GPU Architectures for Wavelet-based Image Coding Acceleration

Pablo Enfedaque

Universitat Autònoma de Barcelona
Departament d'Enginyeria de la Informació i de les Comunicacions

GPU Architectures for Wavelet-based Image Coding Acceleration

Submitted to Universitat Autònoma de Barcelona
in partial fulfillment of the requirements for the
degree of Doctor of Philosophy in Computer Science

by Pablo Enfedaque
Bellaterra, April 2017

Supervised by
Dr. Francesc Aulí-Llinàs and
Dr. Juan C. Moure

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Bellaterra, April 2017

Dr. Francesc Aulí-Llinàs

Dr. Juan C. Moure

Committee:

Dr. Manuel Ujaldón Martínez
Dr. Joan Bartrina Rapesta
Dr. Miguel Ángel Luján Moreno
Dr. Antonio Miguel Espinosa Morales (substitute)
Dr. Miguel Hernández Cabronero (substitute)

Abstract

Modern image coding systems employ computationally demanding techniques to achieve image compression. Image codecs are often used in applications that require real-time processing, so it is common in those scenarios to employ specialized hardware, such as Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). GPUs are throughput-oriented, highly parallel architectures that represent an interesting alternative to dedicated hardware. They are software re-programmable, widely available, energy efficient, and they offer very competitive peak computational performance.

Wavelet-based image coding systems are those that employ some kind of wavelet transformation before the data coding stage. Arguably, JPEG2000 is the most representative of those systems. Many research projects have tried to develop GPU implementations of JPEG2000 to speed up the coding pipeline. Although some stages of the pipeline are very suitable for GPU computing, the data coding stage does not expose enough fine-grained parallelism. Data coding is the most computationally demanding stage (75% of the total execution time) and represents the bottleneck of the pipeline. The research presented in this thesis focuses on the GPU computing of the most critical stages of wavelet-based image coding systems: the wavelet transform and the data coding stage.

This thesis proposes three main contributions. The first is a GPU-accelerated implementation of the Discrete Wavelet Transform. The proposed implementation achieves speedups up to 4$\times$ with respect to the previous state-of-the-art GPU solutions. The second contribution is the analysis and reformulation of the data coding stage of JPEG2000. We propose a new parallel-friendly high performance coding engine: Bitplane Image Coding with Parallel Coefficient Processing (BPC-PaCo). BPC-PaCo reformulates the mechanisms of data coding, without renouncing to any of the advanced features of traditional data coding. The last contribution of this thesis presents an optimized GPU implementation of BPC-PaCo. It compares its performance with the most competitive JPEG2000 implementations in both CPU and GPU, revealing speedups up to 30$\times$ with respect to the fastest implementation.

Acknowledgements

I would like to express my gratitude to everyone who has participated in the development of this thesis. Most of all, to my advisors: Francesc Aulí and Juan C. Moure. Thank you for all the ideas, all the advice, and also for your patience. But thank you above all for your time, for the many hours you have spent with me throughout this project. Without your help, I would not be the professional, researcher, and person I am today.

Secondly, I would like to thank my colleagues in both dEIC and CAOS. Special mention to Naoufal, Emilio, Roland, and Roger, but also to other colleagues who have left (some of the former master students) and the ones that arrived afterwards. Without you guys, this thesis would have been a significantly less exciting, fun, and overall less enjoyable experience. Thanks also to all the professors, staff and students of both CAOS and dEIC departments that have worked with me throughout the last few years.

Finally, I would like to thank everyone who, without knowing it, have contributed or helped me with this work. All the people that have been by my side throughout these last four years: friends, family, housemates, and all the amazing people that I have met. Thank you to the ones that will always be beside me and to the ones that shared my life for a shorter period of time: this thesis was possible also because of you.

Contents

Chapter 1

Introduction

The amount of digital images generated and stored in electronic devices has dramatically increased over the past 30 years. In this context, image coding has become increasingly important, developing new techniques to reduce the number of bits employed to represent images. Traditionally, image compression advances were based on the development of elaborated algorithms that increased the computational load of the codecs. Coding systems have also incorporated many features in addition to compression over the last years. Capabilities such as region of interest coding, progressive transmission or lossy-to-lossless support are generally inbuilt in modern codecs. Overall, current image coding systems are commonly computationally demanding applications.

Many current image coding applications, such as digital cinema or medical imaging, demand state-of-the-art compression performance and advanced features, in scenarios that also require real-time processing. Traditionally, these applications employed specialized hardware, such as Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs) to meet their execution time constraints. However, in the last decade, GPUs have become an increasingly popular alternative to specialized hardware due to their outstanding computational power and availability.

GPUs were originally devised to process graphics workload in video games or computer-aided design. Nowadays, they are also used for general purpose processing. This shift was motivated by the desire to employ the computational power of GPUs in different applications, and also because they are software re-programmable, widely available and highly energy efficient. General Purpose GPU (GPGPU) computing emerged as a widely employed technology in 2006, with the release of the Nvidia Compute Unified Device Architecture (CUDA), which provides a suite of tools and a GPU compiler for the acceleration of non-graphic applications.

In recent years, many research projects employed GPUs in diverse image coding applications [1–13]. However, not all projects succeed at employing GPUs to accelerate a specific application. In most cases, suboptimal performance was obtained because GPUs are not well-suited for all kind of algorithms. There is a set of requirements that an algorithm has to fulfill to make full use of the GPU computational resources. Those requirements can be difficult to meet depending on the application, rendering the GPU programming and GPU algorithm (re-) design as non-trivial problems. As a consequence, GPU computing is a very active research focus in many areas, such as linear algebra [14], signal processing [15], or computational biology [16, 17], among others. To further illustrate this matter it is necessary to take a closer look at the peculiarities of the GPU architecture.

The peak computational power of modern GPUs significantly surpasses that of the comparable CPUs. The reason behind GPUs' capabilities lies in their innermost architectural design: they are based on the Single Instruction Multiple Data (SIMD) principle, in which flows of instructions are executed in parallel and synchronously to different pieces of data. This layout simplifies the chip design, permitting the allocation of additional computational resources. Arguably, the GPU architecture is juxtaposed to that of the CPU's: the latter is commonly referred to as a latency-oriented processor conceived to run few instructions with low latency, whereas GPUs are throughput-oriented processors engineered to continuously process a pipeline of thousands of high-latency instructions.

GPUs' computational power comes with a very significant drawback. Due to their SIMD throughput-oriented architecture, GPUs require high amounts of parallel and homogeneous workload to achieve close-to-peak performance. Inherently sequential algorithms, or procedures with significant heterogeneous computation, can be challenging to implement in a GPU. Applications with those characteristics are commonly implemented in CPUs, which are more flexible and less sensitive to the algorithm's singularities. Alternatively, when acceleration is a top priority, an un-fitting algorithm can sometimes be successfully re-formulated to be efficiently implemented in a GPU. This thesis explores and overcomes the described challenges of GPU computing with the main operations of wavelet-based image coding, namely, the Discrete Wavelet Transform, the Bitplane coder, and the Arithmetic Coder.

1.1 Wavelet-based image coding and JPEG2000

Image coding systems are traditionally composed of two main stages: data transformation and data coding. Data transformation removes the spatial redundancy of the image by means of applying a given linear transform. Afterwards, data coding exploits the visual redundancy of the image to code the transformed coefficients. Wavelet-based image coding systems are those that employ some sort of wavelet transform [18] in the first stage. Wavelet transforms provide many valuable features for image coding, and have been widely adopted in many modern codecs. Arguably, the most representative wavelet-based image coding system is that of the JPEG2000 standard (ISO/IEC 15444-1) [19, 20]. JPEG2000 is employed in a wide range of commercial products and applications, as it provides state-of-the-art coding performance and advanced features. JPEG2000 is used as the reference coding system in this thesis, although the analysis and solutions presented herein can also be employed in other wavelet-based image codecs.

JPEG2000 employs the Discrete Wavelet Transform (DWT) in the data transformation stage [21]. The DWT is the prevalent wavelet-based data decorrelation technique in the field of image coding. It is employed in CCSDS-ILDC [22], SPIHT [23], EBCOT [24], and SPECK [25], among others. From a computational point of view, the DWT corresponds to the application of a pair of (high- and low-pass) filters to each image dimension (vertical and horizontal). Those operations can be seen as a 2-dimensional stencil pattern [26], which exposes very high parallelism and homogeneous computation among parallel flows. Because of this, the DWT perfectly fits the requirements of GPU computing. In the literature, many different methods are proposed to compute the DWT in a GPU [3, 5, 7–12, 27, 28]. Although they achieve very competitive speedups with respect to comparable CPU implementations, none of them fully saturates the GPU computational resources.

Data coding is performed in JPEG2000 using two different technologies: the bitplane coder and the arithmetic coder. The Bitplane Coder (BPC) selectively scans the coefficients of an image in a bit-by-bit fashion, and feeds them to the arithmetic coder, which produces a compressed bitstream. Data coding is carried out independently for different square tiles within the image, called codeblocks (commonly of size 64×64 pixels). This scheme exposes a very suitable degree of coarse-grained parallelism that can be efficiently implemented in CPU multi-thread architectures. Unfortunately, the scheme is not suitable for GPU computing. In order to efficiently compute data coding in a GPU, additional SIMD parallelism within the codeblock's computation is required. Promoting this parallelism is extremely challenging with the algorithm at hand: the fundamental problem is that the key operations performed within a codeblock are inherently sequential. Nonetheless, many implementations of bitplane coding with arithmetic coding in GPUs can be found in the literature [4,6,7,9,29,30]. None of them is able to achieve competitive GPU performance due to the aforementioned parallelism constraints of the algorithm.

1.2 Overview of the GPU architecture and CUDA

All the research presented in this thesis has been carried out taking CUDA-enabled Nvidia GPU architectures as a reference. This subsection contains a brief introduction to some relevant GPU and CUDA technical concepts employed thorough this manuscript.

GPUs contain multiple throughput-oriented SIMD units called Streaming Multiprocessors (SMs). Modern GPUs have up to several dozens of 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 of them is assigned and run until completion in a specific SM. Warps inside the same block are executed asynchronously, but they can cooperate sharing data and can synchronize using explicit barrier instructions. The unit of work sent from the CPU (host) to the GPU (device) is called a kernel. The host can launch some kernels for parallel execution, each composed from tens to millions of thread blocks.

The memory hierarchy of GPUs is organized in 1) a space of big, off-chip global memory that is public to all threads, 2) a space of small, on-chip shared memory that is private to each thread block, and 3) a space of local memory that is private to each thread. 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. The registers have the highest bandwidth and lowest latency, whereas the shared memory bandwidth is significantly lower than that of the registers. The shared memory provides flexible accesses, while the accesses to the global memory must be coalesced to achieve higher efficiency. A coalesced access occurs when consecutive threads of a warp access consecutive memory positions. 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 communication between the threads in a thread block is commonly carried out via the shared memory. Threads in a warp can also communicate using the shuffle instruction, which permits the access to another thread register inside the same warp.

The SM activity is defined as the time that each SM is active during the execution of a CUDA kernel. It is commonly expressed as an average percentage. A SM is considered active if it has, at least, one warp assigned for execution. A single kernel 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 frequently.

1.3 Contributions and thesis organization

The contributions of this thesis are organized in three chapters. The research presented in each chapter has been published in different papers in some of the most relevant conferences and journals of the field:

Chapter 2: *Acceleration of the DWT in a GPU*, embodies the research published in:

[13] 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

The chapter presents a novel register-based GPU implementation of the DWT. A solution is proposed based on a deep analysis of the algorithm's computational bounds. The implementation achieves speedups of 4, on average, with respect to previous state-of-the-art GPU implementations. New GPU capabilities are analyzed and employed in the method, and the implementation decisions are described and evaluated for different GPU architectures. The research demonstrates that the proposed DWT implementation effectively saturates the GPU resources, achieving close-to-peak performance in the GPU architectures analyzed.

Chapter 3: *Bitplane Image Coding with Parallel Coefficient Processing*, corresponds to the research published in:

[31] 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.

[32] 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*, pp. 163–172, Apr. 2015.

The second contribution of the thesis is focused on the reformulation of the bitplane coding and arithmetic coding techniques to promote the kind of fine-grained parallelism suitable for GPU computing. The paper introduces Bitplane Image Coding with Parallel Coefficient Processing (BPC-PaCo), a novel coding engine that exposes explicit SIMD parallelism, without renouncing to any coding feature of traditional bitplane coders. BPC-PaCo reformulates some of the key mechanisms of the JPEG2000's data coding engine to promote 32 times more parallelism, in exchange for less than 2% compression performance. BPC-PaCo employs a new parallel scanning order, a novel context formation approach, a stationary probability model and the use of multiple arithmetic coders within the codeblock. The paper evaluates the coding performance loss of each of the proposed techniques and describes the method, but its computational performance is not analyzed in practice.

Chapter 4: *Implementation of BPC-PaCo in a GPU*, presents the main insights of the research published in:

[33] 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.*, in Press, 2017.

[34] 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*, pp. 345–354, Dec. 2015.

The last contribution of this thesis corresponds to the design and performance evaluation of the GPU implementation of BPC-PaCo. A detailed analysis of BPC-PaCo is presented and employed to devise an optimized implementation of the method. The proposed GPU solution is based on an efficient thread-to-data mapping, a smart memory management, and the use of sophisticated cooperation mechanisms to implement inter-thread communication. The performance of the codec is compared with the most competitive state-of-the-art bitplane coding implementations. The experimental results indicate that the proposed implementation is up to 30 times faster than the best CPU bitplane codec, while being highly power efficient. Experimental results also demonstrate that BPC-PaCo achieves an order of magnitude better performance than other GPU implementations.

Chapter 2

Acceleration of the DWT in a GPU

The release of the CUDA Kepler architecture in March 2012 provided Nvidia GPUs with a larger register memory space and instructions for the communication of registers among threads. This facilitates a new programming strategy that utilizes registers for data sharing and reusing, in detriment of the shared memory. Such a programming strategy can significantly improve the performance of applications that reuse data heavily. This chapter presents a register-based implementation of the Discrete Wavelet Transform. Experimental results indicate that the proposed method is, at least, four times faster than the best GPU implementation of the DWT found in the literature. Furthermore, theoretical analysis coincide with experimental tests in proving that the execution times achieved by the proposed implementation are close to the GPU's performance limits.

As previously discussed, implementations in GPUs must be carefully realized to fully exploit the potential of the devices. Data management is one of the most critical aspects. The key is to store the data in the appropriate memory spaces. From a microarchitecture point of view, the GPU has three main memory spaces: global, shared, and registers. The shared memory and the registers are located on-chip and can be explicitly managed. They are two orders of magnitude faster than the global memory, but their size is much smaller. The main difference between the shared memory and the registers is that the first is commonly employed to store and reuse intermediate results and to efficiently share data among threads. The registers are private to each thread, and they are many times solely employed to perform the arithmetic and logical operations of the program.

The CUDA performance guidelines [35] recommended the use of the shared memory for data reuse and data sharing. Surprisingly, these recommendations were challenged in November 2008 by Volkov and Demmel [14, 36], who stated that an extensive use of the shared memory may lead to suboptimal performance. This is caused by a combination of three factors:

1. The bandwidth of the shared memory may become a bottleneck in applications that need to reuse data heavily.
2. Arithmetic or logical operations carried out with data located in the shared memory implicitly need to move this data to the registers before performing the operations, which requires additional instructions.
3. The register memory space is commonly larger than that of the shared memory.

In their paper, Volkov and Demmel indicated that the only way to increase the GPU's performance is to directly use the registers, minimizing the use of the shared memory. Their results suggest that maximum performance is achieved when the registers are employed as the main local storage space for data reusing, though the results may vary depending on the algorithm. At that time, there was no operations to share the data in the registers among threads, and the register memory space was very limited. This restrained the use of register-based implementations.

The release of the Kepler CUDA architecture in March 2012 unlocked these restrictions. The size of the register memory space was doubled, the number of registers that each thread can manage was quadruplicated, and a new set of instructions for data sharing in the register space was introduced. These improvements facilitated register-based strategies to program the GPU. This was an emerging approach that significantly enhanced the potential performance of many GPU implementations. In the fields of image processing and computational biology, for example, this trend was employed to achieve very competitive results [15–17].

This chapter explores the use of a register-based strategy to implement the Discrete Wavelet Transform. Realizations of the DWT in GPUs require carefully implemented strategies of data reuse. As seen below, there are many different approaches in the literature. The use of a register-based strategy allows a particular approach that differs from the state-of-the-art methods. The implementation has to be rethought from scratch. The most critical aspects are data partitioning and thread-to-data mapping (see below). The implementation presented herein is the first implementation of the DWT employing a register-based strategy. The proposed method achieves speedups of 4 compared to the previous best implementation found in the literature.

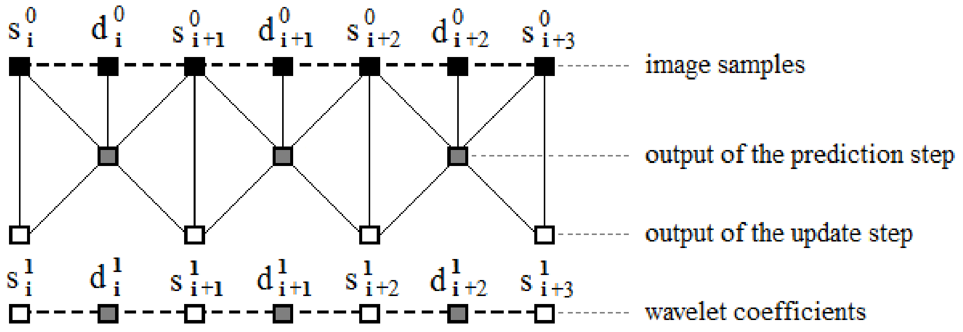
This chapter is structured as follows. Section 2.1 provides a general description of the DWT. Section 2.2 reviews state-of-the-art implementations of the DWT in GPUs. Section 2.3 describes the proposed method detailing the data partitioning scheme employed and its implementation in the GPU. Section 4.4 assesses the performance of the implementation through extensive experimental results. The last section summarizes this work.

2.1 The Discrete Wavelet Transform

The DWT is a signal processing technique derived from the analysis of Fourier. It applies a bank of filters to an input signal that decompose its low and high frequencies. In image coding, the forward operation of the DWT is applied to the original samples (pixels) of an image in the first stage of the encoding procedure. In general, coding systems use a dyadic decomposition of the DWT that produces a multi-resolution representation of the image [18]. This representation organizes the wavelet coefficients in different levels of resolution and subbands that capture the vertical, horizontal, and diagonal features of the image. The decoder applies the reverse DWT in the last stage of the decoding procedure, reconstructing the image samples.

The filter bank employed determines some features of the transform. The most common filter banks in image coding are the irreversible CDF 9/7 and the reversible CDF 5/3 [21], which are employed for lossy and progressive lossy-to-lossless compression, respectively. The proposed method implements these two filter banks since they are supported in JPEG2000, though other banks could also be employed achieving similar results.

