Received March 10, 2020, accepted March 25, 2020, date of publication April 6, 2020, date of current version April 22, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2985859*

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

CARLOS DE CEA-DOMINGUEZ1, JUAN C. MOURE-LOPEZ2, JOAN BARTRINA-RAPESTA1, AND FRANCESC AUL˝-LLINÀS1, (Senior Member, IEEE)



* Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, 08193 Barcelona, Spain

2Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, 08193 Barcelona, Spain

Corresponding author: Carlos de Cea-Dominguez (carlos.decea@uab.cat)

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

 **ABSTRACT** Modern image and video compression standards employ computationally intensive algorithmsthat provide advanced features to the coding system. Current standards often need to be implemented in hard-ware 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 stan-dard (ISO/IEC 15444) [4] introduced more computationally demanding algorithms such as the discrete wavelet trans-form (DWT) [5] and bitplane coding [6]. In the last years, HEVC/H.265 (ISO/IEC 23008) [7] doubled the compression ef ciency of previous standards by using complex techniques that exploit intra- and inter-redundancy of frames. Nowa-days, most codecs (including JPEG2000 and HEVC) provide advanced features such as scalability by quality, interac-tive transmission, and error resilience, among others. To do so, they use algorithms that scan, transform, and code the

The associate editor coordinating the review of this manuscript and approving it for publication was Zhaoqing Pan .



samples[1](#page1) of the image multiple times, consuming signi cant processing time even when executed in the latest processors.

JPEG2000 is a widespread standard in elds that deal with large sets of images and/or videos. Its coding pipeline has three main stages [8]. The rst 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 coef cients. The third reorganizes the data to produce the nal codestream. The high computational complexity of these stages poses a challenge to meet the real-time requirements of some sce-narios. 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

1A 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).

|  |  |  |  |
| --- | --- | --- | --- |
| 68474 | This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/ | VOLUME 8, 2020 |  |
|  |  |

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

daily [10], [11]. This has motivated many works in the litera-ture that propose hardware architectures to accelerate partic-ular stages of the JPEG2000 coding pipeline [12] [22].

Highly parallel architectures may help to reduce process-ing time and costs in some environments. Graphics Process-ing 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 ow of instructions on mul-tiple data in a lock-step synchronous way. When the pro-gram allows data (in addition to task) parallelism, thou-sands 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 exe-cution of fewer threads over different sets of data.

Most of the workload in the rst stage of the JPEG2000 pipeline lies in the DWT, which is well-suited to the SIMD paradigm. The rst implementations of the DWT for GPUs appeared in the 2000s making use of the graphics pipeline [24] [26]. Later, the use of the Compute Uni ed Device Architecture (CUDA) programming language introduced by Nvidia increased the throughput of such imple-mentations signi cantly [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 coef cients, 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 coef cients. This means that the coding of a coef cient 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 solu-tions do not to fully occupy the resources of the GPU due to the lack of ne-grain parallelism. In 2014, we started a line of research [41] [45] focused on providing ne-grain parallelism to this stage *without* sacri cing 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 coef cient 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 cod-ing pipeline. This new part of JPEG2000 (ISO/IEC 15444-

VOLUME 8, 2020

1. adopts the algorithm proposed in [46]. Such algorithm is devised to mostly bene t 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 per-formance in approximately 10%. Also, it sacri ces 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 nal piece of our research line that was aimed to explore new cod-ing techniques for image/video compression tailored for the ne-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 sacri cing any feature. Evidently, compliance with the stan-dard is lost since the proposed techniques require signi cant 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 report-ing 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](#page2) brie y overviews the architecture of Nvidia GPUs and JPEG2000. Section [III](#page4) describes the proposed codec from a top-down perspective and Section [IV](#page7) details each kernel employed. Section [V](#page9) evaluates the throughput of our archi-tecture and compares it to some of the fastest JPEG2000 and HEVC implementations. The last section contains conclusions.

**II. BACKGROUND**

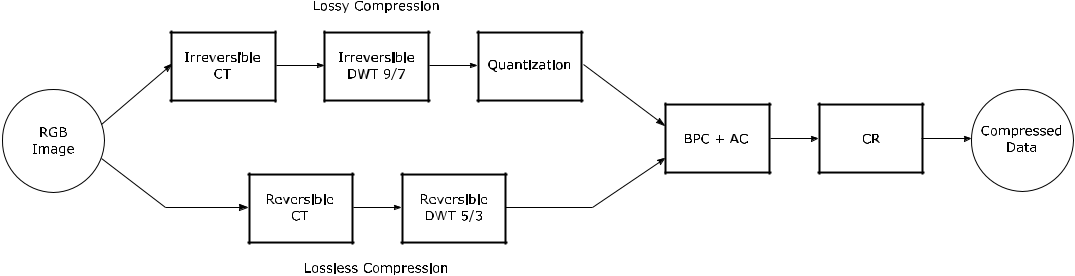
**A. NVIDIA GPU ARCHITECTURE**

Nvidia GPUs are hardware devices that are mainly consti-tuted by individual computing units called Streaming Mul-tiprocessors (SMs). Depending on the model and the archi-tecture, 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 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 rst CUDA-compatible architecture (v1.0) up to Pascal (v6.2), warps are always executed synchronously and in a lock-step

68475

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000



**FIGURE 1.** JPEG2000 coding pipeline.

fashion, featuring an implicit synchronization at the end of any divergence [50]. Volta (v7.0) introduced a modi cation 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 synchroniza-tion.

The memory architecture of the GPU is organized in three levels: global, shared, and local. The size of the global mem-ory 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 ef ciently 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 differ-ent 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 shuf e operation. This kind of memory sharing technique proved to be very ef cient 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 par-allel 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.

68476

**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 car-ries out almost the same operations as JPEG2000, so they are brie y described herein for completeness. Figure [1](#page3) depicts these operations. Depending on whether lossy or lossless compression is needed, some of these operations are irre-versible or reversible. As stated before, the rst stage of JPEG2000 applies several transformations to the image. The rst is carried out for color images, converting the red, green, and blue (RGB) components to the lesser redundant color space YC*b*C*r* , which holds the luminance information in the rst 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 oating-point or integer operations for the irreversible or reversible path, respectively.

