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

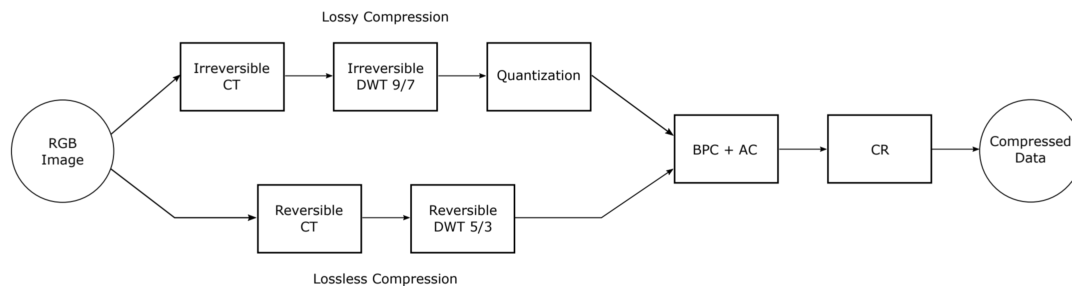

FIGURE 1: JPEG2000 coding pipeline.

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.

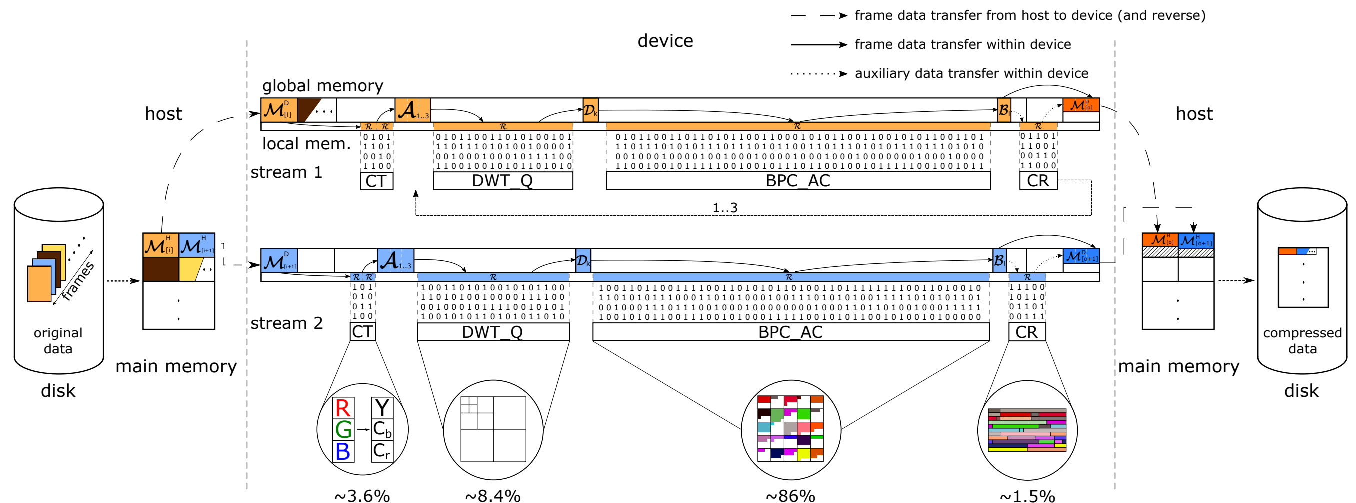

FIGURE 2: Illustration of the codec architecture when using 2 CUDA streams. The cycle of the data is as follows. First, frame data (individually identified by color) are read from disk to a RAM buffer. Then the data are managed by a stream in the GPU. Within the device the data are transferred from global 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
  t_1  { 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
  S_j  { 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
  t_2  {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(·)) 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(·) 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(·) 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($\cdot$) 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($\cdot$) 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($\cdot$) kernel is similar to that achieved by ICT($\cdot$) and DWT_Q($\cdot$). 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($\cdot$) 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

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

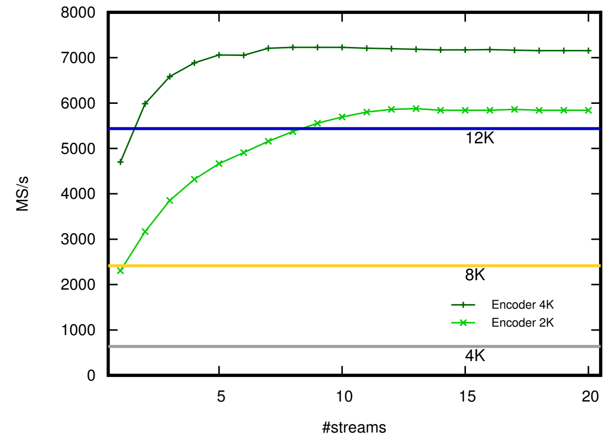

FIGURE 3: Analysis of the throughput achieved by the proposed codec when encoding 2K and 4K video using different number of execution streams, for the RTX 2080 Ti.