The DWT can be implemented via a convolution operation [18] or by means of the



s_i^0 d_i^0 s_{i+1}^0 d_{i+1}^0 s_{i+2}^0 d_{i+2}^0 s_{i+3}^0
image samples
output of the prediction step
output of the update step
s_i^1 d_i^1 s_{i+1}^1 d_{i+1}^1 s_{i+2}^1 d_{i+2}^1 s_{i+3}^1
wavelet coefficients

Figure 2.1: Illustration of the lifting scheme for the forward application of the reversible CDF 5/3 transform.

lifting scheme [37]. The lifting scheme is an optimized realization of the transform that reduces the memory usage and the number of operations performed, so it is more commonly employed. It carries out several steps in a discretely-sampled one-dimensional signal, commonly represented by an array. Each step computes the (intermediate) wavelet coefficients that are assigned to the even, or to the odd, positions of the array. Each coefficient is computed using three samples: that in the even (or odd) position of the array, and its two adjacent neighbors. Such a procedure can be repeated several times depending on the filter bank employed. An important aspect of the lifting scheme is that all coefficients in the even (or odd) positions can be computed in parallel since they do not hold dependencies among them.

Formally expressed, the lifting scheme is applied as follows. Let $\{c_i\}$ with $0 \leq i < I$ be the original set of image samples. First, c_i is split into two subsets that contain the even and the odd samples, referred to as $\{d_i^0\}$ and $\{s_i^0\}$, respectively, with $0 \leq i < I/2$. The arithmetic operations are performed in the so-called prediction and update steps. As seen in Fig. 2.1, the prediction step generates the subset $\{d_i^1\}$ by applying to each sample in $\{d_i^0\}$ an arithmetic operation that involves d_i^0, s_i^0, and s_{i+1}^0. This operation is generally expressed as

$$d_i^{j+1} = d_i^j - \alpha^j (s_i^j + s_{i+1}^j) \,. \tag{2.1}$$

The update step is performed similarly, producing the subset $\{s_i^{j+1}\}$ that is computed according to

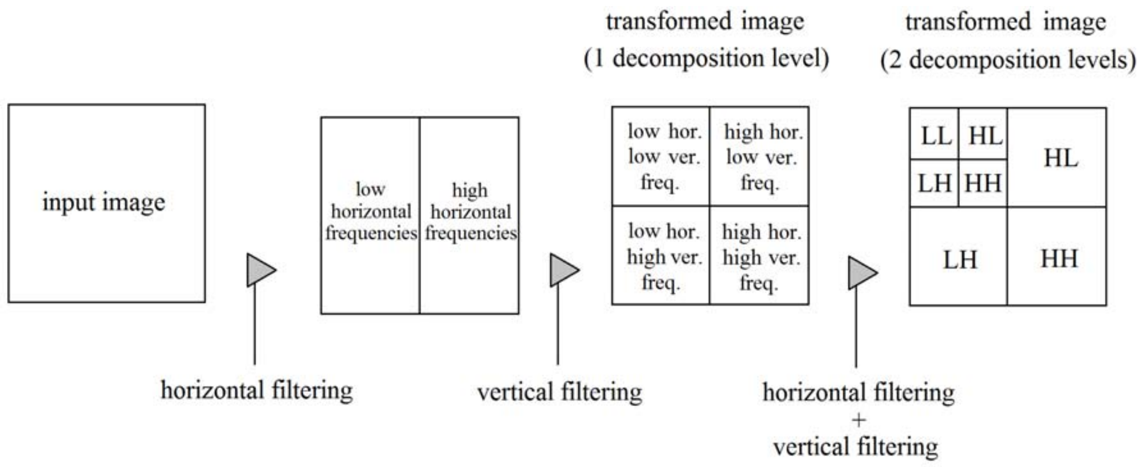


Figure 2.2: Application of two levels of DWT decomposition to an image.

$$s_i^{j+1} = s_i^j + \beta^j (d_i^{j+1} + d_{i+1}^{j+1}) \ . \tag{2.2}$$

Depending on the wavelet filter bank, (2.1) and (2.2) may be repeated several times. The result of these steps are the subsets $\{d_i^J\}$ and $\{s_i^J\}$, which contain the low and high frequencies of the original signal, respectively, with J denoting the number of iterations performed. α^j and β^j in the above equations depend on the filter bank and change in each step j. The 5/3 transform has $J = 1$ whereas the 9/7 has $J = 2$. The reverse application of the transform applies the same procedure but it swaps additions for subtractions.

The application of the lifting scheme to an image is carried out in two stages. First, the lifting scheme is applied to all rows of the image, which is called horizontal filtering. Then, it is applied to the resulting coefficients in a column-by-column fashion in the so-called vertical filtering. The order in which the horizontal and the vertical filtering are applied does not matter as far as the decoder reverses them appropriately. As seen in Fig. 4.2, these filtering stages produce a dyadic decomposition that contains four subsets of coefficients called wavelet subbands. Subbands are commonly referred to as LL, HL, LH, and HH, with each letter denoting Low or High frequencies in the vertical and horizontal direction. The size of the LL subband is one quarter of that of the original image. Its content is similar to the original image, so coding systems commonly generate new levels of decomposition by applying the DWT to the resulting LL subband. Fig. 4.2 depicts this scheme when two levels of decomposition are applied to an image. The reverse DWT applies the inverse procedure starting at the last level of decomposition. In general, five levels of decomposition are enough to achieve maximum decorrelation.

2.2 Previous and related work

The pre-CUDA GPU-based implementations of the DWT employed manifold devices and programming languages.The implementation proposed in [38], for instance, was based on OpenGL, whereas [1, 39] employed OpenGL and Cg. Most of these earliest methods used convolution operations and were tailored to each filter bank. [1] evaluated for the first time the use of the lifting scheme, though the convolution approach was preferred because the lifting requires the sharing of intermediate values among coefficients. At that time there were no tools to implement that efficiently. This was experimentally confirmed in [2], in which both the convolution and the lifting approach were implemented.

The aforementioned pre-CUDA implementations were constrained by the lack of a general-purpose GPU architecture and its programming tools. The operations of the DWT had to be mapped to graphics operations, which are very limited. Though these works accelerated the execution of the DWT with respect to a CPU-based implementation, their performance is far from that achieved with current GPUs that have an enhanced memory hierarchy and support general-purpose computing. Key in current CUDA implementations is how the image is partitioned to permit parallel processing. Fig. 2.3 illustrates the three main schemes employed in the literature. They are named row-column, row-block, and block-based.

The first DWT implementation in CUDA was proposed in [27]. It employs the row-column scheme. First, a thread block loads a row of the image to the shared memory and the threads compute the horizontal filtering on that row. After the first filtering stage, all rows of the image are returned to the global memory, in which the image is stored as a matrix. After transposing it, the same procedure is applied, with the particularity that in this second stage the rows are in reality the columns of the original image, so the vertical filtering is actually executed (see Fig. 2.3(a)). Even though this work still uses convolution operations, speedups between 10 to 20 compared to a multi-core OpenMP

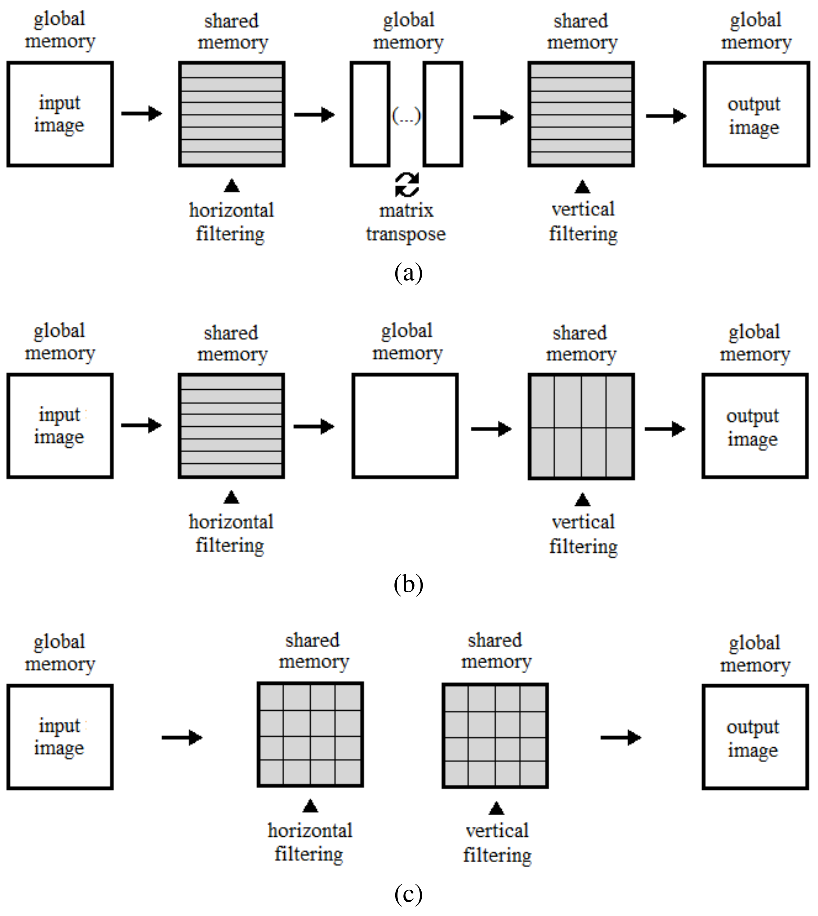


Figure 2.3: Illustration of the a) row-column, b) row-block, and c) block-based partitioning schemes to allow parallel processing in the GPU. Horizontal arrows indicate the main data transfers to/from the global memory.

implementation are achieved. Its main drawback is the matrix transpose, which is a time-consuming operation. A similar strategy is utilized in [5] and [28] for other types of wavelet transform.

The first CUDA implementation based on the lifting scheme was presented in [3] employing the block-based scheme. The main advantage of this scheme is that it minimizes transfers to the global memory since it computes both the horizontal and the vertical filtering in a single step. It partitions the image in rectangular blocks that are loaded to the shared memory by a thread block. Both the horizontal and the vertical filtering are applied in these blocks, neither needing further memory transfers nor a matrix transpose. The only drawback of such an approach is that there exist data dependencies among adjacent blocks. In [3] these dependencies are not addressed. The common solution to avoid them is to extend all blocks with some rows and columns that overlap with adjacent blocks. These extended rows/columns are commonly called halos.

The fastest implementation of the DWT found in the literature is that proposed in [8], in which the row-block scheme is introduced. The first step of this scheme is the same as that of the row-column, i.e., it loads rows of the image to the shared memory to apply the horizontal filtering on them. Then, the data are returned to the global memory. The second step is similar to what the block-based scheme does. It partitions the image in vertically stretched blocks that are loaded to the shared memory. Consecutive rectangular blocks in the vertical axis are processed by the same thread block employing a sliding window mechanism. This permits the thread block to reuse data in the borders of the blocks, handling the aforementioned problem of data dependencies. The only drawback of such a scheme is that it employs two steps, so more accesses to the global memory are required. The speedups achieved by [8] are approximately from 10 to 14 compared to a CPU implementation using MMX and SSE extensions. They also compare their implementation to convolution-based implementations and to the row-column scheme. Their results suggest that the lifting scheme together with the row-block partitioning is the fastest. The implementation of [8] is employed in the experimental section below for comparison purposes.

Other works in the literature implement the DWT in specific scenarios. [7] employs it in a real-time SPIHT decoding system that uses Reed-Solomon codes. The partitioning scheme used is similar to the row-block but without the sliding window, which forces the reading of more data from the global memory. [9] utilizes a block-based scheme for the compression of hyperspectral images. [10, 11] examines the convolution approach again, whereas [12] implements a variation of the DWT.

Regardless of the partitioning scheme employed, all works in the literature store each partition of the image in the shared memory and assign a thread block to compute each one of them. This is the conventional programming style recommended in the CUDA programming guidelines.

2.3 Proposed method

2.3.1 Analysis of the partitioning scheme

As most modern implementations, our method employs the lifting scheme. Contrarily to previous work, the proposed approach stores the data of the image partitions in the registers –rather than in the shared memory. It is clear in the literature that the row-column scheme is the slowest [3, 8]. Also, the analysis in [8] indicates that the block-based scheme may not be effective due to its large need of shared memory. This is the main reason behind the introduction of the row-block scheme in [8]. Nonetheless, that analysis is for CUDA architectures prior to Kepler. So it is necessary to study the differences between the row-block and the block-based scheme in current architectures to decide which is adopted in our method. The following analysis assesses memory accesses, computational overhead, and task dependencies.

Both the row-block and the block-based schemes permit data transfers from/to the global memory via coalesced accesses. The main difference between them is the number of global memory accesses performed. The row-block requires the reading and writing of the image (or the LL subband) twice. All data are accessed in a row-by-row fashion in the first step. After the horizontal filtering, the data are returned to the global memory. The whole image is accessed again using vertically stretched blocks to perform the vertical filter. For images with a large width, it may be more efficient to divide the rows in slices that are processed independently due to memory requirements. Data dependencies among the first and the last samples of the slice have to be handled appropriately. This may slightly increase the number of accesses to the global memory, though not seriously so. A similar issue may appear with images with a large height.

Let m and n respectively be the number of columns and rows of the image. When the row-block scheme is applied, the computation of the first level of decomposition requires, at least, two reads and two writes of all samples to the global memory, i.e., $4mn$ accesses. In general, the application of L levels of wavelet decomposition requires $4M$ accesses, with M being

$$M = \sum_{k=0}^{L-1} \frac{m \cdot n}{4^k} \; . \tag{2.3}$$

Contrarily to the row-block, the block-based scheme reuses the data after applying the horizontal filtering. If it were not for the data dependencies that exist on the borders of the blocks, this partitioning scheme would require $2M$ accesses, half those of the row-block scheme. To address these dependencies, the partitioning has to be done so that the blocks include some rows and columns of the adjacent blocks, the so-called halo. The size of the halo depends on the lifting iterations of the wavelet filter bank (i.e., J). Let $\hat{m}$ and $\hat{n}$ denote the number of columns and rows of the block –including the halo. The application of an iteration of the lifting scheme in a sample involves dependencies with 2 neighboring coefficients. For the reversible CDF 5/3 transform (with $J = 1$), for instance, these dependencies entail two rows/columns on each side of the block. The samples in these rows/columns are needed to compute the remaining samples within the block, but they must be disregarded in the final result.[1] The number of samples computed without dependency conflicts in each block is $(\hat{m} - 4J) \cdot (\hat{n} - 4J)$. The ratio between the size of the block with and without halos is determined according to

$$H = \frac{\hat{m} \cdot \hat{n}}{(\hat{m} - 4J) \cdot (\hat{n} - 4J)} \; . \tag{2.4}$$

Since the halos have to be read but not written to the global memory, the number of global memory accesses required by this partitioning scheme is $HM + M$.

Table 2.1 evaluates the number of memory accesses needed by the row-block and block-based partitioning schemes for different block sizes and wavelet transforms. The row-block scheme always requires $4M$ accesses. For the block-based scheme, the larger the block size, the fewer the accesses to the global memory, with the lower bound at $2M$. Except for blocks of 16×16 and the use of the 9/7 transform, the number of accesses required by the block-based scheme is always lower than for the row-block scheme. So compared to the row-block scheme, results of Table 2.1 suggest that the block-based scheme can reduce the

[1]Sides of blocks that coincide with the limits of the image do not have rows/columns with data dependencies. The number of samples corresponding to this is negligible for images of medium and large size, so it is not considered in the discussion for simplicity.

Table 2.1: Evaluation of the number of accesses to the global memory required by two partitioning schemes. The row-block scheme requires the same number of accesses regardless of the lifting iterations and block size.

block size	row-block	block-based	
($\hat{m} \times \hat{n}$)		CDF 5/3	CDF 9/7
16×16		$2.78M$	$5M$
32×32		$2.31M$	$2.78M$
64×64	$4M$	$2.14M$	$2.31M$
128×128		$2.07M$	$2.14M$
64×20		$2.33M$	$2.90M$

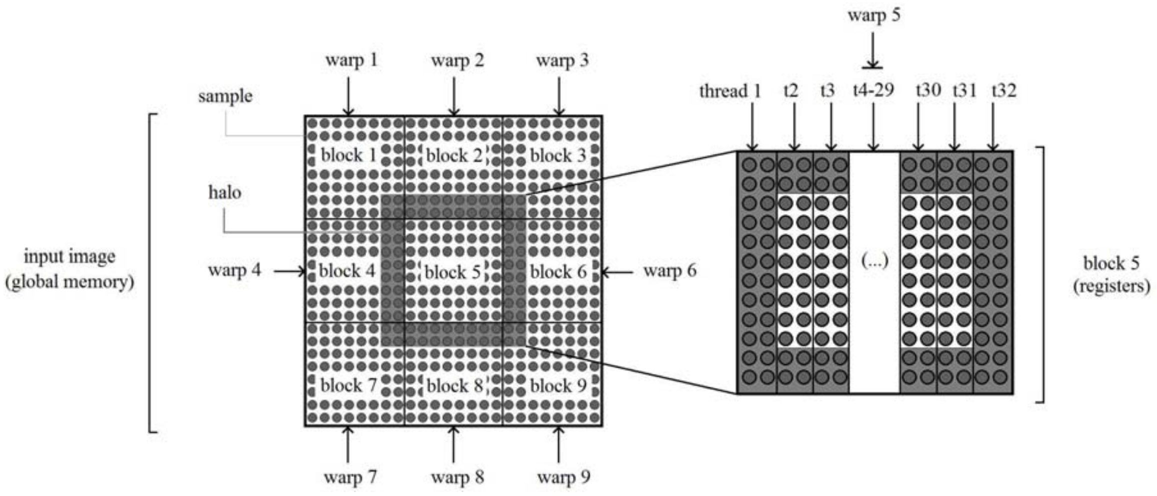


Figure 2.4: Illustration of the partitioning scheme and the thread-to-data mapping employed by the proposed method when applying a CDF 5/3 transform.

execution time devoted to the memory accesses in a similar proportion. Furthermore, the accesses corresponding to the halos may be accelerated by means of the on-chip caches.

The evaluation reported in Table 2.1 is theoretical. The following test evaluates the real execution time that is devoted to the memory accesses achieved by the block-based strategy. In this artificial test none logical or arithmetic operation is performed. A warp is assigned to read and write the block data from/to the global memory to/from the registers. Blocks are of 64×20 since this block size fits well our implementation. Results hold for other block sizes too. The same experiment is carried out with and without using the aforementioned halos. Evidently, the writing of data to the global memory is carried out

Table 2.2: Evaluation of the practical and theoretical increase in computational time due to the inclusion of halos in each block. Blocks of size 64×20 are employed. Experiments are carried out with a Nvidia GTX TITAN Black.

	image size	exec. time (in μs)		real	theor.
	($m \times n$)	no halos	halos	inc.	inc.
CDF 5/3	1024×1024	17	23	1.35	1.17
	2048×2048	63	74	1.17	
	4096×4096	245	279	1.14	
	8192×8192	981	1091	1.11	
CDF 9/7	1024×1024	19	34	1.8	1.45
	2048×2048	67	116	1.73	
	4096×4096	255	444	1.74	
	8192×8192	1015	1749	1.72	

only for the relevant samples when halos are used. When no halos are utilized, $2M$ accesses are performed, whereas the use of halos performs $HM + M$ accesses as mentioned earlier.

