Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200028

GPU Architecture for Wavelet-Based Video Coding Acceleration

Carlos DE CEA-DOMINGUEZ [a,1], Juan C. MOURE [b], Joan BARTRINA-RAPESTA [a] and Francesc AULÍ-LLINÀS [a]

[a] *Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain*
[b] *Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain*

Abstract. The real time coding of high resolution JPEG2000 video requires specialized hardware architectures like Field-Programmable Gate Arrays (FPGAs). Commonly, implementations of JPEG2000 in other architectures such as Graphics Processing Units (GPUs) do not attain sufficient throughput because the algorithms employed in the standard are inherently sequential, which prevents the use of fine-grain parallelism needed to achieve the full GPU performance. This paper presents an architecture for an end-to-end wavelet-based video codec that uses the framework of JPEG2000 but introduces distinct modifications that enable the use of fine-grain parallelism for its acceleration in GPUs. The proposed codec partly employs our previous research on the parallelization of two stages of the JPEG2000 coding process. The proposed solution achieves real-time processing of 4K video in commodity GPUs, with much better power-efficiency ratios compared to server-grade Central Processing Unit (CPU) systems running the standard JPEG2000 codec.

Keywords. Wavelet-based video coding, high-throughput video coding, JPEG2000, GPU, CUDA.

1. Introduction

High definition video with resolutions ranging from 2K to 4K is nowadays common in devices such as TVs, digital cinema projectors, and mobiles. Among others, the JPEG2000 standard (ISO/IEC 15444) is employed for such video content in fields like TV production or digital cinema. This standard provides excellent coding performance and advanced features, such as quality progression, partial transmission, or error resilience [1]. Nonetheless, the algorithms employed to achieve them are very demanding computationally. They transform, code, and reorganize the data in three main stages that require multiple scans and coding operations. This results in long execution times and complex implementations. In the case of digital cinema, for instance, field-programmable gate arrays are required to achieve real-time decoding of 2K and 4K video.

[1] This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under Grants TIN2015-71126-R, TIN2017-84553-C2-1-R and RTI2018-095287-B-I00 (MINECO/FEDER, UE), and by the Catalan Government under Grants 2017SGR-463 and 2017SGR-313.

The first stage of most wavelet-based image/video codecs (including JPEG2000) reduces the image redundancy through the discrete wavelet transform (DWT). The operations performed by the DWT can be parallelized, so DWT implementations for parallel architectures have been widely studied in the literature [2–4]. The second stage employs bitplane coding together with arithmetic coding to reduce the statistical redundancy of wavelet coefficients. This stage scans the coefficients one-by-one, producing a result for each that can not be obtained without all the previous. This causality renders parallelism at the bitplane coding engine a very challenging task. Even though there are some works in the literature with this purpose [5–12], none of them is able to exploit the full potential of GPUs. The third stage of the pipeline reorganizes the data and forms the compressed file.

Aware of the high complexity of JPEG2000, the Joint Photographics Experts Group launched in June 2017 a call for proposals [13] aimed to develop an alternate algorithm for the bitplane and arithmetic coding stage that increases the throughput of the codec. This JPEG2000 part is described in [14, 15]. It increases performance about $10\times$ though penalizes coding performance about 10%. Also, it sacrifices quality scalability, which may become an issue in image/video transmission scenarios.

In the same line of work but well before this call for proposals, we started a line of research whose final goal is to devise an end-to-end image/video codec that can exploit the fine-grain parallelism of GPUs while maintaining the same features of JPEG2000. For the bitplane and arithmetic coding engine, we introduced a bitplane coding strategy with parallel coefficient processing (BPC-PaCo) that does not hold dependencies among coefficients, allowing efficient implementations in GPUs [16–19]. This engine can effectively exploit the parallelism of SIMD architectures, which results in high speedup factors and lower power consumption with respect to the fastest implementations of JPEG2000, either executed in multi-core CPUs or in GPUs. Evidently, BPC-PaCo is not compliant with the standard, but it does *not* sacrifice quality scalability and it penalizes coding performance only about 2%. For the DWT, we also proposed an efficient architecture in [2] that achieves high performance in GPUs.

