GPU Implementation of Bitplane Coding with Parallel Coefficient Processing for High Performance Image Compression

Pablo Enfedaque, Francesc Aulí-Llinàs, *Senior Member, IEEE*, and Juan C. Moure

***Abstract*—The fast compression of images is a requisite in many applications like TV production, teleconferencing, or digital cinema. Many of the algorithms employed in current image compression standards are inherently sequential. High performance implementations of such algorithms often require specialized hardware like field integrated gate arrays. Graphics Processing Units (GPUs) do not commonly achieve high performance on these algorithms because they do not exhibit fine-grain parallelism. Our previous work introduced a new core algorithm for wavelet-based image coding systems. It is tailored for massive parallel architectures. It is called bitplane coding with parallel coefficient processing (BPC-PaCo). This paper introduces the first high performance, GPU-based implementation of BPC-PaCo. A detailed analysis of the algorithm aids its implementation in the GPU. The main insights behind the proposed codec are an efficient thread-to-data mapping, a smart memory management, and the use of efficient cooperation mechanisms to enable inter-thread communication. Experimental results indicate that the proposed implementation matches the requirements for high resolution (4K) digital cinema in real time, yielding speedups of 30× with respect to the fastest implementations of current compression standards. Also, a power consumption evaluation shows that our implementation consumes 40× less energy for equivalent performance than state-of-the-art methods.**

***Index Terms*—Image coding, SIMD computing, Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA).**

I. Introduction

Arguably, the main goal of image coding systems is to represent the samples (i.e., pixels) of an image with the minimum number of bits possible, thereby achieving compression. In addition to compression, current image codecs provide other features such as region of interest coding, error resilience, or capabilities for interactive transmission. The high compression efficiency and novel features of the latest image compression schemes, however, come at the expense of computationally demanding algorithms. As a result, current codecs often require specialized hardware to meet the (real-time) demands of applications such as digital cinema, surveillance, or medical imaging. In such scenarios, the use of Graphics Processing Units (GPUs) might be a solution to accelerate the coding process in a cost- and energy-efficient way.

Pablo Enfedaque and Francesc Aulí-Llinàs are with the Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain (phone: +34 935811861; fax: +34 935813443; e-mail: {pablo | fauli}@deic.uab.cat). Juan C. Moure is with the Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain (e-mail: juancarlos.moure@uab.es). This work has been partially supported by the Universitat Autònoma de Barcelona, by the Spanish Government (MINECO), by FEDER, and by the Catalan Government, under Grants 472-02-2/2012, TIN2015-71126-R, TIN2014-53234-C2-1-R, and 2014SGR-691.

Copyright (c) 2017 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.

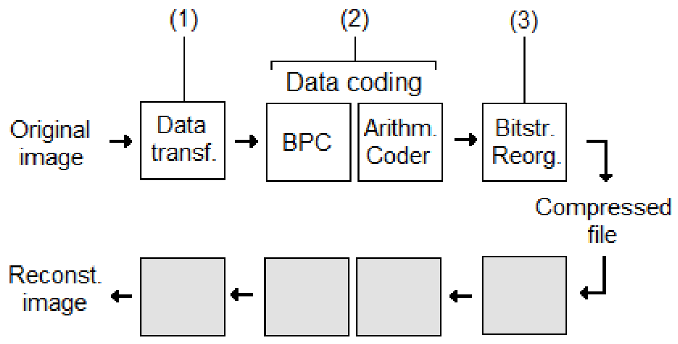

Fig. 1: Main stages of the JPEG2000 coding pipeline: (1) data transformation, (2) data coding through bitplane coding (BPC) and arithmetic coding, and (3) bitstream reorganization. The decoding process (depicted in gray) carries out the inverse operations.

A representative coding scheme of modern image codecs is that of the JPEG2000 standard (ISO/IEC 15444-1), which provides excellent coding performance and advanced features [1]. The coding pipeline of JPEG2000 is structured in three main stages (see Fig. 1): data transformation, data coding, and bitstream reorganization. The data transformation stage removes the spatial redundancy of the image through the discrete wavelet transform [2]. Data coding codes the transformed samples, called coefficients, by means of exploiting visual redundancy. Bitplane coding and arithmetic coding are two efficient techniques to do so. The bitplane coder repetitively scans the coefficients in a bit-by-bit fashion. These bits are fed to the arithmetic coder, which produces the bitstream. The last stage of the coding pipeline codes auxiliary information and reorganizes the data. The techniques employed in the data coding stage are fundamental to achieve compression, though they need abundant computational resources. A common codec approximately spends 80% of the total coding time in this stage, whereas the first and the last stage take 15% and 5% of the execution time, respectively [3].

Many Single Instruction, Multiple Data (SIMD) implementations of image codecs on GPU architectures are devised to accelerate the coding process [4]–[16]. Their aim is to

TABLE I: Execution time (in seconds) of Kakadu, CUJ2K, and JPEG2K when coding 3 images of different size in lossless mode. Kakadu is executed in a Core Duo E8400 at 3 GHz, whereas the GPU implementations are executed in a GeForce GTX 480. These results are reproduced from [17] with the permission of the authors.

	image samples ($\times 2^{20}$)		
	12	28	39
Kakadu	1.65	7.05	8.3
CUJ2K	1.25	2.95	3.9
JPEG2K	0.72	2.35	2.75

extract massive data-level parallelism in the first and second stage of the coding scheme to achieve higher computational performance than implementations optimized for Central Processing Units (CPUs). The operations carried out in the data transformation stage are well-fitted to SIMD computing. Many GPU implementations of such transforms can be found in the literature [4]–[6], [11], [13]–[16]. To implement the bitplane coder and the arithmetic coder efficiently in SIMD architectures is a much greater challenge. The problem is to extract fine-grained data-level parallelism from algorithms that were not originally devised for SIMD. Due to this difficulty, current GPU implementations of bitplane coding engines [7], [9], [10], [12] are unable to fully extract the computational power of the GPU architectures. Table I shows a comparison presented in [17] reporting the execution time of JPEG2000 codecs optimized for CPUs and GPUs. Kakadu [18] is among the fastest CPU implementations of the standard, whereas CUJ2K [19] and JPEG2K [17] are the most competitive open-source implementations for GPUs. The GPU employed in this comparison has a peak performance approximately 10 times superior to that of the employed CPU. Even so, the GPU implementations achieve (at most) a $3\times$ speedup with respect to Kakadu.

Our previous work [20] introduced a bitplane coding engine that unlocks the data dependencies of traditional algorithms. In that work, the proposed bitplane coding with parallel coefficient processing (BPC-PaCo) is introduced in the framework of JPEG2000 without sacrificing any feature of the coding system. The bitstream generated by BPC-PaCo is not compliant with the standard since the parallel coefficient processing modifies the way that the bitstream is constructed. Also, it slightly penalizes coding performance, though in general the efficiency loss is less than 2%. Our previous paper focused on the image coding perspective of the method, analyzing its features and coding performance. A preliminary version of the implementation of the encoder in a GPU was introduced in [21]. Herein, we introduce the optimized GPU implementation for both the encoder and decoder, and extend our previous work with additional experimental results, a detailed analysis of the implementation, and a revised writing. The comparison of the proposed implementation with the most efficient CPU and GPU implementations of JPEG2000 suggests that BPC-PaCo is approximately 30 times faster and 40 times more power-efficient than the best JPEG2000 implementations. This increase in performance is because BPC-PaCo can exploit the resources of the GPU more efficiently than the conventional bitplane coding engine of JPEG2000. The experimental assessment considers the level of divergence, parallelism, and instructions executed of the codecs evaluated.

The rest of the paper is structured as follows. Section II provides a general background of bitplane coding and GPU architectures. Section III reviews BPC-PaCo and Section IV describes the proposed implementation. Section V provides experimental results. The last section summarizes this work.

II. Background

A. *Overview of bitplane image coding*

Fig. 2 depicts an overview of the bitplane coding process of JPEG2000. The image on the left represents the coefficients produced by the data transformation stage. Then, the coding system conceptually partitions the image into rectangular tiles that contain a predefined number of coefficients. These tiles are referred to as codeblocks. Although the size of the codeblock can vary, in general codeblocks of 64×64 are preferred since they provide competitive coding efficiency. The bitplane coding process is applied independently in each codeblock, producing a bitstream per codeblock. All the bitstreams are then re-organized in the third stage of the coding pipeline to produce the final file.

The main insight of bitplane coding is to scan the coefficients in planes of bits. Bitplane j is defined as the collection of bits in the j^{th} position of the binary representation of the coefficients (excluding the sign). Bitplane coding engines code the bits of the coefficients from bitplane $M-1$ to 0, with M representing a sufficient number of bits to represent all coefficients. This is depicted in the middle image of Fig. 2. The bits of the bitplane are not processed sequentially. Instead, the bits that are more likely to reduce the distortion of the image are emitted to the output bitstream first. This is implemented in practice via the so-called coding passes [22]. JPEG2000 employs three coding passes called significance propagation pass (SPP), magnitude refinement pass (MRP), and cleanup pass (CP). Each coding pass processes the bits of a set of coefficients. The procedure ensures that all coefficients are processed once in each bitplane by one –and only one– coding pass.