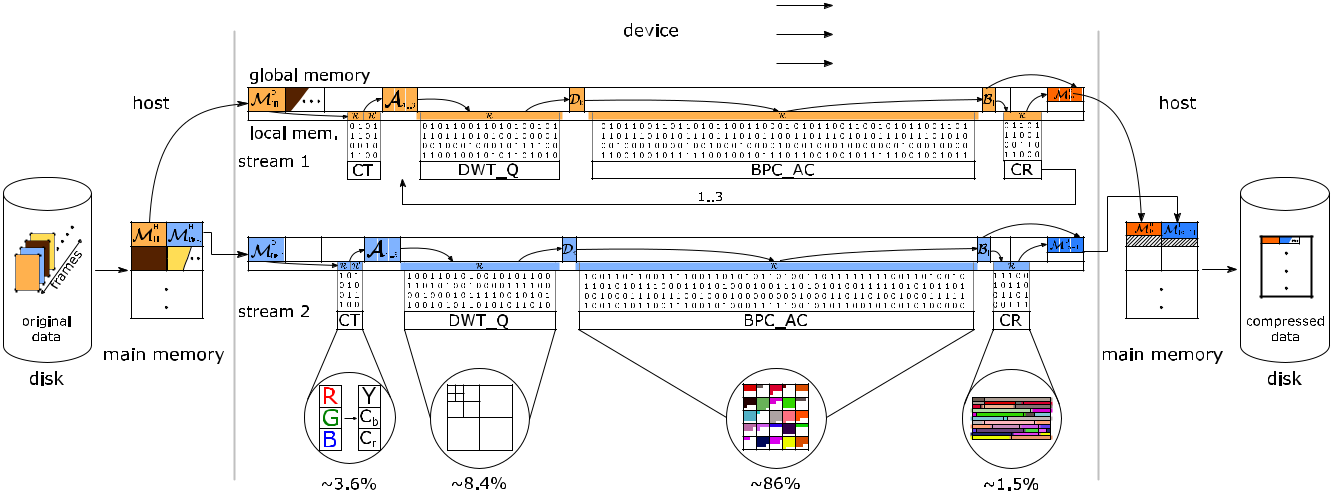
The second operation is the DWT. Most implementations apply it via the lifting scheme [55] since it has low com-putational complexity. The main idea behind this scheme is to rst apply a series of arithmetic operations to all rows of the image and then to all columns. These operations can be carried out in parallel to all rows and then to all columns since there are no inter-row/column dependencies. Then, the result-ing coef cients are re-ordered taking the coef cients 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 lter. JPEG2000 uses the irreversible CDF 9/7 and the reversible CDF 5/3.

The irreversible lter bank employs oating-point arith-metic, so the resulting coef cients need to be converted to integers before bitplane coding. This operation is called dead-zone quantization [8]. It multiplies the coef cients by a step size and keeps the integer part. This operation is not necessary for the reversible transform since it already produces integer coef cients.

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.

VOLUME 8, 2020

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000



**FIGURE 2.** Illustration of the codec architecture when using 2 CUDA streams. The cycle of the data is as follows. First, frame data (individually identifiedby 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 M**D** to local memory 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 M**H** and disk.

Through the binary representation of the integer coef cients (without sign), a bitplane is de ned as the set of bits from all coef cients in the same binary position. Bitplanes are coded from the most to the least signi cant. Just after the rst non-zero bit of a coef cient is coded (referred to as signi cance bit), its sign is coded too so that the decoder can reconstruct that coef cient. The bits coded for a coef - cient after its signi cance bit are called re nement bits. The coef cients within a codeblock are scanned in a pre-de ned order that visits four rows of coef cients, called stripes, consecutively. In each stripe, coef cients 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 rst is called signi cance propagation. It follows the scanning order processing only those coef - cients that have at least one signi cant neighbor. The second is called magnitude re nement. It processes coef cients that were found signi cant in previous bitplanes. The third pass processes the remaining coef cients. It is called cleanup. This multiple-pass coding is aimed to code rst 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 sig-ni cance, or sign, of its eight neighbors. One of 18 different pre-de ned contexts is chosen depending on this information. The context of the coef cient 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 nal codestream, so bitstreams may be truncated to t the target rate. This rate-distortion optimization proce-dure is not de ned in the standard, so each codec can choose among a great variety of methods [56]. The nal operation

re-organizes these bitstreams to put them in the compressed le 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.

1. **OVERVIEW OF THE CODEC ARCHITECTURE A. OVERVIEW**

Except for bitplane and arithmetic coding, all operations of the JPEG2000 coding pipeline offer ne-grain parallelism. Our codec implements these operations following the stan-dard, 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 signi - cantly 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.

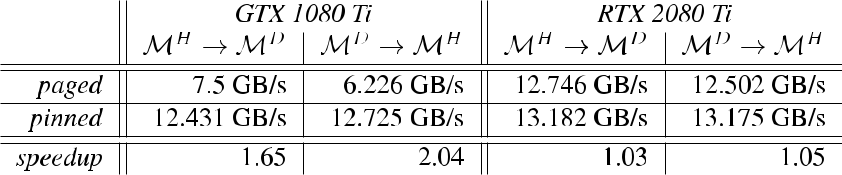
When possible, the proposed architecture joins operations in a single kernel instead of using a straightforward approach that uses one kernel per operation. Within the same kernel, the data are always accessed in the same fashion and the data types do not change. This permits the kernels to maximize 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 signi cantly increases performance.

Algorithm [1](#page5) describes the main routine of the codec. Its architecture is also illustrated in Figure [2.](#page4) First, all memory

|  |  |
| --- | --- |
| VOLUME 8, 2020 | 68477 |

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

**TABLE 1.** Evaluation of the memory bandwidth achieved by our codec when transferring data from host to device (M**H** !M**D**) and device to host(M**D** ! M**H** ) with pinned and paged memory, for two different GPUs.



**Algorithm 1** Main Routine of the Codec

1. CPUMemoryAllocation()
2. GPUGlobalMemoryAllocation()

* 3: **for** each empty M*H* [*i*] **do**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *t*1 | > 5: | | M*D*[[*ii*]] |  | *H* [*i*] |  |
|  | < | 4: | *H* | HDRead() | |  |
|  |  | M | M |  |  |

>

:

1. **end for**

8 7: A1::3 CT(M*D*[*i*])

>

* 8: **for** *k* 2 f1::3g **do**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *S* | *j* | > | 9: | *k* |  | DWT\_Q(A*k* ) | | |  |
|  | 10: | D *l* | | BPC\_AC( | D | *k* ) |  |
|  |  | > |  | fB g | |  |  |  |
|  |  | < |  | M | *D* | [*o*] CR(fB*l* g) | | |  |
|  |  | >11: | |  |  |

>

>

>

>

:

1. **end for**

* 13: **for** each lled M*D*[*o*] **do**

>14: M*H* [*o*] M*D*[*o*]

<

*t*2 15: HDWrite(M*H* [*o*])

>

:

1. **end for**

needed during the coding process is pre-allocated both in the host RAM and the device DRAM, which are respectively referred to as M*H* and 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](#page5) 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](#page5) the difference in the bandwidth achieved by our codec when coding a 4K video (with the test environment described in Section [V)](#page9) 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 dif-ferences 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

* Pinned memory indicates that the allocated space has a xed location in the RAM module(s) during the whole execution.

each frame. The reading of frames is managed by a thread, denoted by *t*1 in Algorithm [1,](#page5) that is executed by the host. Each stream, denoted by *Sj*; *j* 2 f1::b*S*g with b*S* being the number of streams, employs two input buffers in both M*H* and M*D* so that when a buffer is being processed the other can be lled. These buffers are referred to as M*H* [*i*], M*D*[*i*] with *i* 2 f1::2b*S*g. This lling is carried out in lines 3-6. *t*1 continuously checks if there is any empty buffer in M*H* . If so, it reads the data from disk and transfers them to M*H* . Then, it issues an asynchronous copy to the device memory 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.

