Wavelet transform for video coders using GPU

By:
Anders T. Eskildsen
Edwin G. W. Peters
Jack H. Leegaard
Palle Ravn
{aeskil08, egwp08, jleega08, pravn08}@student.aau.dk

Group 841
Signal Processing and Computing 8th semester

31-05-2012

Department of Electronic systems
Electronics & IT
Fredrik Bajers Vej 7
9220 Aalborg Ø
Tel. 99 40 86 00
http://es.aau.dk

Title:
Wavelet transform for
video coders using GPU

Theme:
Scientific Computing

Project period:
SPC8, spring semester 2012

Group:
12gr841

Group members:
Anders T. Eskildsen
Edwin G. W. Peters
Jack H. Leegaard
Palle Ravn

Supervisors:
Tobias Lindstrøm Jensen
Søren Holt Jensen
Torben Larsen

Impressions:

Number of pages: 104

Appendix: 4 + CD

Completed: 31^{th} of May, 2012

Synopsis:

Compression is essential for video coding, but common used methods result in blocking artifacts. To improve this the wavelet transform can be used to transform entire frames. This has a high potential concerning compression, but is very heavy computationally, which is critical for real-time video coding.
This report investigates the potential to perform the CDF 9/7 wavelet transform for real-time (25 FPS) video coding on a GPU. To obtain knowledge of the wavelet transform this is investigated, including filtering and lifting as transform methods. The architectures of the CPU and GPU are examined including OpenCL as API for the GPU. Three commonly used algorithms called Row-Column, Line-Based and Block-Based are analysed, and their potential for GPU implementation is discussed. The implementation is based on Row-Column
Implementations for both filtering and lifting-have been developed for comparison. Both the forward and reverse transforms are implemented for lifting, whereas only forward for filtering. The implementation can perform a full decomposition of square power of two resolutions, and as many levels as possible for HD. As full decomposition is desired due to compression, a padding functionality is implemented, padding HD resolutions into square resolutions.
The implementations are benchmarked using various image resolutions, measuring internal times, giving a thorough specification of the performance. The results show that lifting is the fastest, but both perform better than 25 FPS for HD 1920×1080 pixel images. An acceptance test is performed by simulating video coding on video frames of the same resolution. This shows an average frame rate of 60.91 FPS for 3 levels of decomposition and 61.63 FPS for reconstruction. The results show that, using the wavelet transform on a GPU for video coding has potential.

Preface

This report is composed in the spring of 2012, by 8th semester students of "Signal Processing and Computing" studying at Aalborg University in Denmark. The semester theme is "Scientific Computing" and the purpose is to gain knowledge of different computational platforms, parallel and sequential computing, and how to optimize implementations for these. The main focus of the report is programming on massively parallel units, and the analysis of wavelet transforms. The target audience is students and others, with a background in programming, science and engineering, interested in these topics.

The report is sorted in parts, each containing chapters and sections. To ease the locating of figures, tables and alike, these are numbered according to the respective chapter. References are referred to by square brackets containing a number like [1]. These numbers refer to the reference list, found on page 103. Acronyms have been used for the technical terms. A list of these can be found on page vii. For the documentation of the software the following notation is used:

- `variable`
- `function()`
- `class`
- `module`
- **file**

When explaining both theoretical and implemented algorithms, pseudo code is provided to give a better understanding. Furthermore when found necessary, code examples have been shown in listings. A CD is attached, containing a digital version of the report, the scripts used to generate figures, the source code for the implementation, and the test results. The scripts used to generate figures are all located in the folder named "figurescripts".

Anders T. Eskildsen

Edwin G. W. Peters

Jack H. Leegaard

Palle Ravn

Contents

1 Introduction **1**
1.1 Introduction . . . 1
1.2 Problem statement . . . 2
1.3 Requirements . . . 2

I Introduction and theory **5**

2 Wavelet theory **7**
2.1 Continuous wavelet transform . . . 8
2.2 Discrete wavelet transform . . . 10
2.3 Daubechies wavelets . . . 15
2.4 Cohen-Daubechies-Feauveau 9/7 . . . 18
2.5 Lifting . . . 25
2.6 Lifting equations . . . 27
2.7 Boundary effects . . . 32

3 Platform architecture **35**
3.1 Basic architecture . . . 35
3.2 Limitations . . . 36
3.3 Actual and theoretical performance . . . 37
3.4 GPU architecture . . . 39

4 Algorithm analysis **45**
4.1 Row-column . . . 45
4.2 Line-based . . . 48
4.3 Block based . . . 49
4.4 Analysis . . . 53

II Implementation and benchmarking **55**

5 Implementation **57**
5.1 Test and development platform . . . 57
5.2 Implementation structure . . . 58
5.3 Initial setup and basic functions . . . 59
5.4 Row-column filtering . . . 62
5.5 Lifting . . . 66
5.6 Padding HD images . . . 77
5.7 Generating test images . . . 78
5.8 Verification . . . 79

6 Benchmarking **81**
6.1 Considerations . . . 81

6.2 Image benchmarking 82
6.3 Image results 84
6.4 Discussion 90

III Closure **93**

7 Acceptance test **95**
7.1 Video script 95
7.2 Results 97
7.3 Conclusions 97

8 Conclusions **99**

9 Perspective **101**

Bibliography **103**

IV Appendix **105**

A Test images **107**

B Benchmark results **109**

C Video result histograms **111**

D Kernel codes **113**

Acronyms

API	Aplication Programming Interface
BB	Block-Based
CDF	Cohen-Daubechies-Feauveau
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CU	Compute Unit
CWT	Continuous Wavelet Transform
DWT	Discrete Wavelet Transform
DtoH	Device to Host
ECC	Error Correcting Code
FLOP	Floating Point Operation
GPU	Graphics Processing Unit
HtoD	Host to Device
HD	High Definition
JPEG	Joint Photographic Experts Group
LB	Line-Based
MIMD	Multiple Instruction Multiple Data
OpenCL	Open Computing language
PE	Processing Element
PSNR	Peak Signal to Noise Ratio
RC	Row-Column
SIMD	Single Instruction Multiple Data

1 Introduction

1.1 Introduction

High-definition (HD) resolutions have become widely used in the television industry among other places. The higher resolution comes with a trade-off since a hight bandwidth and more computational power is needed for both encoding and decoding videos. Coding video can be done either offline or online. Offline refers to encoding is done and stored in non real-time, i.e., for later viewing like with DVDs. Whereas online means it is done in real-time and transmitted, i.e., a live television broadcast transmitted to the viewers or a video conference. As more and more producenter incorporate HD resolutions in their products, it is desired to use HD resolutions outside the TV-industry as well.

The "Network Sound Project" focuses on making real-time video and sound interaction, allowing musicians to play with each other across large distances via IP based networks[1]. With the current implementation it is only possible to run with a resolution of 382×258 pixels due to high complexity and computational cost. It is desired to have higher resolutions like HD 1280×720 or 1920×1080 pixels. It is also desired that the implementation should be able to run on a consumer grade PC with limited computational power.

Some video and images codecs used to day are: MPEG-4[2], h.264 [1] and JPEG2000[3]. The first two codecs use block and motion estimation which under lossy conditions results in blocking artifacts which are unwanted. The JPEG2000 uses wavelet transforms to compress the entire frame at once, which avoids the blocking effects, but this is also more demanding both computationally and memory wise [1]. Another benefit of the wavelet transform is that it produces a very sparse result, which is ideal for compression and hereby network transmission [2].

Using the Graphics Processing Unit (GPU) of a PC for parallel computing, these computational heavy transformations can be computed faster than on a regular Central Processing Unit (CPU). Many consumer grade PCs today have GPUs which support either Open Computing language (OpenCL) [3] or Compute Unified Device Architecture (CUDA)s [4] parallel computing architectures. Therefore the potential of performing the Cohen-Daubechies-Feauveau (CDF) 9/7 wavelet transform, for real-time (25 fps) video coding on a GPU is investigated. By customizing this to the GPU architecture it might be possible to encode and decode HD video in near real-time. The proposed setup is illustrated on Figure 1.1. The implementation and testing of the CDF 9/7 wavelet transform makes the ground for this project.

[1] Stated by the project proposal by Jan Østergaard and Tobias L. Jensen.
[2] http://mpeg.chiariglione.org/standards/mpeg-4/mpeg-4.html
[3] http://jpeg.org/jpg2000/CDs15444.html

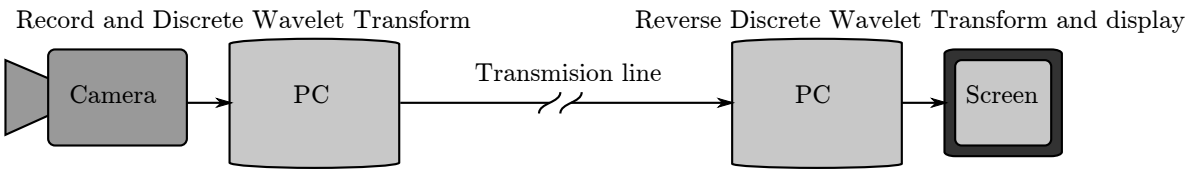


Figure 1.1. Illustration of the project setup. A camera is interfaced with a PC which performs the CDF 9/7 on a GPU, transmits the data to a receiving PC, which performs the inverse CDF 9/7 wavelet transforms and displays the recorded content .

1.2 Problem statement

This project is limited to analyzing, implementing and testing the CDF 9/7 wavelet transform on a consumer grade PC with a Graphics Processing Unit (GPU). The overall problem covered in this report is

> **Can the Cohen-Daubechies-Feauveau (CDF) 9/7 wavelet transform be implemented on a consumer grade PC with a GPU?**

The main problem is divided into subproblems which will be addressed throughout this report:

1. How does the Cohen-Daubechies-Feauveau (CDF) 9/7 wavelet transform work?
2. How is the CPU/GPU architecture defined and what are the differences?
3. How can the wavelet transform algorithm be customized and implemented to fit the GPU architecture?
4. How well does a GPU wavelet implementation perform vs. a CPU implementation?
5. Is it possible perform the wavelet transform in near real-time for HD resolution?

1.3 Requirements

The requirements are partly taken from the project proposal since the main focus for this report is the implementation the CDF 9/7 wavelet transform and test of this. This implementation will only address the following requirements and is limited to wavelet transformation of image/video, hence sound, data transmission, recording and displaying is not considered/implemented. The requirements are:

1. Must be able to perform the CDF 9/7 forward and reverse wavelet transform and on a color image with a resolution up to 1920×1080 pixels.
2. Must perform the forward transform in real-time with a frame rate of 25 FPS.
3. Must perform the reverse transform in real-time with a frame rate of 25 FPS.
4. Must fulfil the above requirements running on a consumer grade PC with a GPU.

This concludes the introduction and ground for this project. It it chosen to implement the CDF 9/7 discrete wavelet transform on GPU. The next part of the report will

explain the theory of wavelet transforms, including the CDF 9/7. The architectural differences between CPUs and GPUs are discussed, followed by a more detailed description of OpenCL. All requirements will be tested either via verification tests or with benchmark in the second part of the report. Finally, a conclusion is made on the implementation and obtained results and additional considerations are put into perspective.

Part I

Introduction and theory

2 Wavelet theory

A wavelet transform [2] is a way to map e.g. a signal into another domain just like the Fourier transform. Comparing the wavelet transform and the Fourier transform the latter uses a kernel consisting of sinusoidal functions, whereas the wavelet transform uses little wavelike functions called *wavelets*. Every wavelet is defined by its function, denoted $\psi(t)$. Figure 2.1 illustrates some commonly used wavelets.

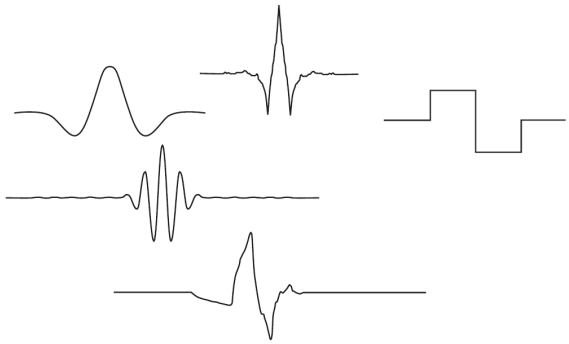


Figure 2.1. Examples of wavelets commonly used in practice [2].

From the figure it is clear that wavelets can look very different, but to be wavelets they must satisfy the following criterias [2].

1. Wavelets must have finite energy:

$$E = \int_{-\infty}^{\infty} |\psi(t)|^2 \, \mathrm{d}t < \infty \tag{2.1}$$

2. If $\bar{\psi}(f)$ is the Fourier transform of $\psi(t)$, the following must hold:

$$C_g = \int_0^{\infty} \frac{|\bar{\psi}(f)|^2}{f} \, \mathrm{d}f < \infty \tag{2.2}$$

   This implies that the wavelet must have zero mean i.e. $\bar{\psi}(0) = 0$. This is called the *admissibility condition* where C_g is the *admissibility constant*.

3. If the wavelet is complex, its Fourier transform must be real and vanish for negative frequencies.

Wavelets are used in various applications, and a few examples are [5]: Image edge detection, digital watermarking, signal denoising and analysis seismic data. One of the most important reason to use a wavelet transform, is the "time" (or local) information that the transform provides, which is something that e.g. the Fourier transform is not capable of.

This chapter first explains the Continuous Wavelet Transform (CWT) together with the principles of performing a wavelet transform. After this the Discrete Wavelet Transform (DWT) is explained including approximation and transform coefficients,

multiresolution representation, filtering, and the 2D DWT. The Daubechies wavelets are then introduced followed by the biorthogonal wavelets and the CDF 9/7. This is followed by the principle of performing the DWT using lifting, and finally boundary effects are considered.

2.1 Continuous wavelet transform

A commonly used wavelet is the *Mexican hat wavelet* [2] given by

$$\hat{\psi}(t) = (1 - t^2) \cdot \mathrm{e}^{-t^2/2}. \tag{2.3}$$

The mexican hat wavelet will be used to explain how the wavelet analysis is performed. Equation (2.3) states the *mother wavelet* or *analysing wavelet*. In the process of analysing a signal, the mother wavelet can be stretched and squeezed (dilation) and/or moved across the signal (translation). The dilation changes the frequency content of the wavelet. The translation on the other hand provides "time" information.

The translation and dilation is controlled by respectively subtracting the *location parameter* b from the time t, and dividing by the *dilation parameter* a. The Mexican hat wavelet in equation (2.3) is then:

$$\hat{\psi}\left(\frac{t-b}{a}\right) = \left[1 - \left(\frac{t-b}{a}\right)^2\right] \cdot \mathrm{e}^{\left[-\frac{1}{2}\left(\frac{t-b}{a}\right)^2\right]}. \tag{2.4}$$

Figure 2.2 and 2.3 illustrate the effect of changing the dilation parameter a and location parameter b.

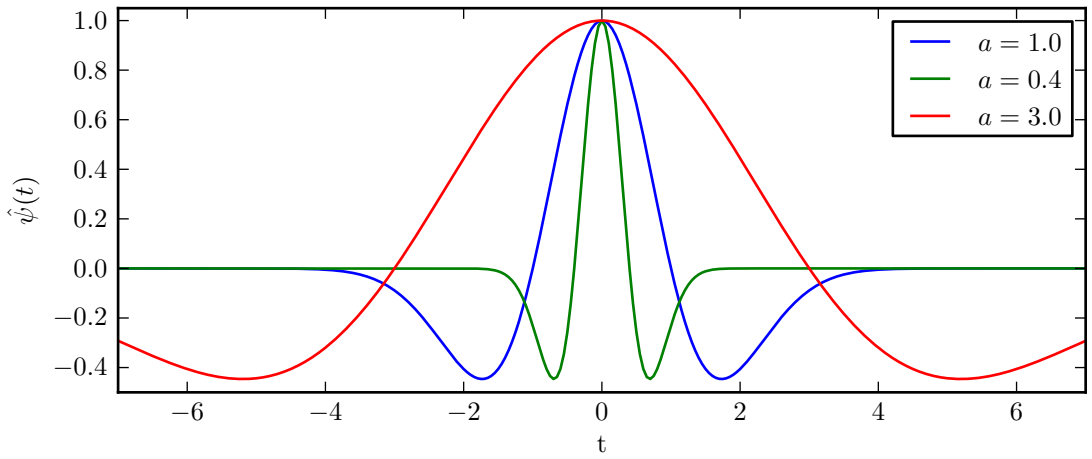


Figure 2.2. Scaling the Mexican hat wavelet (dilation). Here $b = 0$.
Code: **wavelet_dilation_translation.py**.

