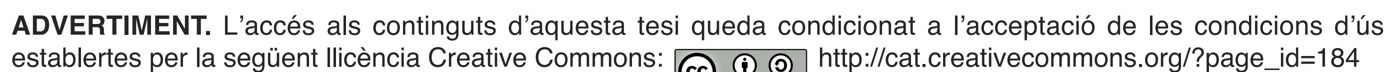
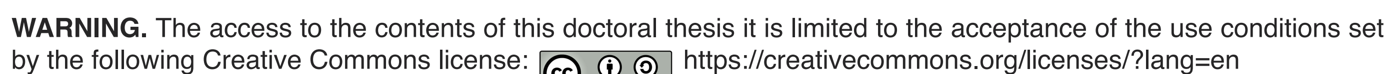
Universitat Autònoma de Barcelona

ADVERTIMENT. L'accés als continguts d'aquesta tesi queda condicionat a l'acceptació de les condicions d'ús establertes per la següent llicència Creative Commons:  http://cat.creativecommons.org/?page_id=184

ADVERTENCIA. El acceso a los contenidos de esta tesis queda condicionado a la aceptación de las condiciones de uso establecidas por la siguiente licencia Creative Commons: http://es.creativecommons.org/blog/licencias/

WARNING. The access to the contents of this doctoral thesis it is limited to the acceptance of the use conditions set by the following Creative Commons license:  https://creativecommons.org/licenses/?lang=en

Universitat Autònoma de Barcelona
Departament d'Enginyeria de la Informació i de les Comunicacions

High Throughput Image/Video Codec with Nvidia GPUs

Submitted to Universitat Autònoma de Barcelona
in partial fulfillment of the requirements for the
degree of Doctor of Philosophy in Computer Science

by Carlos de Cea Domínguez
Bellaterra, Oct 2021

Supervised by:
Dr. Francesc Aulí Llìnas
Dr. Joan Bartrina Rapesta

© Copyright 2021 by Carlos de Cea Domínguez

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bellaterra, Oct 2021

Dr. Francesc Aulí Llìnas and Dr. Joan Bartrina Rapesta
(Supervisors)

Committee:

Dr. Victor Sanchez
Dr. Miguel Hernandez
Dr. Daniel Hernandez
Dr. Javier Melenchón Maldonado (substitute)
Dr. Marc Vivet (substitute)
Dr. Ian Blanes (substitute)

"A mis padres y hermanos.
A los amigos que me han
acompañado en este viaje."

Abstract

The increasing number of image and video content, and the adoption of 8K resolution and high dynamic range technologies, demand faster and more efficient digital coding solutions to store and transfer these data. State-of-the-art solutions like HEVC or JPEG2000 are widely adopted but their computational requirements pose a challenge even for current hardware. For environments like digital cinema or medical image, specific FPGA boards are used to accelerate image processing without affecting image quality. In the last years, a massive parallel hardware have started to gain attraction: Graphical Processing Units (GPUs).

GPUs are massive parallel architectures originally suited for videogames or 3D simulations. In the recent years, their adoption as general purpose devices have allowed to use them as accelerators for a myriad of applications. Algorithms properly adapted to run on GPUs get significant throughput improvements when compared to their CPU implementation. This research focuses on creating an end-to-end codec based on the JPEG2000 standard tailored for GPUs.

This thesis proposes five main contributions, all of which have been published in relevant conferences or journals. The first one focuses on the first end-to-end GPU codec version, which can code and decode gray-scale images. The second version includes the implementation of the video engine within the codec, which can process up to two frames simultaneously. The third contribution consists of an in-depth analysis of the end-to-end codec with multiple throughput improvements and the addition of a multi-frame processing approach, which allows to process multiple frames simultaneously when coding video. The fourth contribution proposes the implementation of an improvement to the core coding engine, tested on a CPU version of the end-to-end codec. The last contribution details an in-depth analysis of the improvement presented in the previous paper but implemented in the end-to-end GPU codec, including results with improvements of more than 10× the performance of the best JPEG2000 commercial implementation when processing 4K RGB video.

Acknowledgements

Una tesis doctoral es un viaje en el que encuentras múltiples obstáculos y diferentes sendas hasta conseguir el ansiado título. Como guías he tenido a mis supervisores, Francesc, Juan Carlos y Joan que me han acompañado y aconsejado a cada paso que he dado. Sin vosotros es evidente que no estaría donde estoy ni habría conseguido todo lo que hemos logrado en tan poco tiempo. Este logro es de todos. Gracias por vuestra inmensa paciencia, vuestro conocimiento y apoyo.

En la facultad, además de a mis supervisores, tuve la fortuna de conocer a gente muy capaz que me acompañó en el viaje. Sin ánimo de ser excluyente, debo dar las gracias a mis dos compañeras de despacho por haber hecho más ameno este trayecto, a Pablo por haber contribuido a introducirme en mi línea de investigación, a mi compañero de facultad, Iván, por todos esos almuerzos donde intercambiábamos opiniones y a todos los profesores asociados con los que compartí docencia estos años.

Saliendo ya del ámbito académico, quiero remarcar la importancia que ha tenido mi familia. Gracias a mis padres y hermanos, que aun en la distancia, siempre se han preocupado e interesado por cómo iba avanzando en este viaje. Os he sentido igual de cerca sin importar los kilómetros que nos han separado. Habéis sido el refugio al que acudir siempre que lo he necesitado.

Acompañándome a cada paso que daba en mi estancia en Barcelona estabais vosotros, Aitor y Daniel. Habéis sido los mejores amigos que podía pedir, consiguiendo que mi vida en Cataluña fuera mucho más familiar, agradable y divertida. Sin nuestras tardes de viernes y nuestras escapadas, los días habrían pesado muchísimo. Puedo decir sin temor a equivocarme que, de no ser por los dos, no creo que hubiera sido capaz de llevar tan bien mi estancia allí. Gracias por conseguir que tres años se sintieran como tres meses.

Tampoco me olvido de todos aquellos amigos que tengo lejos tanto en León como en muchas otras partes de España. Cierto es que durante la tesis los momentos de reunión han escaseado, y aun así habéis permanecido atentos y dispuestos a ayudar en lo que buenamente pudierais. Especialmente gracias a vosotros, Sandra y Diego, por vuestro apoyo incondicional este año. Sois muchos, nombraros a todos me es imposible. ¡Vosotros sabéis quienes sois!

Por último, y no menos importante, daros las gracias a vosotros, Javier, Agustín, Roger y Miguel, que me acompañasteis virtualmente durante todos estos años, siempre poniendo esa guinda de diversión y esparcimiento tan necesaria, sobre todo durante los últimos meses que estuve en Barcelona mientras duró el confinamiento sanitario estricto.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Thesis contributions 4

1.2 Organization 6

2 High Throughput Image Codec for High-Resolution Satellite Images **9**

3 GPU architecture for wavelet-based video coding acceleration **15**

4 GPU Oriented Architecture for an End-to-End Image Video Codec Based on JPEG2000 **27**

5 Complexity Scalable Bitplane Image Coding with Parallel Coefficient Processing **43**

6 Real-time 16K Video Coding on a GPU with Complexity Scalable BPC-PaCo **51**

7 Conclusions **67**

7.1 Summary 67

7.2 Future research lines 68

A List of Publications **71**

B Acronyms **73**

Chapter 1

Introduction

Digital image and video coding is used nowadays in a myriad of fields and disciplines. The amount of images and videos grow day after day and its storage and processing is getting more challenging as quality, resolutions and samples bit-depths increase. In order to process all these data and store it appropriately, usage of image and video coding techniques are a must. There are three main expert groups that define image and video compression standards that conform the state-of-the-art in terms of digital image encoding/decoding: the consultative committee for space data systems (CCSDS), the moving pictures expert group (MPEG) and the joint photographic expert group (JPEG). The CCSDS [1] coding standards, including CCSDS121 [2], CCSDS122 [3] and CCSDS123 [4] are mainly employed on-board satellites to compress/decompress data obtained on embedded boards, hence its lower computational complexity and feature set. For digital processing in environments with less computational restrictions, HEVC [5] and JPEG2000 [6] are the ones employed for most professional and user-based tasks. Both standards include advanced features like quality scalability, interactive transmission or error resilience. However, both of them, specially JPEG2000, are very demanding computationally, and scenarios like digital cinema or medical usage require specific hardware like Floating Point Gate Arrays (FPGAs) [7, 8, 9, 10] to process data in real-time at high quality. With the increasing demand on higher resolutions and the inclusion of High Dynamic Range (HDR) technologies, the importance of getting faster compression tools to store and transfer

these data is growing more than ever.

The aforementioned standards, HEVC and JPEG2000, were designed based on the mainstream hardware to process data available to the date: Central Processing Units (CPUs). CPUs follow the multiple-instructions multiple-data paradigm (MIMD), which in short means that they are designed to have few but very powerful processing threads. Images, and videos, are data structures consisting of, commonly, thousands of data points which may undergo a series of transformations while they are processed. In the last decade, Graphical Processing Units (GPUs) have become more popular to accelerate applications, specially due to their computational power, with 10× more TFlops when compared to CPUs, and their affordable price. Furthermore, many supercomputers are including Nvidia GPUs, hence becoming more relevant in scientific researches and simulations [11].

GPUs are hardware devices which were initially designed to aid with graphical applications or videogames. Currently, they are used as well as general purpose devices to accelerate certain parts of a workload. GPUs are based on the single-instruction multiple-data (SIMD) paradigm, which, contrary to the MIMD one, means that the device is designed to process, typically, thousands of data points simultaneously using thousands of threads in a synchronous fashion. GPUs are composed of, typically, multiple streaming multiprocessors (SMs), each of one is responsible of running multiple 32-wide vector instructions in parallel. These groups of 32-threads are referred to as warps, and warps are grouped in thread blocks, which are independent execution units running on an SM. Thread blocks within an SM can be running kernels (i.e. functions) from the same or different applications. Nvidia GPUs make use of the so-called streams to manage the execution of these independent kernels. Using multiple streams can be beneficial to balance the resource usage and throughput of the GPU.

Comparing each architecture, CPU threads pose a lower latency and higher computational power than those of a GPU. However, having more threads to process data simultaneously can be beneficial, specially for data loads that require simple operations per data point. Algorithms which expose a huge amount of data points with no dependencies among them are perfect scenarios for these kind of devices.

Nonetheless, inherently sequential algorithms conform a difficult task for GPUs. In this situation, either the algorithm is slightly modified to run on GPUs without removing the causality among data points, or the algorithm is re-engineered to decrease (or remove) the dependencies. The drawback on the first approach is that throughput increases are usually not that impressive, but compliance with the original implementation is maintained. The second option usually achieves high boosts in throughput, although as a consequence the new application may lose compliance with the original implementation. Applications which can take advantage of this type of parallelism can get huge throughput improvements if the algorithms are properly adapted. Fields like Bioinformatics can get boosts of up to 20× more performance when adapting its algorithms to run over GPUs [12].

Of the aforementioned codecs, JPEG2000 is the one used in very high-quality demanding environments due to its coding methodology and its excellent resilience to image artifacts. In the last decade, there have been multiple attempts to port JPEG2000 to GPUs without losing its compliance in an effort to increase throughput. The core coding aspect within this standard is composed of four main parts: a color transform (CT), a discrete wavelet transform (DWT), a bitplane coding engine (BPC) and a codestream reorganizer (CR). Of those four, the third one is the heaviest in terms of computational complexity, and it is the only one which exhibits strong dependencies among samples. Thus, unless causality between data samples is broken, the throughput improvements are minimal when porting the algorithm to a SIMD-based architecture. The different attempts to port the algorithm to GPUs, either implementing a full end-to-end codec or just the BPC stage [13, 14, 15, 16, 17, 18, 19, 20, 21], did not achieve competitive throughput, getting lower results than current JPEG2000 CPU implementations like Kakadu [22] running on current hardware. There are other approaches that follow a different line, using GPUs implementations like Comprimato [23] to aid in the coding process, although their performance is limited by the sample inter-dependencies. Finally, the last approach consists on redesigning the algorithm, which achieves better results but the compliance with the traditional implementation is lost [24, 25].

This thesis researches on how to create an efficiently end-to-end image and video

Title	Year	Description	Conf. / Journal	Core / Quartile	Imp. Factor
High throughput image codec for high-resolution satellite images	2018	First end-to-end version which codes and decodes images	IGARSS2018	C	N/A
GPU architecture for wavelet-based video coding acceleration	2020	First end-to-end version with support to code/decode video with up to two streams	ParCo2019	N/A	N/A
GPU-oriented architecture for an end-to-end image/video codec based on JPEG2000	2020	Multi-stream image/video coding/decoding software with in-depth analysis on the GPU resources usage, different amount of streams and a final comparison with other state-of-the-art solutions	IEEE Access	Q1	3.367
Complexity scalable bitplane image coding with parallel coefficient processing	2020	New feature to improve throughput of the BPC-PaCo kernel on the CPU version of the codec with slight coding performance penalizations	IEEE Signal Processing Letters	Q1	3.109
Real-time 16K Video Coding on a GPU with Complexity Scalable BPC-PaCo	2021	Complexity scalable implementation on the GPU codec with in-depth analysis on the throughput improvements and coding performance	Signal Processing: Image Communication	Q2	3.256

Table 1.1: List of publications presented during the Ph.D. and included in the proceedings afterwards.

codec to maximize the throughput while keeping coding performance on par with the original implementation. The proposed solution is based on JPEG2000, it runs entirely on Nvidia GPUs without sacrificing any existing feature, and with increases of over 10× the throughput over the best commercial JPEG2000 implementation.

1.1 Thesis contributions

This thesis continues the line of research started about ten years ago that aims to develop new coding techniques for image/video compression tailored for the fine-grain parallelism of GPUs. It started with the development of coding techniques meant to break the causality of classical coding strategies [26, 27, 28]. Those results lead to the development of a lightweight arithmetic coder that allowed fine-grain parallelism [29, 30]. After that, the research focused on designing and building the different stages of a wavelet-based coding pipeline. The first stage was to develop the DWT [31, 32], which achieved highly competitive performance thanks to a novel register-based strategy. Secondly, an algorithm for the BPC with fine-grain parallelism was presented and evaluated in terms of coding performance in [33, 34]. Then, the GPU implementation of [33, 34] was introduced in [35]. The coding technique used in [33, 34, 35] is referred to as BPC-PaCo. These contributions define the isolated parts of an end-to-end GPU-based codec. This thesis starts when those contributions are published.

The first end-to-end version included the two GPU kernels, DWT and BPC-PaCo, with further adaptations to communicate them, and a new GPU kernel called code-stream reorganization, which reorganizes the data from BPC-PaCo into a compressed data structure ready to be written to the disk. Apart from the new kernel, the challenge of this version was designing an infrastructure capable of taking care of memory transfers between kernels efficiently. Managing memory allocations can be expensive, so it is preallocated taking in consideration the amount of data needed per kernel, which depends on the input data size. The codec makes use as well of Nvidia libraries to implement some primitives within the CR kernel. It also included the first I/O algorithms to read and write images with the minimal impact on the overall performance. This version of the codec can code/decode gray-scale 8-bits images. It was presented at IGARSS2018 [36] and published in the proceedings afterwards.

After the first end-to-end version, the codec was improved by adding video processing, while also including color processing. For this version, the GPU can process two frames simultaneously from a single video feed. It can handle multiple CPU threads to manage two GPU streams simultaneously. Memory preallocation and assignation is carefully handled per stream. The I/O processing is improved, with capability to read information from the disk and send the data to the GPU asynchronously while other frames are being processed. This version yields 4K real-time throughput performance in a Nvidia GTX 1080 Ti GPU. It was presented at ParCo2019 [37] and published in the proceedings afterwards.

The third implementation includes multi-stream capabilities and multiple throughput improvements across the coding pipeline, including all the kernels and the I/O functions. It details each kernel from a GPU perspective, including memory transfers from the device memory to the individual registers, bandwidth values, device occupancy, etc. It also compares the proposed codec against state-of-the-art solutions like JPEG2000 Kakadu [22] or Nvidia HEVC NVENC [38]. It gets better throughput results than existing JPEG2000 compliant GPU implementations, which run slower than Kakadu. It is published in the IEEE Access on 2020 [24].

For the fourth publication, the BPC-PaCo algorithm receives a new feature called Complexity Scalability, getting renamed to CS BPC-PaCo. Traditional BPC-PaCo

takes about 86% of the execution time of the entire coding pipeline. CS BPC-PaCo is aimed to improve coding throughput by using a K factor that works as a trade-off between coding performance and coding throughput. This version is implemented in Java, running on an Intel CPU to test its viability and efficiency. The results showed double digit throughput increments with a coding performance penalty of about 10% with the highest factor. These results were sent to the journal IEEE Signal and Processing Letters, getting published on 2020 [39].

The final publication details the implementation of CS BPC-PaCo in the GPU end-to-end codec. It includes an exhaustive and detailed analysis from both kernel and codec perspective, thoroughly analyzing the impact in memory transfers, instructions per cycle, registers and bandwidth with different K factors, effectively explaining the achieved improvements. Results show that the codec achieves 16K coding throughput in real-time at 50 dB of PSNR when coding a 2 minute long 4K RGB 8 bps in the Nvidia RTX 2080 Ti, faster than the new JPEG2000 standard part "High-Throughput JPEG2000" codec running on a i9-9900K Intel CPU. It was sent to the journal Signal Processing: Image Communication and published on 2021 [40].

1.2 Organization

Each of the following chapters correspond to each of the published papers unedited and in order of publication:

Chapter 2: High Throughput Image Codec For High-Resolution Satellite Images - Includes the research published in [36].

Chapter 3: GPU architecture for wavelet-based video coding acceleration - Includes the research published in [37].

Chapter 4: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000 - Includes the research published in [24].

Chapter 5: Complexity Scalable Bitplane Image Coding With Parallel Coefficient Processing - Implementation of the new CS BPC-PaCo published in [39].

Chapter 6: Real-time 16K video coding on a GPU with Complexity Scalable BPC-PaCo - Includes the GPU implementation of the new CS BPC-PaCo published

in [40].

Chapter 7: Conclusions. It includes the final thoughts of the thesis and future lines of research.

Chapter 2

High Throughput Image Codec for High-Resolution Satellite Images

HIGH THROUGHPUT IMAGE CODEC FOR HIGH-RESOLUTION SATELLITE IMAGES

Carlos de Cea-Dominguez[†], *P. Enfedaque*[⋆], *Juan C. Moure*[‡], *Joan Bartrina-Rapesta*[†], *Francesc Auli-Llinas*[†]

[†] Department of Information and Communications Engineering
[‡] Department of Computer Architecture and Operating Systems
[†] [‡] Universitat Autònoma de Barcelona, Spain
[⋆] Computational Research Division, Lawrence Berkeley National Laboratory

ABSTRACT

The growth in the use of satellite images has generated the need for their fast compression, processing, and distribution. JPEG2000 is a widespread standard for the compression and transmission of such images once they are in the ground. Despite its advanced features and excellent coding performance, JPEG2000 demands significant computational resources. This paper introduces a wavelet-based codec that uses the JPEG2000 framework, but replaces its most computationally demanding coding stage by a highly parallel engine. When executed in Graphics Processing Units to code high-resolution satellite images, the proposed codec achieves speed-ups of up to 8× when compared to the fastest implementation of JPEG2000 executed in a multi-core platform.

***Index Terms*—** High throughput image coding, bitplane image coding, JPEG2000.

1. INTRODUCTION

Satellite images are nowadays employed in myriad fields such as management of natural resources, study of climate change, weather forecast, or map making, among others. These images are captured by missions formed by a (constellation of) Earth Observation satellite(s) like the LandSat, GeoEye, Sentinel, or SEOSat. The huge amount of data acquired by these satellites pose several challenges. The first is to download the images to the ground with minimum loss in quality. To do so, compression standards such as those proposed by the Consultative Committee for Space Data Systems are commonly employed. Once the data are on the ground, other compression standards with more advanced features are used to store and transmit these images. JPEG2000 (ISO/IEC 15444) is a widespread choice to do so.

Despite its excellent capabilities, the bitplane coding engine of JPEG2000, called tier-1 coding [1], is highly demanding in terms of computational resources. Its algorithm scans the wavelet-transformed coefficients of the image numerous times, producing a stream of bits with interesting features such as quality progression, possibility of partial transmission, or error resilience [1]. This comes at the expense of long execution times and complex software/hardware implementations.

The image coding community have studied ways to increase the throughput of JPEG2000 implementations for almost ten years. A promising alternative is to use the highly parallel architectures of Graphics Processing Units (GPUs). Modern GPUs are mostly based on the Single Instruction Multiple Data (SIMD) paradigm. Its main idea is to execute a flow of instructions on multiple pieces of data in a lock-step synchronous way. To effectively apply such vector instructions on an algorithm there must be as few dependencies among the data as possible. The main difficulty to implement JPEG2000 in GPUs is that the tier-1 coding stage employs an inherently sequential algorithm. It scans the coefficients one-by-one, producing a result for each that can not be obtained without all the previous. This causality renders parallelism at the bitplane coding engine a very challenging task. Even though there are some works in the literature with this purpose [2–4], some centered on satellite imagery only [5], none of them is able to exploit the full potential of GPUs.

Aware of this issue, the Joint Photographics Experts Group (JPEG) launched in June 2017 a call for proposals [6] aimed to develop an alternate coding engine to increase the throughput of the standard while sacrificing as few as possible of its features. This initiative is currently under development. In the same line of this call for proposals but well before it was launched, we started a line of research whose purpose is to devise a bitplane coding engine that can exploit the fine-grain parallelism of GPUs while maintaining the same features as those of JPEG2000. First, we studied stationary probability models [7,8] for the coding of coefficients that did not require previously coded data and/or adaptive algorithms like those employed in the standard. Then, we proposed an arithmetic coder that produces fixed-length codewords [9], allowing their interleaving in a single stream instead of producing a single codeword like the MQ coder of JPEG2000. These two techniques were combined in [10] to devise a bitplane cod-

This work has been partially supported by the Spanish Government (MINECO), by FEDER, and by the Catalan Government, under Grants TIN2015-71126-R, TIN2014-53234-C2-1-R, and 2014SGR-691.

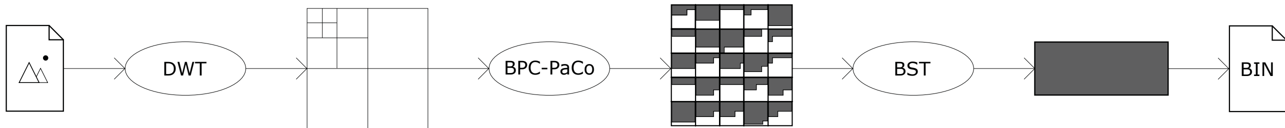


Fig. 1: Pipeline of the proposed codec.

ing strategy with parallel coefficient processing (BPC-PaCo), the first coding engine that breaks the dependencies among coefficients. Its implementation in a GPU [11] indicates that it can effectively exploit the parallelism of SIMD architectures, which results in high speed-up factors with respect to the fastest implementations of JPEG2000, either executed in multi-core Central Processing Units (CPUs) or in GPUs.

Our previous work is solely centered on the bitplane coding stage. As explained below, wavelet-based image codecs (including JPEG2000) are commonly structured in three coding stages: wavelet transform, bitplane coding, and bitstream re-organization. This paper introduces an end-to-end codec based on BPC-PaCo. It employs the framework of JPEG2000 and provides the same features of the standard. When employed to code high-resolution satellite images in consumer-grade GPUs from Nvidia, it achieves speed-ups of up 8× as compared to the fastest implementations of JPEG2000.

The rest of the paper is structured as follows. Section 2 explains the basics of SIMD architectures and JPEG2000. Section 3 describes the proposed end-to-end codec in detail. Section 4 provides experimental results achieved when coding high-resolution images captured by the GeoEye satellite. The last section summarizes this work.

2. BACKGROUND

Nvidia GPUs are hardware devices with tens of individual compute units called streaming multiprocessors (SM). Each SM can execute multiple 32-wide SIMD (or also called vector) instructions simultaneously. GPUs from Nvidia employ the CUDA programming model, which defines a computation structure composed by (potentially) thousands of threads grouped into warps and thread blocks. While the hardware device executes 32-wide SIMD instructions, a software CUDA thread is the virtualization of one of the lanes of the SIMD instruction. A warp is the group of 32 consecutive CUDA threads executing vector instructions and advancing their execution in a lock-step synchronous fashion. A CUDA program should reduce control flow divergence (i.e., conditional instructions) among warps since that results in the sequential execution of the divergent paths, which increases the amount of instructions executed. A thread block is a group of warps, each one assigned to run until completion in a specific SM. Warps in a thread block are executed asynchronously and can cooperate via on-chip fast memories, using explicit synchronizing barrier instructions when required.

The CUDA memory model is hierarchically organized as follows: there is a space of local memory private to each thread, a shared memory private to each thread block, and a global memory public to all threads in the device. From a microarchitecture point of view, the local memory reserved per thread is located either in the registers or the off-chip memory, depending on the available resources. GPUs have two levels of cache as well.

As previously stated, the proposed image codec is structured in three coding stages. The discrete wavelet transform (DWT) is employed in the first stage to decorrelate the spatial redundancy of the image. After that, the image is conceptually partitioned in small sets of wavelet coefficients called codeblocks. Bitplane coding is applied in each codeblock independently. The main idea behind this technique is to code the wavelet-transformed coefficients bitplane by bitplane. A bitplane is defined as the collection of bits in the same position of the binary representation of the magnitude of all coefficients. The bits emitted by the coding engine are fed to an arithmetic coder that employs a probability model to compress them. The last stage of the codec truncates and reorders the bitstreams generated for each codeblock to produce the final codestream.

3. PROPOSED CODEC

The proposed codec is implemented in CUDA. Its coding pipeline is depicted in Fig. 1. The encoder (decoder) reads the image (compressed file) from the hard disk to the CPU memory and then moves the data to the main memory of the GPU. The three stages communicate using this main memory. The final stage moves the compressed data (decompressed image) from the GPU to the CPU memory and writes it back to disk. Each coding stage is implemented to exploit the high amount of fine-grain parallelism required for the effective utilization of the GPU's resources. The algorithm employed to apply the DWT already exhibits fine-grain parallelism and its implementation in GPUs has been widely studied in the literature [12–14]. Herein, we employ the work that we proposed in [13] since, to the best of our knowledge, it is the fastest implementation of the DWT for CUDA. BPC-PaCo has been adapted from our previous work [11]. The third stage of the coding pipeline is called bitstream tightening (BST) and is introduced in this paper.

3.1. Discrete wavelet transform

The GPU-adapted DWT implementation employed herein uses a register-based approach [13]. This allows data reuse

and data sharing, taking advantage of the fine-grain parallelism and data access locality of the algorithm. The DWT is applied via the lifting scheme, which processes, alternatively in the vertical or horizontal axis, odd and even image samples by using its adjacent values. The image is divided into tiles, which are processed by independent warps to provide the coarse-grained parallelism needed to populate the SMs of the GPU. With an appropriate tile size, the coefficients of each tile can be completely stored in the local registers of each thread and processed sequentially in both axes without saving the intermediate information. This halves the memory access operations, significantly increasing the throughput of the algorithm.

3.2. Bitplane coding with parallel coefficient processing

The wavelet coefficients are conceptually partitioned in codeblocks, typically containing 32×32 or 64×64 coefficients, that do not have dependencies among them. The processing of codeblocks in parallel typically requires coarse-grained, control-divergent strategies that are employed in many implementations of the original JPEG2000 standard. To solely use this kind of parallelism does not suit well the architecture of GPUs. As previously stated, BPC-PaCo is devised to promote fine-grain parallelism within the codeblock. Briefly described, BPC-PaCo further partitions the codeblock in 32 stripes of coefficients that can be coded by the threads in a warp. The scanning order, context formation, probability model, coding passes, and arithmetic coder are redevised to permit such a parallel processing. In particular, each stripe employs an arithmetic coder and the codewords produced in each stripe are collaboratively interleaved in the bitstream by all the threads in the warp. This permits the use of coarse- and fine-grain parallelism, since both the codeblocks and the coefficients within them are coded in parallel.

3.3. Bitstream tightening

The bitstreams produced by BPC-PaCo for each codeblock have different sizes depending on the data coded. As a consequence, they are scattered throughout the entire output buffer. The BST stage must tighten the information generated by BPC-PaCo reorganizing them into a compact codestream that can then be written to a file. Also, it adds some auxiliary information (i.e., headers) to be able to identify the codeblock data within the final file in addition to general image information.

The first step of the proposed BST algorithm is the generation of a vector of integers that contains the size of the bitstreams generated for consecutive codeblocks. This vector is then employed to generate a new vector that contains aggregate bitstream sizes, which represent the offset of the bitstream with respect to the start position. The vector of offsets is used as a memory map in the data reorganizing process. The vector of offsets is generated via the parallel prefix sum algorithm [15] using the state-of-the-art implementation in the open-source CUB framework.

To perform efficient memory accesses, the workload is distributed among threads promoting that adjacent threads access adjacent data. To do so, the threads are assigned to consecutive positions in the output codestream. Each thread performs a binary search in the vector of offsets to compute the position of the input value corresponding to the assigned output position. A lookup table is added to accelerate the binary search operation: a direct access to the LUT provides a bounded search interval that reduces the average number of search iterations. This auxiliary LUT accelerates the performance of the BST algorithm by 2×.

The decoder reverses the operations that are carried out in the encoder. With regard to the BST stage, the vector containing the sizes of the bitstreams is in the auxiliary information of the compressed file, so only the vector offset needs to be recomputed.

4. EXPERIMENTAL RESULTS

The proposed codec is compared with Kakadu v7.8. Kakadu is the fastest implementation of JPEG2000. It is programmed in C++ and is heavily optimized via assembly instructions. It supports multithread parallelism when executed in multi-core CPUs. Kakadu is executed in a platform that has a total of 32 Intel Xeon cores running at 2.2 GHz. Our implementation is executed in the consumer-grade GPUs reported in Table 1. The GTX 1080 Ti is a high-end desktop GPU whereas the GTX 960M is a low-end GPU for laptops. The tests report the execution time employed for each implementation without considering the time needed to write the final file to the disk, since that significantly varies depending on the hard disk. The tests employ 3 images captured by the GeoEye satellite. The images are 10240×10240, gray-scale, and have a bit-depth of 8 bits per sample.

