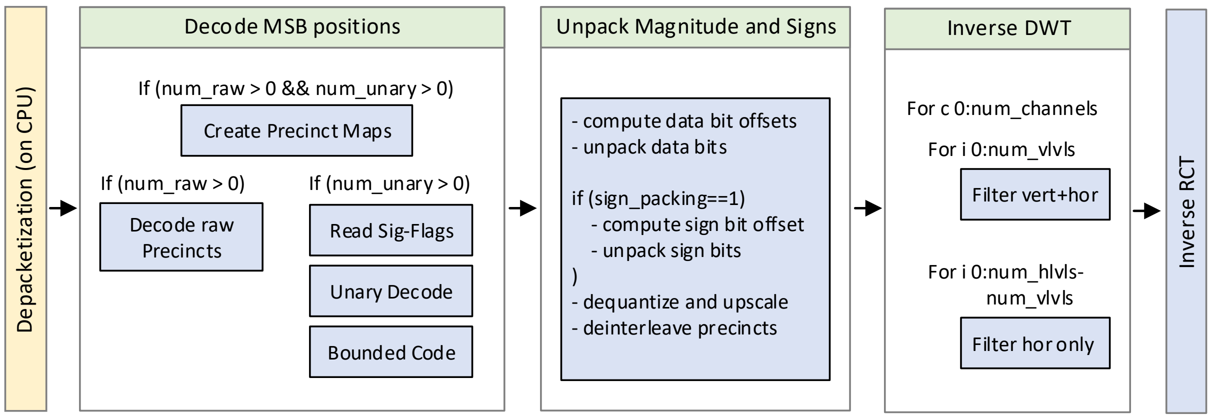


Figure 10-5 - Overview of decoder architecture. Blue boxes represent GPU kernels.

The processing power of a GPU stems from its hundreds or thousands of cores that can compute in parallel. When mapping any task to a GPU, it needs to be broken down into thousands of individually computable subtasks. It is important to oversubscribe the available physical cores in order to enable the GPU scheduler to hide unavoidable latencies introduced by accesses to the slow global memory by swapping in other threads in the meantime. In light of these requirements, the decoder architecture described here is designed to operate on an entire code stream in parallel at the cost of increased latency. In this chapter, the blocks lined out in Figure 10-5 will be described.

Research questions: Has the JPEG XS design goal of making the decoder fully parallelizable been achieved? What throughput is achievable on current high-end and mid-range GPUs? Does the use of a GPU as a hardware platform in any way limit the use cases of a JPEG XS decoder?

The contents of this chapter have been published in [89].

10.3.1 Depacketization

Before the decoder can start reconstructing the MSB positions from the code stream it needs to find the locations of each subpacket's MSB position segment in the code stream. Determining the offsets on the decoder side is an inherently serial task that consists of iterating over all precinct headers one by one. This task will be referred to here as *depacketization* and is realized in a CPU-function that is executed prior to any GPU-kernels.

10.3.2 Reconstruction of MSB Position

The first task is to reconstruct the MSB positions for all coding groups in the image. In the design proposed here, separate independent paths for raw-coded and unary-coded precincts exist (see Figure 10-5). *Raw-coded* precincts can be reconstructed without any further preparation. Given the code stream bit offsets, one thread per coding group can be spawned, read the 4-bit MSB position value and store it in the MSB position grid. In case not all of the precincts are coded raw, it is beneficial to spawn threads only for raw-coded precincts. This requires a precinct-map that maps thread blocks to precincts. First, a list is created with one entry per precinct, stating the precinct index and whether the precinct is raw-coded or not. The raw-coded precinct map is then derived by removing all entries of unary-coded precincts and compacting the remaining entries. The unary-coded precinct map can be created analogously.

Reconstructing the *bounded-coded* MSB positions requires a more elaborate workflow. We propose to first decode the unary-coded values from the code stream and defer their mapping to MSB position prediction residuals to a subsequent kernel. Essentially, the lengths of 1-bit sequences that are separated by 0-bit "commas" have to be identified. One thread block is assigned to each subpacket. Each thread is tasked to decode all code words that end in a 4 byte-range of the subpacket's bit stream. The threads collaboratively loop over the data until the subpacket's entire MSB position segment has been decoded. If the 4 bytes start with a 1-bit, the thread cannot be sure whether the 1-bit-sequence extends further into the previous 2 bytes (given a maximum code word length of 16 bits) and so it needs to access those as well. In order to compute the location where the decoded symbol must be written, the numbers of decoded symbols in all 4-byte segments need to be accumulated. This is efficiently done using the parallel prefix-primitive.

After the variable-length-code has been read, the next kernel, denoted *bounded-code-kernel*, interprets the decoded symbols and restores the MSB position values. Since the MSB positions in many subbands will be vertically predicted from the corresponding subbands in the precinct above, this kernel processes chains of vertically predicted precincts sequentially. Leveraging that the first precinct in each slice is never vertically predicted, all slices are processed concurrently by spawning threads (one per coding-group) only for the first precinct in each slice. The threads then loop over all precincts in the slice from top to bottom, inverting the vertical prediction and interpreting the bounded-code symbols.

In case the encoder omitted runs of zero-valued MSB positions and signaled this using the significance-flags, the *bounded-code-kernel* cannot directly compute which symbol corresponds to which coding group. A mapping that states for each significance group the total number of absent symbols in the subpacket up to that point needs to be created first. This is achieved by calculating an exclusive prefix-sum, where one thread per significance-group

a) contributes a zero, if the corresponding significance-flag indicates that all coding-group values in that significance-group were zero,

b) or contributes the number of coding-groups in the significance-group, if the significance-flag indicates that at least one coding-group value is non-zero.

10.3.3 Unpacking Magnitude and Sign bits

After the MSB positions for all coding-groups have been reconstructed, the number of coded magnitude bits per coding-group can be computed and the bits can be extracted ("unpacked") from the code stream.

A single kernel locates and unpacks all magnitude and sign bits, where one block of threads is in charge of an entire subpacket. The block loops over the coding groups piece by piece. As indicated in Figure 10-5, during each iteration, the threads first execute a prefix scan over the number of coded data bits per coding group in order to calculate each coding group's bit offset within the data segment. Then they extract the data bits from the code stream. If sign bits are packed and stored in a separate subpacket segment, the threads need to calculate another prefix sum over the number of non-zero coefficients per coding group, thereby calculating the bit offset in the subpacket's signs-segment. Next, each thread reads in their up to four sign bits. At the end of each iteration, each thread stores their coding group's four signed coefficients to global memory. The aggregates of the current loop's two prefix scan operations are carried over into the next iteration and added to the next prefix sum result.

Before the inverse DWT can be applied, coefficients first need to be dequantized, converted from sign-magnitude representation into two-complement representation and upscaled. We propose to carry out these operations also within the unpacking kernel in order to minimize accesses to slow global memory. Whether the encoder chose the deadzone or uniform quantization option, is signaled globally in the picture header. In both cases, the dequantization routine requires the MSB and truncation positions, both of which are required in the unpacking kernel anyway. Finally, the unpacking kernel also deinterleaves the precincts and color channels. It stores each color channel separately with low vertical frequencies in the top half and high vertical frequencies in the bottom half (in the case of only one vertical decomposition).

10.3.4 Inverse DWT

The inverse DWT employed in this design is based on the fast lifting implementation described in [38], which proposes to fuse the horizontal and vertical filter operations into a single kernel. In order to accommodate for the asymmetrical DWT required for JPEG XS, a second variant was derived that filters only in the horizontal direction. The workflow is shown in Figure 10-5.

10.3.5 Inverse Reversible Color Transform

The RCT kernel not only transforms back from YCbCr to RGB, but also scales the samples to the desired output bit depth and arranges them in the required pixel format. The kernel is spawned with one thread per pixel (two in the case of 4:2:2 chroma subsampling).

10.4 Experimental Results

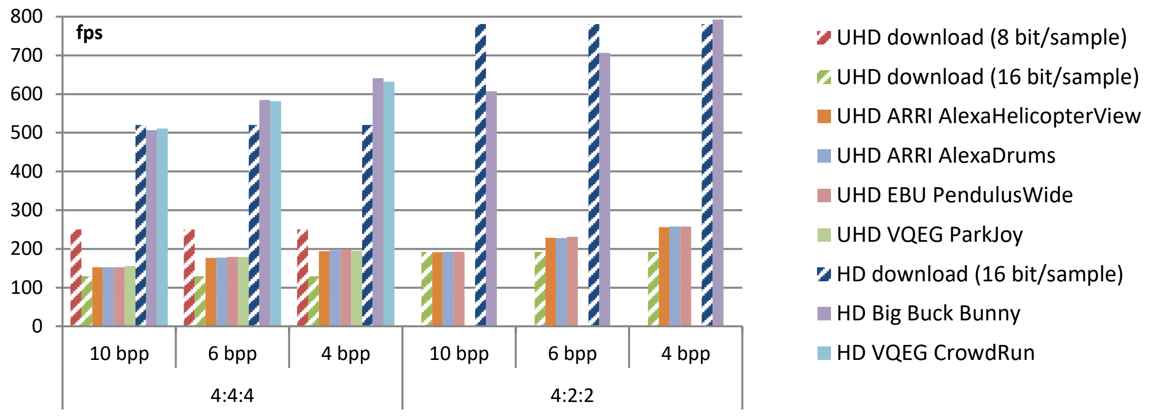


Figure 10-6 - Throughput on GeForce GTX 1080, PCI-E 2.0 x16. Sign-packing and sig-flags enabled, 5 horizontal and 1 vertical DWT level

The decoding throughput for various test sequences is compared in Figure 10-6. While the code stream upload is included in the throughput calculation, the download of the decoded image is listed separately as the decoder uses a three-stage pipeline as shown in Figure 10-7: a frame is downloaded while the next frame is decoded. In parallel, the CPU depacketizes frames, but this is uncritical, as it only takes a little over one millisecond for a 10 MB UHD 4:4:4 code stream. In theory, a four-stage pipeline (Figure 10-7 c) could be employed, where upload, decode and download are all executed concurrently at the cost of increased latency.

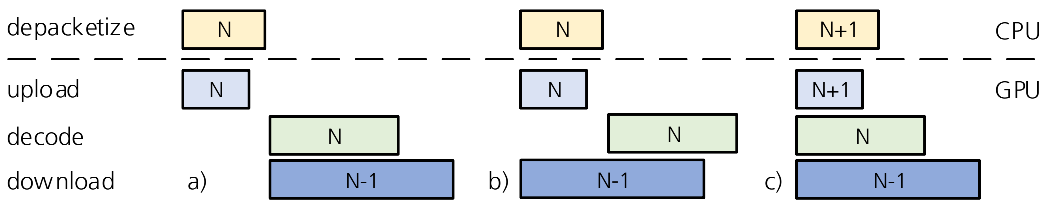


Figure 10-7 - three-stage pipeline without (a) and with (b) hardware support for concurrent transfers. c) four-stage pipeline with concurrent transfers

It can be seen in Figure 10-6 that the decoding throughput is not very sensitive to the image content, but mostly to the resolution and data rate. Another observation is that on this high-end GPU the download poses a bottleneck for UHD sequences reconstructed with 16 bit per sample.

Table 10-2 - Decoding FPS, excluding device-to-host transfer. Encoding options: sign-packing and sig-flags enabled; 5 horizontal and 1 vertical DWT levels; 10 bpp

Format	upload and decode		download
	GeForce GTX 750 Ti (640 cores)	*GeForce GTX 1080* (2560 cores)	PCI-E 2.0 x16 16 bit/sample
UHD 422	54	191	193
UHD 444	42	153	129
HD 444	145	507	520
HD 422	189	607	781

For a detailed discussion of the available data transfer rates, the reader is referred to chapter 2.3.4. In short, already today the download would not pose a bottleneck, when using a mainboard that supports PCI 3.0 or newer. The available bandwidth have grown significantly in recent years and so this bottleneck will disappear.

Table 10-2 compares the throughput measured with two different graphics devices. The performance scales very well with the number of cores: the penalty for the high-end card is usually within 5% when comparing the frames per core.

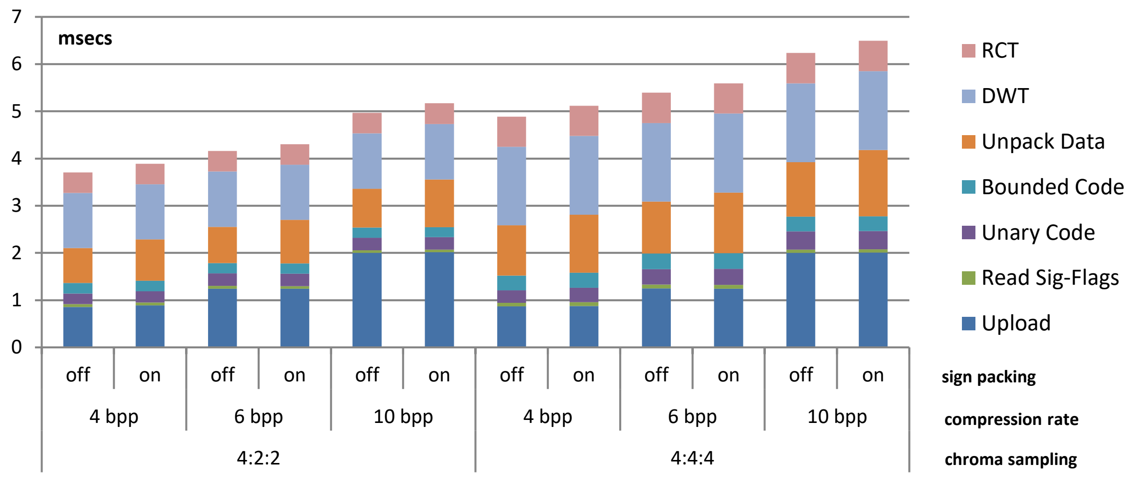


Figure 10-8 - Detailed throughput for 12 bit UHD *Alexa Heli* test sequence on *GeForce GTX 1080*. Encoding options: significance-flags enabled, 5 horizontal and 1 vertical DWT level

In Figure 10-8, the runtime is broken down for the individual kernels. On a high-end GPU, uploading the code stream is often the slowest operation. On slower GPUs, the DWT and data unpacking kernels are the most expensive kernels. Sign-packing reduces the throughput only by 3-4%.

At the time of the author's PCS 2018 publication [89], no other speed-optimized CPU, GPU or FPGA implementations of JPEG XS were available. A speed comparison with the verification model would be unfair and pointless. The outputs of the GPU decoder and verification model are bit identical. A comparison of GPU-based JPEG XS and JPEG 2000 decoders is displayed in Figure 10-9. The throughput decreases inversely to the bit rate, but this effect is more drastic for the JPEG 2000 decoder: at higher bit rates the JPEG XS decoder is 2x faster and at lower bit rates 3x faster. Moreover, the JPEG 2000 implementation needs to decode eight frames at once in order to fully load the GPU, because the entropy decoder processes code blocks of 32x32 samples with a single thread each. Exploiting intra-block parallelism is difficult due to the feedback loop between arithmetic decoder and context modeler. This increases both latency and memory footprint. The JPEG XS decoder, in contrast, does not require frame grouping. Due to the finely grained parallelism exploited in all kernels, it can fully load even a high end GPU with only a single frame and will automatically benefit from future GPU generations with even more computing cores.

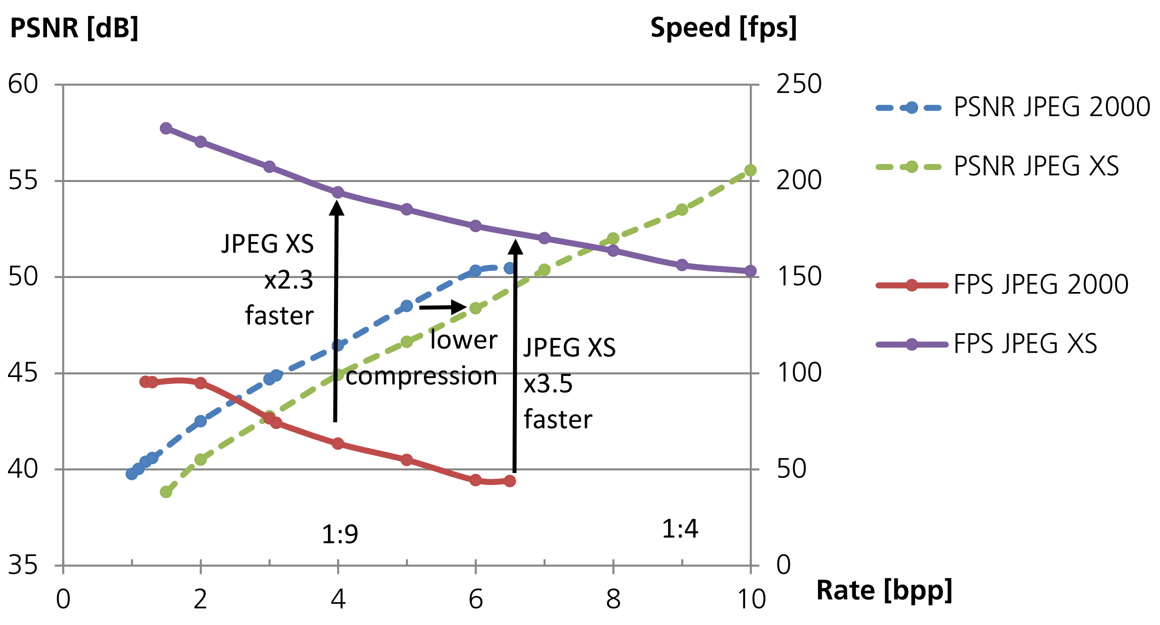


Figure 10-9 - JPEG XS vs JPEG 2000. Throughput (upload & compute) measured on a GeForce GTX 1080 for 12 bit UHD 444 *Alexa Heli* sequence. JPEG 2000: lossy, 12 bit, 32x32 code blocks

10.5 Conclusion

The first research question "*Has the JPEG XS design goal of making the decoder fully parallelizable been achieved?*" can be positively answered. All major kernels utilize finely grained parallelism by mapping one thread to a coding group (four coefficients), a pixel or a sample.