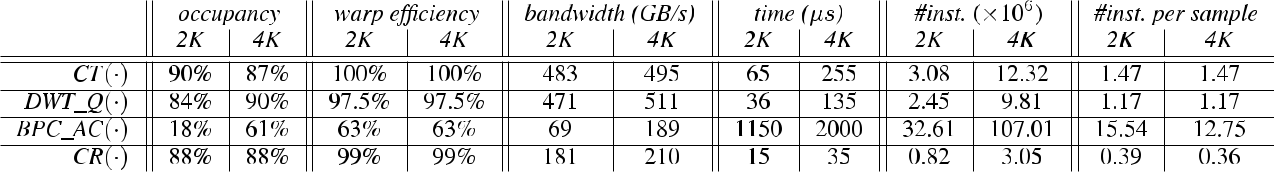
The writing of the compressed data to the disk is done similarly by thread *t*2, which is executed by the host in lines 13-16. A double-buffer strategy is also employed so that when a stream nishes coding a frame, it can readily start coding another without waiting for the compressed data to be transferred to the host memory. These output buffers are referred to as M*H* [*o*], M*D*[*o*] with *o* 2 f1::2b*S*g. Again, the data transfer from device to host is carried out via an asynchronous copy in line 14. Once the transfer is done, *t*2writes them to the disk. The data are copied in the diskorderly, i.e., following the same frame order of the original sequence.

Lines 7-12 in Algorithm [1](#page5) describe the calls to the ker-nels and the auxiliary memory structures employed in the GPU. Four kernels are used. The rst carries out the color transform. It transfers all frame data from M*D*[*i*] to local memory converting them to 32-bits integers ( oats) for the (ir)reversible path and performs the arithmetic operations on the registers. The result is left in the auxiliary struc-ture denoted by A1::3 using the same data type employed in the 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 the quantization step employed in this operation [56]. It transfers the data from A*k* to the registers, applies the lifting scheme, and leaves the result in 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 coef cients. Each codeblock is processed by an individual warp of 32 threads. The result of

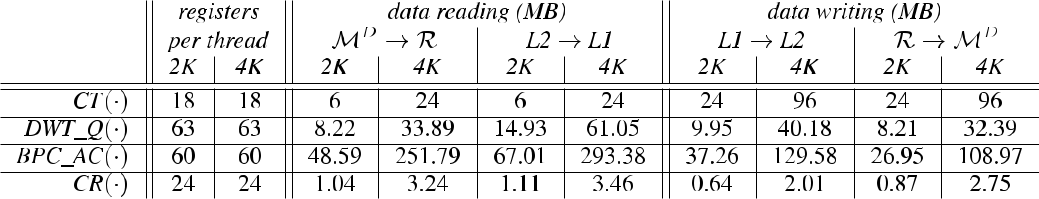
68478 VOLUME 8, 2020

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

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



**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.



this kernel is stored in the set fB*l* g that contains one bitstream per codeblock, with *l* 2 f1::b*L*g and b*L* being the number of codeblocks per component. The length of each bitstream is not known before coding, so the space for bitstreams fB*l* g 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 nal 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](#page6) and [3](#page6) 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.](#page9) The rst kernel (i.e, ICT( )) achieves high occupancy, optimal warp ef ciency (since it does not have divergence), and very high memory bandwidth (see Table [2)](#page6). 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,](#page6) the three image components are trans-ferred from M*D* to 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 oats depending on whether the reversible or irreversible transform is selected. This conversion is seen in the memory transfers

VOLUME 8, 2020

when the data are transferred back from the registers to the device memory via the L1 and L2 caches.

The DWT\_Q( ) kernel can perform a variable number of transformation levels, typically 5. The metrics reported in Table [2](#page6) correspond to the rst call to the kernel, which performs the rst level of transformation. The achieved occu-pancy 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 ef ciency 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,](#page6) 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 M*D* to 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( ) 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 ll the resources of the GPU. 4K frames achieve higher occupancy, though it is still below that achieved by the other kernels. Second, the warp ef ciency is 63% due to the multiple diver-gent 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.,

68479

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

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)](#page6). Differently from the previous kernels, BPC\_AC( ) visits each coef cient of the codeblock many times. The number of visits depends on the codeblock’s data, but is approximately 8 or 9 times per coef cient on average. Since the size of the register space is limited, once a coef cient is visited it is transferred back to the device memory so the register can be employed for other coef cients. When the coef cient is needed again, it is transferred from the device memory to the registers. Many of these coef cients 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 ef ciency 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 *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](#page7) requiring 10 *s* and 30 *s*. The memory bandwidth is lower than that obtained in the rst two kernels since the transferred data are already compressed(also seen in Table [3)](#page6).

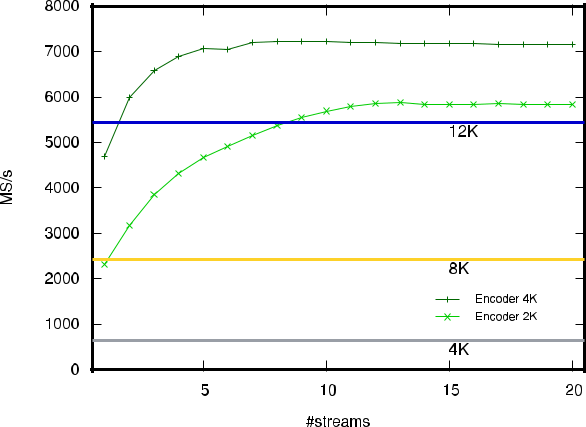
This analysis indicates that the BPC\_AC( ) kernel con-sumes most of the total execution time and it achieves the lowest occupancy. This suggests that the codec may underuse the resources of the GPU when coding large sets of images or video unless more workload is feed to the device. The pro-posed architecture 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](#page7) 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 gure 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 gure, the throughput increases notably when multiple streams are employed. In the case of 2K (4K) video, 13 14 (7 8) streams obtain maximum ef ciency. Again, the coding of 4K video achieves higher throughput due to the nature of the data.

As seen in Section [V](#page9) the throughput achieved by the decoder is only slightly lower than that of the encoder because

* 4K frames are compressed more ef ciently than 2K frames, so they generate fewer data per sample coded.