The first implementation of the end-to-end codec for GPUs was presented in [20]. Nonetheless, that implementation is only able to process individual high-resolution images. This paper presents a vastly improved implementation that processes video sequences in real-time thanks to the introduction of stream management with multiple CPU threads, a double-buffer strategy, and event handling to synchronize GPU and CPU operations. Experimental results achieved with consumer-grade GPUs suggest that the proposed codec achieves a throughput that allows encoding and decoding 4K video in real-time and yields highly better power consumption ratios than JPEG2000 codecs executed in CPUs.

The rest of the paper is structured as follows. Section 2 explains the basics of the Nvidia GPU architecture. Section 3 briefly describes the different parts of the JPEG2000 standard. Section 4 describes the proposed end-to-end codec in detail. Section 5 provides experimental results achieved when coding high resolution video in two GPUs and compares our results with Kakadu [21], one of the best multi-thread JPEG2000 implementations. The last section concludes summarizing this work.

2. Overview of Nvidia GPUs

Nvidia GPUs are hardware devices with tens of individual computing units called streaming multiprocessor (SMs). These SMs can work independently, allowing the GPU to process different sequences of operations, called streams, in parallel. This permits the execution of multiple algorithms in an interleaved fashion, which increases the opportunities for parallelism and thereby the throughput achieved. The SMs execute multiple 32-wide SIMD instructions (i.e., vector instructions) simultaneously.

GPUs from Nvidia employ the CUDA programming model, which defines a computation structure composed by (potentially) hundreds of thousands of threads grouped into warps (each with 32-threads), with each warp assigned to a thread block [22]. While the hardware device executes 32-wide SIMD instructions, a software CUDA thread is the virtualization of one of the lanes of the instruction. From the first CUDA-compatible architecture (v1.0) up to Pascal (v6.2), warps execute instructions in a lock-step synchronous fashion, featuring an implicit synchronization at the end of any divergence [23]. From Volta (v7.0) onward, the implicit synchronization is not included at the end of the branching instructions, and must be coded explicitly if needed [24]. Warps in a thread block are executed asynchronously and can cooperate via on-chip fast memories, using explicit synchronizing barrier instructions when required.

The CUDA memory model is hierarchically organized as follows: there is a space of local memory private to each thread, a shared memory private to each thread block, and a global memory public to all threads in the kernel application. From a microarchitecture point of view, the local memory reserved per thread is located either in the registers or the off-chip memory, depending on the available resources. In the proposed implementation, the host memory (CPU RAM) and the device memory (GPU VRAM) are accessed as different, non-unified memory regions, explicitly managing the moment and the amount of data which are copied between them.

3. Overview of JPEG2000

Our GPU codec carries out the same steps as JPEG2000, so they are briefly described below. Depending on the encoding mode employed, either lossy or lossless, the operations involved may be irreversible or reversible, respectively. Irreversible operations improve the compression ratio though they sacrifice image quality slightly.

The first stage of the coding pipeline is the DWT. Our implementation uses a lifting scheme approach [25] due to its low computational complexity. It applies a series of arithmetic operations first row by row and then column by column. The DWT outputs four different subbands, with three of them including smaller detail images and the fourth including the original image at lower resolution and higher energy. These operations are carried out (typically) 5 times within the fourth subband, with each iteration in a lower resolution subband. For lossy compression, the operations employ floating-point arithmetic, so the resulting data are converted to integers before the next stage. This conversion is performed via deadzone quantization [1].

The second stage applies bitplane coding with arithmetic coding. The wavelet coefficients are conceptually partitioned in small sets of typically 64×64 wavelet coefficients, called codeblocks. The strategy adopted to process each codeblock consists in coding

the most relevant information first. The data are divided in bitplanes, with each bitplane containing the set of bits from the same binary position of the unsigned binary representation of each coefficient. Encoding begins from the most significant bitplane (i.e., the one with the highest magnitude within the codeblock) to the lowest one. In JPEG2000, each bitplane is scanned in three coding passes. The first pass is called Significance Propagation Pass. This pass only processes the bits of those coefficients that have at least one significant neighbor. A coefficient is called significant from that bitplane that holds the first non-zero bit to the lowest. The second coding pass is called Magnitude Refinement Pass. It visits the coefficients that are significant in higher bitplanes. The Cleanup Pass processes the coefficients not visited in the previous passes. This coding strategy aims to code the information which holds more information first, effectively reducing distortion [26]. Each bit emitted by the bitplane coder is fed to the arithmetic coder along with its contextual information. The context considers the number of neighbors that are significant, employed to determine a probability for the processed bit. This probability is employed by the arithmetic coder to generate the final bitstream.