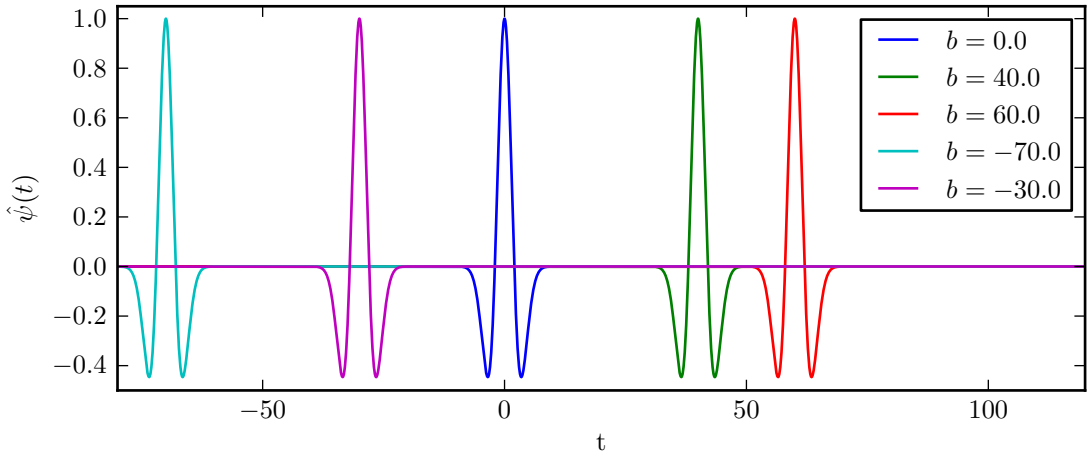


Figure 2.3. Moving the Mexican hat wavelet in time (translation). Here $a = 2$.
Code: **wavelet_dilation_translation.py**.

The wavelet functions are often normalized resulting in unit energy [2]. The energy of the Mexican hat (without dilation and translation) is:

$$E = \int_{-\infty}^{\infty} \hat{\psi}(t)^2 \, \mathrm{d}t = \int_{-\infty}^{\infty} \left[(1 - t^2) \cdot \mathrm{e}^{-t^2/2} \right] \mathrm{d}t = \frac{3}{4}\sqrt{\pi}. \tag{2.5}$$

The normalized Mexican hat wavelet is then:

$$\psi(t) = \frac{1}{\sqrt{E}} \cdot \hat{\psi}(t) = \frac{1}{\sqrt{3/4} \cdot \pi^{1/4}} \cdot (1 - t^2) \cdot \mathrm{e}^{-t^2/2}. \tag{2.6}$$

Using this, the CWT of a signal, $x(t)$, can be computed from the definition [2]:

$$T(a, b) = w(a) \int_{-\infty}^{\infty} x(t) \cdot \psi^* \left(\frac{t - b}{a} \right) \, \mathrm{d}t \tag{2.7}$$

where $x(t)$ is the input signal, $w(a)$ is a weighting function and " * " denotes complex conjugated. Typically $w(a)$ is set to $1/\sqrt{a}$, as this ensures that every scale of the wavelet has the same energy (energy conservation). The wavelet function is often written with the translation, dilation and weighting included:

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}} \cdot \psi \left(\frac{1 - b}{s} \right). \tag{2.8}$$

The CWT is then:

$$T(a, b) = \int_{-\infty}^{\infty} x(t) \cdot \psi^*_{a,b}(t) \, \mathrm{d}t. \tag{2.9}$$

The inverse CWT [2] can then be calculated as:

$$x(t) = \frac{1}{C_g} \int_{-\infty}^{\infty} \int_{0}^{\infty} T(a, b) \cdot \psi_{a,b}(t) \, \frac{\mathrm{d}a \, \mathrm{d}b}{a^2} \tag{2.10}$$

where C_g is the admissibility constant of equation (2.2).

2.2 Discrete wavelet transform

The DWT is performed by discretely shifting the dilation and translation coefficients [2]. Considering the analysis performed on a continuous signal, the wavelet function is given by:

$$\psi_{m,n}(t) = \frac{1}{\sqrt{a_0^m}} \cdot \psi\left(\frac{t - n \cdot b_0 \cdot a_0^m}{a_0^m}\right). \tag{2.11}$$

The dilation parameter, a, is logarithmic discretized, and linked to the location parameter (step size), b. Here m adjusts the logarithmic scaling, with step size a_0, and n adjusts the translation, with step size b_0. Common choices of these parameters are: $a_0 = 2$ and $b_0 = 1$, giving a logarithmic power-of-two scaling of both dilation and location, known as the *dyadic grid* arrangement [2]. Substituting into (2.11) results in

$$\psi_{m,n}(t) = \frac{1}{\sqrt{2^m}} \cdot \psi\left(\frac{t - n \cdot 2^m}{2^m}\right). \tag{2.12}$$

Using this, the (one-step) DWT is

$$T_{m,n} = \int_{-\infty}^{\infty} x(t) \cdot \psi_{m,n}(t)\ \mathrm{d}t \tag{2.13}$$

where $T_{m,n}$ is called the transform coefficient.

Wavelets used in a dyadic grid are commonly chosen to be orthonormal [2], and the following condition must be true:

$$\int_{-\infty}^{\infty} \psi_{m,n}(t) \cdot \psi_{m',n'}(t)\ \mathrm{d}t = \begin{cases} 1 & \text{if } m = m' \text{ and } n = n' \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

This means that all wavelets in the system, translated and/or dilated, are orthonormal. Due to this, the information in the translation coefficient $T_{m,n}$ is unique, and a full reconstruction of the original signal is possible without redundancy.

2.2.1 Approximation and transform coefficients

When performing a DWT on a continuous signal, $x(t)$, this can be transformed into a number of approximation and transform (detail) coefficients [2]. The transform coefficients $T_{m,n}$ was given by equation (2.13). The approximation coefficients $S_{m,n}$ uses the *scaling function* $\phi_{m,n}(t)$ similar to $\psi_{m,n}(t)$ of equation (2.12), and is given by:

$$S_{m,n} = \int_{-\infty}^{\infty} x(t) \cdot \phi_{m,n}(t)\ \mathrm{d}t \tag{2.15}$$

The scaling function $\phi_{m,n}(t)$ performs a smoothing of the signal, and is only orthogonal in translation and not dilation, as is the case for the wavelet function $\psi_{m,n}(t)$ (see (2.14)).

The scaling function of (2.15) can be be build up by contracted and shifted versions of itself. This is stated by the *scaling equation* [2]:

$$\phi(t) = \sum_k h_k \cdot \phi(2t - k) \tag{2.16}$$

where h_k is a *scaling coefficient*, and $\phi(2t-k)$ is a contracted version of $\phi(t)$, shifted in time by step k. This means that a scaling function at one scale can be build up from a number of scaling equations at previous scale. If both sides of (2.16) are integrated, it can be seen that the scaling coefficients, h_k, must fulfill

$$\sum_k h_k = 2. \tag{2.17}$$

For the dyadic grid system to be orthogonal it is in addition required that

$$\sum_k h_k \cdot h_{k+2k'} = \begin{cases} 2 & \text{if } k' = 0 \\ 0 & \text{otherwise} \end{cases}. \tag{2.18}$$

The wavelet function can be build up similar to (2.16) [2]:

$$\psi(t) = \sum_k (-1)^k \cdot h_{N_k-1-k} \cdot \phi(2t-k) \tag{2.19}$$

where N_k is the finite number of scaling coefficients, meaning the wavelet is of compact support. A wavelet of compact support is one which has a finite length sequence of non-zero scaling coefficients. By defining a reconfigured version of the scaling coefficients as

$$g_k = (-1)^k \cdot h_{N_k-1-k} \tag{2.20}$$

and using equations (2.12), (2.16) and (2.19), the scaling and wavelet functions can be written as [2]:

$$\phi_{m+1,n}(t) = \frac{1}{\sqrt{2}} \sum_k h_k \cdot \phi_{m,2n+k}(t) \tag{2.21}$$

and

$$\psi_{m+1,n}(t) = \frac{1}{\sqrt{2}} \sum_k g_k \cdot \psi_{m,2n+k}(t). \tag{2.22}$$

The above equations state that the functions at one scale consist of a sequence of shifted versions of the smaller scale, factored by scaling coefficients.

Finding the approximation and transform coefficients at a specific scale index can then be expressed as [2]:

$$S_{m+1,n} = \frac{1}{\sqrt{2}} \sum_k h_k \cdot S_{m,2n+k} = \frac{1}{\sqrt{2}} \sum_k h_{k-2n} \cdot S_{m,k} \tag{2.23}$$

and

$$T_{m+1,n} = \frac{1}{\sqrt{2}} \sum_k g_k \cdot S_{m,2n+k} = \frac{1}{\sqrt{2}} \sum_k g_{k-2n} \cdot S_{m,k}. \tag{2.24}$$

The wavelet and scaling coefficients at scale m can then be used to find the scaling coefficients at the previous scale $m-1$ [2]:

$$S_{m-1,n} = \frac{1}{\sqrt{2}} \sum_k h_{n-2k} \cdot S_{m,k} + \frac{1}{\sqrt{2}} \sum_k g_{n-2k} \cdot T_{m,k}. \tag{2.25}$$

2.2.2 Multiresolution representation

When applying the DWT repeatedly, which results in a *multiresolution representation*, the continuous signal, $x(t)$, is first transformed into approximation coefficients, $S_{0,n}$ ($n = 0, \ldots, N-1$), at scale index $m = 0$ [2]. Each subsequent time the approximation coefficients, $S_{m,n}$, are decomposed into transform (detail) coefficients $T_{m+1,n}$ and new approximation coefficients $S_{m+1,n}$ [2]. This decomposition is illustrated in Figure 2.4. In the last step of the transform, the approximation coefficients are split into one transform coefficient $T_{M,0}$ and one approximation coefficient $S_{M,0}$, where the latter is the signal mean value.

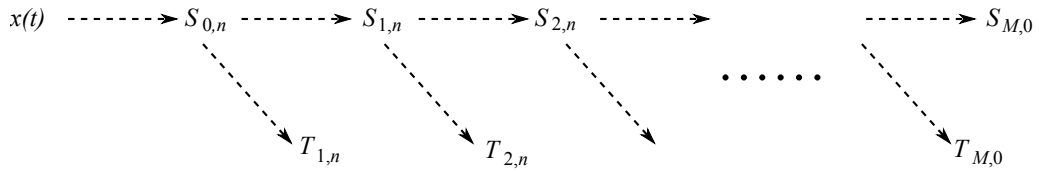


Figure 2.4. Decomposition of the wavelet transform. At start the continuous signal $x(t)$ is transformed into approximation coefficients. Each subsequent time the approximation coefficients are decomposed into approximation and detail coefficients [2].

The approximation and transform coefficients are calculated at different scales, m, using equation (2.23) and (2.24) (and inverse by (2.25)).

2.2.3 Decomposition of a discrete signal

Knowing the continuous signal, $x(t)$, the approximation coefficients at scale index $m = 0$ are

$$S_{0,0}, S_{0,1}, S_{0,2}, \ \ldots \ S_{0,N-1} \tag{2.26}$$

calculated by equation (2.15) on page 10. The original continuous signal can be reconstructed as:

$$x(t) = \sum_{n=-\infty}^{\infty} S_{0,n} \cdot \phi_{0,n}(t). \tag{2.27}$$

If the original continuous signal is not available, but only a discrete representation, $x[n]$ of length N, the coefficients at scale $m = 0$ are set to [2]:

$$S_{0,n} = x[n], \qquad n = 0, \ldots, N-1. \tag{2.28}$$

The multiresolution representation of the transform is then performed as explained earlier. The decomposition can be illustrated schematically as shown in Figure 2.5.

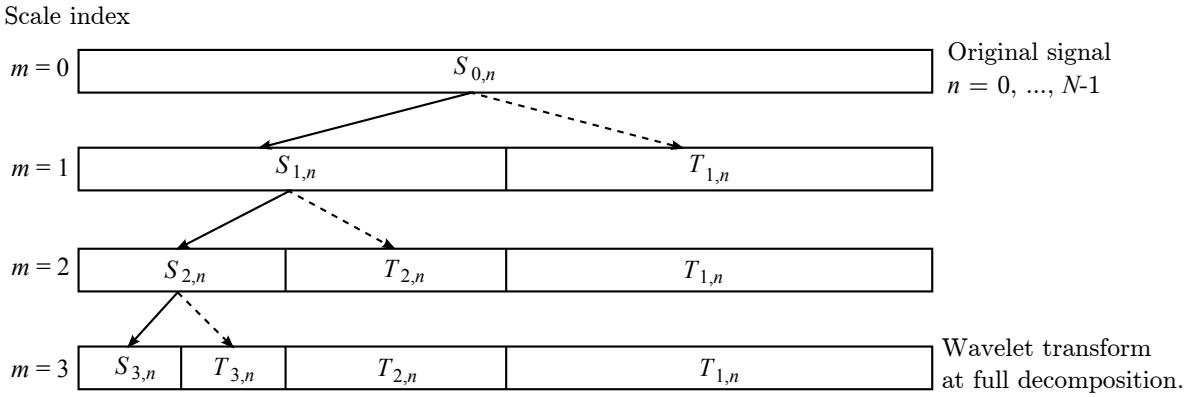


Figure 2.5. Schematic representation of the multiresolution wavelet transform with four scale index levels [2].

2.2.4 Filtering

When looking at the equations (2.21) and (2.21) for the scaling and wavelet functions, we see that these perform convolutions. In fact they work on the input signal as filters. The coefficients h_k and g_k are chosen such that the scaling function acts as lowpass (smoothing) and the wavelet function as highpass [2] (the moment condition, related to the smoothness (lowpass ability) of the filter is presented in section 2.3). Furthermore the filters are moved in steps of two samples resulting in downsampling of the signal. The approximation and transform coefficients (equation (2.23) and (2.24)) are then the downsampled and respectively lowpass and highpass filtered signal. Moving from one scale index level to another in figure 2.5, is then performed as illustrated in figure 2.6. Each time the signal has been filtered and downsampled, the lowpass part is again filtered and downsampled. A full decomposition is obtained when only the signal mean is left as the final value. The reconstruction of the signal is performed as seen in figure 2.7, consistent with equation (2.25).

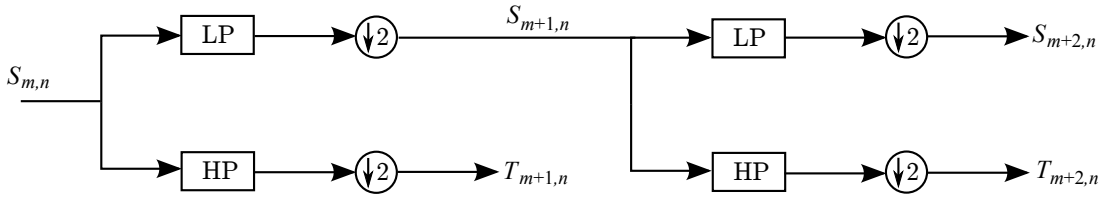


Figure 2.6. Constructing the multiresolution representation by the use of filters and downsampling [2].

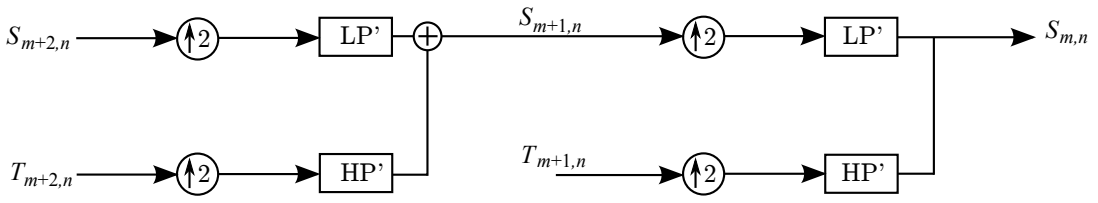


Figure 2.7. Reconstructing the original signal by use of filters and upsampling [2].

2.2.5 DWT by matrix multiplications

The DWT is a linear function and can hence be written as a matrix-vector multiplication. If the signal is represented in the vector $\mathbf{x} = [x[0]\, x[1]\, \ldots\, x[N-1]]^T$, and the filter is the matrix $\mathbf{W}_N \in \mathbb{R}^{N\times N}$, the decomposition vector $\mathbf{y}$ can be found as:

$$\mathbf{y} = \mathbf{W}_N\, \mathbf{x}. \tag{2.29}$$

For a filter of length four, and a signal of lenght $N = 8$, the filter matrix is build up as[1]:

$$\mathbf{W}_8 = \begin{bmatrix} \mathbf{H} \\ \mathbf{G} \end{bmatrix} = \left[\begin{array}{cccccccc} h_3 & h_2 & h_1 & h_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_3 & h_2 & h_1 & h_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_3 & h_2 & h_1 & h_0 \\ h_1 & h_0 & 0 & 0 & 0 & 0 & h_3 & h_2 \\ \hline g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & g_3 & g_2 & g_1 & g_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g_3 & g_2 & g_1 & g_0 \\ g_1 & g_0 & 0 & 0 & 0 & 0 & g_3 & g_2 \end{array}\right] \tag{2.30}$$

The resulting decomposition vector $\mathbf{y}$ is therefore split into an approximation and a detail part, as shown in figure 2.5.

