Strategy of Microscopic Parallelism for Bitplane Image Coding

Francesc Aulí-Llinàs[†], Pablo Enfedaque[†], Juan C. Moure[‡], Ian Blanes[†], and Victor Sanchez[§]

[†]Dep. of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain
[‡]Dep. of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain
[§]Dep. of Computer Science, The University of Warwick, United Kingdom

Abstract

Recent years have seen the upraising of a new type of processors strongly relying on the Single Instruction, Multiple Data (SIMD) architectural principle. The main idea behind SIMD computing is to apply a flow of instructions to multiple pieces of data in parallel and synchronously. This permits the execution of thousands of operations in parallel, achieving higher computational performance than with traditional Multiple Instruction, Multiple Data (MIMD) architectures. The level of parallelism required in SIMD computing can only be achieved in image coding systems via microscopic parallel strategies that code multiple coefficients in parallel. Until now, the only way to achieve microscopic parallelism in bitplane coding engines was by executing multiple coding passes in parallel. Such a strategy does *not* suit well SIMD computing because each thread executes different instructions. This paper introduces the first bitplane coding engine devised for the fine grain of parallelism required in SIMD computing. Its main insight is to allow parallel coefficient processing in a coding pass. Experimental tests show coding performance results similar to those of JPEG2000.

I. Introduction

In general, image compression systems deal with the computational complexity of coding large sets of data by fragmenting the image(s) in pieces that can be processed independently. JPEG2000, for instance, partitions the image in small sets of wavelet coefficients called codeblocks. The coding of small data pieces, called codeblocks from now on, is efficiently handled in processors whose architecture is mainly based on the Multiple Instruction, Multiple Data (MIMD) principle like common Central Processing Units (CPUs). Each executing core in the CPU can execute a flow of instructions independently and asynchronously from the others. The tasks of the coding system are straightforwardly mapped to a multi-core CPU: each codeblock is assigned to a processing core that codes its data. Such a strategy of fragmentation and parallel processing is referred to as macroscopic parallelism [1].

In the field of image compression, the most popular techniques to code the codeblock's data are bitplane coding and context-adaptive arithmetic coding [2]. These techniques scan the coefficients within the codeblock sequentially, so that the probability model can adjust the probability estimates of the emitted symbols as more data are coded. Such techniques are computationally simple, achieve high compression efficiency, and avoid a pre-processing step to collect statistics of the data. Their only drawback is that they

complicate microscopic parallelism, i.e., strategies to code in parallel multiple coefficients within the codeblock.

So far, microscopic parallel strategies have not been very attractive due to implementation difficulties and because most image coding systems are tailored to use the MIMD capabilities of conventional processors. This trend is starting to change due to the arrival of a new generation of processors that extensively employ the Single Instruction, Multiple Data (SIMD) principle [3]. SIMD computing applies a flow of instructions to multiple pieces of data in parallel and synchronously. Nowadays, SIMD computing achieves higher computational performance than MIMD while consuming less power. The Graphics Processing Units (GPUs) are the most representative processors of such architecture.

The fine level of parallelism required in SIMD computing can only be fully achieved via microscopic parallel strategies. Even so, the current approach is to implement *already developed* coding systems for their execution in SIMD-based processors. Without aiming to be exhaustive, GPU implementations of JPEG2000 are found in [4], [5] and there exist commercial products like [6] as well. The JPEG XR standard is implemented in [7], and video coding standards are studied in [3], [8]. Such implementations reduce the execution time of CPU-based implementations. Nevertheless, none of them can fully exploit the resources of the GPU due to the aforementioned sequential coefficient processing.

This paper introduces a bitplane coding strategy tailored for the type of microscopic parallelism required in SIMD computing. Its main features are: 1) a new scanning order that permits the processing of multiple coefficients in parallel, 2) a computationally-simple context formation approach, 3) a stationary model of probabilities that does not require adaptive mechanisms and, 4) the use of multiple arithmetic coders producing fixed-length codewords that are optimally sorted in the bitstream. To the best of our knowledge, this is the first bitplane coder that suits well SIMD computing. This paper describes the techniques employed from an image coding point of view. Future work will detail its implementation in a Nvidia GPU.

