FACULTY OF INFORMATION TECHNOLOGY
department of computer graphics and multimedia

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ústav počítačové grafiky a multimédií

Lifting Scheme Cores for Wavelet Transform
Jádra schématu lifting pro vlnkovou transformaci

Ph.D. thesis / dizertační práce

Ing. David Bařina

supervised by / školitel
prof. Dr. Ing. Pavel Zemčík

Brno, 2015

Abstract

The thesis focuses on efficient computation of the two-dimensional discrete wavelet transform. The state-of-the-art methods are extended in several ways to perform the transform in a single loop, possibly in a multi-scale fashion, using a compact streaming core. This core can further be appropriately reorganized to target the minimization of certain platform resources. The approach presented here nicely fits into common SIMD extensions, exploits the cache hierarchy of modern general-purpose processors, and is suitable for parallel evaluation. Finally, the approach presented is incorporated into the JPEG 2000 compression chain, in which it has proven to be fundamentally faster than widely used implementations.

Abstrakt

Práce se zaměřuje na efektivní výpočet dvourozměrné diskrétní vlnkové transformace. Současné metody jsou v práci rozšířeny v několika směrech a to tak, aby spočetly tuto transformaci v jediném průchodu, a to případně víceúrovňově, použitím kompaktního jádra. Tohle jádro dále může být vhodně přeorganizováno za účelem minimalizace užití některých prostředků. Představený přístup krásně zapadá do běžně používaných rozšíření SIMD, využívá hierarchii cache pamětí moderních procesorů a je vhodný k paralelnímu výpočtu. Prezentovaný přístup je nakonec začleněn do kompresního řetězce formátu JPEG 2000, ve kterém se ukázal být zásadně rychlejší než široce používané implementace.

Keywords

discrete wavelet transform, lifting scheme, Cohen-Daubechies-Feauveau wavelet, SIMD, CPU cache, parallelization, JPEG 2000

Klíčová slova

diskrétní vlnková transformace, schéma lifting, vlnka Cohen-Daubechies-Feauveau, SIMD, cache CPU, paralelizace, JPEG 2000

Citation

Please cite this work as: D. Barina. *Lifting Scheme Cores for Wavelet Transform.* PhD thesis, Brno University of Technology, Brno, 2015.

Declaration

I declare that this dissertation thesis is my original work and that I have written it under the guidance of prof. Dr. Ing. Pavel Zemcik. All sources and literature that I have used during my work on the thesis are correctly cited with complete reference to the respective sources.

Prohlášení

Prohlašuji, že jsem tuto dizertační práci vypracoval samostatně pod vedením prof. Dr. Ing. Pavla Zemčíka. Rovněž prohlašuji, že jsem řádně uvedl a citoval všechny použité prameny, ze kterých jsem čerpal.

December 2, 2015

Acknowledgement

I would like to thank my supervisor prof. Dr. Ing. Pavel Zemcik. I also would like to thank my wife, my family, my friends, and my colleagues. This document was created using LaTeX and BibTeX.

Poděkování

Rád bych touto cestou poděkoval svému školiteli prof. Dr. Ing. Pavlovi Zemčíkovi. Rád bych rovněž poděkoval své ženě, rodině, přátelům a kolegům. Tento dokument byl vysázen systémem LaTeX a BibTeX.

Contents

List of Figures

List of Tables

List of Abbreviations

ASVP	Application-Specific Vector Processor
AVX	Advanced Vector Extensions
BCE	Basic Computing Element
BRAM	Block RAM
CDF	Cohen-Daubechies-Feauveau
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DWT	Discrete Wavelet Transform
EAW	Edge-Avoiding Wavelets
EBCOT	Embedded Block Coding with Optimal Truncation
FF	Flip-Flop
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
(GP)GPU	(General-Purpose computing on) Graphics Processing Units
GPP	General-Purpose Processor
LUT	Look-Up Table
MAC	Multiply–Accumulate operation
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions

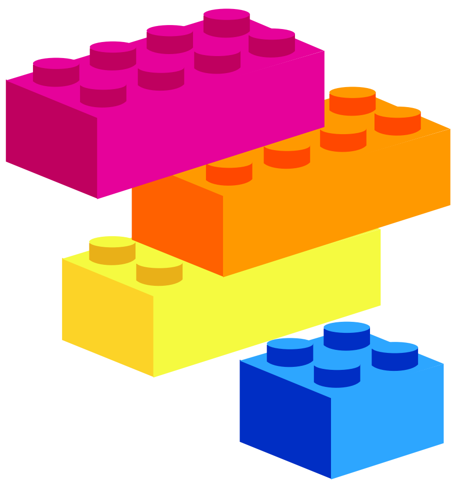
List of Symbols

ψ	wavelet
M, N	lengths of the signal in horizontal and vertical directions
m, n	free variables associtated to horizontal and vertical directions
G, H	FIR filters used in DWT
$\overline{H}$	reversed filter
H^*	transposed filter
a, d	wavelet transform subbands in 1-D
$\mathrm{a}, \mathrm{h}, \mathrm{v}, \mathrm{d}$	wavelet transform subbands in 2-D
I	pairs of lifting steps
$\alpha, \beta, \gamma, \delta$	filter coefficients in the lifting scheme
$K, 1/K, \zeta, 1/\zeta$	scaling factors in the lifting scheme
$P(z), \tilde{P}(z)$	polyphase matrices expressing the lifting scheme
$S_i(z), T_i(z)$	lifting steps (predict and update operators)
$\downarrow 2, \uparrow 2$	subsampling, upsampling by a factor of two
J	number of scales
j	particular scale
F	lag (delay) of the core
B	auxiliary buffer of the core

C	matrix describing the core
$\boldsymbol{x}, \boldsymbol{y}$	vectors comprising the input and output of the core
κ	number of elements at one position of the auxiliary buffer
z	complex variable (z-transform)
$.^T, .^*$	transposition (of matrix/vector)
$\|$	concatenation (of vectors)

Preface

I have always been fascinated by a construction of complex shapes composed of basic building blocks. Indeed, Lego used to be my favorite toy when I was a child. These dyadic blocks can be seen as wavelets in disguise. They come in different sizes (one, two, and four). They are governed by multi-scale relations. This means that one can combine the small sizes into to larger ones, and then engage them together…



Chapter 1

Introduction

Information contained in many different physical phenomena (e.g. sounds, images) can be described using signals. Manipulation with these signals using computers is the subject of the signal processing field, which uses a variety of mathematical tools to analyse, process, and synthesize them. The wavelet transform is one of these tools, allowing for the time–frequency signal analysis. In other words, one can view the information associated with a particular time and frequency.

The thesis focuses on methods for computing the discrete wavelet transform. Specifically, it extends existing single-loop methods to enable dealing with a two-dimensional multi-scale decomposition and to efficiently utilize features of modern CPUs.

The discrete wavelet transform (DWT) is a signal-processing tool suitable to decompose an analysed signal into several scales. For each such scale, the resulting coefficients are formed in several subbands. In the one-dimensional case, the subbands correspond to low-pass (a) and high-pass (d) filtered subsampled variants of the original signal. Plenty of applications are built over the discrete wavelet transform. One of them, nevertheless, stands out quite markedly. The transform is often used as a basis for sophisticated compression algorithms. Particularly, JPEG 2000 is an image-coding system based on such a wavelet compression technique. Unfortunately, there exists several major issues with effective implementation of the discrete wavelet transform. This holds true in particular for images with high resolution (4K, 8K, aerial imagery) decomposed into a number of scales (e.g. 8 scales). These issues are discussed below.

In the case of the two-dimensional transform, one level of the transform can be realized using the separable decomposition scheme. In this scheme, the coefficients are evaluated by successive horizontal and vertical 1-D filtering, resulting in four disjoint groups (a, h, v, and d subbands). A naive algorithm of 2-D DWT computation directly follows the horizontal and vertical filtering loops. Unfortunately, this approach is encumbered with several accesses to intermediate results. State-of-the-art algorithms fuse the horizontal and vertical loops into a single one, which results in the single-loop approach.

One level of the just described 1-D transform can be computed utilizing a convolution with two complementary filters. However, on most architectures there exists a more efficient scheme to calculate the transforms coefficients. This scheme is called lifting and, in contrast to convolution, it benefits from sharing intermediate results.

As indicated above, for high-resolution data decomposed into several scales by a typical separable transform, many CPU cache misses occur. These cache misses significantly slow down the overall calculation. Moreover, in real implementations, a large image block often needs to be buffered, which makes the transform memory-demanding. The motivation behind this work is to overcome these issues.

The thesis contributes to the state of the art of discrete wavelet transform computation methods. The following paragraph particularly outlines the issues that are not solved satisfactorily when using the existing methods.

The state-of-the-art approaches treat signal boundaries in a complicated and inflexible way. When we take these approaches into consideration, we find that parallelization, SIMD vectorization, and the cache hierarchy exploitation are not handled well. This is especially true in conjunction with multi-scale decomposition. Furthermore, the transform fragments cannot be computed according to arbitrary application requirements. For example, a particular transform block at a particular scale cannot be obtained with minimal or no unnecessary calculations. Finally, these approaches do not address the problem of scheme reorganization aimed at minimizing some of the platform's resources at the expense of others.

The thesis focuses on the CDF (Cohen-Daubechies-Feauveau) 5/3 and 9/7 wavelets, which are often used for image compression (e.g. the JPEG 2000 or Dirac standards). However, the methods are general and they are not limited to any specific type of transform.

The rest of the thesis is organized as follows. Chapter Discrete Wavelet Transform discusses the basic principles of discrete wavelet transforms and lifting scheme, and outlines the implementation issues. Chapter Computation Schedules iterates the existing approaches of computing the two-dimensional lifting scheme on various contemporary platforms comprising GPPs, FPGAs, and GPUs. Chapter Lifting Vectorization serves as a bridge between the existing methods and my own work. The main contribution of this thesis is presented in Chapter Lifting Core. Chapter Multi-Dimensional Cores extends the core presented into multiple dimensions. The subsequent Chapter Evaluation provides performance evaluation and discussion of the approach presented above. Finally, Chapter Conclusions concludes the thesis.

Chapter 2

Discrete Wavelet Transform

The discrete wavelet transform can be understood as a method suitable for the decomposition of a signal into low-pass and high-pass frequency components through so-called wavelets. This chapter introduces wavelet theory in a level of detail necessary to understand the thesis.

Wavelets are functions generated from one basic function by dilations and translations. Many constructions of wavelets have been introduced in the literature in the past three decades; for example [1]. As a key advance, I. Daubechies [2] constructed orthonormal bases of compactly supported wavelets in 1988. Subsequently, in 1992, Cohen–Daubechies–Feauveau (CDF) [3] biorthogonal wavelets provided several families of symmetric (linear phase) biorthogonal wavelet bases. Earliery, in 1989, S. Mallat [4, 5] demonstrated the orthogonal wavelet representation of images. It was computed with a pyramidal algorithm based on convolutions with quadrature mirror filters (QMF). In the mid-1990s, W. Sweldens [6, 7, 8] presented the lifting scheme which sped up decomposition. He showed us how any discrete wavelet transform can be decomposed into a sequence of simple filtering steps (lifting steps).

For a description of the filters, the well known z-transform notation is employed. The transfer function of the one-dimensional FIR filter $h(k)$ is defined as

$$H(z) = \sum_k h(k)\, z^{-k}. \tag{2.1}$$

For a better insight, the discrete wavelet transform can be understood as the approximation of a continuous signal by superposition of the individual wavelets. Generally, the wavelets $\psi \in L^2(\mathbb{R})$ are continuous functions from the Hilbert space of finite energy functions localized in both time and frequency. However, if we limit ourselves to the discrete wavelet transform, the wavelets are further constrained by the following equations. For illustration, two wavelets frequently used for DWT are plotted in Figure 2.1. The

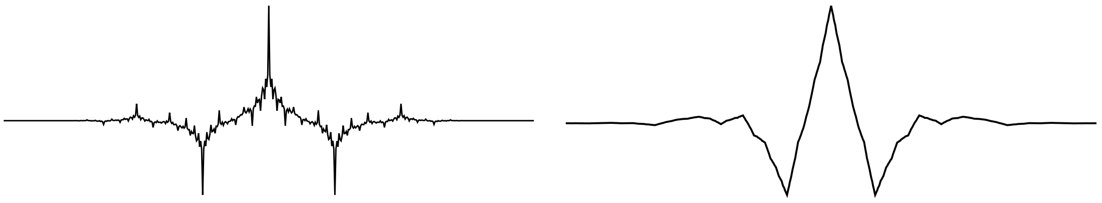


Figure 2.1: Shape of CDF 5/3 and CDF 9/7 wavelets. CDF 5/3 situated on the left, while CDF 9/7 on the right.

approximation is calculated through two conjugated quadrature filters often referred to as h, g. The relation between the wavelet and these filters is

$$\phi(t) = \sqrt{2} \sum_n h(n)\, \phi(2t - n), \tag{2.2}$$

$$\psi(t) = \sqrt{2} \sum_n g(n)\, \phi(2t - n), \tag{2.3}$$

where $\phi \in L^2(\mathbb{R})$ is a scaling function, which was formulated [4, 9] by S. Mallat. As a consequence of these equations, the multi-scale DWT can be computed by passing the signal through a filter bank comprising the $\tilde{h}$, $\tilde{g}$ filters followed by subsampling. One level of the decomposition linked with the synthesis is shown in Figure 2.2. The method is also referred to as the multiresolution analysis (MRA).

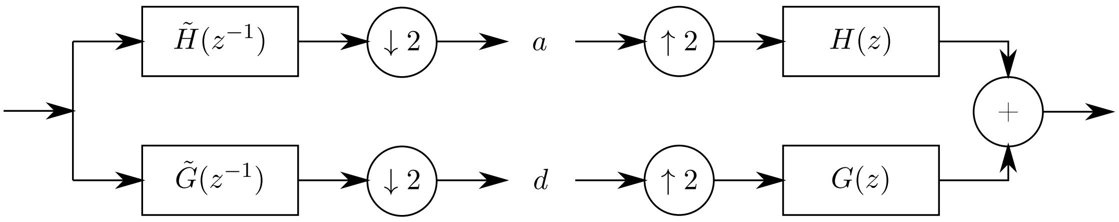


Figure 2.2: Analysis and synthesis part of DWT using FIR filters.

2.1 Lifting Scheme

DWT decomposes the signal into low-pass (a) and high-pass (d) frequency components using two analysis filters – $\tilde{h}$ (low-pass) and $\tilde{g}$ (high-pass) – followed by subsampling. The inverse transform first upsamples the a and d components and then uses two synthesis filters h (low-pass) and g (high-pass). The signal-processing view of such a decomposition and analysis is shown in Figure 2.2. Readers not very familiar with DWT are referred to the excellent book [10] by S. Mallat. For details about the lifting scheme, see [8, 7].

The polyphase representation [11, 8] is a convenient tool to express the $h, g, \tilde{h}, \tilde{g}$ filters as a sum of shorter filters formed by even (e) and odd (o) coefficients of the original ones. Having these filters, the assembled polyphase matrix

$$P(z) = \begin{bmatrix} H_e(z) & G_e(z) \\ H_o(z) & G_o(z) \end{bmatrix} \tag{2.4}$$

expresses the inverse transform. Such a polyphase matrix can be factorized (e.g. using the Euclidean algorithm [12]), so that

$$P(z) = \prod_{i=0}^{I-1} \left\{ \begin{bmatrix} 1 & S_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ T_i(z) & 1 \end{bmatrix} \right\} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix}, \tag{2.5}$$

where K is a non-zero constant, and polynomials $S_i(z), T_i(z)$ for $0 \le i \le I-1$ represent the individual lifting steps. Since the DWT has the perfect reconstruction property

$$P(z)\, \tilde{P}(z^{-1})^T = \mathbf{I}, \tag{2.6}$$

where $\mathbf{I}$ is the identity matrix and $\cdot^T$ denotes the transposition, the dual polyphase matrix

$$\tilde{P}(z) = \prod_{i=0}^{I-1} \left\{ \begin{bmatrix} 1 & 0 \\ -S_i(z^{-1}) & 1 \end{bmatrix} \begin{bmatrix} 1 & -T_i(z^{-1}) \\ 0 & 1 \end{bmatrix} \right\} \begin{bmatrix} 1/K & 0 \\ 0 & K \end{bmatrix} \tag{2.7}$$

describes the analytical part of DWT (forward transform). The $-T_i(z^{-1})$ is called the predict, whereas $-S_i(z^{-1})$ is called the update. The system is illustrated in Figure 2.3.

Let us take a closer look to the decomposition. At first, the input signal is split into two disjoint groups a, d, typically using even/odd sample indices. Then, the individual lifting steps are performed

$$\begin{bmatrix} d \\ a \end{bmatrix} = \tilde{P}(z^{-1})^T \begin{bmatrix} d \\ a \end{bmatrix} \tag{2.8}$$

resulting in a, d subbands.

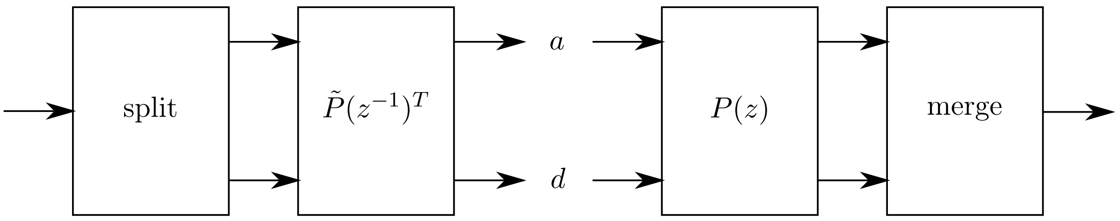


Figure 2.3: Analysis and synthesis part of DWT using lifting schemes.

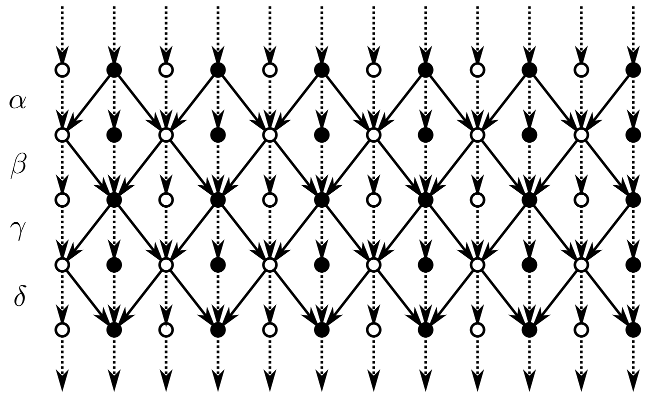


Figure 2.4: Data-flow diagram of CDF 9/7 lifting scheme. The blank bullets represent d coefficients, whereas the solid ones a coefficients. The solid arrows denote multiply operations. The dotted arrows just forward the value. The arrows are accumulated into the bullets.

Focusing on the CDF 9/7 wavelet as an example, the forward transform can be expressed [8] by the dual polyphase matrix

$$\begin{aligned}\tilde{P}(z) = &\begin{bmatrix} 1 & \alpha\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta\left(1+z\right) & 1 \end{bmatrix} \\ &\begin{bmatrix} 1 & \gamma\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \delta\left(1+z\right) & 1 \end{bmatrix} \begin{bmatrix} \zeta & 0 \\ 0 & 1/\zeta \end{bmatrix},\end{aligned} \tag{2.9}$$

where ζ is called the scaling constant. The $\alpha, \beta, \gamma, \delta, \zeta$ are real constants [8] specific to the CDF 9/7 transform. The forms $(1+z^{-1})$ and $(1+z)$ of the polynomials indicate symmetry of the lifting steps as well as the original filters. The corresponding data-flow diagram is shown in Figure 2.4 (scaling is omitted for simplicity). It should be noted that this particular wavelet is widely used in image processing, for example, in JPEG 2000 compression standard.

2.2 2-D Decomposition

S. Mallat [4] demonstrated the wavelet representation of two-dimensional signals computed with a pyramidal algorithm based on convolutions with quadrature mirror filters. One level of such a 2-D pyramid leads to a quadruple of wavelet coefficients (a, h, v, d) as

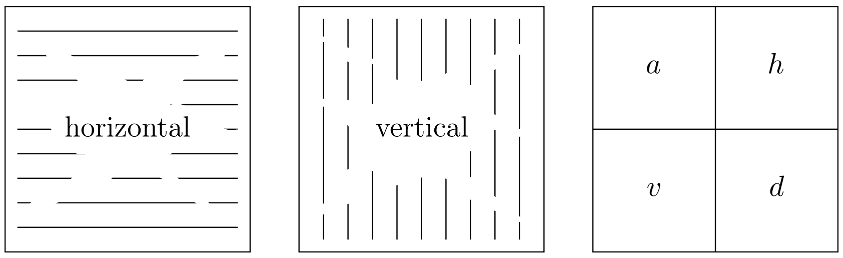


Figure 2.5: Illustrative 2-D decomposition using double sequence of 1-D transforms. From left: horizontal pass, vertical pass, the resulting subbands.

outlined in Figure 2.5.[1] The transform is defined as the tensor product of 1-D transforms. In this case, the two-dimensional transform consists of horizontal and vertical filtering steps. Considering the lifting scheme [8], the order of these steps has some constraints, but it is not strictly fixed (the horizontal and vertical steps can be interleaved). The decomposition is repeatedly applied on a subband which leads to the pyramidal decomposition. It should be noted that a naive algorithm implementing this 2-D scheme could perform a series of 1-D transforms horizontally, followed by a series of 1-D transforms vertically (or vice versa). The above mentioned 1-D transform could be implemented through the filter bank (convolution) or the lifting scheme.

The following discussion considers the situation in the context of a naive implementation. It does not matter whether the convolution or the lifting scheme is used. In both cases, the data coefficients are accessed at least twice (firstly for horizontal, secondly for vertical pass). Thus, the approach is inherently burdened with several accesses to intermediate results. More sophisticated algorithms [13] could perform these separable steps joined together which could even lead into a single-loop transform. In any case, the decomposition is further applied to a subband in order to get multi-scale representation. As in the previous case, individual scales of the decomposition can be performed in an interleaved manner. Performing the multi-scale decomposition in this way is described as the multi-scale single-loop approach.

This decomposition can be naturally applied to images. Images can be understood as finite two-dimensional arrays (matrices), where the values of individual elements represent image pixels. As these matrices are finite, a problem with an appropriate treatment of transform margins arises.

[1]The notation has the following meanings: a for image approximation, h, v for horizontally and vertically filtered features, d for diagonal features (residual signal)

2.3 Non-Separable Lifting Scheme

As stated above, the two-dimensional transform is defined through a series of one-dimensional steps. From the implementation point of view, this construction offers the possibility to merge and reorganize the underlying operations in order to minimize some of the consumed resources.

For instance, M. Iwahashi *et al.* [14, 15, 16] presented the non-separable lifting scheme employing two-dimensional (thus genuinely spatial) filtering steps. As a result, they reduced the number of lifting steps at the cost of increasing the total number of arithmetic operations.

To keep consistency with [16], the $H^*(z_m, z_n) = H(z_n, z_m)$ denotes a filter transposed to the $H(z_m, z_n)$. Furthermore, the $\overline{H}(z_m, z_n) = H(z_n^{-1}, z_m^{-1})$ denotes a filter reversed along the m- as well as n-axis. Coupled together, the $\overline{H}^*(z_m, z_n)$ denotes a transposed and reversed filter to the original $H(z_m, z_n)$.

For the CDF 5/3 and CDF 9/7 factorizations, their filtering steps take the following form

$$\begin{bmatrix} \mathrm{H_c} \\ \mathrm{H_c^*} \\ \mathrm{H_c H_c^*} \end{bmatrix} = \begin{bmatrix} c\,(1 + z_m) \\ c\,(1 + z_n) \\ c^2\,(1 + z_m + z_n + z_m z_n) \end{bmatrix}, \tag{2.10}$$

$$\begin{bmatrix} \overline{\mathrm{H}}_\mathrm{c} \\ \overline{\mathrm{H}}_\mathrm{c}^* \\ \overline{\mathrm{H}}_\mathrm{c}\overline{\mathrm{H}}_\mathrm{c}^* \end{bmatrix} = \begin{bmatrix} c\,(1 + z_m^{-1}) \\ c\,(1 + z_n^{-1}) \\ c^2\,(1 + z_m^{-1} + z_n^{-1} + z_m^{-1} z_n^{-1}) \end{bmatrix}. \tag{2.11}$$

In this scheme, it is no longer possible to distinguish between vertical and horizontal filtering. In their construction, the authors derived the non-separable 2-D scheme for CDF 5/3 and subsequently CDF 9/7 transforms. As an initial step of the CDF 5/3 transform, the input signal is split into quadruples (a, h, v, d).

$$\begin{aligned} \boldsymbol{x} &= \begin{bmatrix} a & h & v & d \end{bmatrix}^\mathrm{T} \\ \boldsymbol{y} &= \begin{bmatrix} a & h & v & d \end{bmatrix}^\mathrm{T} \end{aligned} \tag{2.12}$$

Then, spatial lifting steps leading to the calculation d coefficients are performed. This is followed by parallel computation of the h and v coefficients. In the third step, the a coefficient is updated. Formally, these steps are compressed [15, 16] into the matrix in (the notation is, strictly speaking, incorrect)

$$\boldsymbol{y} = C_{\alpha,\beta}\,\boldsymbol{x}, \tag{2.13}$$

where

$$C_{\alpha,\beta} = \begin{bmatrix} 1 & \overline{\mathrm{H}}_\beta & \overline{\mathrm{H}}^*_\beta & -\overline{\mathrm{H}}_\beta\overline{\mathrm{H}}^*_\beta \\ \mathrm{H}_\alpha & 1 & 0 & \overline{\mathrm{H}}^*_\beta \\ \mathrm{H}^*_\alpha & 0 & 1 & \overline{\mathrm{H}}_\beta \\ \mathrm{H}_\alpha\mathrm{H}^*_\alpha & \mathrm{H}^*_\alpha & \mathrm{H}_\alpha & 1 \end{bmatrix}. \tag{2.14}$$

The scheme for CDF 9/7 comprises two such connected transforms (α, β substituted for γ, δ).

2.4 Capabilities of Lifting Scheme

Many irreplaceable lifting scheme applications can be found in the literature. Several of them are discussed in this section. All these algorithms fit into the framework presented in this thesis. This is because the algorithms are of local nature, and can consequently be computed in a single loop with appropriate coefficients sharing. In other words, these can be incorporated into the presented computation scheme.

Unlike a convolution scheme, the lifting allows [17, 18, 19] the formulation of transforms mapping the integers to integers. For example, JPEG 2000 defines the reversible CDF 5/3 transform as

$$y(2n+1) = x(2n+1) - \left\lfloor \frac{x(2n) + x(2n+2)}{2} \right\rfloor, \tag{2.15}$$

$$y(2n) = x(2n) + \left\lfloor \frac{y(2n-1) + y(2n+1) + 2}{4} \right\rfloor. \tag{2.16}$$

After these two steps, the even samples $x(2n)$ will correspond to subband a, whereas the odd ones $x(2n+1)$ to subband d. Only additions, subtractions and shifts are needed to implement this particular transform. Many other integer-to-integer transforms can be found in [18, 19].

R. Fattal recently proposed [20] a new family of wavelets constructed using a robust data-prediction lifting scheme. This family, referred to as edge-avoiding wavelets (EAW), exhibits a better decorrelation of the data compared to the conventional lifting scheme. In their shape, the new wavelets encode the edginess of the analysed image at every scale. More specifically, EAW [20] use the edge-stopping function to define the control weights

$$w(\boldsymbol{p}, \boldsymbol{q}) = \Big(|x(\boldsymbol{p}) - x(\boldsymbol{q})|^a + \epsilon \Big)^{-1}, \tag{2.17}$$

where a is between 0.8 and 1.2, and $\epsilon = 10^{-5}$. These weights attenuate the coefficients of the subsequent lifting scheme. The effectiveness of this construction is demonstrated on

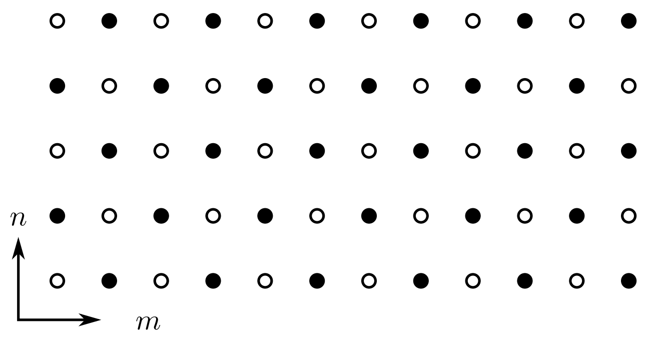


Figure 2.6: Quincunx lattices. Two lattices corresponding to r and b samples (analogously to a and d samples). The original input 2-D raster was split into these two groups.

various applications, including dynamic-range compression, or edge-preserving smoothing.