The compressed data of codeblocks can be truncated to fit a target bitrate. The method to carry out this optimization process is not defined in the standard, so each implementation may adopt its own solution. The final stage reorganizes the data and adds ancillary information needed by the decoder to decode the original image. The decoder carries out the same steps of the encoder in reverse order.

4. Proposed Codec

The proposed codec is implemented in CUDA. CUDA is employed instead of OpenCL because it provides the latest improvements in the newest architectures. JPEG2000 exposes fine-grain parallelism in all coding stages except for the bitplane coder. Except from the bitplane and arithmetic coder, our proposal produces the same output as JPEG2000 in each stage employing a parallel architecture that extracts most of the GPU performance. BPC-PaCo is employed in the bitplane coding engine [18, 19]. As previously stated, this engine is not compliant with the standard though it preserves the same features and allows parallelism at a fine-grain level.

Algorithm 1 describes the main steps of the proposed codec. Fig. 1 also illustrates its main stages. First, the required memory to process the entire video is allocated in the host CPU RAM (lines 1-2) and in the GPU DRAM, which are respectively referred to as M^H and M^D. Next, two CPU threads are created, denoted by t_1 and t_2 in Algorithm 1, to manage the input/output from/to the hard disk (lines 3-6 and 10-13, respectively). The codec utilizes a double-buffer strategy per stream. This double-buffer is employed for both reading the raw data and writing the compressed file, so four buffers are allocated for each stream. These buffers are referred to as $M^H[i]$ and $M^D[i]$ for the input, and $M^H[o]$ and $M^D[o]$ for the output, with $\{i,o\} \in \{1..2\}$. When reading, the data from one buffer are processed while the other is filled. For writing, the compressed data are transferred to the host from one GPU buffer while the other is already empty and can be filled with compressed data from the frame that is being processed. This buffer structure enables the parallelization of the processing task in two streams, removes the risk of a system memory overflow and increases the utilization of the system resources. Both threads are constantly checking the buffers to start data transfers as soon as possible.

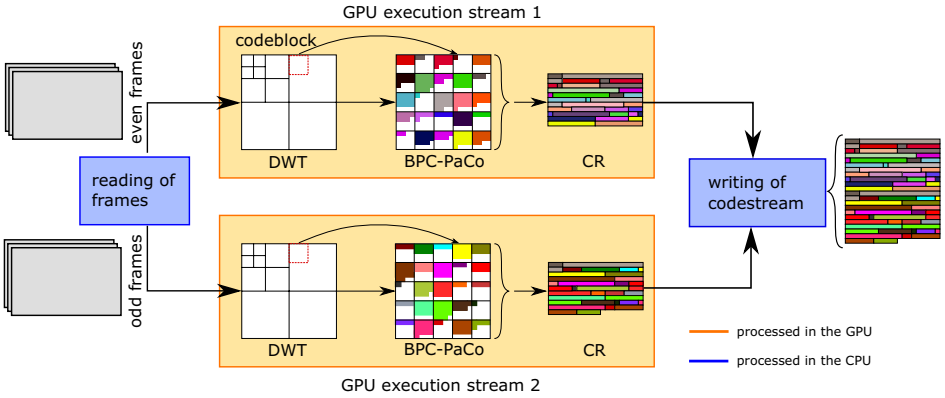

Figure 1. Illustration of the steps performed by the proposed video codec using two streams of execution.