Due to the orthogonality requirements of the transform [5], the filter matrix is orthonormal, meaning $\mathbf{W}_N^T\mathbf{W}_N = \mathbf{I}_N$ and $\mathbf{W}_N^T = \mathbf{W}_N^{-1}$. Because of this the inverse transform can easily be calculated as:

$$\mathbf{x} = \mathbf{W}_N^T\, \mathbf{y} = \mathbf{W}_N^T\, \mathbf{W}_N\, \mathbf{x} = \mathbf{I}_N\, \mathbf{x} \tag{2.31}$$

2.2.6 2D Discrete wavelet transform

The 2D DWT is used when the signal is of 2 dimensions. This would for instance be the case when working with images. Equation (2.15) and (2.24) of the approximation and detail coefficients can be modified into the 2D case as [2]:

Approximation LL:

$$S_{m+1,(n_1,n_2)} = \frac{1}{2}\sum_{k_1}\sum_{k_2} h_{k_1} h_{k_2} S_{m,(2n_1+k_1,2n_2+k_2)} \tag{2.32}$$

Horizontal details LH:

$$T^h_{m+1,(n_1,n_2)} = \frac{1}{2}\sum_{k_1}\sum_{k_2} g_{k_1} h_{k_2} S_{m,(2n_1+k_1,2n_2+k_2)} \tag{2.33}$$

Vertical details HL:

$$T^v_{m+1,(n_1,n_2)} = \frac{1}{2}\sum_{k_1}\sum_{k_2} h_{k_1} g_{k_2} S_{m,(2n_1+k_1,2n_2+k_2)} \tag{2.34}$$

[1] Here we use periodicity for the boundary as in [5]. Boundary effects are considered in section 2.7.

Diagonal details HH:

$$T^d_{m+1,(n_1,n_2)} = \frac{1}{2}\sum_{k_1}\sum_{k_2} g_{k_1} g_{k_2} S_{m,(2n_1+k_1,2n_2+k_2)} \tag{2.35}$$

It should be clear from equations (2.32) to (2.35) that the 2D DWT is performed in two steps. First the one level decomposition of the 1D DWT of all rows is calculated. Then the decomposition of the 1D DWT of all the resulting columns is calculated. Equation (2.32) is the lowpass filtered signal, and (2.33) to (2.35) are the higpass content or details of the signal in each their direction. Much of the literature dealing with 2D DWT uses the notation LL, LH, HL and HH given above. These state the filtering performed on the signal. E.g. LL is two times lowpass filtered, and LH is first lowpass and then highpass filtered. Figure 2.8 illustrates how the decomposition is ordered when performing the transform.

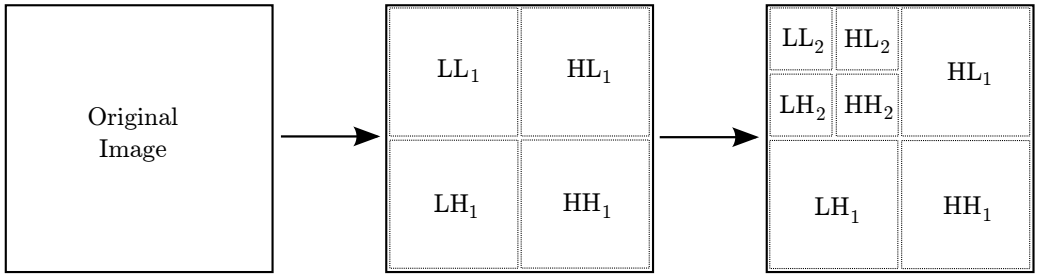


Figure 2.8. Illustration of a 2 level wavelet decomposition performed on a 2D signal, using equation (2.32) to (2.35).

Also for the 2D case, matrix computations are often used for the transform. Here the signal matrix $\mathbf{X}$ must be multiplied on both sides by the filter and transposed filter:

$$\mathbf{Y} = \mathbf{W}_N \, \mathbf{X} \, \mathbf{W}_N^T \tag{2.36}$$

The inverse is found in a similar way to the 1D case:

$$\mathbf{X} = \mathbf{W}_N^T \, \mathbf{Y} \, \mathbf{W}_N = \mathbf{W}_N^T \, \mathbf{W}_N \, \mathbf{X} \, \mathbf{W}_N^T \, \mathbf{W}_N = \mathbf{I}_N \, \mathbf{X} \, \mathbf{I}_N \tag{2.37}$$

2.3 Daubechies wavelets

The Daubechies wavelets are a family of wavelets that satisfy certain conditions [2]. First of all they must fulfill the sum and sum of squares of the scaling coeficients, h_k, specified by equation (2.17) and (2.18). Furthermore they must be of compact support. As a last requirement, the Daubechies wavelets must be smooth to some degree, meaning they must satisfy the moment condition [2]

$$\sum_{k=0}^{N-1} (-1)^k \cdot h_k \cdot k^m = 0 \tag{2.38}$$

where $m = 0, 1, 2 \ldots, N/2 - 1$, and N is the filter length. Formula (2.38) is related to the Fourier transform of a finite length filter:

$$H(\omega) = \sum_{k=0}^{N} h_k \cdot e^{jk\omega}. \tag{2.39}$$

For a $N = L+1$ length filter to be smooth enough (lowpass) the following condition must be satisfied [5]:

$$H^{(d)}(\pi) = 0, \qquad d = 1, \dots, \frac{L-1}{2} \tag{2.40}$$

where d denotes the order of the derivative. The smoothness of the wavelets enables them to suppress parts of the signal up to a degree of $N/2-1$, meaning they have $N/2$ vanishing moments. This can be explained from equations (2.23) and (2.24) of the approximation and detail coefficients. Here a weighted average of the signal is computed at each level. If neighbouring signal values are close in size, the computed value will be near zero [5]. If there are large variations, the computed value will also be large [5].

As an example we will consider the Daubechies wavelets of length four and the shortest of length two, called the Haar wavelet. These are given below [2]:

Daubechies 2 (Haar):

$\mathbf{h} = [1\ 1]^T$
$\mathbf{g} = [1\ \text{-}1]^T$

Daubechies 4:

$\mathbf{h} = [0.483\ 0.837\ 0.224\ \text{-}0.129]^T$
$\mathbf{g} = [\text{-}0.129\ \text{-}0.224\ 0.837\ \text{-}0.483]^T$.

The Haar wavelet leads to a fast computation due to the few coefficients, but it has only 1 vanishing moment. If the analysed signal has large variations of high polynomial degree, the Haar wavelet transform will not catch these [5]. The Haar wavelet can only suppress mean values of the signal, meaning the resulting transform is often not sparse. The Daubechies 4 wavelet has two vanishing moments, and can therefore suppress both the mean and linearity in signals. The Daubechies 6 wavelet can suppress mean, linear and quadratic signals [2].

In figure 2.9 a ramp signal of length 32 is shown, which is a linear signal. The resulting full decompositions, calculated by use of the Haar and Daubechies wavelets, is shown in figure 2.10 [2].

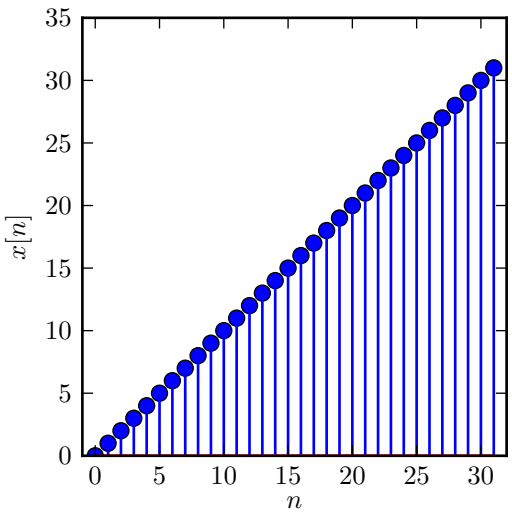


Figure 2.9. Ramp signal used for decomposition example. This is a linear or first order signal.
Code: **vanishing_moments.py**.

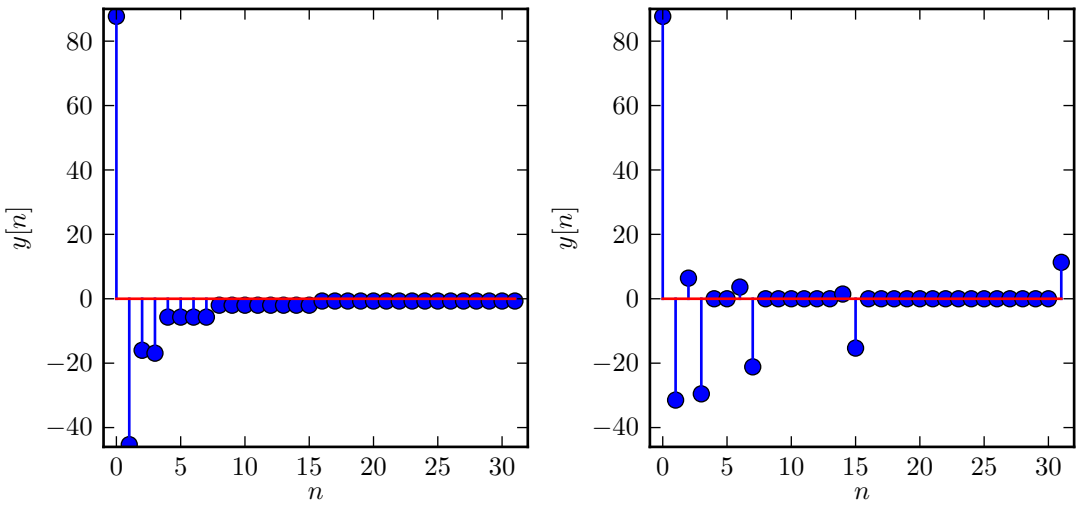


Figure 2.10. Decompositions of the ramp signal of figure 2.9. (a) Calculated by use of the Haar wavelet. (b) Calculated by use of the Daubechies 4 wavelet. [2]
Code: **vanishing_moments.py**.

The values in the figure might look very similar, but when inspecting the decomposition vectors given below, it is clear that this is not the case.

Daubechies 2 (Haar) [2]:

$$\mathbf{y} = [87.7\ -45.3\ -16.0\ -16.9\ -5.7\ -5.7\ -5.7\ -5.7\ -2.0\ -2.0\ -2.0\ -2.0\ -2.0\ -2.0\ -2.0\ -2.0$$
$$-0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7\ -0.7]^T$$

Daubechies 4 [2]:

$$\mathbf{y} = [87.681\ -31.460\ 6.405\ -29.530, 0\ 0\ 3.624\ -21.149\ 0\ 0\ 0\ 0\ 0\ 0\ 1.464\ -15.321$$
$$0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 11.314]^T$$

The peaks of the Daubechies 4 decomposition are the results of periodic boundary handling. The issue of boundary effects is considered in section 2.7. This is something the Haar do not suffer from, as the filters are of length two. As seen there are no zero coefficients in the Haar decomposition, as this only has one vanishing moment. Concerning compression it is desirable to have as sparse a decomposition as possible. The best polynomial approximation is obtained with a long filter, but this results in more computations, and possibly also worse boundary effects. Therefore the most suitable wavelet filter coefficients, depend on the signal.

2.4 Cohen-Daubechies-Feauveau 9/7

In applications like image processing it is desired to have symmetric wavelets, as some argue that the human visual system is more tolerant of symmetric errors [2]. Furthermore, symmetric filters work best for image compression [5]. A symmetric wavelet filter must satisfy:

$$h_k = h_{-k} \qquad \text{for all } k \in \mathbb{Z} \text{ with filter length of odd } N. \tag{2.41}$$

$$h_k = h_{1-k} \qquad \text{for all } k \in \mathbb{Z} \text{ with filter length of even } N. \tag{2.42}$$

Unfortunately the Haar wavelet is the only finite-length, symmetric, orthogonal filter that satisfies the zero derivative conditions of equation (2.40) [5]. Therefore another approach must be used.

2.4.1 Biorthogonal filters

We want to have a symmetric finite-length filter that satisfies the lowpass condition. For the Daubechies wavelets, the filter matrix is orthogonal satisfying $\mathbf{W}_N^T = \mathbf{W}_N^{-1}$. This means that the inverse transform is easily computed. Furthermore the sorting of the output, due to the lowpass/highpass structure, results in fast computation possibilities for certain algorithms (see [5]).

For a non-orthogonal case, the inverse of the wavelet transform matrix must be the transpose of another wavelet transform matrix [5]. To solve this problem, two sets of filter pairs are found: $\mathbf{h}$, $\mathbf{g}$ and $\tilde{\mathbf{h}}$, $\tilde{\mathbf{g}}$. These are used to form the transform matrices $\mathbf{W}_N$ and $\tilde{\mathbf{W}}_N$, such that $\tilde{\mathbf{W}}_N^{-1} = \mathbf{W}_N^T$. Here $\mathbf{h}$ and $\tilde{\mathbf{h}}$ ($\mathbf{g}$ and $\tilde{\mathbf{g}}$) are called a *biorthogonal filter pair* [5]. The 2D DWT using biorthogonal filters is then:

$$\mathbf{Y} = \tilde{\mathbf{W}}_N \, \mathbf{X} \, \mathbf{W}_N^T \tag{2.43}$$

and the inverse:

$$\mathbf{X} = \mathbf{W}_N^T \, \mathbf{Y} \, \tilde{\mathbf{W}}_N = (\mathbf{W}_N^T \, \tilde{\mathbf{W}}_N) \, \mathbf{X} \, (\mathbf{W}_N^T \, \tilde{\mathbf{W}}_N) = \mathbf{I}_N \, \mathbf{X} \, \mathbf{I}_N. \tag{2.44}$$

The filter matrices have the form

$$\mathbf{W}_N = \begin{bmatrix} \mathbf{H} \\ \hline \mathbf{G} \end{bmatrix} \qquad \text{and} \qquad \tilde{\mathbf{W}}_N = \begin{bmatrix} \tilde{\mathbf{H}} \\ \hline \tilde{\mathbf{G}} \end{bmatrix}. \tag{2.45}$$

When finding the filter coefficients, $\mathbf{h}$ and $\tilde{\mathbf{h}}$ must be chosen under certain conditions (explained later) and the detail coefficients can then be found from these, in a similar way to equation (2.20) on page 11, namely as [5]:

$$g_k = (-1)^k \cdot \tilde{h}_{1-k} \tag{2.46}$$
$$\tilde{g}_k = (-1)^k \cdot h_{1-k} \tag{2.47}$$

An example of how the filter coefficients of a biorthogonal 5/3 filters are organized is [5]:

$$\mathbf{W}_8 = \begin{bmatrix} \mathbf{H} \\ \hline \mathbf{G} \end{bmatrix} = \left[\begin{array}{cccccccc} h_0 & h_1 & h_2 & 0 & 0 & 0 & h_{-2} & h_{-1} \\ h_{-2} & h_{-1} & h_0 & h_1 & h_2 & 0 & 0 & 0 \\ 0 & 0 & h_{-2} & h\_1 & h_0 & h_1 & h_2 & 0 \\ h_2 & 0 & 0 & 0 & h_{-2} & h_{-1} & h_0 & h_1 \\ \hline g_0 & g_1 & g_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & 0 \\ g_2 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 \end{array}\right] \tag{2.48}$$

and

$$\tilde{\mathbf{W}}_8 = \begin{bmatrix} \tilde{\mathbf{H}} \\ \hline \tilde{\mathbf{G}} \end{bmatrix} = \left[\begin{array}{cccccccc} \tilde{h}_0 & \tilde{h}_1 & 0 & 0 & 0 & 0 & 0 & \tilde{h}_{-1} \\ 0 & \tilde{h}_{-1} & \tilde{h}_0 & \tilde{h}_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \tilde{h}_{-1} & \tilde{h}_0 & \tilde{h}_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \tilde{h}_{-1} & \tilde{h}_0 & \tilde{h}_1 \\ \hline \tilde{g}_0 & \tilde{g}_1 & \tilde{g}_2 & \tilde{g}_3 & 0 & 0 & 0 & \tilde{g}_{-1} \\ 0 & \tilde{g}_{-1} & \tilde{g}_0 & \tilde{g}_1 & \tilde{g}_2 & \tilde{g}_3 & 0 & 0 \\ 0 & 0 & 0 & \tilde{g}_{-1} & \tilde{g}_0 & \tilde{g}_1 & \tilde{g}_2 & \tilde{g}_3 \\ \tilde{g}_2 & \tilde{g}_3 & 0 & 0 & 0 & \tilde{g}_{-1} & \tilde{g}_0 & \tilde{g}_1 \end{array}\right]. \tag{2.49}$$

The structure in equations (2.48) and (2.49) should be compared to (2.30). Note that the symmetry of the filters are around the coefficients h_0, g_1 and $\tilde{h}_0$, $\tilde{g}_1$ respectively, in accordance with equations (2.46) and (2.47). Because of the orthogonality between the filter pairs the following must be true:

$$\mathbf{W}_N \tilde{\mathbf{W}}_N^T = \begin{bmatrix} \mathbf{H} \\ \hline \mathbf{G} \end{bmatrix} \left[\tilde{\mathbf{H}}^T \;\middle|\; \tilde{\mathbf{G}}^T\right] = \left[\begin{array}{c|c} \mathbf{H}\tilde{\mathbf{H}}^T & \mathbf{H}\tilde{\mathbf{G}}^T \\ \hline \mathbf{G}\tilde{\mathbf{H}}^T & \mathbf{G}\tilde{\mathbf{G}}^T \end{array}\right] = \mathbf{I}_N. \tag{2.50}$$