What throughput is achievable on current high-end and mid-range GPUs? Experiments were carried out with a 2017 high-end *GeForce GTX 1080* with 2560 cores and a 2014 mid-range G*eForce GTX 750 Ti* with 640 cores. The high-end GPU can decode a 10 bpp UHD 4:4:4 sequence with 153 fps. When decoded frames need to be downloaded and more than 24 bits precision are required, the throughput is limited by the PCI-Express 2.0 bandwidth at 129 fps for a sample format with 6 Bytes per pixel. On the mid-range GPU the computation itself presents the bottleneck. UHD sequences can only be decoded with 42 fps (4:4:4), respectively 54 fps at this point and so further optimizations are required to enable real-time processing of UHDp60.

Does the use of a GPU as a hardware platform in any way limit the use cases of a JPEG XS decoder? The architecture proposed in this paper sacrifices the low-latency feature for increased parallelism. The GPU kernels operate back to back on the entire image whereas a decoder optimized for low-latency would need to process the precincts one by one, as they arrive at the decoder. Notably, a single frame exposes sufficient opportunity for parallel processing to fully load even a high-end GPU.

11 Evaluation

11.1 Impact of Quantization

In baseline JPEG the compression rate is determined solely by the quantization step size. A coarser quantization will lead to more DCT coefficients becoming zero, which in turn leads to a higher information loss and increased compression. The codec offers no possibility to specify a target bit rate other than experimenting with various quantization step sizes until one results in a compressed size that is close to the desired rate.

JPEG 2000 on the other hand has two means of dropping information. In addition to the quantization, it is able to truncate the compressed code block bit streams. This is possible due to their embeddedness property. The important difference when considering the throughput of an encoder is that the quantization drops information *before* the time consuming block coder, whereas the post compression truncation drops it *after* the quantization indices have already been encoded. The two mechanisms should be carefully coordinated or otherwise two undesirable situations can occur:

1. **coarse quantization with high target bit rate yields sub-optimal quality**: the target bit rate serves as a maximum budget barrier that must not be exceeded. The encoder will first decorrelate, quantize and entropy-code the image and then only activate the rate distortion optimization machine when required. If the compressed image size does not exceed the afforded bit rate budget, there is no need to truncate any code block bit streams. This situation is undesirable as the coarse quantization caused the quality of the resulting compressed image to be lower than possible since the bit rate budget is not being exhausted. Selecting a finer quantization step size would have yielded a better result with less distortion
2. **fine quantization with low target bit rate yields sub-optimal throughput**: The drawback of selecting a very finely grained quantization step size is that the block coder will waste time on compressing information (low bit planes) that will end up being truncated anyway. Some coders employ a heuristic to anticipate preemptively for each code block exactly how many passes need to be coded instead of simply always coding all the passes down to the least significant bit of the quantized wavelet coefficients [49].

Since the JPEG 2000 framework employed here to compare the various block coders does not use such a heuristic, the selected quantization step size will be reported for all experiments.

In JPEG 2000, the quantization step sizes are specified per subband. An encoder can choose to include only the base quantization step size in the code stream, in which case the decoder will derive all band specific step sizes by dividing the base step size by two for every additional decomposition level. The JPEG 2000 encoder framework used in this work utilizes the algorithm implemented in the *Kakadu* library to derive discrete step sizes for each subband and signal them explicitly in the code stream. It computes the energy gains of both the vertical (g_v) and horizontal (g_h) wavelet kernels. Given a base quantization step size of Δ_{ref}, e.g. 128^{-1}, and band *b*, the quantization step size Δ_b is then given by

$$\Delta_b = \frac{\Delta_{ref}}{\sqrt{g_v(\mathrm{b}) * g_h(\mathrm{b})}}$$

For $\Delta_{ref} = 128^{-1}$, the step size for the LL band after five decompositions, Δ_{LL0}, is 0.000115. The band in the outermost resolution level, Δ_{HH5}, on the other hand has a step size of 0.003754. Let ε_b denote the number of bit planes used to store the quantization indices in band *b*. It is derived from Δ_b such that ε_b is the highest possible integer that guarantees $\frac{1}{2^{\varepsilon_b}} < \Delta_b$. For the example of $\Delta_{ref} = 128^{-1}$ and five decomposition levels (HD/2K image), the bands from the outermost to the innermost resolution levels use 9, 9, 11, 12, 13, 14 bit planes. For further details regarding the procedure for deriving the step sizes as well as the theory on quantization and JPEG 2000 specifics, the reader is referred to [70].

The following rate-distortion plot (Figure 11-1) shows for all image sequences how the chosen base quantization step sizes limit the maximally achievable quality. For the *Alexa Heli* sequence, for example, a coarse base step size of only $\frac{1}{64}$ is only sufficient for a maximum quality of 42.5 dB PSNR at a bit rate of 1.6 bpp. Increasing the bit rate further does not improve the quality without also selecting a finer quantization. With $\frac{1}{128}$ the distortion can be reduced to 45 dB and with $\frac{1}{256}$ to 49 dB or more. The peak quality for coding *Alexa Heli* with a base step size of $\frac{1}{256}$ lies beyond the maximally probed bit rate of 6 bpp.

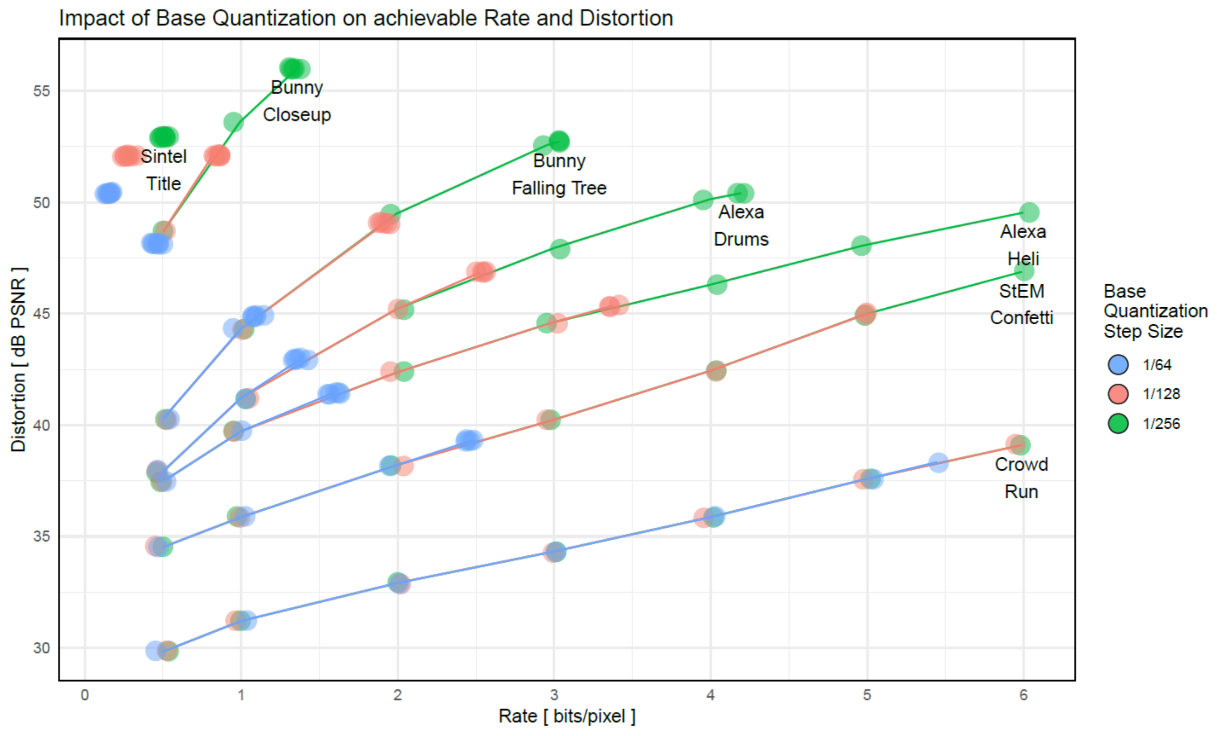


Figure 11-1 - Impact of base quantization step size on rate and distortion with EBCOT coder.

As discussed above, the throughput is significantly impacted by the quantization step sizes as it influences the total number of bit planes that need to be processed by the time consuming block coding stage. Figure 11-2 plots the encoder throughput for the EBCOT block coder over the bit rate, grouped by sequence and base quantization step size.

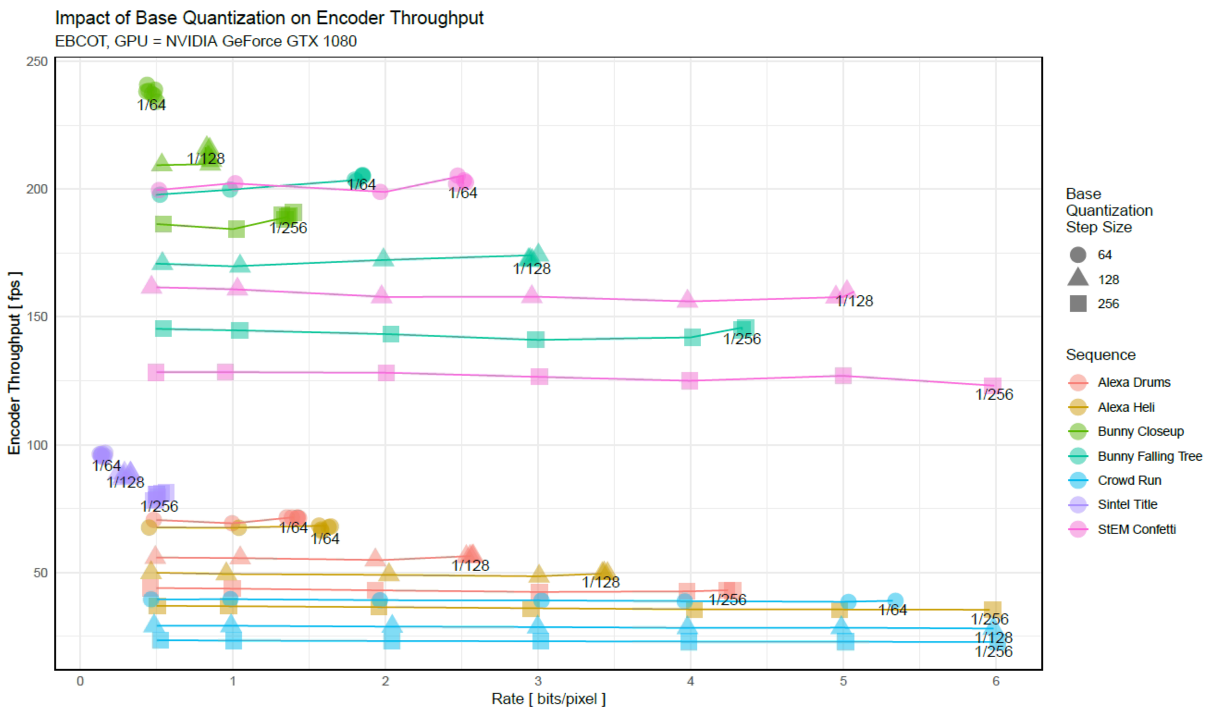


Figure 11-2 - Impact of base quantization step size on the encoder throughput of EBCOT, measured on a *GeForce GTX 1080*.

Figure 11-2 validates that the encoding speed is impacted by the quantization step size rather than the bit rate. When comparing only the various block coders with each other in the context of the JPEG 2000 framework, the chosen quantization is not essential as long as it was the same for all block coders within an experiment. However, in order to allow for a comparison also with the JPEG XS codec, the solution going forward will be to select for each bit rate, sequence and mode the most appropriate quantization step size: as long as two or more step sizes yield the same quality, the coarsest step size will be selected, because it yields the best throughput.

Figure 11-3 below plots on the left as a reference the rate-distortion curve for a constant base quantization step size of $\frac{1}{256}$ and on the right the resulting curve based on the lowest possible quantization step sizes. For this sequence, the bit rates of 0.5 bpp and 1 bpp use a base quantization step size of $\frac{1}{64}$, 2 bpp and 3 bpp use $\frac{1}{128}$ and 4-6 bpp use $\frac{1}{256}$.

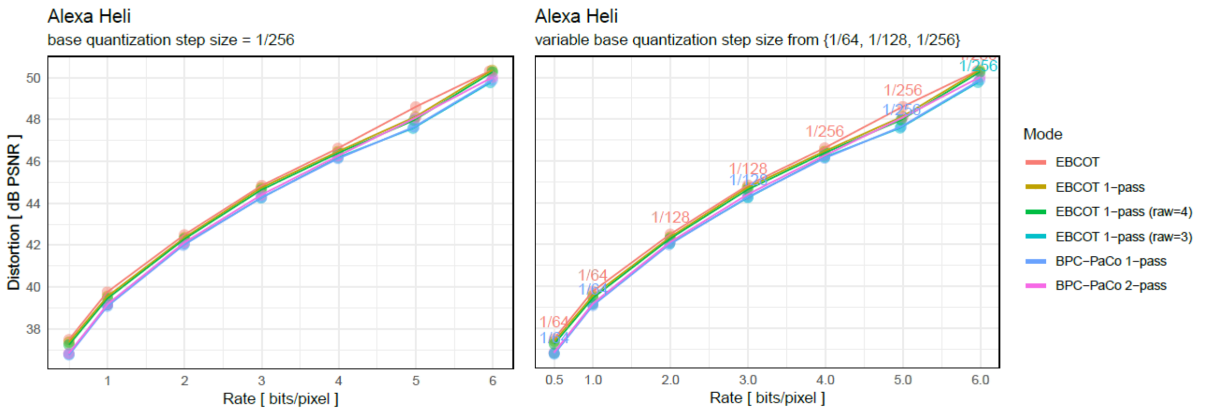


Figure 11-3 - RD-plot for *Alexa Heli* sequence coded with various block coding modes. **Left**: base quantization step size is 256^{-1}. **Right**: the smallest possible quantization step size is chosen for each bit rate, as long as it yields an identical distortion to the finer step sizes.

11.2 Quality

This chapter will discuss the quality differences of the various coding modes. Figure 11-4 shows an overview. A visible trend is that there is a clearly visible gap between JPEG XS and JPEG 2000 curves. For all sequences except *Sintel Title* and *Bunny Closeup* - the two sequences containing the fewest details - the differences among the various JPEG 2000 coders are hardly visible in this large plot. While the JPEG 2000 modes for these two sequences stop at a maximum quality of close to 60 dB PSNR, the JPEG XS curves continue to climb throughout the sample bit rate range. The JPEG 2000 information loss could be further reduced beyond 60 dB by decreasing the quantization step size.

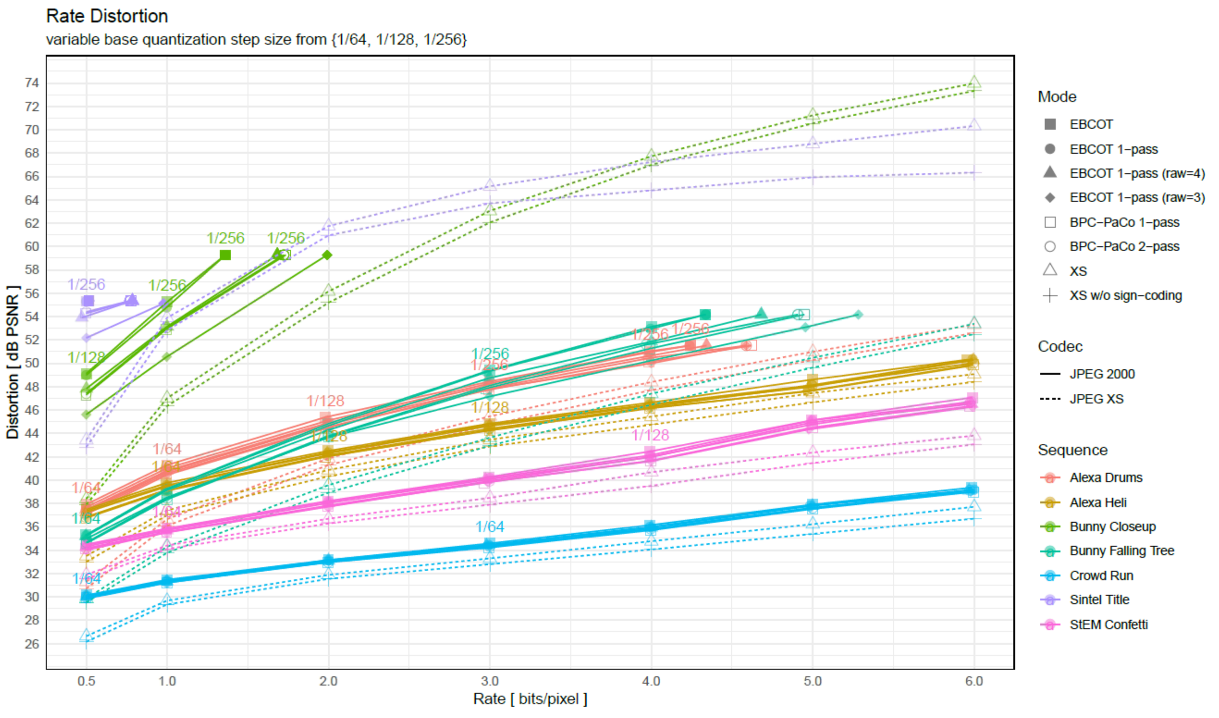


Figure 11-4 - RD-plot for all sequences and all modes.