Let us define the significance state of coefficient x as $\mathcal{S}(x)=1$ when the first non-zero bit of its binary representation has already been emitted, and as $\mathcal{S}(x)=0$ otherwise. When $\mathcal{S}(x)=1$ the coefficient is called significant. The SPP processes the bits of non-significant coefficients that have some immediate neighbor that is significant. This aims at emitting first the bits of those coefficients that are more likely to become significant in the current bitplane. These bits reduce the most the distortion of the image. When a coefficient is significant, its sign bit is emitted just after its significance bit. The MRP is applied after the SPP, processing the bits of coefficients that were significant in previous bitplanes. The CP is the last coding pass applied in each bitplane, processing the bits of non-significant coefficients that were *not* emitted in the

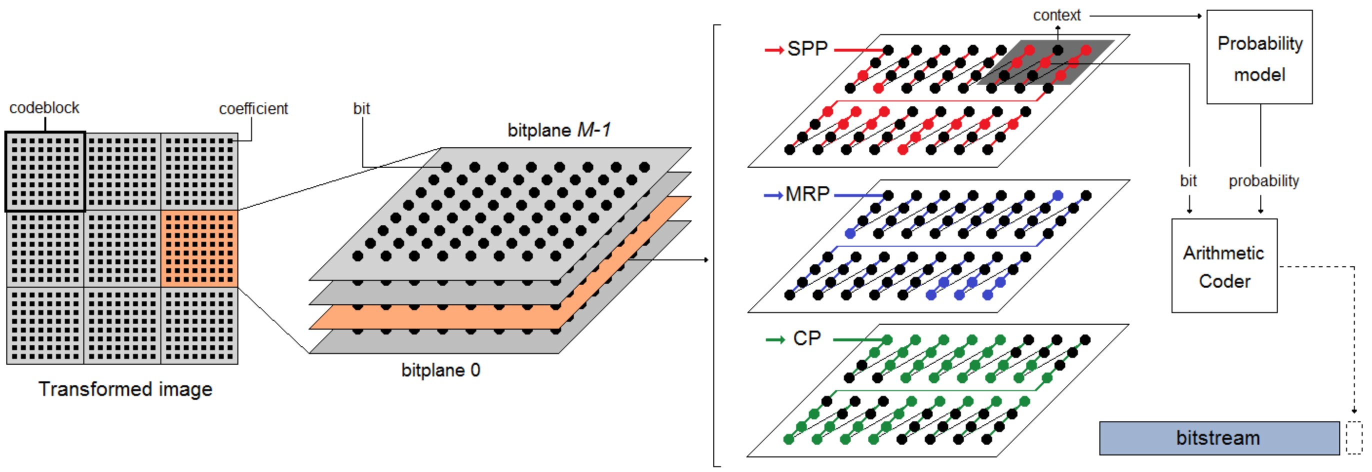

Fig. 2: Overview of the JPEG2000 bitplane coding process. Codeblocks containing 8×8 coefficients are depicted for simplicity. The coefficients processed in the coding passes SPP, MRP, and CP are depicted in red, blue, and green, respectively.

SPP. As seen in the right image of Fig. 2, the three coding passes utilize the same scanning order, though each processes only the coefficients that fulfill the aforementioned conditions. The scanning order of JPEG2000 partitions the codeblock in sets of four rows, visiting the coefficient in each set from the top-left to the bottom-right coefficient.

Two important mechanisms of bitplane coding strategies are the context formation and the probability model. The context of a coefficient is determined via the significance state, or the sign, of its eight immediate neighbors (see Fig. 2, right-top corner). The function that computes the context considers the number and position of the significant neighbors and their signs (when already coded). The probability model then employs this context to adaptively adjust the probabilities of the bits emitted in each context. The bit and the probability are fed to an arithmetic coder, generating a compressed representation of the data.

Arithmetic coding is an entropy coding technique extensively employed in the coding field due to its high efficiency [23]. From an algorithmic point of view, an arithmetic coder divides an arithmetic interval in two subintervals whose sizes are proportional to the estimated probability of the coded bit. The subinterval corresponding to the value of the bit coded is chosen. Then the same procedure is repeated for following bits. The transmission of any number within the final interval, referred to as codeword, permits the decoding of the original bits. As it is traditionally formulated, it renders the coding algorithm as a causal system in which each bit can not be coded without processing all the previous bits of that codeblock.

B. *Overview of Compute Unified Device Architecture (CUDA)*

GPUs are massively parallel devices containing multiple throughput-oriented SIMD units called streaming multiprocessors (SM). Modern GPUs have up to 24 SMs and each SM can execute multiple 32-wide SIMD instructions simultaneously. The CUDA programming model defines a computation hierarchy formed by threads, warps, and thread blocks. A CUDA thread represents a single lane of a SIMD instruction. Warps are sets of 32 threads that advance their execution in a lockstep synchronous way as single SIMD operations. Control flow divergence among the threads of the same warp results in the sequential execution of the divergent paths and the increase of the total number of instructions executed, so it should be avoided. Thread blocks group warps, and each one is assigned and run until completion in a specific SM. Warps inside the same block are executed asynchronously but they can cooperate sharing data via on-chip fast memories and can synchronize using explicit barrier instructions.

From a programming point of view, the memory hierarchy of GPUs is organized in 1) a space of local memory that is private to each thread, 2) a space of shared memory that is private to each thread block, and 3) a space of global memory that is public to all threads. From a microarchitecture point of view, the amount of local memory reserved for each thread is located in the registers or in the off-chip memory, depending on the available resources. GPUs also have two levels of cache. In recent CUDA architectures, local memory located in the off-chip memory has exclusive use of the level-1 (L1) cache.

The SM activity is defined as the time that each SM is active during the execution of a CUDA program, also called kernel. It is commonly expressed as an average percentage. A SM is considered active if it has, at least, a warp assigned for execution. A CUDA program may not occupy all the SMs of the GPU. This may happen when the kernel does not launch sufficient thread blocks. Also, high workload imbalances caused by different execution times of the thread blocks may reduce the SM activity and affect the overall performance. The occupancy of active SMs is defined as the percentage of active warps relative to the maximum supported by the SM. The theoretical occupancy of a kernel is the maximum occupancy when considering the static execution configuration. It can be limited by the amount of shared memory and registers assigned to each thread block. The achieved occupancy may be lower than the theoretical when the warps have high variability in their execution times or when they need to synchronize

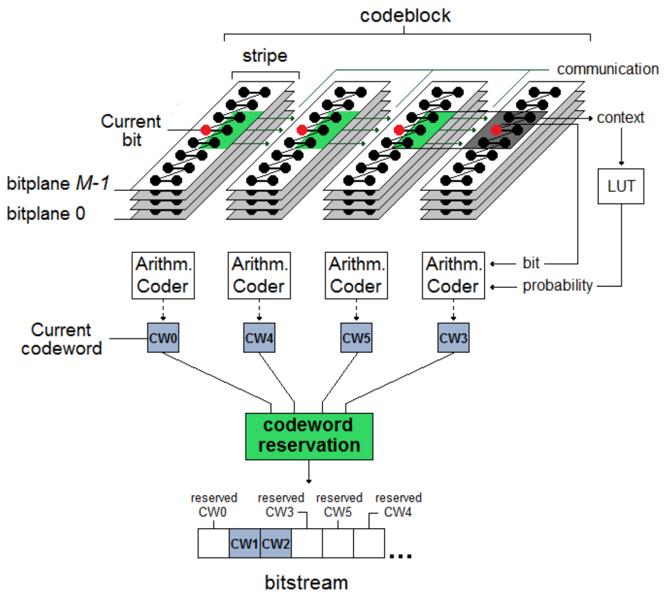

Fig. 3: Illustration of the coding strategy of BPC-PaCo. The currently coded coefficients are depicted in red and the cooperation between stripes is depicted in green. The codewords generated by the arithmetic coders are depicted in blue.

frequently.

III. Review of BPC-PaCo

Traditional implementations of bitplane coding engines code the codeblocks independently and (possibly) in parallel. Unfortunately, this parallelism is not fine-grained enough and the parallel control flows are too divergent to employ the resources of the GPU efficiently. BPC-PaCo redefines the mechanisms of traditional bitplane coding engines to promote SIMD parallelism within the codeblock. The main idea behind BPC-PaCo is to partition the codeblock in N vertical stripes, each containing two columns of coefficients, that can be coded in parallel. The coding process within the codeblock advances its execution in a lock-step synchronous fashion for all stripes, collaborating to share some data when necessary. The scanning order, coding passes, context formation, probability model, and arithmetic coding are redevised to permit such a parallel processing.