The paper is organized as follows. Section II briefly reviews preliminary concepts. Section III describes the proposed bitplane coding strategy. Its coding performance is assessed in Section IV. The last section concludes with a summary.

II. Background

The framework of JPEG2000 is adopted to test the proposed bitplane coding engine. A conventional JPEG2000 implementation is structured in three main coding stages [1]: data transformation, data coding, and codestream re-organization. The first stage applies the wavelet transform and quantizes wavelet coefficients. After data transformation, the image is partitioned in small sets of wavelet coefficients, the so-called codeblocks. Data coding is carried out in each codeblock independently. It takes approximately 70$\sim$75% of the total coding time. As stated before, the routines employed in this stage are based on bitplane coding and context-adaptive arithmetic coding. Herein, the original JPEG2000 engine is replaced by that proposed in this work. The last stage of the coding system re-organizes the final codestream constructing layers of quality.

Bitplane coding works as follows. Let $[b_{M-1}, b_{M-2}, ..., b_1, b_0]$, $b_i \in \{0, 1\}$ be the binary representation of an integer v which represents the magnitude of the index obtained by quantizing wavelet coefficient ω, with M being a sufficient number of bits to represent all coefficients. The collection of bits b_j from all coefficients is called a bitplane. Bits are coded from the most significant bitplane $j = M - 1$ to the least significant bitplane $j = 0$. The first non-zero bit of the binary representation of v is denoted by b_s and is referred to as the significant bit. The sign of the coefficient is denoted by $d \in \{+, -\}$ and is coded immediately after b_s, so that the decoder can begin approximating ω as soon as possible. The bits $b_r,\ r < s$ are referred to as refinement bits. In general, each bitplane is coded employing various coding passes that scan a subset of the coefficients. JPEG2000 employs three coding passes. Two of them are devoted to significance coding and one to refinement coding.

With regard to SIMD architectures, particularly those concerning GPUs, it is worth knowing that threads are executed as lanes of a vector instruction. Sets of T threads advance their execution in a lockstep synchronous way. Commonly, $T = 32$. All T threads execute the same instruction at the same time. When flow divergence occurs (due to conditionals, for instance), the divergent paths are executed sequentially one after another. In general, divergent paths are to be minimized. The vector instruction, that includes the T threads, represents the smallest scheduling unit in a GPU. This is called a warp in the Compute Unified Device Architecture (CUDA) employed in Nvidia GPUs.

III. Proposed method

A. Scanning order

The proposed method achieves microscopic parallelism by means of synchronously coding T coefficients in parallel during the execution of a coding pass. All threads perform the same operation to different coefficients, so CUDA implementations can assign a warp of threads to each codeblock. Fig. 1 depicts the scanning order employed. The light- and dark-blue dots in the figure represent the coefficients within a codeblock. The coefficients are organized in pairs of columns. Each pair is assigned to a thread, which scans all coefficients from the top to the bottom row, and from the left to the right coefficient. The threads are tightly synchronized, all scanning the same coefficient of their columns at the same time.

As seen below, the context of a coefficient is determined via its eight adjacent neighbors. The scanning order of Fig. 1 is highly efficient for context formation purposes. We recall that all information coded in previous passes is available when forming the context. Also, information coded in the *current* coding pass belonging to those neighbors that are visited before the current coefficient can also be employed. The higher the average number of already visited neighbors in the current coding pass (AVNP), the better to achieve competitive coding performance. The AVNP is computed without considering those coefficients in the border of the codeblock. Fig. 1 depicts in gray areas the eight adjacent neighbors of a coefficient in the left and in the right column assigned to a thread. The coefficient for which the context is formed is depicted with a red circle. The

