Single-Loop Architecture for JPEG 2000

David Barina, Ondrej Klima, and Pavel Zemcik

Brno University of Technology
Bozetechova 1/2, 612 66 Brno, Czech Republic
{ibarina,iklima,zemcik}@fit.vutbr.cz

Abstract. We present a novel and very efficient software architecture designed for JPEG 2000 coders. The proposed method employs a strip-based data processing technique while performing a single-pass multi-scale wavelet transform. The overall compression chain is driven by incoming data while the fragments of the resulting bitstream are produced immediately after loading the corresponding data and additionally in parallel. The method is friendly to the CPU cache and nicely exploits the SIMD capabilities of the modern CPUs. Implanted into reference OpenJPEG implementation, our method has significantly better performance in terms of the execution time.

Keywords: Discrete wavelet transform, lifting scheme, JPEG 2000

1 Introduction

The discrete wavelet transform (DWT) is a signal-processing method suitable for decomposition of a signal into several scales. It is often used as a basis for sophisticated compression algorithms. Particularly, JPEG 2000 is an image coding system based on this wavelet compression technique. The format has wide application, especially with professional use cases. For example, Digital Cinema Initiatives established uniform specifications for digital cinemas in which JPEG 2000 is the only accepted compression format. Other applications include medical imaging, meteorology, image archiving (printed books, handwritten manuscripts), or aerial documentation.

Unfortunately, several major issues exist with the efficient implementation of the JPEG 2000 codec. This is especially true for images with high resolution (4K, 8K, aerial imagery) decomposed into a number of scales. For high resolution data decomposed into several scales using a typical separable transform, immensely many CPU cache misses occur. These cache misses significantly slow down the overall calculation. Furthermore, by following the typical data processing, the fundamental coding units of the JPEG 2000 format (referred to as code-blocks) are generated in the order that corresponds to scales. Consequently, it is not possible to produce a bitstream fragment which corresponds to a spatial image region earlier than the complete DWT decomposition is finished. Following the decomposition procedure as defined in the standard, the coefficients of a single resolution appears all at once. Therefore, the entropy coder (EBCOT) needs to

once again return to the data already touched. Finally, current implementations are built using 1-D transform which is unable to fully exploit the potential of modern CPUs.

This paper presents an efficient architecture for JPEG 2000 encoders. Our approach generates multi-scale wavelet transform coefficients in a purely single pass maner and even on the code-block basis. Our fundamental processing core nicely fits contemporary SIMD instruction sets (e.g., SSE).

The rest of the paper is organized as follows. The Related Work section summarizes the state of the art, especially efficient software realizations. The proposed approach is presented in Single-Loop Design section. Additionally, Performance section provides a performance evaluation. Finally, Conclusion section summarizes the paper.

2 Related Work

Many constructions of wavelets have been introduced in past three decades. As a key advance for image compression, Cohen–Daubechies–Feauveau [4] (CDF) biorthogonal wavelets provided several families of symmetric biorthogonal wavelet bases. As another important element, S. Mallat [7] demonstrated the orthogonal wavelet representation of images, today referred to as the 2-D DWT. It was originally computed with a pyramidal algorithm based on convolutions with quadrature mirror filters. In mid-1990s, W. Sweldens [9,5] presented the lifting scheme which speeded up such decomposition. He had shown how any discrete wavelet transform can be decomposed into a sequence of simple filtering steps (lifting steps). Finally, D. Taubman [10] proposed a new image compression algorithm – Embedded Block Coding with Optimized Truncation (EBCOT). The algorithm was quickly adopted into JPEG 2000 standard finalized in 2000.

Efficient realization of JPEG 2000 transform was outlined in [11]. The author described his implementation built with 16-bit fixed-point numbers. However, he did not provide much implementation details and he did not consider any friendliness to the CPU cache nor the SIMD set. Nevertheless, he expressed the memory requirements for multi-scale DWT as $(4 + 2S)M$ samples, where S is the number of lifting steps, and M is the width of the image. As the transform coefficients have to be arranged into code-blocks, the total memory requirements for JPEG 2000 codec are $(4+2S+3\times 2^{c_n})M$ samples, where 2^{c_n} is the code-block height. The initial 4 term corresponds to 2 lines per one decomposition scale. This imposes that his implementation generates all code-blocks at the same time, not one after another. Here we would like to make a short comment. According to the description in [11], their implementation does not process the data in a single loop. However, for a moment, let us assume that their implementation would do so. Still, this strategy is fundamentally different from the architecture proposed in this paper which generates individual blocks sequentially while reusing the same memory area for output coefficients all the time. Regarding the input processing, we have compared these two strategies (line-based and block-based) in [1]. They were almost equally fast. However, the line-based processing does not

fit the JPEG 2000 code-blocks, does not allow the parallel code-block processing, and does not allow to reuse the memory for HL, LH, and HH sub-bands. The motivation behind our work is to overcome these issues.