Fig. 3 depicts the coding strategy of BPC-PaCo. The scanning order in each stripe visits the coefficients from the top to the bottom row and from the left to the right column. The context formation for the SPP and CP sums the significance state of the eight neighbors of the coefficient, i.e., $C(x) = \sum_{i=1}^{8} \mathcal{S}(n_i)$, with n_i denoting the immediate neighbors of x. The sign of the coefficient is emitted, when necessary, employing another set of contexts. These contexts are computed via the sign (when already coded) of the top, right, bottom, and left neighbors, employing simple comparisons and logical operations. The bits emitted in the MRP are all coded with a single context. The employed context formation approach has been devised to reduce both computational load and control-flow divergence. More details on its underlying ideas can be found in [24], [25]. As shown in Fig. 3, the computation of the contexts needs that stripes of the same codeblock communicate among them.

Traditional probability models adjust the probabilities of the emitted bits as more data are coded. The adaptation is sequential. There are no simple solutions to update the probabilities in parallel. To adapt the probabilities for each stripe independently is not effective either because too little data are coded, resulting in poor coding performance [22]. BPC-PaCo adopts an approach in which the probabilities are not adjusted depending on the coded data but they are precomputed off-line using a training set of images. These stationary probabilities are stored in a lookup table (LUT) that is known by the encoder and the decoder (so it is not included in the codestream). Such a model exploits the fact that the transformed coefficients have similar statistical behavior for similar images [26]. Once the LUT is constructed, it can be employed to code any image with similar features as those in the training set. Evidently, different sensors (such as those in the medical or remote sensing fields) produce images with very different statistical behaviors, so individual LUTs need to be computed for each [26].

The probability of a bit to be 0 or 1 is extracted from the LUT using its context and bitplane. The bit and its probability are fed to an arithmetic coder. BPC-PaCo employs N independent arithmetic coders, one for each stripe of the codeblock. This allows the synchronous parallel coding of the bits emitted in each stripe. The main difficulty with such a procedure is that the codewords produced by the N coders must be combined in the bitstream in an optimized order so that the bitstream can be partially transmitted and decoded (see below).

Besides using multiple arithmetic coders, BPC-PaCo employs a coder that is simpler than that employed in traditional systems. The main difference is that it generates multiple fixed-length codewords instead of a single and long codeword that has to be processed in small segments [23]. The fixed-length codeword arithmetic coder is adopted by BPC-PaCo because it reduces computational complexity and control flow-divergence. Fig. 3 depicts the codewords generated by each coder below each stripe. At the beginning, the arithmetic interval of each coder is as large as the codeword. As more bits are coded, the interval is reduced. When the minimum size is reached, the codeword is exhausted and so it is dispatched in a reserved position of the bitstream. Then, a new position is reserved at the end of the bitstream for the to-be-coded codeword. The reservation of this space needs cooperation among stripes.

As described in [20], BPC-PaCo uses three coding passes. We note that the more coding passes employed, the more divergence that occurs in SIMD computing. This is because the bit of the currently visited coefficient in each stripe may, or may not, need to be emitted. The threads coding the stripes in which the bit is not emitted are idle while the others perform the required operations. Three coding passes achieve competitive efficiency [22], though the method can also use two passes without penalizing coding performance significantly. This can be seen in Fig. 4, which reports the coding performance achieved by BPC-PaCo when using two or

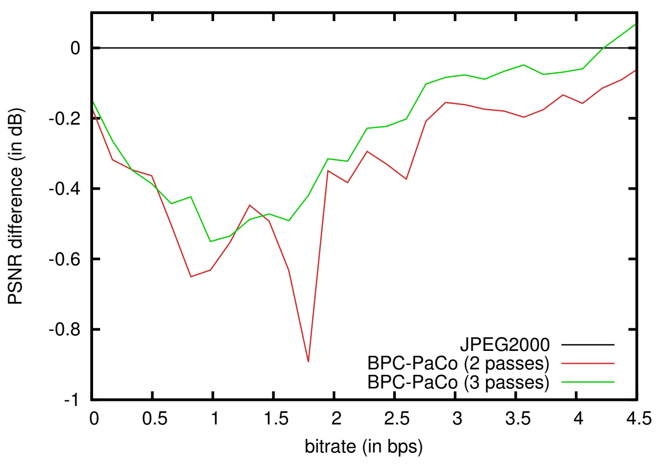

Fig. 4: Coding performance comparison between JPEG2000 and BPC-PaCo with two and three coding passes.

three coding passes with respect to the performance achieved by JPEG2000. The vertical axis of the figure is the peak signal to noise ratio (PSNR) difference between BPC-PaCo and JPEG2000. PSNR is a common metric to evaluate the quality of the image. In general, differences of 1 dB in PSNR are considered visually relevant, whereas differences below 1 dB are not commonly perceived by the human eye. The horizontal axis of the figure is the bitrate, expressed in bits per sample (bps). A low bitrate indicates a small size of the final bitstream. As seen in the figure, BPC-PaCo with three coding passes achieves a PSNR that is, at most, 0.5 dB below that of JPEG2000. BPC-PaCo with two coding passes achieves a slightly inferior coding performance, with peaks of at most 0.9 dB below that of JPEG2000. These results are generated for an image of the corpus employed in the experimental section. The results hold for other images of this and other corpora.

IV. Analysis and Implementation

This section details the implementation of BPC-PaCo in CUDA. We consider the two- and three-pass version of the algorithm since the use of only two passes helps to accelerate the coding process. This requires two versions for the encoder and two for the decoder. The first part of this section overviews the common aspects to all versions of the codec, namely, work decomposition, memory management, and cooperation mechanisms. Then, the particular algorithms for the two versions of the encoder are presented. The decoder is discussed in the last part.

A. Overview

Our implementation decomposes the work following the intrinsic data partitioning of the algorithm. More precisely, a CUDA warp is assigned to each codeblock, and each thread of the warp processes a stripe within the codeblock. This thread-to-data mapping exposes fine-grained parallelism and avoids the use of explicit synchronization instructions among threads. Since there are not data dependencies among codeblocks, the thread block size can be adjusted without algorithmic restrictions.

TABLE II: Occupancy and execution time achieved when limiting the number of registers per thread from 16 to 128. Results achieved with a GTX TITAN X when coding a 5120×5120 GeoEye satellite image. The codeblock size is 64×64.

registers	2 coding passes		3 coding passes	
per thread	occupancy	time (in ms)	occupancy	time (in ms)
16	89%	32.81	89%	45.66
24	89%	17.97	89%	25.41
32	89%	17.07	89%	23.81
40	67%	19.10	66%	27.37
48	56%	21.10	54%	30.45
56	51%	22.44	48%	32.58
64	45%	24.35	42%	35.16
72	40%	26.57	37%	38.07
128	23%	39.23	22%	56.27

Key to maximize performance is the memory management. The two larger and most frequently accessed data structures, both in the encoder and the decoder, are the coefficients of the codeblock and its bitstream. The most efficient strategy is to store the coefficients in the local memory, making use of the rapid on-chip registers, whereas the bitstream is stored in the global memory. With a codeblock size of 64×64 and 32 threads per warp, each thread must hold 128 coefficients in its local memory plus other temporary variables. This large amount of local memory per thread demands a compromise. There is a well-known tradeoff between the registers employed per thread, the amount of register spilling traffic that is redirected to the device memory, and the achieved occupancy. The higher the number of registers per thread, the lower the number of warps that can be executed simultaneously, and also the lower the amount of local data accesses that must be spilled to the device memory. Table II shows the occupancy and the execution time achieved when limiting the number of registers per thread at compilation time from 16 to 128. Results for the two versions of the encoder are reported. The results indicate that the lowest execution time is achieved when using 32 registers per thread. In our implementation the amount of data spilling appears to be moderate and it does not significantly degrade the performance thanks to the high thread-level parallelism achieved. These results also hold for the decoder and for other images.

The bitstream of each codeblock is stored in the global memory to save on-chip resources. As previously explained, the bitstream contains individual codewords. While a codeword is still in use, it is temporarily stored in the local memory. Each codeword is used to code a variable number of symbols. The different probabilities of the symbols causes that codewords from different stripes are exhausted at different instants. Therefore, when a codeword is exhausted, it is written into the bitstream (commonly) in a non-coalesced way. This means that to write codewords in the bitstream is an expensive operation. Fortunately, this task is not carried out frequently because many symbols are coded before a codeword is exhausted. Our experience indicates that to use the global memory to store the bitstream offers optimal performance for the encoder. Once a codeword is written, it is not further used, so the latency of the memory transaction is hidden due to the high arithmetic intensity of the algorithm. The case for the decoder is slightly different and is discussed below.

In addition to these data structures, BPC-PaCo utilizes two ancillary structures, namely, a set of LUTs that store the static probabilities for the coded symbols, and a status map that keeps auxiliary information for each coefficient. The LUTs are read-only and are heavily accessed, so they are put in the constant memory of the device. The status map is employed to know whether a coefficient is significant or not, and in what coding pass it has to be coded. This information requires 2 or 3 bits per coefficient depending on whether 2 or 3 coding passes are employed, respectively. These bits are stored in the most significant bits of the coefficients since the number of operative bits is always below 29 (i.e., $M < 29$) and its representation employs 32 bits. We remark that this status map could be avoided by means of explicitly computing the coefficient status before coding each symbol. This computation is trivial when using 2 coding passes, but it has a significant impact in execution time when 3 coding passes are employed. Our implementation uses such a status map for both versions of the codec.