Algorithm 1. Main routine of the codec

1: CPUMemoryAllocation()
2: GPUGlobalMemoryAllocation()
t_1
3: **for** each empty $M^H[i]$ **do**
4: $M^H[i] \leftarrow$ HDRead()
5: $M^D[i] \leftarrow M^H[i]$
6: **end for**
$S_{1,2}$
7: $D \leftarrow$ DWT_Q($M^D[i]$)
8: $\{B_l\} \leftarrow$ BPC_AC(D)
9: $M^D[o] \leftarrow$ CR($\{B_l\}$)
t_2
10: **for** each filled $M^D[o]$ **do**
11: $M^H[o] \leftarrow M^D[o]$
12: HDWrite($M^H[o]$)
13: **end for**

From lines 7 to 9 in Algorithm 1 the GPU functions, or kernels, to code a frame are called. The compression of each frame is carried out with three kernels. Two GPU streams, denoted by $S_{1,2}$, are employed to process a maximum of two frames simultaneously. Typically, each kernel transfers the data to be processed from the global memory $M^D[i]$ to the local memory R to accelerate memory accesses. The kernel DWT_Q$(\cdot)$ receives the original frame data as input and generates quantized wavelet coefficients that are the input of BPC_AC$(\cdot)$. BPC_AC$(\cdot)$ generates a bitstream per codeblock, referred to as B_l, with $l \in \{1..\widehat{L}\}$, $\widehat{L}$ being the number of codeblocks in each frame. This set of bitstreams is reorganized in the last kernel CR$(\cdot)$, which also adds ancillary information for decoding. This kernel does not transfer the compressed data to local registers since it only needs to reorganize the data in global memory.

As illustrated in Fig. 1, the use of two GPU streams allows the processing of two frames in parallel, increasing the throughput of the codec. Evidently, the three stages of the coding pipeline are carried out sequentially in each stream. Once a frame is coded, the resulting data are sent asynchronously to the host memory and the stream begins processing the next frame immediately. The three kernels are devised and implemented to extract fine-grain parallelism in the GPU. The proposed GPU-oriented architecture is able to process either high-resolution images or video in real time. Next, a brief description of each kernel is provided.

4.1. Discrete wavelet transform

The adopted DWT implementation [2] in our codec employs a register-based acceleration strategy [27] that transfers data from the global memory $M^D[i]$ to local registers, avoiding the use of shared memory. Threads communicate among them via shuffle instructions. This strategy allows data reuse and sharing, taking advantage of the fine-grain parallelism and data access locality of the algorithm. First, the image is conceptually divided in blocks that are independently processed by warps. This permits coarse-grain parallelism, populating more SMs of the GPU. The blocks take into account data dependencies of the transform, so they include some samples from adjacent blocks forming a halo. The halo is needed to obtain the same result as if the DWT was applied to the whole image at once. Within each block, the DWT is applied via the lifting scheme, which alternatively processes in the vertical and horizontal axis odd and even samples. If the compression mode is lossy, quantization is applied after the DWT since the next kernel requires integers.

4.2. Bitplane coding with parallel coefficient processing

The coefficients resulting from the DWT_Q$(\cdot)$ are conceptually partitioned in small sets called codeblocks. Typically, each codeblock contains 64×64 coefficients. They are transferred to the local memory to speed up the processing, like in the previous kernel. Codeblocks do not hold dependencies among them, so they are processed independently. The processing of codeblocks in parallel requires coarse-grain, control-divergent strategies that are employed in many implementations of the original JPEG2000 standard. In addition to this parallelism, the coding engine BPC-PaCo employed in this stage extracts fine-grain parallelism in the coding of the codeblock.

BPC-PaCo is based on bitplane coding, like JPEG2000. A particular feature of BPC-PaCo is that it conceptually divides the codeblock in 32 columns holding two coefficients each. Each codeblock is processed by a warp, and each 2-coefficient column is processed by a thread of the warp. Each thread carries out a modified version of significance coding that does not require cleanup, and the magnitude refinement pass. To employ only 2 coding passes instead of 3 like in JPEG2000 significantly increases the throughput [19]. Each emitted bit is coded via context-based arithmetic coding. To form the context of the coefficient, threads need to cooperate among them so that each coefficient can obtain information of all its adjacent neighbors. Again, this cooperation is performed via shuffle instructions. BPC-PaCo utilizes 32 arithmetic coders so that each thread in the warp can code all bits that it emits. The codewords generated by the arithmetic coders are interleaved in an optimal fashion in the final bitstream generated for the codeblock. The result of kernel BPC_AC$(\cdot)$ is the set $\{B_l\}$ that contains a bitstream per codeblock, with $l \in \{1..\widehat{L}\}$ and $\widehat{L}$ being the number of codeblocks per component. The probability model employed by the arithmetic coders is static, i.e., it employs pre-defined probabilities that are computed via a training set of images. This coding strategy permits the use of coarse- and fine-grain parallelism, since both the codeblocks and the coefficients within them are coded in parallel.