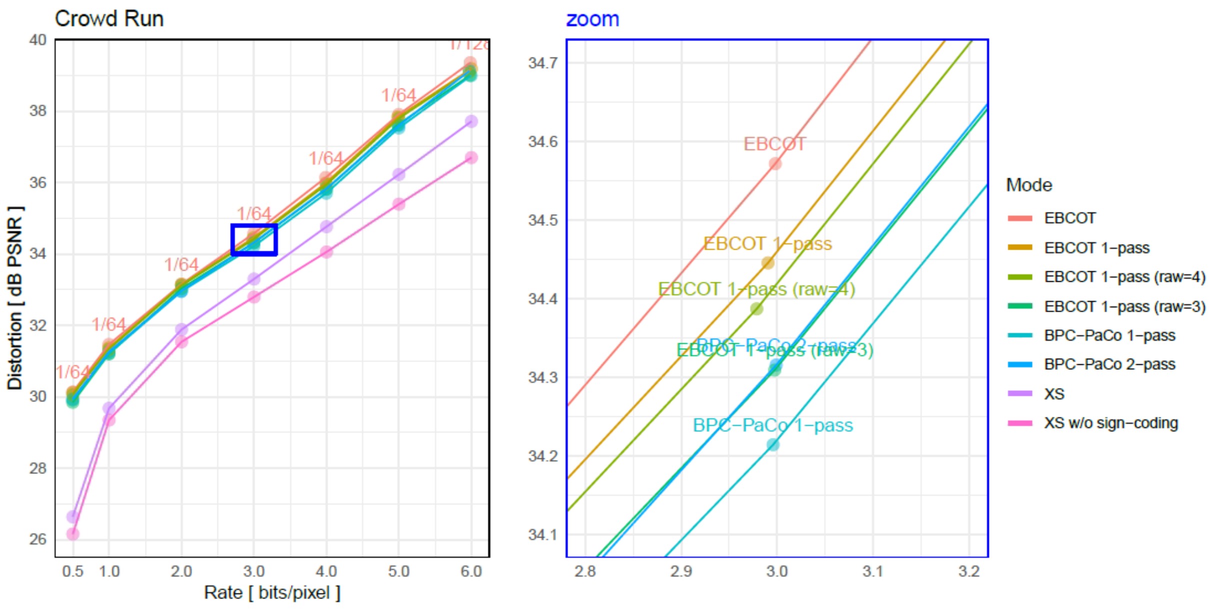


Figure 11-5 - RD-plot for *Crowd Run* sequence.

Figure 11-5 is a rate-distortion plot only for the *Crowd Run* sequence. The EBCOT and BPC-PaCo modes yield very similar quality within 0.4 dB of each other. Zooming in at the 3 bpp bit range shows the order: EBCOT performs best, followed by the singe-pass EBCOT variants, where all ("EBCOT 1-pass"), only the first three ("raw=3") and only the first four ("raw=4") bit planes are arithmetically coded. The raw-coded bit planes are directly appended to the bit stream without being prioritized by a sophisticated scan order or being arithmetically coded. The two BPC-PaCo modes with two passes and one pass per bit plane follow closely. The JPEG XS modes with and without sign coding achieve a PSNR of 33 dB and 32.8 dB, respectively, which are 1.55 dB and 1.75 dB lower than EBCOT.

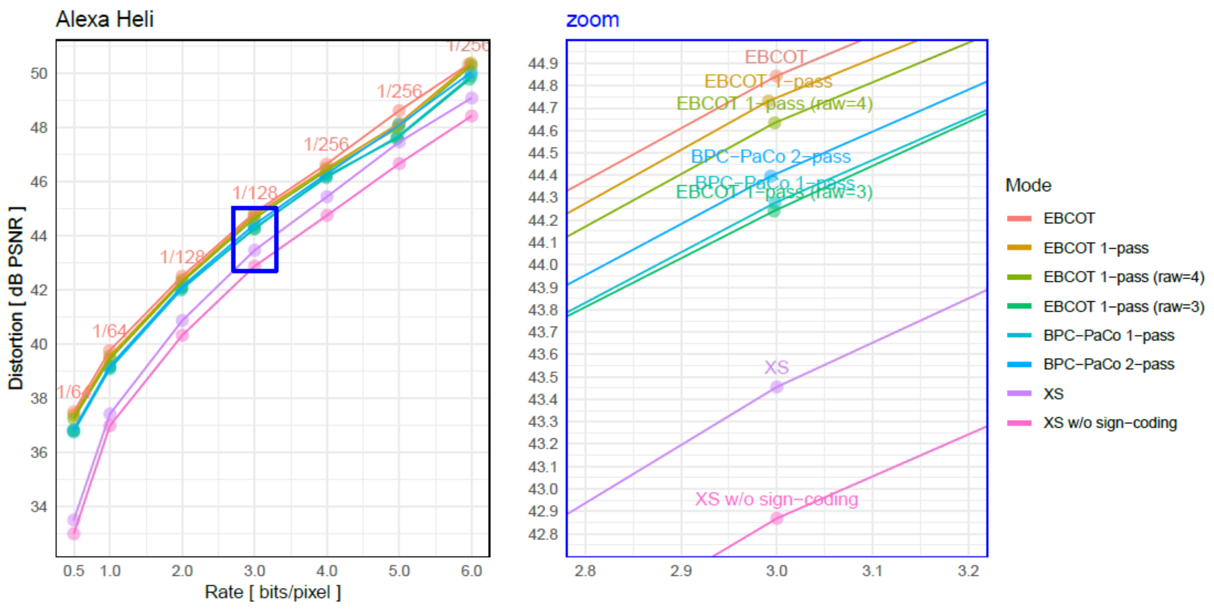


Figure 11-6 - RD-plot for *Alexa Heli* sequence.

The results are consistent with the *Alexa Heli* (Figure 11-6) and *Bunny Falling Tree* (Figure 11-7) sequences. The JPEG 2000 block coder's PSNR values lie within 0.6 dB of each other in the same order as for the *Crowd Run* sequence. JPEG XS with and without sign coding perform 1.4 dB and 2 dB below EBCOT.

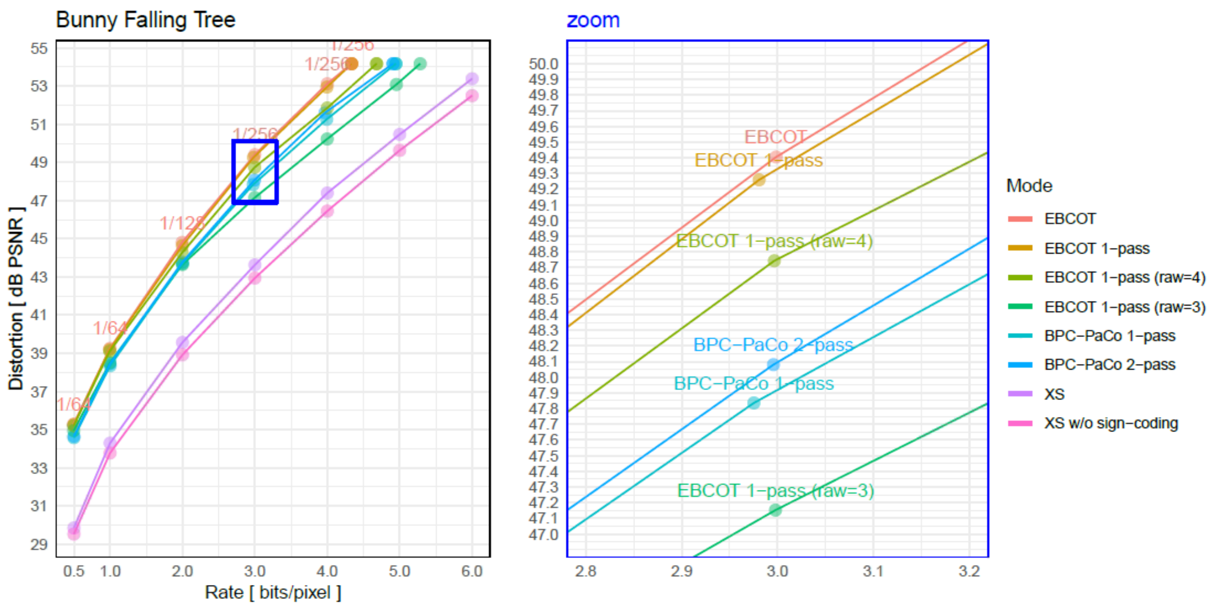


Figure 11-7 - RD-plot for *Bunny Falling Tree* sequence.

11.3 Throughput

The previous chapter has compared the various introduced coding modes in terms of their impact on the achieved quality. Since the objective of this work is to devise coding schemes that prefer throughput over quality, the loss in quality is regarded as the *cost*. The *gain*, on the other hand, is an increased throughput. This metric will be evaluated and discussed in this chapter.

Figure 11-8 shows for the UHD *Alexa Heli* sequence the encoder's and decoder's throughput over the examined range of bit rates. A significant difference between encoder and decoder is that the encoding throughput is largely independent of the bit rate, while the decoder performs faster at lower bit rates and the performance decreases steadily as the bit rate increases. However, the decoding performance, regardless which entropy coding mode, is limited at close to 125 fps, which suggests that the performance is not computation-bound. Indeed, the decoding throughput in this experiment is limited by the device-to-host transfer of the reconstructed frames. The previously decoded group-of-frames is transferred in parallel to the decompression of the current group-of-frames. The JPEG XS decoder implementation achieves a better overlap of data transfers with computations compared to the JPEG 2000 based codecs for reasons explained below.

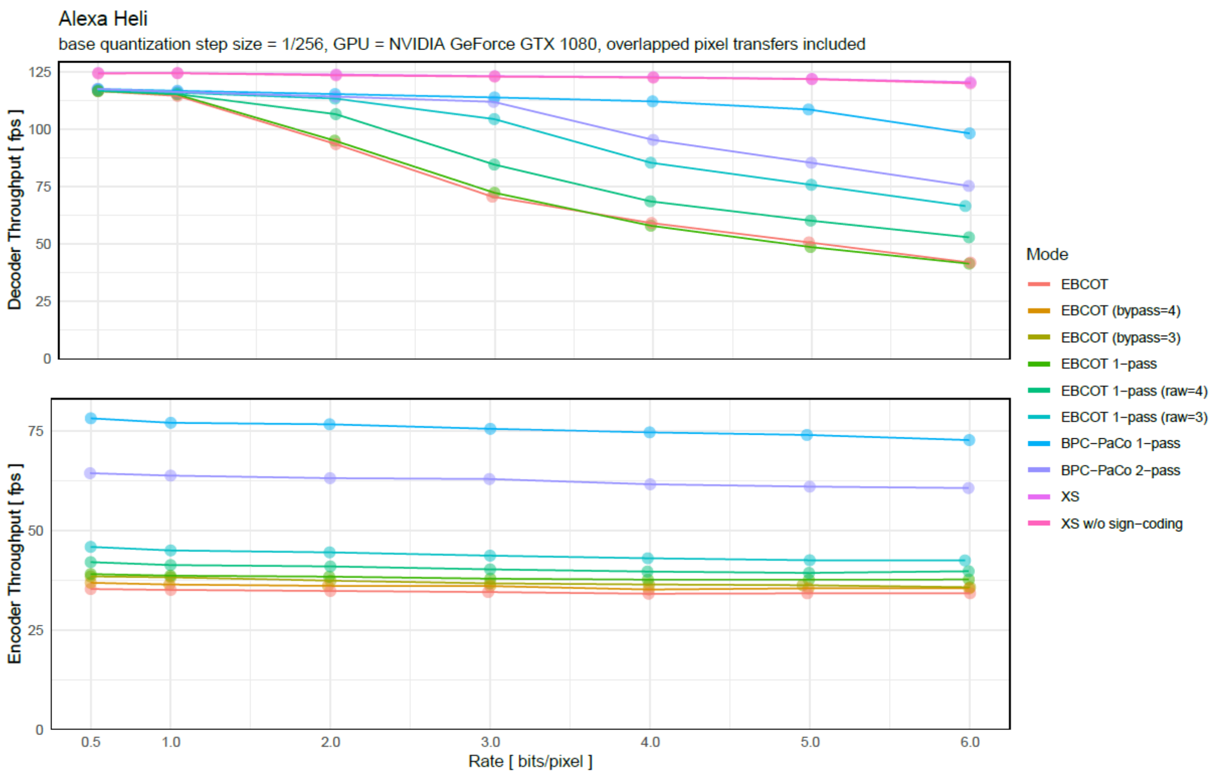
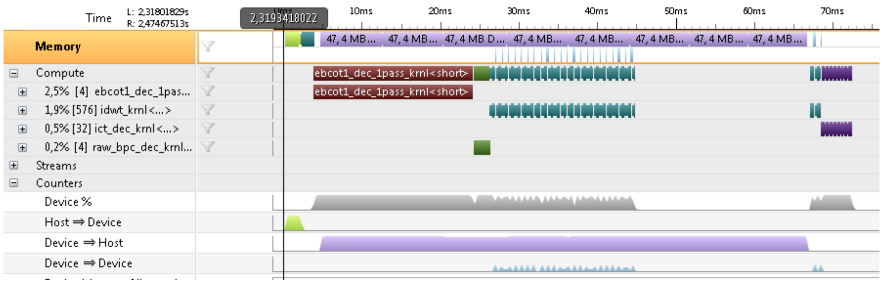


Figure 11-8 - Encoding and decoding throughput of all coding modes for the *Alexa Heli* sequence encoded with a base quantization step size of 256^{-1}. The decoding throughput bottleneck is the device-to-host transfer of the previously decoded group-of-frames. Frames are reconstructed with 6 bytes per pixel, so that a single uncompressed frame has a size of close to 50 MB.

The reason for this limitation is of an implementational rather than a conceptual nature. The timeline of the executed kernels depicted in Figure 11-9 shows that the device-to-host transfer is largely overlapped with the computation kernels. In the case of the high bit rate of 6 bpp (bottom), the EBCOT decoding kernel alone takes longer than the transfer. The transfer time is completely hidden. For the low bit rate, however, the transfer takes longer than the computation kernels. However, for an unknown reason the CUDA scheduler only overlaps part of the wavelet kernels with the transfer and delays a few DWT kernels as well as all color-transform kernels until after the transfer finished even though they do not depend on the transferred data. Possibly, the scheduler queue's depth is too low.



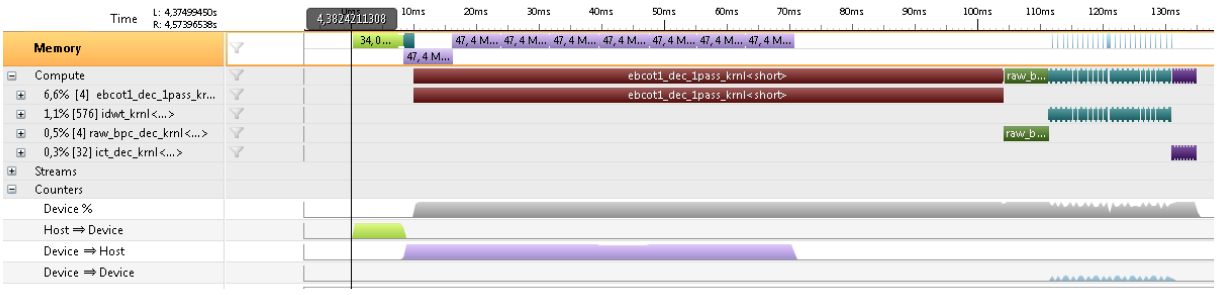
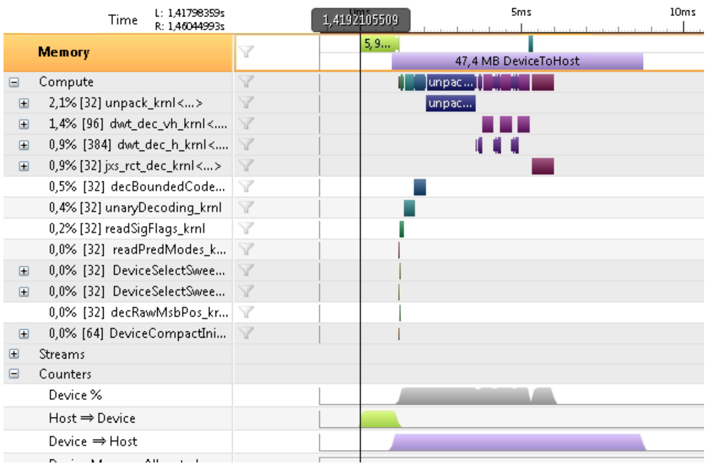


Figure 11-9 – NVIDIA Nsight timeline for JPEG 2000 decoder for the *Alexa Drums* UHD sequence encoded with 1 bpp (**top**) and 6 bpp (**bottom**). The decoder process 8 frames in parallel and concurrently transfers the previously decoded 8 frames from device into host memory (pink blocks, 47,4 MB per frame).