The cooperation of threads within the same warp is needed for two purposes: 1) to compute the context of each coefficient, and 2) to reserve the space of the codewords in the bitstream. The former operation is implemented via shuffle instructions using the coefficients of the stripes stored in the local memory. A shuffle instruction fetches a value from the local memory of another thread within the warp. This instruction was introduced in Kepler architectures and its latency is the same as that of accessing a register. The communication of threads in older architectures needs to use a small buffer in the shared memory [16]. The reservation of the codewords space is implemented via vote and pop-count instructions. The vote instruction allows all threads within the warp to evaluate a condition, leaving the result in a register visible to all of them. The pop-count instruction sums all non-zero bits of a register. In addition to these two instructions, the reservation of space for codewords utilizes a shared pointer to the last free position of the bitstream, which is stored in the shared memory and accessible for all threads. Further details of this cooperation mechanism are described in Algorithm 4. We recall that *no* special synchronization instructions are needed due to the inherent synchronization of the threads within the warp.

B. Encoder with 2 passes

Algorithm 1 details the CUDA kernel implemented for the two-pass encoder. The parameters of the algorithm are thread identifier T, top-left codeblock coordinates (with respect to the image) X and Y, and codeblock height H. First (in lines 2-8), the coefficients of the stripe are read from the global memory, which is denoted by $\mathcal{G}$, and stored in the local memory, which is denoted by $\mathcal{L}$. The status map, referred to as $\mathcal{S}$, is initialized in the same loop. As seen in the algorithm, both bits of $\mathcal{S}$ are initialized to 0. When the coefficient becomes

Algorithm 1 - BPC-PaCo Encoder (with 2 coding passes)
Parameters: thread $T \in [0, 31]$, codeblock coordinates X, Y, and codeblock height H

```
1: allocate $\mathcal{L}[H][2]$ in local memory
2: for $y \in \{0, ..., H-1\}$ do
3:   for $x \in \{0, 1\}$ do
4:     $\mathcal{L}[y][x] \leftarrow \mathcal{G}[Y+y][X+T*2+x]$
5:     $\mathcal{S}[y][x][0] \leftarrow 0$
6:     $\mathcal{S}[y][x][1] \leftarrow 0$
7:   end for
8: end for
9: for $j \in \{M-1, ..., 0\}$ do
10:   for $y \in \{0, ..., H-1\}$ do
11:     for $x \in \{0, 1\}$ do
12:       $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3 \leftarrow$ getNeighbors$(T, y, x)$
13:       if $\mathcal{S}[y][x][0] = 0$ then
14:         $\mathcal{C}_{sig} \leftarrow$ significanceContext$(\mathcal{S}, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, y, x)$
15:         $\mathcal{P}_{sig} \leftarrow \mathcal{U}_{sig}[\mathcal{C}_{sig}][j]$
16:         $b \leftarrow (|\mathcal{L}[y][x]| \gg j)$ & $1$
17:         encodeBit$(b, \mathcal{P}_{sig})$
18:         if $b = 1$ then
19:           $\mathcal{S}[y][x][0] \leftarrow 1$
20:           $\mathcal{C}_{sign} \leftarrow$ signContext$(\mathcal{L}, \mathcal{L}_2, y, x)$
21:           $\mathcal{P}_{sign} \leftarrow \mathcal{U}_{sign}[\mathcal{C}_{sign}][j]$
22:           $s \leftarrow \mathcal{L}[y][x] < 0$ ? $1 : 0$
23:           encodeBit$(s, \mathcal{P}_{sign})$
24:         end if
25:       end if
26:     end for
27:   end for
28:   refineMagnitude$(\mathcal{L}, \mathcal{S}, j)$
29: end for
```

Algorithm 2 - refineMagnitude
Parameters: local data $\mathcal{L}$, status map $\mathcal{S}$, and bitplane j

```
1: for $y \in \{0, ..., H-1\}$ do
2:   for $x \in \{0, 1\}$ do
3:     if $\mathcal{S}[y][x][1] = 1$ then
4:       $\mathcal{P}_{ref} \leftarrow \mathcal{U}_{ref}[j]$
5:       $b \leftarrow (|\mathcal{L}[y][x]| \gg j)$ & $1$
6:       encodeBit$(b, \mathcal{P}_{ref})$
7:     else if $\mathcal{S}[y][x][0] = 1$ then
8:       $\mathcal{S}[y][x][1] \leftarrow 1$
9:     end if
10:   end for
11: end for
```

Algorithm 3 - getNeighbors
Parameters: thread $T \in [0, 31]$, and coefficient coordinates y, x

```
1: return ( $\Phi(\mathcal{L}[y-1][x \pm 1], T \pm 1)$,
     $\Phi(\mathcal{L}[y][x \pm 1], T \pm 1)$,
     $\Phi(\mathcal{L}[y+1][x \pm 1], T \pm 1)$)
```

Algorithm 4 - encodeBit
Parameters: thread $T \in [0, 31]$, bit b, and probability $\mathcal{P}$
Initialization: $\mathcal{B} \leftarrow 0$ (bitstream index) , $Z \leftarrow 0$ (size of the interval), $L \leftarrow 0$ (lower bound of the interval)

```
1: if $Z = 0$ then
2:   $L \leftarrow 0$
3:   $Z \leftarrow 2^W - 1$
4:   $v \leftarrow \Omega(true)$
5:   $\widehat{\mathcal{B}} \leftarrow \mathcal{B} + \Psi(v \ll (32-T))$
6:   $\mathcal{B} \leftarrow \mathcal{B} + \Psi(v)$
7: end if
8: if $b = 0$ then
9:   $Z \leftarrow Z \cdot \mathcal{P}$
10: else
11:   $t \leftarrow (Z \cdot \mathcal{P}) + 1$
12:   $L \leftarrow L + t$
13:   $Z \leftarrow Z - t$
14: end if
15: if $Z = 0$ then
16:   $\mathcal{G}[\widehat{\mathcal{B}}] \leftarrow L$
17: end if
```

significant, its first bit is set to 1 (in line 19) to facilitate the context computation. The second bit of the status map indicates whether the coefficient has to be coded in the SPP or the MRP, so it is set to 1 (in line 8 of Algorithm 2) when the coefficient needs to be refined. Note that, for simplicity, we use SPP in this version of the coder to refer to the significance coding (despite that the CP is not in use).

Line 9 in Algorithm 1 is the loop that iterates from bitplane $M-1$ to 0. M is computed beforehand by each warp via a reduction operation. The SPP is applied in lines 10-27, whereas the MRP, embodied in Algorithm 2, is applied afterwards. The first operation (in line 12) of the SPP is to get the neighbors within the adjacent stripes needed to compute the context of the coefficient. This operation must be carried out before the potentially divergent step of line 13 because otherwise some threads may become inactive, being unable to participate in the communication. The communication among threads is done via the shuffle instruction, denoted by $\Phi(\cdot)$ in Algorithm 3. The function "getNeighbors$(\cdot)$" fetches the adjacent neighbors to $\mathcal{L}[y][x]$ that, depending on whether it is in the left or right column of the stripe, needs the $x+1$ or $x-1$ coefficient from the $T-1$ or $T+1$ thread, respectively. Algorithm 3 simplifies this with the operator $\pm$.

After fetching the neighbors, the algorithm checks whether the coefficient needs to be coded in the SPP or not. If so, the "significanceContext$(\cdot)$" function computes the significance context, denoted by $\mathcal{C}_{sig}$, employing the eight adjacent neighbors of the coefficient, as described in Section III. This function is not further detailed herein. Probability $\mathcal{P}_{sig}$ is accessed through $\mathcal{C}_{sig}$ and bitplane j in the corresponding LUT, which is referred to as $\mathcal{U}_{sig}$. The significance bit (computed in line 16, with & denoting a bit-wise AND operation) and its probability are fed to the arithmetic coder embodied in procedure "encodeBit$(\cdot)$". If the coefficient becomes significant (i.e., if $b=1$), then its sign has to be coded too. Lines 20-23 do so. The operations are similar to the coding of the significance bit.

The arithmetic interval employed by the arithmetic coder is represented by L and Z in Algorithm 4. L is its lower boundary and Z its size. The length of the codeword is denoted by W, so both L and Z are integers in the range $[0, 2^W-1]$. W is $W=16$ in our implementation, though other values are also valid [23]. The interval division is carried out in lines 8-14. When $b=0$, the lower subinterval is kept, otherwise the upper subinterval is kept. The codeword is exhausted when $Z=0$. As seen in line 16, then the codeword is put in position $\widehat{\mathcal{B}}$ of the bitstream. Note that $\widehat{\mathcal{B}}$ is computed in lines 1-7 when a new symbol is coded and the last codeword is exhausted (or at the beginning of coding). The vote and pop-count functions are denoted by $\Omega(\cdot)$ and $\Psi(\cdot)$, respectively. $\Omega(\cdot)$ is employed to compute how many concurrent threads reserve space in the bitstream. In line 5, $\Psi(\cdot)$ computes the number of threads with higher priority than T (i.e., all those processing the stripes on the left of the current). $\mathcal{B}$ is the length of the bitstream, stored in the shared memory. It is updated in line 6 considering all threads that have reserved a codeword in the bitstream.