Table 2.2 reports the results achieved when using different image sizes, for both the reversible CDF 5/3 (with $J = 1$) and the irreversible CDF 9/7 (with $J = 2$) transform. The theoretical increase in memory accesses (i.e., $(HM + M)/2M$) due to the use of halos for this experiment is 1.17 and 1.45, respectively for the 5/3 and 9/7 transform. The experimental results suggest that the theoretical analysis is approximately accurate for the 5/3 transform, especially for images of medium and large size. Contrarily, the real increase for the 9/7 transform is larger than the theoretical. This is caused because the writing of samples is *not* done in a fully coalesced way. The threads assigned to the halo hold irrelevant data, so they are idle when writing the results. In spite that the real increase is higher than the theoretical, we note that it is always below 2, which is the point at which the row-block scheme would be more effective than the block-based.

Another aspect that may be considered when analyzing these partitioning schemes is the arithmetic operations that are performed. The row-block applies the lifting scheme to all image coefficients (or to those in the LL subband) once, so it performs λM operations, with λ denoting the number of arithmetic operations needed to apply the lifting scheme to each coefficient. Samples within the halos in the block-based scheme compel the application of the lifting scheme in some coefficients more than once. Again, the increase can be determined through the percentage of samples within the halos in each block, resulting in λHM. Although the block-based scheme performs more arithmetic operations, in practice this becomes inconsequential since the performance of the DWT implementation is bounded by the memory accesses (see next section).

The final aspect of this analysis studies the task dependencies of the algorithm. There are two task dependencies that must be considered. They are the application of the horizontal and vertical filtering, and the application of the prediction and update steps. In both cases, the tasks have to be applied one after the other. The steps dependency can be handled in both partitioning schemes with local synchronization within each thread block. The horizontal-vertical filtering dependency has different constrains in each partitioning scheme. The row-block needs to synchronize all the thread blocks after each filter pass. This can only be implemented via two kernels that are executed sequentially. The block-based scheme does not require synchronization among different thread blocks since all data is within the block, so local synchronization can be employed. This is generally more effective than executing two sequential kernels.

2.3.2 Thread-to-data mapping

The analysis of the previous section indicates that the block-based partitioning scheme requires fewer global memory accesses and that it can be implemented employing effective synchronization mechanisms. The proposed method uses a scheme similar to the block-based. Besides storing the data of the image partitions in the registers, another important difference with respect to previous works is that each partition is processed by a warp instead of using a thread block. This strategy does not need shared memory since threads within a warp can communicate via shuffle instructions. It also avoids the use of synchronization operations required by inter-lifting dependencies since the threads in a warp are intrinsically synchronized and there is no need to communicate data between warps. The removal of all synchronization operations elevates the warp-level parallelism.

Fig. 2.4 illustrates the partitioning strategy employed. The rectangular divisions of the image represent the samples within each block that can be computed with*out* dependency conflicts. The surrounding gray rows/columns of block 5 in the figure represent the real extension (including the halo) of that block. For the 5/3 transform, two rows/columns are added to each side to resolve data dependencies. The real size of all other blocks in the figure also includes two rows/columns in each side, though it is not illustrated for clarity.

The size of the block directly affects the overall performance of the implementation since it has impact on the memory access pattern, the register usage, the occupancy, and the total number of instructions executed. Key to achieve maximum efficiency is that the threads in a warp read and write the rows of the block performing coalesced accesses to the global memory. To achieve it, the block width must be a multiple of the threads within a warp, which is 32 in current architectures. To assign pairs of samples to each thread is highly effective. The processing of two adjacent samples per thread permits the application of the lifting scheme first for samples in the even positions of the array and then for samples in the odd positions. We note that to assign only one sample per thread would generate divergence since only half the threads in a warp could compute the (intermediate) wavelet coefficients. So the width of the block that we employ is $\hat{m} = 64$. This is illustrated in the right side of Fig. 2.4.

In our implementation, each thread holds and processes all pairs of samples of two consecutive columns. With such a mapping, the application of the vertical filtering does not require communication, whereas the horizontal filtering requires collaboration among threads. With this mapping, the threads collaboratively request $\hat{n}$ rows from the global memory before carrying out any arithmetic operation. This generates multiple on-the-fly requests to the global memory, key to hide the latency of the global memory since arithmetic operations are then overlapped with memory accesses.

The height of the block permits some flexibility. Fig. 2.5 reports the results that are achieved by the proposed method when employing different block heights. Results are for images of different size and for the forward CDF 5/3 transform, though they hold for other wavelet filter banks and for the reverse application of the transform. The results in this figure indicate that the lowest execution times are achieved when the height of the block is between 12 to 26, approximately.

Table 2.3 extends the previous test. It reports the registers employed by each thread, the device occupancy, and the number of instructions executed when applying the proposed method to an image of size 7168×7168 employing blocks of different heights. We assure

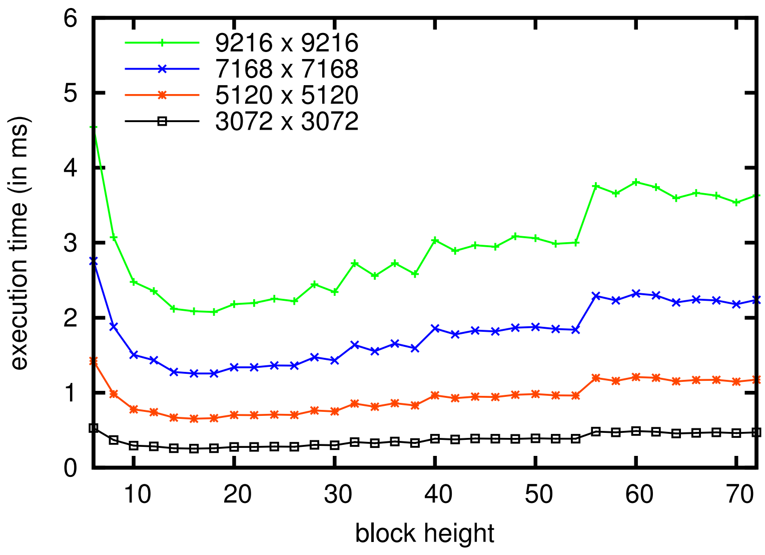


Figure 2.5: Evaluation of the execution time achieved by the proposed method when employing blocks of different height. 5 decomposition levels of forward CDF 5/3 wavelet transform are applied to images of different size. Experiments are carried out with a Nvidia GTX TITAN Black.

that register spilling does not occur in any of these, and following, tests. The smaller the block height, the fewer registers used per thread and the higher the occupancy of the device. Then, more instructions are executed due to larger halos. As seen in Fig. 2.5, the tradeoff between the device occupancy and the number of instructions executed is maximized with blocks of 64 $\times$ 20, approximately. For this block size, the occupancy of the device is 45% and the number of instructions executed is much lower than when using blocks of 64 $\times$ 10. These results hold for the CDF 9/7 transform and for images of other sizes. The results of the next section employ a block size of 64×20.

2.3.3 Algorithm

Algorithm 1 details the CUDA kernel implemented in this work for the forward application of the wavelet transform. We recall that a CUDA kernel is executed by all threads in each warp identically and synchronously. The parameters of the algorithm are the thread identifier (i.e., T), the first column and row of the image corresponding to the block processed by the current warp (i.e., X, Y), and the first column and row of the wavelet subbands in

Table 2.3: Evaluation of some GPU metrics when the proposed method is applied to an image of size 7168×7168 employing different block heights, for the forward CDF 5/3 transform. Experiments are carried out with a Nvidia GTX TITAN Black.

block height	registers used	device occupancy	instructions executed (x10^3)
10	34	69%	70655
20	57	45%	45153
30	78	33%	39749
40	97	22%	37106
50	121	22%	35599
60	141	16%	34604
70	161	16%	34222

which the current warp must leave the resulting coefficients (i.e., X_S, Y_S). The height of the block is denoted by $\bar{Y}$ and is a constant.

The first operation of Algorithm 1 reserves the registers needed by the thread. The registers are denoted by $\mathcal{R}$, whereas the global memory is denoted by $\mathcal{G}$. From line 2 to 5, the thread reads from the global memory the two columns that it will process. The reading of two consecutive columns can be implemented with coalesced accesses to the global memory. The reading of all data before carrying out any arithmetic operation generates the aforementioned on-the-fly memory accesses.

The horizontal filtering is carried out in lines 6-13 as specified in Eq. (2.1) and (2.2) employing that α^j and β^j corresponding to the wavelet filter bank. Since this filtering stage is applied along each row, the threads must share information among them. The operation $\Phi(\cdot)$ in lines 8,10 is the shuffle instruction introduced in the CUDA Kepler architecture. This operation permits thread T to read a register from any other thread in the warp. The register to be read is specified in the first parameter of this function. The second parameter of $\Phi(\cdot)$ is the thread identifier from which the register is read.

The vertical filtering is applied in the loops of lines 14-23. In this case, the thread has all data needed to apply it, so the prediction step is carried out first in the loop of lines 15-18 followed by the update step. Note that in the horizontal filtering, the prediction and update steps were carried out within the same loop since all threads process the same row simultaneously.

Algorithm 1 Forward DWT kernel

Parameters:
T thread with $T \in [0, 31]$
X, Y first column and row of the block in the image
X_S, Y_S first column and row of the block in the S subband

```
1: allocate $\mathcal{R}[\bar{Y}][2]$ in register memory space
2: for $y \in \{0, 1, 2, ..., \bar{Y} - 1\}$ do
3:     $\mathcal{R}[y][0] \leftarrow \mathcal{G}[Y + y][X + T * 2]$
4:     $\mathcal{R}[y][1] \leftarrow \mathcal{G}[Y + y][X + T * 2 + 1]$
5: end for
6: for $j \in \{0, 1, 2, ..., J - 1\}$ do
7:     for $y \in \{0, 1, 2, ..., \bar{Y} - 1\}$ do
8:         $\mathcal{R}' \leftarrow \Phi(\mathcal{R}[y][0], T + 1)$
9:         $\mathcal{R}[y][1] \leftarrow \mathcal{R}[y][1] - \alpha^j(\mathcal{R}[y][0] + \mathcal{R}')$
10:        $\mathcal{R}' \leftarrow \Phi(\mathcal{R}[y][1], T - 1)$
11:        $\mathcal{R}[y][0] \leftarrow \mathcal{R}[y][1] - \beta^j(\mathcal{R}[y][1] + \mathcal{R}')$
12:    end for
13: end for
14: for $j \in \{0, 1, 2, ..., J - 1\}$ do
15:    for $y \in \{1, 3, 5, ..., \bar{Y} - 1\}$ do
16:        $\mathcal{R}[y][0] \leftarrow \mathcal{R}[y][0] - \alpha^j(\mathcal{R}[y-1][0] + \mathcal{R}[y+1][0])$
17:        $\mathcal{R}[y][1] \leftarrow \mathcal{R}[y][1] - \alpha^j(\mathcal{R}[y-1][1] + \mathcal{R}[y+1][1])$
18:    end for
19:    for $y \in \{0, 2, 4, ..., \bar{Y} - 2\}$ do
20:        $\mathcal{R}[y][0] \leftarrow \mathcal{R}[y][0] - \beta^j(\mathcal{R}[y-1][0] + \mathcal{R}[y+1][0])$
21:        $\mathcal{R}[y][1] \leftarrow \mathcal{R}[y][1] - \beta^j(\mathcal{R}[y-1][1] + \mathcal{R}[y+1][1])$
22:    end for
23: end for
24: for $y \in \{2J, 2J + 2, ..., \bar{Y} - 2J\}$ do
25:    $\mathcal{G}[Y_{LL} + y/2][X_{LL} + T] \leftarrow \mathcal{R}[y][0]$
26:    $\mathcal{G}[Y_{HL} + y/2][X_{HL} + T] \leftarrow \mathcal{R}[y][1]$
27: end for
28: for $y \in \{2J + 1, 2J + 3, ..., \bar{Y} - 2J + 1\}$ do
29:    $\mathcal{G}[Y_{LH} + y/2][X_{LH} + T] \leftarrow \mathcal{R}[y][0]$
30:    $\mathcal{G}[Y_{HH} + y/2][X_{HH} + T] \leftarrow \mathcal{R}[y][1]$
31: end for
```

The last two loops in Algorithm 1 (lines 24-31) write the resulting coefficients in the corresponding wavelet subbands stored in the global memory. In this case, accesses to the global memory are not fully coalesced as mentioned earlier. These loops only transfer the rows that do not belong to the halos. Our implementation also takes into account that the threads containing the first and last columns of the block do not write their coefficients in the global memory since they have dependency conflicts, though it is not shown in Algorithm 1 for simplicity.

This algorithm details the forward application of the wavelet transform for one decomposition level. The application of more decomposition levels carries out the same procedure but taking the resulting LL subband of the previous decomposition level as the input image. Also, the reverse operation of the transform is implemented similarly as the procedure

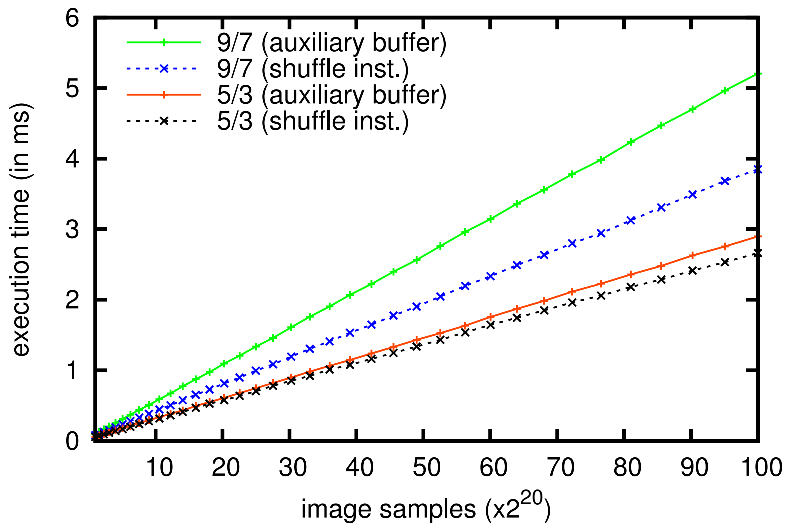


Figure 2.6: Evaluation of the execution time achieved by the proposed method when employing shuffle instructions or an auxiliary buffer in the shared memory to communicate data among threads. Five decomposition levels of forward CDF 5/3 or 9/7 wavelet transform are applied to images of different size. Experiments are carried out with a Nvidia GTX TITAN Black.

specified in Algorithm 1.

Although the proposed method has been devised for the Kepler CUDA architecture and following, Algorithm 1 could also be employed in previous architectures. Before Kepler, the data sharing among threads in a warp could be implemented by using an auxiliary buffer in the shared memory. By only replacing the shuffle instructions in lines 8,10 by the use of this auxiliary buffer, our register-based strategy could be employed in pre-Kepler architectures. This strategy is employed in the next section to assess the performance achieved with GPUs of the Fermi architecture. Evidently, the shuffle instruction is faster than the use of an auxiliary buffer due to the execution of fewer instructions. See in Fig. 2.6 the execution time spent by the proposed method when employing shuffle instructions or the auxiliary buffer. Shuffle instructions accelerate the execution of the 9/7 transform in approximately 20%.

On another note, the proposed method can also be employed to perform strategies of wavelet transformation that involve three dimensions in images with multiple components, such as remote sensing hyperspectral images or 3D medical images. The conventional

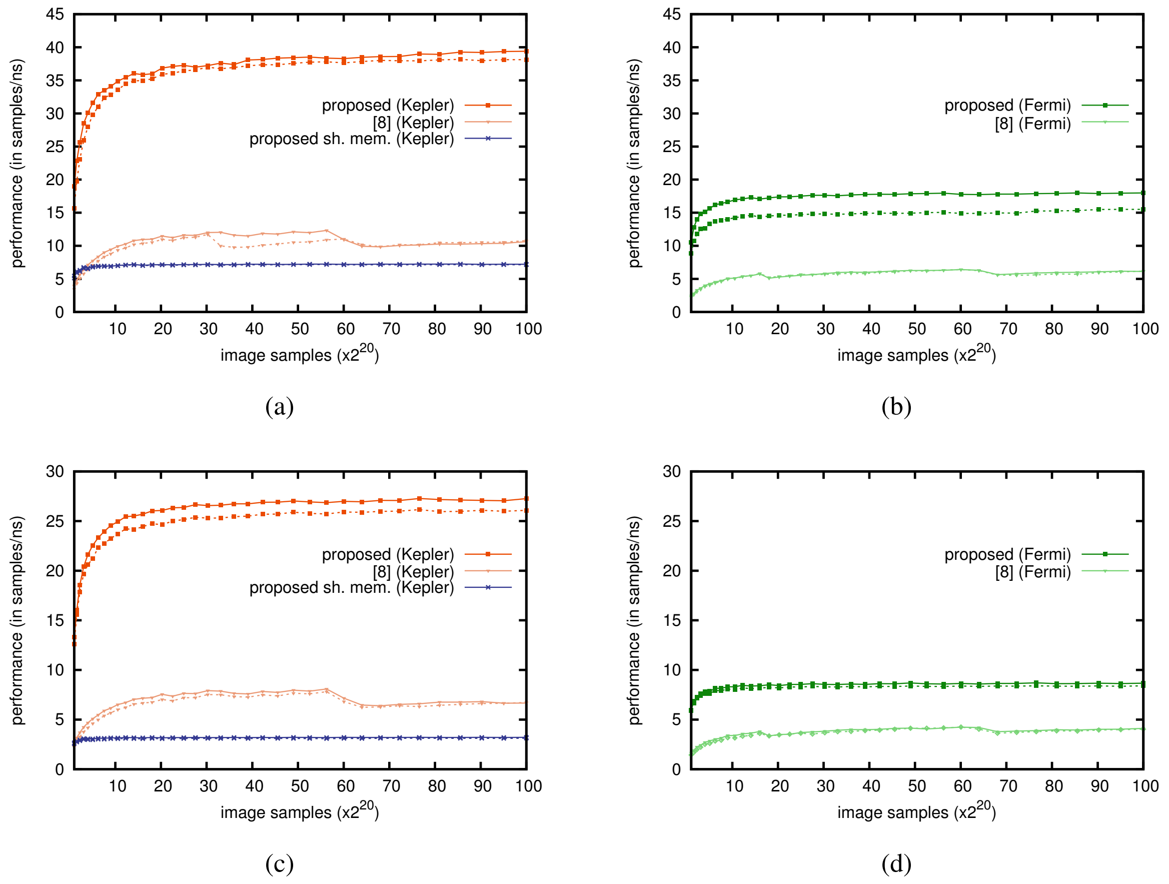


Figure 2.7: Evaluation of the performance achieved by the proposed method and [8], for (a),(b) the CDF 5/3 transform and (c),(d) the CDF 9/7 transform. Solid lines indicate the forward application of the transform and dashed lines indicate the reverse.

way to apply such strategies is to reorder the original samples and apply the DWT afterwards [40].

2.4 Experimental results

Except when indicated, the experimental results reported in this section are carried out with a Nvidia GTX TITAN Black GPU using the CUDA v5.5 compiler. This GPU has 15 SMs and a peak global memory bandwidth of 336 GB/s. Results have been collected employing the Nvidia profiler tool nvprof. All the experiments apply five levels of decomposition to images of size ranging from 1024×1024 to 10240×10240. The data structures employed to store the image samples are of 16 bits. Floats of 32 bits are employed to perform all computations of the CDF 9/7 since they provide enough arithmetic precision for image coding applications.