In [21], G. Uytterhoeven presented a construction of so-called Red-Black wavelets. These are constructed using a lifting scheme on 2-D grid where the samples are divided into two groups – red (r) and black (b) samples – forming so-called quincunx lattice (see Figure 2.6). The construction is based on two spatial lifting steps

$$\begin{aligned} b(m,n) &= b(m,n) \\ &\quad - \left(\frac{r(m-1,n) + r(m,n-1) + r(m,n+1) + r(m+1,n)}{4} \right), \end{aligned} \tag{2.18}$$

$$\begin{aligned} r(m,n) &= r(m,n) \\ &\quad + \left(\frac{b(m-1,n) + b(m,n-1) + b(m,n+1) + b(m+1,n)}{8} \right). \end{aligned} \tag{2.19}$$

This construction is rotated by 45 degrees on the next decomposition level. The wavelets show less anisotropy.[2] As a result, compared to a similar separable wavelets, image denoising performs better for lines that are not horizontal, vertical, or diagonal.

Denoising techniques based on the discrete wavelet transform (see Chapter 11 of [10]) use a thresholding operator which is applied separately to each wavelet coefficient. For example, the hard thresholding operator

$$\rho_\lambda^{hard}(d) = \begin{cases} d & |d| \geq \lambda \\ 0 & |d| < \lambda \end{cases} \tag{2.20}$$

[2]The wavelet coefficients respond to arbitrarily oriented edges.

with λ threshold is often implemented. Donoho *et al.* [22] derived $\lambda = \sigma\sqrt{2 \ln N}$ for Gaussian white noise of variance σ^2.

Except the last one, it would be very difficult to built the algorithms described in this section over the convolution scheme. Roughly speaking, the lifting scheme is nicely connectable with other algorithms.

Chapter 3

Computation Schedules

This chapter discusses existing methods of computing the 2-D discrete wavelet transform on various platforms, especially contemporary general-purpose processors (GPPs). With the exception of GPPs, implementations using programmable hardware and modern graphics cards are reviewed. The GPP and FPGA approaches are based on the same principles. Unlike them, the GPU approaches operate quite differently.

3.1 Processors

The thesis is mainly focused on contemporary processors (GPPs), especially on the x86 architecture. A type of the CPU cache is present in all modern platforms. An excellent introduction to this topic can be found in [23]. The cache is usually organized as a hierarchy of more cache levels. In the cache hierarchy, the individual coefficients of the transform are stored inside larger and integral blocks – cache lines, typically 64-bytes long. A hardware prefetcher attempts to speculatively load these lines in advance, before they are actually required. Moreover, due to a limited cache associativity, it is also impossible to hold in the cache lines corresponding to the arbitrary memory location at the same time. In detail, the cache lines are divided into several sets according to an associativity of the cache (e.g. four sets for typical 4-way associativity). The cache associativity indicates the number of lines from a particular set which can be held in the cache at one time. When more lines from this set are accessed, the older lines are evicted in favour of the new ones. Considering such cache, a memory address is split into three parts. Such an address structure is outlined in Figure 3.1. Typically, the low six[1] bits specify the offset in a cache line. A few upper bits specify the associativity set of the cache. The rest of the bits represent a tag stored for each individual cache

[1]for 64-bytes lines, $\log_2(64) = 6$

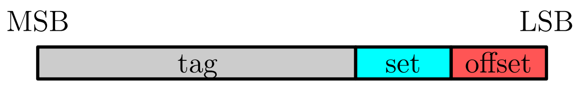


Figure 3.1: Address structure in relation to the CPU cache. The sizes of the individual parts depend on the particular architecture.

line. Another notable features typical for modern GPPs are SIMD extensions, symmetric multiprocessing, large virtual memory address space, paging, or a highly sophisticated branch prediction.

Originally, the problem of efficient implementation of the 1-D lifting scheme was addressed in [24] by Ch. Chrysafis and A. Ortega. Their approach is very general and is not focused on parallel processing. Nonetheless, this is essentially the same method as the on-line or pipelined method mentioned in other papers (although not necessarily using the lifting scheme nor the 1-D transform). The key idea is to make the lifting scheme causal, so that it may be evaluated as a running scheme without buffering of the whole signal.

Many authors have tried to find an efficient schedule for calculating the 2-D lifting scheme. Having the input 2-D image in the main memory, different strategies of 2-D transform implementation can be used. These strategies can be divided into three groups – row-column (fully separable), block-based, and line-based methods. The groups will be discussed with the individual techniques below. Aside from these basic strategies, several techniques were independently presented in several papers. All of them led to performance improvements. These techniques will be discussed now.

The separable implementation of the 2-D transform is performed by two passes of the 1-D transform – the horizontal and vertical pass. The horizontal pass densely visits the coefficients likely prefetched in the cache. Usually, there is no bottleneck in the horizontal pass. However, the vertical pass accesses the coefficients using a stride that prevents the hardware prefetcher from doing its job well. Moreover, usually only one coefficient from each cache line is accessed; the rest remains unused. Finally, considering the vertical access pattern, the coefficients lying in a particular column are likely mapped into the same cache set, especially considering the power-of-two [25, 13] data sizes.

In order to solve the last of the mentioned issues, several authors [25, 13, 26, 27, 28] suggested adding a padding after each data row (or the resulting subband row in some cases). This row extension causes the coefficients in a particular column to be mapped into different sets with a high probability. In particular, the odd or prime strides are often used.

Another technique to address the limited cache associativity is the loop fission used by A. Shahbahrami in [26, 27, 28]. This technique splits the vertical loop so it accesses at most as many rows as the cache associativity. As a consequence, several vertical passes are needed. Nevertheless, the pipelined [24] lifting scheme often does not need to access too many coefficients in a single processing step. So, this particular technique is broadly useful for a convolution-based transform only, where longer FIR filters are employed.

Since only one of the coefficients settled in each cache line is used in a vertical pass, many authors [25, 29, 30, 28] discovered a technique leading to a better utilization of cache lines. This technique is referred to as the aggregation, strip-mine, or loop tiling. Using such a technique, several adjacent columns are filtered concurrently, likely using all the coefficients in each cache line.

So far, the input as well as the output data were stored using a linear-memory layout (particularly, the row-major layout). Several authors investigated the influence of more complicated (mallat,[2] recursive), possibly non-linear memory layouts (4-D, Morton). The 4-D, Morton layouts are internally organized into blocks and thus imply the block-based strategy mentioned above. The working set for each block can now fit into the cache. The performance of these layouts was investigated in [31, 32, 33]. The mallat layout utilized in [30, 29] uses an auxiliary matrix in order to store the results of the horizontal filtering. As a result, no rearrangement stage is needed after the transform, since the coefficients can be directly stored at arbitrary locations in the original memory area. Using the recursive layout, each subband is laid out contiguously in the memory. This is especially useful for multi-scale decomposition, where the resulting subbands are transformed once more. This layout was employed in [30, 29].

[2]In the literature, the lower case initial letter is used.

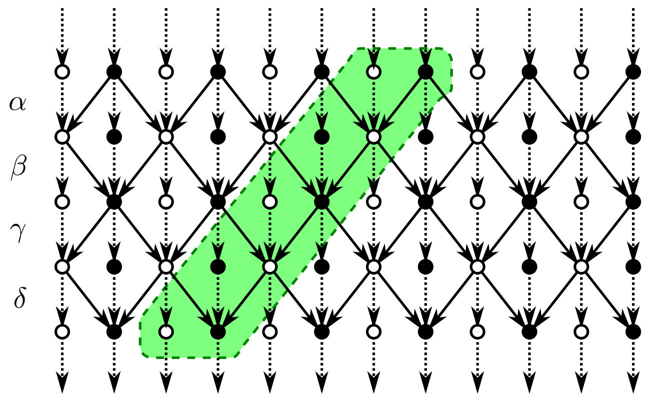


Figure 3.2: Single-loop vectorization of CDF 9/7 lifting scheme. The highlighted area is evaluated in a single iteration of the loop. Note that some intermediate results need to be forwarded into next iteration.

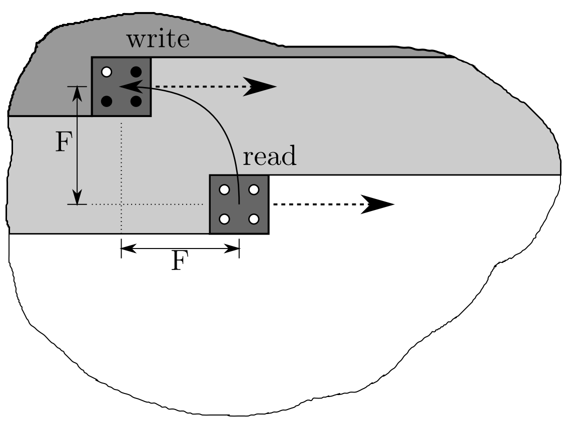


Figure 3.3: A core of the single-loop approach. Already read/written area is shown in light/dark gray.

Among all these disclosed techniques, probably the most important one is to interleave processing of the vertical and horizontal loop. This 2-D technique is often referred to as the pipelined, line-based, or single-loop transform. Some granularity (e.g. several input lines, large blocks) is used for interleaving of the loops. For instance, D. Chaver [34] used the block-based interleaving with a non-linear 4-D memory layout. Moreover, in [34, 35, 36, 37, 38], the line-based interleaving was used (at least two lines are needed). The most sophisticated techniques were investigated by R. Kutil in [13], which focused on CDF 9/7 wavelet and SIMD vectorization. In Kutil's work, one step of the lifting processing requires two values (a pair) to perform a loop iteration (see the data-flow graph in Figure 3.2). Thus, the algorithm needs to perform two horizontal filterings (on two consecutive rows) at once. For each row, a low-pass and a high-pass coefficient is produced, which makes 2×2 values in total. The image processing by this "core" is outlined in Figure 3.3. The lag $F = 4$ coefficients can be recognized from the data-flow graph in Figure 3.2. The algorithm passes four values from one iteration to the other in the horizontal direction for each row (eight in total). In the vertical direction, the algorithm needs to pass four rows between iterations. The length of the corresponding prolog as well as epilog phases is 4 coefficients. The situation is illustrated in Figure 3.4. This algorithm can be vectorized by handling the coefficients in blocks. Special prolog and epilog parts are needed (at least nine variants, if even/odd signal lengths are not considered).

Another important group of techniques covers the parallelization. Basically, two kinds of parallelization can be identified in the literature – the fine-grained and coarse-grained. The fine-grained parallelization refers to exploiting the SIMD extensions (namely, MMX,

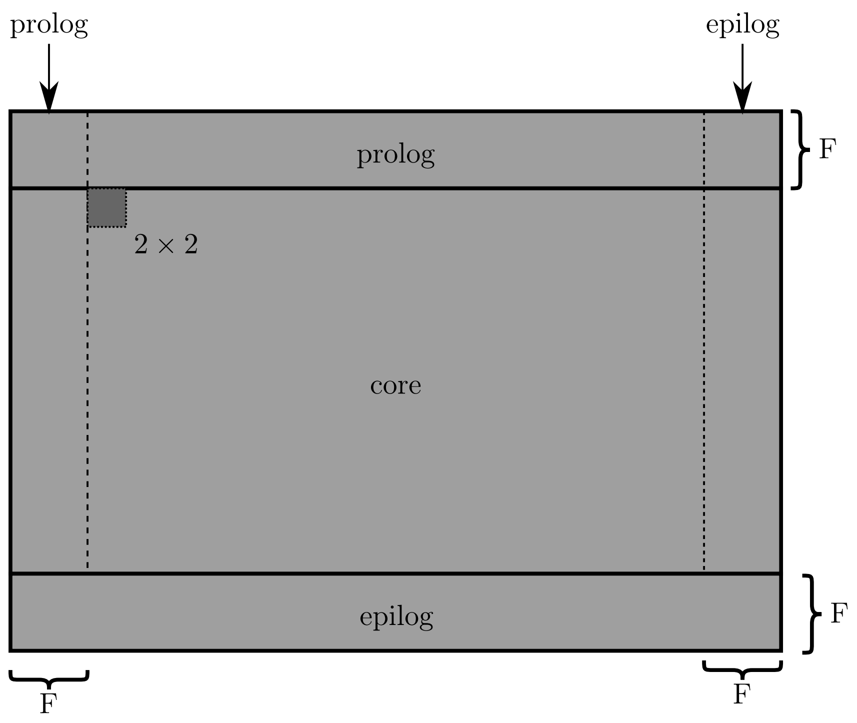


Figure 3.4: Simplified view of the single-loop approach showing the prolog and epilog phases. The length F of these phases is 4 (vertical vectorization) or 10 (diagonal one) coefficients.

and SSE). This kind was investigated at various levels in [32, 34, 29, 35, 27, 39, 36, 40, 41, 13, 42]. The most efficient solutions are presented in [13]. In this work, the author reads two 8×4 blocks and performs filterings and transpositions on them by exploiting the SIMD instructions. The main problem are the arduous prolog and epilog phases. The latter kind of parallelization was investigated, e.g., by D. Chaver in [33]. Moreover, various implementation details are occasionally considered in some papers; for example, loop unrolling and data alignments in [32]. Note that a slightly different viewpoint on strategies used to implement the transform is discussed in [43].

Furthermore, many papers exist which present an efficient 3-D DWT implementation. Let me to mention the most significant works. In [44], Bernabe *et al.* presented two methods reducing the 3-D transform execution time. However, in both of these methods, they employed the convolution scheme that does not take advantage of the benefits of the lifting scheme. In their first method, they split the original 3-D volume into several independent sub-volumes. Thus, they have performed several independent transforms (introducing a block effect) which is a different and easier task, compared to what is proposed in this thesis. On the other hand, such a method benefits from a small working set of the transformed data. The independent transforms can be further applied in an overlapped and non-overlapped manner. The second method is just a modification of the first, where the independent transforms are applied on cuboid sub-volumes instead of cubes. The method should better exploit the memory locality occurred due to their particular memory layout. The authors also exploited a fine-grained parallelism by vectorizing loops using the SSE instructions. Unfortunately, their methods are far away from the single-loop approaches. In [45], Bernabe *et al.* exploited the advantages of a parallel processing using multiple threads. The work is closely focused on hyper-threading (HT) technology. However, the principles of the methods employed remained the same as in previous paper. In [46], Lopez *et al.* introduced a fast frame-based 3-D DWT video encoder with low memory usage. The authors used the convolution scheme. In their approach, the video frames are continuously consumed by the 2-D DWT algorithm. Then, this transformed frame is stored in a buffer. Unfortunately, this buffer must be able to hold as many frames as the number of taps for the FIR filter in the temporal direction. Although their encoder reduces memory as well as computational requirements compared to the original 3D-SPIHT algorithm, it is still far away from the true 3-D pipelined transform. In another two papers [47, 48], the authors applied separately 2-D spatial and 1-D temporal transform. Both of the works deal with video compression. As in the previous case, their approaches still need several input frames to be accumulated in a buffer in order to filter the frames along the third dimension.

3.2 Field-Programmable Gate Arrays

The efficient implementation of DWT was also extensively studied on hardware platforms. This section focuses on Field-Programmable Gate Arrays (FPGAs).

Programmable logic devices (FPGAs) are one of the platforms suitable for implementations of the wavelet transformation. From the external memory bandwidth point of view, they cannot be compared to current GPGPU cards. Moreover, the advantage of FPGA implementation is mainly in small embedded devices, such as cameras which are already based on FPGA and/or have a fixed requirements on real-time processing, dimensions, low resource and power consumption and where the GPGPU or other platforms simply cannot be deployed.

Considering the 2-D signals, such as images, the transform can be computed using several strategies. These are typically referred to as the separable transform (row–column), the block-based transform, and the line-based transform. These strategies will now be briefly reviewed. Their detailed description can be found, e.g., in [49].

The simplest strategy is to perform the separable transform using sequential horizontal and vertical passes over the whole input image. This approach requires the use of large off-chip memory blocks to store the intermediate results. Unlike this strategy, the two following strategies do not require to store the intermediate results into off-chip memory. The block-based and line-based strategies perform the horizontal and vertical filtering onto smaller image fragments. These fragments consist of rectangular areas or small groups of lines in case of the block-based or the line-based strategy, respectively.

In all the previous strategies, the output coefficients are generated in chunks of various sizes. None of them generate the coefficients continuously with granularity corresponding to the essence of 2-D DWT. Note, please, that this elementary granularity of DWT is a quadruple of a, h, v, and d coefficients.

In this paragraph, several significant works on FPGA implementation of 2-D DWT are analysed. In [50], the authors implemented separable transform using the convolution rather than the lifting scheme. However, their implementation was able to deal with images of the size of 512×512 samples only, although, as the authors showed, bigger tiles are also possible for the price of much higher BRAM consumption. In [51], the authors proposed a line-based architecture with focus on JPEG 2000. Similarly to previous work, this architecture was able to process images of size 512×512. However, the transform is implemented using the lifting scheme. Another work focused on JPEG 2000 was done in [52]. As in the previous two cases, this implementation can deal with the tiles of size 512×512 pixels. Similarly, it is build upon the lifting scheme and processes images line by line. Yet another line-based 2-D wavelet transform implementation of JPEG 2000 was proposed in [53]. Again, it is based on the lifting scheme. This time, the implementation deals with 256×256 images. Many other papers can be found. However, none of them address the problem of efficient processing of high resolution, e.g. Full HD or 4K UHD, images.

3.3 Graphics Processing Units

The implementation of DWT was comprehensively studied on various platforms, including the modern programmable graphics cards (GPUs).

This section is focused on the implementation of DWT using modern graphics cards capable of a general-purpose computing. In these architectures, the GPU contains thousands of stream processors that are clustered into blocks. All processors in such a block execute the same instruction with different operands at one time. The blocks are grouped into multiprocessors which form the basic functional units of the GPUs. The thread scheduler allocates as many work-groups to multiprocessors as their resources allow. The work-groups are defined as an allocatable pack of threads that can interoperate with each other using the local memory and memory barriers. Thus, resources such as the local memory size should be minimized. The multiprocessor contains blocks of processors, warp schedulers, local memory, load store units, etc. The allocated work-groups created by the OpenCL framework is then divided into warps (hardware blocks with 32 threads). Execution instructions of these warps on blocks of processors are provided using warp schedulers dynamically. Due to the fact that each instruction is executed on the whole warp at once, recommendations for ensuring good performance of memory operations exist. Global memory indices in warp should be coalesced. For explanation, the thread in warp issues a coalesced memory access using 1–16-bytes width memory operation where neighboring threads access to neighboring addresses. Otherwise, additional memory operations are executed. Local memory is organized into banks. Access to the same banks from warp causes serialization. This issue is referred to as the bank conflict. The bank conflict is caused by a memory access from threads of the same warp to different positions of same bank in the shared memory. The serialization of local memory operations and uncoalesced global memory access can cause a degradation of performance.

This thesis is further focused on the OpenCL framework. OpenCL is a framework for general-purpose parallel programming across multiple device types (like GPUs, CPUs). In this framework, a platform independent executable program is called the kernel. The kernel is executed on the required number of threads that identify their data and control flow by their indices. These threads are organized into work-groups with an identical user-defined number of threads. The threads in such a group can cooperate with each other through local memory and barriers.

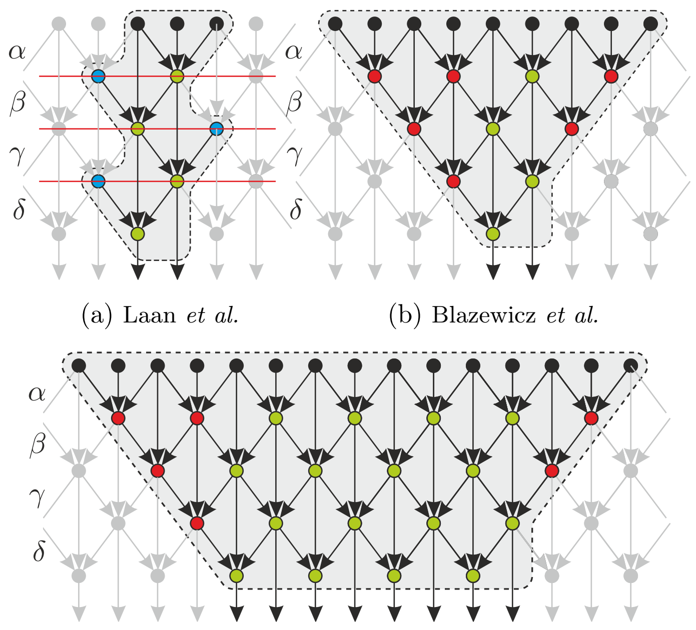
Considering implementations on modern GPUs, the input image has to be initially transferred from main memory into memory on the graphics card. Similarly, the resulting coefficients could be transferred back. Having the input 2-D image in the GPU global memory, different strategies of 2-D DWT implementation can be used. These strategies can be divided into three groups – row–column, block-based, and pipelined methods.

The row–column method applied on the entire 2-D image was used for instance in [54, 55, 56, 57, 58, 59]. In [56] and [57], a data transposition was performed in between the horizontal and vertical series of 1-D transforms. In [54] and [55], Tenllado *et al.* adapted the discrete wavelet transform on fragment shaders of GPU. They used the Cg programming language, and mapped the input image into textures. As this thesis is focused on the OpenCL framework, their paper is not discussed in more detail. The other cited papers are focused on the CUDA architecture. In [56], the convolution scheme is applied on each row. Then, the image matrix is transposed and the convolutions are applied on each column. Finally, the image is transposed back. In [58] and [59], V. Galiano *et al.* compared several CUDA implementations of DWT. They used the CDF 9/7 wavelet and convolution-based algorithm on entire rows/columns. Their fastest implementation uses the coalesced memory access.

In [57], Blazewicz *et al.* presented two wavelet transform approaches. The first of them calculates the wavelet transform through 4 kernels. The first kernel performs an image transposition using work-groups of size 16×16, where each single thread processes exactly one image element. To ensure coalesced global memory access, transposition is used in the shared memory, rather than directly in the global memory. In the second kernel, the vertical wavelet transform is performed as follows. The image is divided into multiple chunks. The size of such chunk is chosen as a work-group size $\times$ a number of pixels per one thread. Each thread in the work-group loads its elements from the global memory and stores them into the shared memory. Then, the adjacent elements that are required for the computation of the output coefficient are loaded from the shared memory into registers. The threads compute their output coefficients using 4 steps of the wavelet scheme independently to each other (with no synchronization). When the computation is finished, the output coefficients are written back to the global memory. To ensure the correct results, a symmetric border extension is used. The third and the fourth kernel calculates the image transposition and the horizontal wavelet transform in the same way as the first two kernels. The calculations that are done by a single thread in the previously described approach can be seen in Figure 3.5b (one pair of coefficients per thread) and Figure 3.5c (four pairs of coefficients per thread).

The pipelined approach was used in [60] and [61]. In [60], Laan *et al.* accelerated the Dirac video codec using the CUDA platform. In [61], the authors provided a detailed analysis of the DWT implementation using the lifting scheme on the CUDA platform. They focused on 2-D and 3-D methods of DWT implementation using several wavelets including CDF 9/7. In the horizontal part of their transform, each work-group is mapped to a single image row. Each thread computes one coefficient for each step and shares it with other threads. Because of non-atomic instructions being issued in the whole group, a memory barrier is needed in between each two steps. See Figure 3.5a. The vertical part of their transform maps each work-group to multiple vertical strips with a width that ensures coalesced global memory accesses and bank-conflict-free shared memory transfers.

On modern GPUs, the direct implementation of 3-D DWT was also studied. For instance, the authors of [62] and [63] used a convolution scheme which keeps transforms separated along three dimensions.



(a) Laan *et al.* (b) Blazewicz *et al.*

(c) Blazewicz *et al.*

🟢 necessary computations 🔴 redundant computations

– thread synchronization 🔵 exchanged

Figure 3.5: A portion of the data-flow graph attributable to a single thread. The method of (a) Laan *et al.* and two methods used by Blazewicz *et al.* – (b) with one pair, (c) with four pairs.

Approaches in [64] as well as [57] are focused on CDF wavelets and the lifting scheme. Their implementations splits the image into small tiles and performs several independent transforms on each of them. Thus, they have performed several independent transforms (introducing a block effect). Finally, let me note that the extension of the above-discussed methods will be presented in the subsequent chapters.

Chapter 4

Lifting Vectorization

This chapter should serve as a bridge between the above-described methods and my own work, presented in the following chapters. Although some parts of my work are presented here, it should not be considered the main contribution of the thesis.

Let me now focus on the data-flow diagrams of signal-processing algorithms. Vectorization is the process of evaluating these diagrams using operations that are applied to whole vectors instead of individual coefficients. Many GPPs have vector instruction sets which apply the same operation simultaneously to several coefficients in such a vector. Exploiting these instruction sets can be a particular reason of the vectorization. The vector instruction sets are also called SIMD sets or SIMD extensions. Intel's SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions) are frequent examples of such sets. Even if no vector instructions can be utilized, dividing the diagram into the vectors may be useful thanks to data locality in such vector.

One particular example of this technique can be found in [65] where the author considered the vectorization of FIR filtering (i.e. convolution). Basically, he identified three methods of the vectorization in the convolution

$$y(n) = \sum_k x(n-k)\, h(k) \tag{4.1}$$

data-flow diagram. In the equation above, two loops can be seen – the inner one for k and the outer for n variable. The causality of the scheme and dependencies of these loop iterations in principle allow three vectorization methods. In [65], these are denoted as A, B, and C. In method A, several consecutive inner loop iterations are combined into one vectorized iteration. In this iteration, a sample of the signal is associated with a sample of the filter coefficient. Unfortunately, dependencies between iterations break the possibility to utilize a parallel evaluation. In method C, a sample of the signal is associated with a vector of several distinct filter coefficients. This allows for parallel processing. However,

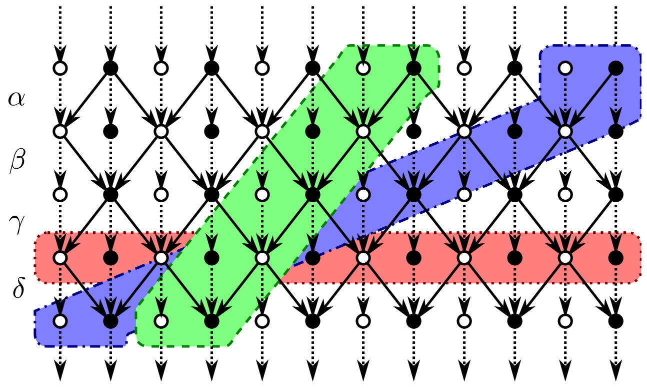


Figure 4.1: Vectorizations of CDF 9/7 lifting scheme. In all cases, the highlighted areas are evaluated in a single iteration of the loop.

several shuffle operations are required to implement this method. In method B, several input samples are associated with a vector of the same filter coefficient. Also this method allows for parallel processing.

Now, consider the data-flow diagram of the lifting scheme for CDF 9/7 wavelet. Three analogous methods can be identified here as well. For their understanding, please refer to the Figure 4.1. The terminology will be outlined with respect to this figure. The dashed green area corresponds to so-called vertical vectorization. This method was employed under different names in many papers, e.g. in [41]. The method can be seen as an analogy to the method A from the previous paragraph. The dash–dot blue area corresponds to the diagonal vectorization which was proposed in [IV]. It can be seen as an analogy to the method C. Finally, similarly to method B, the dotted red area depicts the horizontal vectorization which was investigated in [V]. The performance comparison of the presented vectorizations is postponed into Chapter 7.

4.1 Horizontal Vectorization

In [V], a general approach of lifting scheme vectorization evaluated on an FPGA-based Application-Specific Vector Processor (ASVP) was presented. This platform was presented in [66], [67], [68], and [69]. This unit can be classified as SIMD computer in Flynn's taxonomy. This platform uses several vector units referred to as Basic Computing Elements (BCEs). BCEs are able to accelerate simple operations (like addition or multiplication) on long single-precision floating-point vectors. Considering the dotted red area in 4.2, the lifting of CDF 9/7 transform can be directly adapted on them.

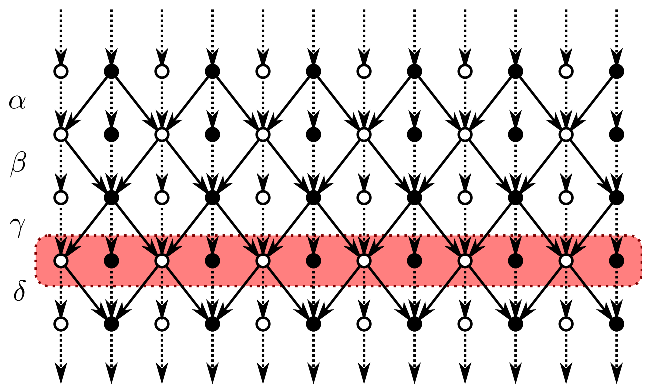


Figure 4.2: Horizontal vectorization of CDF 9/7 lifting scheme. The highlighted area is successively evaluated in the direction from the inputs to the outputs.

Consider the decomposition of the signal of length of N samples. Without loss of generality one can assume only signals with even length N. Possible remaining coefficient can treated separately in the prolog or epilog phases together with border extension. Thus, the transform contains $N/2$ pairs of resulting wavelet coefficients (a, d). The a coefficients represent the smoothed signal. On the contrary, the d coefficients form a difference or detail signal. When coefficient scaling is omitted, the calculation of a pair of the DWT coefficients at the position n (s_n and d_n) is performed through four lifting steps. Intermediate results ($a_n^{(i)}$ and $d_n^{(i)}$) can be appropriately shared between neighbouring pairs of coefficients (a_n and d_n).