Algorithm 5 - BPC-PaCo Encoder (with 3 coding passes)
Parameters: thread $T \in [0,31]$, codeblock coordinates X, Y, and codeblock height H

```
 1: allocate $\mathcal{L}[H][2]$ in local memory
 2: for $y \in \{0,...,H-1\}$ do
 3:   for $x \in \{0,1\}$ do
 4:     $\mathcal{L}[y][x] \leftarrow \mathcal{G}[Y+y][X+T*2+x]$
 5:     $\mathcal{S}[y][x][0] \leftarrow 0$
 6:     $\mathcal{S}[y][x][1] \leftarrow 0$
 7:     $\mathcal{S}[y][x][2] \leftarrow 1$
 8:   end for
 9: end for
10: for $j \in \{M-1,...,0\}$ do
11:   for $y \in \{0,...,H-1\}$ do
12:     for $x \in \{0,1\}$ do
13:       $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3 \leftarrow$ getNeighbors$(T,y,x)$
14:       if $\mathcal{S}[y][x][0] = 0$ then
15:         if any neighbor of $\mathcal{L}[y][x]$ has $\mathcal{S}[\cdot][\cdot][0] = 1$ then
16:           $\mathcal{C}_{sig} \leftarrow$ significanceContext$(\mathcal{S}, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, y, x)$
17:           $\mathcal{P}_{sig} \leftarrow \mathcal{U}_{sig}[\mathcal{C}_{sig}][j]$
18:           $b \leftarrow (|\mathcal{L}[y][x]| \gg j)$ & $1$
19:           encodeBit$(b, \mathcal{P}_{sig})$
20:           if $b = 1$ then
21:             $\mathcal{S}[y][x][0] \leftarrow 1$
22:             $\mathcal{C}_{sign} \leftarrow$ signContext$(\mathcal{L}, \mathcal{L}_2, y, x)$
23:             $\mathcal{P}_{sign} \leftarrow \mathcal{U}_{sign}[\mathcal{C}_{sign}][j]$
24:             $s \leftarrow \mathcal{L}[y][x] < 0$ ? $1 : 0$
25:             encodeBit$(s, \mathcal{P}_{sign})$
26:           end if
27:         else
28:           $\mathcal{S}[y][x][2] \leftarrow 1$
29:         end if
30:       end if
31:     end for
32:   end for
33:   refineMagnitude$(\mathcal{L}, \mathcal{S}, j)$
34:   for $y \in \{0,...,H-1\}$ do
35:     for $x \in \{0,1\}$ do
36:       $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3 \leftarrow$ getNeighbors$(T,y,x)$
37:       if $\mathcal{S}[y][x][2] = 1$ then
38:         $\mathcal{S}[y][x][2] \leftarrow 0$
39:         $\mathcal{C}_{sig} \leftarrow$ significanceContext$(\mathcal{S}, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, y, x)$
40:         $\mathcal{P}_{sig} \leftarrow \mathcal{U}_{sig'}[\mathcal{C}_{sig}][j]$
41:         $b \leftarrow (|\mathcal{L}[y][x]| \gg j)$ & $1$
42:         encodeBit$(b, \mathcal{P}_{sig})$
43:         if $b = 1$ then
44:           $\mathcal{S}[y][x][0] \leftarrow 1$
45:           $\mathcal{S}[y][x][1] \leftarrow 1$
46:           $\mathcal{C}_{sign} \leftarrow$ signContext$(\mathcal{L}, \mathcal{L}_2, y, x)$
47:           $\mathcal{P}_{sign} \leftarrow \mathcal{U}_{sign'}[\mathcal{C}_{sign}][j]$
48:           $s \leftarrow \mathcal{L}[y][x] < 0$ ? $1 : 0$
49:           encodeBit$(s, \mathcal{P}_{sign})$
50:         end if
51:       end if
52:     end for
53:   end for
54: end for
```

C. Encoder with 3 passes

Algorithm 5 details the CUDA kernel of the BPC-PaCo encoder with three coding passes. It uses the same functions as before. The structure of the algorithm is similar to that of Algorithm 1 too. The main difference is that significance coding is carried out in two different passes, the SPP and the CP. The SPP is applied from line 11 to 32, whereas the CP is carried out from line 34 to 53. As seen in lines 14 and 15, SPP only codes non-significant coefficients that have some significant neighbor. The CP codes the remaining non-significant coefficients.

The status map of this version of the encoder uses 3 bits per coefficient. The first two have the same meaning as before. The third flags the non-significant coefficients that are to be coded in the CP. It is initialized to 1 at the beginning of coding (in line 7) because only the CP is applied in the highest bitplane.

The probabilities employed for SPP and CP are different, so different LUTs are employed in each coding pass.

Clearly, the three-pass version of the encoder executes more instructions than the two-pass version. The addition of a third coding pass also increases the control-flow divergence, which results in longer execution times. Table III reports the number of instructions executed normalized by the problem size, the warp efficiency, and the normalized execution time achieved by both encoders. On average, the three-pass version executes 1.35× more instructions than the two-pass version, which corresponds with the increase in execution time. The warp efficiency is a metric to assess the control-flow divergence. It is measured as the average percentage of active threads per warp during execution time. The two-pass version of the algorithm achieves a 49% warp efficiency since, on average, half the threads in a coding pass are idle while the others code the coefficients. The three-pass version of the algorithm achieves a warp efficiency only 4% lower than that of the two-pass version since the CP does not produce much divergence among threads.

D. Decoder

The algorithmic structure and the cooperation mechanisms of the decoder are the same as those of the encoder. The bitstream is also stored in the global memory and the reconstructed coefficients are kept in the local memory. Contrarily to the encoder, the decoder reads the codewords from the bitstream and uses them to decode the symbols. Again, the codewords are read in a non-coalesced way, decreasing the efficiency of the memory transactions. In this case, the memory transactions can not be hidden by executing independent arithmetic operations as effectively as in the encoder. This is because the value of a codeword is required immediately after fetching it. This is the cause behind the slightly longer execution times of the decoder with respect to the encoder.

Table III reports the normalized execution time for both versions of the decoder. On average, the two-pass version of the decoder is 10.3% slower than the encoder, whereas the three-pass version is 9.2% slower. Despite this decrease in performance, our experience indicates that to store the bitstream in the global memory is more efficient than to use the shared memory or other strategies since they increase the number of instructions executed and decrease the occupancy.

V. Experimental results

The proposed implementation is compared with Kakadu v7.8 [18] and JPEG2K v1.0 [17]. As previously stated, Kakadu is one of the fastest JPEG2000 implementations. It is a C++ CPU multi-thread implementation heavily optimized via assembler. JPEG2K is an open-source CUDA implementation of JPEG2000. It is not optimized for the latest CUDA architectures, but still offers the most competitive performance among open-source implementations. BPC-PaCo and JPEG2K are compiled with CUDA 7.5 and executed in five devices, namely, a GTX TITAN X, GTX TITAN Black, GTX 480, GTX 750, and a Tegra X1. Kakadu is executed in a workstation with 4 Intel Xeon E5-4620 at 2.20 GHz (8 cores and 16 threads per processor, for a total of 32 cores and 64 threads). It is compiled using GCC 4.8. The GPU metrics are collected employing "nvprof". The images employed in the experiments are captured by the GeoEye and Ikonos satellites. They have a maximum size of 10240×10240, are eight-bit gray scale, and have one component. These images are employed herein due to their very high resolution, which facilitates performance tests. The type of the image (e.g., natural, satellite, etc.) or its shape does not affect the computational performance. The obtained results hold for different types of images such as those employed in digital cinema, TV production, surveillance, or digital cameras, among others. The performance achieved by BPC-PaCo for different types of images is extensively studied in [20], [21], [24], [26]. Some of the following experiments employ reduced-size versions of these images. The irreversible 9/7 wavelet transform is employed to transform them with 5 levels of decomposition. Wavelet data are partitioned in codeblocks of 64×64. The GPU tests employ a block size of 128 CUDA threads. In all experiments, the results reported for Kakadu are obtained when using the optimal number of threads. See in Fig. 5 the performance achieved by this implementation when using different number of threads to code an image of the corpus. Results also hold for the other images. The vertical axis of the figure is the number of coefficients coded per unit of time (in Msamples/second). The scalability achieved from 2 to 8 threads is almost linear, though for a higher number of threads is notably decreased. In the workstation employed, the use of 32 threads achieves maximum performance.