The first test evaluates the performance achieved by the proposed method and compares it to the best implementation found in the literature [8]. The implementation in [8] is configured to obtain maximum performance in this GPU using the maximum shared memory size. Fig. 2.7(a) and (c) depict the results achieved for both the reversible CDF 5/3 and the irreversible CDF 9/7 wavelet transforms. The horizontal axis of the figures is the size of the image, measured as the number of image samples, whereas the vertical axis is the performance. The metric employed to evaluate the performance is the number of samples processed per unit of time. The plots with the label "proposed" depict the performance of our method when the data of the blocks are stored in the registers. To compare the performance achieved by the use of registers vs the use of shared memory, these figures also report the performance achieved with the GTX TITAN Black when our implementation uses a buffer in the shared memory to hold all the data.[2] To assess the increase in performance achieved with different Nvidia architectures, Fig. 2.7(b) and (d) depict the results achieved with a Tesla M2090 GPU (Fermi architecture). Data blocks of size 64×20 and thread blocks of 128 are employed for all implementations. In our implementation, thread blocks of 128 achieve the best results. Thread blocks of 64 may obtain slightly better performance in some applications, especially when using Maxwell architectures (and onwards). In our case the differences between blocks of 128 and 64 are negligible. As seen in Fig. 2.7, both the forward and the reverse application of the wavelet transform achieve similar performance since they perform practically the same operations in the inverse order.

The experimental results of Fig. 2.7 indicate that the performance speedup between Fermi and Kepler achieved by [8] is approximately 1.7 for both the CDF 5/3 and 9/7 transform. The performance speedup achieved by our implementation is 2.3 and 3, respectively for the 5/3 and 9/7. This difference is because our implementation exploits very efficiently the resources of the Kepler architecture. Furthermore, note that the performance achieved

[2]Such a strategy does *not* explicitly use the registers, so all data are kept in the shared memory. It is configured to avoid bank conflicts in the shared memory and to employ the maximum shared memory size, maximizing performance.

Table 2.4: Evaluation of the total number of instructions executed and global memory accesses performed by the proposed method and by the implementation in [8] (Kepler architecture). Results are for the forward transform.

		instructions executed (x10^3)					mem. accesses (x10^3)		
	image size	proposed	[8]	incr.	proposed (sh. mem.)	incr.	proposed	[8]	incr.
CDF 5/3	1024×1024	982	3554	3.62	1618	1.64	208	348	1.67
	2048×2048	3804	12350	3.25	6224	1.63	822	1395	1.70
	4096×4096	14926	44541	2.98	24357	1.63	3287	5587	1.70
	8192×8192	59 K	167 K	2.83	96 K	1.63	13 K	22 K	1.69
CDF 9/7	1024×1024	2026	5370	2.65	3743	1.84	210	366	1.74
	2048×2048	7847	18921	2.41	14456	1.84	825	1396	1.69
	4096×4096	31200	69266	2.22	57513	1.84	3313	5588	1.69
	8192×8192	124 K	263 K	2.12	229 K	1.84	13 K	23 K	1.67

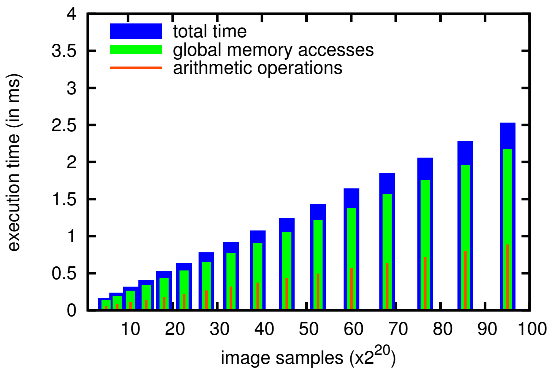
by our implementation when using the Tesla M2090 GPU (Fermi architecture) is even higher (1.4 on average) than that of [8] when using the GTX TITAN Black (Kepler architecture). When comparing the results achieved by both implementations with the Kepler architecture, the results of Fig. 2.7 show that the proposed register-based implementation is significantly faster than the method presented in [8]. Though it depends on the wavelet transform and the size of the input data, speedups ranging approximately from 3.5 to almost 5 are achieved. The gain in performance is caused by the novel programming methodology based on the use of registers, which results in a lower number of instructions executed and a lower number of global memory accesses. This can be seen in Table 2.4. The columns with the label "increase" in this table report the increase ratio in the number of instructions or accesses with respect to the proposed method. The implementation in [8] executes three times more instructions and around 70% more memory accesses than ours. This corresponds with the theoretical analysis of Section 2.3 (Table 2.1).

The results of Fig. 2.7(a) and (c) also indicate that, in our implementation, the use of registers speedups the execution from 3.5 to 9 times with respect to the use of shared memory. As mentioned previously, the use of shared memory significantly decreases the performance due to a low occupancy of the device and the fact that data have to be moved from the shared memory to the registers to perform arithmetic operations. The occupancy achieved by "proposed sh. mem." is 12%, as opposed to the 45% achieved when using registers. The low occupancy achieved by the use of shared memory is constrained by the amount of shared memory assigned per thread block. Though this occupancy could be increased by reducing the data block size, the number of instructions executed is then significantly increased and so the overall performance is reduced. As seen in Table 2.4, when the proposed method employs shared memory instead of registers, the number of instructions is increased in approximately 60% and 80% for the 5/3 and 9/7 transform, respectively. This is due to the operations that move the data from the shared memory to the registers. The number of memory accesses performed by the proposed method is the same regardless of using registers or shared memory, so it is not shown in the table. We note that these results correspond with [8], in which it was already indicated that the block-based scheme employing shared memory is *not* efficient. The use of the proposed register-based strategy enhances the performance of such a scheme greatly.

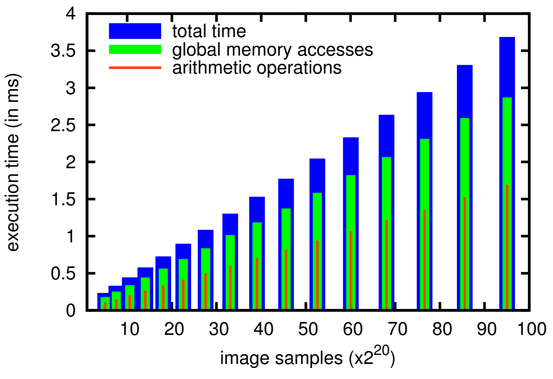
Another observation that stems from Fig. 2.7 is that our method achieves regular performance regardless of the image size, indicating that it scales well with the size of the input data. This is seen in the figure as the almost straight plot of our implementation. Only for small images the performance decreases due to low experimental occupancy.

2.4.1 Analysis of execution bottleneck

The aim of the next test is to identify the execution bottleneck. To this end, it separately evaluates the time spent by the arithmetic operations and the time spent by the global memory accesses in our register-based implementation. Fig. 2.8 depicts the results achieved by the forward application of both the 5/3 and 9/7 transform. Again, the horizontal axis of the figure is the number of image samples, whereas the vertical axis is the execution time. The plot with the label "global memory accesses" reports the time spent by reading and writing all data from/to the global memory in an implementation in which all arithmetic operations are removed. The plot with the label "arithmetic operations" reports the time spent by the arithmetic operations in an implementation in which all memory accesses are removed. The plot with the label "total time" is our original implementation with both



(a)



(b)

Figure 2.8: Evaluation of the execution time spent to perform only the arithmetic operations, accesses to the global memory, and both the arithmetic operations and accesses (total time), for the forward application of the (a) CDF 5/3 transform and (b) the CDF 9/7 transform.

memory accesses and arithmetic operations.

It is worth noting in Fig. 2.8 that the time spent to perform the arithmetic operations is less than that spent for the memory accesses. As it is also observed in the figure, the overlapping of the arithmetic operations with the memory accesses is carried out efficiently. If the arithmetic operations were not overlapped with the memory accesses, the total execution time should be the sum of both. If the overlapping were realized perfectly, the execution time should be the maximum of both. These results indicate that the proposed method is mostly memory bounded, especially for the 5/3 transform. As previously stated, such an overlapping is achieved thanks to the large number of on-the-fly requests to the global memory carried out in our implementation.

As seen in Fig. 2.8 (and also in Fig. 2.7), the performance achieved with the reversible CDF 5/3 transform is approximately 40% higher than that achieved with the irreversible CDF 9/7. This difference is caused by two factors. First, the 9/7 has $J = 2$, so its lifting scheme requires twice the number of arithmetic operations as that of the 5/3 (see Table 2.4). The amount of time required to execute the arithmetic instructions of the 9/7 is also twice that required by the 5/3 (see Fig. 2.8). Thanks to the overlapping of the arithmetic operations with the memory accesses, the total execution time of the 9/7 is *not* doubled. Nonetheless, the overlapping achieved by the 9/7 is not as effective as that achieved by the 5/3. The second factor behind the lower performance achieved with the 9/7 is that the blocks in the 9/7 need larger halos than with the 5/3, requiring more memory accesses. The time spent by the 9/7 transform to carry out the memory accesses is approximately 30% higher than that spent by the 5/3. Even so, in Table 2.4 the number of memory accesses carried out by both transforms is the same because the extra memory accesses (corresponding to the larger halos of the 9/7) are reused in the on-chip cache. In practice, the 9/7 performs 40% more memory accesses to the on-chip cache than the 5/3 (data not shown), increasing the time spent by the memory accesses. We note that large on-chip caches can hold more data reused for the halos, reducing the computational time.

The aim of the next test is to appraise whether our implementation is bounded by the memory bandwidth or by the memory latency. The results of Fig. 2.9 depict the experimentally measured bandwidth. As reported by Nvidia, 100% usage of the peak memory bandwidth is not attainable in practice. The maximum attainable can be approximated by that obtained by the Nvidia SDK bandwidth test. In the GTX TITAN Black, this test achieves an usage of 70%. Our implementation achieves an average bandwidth of 65% and 50% for the 5/3 and 9/7 transform, respectively. These results reveal that the implementation applying the 5/3 transform is bounded by the global memory bandwidth. The 9/7 does not reach the maximum bandwidth usage and so more parallelism (by means of more thread- and instruction-level parallelism) could improve its performance. The reverse application of the transforms achieves lower bandwidth usage due to the more scattered access pattern that they use when reading the image since each warp fetches data from four different subbands.

2.4.2 Evaluation in other devices

The last test evaluates the performance of our register-based implementation in four different GPUs. The features of the employed devices are shown in Table 2.5. The Tesla M2090 has a Fermi architecture, the GTX 680 and GTX TITAN Black have a Kepler architecture, and the GTX 750 Ti has the Maxwell architecture. The four devices have different memory bandwidth. The experimental bandwidth depicted in the table is computed with the Nvidia SDK bandwidth test. The results achieved with these devices can be found in Table 2.6 and

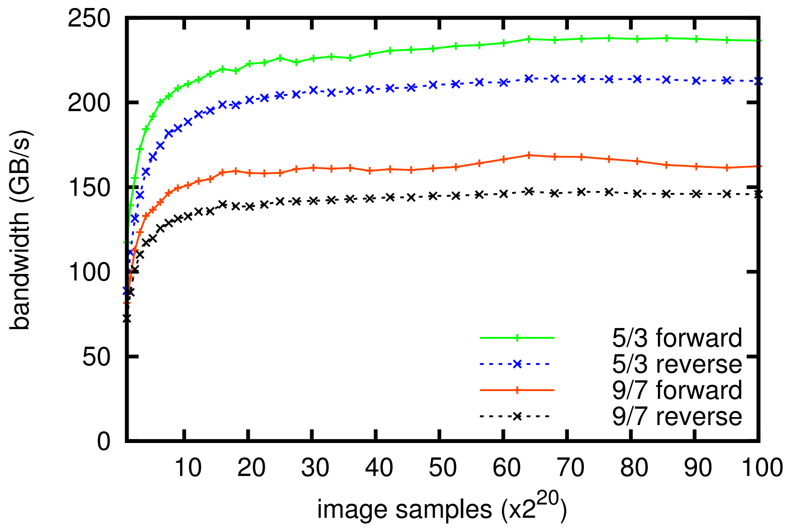


Figure 2.9: Evaluation of the global memory bandwidth usage achieved by the proposed method.

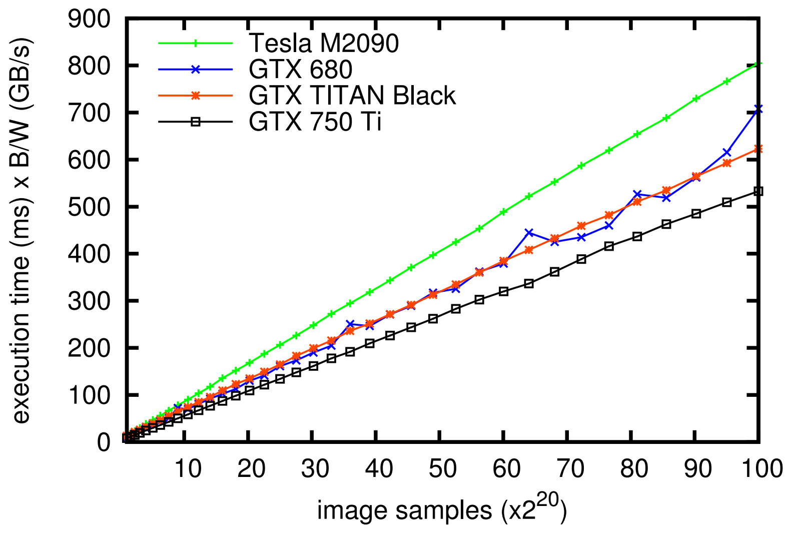


Figure 2.10: Evaluation of the execution time weighted by the experimentally measured global memory bandwidth in different GPUs, for the forward application of the 5/3 transform.

Fig. 2.10. The table reports the execution time for both the 5/3 and 9/7 transform, whereas the figure depicts the execution time multiplied by the global memory bandwidth of each device, for the forward application of the 5/3 transform. As seen in Table 2.5, the execution

Table 2.5: Features of the GPUs employed.

	Tesla M2090	GTX 680	GTX TITAN Black	GTX 750 Ti
compute capability	2.0	3.0	3.5	5.0
clock frequency	1301 MHz	1006 MHz	889 MHz	1020 MHz
SMs	16	8	15	5
number of cores	512	1536	2880	640
register space per SM	128 KB	256 KB	256 KB	256 KB
shared memory per SM	48 KB	48 KB	48 KB	64 KB
size of global memory	6144 MB	2048 MB	6144 MB	2048 MB
memory bandwidth (theoretical)	177.6 GB/s	192.2 GB/s	336 GB/s	86.4 GB/s
memory bandwidth (experimental)	138.11 GB/s	146.4 GB/s	234 GB/s	67.1 GB/s
size of on-chip L2 cache	768 KB	512 KB	1536 KB	2048 KB
peak GFLOPS (single precision)	1331	3090	5121	1306

Table 2.6: Evaluation of the execution time achieved with different GPUs, for the forward application of the 5/3 and 9/7 transform using 5 decomposition levels.

		execution time (in μs)			
	image size	Tesla M2090	GTX 680	GTX TITAN Black	GTX 750 Ti
CDF 5/3	1024×1024	100	70	55	125
	2048×2048	278	204	139	370
	4096×4096	983	698	467	1305
	8192×8192	3782	3038	1741	5021
CDF 9/7	1024×1024	176	97	79	171
	2048×2048	538	282	194	540
	4096×4096	1997	1024	652	1990
	8192×8192	7811	3960	2490	7686

time achieved is mainly related to the memory bandwidth. The GTX TITAN Black has the highest bandwidth and so the lowest execution time, followed by the GTX 680, the GTX 750 Ti, and the Tesla M2090. Despite the differences seen in this table, note in Fig. 2.10 that the performance of both the GTX TITAN Black and the GTX 680 is almost the same considering their difference in the global memory bandwidth. This figure also discloses that the GTX 750 Ti, which has the highest compute capability and the largest on-chip cache, achieves slightly better performance than the remaining devices considering its memory bandwidth. The small irregularities achieved by the GTX 680 in Fig. 2.10 may be because this GPU has the smallest on-chip cache and the fewest number of SMs, which may affect its performance for some image sizes. As seen in the figure, the Tesla M2090 achieves the lowest performance because the Fermi architecture has half the register memory space per SM as that of Kepler and Maxwell architectures, which reduces the device occupancy. Also, because it does not employ shuffle instructions.

2.5 Summary

This research proposes an implementation of the DWT in a GPU using a register-based strategy. This kind of implementation strategy became feasible in the Kepler CUDA architecture due to the expansion of the register memory space and the introduction of instructions to allow data sharing in the registers. The key features of the proposed method are the use of the register memory space to perform all operations and an effective block-based partitioning scheme and thread-to-data mapping that permit the assignment of warps to process all data of a block. Experimental evidence indicates that the proposed register-based strategy obtains better performance than the ones using shared memory, since it requires fewer instructions and achieves higher GPU occupancy.

Experimental analyses suggest that the proposed implementation is memory bounded. The global memory bandwidth achieved is close to the experimental maximum, and most of the computation is overlapped with the memory accesses. Since most of the global memory traffic is unavoidable (i.e., employed to read the input image and to write the output data), we conclude that the execution times achieved by the proposed implementation are close to the limits attainable in current architectures. Compared to the state of the art, our register-based implementation achieves speedups of 4, on average. The implementation employed in this work is left freely available in [41].

Conceptually, the application of the DWT can also be seen as a stencil pattern [26]. Stencils, and other algorithms with similar data reuse patterns, may also benefit from an implementation strategy similar to that described in this work.

Chapter 3

Bitplane Image Coding with Parallel Coefficient Processing

Image coding systems have been traditionally tailored for Multiple Instruction, Multiple Data (MIMD) computing. Bitplane coding techniques are no exception. They partition a (transformed) image in codeblocks that can be efficiently coded in the cores of MIMD-based processors. Unfortunately, current bitplane coding strategies can not fully profit from SIMD processors, such as GPUs, due to its inherently sequential coding task. This chapter presents Bitplane Image Coding with Parallel Coefficient Processing (BPC-PaCo), a coding method that can process many coefficients within a codeblock in parallel and synchronously. The scanning order, the arithmetic coder, the context formation, and the probability model of the coding engine have been re-formulated. Experimental results suggest that the penalization in coding performance of BPC-PaCo with respect to traditional strategies is almost negligible.

Over the past 20 years, the computational complexity of image coding systems has been increased notably. Codecs of the early nineties were based on computationally simple techniques like the discrete cosine transform (DCT) and Huffman coding [42]. Since then, techniques have been sophisticated to provide higher compression efficiency and enhanced features. Currently, image compression standards such as JPEG2000 [19] or HEVC intra-coding [43] employ complex algorithms that transform and scan the image multiple times. This escalation in computational complexity continues in each new generation of coding systems.

In general, modern coding schemes tackle the computational complexity by means of fragmenting the image in sets of (transformed) samples, called codeblocks, that do not hold (or hold in a well-orderly way) dependencies among them. Each codeblock [21], or group of codeblocks [44], can be coded independently from the others employing the innermost algorithms of the codec. These algorithms scan the samples repetitively, producing symbols that are fed to an entropy coder. Key in such a system is the context formation and the probability model, which determine probability estimates employed by the entropy coder. Commonly, the samples are visited in a sequential order so that the probability model can adaptively adjust the estimates as more data are coded. In many image coding systems [23–25, 45, 46], these algorithms employ bitplane coding strategies and context-adaptive arithmetic coders.