The results achieved are depicted in Fig. 2. The vertical axis is the performance achieved, reported in Msamples per second. Each pair of columns is the encoding and decoding of an image. The results indicate that the proposed codec is approximately 8× faster than Kakadu when executed in the GTX 1080 Ti and approximately 2× faster than Kakadu when executed in the GTX 960M. As seen in Table 1, the GTX 1080 Ti has six times more cores and memory bandwidth than the GTX 960M, though in the results of Fig. 2 the GTX 1080 is only 4× faster than the GTX 960M. This is due to the processes involved in the codec, though further analysis is required.

From a power efficiency point of view, the consumption used by Kakadu with the Intel Xeon platform is up to 380W, whereas the GTX 1080 Ti and GTX 960M consume up to 250W and 60W, respectively. This indicates that the proposed codec achieves a much higher throughput than Kakadu while consuming significantly less power, making it suitable for mobile solutions.

device	#SMs	cores × SM	#cores	clock freq.	mem. bandwidth
GTX 1080 Ti	28	128	3584	1582 MHz	484 GB/s
GTX 960M	5	128	640	1176 MHz	80 GB/s

Table 1: Features of the GPUs employed.

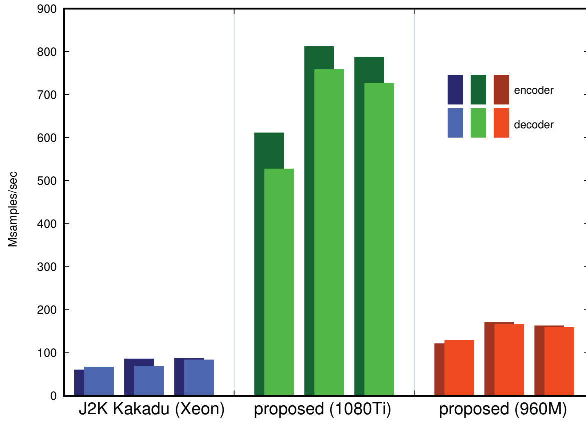


Fig. 2: Results achieved when coding three high-resolution images captured by the GeoEye satellite.

5. CONCLUSIONS

The current image compression standards employed for the compression of satellite imagery are mainly devised to exploit the large-degree of parallelism provided in CPUs. This paper introduces an end-to-end codec that employs the framework of JPEG2000 but that provides –in all stages of the coding pipeline– a fine-degree of parallelism. This can be effectively exploited when executed in architectures that highly rely on SIMD parallelism such as those found in Nvidia GPUs. Experimental results coding large-resolution satellite images indicates that the proposed codec is 8× faster than the most efficient implementations of JPEG2000 while significantly reducing the power consumption. None of the advanced features of JPEG2000 are sacrificed in the proposed codec, so it is ideal for scenarios that deal with massive datasets, such as in the compression an transmission of satellite imagery. Future work will explore the adaptation of the proposed codec for the coding of other types of images and video.

6. REFERENCES

[1] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image compression fundamentals, standards and practice*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.

[2] J. Matela, V. Rusnak, and P. Holub, "Efficient JPEG2000 EBCOT context modeling for massively parallel architectures," in *Proc. IEEE Data Compression Conference*, Mar. 2011, pp. 423–432.

[3] M. Ciznicki, M. Kierzynka, P. Kopta, K. Kurowski, and P. Gepnerb, "Benchmarking JPEG 2000 implementations on modern CPU and GPU architectures," *ELSEVIER Journal of Computational Science*, vol. 5, no. 2, pp. 90–98, Mar. 2014.

[4] Comprimato. (2014, Apr.) Comprimato JPEG2000@GPU. [Online]. Available: http://www.comprimato.com

[5] M. Ciznicki, K. Kurowski, and A. Plaza, "Graphics processing unit implementation of JPEG2000 for hyperspectral image compression," *SPIE Journal of Applied Remote Sensing*, vol. 6, pp. 1–14, Jan. 2012.

[6] *High Throughput JPEG 2000 (HTJ2K): Call for Proposals*, ISO/IEC Std., 2017, document ISO/IEC JTC 1/SC29/WG1 N76037.

[7] F. Auli-Llinas, "Stationary probability model for bitplane image coding through local average of wavelet coefficients," *IEEE Trans. Image Process.*, vol. 20, no. 8, pp. 2153–2165, Aug. 2011.

[8] 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, Jun. 2014.

[9] F. Auli-Llinas, "Context-adaptive binary arithmetic coding with fixed-length codewords," *IEEE Trans. Multimedia*, vol. 17, no. 8, pp. 1385–1390, Aug. 2015.

[10] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, "Bitplane image coding with parallel coefficient processing," *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 209–219, Jan. 2016.

[11] 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, Aug. 2017.

[12] W. J. van der Laan, A. C. Jalba, and J. B. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 132–146, Jan. 2011.

[13] 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, Dec. 2015.

[14] T. M. Quan and W.-K. Jeong, "A fast discrete wavelet transform using hybrid parallelism on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3088–3100, Nov. 2017.

[15] D. Merrill and M. Garland. (2016) Single-pass parallel prefix scan with decoupled look-back. [Online]. Available: http://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf

Chapter 3

GPU architecture for wavelet-based video coding acceleration

GPU architecture for wavelet-based video coding acceleration

Carlos DE CEA-DOMINGUEZ [a,1], Juan C. MOURE [b], Joan BARTRINA-RAPESTA [a] and Francesc AULÍ-LLINÀS [a]

[a] *Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain*

[b] *Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain*

Abstract. The real time coding of high resolution JPEG2000 video requires specialized hardware architectures like Field-Programmable Gate Arrays (FPGAs). Commonly, implementations of JPEG2000 in other architectures such as Graphics Processing Units (GPUs) do not attain sufficient throughput because the algorithms employed in the standard are inherently sequential, which prevents the use of fine-grain parallelism needed to achieve the full GPU performance. This paper presents an architecture for an end-to-end wavelet-based video codec that uses the framework of JPEG2000 but introduces distinct modifications that enable the use of fine-grain parallelism for its acceleration in GPUs. The proposed codec partly employs our previous research on the parallelization of two stages of the JPEG2000 coding process. The proposed solution achieves real-time processing of 4K video in commodity GPUs, with much better power-efficiency ratios compared to server-grade Central Processing Unit (CPU) systems running the standard JPEG2000 codec.

Keywords. Wavelet-based video coding, high-throughput video coding, JPEG2000, GPU, CUDA.

1. Introduction

High definition video with resolutions ranging from 2K to 4K is nowadays common in devices such as TVs, digital cinema projectors, and mobiles. Among others, the JPEG2000 standard (ISO/IEC 15444) is employed for such video content in fields like TV production or digital cinema. This standard provides excellent coding performance and advanced features, such as quality progression, partial transmission, or error resilience [1]. Nonetheless, the algorithms employed to achieve them are very demanding computationally. They transform, code, and reorganize the data in three main stages that require multiple scans and coding operations. This results in long execution times and complex implementations. In the case of digital cinema, for instance, field-programmable gate arrays are required to achieve real-time decoding of 2K and 4K video.

[1]This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under Grants TIN2015-71126-R, TIN2017-84553-C2-1-R and RTI2018-095287-B-I00 (MINECO/FEDER, UE), and by the Catalan Government under Grants 2017SGR-463 and 2017SGR-313.

The first stage of most wavelet-based image/video codecs (including JPEG2000) reduces the image redundancy through the discrete wavelet transform (DWT). The operations performed by the DWT can be parallelized, so DWT implementations for parallel architectures have been widely studied in the literature [2–4]. The second stage employs bitplane coding together with arithmetic coding to reduce the statistical redundancy of wavelet coefficients. This stage scans the coefficients one-by-one, producing a result for each that can not be obtained without all the previous. This causality renders parallelism at the bitplane coding engine a very challenging task. Even though there are some works in the literature with this purpose [5–12], none of them is able to exploit the full potential of GPUs. The third stage of the pipeline reorganizes the data and forms the compressed file.

Aware of the high complexity of JPEG2000, the Joint Photographics Experts Group launched in June 2017 a call for proposals [13] aimed to develop an alternate algorithm for the bitplane and arithmetic coding stage that increases the throughput of the codec. This JPEG2000 part is described in [14, 15]. It increases performance about $10\times$ though penalizes coding performance about 10%. Also, it sacrifices quality scalability, which may become an issue in image/video transmission scenarios.

In the same line of work but well before this call for proposals, we started a line of research whose final goal is to devise an end-to-end image/video codec that can exploit the fine-grain parallelism of GPUs while maintaining the same features of JPEG2000. For the bitplane and arithmetic coding engine, we introduced a bitplane coding strategy with parallel coefficient processing (BPC-PaCo) that does not hold dependencies among coefficients, allowing efficient implementations in GPUs [16–19]. This engine can effectively exploit the parallelism of SIMD architectures, which results in high speedup factors and lower power consumption with respect to the fastest implementations of JPEG2000, either executed in multi-core CPUs or in GPUs. Evidently, BPC-PaCo is not compliant with the standard, but it does *not* sacrifice quality scalability and it penalizes coding performance only about 2%. For the DWT, we also proposed an efficient architecture in [2] that achieves high performance in GPUs.

The first implementation of the end-to-end codec for GPUs was presented in [20]. Nonetheless, that implementation is only able to process individual high-resolution images. This paper presents a vastly improved implementation that processes video sequences in real-time thanks to the introduction of stream management with multiple CPU threads, a double-buffer strategy, and event handling to synchronize GPU and CPU operations. Experimental results achieved with consumer-grade GPUs suggest that the proposed codec achieves a throughput that allows encoding and decoding 4K video in real-time and yields highly better power consumption ratios than JPEG2000 codecs executed in CPUs.

The rest of the paper is structured as follows. Section 2 explains the basics of the Nvidia GPU architecture. Section 3 briefly describes the different parts of the JPEG2000 standard. Section 4 describes the proposed end-to-end codec in detail. Section 5 provides experimental results achieved when coding high resolution video in two GPUs and compares our results with Kakadu [21], one of the best multi-thread JPEG2000 implementations. The last section concludes summarizing this work.

2. Overview of Nvidia GPUs

Nvidia GPUs are hardware devices with tens of individual computing units called streaming multiprocessor (SMs). These SMs can work independently, allowing the GPU to process different sequences of operations, called streams, in parallel. This permits the execution of multiple algorithms in an interleaved fashion, which increases the opportunities for parallelism and thereby the throughput achieved. The SMs execute multiple 32-wide SIMD instructions (i.e., vector instructions) simultaneously.

GPUs from Nvidia employ the CUDA programming model, which defines a computation structure composed by (potentially) hundreds of thousands of threads grouped into warps (each with 32-threads), with each warp assigned to a thread block [22]. While the hardware device executes 32-wide SIMD instructions, a software CUDA thread is the virtualization of one of the lanes of the instruction. From the first CUDA-compatible architecture (v1.0) up to Pascal (v6.2), warps execute instructions in a lock-step synchronous fashion, featuring an implicit synchronization at the end of any divergence [23]. From Volta (v7.0) onward, the implicit synchronization is not included at the end of the branching instructions, and must be coded explicitly if needed [24]. Warps in a thread block are executed asynchronously and can cooperate via on-chip fast memories, using explicit synchronizing barrier instructions when required.

The CUDA memory model is hierarchically organized as follows: there is a space of local memory private to each thread, a shared memory private to each thread block, and a global memory public to all threads in the kernel application. From a microarchitecture point of view, the local memory reserved per thread is located either in the registers or the off-chip memory, depending on the available resources. In the proposed implementation, the host memory (CPU RAM) and the device memory (GPU VRAM) are accessed as different, non-unified memory regions, explicitly managing the moment and the amount of data which are copied between them.

3. Overview of JPEG2000

Our GPU codec carries out the same steps as JPEG2000, so they are briefly described below. Depending on the encoding mode employed, either lossy or lossless, the operations involved may be irreversible or reversible, respectively. Irreversible operations improve the compression ratio though they sacrifice image quality slightly.

The first stage of the coding pipeline is the DWT. Our implementation uses a lifting scheme approach [25] due to its low computational complexity. It applies a series of arithmetic operations first row by row and then column by column. The DWT outputs four different subbands, with three of them including smaller detail images and the fourth including the original image at lower resolution and higher energy. These operations are carried out (typically) 5 times within the fourth subband, with each iteration in a lower resolution subband. For lossy compression, the operations employ floating-point arithmetic, so the resulting data are converted to integers before the next stage. This conversion is performed via deadzone quantization [1].

The second stage applies bitplane coding with arithmetic coding. The wavelet coefficients are conceptually partitioned in small sets of typically 64×64 wavelet coefficients, called codeblocks. The strategy adopted to process each codeblock consists in coding

the most relevant information first. The data are divided in bitplanes, with each bitplane containing the set of bits from the same binary position of the unsigned binary representation of each coefficient. Encoding begins from the most significant bitplane (i.e., the one with the highest magnitude within the codeblock) to the lowest one. In JPEG2000, each bitplane is scanned in three coding passes. The first pass is called Significance Propagation Pass. This pass only processes the bits of those coefficients that have at least one significant neighbor. A coefficient is called significant from that bitplane that holds the first non-zero bit to the lowest. The second coding pass is called Magnitude Refinement Pass. It visits the coefficients that are significant in higher bitplanes. The Cleanup Pass processes the coefficients not visited in the previous passes. This coding strategy aims to code the information which holds more information first, effectively reducing distortion [26]. Each bit emitted by the bitplane coder is fed to the arithmetic coder along with its contextual information. The context considers the number of neighbors that are significant, employed to determine a probability for the processed bit. This probability is employed by the arithmetic coder to generate the final bitstream.

The compressed data of codeblocks can be truncated to fit a target bitrate. The method to carry out this optimization process is not defined in the standard, so each implementation may adopt its own solution. The final stage reorganizes the data and adds ancillary information needed by the decoder to decode the original image. The decoder carries out the same steps of the encoder in reverse order.

4. Proposed Codec

The proposed codec is implemented in CUDA. CUDA is employed instead of OpenCL because it provides the latest improvements in the newest architectures. JPEG2000 exposes fine-grain parallelism in all coding stages except for the bitplane coder. Except from the bitplane and arithmetic coder, our proposal produces the same output as JPEG2000 in each stage employing a parallel architecture that extracts most of the GPU performance. BPC-PaCo is employed in the bitplane coding engine [18, 19]. As previously stated, this engine is not compliant with the standard though it preserves the same features and allows parallelism at a fine-grain level.

Algorithm 1 describes the main steps of the proposed codec. Fig. 1 also illustrates its main stages. First, the required memory to process the entire video is allocated in the host CPU RAM (lines 1-2) and in the GPU DRAM, which are respectively referred to as M^H and M^D. Next, two CPU threads are created, denoted by t_1 and t_2 in Algorithm 1, to manage the input/output from/to the hard disk (lines 3-6 and 10-13, respectively). The codec utilizes a double-buffer strategy per stream. This double-buffer is employed for both reading the raw data and writing the compressed file, so four buffers are allocated for each stream. These buffers are referred to as $M^H[i]$ and $M^D[i]$ for the input, and $M^H[o]$ and $M^D[o]$ for the output, with $\{i,o\} \in \{1..2\}$. When reading, the data from one buffer are processed while the other is filled. For writing, the compressed data are transferred to the host from one GPU buffer while the other is already empty and can be filled with compressed data from the frame that is being processed. This buffer structure enables the parallelization of the processing task in two streams, removes the risk of a system memory overflow and increases the utilization of the system resources. Both threads are constantly checking the buffers to start data transfers as soon as possible.

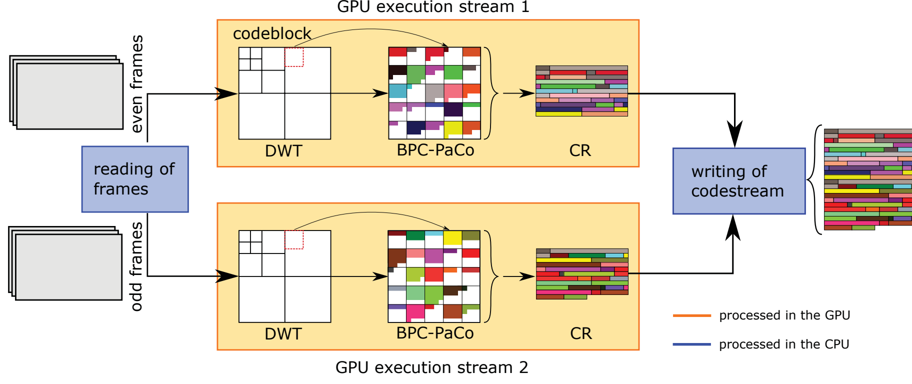


Figure 1. Illustration of the steps performed by the proposed video codec using two streams of execution.

Algorithm 1. Main routine of the codec

1: CPUMemoryAllocation()
2: GPUGlobalMemoryAllocation()
3: **for** each empty $M^H[i]$ **do** (t_1)
4: $\quad M^H[i] \leftarrow$ HDRead()
5: $\quad M^D[i] \leftarrow M^H[i]$
6: **end for**
7: $D \leftarrow$ DWT_Q($M^D[i]$) ($S_{1,2}$)
8: $\{B_l\} \leftarrow$ BPC_AC(D)
9: $M^D[o] \leftarrow$ CR($\{B_l\}$)
10: **for** each filled $M^D[o]$ **do** (t_2)
11: $\quad M^H[o] \leftarrow M^D[o]$
12: $\quad$ HDWrite($M^H[o]$)
13: **end for**

From lines 7 to 9 in Algorithm 1 the GPU functions, or kernels, to code a frame are called. The compression of each frame is carried out with three kernels. Two GPU streams, denoted by $S_{1,2}$, are employed to process a maximum of two frames simultaneously. Typically, each kernel transfers the data to be processed from the global memory $M^D[i]$ to the local memory R to accelerate memory accesses. The kernel DWT_Q($\cdot$) receives the original frame data as input and generates quantized wavelet coefficients that are the input of BPC_AC($\cdot$). BPC_AC($\cdot$) generates a bitstream per codeblock, referred to as B_l, with $l \in \{1..\widehat{L}\}$, $\widehat{L}$ being the number of codeblocks in each frame. This set of bitstreams is reorganized in the last kernel CR($\cdot$), which also adds ancillary information for decoding. This kernel does not transfer the compressed data to local registers since it only needs to reorganize the data in global memory.

As illustrated in Fig. 1, the use of two GPU streams allows the processing of two frames in parallel, increasing the throughput of the codec. Evidently, the three stages of the coding pipeline are carried out sequentially in each stream. Once a frame is coded, the resulting data are sent asynchronously to the host memory and the stream begins processing the next frame immediately. The three kernels are devised and implemented to extract fine-grain parallelism in the GPU. The proposed GPU-oriented architecture is able to process either high-resolution images or video in real time. Next, a brief description of each kernel is provided.

4.1. Discrete wavelet transform

The adopted DWT implementation [2] in our codec employs a register-based acceleration strategy [27] that transfers data from the global memory $M^D[i]$ to local registers, avoiding the use of shared memory. Threads communicate among them via shuffle instructions. This strategy allows data reuse and sharing, taking advantage of the fine-grain parallelism and data access locality of the algorithm. First, the image is conceptually divided in blocks that are independently processed by warps. This permits coarse-grain parallelism, populating more SMs of the GPU. The blocks take into account data dependencies of the transform, so they include some samples from adjacent blocks forming a halo. The halo is needed to obtain the same result as if the DWT was applied to the whole image at once. Within each block, the DWT is applied via the lifting scheme, which alternatively processes in the vertical and horizontal axis odd and even samples. If the compression mode is lossy, quantization is applied after the DWT since the next kernel requires integers.

4.2. Bitplane coding with parallel coefficient processing

The coefficients resulting from the DWT_Q($\cdot$) are conceptually partitioned in small sets called codeblocks. Typically, each codeblock contains 64$\times$64 coefficients. They are transferred to the local memory to speed up the processing, like in the previous kernel. Codeblocks do not hold dependencies among them, so they are processed independently. The processing of codeblocks in parallel requires coarse-grain, control-divergent strategies that are employed in many implementations of the original JPEG2000 standard. In addition to this parallelism, the coding engine BPC-PaCo employed in this stage extracts fine-grain parallelism in the coding of the codeblock.

BPC-PaCo is based on bitplane coding, like JPEG2000. A particular feature of BPC-PaCo is that it conceptually divides the codeblock in 32 columns holding two coefficients each. Each codeblock is processed by a warp, and each 2-coefficient column is processed by a thread of the warp. Each thread carries out a modified version of significance coding that does not require cleanup, and the magnitude refinement pass. To employ only 2 coding passes instead of 3 like in JPEG2000 significantly increases the throughput [19]. Each emitted bit is coded via context-based arithmetic coding. To form the context of the coefficient, threads need to cooperate among them so that each coefficient can obtain information of all its adjacent neighbors. Again, this cooperation is performed via shuffle instructions. BPC-PaCo utilizes 32 arithmetic coders so that each thread in the warp can code all bits that it emits. The codewords generated by the arithmetic coders are interleaved in an optimal fashion in the final bitstream generated for the codeblock. The result of kernel BPC_AC($\cdot$) is the set $\{B_l\}$ that contains a bitstream per codeblock, with $l \in \{1..\widehat{L}\}$ and $\widehat{L}$ being the number of codeblocks per component. The probability model employed by the arithmetic coders is static, i.e., it employs pre-defined probabilities that are computed via a training set of images. This coding strategy permits the use of coarse- and fine-grain parallelism, since both the codeblocks and the coefficients within them are coded in parallel.

	SMs	*cores × SM*	*clock frequency*	*memory bandwidth*	*peak FP32 throughput*	*TDP*	*memory size*
GTX 1080 Ti	28	128	1923 MHz	484 GB/s	13.78 TFlops	250 W	11 GB
GTX 960M	5	128	1176 MHz	80 GB/s	1.5 TFlops	60 W	2 GB

Table 1. Features of the GPUs employed.

4.3. Codestream reorganization (CR)

The bitstreams produced for each codeblock have different size depending on the data coded, as depicted in Fig. 1. This results in data scattered in the output buffer of the global memory. The final stage of the coding process reorganizes these data putting them in a compact structure that can be transferred to the main memory of the host $M^H[o]$ and then written to the disk. This stage also includes auxiliary information in the final codestream for decoding.

When a warp compresses a codeblock, the lengths of the bitstreams are stored in a vector of integers $L = \{L_1, L_2, \cdots, L_{\hat{L}}\}$. Then an aggregated list of lengths, i.e., $L' = \{0, L_1, L_1 + L_2, \cdots, L_1 + \cdots + L_{\hat{L}}\}$ is generated via the Device Scan primitive from the Nvidia CUB framework [28]. To accelerate the access to this list, a fast lookup table, denoted by $\mathrm{LUT}_{L'}$ is created. This LUT is generated applying a binary search over L' in which each position represents some positions of the original map. Our experience indicates that speedups about 2× are achieved by using such a strategy. Kernel $\mathrm{CR}(\cdot)$ then uses this LUT so that each thread transfers 2 bytes of the codeblock's data to the final output buffer.

5. Experimental Results

The throughput achieved by the proposed codec is compared with Kakadu v7.A.2 [21] in the experiments below. Kakadu is among the fastest implementations of JPEG2000, with multi-thread support for multi-core CPUs. It is programmed in C++ and is heavily optimized via assembly instructions. In the tests below, Kakadu is executed in a platform that has 4 AMD Opteron 6376 CPUs running at 2.3 GHz, employing a total of 32 threads of execution. Results from other JPEG2000 implementations in GPUs [11, 12] are not included herein because their throughput is similar or inferior to that of Kakadu, with the exception of Comprimato [10], which does not offer any option to test its implementation. Our codec is executed in the consumer-grade GPUs reported in Table 1, namely, the high-end GTX 1080 Ti for desktops, and the low-end GTX 960M for laptops. The tests code a video sequence of 948 frames of size 2048×832, gray-scale, and bit-depth of 8 bits per sample. The results shown below do not consider the I/O time needed to read/write the files from/to the disk since that may affect execution times significantly depending on the device employed. In all implementations, data are read from the host memory, where they are preloaded before starting the execution.

The first test evaluates the throughput achieved by our codec when using 1 or 2 GPU streams. The results are depicted in Fig. 2. The vertical axis reports the throughput achieved, in Mega samples per second (Msamples/sec.). The results for 1 and 2 streams are depicted for each GPU and for the encoding and decoding process. Using 2 streams provides a performance increase about 26% (7%) in the encoding process and 22% (9%)

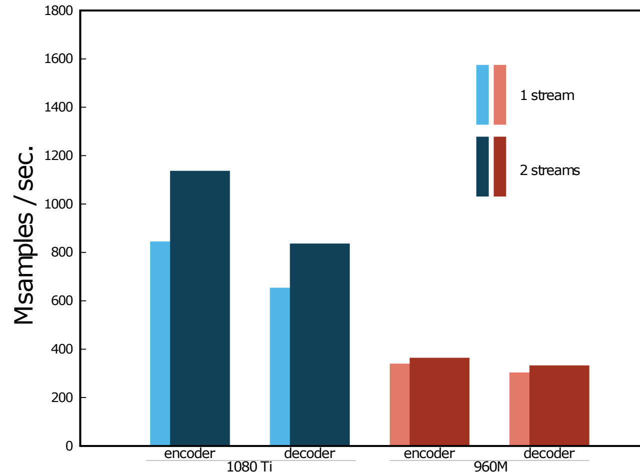


Figure 2. Evaluation of the throughput achieved when using 1 and 2 execution streams.

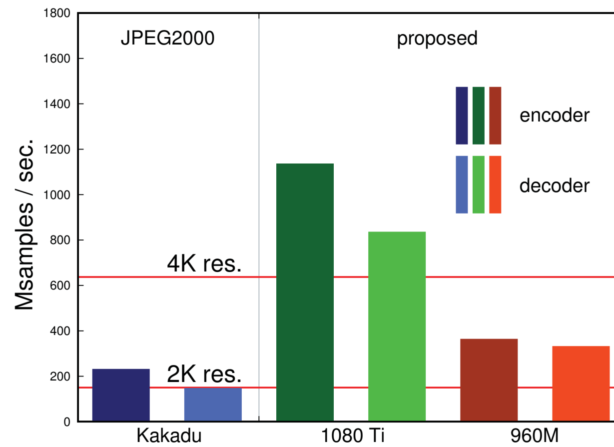


Figure 3. Evaluation of the throughput achieved by the proposed codec and Kakadu.

in the decoding process for the 1080 Ti (960M) GPU. The performance gain depends on the peak throughput of each GPU. The 1080 Ti has ample resources to process more than one frame whereas the 960M almost saturates its resources when coding a single frame (i.e., 1 stream).

The second test evaluates the throughput of the proposed codec running 2 streams and Kakadu. Fig. 3 depicts the obtained results. The proposed codec executed in the 1080 Ti yields a throughput about $5\times$ higher than that of Kakadu. For the 960M, the throughput achieved is about $2\times$ higher than that of Kakadu. Nonetheless, we recall that Kakadu is executed in an expensive multi-CPU platform, whereas the proposed codec employs commodity GPUs. Fig. 3 also depicts the throughput needed to process digital cinema video at resolutions of 2K and 4K in real time (straight horizontal lines). The results suggest that the proposed codec running in the 1080 Ti (960M) can process 4K

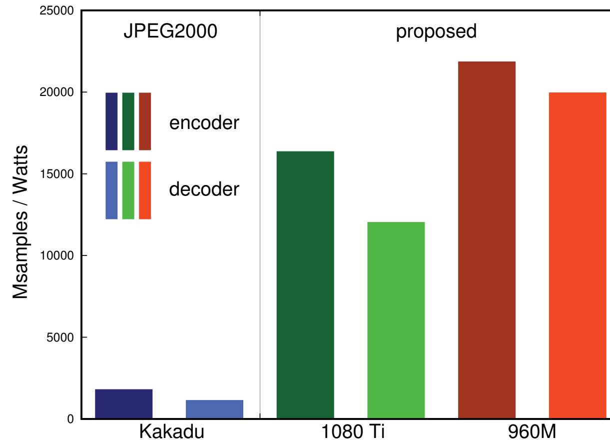


Figure 4. Evaluation of the power efficiency achieved by the proposed codec and Kakadu.

(2K) video in real time.

The third test evaluates power consumption. Fig. 4 depicts the results yield by Kakadu and our codec. In this case, the vertical axis reports Msamples processed per Watt consumed. Kakadu employs four high-end AMD Opteron processors, each with a thermal design power (TDP) of 115W, whereas the 1080 Ti and 960M GPUs have a TDP of 250W and 60W, respectively. The low power consumption of the 960M makes it the most efficient, being approximately 12$\times$ more power efficient than Kakadu for encoding and about 17$\times$ for decoding. The proposed codec executed in the 1080 Ti is less power-hungry than Kakadu too, with increases in efficiency about 9$\times$ and 10$\times$ for the encoder and decoder, respectively.

6. Conclusions

The JPEG2000 standard is mainly devised to exploit the coarse-grain parallelism provided in CPUs. When employed to code high-resolution video in scenarios such as TV production or digital cinema, implementations need specialized hardware or expensive computer platforms to meet real-time requirements. So far, implementations for cheaper devices such as GPUs are not able to achieve high throughput because the innermost algorithms of the coding system do not exhibit enough fine-grain parallelism. This paper introduces a fully parallel end-to-end codec that employs the framework of JPEG2000 but that provides –in all stages of the coding pipeline– distinct modifications that permits the use of fine-grain parallelism. This can be effectively exploited when executed in architectures that highly rely on SIMD parallelism such as those found in Nvidia GPUs. Experimental results coding high-resolution video indicates that the proposed codec is 5$\times$ faster than the most efficient implementations of JPEG2000 while reducing the power consumption more than 10$\times$. None of the advanced features of JPEG2000 are sacrificed in the proposed codec, so it is ideal for scenarios that deal with massive data sets and/or power constraints.

References

[1] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image compression fundamentals, standards and practice*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.

[2] 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, Dec. 2015.

[3] T. M. Quan and W.-K. Jeong, “A fast discrete wavelet transform using hybrid parallelism on GPUs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3088–3100, Nov. 2017.

[4] W. J. van der Laan, A. C. Jalba, and J. B. Roerdink, “Accelerating wavelet lifting on graphics hardware using CUDA,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 132–146, Jan. 2011.

[5] S. Datla and N. S. Gidijala, “Parallelizing motion JPEG 2000 with CUDA,” in *Proc. IEEE International Conference on Computer and Electrical Engineering*, Dec. 2009, pp. 630–634.

[6] R. Le, I. R. Bahar, and J. L. Mundy, “A novel parallel tier-1 coder for JPEG2000 using GPUs,” in *Proc. IEEE Symposium on Application Specific Processors*, Jun. 2011, pp. 129–136.