Many authors have tried to find an efficient schedule for 2-D DWT calculation. In [2], the authors proposed several cache-related optimizations of DWT phase. Although they still kept separated 1-D filtering, they interleaved the vertical pass of multiple columns. They also stored the LL sub-band contiguously in memory which is suitable for the next level of decomposition. Also, the authors [3] proposed several cache-related improvements and SIMD vectorization of DWT. At first, they used three specific memory layouts to improve cache locality under the multi-scale decomposition. Then, they used the same technique as the authors of [2] for interleaving the 1-D filtering on several adjacent columns. Finally, they vectorized the only the horizontal filtering using SSE instruction set. In [8], P. Meerwald *et al.* observed many cache misses especially when using large images with a width equal to a power of two. In order to overcome this problem, they have considered two improvements. Firstly, they added padding after each image row leading to a better utilization of limited set-associativity cache. Secondly, they filtered several adjacent columns concurrently as the authors of [3] and [2]. In [6], R. Kutil focused on the 2-D transform in which he merged vertical and horizontal passes into the single loop. Two nested loops (an outer vertical and an inner horizontal loop) are considered as the single loop processing all pixels of the image. His single-loop approach is line-based and vectorized using SSE set. However, they did not extend its approach to the whole multi-scale wavelet transform.

In [1], we have proposed a stand-alone unit able to transform the image in the single loop. Using this core, one can instantly produce the wavelet coefficients while the input data are visited only once. The processing of a particular scale can be suspended anytime and appropriate portions of the subsequent transform scale can be executed. In this paper, we extend this approach into a multi-scale single-loop approach on the code-block basis. This newly proposed approach is further parallelized and vectorized.

3 Single-Loop Design

In this section, we describe the single-loop core and its adaptation into the JPEG 2000 system. The established strip-based transform directly produces the code-blocks one by one. The processing of code-blocks is then chained together to progressively produce the multi-scale transform. On any level, such processing can be further parallelized in such a manner that the code-blocks are generated in parallel. As a consequence, this parallelism involves interleaving of the DWT and Tier-1 stages. Finally, the core is vectorized using the most widely used SIMD extensions.

The transform employed in JPEG 2000 decomposes the input image

$$\left(\mathrm{LL}^0_{m_0,n_0}\right)_{(0,0)\leq(m_0,n_0)<(M_0,N_0)} \tag{1}$$

of size $M_0 \times N_0$ pixels into $J > 0$ scales giving rise to the resulting wavelet bands

$$\left(\mathrm{HL}^j_{m_j,n_j}\right), \left(\mathrm{LH}^j_{m_j,n_j}\right), \left(\mathrm{HH}^j_{m_j,n_j}\right), \left(\mathrm{LL}^j_{m_j,n_j}\right), \quad \Big|_{(0,0)\le(m_j,n_j)<(M_j,N_j)} \tag{2}$$

at scales $0 < j < J$, and the residual LL band

$$\left(\mathrm{LL}^J_{m_J,n_J}\right)_{(0,0)\le(m_J,n_J)<(M_J,N_J)}, \tag{3}$$

at the topmost scale J. Such decomposition is performed using the 2×2 core with a lag $F = 3$ samples in both directions proposed in [1]. For each scale $0 \le j < J$, the core requires an access to two auxiliary buffers

$$\left({}^{M}B^j_{m_j}\right)_{0\le m_j<M_j}, \left({}^{N}B^j_{n_j}\right)_{0\le n_j<N_j}. \tag{4}$$

These buffers hold intermediate results of the underlying lifting scheme. The size of the buffer can be expressed as $M \times 4$ (x-buffer) and $N \times 4$ coefficients (y-buffer), where 4 is the number of values that have to be passed between adjacent 1-D cores. Taken together, the 2×2 core needs the access to 8 values in x-buffer and 8 values in y-buffer.