Modern CPUs are mainly based on the MIMD principle, and because of this, they handle well the computational complexity of image coding systems. The tasks of the image codec are straightforwardly mapped to the CPU: each codeblock is simply assigned to a core that runs a bitplane coding engine. This parallel processing of codeblocks is called *macro*scopic parallelism [21]. *Micro*scopic parallelism refers to parallel strategies of data coding within a codeblock. There are few such strategies due to the difficulty to unlock the data dependencies that arise when the coefficients are processed in a sequential fashion. Also, because most codecs are tailored for their execution in CPUs, so parallelization in the bitplane coding stage is not appealing. It has not been until recent years that microscopic parallelism has become attractive due to the upraising of GPUs, among other SIMD accelerators.

The fine level of parallelism required for SIMD computing can only be achieved in image coding systems via microscopic parallel strategies. Even so, the current trend is to implement *already developed* coding schemes for their execution in GPUs. GPU implementations of JPEG2000 are found in [4, 9, 47, 48] and there exist commercial products like [49] as well. The JPEG XR standard is implemented in [50], and video coding standards are studied in [51, 52]. Other coding schemes such as EBCOT and wavelet lower trees are also implemented in GPUs in [6] and [53], respectively. Such implementations reduce the execution time of CPU-based implementations. Nonetheless, none of them can fully exploit the resources of the GPU due to the aforementioned sequential coefficient processing.

This chapter introduces Bitplane Image Coding with Parallel Coefficient Processing (BPC-PaCo), a wavelet-based coding strategy tailored for SIMD computing. To this end, a new scanning order, context formation, probability model, and arithmetic coder are devised. All the proposed mechanisms permit the processing of the samples in parallel or sequentially, allowing efficient implementations for both SIMD and MIMD computing. The coding performance achieved by the proposed method is similar to that of JPEG2000. This chapter describes the employed techniques and assesses their performance from an image coding perspective.

This chapter is structured as follows. Section 3.1 presents some preliminary background concepts. Section 3.2 describes the proposed bitplane coding strategy. Section 3.3 assesses its coding performance through experimental results carried out for four different corpora of images. The last section concludes with a brief summary.

3.1 Introduction to Bitplane Coding

The bitplane coding strategy proposed in this chapter can be employed in any wavelet-based compression scheme. We adopt the framework of JPEG2000 due to its excellent coding performance and advanced features. A conventional JPEG2000 implementation is structured in three main coding stages [21]: data transformation, data coding, and codestream re-organization. The first stage applies the wavelet transform and quantizes wavelet coefficients. This represents approximately 15$\sim$20% of the overall coding task and does not pose a challenge for its implementation in SIMD architectures [1,3,8,10,13,27,54]. After data transformation, the image is partitioned in small sets of wavelet coefficients, the so-called codeblocks. Data coding is carried out in each codeblock independently. It represents approximately 70$\sim$75% of the coding task. The routines employed in this stage are based on bitplane coding and context-adaptive arithmetic coding. The last stage re-organizes the final codestream in quality layers that include segments of the bitstreams produced for each codeblock in the previous stage. Commonly, the codestream re-organization is carried out employing rate-distortion optimization techniques [55,56], representing less than 10% of the coding task.

Bitplane coding strategies work 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 υ 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 $M - 1$ to the least significant bitplane 0. The first non-zero bit of the binary representation of υ 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.

JPEG2000 codes each bitplane employing three coding passes [21] called significance propagation pass (SPP), magnitude refinement pass (MRP), and cleanup pass (CP). The SPP and CP perform significance coding. They visit those coefficients that did not become significant in previous bitplanes, coding whether they become significant in the current bitplane or not. The difference between them is that the SPP visits coefficients that are more likely to become significant. The MRP refines the magnitude of coefficients that became significant in previous bitplanes. The order of the coding passes in each bitplane is SPP, MRP, and CP except for the most significant bitplane, in which only the CP is applied. This three coding pass scheme is convenient for rate-distortion optimization purposes [46].

3.2 Proposed bitplane coding strategy

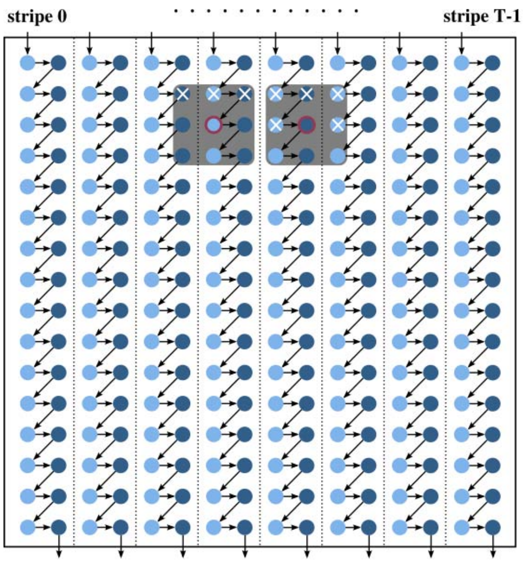
A parallel bitplane coding strategy must be deterministic, i.e., the parallel execution must unambiguously correspond to an equivalent sequential execution. The codestream generated or processed by both the parallel and sequential versions of the algorithm must be the same. Three mechanisms of the bitplane coder have been re-formulated keeping in mind this purpose: the scanning order, the context formation and its probability model, and the arithmetic coder.

3.2.1 Scanning order

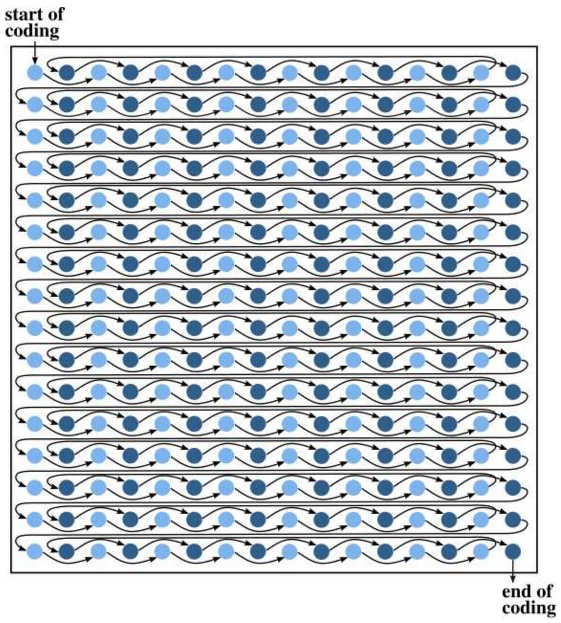
Scanning orders visit coefficients employing a pre-defined sequence. Typical sequences are row by row or column by column [44], in zig zag [57], using stripes of 4 rows that are scanned from left to right [19], or via quadtree strategies [25]. Regardless of the scanning sequence, all methods visit coefficients in a consecutive fashion, which prevents parallelism while executing a coding pass. The only way to achieve microscopic parallelism in current bitplane coding engines is to execute coding passes in parallel. JPEG2000, for instance, provides the RESET, RESTART, and CAUSAL coding variations to achieve it. The main problem of coding pass parallelism is that in order to code a coefficient in the current pass, some information of its neighbors coded in previous passes may be needed. This is addressed by delaying the beginning of the execution of each coding pass some coefficients with respect to its immediately previous pass [21, 58]. Such an elaborate strategy is not suitable for SIMD computing since each coding pass carries out different operations, which generates divergence among threads.

The proposed method achieves microscopic parallelism by means of coding T coefficients in parallel during the execution of a coding pass, where T is the width of an SIMD vector instruction. Sets of T threads perform the same operation to different coefficients, so vector instructions can be naturally mapped to process each codeblock. Fig. 3.1(a) 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 vertical stripes that contain two columns. Each stripe is processed by a thread. Coefficients are scanned from the top to the bottom row, and from the left to the right coefficient. All coefficients in the same position of the stripes are processed at the same time.

The scanning order of Fig. 3.1(a) is highly efficient for context formation purposes. Let us explain further. As seen in the following section, the context of a coefficient is determined via its eight adjacent neighbors. All information coded in previous passes is available when forming the context since such information has been already transmitted to the decoder. Also, information coded in the current coding pass that belongs to those neighbors visited before the current coefficient can also be employed. This information is valuable since it helps to predict with higher precision the symbols coded. The higher the



(a)



(b)

Figure 3.1: Illustration of the proposed scanning order for (a) parallel and (b) sequential processing.

Average number of already Visited Neighbors in the current coding Pass (AVNP), the better the coding performance. The AVNP is computed without considering those coefficients in the border of the codeblock. Fig. 3.1(a) depicts in gray the eight adjacent neighbors of two coefficients, one in the left and the other in the right column of a stripe. The coefficient for which the context is formed is depicted with a red circle. The neighbors that were already visited in the current coding pass are depicted with a white cross. The coefficients in the left column (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. JPEG2000 and other coding systems employing sequential scanning orders also achieve an AVNP of 4.

The sequential version of the proposed scanning order is depicted in Fig. 3.1(b). As seen in the figure, all light-blue coefficients of a row are visited first from left to right, followed by the dark-blue coefficients. The same routine is carried out in each row, from the top to the bottom of the codeblock. Since the parallel operation is synchronous and deterministic, the context formation resulting from the parallel and sequential version of this scanning order is identical. This scanning order does not use fast-coding primitives such as the run mode of JPEG2000 since they do not provide significant coding gains when employed with the proposed probability model [46].

The scanning order of BPC-PaCo is employed with the same coding passes as those defined in JPEG2000. Though other schemes may be utilized, the three-coding pass strategy of JPEG2000 is adopted herein due to its high coding efficiency [46]. The number of significant bitplanes coded for each codeblock is signaled in the headers of the codestream.

3.2.2 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 (i.e., b_s) has already been coded. Clearly, 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 context of υ at bitplane j is computed as

$$\phi_{sig}(\upsilon, j) = \sum_k \Phi(\upsilon^k, j) \; . \tag{3.1}$$

Therefore, $\phi_{sig}(\cdot) \in \{0, ..., 8\}$. Although other works in the literature [24,45,59] determine the context depending on the position of the significance neighbors, the analysis in [60] shows that simple context formation approaches like (3.1) also achieve competitive coding performance. 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

$$\phi_{sign}(\omega, j) = \begin{cases} 0 & \text{if } (\chi^V > 0 \text{ and } \chi^H > 0) \text{ or} \\ & \quad (\chi^V < 0 \text{ and } \chi^H < 0) \\ 1 & \text{if } \chi^V = 0 \text{ and } \chi^H \neq 0 \\ 2 & \text{if } \chi^V \neq 0 \text{ and } \chi^H = 0 \\ 3 & \text{otherwise} \end{cases} \; . \tag{3.2}$$

Contexts for refinement coding should be based on computationally intensive techniques such as the local average, or otherwise use only one context for all refinement bits, as suggested in [60]. Herein, the latter approach is used for computational 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 probability models adaptively adjust the probability estimates of the symbols as more data are coded. Such models are convenient since they are computationally simple, achieve high compression efficiency, and avoid a pre-processing step to collect statistics of the data. Compression standards such as JBIG [61], JPEG2000 [19], and HEVC [43] employ them. Unfortunately, context-adaptive models cannot be employed herein. To do so, the probability adaptation should be carried out for all data of the codeblock, which is not possible due to the parallel processing of coefficients. Such models achieve poor performance when coding short sequences [58], so to use them independently for each stripe is not effective.

The proposed bitplane coder employs a stationary probability model that uses a fixed probability for each context and bitplane. As shown in [58], 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, so there is no need to transmit it. The LUT contains one probability estimate per context and bitplane for each wavelet subband. It is accessed as $\mathcal{P}_u[j][\phi_{\{sig|sign|ref\}}(\cdot)]$, providing the probability of the symbol coded. u denotes the wavelet subband. Note that such a probability model does not need the adaptive probability tables employed in context-adaptive arithmetic coders such as the MQ.

The probability estimates needed to populate the LUTs are determined as follows. Let $F_u(v \mid \phi_{sig}(v, j))$ 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. Let us denote the probability that b_j is 0 during significance coding by $P_{sig}(b_j = 0 \mid \phi_{sig}(v, j))$. This probability is determined from the corresponding pmf according to

$$P_{sig}(b_j = 0 \mid \phi_{sig}(\upsilon, j)) = \frac{\sum\limits_{\upsilon=0}^{2^j-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j))}{\sum\limits_{\upsilon=0}^{2^{j+1}-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j))} = \frac{\sum\limits_{\upsilon=0}^{2^j-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j))}{1} = \sum_{\upsilon=0}^{2^j-1} F_u(\upsilon \mid \phi_{sig}(\upsilon, j)) \,. \tag{3.3}$$

The probability estimates for refinement and sign coding are derived similarly. The LUT is different for each image type since the probability model exploits the fact that the data produced after transforming images of the same type (e.g., natural, medical, etc.) with the same wavelet filter-bank are statistically similar [60, 62, 63]. A more in-depth study on this stationary probability model can be found in [58].

3.2.3 Arithmetic coder

The symbol and its probability estimate are fed to an arithmetic coder. Conventional arithmetic coding works as follows. The coder begins by segmenting the interval of real numbers $[0, 1)$ into two subintervals. The size of the subintervals is chosen according to the probability estimate of the symbol. The first symbol is coded by selecting its corresponding subinterval. Then, this procedure is repeated within the selected subintervals for the following symbols. The transmission of any number within the range of the final subinterval guarantees that the reverse procedure decodes the original message losslessly. The number transmitted is generally referred to as codeword.

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. Roughly described, this is done as follows. Let $[L, R)$ denote the current interval of the coder, with L and R being the fractional part of the left and right boundaries of the interval stored in hardware registers. Assume that the leftmost bits of the binary representations of L and R are not equal in the current interval. When a new symbol is coded, this interval is further reduced to $[L', R')$. If the leftmost bits of L' and R' are then equal, all following segmentations of the interval will also start with those same bit(s) since $L \leq L' \leq \ldots \leq R' \leq R$. This permits to dispatch the leftmost bits of L' and R' that are identical and to shift the remaining bits of the registers to the left. This procedure is called renormalization.

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. Part of this complexity is due to the renormalization procedure, which requires conditionals and repositioning operations as explained before.

These aspects are addressed herein by means of a new technique that employs multiple arithmetic coders that work in parallel and generate fixed-length codewords that are optimally positioned in the bitstream. As previously described, each thread codes all data of a stripe. 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 for a stripe. Instead of using conventional arithmetic coding, we employ an arithmetic coder that generates codewords of fixed length [64–68]. Variable-to-fixed length arithmetic coding avoids renormalization, reducing the complexity of the coder [68]. 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).

The codewords produced in each stripe are sorted generating a *single* quality-embedded bitstream for all stripes that can be truncated at any point so that the quality of the recovered image is maximized. Such a bitstream is similar to that produced by conventional image codecs, so it can be employed in the same framework of rate-distortion optimization defined in JPEG2000 to construct layers of quality and/or different progression orders [21]. In the

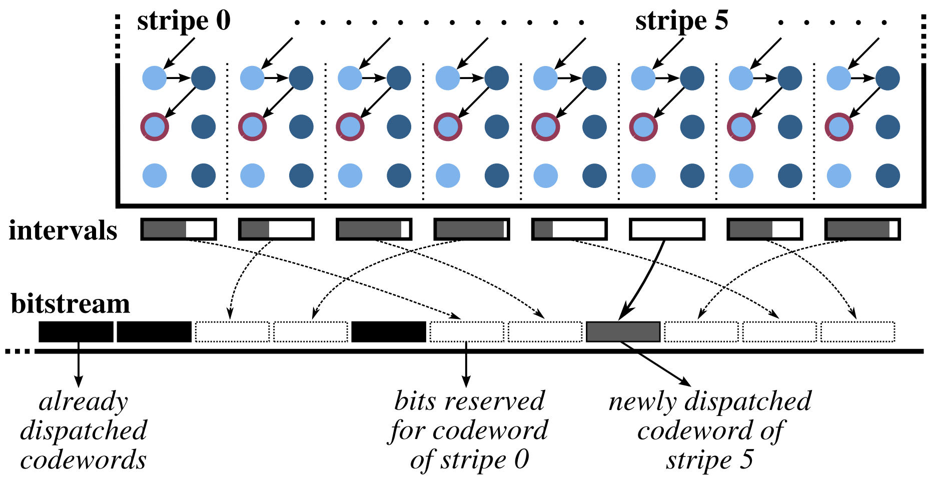

stripe 0
stripe 5
intervals
bitstream
already dispatched codewords
bits reserved for codeword of stripe 0
newly dispatched codeword of stripe 5

Figure 3.2: Illustration of the sorting technique employed to situate the codewords in the bitstream when encoding.

encoder, the bitstream is constructed as follows. Each time that a thread initializes its interval (because is the beginning of coding or because the interval is exhausted and a new symbol needs to be coded), W bits are reserved at the end of the bitstream. This space is reserved –but it is not filled– at this instant because the interval of the thread has just been initialized, so the codeword is still not available. After coding some symbols (possibly from different coding passes), the interval of this thread is exhausted, so its codeword is put in the reserved space. Fig. 3.2 illustrates an example of this sorting technique. All stripes in the figure have its own space in the bitstream, which was reserved when needed. The coefficients depicted with a red circle are those currently visited. When the thread processing the fifth stripe emits its symbol, it exhausts its interval, so the codeword is put in the space that was reserved for this thread. Note that this thread does *not* reserve a new space at the end of the bitstream at this instant but it will do it when coding a new symbol. Evidently, if two or more threads need to reserve space at the same instant, some priority must be employed. In order to provide determinism, stripes on the left have higher priority. When the coding of the codeblock data finishes, the arithmetic coders put their codewords in the bitstream, without needing a byte flush operation.

As previously stated, the order in which the codewords are sorted minimizes the distortion at any truncation point. This can be seen from the perspective of the decoder. All the threads need a non-exhausted interval to decode the data of their corresponding stripes. The first thread that –while decoding– exhausts its interval stops the whole decoding procedure for that codeblock since all threads are synchronized. The codewords are sorted so that, at any instant of the decoding, the thread that exhausts its interval and needs to decode a new symbol can found its immediately next codeword at the immediately next position of the bitstream. In other words, 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 to be decoded. This decodes the maximum amount of data for any given segment of the bitstream, thus the distortion of the reconstructed coefficients is minimized.

The proposed arithmetic coding technique slightly penalizes the coding performance with respect to an implementation that produces a single codeword. This is because either if the bitstream is truncated for rate-distortion optimization purposes, or if it is fully transmitted, the last codeword that is read for each stripe may contain some bits that are not really needed to decode the data of the corresponding coding pass. Since the proposed strategy utilizes T stripes, these excess bits may not be negligible. The penalization in coding performance decreases as more data are coded in each stripe. We found that the coding of two columns is a good tradeoff between coding performance and parallelism. Evidently, the implementation of the proposed method in hardware architectures such as FPGAs would require the replication of the arithmetic coder. Replication is a common strategy to obtain high performance codecs [69].

3.2.4 Algorithm

The encoding procedures of BPC-PaCo are embodied in Algorithm 2. One procedure per coding pass is specified. These procedures detail the operations carried out for a stripe. 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 its row and column within 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 of stripe t is stored in registers $L[t]$ and $S[t]$, which are the left 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. Note that when $S[t] = 0$, $L[t]$ represents the final number within the interval or, in other words, the emitted codeword. If a new symbol is coded and $S[t] = 0$, the procedure reserves W bits and sets $L[t] \leftarrow 0$ and $S[t] \leftarrow 2^W - 1$ (see lines 1-5).