For the JPEG XS decoder, this problem does not exist (see Figure 11-10). Even if the transfer poses the bottleneck, the kernels are entirely overlapped and the only additional overhead on top of the device-to-host transfer is the host-to-device transfer of the current compressed frame. A *GeForce* card does not overlap host-to-device with device-to-host transfers. In NVIDIA GPU products, this functionality is limited exclusively to the professional *Quadro* series.



Time
L: 1,41798359s
R: 1,46044993s
1,4192105509
5ms
10ms
Memory
5, 9...
47,4 MB DeviceToHost
Compute
2,1% [32] unpack_krnl<...>
1,4% [96] dwt_dec_vh_krnl<...
0,9% [384] dwt_dec_h_krnl<...
0,9% [32] jxs_rct_dec_krnl<...>
0,5% [32] decBoundedCode...
0,4% [32] unaryDecoding_krnl
0,2% [32] readSigFlags_krnl
0,0% [32] readPredModes_k...
0,0% [32] DeviceSelectSwee...
0,0% [32] DeviceSelectSwee...
0,0% [32] decRawMsbPos_kr...
0,0% [64] DeviceCompactIni...
unpac...
Streams
Counters
Device %
Host ⇒ Device
Device ⇒ Host

Figure 11-10 – NVIDIA Nsight timeline for JPEG XS decoder for the *Alexa Heli* UHD sequence. While decoding the current frame, the previously reconstructed frame is transferred from GPU to host memory concurrently. In this example, this transfer poses the overall bottleneck.

Considering use cases, where a transfer of the reconstructed frames back into host memory is not required, it is nonetheless worthwhile to examine the throughput when only considering the host-to-device upload as well as the computation and disregarding the device-to-host transfer of reconstructed frames. Figure 11-11 is identical to the plot in Figure 11-8, except that now the time-limiting transfer is excluded. The decoding throughput is now no longer limited for the lower bit rates. The decoding throughput is highest for the lowest bit rate (0.5 bpp) and then decreases logarithmically. For 1 bpp and higher, the JPEG XS decoder is the fastest. The BPC-PaCo coders are the fastest JPEG 2000 variants. The difference to JPEG XS increases from 25 fps at 1 bpp to almost 75 fps at 6 bpp, i.e. the JPEG XS decoder is less affected by the increasing bit rate compared to the JPEG 2000 coders.

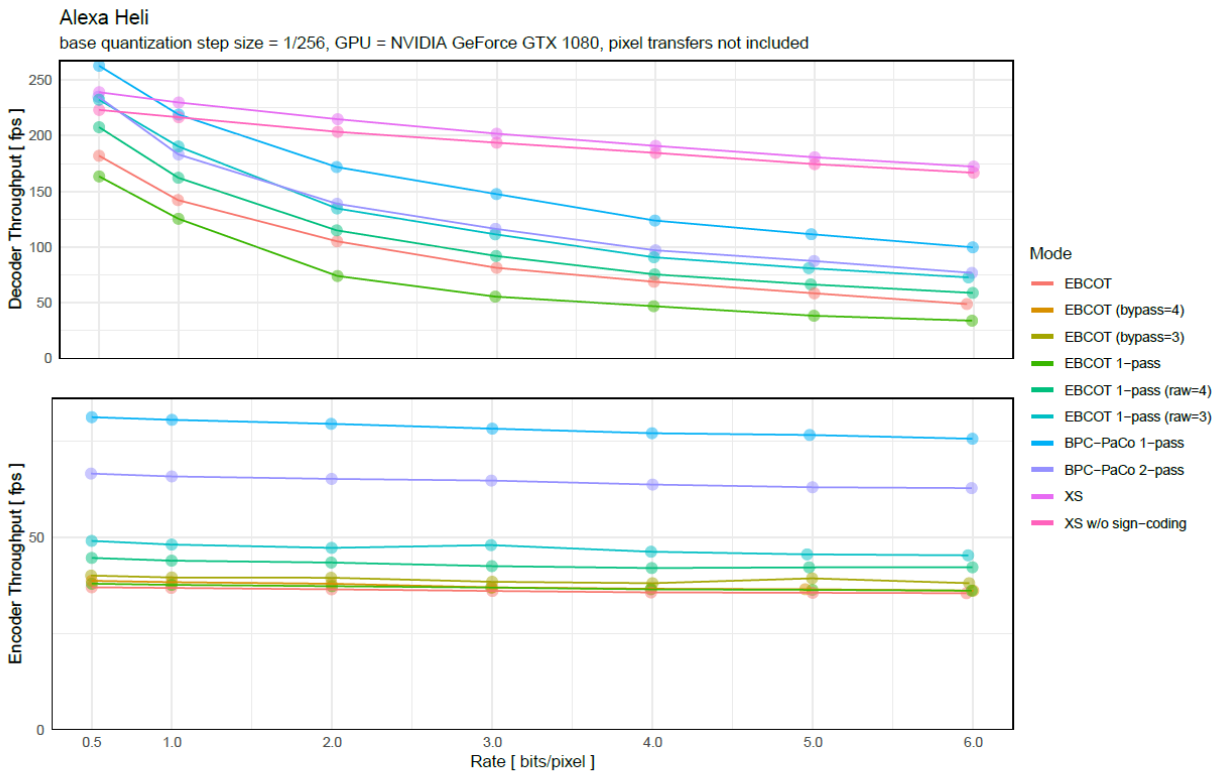


Figure 11-11 - Encoding and decoding throughput of all coding modes for the *Alexa Heli* sequence encoded with a base quantization step size of 256^{-1}. Measurements only include the code stream upload and computation, not the device-to-host transfer of reconstructed images.

As discussed previously in chapter 11.1, the encoder throughput is almost not affected by the selected target bit rate as long as the quantization step size remains unchanged. For lower bit rates, bit planes will still be time-consumingly encoded first by the block-coder only to be dropped in the PCRD-Opt. stage. Figure 11-12 shows the *Alexa Heli* measurement again, this time with a dynamically selected quantization step size. Since the measurements have only been carried out for three base quantization step sizes, the encoding throughput curves have three steps. This effect could be alleviated by introducing further quantization step sizes. At 0.5 bpp, the fastest mode, BPC-PaCo 1-pass, achieves now around 110 fps compared to 80 fps, because for this image at the low bit rate, it is safe to increase the base quantization step size from $\frac{1}{256}$ to $\frac{1}{64}$.

Figure 11-13 shows he results of the same measurement for the *Crowd Run* sequence. The fact that this sequence contains the most details of all the test sequences is reflected in the significantly lower encoding times. Compared to throughputs ranging from 70 fps to 110 fps at 0.5 bpp for the *Alexa Heli* sequence, the encoders reach only 40 fps to 85 fps for the more complex *Crowd Run* sequence. The order of the coding modes is consistent, though. Whereas for the *Alexa Heli* sequence, the quantization needs to be decreased already between 1 bpp and 2 bpp, for *Crowd Run* the strong quantization only becomes limiting between 5 bpp and 6 bpp. Likely, the quantization could have been decreased even further beyond 64^{-1} for the low bit rates for this test sequence.

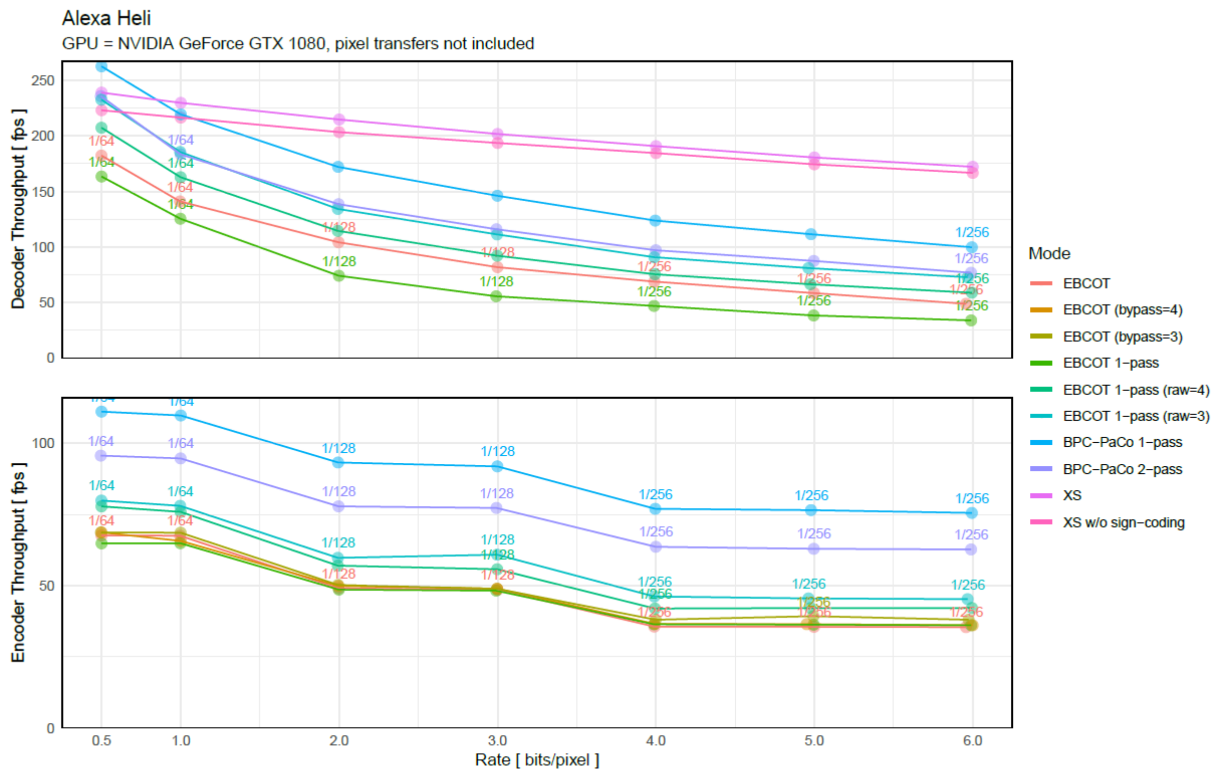


Figure 11-12 - Encoding and decoding throughput of all coding modes for the *Alexa Heli* sequence encoded with the appropriate base quantization step size of $\{64^{-1}, 128^{-1}, 256^{-1}\}$. Measurements only include the code stream upload and computation, not the device-to-host transfer of reconstructed images.

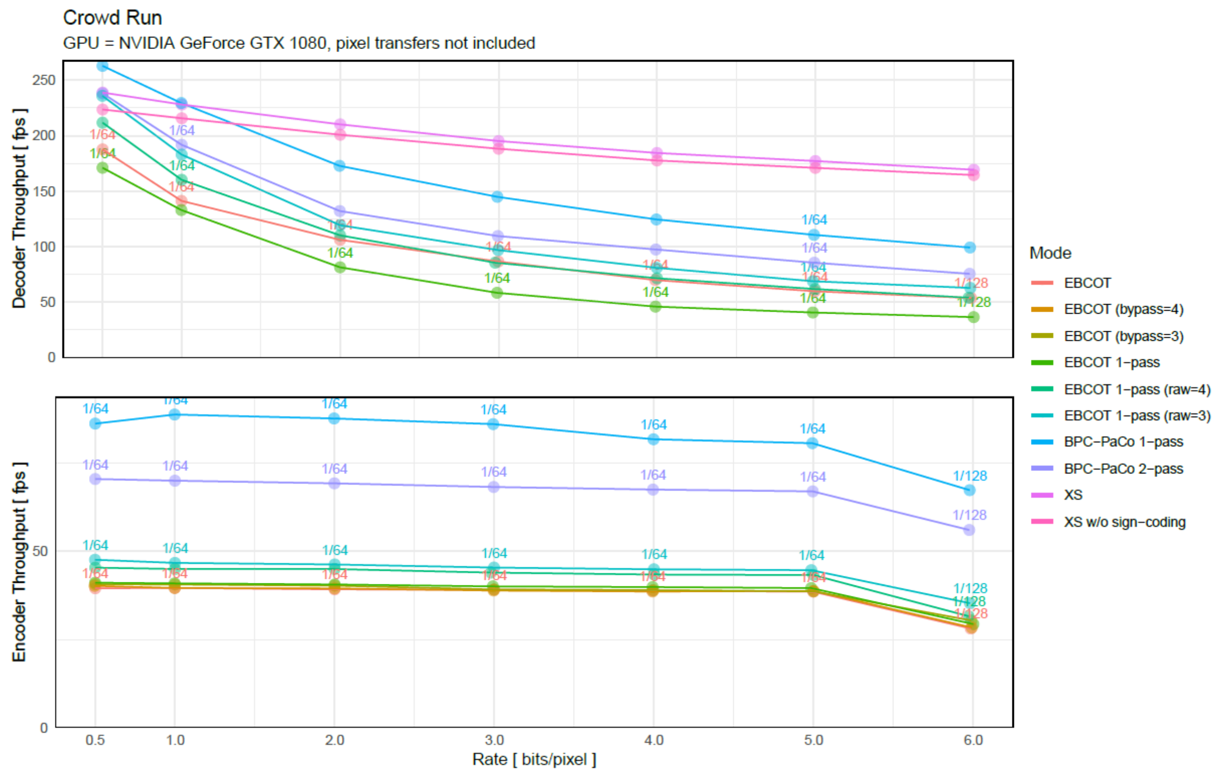


Figure 11-13 - Encoding and decoding throughput of all coding modes for the UHD *Crowd Run* sequence.

The decoding throughput for the *Alexa Heli* and *Crowd Run* sequences is largely identical. As long as the encoder exhausts the available bit rate budget this is to be expected, because then the workload for the decoder will be comparable and mostly determined by the resolution (which is UHD in both cases).

The *StEM Confetti* sequence also contains many details, but only has a resolution of 2K. Figure 11-14 and Figure 11-15 show the encoding and decoding results with and without taking into account the time required for the device-to-host transfer. For a resolution of 2048 x 857 at 6 bytes per pixel, a single uncompressed frame has a size of 10 MB. The transfer limits the decoding throughput at just over 500 fps. The decoder is not affected by the transfer limit as even the fastest coding mode achieves a throughput below this limit. The order of how the coding modes perform is consistent with the previously discussed test sequences.

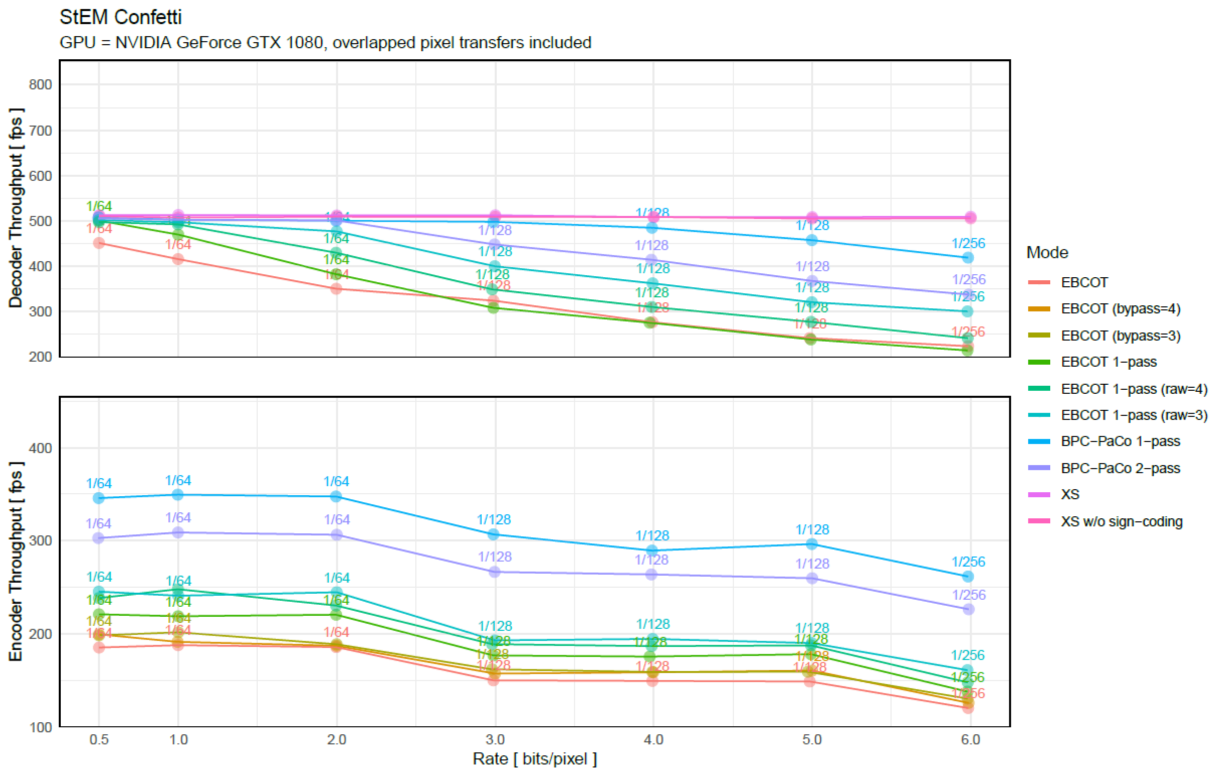


Figure 11-14 - Encoding and decoding throughput of all coding modes for the *StEM Confetti* sequence. The decoding throughput bottleneck is the device-to-host transfer of the previously decoded group-of-frames.