68480

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



**Algorithm 2** Kernel Routine CT(M*D*[*i*])

1. GPULocalMemoryAllocation()
2. R1::3 M*D*[*i*]

|  |  |  |  |
| --- | --- | --- | --- |
| 3: | 0 | ( | 1::3) |
|  | R1::3 | R |  |
| 4: | 1::3 | 0 |  |
|  | A | R1::3 | |

1. **return(**A1::3**)**

the decoder requires more local memory, which reduces the occupancy. The rest of the decoding process is very simi-lar to that of the encoder, so it is not reported herein for brevity.

**IV. DESCRIPTION OF THE KERNELS**

Algorithm [2](#page7) details the routine of the CT( ) kernel. In this and following kernels, the algorithm describes the main oper-ations that are performed at a thread level. Like in the other kernels, the rst 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 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 ( ) in line 3, and the result is left in the output register space R0. 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 data blocks adjacent to that requested in the L2 cache for (possi-ble) future requests. Depending on whether lossy or lossless compression is selected, the operations and the data types employed in the registers are oating points or integers, respectively.

The second kernel is detailed in Algorithm [3.](#page8) The wavelet transform is applied in blocks of 64 b*Y* samples that are

VOLUME 8, 2020

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

**Algorithm 3** Kernel Routine DWT\_Q(A*k*)

1. GPULocalMemoryAllocation()
2. R A*k*
3. **for** *y*2 f1::b*Y*g **do**
4. **for** *x*2 f0::1g **do**
5. R[*y*][*x*] ’(R[*y*][*x*])
6. **end for**
7. **end for**
8. **for** *x*2 f0::1g **do**
9. **for** *y*2 f1::b*Y*g **do**
10. R[*y*][*x*] ’(R[*y*][*x*])
11. **end for**
12. **end for**
13. **for** *y*2 f1::b*Y*g **do**
14. **for** *x*2 f0::1g **do**
15. **if** (*y*;*x*)2=halo **then**

16: R[*y*][*x*] R[*y*][*x*] *Q*

1. D*k* R[*y*][*x*]
2. **end if**
3. **end for**
4. **end for**
5. **return(**D*k* **)**

processed by a single warp.[4](#page8) This allows communication among threads without needing shared memory. The height of the block is denoted by b*Y*. Each thread processes two columns of a block. The kernel applies a 2D high-pass/low-pass lter to all samples. First, the lter ’( ) is applied horizontally (lines 3-7) and then vertically (lines 8-12). The lter consists in a series of arithmetic operations that use the adjacent samples to the processed coef cient, 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 R. When the thread needs data from other threads, it uses shuf e instructions (not shown in Algorithm [3)](#page8) 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 lter. This is carried out calling the kernel again. It is not detailed in Algorithm [3](#page8) for the sake of clarity. The nal 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](#page8) 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 oating points and integers when applying ’( ). Quantization is only applied for lossy compression.

* Note that these blocks are *not* the codeblocks utilized in BPC\_AC( ), but a tile of the original image. Although the partitioning is similar for paral-lelism purposes, the block transformed by DWT\_Q( ) contains overlapped samples of adjacent blocks.

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

VOLUME 8, 2020

The BPC\_AC( ) kernel is detailed in Algorithm [4.](#page9) 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 parti-tioned in codeblocks of 64 b*Y*0 coef cients, with typically b*Y*0 D 64. The data for the codeblock are implicitly transferred to the local memory in line 2 of the algorithm. Then the coef cients are coded from bitplane b*B*, which is a suf cient number of magnitude bits to code all coef cients 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 compres-sion ef ciency is achieved [6], [44], [45] while increasing throughput about 40%. The loop in lines 4-17 performs signi cance coding. It checks whether the coef cient was signi cant in previous bitplanes via the ( ) function, which returns the signi cance bitplane of the coef cient. If not, signi cance coding is performed. First, context *C* of the coef cient is determined via 8( ) 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( ) is not detailed in the algorithm for simplicity. It can be found in [45]. If the coef cient is signi cant in the current bitplane (i.e., (R[*y*][*x*]) D *b*), its sign is coded in lines 11-13 with a similar procedure to that of signi cance coding. Re nement coding is carried out in lines 18-25. In this case, no context is employed. The return of the AC( ) function is the bitstream B*l* that contains the compressed information. Each time that this function is called, some data may be added to B*l* . We note that B*l* is in the global memory. Each thread puts data in 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 B*l* in a separate global memory region, denoted by L.

The last kernel (i.e., CR( )) is detailed in Algorithm [5.](#page9) It receives the set of bitstreams fB*l* g. As previously stated, its purpose is to reorganize the bitstream data in a compact struc-ture. To do so, blocks of 2 bytes are assigned to each thread in the warp to be written in the nal memory positions. The rst step is to generate a memory map to know these positions.

This map is denoted as L0 and contains an aggregated list of lengths, more precisely, L0 D f0; L1; L1C L2; ; L1C C L*L* g. L0 is generated via the Device Scan primitive from the

b

Nvidia CUB framework [57]. To accelerate the access to this map, a fast lookup table, denoted by LUTL0 , is created. This LUT is generated applying a binary search over L0 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. These operations are carried out

68481

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

**Algorithm 4** Kernel Routine BPC\_AC(D*k*)

1. GPULocalMemoryAllocation()
2. R D*k*
3. **for** *b*2 fb*B*::0g*do*
4. **for** *y*2 f1::b*Y*0g **do**
5. **for** *x*2 f0::1g **do**
6. **if** (R[*y*][*x*])*b* **then**

7: *C* 8(R[*y*][*x*])

1. *P* LUT*sig*[*C*][*b*]
2. B*l* AC(R[*y*][*x*]; *P*)
3. **if** (R[*y*][*x*])D*b* **then**

|  |  |  |
| --- | --- | --- |
| 11: | *C*0 | 80(R[*y*][*x*]) |
| 12: | *P*0 | LUT*sign*[*C*0][*b*] |

1. B*l* AC(R[*y*][*x*]; *P*0)
2. **end if**
3. **end if**
4. **end for**
5. **end for**
6. **for** *y*2 f1::b*Y*0g **do**
7. **for** *x*2 f0::1g **do**
8. **if** (R[*y*][*x*])>*b* **then**

|  |  |  |
| --- | --- | --- |
| 21: | *P*00 | LUT*ref* [*b*] |
| 22: | B*l* | AC(R [*y*][*x*]; *P*00) |

1. **end if**
2. **end for**
3. **end for**
4. **end for**
5. L*l* length(B*l* )
6. **return(**B*l* **)**

**Algorithm 5** Kernel Routine CR(fB*l*g)

1: *S* computePosition(*T* ; LUTL0 ; L0)

1. **if** (*S*2 fL0g) **then**
2. M*D*[*o*][*H* ] B*l* [*S*]
3. **else**
4. M*D*[*o*][*D*] B*l* [*S*]
5. **end if**
6. **return(**M*D*[*o*]**)**

before running the CR( ) kernel, so they are not speci ed in Algorithm [5.](#page9)

Once the LUTL0 is created, each warp thread *T* com-putes 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 signi cant bitplane), or compressed data. This is carried out in line 2 checking if the thread is copying the rst bytes of the codeblock’s bitstream. The corresponding bytes are either copied to the header or body section of the nal structure, respectively denoted by M*D*[*o*][*H* ] and 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.