Let me now review the lifting scheme of the CDF 9/7 wavelet [8] from Chapter 2.

$$\begin{aligned}\tilde{P}(z) = &\begin{bmatrix} 1 & \alpha\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta\left(1+z\right) & 1 \end{bmatrix} \\ &\begin{bmatrix} 1 & \gamma\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \delta\left(1+z\right) & 1 \end{bmatrix} \begin{bmatrix} \zeta & 0 \\ 0 & 1/\zeta \end{bmatrix}.\end{aligned} \tag{4.2}$$

The general diagram of these lifting steps is shown in Figure 4.3. This particular factorization leads to the following implementation. To simplify the description, the scaling is omitted.

$$d_n = d_n + \alpha\left(a_n + a_{n+1}\right) \tag{4.3a}$$

$$a_n = s_n + \beta\left(d_n + d_{n-1}\right) \tag{4.3b}$$

$$d_n = d_n + \gamma\left(a_n + a_{n+1}\right) \tag{4.3c}$$

$$a_n = s_n + \delta\left(d_n + d_{n-1}\right) \tag{4.3d}$$

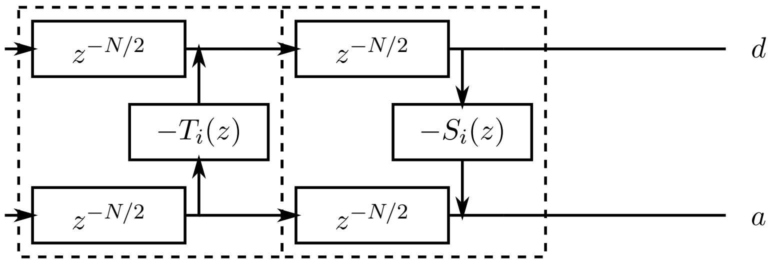


Figure 4.3: Block diagram of the horizontal vectorization. The parts bound with a dashed line correspond to the areas of parallel computation.

This algorithm requires several reads and writes of the intermediate results a_n and d_n. Considering the GPP implementation together with long signals, these intermediate results will be evicted several times from the CPU cache [23] in favor of other intermediate results. Consequently, many cache misses during such a computation will occur. Considering the vector processor implementation, the individual equations (4.3) correspond to several elementary vector operations (data move, addition, multiplication).

4.2 Vertical Vectorization

Another way of lifting data flow graph evaluation is the vertical vectorization. In the literature, this approach can be found under various names, e.g. the double-loop approach in [13]. Earlier, it was described in [24] with focus on low memory systems. This method was evaluated in [IV, V].

Any discrete wavelet transform with finite filters can be factored into a finite sequence (I pairs) of predict and update convolution operators T_i and S_i. As stated earlier, this decomposition can be described by the dual polyphase matrix

$$\tilde{P}(z) = \prod_{i=1}^{m} \left\{ \begin{bmatrix} 1 & 0 \\ -S_i(z^{-1}) & 1 \end{bmatrix} \begin{bmatrix} 1 & -T_i(z^{-1}) \\ 0 & 1 \end{bmatrix} \right\} \begin{bmatrix} 1/K & 0 \\ 0 & K \end{bmatrix}, \tag{4.4}$$

where each predict operator T_i corresponds to a filter $t_k^{(i)}$ and each update operator S_i to a filter $s_k^{(i)}$, i.e.

$$T_i(z) = \sum_{k=-l_n}^{g_n} t_k^{(i)} z^{-i}, \tag{4.5}$$

$$S_i(z) = \sum_{k=-m_n}^{f_n} s_k^{(i)} z^{-i}. \tag{4.6}$$

Let me note that this factorization is not unique. For symmetric filters, this non-uniqueness can be exploited to maintain symmetry of lifting steps.

The $T_i(z)$ and $S_i(z)$ filters need not be causal. In general, non-causal systems require storing of the whole input signal into memory (as can be seen from Figure 4.1). This is not suitable for fast or memory limited signal processing, nor for a vectorization. Therefore, it would be appropriate to convert non-causal lifting steps (T_i and S_i) to causal systems. The key to force these filtering steps to be causal is the introduction of appropriate delays. The transition from non-causal to causal system introduces [24] a delay z^{-l_i} on both inputs of the prediction filtering step S_i. In the bottom input a, the delay can be distributed into both branches. This leads to a causal system

$$\mathcal{S}_i(z) = z^{-l_i} S_i(z) = \sum_{k=0}^{g_i+l_i} s_{k-l_i}^{(i)} z^{-k}. \tag{4.7}$$

Similarly, a delay of m_i samples is introduced on both inputs of update step T_i. Again, this delay can be distributed into branches of upper input d. The resulting equation is given as

$$\mathcal{T}_i(z) = z^{-m_i} T_i(z) = \sum_{k=0}^{f_i+m_i} t_{k-m_i}^{(i)} z^{-k}. \tag{4.8}$$

For simplicity, the adjacent delays can be combined into a single one. Finally in

$$\eta_i = l_i, \tag{4.9a}$$
$$\mu_i = l_i + m_i, \tag{4.9b}$$
$$\nu_i = m_i, \tag{4.9c}$$

the delays of η_i, μ_i, and ν_i samples appear around each pair of filtering steps S_i and T_i. The resulting block diagram is shown in Figure 4.4.

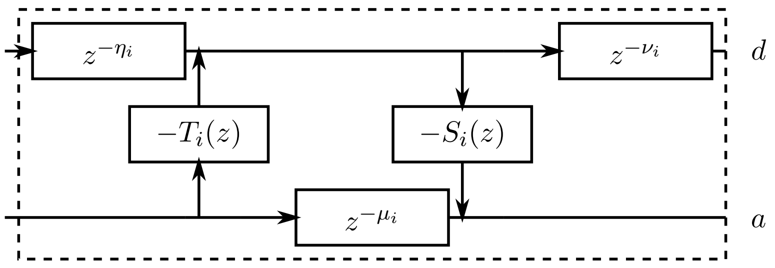


Figure 4.4: Block diagram of vertical vectorization. The area bound by the dashed line corresponds to the area of parallel computation.

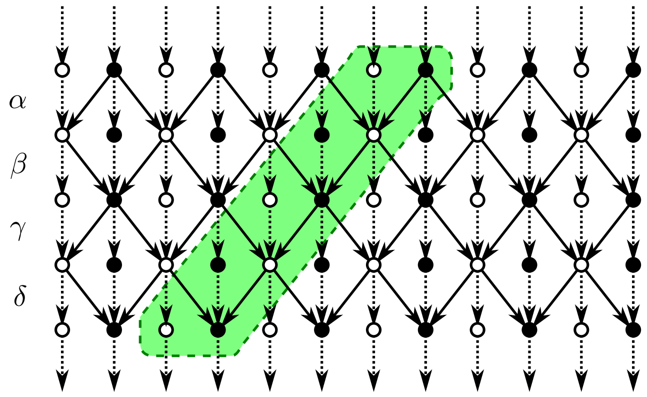


Figure 4.5: Vertical vectorization of CDF 9/7 lifting scheme. The highlighted area moves from the left to the right. Due to the internal data dependencies, the area cannot be computed in parallel.

In this method, the lifting computation is transformed into one loop, instead of multiple loops over all of the coefficients. Therefore, one pair of lifting coefficients a_n and d_n is computed in each iteration. However, the computations within each of these areas cannot be directly parallelized due to data dependencies. Even so, this procedure is advantageous because the coefficients are read and written only once. Consequently, this prevents unnecessary cache misses. In 1-D case, the SIMD vectorization of this method lies in processing of several adjacent areas in parallel, like in [40]. The data flow graph is split into vertical areas of width of two coefficients as in Figure 4.5.

4.3 Diagonal Vectorization

In [IV], the diagonal vectorization of the lifting scheme was proposed and subsequently evaluated. It can be appropriate for limited memory scenarios because it can start iteration of the vectorized loop immediately after a new pair of coefficients is available.

The subsequent lifting operations $\mathcal{S}_i$ and $\mathcal{T}_i$ inside the area of vectorization above cannot be computed in parallel due to data dependencies. To eliminate these dependencies another delay of one sample is introduced on both lines a and d, see Figure 4.6. Similarly to the case of vertical vectorization, multiple loops of the naive approach are transformed into the single loop over all of the coefficients. One pair of resulting coefficients a and d is produced in each iteration. Unlike the vertical vectorization, the elementary lifting operations evaluated in single loop iterations are shifted with respect to each other. This shift removes the data dependency within this loop iteration. Therefore, the elementary oper-

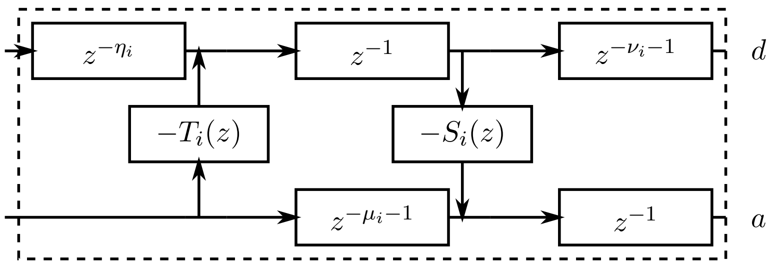


Figure 4.6: Block diagram of diagonal vectorization. The lifting operators can be evaluated in parallel.

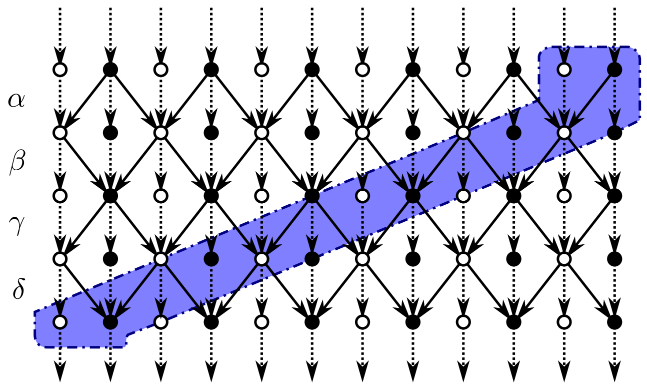


Figure 4.7: Diagonal vectorization of CDF 9/7 lifting scheme. The highlighted area moves from left to right. The operations inside this area can be computed in parallel.

ations can be now computed in parallel. The corresponding slice of the data flow graph is depicted in Figure 4.7. In contrast to the vertical vectorization, the proposed method does not require buffering of the input samples into groups of width corresponding to the used SIMD instruction set. A pair of resulting coefficients is available immediately after processing a pair of input samples. On the other hand, it is necessary to choose a wavelet with such a lifting factorization which has the same number of lifting steps (i.e. $2I$) as the number of components of the SIMD set (R). Depending on the instruction set being used, more shuffling instruction may be needed to implement the proposed diagonal vectorization (which is the case of Intel's SSE utilized in [IV]).

Table 4.1 shows the comparison of the different algorithms in terms of memory consumption. Each of the methods require several samples (2nd column) to start iteration of the vectorized loop and several memory cells (3rd column) to store intermediate results. The number in the 4th column indicates the number of operations that can be evaluated in parallel in each iteration. The horizontal vectorization requires an entire

vectorization	samples	coefficients	operations
horizontal	N	N	$N/2$
vertical	$2R$ (8)	$2I$ (4)	R (4)
diagonal	2 (2)	$6I$ (12)	$2I$ (4)

Table 4.1: Memory consumption of vectorization methods for two-tap lifting steps. The numbers in parentheses are related to SSE implementations of the CDF 9/7 transform.

signal of N samples ($N/2$ pairs of coefficients) to be loaded into memory. On this signal, up to $N/2$ independent operations can evaluated in parallel. In contrast, the vertical vectorization needs only $2R$ samples to start iterating over the vectorized loop in which R lifting operators can be evaluated in parallel. In the case of 2-tap T and S operators, this vectorization needs only $2I$ memory words to store intermediate results between such subsequent iterations. Finally, the diagonal vectorization requires only two new samples for each iteration which evaluates $2I$ lifting operators in parallel.

This chapter discussed three fundamental vectorizations of the lifting scheme. The simplest horizontal vectorization directly follows lifting scheme steps. However, several passes through the data are required in this case. Considering a limited cache size, this strategy would lead to extensive cache misses. The other two vectorizations (vertical and diagonal) allow for the computation the scheme in a single pass. Consequently, no unnecessary cache misses can be expected here.

Chapter 5

Lifting Core

The main contribution of the thesis is presented in this chapter. The contribution is a formulation of a computation unit built over the lifting scheme technique. The direct consequence of this formulation is the possibility of reorganizing operations in order to minimize the requirements for certain resources. Moreover, several other possibilities arise – for example, an elegant treatment of signal boundaries, or, in the case of multi-dimensional signals, a variety of allowed processing orders. The presented unit is further referred to as *the core*. In this chapter, the core is formally specified. Additionally, the subsequent chapter extends this core into multiple dimensions. To keep these two chapters compact, the evaluation and experiments are presented in Chapter 7.

In this paragraph, some terminology necessary for understanding the following text is clarified. Lag F describes a delay of the output samples with respect to the input samples. The stage is used in the sense of the scheme step, usually the lifting step. In linear algebra, such a stage can be described by the linear operator (a matrix) mapping the input vector onto the output vector. In this context, the operation denotes the multiply–accumulate (MAC) operation. Considering the output coefficient, the most demanding operation is identified as the operation having the highest number of operands. Please note that the notation is slightly different in some parts of this chapter – using a subscript for indexing the signals.

The following part of the chapter leads to the formulation of the core. Although the thesis has focused on image processing, the one-dimensional transform will be discussed first. The multi-scale discrete wavelet transform decomposes the input signal

$$\left(a^0_{n_0}\right)_{0 \leq n_0 < N_0} \tag{5.1}$$

of size $N_0 = N$ samples into $J > 0$ scales giving rise to the resulting wavelet bands

$$\left(d^j_{n_j}\right)_{0 \leq n_j < N_j}, \tag{5.2}$$

the temporary bands

$$\left(a^j_{n_j}\right)_{0 \leq n_j < N_j}, \tag{5.3}$$

at scales $0 < j < J$, and the residual signal

$$\left(a^J_{n_J}\right)_{0 \leq n_J < N_J}, \tag{5.4}$$

at the topmost scale J.

In order to solve the issues summarized at the beginning of this thesis, a unit which continuously consumes the input signal a^j and produces the output a^{j+1}, d^{j+1} subbands is proposed. This unit was also presented in [VI]. As mentioned above, this unit is referred to as the "core" in this thesis. As a consequence of the DWT nature, the core has to consume pairs of input samples. The input signal is processed progressively from the beginning to the end, therefore in a single loop. It should be noted that it is also possible to run these cores parallel – this possibility is discussed at the end of the chapter. The corresponding output samples are produced with lag F samples depending on the underlying computation scheme. For each scale $0 \leq j < J$, the core requires access to an auxiliary buffer B^j. These buffers hold intermediate results of the underlying computation scheme. At each scale, the size of the buffer can be expressed as κ coefficients, where κ is the number of values that have to be passed between adjacent cores.

Considering the single-loop approach, the vertical and diagonal vectorizations formulated in the previous chapter can be understood as baseline examples of the underlying computation scheme. However, the possibilities are much larger, as disclosed below in the thesis.

The characteristic attributes of the simplest cores are listed in Table 5.1. Although the core built above the vertical vectorization does not have the ability of SIMD evaluation, it is possible to link several such cores into longer "supercore" in order to fit the operations to SIMD instruction set. The last column indicated depth of calculations inside the core.

vectorization	lag F	buffer κ	SIMD-capability	latency
vertical	$2I$	$2I$	no	$2I$
diagonal	$6I-2$	$6I$	yes	1

Table 5.1: Attributes of the baseline single-loop cores. Valid for two-tap lifting steps.

This number corresponds to the number of data dependent blocks of data-flow graph inside the core.

To simplify relations, two functions will be introduced given by

$$\Theta(n) = n + F, \text{ and } \Omega(n) = \lceil n/2 \rceil. \tag{5.5}$$

The function $\Theta(n)$ maps core output coordinates onto core input coordinates with the lag F. The function $\Omega(n)$ maps the coordinates at the scale j onto coordinates at the scale $j+1$ with respect to the chosen coordinate system. Note that the $\Omega(n)$ can be defined in many ways.

The core transforms the fragment I_n^j of an input signal onto the fragment O_n^j of an input signal

$$\mathrm{I}_n^j = \begin{pmatrix} a_{\Theta(n)}^j & a_{\Theta(n+1)}^j \end{pmatrix}^T, \tag{5.6}$$

$$\mathrm{O}_n^j = \begin{pmatrix} a_{\Omega(n)}^{j+1} & d_{\Omega(n+1)}^{j+1} \end{pmatrix}^T, \tag{5.7}$$

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

$$\boldsymbol{y} = C\,\boldsymbol{x} \tag{5.8}$$

from the input vector

$$\boldsymbol{x} = \mathrm{I}_n^j \parallel \mathrm{B}^j \tag{5.9}$$

onto the output vector

$$\boldsymbol{y} = \mathrm{O}_n^j \parallel \mathrm{B}^j, \tag{5.10}$$

where $\parallel$ denotes the concatenation operator. The (5.8) is the most fundamental equation of this thesis. In this linear mapping, the matrix C defines the core.

The meaning and the number of individual coefficients in B^j is not firmly given. The choice of the matrix C is a degree of freedom of the presented framework. Particularly, this matrix can be reorganized in order to minimize some of the resources (e.g. memory cells, operations, latency). An illustrative example of such a reorganization is demonstrated in the next section.

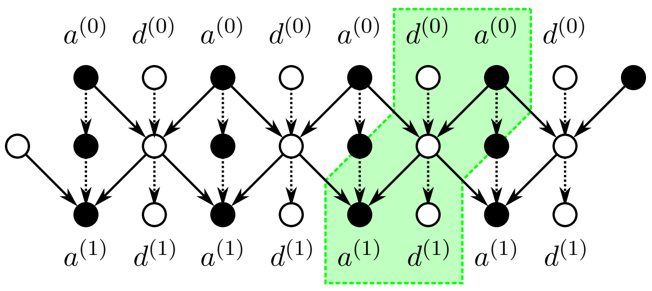


Figure 5.1: Implementation of CDF 5/3 transform. The core is the highlighted region. Circles represent the results of operations.

5.1 Core Reorganization

The following discussion is focused on a single pair of lifting steps of the vertically vectorized core. Since such a core consists of two lifting steps (predict and update), two data dependent blocks of operations have to be evaluated. This section demonstrates how to reduce the depth of the calculation per the output coefficients. These ideas were also presented in [III]. The basic principles of this idea is disclosed discussing the CDF 5/3 transform.

At the beginning, the CDF 5/3 lifting scheme is described. This description is related to Figure 5.1 (the notation in accordance with [8]). As in [8], CDF 5/3 lifting scheme is defined using α and β constants. The resulting core has 2 independent stages suitable for hardware pipelining. The core consists of 5 operations in total. The number of operands of the most demanding expression is 3 in both stages. The lag $F = 2$ or $F = 1$ can be obtained by a trivial reorganization of coefficients stored in the auxiliary buffer. The following discussion focuses on the latter case.

In more detail, the core transforms the vector $\boldsymbol{x}$ into $\boldsymbol{y}$. These two vectors are composed of two samples of input signal and some specific intermediate results as

$$\begin{aligned} \boldsymbol{x} &= \begin{bmatrix} a_n^{(0)} & d_n^{(0)} & \cdots \end{bmatrix}^T, \\ \boldsymbol{y} &= \begin{bmatrix} a_{n-1}^{(\cdot)} & d_n^{(\cdot)} & \cdots \end{bmatrix}^T. \end{aligned} \tag{5.11}$$

The individual stages correspond to predict T_α and update S_β steps, respectively. The core can be then described in matrix notation as

$$\boldsymbol{y} = C_{\alpha,\beta}\,\boldsymbol{x} = S_\beta\,T_\alpha\,\boldsymbol{x}. \tag{5.12}$$

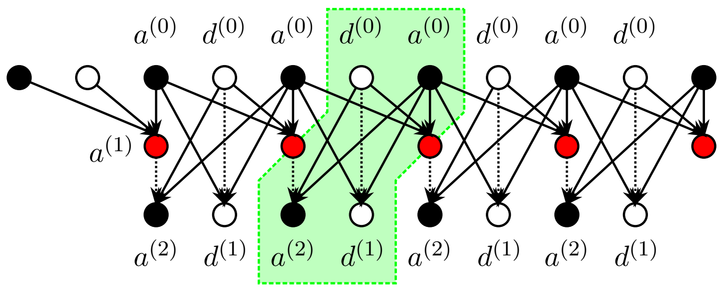

$a^{(0)}$ $d^{(0)}$ $a^{(0)}$ $d^{(0)}$ $a^{(0)}$ $d^{(0)}$ $a^{(0)}$ $d^{(0)}$
$a^{(1)}$
$a^{(2)}$ $d^{(1)}$ $a^{(2)}$ $d^{(1)}$ $a^{(2)}$ $d^{(1)}$ $a^{(2)}$ $d^{(1)}$

Figure 5.2: The 1-D implementation of CDF 5/3 filter with reduced latency. The circles in different color correspond to $a^{(1)}$ coefficients from (5.16). The core is highlighted.

The computation inside the $S_\beta T_\alpha$ may be implemented as

$$d^{(1)}_n = d^{(0)}_n + \alpha\left(a^{(0)}_{n-1} + a^{(0)}_n\right), \tag{5.13}$$

$$a^{(1)}_{n-1} = a^{(0)}_{n-1} + \beta\left(d^{(1)}_{n-1} + d^{(1)}_n\right). \tag{5.14}$$

For a better understanding, the slice of computation is depicted in Figure 5.1.

The reference implementation above has latency of 2 lifting steps. In the same single-loop framework, one can reduce [III] this latency to just one step. Such a core still uses the auxiliary buffer of $\kappa = 2$ coefficients, has the lag of one sample and produces numerically exactly the same results. However, the operations inside the core are appropriately reorganized. The key idea is a direct calculation of a coefficient from one intermediate coefficient and two current input samples. The intermediate coefficient is forwarded by previous core iteration through the auxiliary buffer. As a result, all the operations inside can be computed in parallel. The auxiliary buffer covers the values of $(a^{(0)}, a^{(1)})$. The resulting data-flow graph is shown in Figure 5.2. For clarity, a new constant $\gamma = 1 + 2\alpha\beta$ was introduced. The core implementation consists of

$$d^{(1)}_n = d^{(0)}_n + \alpha\left(a^{(0)}_{n-1} + a^{(0)}_n\right), \tag{5.15}$$

$$a^{(1)}_n = \gamma a^{(0)}_n + \alpha\beta a^{(0)}_{n-1} + \beta d^{(0)}_n, \tag{5.16}$$

$$a^{(2)}_{n-1} = a^{(1)}_{n-1} + \beta d^{(0)}_n + \alpha\beta a^{(0)}_n. \tag{5.17}$$

This new scheme is fundamentally different from the original one described earlier. In the original scheme, the a coefficient was calculated based on the value of the d coefficient. Unfortunately, the d coefficient had first to be calculated from the input samples. This process implied a delay of 2 steps. However, the newly formed core has a latency of 1 step. Note that the new scheme cannot be evaluated using the horizontal vectorization due to the formation of the new intermediate results $a^{(1)}$. On the other hand, the core

core	lag F	buffer κ	latency	max. operands	total operands
vertical lag-2	2	2	2	5	6
vertical lag-1	1	2	2	5	6
reorganized	1	2	1	3	9

Table 5.2: Attributes of CDF 5/3 cores. Only the cores based on the vertical vectorization are shown. The latency is the number of subsequent steps, max. operands identifies the most demanding operation of the core.

consists of 9 operations in total. See Table 5.2 for the comparison of all three mentioned cores. The total depth of the entire calculation is smaller (as the number of subsequent stages was halved) than in the original case. In matrix notation, the reorganized core can be expressed as

$$\boldsymbol{y} = C_{\alpha,\beta}\,\boldsymbol{x}. \tag{5.18}$$

The core reorganization takes new depths with the increasing number of dimensions. Before switching to multiple dimensions, further in the text, a graceful signal border treatment is demonstrated.

5.2 Treatment of Signal Boundaries

In order to keep the total number of wavelet coefficients equal to the number of input samples, symmetric border extension is widely used. A particular variant of this extension is employed in JPEG 2000 standard. Please, consult particular details with [70].

This section describes the core calculating the CDF 5/3 transform. The core described in the previous section consumes the input signal $\left(a^j_n, a^j_{n+1}\right)$ per fragments of two samples. After performing the calculations, the $\left(d^{j+1}_{\Omega(n)}, a^{j+1}_{\Omega(n+1)}\right)$ is produced with a lag F. For the purposes of the following discussion, only even length signals are considered. The core consists of two stages suitable for hardware pipelining.

As mentioned earlier, the core processes the signal in a single loop. The naive way [VIII] of border handling is described first. Due to the symmetric border extension, the core begins the processing at a certain position before the start of the actual signal. Analogously, the processing stops at a certain position after the end of the signal. The samples outside the actual signal are mirrored into the valid signal area. This processing introduces the need for buffering of the input, at least at the beginning and end of the signal. This buffering breaks the ability of simple stream processing, especially considering the multi-scale decomposition. All the approaches described in Chapter 3 also suffer from this issue.

The situation can be neatly resolved changing the core near the signal border. In more detail, the "mutable" core performs 5 different calculations depending on the position of the input signal. Therefore, the core comprises 2×5 slightly different steps (stages) in total. As in the previous section, each of them is implemented by a linear transformation operating with four-component vectors. This can be written in matrix notation as

$$\boldsymbol{y} = S_{\beta,\Theta(n)}\, T_{\alpha,\Theta(n)}\, \boldsymbol{x}, \tag{5.19}$$

where $T_{\alpha,\Theta(n)}, S_{\beta,\Theta(n)}$ are the linear transformations of the predict and update stages performed at the subsampled output position $\Theta(n)$. Moreover, $\boldsymbol{x} = \begin{bmatrix} a^b & d^b & a_n & d_n \end{bmatrix}^T$ and $\boldsymbol{y} = \begin{bmatrix} a^b & d^b & a_{n-1} & d_{n-1} \end{bmatrix}^T$ are the input and output vectors, respectively. Here, the b superscript denotes the content of the auxiliary buffer. These coefficients are generated in $T_{\alpha,\Theta(n)}$ so that these can be used by $T_{\alpha,\Theta(n+2)}$ at the same time when $S_{\beta,\Theta(n+2)}$ runs. It is essential that the coefficients a^b, d^b are initially set to zero. The output signal is generated with a lag $F = 1$ sample with respect to the input signal. The input a samples outside of the input signal are treated as zeros. Similarly, the output a, d coefficients outside of the output signal are discarded. The following table describes the individual $T_{\alpha,\Theta(n)}, S_{\beta,\Theta(n)}$ transformations. The transform is defined using the α, β constants. Table 5.3 enumerates the individual core stages. In addition, Figure 5.3 illustrates theirs usage. As a result, the signal is transformed without buffering, possibly on a multi-scale basis.

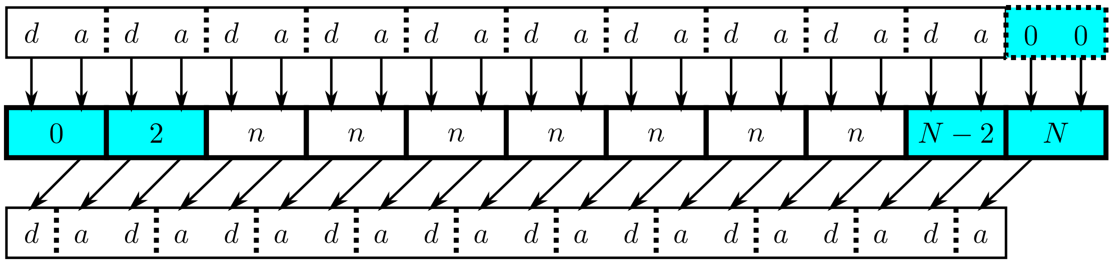

d a d a d a d a d a d a d a d a d a d a 0 0
0
2
n
n
n
n
n
n
n
N − 2
N
d a d a d a d a d a d a d a d a d a d a

Figure 5.3: Signal processing using the mutable core. The input position on the original position n is shown. The first and last two cores (highlighted) differ from the others. The very first and last cores access outside the signal. The input samples already split into a, d subbands.

$\Theta(n)$	$T^{-1}_{\alpha,\Theta(n)}$	$S^{-1}_{\beta,\Theta(n)}$	$T_{\alpha,\Theta(n)}$	$S_{\beta,\Theta(n)}$
0	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 2\alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$
1	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 2\beta & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$
$\Theta(n)$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$
$\Theta(N-2)$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$
$\Theta(N)$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2\alpha & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \beta & 1 & \beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ \alpha & 0 & \alpha & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2\beta \\ 0 & 1 & 0 & 0 \end{bmatrix}$

Table 5.3: Individual linear transformations inside the mutable CDF 5/3 core. Also inverse lifting steps $T^{-1}_{\alpha,\Theta(n)}, S^{-1}_{\beta,\Theta(n)}$ are shown. Changes are displayed in different color.