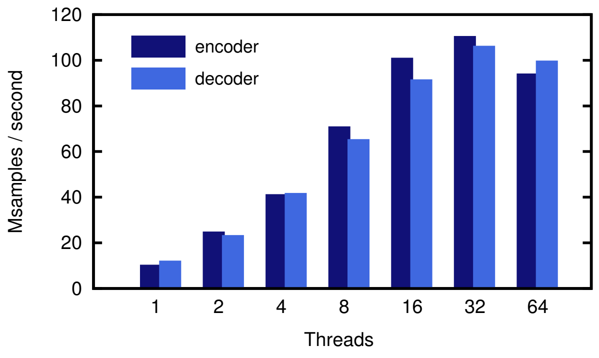

Fig. 5: Evaluation of the performance achieved by Kakadu when using different number of execution threads. Each pair of bars corresponds to an image.

In our implementation, CPU-GPU memory transfers are implemented synchronously using pinned memory. Table IV reports the time spent by the CPU-GPU transfers and the computation time spent by the BPC-PaCo encoder with 2 coding passes for different image sizes. Memory transfers represent 40% and 33% of the execution time, on average, when using 2 and 3 (not shown in the table) coding passes, respectively. These results hold for the decoder. In throughput-oriented scenarios, the memory transfers can be asynchronously overlapped with the computation task when coding large resolution images or video sequences. Only the bitplane coding time

TABLE III: Evaluation of GPU metrics achieved by the different versions of the codec. The experiments are carried out with a GTX TITAN X.

	encoder						decoder	
	2 coding passes			3 coding passes			2 coding passes	3 coding passes
image size	$\frac{\text{inst. executed}}{\text{\#coefficients}}$	warp efficiency	$\frac{\text{exec. time}}{\text{\#coefficients}}$	$\frac{\text{inst. executed}}{\text{\#coefficients}}$	warp efficiency	$\frac{\text{exec. time}}{\text{\#coefficients}}$	$\frac{\text{exec. time}}{\text{\#coefficients}}$	$\frac{\text{exec. time}}{\text{\#coefficients}}$
2048 × 2048	≈ 32.5	49%	0.98 ns	≈ 43.5	45%	1.36 ns	1.15 ns	1.56 ns
3072 × 3072			0.76 ns			1.06 ns	0.83 ns	1.15 ns
4096 × 4096			0.68 ns			0.94 ns	0.74 ns	1.02 ns
5120 × 5120			0.65 ns			0.90 ns	0.71 ns	0.99 ns
6144 × 6144			0.62 ns			0.86 ns	0.67 ns	0.94 ns
7168 × 7168			0.58 ns			0.85 ns	0.62 ns	0.89 ns

TABLE IV: CPU-GPU memory transfers and computation time of BPC-PaCo with 2 coding passes. The experiments are carried out with a GTX TITAN X (with a PCI 3.0 bus).

image size	mem. transfer CPU → GPU		BPC-PaCo (2 passes)		mem. transfer GPU → CPU		total time (in ms)
	time	%	time	%	time	%	
2048 × 2048	1.41	23	4.11	67	0.63	10	6.15
4096 × 4096	5.63	29	11.41	58	2.48	13	19.52
6144 × 6144	12.52	30	23.32	56	5.69	14	41.53
8192 × 8192	22.24	31	39.67	55	9.72	14	71.63

is reported in the following tests, excluding pre- and post-processing operations.

A. Computational performance evaluation

The first test evaluates computational performance. Fig. 6 depicts the achieved results. Each bar in the figure corresponds to the performance achieved when en/de-coding a particular image. Note that the figure is vertically split for illustration purposes. The results indicate that BPC-PaCo is significantly faster than the other implementations. The two-pass version of the encoder (decoder) achieves average speedups of 27.4× (25.1×) and 94.2× (121.1×) with respect to Kakadu (and JPEG2K). The three-pass version of BPC-PaCo achieves average speedups of 19.3× (18.2×) and 66× (87.7×). The two-pass version of the algorithm is, approximately, 1.4 times faster than the three-pass version, for both the encoder and the decoder. JPEG2K is slower than Kakadu because of the fine-tuning optimization carried out in Kakadu and because the GPU implementation of JPEG2000 can not fully exploit SIMD parallelism due to the sequential coding algorithms of JPEG2000. Fig. 6 also depicts the minimum performance needed to compress in real time high resolution digital cinema. The proposed implementation is the only that could be employed. We recall that, currently, implementations of real-time digital cinema need to employ field programmable gate arrays.

Table V reports some GPU performance metrics achieved when BPC-PaCo and JPEG2K code an image of the corpus. These metrics help to appraise the performance of our method and to explain the performance difference between our method and JPEG2K. As seen in the table, the 73× (53×) speedup of the two-pass (and three-pass) version of BPC-PaCo with respect to JPEG2K is due to improvements in three aspects: 1) the execution of 15 (11.1) times fewer instructions, 2) the execution of these instructions 4.5 (4.2) times faster, and 3) a slightly higher usage of the SMs. BPC-PaCo executes fewer instructions than JPEG2K because its algorithm is simpler and because it exhibits lower control flow divergence, as shown by its 1.49× (1.36×) higher warp efficiency. The fine-grained parallelism available in BPC-PaCo and the thread-to-data mapping strategy are behind the better GPU utilization metrics achieved by our method. The total IPC is defined as the aggregated number of instructions executed per clock cycle by all SMs. The higher IPC of BPC-PaCo is due to a higher SM occupancy (85% vs 23%) achieved by the abundant thread- and instruction-level parallelism of our implementation. BPC-PaCo achieves higher SM activity because it exposes more parallelism in the form of thread blocks. This analysis also holds for the decoder.

In order to assess the performance bottleneck of our implementation, additional performance metrics have been collected via the Nvidia profiler. The main results obtained for the two-pass encoder indicate that:

1) The computational utilization, determined as the ratio between the achieved instruction throughput (i.e., IPC) and the maximum throughput theoretically attainable is 53%. The throughput for specific instruction types, like integer, load, or shift operations, is also well below their peak limits. This suggests that performance is not bounded by the computational throughput.
2) The memory bandwidth achieved is 28% since most of the memory traffic is filtered by the L1 and L2 caches. This indicates that the memory bandwidth can be discarded as the performance bottleneck too.
3) Most of the stalling cycles occurring in the execution pipeline are due to general computing operations. Only 20% of the stalling cycles are caused by memory dependencies. This indicates that our implementation is mainly bounded by the execution latency of dependent computing instructions.

Similar results are obtained with the three-pass version of the encoder and with both versions of the decoder.

Table VI reports the computational performance of BPC-PaCo when using different codeblock sizes. The width of the codeblock is 64 for all the tests so that the same optimized memory access pattern is employed. As seen in the table, the use of 64×32 codeblocks obtains the highest performance, which is approximately 12% higher than when using 64×64

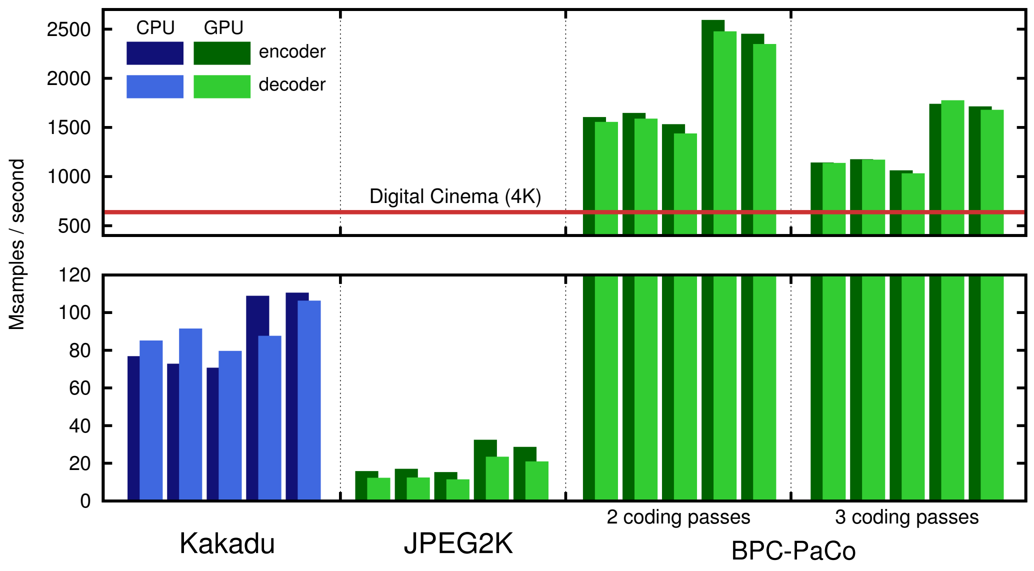

Fig. 6: Computational performance evaluation. Kakadu is executed with 32 CPU threads and GPU implementations are executed in a GTX TITAN X. Each pair of bars corresponds to an image.