68482

**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 speci cations are reported in Table [4.](#page10) 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 ef ciency 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 per-formance 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 com-pression as indicated, 5 levels of DWT, and codeblocks of 64 64. Although there are other competitive GPU imple-mentations 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 cor-responding webpages suggest that they obtain competitive throughput, though lower to that achieved by the proposed codec.

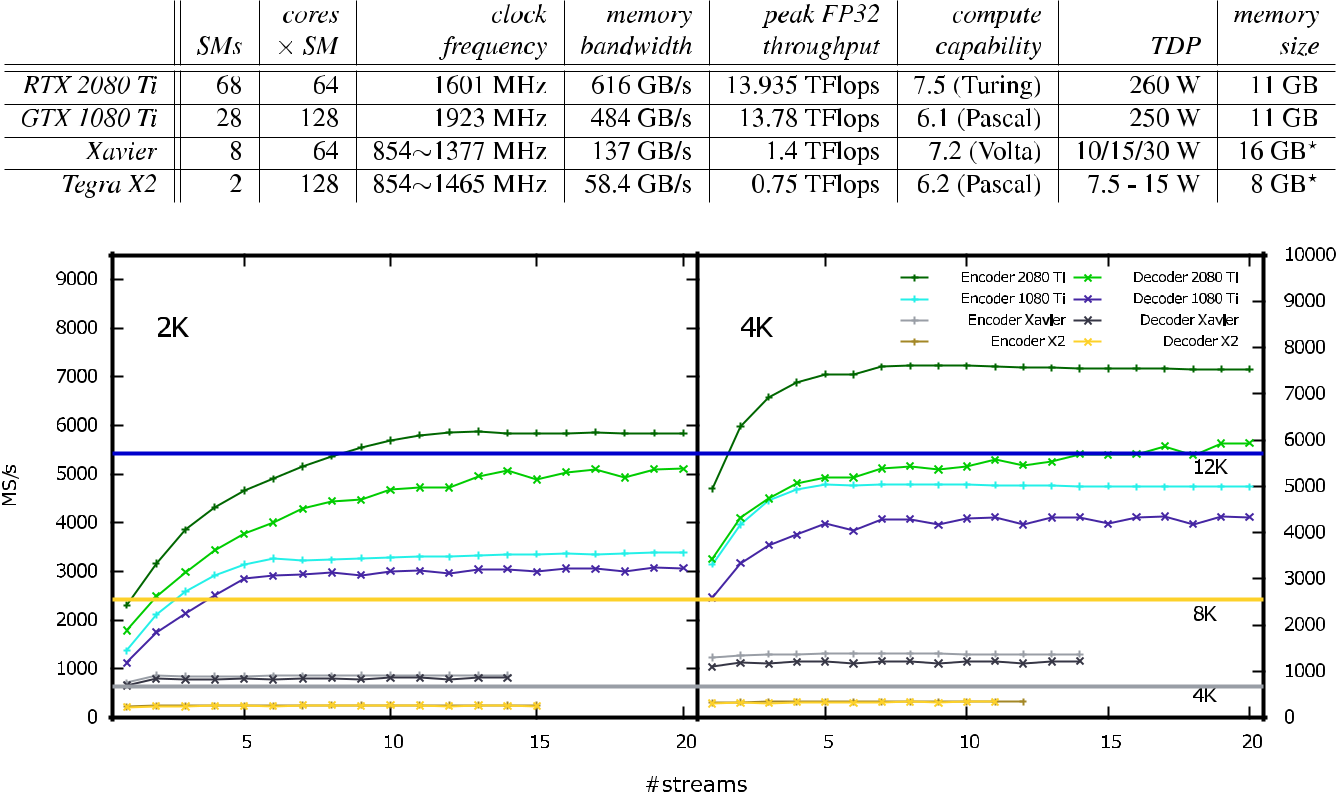
For comparison purposes, the following experiments also provide the throughput achieved with the HEVC implemen-tation developed by Nvidia [64], which is executed with the RTX 2080 Ti and the GTX 1080 Ti. This codec runs in the GPU employing in-chip support and dedicated hardwired components. The parameters for HEVC are: rate control with constant quantization 1-51 (0) for lossy (lossless), inter-frame coding with GOPD32, and high performance mode. This con guration achieves maximum throughput in our tests. We note that HEVC is not supported in Jetson GPUs.

The data set employed in the experiments is a 2-minute segment of the movie ‘‘Star Wars: The Last Jedi,’’ at a reso-lution 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

VOLUME 8, 2020

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

**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.

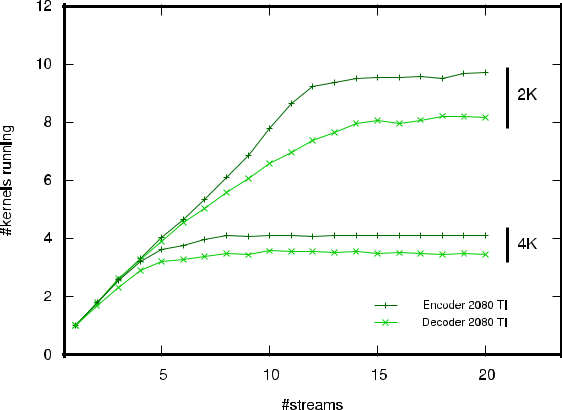


**FIGURE 4.** Analysis of the throughput achieved by the proposed codec when coding 2K (left) and 4K (right) video using different number of executionstreams, for lossy compression at maximum quality.

performance achieved. In general, the size of this data set is suf ciently large to ll the resources of the GPU. Larger data sets achieve similar results as those reported below. In all results, the execution time is measured without con-sidering the I/O time spent to read/write the les from/to the disk since that would affect results signi cantly depend-ing 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 rst test evaluates the throughput achieved by the proposed codec with the four GPUs when using a differ-ent 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](#page10) reports the results achieved. Again, this g-ure depicts with horizontal lines the throughput needed to yield 4K, 8K, and 12K video compression in real time, assum-ing 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 bene t as much of using multiple streams because they have fewer SMs, so their resources are mostly lled with a single execution stream.