[7] J. Matela, V. Rusnak, and P. Holub, “Efficient JPEG2000 EBCOT context modeling for massively parallel architectures,” in *Proc. IEEE Data Compression Conference*, Mar. 2011, pp. 423–432.

[8] M. Ciznicki, K. Kurowski, and A. Plaza, “Graphics processing unit implementation of JPEG2000 for hyperspectral image compression,” *SPIE Journal of Applied Remote Sensing*, vol. 6, pp. 1–14, Jan. 2012.

[9] M. Ciznicki, M. Kierzynka, P. Kopta, K. Kurowski, and P. Gepnerb, “Benchmarking JPEG 2000 implementations on modern CPU and GPU architectures,” *ELSEVIER Journal of Computational Science*, vol. 5, no. 2, pp. 90–98, Mar. 2014.

[10] Comprimato. (2014, Apr.) Comprimato. [Online]. Available: http://www.comprimato.com

[11] (2016, Jun.) CUDA JPEG2000 (CUJ2K). [Online]. Available: http://cuj2k.sourceforge.net

[12] (2016, Jun.) GPU JPEG2K. [Online]. Available: http://apps.man.poznan.pl/trac/jpeg2k/wiki

[13] *High Throughput JPEG 2000 (HTJ2K): Call for Proposals*, ISO/IEC Std., 2017, document ISO/IEC JTC 1/SC29/WG1 N76037.

[14] D. Taubman, A. Naman, and R. Mathew, “FBCOT: a fast block coding option for JPEG 2000,” in *Proc. SPIE Applications of Digital Image Processing*, vol. 10396, Sep. 2017, pp. 1–18.

[15] D. Taubman, A. Naman, R. Mathew, and M. D. Smith, “High throughput JPEG 2000 (HTJ2K): New algorithms and opportunities,” *SMPTE Motion Imaging Journal*, vol. 127, no. 9, pp. 1–7, Oct. 2018.

[16] F. Auli-Llinas, “Stationary probability model for bitplane image coding through local average of wavelet coefficients,” *IEEE Trans. Image Process.*, vol. 20, no. 8, pp. 2153–2165, Aug. 2011.

[17] 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, Jun. 2014.

[18] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, “Bitplane image coding with parallel coefficient processing,” *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 209–219, Jan. 2016.

[19] 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, Aug. 2017.

[20] C. de Cea-Dominguez, P. Enfedaque, J. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, “High throughput image codec for high-resolution satellite images,” in *Proc. IEEE International Geoscience and Remote Sensing Symposium*, Jul. 2018, pp. 6524–6527.

[21] D. Taubman. (2018, Dec.) Kakadu software. [Online]. Available: http://www.kakadusoftware.com

[22] “CUDA, C Programming guide,” Tech. Rep., Jan. 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[23] Nvidia. (2018, Jan.) Warp level primitives. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

[24] ——. (2019, Jun.) Nvidia Tesla V100 GPU architecture. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[25] W. Sweldens, “The lifting scheme: A construction of second generation wavelets,” *SIAM Journal on Mathematical Analysis*, vol. 29, no. 2, pp. 511–546, 1998.

[26] F. Auli-Llinas and M. W. Marcellin, “Scanning order strategies for bitplane image coding,” *IEEE Trans. Image Process.*, vol. 21, no. 4, pp. 1920–1933, Apr. 2012.

[27] A. Chacon, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, “Boosting the FM-index on the GPU: effective techniques to mitigate random memory access,” *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 12, no. 5, pp. 1048–1059, Sep. 2015.

[28] Nvidia. (2018, Dec.) CUB framework. [Online]. Available: https://nvlabs.github.io/cub/

Chapter 4

GPU Oriented Architecture for an End-to-End Image Video Codec Based on JPEG2000

Digital Object Identifier

GPU-oriented architecture for an end-to-end image/video codec based on JPEG2000

CARLOS DE CEA-DOMINGUEZ[1], JUAN C. MOURE [2], JOAN BARTRINA-RAPESTA[1], AND FRANCESC AULÍ-LLINÀS, (Senior Member, IEEE)[1]
[1]Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain (phone: +34 935811861; fax: +34 935813443; e-mail: carlos.decea@uab.cat, joan.bartrina@uab.cat, francesc.auli@uab.cat)
[2]Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain (e-mail: juancarlos.moure@uab.es)
Corresponding author: Carlos de Cea-Dominguez (e-mail: carlos.decea@uab.cat).

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under Grants TIN2017-84553-C2-1-R and RTI2018-095287-B-I00 (MINECO/FEDER, UE), and by the Catalan Government under Grants 2017SGR-463 and 2017SGR-313. **Copyright (c) 2020 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.**

ABSTRACT Modern image and video compression standards employ computationally intensive algorithms that provide advanced features to the coding system. Current standards often need to be implemented in hardware or using expensive solutions to meet the real-time requirements of some environments. Contrarily to this trend, this paper proposes an end-to-end codec architecture running on inexpensive Graphics Processing Units (GPUs) that is based on, though not compatible with, the JPEG2000 international standard for image and video compression. When executed in a commodity Nvidia GPU, it achieves real time processing of 12K video. The proposed S/W architecture utilizes four CUDA kernels that minimize memory transfers, use registers instead of shared memory, and employ a double-buffer strategy to optimize the streaming of data. The analysis of throughput indicates that the proposed codec yields results at least 10× superior on average to those achieved with JPEG2000 implementations devised for CPUs, and approximately 4× superior to those achieved with hardwired solutions of the HEVC/H.265 video compression standard.

INDEX TERMS Wavelet-based image coding, high-throughput image coding, JPEG2000, GPU, CUDA.

I. INTRODUCTION

OVER the past decades, the computational complexity of image and video coding systems has increased notably. In the early nineties, the JPEG standard (ISO/IEC 10918) [1] employed the low-complexity discrete cosine transform [2] and Huffman [3] coding. Ten years after, the JPEG2000 standard (ISO/IEC 15444) [4] introduced more computationally demanding algorithms such as the discrete wavelet transform (DWT) [5] and bitplane coding [6]. In the last years, HEVC/H.265 (ISO/IEC 23008) [7] doubled the compression efficiency of previous standards by using complex techniques that exploit intra- and inter-redundancy of frames. Nowadays, most codecs (including JPEG2000 and HEVC) provide advanced features such as scalability by quality, interactive transmission, and error resilience, among others. To do so, they use algorithms that scan, transform, and code the samples[1] of the image multiple times, consuming significant processing time even when executed in the latest processors.

JPEG2000 is a widespread standard in fields that deal with large sets of images and/or videos. Its coding pipeline has three main stages [8]. The first reduces the image redundancy through a color transform (CT) and the DWT. The second employs bitplane coding together with arithmetic coding to reduce the statistical redundancy of wavelet coefficients. The third reorganizes the data to produce the final codestream. The high computational complexity of these stages poses

[1]A sample is the basic unit of a digital image, representing a level of brightness in a grayscale or color component (each RGB pixel has three samples).

a challenge to meet the real-time requirements of some scenarios. In Digital Cinema, for instance, JPEG2000 needs to be implemented in Field-Programmable Gate Arrays to process 2K (i.e., 2048×1024) and 4K (i.e., 4096×2048) resolution [9]. In medical and remote sensing applications, dedicated servers and workstations are employed to manage and store the large quantity of images that are produced daily [10], [11]. This has motivated many works in the literature that propose hardware architectures to accelerate particular stages of the JPEG2000 coding pipeline [12]–[22].

Highly parallel architectures may help to reduce processing time and costs in some environments. Graphics Processing Units (GPUs) may be ideal due to their high throughput, low cost, and widespread availability. Their architecture is mainly based on the Single Instruction Multiple Data (SIMD) paradigm, which executes a flow of instructions on multiple data in a lock-step synchronous way. When the program allows data (in addition to task) parallelism, thousands of threads can be executed in parallel, achieving a throughput that is potentially an order of magnitude higher than that achieved by conventional Central Processing Units (CPUs) [23]. This is in part because the architecture of the CPUs is more based on the Multiple Instruction Multiple Data (MIMD) paradigm, which allows the asynchronous execution of fewer threads over different sets of data.

Most of the workload in the first stage of the JPEG2000 pipeline lies in the DWT, which is well-suited to the SIMD paradigm. The first implementations of the DWT for GPUs appeared in the 2000s making use of the graphics pipeline [24]–[26]. Later, the use of the Compute Unified Device Architecture (CUDA) programming language introduced by Nvidia increased the throughput of such implementations significantly [27]–[31]. Recently, we proposed a register-based implementation of the DWT for GPUs [32] that yields 40× speedups compared to CPU implementations. Similar results are also achieved in [33].

In general, the DWT takes 15% of the total execution time of the codec. The most expensive stage is the bitplane and arithmetic coding, which spends about 80% of the time. This stage poses the major challenge for GPUs because it is not well-suited to the SIMD paradigm. In this stage, the wavelet-transformed image is partitioned in small sets of typically 64×64 wavelet coefficients, called codeblocks, and codes them independently. This provides coarse-grain parallelism. The coding within each codeblock must be carried out by a single thread, since there exist causal relationships among coefficients. This means that the coding of a coefficient depends on the output of the previous, so they can not be processed in parallel. Even so, there have been efforts to implement this stage in GPUs [34]–[40], though these solutions do not to fully occupy the resources of the GPU due to the lack of fine-grain parallelism. In 2014, we started a line of research [41]–[45] focused on providing fine-grain parallelism to this stage without sacrificing any feature of the system. The goal was not to implement the compliant JPEG2000 algorithm, but to redevise it keeping in mind the SIMD architecture of GPUs. The proposed algorithm is not compatible with the standard, but it allows parallel coefficient processing within the codeblock.

Following a similar line, in 2017 the Joint Photographics Experts Group launched a call for proposals with the aim to augment the parallelism in the second stage of the coding pipeline. This new part of JPEG2000 (ISO/IEC 15444-15) adopts the algorithm proposed in [46]. Such algorithm is devised to mostly benefit from the modern instruction sets like AVX2, NEON, and BMI2 included in new CPUs, though it can also be implemented in GPUs [47]. It is about 10× faster than the standard, but it penalizes coding performance in approximately 10%. Also, it sacrifices quality scalability, which is a valued feature of the system since it permits the transmission of an image progressively by quality.

This paper introduces a highly-parallel, GPU-oriented codec based on JPEG2000. The proposed codec is the final piece of our research line that was aimed to explore new coding techniques for image/video compression tailored for the fine-grain parallelism of GPUs. The JPEG2000 framework is employed to show that the proposed techniques can virtually obtain the same coding performance of this standard without sacrificing any feature. Evidently, compliance with the standard is lost since the proposed techniques require significant changes in the core coding system. A preliminary version of the proposed codec was partially described in [48], [49]. This paper vastly improves our previous work by describing the complete coding pipeline with the needed machinery to avoid bottlenecks, providing the color transform and the codestream reorganization stages with an in-depth analysis of the kernel metrics and memory transfers, and reporting extensive experimental tests. The obtained results show that the proposed S/W architecture can process real-time 12K (i.e., 12288×6144) video, achieving a throughput 4× superior to that achieved by the state-of-the-art Nvidia codec of HEVC that is supported by in-chip dedicated hardware.

The rest of the paper is structured as follows. Section II briefly overviews the architecture of Nvidia GPUs and JPEG2000. Section III describes the proposed codec from a top-down perspective and Section IV details each kernel employed. Section V evaluates the throughput of our architecture and compares it to some of the fastest JPEG2000 and HEVC implementations. The last section contains conclusions.

II. BACKGROUND

A. NVIDIA GPU ARCHITECTURE

Nvidia GPUs are hardware devices that are mainly constituted by individual computing units called Streaming Multiprocessors (SMs). Depending on the model and the architecture, a Nvidia GPU may contain from one to tens of SMs. Each SM can work independently, allowing the GPU to process sequences of instructions from different algorithms. Typically, SMs execute multiple 32-wide vector instructions in parallel.

CUDA refers vector instructions as warps. Each lane of

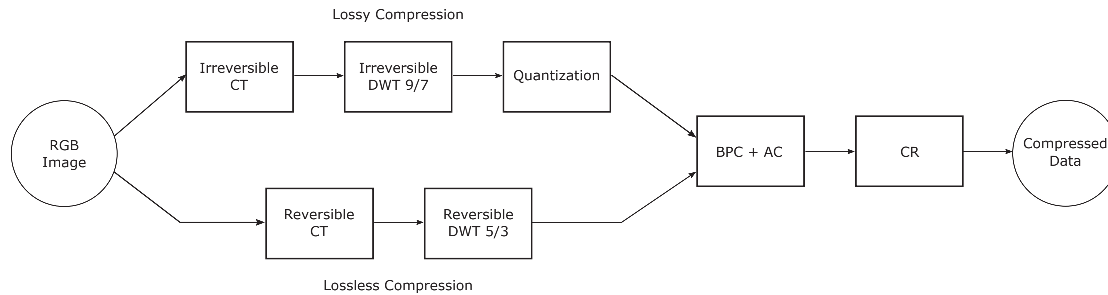


FIGURE 1: JPEG2000 coding pipeline.

a vector is virtualized into a software thread. Aggregations of 32 threads form a warp. A group of warps, called thread block, is assigned to a SM for execution. From the first CUDA-compatible architecture (v1.0) up to Pascal (v6.2), warps are always executed synchronously and in a lock-step fashion, featuring an implicit synchronization at the end of any divergence [50]. Volta (v7.0) introduced a modification in the warp scheduler that allows the execution of warp threads asynchronously [51], so the synchronization among threads must be explicitly programmed when needed. Our codec is adapted to work with both implicit and explicit synchronization.

The memory architecture of the GPU is organized in three levels: global, shared, and local. The size of the global memory is, in general, in the order of GBs and is accessible by all SMs. When this memory is accessed in a coalesced way (i.e., via consecutive positions) the available bandwidth is used efficiently and the latency is minimized. The size of the shared memory is in the order of MBs and its latency is lower than that of the global, though it can only be shared within the thread blocks. The local memory is the fastest though it is also limited in size and is only accessible by the threads within a warp. The data allocated in the local memory are commonly stored in the registers, though they may be temporarily moved to the device memory (i.e., DRAM of the GPU) when the register space is saturated. Typically, the global memory is employed to read and store the application's data, the shared memory is used for communication among threads of different warps, and the local memory is utilized for intermediate computation. The local memory can be shared among threads within a warp via the low-level shuffle operation. This kind of memory sharing technique proved to be very efficient in some applications [32], [52]–[54]. The GPU has two levels of cache, denoted by L1 and L2. The registers and the L1 cache are in the SM. The data transferred from the device memory to the registers passes through the L1 and L2 caches, which are reservoirs of the most recently accessed data to be (possibly) reused in future petitions.

As previously mentioned, each SM runs thread blocks. These blocks can execute code from one or more CUDA functions, called kernels, independently. This allows the parallel execution of many different kernels from a single or various applications. CUDA provides the so-called streams to organize the execution of running kernels. Each stream may process a sequence of kernels of an application in a set of SMs asynchronously from the rest. An appropriate use of the streams optimizes the use of the GPU resources, which can help to increase the throughput.

B. JPEG2000

As previously stated, JPEG2000 is an image/video coding standard employed in professional environments due to its excellent features and performance. The proposed codec carries out almost the same operations as JPEG2000, so they are briefly described herein for completeness. Figure 1 depicts these operations. Depending on whether lossy or lossless compression is needed, some of these operations are irreversible or reversible. As stated before, the first stage of JPEG2000 applies several transformations to the image. The first is carried out for color images, converting the red, green, and blue (RGB) components to the lesser redundant color space YC_bC_r, which holds the luminance information in the first component and the chrominance with respect to blue and red in the second and third components, respectively. This is a pixel-wise operation that holds no dependencies among pixels. It is carried out applying floating-point or integer operations for the irreversible or reversible path, respectively.

The second operation is the DWT. Most implementations apply it via the lifting scheme [55] since it has low computational complexity. The main idea behind this scheme is to first apply a series of arithmetic operations to all rows of the image and then to all columns. These operations can be carried out in parallel to all rows and then to all columns since there are no inter-row/column dependencies. Then, the resulting coefficients are re-ordered taking the coefficients in the even and odd positions in each direction. This produces four different subbands of one quarter the size of the original image. In general, the same procedure is applied again four more times in the subband that contains the low-detail image. The operations carried out in each step apply a low- and high-pass filter. JPEG2000 uses the irreversible CDF 9/7 and the reversible CDF 5/3.

The irreversible filter bank employs floating-point arithmetic, so the resulting coefficients need to be converted to integers before bitplane coding. This operation is called deadzone quantization [8]. It multiplies the coefficients by a step size and keeps the integer part. This operation is

not necessary for the reversible transform since it already produces integer coefficients.

The second main stage of the coding pipeline carries out bitplane coding together with arithmetic coding. As stated before, this stage is applied in each codeblock independently. Through the binary representation of the integer coefficients (without sign), a bitplane is defined as the set of bits from all coefficients in the same binary position. Bitplanes are coded from the most to the least significant. Just after the first non-zero bit of a coefficient is coded (referred to as significance bit), its sign is coded too so that the decoder can reconstruct that coefficient. The bits coded for a coefficient after its significance bit are called refinement bits. The coefficients within a codeblock are scanned in a pre-defined order that visits four rows of coefficients, called stripes, consecutively. In each stripe, coefficients are scanned from the left- to right-most column and, in each column, from the top to the bottom row. JPEG2000 codes each bitplane in three coding passes. The first is called significance propagation. It follows the scanning order processing only those coefficients that have at least one significant neighbor. The second is called magnitude refinement. It processes coefficients that were found significant in previous bitplanes. The third pass processes the remaining coefficients. It is called cleanup. This multiple-pass coding is aimed to code first the information that reduces the most the distortion of the image [6].

Each processed bit is fed to the arithmetic coder together with its contextual information. The context considers the significance, or sign, of its eight neighbors. One of 18 different pre-defined contexts is chosen depending on this information. The context of the coefficient is employed by the arithmetic coder to establish a probability for the currently processed bit, generating a compacted stream of bits.

The output produced in this stage for each codeblock is a bitstream that can be truncated at the end of each coding pass. Like most coding systems, JPEG2000 permits specifying a size for the final codestream, so bitstreams may be truncated to fit the target rate. This rate-distortion optimization procedure is not defined in the standard, so each codec can choose among a great variety of methods [56]. The final operation re-organizes these bitstreams to put them in the compressed file together with ancillary information for decoding. The decoder carries out the same operations in reverse order except the rate-distortion optimization stage, which is not necessary.

III. OVERVIEW OF THE CODEC ARCHITECTURE

A. OVERVIEW

Except for bitplane and arithmetic coding, all operations of the JPEG2000 coding pipeline offer fine-grain parallelism. Our codec implements these operations following the standard, so their input/output is the same as that obtained by a conventional JPEG2000 implementation. To use the JPEG2000's bitplane and arithmetic coder would significantly hinder the throughput of the GPU, so this is the only stage that is not compliant with the standard. This stage is replaced by the coding engine proposed in [44], [45]. The aim of our codec is to code large quantities of images. The input data set may contain frames of a video sequence or images of the same size. For convenience, frame is used to refer both terms in the following.

When possible, the proposed architecture joins operations in a single kernel instead of using a straightforward approach that uses one kernel per operation. Within the same kernel, the data are always accessed in the same fashion and the data types do not change. This permits the kernels to maximize the use of local memory in detriment of shared memory, using a register-based strategy [52]–[54] that minimizes memory latencies. When the data set needs to be re-organized or the data type is changed, then the data are transferred to the global memory preparing them for the next kernel. This architecture minimizes the overall memory transfers and significantly increases performance.

Algorithm 1 describes the main routine of the codec. Its architecture is also illustrated in Figure 2. First, all memory needed during the coding process is pre-allocated both in the host RAM and the device DRAM, which are respectively referred to as $\mathcal{M}^H$ and $\mathcal{M}^D$. This allocation (lines 1 and 2 in the algorithm) considers the space needed for a double buffer strategy to load the frames (see below), auxiliary memory structures, and number of GPU streams employed. The host RAM allocation is performed in pinned memory[2] to avoid memory positions requests to the CPU when transferring data. This allocation greatly improves the memory bandwidth achieved in some GPUs. See, for instance, in Table 1 the difference in the bandwidth achieved by our codec when coding a 4K video (with the test environment described in Section V) using pinned or paged memory. To use pinned memory in the Nvidia GTX 1080 Ti (Pascal architecture) almost doubles the bandwidth achieved as compared to paged memory. For the RTX 2080 Ti (Turing architecture), the differences are much smaller due to the use of DDR4 RAM modules in the host, though there is a slight increase of 4% in the bandwidth achieved. It is worth noting that the practical maximum speed of the PCI-E 3.0 bus employed is 13.2 GB/s (with 15.8 GB/s of theoretical maximum), so our codec yields maximum bandwidth in practice.

Memory transfers are programmed to be asynchronous so they can absorb variations in the time spent to process each frame. The reading of frames is managed by a thread, denoted by t_1 in Algorithm 1, that is executed by the host. Each stream, denoted by $S_j, j \in \{1..\widehat{S}\}$ with $\widehat{S}$ being the number of streams, employs two input buffers in both $\mathcal{M}^H$ and $\mathcal{M}^D$ so that when a buffer is being processed the other can be filled. These buffers are referred to as $\mathcal{M}^H[i]$, $\mathcal{M}^D[i]$ with $i \in \{1..2\widehat{S}\}$. This filling is carried out in lines 3-6. t_1 continuously checks if there is any empty buffer in $\mathcal{M}^H$. If so, it reads the data from disk and transfers them to $\mathcal{M}^H$. Then, it issues an asynchronous copy to the device memory

[2]Pinned memory indicates that the allocated space has a fixed location in the RAM module(s) during the whole execution.

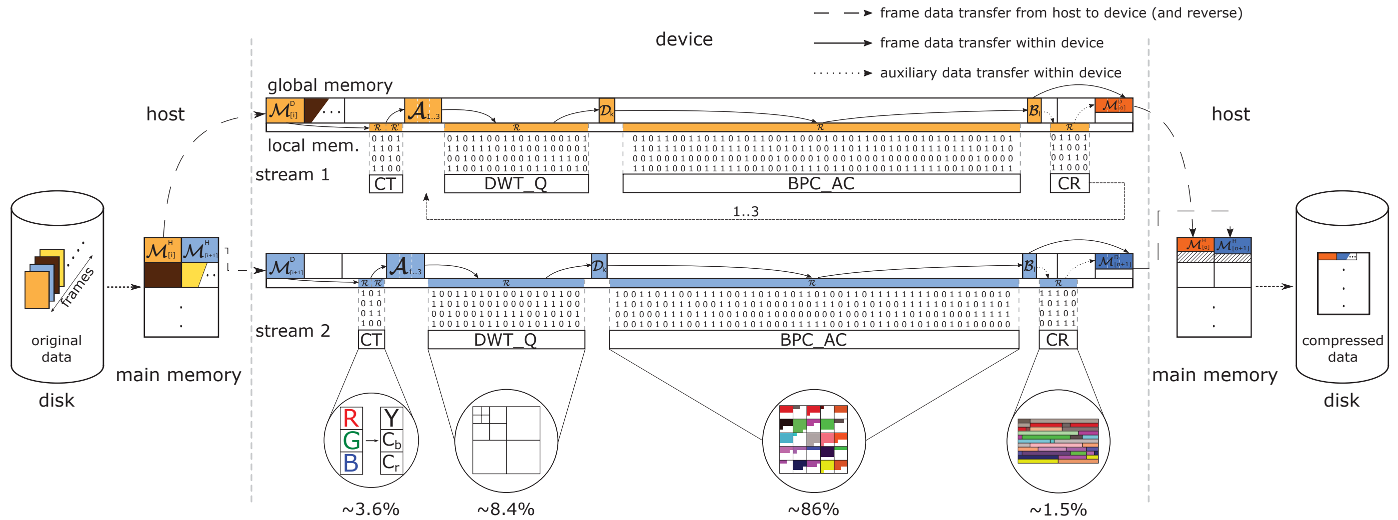


FIGURE 2: Illustration of the codec architecture when using 2 CUDA streams. The cycle of the data is as follows. First, frame data (individually identified by color) are read from disk to a RAM buffer. Then the data are managed by a stream in the GPU. Within the device the data are transferred from global memory $\mathcal{M}^D$ to local memory $\mathcal{R}$ and inversely before and after running each kernel. The kernel execution is illustrated by the matrix of 0s and 1s. Each stream processes the three components of the frame before transferring the compressed data back to the host memory $\mathcal{M}^H$ and disk.

	GTX 1080 Ti		*RTX 2080 Ti*	
	$\mathcal{M}^H \rightarrow \mathcal{M}^D$	$\mathcal{M}^D \rightarrow \mathcal{M}^H$	$\mathcal{M}^H \rightarrow \mathcal{M}^D$	$\mathcal{M}^D \rightarrow \mathcal{M}^H$
paged	7.5 GB/s	6.226 GB/s	12.746 GB/s	12.502 GB/s
pinned	12.431 GB/s	12.725 GB/s	13.182 GB/s	13.175 GB/s
speedup	1.65	2.04	1.03	1.05

TABLE 1: Evaluation of the memory bandwidth achieved by our codec when transferring data from host to device ($\mathcal{M}^H \rightarrow \mathcal{M}^D$) and device to host ($\mathcal{M}^D \rightarrow \mathcal{M}^H$) with pinned and paged memory, for two different GPUs.

in line 5. t_1 is active until all frames have been buffered. The data are read and stored considering their original bit-depth to optimize transfers and memory space. In general, 8-bit integers are employed.

The writing of the compressed data to the disk is done similarly by thread t_2, which is executed by the host in lines 13-16. A double-buffer strategy is also employed so that when a stream finishes coding a frame, it can readily start coding another without waiting for the compressed data to be transferred to the host memory. These output buffers are referred to as $\mathcal{M}^H[o]$, $\mathcal{M}^D[o]$ with $o \in \{1..2\widehat{S}\}$. Again, the data transfer from device to host is carried out via an asynchronous copy in line 14. Once the transfer is done, t_2 writes them to the disk. The data are copied in the disk orderly, i.e., following the same frame order of the original sequence.

Lines 7-12 in Algorithm 1 describe the calls to the kernels and the auxiliary memory structures employed in the GPU. Four kernels are used. The first carries out the color transform. It transfers all frame data from $\mathcal{M}^D[i]$ to local memory converting them to 32-bits integers (floats) for the (ir)reversible path and performs the arithmetic operations on the registers. The result is left in the auxiliary structure denoted by $\mathcal{A}_{1..3}$ using the same data type employed in the

Algorithm 1 Main routine of the codec

```
      1:  CPUMemoryAllocation()
      2:  GPUGlobalMemoryAllocation()
    { 3:  for each empty M^H[i] do
t1  { 4:      M^H[i] ← HDRead()
    { 5:      M^D[i] ← M^H[i]
    { 6:  end for
    { 7:  A_{1..3} ← CT(M^D[i])
    { 8:  for k ∈ {1..3} do
Sj  { 9:      D_k ← DWT_Q(A_k)
    {10:      {B_l} ← BPC_AC(D_k)
    {11:      M^D[o] ← CR({B_l})
    {12:  end for
    {13:  for each filled M^D[o] do
t2  {14:      M^H[o] ← M^D[o]
    {15:      HDWrite(M^H[o])
    {16:  end for
```

kernel. After this, each component is processed independently. The next kernel carries out the DWT and, if using lossy compression, quantization. Our codec employs a rate-distortion optimization method that controls the rate through

	occupancy		*warp efficiency*		*bandwidth (GB/s)*		*time (μs)*		*#inst. ($\times 10^6$)*		*#inst. per sample*	
	2K	*4K*	*2K*	*4K*	*2K*	*4K*	*2K*	*4K*	*2K*	*4K*	*2K*	*4K*
CT$(\cdot)$	90%	87%	100%	100%	483	495	65	255	3.08	12.32	1.47	1.47
DWT_Q$(\cdot)$	84%	90%	97.5%	97.5%	471	511	36	135	2.45	9.81	1.17	1.17
BPC_AC$(\cdot)$	18%	61%	63%	63%	69	189	1150	2000	32.61	107.01	15.54	12.75
CR$(\cdot)$	88%	88%	99%	99%	181	210	15	35	0.82	3.05	0.39	0.36

TABLE 2: Analysis of the codec's kernels when coding a 2K and 4K frame with the Nvidia RTX 2080 Ti.

	registers per thread		*data reading (MB)*				*data writing (MB)*			
			$\mathcal{M}^D \rightarrow \mathcal{R}$		*L2* $\rightarrow$ *L1*		*L1* $\rightarrow$ *L2*		$\mathcal{R} \rightarrow \mathcal{M}^D$	
	2K	*4K*	*2K*	*4K*	*2K*	*4K*	*2K*	*4K*	*2K*	*4K*
CT$(\cdot)$	18	18	6	24	6	24	24	96	24	96
DWT_Q$(\cdot)$	63	63	8.22	33.89	14.93	61.05	9.95	40.18	8.21	32.39
BPC_AC$(\cdot)$	60	60	48.59	251.79	67.01	293.38	37.26	129.58	26.95	108.97
CR$(\cdot)$	24	24	1.04	3.24	1.11	3.46	0.64	2.01	0.87	2.75

TABLE 3: Analysis of the hierarchical memory transfers of the codec's kernels when coding a 2K and 4K frame with the Nvidia RTX 2080 Ti.

the quantization step employed in this operation [56]. It transfers the data from $\mathcal{A}_k$ to the registers, applies the lifting scheme, and leaves the result in $\mathcal{D}_k$. The third kernel is the most complex. It applies bitplane and arithmetic coding. Like the other kernels, it reads the data from the global memory and puts them in the local. These data are organized in codeblocks holding 64×64 coefficients. Each codeblock is processed by an individual warp of 32 threads. The result of this kernel is stored in the set $\{\mathcal{B}_l\}$ that contains one bitstream per codeblock, with $l \in \{1..\widehat{L}\}$ and $\widehat{L}$ being the number of codeblocks per component. The length of each bitstream is not known before coding, so the space for bitstreams $\{\mathcal{B}_l\}$ is pre-allocated amply. As a result, the bitstream data are scattered throughout the whole structure. These data must be compacted before transferring them to the host memory and disk, which is the function of the last kernel. Contrarily to the other kernels, it does not put the frame data to the registers but only the lengths of the generated bitstreams (via pointers to memory positions), so that it can compute the final position of each compressed byte. Then, it re-organizes the compressed frame data in the global memory leaving them in one of the two output buffers.