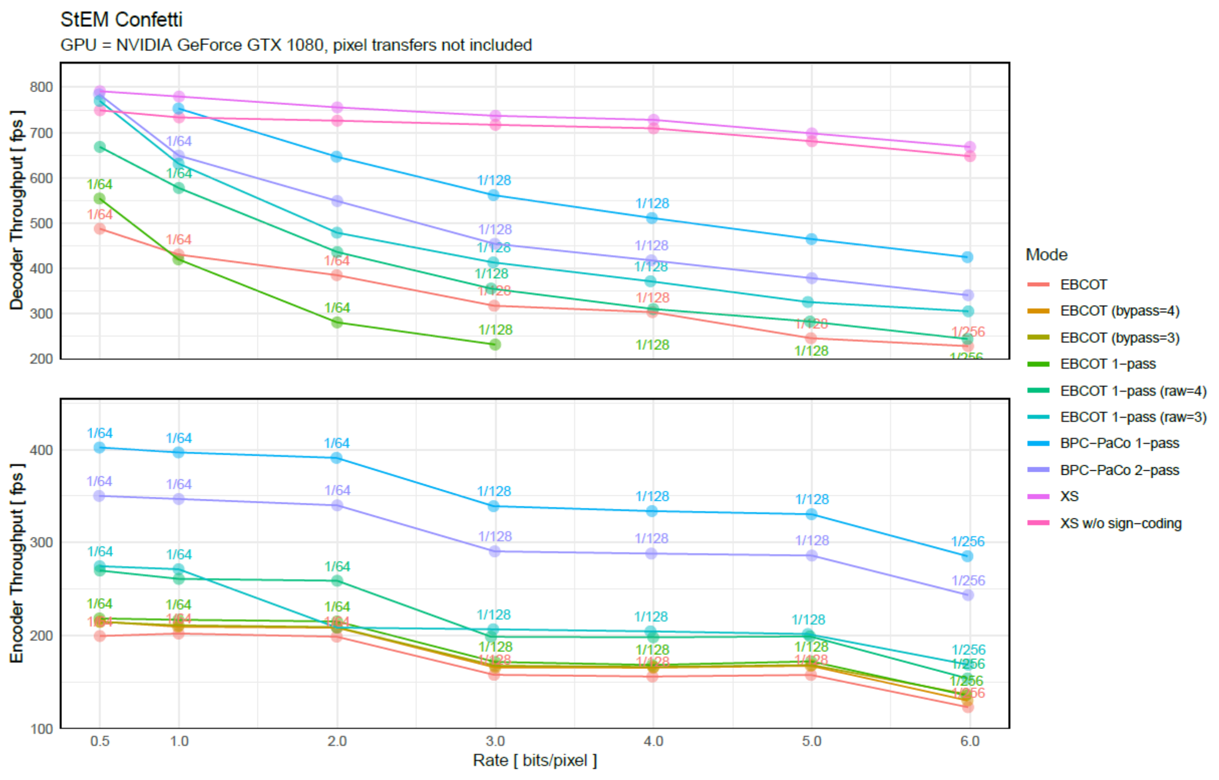


Figure 11-15 - Encoding and decoding throughput of all coding modes for the *StEM Confetti* sequence. Measurements only include the code stream upload and computation, not the device-to-host transfer of reconstructed images.

11.4 Case Study: Compressed Video Transmission

This chapter discusses the scenario of transmitting video over a channel with limited bandwidth, e.g. uploading a video stream into the cloud. The question is whether it is beneficial to compress video frames before sending them.

It is assumed that the scenario is realized in a streaming fashion, where the following processing steps are executed concurrently:

1. Decompression of frame N-1
2. Transmission of frame N
3. Compression of frame N+1

The overall throughput will thus be equal to the lowest of the three individual processing steps. We will assume that the following conditions holds:

$$\textit{Compression Throughput} < \textit{Decompression Throughput}$$

As a result, the decompression throughput can be disregarded and either the compression or the transmission will pose the bottleneck. The situation can then be modeled using the following variables:

- Compression throughput β_{enc} [fps]
- Available transmission bandwidth B [Mbit/sec]
- Video quality Q [dB PSNR]

- Image resolution $W \, x \, H$ [pixels]
- Uncompressed pixel size P_{raw} [bpp]
- Compressed pixel size P_{enc} [bpp]

Compressing the frames prior to transmission leads to an increased throughput when the compression duration is lower than the transmission duration:

$$t_{enc}\,[sec] < \frac{S_{raw}[MB]}{B\left[\frac{MB}{sec}\right]}$$

where

$$t_{enc}\,[sec] = \frac{1000}{\beta_{enc}\,[fps]} \quad and \quad S_{raw}[MB] = \frac{W * H * P_{raw}[\frac{bit}{pixel}]}{1024 * 1024}$$

Figure 11-16 plots for the *Alexa Drums* sequence and a given desired quality of 48 dB PSNR the achievable total throughput over a range of available transmission bandwidths.

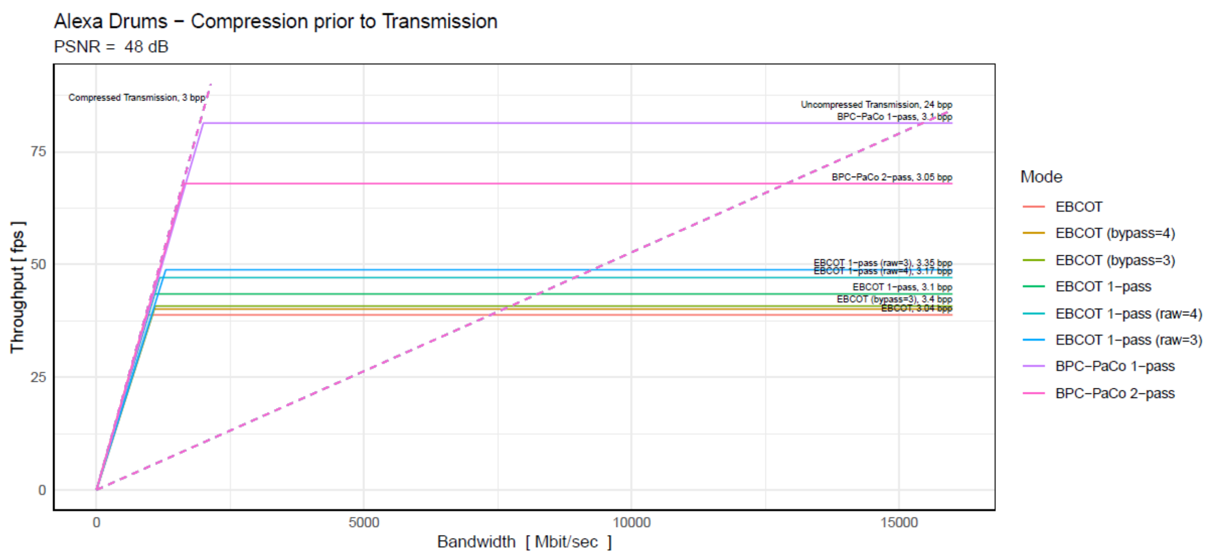


Figure 11-16 - Overview chart of achievable transmission throughput with the various coding modes for the *Alexa Drums* sequence, when the requirement is a quality of 48 dB PSNR. A single frame has an uncompressed size of 3840 x 2160 x 48 bit = 48 MB

The above diagram gives an overview across a wide transmission channel bandwidth range up to 15 Gbit/s. The flat dashed line shows the achievable throughput in frames/second when transmitting the uncompressed UHD sequence with 24 bit per pixel (3840 x 2160 x 24 bit = 48 MB). The solid lines show the achievable total throughput when compressing the frames before transmission. At low bandwidths where the solid curves rise linearly, the throughput is limited by the transmission and not by the compression. Oppositely, the flat plateau portions of the curves indicate the bandwidth range where the total throughput is limited by the encoder throughput. The location of the break-even points vary for the coding modes, as they require different bit rates for achieving the desired quality and yield different encoding throughputs. The required bit rates lie between 3.04 bpp for EBCOT and 3.5 bpp for the BPC-PaCo 1-pass. As a reference, the steeper dashed line shows the achievable throughput for transmitting frames compressed at 3 bpp, disregarding any bottleneck posed by the encoder.

The intersection points between the solid lines and the flatter dashed line indicates the upper limit where compression is still beneficial. If the bandwidth of the transmission channel rose beyond this point, transmitting the uncompressed video would yield a higher throughput. Given an available bandwidth of 10 Gbit/s, compressing the video with the original EBCOT mode first would limit the achievable throughput at ~28 fps and in fact transmitting the frames uncompressed would increase the throughput to a little over 50 fps. By employing the BPC-PaCo 1-pass codec, the throughput can be increased to a little over 80 fps while decreasing the bandwidth usage from 10 Gbit/s to only ~2300 Mbit/s (point where BPC-PaCo 1-pass curve plateaus).

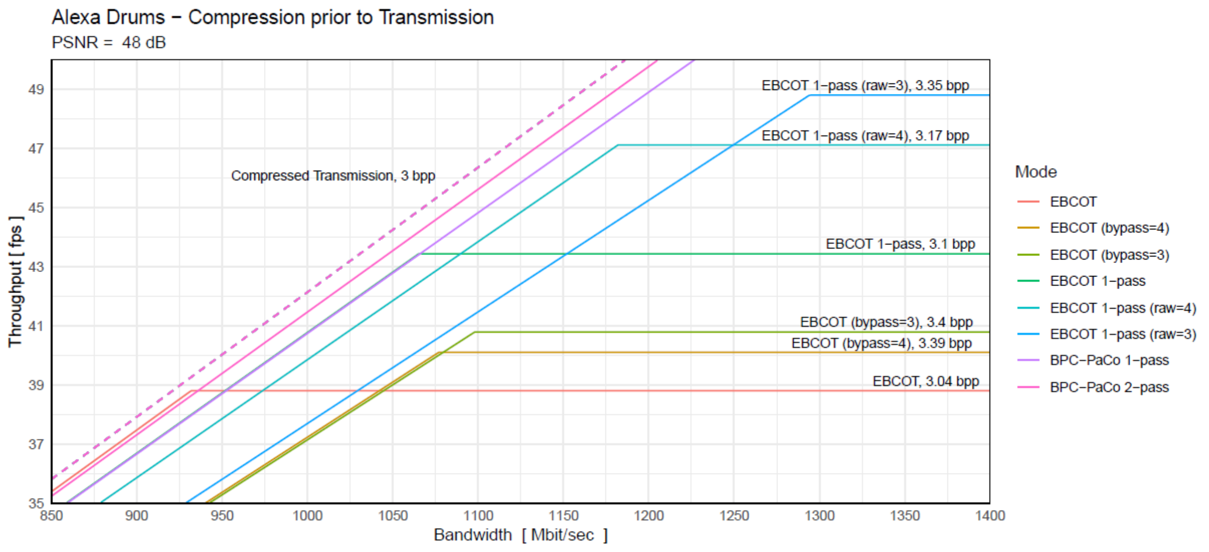


Figure 11-17 – Zoom-in on the a portion of the previous plot

Figure 11-17 zooms in at the region of the plot where the throughput transitions from being transmission-bound to being compression-bound. If the transmission bandwidth were limited to 1000 Mbit/s, for example, BPC-PaCo 2-pass outperforms BPC-PaCo 1-pass slightly, because it compresses the images better (3.05 bpp instead of 3.5 bpp), which leads to a minimal increase of the total throughput from 40.8 fps to 41.5 fps. For bandwidths below 930 Mbit/s, employing the original EBCOT instead is the best choice, albeit by a very small margin.

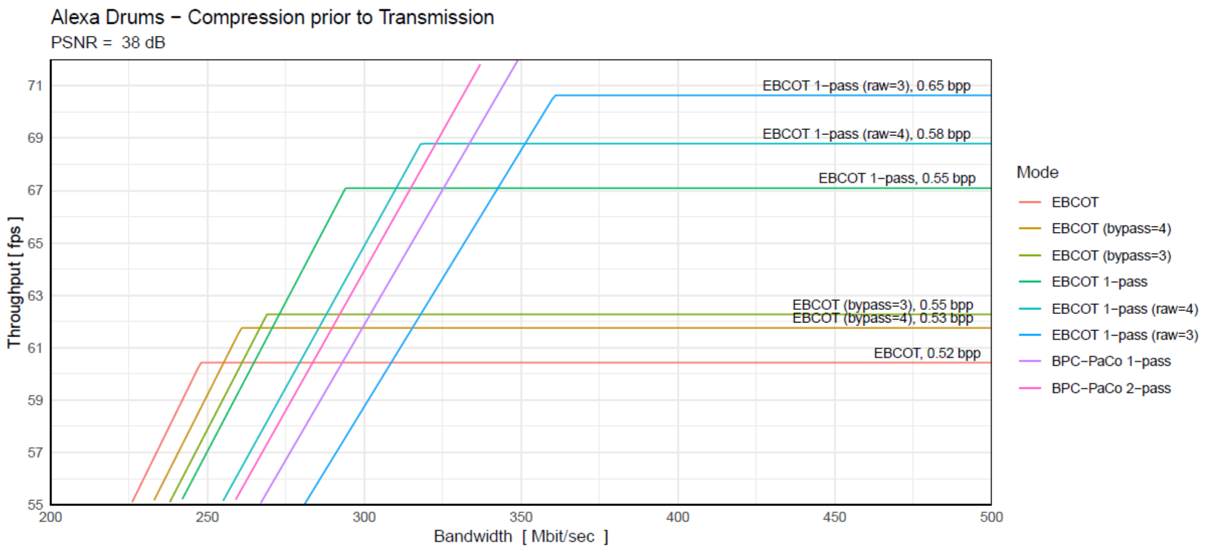


Figure 11-18 – Transmission of *Alexa Drums* sequence with a desired quality of 38 dB

When the video quality is lowered from 48 to 38 dB PSNR, the performance differences at the low-bandwidth end increase (see Figure 11-18). The most appropriate codec given for an available bandwidth is summarized in the following table.

Table 11-1 – optimal codec choice by available transmission channel bandwidth when transmitting the *Alexa Heli* sequence with 38 dB PSNR. Rows in grey font do not provide a significant benefit and could be omitted.

Available Bandwidth B [Mbit/sec]	Codec	Throughput [fps]	Compressed Bit rate [bpp]
0 < B < 247	EBCOT	60.42	0.518
247 < B < 260	EBCOT (bypass=4)	61.76	0.533
260 < B < 269	EBCOT (bypass=3)	62.28	0.546
269 < B < 294	EBCOT 1-pass	67.1	0.554
294 < B < 318	EBCOT 1-pass (raw=4)	68.78	0.584
360 < B < 448	BPC-PaCo 2-pass	95.67	0.593
448 < B < 511	BPC-PaCo 1-pass	105.41	0.612

All codecs, except the “EBCOT 1-pass (raw=3)”, are optimal in certain bandwidth ranges. This shows that it is advantageous to have access to a wide range of codecs that can be carefully tuned to a given scenario – available bandwidth and desired quality in this case study. However, as some of the windows are very small, some of the evaluated coding modes only provide a negligible advantage and could be omitted. The coding modes that do not provide a significant benefit in this scenario are indicated by a grey text color.

12 Summary

This work presents novel high quality intra-frame image compression approaches tailored for an execution on a massively parallel device architecture – in this case on a GPU. The focus is put on evaluating and fine-tuning the inherent trade-off between compression efficiency and computational complexity, subject to the constraints posed by the architecture of parallel devices.

Concretely, a majority of this work is concerned with the JPEG 2000 framework. An already heavily optimized implementation of the DCI, IMF and broadcast profiles for the NVIDIA CUDA GPU architecture represents the starting point. The bottleneck is JPEG 2000's entropy coder – the *Embedded Block Coder with Optimized Truncation* (EBCOT).