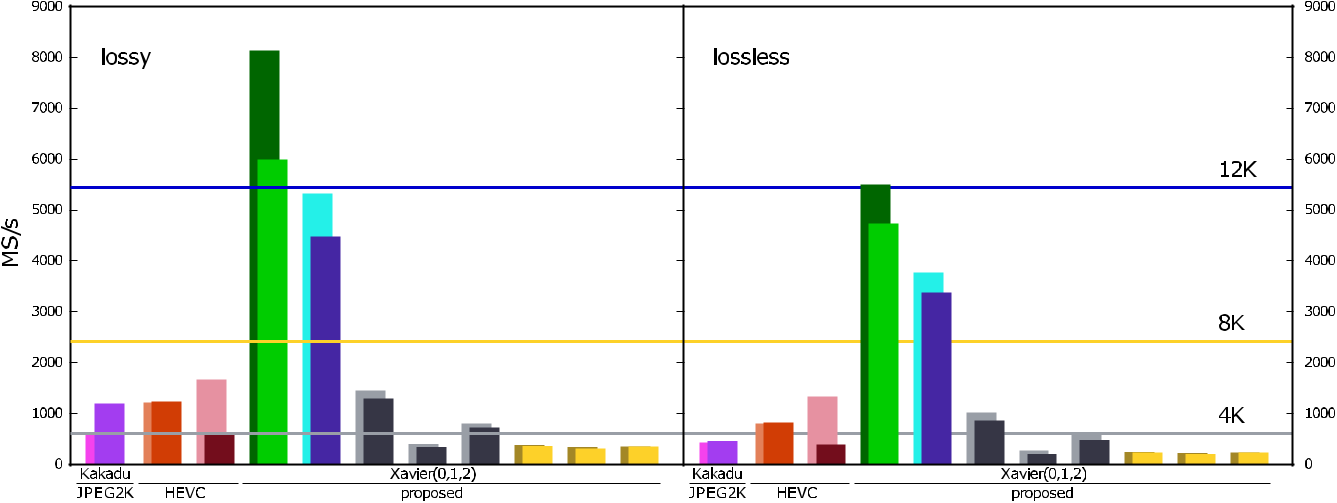
**FIGURE 5.** Evaluation of the average number kernels executed per unit oftime depending on the number of streams employed.



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

|  |  |
| --- | --- |
| VOLUME 8, 2020 | 68483 |

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000



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

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 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](#page10) 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 ll 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 gure. 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 ll its resources.

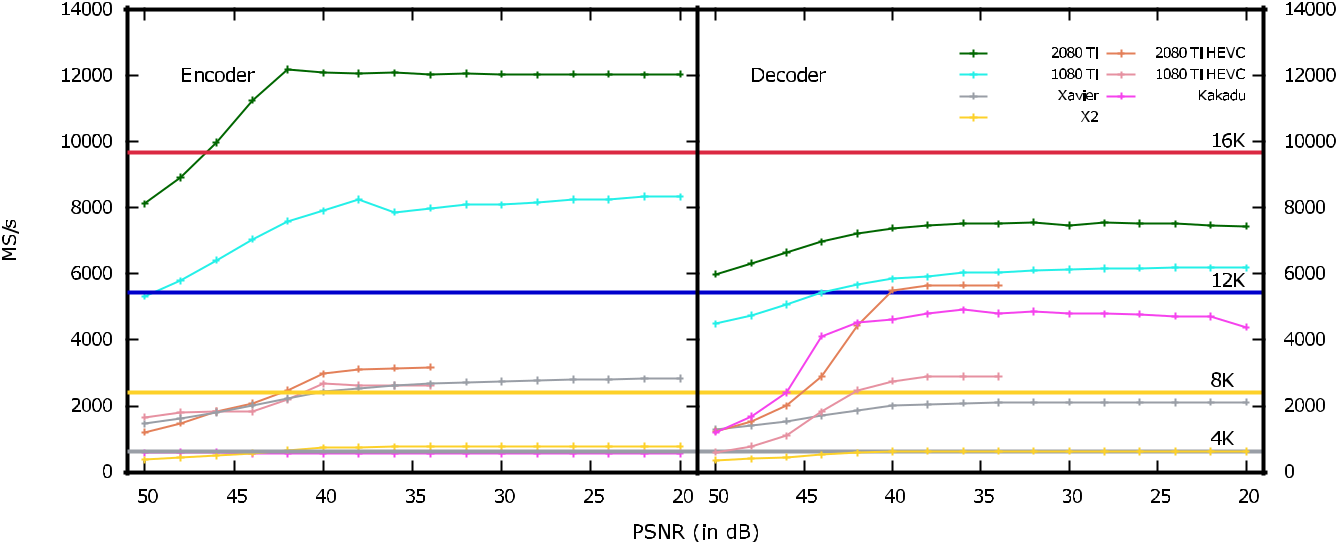
Figure [6](#page11) reports the throughput achieved by the proposed codec with the four GPUs, Kakadu, and HEVC when coding 4K video in lossy and lossless mode. For lossy compres-sion, the average image quality yield for all codecs is about 50 dB. At this level of quality, distortion is not perceptible by the human eye. Each codec has a pair of columns. The rst 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 gure. Results for the Xavier and Tegra X2 are reported when using three power modes, namely, maximum (0), min-imum (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 com-pression. In all codecs, the performance in lossless mode is

slightly lower than that achieved in lossy since more data are processed, generating larger compressed les. 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 sig-ni cantly 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. Surpris-ingly, 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](#page12) 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 quan-tization parameter *Q* in our codec, and similarly in HEVC and Kakadu. As seen in the gure, reducing the quality has

68484 VOLUME 8, 2020

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000



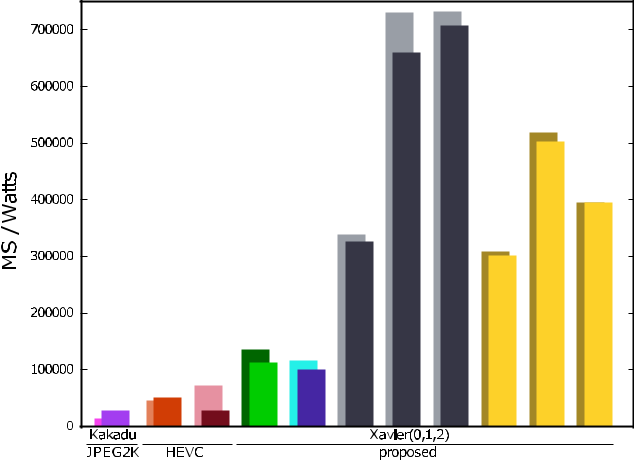
**FIGURE 7.** Throughput evaluation for lossy compression of 4K video at different quality levels. Results are for the proposed codec except when indicated.

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 perfor-mance 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 bene ts from this architectural improvement since it employs registers extensively. The highest speedups reported in Figure [6](#page11) are achieved by the HEVC decoder, which increases the throughput almost 6 .