The decoder employs a similar structure to that of the encoder. It executes the kernels in inverse order, performing the reverse operations.

B. ANALYSIS

Table 2 and 3 report the kernels' metrics obtained via the Nvidia Nsight Compute tool when coding a 2K and 4K frame using the test environment described in Section V. The first kernel (i.e, ICT$(\cdot)$) achieves high occupancy, optimal warp efficiency (since it does not have divergence), and very high memory bandwidth (see Table 2). These results are due to the pixel-wise operation that it carries out. The differences between the 2K and 4K frame with respect to execution time and total number of instructions executed are a 4 fold increase, coinciding with the increase in number of processed samples. We recall that this kernel processes the three image components, whereas the following kernels process only one. As seen in Table 3, the three image components are transferred from $\mathcal{M}^D$ to $\mathcal{R}$ requiring 6 and 24 MB for a 2K and 4K frame, respectively. Once the data are in the SM, they are converted from 8-bit integers to 32-bit integers or floats depending on whether the reversible or irreversible transform is selected. This conversion is seen in the memory transfers when the data are transferred back from the registers to the device memory via the L1 and L2 caches.

The DWT_Q$(\cdot)$ kernel can perform a variable number of transformation levels, typically 5. The metrics reported in Table 2 correspond to the first call to the kernel, which performs the first level of transformation. The achieved occupancy is about 84% for 2K and 90% for 4K. This indicates that other computations can be done while this kernel is running. Similar to the previous kernel, the warp efficiency is almost 100% since there are no divergent paths. The increase in execution time and total number of instructions between 2K and 4K is also proportional to the frame size. As seen in Table 3, this kernel utilizes more registers per thread due to a larger data tile processed by each warp. The data require 8 MB and 32 MB for the 2K and 4K frame, respectively, which approximately correspond to the transfers between $\mathcal{M}^D$ to $\mathcal{R}$ and inversely. The extra data transferred correspond to auxiliary information. The transfers between the L1 and L2 cache are higher than those from the device memory to the registers because this kernel processes the data tiles employing a redundant halo.

As shown by the metrics, the BPC_AC$(\cdot)$ kernel is the most complex. First, the occupancy is much lower than that achieved by the other kernels, especially for 2K frames. This is because 2K frames do not have enough data to fill the resources of the GPU. 4K frames achieve higher occupancy, though it is still below that achieved by the other kernels. Second, the warp efficiency is 63% due to the multiple divergent paths of the algorithm. Third, the memory bandwidth

is much lower than that achieved by the other kernels since BPC_AC(·) is bounded by the latency of the computing instructions [45]. Fourth, the time spent for coding a 2K and 4K frame is not proportional to the frame size. This is due to the low occupancy that is achieved for 2K frames and due to the image content. Let us explain further. The codeblock size is 64×64 regardless of the frame size. This causes that codeblocks of 2K frames have more details (i.e., more entropy) than codeblocks of 4K frames, requiring more instructions to code their information. This is manifested in the total number of instructions and instructions per sample executed, since the 4K frame requires approximately 20% fewer instructions to code each sample. The memory transfers when reading the data are higher than in the other kernels mainly due to the register pool size (see Table 3). Differently from the previous kernels, BPC_AC(·) visits each coefficient of the codeblock many times. The number of visits depends on the codeblock's data, but is approximately 8 or 9 times per coefficient on average. Since the size of the register space is limited, once a coefficient is visited it is transferred back to the device memory so the register can be employed for other coefficients. When the coefficient is needed again, it is transferred from the device memory to the registers. Many of these coefficients are kept in cache and are reused, so the transfers between the L2 and L1 cache are high as well. The data transfers when writing are not as high because the kernel only stores the compressed data. Even so, the data in the compressed bitstream are accessed many times, so the transfers between registers and device memory are higher than in the previous kernels.

The occupancy and efficiency of the CR(·) kernel is similar to that achieved by ICT(·) and DWT_Q(·). The execution time for 4K frames is twice as that needed for 2K. This is because both frames require $5\mu s$ to generate preliminary tables, and then the data to be reorganized are about 1 MB and 3 MB respectively for the 2K and 4K frame,[3] requiring $10\mu s$ and $30\mu s$. The memory bandwidth is lower than that obtained in the first two kernels since the transferred data are already compressed(also seen in Table 3).

This analysis indicates that the BPC_AC(·) kernel consumes most of the total execution time and it achieves the lowest occupancy. This suggests that the codec may underuse the resources of the GPU when coding large sets of images or video unless more workload is feed to the device. The proposed architecture alleviates this issue by employing multiple streams of execution. Each stream processes a frame, so more data are processed in parallel, employing more resources and increasing the overall throughput. See in Figure 3 the throughput achieved by our multiple-streamed codec when encoding 2K and 4K video in the same conditions as before. The results are reported as the number of Mega Samples coded per second (MS/s). The figure depicts the throughput needed to code 4K, 8K, and 12K video in real-time with straight horizontal lines for the convenience of the reader. As seen in the figure, the throughput increases notably when multiple streams are employed. In the case of 2K (4K) video, 13∼14 (7∼8) streams obtain maximum efficiency. Again, the coding of 4K video achieves higher throughput due to the nature of the data.

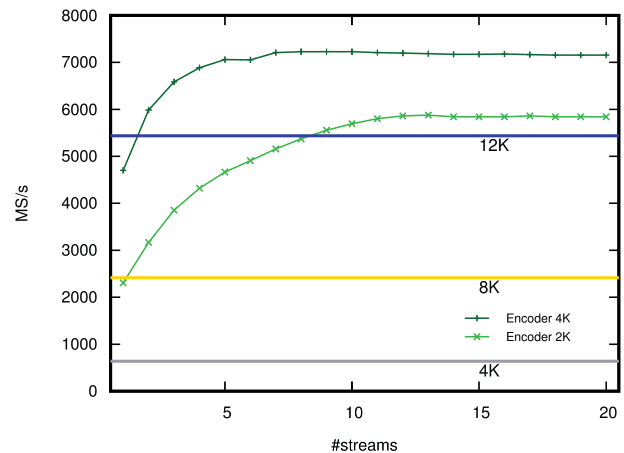


FIGURE 3: Analysis of the throughput achieved by the proposed codec when encoding 2K and 4K video using different number of execution streams, for the RTX 2080 Ti.

Algorithm 2 Kernel routine CT($\mathcal{M}^D[i]$)

1: GPULocalMemoryAllocation()
2: $\mathcal{R}_{1..3} \leftarrow \mathcal{M}^D[i]$
3: $\mathcal{R}'_{1..3} \leftarrow \phi(\mathcal{R}_{1..3})$
4: $\mathcal{A}_{1..3} \leftarrow \mathcal{R}'_{1..3}$
5: **return**($\mathcal{A}_{1..3}$)

As seen in Section V the throughput achieved by the decoder is only slightly lower than that of the encoder because the decoder requires more local memory, which reduces the occupancy. The rest of the decoding process is very similar to that of the encoder, so it is not reported herein for brevity.

IV. DESCRIPTION OF THE KERNELS

Algorithm 2 details the routine of the CT(·) kernel. In this and following kernels, the algorithm describes the main operations that are performed at a thread level. Like in the other kernels, the first instruction allocates the local memory. All kernels only use registers since this increases the throughput. After allocating the required space, the data of the three frame components are transferred from the global memory to the register space, referred to as $\mathcal{R}$ for the input data. This is the only kernel that needs the three components of the frame. It applies a transformation that involves several arithmetic operations, denoted by $\phi(\cdot)$ in line 3, and the result is left in the output register space $\mathcal{R}'$. Then the data are returned to the global memory, ready to be fetched by the next kernel. Both reading and writing in the global memory in this and following kernels is carried out in a coalesced way to maximize memory performance since the GPU stores

[3] 4K frames are compressed more efficiently than 2K frames, so they generate fewer data per sample coded.

Algorithm 3 Kernel routine DWT_Q($\mathcal{A}_k$)

```
1:  GPULocalMemoryAllocation()
2:  R ← A_k
3:  for y ∈ {1..Ŷ} do
4:      for x ∈ {0..1} do
5:          R[y][x] ← φ(R[y][x])
6:      end for
7:  end for
8:  for x ∈ {0..1} do
9:      for y ∈ {1..Ŷ} do
10:         R[y][x] ← φ(R[y][x])
11:     end for
12: end for
13: for y ∈ {1..Ŷ} do
14:     for x ∈ {0..1} do
15:         if (y, x) ∉ halo then
16:             R[y][x] ← R[y][x] · Q
17:             D_k ← R[y][x]
18:         end if
19:     end for
20: end for
21: return(D_k)
```

data blocks adjacent to that requested in the L2 cache for (possible) future requests. Depending on whether lossy or lossless compression is selected, the operations and the data types employed in the registers are floating points or integers, respectively.

The second kernel is detailed in Algorithm 3. The wavelet transform is applied in blocks of $64 \times \widehat{Y}$ samples that are processed by a single warp.[4] This allows communication among threads without needing shared memory. The height of the block is denoted by $\widehat{Y}$. Each thread processes two columns of a block. The kernel applies a 2D high-pass/low-pass filter to all samples. First, the filter $\varphi(\cdot)$ is applied horizontally (lines 3-7) and then vertically (lines 8-12). The filter consists in a series of arithmetic operations that use the adjacent samples to the processed coefficient, in which the result is left. This type of operation does not require two register spaces (for input and output) like in the previous kernel, but only one that is referred to as $\mathcal{R}$. When the thread needs data from other threads, it uses shuffle instructions (not shown in Algorithm 3) since they have lower latency than using shared memory [32]. If more than one level of wavelet transform is selected, the instructions from line 3 to 12 are repeated each time over a quarter of the last data processed, which contains the results of the low-pass filter. This is carried out calling the kernel again. It is not detailed in Algorithm 3 for the sake of clarity. The final step in this routine is to transfer the data from the local space to the global memory. It is only done for those samples that do not belong to the halo.[5] Before transferring the data, a quantization step size, denoted by Q in line 16, may be applied. Again, lossy and lossless compression respectively requires the use of floating points and integers when applying $\varphi(\cdot)$. Quantization is only applied for lossy compression.

The BPC_AC($\cdot$) kernel is detailed in Algorithm 4. It is applied to all codeblocks of the component, though we recall that the algorithm details the operations carried out at thread level. The kernel receives a frame component that is partitioned in codeblocks of $64 \times \widehat{Y}'$ coefficients, with typically $\widehat{Y}' = 64$. The data for the codeblock are implicitly transferred to the local memory in line 2 of the algorithm. Then the coefficients are coded from bitplane $\widehat{B}$, which is a sufficient number of magnitude bits to code all coefficients within the codeblock, to the lowest bitplane 0. This is performed in the loop of line 3. Like in the previous kernel, each thread processes two columns and each codeblock is processed by a warp. Contrarily to JPEG2000, this kernel carries out 2 coding passes instead of 3 since virtually same compression efficiency is achieved [6], [44], [45] while increasing throughput about 40%. The loop in lines 4-17 performs significance coding. It checks whether the coefficient was significant in previous bitplanes via the $\gamma(\cdot)$ function, which returns the significance bitplane of the coefficient. If not, significance coding is performed. First, context C of the coefficient is determined via $\Phi(\cdot)$ and, through this context and the current bitplane, probability P for the coded bit is extracted from the lookup table LUT_{sig}. This table contains pre-computed probabilities determined with a training set of images. Then, the bit is coded via arithmetic coding. The procedure for AC($\cdot$) is not detailed in the algorithm for simplicity. It can be found in [45]. If the coefficient is significant in the current bitplane (i.e., $\gamma(\mathcal{R}[y][x]) = b$), its sign is coded in lines 11-13 with a similar procedure to that of significance coding. Refinement coding is carried out in lines 18-25. In this case, no context is employed. The return of the AC($\cdot$) function is the bitstream $\mathcal{B}_l$ that contains the compressed information. Each time that this function is called, some data may be added to $\mathcal{B}_l$. We note that $\mathcal{B}_l$ is in the global memory. Each thread puts data in $\mathcal{B}_l$ asynchronously from the others ensuring mutual exclusion. This exclusion is guaranteed considering the threads that need a new chunk of memory to write their information, assigning positions based on the thread index within the warp. This kernel also stores the length of $\mathcal{B}_l$ in a separate global memory region, denoted by $\mathcal{L}$.

The last kernel (i.e., CR($\cdot$)) is detailed in Algorithm 5. It receives the set of bitstreams $\{\mathcal{B}_l\}$. As previously stated, its purpose is to reorganize the bitstream data in a compact structure. To do so, blocks of 2 bytes are assigned to each thread in the warp to be written in the final memory positions. The first step is to generate a memory map to know these positions. This map is denoted as $\mathcal{L}'$ and contains an aggregated list of

[4]Note that these blocks are *not* the codeblocks utilized in BPC_AC($\cdot$), but a tile of the original image. Although the partitioning is similar for parallelism purposes, the block transformed by DWT_Q($\cdot$) contains overlapped samples of adjacent blocks.

[5]The halo is an area surrounding the processed samples that is employed by the warp to obtain the correct result of the wavelet transform.

Algorithm 4 Kernel routine BPC_AC($\mathcal{D}_k$)

```
1:  GPULocalMemoryAllocation()
2:  R ← D_k
3:  for b ∈ {B̂..0} do
4:      for y ∈ {1..Ŷ'} do
5:          for x ∈ {0..1} do
6:              if γ(R[y][x]) ≤ b then
7:                  C ← Φ(R[y][x])
8:                  P ← LUT_sig[C][b]
9:                  B_l ← AC(R[y][x], P)
10:                 if γ(R[y][x]) = b then
11:                     C' ← Φ'(R[y][x])
12:                     P' ← LUT_sign[C'][b]
13:                     B_l ← AC(R[y][x], P')
14:                 end if
15:             end if
16:         end for
17:     end for
18:     for y ∈ {1..Ŷ'} do
19:         for x ∈ {0..1} do
20:             if γ(R[y][x]) > b then
21:                 P'' ← LUT_ref[b]
22:                 B_l ← AC(R [y][x], P'')
23:             end if
24:         end for
25:     end for
26: end for
27: L_l ← length(B_l)
28: return(B_l)
```

Algorithm 5 Kernel routine CR($\{\mathcal{B}_l\}$)

```
1: S ← computePosition(T, LUT_L', L')
2: if (S ∈ {L'}) then
3:     M^D[o][H] ← B_l[S]
4: else
5:     M^D[o][D] ← B_l[S]
6: end if
7: return(M^D[o])
```

lengths, more precisely, $\mathcal{L}' = \{0, \mathcal{L}_1, \mathcal{L}_1 + \mathcal{L}_2, \cdots, \mathcal{L}_1 + \cdots + \mathcal{L}_{\widehat{L}}\}$. $\mathcal{L}'$ is generated via the Device Scan primitive from the Nvidia CUB framework [57]. To accelerate the access to this map, a fast lookup table, denoted by $\text{LUT}_{\mathcal{L}'}$, is created. This LUT is generated applying a binary search over $\mathcal{L}'$ in which each position represents some positions of the original map. Our experience indicates that speedups about $2\times$ are achieved by using such a strategy. These operations are carried out before running the CR($\cdot$) kernel, so they are not specified in Algorithm 5.

Once the $\text{LUT}_{\mathcal{L}'}$ is created, each warp thread T computes the position S of the data to be written (line 1). Then, it checks whether the information to be copied is auxiliary information of the codeblock (i.e., most significant bitplane), or compressed data. This is carried out in line 2 checking if the thread is copying the first bytes of the codeblock's bitstream. The corresponding bytes are either copied to the header or body section of the final structure, respectively denoted by $\mathcal{M}^D[o][H]$ and $\mathcal{M}^D[o][D]$. The data transfers are also performed in a coalesced fashion to maximize throughput.

Again, the kernels employed in the decoder are very similar to those of the encoder, so they are not detailed herein.

V. EXPERIMENTAL RESULTS

The proposed codec is evaluated with four Nvidia GPUs, namely, the RTX 2080 Ti, the GTX 1080 Ti, the Xavier, and the Tegra X2. These devices are commodity GPUs, with prices ranging from 650€ to 1350€. Their specifications are reported in Table 4. Both the RTX 2080 Ti and the GTX 1080 Ti are commonly employed in workstations for design applications and gaming. The RTX 2080 Ti has the highest peak throughput. It is employed with an i9 9900K CPU workstation with 16 GB of DDR4 RAM. The 1080 Ti is used on an i7-3770 workstation with 8 GB of DDR3 RAM. Both the Xavier and the Tegra X2 are GPUs devised for devices in which efficiency and size are important aspects, for example in the Nintendo Switch. In our tests, they run on a Jetson SDK platform [58]. Both GPUs have low performance, but consume very little power. Both allow different power modes with varying performance and Thermal Design Power (TDP). The results reported below correspond to the maximum performance mode except when indicated.

JPEG2000 results are obtained with Kakadu (v8.0.2) [59]. Kakadu is among the fastest CPU implementations of the standard. It is heavily optimized in assembler, achieving superior throughput than other implementations for GPUs such as CuJ2K [60] and GPU-J2K [61]. It is executed in a workstation with an Intel i9-9900K CPU with 8 cores and 16 GB of DDR4 RAM. Kakadu is compiled for this architecture and it is run with 16 threads of execution to achieve maximum throughput. The compression parameters for both Kakadu and our codec are: lossy or lossless compression as indicated, 5 levels of DWT, and codeblocks of 64×64. Although there are other competitive GPU implementations of JPEG2000 such as Comprimato [62] and CUDA-JPEG2000 [63], it was not possible to compare them in our test environment. Some results reported in their corresponding webpages suggest that they obtain competitive throughput, though lower to that achieved by the proposed codec.

For comparison purposes, the following experiments also provide the throughput achieved with the HEVC implementation developed by Nvidia [64], which is executed with the RTX 2080 Ti and the GTX 1080 Ti. This codec runs in the GPU employing in-chip support and dedicated hardwired components. The parameters for HEVC are: rate control with constant quantization 1-51 (0) for lossy (lossless), inter-frame coding with GOP=32, and high performance mode. This configuration achieves maximum throughput in our tests. We note that HEVC is not supported in Jetson GPUs.

	SMs	cores × SM	clock frequency	memory bandwidth	peak FP32 throughput	compute capability	TDP	memory size
RTX 2080 Ti	68	64	1601 MHz	616 GB/s	13.935 TFlops	7.5 (Turing)	260 W	11 GB
GTX 1080 Ti	28	128	1923 MHz	484 GB/s	13.78 TFlops	6.1 (Pascal)	250 W	11 GB
Xavier	8	64	854∼1377 MHz	137 GB/s	1.4 TFlops	7.2 (Volta)	10/15/30 W	16 GB*
Tegra X2	2	128	854∼1465 MHz	58.4 GB/s	0.75 TFlops	6.2 (Pascal)	7.5 - 15 W	8 GB*

TABLE 4: Features of the GPUs employed. *Both the Xavier and Tegra X2 do not have dedicated GPU memory. Memory is shared by both the CPU and GPU.

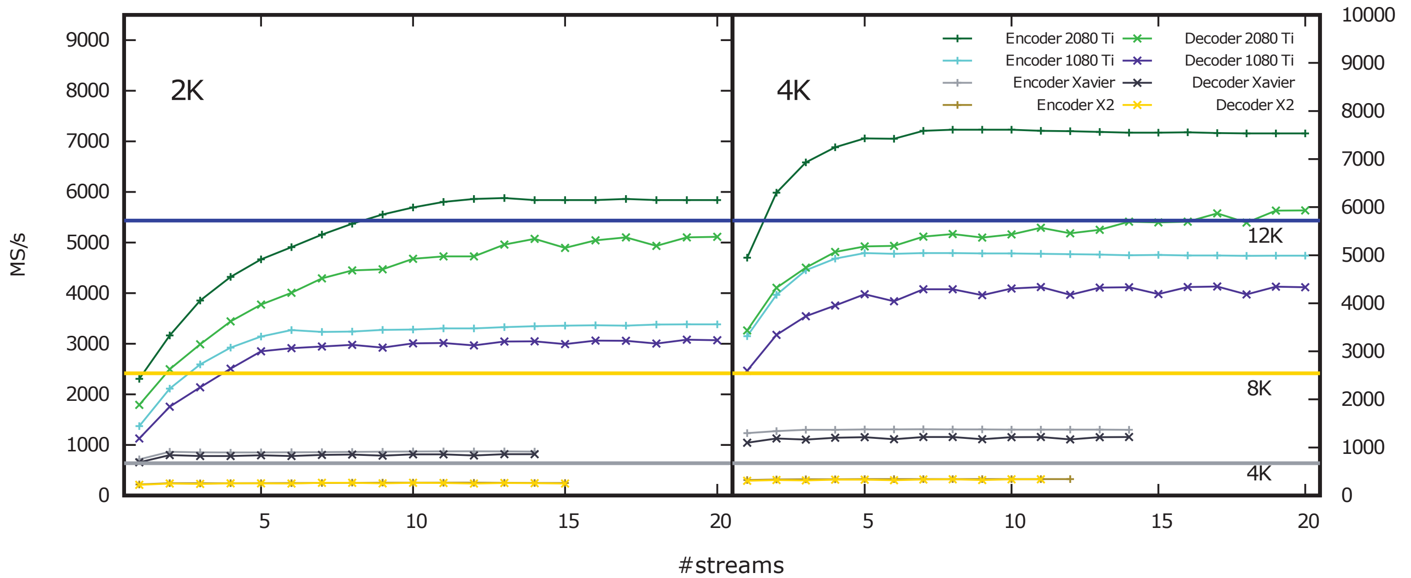


FIGURE 4: Analysis of the throughput achieved by the proposed codec when coding 2K (left) and 4K (right) video using different number of execution streams, for lossy compression at maximum quality.

The data set employed in the experiments is a 2-minute segment of the movie “Star Wars: The Last Jedi,” at a resolution of 2K and 4K. The video contains 2,880 color frames with a bit-depth resolution of 24 bits per pixel (i.e., 8 bits per pixel per component), resulting in 67,5 GB (16,875 GB) of uncompressed data for the 4K (2K) resolution. The HEVC codec uses a subsampled 4:2:0 version of the video for compatibility issues with the 4K resolution in the GTX 1080 Ti. This is taken in consideration when measuring the performance achieved. In general, the size of this data set is sufficiently large to fill the resources of the GPU. Larger data sets achieve similar results as those reported below. In all results, the execution time is measured without considering the I/O time spent to read/write the files from/to the disk since that would affect results significantly depending on the hard drive employed. The results below evaluate only the throughput achieved since coding performance of the proposed codec is extensively analyzed in [44]. Herein, the codecs are compared when their coding options yield equivalent image quality.

The first test evaluates the throughput achieved by the proposed codec with the four GPUs when using a different number of execution streams. The test evaluates both the encoder and decoder in lossy mode with a quantization step size that achieves maximum quality (about 50 dB). Figure 4 reports the results achieved. Again, this figure depicts with horizontal lines the throughput needed to yield 4K, 8K, and 12K video compression in real time, assuming a frame rate of 24 frames per second. The results indicate that both the RTX 2080 Ti and GTX 1080 Ti increase the throughput as more streams are employed, yielding optimal performance depending on the frame resolution and GPU employed. The Xavier and Tegra X2 do not benefit as much of using multiple streams because they have fewer SMs, so their resources are mostly filled with a single execution stream. In all results, the decoder yields slightly lower throughput than the encoder because it requires more local memory. This behavior is not common in software implementations of image and video codecs since the encoder generally requires more computations. Highly optimized implementations such as the presented herein, however, may obtain different results due to the need of different data structures in the decoder. In the following tests, 20 and 9 streams are employed for the RTX 2080 Ti and GTX 1080 Ti, respectively, to achieve maximum throughput. The Xavier and Tegra X2 employ 14 and 10 streams, respectively, though their throughput is almost the same as when using only 2.

The next test evaluates the number of kernels that are

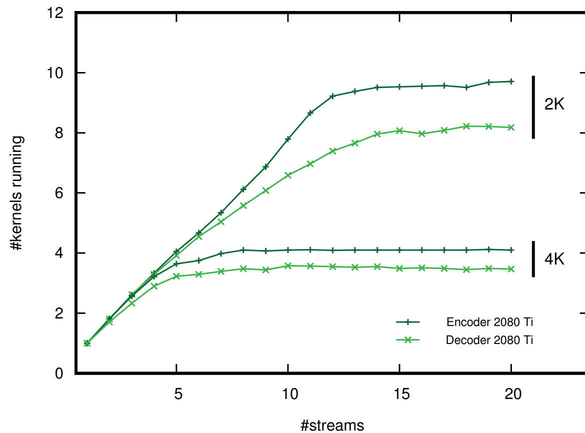


FIGURE 5: Evaluation of the average number kernels executed per unit of time depending on the number of streams employed.

executed in parallel depending on the number of streams employed. This analysis complements the previous for the RTX 2080 Ti. The GTX 1080 Ti, Xavier, and Tegra X2 are not included in this analysis. Figure 5 depicts the results achieved. For 4K video, the maximum number of running kernels is 4, which is yield when employing 10 streams. 4 parallel kernels already fill the resources of the GPU. This indicates that no more kernels can be executed despite increasing the number of streams employed, although a slight increase in throughput can be achieved as it seen in the previous figure. 2K video obtains a different behavior. Number of streams and running kernels are almost directly related, reaching a peak at 20 streams and 10 parallel kernels. This is because 2K frames have only a quarter of the data of 4K frames, so the GPU requires more kernels to fill its resources.

Figure 6 reports the throughput achieved by the proposed codec with the four GPUs, Kakadu, and HEVC when coding 4K video in lossy and lossless mode. For lossy compression, the average image quality yield for all codecs is about 50 dB. At this level of quality, distortion is not perceptible by the human eye. Each codec has a pair of columns. The first reports the results for the encoder whereas the second for the decoder. The results for 2K video are similar but with lower performance, so they are not included in this figure. Results for the Xavier and Tegra X2 are reported when using three power modes, namely, maximum (0), minimum (1), and mid-tier (2) performance. The results show that the proposed codec yields superior performance to that achieved by Kakadu and HEVC for both the RTX 2080 Ti and GTX 1080 Ti regardless of using lossy or lossless compression. In all codecs, the performance in lossless mode is slightly lower than that achieved in lossy since more data are processed, generating larger compressed files. Even so, real-time 12K video can be managed by our codec for both compression modes. The Xavier and Tegra X2 GPUs do not achieve such a high performance, but the Xavier is able to process 4K video in real time when employing the maximum performance mode. This throughput is similar to that obtained by Kakadu, though we recall that Kakadu employs a modern CPU and the Xavier is an embedded mobile solution. Both for the Xavier and the Tegra X2, the minimum power mode significantly lowers performance and the mid-tier mode achieves an intermediate performance. This is more pronounced in the Xavier. HEVC yields higher performance than Kakadu, though it is lower than that achieved by our codec. Surprisingly, the HEVC encoder achieves higher throughput with the GTX 1080 Ti than with the RTX 2080 Ti. Even though it is executed using the Nvidia SDK HEVC software (v9.0) [64] in maximum performance mode in both, each GPU has its own hardwired solution for this codec. More precisely, the RTX 2080 Ti includes one NVEnc Turing engine whereas the 1080 Ti includes two Pascal engines. Note also that the GTX 1080 Ti obtains higher throughput for the encoder than for the decoder, whereas the RTX 2080 Ti yields more balanced results.

The previous test evaluates the performance achieved when there is (almost) no quality loss. Scenarios such as video streaming or TV broadcast may tolerate more distortion. Reducing the image quality results in higher throughput since fewer data are coded. Figure 7 depicts the throughput achieved by Kakadu, HEVC, and the proposed codec when coding 4K video at different levels of quality, namely, from 50 dB to 20 dB, which is the quality range employed in most scenarios. The image quality is controlled via the quantization parameter Q in our codec, and similarly in HEVC and Kakadu. As seen in the figure, reducing the quality has a direct impact on throughput for all codecs. The proposed codec achieves real-time encoding of 16K video for qualities below 46 dB. The decoder has a lower increase in performance as the quality decreases because the aforementioned need of more local memory. The Xavier and Tegra X2 also increase their throughput, though more gradually due to their inferior performance power. It is worth noting that, even though the RTX 2080 Ti and GTX 1080 Ti have a similar peak throughput (about 14 TFlops), the RTX 2080 Ti obtains approximately 50% more throughput when encoding. This is due to the distribution of performance power in the GPU. The RTX 2080 Ti has fewer CUDA cores in each SM, but more than twice SMs than the GTX 1080 Ti. This provides more resources per thread, especially, more local memory. Our codec greatly benefits from this architectural improvement since it employs registers extensively. The highest speedups reported in Figure 6 are achieved by the HEVC decoder, which increases the throughput almost $6\times$.

Power consumption is nowadays an important aspect due to the advent of mobile devices. Figure 8 evaluates the power consumption of our codec, HEVC, and Kakadu when coding 4K video at 50 dB, like in Figure 6. The results are depicted in MS processed per Watts consumed. A Nvidia tool that measures consumption in real-time is employed to obtain these results. Kakadu's consumption is measured via the utility PowerTOP. The results depicted in Figure 8 suggest that the proposed codec is the most efficient in terms of power

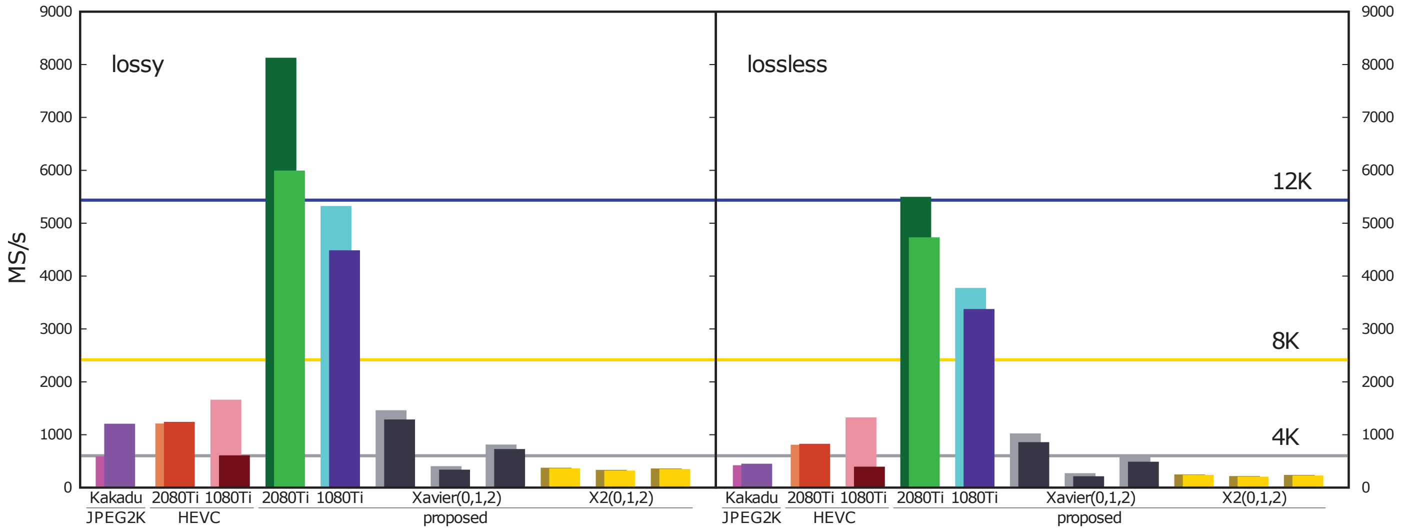


FIGURE 6: Throughput evaluation for lossy (with highest image quality) and lossless compression of 4K video, for all codecs and GPUs. Each pair of columns reports the results for the encoder (back) and decoder (front).