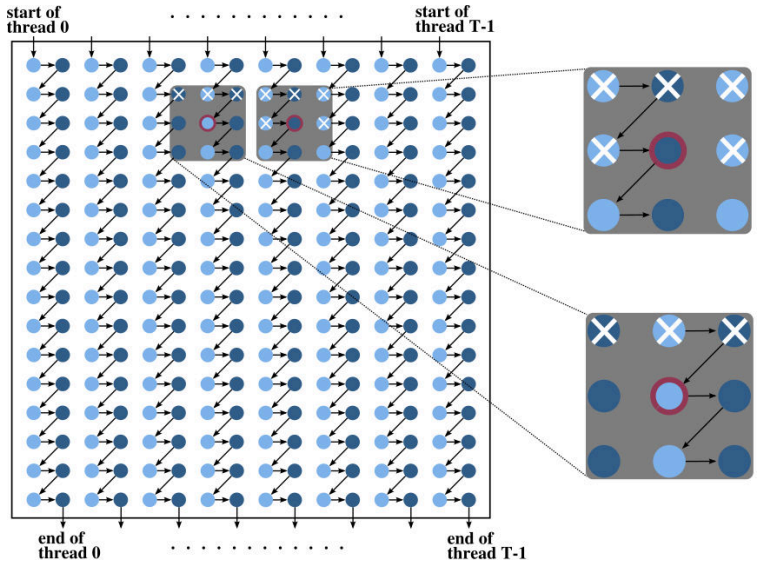


Fig. 1: Illustration of the scanning order and context formation for two coefficients.

neighbors that are already visited in the current coding pass are depicted with a white cross. The coefficients of the left column within the gray areas (depicted in light blue) have 3 already visited neighbors, whereas the coefficients in the right column have 5. So the AVNP achieved by the proposed scanning order is 4. We note that JPEG2000 and other coding systems employing sequential scanning orders also achieve an AVNP of 4.

B. Context formation and probability model

The contexts employed for significance coding use the significance state of the eight adjacent neighbors of coefficient ω. The neighbors of ω are denoted by ω^k, with $k \in \{\uparrow, \nearrow, \rightarrow, \searrow, \downarrow, \swarrow, \leftarrow, \nwarrow\}$ referring to the neighbor in the top, top-right, right,... position, respectively. The magnitude of the quantization index of these neighbors is denoted by υ^k. The significance state of υ^k in bitplane j is denoted by $\Phi(\upsilon^k, j)$. It is 1 when its significance bit has already been coded. This definition includes all neighbors that became significant in bitplanes higher than the current, i.e., $\Phi(\upsilon^k, j) = 1$ if $s > j$. It also includes the neighbors that become significant in the current bitplane –and that are already visited in the current coding pass–, i.e., $\Phi(\upsilon^k, j) = 1$ if $s = j$ and υ^k is already visited. Otherwise, $\Phi(\upsilon^k, j) = 0$.

The contexts employed for significance coding are denoted by $\phi_{sig}(\cdot)$. They are computed as the sum of the significance state of the eight adjacent neighbors of ω, more precisely, the significance state of υ at bitplane j is computed as $\phi_{sig}(\upsilon, j) = \sum_k \Phi(\upsilon^k, j)$. Evidently, $\phi_{sig}(\cdot) \in \{0, ..., 8\}$. It is shown in [9] that simple context formation approaches like this achieve competitive coding performance, so this approach is employed herein due to its computational simplicity.

The contexts employed for sign coding are similar to those of JPEG2000 since they obtain high efficiency. Sign contexts employ the sign of the neighbors in the vertical and horizontal positions. Let $\chi(\omega^k, j)$ represent the sign of ω^k when coding bitplane j.

$\chi(\omega^k, j)$ is 0 if the coefficient is not significant, otherwise is 1 and -1 for positive and negative coefficients, respectively. Then, $\chi^V = \chi(\omega^{\uparrow}, j) + \chi(\omega^{\downarrow}, j)$ and $\chi^H = \chi(\omega^{\leftarrow}, j) + \chi(\omega^{\rightarrow}, j)$. Context $\phi_{sign}(\omega, j)$ is computed according to

$$\phi_{sign}(\omega, j) = \begin{cases} 0 & \text{if } (\chi^V > 0 \text{ and } \chi^H > 0) \text{ or} \\ & \quad (\chi^V < 0 \text{ and } \chi^H < 0) \\ 1 & \text{if } \chi^V = 0 \text{ and } \chi^H \neq 0 \\ 2 & \text{if } \chi^V \neq 0 \text{ and } \chi^H = 0 \\ 3 & \text{otherwise} \end{cases} . \tag{1}$$

As suggested in [9], contexts for refinement coding should be based on techniques such as the local average, which are computationally intensive, or otherwise use only one context for all refinement bits. Herein, the latter approach is used for simplicity, so $\phi_{ref}(\upsilon, j) = 0$.