Power consumption is nowadays an important aspect due to the advent of mobile devices. Figure [8](#page12) evaluates the power consumption of our codec, HEVC, and Kakadu when coding 4K video at 50 dB, like in Figure [6.](#page11) 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](#page12) suggest that the proposed codec is the most ef cient in terms of power 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 ef ciency. The proposed codec is more ef cient 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

VOLUME 8, 2020

**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).

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. CONCLUSION**

Faster and less power-hungry image and video codecs are cur-rently needed in multiple scenarios. Typically, high through-put 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 ne-grain parallelism to fully exploit the SIMD archi-tecture of GPUs. This paper introduces an image/video codec

68485

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

based on the JPEG2000 standard. All stages of the coding pipeline have been devised to extract ne-grain parallelism. All stages are compliant with the standard except for the core algorithm called bitplane and arithmetic coding. 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 stan-dard. The throughput of the resulting architecture when exe-cuted 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 Standard 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, no. 9, pp. 1098 1101, 1952.
4. *Information Technology JPEG 2000 Image Coding System Part 1: Core Coding System*, ISO/IEC Standard 15 444-1, Dec. 2000.
5. I. Daubechies, *Ten Lectures on Wavelets*. Philadelphia, PA, USA: 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 Ef ciency Video Coding Standard*, ITU Standard H.265, 2013.
8. D. S. Taubman and M. W. Marcellin, *JPEG2000: Image Compression*

*Fundamentals, Standards and Practice*. Norwell, MA, USA: Academic,2002.

1. A. Descampe, F.-O. Devaux, G. Rouvroy, J.-D. Legat, J.-J. Quisquater, and B. Macq, ‘‘A exible hardware JPEG 2000 decoder for digital cinema,’’ *IEEE Trans. Circuits Syst. Video Technol.*, vol. 16, no. 11, pp. 1397 1410,Nov. 2006.
2. T. Bruylants, A. Munteanu, and P. Schelkens, ‘‘Wavelet based volumetric medical image compression,’’ *Signal Process., Image Commun.*, vol. 31, pp. 112 133, Feb. 2015.
3. 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.
4. 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.
5. H.-C. Fang, Y.-W. Chang, T.-C. Wang, C.-J. Lian, and L.-G. Chen, ‘‘Par-allel embedded block coding architecture for JPEG 2000,’’ *IEEE Trans.* *Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1086 1097, Sep. 2005.
6. G. Pastuszak, ‘‘A high-performance architecture for embedded block cod-ing in JPEG 2000,’’ *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1182 1191, Sep. 2005.
7. 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.
8. 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.
9. 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.
10. K. Mei, N. Zheng, C. Huang, Y. Liu, and Q. Zeng, ‘‘VLSI design of a high-speed and area-ef cient JPEG2000 encoder,’’ *IEEE Trans. Circuits* *Syst. Video Technol.*, vol. 17, no. 8, pp. 1065 1078, Aug. 2007.

68486

1. 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.
2. M. Dyer, S. Nooshabadi, and D. Taubman, ‘‘Design and analysis of sys-tem on a chip encoder for JPEG2000,’’ *IEEE Trans. Circuits Syst. Video* *Technol.*, vol. 19, no. 2, pp. 215 225, Feb. 2009.
3. K. Sarawadekar and S. Banerjee, ‘‘An ef cient 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.
4. K. Liu, E. Belyaev, and Y. Li, ‘‘A high throughput JPEG2000 entropy decoding unit architecture,’’ *J. Signal Process. Syst.*, vol. 91, no. 8,
   1. 899 913, Aug. 2019.
5. Nvidia. (Dec. 2018). *GPU vs CPU Theoretical GFLOP/s*. [Online].

Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ graphics/ oating-point-operations-per-second.png

1. M. Hopf and T. Ertl, ‘‘Hardware accelerated wavelet transformations,’’ in *Proc. IEEE Symp. Vis.*, May 2000, pp. 93 103.
2. T.-T. Wong, C.-S. Leung, P.-A. Heng, and J. Wang, ‘‘Discrete wavelet transform on consumer-level graphics hardware,’’ *IEEE Trans. Multime-dia*, vol. 9, no. 3, pp. 668 673, Apr. 2007.
3. C. Tenllado, J. Setoain, M. Prieto, L. Piæuel, and F. Tirado, ‘‘Parallel implementation of the 2D discrete wavelet transform on graphics process-ing units: Filter bank versus lifting,’’ *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 3, pp. 299 310, Mar. 2008.
4. J. Matela, ‘‘GPU-based DWT acceleration for JPEG2000,’’ in *Proc.* *Annu. Doctoral Workshop Math. Eng. Methods Comput. Sci.*, Nov. 2009,
   1. 136 143.
5. J. Franco, G. Bernabe, J. Fernandez, and M. E. Acacio, ‘‘A parallel imple-mentation of the 2D wavelet transform using CUDA,’’ in *Proc. IEEE Int.* *Conf. Parallel, Distrib. Netw.-Based Process.*, Feb. 2009, pp. 111 118.
6. W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink, ‘‘Accelerating wavelet lifting on graphics hardware using CUDA,’’ *IEEE Trans. Parallel* *Distrib. Syst.*, vol. 22, no. 1, pp. 132 146, Jan. 2011.
7. 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. Int. Conf. Comput. Math. Methods Sci. Eng.*, Jan. 2012, pp. 151 158.
8. V. Galiano, O. López, M. P. Malumbres, and H. Migallón, ‘‘Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs,’’ *J. Supercomput.*, vol. 64, no. 1, pp. 4 16, Apr. 2013.
9. 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.
10. 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.
11. S. Datla and N. S. Gidijala, ‘‘Parallelizing motion JPEG 2000 with CUDA,’’ in *Proc. IEEE Int. Conf. Comput. Elect. Eng.*, Dec. 2009,
    1. 630 634.
12. R. Le, I. R. Bahar, and J. L. Mundy, ‘‘A novel parallel tier-1 coder for JPEG2000 using GPUs,’’ in *Proc. IEEE 9th Symp. Appl. Speci c Process.* *(SASP)*, Jun. 2011, pp. 129 136.
13. J. Matela, V. Rusnak, and P. Holub, ‘‘Ef cient JPEG2000 EBCOT context modeling for massively parallel architectures,’’ in *Proc. Data Compress.* *Conf.*, Mar. 2011, pp. 423 432.
14. F. Wei, Q. Cui, and Y. Li, ‘‘Fine-granular parallel EBCOT and optimization with CUDA for digital cinema image compression,’’ in *Proc. IEEE Int.* *Conf. Multimedia Expo*, Jul. 2012, pp. 1051 1054.
15. M. Ciznicki, K. Kurowski, and A. Plaza, ‘‘Graphics processing unit imple-mentation of JPEG2000 for hyperspectral image compression,’’ *SPIE J.* *Appl. Remote Sens.*, vol. 6, pp. 1 14, Jan. 2012.
16. J.-W. Lee, B. Kim, and K.-S. Yoon, ‘‘CUDA-based JPEG2000 encoding scheme,’’ in *Proc. IEEE Int. Conf. Adv. Commun. Technol.*, Feb. 2014,
    1. 671 674.
17. X. Wu, Y. Li, K. Liu, K. Wang, and L. Wang, ‘‘Massive parallel implemen-tation of JPEG2000 decoding algorithm with multi-GPUs,’’ *Proc. SPIE*, vol. 9124, pp. 1 6, May 2014.
18. 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.
19. F. Auli-Llinas, ‘‘Context-adaptive binary arithmetic coding with xed-length codewords,’’ *IEEE Trans. Multimedia*, vol. 17, no. 8,
    1. 1385 1390, Aug. 2015.