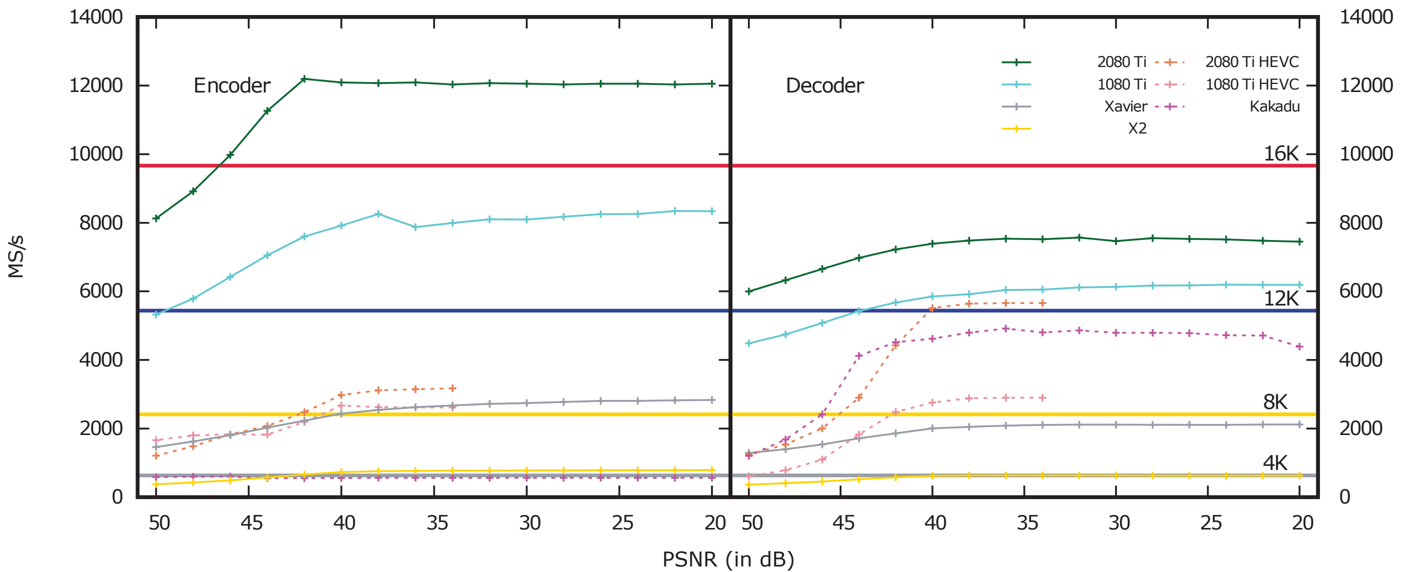


FIGURE 7: Throughput evaluation for lossy compression of 4K video at different quality levels. Results are for the proposed codec except when indicated.

consumption. Evidently, the Xavier and Tegra X2 yield the best results due to its architecture. Our codec employed with the three power modes of these GPUs is less power-hungry than the remaining, with the minimum mode achieving the highest efficiency. The proposed codec is more efficient than HEVC even when executed in the RTX 2080 Ti and GTX 1080 Ti, though moderately so. In general, CPUs consume more power than GPUs, so Kakadu seems to consume the most. The low power consumption of our codec means that, in practice, it can allow batteries of mobile devices last much longer and/or code more minutes of video for the same battery capacity.

VI. CONCLUSIONS

Faster and less power-hungry image and video codecs are currently needed in multiple scenarios. Typically, high throughput codecs are achieved by means of integrated hardware architectures such as ASICs or FPGAs. GPUs are also a widely pursued means to accelerate codecs, though these architectures do not commonly obtain the high performance of their counterparts. This is because the core algorithms of conventional image and video coding systems do not provide enough fine-grain parallelism to fully exploit the SIMD architecture of GPUs. This paper introduces an image/video codec based on the JPEG2000 standard. All stages of the coding pipeline have been devised to extract fine-grain parallelism. All stages are compliant with the standard except for the core algorithm called bitplane and arithmetic coding.

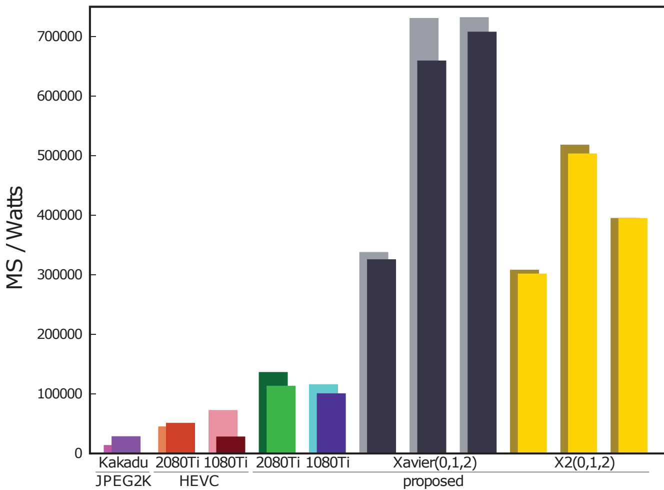


FIGURE 8: Power consumption evaluation when encoding 4K video at 50 dB. Each pair of columns reports the results for the encoder (back) and decoder (front).

The proposed codec introduces a similar algorithm to that of JPEG2000 that augments its parallel capabilities. Although the resulting codestream is not compliant with JPEG2000, the coding system has the same advanced features of the standard. The throughput of the resulting architecture when executed in consumer-grade GPUs is at least 10× higher than that achieved with CPU implementations executed in high-end workstations, and superior to that achieved by Nvidia's SDK implementation of the HEVC video standard. Experimental results suggest that our codec can encode (decode) real-time 12K (8K) video in a Nvidia RTX 2080 Ti and that it consumes very little power, especially in mobile GPUs.

REFERENCES

[1] Digital compression and coding for continuous-tone still images, ISO/IEC Std. 10 918-1, 1992.
[2] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," IEEE Trans. Comput., vol. C-23, no. 1, pp. 90–93, Jan. 1974.
[3] D. Huffman, "A method for the construction of minimum redundancy codes," Proc. IRE, vol. 40, pp. 1098–1101, 1952.
[4] Information technology - JPEG 2000 image coding system - Part 1: Core coding system, ISO/IEC Std. 15 444-1, Dec. 2000.
[5] I. Daubechies, Ten lectures on wavelets. Philadelphia, PA: SIAM, 1992.
[6] F. Auli-Llinas and M. W. Marcellin, "Scanning order strategies for bitplane image coding," IEEE Trans. Image Process., vol. 21, no. 4, pp. 1920–1933, Apr. 2012.
[7] High Efficiency Video Coding Standard, International Telecommunication Union Std. H.265, 2013.
[8] D. S. Taubman and M. W. Marcellin, JPEG2000 Image compression fundamentals, standards and practice. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.
[9] A. Descampe, F.-O. Devaux, G. Rouvroy, J.-D. Legat, J.-J. Quisquater, and B. Macq, "A flexible hardware JPEG 2000 decoder for digital cinema," IEEE Trans. Circuits Syst. Video Technol., vol. 16, no. 11, pp. 1397–1410, Nov. 2006.
[10] T. Bruylants, A. Munteanu, and P. Schelkens, "Wavelet based volumetric medical image compression," ELSEVIER Signal Processing: Image Communication, vol. 31, no. C, pp. 112–133, Feb. 2015.
[11] B. Penna, T. Tillo, E. Magli, and G. Olmo, "Transform coding techniques for lossy hyperspectral data compression," IEEE Trans. Geosci. Remote Sens., vol. 45, no. 5, pp. 1408–1421, May 2007.
[12] Q. Huang, R. Zhou, and Z. Hong, "Low memory and low complexity VLSI implementation of JPEG2000 codec," IEEE Trans. Consum. Electron., vol. 50, no. 2, pp. 638–646, May 2004.
[13] H.-C. Fang, Y.-W. Chang, T.-C. Wang, C.-J. Lian, and L.-G. Chen, "Parallel embedded block coding architecture for JPEG 2000," IEEE Trans. Circuits Syst. Video Technol., vol. 15, no. 9, pp. 1086–1097, Sep. 2005.
[14] G. Pastuszak, "A high-performance architecture for embedded block coding in JPEG 2000," IEEE Trans. Circuits Syst. Video Technol., vol. 15, no. 9, pp. 1182–1191, Sep. 2005.
[15] A. K. Gupta, S. Nooshabadi, D. Taubman, and M. Dyer, "Realizing low-cost high-throughput general-purpose block encoder for JPEG2000," IEEE Trans. Circuits Syst. Video Technol., vol. 16, no. 7, pp. 843–858, Jul. 2006.
[16] Y. Li and M. Bayoumi, "A three-level parallel high-speed low-power architecture for EBCOT of JPEG 2000," IEEE Trans. Circuits Syst. Video Technol., vol. 16, no. 9, pp. 1153–1163, Sep. 2006.
[17] Y.-W. Chang, C.-C. Cheng, C.-C. Chen, H.-C. Fang, and L.-G. Chen, "124 MSamples/s pixel-pipelined Motion-JPEG 2000 codec without tile memory," IEEE Trans. Circuits Syst. Video Technol., vol. 17, no. 4, pp. 398–406, Apr. 2007.
[18] K. Mei, N. Zheng, C. Huang, Y. Liu, and Q. Zeng, "VLSI design of a high-speed and area-efficient JPEG2000 encoder," IEEE Trans. Circuits Syst. Video Technol., vol. 17, no. 8, pp. 1065–1078, Aug. 2007.
[19] Y.-W. Chang, H.-C. Fang, C.-C. Chen, C.-J. Lian, and L.-G. Chen, "Word-level parallel architecture of JPEG 2000 embedded block coding decoder," IEEE Trans. Multimedia, vol. 9, no. 6, pp. 1103–1112, Oct. 2007.
[20] M. Dyer, S. Nooshabadi, and D. Taubman, "Design and analysis of system on a chip encoder for JPEG2000," IEEE Trans. Circuits Syst. Video Technol., vol. 19, no. 2, pp. 215–225, Feb. 2009.
[21] K. Sarawadekar and S. Banerjee, "An efficient pass-parallel architecture for embedded block coder in JPEG 2000," IEEE Trans. Circuits Syst. Video Technol., vol. 21, no. 6, pp. 825–836, Jun. 2011.
[22] K. Liu, E. Belyaev, and Y. Li, "A high throughput JPEG2000 entropy decoding unit architecture," SPRINGER Journal of Digital Imaging, 2019, in Press.
[23] Nvidia. (2018, Dec.) GPU vs CPU theoretical GFLOP/s. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/floating-point-operations-per-second.png
[24] M. Hopf and T. Ertl, "Hardware accelerated wavelet transformations," in Proc. Eurographics and IEEE Symposium on Visualization, May 2000, pp. 93–103.
[25] T.-T. Wong, C.-S. Leung, P.-A. Heng, and J. Wang, "Discrete wavelet transform on consumer-level graphics hardware," IEEE Trans. Multimedia, vol. 9, no. 3, pp. 668–673, Apr. 2007.
[26] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, 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, Mar. 2008.
[27] J. Matela, "GPU-Based DWT acceleration for JPEG2000," in In Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, Nov. 2009, pp. 136–143.
[28] J. Franco, G. Bernabe, J. Fernandez, and M. E. Acacio, "A parallel implementation of the 2D wavelet transform using CUDA," in Proc. IEEE International Conference on Parallel, Distributed and Network-based Processing, Feb. 2009, pp. 111–118.
[29] W. J. van der Laan, A. C. Jalba, and J. B. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," IEEE Trans. Parallel Distrib. Syst., vol. 22, no. 1, pp. 132–146, Jan. 2011.
[30] V. Galiano, O. Lopez, M. P. Malumbres, and H. Migallon, "Speeding-up the discrete wavelet transform computation with multicore and GPU-based algorithms," in Proc. International Conference on Computational and Mathematical Methods in Science and Engineering, Jan. 2012, pp. 151–158.
[31] ——, "Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs," SPRINGER The Journal of Supercomputing, vol. 64, no. 1, pp. 4–16, Mar. 2012.
[32] 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, Dec. 2015.
[33] T. M. Quan and W.-K. Jeong, "A fast discrete wavelet transform using hybrid parallelism on GPUs," IEEE Trans. Parallel Distrib. Syst., vol. 27, no. 11, pp. 3088–3100, Nov. 2016.
[34] S. Datla and N. S. Gidijala, "Parallelizing motion JPEG 2000 with CUDA," in Proc. IEEE International Conference on Computer and Electrical Engineering, Dec. 2009, pp. 630–634.

[35] R. Le, I. R. Bahar, and J. L. Mundy, "A novel parallel tier-1 coder for JPEG2000 using GPUs," in Proc. IEEE Symposium on Application Specific Processors, Jun. 2011, pp. 129–136.
[36] J. Matela, V. Rusnak, and P. Holub, "Efficient JPEG2000 EBCOT context modeling for massively parallel architectures," in Proc. IEEE Data Compression Conference, Mar. 2011, pp. 423–432.
[37] F. Wei, Q. Cui, and Y. Li, "Fine-granular parallel EBCOT and optimization with CUDA for digital cinema image compression," in Proc. IEEE International Conference on Multimedia and Expo, Jul. 2012, pp. 1051–1054.
[38] M. Ciznicki, K. Kurowski, and A. Plaza, "Graphics processing unit implementation of JPEG2000 for hyperspectral image compression," SPIE Journal of Applied Remote Sensing, vol. 6, pp. 1–14, Jan. 2012.
[39] J. Lee, B. Kim, and K. Yoon, "CUDA-based JPEG2000 encoding scheme," in Proc. IEEE International Conference on Advanced Communication Technology, Feb. 2014, pp. 671–674.
[40] X. Wu, Y. Li, K. Liu, K. Wang, and L. Wang, "Massive parallel implementation of JPEG2000 decoding algorithm with multi-GPUs," in Proc. SPIE Satellite Data Compression, Communications, and Processing X, vol. 9124, May 2014, pp. 1–6.
[41] 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, Jun. 2014.
[42] F. Auli-Llinas, "Context-adaptive binary arithmetic coding with fixed-length codewords," IEEE Trans. Multimedia, vol. 17, no. 8, pp. 1385–1390, Aug. 2015.
[43] F. Auli-Llinas, P. Enfedaque, J. C. Moure, I. Blanes, and V. Sanchez, "Strategy of microscopic parallelism for bitplane image coding," in Proc. IEEE Data Compression Conference, Apr. 2015, pp. 163–172.
[44] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, "Bitplane image coding with parallel coefficient processing," IEEE Trans. Image Process., vol. 25, no. 1, pp. 209–219, Jan. 2016.
[45] 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, Aug. 2017.
[46] D. Taubman, A. Naman, and R. Mathew, "High throughput block coding in the HTJ2K compression standard," in Proc. IEEE International Conference on Image Processing, Sep. 2019, pp. 1079–1083.
[47] A. Naman and D. Taubman, "Decoding high-throughput JPEG2000 (HTJ2K) on a GPU," in Proc. IEEE International Conference on Image Processing, Sep. 2019, pp. 1084–1088.
[48] C. de Cea-Dominguez, P. Enfedaque, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "High throughput image codec for high-resolution satellite images," in Proc. IEEE International Geoscience and Remote Sensing Symposium, Jul. 2018, pp. 6524–6527.
[49] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "GPU architecture for wavelet-based video coding acceleration," in Parallel Computing: Technology Trends, vol. 36, Apr. 2020, pp. 83–92, IOSPress Series in Advances in Parallel Computing.
[50] Nvidia, "Warp level primitives," Tech. Rep., Jan. 2018. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/
[51] ——, "Nvidia Tesla V100 GPU architecture," Tech. Rep., Jun. 2019. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[52] F. N. Iandola, D. Sheffield, M. Anderson, P. M. Phothilimthana, and K. Keutzer, "Communication-minimizing 2d convolution in gpu registers," in Proc. IEEE International Conference on Image Processing, Sep. 2013, pp. 2116–2120.
[53] A. Chacon, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Thread-cooperative, bit-parallel computation of Levenshtein distance on GPU," in Proc. ACM International Conference on Supercomputing, Jun. 2014, pp. 103–112.
[54] ——, "FM-index on GPU: a cooperative scheme to reduce memory footprint," in Proc. IEEE International Symposium on Parallel and Distributed Processing with Applications, Aug. 2014, pp. 1–9.
[55] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," SIAM Journal on Mathematical Analysis, vol. 29, no. 2, pp. 511–546, 1998.
[56] F. Auli-Llinas and J. Serra-Sagrista, "JPEG2000 quality scalability without quality layers," IEEE Trans. Circuits Syst. Video Technol., vol. 18, no. 7, pp. 923–936, Jul. 2008.
[57] Nvidia. (2018, Dec.) CUB framework. [Online]. Available: https://nvlabs.github.io/cub/
[58] ——, "Jetson SDK," Tech. Rep., May 2019. [Online]. Available: https://developer.nvidia.com/embedded-computing
[59] D. Taubman. (2020, Feb.) Kakadu software. [Online]. Available: http://www.kakadusoftware.com
[60] University of Stuttgart. (2020, Jan.) CuJ2K. [Online]. Available: http://cuj2k.sourceforge.net/
[61] Poznan Supercomputing and Networking Center. (2020, Feb.) GPUJ2K. [Online]. Available: http://apps.man.poznan.pl/trac/jpeg2k/wiki
[62] Comprimato. (2020, Feb.) Comprimato JPEG2000@GPU. [Online]. Available: http://www.comprimato.com
[63] Fastvideo LLC. (2020, Feb.) CUDA-JPEG2K. [Online]. Available: https://www.fastcompression.com/products/gpu-jpeg2000.htm
[64] Nvidia. (2018, Dec.) HEVC SDK. [Online]. Available: https://developer.nvidia.com/nvidia-video-codec-sdk

Chapter 5

Complexity Scalable Bitplane Image Coding with Parallel Coefficient Processing

Complexity Scalable Bitplane Image Coding with Parallel Coefficient Processing

Carlos de Cea-Dominguez, Juan C. Moure, Joan Bartrina-Rapesta, and Francesc Aulí-Llinàs

***Abstract*—Very fast image and video codecs are a pursued goal both in the academia and the industry. This paper presents a complexity scalable and parallel bitplane coding engine for wavelet-based image codecs. The proposed method processes the coefficients in parallel, suiting hardware architectures based on vector instructions. Our previous work is extended with a mechanism that provides complexity scalability to the system. Such a feature allows the coder to regulate the throughput achieved at the expense of slightly penalizing compression efficiency. Experimental results suggests that, when using the fastest speed, the method almost doubles the throughput of our previous engine while penalizing compression efficiency by about 10%.**

***Index Terms*—High-throughput image coding, JPEG2000.**

I. Introduction

THE pursuit of faster image and video coding systems began shortly after the development of the first codecs and compression standards. Traditional image coding systems, such as SPIHT [1] or EBCOT [2], have been revisited many times introducing modifications that accelerate their coding process and/or alleviate computational resources [3]–[8]. Also, many hardware architectures of such systems are optimized to reduce execution time and meet the real-time requirements of some environments [9]–[12]. These works focus on the improvement, or efficient implementation, of the most demanding tasks of the codec, without modifying the techniques of the original system. In general, such techniques code the data via a single-thread procedure. This strategy together with the soaring of the processor's clock speed for more than three decades, enhanced the codecs' throughput significantly. Since 2005 the increase in the clock's speed slowed and processors began augmenting their processing power via parallel architectures.

The transition from single- to multi-thread algorithms in the image coding field began with the advent of multi-core Central Processing Units (CPUs) in the 2000s [13]–[16]. The first multi-thread codecs partitioned the image in multiple pieces (referred to as codeblocks onward) that can be processed independently. In international standards such as JPEG2000 (ISO/IEC 15444) or HEVC (ISO/IEC 23008-2), for instance, the coding system provides multiple opportunities for such coarse-grain parallelism. However, the core algorithms do not allow fine-grain parallelism since they are envisaged from a single-thread perspective, resulting in a causal relationship among samples that makes parallel processing difficult.

This drawback was inconsequential while highly parallel computing was not widely available. This changed in the last years, when CPUs began including vector instructions to exploit fine-grain parallelism and, more importantly, when parallel architectures and platforms like CUDA were introduced allowing massive parallelism in commodity Graphics Processing Units (GPUs). Algorithms of other fields that were well suited to fine-grain parallelism were rapidly adapted in GPUs, achieving 20× speedups or more [17]. When implemented in GPUs, image and video coding systems did not achieve such speedups due to the sequential techniques employed.

Aware of this fact, the Joint Photographics Experts Group launched a call for proposals in 2017 [18] to introduce a new part to the JPEG2000 standard that defines a new tier-2 coding variant that offers high throughput [19]. This part is called HTJ2K (ISO/IEC 15444-15). It is devised to benefit from the modern instruction sets like AVX2, NEON, and BMI2 included in new CPUs, and also from the GPU's highly parallel architecture [20]. It is about 10× faster than the standard when executed in a CPU, though it penalizes coding performance in approximately 10%. Also, it sacrifices quality scalability, which is a valued feature of the standard that allows transmitting the image progressively by quality.

In a similar line, in 2014 we started a research whose goal is a JPEG2000-like codec that provides opportunities for fine-grain parallelism in all the stages of the coding process [21]–[25]. The proposed codec was recently evaluated using a commodity GPU. Experimental results suggest that it achieves 10× speedups compared to an implementation of JPEG2000 that is executed in a workstation with 4 CPUs [26]. The adaptation of the bitplane coding engine was the most demanding task since it requires the modification of the original techniques of JPEG2000, losing compliance. The proposed bitplane coding engine with parallel coefficient processing (BPC-PaCo) uses vector instructions of 32 lanes (or, equivalently, 32 CUDA threads) to process 32 coefficients within a codeblock in parallel. BPC-PaCo sacrifices coding efficiency as compared to JPEG2000 by about 2% but maintains all its features.

This paper introduces a mechanism that provides a new feature to BPC-PaCo: complexity scalability. The proposed mechanism allows trading computational complexity for compression efficiency. The underlying motivation is that some environments may be willing to sacrifice coding performance in exchange of throughput. Complexity scalable BPC-PaCo (CS BPC-PaCo) allows tuning the codec to accelerate more or less the coding process. Evidently, the higher the throughput achieved, the more affected are the compression efficiency and quality scalability of the system. Experimental results indicates

Carlos de Cea-Dominguez, Joan Bartrina-Rapesta and Francesc Aulí-Llinàs are with the Dep. of Information and Communications Engineering and Juan C. Moure is with the Dep. of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain (phone: +34 935811861; fax: +34 935813443; e-mail: carlos.decea@uab.cat). This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under Grants TIN2017-84553-C2-1-R and RTI2018-095287-B-I00 (MINECO/FEDER, UE), and by the Catalan Government under Grants 2017SGR-463 and 2017SGR-313. **Copyright (c) 2020 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.**

that speedups of almost 2× are achieved compared to BPC-PaCo while penalizing performance by about 10%.

The rest of the paper is structured as follows. Section II reviews BPC-PaCo and Section III describes the proposed complexity scalable mechanism. Experimental results are presented in Section IV. The last section summarizes this work.

II. Review of BPC-PaCo

BPC-PaCo utilizes a traditional bitplane coding strategy that codes the wavelet coefficients from the most significant bitplane $M-1$ to the least, with M being a sufficient number of bits to represent all coefficients within a codeblock. A bitplane is the collection of bits b_j from all coefficients, with $[b_{M-1}, b_{M-2}, ..., b_1, b_0]$, $b_i \in \{0,1\}$ denoting the binary representation of an integer υ that represents the magnitude of the index obtained by quantizing wavelet coefficient ω. The first non-zero bit of the binary representation of υ is denoted by b_s and is referred to as the significant bit. The sign of the coefficient is denoted by $d \in \{+,-\}$ and is coded immediately after b_s, so that the decoder can begin approximating ω as soon as possible. The bits $b_r,\ r < s$ are referred to as refinement bits. Although two or three coding passes may be employed [24], [25], the two coding pass version is employed herein as baseline since it achieves higher throughput. The first is called significance coding. It processes the bits of non-significant coefficients, i.e., those coefficients whose $s \leq j$ or, more precisely, whose significance state $\Phi(\upsilon, j) = 0$. The second pass is called refinement coding and processes the bits of the remaining coefficients (i.e., those whose $\Phi(\upsilon, j) = 1$).

The main difference between BPC-PaCo and other bitplane coding engines is that BPC-PaCo codes multiple coefficients in parallel. The scanning order is organized in stripes of two columns. The stripes are processed by threads that advance their execution synchronously, all coding the coefficient in the same position of their corresponding stripe. This is the key to achieve fine-grain parallelism, since a single vector instruction is executed to code T coefficients of the codeblock at the same clock cycle. In general, the codeblock contains 64×64 coefficients, so $T = 32$. Evidently, this strategy must be accompanied with parallel techniques for context formation, probability estimation, and entropy coding.

For significance coding, the context of υ at bitplane j is determined considering its eight adjacent neighbors, denoted by υ^k, via $\phi_{sig}(\upsilon, j) = \sum_k \Phi(\upsilon^k, j)$. The context for sign coding, denoted by $\phi_{sign}(\omega, j)$, employs a similar strategy, whereas the refinement pass employs a single context since little gain is achieved with more complex models [27], so $\phi_{ref}(\upsilon, j) = 0$. Through the context, the probability estimate of the encoded bit is extracted from a lookup table (LUT) known by encoder and decoder [21]. The LUT for significance coding is accessed as $\mathcal{P}_u[j][\phi_{sig}(\cdot)]$, with u denoting the wavelet subband. This LUT contains the probability that b_j is 0, which is determined according to

$$P_{sig}(b_j = 0 \mid \phi_{sig}(\upsilon, j)) = \frac{\sum_{\upsilon=0}^{2^j-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j))}{\sum_{\upsilon=0}^{2^{j+1}-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j))}, \quad (1)$$

where $F_u(\upsilon \mid \phi_{sig}(\upsilon, j))$ is the probability mass function (pmf) of the quantization indices at bitplane j given their context. Its support is $[0, ..., 2^{j+1}-1]$ since it contains quantization indices that were not significant in bitplanes greater than j. Probabilities for sign and refinement coding are derived similarly. Their respective LUTs are denoted by $\mathcal{P}'_u$ and $\mathcal{P}''_u$.

Entropy coding is carried out through multiple arithmetic coders that produce fixed-length codewords [22] as data are coded. Each thread employs one such a coder. The dispatching of the codewords in the quality embedded bitstream generated for the codeblock requires cooperation among threads. It is optimally constructed so that the bitstream can be truncated at the end of coding passes yielding minimum distortion (see Section III.C and III.D in [24]).

III. Complexity scalable BPC-PaCo

A distinct feature of bitplane coding engines, including BPC-PaCo, is that they code the coefficients in multiple passes per bitplane. This strategy is aimed to code first those data that mostly decrease the image distortion. At the decoder, the wavelet coefficients are progressively reconstructed, allowing a fine refinement of the estimates of the incoming data. These estimates are key to achieve compression. They are commonly embodied in the context formation and probability model. Another advantage of using multiple passes per bitplane is that the bitstream contains multiple truncation points, one at the end of each coding pass. They are key to achieve quality scalability since they are employed by the rate-distortion optimization method to minimize the distortion at a target rate(s). Unfortunately, more coding passes entail more computational complexity. Each pass scans all coefficients of the codeblock despite coding the bits for only some of them. This is repeated in each coding pass, so a coefficient is accessed as many times as coding passes are executed.

The main idea behind Complexity Scalable BPC-PaCo (CS BPC-PaCo) is to reduce the computational complexity of the coding engine by reducing the number of times that each coefficient is visited. To do so while minimizing the impact on compression efficiency and quality scalability, bitplanes $[M-1, N]$ are coded as defined in BPC-PaCo. From bitplane $N-1$ to the lowest, each coefficient is coded with a fast mode that uses a single pass. Differently from conventional bitplane coding strategies, this single pass carries out inter-bitplane coding since it transmits the information of multiple bitplanes at once. Through N, the granularity of the quality scalability, the compression efficiency, and the computational complexity of the algorithm are controlled. When N is low, more bitplanes are coded with two coding passes, producing many truncation points that can be employed by rate-distortion optimization procedures. Also, coefficients are reconstructed progressively, allowing fine estimates. Evidently, low Ns do not reduce computational complexity significantly. When N is high, more bitplanes are coded in fast mode, reducing computational complexity though producing fewer truncation points and penalizing compression efficiency due to rougher estimates. This mechanism provides complexity scalability to the codec, since it can be employed to favor the application's throughput or the compression efficiency/quality scalability.

The same coding techniques of [24], with the modifications described below, are valid in the fast mode to remove the data dependency when coding coefficients in parallel.

Algorithm 1 describes the proposed coding engine from a thread (or a single lane of a vector instruction) perspective. From line 1 to 14, it employs the same procedure as that of BPC-PaCo (see Section III.D in [24]). The only difference is that the loop in line 1 codes bitplanes $[M-1,N]$ instead of $[M-1,0]$. The position of the coefficient within the codeblock is denoted by y and x for the row and column, respectively. The fast mode is embodied in lines 15 to 29. It encodes bits $[N-1,0]$ at once for each coefficient. The significance context is computed in line 17 before start coding and it is employed until b_s is found. This context does not change from bitplane $N-1$ to 0 since no more information of the adjacent neighbors is available once the fast mode begins. This also needs to be considered in the probability model, so the LUT employed in the fast mode for significance coding in bitplanes $j' = [N-1,0]$ is populated according to

$$P_{sig}(b_{j'} = 0 \mid \phi_{sig}(\upsilon, N-1)) = \frac{\sum_{\upsilon=0}^{2^{j'}-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, N-1))}{\sum_{\upsilon=0}^{2^{j'+1}-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, N-1))} \tag{2}$$

Probabilities for sign coding are determined accordingly. The LUT for refinement is unchanged due to the use of a single context.