The contexts are employed together with the probability model to determine the probability estimate that is fed to the arithmetic coder. Conventional context-adaptive probability models cannot be employed herein since they adjust the probabilities in a sequential fashion. Therefore, the proposed bitplane coder employs a stationary probability model. The main idea is to use a fixed probability for each context and bitplane. As shown in [10], this model is based on the empirical evidence that the probabilities employed to code all symbols with a context are mostly regular in the same bitplane. The probability estimates are precomputed off-line and stored in a lookup table (LUT) that is known by the encoder and the decoder. The LUT contains one probability estimate per context and bitplane for each wavelet subband. They are denoted by $\mathcal{P}_u[j][\phi_{sig|ref|sign}(\cdot)]$, with u referring to the wavelet subband.

The probability estimates needed to populate the LUTs are determined as follows. Let $F_u(\upsilon \mid \phi_{sig}(\upsilon, j))$ denote the probability mass function (pmf) of the quantization indices at bitplane j given their significance context. This pmf is computed for each wavelet subband using the data from all images in a training set. Its support is $[0, ..., 2^{j+1} - 1]$ since it contains quantization indices that were not significant in bitplanes greater than j. The probability estimates used to populate the LUTs are generated by integrating the pmfs to obtain the probabilities of emitting 0 or 1 in the corresponding contexts. Denote the probability that bit b_j is 0 during significance coding by $P_{sig}(b_j = 0 \mid \phi_{sig}(\upsilon, j))$. This probability is determined from the corresponding pmf according to

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

The probability estimates for refinement and sign coding are derived similarly. A more in-depth study of this stationary probability model can be found in [10].

C. Arithmetic coding

The symbols and their probability estimates are fed to an arithmetic coder. Most arithmetic coders employed for image compression produce variable-to-variable length codes. This is, a variable number of input symbols are coded with a codeword of a priori unknown length. In JPEG2000, for instance, all data of a codeblock is coded with a single –and commonly very long– codeword. Practical realizations of arithmetic coders operate with hardware registers of 16 or 32 bits, so the generation of the codeword is carried out progressively.

Two aspects of conventional arithmetic coding prevent its use in the proposed bitplane coding strategy. The first is the generation of a single codeword. The scanning order described above utilizes T threads that code data in parallel. Forcing them to produce a single codeword would require to code their output in a sequential order, ruining the parallelism. The second aspect is the computational complexity of current arithmetic coders. Even the simplest executes five or more conditional instructions to code every symbol [11]. Part of this complexity is due to the conditionals and repositioning operations needed to control the generation of a long codeword.

These aspects are addressed herein by means of a new technique that employs multiple arithmetic coders that generate fixed-length codewords that are optimally placed in the bitstream. As previously described, each thread codes all data of two columns of coefficients. The coefficients coded by a thread are visited in a sequential order, so an arithmetic coder can be *individually* employed to code all symbols emitted by a thread. Instead of using conventional arithmetic coding, we employ an arithmetic coder that generates codewords of fixed length [11]. Variable-to-fixed length arithmetic coding avoids the operations needed to progressively process a long codeword, reducing the complexity of the coder. It uses an integer interval with a pre-defined range, say $[0, 2^W - 1]$ with W being the length of the codeword (in bits). The division of the interval is carried out in a similar way as with conventional arithmetic coding until its size is less than 2. Then, the number within the last interval is dispatched to the bitstream and a new interval is set (see below). In our implementation $W = 16$.

The codewords produced by the T threads of a codeblock generate a quality-embedded bitstream that can be truncated at any point so that the quality of the recovered image is maximized. In the encoder, the bitstream is constructed as follows. Each time that a thread initializes its interval (because is the beginning of the coding or the interval is exhausted), W bits are reserved in the bitstream. This space is hold until the thread exhausts its interval. Then, the codeword of that thread is put in the reserved space. Fig. 2 illustrates an example of this technique. All threads in the figure have its own space reserved in the bitstream. The coefficients depicted with a red circle are those currently visited by the threads. When thread 5 emits the symbol to code the coding pass of the coefficient, it exhausts its interval (depicted with the white codeword in the figure), so the codeword is put in the space reserved in the bitstream. Note that thread 5 does *not* reserve a new space in the bitstream at this instant but it will do it when coding a new symbol. This ordering strategy can be implemented via thread-collaborative operations in