In detail, the core consumes a 2×2 fragment of the input signal and immediately produces a four-tuple of coefficients (LL, HL, LH, HH). The produced coefficients have a lag of 3 samples in the vertical as well as the horizontal direction with respect to the input coordinate system. In the JPEG 2000 coordinate system, the core consumes the fragment of the input starting on odd (m, n) coordinates. Every code-block starts on even (m, n) coordinates (the LL coefficient). Note that any shorter lag is not possible due to the nature of CDF 9/7 lifting scheme. To simplify relations, we also introduce two functions

$$\Theta(m, n) = (m + F, n + F), \text{ and } \Omega(m, n) = (\lceil m/2 \rceil, \lceil n/2 \rceil). \tag{5}$$

The function $\Theta(m, n)$ maps core output coordinates onto core input coordinates with a lag F. The function $\Omega(m, n)$ maps the coordinates at the scale j onto coordinates at the scale $j + 1$ with respect to the JPEG 2000 coordinate system. The 2×2 core transforms the fragment $I_{m,n}$ of an input signal onto the fragment $O_{m,n}$ of an input signal

$$I_{m,n} = \left(\mathrm{LL}^j_{\Theta(m,n)}\ \mathrm{LL}^j_{\Theta(m+1,n)}\ \mathrm{LL}^j_{\Theta(m,n+1)}\ \mathrm{LL}^j_{\Theta(m+1,n+1)} \right)^T, \tag{6}$$

$$O_{m,n} = \left(\mathrm{LL}^{j+1}_{\Omega(m,n)}\ \mathrm{HL}^{j+1}_{\Omega(m+1,n)}\ \mathrm{LH}^{j+1}_{\Omega(m,n+1)}\ \mathrm{HH}^{j+1}_{\Omega(m+1,n+1)} \right)^T, \tag{7}$$

while updating the two auxiliary buffers. Finally, operations performed inside the core can be described using a matrix C as

$$\boldsymbol{y} = C\boldsymbol{x} \tag{8}$$

with the input vector

$$\boldsymbol{x} = I_{m,n} \parallel {}^{M}\mathrm{B}_{m}^{j} \parallel {}^{M}\mathrm{B}_{m+1}^{j} \parallel {}^{N}\mathrm{B}_{n}^{j} \parallel {}^{N}\mathrm{B}_{n+1}^{j} \tag{9}$$

and the output vector

$$\boldsymbol{y} = O_{m,n} \parallel {}^{M}\mathrm{B}_{m}^{j} \parallel {}^{M}\mathrm{B}_{m+1}^{j} \parallel {}^{N}\mathrm{B}_{n}^{j} \parallel {}^{N}\mathrm{B}_{n+1}^{j}, \tag{10}$$

where $\parallel$ denotes the concatenation operator.

As a next step, we have encapsulated the processing of the code-blocks into monolithic units. These units are evaluated in horizontal "strips" due to the assumed line-oriented processing order. Inside the code-block unit, the 2×2 core can be used. Moreover, the unit requires access to two auxiliary buffers (one for each direction). The size of the buffer can be expressed as $2^{c_m} \times 4$ (for the x-buffer) and $2^{c_n} \times 4$ (for the y-buffer). As we are using the strip-based processing with a granularity of the code-block size, the y-buffer is straightly passed to the subsequent code-block processing unit. The x-buffer will be used by a strip of code-blocks lying below. At the beginning of the strip, the y-buffer contains arbitrary values. The first code-block unit initializes this buffer and passes it to the subsequent unit in x-direction. The transform of this subsequent unit is started not earlier than the EBCOT on the current unit has been finished. This allows for reusing the memory for HL, LH, and HH sub-bands.

The above-described procedure is in effect friendly to the cache hierarchy. The processing engine uses several memory regions of a different purpose. (1) The resulting code-block sub-bands occupy a few KiB of memory likely settled in the top-level cache. (2) The y-buffer occupies several hundreds of bytes. (3) The fragments of x-buffers occupy the same size as the total size of y-buffer. However, these are used only for short time and then can be evicted from all levels of the cache hierarchy. (4) The input strip can be simply streamed into the same memory region which may be in part mirrored in the cache. (5) The temporary LL bands can be partially mirrored as well. For a smaller resolution, there is a good chance that the entire working set can fit into the cache hierarchy.