This can also be stated as:

$$\mathbf{H}\tilde{\mathbf{G}}^T = \mathbf{G}\tilde{\mathbf{H}}^T = \mathbf{0}_{N/2} \tag{2.51}$$
$$\mathbf{H}\tilde{\mathbf{H}}^T = \mathbf{G}\tilde{\mathbf{G}}^T = \mathbf{I}_{N/2} \tag{2.52}$$

where $\mathbf{0}_{N/2} \in \mathbb{R}^{N/2 \times N/2}$ is the zero matrix and $\mathbf{I}_{N/2} \in \mathbb{R}^{N/2 \times N/2}$ is the identity matrix. The Fourier transform of a sequence of filter coefficients $\mathbf{h}$ is given by:

$$H(\omega) = \sum_{k=-\infty}^{\infty} h_k \cdot e^{jk\omega}. \tag{2.53}$$

For the biorthogonal filter pairs $\mathbf{h}$ and $\tilde{\mathbf{h}}$ to be orthogonal, they must satisfy the *biorthogonality condition* [5]:

$$\tilde{H}(\omega)H^*(\omega) + \tilde{H}(\omega+\pi)H^*(\omega+\pi) = 2. \tag{2.54}$$

The filter must also satisfy the lowpass conditions [5]:

$$H(0) = \tilde{H}(0) = \sqrt{2} \tag{2.55}$$
$$H(\pi) = \tilde{H}(\pi) = 0 \tag{2.56}$$

2.4.2 Biorthogonal spline filters

A certain kind of filters, called *biorthogonal spline filters*, use the fact that the Fourier series for a filter of length $\tilde{N}+1$, with $\tilde{N}$ a positive integer, can be written as [5]:

$$\tilde{H}(\omega) = \sqrt{2} \cdot \cos^{\tilde{N}}\left(\frac{\omega}{2}\right) \qquad \text{for } \tilde{N} \text{ even} \tag{2.57}$$

and

$$\tilde{H}(\omega) = \sqrt{2} \cdot e^{j\omega/2} \cdot \cos^{\tilde{N}}\left(\frac{\omega}{2}\right) \qquad \text{for } \tilde{N} \text{ odd.} \tag{2.58}$$

The filter coefficients can then be calculated as [5]:

$$\tilde{h}_k = \frac{\sqrt{2}}{s^{\tilde{N}}} \cdot \binom{\tilde{N}}{\frac{\tilde{N}}{2} - k} \qquad \text{for } \tilde{N} \text{ even}, \quad k = 0, \pm 1, \ldots, \pm\tilde{N}/2 \tag{2.59}$$

and

$$\tilde{h}_k = \frac{\sqrt{2}}{s^{\tilde{N}}} \cdot \binom{\tilde{N}}{\frac{\tilde{N}-1}{2} + k} \qquad \text{for } \tilde{N} \text{ odd}, \quad k = -\frac{\tilde{N}-1}{2}, \ldots, \frac{\tilde{N}+1}{2} \tag{2.60}$$

where the binomial coefficient is calculated as:

$$\binom{a}{b} = \frac{a!}{b! \cdot (a-b)!}. \tag{2.61}$$

A detailed explaination of the derivations can be found in [5]. Using the conditions for orthogonality, symmetry and lowpass, the filter $\mathbf{h}$ can be found by solving a system of equations. Daubechies found the solution to the fourier transform as [5]:

$$H(\omega) = \sqrt{2} \cdot \cos^{N}\left(\frac{\omega}{2}\right) \sum_{s=0}^{l+\tilde{l}-1} \binom{l+\tilde{l}-1+s}{s} \cdot \sin^{2j}\left(\frac{\omega}{2}\right), \qquad N = 2l \tag{2.62}$$

and

$$H(\omega) = \sqrt{2} \cdot e^{j\omega/2} \cdot \cos^{N}\left(\frac{\omega}{2}\right) \sum_{s=0}^{l+\tilde{l}} \binom{l+\tilde{l}+s}{s} \cdot \sin^{2j}\left(\frac{\omega}{2}\right), \qquad N = 2l+1 \tag{2.63}$$

where $N+1$ is the filter length, $\tilde{N} = 2\tilde{l}+1$ for N odd, and $\tilde{N} = 2\tilde{l}$ for N even. The detail coefficients can then be found from $\tilde{\mathbf{h}}$ and $\mathbf{h}$, by use of equations (2.46) and (2.47).

2.4.3 Cohen-Daubechies-Feauveau 9/7

It is desired that the wavelets are very smooth, i.e. they satisfy equation (2.40). This though, causes the biorthogonal filter length to grow [5], leading to more computations when performing the transform. Another problem is that the biorthogonal pairs e.g. $\mathbf{h}$ and $\tilde{\mathbf{h}}$ often have very different lengths. This could be a problem since $\tilde{\mathbf{g}}$ is constructed from $\mathbf{h}$, resulting in the lowpass approximation of $\tilde{\mathbf{W}}_N$ to be worse than the highpass. For image compression it is therefore preferred to have filters of similar lengths. Because of this the JPEG2000 (Joint Photographic Experts Group) lossy compression uses the Cohen-Daubechies-Feauveau (CDF) 9/7 filter pairs, developed by Cohen [5]. These filters are not a member of the biorthogonal spline filters, but they feature symmetry and satisfy the necessary conditions of their Fourier transform to be lowpass filters [5]. The filter lengths of $\mathbf{h}$ and $\tilde{\mathbf{h}}$ are 9 and 7, respectively.

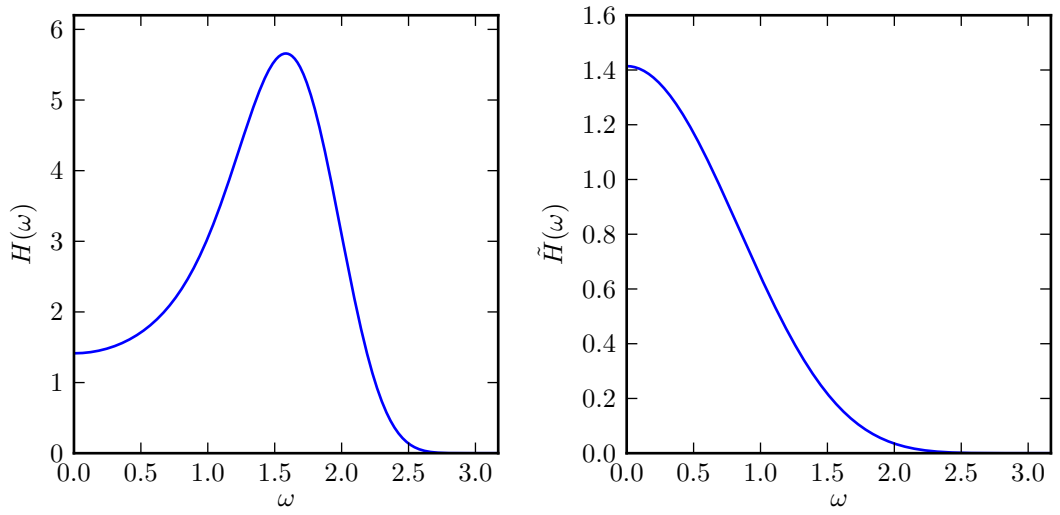
It is possible to produce a biorthogonal spline filter pair also of lengths 9 and 7. This though, does not produce a filter matrix as close to orthogonal as the one for the CDF 9/7 [5]. Another thing is that the biorthogonal spline 9/7 pair satisfy:

$$H(\pi) = H'(\pi) = 0 \qquad \text{and} \qquad \tilde{H}^{(d)}(\pi) = 0, \qquad d = 0, \dots, 5 \tag{2.64}$$

whereas the CDF 9/7 satisfy:

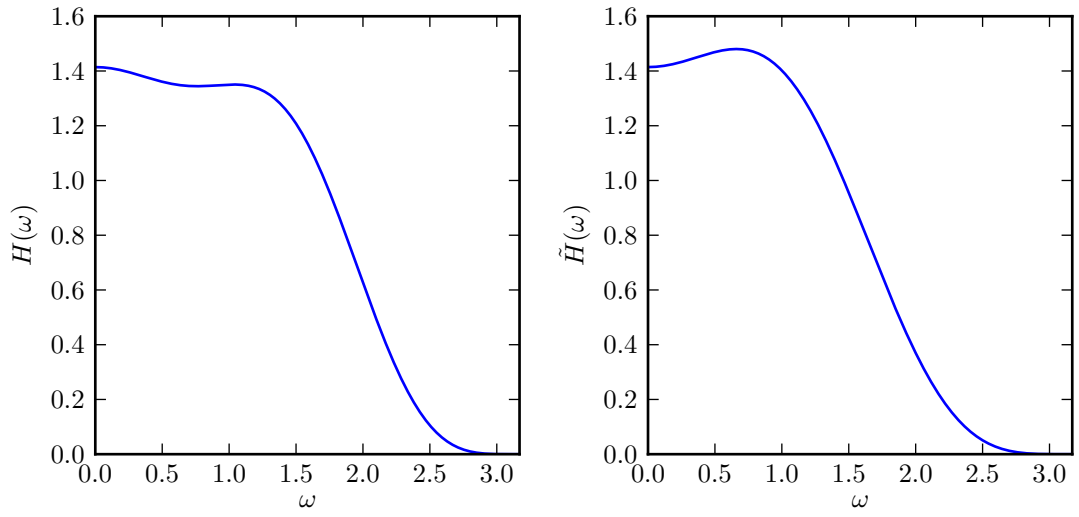
$$H^{(d)}(\pi) = \tilde{H}^{(d)}(\pi) = 0, \qquad d = 0, \dots, 3. \tag{2.65}$$

where d denotes the order of derivatives (see equation (2.40)). This means that the zeroes of the filters are equally balanced for the CDF 9/7, and not for the biorthogonal spline 9/7. For an illustrative comparison, $|H(\omega)|$ and $|\tilde{H}(\omega)|$ for both filter pairs are plotted in figures 2.11 and 2.12. These all satisfy the lowpass conditions from equations (2.55) and (2.56), as they cross the y-axis in $\sqrt{2}$ and are zero at π. The plots for the biorthogonal 9/7 pair are generated by use of equation (2.57) and (2.62), and for the CDF 9/7 the Fourier transform of the filter coefficients i used (see table 2.1).



(a) (b)

Figure 2.11. Fourier transform of the biorthogonal 9/7 smooth filter pair. (a) $|H(\omega)|$. (b) $|\tilde{H}(\omega)|$. Code: **biorspline97_cdf97_fourier.py**.



(a) (b)

Figure 2.12. Fourier transform of the CDF 9/7 smooth filter pair. (a) $|H(\omega)|$. (b) $|\tilde{H}(\omega)|$. Code: **biorspline97_cdf97_fourier.py**.

The equal balancing is obtained using that the Fourier transform of symmetric odd length filters can be expressed as [5]:

$$H(\omega) = \sqrt{2}\cos^{2l}\left(\frac{\omega}{2}\right) \cdot p(\cos(\omega)) \tag{2.66}$$

and

$$\tilde{H}(\omega) = \sqrt{2}\cos^{2\tilde{l}}\left(\frac{\omega}{2}\right) \cdot \tilde{p}(\cos(\omega)) \tag{2.67}$$

where l and $\tilde{l}$ are nonnegative integers and $p(-1) \neq 0$, $\tilde{p}(-1) \neq 0$, $p(1) = 1$ and $\tilde{p}(1) = 1$. To satisfy the biorthogonality condition of equation (2.54), the following condition must be fulfilled [5]:

$$p(\cos(\omega)) \cdot \tilde{p}(\cos(\omega)) = \sum_{s=0}^{K-1} \binom{K-1+s}{s} \cdot \sin^{2s}\left(\frac{\omega}{2}\right) \tag{2.68}$$

where $K = l + \tilde{l}$ and the degree of $p(t)\tilde{p}(t)$ is less than K. To find the filter coefficients for the CDF 9/7, Cohen factorized the right hand side of (2.68), spreading the factors as equally as possible between $p(t)$ and $\tilde{p}(t)$, and used these and the lowpass conditions to solve a system of equations [5].

The wavelet filter coefficients of the CDF 9/7 are listed in table 2.1, rounded to eight decimals. The detail coefficients are found using formula (2.46) and (2.47). These are listed in table 2.2. For the purpose of illustration, the filters are plotted in figures 2.13 and 2.14.

As stated earlier, using biorthogonal filter pairs the 2D DWT is calculated as:

$$\mathbf{Y} = \tilde{\mathbf{W}}_N \, \mathbf{X} \, \mathbf{W}_N^T \tag{2.69}$$

and the inverse:

$$\mathbf{X} = \mathbf{W}_N^T \, \mathbf{Y} \, \tilde{\mathbf{W}}_N = (\mathbf{W}_N^T \, \tilde{\mathbf{W}}_N) \, \mathbf{X} \, (\mathbf{W}_N^T \, \tilde{\mathbf{W}}_N) = \mathbf{I}_N \, \mathbf{X} \, \mathbf{I}_N. \tag{2.70}$$

For the CDF 9/7 it is common to use the filter matrix $\mathbf{W}_N$, containing $\mathbf{h}$ and $\mathbf{g}$, for the decomposition, and $\tilde{\mathbf{W}}_N$, containing $\tilde{\mathbf{h}}$ and $\tilde{\mathbf{g}}$, for the reconstruction. This results in the 2D DWT:

$$\mathbf{Y} = \mathbf{W}_N \, \mathbf{X} \, \mathbf{W}_N^T \tag{2.71}$$

and the inverse:

$$\mathbf{X} = \mathbf{W}_N^T \, \mathbf{Y} \, \tilde{\mathbf{W}}_N = (\tilde{\mathbf{W}}_N^T \, \mathbf{W}_N) \, \mathbf{X} \, (\mathbf{W}_N^T \, \tilde{\mathbf{W}}_N) = \mathbf{I}_N \, \mathbf{X} \, \mathbf{I}_N. \tag{2.72}$$

CDF 9/7 smooth filter pair			
h_0	0.85269868	$\tilde{h}_0$	0.78848562
$h_{-1} = h_1$	0.37740286	$\tilde{h}_{-1} = \tilde{h}_1$	0.41809227
$h_{-2} = h_2$	-0.11062440	$\tilde{h}_{-2} = \tilde{h}_2$	-0.040689418
$h_{-3} = h_3$	-0.023849465	$\tilde{h}_{-3} = \tilde{h}_3$	-0.064538883
$h_{-4} = h_4$	0.037828456		

Table 2.1. CDF 9/7 biorthogonal smooth filter coefficients. [6][5]

CDF 9/7 detail filter pair			
g_1	-0.78848562	$\tilde{g}_1$	-0.85269868
$g_2 = g_0$	0.41809227	$\tilde{g}_2 = \tilde{g}_0$	0.37740286
$g_3 = g_{-1}$	0.040689418	$\tilde{g}_3 = \tilde{g}_{-1}$	0.11062440
$g_4 = g_{-2}$	-0.064538883	$\tilde{g}_4 = \tilde{g}_{-2}$	-0.023849465
		$\tilde{g}_5 = \tilde{g}_{-3}$	-0.037828456

Table 2.2. CDF 9/7 biorthogonal detail filter coefficients. [6][5]

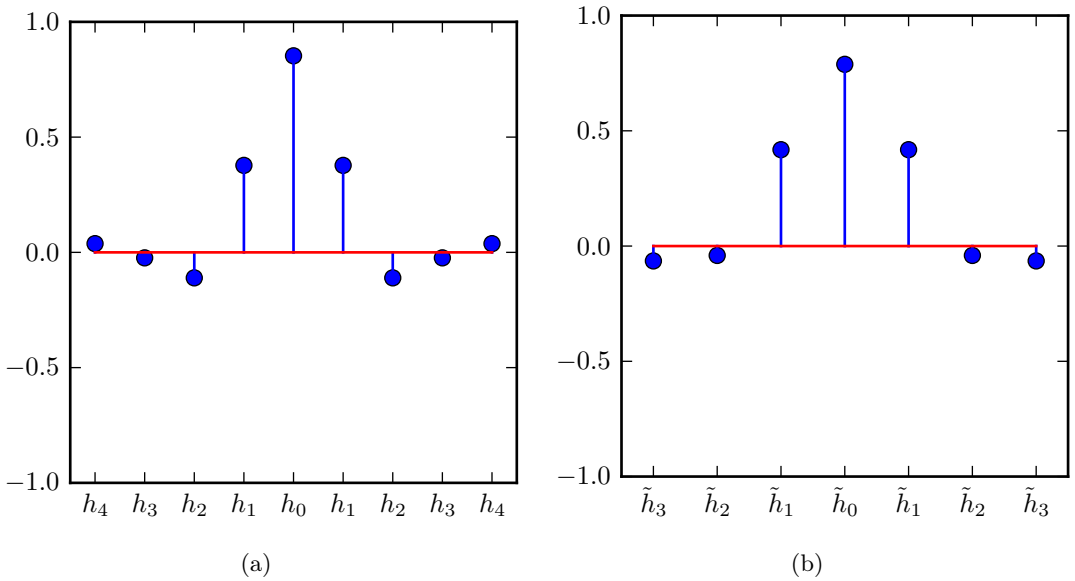


Figure 2.13. CDF 9/7 biorthogonal smooth filter pair. (a) The 9 length filter coefficients. (b) The 7 length filter coefficients.
Code: **cfd97coef.py**.