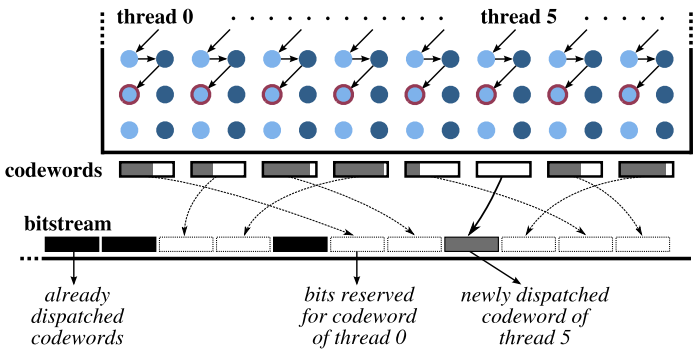


Fig. 2: Illustration of the technique employed to sort the codewords in the bitstream.

SIMD architectures. The order in which the codewords are sorted facilitates the decoding procedure. Any thread of the decoder only needs to read the next W bits of the bitstream when its interval is exhausted and a new symbol is decoded.

D. Algorithm

The encoding procedures of the proposed BitPlane Coding strategy with Parallel Coefficient processing (BPC-PaCo) are embodied in Algorithm 1. The same coding passes as those defined in JPEG2000 are employed since they achieve high coding performance [2]. One procedure per coding pass is specified. These procedures detail the operations carried out by one thread. The "ACencode" procedure describes the operations of the arithmetic coder. The scanning order is specified in the first two lines of the "SPP", "MRP", and "CP" procedures. The (quantized) coefficient visited is denoted by $(\upsilon_{y,x})$ $\omega_{y,x}$, with y, x indicating the row and column of the codeblock, respectively. The SPP and CP check whether the visited coefficient is significant in previous bitplanes or not. If not, they code bit b_j of the quantized coefficient. The SPP only visits coefficients that have at least one significant neighbor (i.e., those that have $\phi_{sig}(\upsilon_{y,x}, j) \neq 0$), whereas the CP visits all non-significant coefficients that were not coded by the SPP. The MRP codes the bit b_j of all coefficients that became significant in previous bitplanes.

The "ACencode" procedure codes all symbols emitted. The interval arithmetic of thread t is stored in registers $L[t]$ and $S[t]$, which are the lower boundary and the size minus one of the interval, respectively. Since the length of the codewords is W, both $L[t]$ and $S[t]$ are integers in the range $[0, 2^W - 1]$. The codeword is dispatched to the bitstream in lines 13-15 of this procedure when the interval is exhausted. If a new symbol is coded and $S[t] = 0$, W bits are reserved and the interval is reset (see lines 1-9).

The interval division is carried out in lines 6-12. When the symbol is 0 or $-$, the lower subinterval is kept, so $S[t] \leftarrow (S[t] \cdot p) \gg \widehat{\mathcal{P}}$ and $L[t]$ is left unmodified. $\gg$ denotes a bit shift to the right. p is the probability of the symbol to be 0 or $+$ expressed in the range $[0, 2^{\widehat{\mathcal{P}}} - 1]$, so $p = \lfloor P_{sig}(b_j = 0 \mid \phi_{sig}(\upsilon, j)) \cdot 2^{\widehat{\mathcal{P}}} \rfloor$ for significance coding, and equivalently for refinement and sign coding. $\lfloor \cdot \rfloor$ denotes the floor operation. As seen in Algorithm 1, p is the value that is stored in the LUTs. $\widehat{\mathcal{P}}$ is the number of bits employed to express the symbol's probability. In our implementation $\widehat{\mathcal{P}} = 7$. The coding of 1 or $+$ keeps the upper subinterval, as described in lines 9-11 of the "ACencode" procedure.

