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

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

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

Abstract

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

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

1. Introduction

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

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

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

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

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

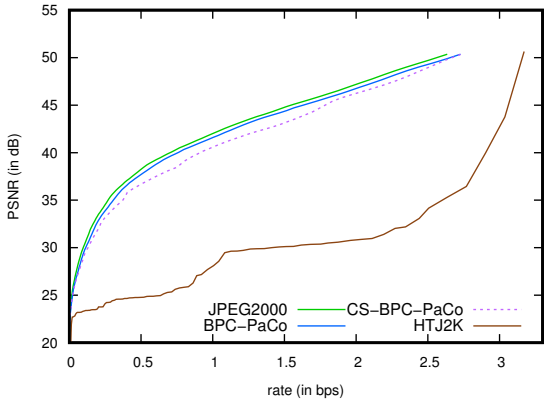

Figure 1: Evaluation of rate-distortion performance for JPEG2000, HTJ2K, BPC-PaCo and CS BPC-PaCo (with $K = 0.5$) when transmitting the color image "Portrait" at 100 different rates.

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

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

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

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

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

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

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

2. Background

2.1. GPU architecture

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

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

As Fig. 2 depicts, the memory architecture of a GPU has three levels: global memory, shared memory, and registers. The global memory is located in the device RAM or DRAM, has a size in the order of GBs, and is accessible by all SMs. This memory has high latency but relatively high bandwidth, so data transfers are to be carried out in a coalesced way (i.e., using consecutive memory positions) to maximize performance. The shared memory has a size in the order of MBs, has low

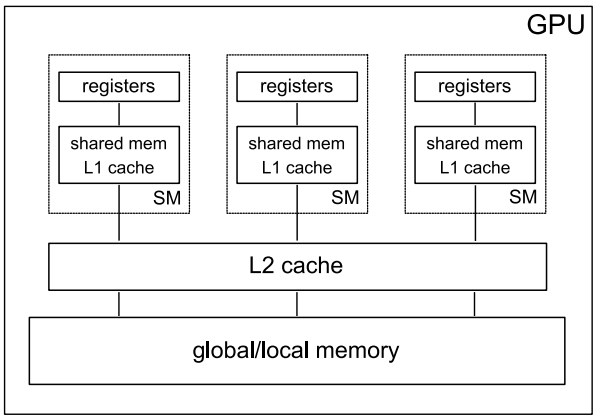

Figure 2: Memory hierarchy of a Nvidia GPU.

latency and higher bandwidth, and can be accessed by all threads of a block. Each SM has an individual memory bank for this memory. The registers have very fast access, very high bandwidth, and a size of typically 256 KB. When the registers can not hold all the data required by the application, some data are temporarily moved to a reserved space in the device memory, the so-called local memory. This is called register spilling. It significantly affects the application's performance because transfers from/to the device memory render threads in an idle state due to the memory latency.

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

2.2. JPEG2000 architecture

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

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

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

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

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

3. Proposed method

3.1. Codec architecture

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

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

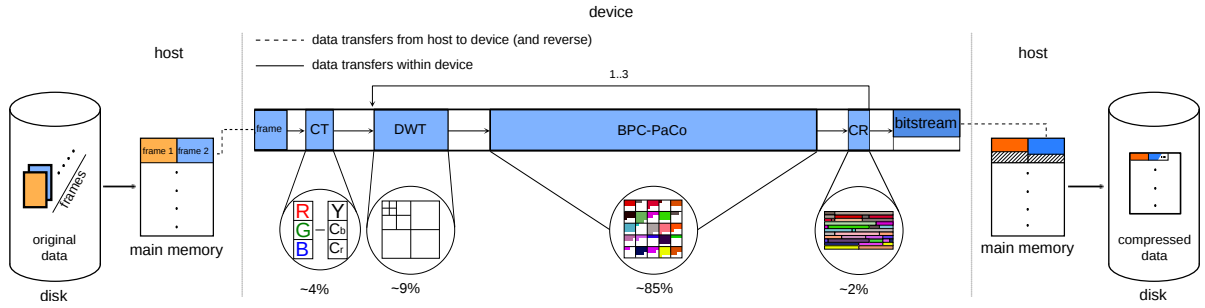

Figure 3: Illustration of the codec architecture when employing a single stream of execution in the GPU.