	SMs	cores × SM	clock frequency	memory bandwidth	peak FP32 throughput	TDP	memory size
GTX 1080 Ti	28	128	1923 MHz	484 GB/s	13.78 TFlops	250 W	11 GB
GTX 960M	5	128	1176 MHz	80 GB/s	1.5 TFlops	60 W	2 GB

Table 1. Features of the GPUs employed.

4.3. Codestream reorganization (CR)

The bitstreams produced for each codeblock have different size depending on the data coded, as depicted in Fig. 1. This results in data scattered in the output buffer of the global memory. The final stage of the coding process reorganizes these data putting them in a compact structure that can be transferred to the main memory of the host $M^H[o]$ and then written to the disk. This stage also includes auxiliary information in the final codestream for decoding.

When a warp compresses a codeblock, the lengths of the bitstreams are stored in a vector of integers $L = \{L_1, L_2, \cdots, L_{\widehat{L}}\}$. Then an aggregated list of lengths, i.e., $L' = \{0, L_1, L_1 + L_2, \cdots, L_1 + \cdots + L_{\widehat{L}}\}$ is generated via the Device Scan primitive from the Nvidia CUB framework [28]. To accelerate the access to this list, a fast lookup table, denoted by $\text{LUT}_{L'}$ is created. This LUT is generated applying a binary search over L' in which each position represents some positions of the original map. Our experience indicates that speedups about $2\times$ are achieved by using such a strategy. Kernel $\text{CR}(\cdot)$ then uses this LUT so that each thread transfers 2 bytes of the codeblock's data to the final output buffer.

5. Experimental Results

The throughput achieved by the proposed codec is compared with Kakadu v7.A.2 [21] in the experiments below. Kakadu is among the fastest implementations of JPEG2000, with multi-thread support for multi-core CPUs. It is programmed in C++ and is heavily optimized via assembly instructions. In the tests below, Kakadu is executed in a platform that has 4 AMD Opteron 6376 CPUs running at 2.3 GHz, employing a total of 32 threads of execution. Results from other JPEG2000 implementations in GPUs [11, 12] are not included herein because their throughput is similar or inferior to that of Kakadu, with the exception of Comprimato [10], which does not offer any option to test its implementation. Our codec is executed in the consumer-grade GPUs reported in Table 1, namely, the high-end GTX 1080 Ti for desktops, and the low-end GTX 960M for laptops. The tests code a video sequence of 948 frames of size 2048×832, gray-scale, and bit-depth of 8 bits per sample. The results shown below do not consider the I/O time needed to read/write the files from/to the disk since that may affect execution times significantly depending on the device employed. In all implementations, data are read from the host memory, where they are preloaded before starting the execution.

The first test evaluates the throughput achieved by our codec when using 1 or 2 GPU streams. The results are depicted in Fig. 2. The vertical axis reports the throughput achieved, in Mega samples per second (Msamples/sec.). The results for 1 and 2 streams are depicted for each GPU and for the encoding and decoding process. Using 2 streams provides a performance increase about 26% (7%) in the encoding process and 22% (9%)

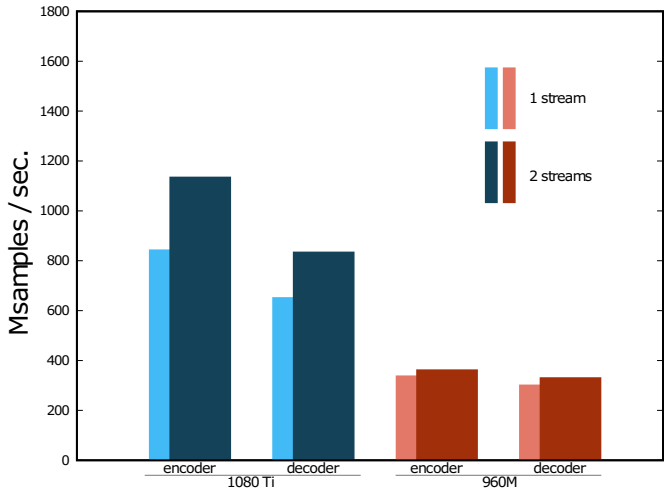

Figure 2. Evaluation of the throughput achieved when using 1 and 2 execution streams.