5.3 Lifting Scheme Choice

So far, only the CDF 5/3 and 9/7 wavelets have been discussed as illustrative examples. However, the presented computation unit is general. The following paragraph generalizes the developed framework to other wavelets.

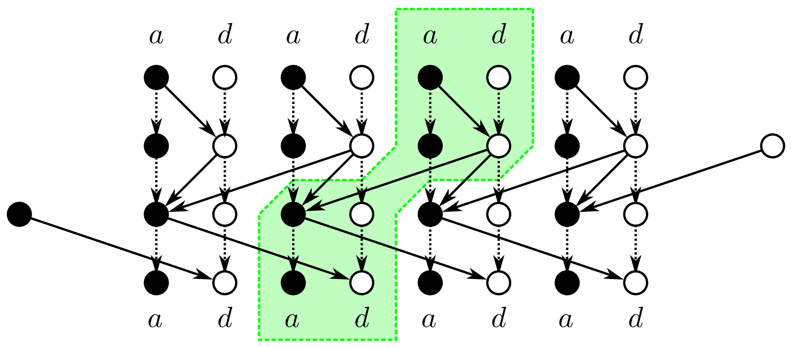
One can identify the core in arbitrary underlying lifting schemes. However, its implementation can be obscure in some cases mainly due to the increasing number of intermediate results in auxiliary buffers. For his reason, a lifting factorization employing steps in the form of two-tap filters having degree 1 is a proven choice. Many such factorizations of various wavelets have been presented in the literature, e.g. [8]. Moreover, the symmetric filters with lengths $2L \pm 1$, as is the case for the CDF 5/3 and 9/7 filters, can be implemented through some sequence of lifting steps having this particular form. See Chapter 6 of [70] for details. More precisely, exactly $L = 2I$ such symmetric two-tap lifting steps are obtained.

In [8], the authors demonstrated three different implementations of a 4-tap orthonormal filter with two vanishing moments (often referred to as D4 wavelet). The corresponding data-flow diagrams are shown in Figure 5.4. For simplicity, the scalings are omitted. Suitable examples of one-dimensional single-loop cores are highlighted. However, these are not the only possible cores which can be identified in given data flows. Some observations can be made from these diagrams. The core in the first implementation needs to pass 3 coefficients between iterations. Similarly, the cores in the second and third implementations need to pass 2 values. All the cores have latency of three lifting steps. The first two cores have a lag of 2 signal samples relative to the input signal. The third core has a lag of one sample.

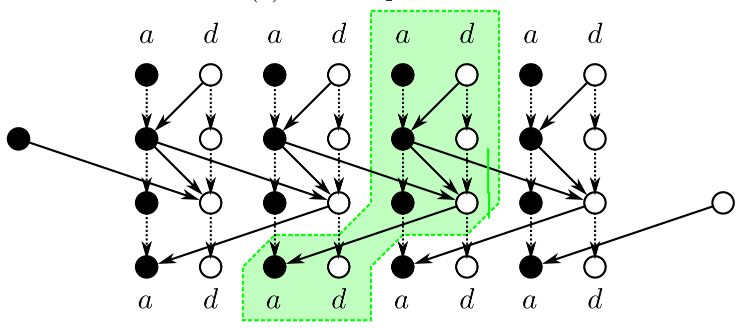
One particular implementation will be presented here, for the purpose of illustration. Thanks to the small amount of forwarded values and the smallest lag, the third implementation has been chosen. At the beginning, the fragment of the input signal is split into the pair $(a, d)^{(0)}$. Then, the sequence of predict $d_n^{(1)}$, update $a_{n-1}^{(1)}$ and predict $d_n^{(2)}$ operations is performed. Finally, the coefficients are scaled resulting into (a_{n-1}, d_n). The $(d^{(1)}, a^{(1)})$ coefficients are forwarded into subsequent iteration through the auxiliary buffer. The implementation

$$\begin{aligned} d_n^{(1)} &= d_n^{(0)} + \alpha a_n^{(0)}, \\ a_{n-1}^{(1)} &= a_{n-1}^{(0)} + \beta d_{n-1}^{(1)} + \gamma d_n^{(1)}, \\ d_n^{(2)} &= d_n^{(1)} + \delta a_{n-1}^{(1)} \end{aligned} \tag{5.20}$$

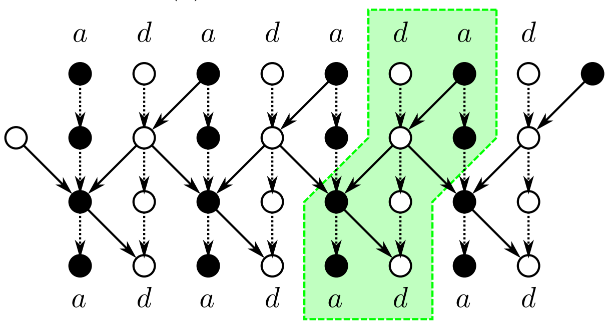
causes latency of 3 lifting steps plus subsequent scaling. Using the matrix notation, the



(a) First implementation.



(b) Second implementation.



(c) Third implementation.

Figure 5.4: Three different implementations of D4 transform. The cores are highlighted.

core can be described as

$$\boldsymbol{y} = C\,\boldsymbol{x} = S_\delta^2\,T_{\beta,\gamma}\,S_\alpha^1\,\boldsymbol{x}, \tag{5.21}$$

where the individual T, S operators implement the equations (5.20).

5.4 Parallel Cores

This section considers conditions under which the cores can run simultaneously. The methods presented here were used earlier in several papers, e.g. [60]. Their work was further examined and improved in [IX, X]. The possibility to run the cores in parallel is discussed mainly as a basis for the next chapter.

Single-loop cores presented in this chapter require the auxiliary buffer [VIII] to pass the intermediate results into subsequent iteration. Such a strategy is however not suitable for all platforms, especially for the massively-parallel architectures. Considering such platforms, it would be preferable to run all the cores in parallel. Indeed, such approaches were already presented in the literature, e.g. [60]. However, they were not referred to as the "cores" so far. These will be referred to as parallel cores in this section.

The single-loop cores can be notionally modified to exchange the intermediate results directly without the auxiliary buffers. In parallel environment, it causes the need of synchronization points. These points are denoted as the memory barriers. Particular example of such a barrier is shown in Figure 5.5. In the referenced figure, the dashed horizontal line indicates the point where the processing units have to be synchronized. The initial values can be loaded without the synchronization. However, the intermediate results after the first lifting steps can be loaded no sooner than after the synchronization

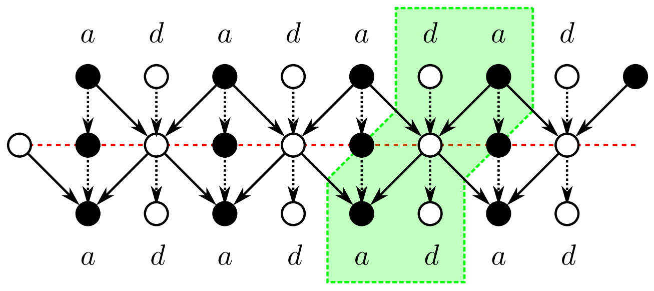


Figure 5.5: Parallel implementation of CDF 5/3 transform. The core is highlighted. The dashed horizontal line identifies the memory barrier. This line intersects the d coefficients (blank bullets), which are, therefore, subject of the synchronization.

point. Note that it is acceptable to pass the intermediate results in either direction – to the left and to the right.

In conclusion, this chapter presented a new formulation of the lifting scheme. This new formulation has the ability to retain certain intermediate results through the introduced auxiliary buffers. This was not possible in the original scheme. As a consequence of the presented formalism, the computation can be modified in such a way to meet different requirements on various platforms. The next chapter extends the formalism into multiple dimensions.

Chapter 6

Multi-Dimensional Cores

The presented core approach can be naturally extended to multiple dimensions. The key ideas of this section were presented in [VI, VIII, II]. This chapter is particularly focused on two-dimensional cores. However, the same principles also apply to more dimensions. Several benefits of the implementation arise by extending the core into two dimensions. Thanks to the linear nature of DWT, the horizontal and vertical steps can be interleaved or even merged. Merging of the final coefficient scaling is a useful involvement of this property.

The extension into two dimensions follows. To simplify the relations, the inequality $(0,0) \leq (m_j, n_j) < (M_j, N_j)$ holds for all $0 \leq j \leq J$. The 2-D transform decomposes the input raster

$$\left(a^0_{m_0,n_0}\right) \tag{6.1}$$

of size $M_0 \times N_0$ pixels into $J > 0$ scales giving rise to the temporary subbands

$$\left(a^j_{m_j,n_j}\right), \tag{6.2}$$

the resulting wavelet subbands

$$\left(h^j_{m_j,n_j}\right), \left(v^j_{m_j,n_j}\right), \left(d^j_{m_j,n_j}\right), \tag{6.3}$$

at scales $0 < j < J$, and the residual signal

$$\left(a^J_{m_J,n_J}\right) \tag{6.4}$$

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

$$\left({}^M B^j_{m_j}\right)_{0 \leq m_j < M_j}, \left({}^N B^j_{n_j}\right)_{0 \leq n_j < N_j}. \tag{6.5}$$

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

Similarly to in the 1-D case, the 2-D core consumes a 2×2 fragment of the input signal and immediately produces a four-tuple of coefficients (a, h, v, d). The produced coefficients have a delay of F samples in the vertical as well as the horizontal direction with respect to the input coordinate system. To simplify relations, two functions will be introduced once again

$$\Theta(m,n) = (m+F, n+F), \text{ and } \Omega(m,n) = (\lceil m/2 \rceil, \lceil n/2 \rceil). \tag{6.6}$$

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 $j+1$. It should be noted that $\Omega(m,n)$ can be defined in many ways. However, this particular example fits into the JPEG 2000 coordinate system. The 2×2 core transforms the fragment $\mathrm{I}^j_{m,n}$ of the input signal onto the fragment $\mathrm{O}^j_{m,n}$ of the output signal

$$\mathrm{I}^j_{m,n} = \begin{pmatrix} a^j_{\Theta(m,n)} & a^j_{\Theta(m+1,n)} & a^j_{\Theta(m,n+1)} & a^j_{\Theta(m+1,n+1)} \end{pmatrix}^T, \tag{6.7}$$

$$\mathrm{O}^j_{m,n} = \begin{pmatrix} a^{j+1}_{\Omega(m,n)} & h^{j+1}_{\Omega(m+1,n)} & v^{j+1}_{\Omega(m,n+1)} & d^{j+1}_{\Omega(m+1,n+1)} \end{pmatrix}^T, \tag{6.8}$$

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

$$\boldsymbol{y} = C\,\boldsymbol{x} \tag{6.9}$$

from the input vector

$$\boldsymbol{x} = \mathrm{I}^j_{m,n} \parallel {}^M\mathrm{B}^j_m \parallel {}^M\mathrm{B}^j_{m+1} \parallel {}^N\mathrm{B}^j_n \parallel {}^N\mathrm{B}^j_{n+1} \tag{6.10}$$

onto the output vector

$$\boldsymbol{y} = \mathrm{O}^j_{m,n} \parallel {}^M\mathrm{B}^j_m \parallel {}^M\mathrm{B}^j_{m+1} \parallel {}^N\mathrm{B}^j_n \parallel {}^N\mathrm{B}^j_{n+1}, \tag{6.11}$$

where $\parallel$ denotes the concatenation operator. One needs to recall that the choice of the C matrix and the consequent arrangement and the size κ of elements in the buffers is the subject of this thesis.

Considering the SIMD extensions, the two-dimensional core can nicely exploit capabilities of modern GPPs. Moreover, as the 2-D data occupy one order of magnitude more memory compared to the 1-D signals, the processing can be divided across multiple independent processing units. Combined with the single-loop approach [13], the cache hierarchy can also be properly utilized. The influence of all these possibilities was investigated in [VIII].

So far, the main ability of the 2-D extension remains undisclosed. The single loop over the data does not have a strictly fixed order. On the contrary, many scan orders are now possible. Some of them are depicted in Figure 6.1. It should be noted that the original single-loop approach from [40] does not have this ability. The above-described degree of freedom allows us to adapt the processing to specific needs of the application. For instance, it turned out that the 2-D core approach can be adapted to JPEG 2000 coding units (so-called codeblocks) in [II]. When associated with the capabilities explained in the previous paragraph, these codeblocks can be generated in parallel. This experiment is further evaluated below.

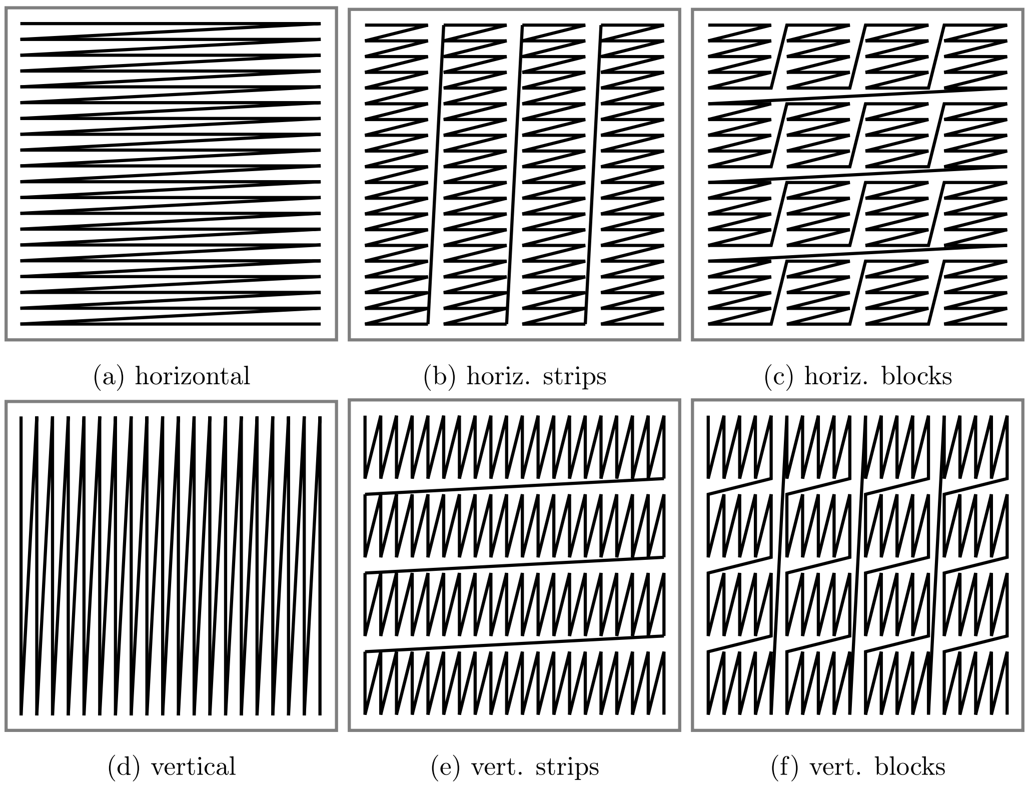


Figure 6.1: Some of the processing orders enabled by the core approach.

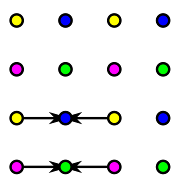
6.1 2-D Core Reorganization

For purposes of illustration, the following text is focused on two-dimensional CDF 5/3 transform. The notation used for the description of 2-D diagrams is explained in Figure 6.2.

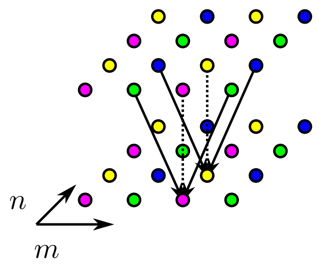
Considering the baseline separable extension into two dimensions resulting into a 2×2 core, the matrix C in the relationship $\boldsymbol{y} = C\boldsymbol{x}$ can be factored into

$$\boldsymbol{y} = {}^{N}S_{\beta}\ {}^{N}T_{\alpha}\ {}^{M}S_{\beta}\ {}^{M}T_{\alpha}\,\boldsymbol{x}, \tag{6.12}$$

where the M superscripts refer to the horizontal direction, whereas N refers to the vertical one. Taken together, ${}^{M}T_{\alpha}$ performs two horizontal predicts, ${}^{M}S_{\beta}$ two horizontal updates, etc. The order of these steps (or stages) is not only strictly fixed but also completely unconstrained. The implementation has the latency of four lifting steps, plus scaling. The scheme is graphically illustrated in Figure 6.3. In total, 16 non-trivial operations (four in each stage) are needed to calculate this core (the scaling is omitted).



(a) 2-D view



(b) 3-D view

Figure 6.2: Two views of 2-D data-flow diagrams. The same diagram in (a) top view and (b) in 3-D space. The dotted arrows are not possible to display in the top view.

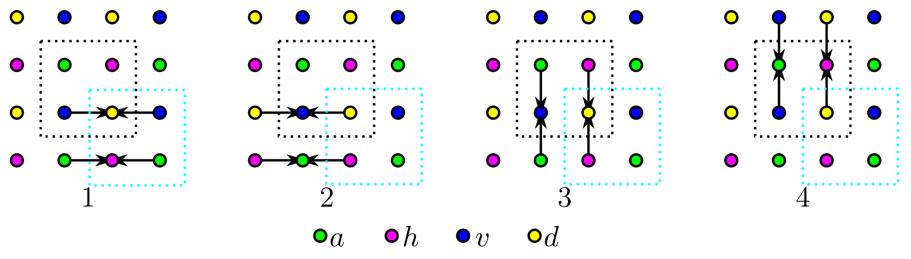


Figure 6.3: The separable implementation of CDF 5/3 core with 4 stages. The output is highlighted in dark, the input in bright. The arrows correspond to the multiply operations which are accumulated into the bullets.

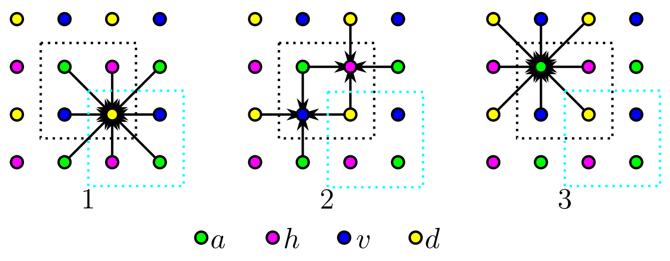


Figure 6.4: The non-separable 3-stage implementation of CDF 5/3 core. The input (in bright) and the output (in dark) of the core are highlighted.

In [16], the authors derived a non-separable 2-D lifting scheme for CDF 5/3 DWT. One can easily identify a suitable core in their construction. The same core can be obtained by proper reorganization of operations inside the separable 2×2 core. The result is shown in Figure 6.4. Thanks to the parallel processing of v and h samples, this implementation has a total latency of 3 steps.

The more detailed description of this non-separable core follows. As initial step, a 2×2 fragment of the input signal is notionally split into the input quadruple $(a, v, h, d)^{(0)}_{m,n}$. Then, the prediction of the detailed coefficient is performed. This is followed by parallel prediction of the horizontal and vertical coefficients. In third step, the approximation coefficient is updated. Finally, the four coefficients corresponding to four subbands can be scaled. Ten coefficients need to be forwarded through the auxiliary buffers. However, it is not necessary to construct such a huge buffer. Instead, the buffers from a separable implementation can be used complemented by small 2×2 buffer for the exchange of intermediate results in 2-D. Therefore, practical implementations with a horizontal image scanning order would require two rows of coefficients to be buffered. The core has the lag F of one sample in each direction.

Using the matrix notation, the core is described as

$$\boldsymbol{y} = A_\beta \, T_{\alpha,\beta} \, D_\alpha \, \boldsymbol{x}, \tag{6.13}$$

where D_α operator computes the d coefficient, $T_{\alpha,\beta}$ computes h and v, and A_β finally computes a coefficient. It is easy to identify buffers forming borders between the consequent cores. Excluding diagonals, the matrices $A_\beta, T_{\alpha,\beta}, D_\alpha$ have a total of 24 non-zero entries implying 24 non-trivial MAC operations.

Using the core approach presented in this thesis, it is possible to go further. For now, ignore the separable core comprising the reorganized cores from the previous chapter. The total number of arithmetic operations in the non-separable scheme [16] can be reduced. The key idea here is not to calculate the wavelet coefficients all at once. Instead, the calculation of these coefficients is subdivided into separate parts. The sum of these parts gives us the desired result.

This approach will be now explained in detail. The starting point of the solution is the non-separable lifting scheme of CDF 5/3 transform as described in [16]. The operations inside the core were further appropriately reorganized. After some rewriting of expressions, the following scheme has appeared. First, the scheme requires only 8 coefficients to be passed between core iterations. This means that this new scheme has the same memory demands as in the original separable case. Second, the number of the non-trivial arithmetic operations was reduced from 24 to 22 compared to the original non-separable scheme. The trick is that the h, v and a coefficients are not calculated at once. Instead, these are calculated separately, split in two parts each.

For better understanding, the new scheme is graphically illustrated in Figure 6.5. The $(v^{(0)}_{m,n}, a^{(0)}_{m,n}, v^{(1)}_{m,n}) \to (v^{(0)}_{m-1,m}, a^{(0)}_{m-1,n}, v^{(1)}_{m-1,n})$ coefficients have to be passed through the buffer horizontally. Additional $(h^{(0)}_{m,n}, a^{(0)}_{m,n}, h^{(1)}_{m,n}) \to (h^{(0)}_{m,n-1}, a^{(0)}_{m,n-1}, h^{(1)}_{m,n-1})$ coefficients have to be passed vertically. Lastly, $(a^{(0)}_{m,n}, a^{(1)}_{m,n}) \to (a^{(0)}_{m-1,n-1}, a^{(1)}_{m-1,n-1})$ coefficients have to be passed in diagonal direction. This gives 8 coefficients per core iteration in total. It should be emphasized that this newly formed scheme cannot be derived using instruments in [16]. This is caused by the fact that the authors of [16] do not specify a sequence of the operations. For comparison, shapes of the individual lifting steps are shown in Figure 6.7. Using the matrix notation, the proposed transform core can be described as

$$\boldsymbol{y} = A_\beta \, T_{\alpha,\beta} \, D_\alpha \, \boldsymbol{x}, \tag{6.14}$$

where

$$\begin{aligned} \boldsymbol{x} &= \begin{bmatrix} a_{m-1,n-1} & a_{m,n-1} & a_{m-1,n} & a_{m,n} & h_{m,n-1} & h_{m,n} & v_{m-1,n} & v_{m,n} & d_{m,n} \end{bmatrix}^T \\ \boldsymbol{y} &= \begin{bmatrix} a_{m-1,n-1} & a_{m,n-1} & a_{m-1,n} & a_{m,n} & h_{m,n-1} & h_{m,n} & v_{m-1,n} & v_{m,n} & d_{m,n} \end{bmatrix}^T \end{aligned} \tag{6.15}$$

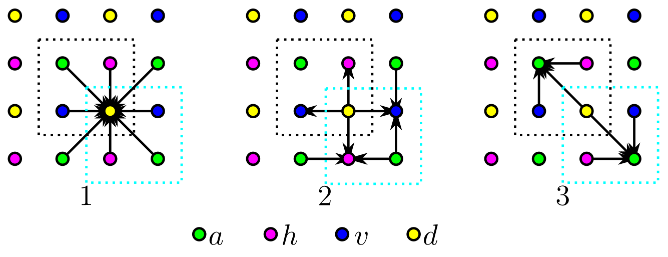


Figure 6.5: Proposed non-separable lifting core of CDF 5/3 with three stages. The input coefficients of active core are in bright box. The output ones are in dark one.

and

$$D_\alpha = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \alpha^2 & \alpha^2 & \alpha^2 & \alpha^2 & \alpha & \alpha & \alpha & \alpha & 1 \end{bmatrix}, \tag{6.16}$$

$$T_{\alpha,\beta} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \beta \\ 0 & 0 & \alpha & \alpha & 0 & 1 & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \beta \\ 0 & \alpha & 0 & \alpha & 0 & 0 & 0 & 1 & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6.17}$$

$$A_\beta = \begin{bmatrix} 1 & 0 & 0 & 0 & \beta & 0 & \beta & 0 & -\beta^2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \beta & 0 & \beta & -\beta^2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.18}$$

The implementation is always a trade-off between latency and the number of operations. In next example, another non-separable implementation with latency of 2 steps is shown. This implementation has the same buffer requirements. The achieved number of operations is 22. In matrix notation, the transform can be written as the product

$$\boldsymbol{y} = A_\beta \, D_\alpha \, \boldsymbol{x}, \tag{6.19}$$

where $\boldsymbol{x}, \boldsymbol{y}$ are defined as shown above. On the contrary, the A_β, D_α are defined below.

$$D_\alpha = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha & \alpha & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \alpha & 0 & \alpha & 0 & 0 & 0 & 1 & 0 \\ \alpha^2 & \alpha^2 & \alpha^2 & \alpha^2 & \alpha & \alpha & \alpha & \alpha & 1 \end{bmatrix} \tag{6.20}$$

$$A_\beta = \begin{bmatrix} 1 & 0 & 0 & 0 & \beta & 0 & \beta & 0 & \beta^2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \beta & 0 & \beta & \beta^2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.21}$$

The steps are graphically illustrated in Figure 6.6.

Table 6.1 provides a summarized comparison of the discussed 2-D single-loop cores. The most complicated calculation from all the steps is indicated in the last column. This number is given in the format of the non-trivial operations plus the trivial operations. As before, the scalings were omitted. When the stages of the core are pipelined (run in

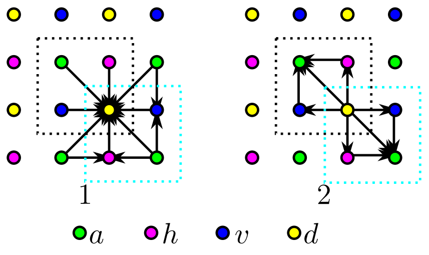


Figure 6.6: Proposed non-separable lifting core of CDF 5/3 with two stages. The input coefficients of the active core are in the bright box. The output ones are in the dark one.

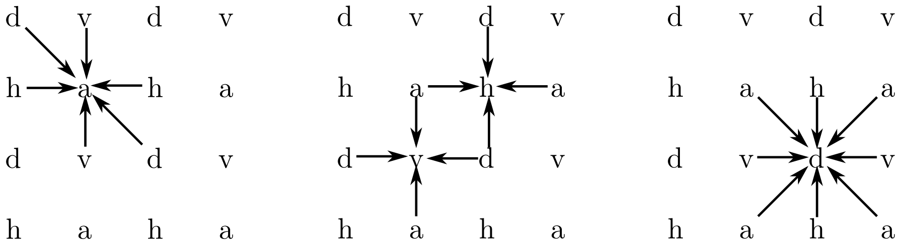

d v d v h a h a d v d v h a h a
d v d v h a h a d v d v h a h a
d v d v h a h a d v d v h a h a

Figure 6.7: Shapes of the individual lifting steps of the newly proposed CDF 5/3 scheme with three steps.

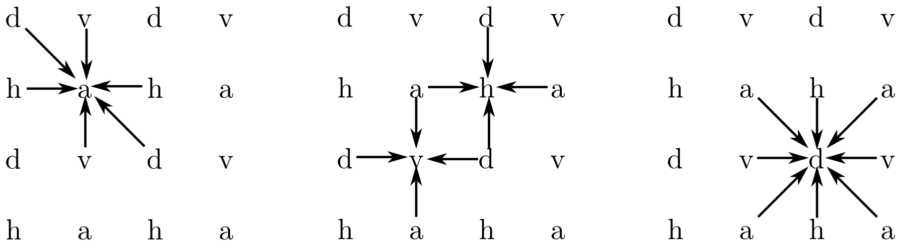

d v d v h a h a d v d v h a h a
d v d v h a h a d v d v h a h a
d v d v h a h a d v d v h a h a

Figure 6.8: Shapes of the individual lifting steps of newly proposed CDF 5/3 scheme with two steps.

core	latency	buffer	operations	max. operands in step
separable	4	8	16	$2+1$
non-separable [16]	3	10	24	$8+1$
non-separable new	3	8	22	$8+1$
non-separable new	2	8	22	$8+1$

Table 6.1: Comparison of the 2-D single-loop cores. The operands are given in format non-trivial plus trivial. The scaling is omitted.

parallel), the clock latency of the core is directly subordinated by the maximum number of operands. The table further indicates the number of stages (steps), the number of coefficients accessed in the auxiliary buffers, and the total number of non-trivial operations.

6.2 Parallel 2-D Cores

So far, only the single-loop two-dimensional cores were discussed. Considering the parallel environment, the cores can be modified in order to run in parallel. In such a case, the cores have to exchange the intermediate results directly, without buffers. This modification introduces the need for synchronization using the memory barrier. Usually, these barriers form the major bottleneck of the overall computation. Taken together, it is desirable to minimize the number of memory barriers (along with another resources). The cores discussed in this section were proposed in [X]. This section is still focused on the CDF 5/3 transform. The generalization is straightforward.