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

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

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

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

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

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

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

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

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

3.2. Complexity Scalable BPC-PaCo

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

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

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

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

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

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

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

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

(a) (b)

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

3.3. Implementation

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

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

Algorithm 1 CS BPC-PaCo

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

```
1:  for j ∈ [M − 1, N] do
2:      SignificancePass()
3:      RefinementPass()
4:  end for
5:  for y ∈ [0, numRows − 1] do
6:      for x ∈ [t · 2, t · 2 + 1] do
7:          c ← φ_sig(υ_{y,x}, N − 1)
8:          for j ∈ [N − 1, 0] do
9:              if Φ(υ_{y,x}, j + 1) = 0 then
10:                 ACencode(b_j, P_u[j][c], t)
11:                 if b_j = 1 then
12:                     ACencode(d, P'_u[j][φ_sign(υ_{y,x}, N − 1)], t)
13:                 end if
14:             else
15:                 ACencode(b_j, P''_u[j][0], t)
16:             end if
17:         end for
18:     end for
19: end for
```

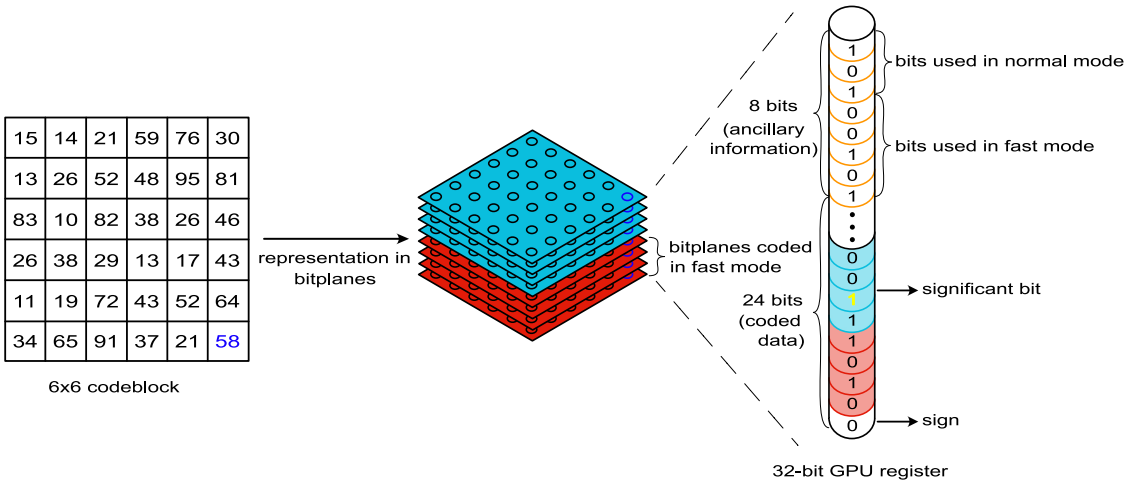

Figure 5: Illustration of a codeblock and the bit-allocation strategy in the 32-bit registers of the GPU employed by CS BPC-PaCo.

in red are coded with the fast mode.