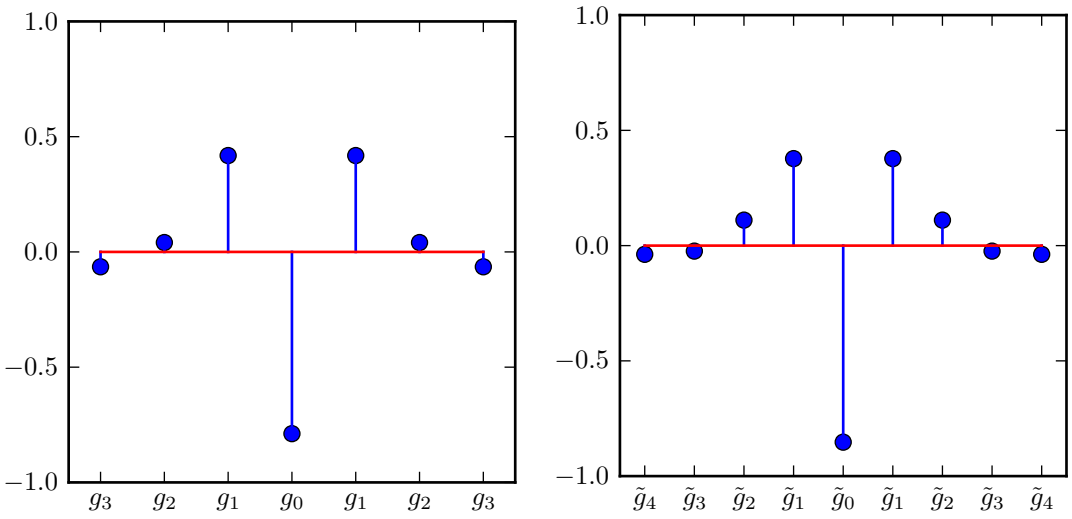


Figure 2.14. CDF 9/7 biorthogonal detail filter pair. (a) The 7 length filter coefficients. (b) The 9 length filter coefficients.
Code: **cfd97coef.py**.

2.5 Lifting

The lifting principle is an alternative approach to filtering, used to decompose any discrete wavelet transform into simple filtering sequences. The idea behind lifting is to reduce the low- and high-pass filters to simple lifting equations, requiring about half of the computations compared to a filter implementation.

An overview of the lifting decomposition is shown in figure 2.15, where the signal is split into even and odd parts, x_e and x_o in the first step, respectively.

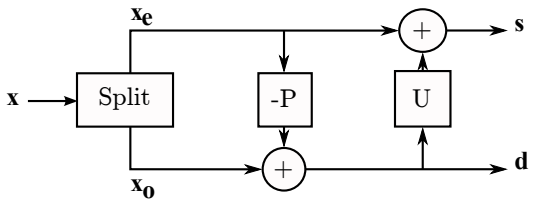


Figure 2.15. Block diagram of the lifting scheme [7].

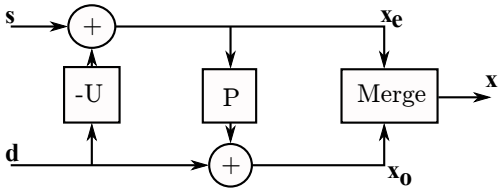


Figure 2.16. Block diagram of the reverse lifting scheme [7].

After the signal is split, a prediction is made . This is done on the assumption, that a sample and its nearest neighbors have some sort of correlation. This could for instance be the difference d between them. Which can be calculated as:

$$d[n] = x_o[n] - P(\mathbf{x}_e) \tag{2.73}$$

where $x_o[n]$ is an odd pixel. $\mathbf{x}_e$ is a sequence of even pixels, neighboring the corresponding x_o. The predictor function, P, operates on a sequence of even pixels, $\mathbf{x}_e$ [8]. These could for instance the pixels located next to x_o. In this way, we can restore x_o from the prediction of the odd sample:

$$x_{\mathrm{o}}[n] = d[n] + P(\mathbf{x}_{\mathrm{e}}) \tag{2.74}$$

The disadvantage is, that this is obtained by subsampling the odd pixels, which results in aliasing [8]. To prevent this, an update step is introduced to preserve the knowledge of the signal. This could be the mean value of the neighboring pixels [7]. Here U is a function on the predictors $\mathbf{d}$:

$$s[n] = x_{\mathrm{e}}[n] + U(\mathbf{d}) \tag{2.75}$$

where $\mathbf{d}$ is a sequence of differences, obtained by using equation (2.73) and $s[n]$ is the mean value of pixel $x_{\mathrm{e}}[n]$ and its neighboring differences $\mathbf{d}$. The results $\mathbf{d}$ and $\mathbf{s}$ are the decomposition of the signal $\mathbf{x}$, where the inverse is called the reconstruction of the signal $\mathbf{x}$.

Both steps are reversible, so the original pixel values x_{e} and x_{o} can be restored. The block diagram of this is shown in figure 2.15.

The reversed lifting steps are easily calculated by reverting the steps in equations (2.73) and (2.75), which become:

$$x_{\mathrm{e}}[n] = s[n] - U(\mathbf{d}) \tag{2.76}$$

$$x_{\mathrm{o}}[n] = d[n] + P(\mathbf{x}_{\mathrm{e}}). \tag{2.77}$$

The decomposition can be repeated on $\mathbf{s}$, which represents the low-pass part, to get further levels of decomposition, just like wavelet filters. The principle of lifting using the Haar wavelet is illustrated in table 2.3.

$x[0]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$x[6]$	$x[7]$	
$s_1[0]$	$s_1[1]$	$s_1[2]$	$s_1[3]$	$\mathbf{d_1[0]}$	$\mathbf{d_1[1]}$	$\mathbf{d_1[2]}$	$\mathbf{d_1[3]}$	**Level 1**
$s_2[0]$	$s_2[1]$	$\mathbf{d_2[0]}$	$\mathbf{d_2[1]}$	$\mathbf{d_1[0]}$	$\mathbf{d_1[1]}$	$\mathbf{d_1[2]}$	$\mathbf{d_1[3]}$	**Level 2**
$s_3[0]$	$\mathbf{d_3[0]}$	$\mathbf{d_2[0]}$	$\mathbf{d_2[1]}$	$\mathbf{d_1[0]}$	$\mathbf{d_1[1]}$	$\mathbf{d_1[2]}$	$\mathbf{d_1[3]}$	**Level 3**

Table 2.3. Three levels of decomposition with the Haar wavelet. The ds are differences whereas the s are mean values.

The Haar wavelet is for instance decomposed into the following steps [7]:

$$d[n] = \frac{1}{2}\left(x[2n+1] - x[2n]\right) \tag{2.78}$$

$$s[n] = x[2n] + d[n]. \tag{2.79}$$

Table 2.4 shows an example using the Haar wavelet on numbers to demonstrate the principle of lifting. The first 4 numbers in level 1 are the mean values of two consecutive numbers in the original signal, shown in row 1, according to equation (2.79), whereas the last 4 numbers are the differences according to equation (2.78). This continues on level 2, where the first two numbers are the mean values of the means in the first level of decomposition, and the next two are the differences. Finally we end up with one mean

$x[0]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$x[6]$	$x[7]$	
24	8	18	36	18	18	12	42	**Original**
16	27	18	27	**-8**	**9**	**0**	**-15**	**Level 1**
21.5	22.5	**5.5**	**4.5**	**-8**	**9**	**0**	**-15**	**Level 2**
22	**-0.5**	**5.5**	**4.5**	**-8**	**9**	**0**	**-15**	**Level 3**

Table 2.4. Lifting performed on 8 numbers. The bold font numbers are differences whereas the normal font numbers are mean values. These values are obtained using equations (2.78) and (2.79).

value at level 3, shown in the last row. Notice that the entire table can be rebuild from the mean value and the differences from the third level of decomposition using equations (2.76) and (2.77).

2.6 Lifting equations

The Haar wavelet explained in section 2.5 is the easiest to derive the lifting equations for.

In general, the prediction steps consist of the difference between an odd sample and the neighboring even samples. The z-transform representation of an input signal x is given by [7]:

$$X(z) = X_{\mathrm{e}}(z^2) + z^{-1} \cdot X_{\mathrm{o}}(z^2) \tag{2.80}$$

where

$$X_{\mathrm{o}}(z) = \sum_n x[2n] \cdot z^{-n} \tag{2.81}$$

$$X_{\mathrm{e}}(z) = \sum_n x[2n+1] \cdot z^{-n}. \tag{2.82}$$

Here $X_{\mathrm{e}}(z)$ are the even x-values and $X_{\mathrm{o}}(z)$ the odd. Equation (2.80) shows, that the original signal x can be obtained by up-sampling the even and odd components by a factor 2, illustrated by z^2, and shift the odd x, $X_{\mathrm{o}}(z)$, time-step to the right by multiplication with z^{-1} [7].

The prediction step computes a linear combination of two even x and subtracts it from an odd x. This combination is only based on the relative location of the even and odd xs under consideration. This is general for all wavelets. Biorthogonal wavelets are always symmetric, so the even sample x_{e} to the left of the sample x_{o}, is multiplied with the same factor, as the x_{e} to the right. [7] In lifting steps, we only want to use the samples, located next to the sample under consideration. The polynomials are thus in z and z^{-1}, also called Laurent polynomials [7]. This gives us the following prediction in the z-domain [7]:

$$d(z) = X_{\mathrm{o}}(z) - T(z) \cdot X_{\mathrm{e}}(z) \tag{2.83}$$

where $T(z)$ is a first order Laurent polynomial. In matrix form the prediction step can be written as:

$$\mathbf{P}_1(z) = \begin{bmatrix} 1 & 0 \\ -T(z) & 1 \end{bmatrix}. \tag{2.84}$$

This gives the relation:

$$\begin{bmatrix} X_{\mathrm{e}}(z) \\ X_{\mathrm{o}}(z) - T(z) \cdot X_{\mathrm{e}}(z) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -T(z) & 1 \end{bmatrix} \begin{bmatrix} X_{\mathrm{e}}(z) \\ X_{\mathrm{o}}(z) \end{bmatrix}. \tag{2.85}$$

The update step can be obtained in a similar way with the matrix:

$$\mathbf{U}_1(z) = \begin{bmatrix} 1 & S_1(z) \\ 0 & 1 \end{bmatrix} \tag{2.86}$$

where $S_1(z)$ also is a Laurent polynomial [7]. In this way, the entire filter can be decomposed into predict and update steps, based on Laurent polynomials containing z and z^{-1}:

$$\begin{aligned} H(z) &= \begin{bmatrix} H_{00}(z) & H_{10}(z) \\ H_{10}(z) & H_{11}(z) \end{bmatrix} \\ &= \begin{bmatrix} K & 0 \\ 0 & K^{-1} \end{bmatrix} \begin{bmatrix} 1 & S_{\mathrm{N}}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -T_{\mathrm{N}}(z) & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & S_1(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -T_1(z) & 1 \end{bmatrix} \end{aligned} \tag{2.87}$$

where $H_{00}(z)$ and $H_{01}(z)$ represent the even and odd parts of the low-pass decomposition filter and $H_{10}(z)$ and $H_{11}(z)$ the high pass filter.

This means, that the filter $H(z)$ can be obtained by running N prediction and update steps, followed by a scaling step, using the scaling factor K to normalize the filter [7]. The reconstruction is made by reverting the lifting steps.

To obtain the lifting steps $T_n(z)$ and $S_n(z)$, we need to find common dividers, relating the even and odd coefficients of the low-pass analysis filter $H_{00}(z)$ and $H_{01}(z)$, respectively. These can be used to reconstruct both the low- and high-pass filters $H_0(z)$ and $H_1(z)$. This is done using partial fraction division.

We define the approximation filter from the CDF 9/7 as:

$$H_0(z) = h_4 \cdot \left(z^{-4} + z^4\right) + h_3 \cdot \left(z^{-3} + z^3\right) + h_2 \cdot \left(z^{-2} + z^2\right) + h_1 \cdot \left(z^{-1} + z\right) + h_0. \tag{2.88}$$

To obtain the lifting steps, we have to factorize the filter equation. To do this, we start by splitting $H_0(z)$ in even and odd parts:

$$H_{00}(z) = h_4 \cdot \left(z^{-2} + z^2\right) + h_2 \cdot \left(z^{-1} + z\right) + h_0 \tag{2.89}$$

$$H_{01}(z) = h_3 \cdot \left(z^{-2} + z\right) + h_1 \cdot \left(z^{-1} + 1\right) \tag{2.90}$$

where $H_{00}(z)$ and $H_{01}(z)$ are related as:

$$H_0(z) = H_{00}(z^2) + z \cdot H_{01}(z^2). \tag{2.91}$$

Let:

$$a_0(z) = H_{00}(z) \tag{2.92}$$
$$b_0(z) = H_{01}(z). \tag{2.93}$$

Now we have to find a divider $q_1(z)$ which results in $r_1(z)$ of one power less than b_0:

$$r_1(z) = a_0(z) - b_0(z) \cdot q_1(z). \tag{2.94}$$

Here we choose $q_1(z)$ on the form $\alpha \cdot (z+1)$, where $\alpha = h_4/h_3$ so the polynomial terms z^{-2} and z^2 of $a_0(z)$ are canceled out:

$$q_1(z) = \frac{h_4}{h_3} \cdot (z+1) \tag{2.95}$$
$$\begin{aligned} r_1(z) &= a_0(z) - b_0(z) \cdot \frac{h_4}{h_3} \cdot (z+1) \\ &= -\frac{-h_2h_3 + h_4h_3 + h_4h_1}{h_3} \cdot z^{-1} + \frac{-2\,h_4h_1 + h_0h_3}{h_3} - \frac{(-h_2h_3 + h_4h_3 + h_4h_1)}{h_3} \cdot z. \end{aligned} \tag{2.96}$$

For the next division we assign:

$$a_1(z) = b_0(z) \tag{2.97}$$
$$b_1(z) = r_1(z) \tag{2.98}$$

and find a new divider $q_2(z)$ and remainder $r_2(z)$. To remove the first and last parts of $a_1(z)$, $q_2(z)$ is on the form $\beta \cdot (z^{-1}+1)$:

$$\beta = -\frac{{h_3}^2}{-h_2h_3 + h_4h_3 + h_4h_1} \tag{2.99}$$
$$q_2(z) = \beta \cdot \left(z^{-1} + 1\right) \tag{2.100}$$

giving the remainder:

$$r_2(z) = a_1(z) - b_1(z) \cdot q_2(z) \tag{2.101}$$
$$\begin{aligned} &= \frac{-2\,h_1h_4h_3 + h_2{h_3}^2 + h_0{h_3}^2 - h_4{h_3}^2 - h_1h_2h_3 + h_4{h_1}^2}{-h_2h_3 + h_4h_3 + h_4h_1} \\ &\quad + \frac{\left(-2h_1h_4h_3 + h_2{h_3}^2 + h_0{h_3}^2 - h_4{h_3}^2 - h_1h_2h_3 + h_4{h_1}^2\right)}{-h_2h_3 + h_4h_3 + h_4h_1} \cdot z. \end{aligned} \tag{2.102}$$

For the third step, $a_2(z)$ and $b_2(z)$ are assigned, and $q_3(z)$ is calculated on the form $\sigma \cdot (z+1)$, where:

$$\sigma = -\frac{(-h_2h_3 + h_4h_3 + h_4h_1)^2}{h_3\left(-2h_1h_4h_3 + h_2{h_3}^2 + h_0{h_3}^2 - h_4{h_3}^2 - h_1h_2h_3 + h_4{h_1}^2\right)} \tag{2.103}$$

such that:

$$q_3(z) = \sigma \cdot (z+1)\,. \tag{2.104}$$

This results in the following remainder:

$$r_3 = a_2(z) - b_2(z) \cdot q_3(z) = h_0 - 2\,h_2 + 2\,h_4. \tag{2.105}$$

For the last step, $q_4(z)$ is on the form $\delta \cdot (z+1)$ with:

$$\delta = \frac{\left(-2\,h_1 h_4 h_3 + h_2 {h_3}^2 + h_0 {h_3}^2 - h_4 {h_3}^2 - h_1 h_2 h_3 + h_4 {h_1}^2\right)(z+1)}{\left(-h_2 h_3 + h_4 h_3 + h_4 h_1\right)\left(h_0 - 2\,h_2 + 2\,h_4\right)} \tag{2.106}$$

giving:

$$q_4(z) = \delta \cdot \left(z^{-1} + 1\right). \tag{2.107}$$

This results in:

$$r_4 = a_3(z) - b_3(z) \cdot q_4(z) = 0 \tag{2.108}$$

finalizing the division with $a_4(z)$:

$$a_4(z) = b_3(z) = r_3 = h_0 - 2\,h_2 + 2\,h_4. \tag{2.109}$$

The dividers can be put into a generalized matrix form::

$$\begin{bmatrix} H_{00}(z) & H_{10}(z) \\ H_{01}(z) & H_{11}(z) \end{bmatrix} = \begin{bmatrix} K & 0 \\ 0 & K^{-1} \end{bmatrix} \prod_{n=N/2}^{1} \begin{bmatrix} 1 & q_{2n}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ q_{2n-1}(z) & 1 \end{bmatrix}. \tag{2.110}$$

where K is set to $a_4(z)$.