The entire process can be efficiently parallelized. We have in mind the coarse-grained parallelism using the threads. The key idea is to split the strip processing into several independent regions. Thus, a single thread is responsible for several adjacent code-blocks. Each thread holds its private copy of y-buffer and the memory region for the resulting sub-bands (HL, LH, HH). Therefore, several EBCOT coders can work in parallel. Moreover, the threads are completely synchronization-free (they do not need to exchange any data). At the beginning of the strip processing, each thread initializes its y-buffer using a short prolog phase. There is only simplified core (without the vertical pass and the output phase) run in this phase. Thanks to the omission of the vertical pass, the x-buffer is not touched here and no interaction between threads is required. After the prolog, the processing continues in the usual way. Disjoint fragments of the x-buffer are accessed by all threads. In our implementation,[1] we have

[1] available on demand

parallelized the wavelet decomposition as well as Tier-1 encoding. On parallel architectures, it is also possible to encode every single code-block of the strip in parallel. However, the parallelization of our implementation is constrained by the number of computing units. Note that more sophisticated implementations could parallelize almost entire compression chain.

4 Performance

In the previous section, we have described our design of wavelet transform engine with the compatibility to JPEG 2000 standard. In this section, we evaluate its performance and compare it to the competitive solutions.

Let us now focus on physical memory demands. The input image is consumed gradually using strips with height of 2×2^{c_m} lines. No more input data are required to be placed in physical memory at the same moment. For the output sub-bands, memory for only $4 \times 2^{c_m+c_n}$ coefficients is allocated (considering all four sub-bands). This memory is reused by all code-blocks in the transform (or a processing thread). Additionally, we need to allocate two auxiliary buffers of size $M_j \times 4$ and $N_j \times 4$ coefficients for each decomposition level j. Note that $M_{j+1} = \lceil M_j/2 \rceil_{c_m}$ and $N_{j+1} = \lceil N_j/2 \rceil_{c_n}$, where $\lceil . \rceil_c$ denotes ceiling to the next multiple of 2^c; initially $M_0 = M$ and $N_0 = N$. For each auxiliary LL band (excluding the input and the final one), the window of physical memory can be maintained and progressively mapped onto the right place in the virtual memory. The size of such window is roughly $3 \times 2^{c_n} \times M_{j+1}$. Note that we need 3 instead of 2 code-block strips due to the periodic symmetric extension on the image borders, additionally, a lag of $F = 3$ lines from the input to the output of the core. Roughly speaking, the code-blocks of the subsequent scales do not exactly fit each other. Taken together, our solution requires

$$(2S + 3 \times 2^{c_n})M \tag{11}$$

samples populated into the physical memory. Please note that these memory requirements are the same as outlined in [11].

Our solution was compared to C/C++ libraries listed on the official JPEG committee web pages. The OpenJPEG, FFmpeg, and JasPer libraries are distributed under the terms of open-source licences. Thus, these could be analyzed through their source code in detail. Note that OpenJPEG and JasPer are approved as reference JPEG 2000 implementations. The Kakadu implementation is a heavy optimized closed-source library. To ensure reproducible experiments, we list versions used – JasPer version 1.900.1, OpenJPEG 2.1.0, and FFmpeg 2.8. The OpenJPEG, JasPer (enforced the 32-bit type), and FFmpeg libraries implement the transform using 32-bit fixed-point format. Our implementation is based on 32-bit floating-point format.

The overview of the above described libraries is shown in Table 1. The naive approach refers to processing the entire image at once while keeping the horizontal and vertical passes as well as the transform scales separated. Furthermore,

library	algorithm
our solution	strip-based, scales interleaved
OpenJPEG	naive
JasPer	naive
FFmpeg	naive
Kakadu	line-based, scales interleaved

Table 1. Software overview in terms of the transform stage.

inside the horizontal and vertical passes, the lifting steps are processed sequentially. As a consequence, samples of the tile are visited many times while being over and over again evicted from the cache. Unlike the naive approach, the other two approaches use sophisticated technique where the processing of consecutive scales is interleaved. Moreover, in case of our strip-based processing, the horizontal and vertical passes were fused into the single loop. Regarding the strip-based processing, the input is consumed using strips, one by one. The subsequent scales are recursively processed as soon as enough data is available. For the line-based processing, no details were provided [11] about the processing of the horizontal and vertical lifting steps.