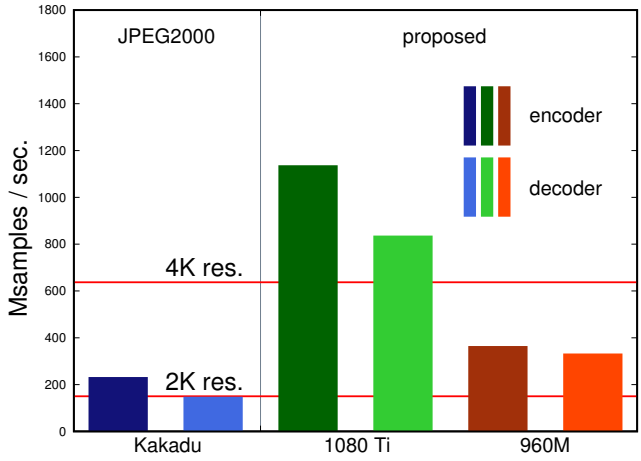

Figure 3. Evaluation of the throughput achieved by the proposed codec and Kakadu.

in the decoding process for the 1080 Ti (960M) GPU. The performance gain depends on the peak throughput of each GPU. The 1080 Ti has ample resources to process more than one frame whereas the 960M almost saturates its resources when coding a single frame (i.e., 1 stream).

The second test evaluates the throughput of the proposed codec running 2 streams and Kakadu. Fig. 3 depicts the obtained results. The proposed codec executed in the 1080 Ti yields a throughput about 5× higher than that of Kakadu. For the 960M, the throughput achieved is about 2× higher than that of Kakadu. Nonetheless, we recall that Kakadu is executed in an expensive multi-CPU platform, whereas the proposed codec employs commodity GPUs. Fig. 3 also depicts the throughput needed to process digital cinema video at resolutions of 2K and 4K in real time (straight horizontal lines). The results suggest that the proposed codec running in the 1080 Ti (960M) can process 4K

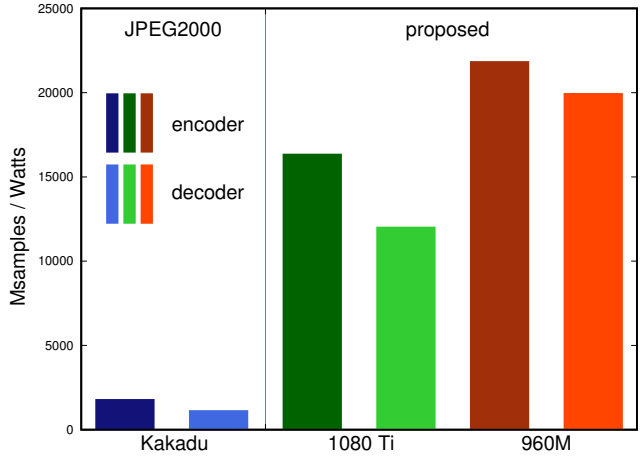

Figure 4. Evaluation of the power efficiency achieved by the proposed codec and Kakadu.

(2K) video in real time.

The third test evaluates power consumption. Fig. 4 depicts the results yield by Kakadu and our codec. In this case, the vertical axis reports Msamples processed per Watt consumed. Kakadu employs four high-end AMD Opteron processors, each with a thermal design power (TDP) of 115W, whereas the 1080 Ti and 960M GPUs have a TDP of 250W and 60W, respectively. The low power consumption of the 960M makes it the most efficient, being approximately 12× more power efficient than Kakadu for encoding and about 17× for decoding. The proposed codec executed in the 1080 Ti is less power-hungry than Kakadu too, with increases in efficiency about 9× and 10× for the encoder and decoder, respectively.