Written out, the matrix for the CDF 9/7 results in:

$$\begin{bmatrix} H_{00}(z) & H_{01}(z) \\ H_{10}(z) & H_{11}(z) \end{bmatrix} = \begin{bmatrix} K & 0 \\ 0 & K^{-1} \end{bmatrix} \begin{bmatrix} 1 & \delta \cdot \left(z^{-1}+1\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sigma \cdot (z+1) & 1 \end{bmatrix} \begin{bmatrix} 1 & \beta \cdot \left(z^{-1}+1\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \alpha \cdot (z+1) & 1 \end{bmatrix}. \tag{2.111}$$

We have found the low-pass filter parts H_{00} and H_{01}, but we still need to find $H_{10}(z)$ and $H_{11}(z)$, representing the high-pass filter of the CDF 9/7 wavelet. These have the following relation:

$$H_1(z) = H_{10}(z^2) + z \cdot H_{11}(z^2) \tag{2.112}$$

The high-pass filter is found calculating the matrix on the right hand side of (2.111), and check the difference between these and the even and odd parts of the original filter

The lifting coefficients can be obtained using the filter coefficients for the CDF 9/7. These are found in table 2.1 derived in section 2.4.

To verify if the decomposition is correct, the matrices can be multiplied together and the coefficients α, β, σ and δ, which can be found in table 2.5, are inserted. This results in the even and odd parts of the high- and low-pass filter:

$$\begin{bmatrix} H_{00}(z) & H_{01}(z) \\ H_{10}(z) & H_{11}(z) \end{bmatrix} = \begin{bmatrix} 0.6029 - 0.0782(z^{-1}+z) + 0.0267(z^{-2}+z^2) & 0.2669(z^{-1}+1) - 0.0169(z^{-2}+z) \\ -0.5913(1+z) + 0.0912(z^{-1}+z^2) & 1.1151 - 0.0575(z^{-1}+z) \end{bmatrix}. \tag{2.113}$$

These coefficients are not scaled with $\sqrt{2}$, as the values in table 2.1 and 2.2 from chapter 2 on page 7. The original filters can be obtained using equation (2.91).

The lifting equations can be obtained by reading the equations from last matrix on the right hand side of the matrix in equation (2.111) and move to the left. The steps are obtained by performing the inverse Z-transform on these. $s_0[n]$ are the even coefficients form the signal, whereas $d_0[n]$ are the odd. This results in:

$$d_0[n] = S[2n+1] \tag{2.114}$$
$$s_0[n] = S[2n] \tag{2.115}$$
$$d_1[n] = d_0[n] + \alpha \cdot (s_0[n] + s_0[n+1]) \tag{2.116}$$
$$s_1[n] = s_0[n] + \beta \cdot (d_1[n-1] + d_1[n]) \tag{2.117}$$
$$d_2[n] = d_1[n] + \sigma \cdot (s_1[n] + s_1[n+1]) \tag{2.118}$$
$$s_2[n] = s_1[n] + \delta \cdot (d_2[n-1] + d_2[n]) \tag{2.119}$$

Where S is the input signal. The final scaling is given by:

$$s = K \cdot s_2 \tag{2.120}$$
$$d = K^{-1} \cdot d_2. \tag{2.121}$$

The block diagram for the CDF 9/7 lifting algorithm is shown in figure 2.17.

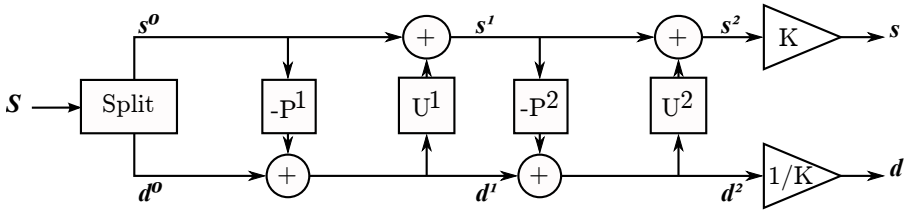


Figure 2.17. Block diagram of the CDF 9/7 lifting implementation.

The lifting coefficients are shown in table 2.5.

The reverse lifting equations, needed to calculate the reconstruction, can be obtained by reversing equations (2.116) to (2.121) and execute them in reverse order. these are shown here:

α	-1.586134342
β	-0.05298011857
σ	0.8829110758
δ	0.4435068519
K	0.8128930661

Table 2.5. The coefficients for the CDF 9/7 lifting equations

$$s_2 = K^{-1} \cdot s \tag{2.122}$$
$$d_2 = K \cdot d \tag{2.123}$$
$$s_1[n] = s_2[n] - \delta \cdot (d_2[n-1] + d_2[n]) \tag{2.124}$$
$$d_1[n] = d_2[n] - \sigma \cdot (s_1[n] + s_1[n+1]) \tag{2.125}$$
$$s_0[n] = s_1[n] - \beta \cdot (d_1[n-1] + d_1[n]) \tag{2.126}$$
$$d_0[n] = d_1[n] - \alpha \cdot (s_0[n] + s_0[n+1]) \tag{2.127}$$
$$S[2n] = s_0[n] \tag{2.128}$$
$$S[2n+1] = d_0[n]. \tag{2.129}$$

2.7 Boundary effects

When working with a finite signal it is necessary to consider how the borders are handled. Many different methods exists, and an overview of some possible solutions is given below.

1. **Zero padding:** All value outside the finite signal are set to zero.
2. **Value padding:** All values from the border in each direction are set to the end value at that border.
3. **Decay padding:** The values next to the borders are decaying in amplitude as they move away from the finite signal.
4. **Periodization:** Assuming that the signal is periodic, like e.g. a sinusoid, the finite signal is just repeated again at the borders.
5. **Reflection:** At the borders the finite signal is reflected or mirrored.
6. **Smoothing window:** A window with a concave shape is used to scale the signal values. This causes the amplitude to decay towards the borders, and end in zero at the borders.
7. **Polynomial fitting:** The values beyond the boders are found by polynomial extrapolation.
8. **Boundary filters:** In this method the used filter is substituted with a shorter one each time it exceeds the boundary.

An issue when handeling the borders is that this can result in false frequency content at the borders. To obtain a high image quality for the wavelet transform it would be

necessary to analyse and test these possibilities, and based on this choose the method best suited. This is out of the scope of this report. Instead the pros and cons of the methods are shortly discussed and one is chosen.

The zero padding, periodization and reflection are easy to perform in an implementation, as they do not require analysis of the signal. A problem is that they might result in high frequency content at the borders. Here the value padding and decay padding should give a more smooth result, but require that the signal is used to "generate" the padding values. This is also the case for the polynomial fitting, which might be even more comprehensive. Using a smoothing window also require processing of the signal, and manipulates the amplitude of the data, which might result in numerical inaccuracy. Boundary filters reduces the corruption of the signal, but requires a more advanced filter handling.

It is chosen that the method used for boundary handling will be periodization.

Summary

In this section the wavelet transform is investigated. The basic principles of performing the transform is explained by use of the CWT. The DWT is presented, including the the dyadic grid, orthogonality and transform and approximation coefficients. The later two are used for the multiresolution decomposition. The DWT can also be seeing as filtering and downsampling a signal. The 2D DWT used for images is presented, including how this is performed by matrix multiplications.

The Daubechies wavelets, their lowpass requirements, and their ability to suppress high order olynomial content within a signal is explained. Symmetric filters are desired when transforming images, as these results in better compression and humans are more tolerant of symmetric errors. As the short Haar wavelet is the only finite-length, symmetric, orthogonal, lowpass filter, biorthogonal filters are instead a possibility. These use the biorthogonal filter pairs $\mathbf{h}$, $\tilde{\mathbf{h}}$ and $\mathbf{g}$, $\tilde{\mathbf{g}}$. Biorthogonal spline filter pairs, can be derived using the Fourier transform, but these are often very different in length and pole balancing, resulting in bad approximations. The CDF 9/7 filter on the other hand have balanced poles and is therefore used by JPEG2000 for lossy compression. The theory of performing the filtering in lifting steps is explained, and the lifting equations for the CDF 9/7 are derived.

Finally possible solutions to handle boundaries of finite signals are considered, and it is chosen to use periodization.

3 Platform architecture

When rendering signals in near real time a lot of processing power may be needed, especially for video. Certain operations like wavelet transforms, require many calculations that can be performed in parallel. In the past few years, the Graphics Processing Unit (GPU) has gained much popularity in the scientific computing sector. This is due to the superior computing performance a GPU can provide compared to a Central Processing Unit (CPU). In this chapter the main differences between the CPU and GPU are investigated as well as their strengths and weaknesses. After this, there will be a short introduction to the GPU architecture explained in OpenCL terminology. Finally in this chapter, some of the important features of NVIDIAs CUDA compute capabilities are listed.

3.1 Basic architecture

The architecture of a device (GPU) and a host (CPU), is shown in figure 3.1. Today, modern host have relatively few cores compared to device. Where a CPU has up to 16 cores[1], a GPU has up to 512 cores[2]. The arithmetic cores in the GPU are very different compared to CPU cores as these have much simpler control structures (CTL on figure 3.1) and an smaller instruction set. The figure also shows, that the ALUs on the GPU are gathered in clusters or cores with a control unit for each core. The individual cores on today's GPUs are generally slower than CPU cores, as the latter ones are running around 3 GHz where the modern GPUs run up to 1 GHz. When solving arithmetic tasks, that can be executed in parallel, the GPU is able to outperform the CPU because of the large number of arithmetic units. but the performance reduces drastically when computing

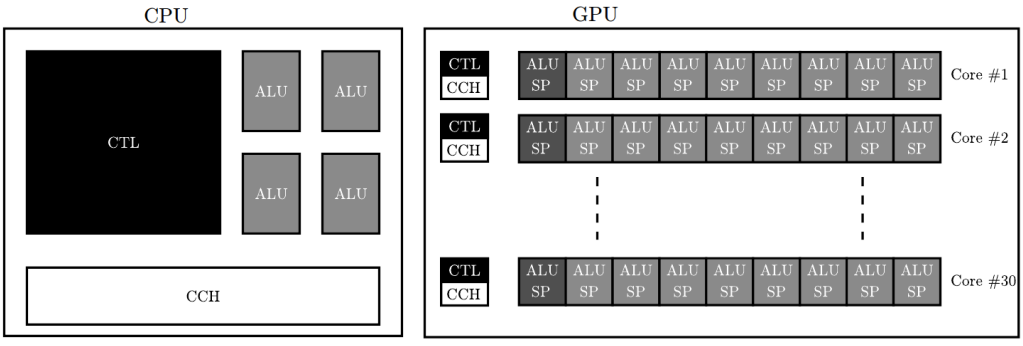


Figure 3.1. Comparison of CPU and GPU architecture concerning arithmetic and control units, from [9]. In the GPU, the cores are gathered in small clusters, each with one CTL unit, whereas the CPU only has one CTL for all cores.

[1] http://www.amd.com/us/products/server/processors/6000-series-platfomr/6200/Pages/6200-series-processors.aspx

[2] http://www.geforce.com/Hardware/gpuS/geforce-gtx-580/specifications

conditional statements due to the architecture of a GPU [9].

The device is dependent on the host to receive instructions and data. The device is unable to run an operating system, and can thereby not run without a host. This introduces other performance related issues, since the data has to be transferred from the host memory to the device memory, which is done through a PCI bus. The data path between the host and device is illustrated in figure 3.2. Here it is seen, that the PCI-e 1.0 bus only operates at 8 GB/s which is much slower than any other of the internal buses shown, newer versions are operating faster. In addition to the lower bandwidth, there is also a delay of some 100 µs[9] when initializing a transfer on the PCI bus. Because of this, it has to be considered carefully, what data has to be transferred to and from the device, and if it pays of, compared to the reduction in calculation time.

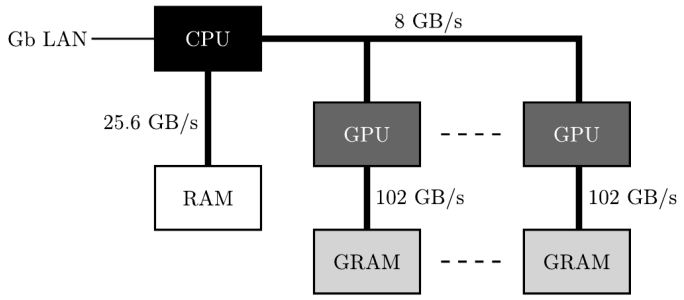


Figure 3.2. The data-path between the CPU and GPU [9]

Another limitation of the GPUs is the amount of memory available. The CPUs in modern workstations easily can support more than 200 GB of memory[3], where the best professional GPUs cards (NVIDIA Tesla C2075) have up to 6 GB[4] of memory. Consumer grade cards (NVIDIA Geforce GTX580) have up to 1.5 GB[5]. This means, that a GTX580 GPU can hold a matrix of

$$\frac{1.5 \cdot 1024^3}{4} \approx 402.6 \cdot 10^6 \tag{3.1}$$

single precision floating point numbers, which is a relatively small size in scientific computing terms, since both the operands and results should fit in the memory at once.

3.2 Limitations

The major limitation of GPUs is, as mentioned before, the long data-path between the CPU and GPU as illustrated in figure 3.2.

When considering the numerical precision of single-, double- or quadruple-precision floating-point numbers, there are different things to take into account. The computational performance drops as the precision increases [9]. Furthermore, many consumer-grade GPUs are limited to single-precision floating-point numbers. This is not considered

[3] http://www.colfax-intl.com/jlrid/SpotLight_more_Acc.asp?L=80&S=8&B=2207
[4] http://www.nvidia.com/object/workstation-solutions-tesla.html
[5] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications

an issue for graphics, as the difference between the true and computer represented number often is insignificant compared to the difference humans can perceive. The precision become a problem when doing critical numerical calculations, especially when the difference between the operands become close to the limit of the machine precision [9]. This of course also depends on the precision wanted in the calculations. Table 3.1 shows the precision for the two floating-point numbers supported by GPUs.

Format	Name	Precision
Binary32	Single-precision	$5.96 \cdot 10^{-8}$
Binary64	Double-precision	$1.11 \cdot 10^{-16}$

Table 3.1. The precision of the floating-point numbers supported by GPUs [9].

When implementing algorithms, it has to be checked, whether the machine precision is sufficient for the desired precision of the outcomes.

The professional GPU cards have Error Correcting Code (ECC) memory, that detects and corrects bit-errors. In scientific computing this can be necessary, since bit-errors can corrupt the obtained results. General purpose GPUs do not feature ECC memory, and are thereby not protected against bit-errors. This can become a problem if it is of high priority to obtain correct results. In an application such as video coding, temporal incorrect results can normally be accepted.

3.3 Actual and theoretical performance

The theoretical matrix multiplication performance for a CPU and GPU is compared in Figure 3.3. The data is obtained by simulating the transfer through the PCI-bus where the up-link is set to 4 GB/s and the down-link to 2.5 GB/s with a delay of 150 µs on the bus. The CPU has a theoretical performance of 109 GFLOPS[6] and the GPU is set to 1030 GFLOPS (single precision)[7] with an efficiency of 50 %[9]. This is due to the fact, that a GPU never reaches its theoretical maximum performance [9]. It is clear, that operations on a matrix larger than 500×500 elements performs significantly faster on a GPU than a CPU.

Figure 3.4 shows the actual performance of a GPU compared to a CPU in MATLAB, where the GPU is accessed through the Jacket[8] interface. The test is performed by multiplying two square single-precision matrices [10], using a Intel Core i7 x975 CPU and a Tesla C2050 GPU. When observing figure 3.4, it should be noted that the results of the GPU are measured when the data is already present on the GPU prior to measurement. This favors the GPU since no data has to be transferred to and from the GPU.