The measurements presented in this paper were obtained on Intel Core2 Quad Q9000 running at 2.0 GHz. The CPU has 32 KiB of level 1 data cache and 3 MiB of level 2 shared cache (two cores share one cache unit). The system is running on 64-bit Linux. All the algorithms below were implemented in the C language, possibly using the SIMD compiler intrinsics. In all cases, a 64-bit code compiled using GCC with `-march=native -O3` flag was used. The performance was measured using the `clock_gettime` call. We measured the average time required to produce a single transform coefficient for various range of image resolutions.

Considering our test implementation, we have vectorized our processing engine using widely spread SIMD extensions. Since we have built our implementation over the 32-bit floating point numbers, we used primarily the SSE (Streaming SIMD Extensions) instruction set. The processing inside the 2×2 core is separable into series of 1-D filtering steps. The first idea was to extend the core to fit the 4-way 128-bit SSE registers. This way, we obtained the 4×4 core inside which all of the filtering steps are performed using 4-way parallelism through the 128-bit SSE register. This case was also studied in [1]. Unfortunately, an issue appears when storing the resulting coefficients into separated memory areas. In detail, the 4×4 core produces four 2×2 fragments of the output sub-bands. This operation does not fit the SSE instruction set and consequently degrades the performance. For this reason, we decided to construct 8×8 "supercore" consisting of four adjacent 4×4 cores. The supercore does not suffer from the above-described issue and provides a slightly better performance. The 8×8 core naturally fits into 8-way 256-bit AVX registers. In this case, the storage of the resulting coefficients is performed in fragments of 4×4 coefficients which again do not fit the AVX registers. This second issue is not possible to solve because

4×4 is the smallest possible code-block size required by the standard. In other words, a theoretical 16×16 core would produce the 8×8 fragments of sub-bands which might not fit the 4×4 code-blocks. We have used the OpenMP interface to parallelize our code; however, many other implementations are possible.

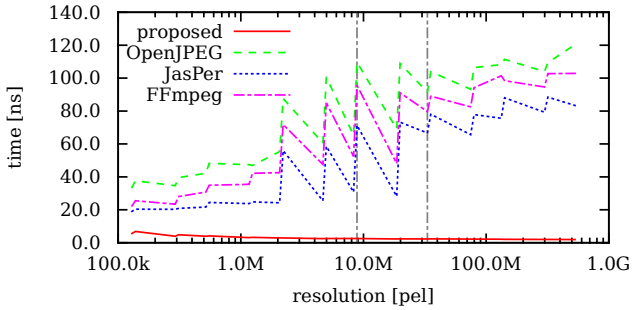

Fig. 1. Performance comparison of major libraries. Time per pixel for the transform stage only. DCI 4K and 8K UHD resolutions indicated by the vertical lines.

We have extracted the transform stage from the libraries described above in order to get accurate results. This stage was then subject of measurements. The results are plotted in Fig. 1. As observed also in [6,1], the single-loop processing has stable performance regardless the input resolution. The proposed implementation was measured using 4 threads and SSE extensions. However, the SSE or AVX extensions boost the performance by at most 5 %.

	single scale		multiple scales	
threads	time [ns/pel]	speedup	time [ns/pel]	speedup
1	3.08	1.00	5.60	1.00
2	1.59	1.94	3.44	1.62
3	1.22	2.53	2.68	2.09
4	0.97	3.16	2.56	2.19

Table 2. Parallel processing, streaming input. 4096×2160 input, 64×64 code-blocks. The tile was decomposed into a single ($J = 1$) and multiple ($J = 8$) scales.

We have evaluated the possibility of parallel processing. The original single-loop approach [1] scaled almost linearly with the number of threads. The JPEG 2000 processing has coarser granularity (code-blocks instead of cores) and it is performed in multiple scales. Higher scales of the decomposition have, unfortunately, significantly lower resolutions in comparison with the input tile. For this reason, the parallelization is not as efficient as in case of the original approach. The results of our measurement are shown in Table 2. It can be seen that the

single-scale decomposition scales slightly less than linearly with the number of threads. As it might be expected, the multi-scale decomposition is not as close to the linear relationship.

implementation	time [ns/pel]	original speedup	proposed speedup
original	528.73	1.00	—
proposed 1	398.36	1.33	1.00
proposed 2	210.77	2.51	1.89
proposed 3	175.27	3.02	2.27
proposed 4	142.09	3.72	2.80

Table 3. The proposed method inside of OpenJPEG library. 4K resolution.