Algorithm 1 BPC-PaCo encoding procedures
Initialization: $S[t] \leftarrow 0 \quad \forall \quad 0 \leq t < T$

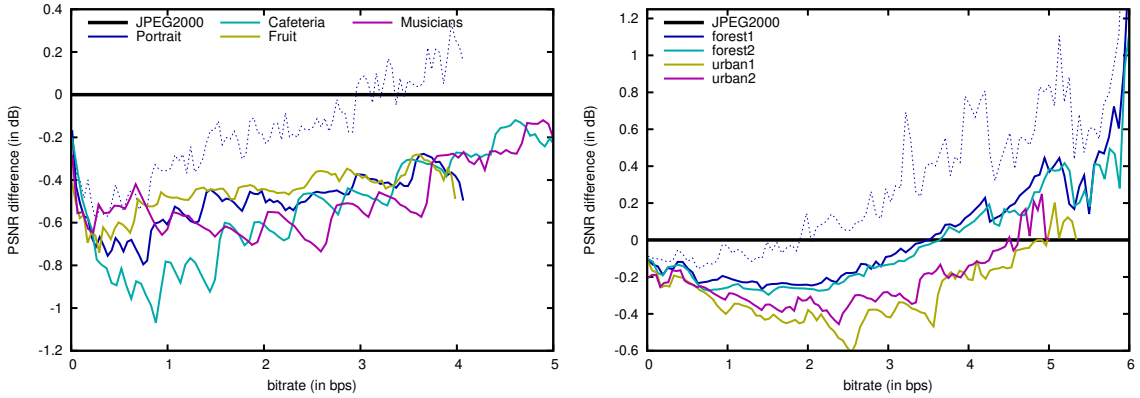
SPP (u subband, j bitplane, t thread)
for $y \in [0, \text{numRows} - 1]$ **do**
  for $x \in [t \cdot 2, t \cdot 2 + 1]$ **do**
    if $\upsilon_{y,x}$ is not significant **AND** $\phi_{sig}(\upsilon_{y,x}, j) \neq 0$ **then**
      ACencode(b_j, $\mathcal{P}_u[j][\phi_{sig}(\upsilon_{y,x}, j)]$, t)
      if $b_j = 1$ **then**
        ACencode(d, $\mathcal{P}_u[j][\phi_{sign}(\omega_{y,x}, j)]$, t)
      end if
    end if
  end for
end for

MRP (u subband, j bitplane, t thread)
for $y \in [0, \text{numRows} - 1]$ **do**
  for $x \in [t \cdot 2, t \cdot 2 + 1]$ **do**
    if $\upsilon_{y,x}$ significant in $j' > j$ **then**
      ACencode(b_j, $\mathcal{P}_u[j][\phi_{ref}(\upsilon_{y,x}, j)]$, t)
    end if
  end for
end for

CP (u subband, j bitplane, t thread)
for $y \in [0, \text{numRows} - 1]$ **do**
  for $x \in [t \cdot 2, t \cdot 2 + 1]$ **do**
    if $\upsilon_{y,x}$ not significant **AND** not coded in SPP **then**
      ACencode(b_j, $\mathcal{P}_u[j][\phi_{sig}(\upsilon_{y,x}, j)]$, t)
      if $b_j = 1$ **then**
        ACencode(d, $\mathcal{P}_u[j][\phi_{sign}(\omega_{y,x})]$, t)
      end if
    end if
  end for
end for

ACencode (c symbol, p probability, t thread)
if $S[t] = 0$ **then**
  Reserve the next W bits of the bitstream
  $L[t] \leftarrow 0$
  $S[t] \leftarrow 2^W - 1$
end if
if $c = 0$ **OR** $c = -$ **then**
  $S[t] \leftarrow (S[t] \cdot p) \gg \widehat{\mathcal{P}}$
else
  $f \leftarrow ((S[t] \cdot p) \gg \widehat{\mathcal{P}}) + 1$
  $L[t] \leftarrow L[t] + f$
  $S[t] \leftarrow S[t] - f$
end if
if $S[t] = 0$ **then**
  Put $L[t]$ in reserved space of the bitstream
end if

The interval division is carried out via integer multiplications and bit shifts because these are the fastest operations in hardware architectures. Other alternatives tested such as the use of LUTs reduce the computational performance. We note that the arithmetic coder embodied in Algorithm 1 performs several arithmetic operations and only three conditionals, which suits well SIMD computing.



(a) (b)

Fig. 3: Coding performance evaluation of BPC-PaCo vs. that of JPEG2000.