[6] http://realworldtech.com/page.cfm?ArticleID=RWT040511235825&p=2

[7] http://www.geforce.com/Hardware/GPUs/geforce-gtx-580/specifications

[8] Jacket optimizes the code to execute in parallel on CPU and GPU systems. See more at http://www.accelereyes.com/

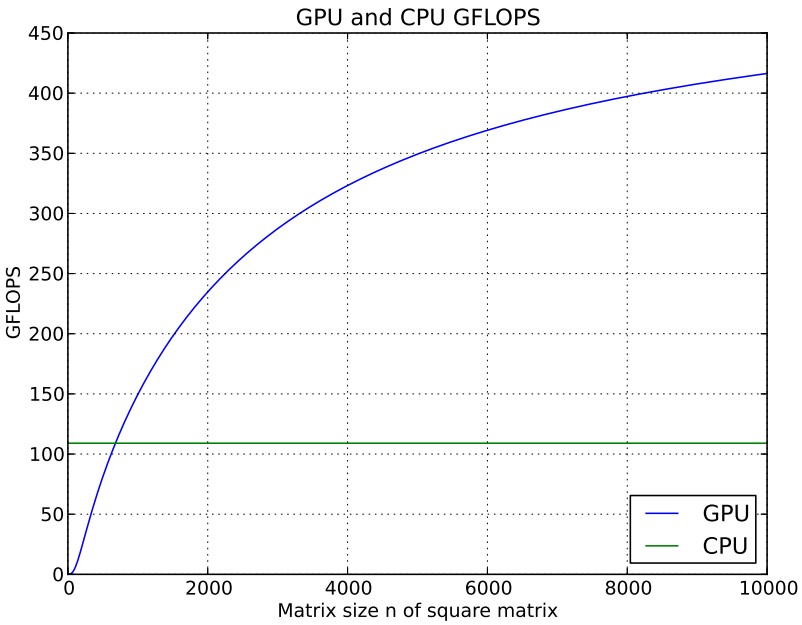


Figure 3.3. Theoretical floating-point performance of a CPU and GPU for different square matrix sizes.
Code: **gpuflops.py**.

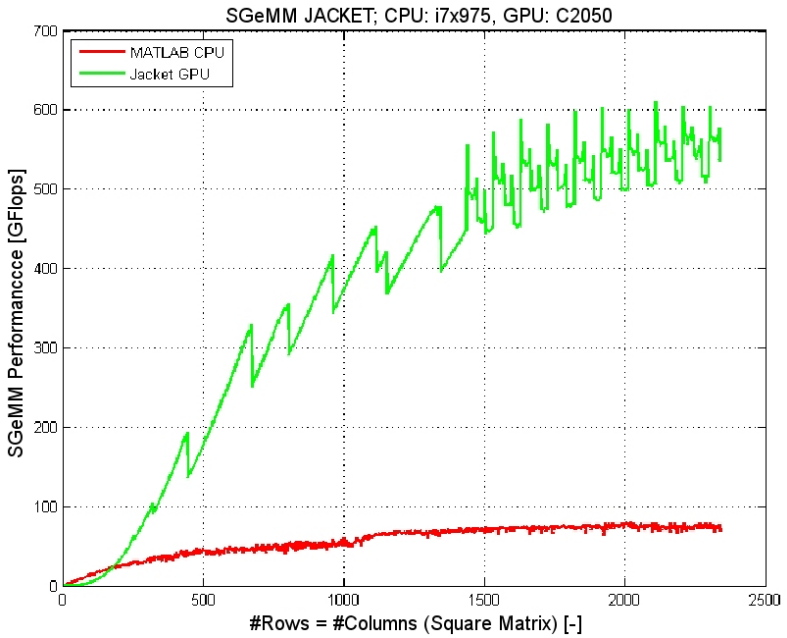


Figure 3.4. Actual GPU performance through Jacket compared to CPU-performance from [10].

Figures 3.3 and 3.4 show that the GPU outperforms a modern CPU in floating-point arithmetic with larger matrix sizes. When considering video-encoding using a wavelet transform (See chapter 2), the performance could be enhanced by using the GPU to perform the "heavy" parallel arithmetic part of the algorithm.

Finally to summarize, a comparison between the CPU and GPU is shown in table 3.2. The GPU cannot be used without a CPU, since the GPU is unable to run an operating

Feature	CPU	GPU
Usage	Versatile	Specialized
Cache memory	Large (MB)	Small (kB)
Global memory	Large (TB)	Small (GB)
Shared memory	Fast (QPI)	Very slow (PCI-e)
Instruction set	Complex	Simple/Reduced
Floating-point performance	Mediocre	Extreme

Table 3.2. Table with the major differences between CPU and GPU [10]

system, and needs to be provided with data from the CPU and its memory. This gives the advantage to execute heavy sequential parts of the algorithm on the CPU while executing the parallel parts on the GPU.

3.4 GPU architecture

To day the are two major programming languages for interfacing with the GPU, these are OpenCL [9] and CUDA [10]. CUDA is invented by NVIDIA and is a parallel computing platform and programming model. OpenCL is a standardized cross-platform Aplication Programming Interface (API), developed specifically for parallel computing, supports multi platforms and has vendor portability, which makes OpenCL useful for a wide variety of applications. It is chosen to implement in OpenCL and we have chosen to focus on explaining the GPU architecture in respect to OpenCL terminology. The explanation of the architecture will therefore use abbreviations and terminology specific to OpenCL.

Architecture

The overall architecture for a CUDA-able (by standards also OpenCL-able) device is shown on figure 3.5. This illustrate the organization of 16 Compute Units (CUs) [11] and with 8 Processing Elements (PEs) [12] in each CU, and two CUs form a Texture Processor Cluster (TPC) or block (CUDA terminology), see table 3.4. The PEs have shared logic and instruction cache. Both PEs and CUs can access the global memory on the device. The number of PEs in a CU and the number of blocks varies for each CUDA generation

[9] http://www.khronos.org/opencl/
[10] `http://www.nvidia.com/object/cuda_home_new.html`
[11] streaming multiprocessor in CUDA terminology
[12] streaming processors or CUDA cores

and graphic card [11]. Each PE has a multiply-add (MAD), a multiplier plus additional special-function units, for performing other floating point operations, such as square root. The PEs can be massively threaded, which essentially means that an application can run thousands of threads [11]. For reference does the geforce G80 chip support up to 768 threads per CU, and with 16 CUs and 8 PEs per CU, this adds up to 12.000 threads. For the GT200 chips with 1024 threads per CU and with 30 CUs, this adds up to 30000 threads [11]. GPU threads differ from CPU threads, due to efficient hardware support and take very few cycles to generate and schedule [11].

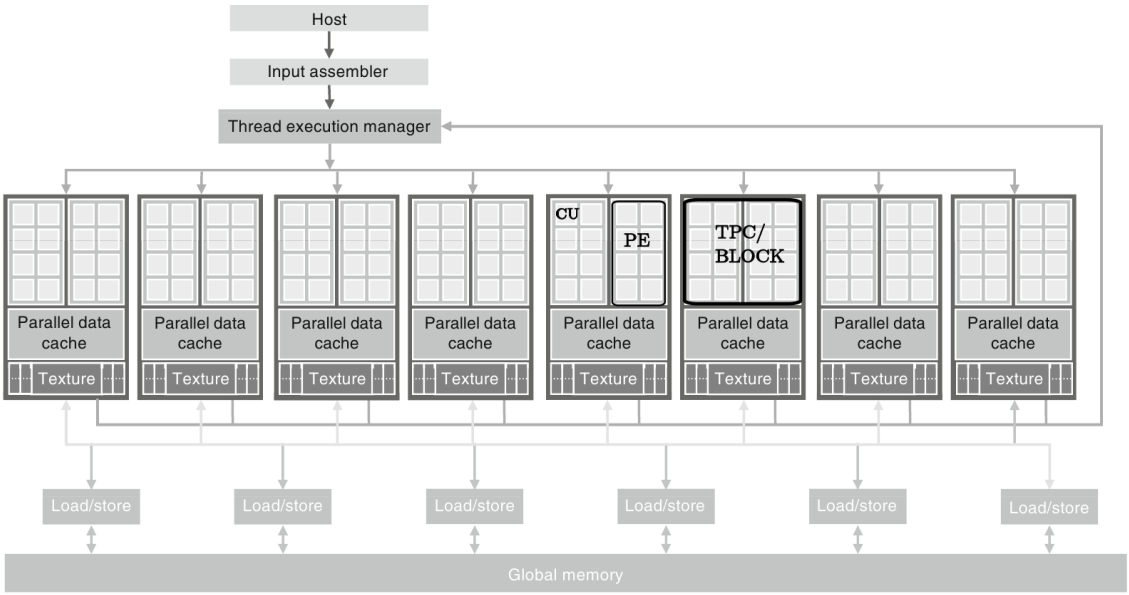


Figure 3.5. Overview of the architecture of a CUDA device [11]. On this figure 8 blocks (TPCs) are shown, where each has 2 CUs. Each CU has 8 PEs. These blocks are connected to the global device memory.

Parallel execution model

The host device (CPU), controls and initiate kernels executions on the GPU-device. Kernels are a abbreviation for small programs capable of running on the GPU. When a kernel is invoked, the GPU generates a grid of threads (CUDA) also know as work items (OpenCL), according to defined settings. The grid is terminated when the kernel execution is done [11]. Dimensions of this grid are application dependent, an is set as an argument when the kernel is executed. The work items are organized into groups known as work groups (blocks in CUDA). The work items in the groups can be organized in either 2 or 3-dimensional array with sizes up to 1024 threads [11] per group. Each work groups is assigned to one CU, with a maximum of 8 or 16 active work groups per CU depending on the device, see table 5.2. The work groups are aligned quadratically or rectangular across the whole grid as shown in figure 3.6. In OpenCL this grid is also referred to as NDRange, but for convenience it is chosen to use grid.

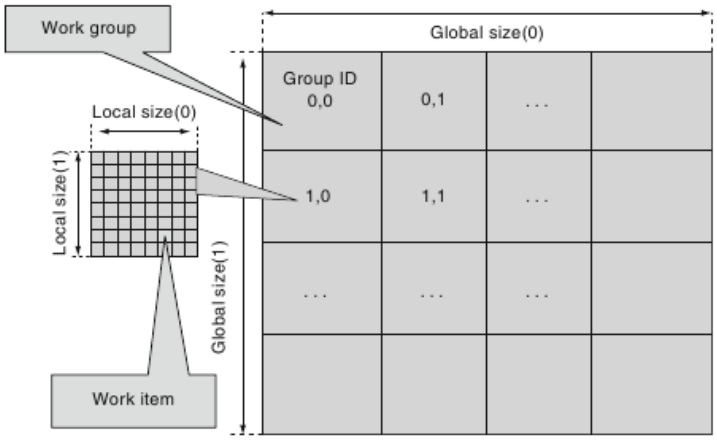


Figure 3.6. Overview of the OpenCL parallel execution model taken from [11]. This shows a 2 dimensional grid, and illustrate the alignment of work groups, how these groups are indexed.

Memory access

Besides the global GRAM on the device, PEs, CUs and blocks can access different memory areas. The memory model for a OpenCL device is shown on figure 3.7. The device has five different memory types which have different accessibility [11]. These are shown in table 3.3.

Memory Type	Access	Permission	Latency	Cached	Location on/off chip	Lifetime
Private	Work item	R/W	Fast	n/a	On	Thread
Local	Work group	R/W	Fast	No	On	Group
Global	Grid + Host	R/W	Slow	Some GPU's	Off	Host allocation
Constant	Grid + Host	R (W host)	Slow	Yes	Off	Host allocation
Texture	Grid + Host	R (W host)	Slow	Yes	Off	Host allocation

Table 3.3. Listed are the different memory types available, their accessibility, device read/write properties, and latencies. This is taken from [11]. Grid does in this cases mean that all work items in the grid can access it.

As listed in table 3.3 and seen on figure 3.7 can these memory types can be accessed by/on different levels internally on the device. Global and constant memory can both be allocated dynamically and read and written to by host, but where global memory support read/write access by the device, constant memory only support read operation. These memory types can be accessed by all work items and groups in the grid. Local memory can be dynamically allocated by the host and statically allocate in code by the device. This memory can, however, not be accessed by host, but is shared among all work items in a work group. Private memory and registers are only accessible by individual work

items. Texture memory will not be used or explained further in this report.

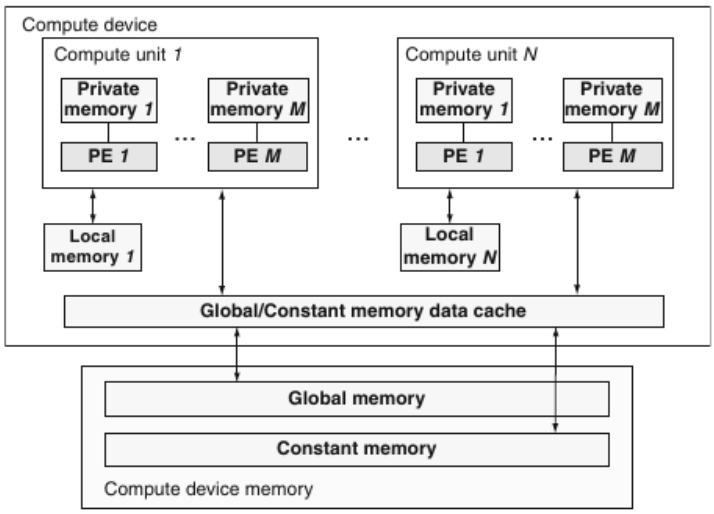


Figure 3.7. Overview of the conceptual OpenCL device architecture model from [11].

When the image is transfered to the device, the images is located linear and row wise in memory, as illustrated on figure 3.8. The three color layer pixels are located adjacent in memory. Accessing the memory is relative to the starting position (0,0,0) and each step in the Y-direction is calculated as the length of row.

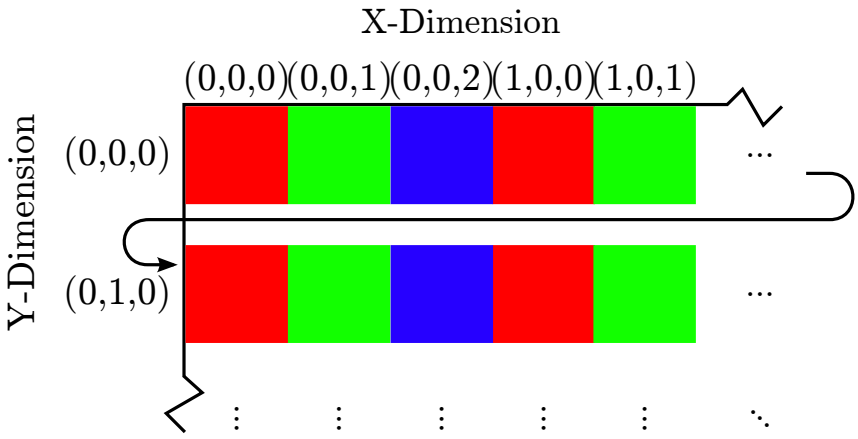


Figure 3.8. Illustrative memory layout on the GPU for a RGB image.

All work items have unique global index values, inside a kernel these can be returned by use of the API function `get_global_id(D)`, where D can be 0 for x-dimension, 1 for y-dimension, and if working on a 3-dimensional grid, 2 is used for the z-dimension. A work item is found using `get_local_id()` and the work group id is found with `get_group_id()`. These are accessed with similar to global id. A global id of a work item, can also be calculated from the relative position in the grid using `get_group_id()` and `get_local_id()`.

3.4.1 Compute capabilities

The number of work items in a group are application specific, but also limited by hardware. Memory, cache and register sizes are somewhat application specific but vary depending on the used hardware used. This should, therefore, be further investigated when the programming/testing platform is defined. CUDA does, however, have standardized specifications for different CUDA compute capabilities which can be mapped to OpenCL. Some of the features worth mentioning are found in table 3.4 [12]. This is a small cut from a long list of features in the specification.

Specifications \ Compute capability Version	1,1	3.0
Maximum dimensionality of grid of thread blocks	2	3
Maximum x-dimensions of a grid of thread blocks	65535	$2^{31} - 1$
Maximum Y- or Z-dimensions of a grid of thread blocks	65535	65535
Maximum x- or y-dimension of a block	512	1024
Maximum z-dimension of a block	64	64
Maximum number of threads per block	512	1024
Amount of local/shared memory per thread (KB)	16	512
Maximum number of resident blocks per multiprocessor	8	16
Maximum amount of shared memory per multiprocessor (KB)	16	48
Number of 32-bit registers per multiprocessor	8 K	64 K
Constant memory size (KB)	64	64
Support Double-precision floating-point numbers	No	Yes

Table 3.4. Specification on CUDA's compute capabilities found at [12].

Summary

Throughout this chapter, the main differences between CPUs and GPUs have been listed and discussed. It is seen that the GPU is powerful and have fast Floating Point Operations (FLOPs). The computational power of the GPU is around 4-5 times that of the CPU. The theoretical performance of the NVIDIAs Tesla C2050 GPU is 1030 GFLOPS compared to 103 GFLOPS for the Intel Core i7 x975 CPU. In practices the GPU performance is lower, around 50 % of the theoretical, due to both data transfers and internal control logic. This is a major drawback on the GPU, but the computational power still out-performs the CPU many times. Minimizing data transfers between the CPU and GPU is beneficial as this is the overall bottleneck when optimizing for speed. In terms of architecture, the GPU is optimized to run Single Instruction Multiple Data (SIMD), and in general parallel execution of operation on large amounts of data.

4 Algorithm analysis

Different algorithms and schemes exist for calculating a 2D DWT. There are three main methods [13]: Row-Column (RC), Line-Based (LB) and Block-Based (BB). These are all investigated and in the end a comparison is made. Those found best suited for GPU computing are chosen for the development process.