The selection of N is key to control the computational complexity of the engine. A straightforward approach is to apply the same N to all codeblocks. Our experience indicates that this may penalize quality scalability significantly because at the lowest N bitplanes there is only one truncation point available for each codeblock. When the bitstreams segments of higher bitplanes are already selected, the rate-distortion optimization method can only include the whole segment of some codeblocks, completely discarding some others. At low rates, this may cause that none information is transmitted for some areas of the image, producing an image with blank areas or with no color information. The quality scalability of the system is less affected when N is chosen depending on the codeblock's data and the wavelet subband. Let us explain further. As indicated in [28], the highest bitplanes of a codeblock contain the information that mostly decreases the distortion. In terms of rate-distortion optimization, this means that is more valuable the data coded in bitplane $j = 4$ for a codeblock with $M = 5$ than for another with $M = 6$, for example. Therefore, our strategy selects N depending on M. The wavelet subband is also considered. The first decomposition levels (i.e., the largest wavelet subbands) contain most codeblocks, whereas the latest contain much fewer, so the use of the fast mode in the codeblocks of the smallest resolution subbands barely affects the throughput achieved. However, these codeblocks contain the rougher details of the image, important for its reconstruction. Our strategy selects N in each codeblock according to

Algorithm 1 Complexity Scalable BPC-PaCo (encoder)
Parameters: u subband, t stripe, M bitplanes to code, N bitplanes in fast mode

1: **for** $j \in [M-1, N]$ **do**
2: **for** $y \in [0, \text{numRows}-1]$ **do**
3: **for** $x \in [t \cdot 2, t \cdot 2 + 1]$ **do**
4: **if** $\Phi(\upsilon_{y,x}, j+1) = 0$ **then**
5: ACencode(b_j, $\mathcal{P}_u[j][\phi_{sig}(\upsilon_{y,x}, j)]$, t)
6: **if** $b_j = 1$ **then**
7: ACencode(d, $\mathcal{P}'_u[j][\phi_{sign}(\omega_{y,x}, j)]$, t)
8: **end if**
9: **else**
10: ACencode(b_j, $\mathcal{P}''_u[j][0]$, t)
11: **end if**
12: **end for**
13: **end for**
14: **end for**
15: **for** $y \in [0, \text{numRows}-1]$ **do**
16: **for** $x \in [t \cdot 2, t \cdot 2 + 1]$ **do**
17: $c \leftarrow \phi_{sig}(\upsilon_{y,x}, N-1)$
18: **for** $j \in [N-1, 0]$ **do**
19: **if** $\Phi(\upsilon_{y,x}, j+1) = 0$ **then**
20: ACencode(b_j, $\mathcal{P}_u[j][c]$, t)
21: **if** $b_j = 1$ **then**
22: ACencode(d, $\mathcal{P}'_u[j][\phi_{sign}(\omega_{y,x}, N-1)]$, t)
23: **end if**
24: **else**
25: ACencode(b_j, $\mathcal{P}''_u[j][0]$, t)
26: **end if**
27: **end for**
28: **end for**
29: **end for**

$$N = \min\left(M, \left\lfloor M \cdot \frac{K}{\mathcal{L}_u} \right\rfloor\right) . \tag{3}$$

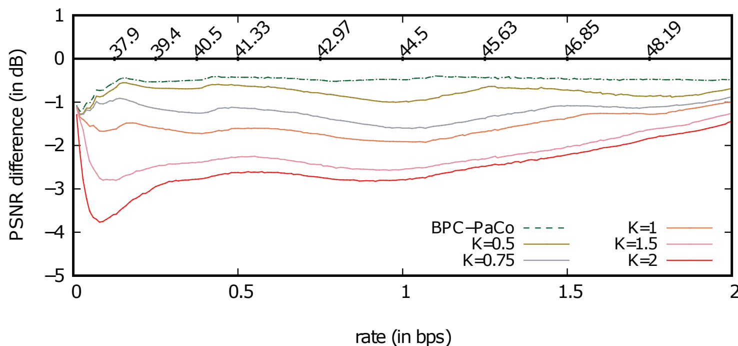
K is the input parameter of our implementation that controls the computational complexity of the codec. Larger Ks achieve larger Ns, so more bitplanes are coded in fast mode, rising the codec's throughput. $\mathcal{L}_u$ is the L_2norm of the synthesis basis vectors of the subband's filter-bank (it is assumed equal energy gain factor in all subbands). The higher the decomposition level, the more decreases the K, resulting in lower Ns in the smallest resolution levels. Through this strategy, the fast mode is applied at different bitplanes depending on the data and subband of the codeblock, providing more variability to the rate-distortion optimization method.

IV. Experimental results

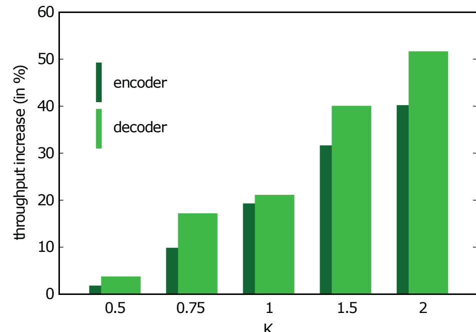
The ISO 12640-1 corpus is employed (8 color images, 2560×2048, and 8 bits per sample (bps)). The results report the performance achieved by JPEG2000, BPC-PaCo, and the proposed CS BPC-PaCo. The same Java framework BOI [29] is used for all codecs, using the same rate-distortion optimization method. Results for throughput are computed when the coding engine is executed with a single thread. This gives an approximation of the computational complexity of the algorithm. 5 levels of wavelet decomposition and codeblocks of 64×64 are employed. These coding parameters are selected as the most commonly used. A smaller codeblock size alleviates the penalization in coding efficiency of the proposed method,

TABLE I: Evaluation of the proposed method for lossless and lossy compression. All results are reported in bps except for the speedup, which is the percentage of CS BPC-PaCo with respect to BPC-PaCo (on average for the encoder and decoder).

	LOSSLESS COMPRESSION								LOSSY COMPRESSION							
		BPC-	*CS BPC-PaCo*							*BPC-*	*CS BPC-PaCo*					
	JP2	*PaCo*	$K = 0.5$		$K = 1.5$		$K = \infty$		*JP2*	*PaCo*	$K = 1$		$K = 2$		$K = \infty$	
Portrait	3.80	+0.21	+0.22	7%	+0.39	56%	+0.43	76%	2.60	+0.10	+0.16	21%	+0.27	51%	+0.31	68%
Cafe.	4.68	+0.13	+0.16	10%	+0.47	59%	+0.52	74%	3.61	+0.08	+0.23	40%	+0.43	63%	+0.50	81%
Fruit	3.96	+0.20	+0.22	5%	+0.40	52%	+0.43	70%	2.73	+0.11	+0.18	24%	+0.28	48%	+0.32	62%
Wine	3.94	+0.20	+0.21	8%	+0.37	55%	+0.41	72%	2.71	+0.08	+0.14	27%	+0.22	50%	+0.27	70%
Bicycle	3.90	+0.20	+0.22	7%	+0.41	54%	+0.46	69%	2.67	+0.11	+0.20	29%	+0.32	55%	+0.36	70%
Orchid	3.44	+0.26	+0.27	7%	+0.39	48%	+0.43	71%	2.15	+0.12	+0.17	20%	+0.25	46%	+0.28	62%
Music.	5.34	+0.20	+0.29	8%	+0.78	56%	+0.84	68%	4.40	+0.11	+0.36	42%	+0.61	67%	+0.66	79%
Candle	4.74	+0.15	+0.20	11%	+0.53	56%	+0.60	73%	3.69	+0.08	+0.26	40%	+0.47	66%	+0.52	76%
average	4.22	+0.20	+0.22	8%	+0.47	54%	+0.51	72%	3.07	+0.09	+0.21	30%	+0.36	56%	+0.40	71%



(a)



(b)

Fig. 1: Evaluation of (a) lossy coding performance and (b) throughput achieved for the "Orchid" image when using different Ks. In (a), the horizontal straight plot depicts the performance achieved by JPEG2000, whereas the other plots depict the performance achieved by the proposed method with different Ks, or by BPC-PaCo.

whereas fewer wavelet levels degrades coding performance significantly at low rates.

Table I (left) reports the results for lossless compression. They use $K = \{0.5, 1.5, \infty\}$. $K = \infty$ achieves the fastest speed since all bitplanes are coded with the fast mode. The results of this table suggest that CS BPC-PaCo can accelerate the coding process of the original engine by 72% whereas the penalization in coding performance is, with the fastest speed, 13% and 10% as compared to JPEG2000 and BPC-PaCo, respectively.

Table I (right) reports lossy compression results when $K = \{1, 2, \infty\}$. This test evaluates the rate and throughput increase achieved when all bitplanes are coded, achieving a quality above 50 dB. The results suggest that when $K = 1$ the engine's throughput is increased by 30% whereas the rate by 4% with respect to BPC-PaCo. For $K = 2$ ($K = \infty$), throughput and rate are respectively increased by 56% (71%) and 9% (10%). Note that, in terms of percentage, the throughput is more increased than the decrease in compression efficiency.

Fig. 1(a) evaluates the quality scalability achieved by the proposed method, for the "Orchid" image. Results hold for the others. The figure reports the coding performance achieved at 200 rates equivalently distributed between 0.01 and 2 bps, in terms of Peak Signal to Noise Ratio (PSNR) difference between JPEG2000 and BPC-PaCo or CS BPC-PaCo when using different Ks. The results are obtained when all bitplanes are coded and then the rate-distortion optimization method constructs the final file, or quality layers, at the target rates. For $K \in (0, 1)$ the losses in coding performance are below 2 dB for the whole rate range. Larger Ks result in higher losses, especially for low rates. This penalization in compression efficiency is caused by both the lack of enough truncation points and the poorer efficiency of the arithmetic coder due to rougher estimates of the coefficients. Fig. 1(b) reports the throughput's increase when using the same Ks as before. When $K = 1$ the throughput is increased in 20% while slightly affecting the coding efficiency (see Fig. 1(a)). Although larger Ks penalize more the image quality, the throughput's enhancement is significant, achieving more than 40% when $K = 2$.

V. Conclusions

Many efforts have been done to increase the throughput of image and video codecs. Common approaches are to implement them in hardware or to simplify their algorithms, which sometimes sacrifices some features. This paper presents a fast bitplane coding engine that, in addition to the features of the JPEG2000 standard, provides complexity scalability. To do so, it uses a coding engine that processes the coefficients in parallel and, when indicated, changes the conventional coding of bitplanes to a fast mode that codes all bits of the coefficients at once. Experimental results indicate that the proposed method can effectively regulate the codec's throughput. When using the minimum complexity, the throughput and rate are increased by about 70% and 13%, respectively, whereas the maximum complexity increases throughput and rate by about 10% and 2%, respectively, on average for the employed corpus and for lossy and lossless compression.

REFERENCES

[1] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 3, pp. 243–250, Jun. 1996.

[2] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Trans. Image Process.*, vol. 9, no. 7, pp. 1158–1170, Jul. 2000.

[3] W. A. Pearlman, A. Islam, N. Nagaraj, and A. Said, "Efficient, low-complexity image coding with a set-partitioning embedded block coder," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 14, no. 11, pp. 1219–1235, Nov. 2004.

[4] G. Xie and H. Shen, "Highly scalable, low-complexity image coding using zeroblocks of wavelet coefficients," *IEEE Trans. Image Process.*, vol. 15, no. 6, pp. 762–770, Jun. 2005.

[5] M. Dyer, D. Taubman, S. Nooshabadi, and A. K. Gupta, "Concurrency techniques for arithmetic coding in JPEG2000," *IEEE Trans. Circuits Syst. I*, vol. 53, no. 6, pp. 1203–1212, Jun. 2006.

[6] M. Rhu and I.-C. Park, "Optimization of arithmetic coding for JPEG2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 20, no. 3, pp. 446–451, Mar. 2010.

[7] F. Auli-Llinas and M. W. Marcellin, "Scanning order strategies for bitplane image coding," *IEEE Trans. Image Process.*, vol. 21, no. 4, pp. 1920–1933, Apr. 2012.

[8] X. Song, Q. Huang, S. Chang, J. He, and H. Wang, "Three-dimensional separate descendant-based SPIHT algorithm for fast compression of high-resolution medical image sequences," vol. 11, no. 1, pp. 80–87, Jan. 2017.

[9] A. K. Gupta, S. Nooshabadi, D. Taubman, and M. Dyer, "Realizing low-cost high-throughput general-purpose block encoder for JPEG2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 7, pp. 843–858, Jul. 2006.

[10] K. Mei, N. Zheng, C. Huang, Y. Liu, and Q. Zeng, "VLSI design of a high-speed and area-efficient JPEG2000 encoder," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 17, no. 8, pp. 1065–1078, Aug. 2007.

[11] M. Dyer, S. Nooshabadi, and D. Taubman, "Design and analysis of system on a chip encoder for JPEG2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 19, no. 2, pp. 215–225, Feb. 2009.

[12] S. Kim, D. Lee, J.-S. Kim, , and H.-J. Lee, "A high-throughput hardware design of a one-dimensional SPIHT algorithm," *IEEE Trans. Multimedia*, vol. 18, no. 3, pp. 392–404, Mar. 2016.

[13] H.-C. Fang, Y.-W. Chang, T.-C. Wang, C.-J. Lian, and L.-G. Chen, "Parallel embedded block coding architecture for JPEG 2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1086–1097, Sep. 2005.

[14] Y. Li and M. Bayoumi, "A three-level parallel high-speed low-power architecture for EBCOT of JPEG 2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 9, pp. 1153–1163, Sep. 2006.

[15] K. Sarawadekar and S. Banerjee, "An efficient pass-parallel architecture for embedded block coder in JPEG 2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 21, no. 6, pp. 825–836, Jun. 2011.

[16] Y. Jin and H.-J. Lee, "A block-based pass-parallel SPIHT algorithm," vol. 22, no. 7, pp. 1064–1075, Jul. 2012.

[17] M. S. Nobile, P. Cazzaniga, A. Tangherloni, and D. Besozzi, "Graphics processing units in bioinformatics, computational biology and systems biology," *Briefings in Bioinformatics*, vol. 18, no. 5, pp. 870–885, Sep. 2017.

[18] *High Throughput JPEG 2000 (HTJ2K): Call for Proposals*, ISO/IEC Std., 2017, document ISO/IEC JTC 1/SC29/WG1 N76037.

[19] D. Taubman, A. Naman, and R. Mathew, "High throughput block coding in the HTJ2K compression standard," in *Proc. IEEE International Conference on Image Processing*, Sep. 2019, pp. 1079–1083.

[20] A. Naman and D. Taubman, "Decoding high-throughput JPEG2000 (HTJ2K) on a GPU," in *Proc. IEEE International Conference on Image Processing*, Sep. 2019, pp. 1084–1088.

[21] 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, Jun. 2014.

[22] F. Auli-Llinas, "Context-adaptive binary arithmetic coding with fixed-length codewords," *IEEE Trans. Multimedia*, vol. 17, no. 8, pp. 1385–1390, Aug. 2015.

[23] 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, Dec. 2015.

[24] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, "Bitplane image coding with parallel coefficient processing," *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 209–219, Jan. 2016.

[25] 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, Aug. 2017.

[26] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "GPU architecture for wavelet-based video coding acceleration," in *Parallel Computing: Technology Trends*, vol. 36, Apr. 2020, pp. 83–92, IOSPress Series in Advances in Parallel Computing.

[27] F. Auli-Llinas, "Stationary probability model for bitplane image coding through local average of wavelet coefficients," *IEEE Trans. Image Process.*, vol. 20, no. 8, pp. 2153–2165, Aug. 2011.

[28] F. Auli-Llinas and J. Serra-Sagrista, "JPEG2000 quality scalability without quality layers," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 7, pp. 923–936, Jul. 2008.

[29] F. Auli-Llinas. (2019, Nov.) BOI codec. [Online]. Available: http://www.deic.uab.cat/~francesc/software/boi

Chapter 6

Real-time 16K Video Coding on a GPU with Complexity Scalable BPC-PaCo

Real-time 16K Video Coding on a GPU with Complexity Scalable BPC-PaCo

Carlos de Cea-Dominguez[a,*], Juan C. Moure[b], Joan Bartrina-Rapesta[a], Francesc Aulí-Llinàs[a]

[a]Dep. of Information and Communications Engineering, Universitat Autònoma de Barcelona - 08193 Bellaterra, Spain
[b]Dep. of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona - 08193 Bellaterra, Spain

Abstract

The advent of new technologies such as high dynamic range or 8K screens has enhanced the quality of digital images but it has also increased the codecs' computational demands to process such data. This paper presents a video codec that, while providing the same coding features and performance as those of JPEG2000, can process 16K video in real time using a consumer-grade GPU. This high throughput is achieved with a technique that introduces complexity scalability to a bitplane coding engine, which is the most computationally complex stage of the coding pipeline. The resulting codec can trade throughput for coding performance depending on the user's needs. Experimental results suggest that our method can double the throughput achieved by CPU implementations of the recently approved High-Throughput JPEG2000 and by hardwired implementations of HEVC in a GPU.

Keywords: High Throughput Image and Video Coding, GPU, CUDA, JPEG2000, HTJ2K

1. Introduction

Image and video coding are primary needs of industries such as digital cinema, content streaming or video production, among others. Two main standards satisfy the requirements of many such industries, namely, JPEG2000 [1] and HEVC [2]. JPEG2000 is commonly employed in digital cinema and medical imaging, whereas HEVC is often used for media streaming and video production. Both standards have advanced features like high compression efficiency, quality scalability, interactive transmission, or error resilience. Both standards also demand ample computational resources, posing a challenge when high quality video (of 4K or more resolution and/or with high dynamic range) need to be coded in real time. In computational-constrained devices, the image quality may need to be reduced to achieve real-time processing. Other scenarios such as digital cinema or medical imaging require the highest quality possible, so expensive hardware solutions are often in use.

The literature employs different approaches to increase the codecs' throughput. Some works focus on the coding algorithms to reduce computational complexity [3, 4, 5]. Others implement the codec in hardware devices such as Field-Programmable Gate Arrays (FPGAs) [6, 7, 8, 9]. FPGAs are attractive despite their high price due to their high performance, so they are used in scenarios such as digital cinema [10] or medical imaging [11, 12, 13]. The highly parallel architecture of Graphics Processing Units (GPUs) has also been employed to parallelize the codec's tasks [14, 15, 16]. The lower cost and the capacity for general-purpose computing of GPUs have made these accelerators very popular in recent years.

When the algorithms exhibit fine-grained parallelism, implementations in GPUs can achieve high throughput thanks to the inherent Single Instruction Multiple Data (SIMD) architecture of these devices in combination with a Multiple Instruction Multiple Data (MIMD) architecture. Together, both characteristics allow processing thousands of threads executing the same instruction on different data. Some algorithms can be accelerated up to 20× as compared to implementations on traditional Central Processing Units (CPUs) [17]. Unfortunately, such speedups are not achieved when implementing conventional image/video codecs because their core algorithms exhibit poor fine-grained parallelism. In

*Corresponding author. Telephone: +34 935811861; Fax: +34 935813443; Postal address: Escola Enginyeria, UAB - 08193 Bellaterra, Spain.

Email addresses: carlos.decea@uab.cat (Carlos de Cea-Dominguez), juancarlos.moure@uab.cat (Juan C. Moure), joan.bartrina@uab.cat (Joan Bartrina-Rapesta), francesc.auli@uab.cat (Francesc Aulí-Llinàs)

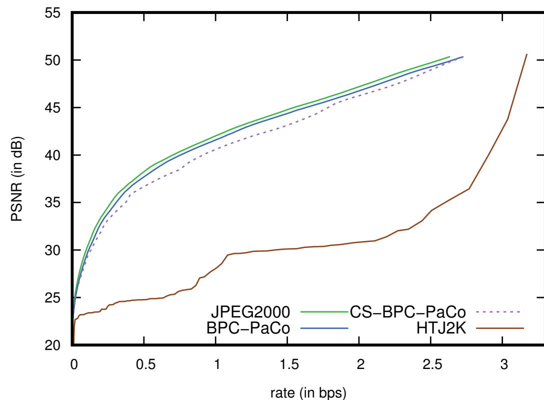


Figure 1: Evaluation of rate-distortion performance for JPEG2000, HTJ2K, BPC-PaCo and CS BPC-PaCo (with $K = 0.5$) when transmitting the color image "Portrait" at 100 different rates.

general, these algorithms are devised to exploit only the MIMD-based architecture of CPUs (or GPUs), which can process tenths of threads executing different instructions on different data.

The coding pipeline of a traditional JPEG2000 codec has three main stages: discrete wavelet transform (DWT), bitplane and entropy coding (BPC), and codestream re-organization (CR). The DWT and CR stages can be easily mapped to a SIMD-based architecture since their operations can be parallelized and do not hold critical data dependencies. Contrarily, the BPC stage has data dependencies that force the samples to be coded sequentially in each tile of data. This stage accounts for 85% of the total execution time, so it is key in the codec's overall throughput.

Aimed to provide more parallelism to the BPC engine, the Joint Photographic Experts Group approved in 2019 Part 15 of the standard, named High-Throughput JPEG2000 (HTJ2K). This new part adopts the algorithm proposed in [18], which exploits vector (SIMD) instructions included in modern CPUs and GPUs. HTJ2K can increase the throughput of a conventional JPEG2000 codec by about 10× at the expense of sacrificing: codestream compliance, compression efficiency (about 10%) and, quality scalability. Codestream compliance and compression efficiency are inevitably affected when modifying the coding engine, but these features are not essential in most scenarios. Quality scalability, on the other hand, is a valuable feature that allows partial decoding of the codestream at different rates while minimizing the distortion of the recovered image. See, for instance, in Fig. 1, the performance achieved by JPEG2000 and HTJ2K when the "Portrait" image (of corpus ISO/IEC 12640-1) is compressed and then transmitted at 100 different rates distributed between 0.01 and 3 bits per sample (bps). The vertical axis of the figure reports the quality of the recovered image in Peak Signal to Noise Ratio (PSNR), whereas the horizontal axis is the transmission rate. The quality achieved by HTJ2K is much lower than that achieved by the original JPEG2000 due to the lack of quality scalability.

This paper continues our line of research focused on providing fine-grained parallelism to all coding stages of an image/video codec. Our work originates in coding techniques that break the causality of classical coding strategies [19, 20, 21]. These techniques led to the development of a lightweight arithmetic coder that allows fine-grained parallelism [22, 23]. After that, the research focused on the stages of a JPEG2000-like codec. First, we proposed a GPU implementation of the DWT [14, 24] employing a highly-efficient register-based strategy. Second, the BPC engine was reformulated, resulting in a BPC with parallel coefficient processing (BPC-PaCo) [25, 26] that can efficiently exploit the resources of a GPU [15]. Third, we presented the GPU architecture for the end-to-end codec [16]. This codec can code up to 12K video in real time, achieves a compression efficiency comparable to that of the original JPEG2000 standard, and does *not* sacrifice quality scalability. See in Fig. 1 that the coding performance achieved by this codec is approximately only 2% inferior to that of JPEG2000.

Our last step proposes a complexity scalable technique for the coding engine. Complexity scalability allows trading computational complexity by compression efficiency so that the user can tune the codec to run more or less rapidly while marginally increasing the size of the compressed file. As it was studied in [27] and seen in Fig. 1, the proposed Complexity Scalable BPC-PaCo (CS BPC-PaCo) decreases only slightly the coding performance with respect to BPC-PaCo. This paper extends that work by first analyzing the computational bottleneck of the original BPC-PaCo in the GPU, which guides the development of the complexity scalable technique. Second, the proposed technique is introduced in our end-to-end codec evaluating its memory footprint, occupancy and performance, as well as the overall throughput achieved in different test conditions. Finally, experimental results evaluate the coding performance, throughput, and power consumption of the proposed method compared to other state-of-the-art codecs.

The rest of the paper is organized as follows. Section 2 reviews the architecture of the GPU, JPEG2000 and HTJ2K. Section 3 overviews the architecture of our codec, examines the aspects of BPC-PaCo that limit its throughput, and describes the implementa-

tion of CS BPC-PaCo. Experimental results are presented in Section 4 comparing the proposed method with JPEG2000, HTJ2K, and HEVC. Section 5 concludes with a brief summary.

2. Background

2.1. GPU architecture

Arguably, the most popular accelerators are currently those manufactured by Nvidia due to their low price, high performance, and capacity for general-purpose computing through the CUDA programming language, so they are employed in this work. Nvidia GPUs are constituted by many individual computing units called Streaming Multiprocessors (SM). Each SM is responsible of managing the execution of multiple 32-wide vector instructions in parallel. A GPU can have from one to a hundred of SMs. CUDA virtualizes each lane of a 32-wide vector instruction into a software thread. The group of 32 threads is referred to as a warp. Warps are organized in thread blocks, which are assigned to a SM for execution. Each thread block within a SM can execute tasks independently from the others, so different kernels (i.e., CUDA functions) from the same or different applications can run concurrently on the same SM. To organize the execution of kernels, CUDA provides the so-called streams. Each stream executes one or various kernels of an application in a pre-determined order. Since the GPU has abundant computational resources, concurrent streams of the same application can be executed in parallel to process different data.

Until CUDA v6.2, every thread in a warp executed instructions in a synchronous, lock-step fashion with the other threads of the warp. Implicit synchronization was featured at the end of every divergence in the execution flow. Since the release of CUDA v7.0, every thread in a warp can be executed asynchronously, so synchronization among threads must be explicitly programmed when needed. Our codec considers this aspect, producing the same result regardless of the architecture employed. For simplicity, the following sections assume that implicit synchronization is employed.

As Fig. 2 depicts, the memory architecture of a GPU has three levels: global memory, shared memory, and registers. The global memory is located in the device RAM or DRAM, has a size in the order of GBs, and is accessible by all SMs. This memory has high latency but relatively high bandwidth, so data transfers are to be carried out in a coalesced way (i.e., using consecutive memory positions) to maximize performance. The shared memory has a size in the order of MBs, has low latency and higher bandwidth, and can be accessed by all threads of a block. Each SM has an individual memory bank for this memory. The registers have very fast access, very high bandwidth, and a size of typically 256 KB. When the registers can not hold all the data required by the application, some data are temporarily moved to a reserved space in the device memory, the so-called local memory. This is called register spilling. It significantly affects the application's performance because transfers from/to the device memory render threads in an idle state due to the memory latency.

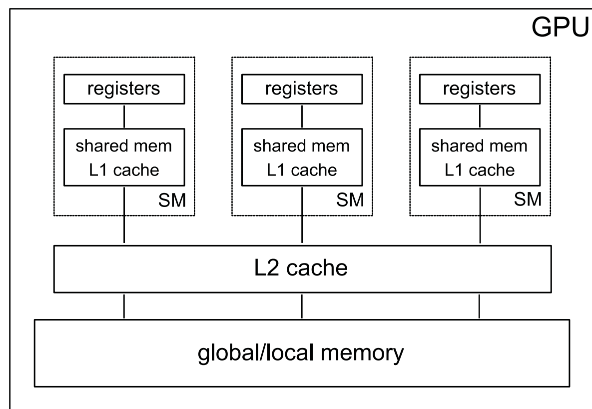


Figure 2: Memory hierarchy of a Nvidia GPU.

The memory architecture of the GPU is devised so that each execution kernel transfers the data required for computation from the global memory to the registers and then transfers back the results to the global memory. Communication among threads is commonly carried out via the shared memory or register shuffling. Each GPU has a Level 1 (L1) and Level 2 (L2) cache to minimize the latency when moving data from/to the device memory to/from the shared memory and registers. The L1 cache is located in the memory bank within the SM that also holds the shared memory, whereas the L2 cache is in a separate memory bank between the SMs and the device memory.

2.2. JPEG2000 architecture

As previously stated, the JPEG2000 coding pipeline has three main stages. The first reduces the spatial redundancy of the image through the DWT. The input to the DWT is either a gray image or a color image that has been converted to a color space that holds the luminance in the first component and the blue and red chrominance in the second and third component, respectively. The color transform (CT) is a pixel-wise operation without dependencies, so it is easily mapped to SIMD-based in-

structions. The DWT applies a series of arithmetic operations to all rows and columns of the image employing the so-called lifting scheme [28]. There are no dependencies between rows/columns, so these operations can be performed in parallel, suiting well SIMD programming too. The resulting wavelet coefficients are then reordered in four different subbands of one quarter the size of the original image. One of these subbands holds the low-pass details of the image, whereas the other three hold the high-pass details. In general, the DWT is applied 5 times on the low-pass subband to further compact the image energy. JPEG2000 provides reversible and irreversible operations for both the CT and DWT operations. The reversible path employs integer operations, so the original image can be recovered losslessly. The irreversible path employs floating-point operations that provide higher compression efficiency but produce losses in the reconstructed image. These losses can be controlled via the dead-zone quantization that is applied just after the DWT.

The second stage of the coding pipeline is the BPC. It is applied independently on data tiles that typically contain a set of 64×64 wavelet coefficients, called codeblocks. The coefficients within each codeblock are processed in a bitplane-by-bitplane fashion. A bitplane is the collection of bits b_j from all coefficients at binary position j, with $[b_{M-1}, b_{M-2}, ..., b_1, b_0]$, $b_i \in \{0, 1\}$ representing the binary representation of integer υ produced by the DWT (and quantization when using the irreversible path). The first non-zero bit of each coefficient, denoted by b_s, is referred to as significant bit. The sign of υ is denoted by $d \in \{+, -\}$ and is coded immediately after b_s so that the decoder can start approximating υ as soon as possible. JPEG2000 codes all bits of each bitplane in three coding passes. Each pass scans all the coefficients within the codeblock but only codes the bits of a group of selected coefficients. This three-pass strategy codes first the information that mostly reduces the distortion of the image. Each bit, with contextual information about the coefficient's neighbors, is fed to an arithmetic coder. The arithmetic coder employs this contextual information to adaptively adjust the probabilities of the processed bits, which is key to achieve compression. The output of the BPC stage is a bitstream per codeblock that can be truncated and re-organized in different layers of quality in the final codestream by the CR stage, the last of the coding pipeline.