Chapter 9 presents a runtime model that permits to predict, based on benchmarking metrics measured for a particular kernel on one GPU, the runtime on another GPU. In two case studies, the original JPEG 2000 compliant implementations of the context modeling and arithmetic-coding kernels are evaluated. The workload for both kernels depends very strongly on the signal statistics of the source image and so a prediction that is not dependent on the benchmark results is infeasible. The benchmarking results confirm that the arithmetic coding block, which does not allow for samples or small groups of samples to be processed in parallel, but requires for all context-decision pairs from an entire code block's bit plane be processed serially, cannot be executed at full efficiency on a GPU. A strategy for increasing the overall throughput must be to break or reduce the long causal dependency chains that occur when encoding a long bit stream with a context-adaptive entropy coder.

Though not permitted in the listed profiles, JPEG 2000 part-2 already contains an optional block coding mode for reducing the computational complexity, namely the *Selective Arithmetic Coding Bypass mode*. Starting after three or four bit planes, symbols from two of the three bit plane pass types are no longer arithmetically entropy-coded, but instead appended directly to the bit stream. *Can an implementation of EBCOT that targets parallel architectures such as GPUs benefit from the more finely grained parallelism exposed by this coding style and thereby increase its throughput?* While the cost in terms of a reduced compression efficiency is very limited, the gain for the throughput is also limited. When coding only a single frame at a time, a speedup by a factor of two can be achieved, but when coding multiple frames at once in order to better utilize the GPU, the speedup decreases to only a few percent. The reason is that even for the lower bit planes the mode demands context modeling and iterating over the bits in multiple passes.

Consequently, chapter 7 introduces more profound changes to the block coder to address these limitations. On the one hand, a true *raw-coding mode* that does not require context modeling is introduced. It can be activated after an arbitrary amount of entropy-coded bit planes. A sample-parallel algorithm for appending magnitude bits as well as sign bits to a code block's bit stream is proposed. On the other hand, a simpler scan order for reducing the runtime of the entropy-coded bit planes is devised. Iterating over all samples in a bit plane in multiple passes increases the amount of memory accesses needlessly given that the most significant bit planes are unlikely to be truncated in a visually lossless operating domain with sufficient bit rate budget. The single-pass mode gives up the concept of intra-bit plane truncation points. *How much of a speed-up can be achieved? What is the cost in terms of a diminished compression efficiency?* The single-pass mode alone yields speed-ups between 1.2x and 1.5x for the encoder and 1.05x to 1.3x for the decoder at virtually no decrease in compression rate. The raw-coding mode yields considerable encoding accelerations of around 50 fps for 2K/HD sequences and 10 fps for the 4K/UHD sequences. The raw-coding mode allows for a high degree of parallelism as individual samples can be processed in parallel. This benefit weighs in more heavily for lower resolution sequences as there the relative increase in parallelism compared to the regular coding mode is higher than for 4K/UHD sequences which have four times as many samples and in turn four times as many independent work units to begin with.

Researchers from the *University of Barcelona* have released an alternative block coder during the course of this work, named *BPC-PaCo*. *How does BPC-PaCo compare to the other coding modes presented in this work?* The block coder gives up the context adaptivity in favor of a static probability model, which in turn allows for a highly parallel implementation that processes individual samples in parallel. Chapter 8 evaluates the block coder in detail. *What are the drawbacks and advantages in terms of compression efficiency and throughput?* A fair comparison is achieved by incorporating the block coder into the same JPEG 2000 framework as the other coding modes. Measurements have shown that the PSNR for any given bit rate between 0.5 and 6 bpp lies no more than 1 dB below that achieved with the original EBCOT coding mode. The achieved encoding speedup lie between 1.35x (*Sintel Title*, fewest details) and 2.15x (*Crowd Run*, most details). The decoder speed up is lower and ranges from 1.05x (*Sintel Title*) to 1.4x (*Crowd Run*) (refer Figure 8-14). *How strongly are the results affected by the choice of the static probability map created offline?* Since the static probability model requires a priori knowledge about the sequences to be coded, an experiment is conducted that evaluates the impact of using various probability maps on the coding efficiency. It shows that a model averaged over all test sequences performs reasonably well. The maximum PSNR difference with respect to the optimal probability table created especially for the respective test sequences is 0.5 dB. *Can the coding mode be further optimized for the use case of visually lossless image compression?* A combination of *BPC-PaCo* with a single-pass scan order is proposed: *BPC-PaCo 1-pass*. It outperforms the original *BPC-PaCo 2-pass* mode, while staying within 0.3 dB PSNR. The encoder speedup range increased from 1.35x-2.15x to 1.5x-2.6x, the decoder speedup from 1.05x-1.4x to 1.25x-1.9x (refer Figure 8-14). To give an example, the compression of the *Crowd Run* sequence is increased from 30 fps to 65 fps by replacing EBCOT with *BPC-PaCo 2-pass* and to 80 fps by using the proposed *BPC-PaCo 1-pass* modification. Decoding at 0.5 bpp, it is increased from 180 fps (EBCOT) to 225 fps (*BPC-PaCo 2-pass*) to 250 fps (1-pass) and at 6 bpp from 50 fps (EBCOT) to 75 fps (2-pass) to 95 fps (1-pass) (refer Figure 8-12).

Importantly, with JPEG XS another coding framework outside of JPEG 2000 is additionally discussed, evaluated and compared against the JPEG 2000 based coding modes. The JPEG XS standard has been developed during the course of this work with involvement of Fraunhofer IIS, who has won the call for proposals alongside Belgian company *IntoPix* who submitted their *TICO* codec. The development of the CUDA-based JPEG XS decoder started based on early specifications and one of the main reasons to start early was to ensure that the codec's design exposes sufficient parallelism to leverage the full potential of a GPU architecture. *Has the JPEG XS design goal of making the decoder fully parallelizable been achieved? What throughput is achievable on current high-end and mid-range GPUs?* A mid-range *GeForce GTX 750 Ti* with 640 cores decodes a UHD (HD) 4:2:2 sequence with 54 fps (190 fps). The fact that a high-end *GeForce GTX 1080* with 2560 cores achieves a throughput of 190 fps for UHD and 600 fps for HD shows that the codec scales well and exposes sufficient parallelism. When reconstructing an accuracy of 48 bit per pixel, transferring decoded frames back into host memory over PCI-E 2.0 x16 poses a bottleneck of 190 fps for 4:2:2 and 130 fps for 4:4:4. *Does the use of a GPU as a hardware platform in any way limit the use cases of a JPEG XS decoder?* One of the capabilities of the JPEG XS standard is its low sub-frame latency that makes it an ideal codec also for live-production or KVM-extension scenarios. An implementation capable of achieving the sub-frame low-latency would process images row by row or rather packet by packet in a pipelined fashion. This, however, would not yield an amount of parallel jobs sufficient to load all the hundreds or thousands of cores available on a modern GPU. The decoder proposed here operates on an entire frame and transfers the previously decoded frame back to the host memory concurrently. Compared to the JPEG 2000 coder implementations that batch multiple frames together in order to increase the amount of parallelizable EBCOT blocks, the JPEG XS decoder implementation has a significantly lower latency. However, it does not fully utilize the sub-frame latency feature available in JPEG XS.

13 Conclusion

Can the throughput be increased considerably while keeping the cost of decreased compression efficiency at a minimum? Yes, all the proposed coding modes yield a relative speedup at a marginal quality loss. Figure 13-1 plots the change in PSNR over the speedup relative to the original EBCOT mode ("1x") for all sequences and all coding modes for a bit rate of 3 bpp. It can be observed that the point cloud spreads horizontally, indicating that the speedup clearly outweighs the quality loss.

The recommended JPEG 2000 variant is BPC-PaCo with a single-pass scan order. With respect to the author's GPU-accelerated standard JPEG 2000 implementation, it speeds up encoding by a factor of two and decoding by a factor of 1.5 in most cases. The quality loss as indicated by a decrease of the PSNR is very limited (see Figure 13-1).

The JPEG XS decoder is even faster – its relative speed up grows towards higher bit rates up to a factor of 3x - 4x for UHD sequences coded at 6 bpp (Figure 11-11). This comes at the cost of a lower compression efficiency, though. For a given bit rate, the achieved quality in PSNR is often 1-2 dB lower than the proposed EBCOT variants.

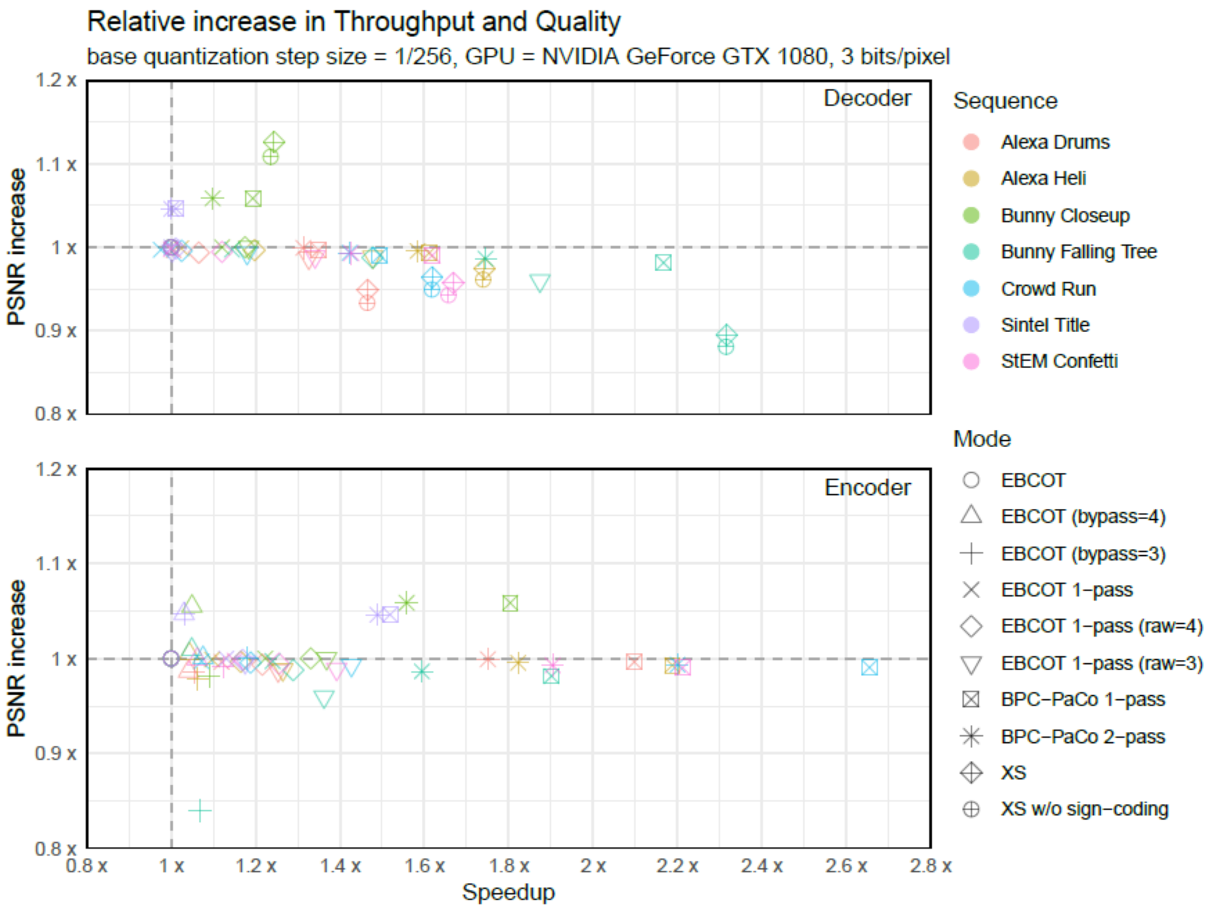


Figure 13-1 – Relative increase in throughput and quality at 3 bpp

Is there much to gain by permitting these features in the JPEG 2000 profiles employed by cinema and broadcast industry? Especially in contexts, where a visually lossless quality is demanded and relatively high bit rates are used, the default EBCOT entropy coder has a poor tradeoff between complexity and quality. At high bit rates, it attempts to code information even in the least significant bit planes where the entropy coder's assumption about a skewed probability estimation function do not hold anymore. Since the profiles state a maximum bit rate irrespective of the resolution and frame rate, the problem is particularly evident at low resolutions and frame rates such as a 2K movie with 24 fps, which is still the most common cinema format today.

Extending beyond the scope of the JPEG 2000 standard, how can the compression scheme be altered when giving up compatibility? Can the throughput bottleneck be remedied by designing a drop-in replacement for the complex entropy coder? Yes, various drop-in replacements for EBCOT have been evaluated. The coding modes benefit from three alterations:

1. only the first few bit planes are entropy coded. Lower bit planes that will not get compressed as efficiently anyway are coded raw
2. intra-bit plane truncation points are sacrificed. A single-pass mode decreases the complexity and required amount of memory accesses
3. BPC-PaCo sacrifices context adaptivity, introducing instead a fixed probability model that is computed offline and transported as side-information. The map contains probabilities for each band, bit plane, coding operation and context.

They all maintain the embeddedness property and still allow a post-compression distortion that is key to utilizing the available data budget down to the last bytes without multi-pass encoding.

How does the requirement to expose as much parallelism as possible to better leverage massively parallel architectures impact the design of a new low complexity codec? The degree of exposed parallelism is an important metric that should be addressed in future codec designs. For the first time, this has been an explicit requirement in the JPEG XS call for proposals. In this example, especially the prediction mode is a technique that stands in the way of finely grained parallelism. Long vertical causal prediction chains are interrupted at the slice boundary. They effectively present sync points and enable a decoder to process slices in parallel. The idea of intra-precinct horizontal prediction has been discarded in the end, enabling a decoder to process all samples in a precinct in a sample-parallel fashion.

In JPEG 2000 EBCOT's default coding mode, the causal dependency chain stretches from a code block's top-left position in the most significant bit plane's first pass to the least significant bit plane's bottom right position in the last pass. Consequently, the unit of parallelism in the decoder is an entire 32x32 EBCOT block, which contradicts the lightweight threading model in a GPU that is designed to enable sample-parallel processing. The JPEG 2000 encoder can somewhat mitigate this problem by decoupling context modeling and arithmetic coding. At least the context modeling routine can be implemented sample-parallel, but the arithmetic coder's context adaptiveness requires that all symbols from an entire code block are processed sequentially, forcing a thread layout where a single thread is in charge of an entire code block containing up to 1024 samples. Such circumstances must be avoided in GPU-friendly low complexity codecs.

Furthermore, the embeddedness property in JPEG 2000 requires that samples are processed bit-plane-wise. This increases the number of computations and memory accesses. In JPEG XS, on the hand, samples are coded as a whole.

In summary, it can be concluded that it is advantageous to constrain a new codec to a narrow bit rate range in order to be able to select an appropriate amount of coding tools. The complexity required to achieve good visual results at very low bit rates needlessly slows down a codec when used at higher bit rates. Additionally, long dependency chains must be avoided. High-end GPUs today already have over 4000 cores and the number can be expected to grow further. In order to hide latencies, the GPU's scheduler prefers a massive oversubscription in terms of the number of spawned threads. Future GPU-friendly codecs should therefore ensure that a single frame exposes multiple thousands concurrently processable coding units.

13.1 Contributions

The novelties introduced in this work and the contributing publications are as follows:

JPEG 2000 part-2 Selective Arithmetic Bypassing Mode for a GPU: A sample-parallel algorithm for executing the significance-propagation pass magnitude-refinement pass is presented and multiple concepts for organizing the passes in GPU kernels are proposed and evaluated.

Parallel Bit Stuffing Algorithm: A sample-parallel bit stuffing algorithm that utilizes efficient thread-group internal voting instructions specific to the CUDA language was devised.

EBCOT single-pass scan order for GPU: The concept of sacrificing intra-bit plane truncation points in favor of increased throughput is formulated and an execution algorithmic is proposed. The effect on compression efficiency and runtime are evaluated.

Parallel EBCOT raw-coding mode: Motivated by the fact that at medium to high bit rates the EBCOT compression efficiency at less significant bit planes drops to the point that bit planes are occasionally expanded and not compressed, a raw-coding mode is formulated. In comparison to the selective arithmetic bypass mode, it deliberately sacrifices context modeling in order to lower the computational complexity further. It accounts for sign bits that might not have been coded yet during the arithmetically coded most significant bit planes.

Parallel JPEG 2000 packetization: When all other coding steps, even EBCOT, are heavily optimized, the *Post-Compression Rate-Distortion Optimization* (PCRD-Opt.) and subsequent packetization may become a bottleneck. The packetization routine including PCRD-Opt. is formulated in a parallel manner so that the completely assembled code stream can be downloaded from the GPU.

Runtime Prediction Model: Two exemplary kernels' benchmarking results are analyzed and it was found that it is not feasible to predict the runtime merely based on statistics of the input image that are available a priori. Instead, an approach is presented where the kernel is benchmarked on one GPU and its runtime is formulated in terms of various GPU and kernel dependent variables. By assuming the kernel dependent variables (workload and work efficiency) to be constant and merely exchanging the GPU-specific variables, the runtime can then be predicted for another GPU.