6. Conclusions

The JPEG2000 standard is mainly devised to exploit the coarse-grain parallelism provided in CPUs. When employed to code high-resolution video in scenarios such as TV production or digital cinema, implementations need specialized hardware or expensive computer platforms to meet real-time requirements. So far, implementations for cheaper devices such as GPUs are not able to achieve high throughput because the innermost algorithms of the coding system do not exhibit enough fine-grain parallelism. This paper introduces a fully parallel end-to-end codec that employs the framework of JPEG2000 but that provides –in all stages of the coding pipeline– distinct modifications that permits the use of fine-grain parallelism. This can be effectively exploited when executed in architectures that highly rely on SIMD parallelism such as those found in Nvidia GPUs. Experimental results coding high-resolution video indicates that the proposed codec is 5× faster than the most efficient implementations of JPEG2000 while reducing the power consumption more than 10×. None of the advanced features of JPEG2000 are sacrificed in the proposed codec, so it is ideal for scenarios that deal with massive data sets and/or power constraints.

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] 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.

[3] T. M. Quan and W.-K. Jeong, "A fast discrete wavelet transform using hybrid parallelism on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3088–3100, Nov. 2017.

[4] 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.

[5] 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.

[6] R. Le, I. R. Bahar, and J. L. Mundy, "A novel parallel tier-1 coder for JPEG2000 using GPUs," in *Proc. IEEE Symposium on Application Specific Processors*, Jun. 2011, pp. 129–136.

[7] 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.

[8] 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.

[9] M. Ciznicki, M. Kierzynka, P. Kopta, K. Kurowski, and P. Gepnerb, "Benchmarking JPEG 2000 implementations on modern CPU and GPU architectures," *ELSEVIER Journal of Computational Science*, vol. 5, no. 2, pp. 90–98, Mar. 2014.

[10] Comprimato. (2014, Apr.) Comprimato. [Online]. Available: http://www.comprimato.com

[11] (2016, Jun.) CUDA JPEG2000 (CUJ2K). [Online]. Available: http://cuj2k.sourceforge.net

[12] (2016, Jun.) GPU JPEG2K. [Online]. Available: http://apps.man.poznan.pl/trac/jpeg2k/wiki

[13] *High Throughput JPEG 2000 (HTJ2K): Call for Proposals*, ISO/IEC Std., 2017, document ISO/IEC JTC 1/SC29/WG1 N76037.

[14] D. Taubman, A. Naman, and R. Mathew, "FBCOT: a fast block coding option for JPEG 2000," in *Proc. SPIE Applications of Digital Image Processing*, vol. 10396, Sep. 2017, pp. 1–18.

[15] D. Taubman, A. Naman, R. Mathew, and M. D. Smith, "High throughput JPEG 2000 (HTJ2K): New algorithms and opportunities," *SMPTE Motion Imaging Journal*, vol. 127, no. 9, pp. 1–7, Oct. 2018.

[16] 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.

[17] 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.

[18] 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.

[19] P. Enfedaque, F. Auli-Llinas, and J. C. Moure, "GPU implementation of bitplane coding with parallel coefficient processing for high performance image compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2272–2284, Aug. 2017.

[20] C. de Cea-Dominguez, P. Enfedaque, J. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "High throughput image codec for high-resolution satellite images," in *Proc. IEEE International Geoscience and Remote Sensing Symposium*, Jul. 2018, pp. 6524–6527.

[21] D. Taubman. (2018, Dec.) Kakadu software. [Online]. Available: http://www.kakadusoftware.com

[22] "CUDA, C Programming guide," Tech. Rep., Jan. 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[23] Nvidia. (2018, Jan.) Warp level primitives. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

[24] ——. (2019, Jun.) Nvidia Tesla V100 GPU architecture. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[25] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM Journal on Mathematical Analysis*, vol. 29, no. 2, pp. 511–546, 1998.

[26] 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.

[27] A. Chacon, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Boosting the FM-index on the GPU: effective techniques to mitigate random memory access," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 12, no. 5, pp. 1048–1059, Sep. 2015.

[28] Nvidia. (2018, Dec.) CUB framework. [Online]. Available: https://nvlabs.github.io/cub/