IV. Experimental results

Two corpora of images are employed to assess the performance of the proposed method. The first corpus consists of the eight natural images of the ISO 12640-1 corpus (2048×2560, gray scale, 8 bits per sample (bps)), whereas the second is composed of four aerial images provided by the Cartographic Institute of Catalonia, covering vegetation and urban areas (7200×5000, gray scale, 8 bps). Our JPEG2000 implementation BOI [12] is employed to compare the performance of the proposed method to JPEG2000. The coding parameters for all tests are: 5 levels of wavelet transform, lossy mode, codeblocks of 64×64, single quality layer, and no precincts.

Fig. 3 evaluates the coding performance achieved by BPC-PaCo as compared to that of JPEG2000. The results are reported as the peak signal to noise ratio (PSNR) difference achieved between BPC-PaCo and JPEG2000. The performance of JPEG2000 is depicted as the horizontal straight line in the figures. Results below this line indicate that the proposed method achieves lower PSNR than that of JPEG2000. To avoid cluttering the figure, results for only four of the eight natural images are reported in Fig. 3(a), though similar plots are achieved for the remaining images. The results of Fig. 3 indicate that, for natural images, the proposed method achieves PSNR values between 0.2 to 1 dB below those of JPEG2000. The results achieved by BPC-PaCo for aerial images are between 0.2 to 0.4 dB below those of JPEG2000 at low and medium bitrates, and from 0 to 0.6 dB above those of JPEG2000 at high bitrates.

For comparison purposes, Fig. 3 also reports the results when the RESET, RESTART, and CAUSAL coding variations of JPEG2000 are in use when coding the first image of each corpus. The results are reported with the plot with dots. These coding variations allow coding pass parallelism. We note that coding pass parallelism requires elaborate implementations [1] and it does not suit well SIMD computing since different threads perform different instructions, causing divergence. Even so, it is the only form of microscopic parallelism allowed in the standard. The results of Fig. 3 indicate that when these coding variations are in use, the coding performance difference between BPC-PaCo and

JPEG2000 is reduced between 0.2 to 0.5 dB, being almost negligible at most bitrates.

V. Conclusions

This paper introduces a microscopic parallel strategy for bitplane image coding. Contrarily to current approaches that only parallelize coding passes, the proposed method processes multiple coefficients in the same coding pass in parallel and synchronously. This fine level of parallelism is achieved rethinking the bitplane coding engine with new techniques for the scanning of coefficients, the formation of contexts and their model of probabilities, and for arithmetic coding. The resulting engine provides more opportunities for parallelism, especially for processors whose architecture strongly relies on the SIMD principle, such as GPUs. Experimental results indicate a slight penalization in coding performance when coding natural images as compared to JPEG2000. For aerial images, the coding performance achieved by the proposed method is almost equivalent to that of JPEG2000. Like all parallel algorithms, the proposed method can also be implemented to process sequentially the coefficients, employing a single thread of execution. The proposed bitplane coding strategy can be efficiently implemented for both MIMD and SIMD computing. Future work will describe the implementation of the proposed method in a GPU.

Acknowledgment

This work has been partially supported by the Spanish Government (MINECO), by FEDER, and by the Catalan Government, under Grants RYC-2010-05671, UAB-472-02-2/2012, TIN2012-38102-C03-03, TIN2011-28689-C02-1, and 2014SGR-691.

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] 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.

[3] N.-M. Cheung, X. Fan, O. C. Au, and M.-C. Kung, "Video coding on multicore graphics processors," *IEEE Signal Process. Mag.*, vol. 27, no. 2, pp. 79–89, Mar. 2010.

[4] 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.

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

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

[7] B. Pieters, J. D. Cock, C. Hollemeersch, J. Wielandt, P. Lambert, and R. V. de Walle, "Ultra high definition video decoding with motion JPEG XR using the GPU," in *Proc. IEEE International Conference on Image Processing*, Sep. 2011, pp. 377–380.

[8] N.-M. Cheung, O. C. Au, M.-C. Kung, P. H. Wong, and C. H. Liu, "Highly parallel rate-distortion optimized intra-mode decision on multicore graphics processors," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 19, no. 11, pp. 1692–1703, Nov. 2009.

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

[10] 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.

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

[12] ——. (2014, Nov.) BOI codec. [Online]. Available: http://www.deic.uab.cat/~francesc/software/boi