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**($\mathcal{A}_{1..3}$)

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.

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($\cdot$) 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

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.

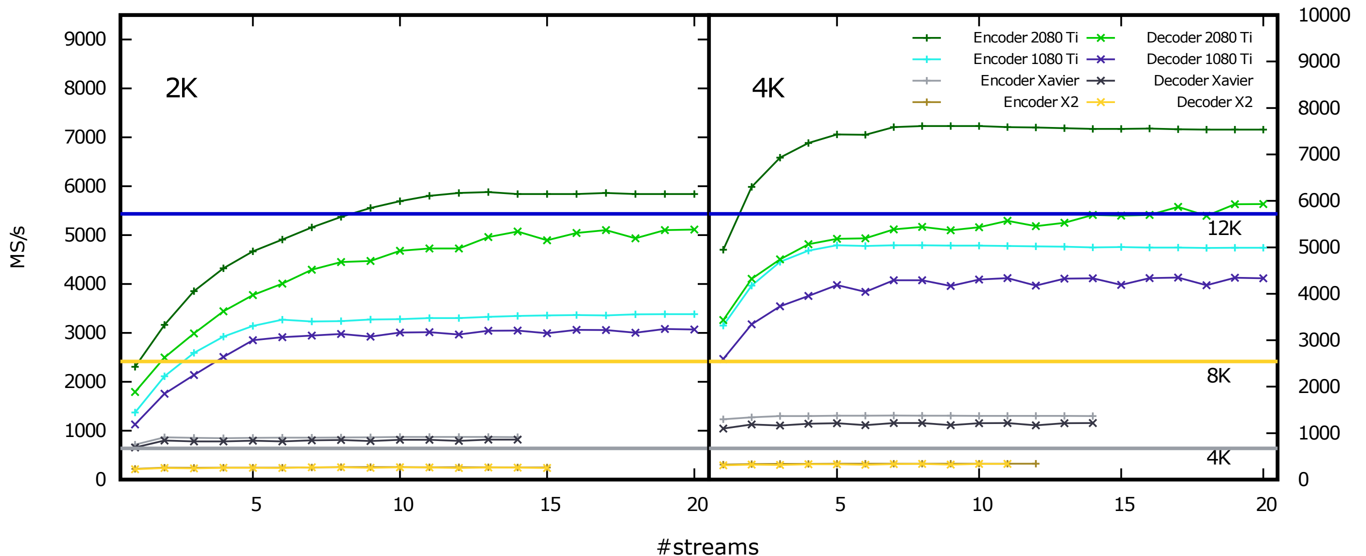

FIGURE 4: Analysis of the throughput achieved by the proposed codec when coding 2K (left) and 4K (right) video using different number of execution streams, for lossy compression at maximum quality.

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

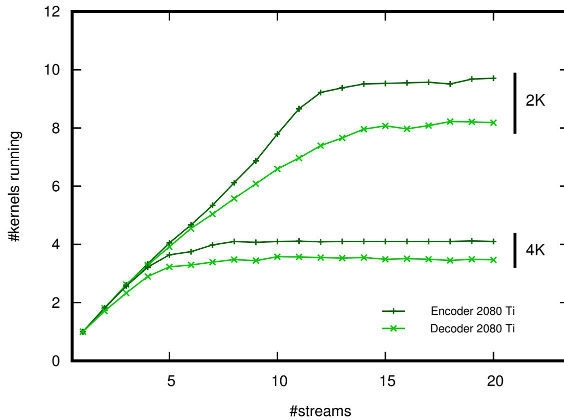

FIGURE 5: Evaluation of the average number kernels executed per unit of time depending on the number of streams employed.