Additional evaluation of BPC-PaCo: BPC-PaCo with two or three passes per bit plane has been introduced in the literature during the course of this work. This work in turn contains a previously unpublished independent evaluation of the coder. Additionally, the impact of the static probability maps that are created offline a priori and are inferred from representative video sequences has been shown with the finding that a map created by averaging over multiple test sequences presents a feasible way.

BPC-PaCo single-pass scan order mode: The concept of the single-pass mode previously formulated for EBCOT has been transferred to BPC-PaCo. This opens up possibilities for algorithmic optimizations such as stenciling that result in an additional significant speedup.

JPEG XS GPU decoder: The JPEG XS standardization took place during the course of this work. The first JPEG XS GPU decoder was presented, including a comparison to the JPEG 2000 GPU decoder.

13.2 Limitations and Future Work

13.2.1 Limitations

NVIDIA CUDA: The experiments have been run with an implementation that uses *NVIDIA CUDA*. The implementations draw heavily on intrinsics and optimization guidelines for the CUDA architecture and porting the proposed algorithms to other frameworks such as OpenCL or Metal may not be straight-forward in all cases. The SIMD-group voting intrinsics are a prominent example.

13.2.2 Future Work

JPEG XS Encoder: As part of this work, an *NVIDIA CUDA* implementation of a JPEG XS decoder has been developed, but no encoder. Future work will be to implement a GPU-accelerated encoder as well and compare it carefully with the introduced JPEG 2000 based encoders.

JPEG 2000 High Throughput: As addressed in chapter 3.4.2, an initiative to standardize a lower-complexity block coder for JPEG 2000 has been started during the course of this work and was published as part 15 of the standard in October 2019. When encoder and decoder implementation are available, it would be interesting to compare them with the modes presented in this work.

Per-Block Raw-Coding mode: The raw-coding mode proposed in chapter 7.2 is signaled globally and activated after a certain bit plane has been reached. Instead, it might be beneficial to activate the raw-coding mode independently for each code block. This way the encoder could activate it precisely when the entropy coding efficiency falls below a given threshold.

BPC-PaCo with raw-coding: Raw-coding has been evaluated here in conjunction with the single-pass mode. Alternatively, it could be combined with the BPC-PaCo mode in order to see if the performance can been increased further.

2D+t DWT: It has been mentioned that the EBCOT kernels process code blocks from multiple frames in parallel in order to fully utilize all available GPU cores. The DWT could also leverage this by executing a wavelet kernel along the time axis in order to compact the present energy further into fewer coefficients. It is an interesting research question whether the time required to execute the DWT along the additional dimension is compensated for by the expected decrease in the EBCOT runtime.

Motion compensated 2D+t DWT: For best results, a motion-compensated wavelet should be employed [92]. It has been shown that the motion vectors from a HEVC video coder can be reused to predict the optical flow [93]. *NVIDIA GeForce* cards are equipped with AVC and HEVC ASICs that are accessible via the *NVIDIA* Codec SDK. This has a motion-vector-only mode that might be leveraged to speed up the runtime of a motion-compensated 2D+t wavelet.

Hardware Dependency: In this work, the algorithms were evaluated on a small set of *NVIDIA GeForce* GPUs. In chapter 9 the runtime on a mid- and high-end GPU within the same *GeForce* series as well as between GPUs from two series has been compared and closely examined. Beyond this, a more detailed investigation of a larger scale into how the algorithms perform on various hardware platforms would be worthwhile. In particular, it would be valuable to gain knowledge on how a codec's runtime performance scales as new hardware generations are released.

Automatic Implementation Optimization: Optimizing an implementation in terms of the achievable throughput is a very time consuming process. In this work, an iterative process has been followed where an implementation was profiled with the development tools provided by *NVIDIA* and then potential improvements have been derived, implemented and the process was repeated over. Future research can be spent on streamlining this process and gaining more objective results that are not subject to manual optimizations. An even more ambitious goal is to strive for a fully automatic optimization that translates a given algorithm for a given hardware, possibly at runtime.

Bibliography

[1] *SMPTE Standard ST 429-4 - D-Cinema Packaging — MXF JPEG 2000 Application*, Piscataway, NJ, USA.

[2] *ISO/IEC 15444-1 Information Technology - JPEG 2000 Image Coding System - Part 1: Core Coding System.* [Online]. Available: https://www.iso.org/standard/27687.html

[3] NVIDIA, *NVIDIA CUDA.* [Online]. Available: https://developer.nvidia.com/cuda-zone (accessed: Aug. 19 2018).

[4] *SMPTE Standard ST 2067-20:2016, Interoperable Master Format - Application #2,* SMPTE, Piscataway, NJ, USA.

[5] *SMPTE RDD 44 Material Exchange Format — Mapping and Application of Apple ProRes*, Piscataway, NJ, USA.

[6] Netflix Inc., *Netflix Originals Delivery Specifications 3.1: version OC-3-1.* [Online]. Available: https://partnerhelp.netflixstudios.com/hc/en-us/articles/214806618-Netflix-Originals-Delivery-Specifications-v3-1 (accessed: Aug. 19 2018).

[7] *ISO/IEC 15444-1 - Information Technology - JPEG 2000 Image Coding System - Part 1: Core Coding System: Amendment 1 - Profiles for Digital Cinema Applications,* ISO/IEC, 2005.

[8] *ISO/IEC 15444-1 - Information Technology - JPEG 2000 Image Coding System: Core Coding System - Part 1: Amendment 4 - Profiles for broadcast applications*, 15444-1:2004/Amd.4, ISO/IEC.

[9] *ISO/IEC 15444-1 - Information Technology - JPEG 2000 Image Coding System: Core Coding System - Part 1: Amendment 7: Profiles for an Interoperable Master Format (IMF),* ISO/IEC, 2015.

[10] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, “Image Quality Assessment: From Error Visibility to Structural Similarity,” *IEEE Trans. on Image Process.*, vol. 13, no. 4, pp. 600–612, 2004, doi: 10.1109/TIP.2003.819861.

[11] Netflix Inc., *Toward A Practical Perceptual Video Quality Metric.* [Online]. Available: https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652 (accessed: Jun. 2 2019).

[12] D. Taubman, “High Throughput JPEG 2000: New Algorithms and Opportunities,” in *IBC 2017.* Accessed: Aug. 27 2018. [Online]. Available: https://www.ibc.org/download?ac=3808

[13] Z. Wang and A. C. Bovik, “Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures,” *IEEE Signal Process. Mag.*, vol. 26, no. 1, pp. 98–117, 2009, doi: 10.1109/MSP.2008.930649.

[14] *High Efficiency Video Coding (HEVC) Text Specification Draft 9,* ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Picture Experts Group (MPEG), Oct. 2012.

[15] P. de Rivaz and J. Haughton, “AV1 Bitstream & Decoding Process Specification,” 2019. [Online]. Available: https://aomediacodec.github.io/av1-spec/av1-spec.pdf

[16] S. G. Mallat, “A theory for multiresolution signal decomposition: the wavelet representation,” *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 11, no. 7, pp. 674–693, 1989, doi: 10.1109/34.192463.

[17] I. Daubechies, “Orthonormal bases of compactly supported wavelets,” *Comm. Pure Appl. Math.*, vol. 41, no. 7, pp. 909–996, 1988, doi: 10.1002/cpa.3160410705.

[18] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado, “Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 3, pp. 299–310, 2008, doi: 10.1109/TPDS.2007.70716.

[19] NVIDIA, *CUDA Programming Guide: version 9.2.148.* [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed: Aug. 19 2018).

[20] *OpenCL - The open standard for parallel programming of heterogeneous systems.* [Online]. Available: https://www.khronos.org/opencl/ (accessed: Aug. 25 2018).

[21] *Intel® FPGA SDK for OpenCL™.* [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html (accessed: Aug. 25 2018).

[22] *Table Comparing Syntax for Different Compute APIs.* [Online]. Available: https://rocm.github.io/languages.html (accessed: Aug. 25 2018).

[23] *C++ AMP (C++ Accelerated Massive Parallelism).* [Online]. Available: https://msdn.microsoft.com/de-de/library/hh265137.aspx (accessed: Aug. 25 2018).

[24] *Apple Inc., Metal 2 - Apple Developer.* [Online]. Available: https://developer.apple.com/metal/ (accessed: Aug. 25 2018).

[25] Khronos Group, *OpenMP Application Programming Interface*, 2015. Accessed: Aug. 25 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[26] M. J. Harvey and G. de Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093–1099, 2011, doi: 10.1016/j.cpc.2010.12.052.

[27] G. Martinez, M. Gardner, and W.-c. Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures," in *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), 2011: 7-9 Dec. 2011, Tainan, Taiwan ; proceedings ; [including workshop papers]*, Tainan, Taiwan, 2011, pp. 300–307.

[28] J. Wu *et al.,* "gpucc: an open-source GPGPU compiler," in *Proceedings of CGO 2016, the 14th International Symposium on Code Generation and Optimization: 12-18 March 2016, Barcelona, Spain*, Barcelona, Spain, 2016, pp. 105–116.

[29] NVIDIA, *Inside Pascal: NVIDIA's Newest Computing Platform.* [Online]. Available: https://devblogs.nvidia.com/inside-pascal/ (accessed: Aug. 19 2018).

[30] PCI-SIG Administration, *Specifications.* [Online]. Available: https://pcisig.com/specifications/pciexpress (accessed: Mar. 18 2021).

[31] *CUB: Main Page.* [Online]. Available: https://nvlabs.github.io/cub/ (accessed: Aug. 27 2018).

[32] *moderngpu/moderngpu.* [Online]. Available: https://github.com/moderngpu/moderngpu/wiki (accessed: Aug. 27 2018).

[33] *Thrust - Parallel Algorithms Library.* [Online]. Available: https://thrust.github.io/ (accessed: Aug. 27 2018).

[34] H. Nguyen, *GPU gems 3*. Boston, Mass., London: Addison-Wesley, 2007.

[35] M. Harris, S. Sengupta, and J. D. Owens, *GPU Gems 3 Chapter 39. Parallel Prefix Sum (Scan) with CUDA.* [Online]. Available: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html (accessed: Aug. 25 2018).

[36] M. Harris, "Optimizing Parallel Reduction in CUDA," 2007. [Online]. Available: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

[37] J. Luitjens, *Faster Parallel Reductions on Kepler.* [Online]. Available: https://devblogs.nvidia.com/faster-parallel-reductions-kepler/ (accessed: Aug. 27 2018).

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

[39] J. Matela, V. Rusnak, and P. Holub, Eds., *GPU-Based Sample-Parallel Context Modeling for EBCOT in JPEG2000*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2011.

[40] R. Le, I. R. Bahar, and J. L. Mundy, "A novel parallel Tier-1 coder for JPEG2000 using GPUs," in *IEEE 9th Symposium on Application Specific Processors (SASP), 2011: 5-6 June 2011, San Diego, CA, USA ; proceedings*, San Diego, CA, USA, 2011, pp. 129–136.

[41] F. Wei, Q. Cui, and Y. Li, "Fine-Granular Parallel EBCOT and Optimization with CUDA for Digital Cinema Image Compression," in *2007 IEEE International Conference on Multimedia and Expo (ICME 2012): Melbourne, Australia, 9-13 July 2012*, Melbourne, Australia, 2012, pp. 1051–1054.

[42] A. Balevic, M. Heide, A. Weiß, S. Papandreou, and N. Fürst, *cuj2k: JPEG 2000 on Cuda.* [Online]. Available: http://cuj2k.sourceforge.net/index.html (accessed: Aug. 27 2018).

[43] *GPU JPEG2K: GPU-based implementation of JPEG2K standard*. http://apps.man.poznan.pl/trac/jpeg2k: Poznan Supercomputing and Networking Center. Accessed: Dec. 25 2018.

[44] ITU-T, *T.24: standardized digitized image set.* [Online]. Available: http://www.itu.int/rec/T-REC-T.24/en (accessed: Mar. 15 2021).

[45] T. Richter and S. Simon, "On the JPEG 2000 ultrafast mode," in *2012 19th IEEE International Conference on Image Processing*, Orlando, FL, USA, Sep. 2012 - Oct. 2012, pp. 2501–2504.

[46] J. Matela, M. Šrom, and P. Holub, "Low GPU Occupancy Approach to Fast Arithmetic Coding in JPEG2000," in *Lecture Notes in Computer Science*, vol. 7119, *Mathematical and engineering methods in computer science: 7th International Doctoral Workshop, MEMICS 2011, Lednice, Czech Republic, October 14-16, 2011: revised selected papers / Zdeněk Kotásek ... [et al.], (eds.)*, Z. Kotásek, Ed., Heidelberg, London: Springer, 2012, pp. 136–145.

[47] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Trans. on Image Process.*, vol. 9, no. 7, pp. 1158–1170, 2000, doi: 10.1109/83.847830.

[48] F. Aulí Llinàs, *Model-based JPEG2000 rate control methods*. Bellaterra: Universitat Autònoma de Barcelona, 2007.

[49] M. Á. Martínez del Amor, V. Bruns, and H. Sparenberg, "Parallel efficient rate control methods for JPEG 2000," in *Applications of Digital Image Processing XL: 7-10 August 2017, San Diego, California, United States / Andrew G. Tescher, editor*, San Diego, United States, 2017, p. 26. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10396/2273005/Parallel-efficient-rate-control-methods-for-JPEG-2000/10.1117/12.2273005.full

[50] F. Auli-Llinas, P. Enfedaque, J. C. Moure, I. Blanes, and V. Sanchez, "Strategy of Microscopic Parallelism for Bitplane Image Coding," in *2015 Data Compression Conference*, Snowbird, UT, USA, Apr. 2015 - Apr. 2015, pp. 163–172.

[51] F. Aulí-Llinàs and M. W. Marcellin, "Scanning order strategies for bit plane image coding," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 21, no. 4, pp. 1920–1933, 2012, doi: 10.1109/TIP.2011.2176953.

[52] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, "Bitplane Image Coding With Parallel Coefficient Processing," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 25, no. 1, pp. 209–219, 2016, doi: 10.1109/TIP.2015.2484069.

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

[54] *SMPTE Standard ST 2022-6 - Transport of High Bit Rate Media Signals over IP Networks (HBRMT)*, ST 2022-6:2012, SMPTE, Piscataway, NJ, USA.

[55] *RTP: A Transport Protocol for Real-Time Applications*, RFC 3550, IETF, Aug. 2018. [Online]. Available: https://tools.ietf.org/html/rfc3550

[56] *SMPTE Standard ST 2110-10 Professional Media Over Managed IP Networks: System Timing and Definitions*, ST 2110-10:201, SMPTE, Piscataway, NJ, USA, 2017.

[57] H. H. Massimo Visca, "HDTV production codec tests: EBU Project Group P/HDTP," [Online]. Available: https://tech.ebu.ch/docs/techreview/trev_2008-Q3_HD-Prod-Codecs.pdf

[58] A. Descampe, J. Keinert, T. Richter, S. Fößel, and G. Rouvroy, "JPEG XS, a new standard for visually lossless low-latency lightweight image compression," in *Applications of Digital Image Processing XL: 7-10 August 2017, San Diego, California, United States / Andrew G. Tescher, editor*, San Diego, United States, 2017, p. 21. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10396/2273625/JPEG-XS-a-new-standard-for-visually-lossless-low-latency/10.1117/12.2273625.full

[59] *SMPTE Standard ST 2042-1 - VC-2 Video Compression*, ST 2042-1, SMPTE, Piscataway, NJ, USA, 2012.

[60] T. Borer, "Open Technology Video Compression for Production and Post Production: Research White Paper - WHP159," 2007. [Online]. Available: http://downloads.bbc.co.uk/rd/pubs/whp/whp-pdf-files/WHP159.pdf

[61] BBC, *Dirac Specification: Version 2.2.3.* [Online]. Available: https://web.archive.org/web/20150503015104if_/http://diracvideo.org/download/specification/dirac-spec-latest.pdf (accessed: Mar. 31 2019).

[62] *SMPTE RDD 36 - Apple ProRes Bitstream Syntax and Decoding Process*, RDD 36, SMPTE, Piscataway, NJ, USA.

[63] Apple Inc., *Apple ProRes Whitepaper.* [Online]. Available: https://www.apple.com/final-cut-pro/docs/Apple_ProRes_White_Paper.pdf (accessed: Sep. 2 2018).

[64] *SMPTE Standard ST 2019-1 - VC-3 Picture Compression and Data Stream Format,* SMPTE, Piscataway, NJ, USA, 2014.

[65] *TICO Lightweight Codec Used in IP Networked or in SDI Infrastructures*, RDD 35, SMPTE, Piscataway, NJ, USA.

[66] T. Richter, S. Fößel, J. Keinert, and A. Descampe, "High speed, low-complexity image coding for IP-transport with JPEG XS," in *Applications of Digital Image Processing XXXIX*, San Diego, California, United States, 2016, 99711N.