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

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

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

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

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

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

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

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

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

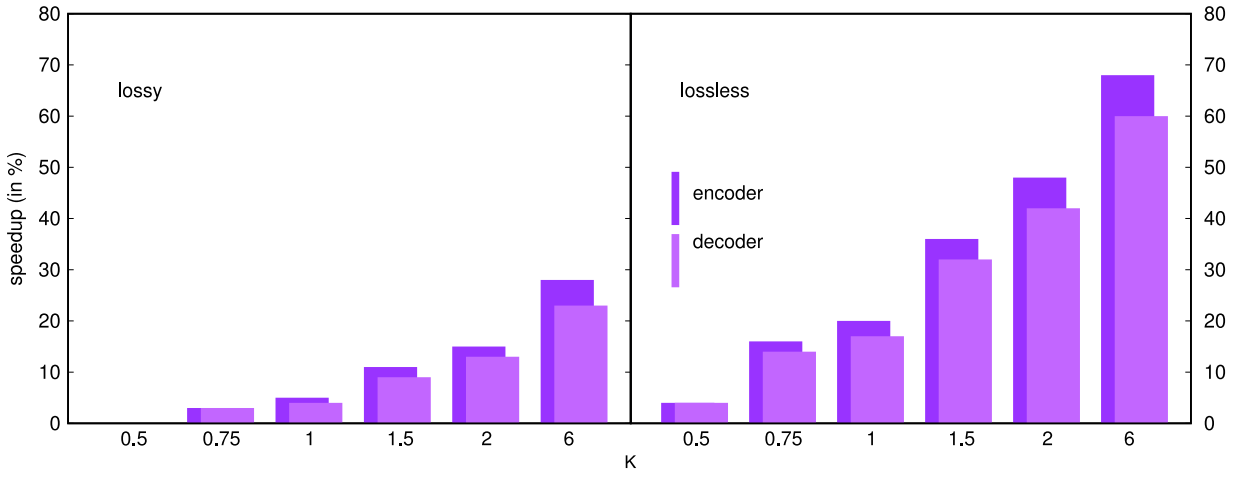

Figure 6: Evaluation of the throughput gain achieved by CS BPC-PaCo with respect to BPC-PaCo for different Ks.

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

4. Experimental Results

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

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

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

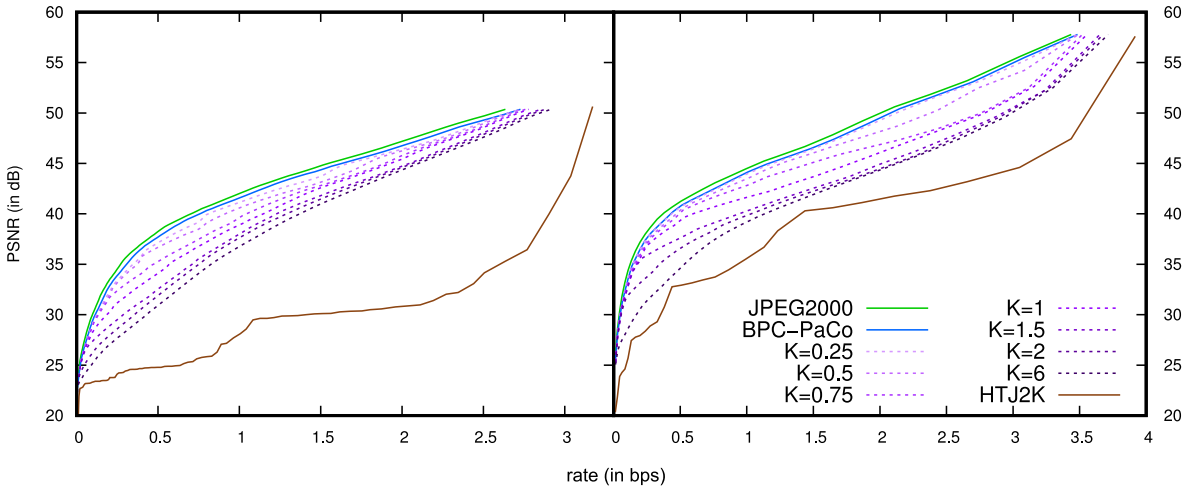

Figure 7: Evaluation of rate-distortion performance for JPEG2000, HTJ2K, BPC-PaCo and CS BPC-PaCo (with different Ks) when transmitting an image at 100 different rates (left) and a video sequence at 30 different rates (right).