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

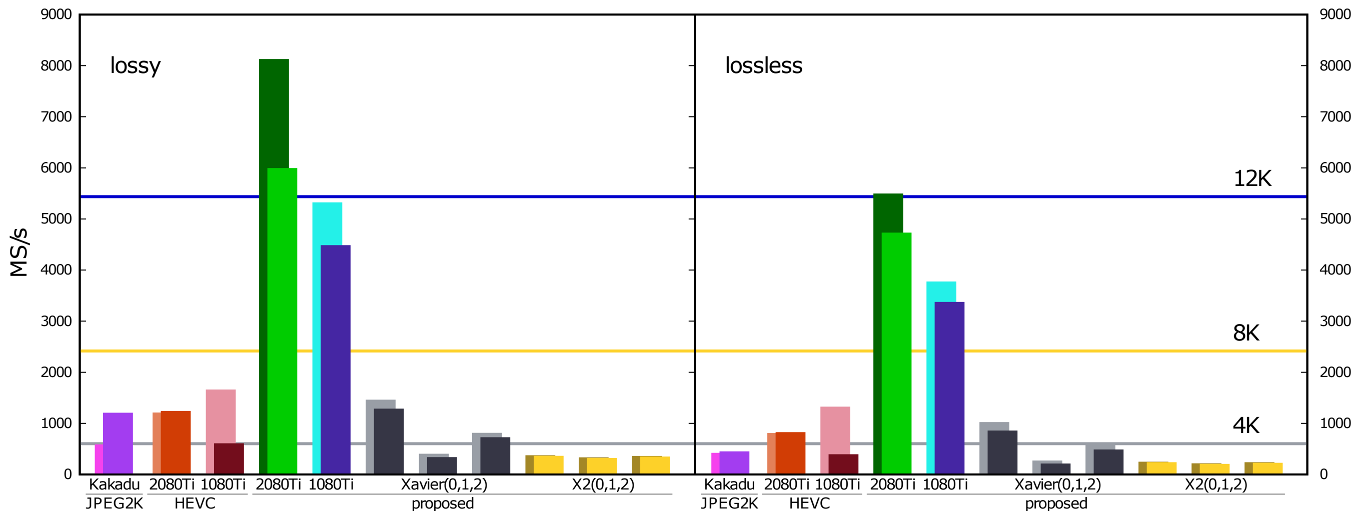

FIGURE 6: Throughput evaluation for lossy (with highest image quality) and lossless compression of 4K video, for all codecs and GPUs. Each pair of columns reports the results for the encoder (back) and decoder (front).

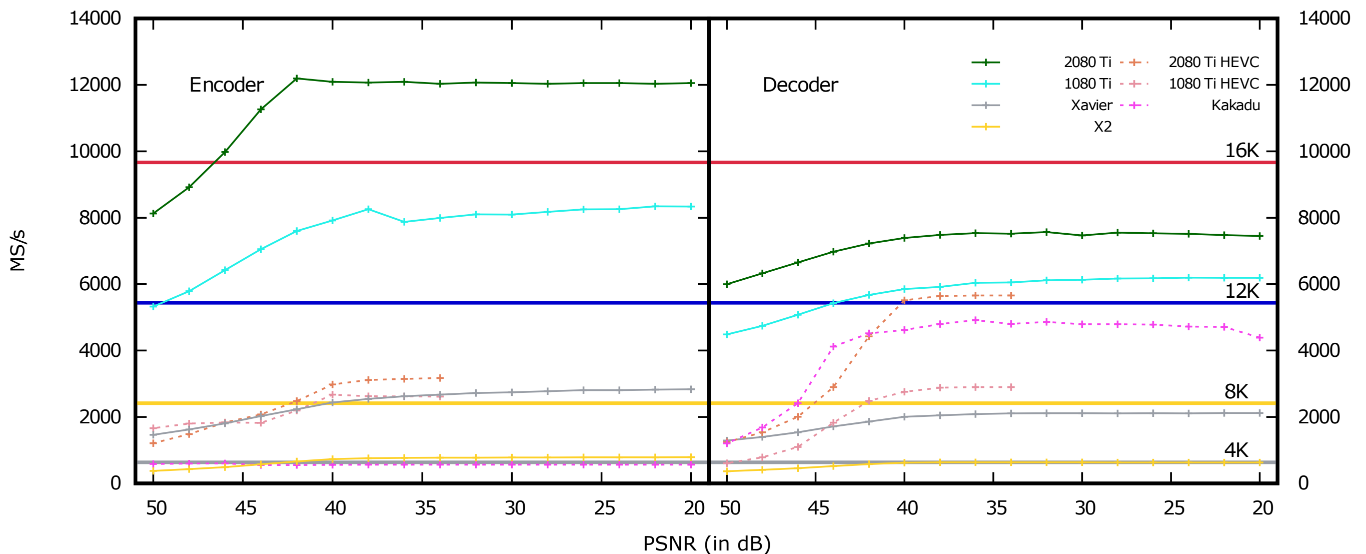

FIGURE 7: Throughput evaluation for lossy compression of 4K video at different quality levels. Results are for the proposed codec except when indicated.

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.

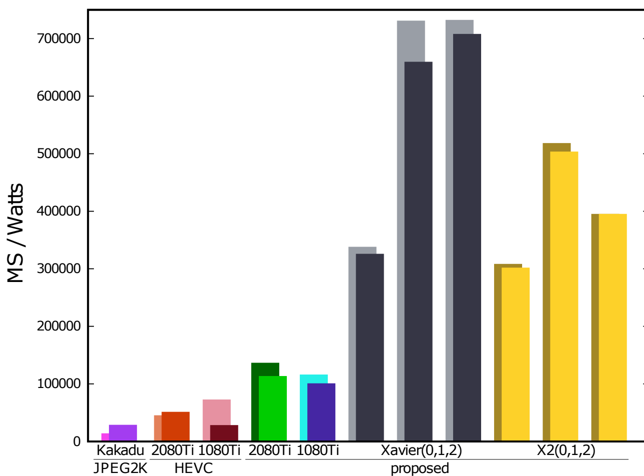

FIGURE 8: Power consumption evaluation when encoding 4K video at 50 dB. Each pair of columns reports the results for the encoder (back) and decoder (front).

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