[67] *ISO/IEC 21122-1 - Information technology — JPEG XS low-latency lightweight image coding system: Part 1: Core coding system,* ISO/IEC, 2019. [Online]. Available: https://www.iso.org/standard/74535.html

[68] *High Throughput JPEG 2000 (HTJ2K): Call for Proposals*, ISO/IEC 15444 (JPEG 2000), ISO/IEC. [Online]. Available: https://jpeg.org/downloads/htj2k/wg1n76037-HTJ2K_final_cfp.pdf

[69] *ISO/IEC 15444-1 - Information technology — JPEG 2000 image coding system: Part 15: High-Throughput JPEG 2000,* ISO/IEC, 2019. [Online]. Available: https://www.iso.org/standard/76621.html

[70] M. W. Marcellin, M. A. Lepley, A. Bilgin, T. J. Flohr, T. T. Chinen, and J. H. Kasner, "An overview of quantization in JPEG 2000," *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 73–84, 2002, doi: 10.1016/S0923-5965(01)00027-3.

[71] V. Bruns, M. A. Martinez-del-Amor, and H. Sparenberg, "Evaluation of GPU/CPU co-processing models for JPEG 2000 packetization," in *2017 IEEE 19th International Workshop on Multimedia Signal Processing (MMSP)*, Luton, Oct. 2017 - Oct. 2017, pp. 1–6.

[72] C.-J. Lian, K.-F. Chen, H.-H. Chen, and L.-G. Chen, "Analysis and architecture design of block-coding engine for EBCOT in JPEG 2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 3, pp. 219–230, 2003, doi: 10.1109/TCSVT.2003.809833.

[73] Digital Cinema Initiatives, *Original Standard Evaluation Material (StEM).* [Online]. Available: https://www.dcimovies.com/Original_StEM_Access/ (accessed: Mar. 15 2021).

[74] R. Yu, C. C. Ko, S. Rahardja, and X. Lin, "Bit-plane Golomb coding for sources with Laplacian distributions," in *2003 IEEE international conference on acoustics, speech and signal processing*, Hong Kong, China, 2003, IV-277-80.

[75] V. Bruns and M. A. Martinez-del-Amor, "Sample-parallel execution of EBCOT in fast mode," in *Proceedings 2016 Picture Coding Symposium (PCS): 4th-7th December 2016, Nürnberg, Germany*, Nuremberg, Germany, 2016, pp. 1–5.

[76] A. Balevic, "Parallel Variable-Length Encoding on GPGPUs," in *Lecture notes in computer science, 0302-9743*, vol. 6043, *Euro-PAR 2009 parallel processing workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, the Netherlands, August 25-28, 2009 workshops*, H.-X. Lin, Ed., Berlin: Springer, 2010, pp. 26–35.

[77] N. Sakharnykh, *GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell.* [Online]. Available: https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/ (accessed: Sep. 13 2018).

[78] D. Taubman, *Kakadu Software: The world's leading JPEG 2000 software development toolkit.* [Online]. Available: http://kakadusoftware.com/ (accessed: Sep. 16 2018).

[79] V. Bruns, M. A. Martinez-del-Amor, and H. Sparenberg, "GPU-friendly EBCOT variant with single-pass scan order and raw bit plane coding," in *2017 IEEE International Conference on Image Processing (ICIP)*, Beijing, Sep. 2017 - Sep. 2017, pp. 3230–3234.

[80] V. Bruns and H. Sparenberg, "Comparison of Code-Pass-Skipping Strategies for Accelerating a JPEG 2000 Decoder,"

[81] F. Aulí-Llinàs, "Stationary probability model for bit plane image coding through local average of wavelet coefficients," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 20, no. 8, pp. 2153–2165, 2011, doi: 10.1109/TIP.2011.2114892.

[82] F. Auli-Llinas and M. W. Marcellin, "Stationary Probability Model for Microscopic Parallelism in JPEG2000," *IEEE Trans. Multimedia*, vol. 16, no. 4, pp. 960–970, 2014, doi: 10.1109/TMM.2014.2307553.

[83] F. Auli-Llinas, I. Blanes, J. Bartrina-Rapesta, and J. Serra-Sagrista, “Stationary model of probabilities for symbols emitted by bit plane image coders,” in *2010 IEEE International Conference on Image Processing*, Hong Kong, Hong Kong, Sep. 2010 - Sep. 2010, pp. 497–500.

[84] F. Auli-Llinas, “Context-Adaptive Binary Arithmetic Coding With Fixed-Length Codewords,” *IEEE Trans. Multimedia*, vol. 17, no. 8, pp. 1385–1390, 2015, doi: 10.1109/TMM.2015.2444797.

[85] H. Li, Di Yu, A. Kumar, and Y.-C. Tu, “Performance Modeling in CUDA Streams - A Means for High-Throughput Data Processing,” *Proceedings : IEEE International Conference on Big Data.*, vol. 2014, pp. 301–310, 2014, doi: 10.1109/BigData.2014.7004245.

[86] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *The 36th Annual International Symposium on Computer Architecture: Austin, Texas, USA, 20-24 June 2009 : Conference proceedings*, Austin, TX, USA, 2009, p. 152.

[87] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan, “A performance prediction model for the CUDA GPGPU platform,” in *2009 International Conference on High performance Computing (HiPAC 2009), Kochi, India, 16-19 December 2009*, Kochi, India, 2009, pp. 463–472.

[88] NVIDIA, *NVIDIA® Nsight™ Application Development Environment for Heterogeneous Platforms, Visual Studio Edition 5.5 User Guide: Instruction Statistics.* [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/nsight/analysis/report/cudaexperiments/kernellevel/instructionstatistics.htm (accessed: Sep. 30 2018).

[89] V. Bruns, T. Richter, B. Ahmed, J. Keinert, and S. Fößel, “Decoding JPEG XS on a GPU,” in *Proceedings 2018 Picture Coding Symposium 2018.*

[90] Fraunhofer IIS, *JPEG XS: the new low complexity codec standard for professional video production.* [Online]. Available: https://www.iis.fraunhofer.de/jpegxs (accessed: Mar. 18 2021).

[91] T. Richter, J. Keinert, A. Descampe, and G. Rouvroy, “Entropy Coding and Entropy Coding Improvements of JPEG XS,” in *DCC 2018: 2018 Data Compression Conference : 27-30 March 2018, Snowbird, Utah, USA : proceedings*, Snowbird, UT, 2018, pp. 87–96.

[92] A. Secker and D. Taubman, “Lifting-based invertible motion adaptive transform (LIMAT) framework for highly scalable video compression,” *IEEE Trans. on Image Process.*, vol. 12, no. 12, pp. 1530–1542, 2003, doi: 10.1109/TIP.2003.819433.

[93] D. Rufenacht and D. Taubman, “Leveraging decoded HEVC motion for fast, high quality optical flow estimation,” in *2017 IEEE 19th International Workshop on Multimedia Signal Processing (MMSP)*, Luton, Oct. 2017 - Oct. 2017, pp. 1–6.

Appendix A – Test Sequences

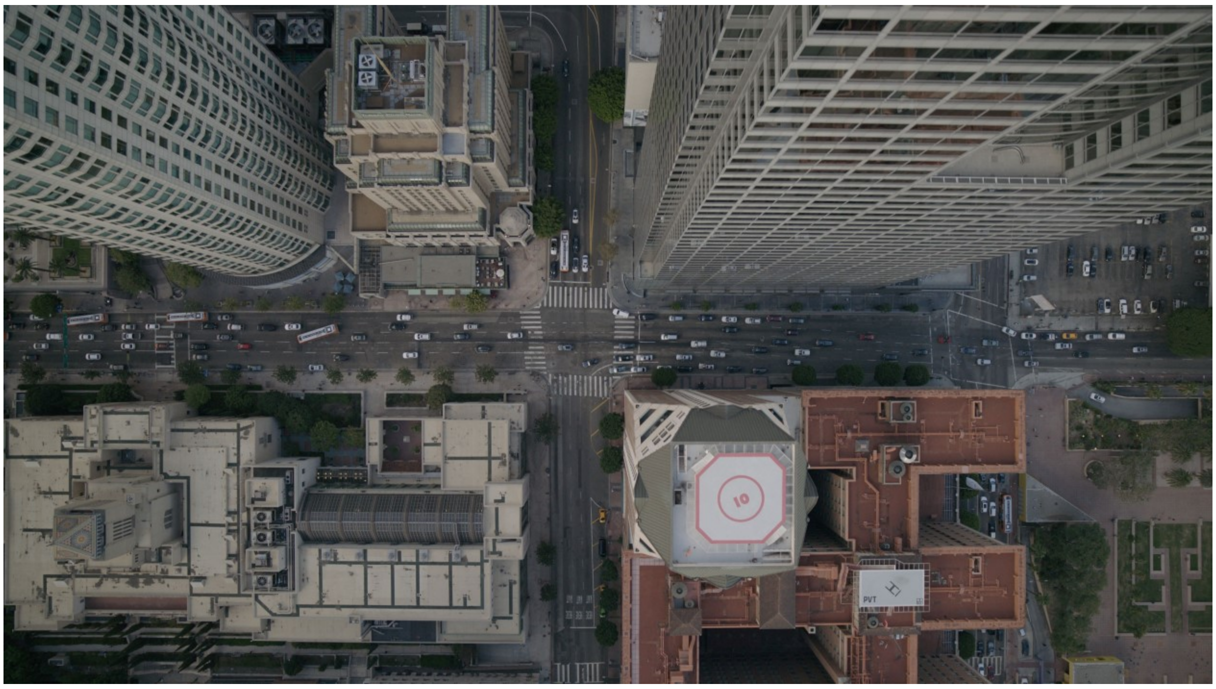
"VQEG Crowd Run", 3840 x 2160, RGB, 48 bit, entropy[6] =7.71

ftp://vqeg.its.bldrdoc.gov/HDTV/SVT_MultiFormat/2160p50_CgrLevels_Master_SVTdec05_/

"ARRI Alexa Drums", 3840 x 2160, RGB, 36 bit, entropy = 7.65

[6] calculated on 24 bit grey image

„ ARRI Alexa Helicopter“, 3840 x 2160, RGB, 36 bit, entropy = 7.10



„StEM Confetti“, 2048 x 857, 48 bit, entropy = 6.48

https://www.smpte.org/store/stem-materials

Big Buck Bunny „Falling Tree Trunk“, 1920 x 1080, 48 bit , entropy = 6.74

https://peach.blender.org/

Big Buck Bunny „Closeup “, 1920 x 1080, 48 bit, entropy = 6.93

https://peach.blender.org/

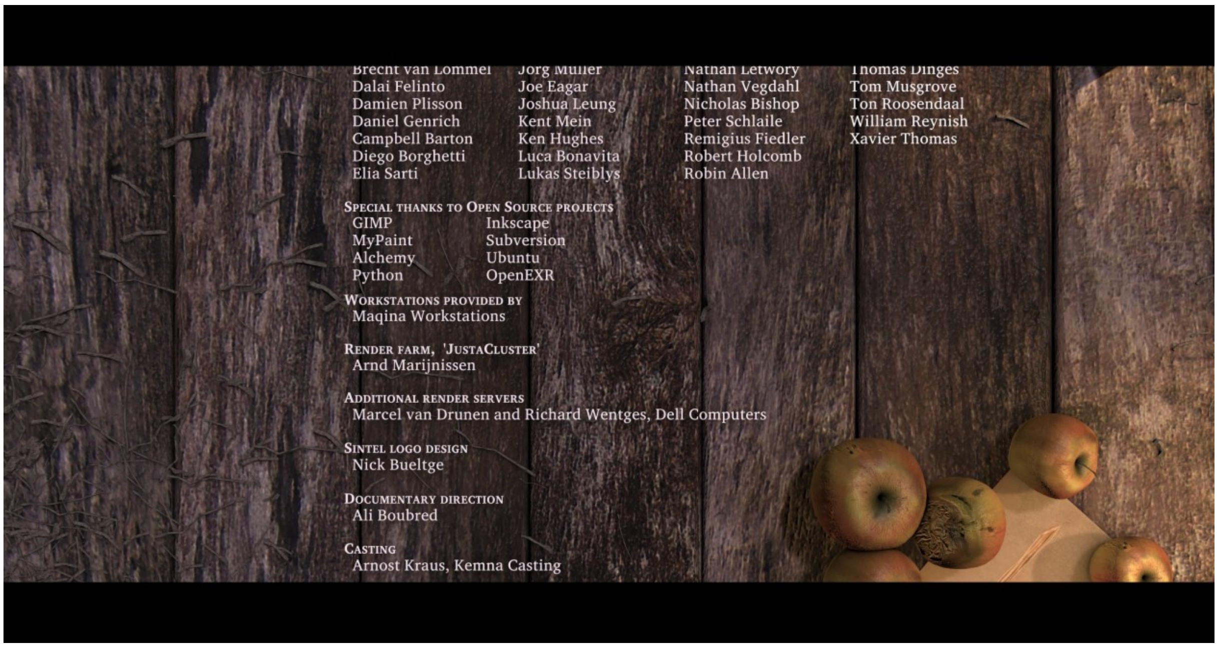
"Sintel title", 4096 x 2160, RGB, 48 bit, entropy = 6.50

https://durian.blender.org/

THE DURIAN OPEN MOVIE PROJECT

"Sintel rolling titles", 4096 x 2160, RGB, 48 bit

https://durian.blender.org/



Brecht van Lommel	Jorg Muller	Nathan Letwory	Thomas Dinges
Dalai Felinto	Joe Eagar	Nathan Vegdahl	Tom Musgrove
Damien Plisson	Joshua Leung	Nicholas Bishop	Ton Roosendaal
Daniel Genrich	Kent Mein	Peter Schlaile	William Reynish
Campbell Barton	Ken Hughes	Remigius Fiedler	Xavier Thomas
Diego Borghetti	Luca Bonavita	Robert Holcomb	
Elia Sarti	Lukas Steiblys	Robin Allen	

SPECIAL THANKS TO OPEN SOURCE PROJECTS

GIMP	Inkscape
MyPaint	Subversion
Alchemy	Ubuntu
Python	OpenEXR

WORKSTATIONS PROVIDED BY
Maqina Workstations

RENDER FARM, 'JUSTACLUSTER'
Arnd Marijnissen

ADDITIONAL RENDER SERVERS
Marcel van Drunen and Richard Wentges, Dell Computers

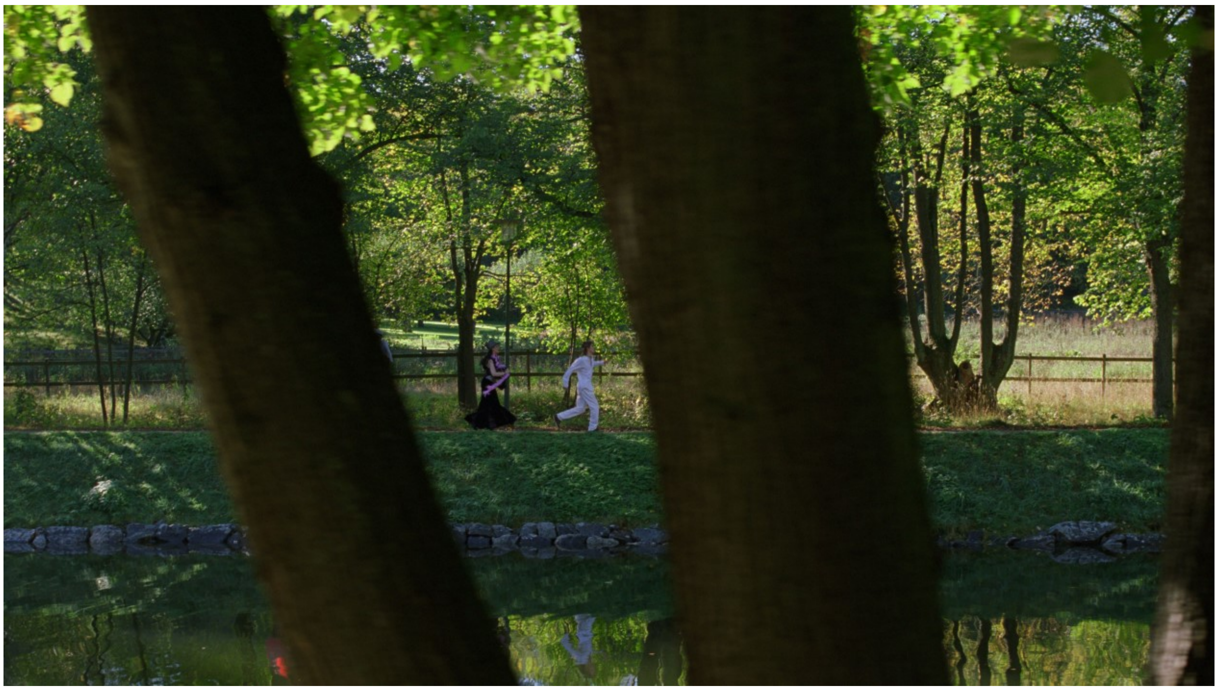
SINTEL LOGO DESIGN
Nick Bueltge

DOCUMENTARY DIRECTION
Ali Boubred

CASTING
Arnost Kraus, Kemna Casting

"VQEG Park Joy", 3840 x 2160, RGB, 48 bit

ftp://vqeg.its.bldrdoc.gov/HDTV/SVT_MultiFormat/2160p50_CgrLevels_Master_SVTdec05_/



"Kodak" test set "001" (top left), "002" (top right), "003" (bottom left), "004" (bottom right) – 3072 x 2048