The coding of codeblocks by independent threads provides the coarse-grained parallelism that suits CPUs, but GPUs need a finer parallelism. The BPC stage has many data dependencies. The most crucial is imposed by the arithmetic coder, which requires the result of the last processed bit to start coding the next. The contextual information and the group of coefficients selected in each coding pass also depend on the previously coded data, though these dependencies might be avoided at the expense of more computations. These aspects prevent parallelism at a coefficient level, which is the kind of fine-grained parallelism that GPUs may exploit more efficiently.

Part 15 of JPEG2000 (ISO/IEC 15444-15) provides more opportunities for fine-grained parallelism [18]. The logical partition in codeblocks is maintained but, instead of using a bitplane coding strategy, the coefficients are coded with a single coding pass in sets of 4×4 coefficients called quads. Most of the operations to code each quad do not hold critical dependencies with other quads. Entropy coding is carried out via variable-to-variable length codes, allowing parallel processing of quads. This coding strategy allows the use of vector instructions in modern CPUs and GPUs. Nonetheless, the use of a single coding pass disables quality scalability because the bitstream of each codeblock can not be truncated and re-organized as in the original JPEG2000. As seen in Fig. 1, this significantly reduces the quality of a compressed image transmitted at progressive rates. Also, the compression efficiency is penalized due to the use of a less efficient entropy coder than that of the original JPEG2000.

3. Proposed method

3.1. Codec architecture

The method proposed in this work extends our previous GPU-based architecture for the end-to-end codec [16] by introducing complexity scalability. The goal is to accelerate the coding process at the expense of decreasing compression efficiency in a way that can be controlled by the user.

First, let us briefly describe the architecture of our codec. Fig. 3 depicts the employed kernels. The architecture is devised so that each kernel performs all operations to a chunk of data before it needs to be re-organized for the following operations. This minimizes the transfers from/to the global memory since the data are fetched and returned to this memory only once in each kernel. More precisely, the CT kernel processes data tiles containing three color components from an image region, the DWT kernel processes data tiles containing samples of a single component, the BPC-PaCo kernel codes codeblocks, and the CR kernel re-organizes the bitstreams produced for each codeblock in the final codestream. This organization allows

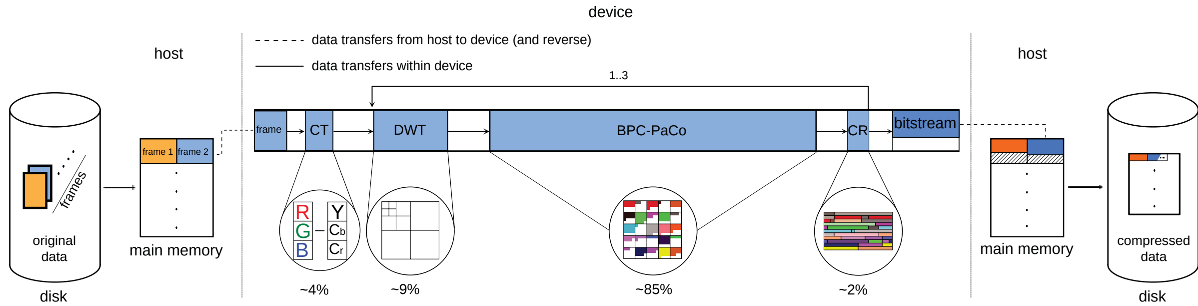


Figure 3: Illustration of the codec architecture when employing a single stream of execution in the GPU.

each kernel to compute many small data tiles in parallel, maximizing the overall throughput. In addition, the codec leverages the computational resources of the GPU through asynchronous I/O and multi-stream processing, and favors the use of register-based operations to communicate among threads in detriment of shared memory to avoid the latency of this memory.

Fig. 3 depicts below each kernel its computational load. BPC-PaCo approximately spends 85% of the total execution time, so its optimization may significantly increase the overall throughput. The remaining kernels represent less than 15% of the total load and their operations are indispensable and already highly optimized. As it is formulated in [16], the BPC-PaCo kernel uses two coding passes per bitplane. The significance pass codes the bits of those coefficients that were not significant in previous bitplanes, more precisely, those with $s \leq j$, with j representing the current bitplane. The refinement pass codes the bits of the remaining coefficients. The scanning order is devised so that each thread of a warp visits two columns of coefficients from the top to the bottom row. For significance coding, the context of the current coefficient υ is determined via the significance state of its eight adjacent neighbors as $\phi_{sig}(\upsilon, j) = \sum_k \Phi(\upsilon^k, j)$, with $\upsilon^k, 1 < k \leq 8$ denoting the neighbors and $\Phi(\upsilon^k, j) = 1$ or 0 when υ^k is significant or not, respectively. The context employed to code sign d is denoted by $\phi_{sign}(\upsilon, j)$ and employs a similar strategy, whereas the refinement pass uses a single context since little gain is achieved with more complex models [21], so $\phi_{ref}(\upsilon, j) = 0$. The probability estimate that the encoded bit b_j is 0 is extracted from a lookup table (LUT) known by encoder and decoder that is accessed as $\mathcal{P}_u[j][\phi_{sig}(\cdot)]$, with u denoting the wavelet subband. This LUT is pre-computed offline with a training set of images according to

$$P_{sig}(b_j = 0 \mid \phi_{sig}(\upsilon, j)) = \frac{\sum_{\upsilon=0}^{2^j-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j))}{\sum_{\upsilon=0}^{2^{j+1}-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j))}, \quad (1)$$

where $F_u(\upsilon \mid \phi_{sig}(\upsilon, j))$ is the probability mass function of the quantized coefficients at bitplane j given their context. Its support is $[0, ..., 2^{j+1} - 1]$ since it contains coefficients that were not significant in bitplanes greater than j. Probabilities for sign and refinement coding are derived similarly. Their respective LUTs are denoted by $\mathcal{P}'_u$ and $\mathcal{P}''_u$. Entropy coding of the emitted bits and their probabilities are carried out by each thread with an arithmetic coder that produces fixed-length codewords [29]. Threads cooperate among them to dispatch these codewords to the bitstream in a quality embedded order.

In BPC-PaCo, coefficients υ and ancillary data to process them are stored in the registers. Typically, each thread processes 128 coefficients (belonging to 2 columns of 64 coefficients), ideally requiring 128 registers of 32 bits plus some more for ancillary data. In current GPUs, this is too much information to hold in the register space. Each SM in current GPUs (Turing architecture) has 256 KB for registers and can run a maximum of 1024 threads. If all threads run in parallel, they can only access a maximum of 64 registers without causing register spilling and rendering some threads in an idle state.

As illustrated in the first row of Table 1, register spilling is the main bottleneck of the BPC-PaCo kernel. The table reports the transfers between memory device $\mathcal{M}^D$ and registers $\mathcal{R}$ that occur when the kernel processes a 4K image (gray scale, 8 bps). The component's data approximately requires 32 MB but, as seen in the table, 251 MB are read from $\mathcal{M}^D$ due to the ex-

	data reading (MB)		data writing (MB)		cache hit rates	
	$\mathcal{M}^D \rightarrow \mathcal{R}$	$L2 \rightarrow L1$	$L1 \rightarrow L2$	$\mathcal{R} \rightarrow \mathcal{M}^D$	L1	L2
BPC-PaCo	251	283	129	108	69%	39%
CS BPC-PaCo K=0.5	234	265	119	97	70%	39%
CS BPC-PaCo K=1	182	212	106	78	72%	43%
CS BPC-PaCo K=2	118	145	96	58	76%	51%
CS BPC-PaCo K=6	91	108	94	57	79%	55%

Table 1: Evaluation of memory and cache transfers when the kernels BPC-PaCo and CS BPC-PaCo code a 4K image in a RTX 2080 Ti GPU.

tensive use of local memory, as much as 8×. This increase is approximately 4× for writing. Table 1 also reports the data transfers between caches, which are similar, and the cache hit rates, which are moderately high because the data employed by the threads are frequently accessed and easily foreseeable. Despite the acceleration that the caches may provide, register spilling severely handicaps this kernel.

3.2. Complexity Scalable BPC-PaCo

The register spilling that occurs in BPC-PaCo is mainly caused by the multiple scanning of the coefficients during the coding process. The average number of coded bitplanes is 8, resulting in 16 accesses per coefficient. This generates multiple data transfers back and forth from the local memory since registers can not hold all the coefficients simultaneously.

The only way to reduce register spilling is minimizing the number of times that the coefficients are visited. However, the two-pass strategy is necessary to provide accurate estimates that achieve compression, and multiple truncation points that achieve quality scalability. The adopted strategy must alleviate the impact on these coding features, regulating the coding passes performed in each codeblock but without affecting the most relevant information in terms of distortion.

The technique employed herein was presented in [27] from a theoretical perspective that evaluates the impact on coding performance and quality scalability, but without implementing it in our end-to-end GPU codec. Its main insight is to code bitplanes $[M-1, N]$ with the same two-pass strategy of BPC-PaCo, and then use a fast mode that codes bitplanes $[N-1, 0]$ in a single pass. This codes the most relevant information in terms of distortion (contained in the highest bitplanes $[M-1, N]$) more progressively than the lesser relevant information, minimizing the impact on compression efficiency and quality scalability.

Choosing a suitable N is key to balance throughput and compression efficiency. A high N causes the coding of many bitplanes in fast mode, increasing the throughput but penalizing coding performance. A low N does not affect coding performance significantly though it does not provide significant throughput gains either. Instead of using the same N for all codeblocks, the strategy proposed in [30] uses different Ns depending on the codeblock's wavelet subband u and magnitude bitplanes M according to

$$N = \min\left(M, \left\lfloor M \cdot \frac{K}{\mathcal{L}_u} \right\rfloor\right) . \quad (2)$$

$\mathcal{L}_u$ is the L_2 norm of the synthesis basis vectors of the subband filter-bank (which is computed offline assuming equal energy gain factor in all subbands). Large Ks result in large Ns, so more bitplanes are coded with the fast mode, increasing the codec's throughput. Note that K is the user parameter that controls the speedup or, in other words, the mechanism through which complexity scalability is managed.

The coding technique embodied in Eq. 2 sets lower Ns to codeblocks within subbands in smaller resolution levels. Although these subbands have fewer codeblocks than in larger levels, these codeblocks have higher entropy than the rest, so coding them with more coding passes significantly enhances the quality scalability of the system. This is illustrated in Fig. 4. It depicts the images recovered when coding the "Portrait" image with (a) the same N in all codeblocks, and (b) the proposed strategy. Both codecs are set to achieve the same throughput.[1] The strategy that uses a fixed N (Fig. 4(a)) significantly degrades the image quality because the bitstream of some codeblocks within the lowest resolution levels are not included in the final codestream. The proposed strategy (Fig. 4(b)) provides an image with much higher quality. This holds for other coding parameters and images.

[1] Coding parameters for this test are: lossy compression, 2 DWT levels, 64×64 codeblocks, target rate 0.25 bps, $N = 15$, and $K = 6$ for the fixed and variable strategy, respectively.

(a) (b)

Figure 4: Visual evaluation of a (a) fixed and (b) variable strategy to set the bitplanes coded in fast mode with CS BPC-PaCo.

3.3. Implementation

Algorithm 1 details the proposed kernel from a thread perspective. The bitplanes in the range $[M-1, N]$ are coded with the original BPC-PaCo (lines 2 and 3) as described in [15, 16]. A significance and refinement pass are employed in each bitplane. The fast mode is used from bitplane $N-1$ to 0. Instead of visiting the coefficients twice per bitplane, the fast mode visits them only once and codes all their bits. Lines 5 and 6 in Algorithm 1 scan the coefficients. Since the context for significance coding is the same from bitplane $N-1$ onward, it is only computed once in line 7. Our implementation avoids this operation when all the coefficients are already significant. The loop in line 8 codes all bits of the coefficient considering its significance state. The arithmetic coder employs the procedure described in [15], which is not detailed herein for simplicity.

As previously stated, key to achieve high throughput is to reduce the number of registers that each thread employs. To this end, the 32-bit registers of the GPU hold all the information needed by the algorithm. In general, 24 bits are enough to hold the value of the coefficient (including the possible data expansion that the lossy DWT may produce), so ancillary data are stored in the remaining bits. Fig. 5 illustrates the binary representation of a GPU register. The lowest 24 bits store the magnitude and sign of υ, with the sign stored at the lowest bit. The highest 8 bits are employed for auxiliary information. The 3 upper bits are used in the SignificancePass() and RefinementPass() of Algorithm 1 to signal information regarding the significance state of the coefficient. The remaining 5 bits are employed to store the significance bitplane of the coefficient (i.e., s), which is employed in the fast mode to accelerate the operations that compute the context (i.e., in $\phi_{sig}(\cdot)$). In the example of Fig. 5, $M = 8$ and $N = 4$. The bitplanes depicted in blue represent those that are coded with two coding passes, whereas the bitplanes depicted

Algorithm 1 CS BPC-PaCo

Parameters: u subband, t stripe, M total magnitude bitplanes, N bitplanes coded in fast mode

1: **for** $j \in [M-1, N]$ **do**
2: SignificancePass()
3: RefinementPass()
4: **end for**
5: **for** $y \in [0, \text{numRows}-1]$ **do**
6: **for** $x \in [t \cdot 2, t \cdot 2 + 1]$ **do**
7: $c \leftarrow \phi_{sig}(\upsilon_{y,x}, N-1)$
8: **for** $j \in [N-1, 0]$ **do**
9: **if** $\Phi(\upsilon_{y,x}, j+1) = 0$ **then**
10: ACencode(b_j, $\mathcal{P}_u[j][c]$, t)
11: **if** $b_j = 1$ **then**
12: ACencode(d, $\mathcal{P}'_u[j][\phi_{sign}(\upsilon_{y,x}, N-1)]$, t)
13: **end if**
14: **else**
15: ACencode(b_j, $\mathcal{P}''_u[j][0]$, t)
16: **end if**
17: **end for**
18: **end for**
19: **end for**

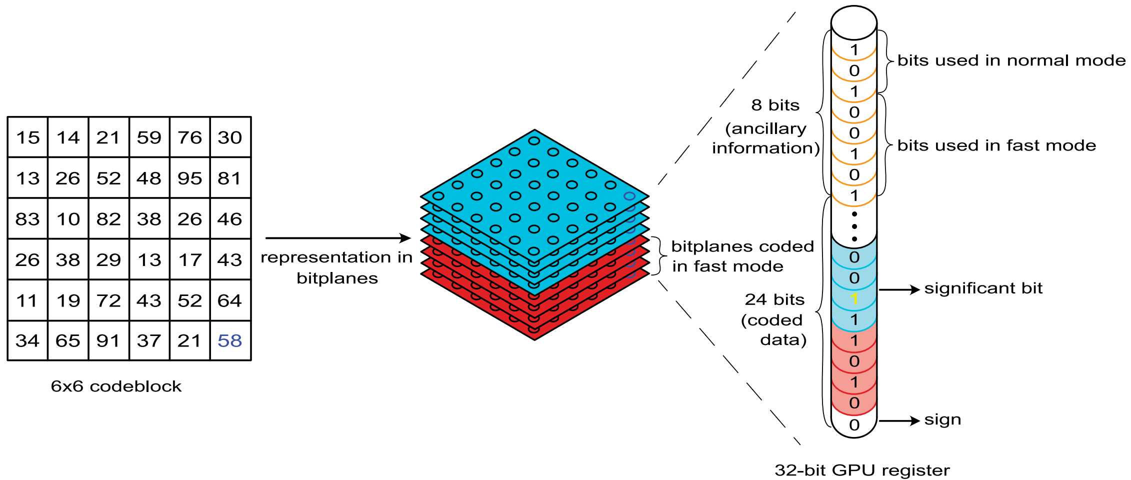


Figure 5: Illustration of a codeblock and the bit-allocation strategy in the 32-bit registers of the GPU employed by CS BPC-PaCo.

in red are coded with the fast mode.

This bit-allocation strategy in the registers minimizes the amount of local memory employed during execution time. The previous analysis of memory transfers for BPC-PaCo depicted in Table 1 also reports the results obtained for CS BPC-PaCo when different Ks are used. Data reading from $\mathcal{M}^D$ to $\mathcal{R}$ is proportionally reduced to the value of K. High Ks employ more extensively the fast mode of CS BPC-PaCo, reducing memory transfers. When $K = 6$, the proposed method only requires 36% of the memory transfers employed by the original BPC-PaCo. Memory transfers from $L2$ to $L1$ are reduced similarly. Data writing is reduced slightly less, though for $K = 6$ the proposed method approximately halves the transfers of BPC-PaCo. Since fewer data are employed by the algorithm, the cache hit rates for both $L1$ and $L2$ are increased, which provides even faster access to the data.

Table 2 illustrates the impact in the throughput achieved by the CS BPC-PaCo kernel when using different Ks as a result of reducing memory transfers. This evaluation employs the same 4K image of the test reported in Table 1. The second column reports the average number of clock cycles that each executed instruction is blocked due to the latency of the local memory, and the average cycles needed to execute each instruction (CPI). These metrics clearly illustrate the beneficial effect of using less local memory. BPC-PaCo blocks almost 10 cycles per instruction, requiring 15 cycles to execute each instruction. CS BPC-PaCo reduces the number of cycles in which instructions are blocked proportionally to the use of the fast mode. For $K = 6$, instructions are blocked only 1.56 cycles, whereas instructions only require 8 cycles to complete, on average. This improvement is also noted in the instructions executed per cycle (IPC) reported in the third column, which is almost doubled as compared to BPC-PaCo. The memory bandwidth (fourth column) employed by the kernel indicates that less bandwidth is needed as fewer coding passes are performed. The warp efficiency and GPU occupancy (fifth and sixth columns) is almost the same for all kernels since the algorithms have similar divergence (i.e., conditional paths in the execution flow). The total number of executed instructions (seventh column) is slightly higher in CS BPC-PaCo due to more instructions are needed when switching to the fast mode. Despite executing more instructions, the execution time of CS BPC-PaCo is reduced for all Ks because of the fewer memory transfers, with a reduction of 31% when $K = 6$.

Nvidia allows developers to manually set the number of registers assigned to the threads of a kernel. Evidently, to use too few registers per thread requires more local memory, while too many may cause an underuse of the GPU. Since the throughput achieved by our method highly depends on the local memory employed, the register assignment is carefully studied to yield maximum performance. Table 3 provides an evaluation of the throughput achieved when a different number of registers per thread is assigned to the proposed kernel. Different Ks are employed to consider different running conditions. The test is carried out for the same conditions as in previous evaluations, though results hold for other images and parameters. The third and fourth columns of the table depict the theoretical maximum and real GPU occupancy achieved, respectively. The maximum occupancy is calculated as the

		#cycles per inst. blocked (total CPI)	*IPC*	*bandwidth (GB/s)*	*warp efficiency*	*occupancy*	*#inst. ($\times 10^6$)*	*time (ms)*
BPC-PaCo		9.58 (15.29)	65	200	53%	46%	153	2.17
CS BPC-PaCo	*K=0.5*	6.49 (12.95)	77	156	56%	46%	159	1.93
	K=1	3.73 (10.33)	94	107	55%	46%	157	1.81
	K=2	1.86 (8.32)	114	74	53%	45%	153	1.61
	K=6	1.56 (8.00)	122	73	53%	46%	152	1.39

Table 2: Evaluation of throughput metrics when the kernels BPC-PaCo and CS BPC-PaCo code a 4K image in a RTX 2080 Ti GPU.

	registers per thread	*occupancy maximum*	*occupancy real*	*time (in ms) kernel*	*time (in ms) total*	*time (in ms) total av.*
K=0.5	96	62.5%	48%	1.86	11.77	10.17
K=1			49%	1.76	10.6	
K=2			48%	1.6	9.65	
K=6			48%	1.4	8.65	
K=0.5	80	75%	56%	1.88	11.43	10.06
K=1			54%	1.81	10.53	
K=2			55%	1.6	9.65	
K=6			56%	1.38	8.65	
K=0.5	72	87.5%	62%	1.93	11.5	9.94
K=1			61%	1.81	10.21	
K=2			62%	1.61	9.47	
K=6			64%	1.39	8.59	
K=0.5	64	100%	63%	1.96	11.62	10.4
K=1			61%	1.86	10.73	
K=2			64%	1.67	9.78	
K=6			66%	1.44	9.47	

Table 3: Evaluation of occupancy and execution time achieved by CS BPC-PaCo with different Ks when assigning a different number of registers to the threads, for a 4K image in a RTX 2080 Ti GPU.

number of threads that can be run in parallel using the assigned number of registers. It is only from a theoretical point of view since, in practice, threads are commonly blocked due to register spilling and other aspects. As seen in Table 3, even though 64 registers per thread achieves a theoretical maximum occupancy of 100%, the real occupancy achieved is about 63%. To assign 72 registers per thread decreases the maximum occupancy to 87,5%, though in practice is about 62% too. Also, to use 72 registers instead of 64 improves the throughput achieved since less register spilling occurs, decreasing the execution time of the kernel and of the end-to-end codec (5th and 6th columns in the table). To use more registers per thread slightly improves the performance of the CS BPC-PaCo kernel because less register spilling occurs, though for the overall end-to-end codec this is not beneficial because the other kernels running in parallel do not have enough resources. In all tests reported in this work, the CS BPC-PaCo kernel uses 72 registers per thread.

The speedup that the proposed kernel achieves with respect to the original BPC-PaCo is evaluated in Fig. 6. The figure reports in the vertical axis the speedup achieved for the Ks depicted in the horizontal axis, for both lossy and lossless compression when using a video sequence (see below). The results indicate that our method yields higher speedups for lossless compression, reaching a speedup of 70% for the highest K evaluated. Lossy compression achieves more moderate speedups, approximately up to 30%. This is because the floating-point DWT produces wavelet coefficients with higher magnitudes, and so more bitplanes are coded. We remark that the increase in throughput is higher than the increase in rate in all cases. When $K = 6$, for instance, CS BPC-PaCo achieves a speedup of approximately 65% (23%) while the rate increase is

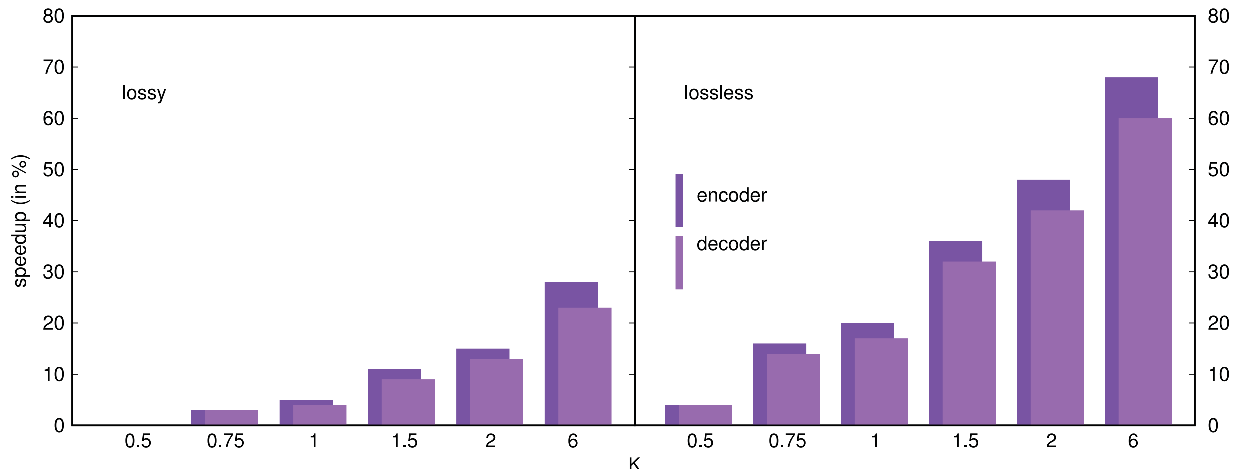


Figure 6: Evaluation of the throughput gain achieved by CS BPC-PaCo with respect to BPC-PaCo for different Ks.

14% (8%) for lossless (lossy) compression.

4. Experimental Results

The proposed CS BPC-PaCo is compared with BPC-PaCo and state-of-the-art codecs widely employed in the field, more precisely, the standard JPEG2000 and its new part HTJ2K, and the standard HEVC. All results below report the coding performance or throughput achieved by the end-to-end codec instead of only focusing on the CS BPC-PaCo kernel as in the previous section. Our method is run in two commodity GPUs from Nvidia, namely, the RTX 2080 Ti (68 SMs with 4352 cores at 1.6 GHz with 11 GB of RAM) and the GTX 1080 Ti (28 SMs with 3585 cores at 1.9 GHz with 11 GB of RAM). The former GPU uses the Nvidia microarchitecture called Turing (CUDA capability v7.5) and runs in a workstation with an Intel i9-9900K CPU with 16 GB of RAM. The latter uses the previous microarchitecture Pascal (CUDA capability v6.0) and runs in a workstation with an Intel i7-3770 CPU with 8 GB of RAM. Results for JPEG2000 and HTJ2K are obtained with Kakadu (v8.0.3) [31], which is among the fastest CPU implementations for JPEG2000 optimized with assembly and vector instructions. It runs in the i9-9900K workstation with 16 execution threads, yielding higher throughput than implementations of JPEG2000 for GPUs such as CuJ2K [32] and GPU-J2K [33]. Although HTJ2K can also be optimized for GPUs [34], to the best of our knowledge, there is no implementation that allows testing in the environment employed herein. Results for HEVC are obtained with the Nvidia implementation of the standard [35] running in both GPUs, which use a hardwired and specialized chip in the device. Coding parameters for our method and JPEG2000 are: lossy or lossless compression as indicated, 5 DWT levels, and codeblocks of 64×64. For HTJ2K, parameter "Cplex={6,EST,0.25,0}" is also employed to allow the codec to attain the specified target rate. HEVC uses a rate control method with constant quantization (1-51) for lossy compression, GOP=32, and high performance mode, which achieves maximum throughput in our tests. Throughput and power consumption results use a 2-minute segment of the "Star Wars: The Last Jedi" movie at 4K that has 2,880 color frames, resulting in 67.8 GB of uncompressed data. Coding performance results use the color image "Portrait" (with a size of 2560×2048) and a segment of the previous video sequence containing 948 gray-scale frames at 2K.

The first test evaluates lossy coding performance. Fig. 7 extends the results of Fig. 1 by including different Ks for the proposed method and the results for video. We recall that this test depicts rate vs. quality when the codestream is compressed and then transmitted at different rates. For both tests, the performance achieved by CS BPC-PaCo decreases as more coding passes are coded in fast mode (i.e., with higher values of K). The results indicate that the quality scalability of the proposed method is significantly better than that of HTJ2K since even when $K = 6$ and most passes are coded in a single pass, the drop in quality is approximately 5 dB with respect to JPEG2000 and BPC-PaCo, as compared to the losses of about 15 dB of HTJ2K.

The second test evaluates lossless compression. Table 4 reports the rate achieved when coding the video with all methods evaluated. BPC-PaCo yields almost

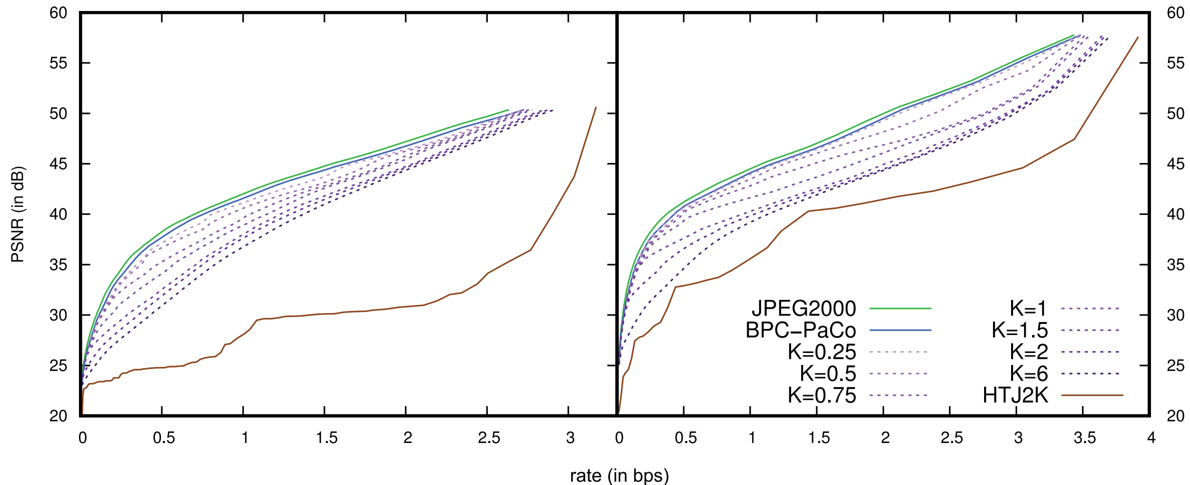


Figure 7: Evaluation of rate-distortion performance for JPEG2000, HTJ2K, BPC-PaCo and CS BPC-PaCo (with different Ks) when transmitting an image at 100 different rates (left) and a video sequence at 30 different rates (right).

CS BPC-PaCo		BPC-PaCo	JPEG2000	HTJ2K	HEVC
$K = 0.25$	3.82				
$K = 0.5$	3.83				
$K = 0.75$	3.89				
$K = 1$	3.91	3.82	3.79	4.06	4.03
$K = 1.5$	4.00				
$K = 2$	4.01				
$K = 6$	4.05				

Table 4: Evaluation of lossless compression performance for JPEG2000, HTJ2K, HEVC, BPC-PaCo, and CS BPC-PaCo (with different Ks). Results are reported in bps.

same performance to that of JPEG2000, while CS BPC-PaCo penalizes it slightly more with increments in rate of about 7% when $K = 6$. This increment is lower than that of HTJ2K, which almost obtains the same performance to that of CS BPC-PaCo when $K = 6$.