The interval division is carried out in lines 6-12. When the symbol is 0 or $-$, the lower subinterval is kept, so the interval size is reduced to

$$S[t] \leftarrow (S[t] \cdot p) \gg \widehat{\mathcal{P}} \,, \tag{3.4}$$

and $L[t]$ is left unmodified. $\gg$ above denotes a bit shift to the right. p is the probability of the symbol to be $0/+$ expressed in the range $[0, 2^{\widehat{\mathcal{P}}} - 1]$, determined according to

$$p = \lfloor P_{sig}(b_j = 0 \mid \phi_{sig}(\upsilon, j)) \cdot 2^{\widehat{\mathcal{P}}} \rfloor \tag{3.5}$$

for significance coding, and equivalently for refinement and sign coding. $\lfloor \cdot \rfloor$ denotes the floor operation. As seen in Algorithm 2, p is the value that is stored in the LUTs, so (3.5) is computed off-line. $\widehat{\mathcal{P}}$ is the number of bits employed to express the symbol's probability. The result of the multiplication in (3.4) (i.e., $(S[t] \cdot p)$) must not cause arithmetic overflow in the hardware registers, so $W + \widehat{\mathcal{P}} \leq 64$ in modern architectures. Experimental evidence indicates that $16 \leq W \leq 32$ and $\widehat{\mathcal{P}} \geq 7$ achieve competitive performance. In our implementation $W = 16$ and $\widehat{\mathcal{P}} = 7$.

Algorithm 2 BPC-PaCo encoding procedures
Initialization: $S[t] \leftarrow 0 \;\; \forall \;\; 0 \leq t < T$

SPP (u subband, j bitplane, t stripe)

```
1:  for $y \in [0, \text{numRows} - 1]$ do
2:      for $x \in [t \cdot 2, t \cdot 2 + 1]$ do
3:          if $\upsilon_{y,x}$ is not significant AND $\phi_{sig}(\upsilon_{y,x}, j) \neq 0$ then
4:              ACencode($b_j$, $\mathcal{P}_u[j][\phi_{sig}(\upsilon_{y,x}, j)]$, $t$)
5:              if $b_j = 1$ then
6:                  ACencode($d$, $\mathcal{P}_u[j][\phi_{sign}(\omega_{y,x}, j)]$, $t$)
7:              end if
8:          end if
9:      end for
10: end for
```

MRP (u subband, j bitplane, t stripe)

```
1:  for $y \in [0, \text{numRows} - 1]$ do
2:      for $x \in [t \cdot 2, t \cdot 2 + 1]$ do
3:          if $\upsilon_{y,x}$ is significant in $j' > j$ then
4:              ACencode($b_j$, $\mathcal{P}_u[j][\phi_{ref}(\upsilon_{y,x}, j)]$, $t$)
5:          end if
6:      end for
7:  end for
```

CP (u subband, j bitplane, t stripe)

```
1:  for $y \in [0, \text{numRows} - 1]$ do
2:      for $x \in [t \cdot 2, t \cdot 2 + 1]$ do
3:          if $\upsilon_{y,x}$ is not significant AND not coded in SPP then
4:              ACencode($b_j$, $\mathcal{P}_u[j][\phi_{sig}(\upsilon_{y,x}, j)]$, $t$)
5:              if $b_j = 1$ then
6:                  ACencode($d$, $\mathcal{P}_u[j][\phi_{sign}(\omega_{y,x})]$, $t$)
7:              end if
8:          end if
9:      end for
10: end for
```

ACencode (c symbol, p probability, t stripe)

```
1:  if $S[t] = 0$ then
2:      Reserve the next $W$ bits of the bitstream
3:      $L[t] \leftarrow 0$
4:      $S[t] \leftarrow 2^W - 1$
5:  end if
6:  if $c = 0$ OR $c = -$ then
7:      $S[t] \leftarrow (S[t] \cdot p) \gg \widehat{\mathcal{P}}$
8:  else
9:      $f \leftarrow ((S[t] \cdot p) \gg \widehat{\mathcal{P}}) + 1$
10:     $L[t] \leftarrow L[t] + f$
11:     $S[t] \leftarrow S[t] - f$
12: end if
13: if $S[t] = 0$ then
14:     Put $L[t]$ in reserved space of the bitstream
15: end if
```

The coding of $1/+$ keeps the upper subinterval, so

$$\begin{aligned} L[t] &\leftarrow L[t] + ((S[t] \cdot p) \gg \widehat{\mathcal{P}}) + 1 \text{ , and} \\ S[t] &\leftarrow S[t] - ((S[t] \cdot p) \gg \widehat{\mathcal{P}}) - 1 \text{ .} \end{aligned} \tag{3.6}$$

The interval division is carried out via integer multiplications and bit shifts because these are the fastest operations in hardware architectures. Also, because floating point arithmetic should be avoided to prevent incompatibilities with different architectures. An alternative to (3.4), (3.6) is the use of LUTs that contain the result of these operations with relative precision, similarly as how it is done in [21,42,70–73]. Our implementation employs the above operations since they are faster than any other alternative tested.

Algorithm 3 BPC-PaCo relevant decoding procedures
Initialization: $S[t] \leftarrow 0 \ \forall \ 0 \leq t < T$

```
ACdecode ($p$ probability, $t$ stripe)
1: if $S[t] = 0$ then
2:     $I[t] \leftarrow$ read the next $W$ bits of the bitstream
3:     $S[t] \leftarrow 2^W - 1$
4:     $L[t] \leftarrow 0$
5: end if
6: $f \leftarrow ((S[t] \cdot p) \gg \widehat{\mathcal{P}}) + 1$
7: $g \leftarrow L[t] + f$
8: if $I[t] \geq g$ then
9:     $c \leftarrow 1$ OR $+$
10:    $S[t] \leftarrow S[t] - f$
11:    $L[t] \leftarrow g$
12: else
13:    $c \leftarrow 0$ OR $-$
14:    $S[t] \leftarrow f - 1$
15: end if
16: return $c$
```

The decoding procedures of the SPP, MRP, and CP are similar to those of the encoder, so they are not detailed. Algorithm 3 describes the decoding procedure of the arithmetic coder. In this procedure, $I[t]$ is the codeword read from the bitstream for stripe t. The procedure is similar to that of the encoder. An extended description of the arithmetic coder employed in Algorithms 2 and 3 can be found in [68].

The sequential version of BPC-PaCo carries out the same instructions detailed above except that the two loops in lines 1 and 2 of the coding passes are replaced by loops that implement the scanning order depicted in Fig. 3.1(b). The call to “ACencode” or “ACdecode” replaces t by $x/2$, so that each stripe employs a different interval. Also, sign coding is computed slightly different. In the parallel version, it is carried out just after emitting bit b_j. In the sequential version, the sign can *not* be emitted just after b_j since that would produce a different bitstream from that obtained by the parallel algorithm. When the coefficients are coded sequentially, sign coding for the odd (even) coefficients must be carried out just before starting the significance coding of the even (odd) coefficients of the same (next) row. This is necessary to ensure that the codewords are sorted in the bitstream identically in both versions of the algorithm.

The replacement of the original algorithms of JPEG2000 by the proposed bitplane coding strategy does not sacrifice any feature of the coding system. The formation of quality layers, the use of different progression orders, the region of interest coding, or the scalability of the system is unaffected by the use of the proposed strategy.

3.3 Experimental Results

Four corpora of images are employed to assess the performance of BPC-PaCo. The first consists of the eight natural images of the ISO 12640-1 corpus (2048×2560, gray scale, 8 bits per sample (bps)). 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). The third corpus has three xRay angiography images from the medical community (512×512 with 15 components, 12 bps). The last corpus contains three AVIRIS (Airbone Visible/Infrared Imaging Spectrometer) hyperspectral images provided by NASA (512×512 with 224 components, 16 bps). BPC-PaCo is implemented in the framework of JPEG2000 by replacing the bitplane coding engine and the arithmetic coder of a conventional JPEG2000 codec. The resulting codestream is not compliant with JPEG2000, though it does not undermine any feature of the standard. Our implementation BOI [74] is employed in these experiments. Except when indicated, the coding parameters for all tests are: 5 levels of wavelet transform, codeblocks of 64×64, single quality layer, and

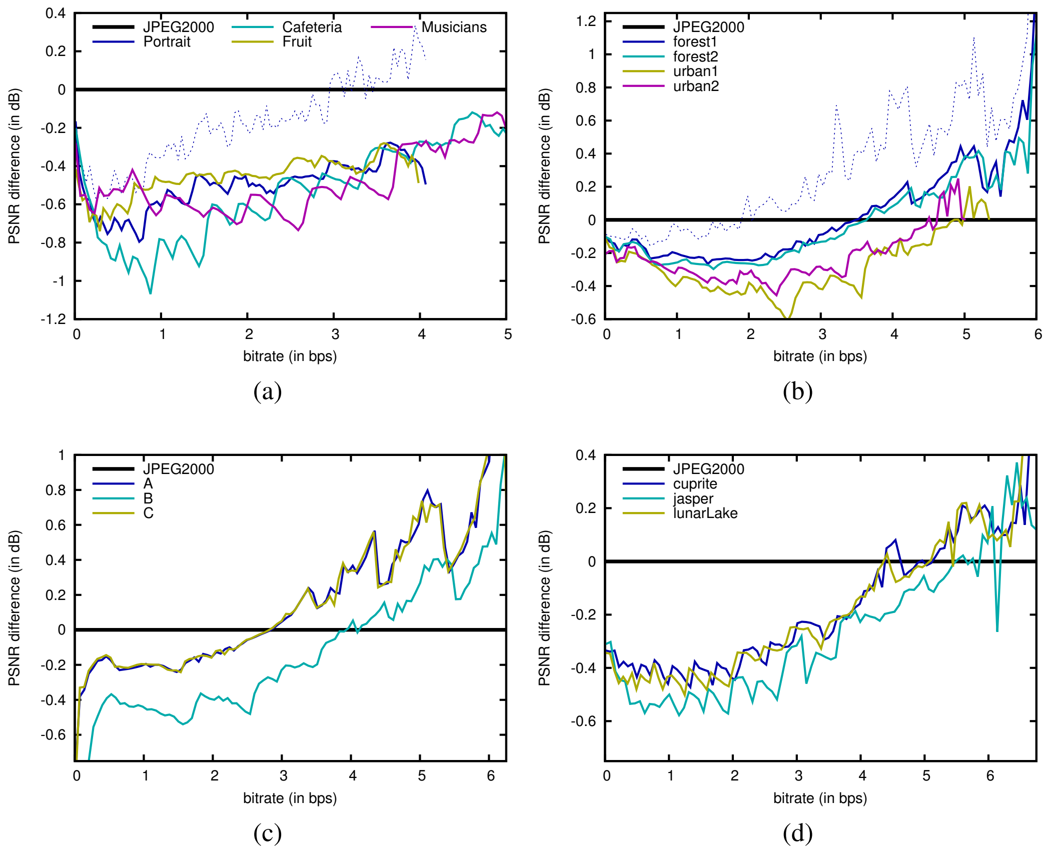


Figure 3.3: Evaluation of the lossy coding performance achieved by BPC-PaCo compared to that of JPEG2000. Each subfigure reports the performance achieved for images from a specific corpus: (a) natural, (b) aerial, (c) xRay, and (d) AVIRIS.

no precincts. The 9/7 and the 5/3 wavelet transforms are employed for lossy and lossless regimes, respectively. BPC-PaCo employs the same rate-distortion optimization techniques as those of JPEG2000, which select the coding passes of each codeblock included in the final codestream.

The first test evaluates the coding performance achieved by BPC-PaCo as compared to that of JPEG2000. Fig. 3.3 depicts the results achieved for the four corpora. 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 BPC-PaCo 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. 3.3(a), though similar plots are achieved for the remaining The results of Fig. 3.3 indicate that, for natural images, the proposed method achieves PSNR values between 0.2 to 1 dB below those of JPEG2000. As it is explained in the previous section and analyzed below, this penalization is mainly due to the use of multiple arithmetic coders. 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 the corpus of xRay and AVIRIS images, the results are similar to those obtained for aerial images.

For comparison purposes, Fig. 3.3(a) and 3.3(b) also report the results when the RESET, RESTART, and CAUSAL coding variations of JPEG2000 are in use when coding the first image of the natural and aerial corpus (i.e., “Portrait” and “forest1”). The results are reported with the plot with dots. We recall that these coding variations are employed to enable coding pass parallelism in JPEG2000 (see Section 3.2.1). When they are in use, the coding performance difference between BPC-PaCo and JPEG2000 is reduced between 0.2 to 0.5 dB.

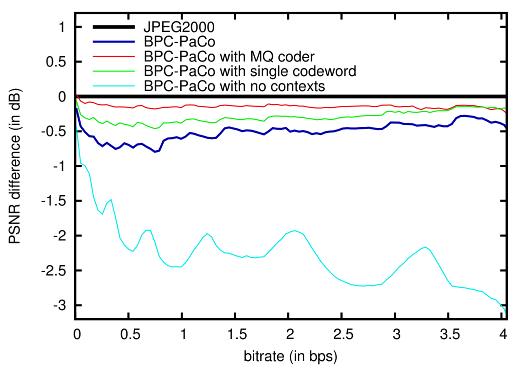
Table 3.1 reports the results achieved when coding all images in lossless mode. The third column of the table reports the bitrate achieved by JPEG2000, in bps. The fourth column reports the bitrate difference between the proposed method and JPEG2000. Again, BPC-PaCo achieves slightly lower and higher compression efficiency than that of JPEG2000 for the corpus of natural images and for the remaining corpora, respectively. On average, BPC-PaCo increases the length of the codestream negligibly.

The aim of the next test is to appraise three key mechanisms of the proposed bitplane coding strategy. To this end, three modifications are carried out to BPC-PaCo. The first replaces its arithmetic coder and utilizes the MQ coder of JPEG2000. The MQ coder employs context-adaptive mechanisms and produces a single codeword for all data coded in a codeblock. The second modification compels the arithmetic coder of BPC-PaCo to employ a single codeword for all stripes. Evidently, these two modifications prevent parallelism. Their sole purpose is to appraise the coding efficiency of these two mechanisms. The third modification removes the context formation approach and employs one context for significance coding, one for refinement coding, and one for sign coding. Fig. 3.4 reports the results obtained for one image of each corpus when these modifications are in use. For

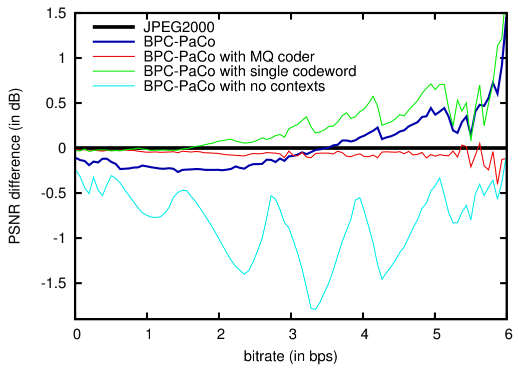
Table 3.1: Evaluation of the lossless coding performance achieved by BPC-PaCo and JPEG2000. Results are reported in bps. The three rightmost columns report the results achieved when variations in BPC-PaCo are employed.

				BPC-PaCo with		
	image	**JP2**	**BPC-PaCo**	**MQ**	**single cwd. AC**	**no ctx.**
ISO 12640-1	"Portrait"	4.38	+0.09	+0.02	+0.06	+0.45
	"Cafeteria"	5.28	+0.08	+0.04	+0.05	+0.49
	"Fruit"	4.29	+0.17	+0.01	+0.14	+0.47
	"Wine"	4.57	+0.16	+0.02	+0.13	+0.43
	"Bicycle"	4.37	+0.20	+0.04	+0.16	+0.57
	"Orchid"	3.58	+0.24	+0.01	+0.21	+0.59
	"Musicians"	5.56	+0.11	+0.02	+0.07	+0.47
	"Candle"	5.65	+0.08	+0.04	+0.04	+0.58
aerial	"forest1"	6.20	-0.04	+0.01	-0.08	+0.12
	"forest2"	6.28	-0.05	+0.01	-0.09	+0.13
	"urban1"	5.54	+0.01	+0.02	-0.03	+0.21
	"urban2"	5.20	+0.03	+0.01	0.00	+0.29
xRay	"A"	6.37	-0.07	0.00	-0.12	0.00
	"B"	6.48	-0.03	0.00	-0.11	+0.02
	"C"	6.35	-0.06	0.00	-0.11	+0.02
AVIRIS	"cuprite"	7.00	-0.03	+0.01	-0.07	+0.41
	"jasper"	7.66	-0.04	+0.02	-0.08	+0.48
	"lunarLake"	6.91	-0.02	+0.01	-0.05	+0.46
	average	*5.65*	*+0.05*	*+0.02*	*+0.01*	*+0.34*

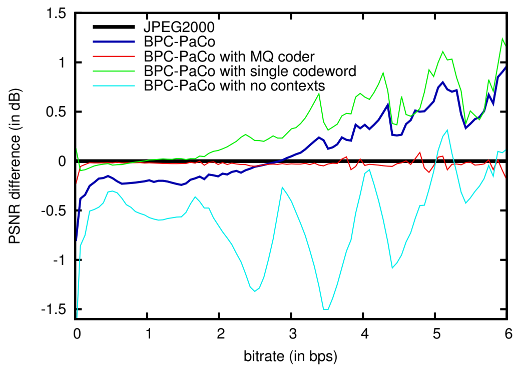
comparison purposes, the figure also reports the performance achieved by the original BPC-PaCo. When the MQ coder is employed, the coding performance achieved by BPC-PaCo is almost the same as that of JPEG2000 for all images. This indicates that the scanning order and the context formation employed in BPC-PaCo do not penalize coding performance significantly. Clearly, the use of multiple arithmetic coders producing multiple codewords is the technique mainly responsible for the penalization in compression efficiency. This can also be seen in Fig. 3.4 via the second modification of BPC-PaCo, which employs a single



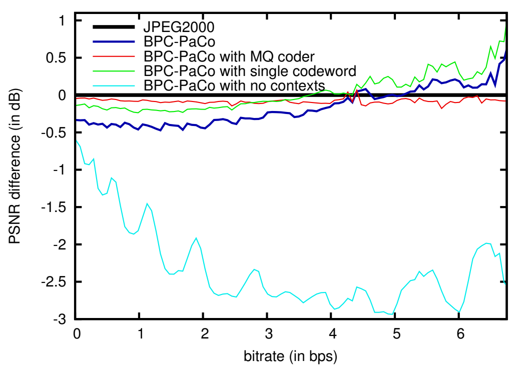
(a)



(b)



(c)



(d)

Figure 3.4: Evaluation of the lossy coding performance achieved by BPC-PaCo when three modifications are employed. Each subfigure reports the performance achieved for one image of a specific corpus: (a) natural image "Portrait", (b) aerial image "forest1", (c) xRay image "A", and (d) AVIRIS image "cuprite".

codeword for all stripes. When this modification is in use, the coding performance of BPC-PaCo is enhanced from 0.25 to 0.5 dB, achieving higher PSNR than that of JPEG2000 for all corpora except the natural. The third modification shows that the proposed context formation approach enhances the coding performance of the proposed method significantly (more than 3 dB in some cases). The results of these modifications are also reported in Table 3.1 for the lossless regime. Similar results are achieved for both lossy and lossless regimes.