Since we have implemented only DWT stage of the JPEG 2000 codec, we have decided to implant our code into OpenJPEG library replacing the original implementation. Note that no part of OpenJPEG is optimized for the performance. Because our implementation is built using the floating-point format and OpenJPEG uses the fixed-point format, we have to convert the samples one by one before and after the transform. The quantization and Tier-1 stage are performed using the original OpenJPEG's code. However, these parts of the compression chain now run in parallel as these are linked to the transform of code-blocks. The rest of the code remains unmodified and runs in sequence. Eight decomposition levels, up to 4 threads, and SSE were used. As expected, the single-loop processing has stable performance regardless of the input resolution. The measurement is summarized in Table 3. It can be seen that the complete compression chain scales better than the standalone transform stage.

5 Conclusion

We have introduced a new schedule for calculation of the discrete wavelet transform with JPEG 2000 compatibility. In contrast to previously presented schemes, the newly proposed scheme: generates the code-blocks one by one while reusing the memory for the resulting coefficients; passes every single code-block to subsequent Tier-1 coding before processing any next code-block (without evicting the code-block from the cache); generates and encodes the code-blocks in parallel (fragments of Tier-1 stage run simultaneously with fragments of the transform stage); exploits SIMD capabilities of modern CPUs as the wavelet coefficients are generated using 2-D processing unit instead of a conventional 1-D vectorization.

We have integrated our test implementation into OpenJPEG library (the reference JPEG 2000 software). The performance of this implementation outperforms the original code even if no parallelization and no SIMD extensions are used. When the parallel processing is enabled, the performance increases proportionally to the input size and number of processing threads.

In future work, we would like to implement a complete JPEG 2000 compression chain.

Acknowledgements This work was supported by the Technology Agency of the Czech Republic (TA CR) Competence Centres project V3C – Visual Computing Competence Center (no. TE01020415), the Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science (no. LQ1602), and Technology Agency of the Czech Republic (TA CR) project TraumaTech (no. TA04011606).

References

1. Barina, D., Zemcik, P.: Vectorization and parallelization of 2-D wavelet lifting. Journal of Real-Time Image Processing (in press)
2. Chatterjee, S., Brooks, C.D.: Cache-efficient wavelet lifting in JPEG 2000. In: IEEE International Conference on Multimedia and Expo. vol. 1, pp. 797–800 (2002)
3. Chaver, D., Tenllado, C., Pinuel, L., Prieto, M., Tirado, F.: Vectorization of the 2D wavelet lifting transform using SIMD extensions. In: International Parallel and Distributed Processing Symposium. p. 8 (2003)
4. Cohen, A., Daubechies, I., Feauveau, J.C.: Biorthogonal bases of compactly supported wavelets. Communications on Pure and Applied Mathematics 45(5), 485–560 (1992)
5. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. Journal of Fourier Analysis and Applications 4(3), 247–269 (1998)
6. Kutil, R.: A single-loop approach to SIMD parallelization of 2-D wavelet lifting. In: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). pp. 413–420 (2006)
7. Mallat, S.: A theory for multiresolution signal decomposition: the wavelet representation. IEEE Transactions on Pattern Analysis and Machine Intelligence 11(7), 674–693 (1989)
8. Meerwald, P., Norcen, R., Uhl, A.: Cache issues with JPEG2000 wavelet lifting. In: Visual Communications and Image Processing. SPIE, vol. 4671, pp. 626–634 (2002)
9. Sweldens, W.: The lifting scheme: A custom-design construction of biorthogonal wavelets. Applied and Computational Harmonic Analysis 3(2), 186–200 (1996)
10. Taubman, D.: High performance scalable image compression with EBCOT. IEEE Transactions on Image Processing 9(7), 1158–1170 (2000)
11. Taubman, D.: Software architectures for JPEG2000. In: Proceedings of the IEEE International Conference for Digital Signal Processing. pp. 197–200 (2002)