The third test evaluates throughput for both lossy and lossless video compression. In this test, the quality of the recovered video for lossy compression yields 50 dB in all codecs. Fig. 8 shows the results for all codecs and GPUs (or CPU for JPEG2000 and HTJ2K), reported in mega samples coded per second (MS/s). Two bars are depicted for each codec. The left bar corresponds to the encoder whereas the right to the decoder. BPC-PaCo is depicted with wide blue bars. The proposed CS BPC-PaCo is depicted with three thinner purple bars within that of BPC-PaCo, corresponding to the throughput achieved when $K = \{0.75, 2, 6\}$, with the thinnest bar for the highest K. The figure also shows with horizontal lines the throughput needed to code 4K, 8K, 12K, and 16K video at 24 frames per second in real time. As seen in the figure, the proposed method significantly increases the throughput with respect to BPC-PaCo, mostly in the encoder. In the decoder the gains are not as significant because the decoding process in our implementation needs more ancillary data, which hinders the overall throughput achieved. HTJ2K yields high throughput too, being slightly superior to that of our method for the GTX 1080 Ti in the case of lossless compression, though being 50% inferior (or more, depending on the K) when CS BPC-PaCo runs in the RTX 2080 Ti. The throughput achieved by JPEG2000 is much lower than that of HTJ2K due to the lack of opportunities for fine-grained parallelism in the algorithm. The throughput achieved by HEVC is modest as compared to the other codecs despite using a hardwired chip in the GPU. This is due to the techniques employed in this coding system, which achieve high coding performance at the expense of higher computational complexity. Finally, we remark that the scalability by complexity introduced in BPC-PaCo allows our codec to encode

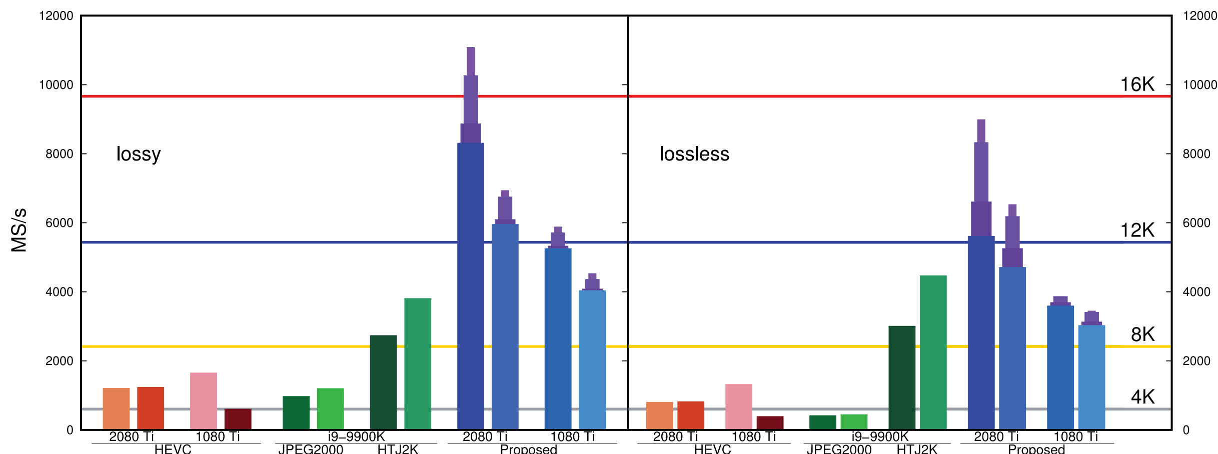


Figure 8: Throughput evaluation of lossy and lossless video compression for all codecs and GPUs/CPUs.

16K (12K) lossy video in real time with the RTX 2080 Ti (GTX 1080 Ti) when $K = 2$, obtaining a good trade-off between coding performance and throughput.

The previous test evaluates throughput for very high video quality. Some scenarios may allow lower video quality due to transmission or visualization aspects. The next test evaluates the throughput achieved when different quality levels are employed. Fig. 9 depicts in the horizontal axis the quality of the recovered video, which is set from 50 to 38 dB in all codecs. Lower quality yields similar results to those obtained for 38 dB. Again, results are reported in MS/s for the encoder and decoder. Blue and purple plots respectively correspond to BPC-PaCo and CS BPC-PaCo with the same Ks as those employed before. As expected, the lower the quality, the higher the throughput since fewer data are coded. The highest gains are achieved by the encoder of the proposed method. At 38 dB, all codecs except the proposed achieve similar throughput when encoding, which is about 3 to 4× lower than that of CS BPC-PaCo. The decoder presents more variations, with HEVC gaining much throughput for low qualities. It is worth noting that HTJ2K yields similar results regardless of the quality, obtaining the same throughput to that of JPEG2000 when encoding or decoding at 38 dB. These results suggest that the proposed CS BPC-PaCo achieves the highest throughput gains when using high quality, while low qualities render the throughput of the codec to almost the same as that of BPC-PaCo.

The last experimental test is aimed at energy consumption. The power demand of codecs running in the GPUs (CPUs) is obtained with the nvidia-smi (PowerTOP) tool, which provides the real consumption of the microprocessor depending on the workload. Fig. 10 reports the results in MS coded per Joule consumed when coding video at 50 dB. The figure illustrates the results in the same form as that of Fig. 8. The proposed method reduces energy consumption with respect to BPC-PaCo, with less consumption for higher Ks. Even so, these improvements are not as high as those obtained with the throughput. This is seen as the larger bars corresponding to CS BPC-PaCo in Fig. 8 vs. those depicted in Fig. 10. These results indicate that more energy has to be spent per coded sample to increment the codec's throughput. Even so, the results achieved with the RTX 2080 Ti suggest that our method consumes less energy than the other codecs evaluated. HTJ2K also consumes little energy compared to JPEG2000, which is the most demanding. The hardwired chip of HEVC in the GPU yields good results as well, except for decoding with the GTX 1080 Ti, which consumes energy similarly to the decoder of JPEG2000.

5. Conclusions

High-throughput and low-power consumption image and video codecs are a current necessity for new applications, cameras, and displays to allow real-time processing of very high resolution video and to extend the battery life of power-constrained devices. International organizations and researchers are proposing novel techniques, systems, and standards to fulfill these requirements. Some works pursue this goal by exploiting the high-performance computing of massively parallel architectures such as those found in Graphics Processing Units (GPUs). This is the line of research followed in this paper, which began by adapting and implementing all stages of a JPEG2000-based coding pipeline to the

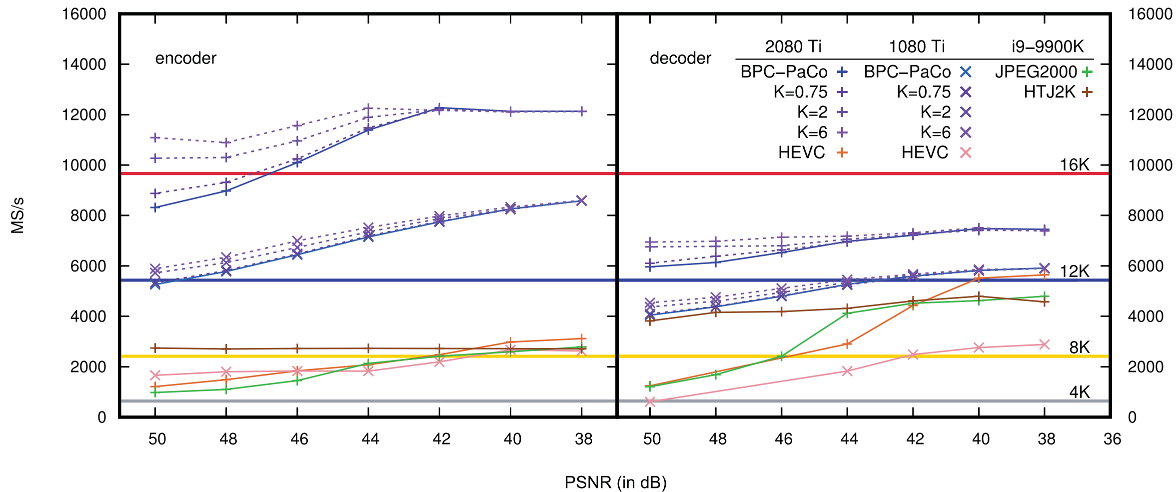


Figure 9: Throughput evaluation for lossy compression of video at different quality levels. Results are for BPC-PaCo and CS BPC-PaCo except when indicated.

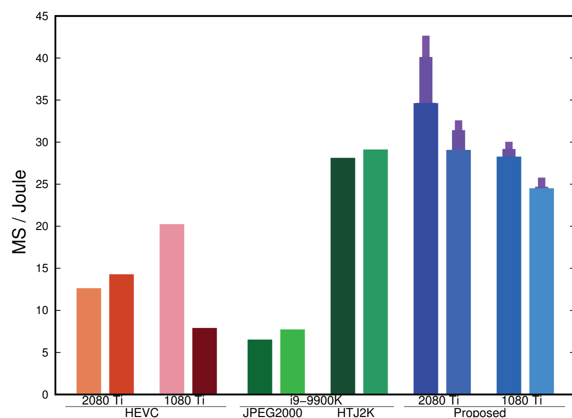


Figure 10: Evaluation of energy consumption for lossy video compression for all codecs and GPUs/CPUs.

fine-grained parallelism that suits GPUs. The bottleneck of the resulting codec is the bitplane and arithmetic coding stage, which spends most of the execution time. This work has analyzed this bottleneck by carefully profiling its execution on a GPU. Its main drawback is that it needs to transfer too much data from the local memory of the GPU to the registers (and viceversa) due to the coding of the image samples in many successive passes. The complexity scalable technique employed herein is tailored to increase the codec throughput in GPUs by reducing the coding passes performed. The proposed technique allows a user-handled control of the speedup achieved while minimizing losses in coding performance and quality scalability. Experimental results suggest that our codec attains higher throughput than other state-of-the-art codecs without sacrificing any feature of the coding system. Under some coding conditions, our method achieves real-time 16K coding of color video in a consumer-grade GPU (considering also memory transfers from host-to-device and viceversa), which is well above the current needs of most practical scenarios.

Acknowledgment

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under Grants TIN2017-84553-C2-1-R and RTI2018-095287-B-I00 (MINECO/FEDER, UE), and by the Catalan Government under Grants 2017SGR-463 and 2017SGR-313.

References

[1] ISO/IEC, Information technology - JPEG 2000 image coding system - Part 1: Core coding system (Dec. 2000).

[2] International Telecommunication Union, High Efficiency Video Coding Standard (2013).

[3] I. Marzuki, J. Ma, Y.-J. Ahn, D. Sim, A context-adaptive fast intra coding algorithm of high-efficiency video coding (HEVC), Journal of Real-Time Image Processing 16 (2019) 883–899.

[4] G. Correa, P. Assuncao, L. Agostini, L. A. da Silva Cruz, Complexity scalability for real-time HEVC encoders, Journal of Real-Time Image Processing 12 (2016) 107–122.

[5] Y. Wu, P. Liu, Y. Gao, K. Jia, Medical ultrasound video coding with H.265/HEVC based on ROI extraction, PLoS One (Nov. 2016).

[6] K. H. Yanzhe Li, Luc Claesen, M. Zhao, A real-time high-quality complete system for depth image-based rendering on FPGA, IEEE Transactions on Circuits and Systems for Video Technology 29 (4) (2019) 1179–1193.

[7] J. W. P. et al., A low-cost and high-throughput FPGA implementation of the retinex algorithm for real-time video enhancement, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 28 (1) (2020) 101–114.

[8] X. W. Xin Guo, Y. Liu, An FPGA implementation of multichannel video processing and 4k real-time display system, in: International Congress on Image and Signal Processing, BioMedical Engineering and Informatics, 2018, pp. 1–6.

[9] Z. He, H. Huang, M. Jiang, Y. Bai, G. Luo, FPGA-Based realtime super-resolution system for ultra high definition videos, in: Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018, pp. 181–188.

[10] A. Descampe, F.-O. Devaux, G. Rouvroy, J.-D. Legat, J.-J. Quisquater, B. Macq, A flexible hardware JPEG 2000 decoder for digital cinema, IEEE Trans. Circuits Syst. Video Technol. 16 (11) (2006) 1397–1410.

[11] I. Chiuchisan, A new FPGA-based real-time configurable system for medical image processing, in: E-Health and Bioengineering Conference (EHB), 2013, pp. 1–4.

[12] V. Kasik, Z. Chvostkova, FPGA in technical resources of medical imaging, in: IEEE 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2013, pp. 193–196.

[13] Junying Chen, Shunfeng Zhou, Huaqing Min, Implementation of parallel medical ultrasound imaging algorithm on CAPI-enabled FPGA, in: International Conference on Field-Programmable Technology (FPT), 2016, pp. 311–314.

[14] P. Enfedaque, F. Auli-Llinas, J. C. Moure, Implementation of the DWT in a GPU through a register-based strategy, IEEE Trans. Parallel Distrib. Syst. 26 (12) (2015) 3394–3406.

[15] P. Enfedaque, F. Auli-Llinas, J. C. Moure, GPU implementation of bitplane coding with parallel coefficient processing for high performance image compression, IEEE Trans. Parallel Distrib. Syst. 28 (8) (2017) 2272–2284.

[16] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, F. Auli-Llinas, GPU-oriented architecture for an end-to-end image/video codec based on JPEG2000, IEEE Access 8 (1) (2020) 68474–68487.

[17] M. S. Nobile, P. Cazzaniga, A. Tangherloni, D. Besozzi, Graphics processing units in bioinformatics, computational biology and systems biology, Briefings in Bioinformatics 18 (5) (2017) 870–885.

[18] D. Taubman, A. Naman, R. Mathew, High throughput block coding in the HTJ2K compression standard, in: Proc. IEEE International Conference on Image Processing, 2019, pp. 1079–1083.

[19] F. Auli-Llinas, Local average-based model of probabilities for JPEG2000 bitplane coder, in: Proc. IEEE Data Compression Conference, 2010, pp. 59–68.

[20] F. Auli-Llinas, I. Blanes, J. Bartrina-Rapesta, J. Serra-Sagrista, Stationary model of probabilities for symbols emitted by bitplane image coders, in: Proc. IEEE International Conference on Image Processing, 2010, pp. 497–500.

[21] F. Auli-Llinas, Stationary probability model for bitplane image coding through local average of wavelet coefficients, IEEE Trans. Image Process. 20 (8) (2011) 2153–2165.

[22] F. Auli-Llinas, Highly efficient, low complexity arithmetic coder for JPEG2000, in: Proc. IEEE International Conference on Image Processing, 2014, pp. 5601–5605.

[23] F. Auli-Llinas, Entropy-based evaluation of context models for wavelet-transformed images, IEEE Trans. Image Process. 24 (1) (2015) 57–67.

[24] P. Enfedaque, F. Auli-Llinas, J. C. Moure, Strategies of SIMD computing for image coding in GPU, in: Proc. IEEE International Conference on High Performance Computing, 2015, pp. 345–354.

[25] F. Auli-Llinas, P. Enfedaque, J. C. Moure, I. Blanes, V. Sanchez, Strategy of microscopic parallelism for bitplane image coding, in: Proc. IEEE Data Compression Conference, 2015, pp. 163–172.

[26] F. Auli-Llinas, P. Enfedaque, J. C. Moure, V. Sanchez, Bitplane image coding with parallel coefficient processing, IEEE Trans. Image Process. 25 (1) (2016) 209–219.

[27] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, F. Auli-Llinas, Complexity scalable bitplane image coding with parallel coefficient processing, IEEE Signal Process. Lett. 27 (2020) 840–844.

[28] W. Sweldens, The lifting scheme: A construction of second generation wavelets, SIAM Journal on Mathematical Analysis 29 (2) (1998) 511–546.

[29] F. Auli-Llinas, Context-adaptive binary arithmetic coding with fixed-length codewords, IEEE Trans. Multimedia 17 (8) (2015) 1385–1390.

[30] F. Auli-Llinas, J. Serra-Sagrista, JPEG2000 quality scalability without quality layers, IEEE Trans. Circuits Syst. Video Technol. 18 (7) (2008) 923–936.

[31] D. Taubman, Kakadu software, http://www.kakadusoftware.com (Jul. 2020).

[32] University of Stuttgart, CuJ2K, http://cuj2k.sourceforge.net/ (Jul. 2020).

[33] Poznan Supercomputing, Networking Center, GPUJ2K, http://apps.man.poznan.pl/trac/jpeg2k/wiki (Feb. 2020).

[34] A. Naman, D. Taubman, Decoding high-throughput JPEG2000 (HTJ2K) on a GPU, in: Proc. IEEE International Conference on Image Processing, 2019, pp. 1084–1088.

[35] Nvidia, HEVC SDK, https://developer.nvidia.com/nvidia-video-codec-sdk (Dec. 2018).

Chapter 7

Conclusions

7.1 Summary

This thesis has successfully created the first end-to-end GPU codec based on the JPEG2000 framework. Its first two publications created the image and video coding pipeline, presenting the first iterations of the infrastructure needed to run the kernels on the GPU. Careful management of data transfers is critical to achieve the best performance, as well as I/O management to avoid potential computation bottlenecks. The first multi-stream approach granted speed-ups of at least 20% over the single stream approach.

The third publication included flexible stream allocation, multiple CPU threads in charge of managing the GPU kernel invocations and smart memory management. Extensive analysis is included in a per-kernel basis to detail how data is processed and moved throughout the different memory systems within the GPU. The multi-stream system is further analyzed to evaluate the throughput gains depending on the amount of streams used and the image resolution of the input data on the 2080 Ti. As the different kernels included in the codec have different requirements in terms of memory transfers vs. computational complexity, this multi-stream approach improves the GPU resources allocation, granting more throughput. Results shows that more streams means more kernels being processed simultaneously within the GPU up to a limit, which depends on both, the size of the input data and the GPU used.

The fourth publication includes the new CS BPC-PaCo engine, with further improvements to coding throughput with a slight penalization in coding performance. The included results are based on a CPU implementation used to verify if the new feature renders any throughput gains to the algorithm design. Results showed double-digit throughput increases with slight coding performance penalizations of about 10%.

The fifth publication adapts CS BPC-PaCo implementation to the GPU codec and evaluates the new kernel in terms of register usage, occupancy, warp efficiency, memory bandwidth and memory transfers. This improvement focuses on reducing the memory transfers between device memory and registers within the kernel, which is the main limiting factor in BPC-PaCo. It is worth noting that it does not sacrifice any existing JPEG2000 feature, and is faster than the new HTJ2K codec using contemporary hardware while preserving quality scalability and roughly achieving the same coding performance results.

The digital image and video industry is demanding increased throughput and quality alike, with the inclusion of advanced technologies like HDR or ultra high resolutions like 8K. Nowadays, fields like digital cinema or medical imaging make use of expensive FPGAs to process data in real time. This thesis proves that there is a new and cheaper way to tackle the throughput challenges: to include native support to SIMD-based architectures such as GPUs, as their low cost compared with their high performance make them ideal for image and video processing. The software project for this work can be found at the following repository url: https://github.com/13Karl/CUDA-Image-and-Video-codec

7.2 Future research lines

The development of this codec in such a short period of time like the one available to a Ph.D. degree forced us to left behind some features that could have been implemented and which would have added more value to our proposed implementation. There are four main research lines that can be taken in consideration for further improving the GPU codec:

1. Codec software improvements: The codec pipeline exhibit some limitations

in certain algorithms or execution modes. First of all, the CS BPC-PaCo engine is limited to only accept blocks with a size of 64×64, thus the subbands generated by the DWT algorithm must be a multiple of that size. Removing this limitation would allow to process any image or video regardless of their size. Secondly, image coding is restricted to use one stream of execution. Multi-stream approach was only included for video as it made sense to process several frames simultaneously while not redesigning the pipeline execution flow to allow per-component processing in different streams. However, for images with several components this design could be improved so that each component is processed by a single stream. Currently, multispectral images can be processed as video files to enhance coding throughput, but at the expense of no additional transformations, achieving sub-optimal performance. Another approach instead of using more streams in images would be to remove the hard limitations between kernels and integrate every kernel within each other, allowing to have some sections of the image or frame to be processed by the DWT, others by the BPC and others by the CR. This would potentially reduce memory transactions as the data would not have to be moved from the registers to the device memory and viceversa several times per frame or component. It is worth noting that implementing this last idea could benefit both, image and video coding.

2. Hardware improvements: At the moment, the codec supports only one GPU at a time. Multi-GPU support would grant throughput improvements, although a fast I/O system would be needed to feed data fast enough to avoid bottlenecks. CPU could also be used to help the GPU process certain aspects of the pipeline, becoming a hybrid approach instead of just a GPU approach.

3. Nvidia architectural improvements from Turing and Ampere: at the moment this codec was designed, Turing and Ampere architectures were not released yet, and most of their new features were unknown to the public. Turing brought to the consumer public a new type of cores: tensor cores. These cores are commonly used in AI implementations as they are finely tuned to process matrices multiplications. Certain aspects of our codec could benefit from these cores, such as the CT kernel. With Ampere, there are two main aspects that could be of interest: DirectStorage functionality and cache improvements. The former one consists on transferring data

directly from a fast SSD, preferably running over the PCI-E 4.0 bus, to the GPU DRAM, without writing any data to the RAM, and with minimal CPU usage. This strategy would decrease the I/O subsystem complexity and remove RAM read/write timings from the pipeline, effectively freeing computational resources. The last one refers to cache improvements consisting on more and faster cache per SM, while also including a feature to bypass the L1 cache to directly copy data from the global memory to the shared memory, rendering shared memory a more valuable resource. All these features would render no benefits to older architectures.

4. Production improvements: the current codec version does not support HDR input data, nor any input data with a bit-depth higher than 8 bps. Also, the codec does not include support for audio.

Appendix A

List of Publications

[36] C. de Cea-Dominguez, P. Enfedaque, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, “High throughput image codec for high-resolution satellite images,” in *Proc. IEEE International Geoscience and Remote Sensing Symposium*, Jul. 2018, pp. 6524–6527

[37] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, “GPU architecture for wavelet-based video coding acceleration,” in *Parallel Computing: Technology Trends*, vol. 36, Apr. 2020, pp. 83–92, IOSPress Series in Advances in Parallel Computing

[24]C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, “GPU-oriented architecture for an end-to-end image/video codec based on JPEG2000,” *IEEE Access*, vol. 8, no. 1, pp. 68 474–68 487, Apr. 2020

[39] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, “Complexity scalable bitplane image coding with parallel coefficient processing,” *IEEE Signal Processing Letters*, vol. 27, pp. 840–844, Jun. 2020

[40] ——, “Real-time 16K video coding on a GPU with Complexity Scalable BPC-PaCo,” *Signal Processing: Image Communication*, vol. 99, p. 116503, Sep. 2021

Appendix B

Acronyms

AC Arithmetic Coder

BPC Bitplane Coding Engine

BPC-PaCo Bitplane Coding Engine with Parallel Coefficients

BPS Bits Per Sample

CPU Central Processing Unit

CR Codestream Reorganization

CS BPC-PaCo Complexity-Scalable Bitplane Coding Engine with Parallel Coefficients

CUDA Compute Unified Device Architecture

DRAM Device Random Access Memory

DWT Discrete Wavelet Transform

E2E End-to-end

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HTJ2K High Throughput JPEG2000

(I)CT (Irreversible) Color transform

MIMD Multiple-Instructions Multiple-Data

PSNR Peak Signal to Noise Ratio

RAM Random Access Memory

RGB Red, Green and Blue color space

SIMD Single-Instruction Multiple-Data

SM Streaming Multiprocessor

TDP Thermal Design Power

VRAM Video Random Access Memory

Bibliography

[1] "Consultative Committee for Space Data Systems (CCSDS)," http://www.ccsds.org, Jul. 2020.

[2] *Lossless Data Compression*, https://public.ccsds.org/Pubs/121x0b3.pdf, Consultative Committee for Space Data Systems, Aug. 2020.

[3] *Image Data Compression*, https://public.ccsds.org/Pubs/122x0b2.pdf, Consultative Committee for Space Data Systems, Sep. 2017.

[4] *Lossless Multispectral & Hyperspectral Image Compression*, https://public.ccsds.org/Pubs/123x0b2c2.pdf, Consultative Committee for Space Data Systems, Feb. 2019.

[5] *High Efficiency Video Coding Standard*, International Telecommunication Union, 2013.

[6] *Information technology - JPEG 2000 image coding system - Part 1: Core coding system*, ISO/IEC, Dec. 2000.

[7] A. Descampe, F.-O. Devaux, G. Rouvroy, J.-D. Legat, J.-J. Quisquater, and B. Macq, "A flexible hardware JPEG 2000 decoder for digital cinema," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 11, pp. 1397–1410, Nov. 2006.

[8] I. Chiuchisan, "A new FPGA-based real-time configurable system for medical image processing," in *E-Health and Bioengineering Conference (EHB)*, Nov. 2013, pp. 1–4.

[9] V. Kasik and Z. Chvostkova, “FPGA in technical resources of medical imaging,” in *IEEE 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Mar. 2013, pp. 193–196.

[10] Junying Chen, Shunfeng Zhou, and Huaqing Min, “Implementation of parallel medical ultrasound imaging algorithm on CAPI-enabled FPGA,” in *International Conference on Field-Programmable Technology (FPT)*, Dec. 2016, pp. 311–314.

[11] Nvidia, “Top supercomputing centers make use of Nvidia technologies,” https://blogs.nvidia.com/blog/2020/06/22/top500-isc-supercomputing/, Jun. 2020.

[12] M. S. Nobile, P. Cazzaniga, A. Tangherloni, and D. Besozzi, “Graphics processing units in bioinformatics, computational biology and systems biology,” vol. 18, no. 5, pp. 870–885, Sep. 2017.

[13] University of Stuttgart, “CuJ2K,” http://cuj2k.sourceforge.net/, Jul. 2020.

[14] Poznan Supercomputing and Networking Center, “GPUJ2K,” http://apps.man.poznan.pl/trac/jpeg2k/wiki, Feb. 2020.

[15] S. Datla and N. S. Gidijala, “Parallelizing motion JPEG 2000 with CUDA,” in *Proc. IEEE International Conference on Computer and Electrical Engineering*, Dec. 2009, pp. 630–634.

[16] R. Le, I. R. Bahar, and J. L. Mundy, “A novel parallel tier-1 coder for JPEG2000 using GPUs,” in *Proc. IEEE Symposium on Application Specific Processors*, Jun. 2011, pp. 129–136.

[17] J. Matela, V. Rusnak, and P. Holub, “Efficient JPEG2000 EBCOT context modeling for massively parallel architectures,” in *Proc. IEEE Data Compression Conference*, Mar. 2011, pp. 423–432.

[18] F. Wei, Q. Cui, and Y. Li, “Fine-granular parallel EBCOT and optimization with CUDA for digital cinema image compression,” in *Proc. IEEE International Conference on Multimedia and Expo*, Jul. 2012, pp. 1051–1054.

[19] M. Ciznicki, K. Kurowski, and A. Plaza, "Graphics processing unit implementation of JPEG2000 for hyperspectral image compression," vol. 6, pp. 1–14, Jan. 2012.

[20] J. Lee, B. Kim, and K. Yoon, "CUDA-based JPEG2000 encoding scheme," in *Proc. IEEE International Conference on Advanced Communication Technology*, Feb. 2014, pp. 671–674.

[21] X. Wu, Y. Li, K. Liu, K. Wang, and L. Wang, "Massive parallel implementation of JPEG2000 decoding algorithm with multi-GPUs," in *Proc. SPIE Satellite Data Compression, Communications, and Processing X*, vol. 9124, May 2014, pp. 1–6.

[22] D. Taubman, "Kakadu software," http://www.kakadusoftware.com, Jul. 2020.

[23] Comprimato, "Comprimato JPEG2000@GPU," http://www.comprimato.com, Feb. 2020.

[24] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "GPU-oriented architecture for an end-to-end image/video codec based on JPEG2000," *IEEE Access*, vol. 8, no. 1, pp. 68 474–68 487, Apr. 2020.

[25] D. Taubman, A. Naman, and R. Mathew, "High throughput block coding in the HTJ2K compression standard," in *Proc. IEEE International Conference on Image Processing*, Sep. 2019, pp. 1079–1083.

[26] F. Auli-Llinas, "Local average-based model of probabilities for JPEG2000 bitplane coder," in *Proc. IEEE Data Compression Conference*, Mar. 2010, pp. 59–68.

[27] F. Auli-Llinas, I. Blanes, J. Bartrina-Rapesta, and J. Serra-Sagrista, "Stationary model of probabilities for symbols emitted by bitplane image coders," in *Proc. IEEE International Conference on Image Processing*, Sep. 2010, pp. 497–500.

[28] F. Auli-Llinas, "Stationary probability model for bitplane image coding through local average of wavelet coefficients," *IEEE Transactions on Image Processing*, vol. 20, no. 8, pp. 2153–2165, Aug. 2011.

[29] ——, "Highly efficient, low complexity arithmetic coder for JPEG2000," in *Proc. IEEE International Conference on Image Processing*, Oct. 2014, pp. 5601–5605.

[30] ——, "Entropy-based evaluation of context models for wavelet-transformed images," *IEEE Transactions on Image Processing*, vol. 24, no. 1, pp. 57–67, Jan. 2015.

[31] P. Enfedaque, F. Auli-Llinas, and J. C. Moure, "Implementation of the DWT in a GPU through a register-based strategy," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3394–3406, Dec. 2015.

[32] ——, "Strategies of SIMD computing for image coding in GPU," in *Proc. IEEE International Conference on High Performance Computing*, Dec. 2015, pp. 345–354.

[33] F. Auli-Llinas, P. Enfedaque, J. C. Moure, I. Blanes, and V. Sanchez, "Strategy of microscopic parallelism for bitplane image coding," in *Proc. IEEE Data Compression Conference*, Apr. 2015, pp. 163–172.

[34] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, "Bitplane image coding with parallel coefficient processing," *IEEE Transactions on Image Processing*, vol. 25, no. 1, pp. 209–219, Jan. 2016.

[35] P. Enfedaque, F. Auli-Llinas, and J. C. Moure, "GPU implementation of bitplane coding with parallel coefficient processing for high performance image compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2272–2284, Aug. 2017.

[36] C. de Cea-Dominguez, P. Enfedaque, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "High throughput image codec for high-resolution satellite images," in *Proc. IEEE International Geoscience and Remote Sensing Symposium*, Jul. 2018, pp. 6524–6527.

[37] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "GPU architecture for wavelet-based video coding acceleration," in *Parallel Computing: Technology Trends*, vol. 36, Apr. 2020, pp. 83–92, IOSPress Series in Advances in Parallel Computing.

[38] Nvidia, "HEVC SDK," https://developer.nvidia.com/nvidia-video-codec-sdk, Dec. 2021.

[39] C. de Cea-Dominguez, J. C. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "Complexity scalable bitplane image coding with parallel coefficient processing," *IEEE Signal Processing Letters*, vol. 27, pp. 840–844, Jun. 2020.

[40] ——, "Real-time 16K video coding on a GPU with Complexity Scalable BPC-PaCo," *Signal Processing: Image Communication*, vol. 99, p. 116503, Sep. 2021.