The last test evaluates an interesting feature of the proposed method. Vector instructions

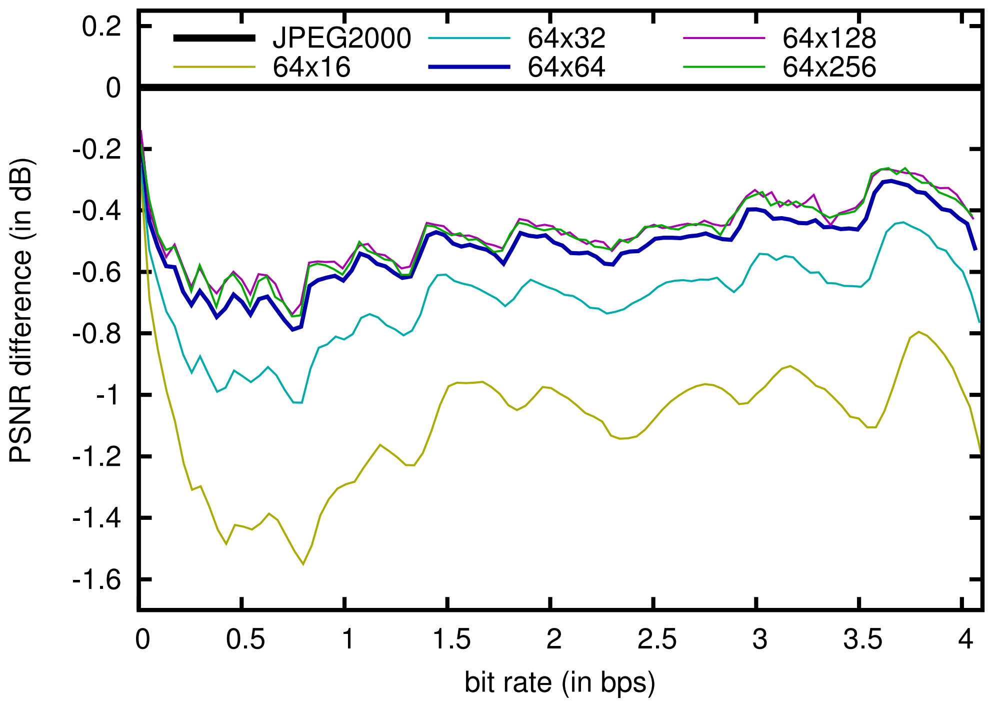


Figure 3.5: Evaluation of the lossy coding performance achieved by BPC-PaCo and JPEG2000 when using different sizes of codeblock. Results are reported for the "Portrait" image of the ISO 12640-1 corpus.

are commonly composed of 32 lanes.[1] Each thread codes a stripe containing two columns, so the use of codeblocks with 64 columns is convenient. The number of rows of the codeblock, on the other hand, strongly influences the coding performance achieved. This is because the more data coded in a stripe, the fewer the excess bits stored in its codewords. This is illustrated in Fig. 3.5 for the natural image "Portrait". Results for codeblocks of 64 columns and a variable number of rows are reported (both JPEG2000 and BPC-PaCo use the same variable codeblock size). The results suggest that the more rows the codeblock has, the better the coding performance. In general, codeblocks of 64$\times$64 already achieve competitive performance while exposing a large degree of parallelism. Results hold for the other images of the corpus and the other corpora.

[1]All Nvidia GPUs, for instance, currently implement vector instructions of 32 lanes.

3.4 Summary

The computational complexity of modern image coding systems cannot be efficiently tackled with SIMD computing. The main difficulty is that the innermost algorithms of current coding systems process the samples in a sequential fashion. This paper presents a bitplane coding strategy tailored to the kind of parallelism required in SIMD computing. Its main insight is to employ vector instructions that process T coefficients of a codeblock in parallel and synchronously. To achieve this coefficient-level parallelism, some aspects of the bitplane coder are modified. First, the scanning order is devised to allow parallel coefficient processing without penalizing the formation of contexts. Second, the context formation approach is implemented via low-complexity techniques. Third, the probability estimates of the emitted symbols employs a stationary probability model that does not need adaptive mechanisms. And fourth, entropy coding is carried out by means of multiple arithmetic coders generating fixed-length codewords that are optimally sorted in the bitstream. The proposed bitplane coding strategy with parallel coefficient processing provides a very fine level of parallelism that permits its efficient implementation for both SIMD and MIMD computing. Experimental results indicate that the coding performance of the proposed method is highly competitive, similar to that achieved by the JPEG2000 standard.

Chapter 4

Implementation of BPC-PaCo in a GPU

This chapter 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$\times$ with respect to the fastest implementations of current compression standards. Also, a power consumption evaluation shows that our implementation consumes 40$\times$ less energy for equivalent performance than state-of-the-art methods.

We recall that the coding pipeline of JPEG2000 is structured in three main stages (Fig. 4.1): data transformation, data coding, and bitstream reorganization. The data transformation stage removes the spatial redundancy of the image through the discrete wavelet transform. Data coding codes the transformed samples, called coefficients, by means of exploiting visual redundancy. It does so using a bitplane coder and an arithmetic coder. 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

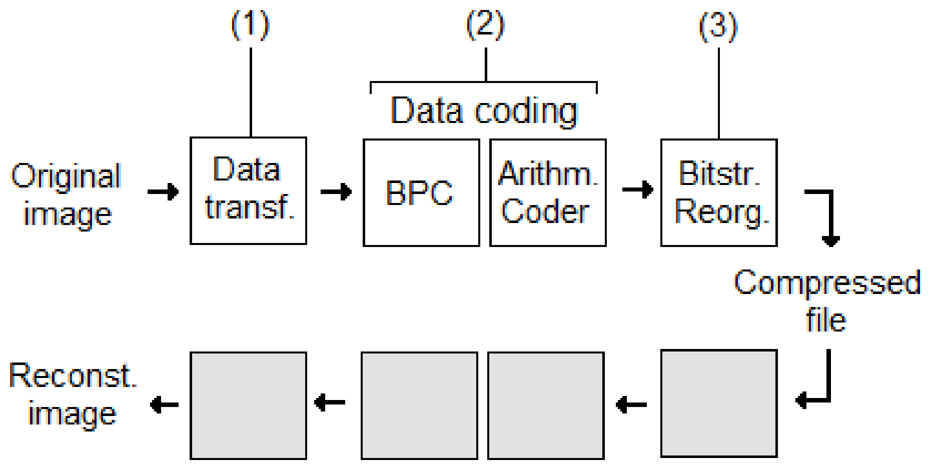


Figure 4.1: Main stages of the JPEG2000 coding pipeline: (1) data transformation, (2) data coding through bitplane coding (BPC) and arithmetic coding, and (3) bitstream reorganization. The decoding process (depicted in gray) carries out the inverse operations.

time, respectively [75].

Many SIMD implementations of image codecs on GPU architectures are devised to accelerate the coding process [1–13]. Their aim is to extract massive data-level parallelism in the first and second stage of the coding scheme to achieve higher computational performance than implementations optimized for CPUs. The operations carried out in the data transformation stage are well-fitted to SIMD computing. 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 [4, 6, 7, 9] are unable to fully extract the computational power of the GPU architectures. Table 4.1 shows a comparison presented in [29] reporting the execution time of JPEG2000 codecs optimized for CPUs and GPUs. Kakadu [76] is among the fastest CPU implementations of the standard, whereas CUJ2K [30] and JPEG2K [29] 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.

In the previous chapter, we introduced BPC-PaCo, a bitplane coding engine that unlocks the data dependencies of traditional algorithms. It can be used in the framework of

Table 4.1: 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 [29] 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

JPEG2000 without sacrificing any feature with respect to traditional bitplane coding. The bitstream generated by BPC-PaCo is not compliant with the standard of JPEG2000, 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%. The previous chapter focused on the image coding perspective of the method, analyzing its features and coding performance. In this chapter we present the optimized GPU implementation of BPC-PaCo. 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 occurs 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.

This chapter is structured as follows. Section 4.1 provides a general background of bitplane coding. Section 4.2 reviews BPC-PaCo and Section 4.3 describes the proposed implementation. Section 4.4 provides experimental results. The last section summarizes this work.

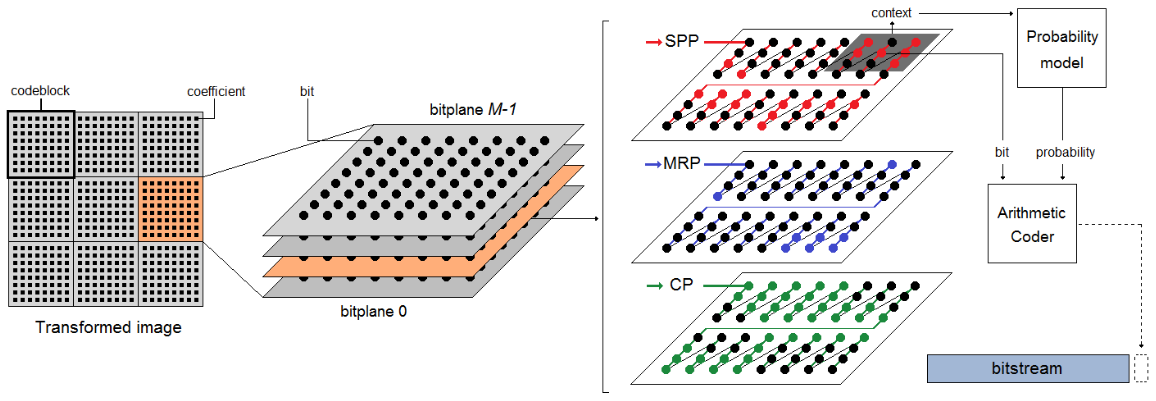


Figure 4.2: Overview of the JPEG2000 bitplane coding process. Codeblocks containing 8×8 coefficients are depicted for simplicity. The coefficients processed in the coding passes SPP, MRP, and CP are depicted in red, blue, and green, respectively.

4.1 Summary of the bitplane coding mechanisms

Fig. 4.2 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. 4.2. 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 [46]. 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 SPP. As seen in the right image of Fig. 4.2, 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. 4.2, 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 [68]. 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.

4.2 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. 4.3 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 [32, 60]. As shown in Fig. 4.3, 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 [46]. BPC-

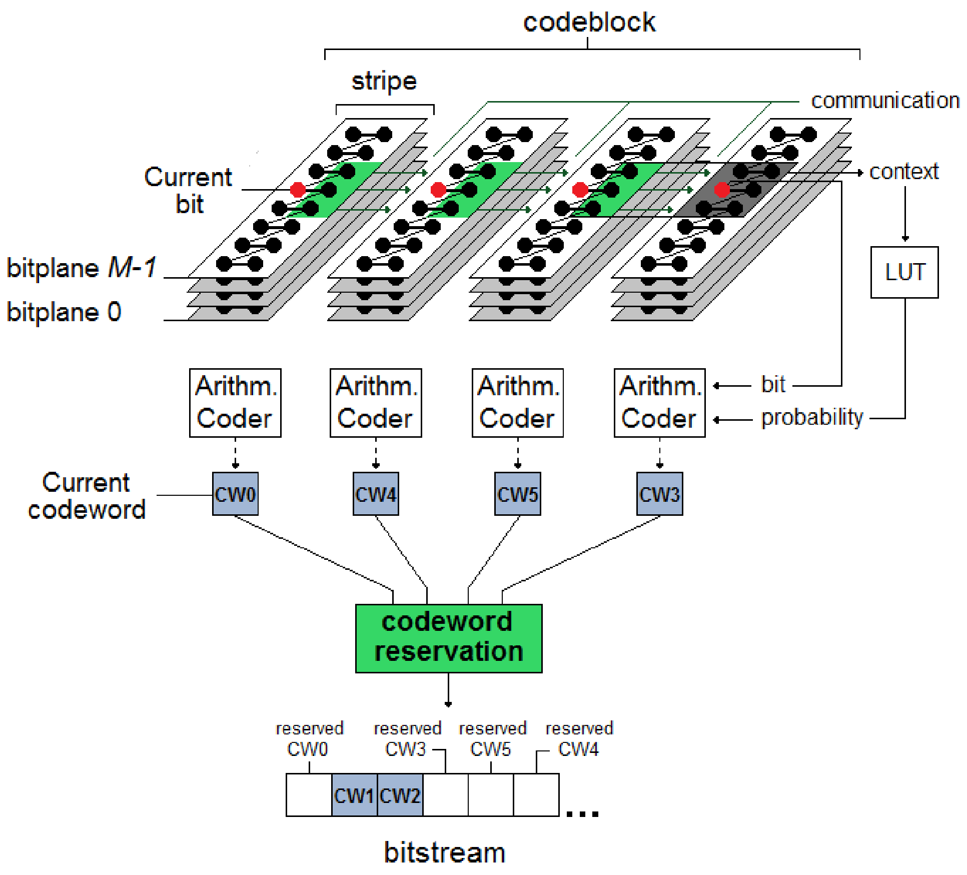


Figure 4.3: Illustration of the coding strategy of BPC-PaCo. The currently coded coefficients are depicted in red and the cooperation between stripes is depicted in green. The codewords generated by the arithmetic coders are depicted in blue.

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 [58]. 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 [58].

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 [68]. The fixed-length codeword arithmetic coder is adopted by BPC-PaCo because it reduces computational complexity and control flow-divergence. Fig. 4.3 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 the previous chapter, 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 [46], though the method can also use two passes without penalizing coding performance significantly. This can be seen in Fig. 4.4, which reports the coding performance achieved by BPC-PaCo when using two or 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

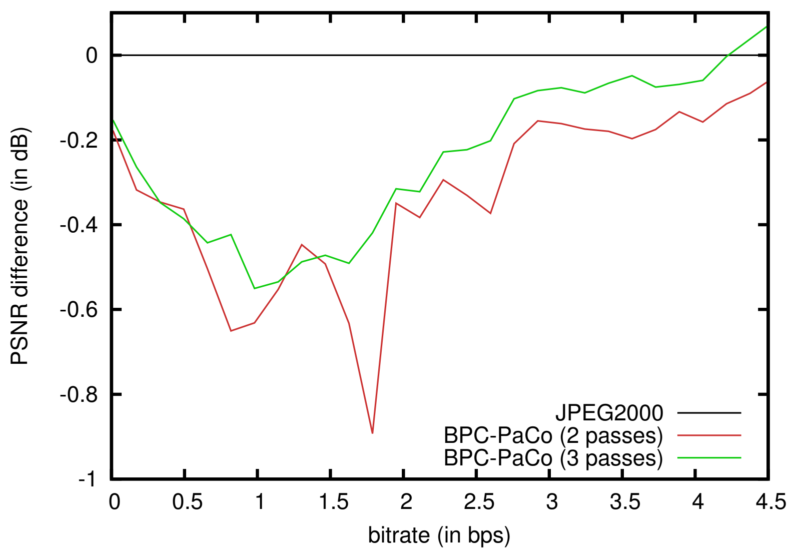


Figure 4.4: Coding performance comparison between JPEG2000 and BPC-PaCo with two and three coding passes.

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.

4.3 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.

4.3.1 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.

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 4.2 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

Table 4.2: 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

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 [13]. 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 7. We recall that *no* special synchronization instructions are needed due to the inherent synchronization of the threads within the warp.

4.3.2 Encoder with 2 passes

Algorithm 4 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 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 5) 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 4 is the loop that iterates from bitplane $M-1$ to 0. M is computed

Algorithm 4 - 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 5 - 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 6 - 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 7 - 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
```

beforehand by each warp via a reduction operation. The SPP is applied in lines 10-27, whereas the MRP, embodied in Algorithm 5, 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 6. 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 6 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 4.2. 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 7. 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 [68]. 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.

4.3.3 Encoder with 3 passes

Algorithm 8 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 4 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-

Algorithm 8 - 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
```

Table 4.3: 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 cod. passes	3 cod. passes
image size	$\frac{\text{inst. exec.}}{\text{\#coeff.}}$	warp efficiency	$\frac{\text{time}}{\text{\#coeff.}}$	$\frac{\text{inst. exec.}}{\text{\#coeff.}}$	warp efficiency	$\frac{\text{time}}{\text{\#coeff.}}$	$\frac{\text{time}}{\text{\#coeff.}}$	$\frac{\text{time}}{\text{\#coeff.}}$
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

pass version. The addition of a third coding pass also increases the control-flow divergence, which results in longer execution times. Table 4.3 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 $1.35\times$ 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.

4.3.4 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 4.3 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.

4.4 Experimental results

The proposed implementation is compared with Kakadu v7.8 [76] and JPEG2K v1.0 [29]. 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 [31, 34, 58, 60]. 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

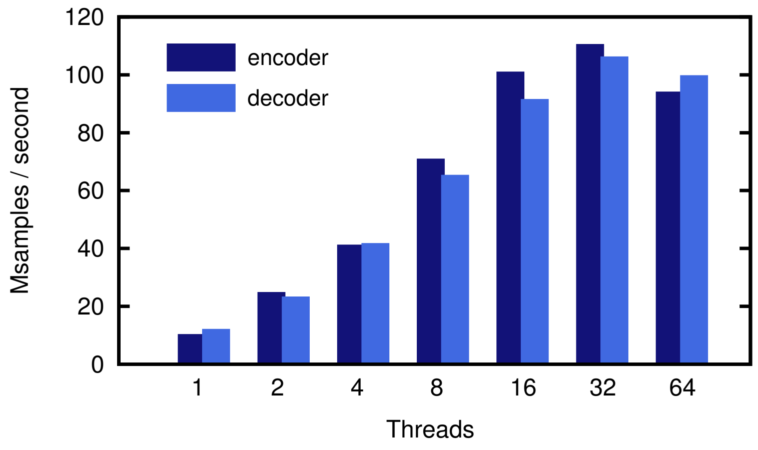


Figure 4.5: Evaluation of the performance achieved by Kakadu when using different number of execution threads. Each pair of bars corresponds to an image.

for Kakadu are obtained when using the optimal number of threads. See in Fig. 4.5 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.

In our implementation, CPU-GPU memory transfers are implemented synchronously using pinned memory. Table 4.4 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 is reported in the following tests, excluding pre- and post-processing operations.