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

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

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

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

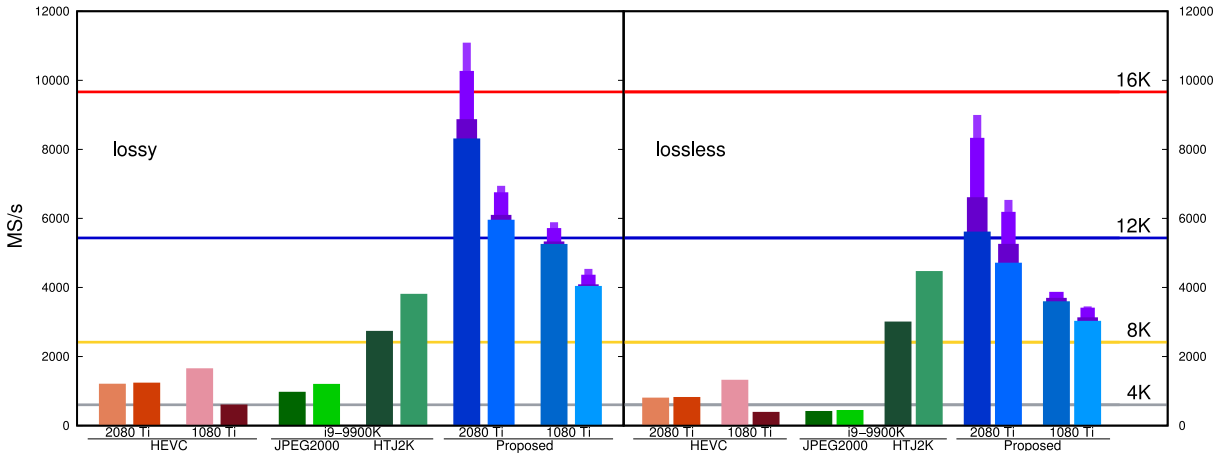

Figure 8: Throughput evaluation of lossy and lossless video compression for all codecs and GPUs/CPUs.

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

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

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

5. Conclusions

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

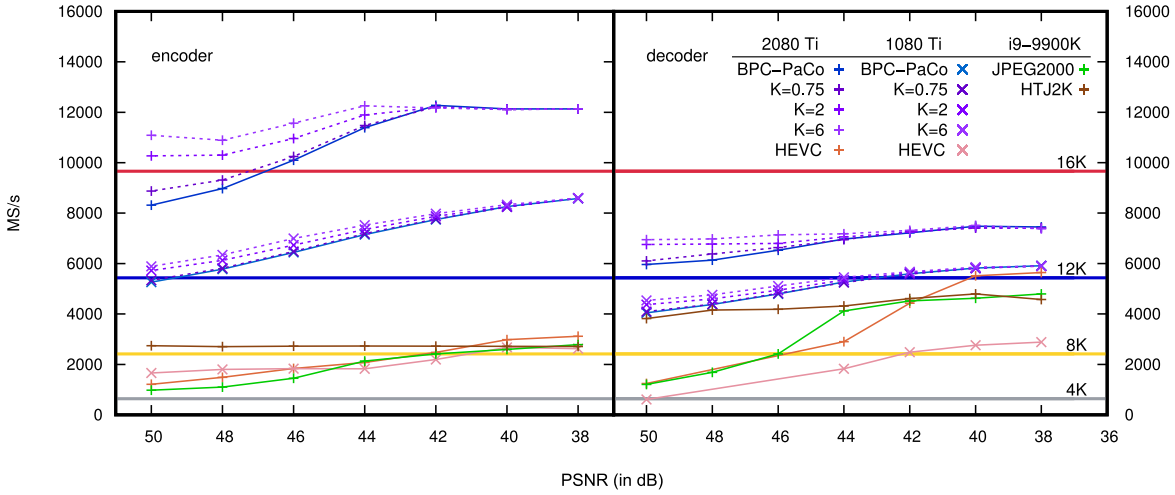

Figure 9: Throughput evaluation for lossy compression of video at different quality levels. Results are for BPC-PaCo and CS BPC-PaCo except when indicated.

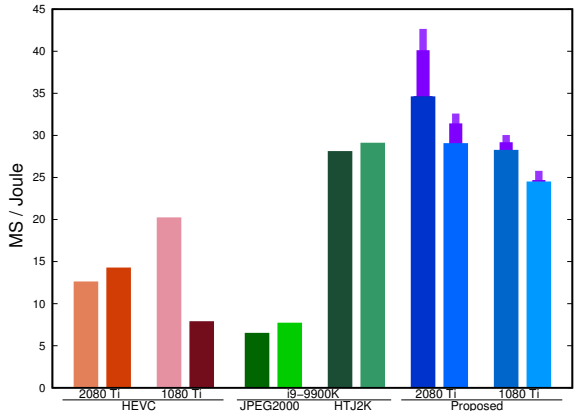

Figure 10: Evaluation of energy consumption for lossy video compression for all codecs and GPUs/CPUs.

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

Acknowledgment

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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