Iwahashi *et al.* [14, 15, 16] recently presented the non-separable lifting scheme employing genuine spatial filtering steps $\mathrm{H_a}, \mathrm{H_a^*}$, and $\mathrm{H_a H_a^*}$. For details, see Section 2.3. In this scheme, it is no longer possible to distinguish the vertical and horizontal filtering. The transform can be described as linear transformations of the vectors

$$\begin{aligned} \boldsymbol{x} &= \begin{bmatrix} a & h & v & d \end{bmatrix}^\mathrm{T}, \\ \boldsymbol{y} &= \begin{bmatrix} a & h & v & d \end{bmatrix}^\mathrm{T}. \end{aligned} \tag{6.22}$$

These transformations can formally be compressed into the matrix $C_{\alpha,\beta}$ in

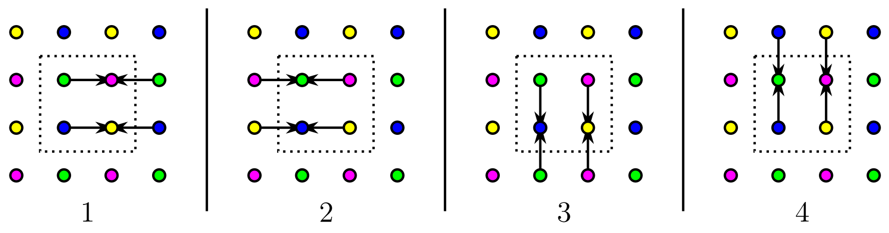
$$\boldsymbol{y} = C_{\alpha,\beta}\,\boldsymbol{x} = A_\beta\,T_{\alpha,\beta}\,D_\alpha\,\boldsymbol{x}, \tag{6.23}$$

where

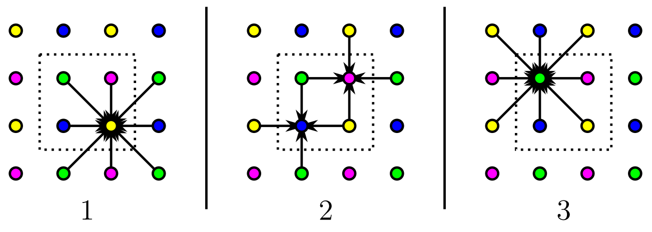
$$C_{\alpha,\beta} = \begin{bmatrix} 1 & \overline{\mathrm{H}}_\beta & \overline{\mathrm{H}}^*_\beta & -\overline{\mathrm{H}}_\beta\overline{\mathrm{H}}^*_\beta \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{H}_\alpha & 1 & 0 & \overline{\mathrm{H}}^*_\beta \\ \mathrm{H}^*_\alpha & 0 & 1 & \overline{\mathrm{H}}_\beta \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \mathrm{H}_\alpha\mathrm{H}^*_\alpha & \mathrm{H}^*_\alpha & \mathrm{H}_\alpha & 1 \end{bmatrix}. \tag{6.24}$$

The scheme is graphically illustrated in Figure 6.9b (referred to as *Iwahashi2007*). Similarly in the original scheme, a memory barrier must be inserted between each of the two steps. As a result, the scheme consists of 24 non-trivial arithmetic operations in three lifting steps separated by two explicit memory barriers. The most complex operation is calculated over 9 operands which leads to a performance issue. This is caused by the number of operands being proportional to the data path with the maximum delay. For the sake of comparison, the baseline separable scheme is illustrated in Figure 6.9a (referred to as *Sweldens1995*). Note that the scheme for CDF 9/7 comprises two such connected transforms.

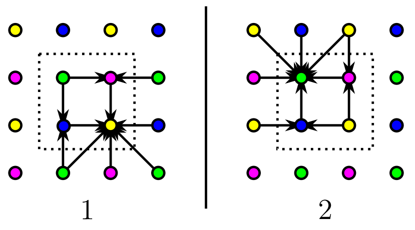
Motivated by the work of Iwahashi *et al.* [16], the elementary lifting filters were reorganized in order to obtain a highly parallelizable scheme. The main purpose of this modification is to minimize the number of memory barriers that slow down the calculation. As a result, several non-separable two-dimensional FIR filters arise.



(a) Sweldens1995 [6]



(b) Iwahashi2007 [14]



(c) proposed

🟢a 🟣h 🔵v 🟡d

Figure 6.9: 2-D data-flow graphs of the parallel cores. The order of the lifting steps is determined by the bottom numbers. The vertical lines indicate the necessary memory barriers.

The formed scheme is composed of three elementary filters F, G, H given by

$$\begin{bmatrix} \mathrm{F_a} \\ \mathrm{G_a} \\ \mathrm{H_a} \end{bmatrix} = \begin{bmatrix} F_a(z_m, z_n) \\ G_a(z_m, z_n) \\ H_a(z_m, z_n) \end{bmatrix} = \begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} 1 \\ z_n \\ 1 + z_m \end{bmatrix}, \tag{6.25}$$

where c denotes a filter parameter.

The filters above are assembled into more complex operations. The scheme consists of two parts between which a memory barrier is placed. The first half of the scheme uses the following filters. Similarly, the second half uses these filters in the reverse orientation.

$$\begin{bmatrix} \mathrm{F_a} \\ \mathrm{G_a} \\ \mathrm{H_a} \\ \mathrm{H_a^*} \\ \mathrm{G_a H_a} \end{bmatrix} = \begin{bmatrix} c \\ c\, z_n \\ c\,(1 + z_m) \\ c\,(1 + z_n) \\ c^2\,(z_n + z_m z_n) \end{bmatrix} \tag{6.26}$$

$$\begin{bmatrix} \overline{\mathrm{F}}_\mathrm{a} \\ \overline{\mathrm{G}}_\mathrm{a} \\ \overline{\mathrm{H}}_\mathrm{a} \\ \overline{\mathrm{H}}_\mathrm{a}^* \\ \overline{\mathrm{G}}_\mathrm{a}\overline{\mathrm{H}}_\mathrm{a} \end{bmatrix} = \begin{bmatrix} c \\ c\, z_n^{-1} \\ c\,(1 + z_m^{-1}) \\ c\,(1 + z_n^{-1}) \\ c^2\,(z_n^{-1} + z_m^{-1} z_n^{-1}) \end{bmatrix} \tag{6.27}$$

Finally, the new scheme is composed of four operators referred to as S^1 to S^4. Between the second S^2 and third S^3 operator, the memory barrier must be inserted in order to properly exchange intermediate results. Thus, S^1 and S^2 form the first lifting step and S^3 and S^4 form the second one. Note that it is also possible to rewrite the scheme using six operators instead of four. It would be also possible to rewrite the scheme with just two operands, however, it is not possible to capture a retention of intermediate results in such a case. Additionally, the scheme requires the induction of two auxiliary variables (the intermediate results) per each quadruple of coefficients a, h, v, and d. These auxiliary variables are denoted as $h^{(1)}, v^{(1)}$. Their initial as well as final values are not important.

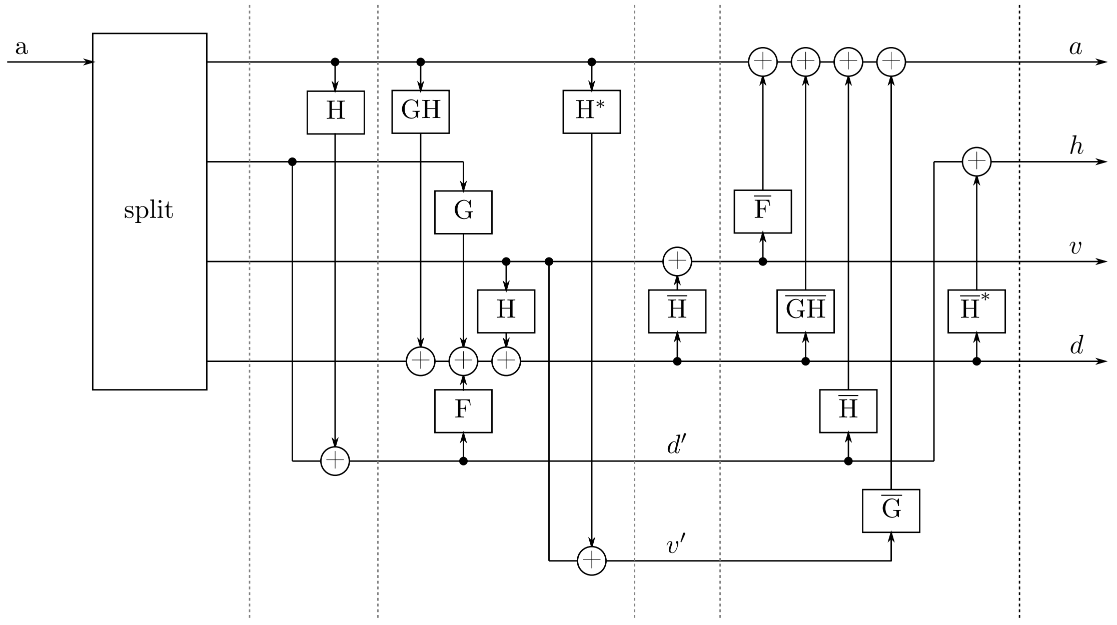


Figure 6.10: Block diagram of the parallel non-separable latency-2 core. The individual operators S are separated by the vertical lines. The memory barrier is placed in between S^2 and S^3 (the middle line).

The scheme

$$\boldsymbol{y} = S^4_\beta \, S^3_\beta \, S^2_\alpha \, S^1_\alpha \, \boldsymbol{x} \tag{6.28}$$

describes the relation between input $\boldsymbol{x}$ and output $\boldsymbol{y}$ vectors. It should be noted that in practical realizations, each single computing unit (e.g. thread) can be responsible of such a vector. The vectors are given by the following equations.

$$\begin{aligned} \boldsymbol{x} &= \begin{bmatrix} a & h & v & d & h^{(1)} & v^{(1)} \end{bmatrix}^{\mathrm{T}} \\ \boldsymbol{y} &= \begin{bmatrix} a & h & v & d & h^{(1)} & v^{(1)} \end{bmatrix}^{\mathrm{T}} \end{aligned} \tag{6.29}$$

Regarding this notation, the individual steps are defined as follows. For a better understanding, the hypothetical signal-processing block diagram of this scheme is shown in Figure 6.10. In addition, the operations are graphically illustrated in Figure 6.9c (referred to as *proposed*). Note that operators S^1 and S^2 are represented by the first lifting step and operators S^2 and S^3 by the second one.

$$S_\alpha^1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \mathrm{H}_\alpha & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.30}$$

$$S_\alpha^2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \mathrm{G}_\alpha\mathrm{H}_\alpha & \mathrm{G}_\alpha & \mathrm{H}_\alpha & 1 & \mathrm{F}_\alpha & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \mathrm{H}_\alpha^* & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{6.31}$$

$$S_\beta^3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \overline{\mathrm{H}}_\beta & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.32}$$

$$S_\beta^4 = \begin{bmatrix} 1 & 0 & \overline{\mathrm{F}}_\beta & \overline{\mathrm{G}}_\beta\overline{\mathrm{H}}_\beta & \overline{\mathrm{H}}_\beta & \overline{\mathrm{G}}_\beta \\ 0 & 0 & 0 & \overline{\mathrm{H}}_\beta^* & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.33}$$

Compared with [16], the total number of arithmetic operations has been reduced from 24 to 20 for the CDF 5/3 wavelet. The calculation of the CDF 9/7 transform consists of two such connected transforms (the first with α, β, the second with γ, δ) and between them another barrier is placed. In total, such a calculation contains three explicit memory barriers.

scheme	steps	operations	max. operands	memory cells
Sweldens1995 [6]	4	16	3	4
Iwahashi2007 [14]	3	24	9	4
proposed	**2**	20	9	6

Table 6.2: Parameters of the 2-D parallel cores for CDF 5/3 wavelet. The columns describe: number of lifting steps, number of arithmetic operations, maximum number of operands per the lifting step result (the complexity of steps), number of memory cells per coefficient quadruple (inclusive).

A quantitative comparison for the CDF 5/3 wavelet of all the cores discussed is provided in Table 6.2. For the CDF 9/7 wavelet, the number of lifting steps and thus the number of operations must be doubled. In general, the cores can be used for any lifting factorization with two-tap filters.

The original *Sweldens1995* scheme provides the best choice in terms of arithmetic operands as well as their complexity. However, it requires three explicit synchronization points (memory barriers) for the CDF 5/3 wavelet. This can be an issue for parallel processing. The recently proposed *Iwahashi2007* scheme uses the highest number of operations of all schemes. On the other hand, it requires only two synchronizations for the CDF 5/3 wavelet and does not need any additional memory. In numbers, this scheme reduces the number of lifting steps to 75 %. Finally, the *proposed* scheme provides a trade-off in the number of operations. Moreover, for the CDF 5/3 wavelet, only one barrier is needed for its realization. In comparison to the original scheme, this scheme reduces the number of lifting steps to only 50 %. In memory limited environments, the 1.5× higher memory consumption can be seen as a restriction.

6.3 Extension to Multiple Dimensions

Similarly to the previous 2-D extension, the 2^3 cores transforming the 3-D data in the single loop were proposed in [VII]. Access to three 2-D auxiliary buffers is required during the computation. As in the previous cases, the implementation can further be extended to allow SIMD-optimizations.

In the most generic variant, each 2-D auxiliary buffer has the same size as the corresponding volume side. The depth of each auxiliary buffer is κ coefficients. See Figure 6.11. This memory consumption can be reduced using an appropriate processing order. When

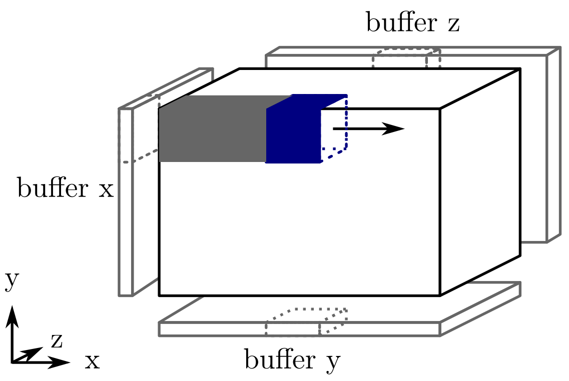


Figure 6.11: Complete processing by the 3-D single-loop core. The auxiliary buffer for each dimension is shown on the sides. The x, y, z notation is used instead of m, n.

using the horizontal (raster scan) order, it is not necessary to allocate the full side size buffers. It is sufficient to allocate only the following sizes. Allocate one full side size buffer for the first dimension. Allocate one edge size buffer for the second dimension. Finally, allocate one point size buffer for the third dimension. For instance, considering the 2^3 core, it is sufficient to allocate buffers of total size $\kappa\,(NM + 2N + 4)$ elements, where N, M are sizes of the volume in first two dimensions.

Considering the separable implementation, the core consists of three blocks performing calculations corresponding to the three dimensions. Each block updates the necessary intermediate results in the corresponding auxiliary buffer. This is followed by scaling of the output coefficients.

The similar extension to an arbitrary number of dimensions can be constructed. In the general case, the core has a size of 2^Π samples, where Π is the number of dimensions. Such a core needs to access into Π buffers. Each of them is indexed by $\Pi - 1$ coordinates. Following the previous discussion, it is not necessary to allocate the complete $(\Pi - 1)$-dimensional buffers. Instead, only a small fraction of them is actually needed. This fraction is proportional to a single $(\Pi - 1)$-dimensional hypercube.

Formally, for each scale $0 \leq j < J$, the 2^Π core requires an access to Π auxiliary buffers

$$\left({}^{\pi}B^{j}_{{}^{\pi}\lambda_j}\right) \tag{6.34}$$

for $0 \leq \pi < \Pi$ and $0 \leq {}^{\pi}\lambda_j < {}^{\pi}\Lambda_j$, where ${}^{\pi}\Lambda_j$ is the size of the data in π-direction at the scale j. For simplicity, the following description is limited on a particular scale j and the scripts j are thus dropped. Moreover, let λ to be the vector of coordinates $({}^{\pi}\lambda)_{0 \leq \pi < \Pi}$, $\mathbf{I}_\lambda$ to be the vector of 2^Π input coefficients, $\mathbf{O}_\lambda$ to be the vector of 2^Π output coefficients, and $\mathbf{B}_\lambda$ to be the vector $({}^\pi B_{{}^\pi l})_{0 \le \pi < \Pi, {}^\pi\lambda \le {}^\pi l < {}^\pi\lambda+2}$ of corresponding buffer fragments. Then the core is described as the mapping

$$\boldsymbol{y} = C\,\boldsymbol{x} \tag{6.35}$$

from the input vector

$$\boldsymbol{x} = \mathbf{I}_\lambda \parallel \mathbf{B}_\lambda \tag{6.36}$$

onto the output vector

$$\boldsymbol{y} = \mathbf{O}_\lambda \parallel \mathbf{B}_\lambda. \tag{6.37}$$

Again, the choice of matrix C and the arrangement of the buffers is not fixed and can be subject to optimization with respect to some criterion.

Chapter 7

Evaluation

This chapter provides deep performance evaluation of the presented core approach. At the beginning, a variety of experiments with the core approach on GPPs is presented. This also includes JPEG 2000 encoding process. Subsequently, the chapter focuses on other platforms. Namely, the decomposition on FPGA and GPU is evaluated. At the end, several experiments investigating the lifting scheme vectorizations are presented.

7.1 Image Processing

First, the SIMD-vectorization and parallelization of the 2-D transform is examined. The content of this section was presented in [VI, VIII].

Let us get back to competing solutions. A useful pipelined implementation of the vertical vectorization of 1-D wavelet lifting was described in [13]. This vectorization can be naively used for 2-D transform by both vertical and horizontal filtering. It is called a naive approach from now on. In [13], R. Kutil considered two nested loops (an outer vertical and an inner horizontal loop) as a single loop over all pixels of the image. He called it the single-loop approach (meant 2-D single-loop). Specifically, he merged two vertically vectorized loops into the single one. Both of the described approaches (the naive and the single-loop one) require complicated treatment of image border areas using several different combinations of prolog and epilog phases. It makes their implementation very tedious. In all cases, no SIMD extensions were used. Both of these discussed approaches were implemented in order to compare them to the core approach. In order to avoid doubts about possible caching issues in the naive implementation, it should be noted that these are avoided using the prime stride between consequent rows of image.

To solve the complicated border treatment problem described above (see Figure 3.4), I decided to remove the difficult parts of the border area processing code. Instead, I

Intel Core2 Quad Q9000 and AMD Opteron 2380

	Intel		AMD	
algorithm	time	speedup	time	speedup
naive	21.9	1.0	47.1	1.0
single-loop	8.4	2.6	15.3	3.1
core	8.5	2.6	8.4	5.6

Intel Core2 Duo E7600 and AMD Athlon 64 X2 4000+

	Intel		AMD	
algorithm	time	speedup	time	speedup
naive	19.4	1.0	154.0	1.0
single-loop	6.2	3.1	20.4	7.5
core	6.3	3.1	12.4	12.4

Table 7.1: Performance evaluation of 1-D and 2-D pipelined approaches. The vertical vectorization was used. No SIMD vectorization was used yet. The time is given in nanoseconds per one pixel. The speedups are given against the naive vertical algorithm.

have involved the 2×2 core of 2-D lifting, which produces a quadruple of coefficients (a, h, v, and d). This approach is thus referred to as the single-loop core approach. This simplification allows the programmer to write a much simpler code. Another consequence is that the code has a reduced footprint in first level instruction cache [23] possibly allowing faster execution.

Moreover, all of the above approaches (naive, single-loop, single-loop core) are compared in Table 7.1. The naive algorithm is used as a reference one. All the measurements was performed on 58-megapixel image. For now, I postpone drawing conclusions further in this chapter. Instead, the influence of CPU caches on a 2-D transform using the core approach is examined.

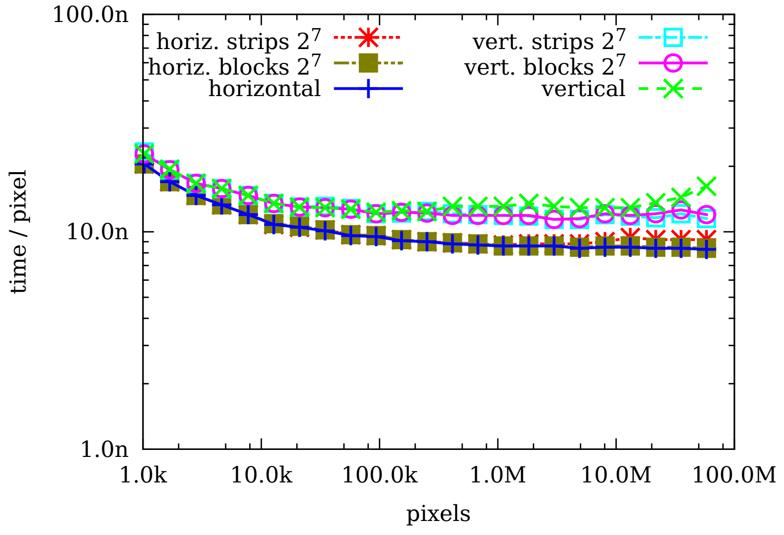
Some version of the CPU cache is present in all modern platforms. However, all the experiments presented in this thesis are closely focused on the x86 architecture. In the cache hierarchy, the individual coefficients of the transform are stored inside larger and integral blocks (called cachelines). A hardware prefetcher attempts to speculatively load these blocks in advance, before they are actually required. Due to the limited size of the cache, the least recently used blocks are evicted (discarded or stored into the memory). Moreover, due to a limited cache associativity, it is also impossible to hold in the cache the blocks corresponding to arbitrary memory location at the same time.

Even though every 4-tuple of pixels is visited only once, certain caching issues can be expected due to the mutual shift of read and write positions of the cores. This mutual shift of positions guarantees that the resulting coefficients are placed into the same (m, n)-coordinates as the corresponding input pixels. However, this is not the only possible memory layout. The read and write positions of the cores can point to the same or, on the contrary, to completely different memory locations. For the reason described above, several processing orders have been evaluated in order to find the most friendly one with respect to CPU caches. In all the cases studied below, the adjacent pixels (coefficients) in image rows were stored without gaps. Note that the coefficients are represented as 32-bit floating-point numbers.

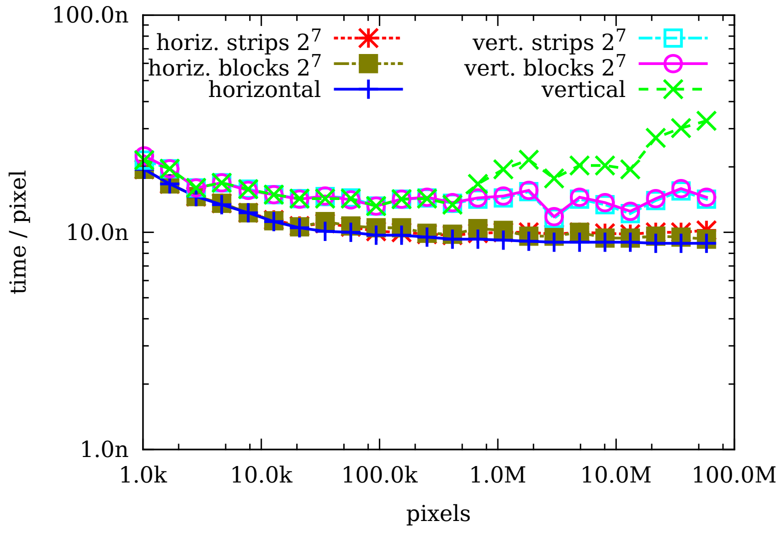
Two different fundamental processing orders are possible – the horizontal order (also known as a raster order) and the vertical order. Considering the limited sizes and possibly limited set associativities of CPU caches, the processing can be performed on the appropriate strips or blocks. This results in six combinations in total. Note that other (more complicated) processing orders also exist. The strip processing order (referred to as strip-mining or aggregation) was used earlier, e.g., in [30], [34], or [29]. All the evaluated processing orders are depicted in Figure 6.1.

A subset of the results of this experiment is shown in Figure 7.1. On the Intel as well as AMD platforms employed, all the horizontal orders perform in most cases almost equally well. On the other hand, all the vertical orders are clearly slower. This is the expected behavior since the hardware prefetcher can prefetch the coefficients only to an extent of one 4 KiB page. Note that these results are not generic and they are dependent on the CPU cache parameters. The measurements performed suggest that the horizontal order should be the best choice for platforms with unknown cache parameters. A summarization of the measurement for 58-megapixel images is shown in Table 7.2. Note also that the implementations used are slightly different from those used in the subsequent sections.

So far, only the baseline 2×2 vertically vectorized core was examined. The following text describes how several vertical as well as diagonal 2×2 cores are fused together in order to better exploit SIMD instructions. Quite a similar fusion was developed by R. Kutil in [13] employing his version of the vertical core. He used a different memory layout and especially a different variant of the single-loop approach (buffering up to 16 whole rows). A more detailed description of the individual cores follows. The best performing vectorized cores are shown in Figure 7.2. Moreover, the complete image processing using the 4×4 vertical core is illustrated in Figure 7.3.

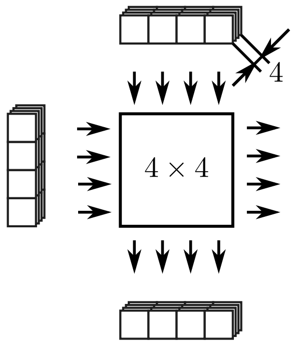


(a) Intel Core2 Quad

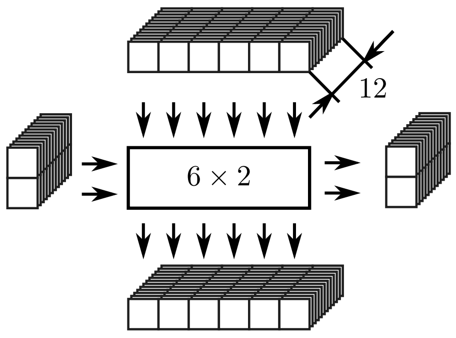


(b) AMD Opteron

Figure 7.1: Evaluation of processing orders. The 2×2 vertically vectorized core was used. Time in nanoseconds per pixel.



(a) vertical



(b) diagonal

Figure 7.2: Best performing vectorized cores. The image is processed in blocks of the indicated size. For each block, the auxiliary buffers are updated during the computation, as indicated by arrows.

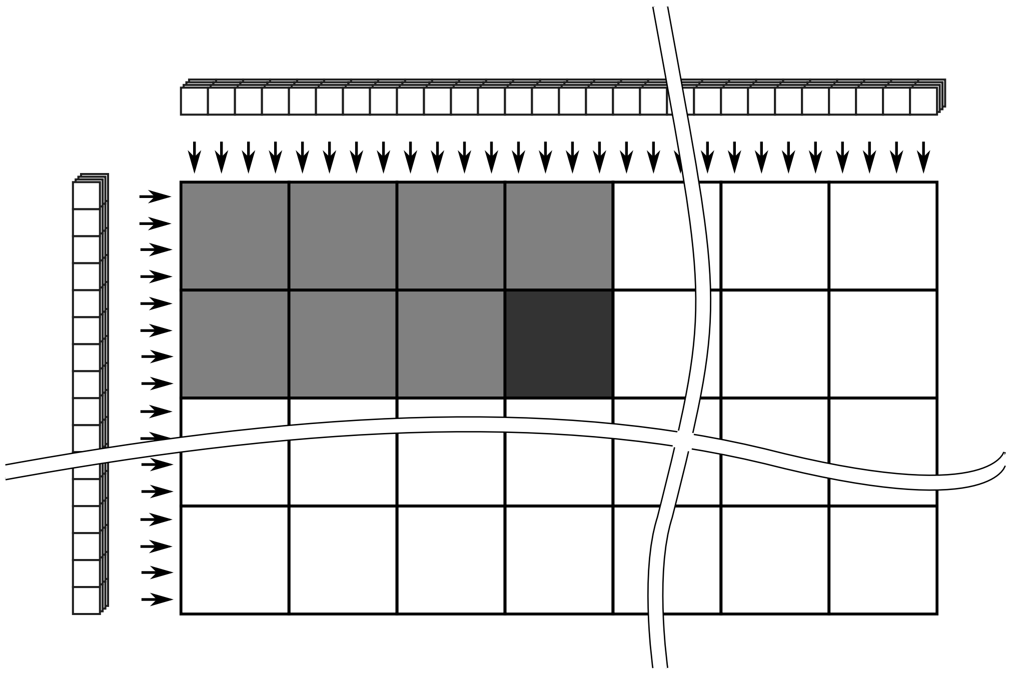


Figure 7.3: Complete image processing using the 4 × 4 core. The auxiliary buffers are shown on the left and top image edges. The light gray cores have to be evaluated before evaluating the dark gray one.

Intel Core2 Quad

order	time
full horizontal	**8.3**
horiz. strips 2^7	9.2
horiz. blocks 2^7	8.4
full vertical	16.2
vert. strips 2^7	11.5
vert. blocks 2^7	12.0

AMD Opteron

order	time
full horizontal	**8.9**
horiz. strips 2^7	10.2
horiz. blocks 2^7	9.3
full vertical	32.6
vert. strips 2^7	14.2
vert. blocks 2^7	14.5

Table 7.2: Cache influence when using different 2-D processing orders. The vertical 2^2 core was used. The time is given in nanoseconds.