VOLUME 8, 2020

C. de Cea-Dominguez *et al.*: GPU-Oriented Architecture for an End-to-End Image/Video Codec Based on JPEG2000

1. F. Auli-Llinas, P. Enfedaque, J. C. Moure, I. Blanes, and V. Sanchez, ‘‘Strategy of microscopic parallelism for bitplane image coding,’’ in *Proc.* *Data Compress. Conf.*, Apr. 2015, pp. 163 172.
2. F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, ‘‘Bitplane image coding with parallel coef cient processing,’’ *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 209 219, Jan. 2016.
3. P. Enfedaque, F. Auli-Llinas, and J. C. Moure, ‘‘GPU implementation of bitplane coding with parallel coef cient processing for high performance image compression,’’ *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8,
   1. 2272 2284, Aug. 2017.
4. D. Taubman, A. Naman, and R. Mathew, ‘‘High throughput block coding in the HTJ2K compression standard,’’ in *Proc. IEEE Int. Conf. Image* *Process. (ICIP)*, Sep. 2019, pp. 1079 1083.
5. A. T. Naman and D. Taubman, ‘‘Decoding high-throughput JPEG2000 (HTJ2K) on a GPU,’’ in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2019,
   1. 1084 1088.
6. C. D. Cea-Dominguez, P. Enfedaque, J. C. Moure, J. Bartrina-Rapesta, and

F. Auli-Llina, ‘‘High throughput image codec for high-resolution satel-lite images,’’ in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, 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. Amsterdam, The Netherlands: IOS Press, Apr. 2020, pp. 83 92.

1. Nvidia. (Jan. 2018). *Warp Level Primitives*. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/
2. (Jun. 2019). *Nvidia Tesla V100 GPU Architecture*. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
3. F. N. Iandola, D. Shef eld, M. J. Anderson, P. M. Phothilimthana, and K. Keutzer, ‘‘Communication-minimizing 2D convolution in GPU regis-ters,’’ in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2013, pp. 2116 2120.
4. A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, ‘‘Thread-cooperative, bit-parallel computation of levenshtein distance on GPU,’’ in *Proc. 28th ACM Int. Conf. Supercomput. (ICS)*, 2014, pp. 103 112.
5. A. Chacon, S. M. Sola, A. Espinosa, P. Ribeca, and J. C. Moure, ‘‘FM-index on GPU: A cooperative scheme to reduce memory footprint,’’ in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl.*, Aug. 2014, pp. 1 9.
6. W. Sweldens, ‘‘The lifting scheme: A construction of second generation wavelets,’’ *SIAM J. Math. Anal.*, vol. 29, no. 2, pp. 511 546, Mar. 1998.
7. 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.
8. Nvidia. (Dec. 2018). *CUB Framework*. [Online]. Available: https://nvlabs. github.io/cub/
9. (May 2019). *Jetson SDK*. [Online]. Available: https://developer.nvidia. com/embedded-computing
10. D. Taubman. (Feb. 2020). *Kakadu Software*. [Online]. Available: http://www.kakadusoftware.com
11. University of Stuttgart. (Jan. 2020). *CuJ2K*. [Online]. Available: http://cuj2k.sourceforge.net/
12. Poznan Supercomputing and Networking Center. (Feb. 2020). *GPUJ2K*. [Online]. Available: http://apps.man.poznan.pl/trac/jpeg2k/wiki
13. Comprimato. (Feb. 2020). *Comprimato JPEG2000@GPU*. [Online]. Available: http://www.comprimato.com
14. Fastvideo LLC. (Feb. 2020). *CUDA-JPEG2K*. [Online]. Available: https://www.fastcompression.com/products/gpu-jpeg2000.htm
15. Nvidia. (Dec. 2018). *HEVC SDK*. [Online]. Available: https://developer. nvidia.com/nvidia-video-codec-sdk

CARLOS DE CEA-DOMINGUEZ received the B.E. and Ms.E. degrees in computer science from Universidad de León, in 2012. He is currently pursuing the Ph.D. degree with the Universitat Autònoma de Barcelona. From 2012 to 2017, he worked as a Technological Consultant. His cur-rent research interests include high-performance computing, image and video coding, and parallel architectures.



VOLUME 8, 2020

JUAN C. MOURE-LOPEZ received the B.Sc. degree in computer science and the Ph.D. degree in computer architecture from the Universitat Autònoma de Barcelona (UAB). Since 2008, he has been an Associate Professor with the Computer Architecture and Operating Systems Department, UAB, where he teaches computer architecture and parallel programming. He has par-ticipated in several European and Spanish projects related to high performance computing. His cur-



rent research interest focuses on the usage of massively parallel architectures and the application of performance engineering techniques to open research problems in bioinformatics, signal processing, and computer vision. He is reviewer for various magazines and symposiums and has authored numerous articles in journals and conferences.

JOAN BARTRINA-RAPESTA received the B.Sc., B.E., M.S., and Ph.D. degrees in computer sci-ence from the Universitat Autònoma de Barcelona (UAB), Spain, in 2002, 2004, 2006, and 2009, respectively, and the M.S. and Ph.D. degrees (Hons.) from UAB. Since 2012, he has been an Associate Professor with the Department of Infor-mation and Communications Engineering, UAB. He was awarded with a Doctoral Fellowship from UAB. He participates as a reviewer for magazines



and symposiums, has (co)authored numerous articles in journals and confer-ences, and has guided Ph.D. students. His research interests include a wide range of image coding topics, computing, and transmission.

FRANCESC AUL˝-LLINÀS (Senior Member, IEEE) received the B.E. (Hons.) and Ph.D. *(cum laude)* degrees in computer science fromthe Universitat Autònoma de Barcelona (UAB), in 2002 and 2006, respectively. From 2002 to 2015, he was consecutively funded in competitive fellowships, including a Ramón y Cajal Grant that was awarded with the Intensi cation Young Inves-tigator (i3) Certi cate. During this time, he carried out two postdoctoral research stages with profes-



sors David Taubman and Michael Marcellin. Since 2016, he has been an Associate Professor (with the Full Professor certi cate) with the Department of Information and Communications Engineering, UAB. He developed and maintains BOI codec, a JPEG2000 implementation that is used in research and professional environments. Also, he is a reviewer for magazines and symposiums, has (co)authored numerous articles in journals and confer-ences, and has guided several Ph.D. students. He has participated and super-vised various projects funded by the Spanish government and the European Union. His research interests lie in the area of image and video coding, computing, and transmission. In 2013, he received a distinguished R-Letter given by the IEEE Communications Society for a paper co-authored with Michael Marcellin.



68487