TABLE V: GPU metrics achieved by BPC-PaCo and JPEG2K when coding a 4096×4096 image. The results are obtained with a GTX TITAN X. The speedup relative to JPEG2K is reported in parentheses.

	JPEG2K	BPC-PaCo encoder	
		2 passes	3 passes
time (ms)	834	11.4 (73×)	15.8 (53×)
#inst. ($\times 10^6$)	8105	541 (15×)	730 (11.1×)
total IPC	11.3	50.4 (4.5×)	48 (4.2×)
SM activity	86%	94% (1.09×)	96% (1.12×)
warp efficiency	33%	49% (1.49×)	45% (1.36×)
SM occupancy	23%	85% (3.7×)	83% (3.61×)

TABLE VI: Evaluation of the execution time (reported in ms) of BPC-PaCo for different codeblock sizes when coding a 4096×4096 image. The experiments are carried out with a GTX TITAN X.

	encoder		decoder	
codeblock size	2 passes	3 passes	2 passes	3 passes
64×32	10.42	14.13	11.21	15.32
64×64	11.41	15.76	12.45	17.01
64×96	12.01	16.34	13.04	17.82
64×128	13.84	18.68	14.82	20.59

codeblocks. This is because a finer subdivision of the data improves further the parallelism, helping to hide the execution latencies. Nonetheless, experimental evidence indicates that the coding performance is penalized when using small codeblocks [20]. In general, 64×64 codeblocks provide a good tradeoff between computational and coding performance.

B. *Scalability evaluation*

The aim of the next experiment is to appraise the scalability of our implementation for different image sizes and different devices. This test uses one of the previous images scaled from 2048×2048 to 9216×9216. The main features of the GPUs employed in the following tests are reported in Table VII. The column that shows the peak computing throughput also depicts the normalized performance with respect to that GPU with the lowest throughput (i.e., the Tegra X1), in parentheses. The GPUs are sorted in this table by their peak throughput.

Fig. 7 depicts the performance results achieved with the two- and three-pass version of BPC-PaCo. For the GTX 480, GTX 750, and Tegra X1, our method achieves regular performance regardless of the image sizes selected in our experiments, i.e., performance scales proportionally with the size of the input data. This is seen in the figure in the form of almost straight plots. The GTX TITAN Black and GTX TITAN X penalize the performance when the images are small. This is because they have more arithmetic cores (3077 and 2880, respectively) than the other devices (a proportion of at least 5 to 1), requiring a minimum input data size larger than in the other devices to fully utilize their resources. As shown in Table III, the performance improves with the problem size until reaching images of 7168×7168. With codeblocks of 64×64, the coding of 2048×2048 image utilizes 1024 warps, which is not enough to use all cores of these GPUs. Since the execution performance is bounded by the latency of the computing instructions, a higher degree of warp-level parallelism helps hiding the waiting time for those latency, improving SM activity and resource utilization, thereby enhancing the performance.

The throughput achieved by BPC-PaCo with each GPU, shown in Fig. 7, does not correspond exactly with the peak throughput of each device, reported in Table VII. The dif-

TABLE VII: Features of the GPUs employed. The devices are sorted by their peak throughput.

device	compute capability	SMs	cores × SM	total cores	clock frequency	peak throughput (normalized)	TDP
GTX TITAN X	Maxwell 5.2	24	128	3072	1000 MHz	3072 Gops/sec (12.00)	250 W
GTX TITAN Black	Kepler 3.5	15	192	2880	890 MHz	2563 Gops/sec (10.01)	250 W
GTX 480	Fermi 2.0	15	32	480	1400 MHz	672 Gops/sec (2.63)	250 W
GTX 750	Maxwell 5.0	4	128	512	1020 MHz	522 Gops/sec (2.04)	55 W
Tegra X1	Maxwell 5.3	2	128	256	1000 MHz	256 Gops/sec (1.00)	10 W

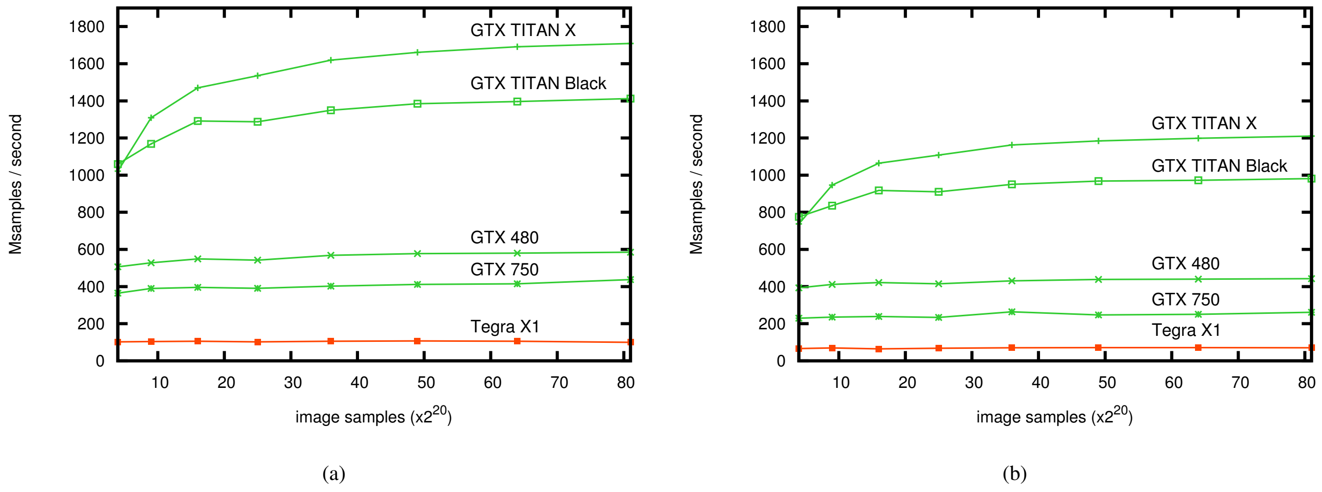

Fig. 7: Performance evaluation of BPC-PaCo with (a) 2 coding passes and (b) 3 coding passes when using different GPUs.

TABLE VIII: Performance metrics evaluation of BPC-PaCo when using different GPUs for coding a 4096×4096 image. Normalized performance is computed as samples/second divided by peak GPU throughput. The percentage of achieved IPC versus the peak IPC that is theoretically attainable by the GPU is reported in parentheses.

	time (in ms)		norm. perf.		#inst. ($\times 10^6$)		total IPC (% of peak)		SM activity	
cod. passes:	2	3	2	3	2	3	2	3	2	3
GTX TITAN X	11.4	15.8	0.479	0.346	541	730	50.4 (53%)	48.0 (50%)	94%	96%
GTX TITAN Black	12.9	18.3	0.507	0.358	485	639	43.4 (48%)	41.5 (45%)	97%	94%
GTX 480	30.5	39.8	0.819	0.628	481	638	11.4 (76%)	11.9 (79%)	99%	96%
GTX 750	42.4	70.1	0.758	0.458	538	764	12.4 (77%)	11.0 (69%)	99%	97%
Tegra X1	159.3	259.4	0.411	0.253	541	730	3.4 (42%)	2.9 (36%)	100%	99%

ferences are assessed in more detail in Table VIII, which reports the execution time, normalized performance, instructions executed, total IPC, and SM activity when an image of 4096×4096 is coded. The GTX 480 is the most cost-effective device for the two-pass (and three-pass) version of BPC-PaCo, with a normalized performance of 0.819 (0.628) Msamples/second for every unit of peak performance throughput. This is more than 1.5× (1.65×) higher than the performance achieved with the more powerful devices GTX TITAN X and GTX TITAN Black, and 2× (2.5×) higher than the performance achieved with the Tegra X1. The GTX TITAN Black and GTX 480 execute between 11% and 14% fewer instructions than the other GPUs, which suggests that the Nvidia compiler generates a more compact code for the instruction set architectures of Kepler and Fermi than for Maxwell. The GTX TITAN X executes an average of 50.4 instructions per clock cycle (which corresponds to 50.4 × 32 = 1612.8 operations per clock cycle, with 32 being the number of operations per SIMD instruction or warp), which represents 53% of the peak computing potential that is theoretically attainable by its 3072 cores. Larger images improve the total IPC around 10% and the SM activity from 94-96% to almost 100% on the GTX TITAN X. However, the GTX 480 and the GTX 750 achieve higher resource efficiency (more than 75%) than the GTX TITAN GPUs (lower than 57%), and considerably higher than the Tegra X1 (around 36-42%). In summary, the GTX 480 benefits both from a more compact code and from a better resource utilization. The lower efficiency of the Tegra X1 demands further analysis, but a plausible explanation is that it has been designed sacrificing operation latency in order to reduce energy consumption.