2×2 **vertical core** This core is not actually SIMD-vectorized. Two adjacent vertical and two subsequent adjacent horizontal iterations of the 1-D vertical vectorization were combined into one compact 2-D core. Additionally, a simple optimization was performed. The coefficient scaling from the first vertical iterations bubbled through the subsequent horizontal iterations and was merged with their scalings. The result of a core optimized in this way is exactly the same as the result of a non-optimized version employing two independent scalings. Although no SIMD instructions are used in the core, some speedup can be expected thanks to the single-loop nature as well as the hardware prefetching into CPU caches. The core formed requires one four-tuple of intermediate results per one pair of coefficients in one direction. The total number of intermediate results is 2×4 horizontally and 2×4 vertically.

4×4 **vertical core** This core consists of two parts. In the first part, two adjacent vertical core iterations are performed on four independent subsequent rows. This can be called 2×4 vertical core iterations. The 4×4 matrix of intermediate results is now transposed.[1] In the second part, two adjacent core iterations are performed on four subsequent columns. Finally, the matrix of coefficients is scaled at once, as explained in the previous core description. Note that the result need not be transposed again. A similar 4×4 core was also used in [13], where the packed words are accessed directly in the main memory. In his work, he have to read two 4×4 blocks at once and store them separately. Figure 7.4 explains how the vectorization of the 4×4 vertical core is actually implemented. The SSE registers are outlined by a dotted line. For simplicity, the buffers are omitted.

[1]using `_MM_TRANSPOSE4_PS` macro

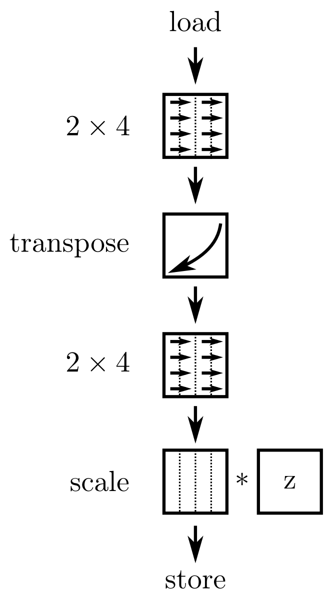


Figure 7.4: Implementation of the 4×4 vertical core. In 2×4 blocks, all operations of the vertical vectorization are performed over SSE vector registers instead of ordinary scalar ones.

8×2 and 2×8 vertical cores The cores are composed of two 8×1 SIMD-vectorized horizontal filterings followed by two 2×4 adjacent vertical core iterations. In the case of the 8×1 horizontal filtering, two (even and odd coefficients) whole packed words are now transformed using SIMD instructions. This actually evaluates four subsequent pairs of lifting coefficients at once. This 8×1 core was also used in [13], working directly over the memory. The 2×4 vertical core iterations were explained in the 4×4 vertical core description. These are applied on even and odd coefficients from 8×1 filterings separately. No transposition is performed in this case.

2×2 diagonal core This core merges two horizontal and two vertical SIMD-vectorized 1-D diagonal cores. Thus, all operations of this core are pure SIMD instructions (with the exception of loads and stores of the coefficients). The optimization of joint scaling operations as mentioned in the description of the 2×2 vertical core is also used here. This newly formed core requires three four-tuples of intermediate results per one pair of coefficients in one direction. The total number of intermediate results is, therefore, 2×12 horizontally plus 2×12 vertically.

Intel Core2 Quad

core	time	speedup
vertical 2×2	8.3	2.6
vertical 4×4	**4.4**	**5.0**
vertical 8×2	4.7	4.7
vertical 2×8	4.5	4.9
diagonal 2×2	7.2	3.0
diagonal 6×2	6.6	3.3
diagonal 2×6	6.6	3.3

AMD Opteron

core	time	speedup
vertical 2×2	8.8	5.4
vertical 4×4	**4.6**	**10.2**
vertical 8×2	5.1	9.2
vertical 2×8	5.1	9.2
diagonal 2×2	8.9	5.3
diagonal 6×2	6.7	7.0
diagonal 2×6	6.4	7.4

Table 7.3: Comparison of SIMD vectorizations of the 2-D cores. The time is given in nanoseconds per one pixel. All the measurements were performed on a 58 megapixel image.

6×2 and 2×6 diagonal cores A series of three consecutive 2×2 diagonal cores can be combined together. At the end of each vertical core iteration, three appropriate auxiliary buffers are exchanged. After three such iterations, the meanings of these buffers are again returned to the original states. Therefore, it is possible to omit this buffer exchange at all if the following diagonal cores are appropriately modified. Note that the three buffers represent the left, center and right input coefficients of the elementary lifting operations.

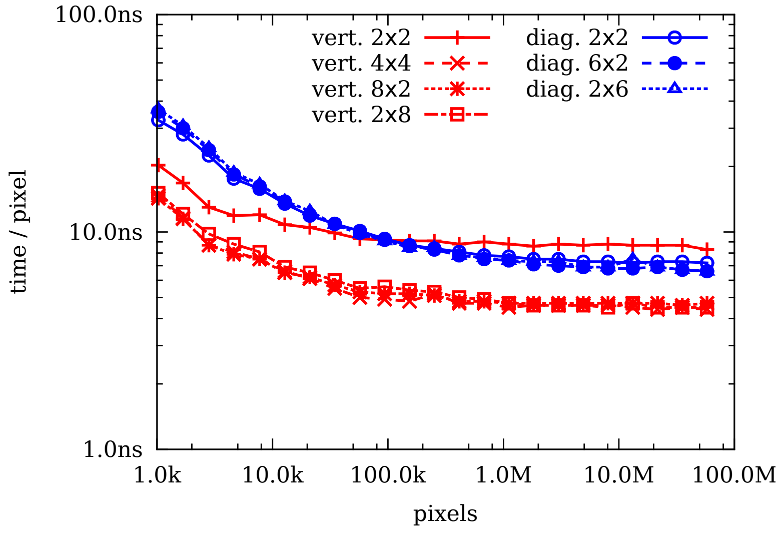
The performance of the above described cores was evaluated. In all cases, a raster scan pattern was used. The results are summarized in Figure 7.5 as well as in Table 7.3. Clearly, the 4^2 vertically vectorized core outperform all others.

So far, only the regular image are was discussed. The following text illustrates how the core approach can be employed with the widely used symmetric border extension.

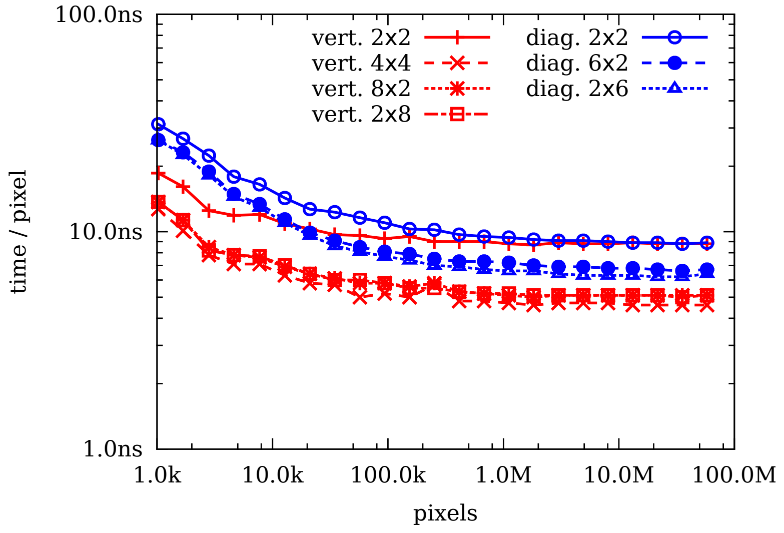
The symmetric border extension method assumes that the input image can be recovered outside its original area by symmetric replication of boundary values. Still, special prolog or epilog parts are not needed.

The best performing 4×4 core using the vertical vectorization was chosen for this purpose. Initially, this core was split into three consecutive parts. The first part loads data from the memory and places them into auxiliary variables.[2] The subsequent second part performs the actual calculation. Finally, the last part stores the results back in the memory. The programmer should be able to fully reuse the already written code. No

[2]the `__m128` data type



(a) Intel Core2 Quad



(b) AMD Opteron

Figure 7.5: Performance comparison of SIMD-vectorized cores.

special prolog or epilog parts are required here. The only change here is a simple memory addressing treatment in the first and the last parts. In the first one, the coefficient addresses pointing outside of an image region are mirrored back inside it. In a similar way, the memory accesses outside of the image in the last part are completely discarded.

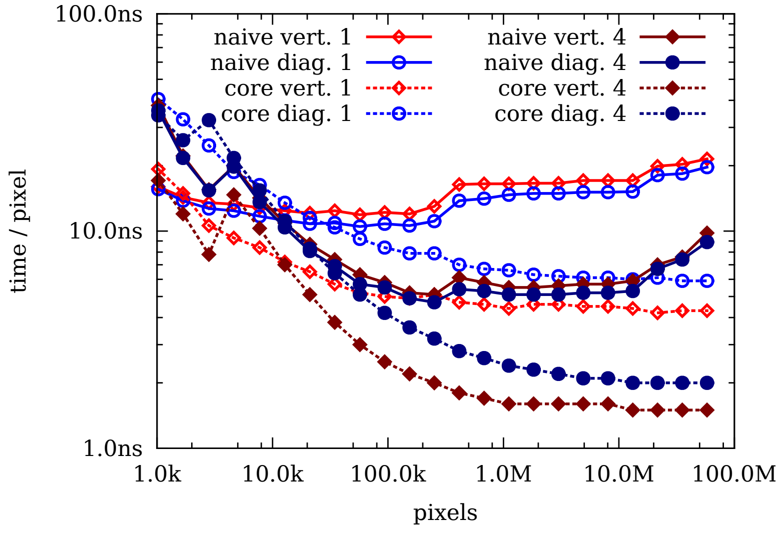
Until now, the cores exploited the advantages of the cache and SIMD extensions. The following discussion shows the effect of coarse-grained parallelization of the above discussed approaches. With the optimizations proposed above, the performance scale almost linearly with the number of threads.

The naive approach that uses the horizontal and vertical 1-D transform was parallelized using multiple threads. The same was done with vectorized core single-loop approaches (4×4 vertical and 6×2 diagonal, both with merged scaling). In the latter case, the image was split into several rectangular regions assigned to different threads. In the first case, this was done twice – for the horizontal and for the vertical filtering. Both multi-threaded implementation were written using the diagonal as well as the vertical vectorizations, resulting in four implementations in total. The performance comparison is shown in Figure 7.6. Both axes are shown in logarithmic scale. It can be seen that the naive approach, even when parallelized, is always slower comparing to just the single-threaded core approach utilizing the vertical vectorization.

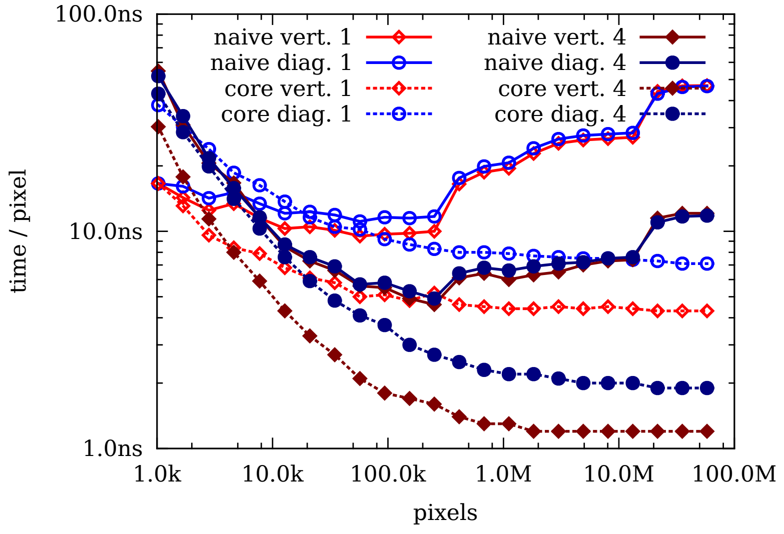
In this case, the parallelization of the single-loop core approach is not as straightforward as the parallelization of the naive approach. In order to produce correct results, each thread must process a segment (several rows) of an input image before its assigned area. In this segment, no coefficients are written to the output. Therefore, this phase can be seen as a prolog. Without the prolog, the threads would produce independent transforms, each with zero border extension.

Finally, a summarized comparison of parallelizations is shown in Table 7.4. The measurements were performed on a 58-megapixel image. The single-threaded algorithm is used as a reference one. Using the core approach, both the vectorizations scale almost linearly with the number of threads.

In summary, the core approach is very suitable for parallelization and vectorization. In contrast to [13], the size of auxiliary memory buffers does not grow with increasing core size (for vertical vectorization, 4 rows for 2×2 as well as 4×4 core). Further, as opposed to [13], the presented method can handle arbitrary memory layouts and process the data in-place as well as out-of-place; also, in case of the symmetric extension, the cores have radically simplified the border treatment and the overhead is now completely hidden in memory latencies. Finally, the core approach can be easily vectorized as well as parallelized as opposed to [13], where parallelization using threads is not considered at all.



(a) Intel



(b) AMD

Figure 7.6: Comparison of the parallelized approaches. The naive and the core approaches are compared.

Intel Core2 Quad

threads	1		2		4	
algorithm	time	speedup	time	speedup	time	speedup
naive vertical	21.5	1.0	15.8	1.4	9.8	2.2
naive diagonal	19.7	1.1	14.4	1.5	8.9	2.5
core vertical	**4.3**	**5.1**	**2.3**	**9.5**	**1.5**	**14.6**
core diagonal	5.9	3.7	3.1	7.1	2.0	11.0

AMD Opteron

threads	1		2		4	
algorithm	time	speedup	time	speedup	time	speedup
naive vertical	46.9	1.0	24.0	2.0	12.1	3.9
naive diagonal	46.6	1.0	23.6	2.0	11.8	4.0
core vertical	**4.3**	**11.0**	**2.3**	**20.5**	**1.2**	**39.3**
core diagonal	7.1	6.6	3.7	12.7	1.9	24.8

Table 7.4: Performance evaluation using threads. The time is given in nanoseconds per pixel. The speedups are shown compared to the non-parallelized naive vertical algorithm.

The experiments above still discuss only the single transform scale. In order to also confirm the performance in a multi-scale scenario, the presented single-loop cores have been incorporated into the JPEG 2000 encoding chain.

7.2 JPEG 2000

JPEG 2000 is an image coding system based on a wavelet compression technique. See the excellent book from D. Taubman in [70] or the brief summary in [71]. The format has wide application, especially with the professional use cases. For example, Digital Cinema Initiatives established uniform specifications for digital cinemas in which JPEG 2000 is the only accepted compression format. Unfortunately, there several major issues exist with the effective 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 (e.g. 8 scales). For high resolution data decomposed into several scales by a typical separable transform, immensely many CPU cache misses occur. These cache misses significantly slows the overall calculation. Furthermore, by following the typical data processing, the fundamental coding units of the JPEG 2000 format (referred to as the codeblocks) 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. As a consequence, the entropy coder [72] (EBCOT) needs to once again return to the data already touched. Finally, current implementations are built over 1-D transform which is unable to fully exploit a potential of modern CPUs (SIMD instruction set, a limited set-associativity cache).

Efficient realization of the JPEG 2000 transform was outlined by D. Taubman in [73]. However, the author 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 $(4 + I)M$ samples. As the transform coefficients have to be arranged into codeblocks, the total memory requirements for the JPEG 2000 codec are $(4 + I + 3 \times 2^{c_n})M$ samples, where 2^{c_n} is the codeblock height. The initial 4 term corresponds to 2 lines per one decomposition scale. This imposes that his implementation generates all codeblocks at the same time, not one after another. Here I would like to make a short comment. According to the description in [73], their implementation does not process the data in a single loop. However, I assumed at the time that their implementation would do so. This strategy is still fundamentally different from the architecture described in this section which generates individual blocks sequentially while all the time reusing the same memory area for output coefficients. Regarding the input processing, the results of comparison these two strategies (line-based and block-based) in the previous section (the scan orders) are interesting. Both were almost equally fast. However, the line-based processing does not fit the JPEG 2000 codeblocks, does not allow for the parallel codeblock processing or for the reuse of the memory for h, v, and d subbands. The motivation behind my work is to overcome these issues.

The cores in the previous section consume a fragment of the input signal and immediately produces a four-tuple of coefficients. The produced coefficients have a lag of $F = 4$ samples in the vertical as well as the horizontal direction with respect to the input coordinate system. In the JPEG 2000 coordinate system, such a core consumes the fragment of the input starting on odd (m, n) coordinates. Unfortunately, the original core does not fit the system of codeblocks in JPEG 2000. For explanation, every codeblock starts on even (m, n) coordinates (which corresponds to the a coefficient). On the other hand, the core with the lag of $F = 4$ samples produces a fragment of coefficients starting

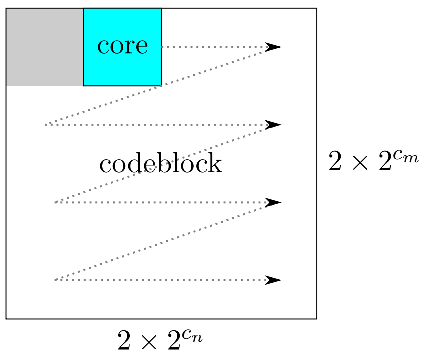


Figure 7.7: Codeblock scan order. Already processed area in gray. Active core is highlighted.

on odd (m, n) coordinates (which corresponds to the d coefficient). In order to solve this incompatibility, the original core has been modified in a way that the output coefficients are produced with a lag of $F = 3$ samples. Note that any shorter lag is not possible due to the nature of CDF 9/7 lifting scheme.

Consider the 4×4 and 2×2 vertically vectorized cores as in the previous section. As a next step, the processing of the codeblocks was encapsulated into monolithic units. These units are evaluated in horizontal "strips" due to the assumed line-oriented processing order. Inside the codeblock unit, the 2×2 core can be used. See Figure 7.7. Moreover, the unit requires access to two auxiliary buffers (one for each direction). The size of the buffer can be expressed as $2^{c_m} \times \kappa$ (for the horizontal buffer) and $2^{c_n} \times \kappa$ (for the vertical buffer), where $\kappa = 4$. As the strip-based processing with a granularity of the codeblock size is used, the vertical buffer is passed straight to the subsequent codeblock processing unit. The horizontal buffer will be used by a strip of codeblocks lying below. At the beginning of the strip, the vertical buffer contains arbitrary values. The first codeblock unit initializes this buffer and passes it to the subsequent unit in horizontal direction. The transform of this subsequent unit is not started earlier than the EBCOT [72] on the current unit has been finished. This allows for reusing the memory for h, v, and d subbands as outlined in Figure 7.8.

The above-described procedure is in effect friendly to the cache hierarchy. As shown, the processing engine uses several memory regions for a different purpose. (1) The resulting codeblock subbands occupy several KiB of memory likely settled in the top-level cache. (2) The vertical buffer occupies several hundreds of bytes. (3) The fragments of horizontal buffers occupy the same size as the total size of the vertical buffer. However, they are used only for a short time and then can be evicted from all levels of the cache

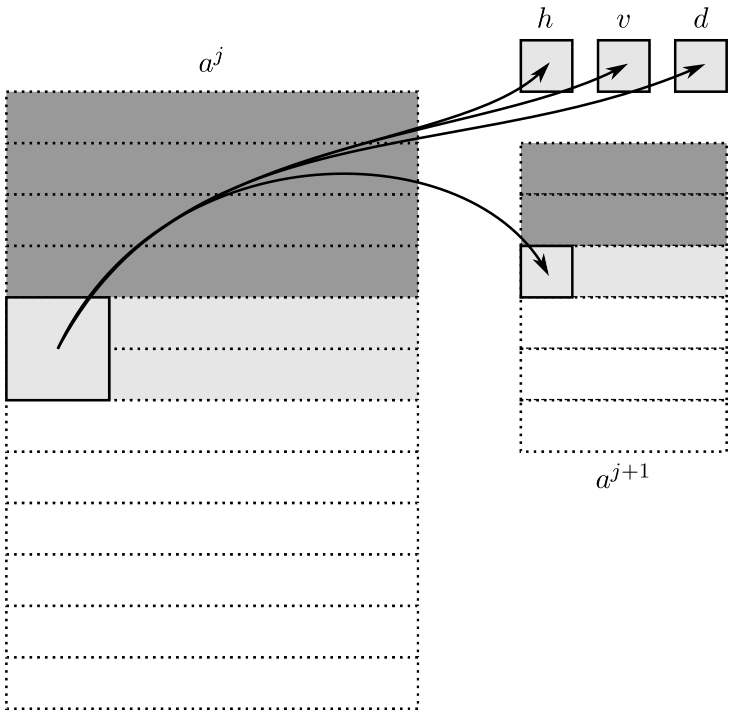


Figure 7.8: Consecutive subbands in virtual memory. Strips are separated with dotted lines. The light gray region indicates the strips right now settled in the physical memory. The dark gray region may already be unmapped, Conversely, the white region memory does not have to be populated yet. The memory for h, v, and d subbands are reused by consecutive codeblocks

hierarchy. (4) The input strip can be simply streamed into the same memory region which may be in part mirrored in the cache (say, 2 MiB for 4K resolution, 2^6 codeblocks and 32-bit samples). (5) The temporary a subbands 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 described above can be efficiently parallelized. I 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 codeblocks. Each thread holds its private copy of the vertical buffer and the memory region for the resulting subbands (h, v, d). 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 the vertical 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 horizontal buffer is not touched here and no interaction between threads is required. After the prolog, the processing continues in the usual way. The disjoint fragments of the horizontal buffer are accessed by all the threads. In the test implementation, the wavelet decomposition as well as Tier-1 encoding was parallelized. On parallel architectures, it is also possible to encode every single codeblock of the strip in parallel. However, the parallelization of the implementation is constrained by the number of computing units. Note that more sophisticated implementations could parallelize almost entire compression chain.

In the following text, the performance of the test implementation and compare it to the competitive solutions was evaluated. Physical memory demands are discussed first. The input image is consumed gradually using strips with a height of 2×2^{c_m} lines. No more input data are required to be placed in the physical memory at the same time. For the output subbands, memory for only $4 \times 2^{c_m + c_n}$ coefficients is allocated (considering all four subbands). This memory is reused by all codeblocks in the transform (or a processing thread). Additionally, two auxiliary buffers of size $M_j \times \kappa$ and $N_j \times \kappa$ coefficients have to be allocated for each decomposition level j. It should be noted 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 a 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 a window is roughly $3 \times 2^{c_n} \times M_{j+1}$. It must be noted that 3 instead of 2 codeblock strips are needed 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 codeblocks of the subsequent scales do not exactly fit each other. Taken together, the presented solution requires

$$(I + 3 \times 2^{c_n})M \tag{7.1}$$

samples populated into the physical memory. Please note that these memory requirements are the same as outlined in [70].

The presented solution was compared to C/C++ libraries listed on the official JPEG committee web pages – OpenJPEG, Kakadu, FFmpeg, and JasPer. The OpenJPEG, FFmpeg, and JasPer libraries are distributed under the terms of open-source licences. Thus, these could be analysed through their source code in detail. It should be noted that OpenJPEG and JasPer are approved as reference JPEG 2000 implementations. The Kakadu implementation is a heavily optimized closed-source library. The implementation

library	algorithm
core approach	strip-based, scales interleaved
OpenJPEG	naive
JasPer	naive
FFmpeg	naive
Kakadu	line-based, scales interleaved

Table 7.5: Software overview in terms of the transform algorithm.

presented in this thesis is based on 32-bit floating-point format.

The overview of the above described libraries is shown in Table 7.5. The naive approach (see Algorithm 1) refers to processing the entire image at once while keeping the horizontal and vertical passes as well as the transform levels (scales) separated. Furthermore, inside the horizontal and vertical passes, the lifting steps are processed sequentially (the horizontal vectorization). 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 (the line-based in Algorithm 2 and the strip-based in Algorithm 3) use sophisticated technique where the processing of consecutive scales is interleaved. Moreover, in case of the presented 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 [73] about the processing of the horizontal and vertical lifting steps.

```
foreach scale do
    foreach dimension do                                   /* M, N axis */
        foreach lifting do
            foreach sample do
                process 1-D step;
            end
        end
    end
end
```

Algorithm 1: The naive approach pseudocode.

```
foreach two available lines do
    process 1-D vertical lifting steps;
    process 1-D horizontal lifting steps;
end
go to next scale;
```

Algorithm 2: Line-based processing.

```
foreach available strip do
    foreach codeblock do
        foreach core do
            process 2-D core;
        end
    end
end
go to next scale;
```

Algorithm 3: Strip-based processing.

Again, the measurements presented in this section were obtained on Intel Core2 Quad Q9000 running at 2.0 GHz. All the algorithms below were implemented in the C language, several of them using the SIMD compiler intrinsics. The average time required to produce a single transform coefficient was measured for various range of image resolutions.

Considering the test implementation, the processing engine was vectorized using widely used SIMD extensions. Since the implementation is built over the 32-bit floating point numbers, primarily the SSE (Streaming SIMD Extensions) instruction set is used. 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, the 4×4 core was obtained. Inside this core, all of the filtering steps are performed using 4-way parallelism through the 128-bit SSE register. This case was also studied the previous section. 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 subbands. Such a operation does not fit the SSE instruction set and consequently degrades the performance. For this reason, an attempt was made to construct 8×8 "supercore" consisting of four adjacent 4×4 cores. This 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 codeblock size required by the standard. In other words, a theoretical 16×16 core would produce the 8×8 fragments of subbands which might not fit the 4×4 codeblocks. The OpenMP interface was exploited to parallelize the code; however, any other implementation is possible.

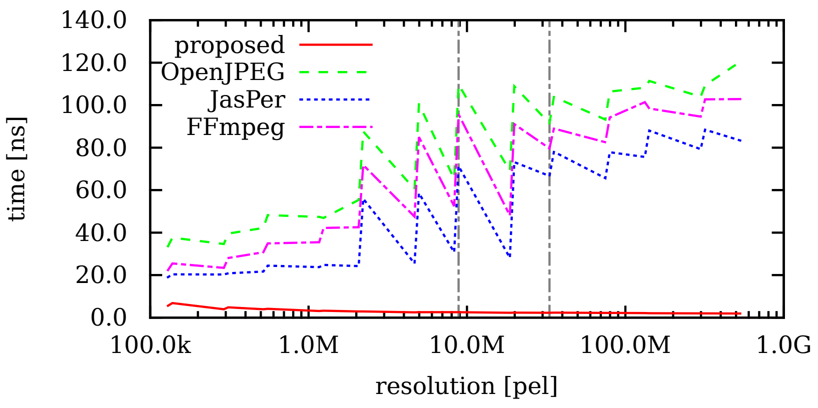


Figure 7.9: Performance comparison of JPEG 2000 libraries. Time per pixel for the transform stage only. DCI 4K and 8K UHD resolutions indicated by the vertical lines.

The transform stage was extracted from the libraries described above in order to get accurate results. This stage was then subjected to measurement. The results are shown in Figure 7.9. As observed also in [13], the single-loop processing has stable performance regardless of the input resolution. The proposed implementation was measured using four threads and SSE extensions. However, the SSE or AVX extensions boost the performance by at most 5 %.

The ability of parallel processing has been evaluated. The original single-loop approach from the previous section scaled almost linearly with the number of threads. The JPEG 2000 processing has coarser granularity (codeblocks instead of cores) and it is performed in multiple scales. Higher scales of the decomposition have, unfortunately, significantly lower resolutions in comparison to the input tile. For this reason, the parallelization is not as efficient as in case of the original approach. The results of the measurement are shown in Table 7.6. 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 so close to the linear relationship.

Since only DWT stage of the JPEG 2000 codec was done, the code was implanted into OpenJPEG library replacing the original implementation. Note that no part of OpenJPEG is optimized for the performance. Because the implementation is built using the floating-point format and OpenJPEG uses the fixed-point format, the samples have to be converted 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 codeblocks. The rest of the code remains unmodified and runs in sequence. The performance mea-

	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 7.6: Parallel JPEG 2000 processing, streaming input. 4096×2160 input, 64×64 codeblocks. The tile was decomposed into a single ($J = 1$) and multiple ($J = 8$) scales.

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 7.7: Integration of the core approach into OpenJPEG library. 4K resolution shown.

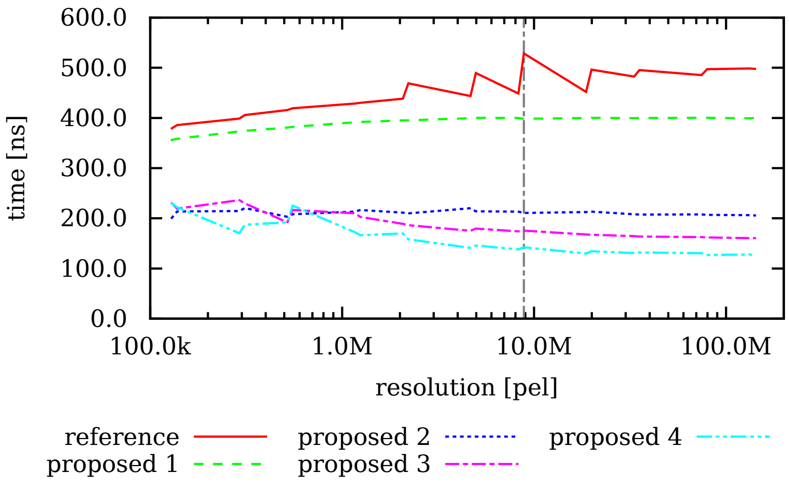


Figure 7.10: Evaluation in OpenJPEG chain. Time/pel for the complete compression chain. The numbers in the legend refers to the number of threads. 4K resolution indicated.

surements for this case are plotted in Figure 7.10. Eight decomposition levels, up to four threads, and SSE extensions were used. As expected, the single-loop processing has stable performance regardless of the input resolution. The measurement is summarized in Table 7.7. It can be seen that the complete compression chain scales better than the standalone transform stage. As in the case of the previous section, the conclusions from this experiment are postponed to be presented together at the end of this chapter.