Table 4.4: 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

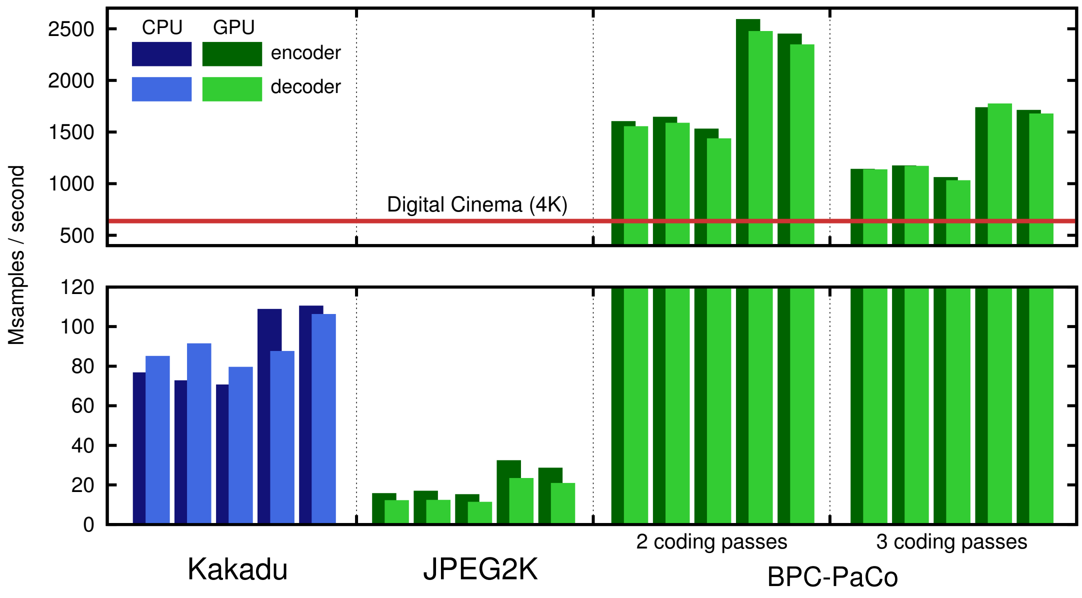


Figure 4.6: Computational performance evaluation. Kakadu is executed with 32 CPU threads and GPU implementations are executed in a GTX TITAN X. Each pair of bars corresponds to an image.

4.4.1 Computational performance

The first test evaluates computational performance. Fig. 4.6 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. 4.6 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 4.5 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

Table 4.5: 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×)

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 4.6 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 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

Table 4.6: 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

Table 4.7: Features of the GPUs employed. The devices are sorted by their peak throughput.

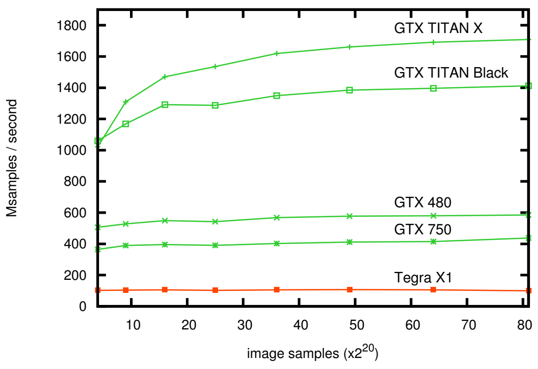
device	compute capability	#SM	cores × SM	total cores	clock frequency	peak throughput (normalized)	TDP
GTX TITAN X	Maxwell 5.2	24	128	3072	1000 MHz	3072 Gops/sec (12.00)	250 W
GTX TITAN Black	Kepler 3.5	15	192	2880	890 MHz	2563 Gops/sec (10.01)	250 W
GTX 480	Fermi 2.0	15	32	480	1400 MHz	672 Gops/sec (2.63)	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

the coding performance is penalized when using small codeblocks [31]. In general, 64×64 codeblocks provide a good tradeoff between computational and coding performance.

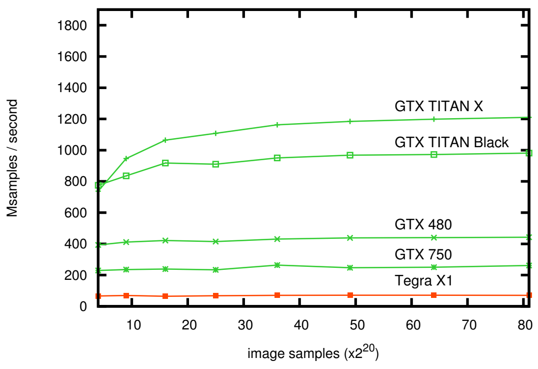
4.4.2 Scalability

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 4.7. 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. 4.7 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



(a)



(b)

Figure 4.7: Performance evaluation of BPC-PaCo with (a) 2 coding passes and (b) 3 coding passes when using different GPUs.

Table 4.8: 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%

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 4.3, 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. 4.7, does not correspond exactly with the peak throughput of each device, reported in Table 4.7. The differences are assessed in more detail in Table 4.8, 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 $1.5\times$ ($1.65\times$) higher than the performance achieved with the more powerful devices GTX TITAN X and GTX TITAN Black, and $2\times$ ($2.5\times$) 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 $50.4 \times 32 = 1612.8$ 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.

4.4.3 Power consumption

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

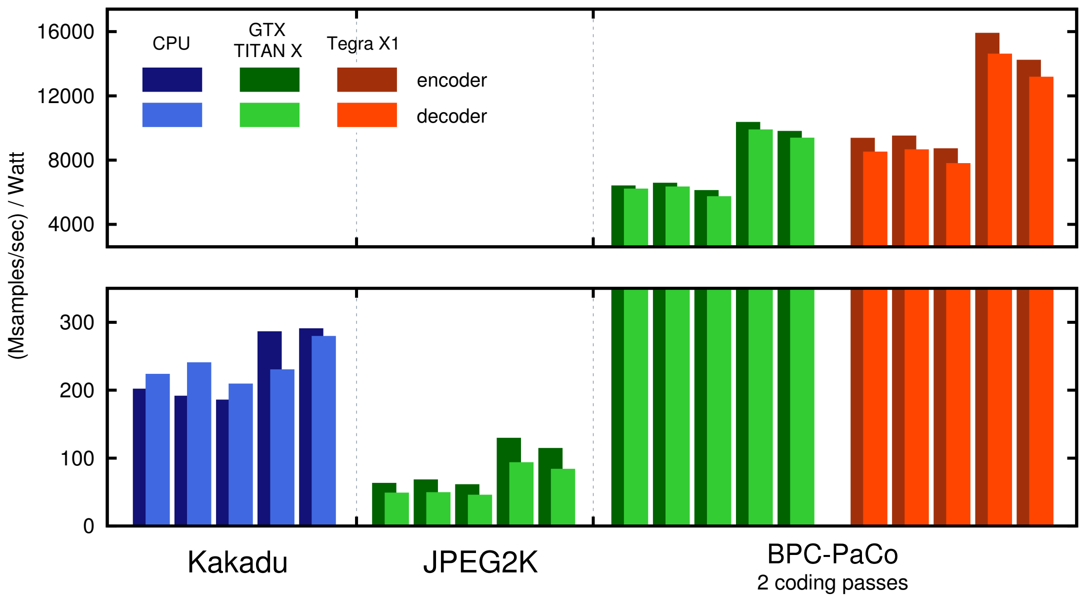


Figure 4.8: Power efficiency evaluation of Kakadu, JPEG2K, and BPC-PaCo with 2 coding passes when using a GTX TITAN X and a Tegra X1. Each pair of bars corresponds to an image.

is much higher than that of other devices. This is seen in the rightmost column of Table 4.7, 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. 4.8 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.

4.5 Summary

This chapter 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 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$\times$ higher than the best implementations of conventional image codecs in CPUs, while its power consumption is 40$\times$ 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 [77].

Chapter 5

Conclusions

This thesis presents a compendium of GPU-based high performance computing advances for the main operations of wavelet-based image coding. The problematics of state-of-the-art GPU computing are studied extensively from three different points of view. The first contribution is focused on the DWT, an operation highly SIMD-friendly with an extensive background of successful GPU implementations. A detailed analysis of the existing GPU solutions, the algorithm bottlenecks, and the architecture limitations is employed to design a new strategy that supposes an incremental improvement to achieve up to $4\times$ speedup with respect to previous solutions.

The second contribution of this thesis tackles a problem that is inherently not well-suited for GPU computing: the massive parallelization of the bitplane coder and arithmetic coder. It focuses on the traditional problem of reformulating a sequential algorithm to increase its parallelism, while maintaining the best possible efficiency and capabilities. A reformulation of some of the coding mechanisms is required to remove the key dependencies of the algorithm. The framework of JPEG2000 is employed as a reference, and a new scanning order, context formation, static probability model and a scheme of 32 cooperative arithmetic coders are proposed. The changes are embodied in BPC-PaCo, a new coding engine that promotes 32 times more SIMD parallelism than state-of-the-art codecs at the expense of 2% in coding performance loss but without renouncing to any advanced coding feature.

The last contribution of this thesis presents the research of developing the first GPU solution of a new state-of-the-art coding engine. A detailed analysis of BPC-PaCo is performed to identify the more suitable mechanisms to implement the method in a GPU. An efficient thread-to-data mapping scheme, a finely-tuned memory management, and advanced cooperation mechanisms are discussed, implemented, and analyzed for different GPU architectures. The resulting GPU-accelerated implementation of BPC-PaCo is compared with the most competitive CPU and GPU implementations of traditional wavelet-based bitplane codecs. The experimental results indicate that BPC-PaCo in a GPU is up to 30 times faster than the best previous implementations, both in CPU and in GPU, while being up to 40 times more power efficient.

Overall, the three contributions of this thesis represent the main pieces required to assemble an end-to-end high-performance wavelet-based image coding framework. All the source code of this thesis is left freely available in [41] and [77]. It is worth noting that the GPU performance discussions, analysis, and techniques presented in this thesis can also be employed for algorithms of other areas. Applications that present the stencil computation pattern in a GPU can benefit from the analysis presented in Chapter 2, whereas Chapter 3 and 4 present insightful guidelines for CPU-GPU algorithm re-formulating and GPU analysis, implementation, and tuning.

5.1 Future Work

The main research line that arises from this thesis is the implementation of an end-to-end wavelet-based GPU-accelerated codec. The main challenge is to assemble the implementations proposed in Chapter 2 and 4, and develop the wrapping coding stages, while maintaining a high performance profile. This task requires additional macroscopic performance analysis that is not explored in this thesis. CPU-GPU work partitioning and overlapping, host-device memory transfers optimization, and multi-GPU support have to be carefully studied to maximize end-to-end performance.

Real-time video coding represents the second step in the future research of this thesis. The proposed DWT and BPC-PaCo implementations can be used as core operations to develop an intra-frame GPU-accelerated video codec solution. This research again requires an in-depth exploration of high-level CPU-GPU optimizations to guarantee an efficient overlapping between the computation of subsequent video frames and CPU-GPU memory transfers.

Besides the main future work described above, an additional research line is to incorporate the implementations proposed in Chapter 2 and 4 into existing real-time image coding applications. This task would bring insights on the end-to-end practical performance of the solutions proposed. The effective performance gain of each GPU implementation would be assessed for different application scenarios, arising hints about application-specific optimizations of both the DWT and BPC-PaCo. Many challenges derived from the generic integration of GPU-accelerated solutions are also worth exploring in industry-wise environments, and they would be analyzed in this research line.

Appendix A

List of All Publications

All publications produced for this thesis are provided below in chronological order:

[13] 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

[32] 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*, pp. 163–172, Apr. 2015.

[34] 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*, pp. 345–354, Dec. 2015.

[31] 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.

[33] 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.*, in Press, 2017.

Appendix B

Acronyms

FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
GPGPU	General Purpose GPU
CUDA	Compute Unified Device Architecture
SIMD	Single Instruction Multiple Data
BPC	Bitplane Coder
SM	Streaming Multiprocessor
BPC-PaCo	Bitplane Image Coding with Parallel Coefficient Processing
MIMD	Multiple Instruction Multiple Data
SPP	Significance Propagation Pass
MRP	Magnitude Refinement Pass
CP	Cleanup Pass

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Nov. 2008, pp. 31–42.

[15] F. N. Iandola, D. Sheffield, M. Anderson, P. M. Phothilimthana, and K. Keutzer, "Communication-minimizing 2D convolution in GPU registers," in *Proceedings of the IEEE International Conference on Image Processing*, Sep. 2013, pp. 2116–2120.

[16] A. Chacón, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "FM-index on GPU: a cooperative scheme to reduce memory footprint," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2014, in Press.

[17] ——, “Thread-cooperative, bit-parallel computation of Levenshtein distance on GPU,” in *Proceedings of the 28th ACM International Conference on Supercomputing*, Jun. 2014, pp. 103–112.

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

[19] *Information technology - JPEG 2000 image coding system - Part 1: Core coding system*, ISO/IEC Std. 15 444-1, Dec. 2000.

[20] A. Skodras, C. Christopoulos, and T. Ebrahimi, “The JPEG2000 still image compression standard,” *IEEE Signal Process. Mag.*, vol. 18, no. 5, pp. 36–58, Sep. 2001.

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

[22] *Image Data Compression*, Consultative Committee for Space Data Systems Std. CCSDS 122.0-B-1, Nov. 2005.

[23] A. Said and W. A. Pearlman, “A new, fast, and efficient image codec based on set partitioning in hierarchical trees,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 3, pp. 243–250, Jun. 1996.

[24] D. Taubman, “High performance scalable image compression with EBCOT,” *IEEE Trans. Image Process.*, vol. 9, no. 7, pp. 1158–1170, Jul. 2000.

[25] W. A. Pearlman, A. Islam, N. Nagaraj, and A. Said, “Efficient, low-complexity image coding with a set-partitioning embedded block coder,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 14, no. 11, pp. 1219–1235, Nov. 2004.

[26] M. Krotkiewski and M. Dabrowski, “Efficient 3D stencil computations using CUDA,” *ELSEVIER Parallel Computing*, vol. 39, pp. 533–548, Oct. 2013.

[27] J. Franco, G. Bernabe, J. Fernandez, and M. E. Acacio, “A parallel implementation of the 2D wavelet transform using CUDA,” in *Proc. IEEE International Conference on Parallel, Distributed and Network-based Processing*, Feb. 2009, pp. 111–118.

[28] Z. Wei, Z. Sun, Y. Xie, and S. Yu, “GPU Acceleration of integer wavelet transform for TIFF image,” in *Proceedings of the Third International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, Dec. 2010, pp. 138–143.

[29] (2016, Jun.) GPU JPEG2K. [Online]. Available: http://apps.man.poznan.pl/trac/jpeg2k/wiki

[30] (2016, Jun.) CUDA JPEG2000 (CUJ2K). [Online]. Available: http://cuj2k.sourceforge.net

[31] 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.

[32] 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.

[33] 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.*, 2017, in Press.

[34] ——, “Strategies of SIMD computing for image coding in GPU,” in *Proc. IEEE International Conference on High Performance Computing*, Dec. 2015, pp. 345–354.

[35] “CUDA, C Programming guide,” Tech. Rep., Jan. 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[36] V. Volkov, “Better performance at lower occupancy,” in *Proceedings of the GPU Technology Conference*, vol. 10, Sep. 2010.

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

[38] M. Hopf and T. Ertl, “Hardware accelerated wavelet transformations,” in *Proceedings of the EG/IEEE TCVG Symposium on Visualization*, May 2000, pp. 93–103.

[39] A. Garcia and H.-W. Shen, “GPU-based 3D wavelet reconstruction with tileboarding,” *The Visual Computer*, vol. 21, no. 8-10, pp. 755–763, Sep. 2005.

[40] B. Penna, T. Tillo, E. Magli, and G. Olmo, “Transform coding techniques for lossy hyperspectral data compression,” *IEEE Trans. Geosci. Remote Sens.*, vol. 45, no. 5, pp. 1408–1421, May 2007.

[41] P. Enfedaque. (2014, Nov.) Implementation of the DWT in a GPU through a register-based strategy. [Online]. Available: https://github.com/PabloEnfedaque/CUDA_DWT_RegisterBased

[42] W. Pennebaker and J. Mitchell, *JPEG still image data compression standard.* New York: Van Nostrand Reinhold, 1993.

[43] *High Efficiency Video Coding Standard*, International Telecommunication Union Std. H.265, 2013.

[44] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, “Overview of the high efficiency video coding (HEVC) standard,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[45] N. Mehrseresht and D. Taubman, “A flexible structure for fully scalable motion-compensated 3-D DWT with emphasis on the impact of spatial scalability,” *IEEE Trans. Image Process.*, vol. 15, no. 3, pp. 740–753, Mar. 2006.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

[53] V. Galiano, O. Lopez-Granado, M. Malumbres, L. A. Drummond, and H. Migallon, "GPU-based 3D lower tree wavelet video encoder," *EURASIP Journal on Advances in Signal Processing*, vol. 1, pp. 1–13, 2013.

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

[55] F. Auli-Llinas and J. Serra-Sagrista, "Low complexity JPEG2000 rate control through reverse subband scanning order and coding passes concatenation," *IEEE Signal Process. Lett.*, vol. 14, no. 4, pp. 251–254, Apr. 2007.

[56] F. Auli-Llinas, J. Bartrina-Rapesta, and J. Serra-Sagrista, "Self-conducted allocation strategy of quality layers for JPEG2000," *EURASIP Journal on Advances in Signal Processing*, vol. 2008, pp. 1–7, 2008, article ID 728794.

[57] *Digital compression and coding for continuous-tone still images*, ISO/IEC Std. 10 918-1, 1992.

[58] 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.

[59] A. J. R. Neves and A. J. Pinho, "Lossless compression of microarray images using image-dependent finite-context models," *IEEE Trans. Med. Imag.*, vol. 28, no. 2, pp. 194–201, Feb. 2009.

[60] 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.

[61] *Information technology - Lossy/lossless coding of bi-level images*, ISO/IEC Std. 14 492, 2001.

[62] R. W. Buccigrossi and E. P. Simoncelli, "Image compression via joint statistical characterization in the wavelet domain," *IEEE Trans. Image Process.*, vol. 8, no. 12, pp. 1688–1701, Dec. 1999.

[63] F. Auli-Llinas, M. W. Marcellin, J. Serra-Sagrista, and J. Bartrina-Rapesta, "Lossy-to-lossless 3D image coding through prior coefficient lookup tables," *ELSEVIER Information Sciences*, vol. 239, no. 1, pp. 266–282, Aug. 2013.

[64] D.-Y. Chan, J.-F. Yang, and S.-Y. Chen, "Efficient connected-index finite-length arithmetic codes," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 5, pp. 581–593, May 2001.

[65] M. D. Reavy and C. G. Boncelet, "An algorithm for compression of bilevel images," *IEEE Trans. Image Process.*, vol. 10, no. 5, pp. 669–676, May 2001.

[66] H. Chen, "Joint error detection and vf arithmetic coding," in *Proc. IEEE International Conference on Communications*, Jun. 2001, pp. 2763–2767.

[67] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression using variable-to-fixed coding based on arithmetic coding," in *Proc. IEEE Data Compression Conference*, Mar. 2003, pp. 382–391.

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

[69] K. Sarawadekar and S. Banerjee, “An efficient pass-parallel architecture for embedded block coder in JPEG 2000,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 21, no. 6, pp. 825–836, Jun. 2011.

[70] P. Howard and J. S. Vitter, “Design and analysis of fast text compression based on quasi-arithmetic coding,” in *Proc. IEEE Data Compression Conference*, Mar. 1992, pp. 98–107.

[71] W. D. Wither, “The ELS-coder: a rapid entropy coder,” in *Proc. IEEE Data Compression Conference*, Mar. 1997, pp. 475–475.

[72] L. Bottou, P. G. Howard, and Y. Bengio, “The Z-Coder adaptive binary coder,” in *Proc. IEEE Data Compression Conference*, Mar. 1998, pp. 1–10.

[73] M. Slattery and J. Mitchell, “The Qx-coder,” *IBM Journal of Research and Development*, vol. 42, no. 6, pp. 767–784, Nov. 1998.

[74] F. Auli-Llinas. (2015, Jun.) BOI codec. [Online]. Available: http://www.deic.uab.cat/~francesc/software/boi

[75] ——, “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

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

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