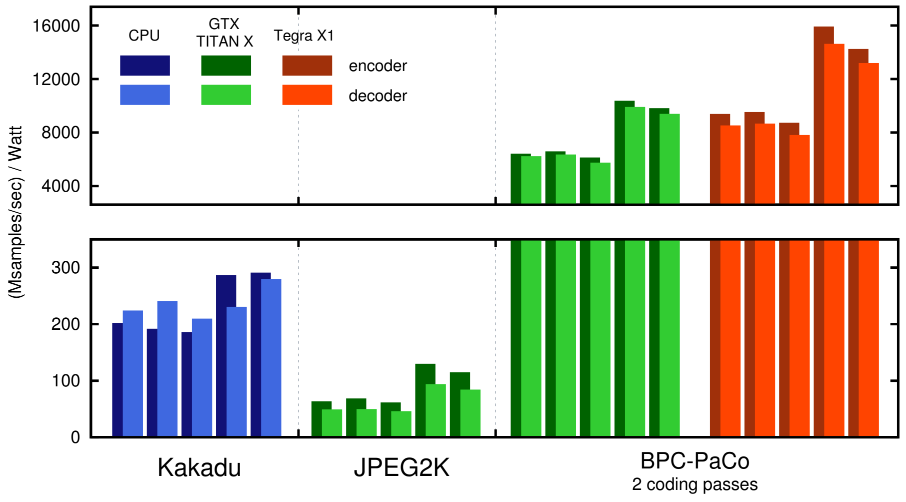

Fig. 8: Power efficiency evaluation of Kakadu, JPEG2K, and BPC-PaCo with 2 coding passes when using a GTX TITAN X and a Tegra X1. Each pair of bars corresponds to an image.

C. Power consumption evaluation

The following test assesses the power consumption of the proposed method as compared to Kakadu and JPEG2K in two devices, namely, the GTX TITAN X and the Tegra X1. The GTX TITAN X is a high-performance GPU, whereas the Tegra X1 is the last generation of Nvidia mobile processors, especially devised to maximize the power-efficiency ratio. Although this device has modest computational power, its peak performance per watt ratio is much higher than that of other devices. This is seen in the rightmost column of Table VII, which reports the Thermal Design Power (TDP) of the device. We recall that the TDP is a metric that measures the maximum amount of heat generated by the device in typical operation. This metric is often used to compare the actual power consumption of different devices. As seen in the table, the TDP of the Tegra X1 is 10W as compared with the 250W of the GTX TITAN X.

Fig. 8 depicts the results achieved when coding the five images of the corpus. The vertical axis of the figure is the performance (in Msamples/second) divided by the TDP of the device. The TDP of Kakadu considers only 4 of the 8 processors of the workstations, so its TDP is 380W since each individual Xeon E5-4620 has a TDP of 95W. The obtained results indicate that the BPC-PaCo encoder (decoder) executed in a GTX TITAN X is 41.6 (38.1) times more power efficient than Kakadu, on average. When running in the Tegra X1, the BPC-PaCo encoder (decoder) is 61.1 (52.7) times more efficient than Kakadu. With respect to JPEG2K, the increase in power efficiency is approximately twice as that of Kakadu. This indicates that, in addition to achieve very high performance, the proposed algorithms and implementations are a lot less power-hungry than the state-of-the-art codecs, making it amenable for mobile devices.

VI. Conclusions

The algorithms in the core of current image coding systems are mainly devised from a sequential point of view. In image compression standards such as JPEG2000, these core algorithms employ bitplane coding strategies and arithmetic coders to code the coefficients of a transformed image. Their main characteristic is to scan the coefficients in a linear fashion producing a bitstream that can only be encoded/decoded sequentially symbol after symbol. This enormously limits the fine-grained parallelism that can be achieved in high-performance devices such as GPUs. Aimed to tackle this issue, our previous work [20] presented BPC-PaCo, an algorithm that, still employing bitplane strategies and arithmetic coders, divides the workload in small pieces that can be processed by lock-step threads of execution in a SIMD fashion. The proposed method does not sacrifice any of the advanced features of the state-of-the-art coding systems and has only a slight penalization in coding performance.

This paper presents the first implementation of BPC-PaCo. The main insights of the proposed implementation are an efficient thread-to-data mapping, a smart memory management, and fast cooperation mechanisms among the threads in a warp. The experimental results achieved with the proposed implementation indicate that it is a very high-performance, power-friendly codec. The main advantage of BPC-PaCo is that its implementation in GPUs achieves a computing performance that is in the order of 30× higher than the best implementations of conventional image codecs in CPUs, while its power consumption is 40× times lower. This suits applications with requirements like real time coding, massive data production, or constrained power. Examples of such applications are digital cinema, TV production, or mobile imaging, among others. The implementation employed in this work is freely available in [27].

References

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

[2] S. Mallat, “A theory of multiresolution signal decomposition: the wavelet representation,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, pp. 674–693, Jul. 1989.

[3] F. Auli-Llinas, “Model-based JPEG2000 rate control methods,” Ph.D. dissertation, Universitat Autònoma de Barcelona, Barcelona, Spain, Dec. 2006. [Online]. Available: http://www.deic.uab.cat/∼francesc

[4] 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.

[5] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado, “Parallel implementation of the 2D discrete wavelet transform on graphics processing units: filter bank versus lifting,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 3, pp. 299–310, Mar. 2008.

[6] J. Matela *et al.*, “GPU-based DWT acceleration for JPEG2000,” in *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, Nov. 2009, pp. 136–143.

[7] 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.

[8] J. Franco, G. Bernabé, J. Fernández, and M. Ujaldón, “Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs,” *Procedia Computer Science*, vol. 1, no. 1, pp. 1101–1110, May 2010.

[9] 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.

[10] C. Song, Y. Li, and B. Huang, “A GPU-accelerated wavelet decompression system with SPIHT and Reed-Solomon decoding for satellite images,” *IEEE J. Sel. Topics Appl. Earth Observations Remote Sens.*, vol. 4, no. 3, pp. 683–690, Sep. 2011.

[11] 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.

[12] 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.

[13] V. Galiano, O. López, M. P. Malumbres, and H. Migallón, “Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs,” *The Journal of Supercomputing*, vol. 64, no. 1, pp. 4–16, Apr. 2013.

[14] V. Galiano, O. López-Granado, M. Malumbres, and H. Migallón, “Fast 3D wavelet transform on multicore and many-core computing platforms,” *The Journal of Supercomputing*, vol. 65, no. 2, pp. 848–865, Aug. 2013.

[15] J. Chen, Z. Ju, C. Hua, B. Ma, C. Chen, L. Qin, and R. Li, “Accelerated implementation of adaptive directional lifting-based discrete wavelet transform on GPU,” *Signal Processing: Image Communication*, vol. 28, no. 9, pp. 1202–1211, Oct. 2013.

[16] 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.

[17] (2016, Jun.) GPU JPEG2K. [Online]. Available: http://apps.man.poznan.pl/trac/jpeg2k/wiki

[18] D. Taubman. (2016, Jun.) Kakadu software. [Online]. Available: http://www.kakadusoftware.com

[19] (2016, Jun.) CUDA JPEG2000 (CUJ2K). [Online]. Available: http://cuj2k.sourceforge.net

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] 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.

[27] P. Enfedaque. (2016, Nov.) Implementation of BPC-PaCo in a GPU. [Online]. Available: https://github.com/PabloEnfedaque/CUDA_BPC-PaCo

Pablo Enfedaque is a Ph.D student with the Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain. He received the B.E. degree in computer science and the M.Sc. degree in high performance computing and information theory in 2012 and 2013, respectively, from Universitat Autònoma de Barcelona. His research interests include image coding, high performance computing and parallel architectures.

Francesc Aulí-Llinàs (S'06-M'08-SM'14) received the B.E. (with highest honors) and Ph.D. (cum laude) degrees in Computer Science from Universitat Autònoma de Barcelona (UAB) in 2002 and 2006, respectively. From 2002 to 2015 he was consecutively funded in competitive fellowships, including a Ramón y Cajal grant that was awarded with the intensification young investigator (i3) certificate. During this time, he carried out two postdoctoral research stages with professors David Taubman and Michael Marcellin. From 2016 to present, he is an associate professor (with the Full Professor certificate) with the Department of Information and Communications Engineering in the UAB. He developed and maintains BOI codec, a JPEG2000 implementation that is used in research and professional environments. In 2013, he received a distinguished R-Letter given by the IEEE Communications Society for a paper co-authored with Michael Marcellin. He has participated and supervised various projects funded by the Spanish government and the European Union. Also, he is reviewer for magazines and symposiums, has (co)authored numerous papers in journals and conferences, and has guided several Ph.D. students. His research interests lie in the area of image and video coding, computing, and transmission.

Juan C. Moure received his B.Sc. degree in computer science and his Ph.D. degree in computer architecture from Universitat Autònoma de Barcelona (UAB). Since 2008 he is associate professor with the Computer Architecture and Operating Systems Department at the UAB, where he teaches computer architecture and parallel programming. He has participated in several European and Spanish projects related to high-performance computing. His current research interest focuses on the usage of massively parallel architectures and the application of performance engineering techniques to open research problems in bioinformatics, signal processing, and computer vision. He is reviewer for various magazines and symposiums and has authored numerous papers in journals and conferences.