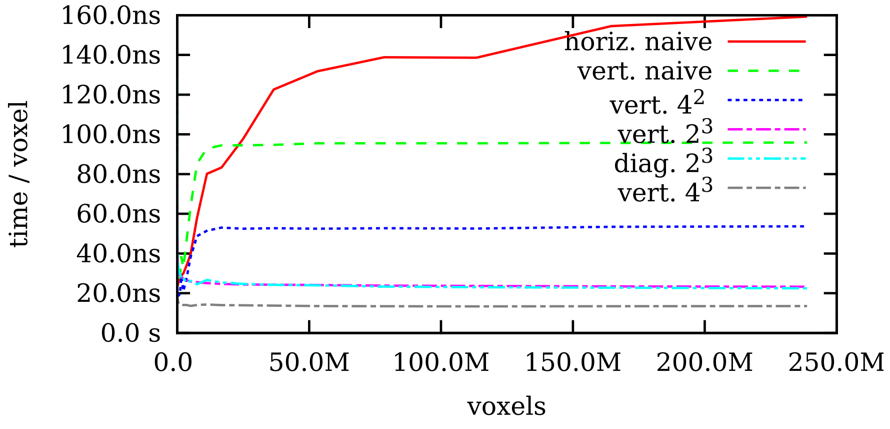
7.3 3-D Decomposition

In this section, the performance of the multi-dimensional, particularly 3-D, transform core was evaluated. Based on the cores in Section 7.1, two baseline implementations transforming the entire volume in the single loop were created. These implementations employ 2^3 cores – first built over the vertical and the second over the diagonal vectorization. Both of them process the data out of a place. The implementation with the diagonal core is inherently accelerated by SIMD instructions. The vertical implementation does not allow SIMD-optimizations at this stage.

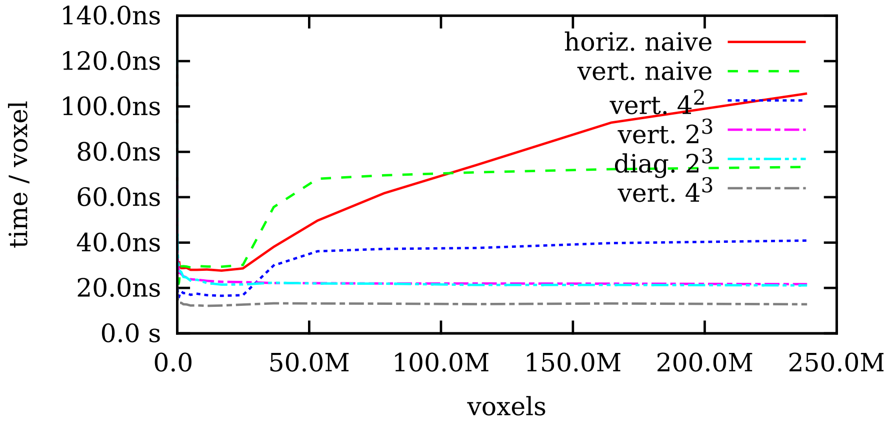
In more detail, both 2^3 cores are composed from three parts. The first part loads the input data from a source memory area. The inner part performs a fragment of the transform computation. Finally, the last part stores the resulting coefficients into a destination area. The first and last parts treat data borders using the symmetric extensions.

Access to three auxiliary buffers is required during the inner part of the computation. This most important part consists of three blocks performing calculations corresponding to the three dimensions. Each block updates the necessary intermediate results in the corresponding auxiliary buffer. This is followed by scaling of the output coefficients. In the most generic variant, each 2-D auxiliary buffer has the same size like the corresponding volume side. The depth of each auxiliary buffer is $\kappa = 4$ coefficients for the vertical vectorization or $\kappa = 12$ coefficients for the diagonal one. This memory consumption was reduced using a appropriate (raster scan) processing order as discussed earlier.

The performance of the two 3-D core approaches has been compared to the methods discussed in Chapter Computation Schedules and Section Image Processing of this chapter. Namely, the 2-D 4×4 SIMD-vectorized core (4^2 slices) and the naive methods (series of single-loop 1-D transforms). The result can be seen in Figure 7.11. Apparently, all of the 3-D approaches (even unvectorized) approximately above 1 megavoxel outperform all the previous methods.



(a) Intel Core2



(b) AMD Opteron

Figure 7.11: Summarizing performance comparison of all 3-D approaches.

Within the context of the vertical vectorization, utilization of SIMD instruction set is straightforward. In the first two dimensions, 2×2 small 2^3 cores are merged together in a manner similar to the one used in Section 7.1. In the third dimension, there is no reason to increase the size of the core as SIMD can be used directly. In all three dimensions, the basic building block of the transform is 2×4 core. In this 2×4 core, four steps of the vertical vectorization are computed in parallel using SIMD instructions. The scaling of coefficients is performed together as the last step of the calculation. As a result, $4 \times 4 \times 2$ SIMD-vectorized core was formed.

Although there is no reasonable justification, it can be tempting to build a compact 4^3 core as an analogy to the well-performing 4^2 counterpart. The core is internally composed

method	Intel Core2		AMD Opteron	
	time	speedup	time	speedup
naive horiz.	159.8	1.0	105.7	1.0
naive vert.	100.1	1.6	73.5	1.4
vert. slice 4^2	53.8	2.9	41.0	2.5
vertical 2^3	23.3	6.8	21.7	4.7
diagonal 2^3	22.8	6.9	21.1	4.9
vertical 4^3	**13.5**	**11.7**	**12.9**	**8.0**

Table 7.8: Performance evaluation for large data in 3-D. Best results are in bold.

of two dependent $4 \times 4 \times 2$ sub-cores. However, such a connection may be slightly advantageous due to prefetching of intermediate results. Moreover, the implementation is very regular.

The SIMD-optimized 3-D cores exhibit the best results. The 4^3 core slightly outperforms the baseline $4 \times 4 \times 2$ one. Above initial transients, all the single-loop approaches confirm the linear asymptotic complexity of the discrete wavelet transform. Table 7.8 summarizes the performances and speedups for a volume of 238 megavoxels. The testing platforms are the same as in the previous sections. The speedups are given comparing to the separable method using the horizontal vectorization and the prime stride. The achieved processing time of 13 nanoseconds per sample is roughly equivalent to process 37 frames per second with Full HD resolution (1920×1080 per frame). For $4 \times 4 \times 2$ core and any infinite video sequence, only two frames (the currently coded and the immediately preceding) have to be held in memory. The summarizing comparison of all of the significant approaches is shown in Figure 7.11.

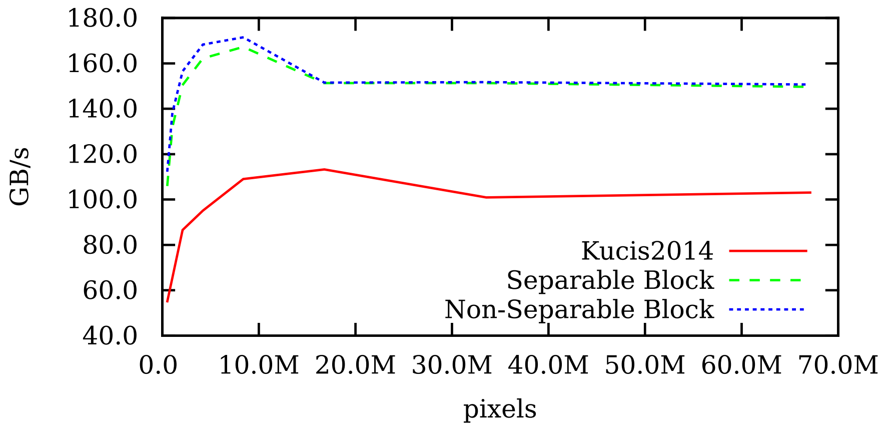
7.4 Parallel Processing on GPU

As described in Section Parallel 2-D Cores, two parallel lifting scheme schedules for GPGPUs were designed and implemented. These schedules are, in fact, based on the separable and non-separable parallel cores presented in the previous chapter. The implementation is based on the OpenCL framework. All of the algorithms are evaluated on CDF 9/7 transform using the most recent GPUs of two biggest vendors, namely, AMD R9 290X and NVIDIA TitanX graphics cards. This experiment was also presented in [X].

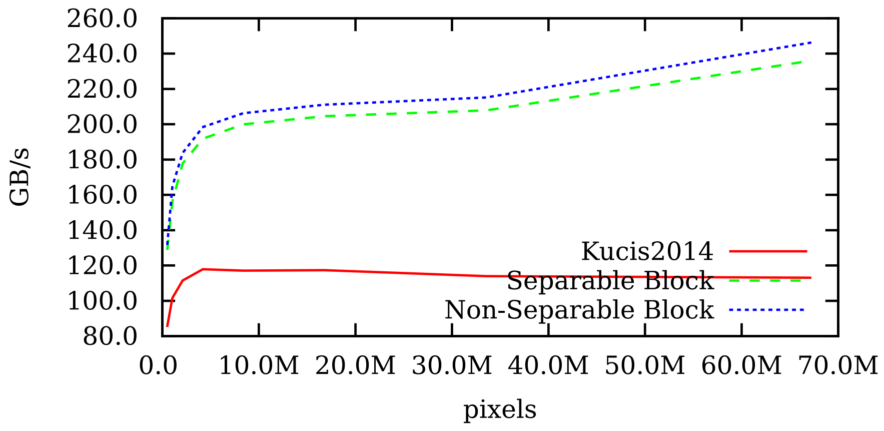
The 4-stage separable approach is discussed first. Except the sliding window, the separable block-based approach uses the same scheme like Laan's [60] method. The threads in each work-group are responsible for processing of 2×2 input coefficients. At the beginning of the computation, the thread (that is in valid image range in the vertical axis) loads their coefficients from the global memory and stores them into separate shared memory locations. The threads on image borders have to correct their pointers to input coefficients. Consequently, the input coefficients on the borders of image are always valid. After loading of the image block, the computation of the wavelet transform using the shared memory can start. In the first step of the horizontal pass, each of the threads computes the v coefficient using two a coefficients of the thread itself and the thread on the right. Additionally, the d coefficient is computed using h coefficients of the current thread and the thread on the right. In the second step, the computation of a and h coefficients is performed in the same way as the computation of the v and d coefficients. After that, these two steps are repeated with a substitution of α and β with γ and δ coefficients. The vertical steps are performed in the same way as the horizontal steps except for a rotation of the scheme by 90 degrees. Unlike the horizontal pass, in the vertical pass a synchronization of threads using a memory barrier is required between each of the steps. Horizontal steps are synchronization-free thanks to the atomicity of hardware block instructions.

The 2-stage non-separable approach was described in the previous chapter. As compared with [16], the total number of arithmetic operations has been reduced form 24 to 20 in relation to CDF 5/3 wavelet. The calculation of CDF 9/7 transform comprises two connected cores (the first with α, β, the second with γ, δ) between them another barrier is placed. In total, the calculation contains 3 memory barriers.

The achieved memory throughput performance is shown in Figure 7.12. As it can be seen, both of the core methods overcome the reference state-of-the-art method. Moreover, the proposed non-separable core performs slightly better compared to the separable one. As it can be expected, this behavior corresponds to the reduced number of the memory barriers.



(a) AMD R9 290X



(b) NVIDIA TitanX

Figure 7.12: Throughput performance of parallel methods. *Kucis2014* denotes the reference method presented in [IX].

7.5 FPGA Devices

In the last substantial experiment, the hardware implementation is evaluated. The implementation is focused on JPEG 2000 system. Particularly, the lossless CDF 5/3 transform was implemented in FPGA.

The transform defined in JPEG 2000 maps the input pairs $(d^{(0)}_m, a^{(0)}_m)$ onto the output ones $(a^{(1)}_{m-1}, d^{(1)}_m)$ as

$$d^{(1)}_m = d^{(0)}_m - \left\lfloor \frac{a^{(0)}_{m-1} + a^{(0)}_m}{2} \right\rfloor , \tag{7.2}$$

$$a^{(1)}_{m-1} = a^{(0)}_{m-1} + \left\lfloor \frac{d^{(1)}_{m-1} + d^{(1)}_m + 2}{4} \right\rfloor . \tag{7.3}$$

Note, please, the rounding term +2 in (7.3). These cores proposed in [III] and presented in this section can be used as standalone computing units or incorporated into the existing block-based or line-based architectures. The general description of the cores is provided in Chapter Lifting Core. Namely, the 2-stage core (baseline) and the 1-stage (reduced latency) core both with $F = 1$ were used as a basis.

As in [8], CDF 5/3 lifting scheme is defined using constants $\alpha = -1/2$ and $\beta = 1/4$. The resulting core has 4 independent stages suitable for hardware pipelining. These stages are shown in more detail in Figure 7.13. The first two of them correspond to the horizontal filtering and do not need to access the coefficients in the auxiliary buffer. The latter two correspond to the vertical filtering and need to exchange the data through the on-chip auxiliary buffer. The core consists of 16 operations in total. The length of the longest data path in both stages is 2 operations. The individual stages correspond to predict T^M_α, T^N_α and update S^M_β, S^N_β steps in the horizontal and vertical direction, respectively. The core can be then described in matrix notation as

$$\boldsymbol{y} = S^N_\beta \, T^N_\alpha \, S^M_\beta \, T^M_\alpha \, \boldsymbol{x}. \tag{7.4}$$

As a second step, I have tried to reduce the depth of the calculation per the output quadruple using the reduced latency 1-D core (see Section 5.1). Inside the core, the newly obtained factors are powers of two as they are composed of the original factors. Finally, the core with two stages reveals – one stage for horizontal and one for vertical filtering. The stages are shown in Figure 7.14. New intermediate results were also introduced. These new results replaced part of the original results in the auxiliary buffer. Therefore,

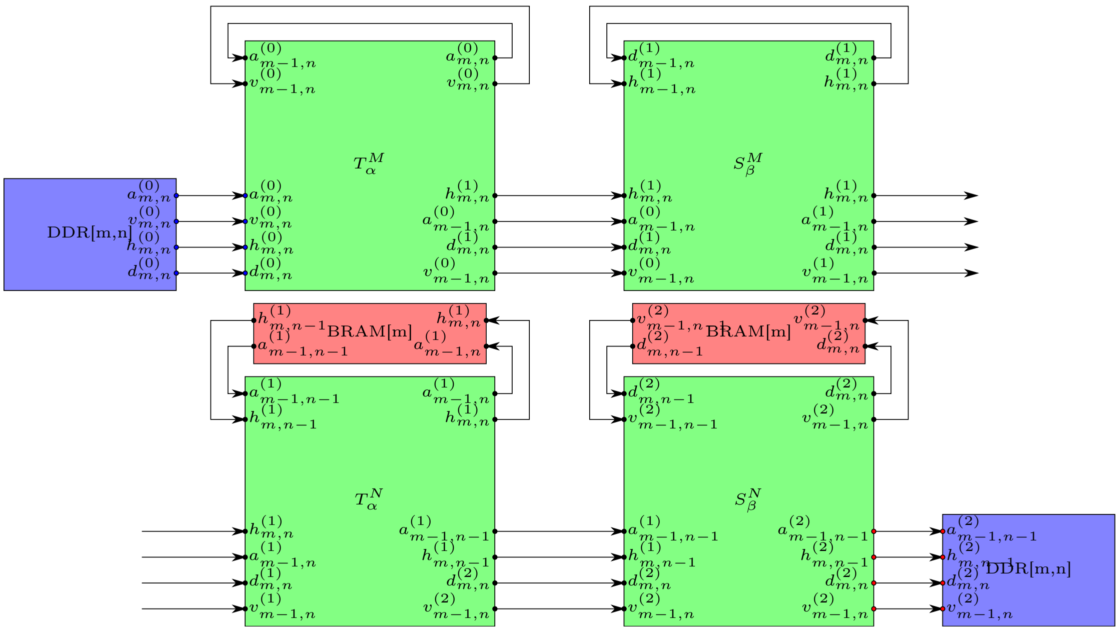


Figure 7.13: The four-stage separable core with 16 additions. The arrows pointing to the right are linked to the arrows coming in from the left.

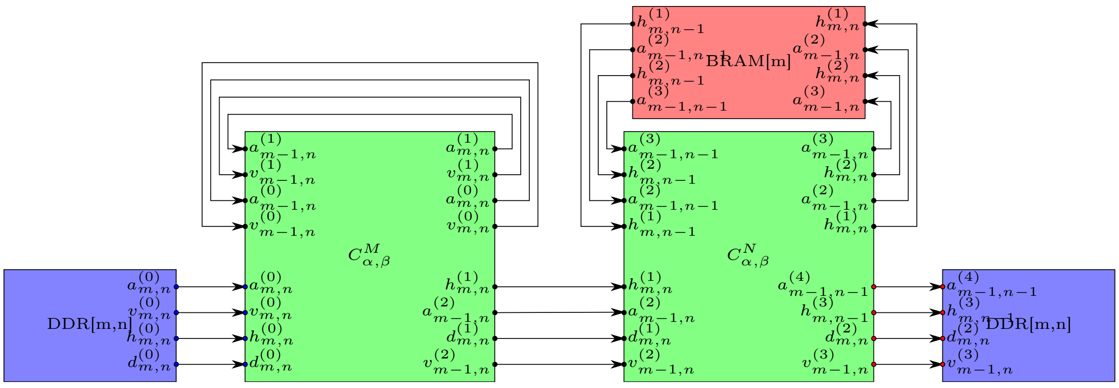


Figure 7.14: The two-stage separable core with 28 additions.

the memory consumption of this buffer remains unchanged. The newly formed core requires 28 MACs per output quadruple. The length of the critical path in each stage is 2 operations. Thus, the total depth of the entire calculation is smaller (as the number of subsequent stages was halved) than in the original case. In the matrix notation, the discussed core can be described by

$$\boldsymbol{y} = C^N_{\alpha,\beta}\, C^M_{\alpha,\beta}\, \boldsymbol{x}. \tag{7.5}$$

Here, the implementation is described in full detail including correct JPEG 2000 rounding. The simplest four-stage separable core implements the equations (7.2) and (7.3). These equations can be rewritten in order to utilize the input carry bit of the adders.

$$d^{(1)}_m = d^{(0)}_m - \left\lfloor \frac{a^{(0)}_{m-1} + a^{(0)}_m}{2} \right\rfloor \tag{7.6}$$

$$a^{(1)}_{m-1} = \left\lfloor \frac{4a^{(0)}_{m-1} + d^{(1)}_{m-1} + d^{(1)}_m + 1 + 1}{4} \right\rfloor \tag{7.7}$$

The transform was implemented using 16-bit signed integers. Such a width is sufficient for few levels of DWT as required by JPEG 2000 standard. The stages reflects the horizontal and vertical predicts and updates, which implies 4 stages. The auxiliary coefficient between the horizontal stages are passed in registers. On the contrary, the vertical intermediate results are exchanged through the Block RAM (BRAM). As all the factors in CDF 5/3 are powers of two, the complete transform consists of 16 additions per output quadruple only.

The implementation of the two-stage core is more sophisticated. At the beginning, 1-D transform is discussed. Considering the core approach, the biggest problem with the lossless DWT as required by JPEG 2000 is the correct rounding of the a_{m-1} coefficient. Now, the a_{m-1} coefficient is computed in two stages. Consequently, a rounding flag (or a rounding bit) has to be distributed from the first stage of the core to the second one. As the a_{m-1} is computed in two stages, the newly formed auxiliary $a^{(1)}_{m-1}$ coefficient have to be passed between the two stages. The final coefficients are then denoted $(a^{(2)}_{m-1}, d^{(1)}_m)$.

To explain the two-stage rounding, the following identity have to be established first. Let $\mathbb{Z}$ denote the set of integers, $2\mathbb{Z}+1$ denote the set of odd integers, and $p, q \in \mathbb{Z}$. One can perform the following expansion

$$\left\lfloor \frac{p+q}{2} \right\rfloor = \lfloor p/2 \rfloor + \lfloor q/2 \rfloor + \mathcal{C}_2(p,q), \tag{7.8}$$

where $\mathcal{C}_2$ is a correction term is defined as

$$\mathcal{C}_2(p,q) = \begin{cases} 1 : p \in 2\mathbb{Z}+1 \land q \in 2\mathbb{Z}+1 \\ 0 \end{cases} . \tag{7.9}$$

Using this identify, one can rewrite the original 1-D transform as follows.

$$d_m^{(1)} = d_m^{(0)} - \left\lfloor \frac{a_{m-1}^{(0)} + a_m^{(0)}}{2} \right\rfloor \tag{7.10}$$

$$a_m^{(1)} = 4a_m^{(0)} - 2\lfloor a_m^{(0)}/2 \rfloor + d_m^{(0)} - \lfloor a_{m-1}^{(0)}/2 \rfloor \tag{7.11}$$

$$b_m = 1 - \mathcal{C}_2(a_{m-1}^{(0)}, a_m^{(0)}) \tag{7.12}$$

$$a_{m-1}^{(2)} = \left\lfloor \frac{a_{m-1}^{(1)} + d_m^{(0)} - \lfloor a_m^{(0)}/2 \rfloor + b_{m-1} + b_m}{4} \right\rfloor \tag{7.13}$$

As in the previous case, the horizontal intermediate values are passed in registers and the vertical ones in BRAM. The core requires 28 additions per output quadruple. Note that the calculation of b_m is implemented in a single NAND gate.

The wavelet engine was experimentally synthesized in a Xilinx Zynq XC7Z045 FPGA and evaluated on the Xilinx ZC706 board (with DDR3 at 1066 MHz). The engine was synthesized for several image resolutions (as seen in Table 7.9) that merely differ in the BRAM size, only to allow comparison with other papers, and also to show that the core is able to process Full HD video (1080p, 60 Hz) faster than in real-time. As it could be seen, the core is ready to process image resolution of up to 4K UHD with the outlook to even higher resolutions without need of any fundamental changes. The computational engine has standardized AXI-Stream interfaces. The input expects streamed video frames in the predefined resolution, the output stream is generating interlaced coefficients of wavelet transform that can be easily split into four separate data streams for further multi-scale

	small tile	Full HD	4K UHD
resolution	512 × 512	1920 × 1080	3840 × 2160
theoretical framerate	3698	477	120
framerate with DRAM	2670	338	84

Table 7.9: The framerates achieved in FPGA implementation. Given for various image resolutions.

decomposition. I would especially like to highlight the ability to process the video stream without the need of using external memory for intermediate results. The design includes mirroring on the image edges which is not performed by the wavelet core itself, but by the engine, which encapsulates the core. The engine itself then represents an independent block, which can be used in a more complex system or which can be easily duplicated and chained to perform more levels of wavelet transform of one image.

Both of the implemented cores are fully pipelined. First of the cores has latency of 2 clock cycles and it represents the four-stages separable wavelet transform from Figure 7.13. The second core has latency of 4 clock cycles and it represents the two-stages separable wavelet transform from Figure 7.14. The wavelet core itself requires just a very small fraction of ZC706 resources, as shown in Table 7.10 and in Table 7.11. As it could be predicted, first solution with 2-stage pipeline leads in higher LUT requirements and less demand for Flip-Flops. At the opposite the 4-stage pipeline consumes more Flip-Flops and smaller amount of LUTs. The requirements for whole wavelet engine are summarized in Table 7.12; besides the core itself, it shows resource demands for engine

	small tile	Full HD	4K UHD
FF	328	332	338
LUT	239	247	257
BRAM	1	2	4

Table 7.10: The resources consumed by the 4 clock latency core.

	small tile	Full HD	4K UHD
FF	280	282	284
LUT	418	440	426
BRAM	1	2	4

Table 7.11: The resources consumed by the 2 clock latency core.

core	FF		LUT		BRAM	
latency 4	441	(0.1 %)	**399**	(0.18 %)	6	(1.1 %)
latency 2	**391**	(< 0.1 %)	592	(0.27 %)	6	(1.1 %)

Table 7.12: Resources consumed for Full HD resolution on ZC706 board.

performing row and column edge mirroring. According to JPEG 2000 standard, the four image lines and columns have to be mirrored, this consumes the extra four BRAMs in case of Full HD resolution.

The overall performance of wavelet engine is summarized by Table 7.9. It incorporates the theoretical and real performance of the engine with relation to image resolution. Both of the wavelet cores have the same throughput – they differ just in output latency. The core is able to process 4 input samples in one clock cycle, producing 4 output samples and the important fact also is that each of the samples needs to be fetched from an external memory and stored into the external memory just once as the design is single pass streaming 2-D DWT unit. The computation has to also be performed on the mirrored image edges that are enlarged by 4 pixels in each direction. The theoretical performance was calculated for maximum speed 250 MHz with respect to edge mirroring, assuming ideal situation that input data are always available. The practical performance was measured on hardware Xilinx ZC706. There it could be observed that the RAM throughput is essential for the overall performance.

The overall comparison with the selected architectures is shown in Table 7.13. The time column is dependent on a clock. Whereas the BRAM and clocks/pel columns are platform-independent. In other words, when the engine is synthesized for some older hardware, the resource consumption and clocks/pel will be the same.

architecture	device	BRAM [bits]	clocks/pel	time [ms]
Dillen [53]	VirtexE1000-8	50K	0.50	1.20
Descampe [52]	Virtex-II XC2V6000	N/A	0.60	1.75
Seo [51]	Altera Stratix	128K	2.64	6.02
Zhang [50]	Virtex-II Pro XC2VP30	6×18K	0.50	0.97
the cores	Zynq XC7Z045	1×36K	**0.26**	**0.27**

Table 7.13: Comparison of various FPGA implementations. Tiles of size 512×512. The processing time and clocks per pixel were projected to the uniform image size. The best results are in bold.

7.6 Vectorization

Finally, it might be interesting to see that the pipelined computation is not always advantageous. For this experiment, the vector processor synthesized in FPGA has been chosen. Meanwhile, the core build above the vertical vectorization was implemented on MicroBlaze CPU. Above that, naive implementation that basically corresponds to the horizontal vectorization was implemented on the CPU as well as the vector processing unit. The platform details are explained below.

As stated in Chapter 4, the vectorizations evaluated on this FPGA-based vector processor was presented in [V]. The heterogeneous multi-core platform referred to as ASVP was presented in [66], [67], [68], and [69]. This platform employs up to several units called Basic Computing Element (BCE) which can accelerate floating-point vector operations. For organization of the platform see Figure 7.15. These elements use a combination of a simple PicoBlaze CPU (sCPU in Figure 7.15) with a configurable pipelined datapath. The computation performed by BCE can be changed through replacing the PicoBlaze firmware. Moreover, the platform contains host CPU (MicroBlaze in this case) that is executing the main program. Thus, the computation is distributed between host CPU and one or more BCE units. This change of the BCE firmware can be made from MicroBlaze CPU in runtime. The BCE contains four memory banks each of 1024 words long (one word denotes 32-bit single-precision floating-point format). Before BCE can

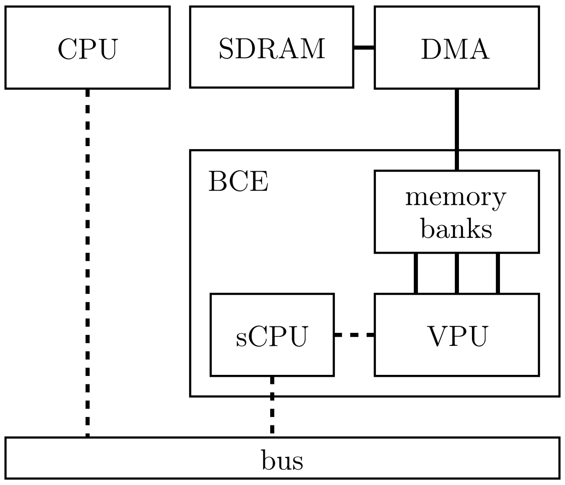


Figure 7.15: Organization of the vector-processor platform. Solid lines indicates data paths. The BCE consists of memory banks and Vector Processing Unit (VPU) and accesses RAM through DMA engine. Its function is controlled by host CPU.

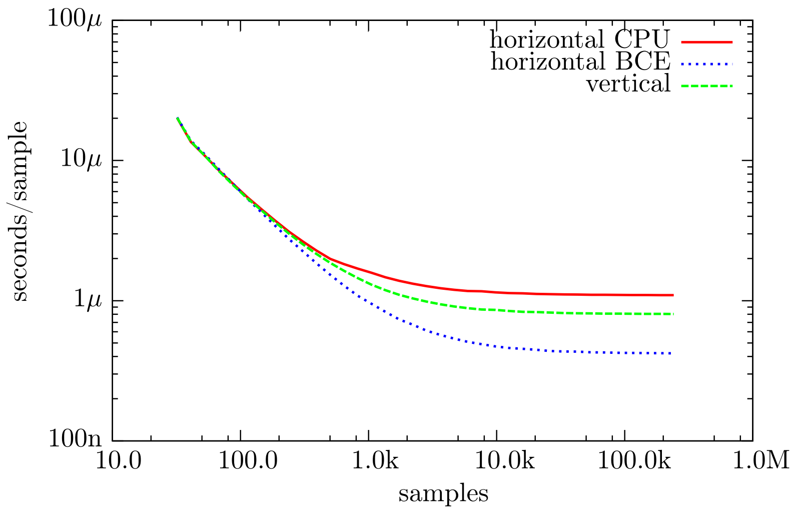


Figure 7.16: Comparison of the vectorizations on the vector processor. The horizontal vectorizations were implemented on the CPU as well as on the vector unit.

start its program, the input data must be transferred from main DDR memory into BCE's memory banks. Similarly, the output data should be transferred back when BCE computation is done. These data are transferred by DMA controller. The operations performed by BCE are element-wise move, addition, multiplication, etc. The results of the evaluation are presented in comparison to the vertical vectorization below in the text.

The comparison was performed on 1-D forward DWT using CDF 9/7 wavelet. All the implementations work over a sequence of single-precision floating point numbers. Evaluation on is summarized in Figure 7.16. The horizontal axis of this graph indicates the sequence length. The vertical axis specifies computation time per one signal sample. Both vectorizations as implemented on MicroBlaze CPU are plotted in this graph. Furthermore, another implementation of the horizontal vectorization accelerated using BCE unit is plotted here. Clearly, the horizontal vectorization is the fastest method when long SIMD operations on BCE can be used. Without appropriate vector operations, the fastest approach is the vertical vectorization. In this particular example, the speedup of the horizontal vectorization with SIMD operations over baseline horizontal approach on CPU is up to 2.6×.

Therefore, another situation occurs, if there is only GPP available. The implementations of all of those vectorizations described in the Chapter 4 were evaluated on x86 platform. As in previous experiment, this comparison was performed on 1-D forward DWT using CDF 9/7 wavelet. All the implementations work over a sequence of single-

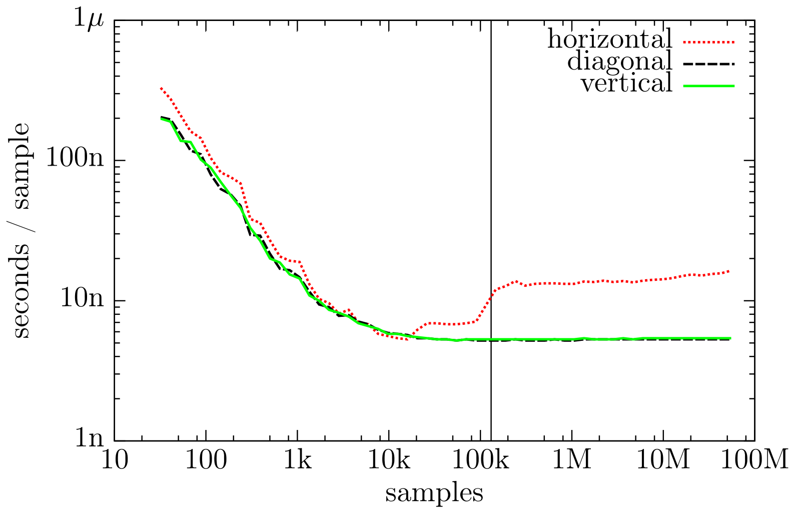


Figure 7.17: Comparison of the vectorizations on the x86 platform. The vertical line represents the size of the last-level CPU cache.

precision floating point numbers. The evaluation on x86-64 (AMD Athlon 64 X2 4000+) is shown in Figure 7.17. The horizontal vectorization fails with samples exceeding the CPU cache size due to extensive cache misses. In contrast, the vertical and diagonal vectorization show stability with increasing input length. As can be observed from these two experiments, the core approach seems to be an appropriate tool considering GPPs.

7.7 Discussion

Several experiments evaluating the performance of different methods were conducted on general-purpose processors. The 1-D transform is considered first. Several findings are evident from these experiments. The most important effect occurs as soon as the working set exceed the cache size. The discussed effect causes the single-loop methods to be faster compared with the naive horizontal vectorization. This is documented in Figure 7.17 in the previous section.

Considering the 2-D transform, the single-loop methods are faster as compared with the naive separable methods. This case is summarized in Table 7.4. The single-loop method can be built above the vertical or diagonal vectorization, or above their variants.

Furthermore, considering the 2-D transform, the core implementation of the single-loop approach was discussed in detail. Its performance is similar the the "hardwired" single-loop code as summarized in Table 7.1. However, the cores disclose several degrees of freedom. This advantage is especially useful when considering the multi-dimensional transform. Namely, a variety of processing orders can be employed during the transform. This is true even in connection with the multi-scale decomposition as shown when integrating into the JPEG 2000 encoder. Moreover, many possible uses of SIMD extension became available in the case of multi-dimensional core. Particularly, 4×4 vertically vectorized core is the best performing one on the Intel x86 platform with SSE extensions. Table 7.3 summarizes the performance of various cores of this platform. Moreover, the transform employing the cores allow for easy coarse-grained parallelization. The results can be seen in Table 7.4 and Table 7.6. As demonstrated in the JPEG 2000 encoding chain, no synchronizations are even required in between threads considering the horizontal adjacency of parallel blocks. The cores incorporated into the JPEG 2000 compression chain have proven to be fundamentally faster than the widely used implementations.

Moreover, the performance of the cores during the 3-D transform has been evaluated. Approximately above 1 megavoxel, all of the 3-D cores (even unvectorized) outperform all the previous 3-D processing methods. The SIMD-optimized 3-D cores exhibit the best results as shown in Table 7.8.

The cores can also be internally reorganized in order to minimize some of the resources. This property was demonstrated on the FPGA where the minimization of the core latency has a direct impact on the utilization of flip-flop circuits and look-up tables (LUT). Specifically, the reduced latency core consumes more LUTs and uses a smaller amount of flip-flops. The overall summary of this experiment for Full HD resolution is shown in Table 7.12.

The cores may also be advantageously used on massively-parallel architectures. This option was demonstrated using the OpenCL framework and the most recent GPUs of two biggest vendors. Specifically, the transform employing the parallel non-separable core reducing the number of memory barriers which was proven to be the fastest way to transform the 2-D data. This behavior is documented in Figure 7.12.

Finally, several general experiments with 1-D lifting scheme was conducted. As can be seen from Figure 7.16, the horizontal vectorization shall be the preferred schedule when long vector operations are available. The situation is opposite at general-purpose processors. As mentioned above, the vertical or diagonal vectorization is faster than the horizontal one as soon as the working set exceed the cache size.

The implementations of the cores discussed in this chapter can be found through the following link: `http://www.fit.vutbr.cz/~ibarina/prods.php`.

As several of the research activities presented in this thesis were performed as a collective work, this paragraph aims to conclude the research achievements of myself. I have designed and formulated the cores of the lifting scheme presented in Chapter 5. This include the single-loop cores as well as the parallel ones. Moreover, this statement also comprises the mutable cores. I have also extended this principle into more dimensions. Based on the formulation of the core, I have proposed several variants of the cores as presented in Section 5.1 and Section 6.1. The 2-stage parallel core disclosed in Section 6.2 was originally the idea of my colleague Michal Kula. All this work was carried out under the guidance of my supervisor Pavel Zemcik.

Chapter 8

Conclusions

The thesis focuses on efficient methods for computing the discrete wavelet transform. The state-of-the-art methods suffer from several ailments. For example, the parallelization, exploitation of SIMD extensions and the cache hierarchy are not handled well. The treatment of signal boundaries is done in a complicated and inflexible way. Additionally, these methods do not address the problem of scheme reorganization in order to minimize some of the resources. The aim of the thesis has been to overcome these issues. This was accomplished with the formation of a compact streaming core which performs the transform in a single loop, possibly in a multi-scale fashion. Using this core, transform fragments can be computed according to application requirements.

New features of the approach presented are indicated by numbers. The presented core can (1) efficiently exploit the capabilities of modern CPUs, especially the cache hierarchy, SIMD extensions, and parallel computing. Operations inside the core can be (2) reorganized in order to minimize some of the platform resources (e.g. the number of memory barriers, the number of steps). Since the core itself (3) treats the signal boundaries, no special prolog or epilog phases are needed. Moreover, the cores can be adapted to (4) massively-parallel environments.

The core can be described as a direct mapping from the input coefficients on the output ones while retaining and exploiting some auxiliary intermediate results. This mapping can be seen as a standalone streaming unit, implemented either in software or hardware. Using the core, the transform fragments can be computed with several (5) new degrees of freedom (the processing order, the interleaving of the multi-scale decomposition). For example, a particular transform block at a particular scale can be obtained with minimal or no unnecessary calculations.

When searching for the best core, I have found that the core can optimize only one criterion at the expense of others. For instance, minimizing the number of arithmetic operations goes against the number of synchronization points (the memory barriers) and the number of scheme steps (the latency). Moreover, a universal core suitable for all cases and environments probably does not exist.

The future work I would like to do comprises the concatenation of the analysis and synthesis cores coupled with some useful algorithm. This can be done on a multi-scale basis. The algorithms can perform, for example, tone-mapping, denoising, compression, etc. Another area of activities can be the generalization to non-linear transforms.

Published Papers

[I] Barina, D.; Klima, O.; Zemcik, P.: Single-Loop Architecture for JPEG 2000. In *International Conference on Image and Signal Processing (ICISP)*, *Lecture Notes in Computer Science (LNCS)*, vol. 9680, Springer, 2016, ISBN 978-3-319-33618-3, pp. 346–355, doi:10.1007/978-3-319-33618-3_35.

[II] Barina, D.; Klima, O.; Zemcik, P.: Single-Loop Software Architecture for JPEG 2000. In *Data Compression Conference (DCC)*, 2016, pp. 582–582, doi:10.1109/DCC.2016.19.

[III] Barina, D.; Musil, M.; Musil, P.; et al.: Single-Loop Approach to 2-D Wavelet Lifting with JPEG 2000 Compatibility. In *Workshop on Applications for Multi-Core Architectures (WAMCA)*, IEEE, 2015, ISBN 978-1-4673-8621-0, pp. 31–36, doi:10.1109/SBAC-PADW.2015.10.

[IV] Barina, D.; Zemcik, P.: Minimum Memory Vectorisation of Wavelet Lifting. In *Advanced Concepts for Intelligent Vision Systems (ACIVS)*, *Lecture Notes in Computer Science (LNCS)*, vol. 8192, Springer, 2013, ISBN 978-3-319-02894-1, pp. 91–101, doi:10.1007/978-3-319-02895-8_9.

[V] Barina, D.; Zemcik, P.: Wavelet Lifting on Application Specific Vector Processor. In *GraphiCon*, GraphiCon Scientific Society, 2013, ISBN 978-5-8044-1402-4, pp. 83–86.

[VI] Barina, D.; Zemcik, P.: Diagonal Vectorisation of 2-D Wavelet Lifting. In *International Conference on Image Processing (ICIP)*, Paris, France: IEEE, 2014, ISBN 978-1-4799-5751-4, pp. 2978–2982, doi:10.1109/ICIP.2014.7025602.

[VII] Barina, D.; Zemcik, P.: Real-Time 3-D Wavelet Lifting. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 2015, ISBN 978-80-86943-65-7, pp. 15–23.

[VIII] Barina, D.; Zemcik, P.: Vectorization and parallelization of 2-D wavelet lifting. *Journal of Real-Time Image Processing (JRTIP)*, in press, ISSN 1861-8200, doi: 10.1007/s11554-015-0486-6.

[IX] Kucis, M.; Barina, D.; Kula, M.; et al.: 2-D Discrete Wavelet Transform Using GPU. In *Workshop on Application for Multi-Core Architectures (WAMCA)*, IEEE, 2014, ISBN 978-1-4799-7014-8, pp. 1–6, doi:10.1109/SBAC-PADW.2014.13.

[X] Kula, M.; Barina, D.; Zemcik, P.: Block-based Approach to 2-D Wavelet Transform on GPUs. In *Information Technology: New Generations (ITNG)*, *Advances in Intelligent Systems and Computing*, vol. 448, Springer, 2016, ISBN 978-3-319-32467-8, pp. 643–653, doi:10.1007/978-3-319-32467-8_56.

References

[1] I. Daubechies, *Ten Lectures on Wavelets*, ser. CBMS-NSF regional conference series in applied mathematics. Philadelphia, Pennsylvania: Society for Industrial and Applied Mathematics, 1992, vol. 61.

[2] ——, “Orthonormal bases of compactly supported wavelets,” *Communications on Pure and Applied Mathematics*, vol. 41, no. 7, pp. 909–996, 1988. doi:`10.1002/cpa.3160410705`

[3] A. Cohen, I. Daubechies, and J.-C. Feauveau, “Biorthogonal bases of compactly supported wavelets,” *Communications on Pure and Applied Mathematics*, vol. 45, no. 5, pp. 485–560, 1992. doi:`10.1002/cpa.3160450502`

[4] S. G. Mallat, “A theory for multiresolution signal decomposition: The wavelet representation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674–693, 1989. doi:`10.1109/34.192463`

[5] ——, “Multifrequency channel decompositions of images and wavelet models,” *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 37, no. 12, pp. 2091–2110, Dec. 1989. doi:`10.1109/29.45554`

[6] W. Sweldens, “The lifting scheme: A new philosophy in biorthogonal wavelet constructions,” in *Wavelet Applications in Signal and Image Processing III*, ser. SPIE, A. F. Laine and M. A. Unser, Eds., vol. 2569. SPIE, 1995, pp. 68–79.

[7] ——, “The lifting scheme: A custom-design construction of biorthogonal wavelets,” *Applied and Computational Harmonic Analysis*, vol. 3, no. 2, pp. 186–200, 1996.

[8] I. Daubechies and W. Sweldens, “Factoring wavelet transforms into lifting steps,” *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, pp. 247–269, 1998. doi:`10.1007/BF02476026`

[9] S. G. Mallat, “Multiresolution approximations and wavelet orthonormal bases of $L_2(R)$,” *Transactions of the American Mathematical Society*, vol. 315, no. 1, pp. 69–87, 1989.

[10] ——, *A Wavelet Tour of Signal Processing: The Sparse Way. With contributions from Gabriel Peyre.*, 3rd ed. Academic Press, 2009.

[11] G. Strang and T. Nguyen, *Wavelets and Filter Banks.* Wellesley-Cambridge, 1996.

[12] R. E. Blahut, *Fast Algorithms for Signal Processing.* Cambridge University Press, 2010.

[13] R. Kutil, “A single-loop approach to SIMD parallelization of 2-D wavelet lifting,” in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2006, pp. 413–420. doi:10.1109/PDP.2006.14

[14] M. Iwahashi, “Four-band decomposition module with minimum rounding operations,” *Electronics Letters*, vol. 43, no. 6, pp. 27–28, 2007. doi:10.1049/el:20073479

[15] M. Iwahashi and H. Kiya, “A new lifting structure of non separable 2D DWT with compatibility to JPEG 2000,” in *Acoustics Speech and Signal Processing (ICASSP)*, 2010, pp. 1306–1309. doi:10.1109/ICASSP.2010.5495427

[16] ——, “Non separable two dimensional discrete wavelet transform for image signals,” in *Discrete Wavelet Transforms – A Compendium of New Approaches and Recent Applications.* InTech, 2013. doi:10.5772/51199

[17] A. R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, “Wavelet transforms that map integers to integers,” *Applied and Computational Harmonic Analysis*, vol. 5, no. 3, pp. 332–369, 1998. doi:10.1006/acha.1997.0238

[18] M. D. Adams and F. Kossentni, “Reversible integer-to-integer wavelet transforms for image compression: performance evaluation and analysis,” *IEEE Transactions on Image Processing*, vol. 9, no. 6, pp. 1010–1024, Jun. 2000. doi:10.1109/83.846244

[19] M. D. Adams, “Reversible integer-to-integer wavelet transforms for image coding,” Ph.D. dissertation, Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada, Sep. 2002.

[20] R. Fattal, “Edge-avoiding wavelets and their applications,” *ACM Transactions on Graphics*, vol. 28, no. 3, pp. 1–10, 2009. doi:10.1145/1531326.1531328

[21] G. Uytterhoeven and A. Bultheel, "The red-black wavelet transform," in *Benelux Signal Processing Symposium.* IEEE, 1998, pp. 191–194.

[22] D. L. Donoho and I. M. Johnstone, "Ideal spatial adaptation by wavelet shrinkage," *Biometrika*, vol. 81, pp. 425–455, 1994.

[23] U. Drepper, "What every programmer should know about memory," Red Hat, Inc., Tech. Rep., Nov. 2007.

[24] C. Chrysafis and A. Ortega, "Minimum memory implementations of the lifting scheme," in *Wavelet Applications in Signal and Image Processing VIII*, ser. SPIE, A. Aldroubi, A. F. Laine, and M. A. Unser, Eds., vol. 4119. SPIE, 2000, pp. 313–324. doi:10.1117/12.408615

[25] P. Meerwald, R. Norcen, and A. Uhl, "Cache issues with JPEG2000 wavelet lifting," in *Visual Communications and Image Processing (VCIP)*, ser. SPIE, vol. 4671, 2002, pp. 626–634.

[26] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Improving the memory behavior of vertical filtering in the discrete wavelet transform," in *Computing frontiers (CF).* ACM, 2006, pp. 253–260. doi:10.1145/1128022.1128056

[27] ——, "Implementing the 2-D wavelet transform on SIMD-enhanced general-purpose processors," *Transactions on Multimedia*, vol. 10, no. 1, pp. 43–51, Jan. 2008. doi:10.1109/TMM.2007.911195

[28] J. Tao and A. Shahbahrami, "Data locality optimization based on comprehensive knowledge of the cache miss reason: A case study with DWT," in *High Performance Computing and Communications (HPCC).* IEEE, Sep. 2008, pp. 304–311. doi:10.1109/HPCC.2008.7

[29] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado, "Vectorization of the 2D wavelet lifting transform using SIMD extensions," in *Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2003. doi:10.1109/IPDPS.2003.1213416

[30] S. Chatterjee and C. D. Brooks, "Cache-efficient wavelet lifting in JPEG 2000," in *International Conference on Multimedia and Expo (ICME)*, vol. 1. IEEE, 2002, pp. 797–800. doi:10.1109/ICME.2002.1035902

[31] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *International Confer-*

ence on Supercomputing (ICS). New York, NY, USA: ACM, 1999, pp. 444–453. doi:10.1145/305138.305231

[32] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado, "Wavelet transform for large scale image processing on modern microprocessors," in *High Performance Computing for Computational Science (VECPAR)*, ser. Lecture Notes in Computer Science (LNCS), J. M. L. M. Palma, A. A. Sousa, J. Dongarra, and V. Hernandez, Eds. Springer, 2003, vol. 2565, pp. 549–562. doi:10.1007/3-540-36569-9_37

[33] D. Chaver, M. Prieto, L. Pinuel, and F. Tirado, "Parallel wavelet transform for large scale image processing," in *Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2002. doi:10.1109/IPDPS.2002.1015472

[34] D. Chaver, C. Tenllado, L. Pinuel, M. Prieto, and F. Tirado, "2-D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and SIMD parallelism exploitation," in *High Performance Computing (HiPC)*, ser. Lecture Notes in Computer Science, S. Sahni, V. K. Prasanna, and U. Shukla, Eds. Springer, 2002, vol. 2552, pp. 9–21. doi:10.1007/3-540-36265-7_2

[35] A. Shahbahrami and B. Juurlink, "A comparison of two SIMD implementations of the 2D discrete wavelet transform," in *Annual Workshop on Circuits, Systems and Signal Processing*, Veldhoven, The Netherlands, Nov. 2007, pp. 169–177.

[36] A. Shahbahrami, "Improving the performance of 2D discrete wavelet transform using data-level parallelism," in *High Performance Computing and Simulation (HPCS)*, Jul. 2011, pp. 362–368. doi:10.1109/HPCSim.2011.5999847

[37] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *Transactions on Image Processing*, vol. 9, no. 3, pp. 378–389, 2000. doi:10.1109/83.826776

[38] J. Oliver, E. Oliver, and M. P. Malumbres, "On the efficient memory usage in the lifting scheme for the two-dimensional wavelet transform computation," in *International Conference on Image Processing (ICIP)*, vol. 1. IEEE, Sep. 2005, pp. I–485–8. doi:10.1109/ICIP.2005.1529793

[39] A. Shahbahrami and B. Juurlink, "SIMD architectural enhancements to improve the performance of the 2D discrete wavelet transform," in *Digital System Design, Architectures, Methods and Tools (DSD)*, Aug. 2009, pp. 497–504. doi:10.1109/DSD.2009.189

[40] R. Kutil, P. Eder, and M. Watzl, “SIMD parallelization of common wavelet filters,” in *Parallel Numerics*, 2005, pp. 141–149.

[41] R. Kutil and P. Eder, “Parallelization of wavelet filters using SIMD extensions,” *Parallel Processing Letters*, vol. 16, no. 3, pp. 335–349, 2006. doi:`10.1142/S012962640600268X`

[42] J. Maly and P. Rajmic, “Lifting-based wavelet transform for images on modern CPU architectures,” in *International Conference on Signals and Electronic Systems (ICSES)*, Sep. 2008, pp. 177–180. doi:`10.1109/ICSES.2008.4673386`

[43] A. Shahbahrami, “Algorithms and architectures for 2D discrete wavelet transform,” *The Journal of Supercomputing*, vol. 62, no. 2, pp. 1045–1064, 2012. doi:`10.1007/s11227-012-0790-x`

[44] G. Bernabe, J. M. Garcia, and J. Gonzalez, “Reducing 3D fast wavelet transform execution time using blocking and the streaming SIMD extensions,” *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 41, no. 2, pp. 209–223, 2005. doi:`10.1007/s11265-005-6651-6`

[45] G. Bernabe, R. Fernandez, J. M. Garcia, M. E. Acacio, and J. Gonzalez, “An efficient implementation of a 3D wavelet transform based encoder on hyper-threading technology,” *Parallel Computing*, vol. 33, no. 1, pp. 54–72, 2007. doi:`10.1016/j.parco.2006.11.011`

[46] O. M. Lopez-Granado, M. O. Martinez-Rach, P. Pinol, M. P. Malumbres, and J. Oliver, “A fast 3D-DWT video encoder with reduced memory usage suitable for IPTV,” in *International Conference on Multimedia and Expo (ICME)*. IEEE, Jul. 2010, pp. 1337–1341. doi:`10.1109/ICME.2010.5583570`

[47] V. Galiano, O. Lopez-Granado, M. P. Malumbres, and H. Migallon, “Multicore-based 3D-DWT video encoder,” *EURASIP Journal on Advances in Signal Processing*, vol. 2013, no. 1, 2013. doi:`10.1186/1687-6180-2013-84`

[48] E. Belyaev, K. Egiazarian, and M. Gabbouj, “Low complexity bit-plane entropy coding for 3-D DWT-based video compression,” in *SPIE*, vol. 8304, 2012. doi:`10.1117/12.912017`

[49] M. E. Angelopoulou, K. Masselos, P. Y. K. Cheung, and Y. Andreopoulos, “Implementation and comparison of the 5/3 lifting 2D discrete wavelet transform compu-

tation schedules on FPGAs," *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 3–21, 2008. doi:`10.1007/s11265-007-0139-5`

[50] C. Zhang, C. Wang, and M. O. Ahmad, "A pipeline VLSI architecture for fast computation of the 2-D discrete wavelet transform," *Transactions on Circuits and Systems I*, vol. 59, no. 8, pp. 1775–1785, Aug. 2012. doi:`10.1109/TCSI.2011.2180432`

[51] Y.-H. Seo and D.-W. Kim, "VLSI architecture of line-based lifting wavelet transform for motion JPEG2000," *Journal of Solid-State Circuits*, vol. 42, no. 2, pp. 431–440, Feb. 2007. doi:`10.1109/JSSC.2006.889368`

[52] A. Descampe, F. Devaux, G. Rouvroy, B. Macq, and J.-D. Legat, "An efficient FPGA implementation of a flexible JPEG2000 decoder for digital cinema," in *European Signal Processing Conference (EUSIPCO)*, Sep. 2004, pp. 2019–2022.

[53] G. Dillen, B. Georis, J.-D. Legat, and O. Cantineau, "Combined line-based architecture for the 5-3 and 9-7 wavelet transform of JPEG2000," *Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 9, pp. 944–950, Sep. 2003. doi:`10.1109/TCSVT.2003.816518`

[54] C. Tenllado, R. Lario, M. Prieto, and F. Tirado, "The 2D discrete wavelet transform on programmable graphics hardware," in *Visualization, Imaging, and Image Processing*, 9 2004, pp. 808–813.

[55] 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 Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 299–310, 2008. doi:`10.1109/TPDS.2007.70716`

[56] J. Franco, G. Bernabe, J. Fernandez, and M. E. Acacio, "A parallel implementation of the 2D wavelet transform using CUDA," in *Parallel, Distributed and Network-based Processing (PDP)*, 2 2009, pp. 111–118. doi:`10.1109/PDP.2009.40`

[57] M. Blazewicz, M. Ciznicki, P. Kopta, K. Kurowski, and P. Lichocki, "Two-dimensional discrete wavelet transform on large images for hybrid computing architectures: GPU and CELL," in *Euro-Par 2011: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science (LNCS). Springer, 2012, vol. 7155, pp. 481–490. doi:`10.1007/978-3-642-29737-3_53`

[58] V. Galiano, O. Lopez, M. Malumbres, and H. Migallon, “Improving the discrete wavelet transform computation from multicore to GPU-based algorithms,” in *International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, 2011, pp. 544–555.

[59] ——, “Parallel strategies for 2D discrete wavelet transform in shared memory systems and GPUs,” *The Journal of Supercomputing*, vol. 64, no. 1, pp. 4–16, 2013. doi:`10.1007/s11227-012-0750-5`

[60] W. J. van der Laan, J. B. T. M. Roerdink, and A. C. Jalba, “Accelerating wavelet-based video coding on graphics hardware using CUDA,” in *International Symposium on Image and Signal Processing and Analysis (ISPA)*, Sep. 2009, pp. 608–613.

[61] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink, “Accelerating wavelet lifting on graphics hardware using CUDA,” *Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 132–146, 2011. doi:`10.1109/TPDS.2010.143`

[62] J. Franco, G. Bernabe, J. Fernandez, and M. Ujaldon, “Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs,” *Procedia Computer Science*, vol. 1, no. 1, pp. 1101–1110, 2010, ICCS 2010. doi:`10.1016/j.procs.2010.04.122`

[63] G. Bernabe, G. D. Guerrero, and J. Fernandez, “CUDA and OpenCL implementations of 3D fast wavelet transform,” in *Latin American Symposium on Circuits and Systems (LASCAS)*. IEEE, Feb. 2012, pp. 1–4. doi:`10.1109/LASCAS.2012.6180318`

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

[65] R. Kutil, “Short-vector SIMD parallelization in signal processing,” in *Parallel Computing*, R. Trobec, M. Vajtersic, and P. Zinterhof, Eds. Springer, 2009, pp. 397–433. doi:`10.1007/978-1-84882-409-6_13`

[66] J. Sykora, L. Kohout, R. Bartosinski, L. Kafka, M. Danek, and P. Honzik, “The architecture and the technology characterization of an FPGA-based customizable Application-Specific Vector Processor,” in *Design and Diagnostics of Electronic Circuits Systems (DDECS)*. IEEE, 2012, pp. 62–67. doi:`10.1109/DDECS.2012.6219026`

[67] J. Sykora, R. Bartosinski, L. Kohout, M. Danek, and P. Honzik, “Reducing instruction issue overheads in Application-Specific Vector Processors,” in *Euromicro Conference on Digital System Design (DSD)*, 2012, pp. 600–607.

[68] R. Bartosinski, M. Danek, J. Sykora, L. Kohout, and P. Honzik, “Foreground detection and image segmentation in a flexible ASVP platform for FPGAs,” in *Design and Architectures for Signal and Image Processing (DASIP)*, 2012, pp. 1–2.

[69] ——, “Video surveillance application based on application specific vector processors,” in *Design and Architectures for Signal and Image Processing (DASIP)*, 2012, pp. 1–8.

[70] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image Compression Fundamentals, Standards and Practice*, ser. The Springer International Series in Engineering and Computer Science. Springer, 2002.

[71] ——, “JPEG2000: standard for interactive imaging,” *Proceedings of the IEEE*, vol. 90, no. 8, pp. 1336–1357, Aug. 2002. doi:10.1109/JPROC.2002.800725

[72] D. S. Taubman, E. Ordentlich, M. Weinberger, and G. Seroussi, “Embedded block coding in JPEG 2000,” *Signal Processing: Image Communication*, vol. 17, no. 1, pp. 49–72, 2002. doi:10.1016/S0923-5965(01)00028-5

[73] D. S. Taubman, “Software architectures for JPEG2000,” in *International Conference for Digital Signal Processing (DSP)*. IEEE, 2002, pp. 197–200.