In the following section the procedure of each method is described. To get an idea of how computational demanding the algorithms are, the number of FLOPs needed for doing a one level decomposition is estimated when both filtering and lifting are used for the transform. The block based method is only estimated for filtering, since lifting does not fit this approach. The memory requirements are also estimated. This is for a single thread implementation, and will therefore be the minimum requirement for an implementation utilizing parallelism.

4.1 Row-column

The Row-Column (RC) wavelet transform is a simple algorithm that applies a 1D DWT horizontal (row-wise) and vertical (column-wise) to produce a full 2D wavelet transformed image. This process is illustrated on figure 4.1.

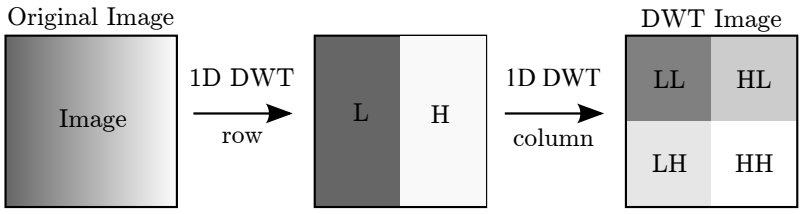


Figure 4.1. Graphical illustration of the RC DWT. A 1D DWT is applied on each row (row-wise) and then again on each column (column-wise), which produces one level of the 2D DWT decomposition. The 1D DWT contains both low pass and high pass filtering which are downsampled into half the image size.

Each row of the image is low pass and high pass filtered, and then downsampled. This divide the image into two pieces, a low pass (L) part to the left and a high pass (H) part to the right, as shown in figure 4.1. When all rows are filtered and downsampled, the procedure is repeated on the columns. After this the image consists of four pieces: LL, LH, HL and HH, which represent the respective filtered parts of the image. This is the one level 2D DWT decomposition.

4.1.1 Floating point operations

Due to the downsampling, half of the filter operations can be skipped, since these do not contribute to the result. Since operations are conducted on the GPU, it is assumed that

multiplications and additions both cost 1 FLOP. The amount of FLOPs for a row-wise filtering, with a low pass filter of 9 taps and high pass filter of 7 taps (CDF 9/7) and an image with dimensions $M \times N$ are

$$\begin{aligned} \mathrm{RC}_{\mathrm{row}} &= N \cdot \frac{1}{2} \cdot (9+8) + N \cdot \frac{1}{2} \cdot (7+6) && (4.1)\\ &= N \cdot 15. && (4.2) \end{aligned}$$

For each low and high pass filtering, the signal is multiplied with the filter coefficients (9 and 7 multiplications respectively) and these are added together (8 and 6 additions respectively) for one pixel value. Since the signal is downsampled, only half the operations are needed. Filtering all the rows the amount of operations are then

$$\mathrm{RC}_{\mathrm{rows}} = M \cdot \mathrm{RC}_{\mathrm{row}} = M \cdot N \cdot 15. \tag{4.3}$$

Applying the 1D DWT column-wise is very similar to row-wise, so the number of floating point operations are

$$\mathrm{RC}_{\mathrm{cols}} = N \cdot \mathrm{RC}_{\mathrm{col}} = N \cdot M \cdot 15. \tag{4.4}$$

The number of operations for a one level decomposition of one color layer is then

$$\begin{aligned} \mathrm{RC}_{\mathrm{layer}} &= \mathrm{RC}_{\mathrm{cols}} + \mathrm{RC}_{\mathrm{rows}} && (4.5)\\ &= (N \cdot M \cdot 15) + (M \cdot N \cdot 15) && (4.6)\\ &= M \cdot N \cdot 30 \text{ FLOP}. && (4.7) \end{aligned}$$

This corresponds to

$$\mathrm{RC}_{\mathrm{fpp}} = 30\,\mathrm{FLOP/pixel}. \tag{4.8}$$

The general formula for calculating the FLOPs needed to decompose an image in RGB colors (three layers) is then

$$\begin{aligned} \mathrm{RC}_{\mathrm{flop}}(N, M) &= 3 \cdot \mathrm{RC}_{\mathrm{layer}} && (4.9)\\ &= 3 \cdot N \cdot M \cdot 30 \text{ FLOP}. && (4.10) \end{aligned}$$

For a full HD image (1920×1080) this corresponds to

$$\mathrm{RC}_{\mathrm{flop}}(1080, 1920) = 3 \cdot 1080 \cdot 1920 \cdot 30 \text{ FLOP} \approx 187\,\mathrm{MFLOP}. \tag{4.11}$$

For the next level of the CDF 9/7 decomposition, the image dimensions are half of the original (960×540), which corresponds to

$$\mathrm{RC}_{\mathrm{flop}}(540, 960) = 3 \cdot 540 \cdot 960 \cdot 30 \text{ FLOP} \approx 47\,\mathrm{MFLOP} \tag{4.12}$$

This is a fourth of the needed FLOPs for the first level of the decomposition. Generally, for every sub-level, the needed FLOPs are a fourth of the previous.

Lifting

In this section, we consider the amount of computations needed to perform the lifting steps. In equations (2.116) to (2.119) in section 2.6 on page 27 it is seend, that every step performs two additions and one multiplication. Every operation is considered to require one FLOP. The amount of FLOPs needed to perform equation (2.116) is 3 for the two additions and one multiplication. The algorithm consists of four lifting steps and two scaling steps. This results in the following amount of FLOPs needed for each row:

$$\mathrm{RC}_{\mathrm{row}_{\mathrm{lifting}}} = N \cdot \frac{1}{2} \cdot (4 \cdot (2+1) + 2 \cdot 1) \tag{4.13}$$

$$= N \cdot 7 \text{ FLOP}. \tag{4.14}$$

Since both low- and high-pass parts are calculated at once due to downsampling, the number is multiplied with $1/2$. For one level of composition on rows and columns, the amount of FLOPs needed is

$$\mathrm{RC}_{\mathrm{layer}_{\mathrm{lifting}}} = (N \cdot M \cdot 7) + (M \cdot N \cdot 7) \tag{4.15}$$

$$= M \cdot N \cdot 14 \text{ FLOP} \tag{4.16}$$

corresponding to

$$RC_{\mathrm{fpp}_{\mathrm{lifting}}} = 14\,\mathrm{FLOP/pixel}. \tag{4.17}$$

For a color image, the amount of FLOPs needed is

$$\mathrm{RC}_{\mathrm{flop}_{\mathrm{lifting}}}(M, N) = 3 \cdot 14 \cdot M \cdot \mathrm{N} \tag{4.18}$$

$$= 42 \cdot M \cdot N \text{ FLOP}. \tag{4.19}$$

This results in

$$\mathrm{RC}_{\mathrm{flop}_{\mathrm{lifting}}}(1080, 1920) \approx 87\,\mathrm{MFLOP} \tag{4.20}$$

which is less than half of the FLOPs compared to a filter bank

4.1.2 Memory usage

Each 1D DWT result can be directly written into the original image, as soon as filtering and downsampling is completed. It is assumed that the image is always in three color layers (RGB) and all elements are represented in 32 bit floating point. The needed memory allocation is listed below.

- Original image ($M \times N \times 3$)
- High pass filter (7 taps)
- Low pass filter (9 taps)

The memory requirements for the image is

$$\begin{aligned} \text{IMG} &= (3 \cdot N \cdot M) \cdot 4\,\text{B} \\ &= M \cdot N \cdot 12\,\text{B} \end{aligned} \tag{4.21}$$

or 12 B/pixel. For a full color HD image of dimensions $1920 \times 1080 \times 3$, the memory requirement is

$$\text{IMG} = 1920 \cdot 1080 \cdot 12 \approx 25\,\text{MB}. \tag{4.22}$$

The memory needed to store the coefficients is

$$\text{RC}_{\text{filter}} = (7 + 9) \cdot 4\,\text{B} = 64\,\text{B}. \tag{4.23}$$

The resulting total memory requirement for a single threaded implementation is then

$$\text{RC}_{\text{mem}} = \text{IMG} + \text{RC}_{\text{filter}} \approx 25\,\text{MB}. \tag{4.24}$$

4.2 Line-based

The LB method is a modified version of the RC, giving certain advantages concerning memory [14, 15]. The amount of FLOPs needed is the same as for RC. The methods differ in the order of filtering, not the actual FLOP count, therefore the estimation of the FLOPs are omitted in this section.

The basic procedure of the LB method is illustrated in figure 4.2. As the first step, the first horizontal line (or row) of the image is filtered one level. The filtered line is saved in a buffer. Then the next line of the image is filtered one level horizontally, and saved in the buffer. For the CDF 9/7 DWT, this continues until nine lines have been filtered, as this corresponds to the filter length. Now each vertical line (column) from the buffer is filtered, resulting in a high pass and a low pass component for each. These are the first lines of each of the four wavelet sub-bands (LL, LH, HL and HH), as seen in figure 4.2.

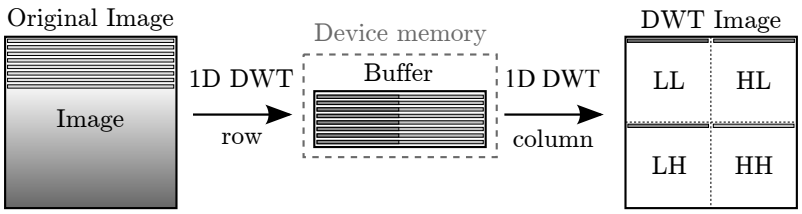


Figure 4.2. Illustration of the procedure for the line-based DWT approach. The first lines corresponding to the filter length are horizontally filtered. Then they are vertically filtered resulting in the first lines in the four sub-bands.

Afterwards another two lines of the image are filtered horizontally, due to downsampling, and written into the buffer, replacing the first two lines. Then the vertical filtering is applied on the updated buffer, giving the second line in each sub-band. This continues until the entire image has been transformed.

To obtain another level of decomposition the procedure just described is applied to the obtained low pass sub-band (LL). When one level is complete and the low pass sub-band LL is processed, only half the buffer size is needed, as the lines of LL are only half the length of the original image. The buffer sizes are then halved for each decomposition level.

The advantage of the LB method is for cases where the available memory is a critical issue [14, 15], like in small embedded systems. The idea is that only the nine buffer lines need to be stored in the device memory. The image lines are loaded into the device one at a time, from e.g. a file, and filtered as described above. The output sub-bands can then be transferred over a data channel, or stored in some external memory, as they are generated, freeing the buffers. Furthermore, if another device require the decomposition, it can begin processing the output sub-bands as soon as they are generated, instead of waiting for the entire decomposition to complete, reducing the transfer delay.

4.2.1 Memory usage

The minimum memory allocation needed on the device for the computations, is the buffer and the filter coefficients. The standard approach is to use the buffer for lines [14, 15], but in principal, the columns could instead be contained by the buffer. This only changes the order of filtering. For a full HD image (1920×1080) it is preferable to filter the columns first, as this result in the smallest buffer size. A full HD RGB image requires a buffer size of

$$\mathrm{LB}_{\mathrm{buffer}} = 3 \cdot M \cdot 9\,\mathrm{B} \cdot 4\,\mathrm{B} = 3 \cdot 1080 \cdot 9\,\mathrm{B} \approx 29\,\mathrm{kB}. \tag{4.25}$$

The required filter memory is the same as for RC found in equation (4.23): $\mathrm{LB}_{\mathrm{filter}} = 64\,\mathrm{B}$. This results in

$$\mathrm{LB}_{\mathrm{mem}} = \mathrm{LB}_{\mathrm{buffer}} + \mathrm{LB}_{\mathrm{filter}} \approx 29\,\mathrm{kB}. \tag{4.26}$$

4.3 Block based

The block based method to perform a wavelet transform is based on element-wise matrix multiplication. As the name implies the block based method is operating on sub-blocks of the image with the same dimensions as the number of filter coefficients. The filter matrix is moved over the picture as illustrated by figure 4.4. This sub-block of the image is multiplied with four different filter matrices. As the wavelet decomposition transform the image into four smaller image blocks, each of these transforms has its own filter matrix which is an outer-product of two vectors containing the filter coefficients. In the case of a CDF 9/7 decomposition, the filter matrices become

$$\mathbf{\Gamma}_{\mathrm{LL}} = \phi \cdot \phi^T \tag{4.27}$$

$$\mathbf{\Gamma}_{\mathrm{HL}} = \psi \cdot \phi^T \tag{4.28}$$

$$\mathbf{\Gamma}_{\mathrm{LH}} = \phi \cdot \psi^T \tag{4.29}$$

$$\mathbf{\Gamma}_{\mathrm{HH}} = \psi \cdot \psi^T \tag{4.30}$$

where $\phi \in \mathbb{R}^{9\times 1}$ is the low pass filter coefficients, and $\psi \in \mathbb{R}^{9\times 1}$ is the high pass filter coefficients, $\mathbf{\Gamma}_{\mathrm{LL}}, \mathbf{\Gamma}_{\mathrm{HL}}, \mathbf{\Gamma}_{\mathrm{LH}}, \mathbf{\Gamma}_{\mathrm{HH}} \in \mathbb{R}^{9\times 9}$ are filter matrices. The subscript on the filter matrices indicate which sub-band block they creates. The transform specifies 7 coefficients for the high pass filter, this is extended with a leading and tailing zero to fit the low pass filters 9 coefficients.

The sum of the filter matrices element-wise multiplication with sub blocks of the image, gives one indices in the next level of the decomposition. If we denote the first decomposition level as $\mathbf{\Upsilon}^1$, then the transform of an image $\mathbf{B}^{M\times N}$ is

$$\mathbf{\Upsilon}^1 = \begin{bmatrix} \mathbf{\Theta}_{\mathrm{LL}} & \mathbf{\Theta}_{\mathrm{LH}} \\ \mathbf{\Theta}_{\mathrm{HL}} & \mathbf{\Theta}_{\mathrm{HH}} \end{bmatrix} \tag{4.31}$$

where

$$\mathbf{\Theta}_{\mathrm{LL}}(x,y) = \sum_{h=1}^{9}\sum_{k=1}^{9} \left[\mathbf{\Gamma}_{\mathrm{LL}} \circ \mathbf{B}(2x-1, 2y-1)\right](h,k) \tag{4.32}$$

$$\mathbf{\Theta}_{\mathrm{LH}}(x,y) = \sum_{h=1}^{9}\sum_{k=1}^{9} \left[\mathbf{\Gamma}_{\mathrm{LH}} \circ \mathbf{B}(2x-1, 2y-1)\right](h,k) \tag{4.33}$$

$$\mathbf{\Theta}_{\mathrm{HL}}(x,y) = \sum_{h=1}^{9}\sum_{k=1}^{9} \left[\mathbf{\Gamma}_{\mathrm{HL}} \circ \mathbf{B}(2x-1, 2y-1)\right](h,k) \tag{4.34}$$

$$\mathbf{\Theta}_{\mathrm{HH}}(x,y) = \sum_{h=1}^{9}\sum_{k=1}^{9} \left[\mathbf{\Gamma}_{\mathrm{HH}} \circ \mathbf{B}(2x-1, 2y-1)\right](h,k) \tag{4.35}$$

where $\mathbf{\Upsilon}^1 \in \mathbb{R}^{M\times N}$, $x \in \mathbb{Z}\ \{1 \leq x \leq \frac{M}{2}\}$, $y \in \mathbb{Z}\ \{1 \leq y \leq \frac{N}{2}\}$ and $[\mathbf{\Gamma} \circ \mathbf{B}] \in \mathbb{R}^{9\times 9}$ is the Hadamard product of a filter matrix and a sub-matrices of the image. The indices (x, y) denote an element in the matrix. For the image $\mathbf{B}$ they denote the upper left index of the blocks placement in the image matrix, see figure 4.3. For a color image with 3 layers the operations are performed for all layers. These equations are equal to equations (2.32) to (2.35), on page 15, with a finite filter.

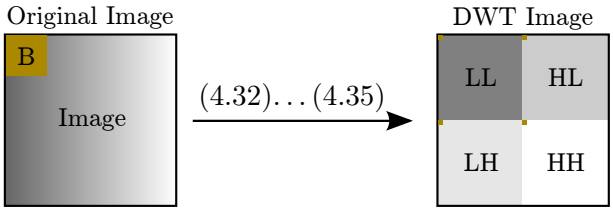


Figure 4.3. Illustration of the mapping at the first decomposition level using equations (4.32) to (4.35) with $(x, y) = (1, 1)$. **B** in this content shows the sub-matrix taken from the image.

$$
\begin{array}{c} \\ M \end{array}
\begin{array}{c}
\overset{N}{\overbrace{}} \\
\begin{bmatrix}
69 & 141 & 57 & 196 & 96 & 231 & 103 & 7 & 183 & 175 & 2 & \cdots \\
92 & 186 & 66 & 185 & 190 & 211 & 170 & 97 & 95 & 9 & 251 & \cdots \\
207 & 158 & 54 & 154 & 82 & 9 & 2 & 116 & 2 & 230 & 237 & \cdots \\
172 & 233 & 232 & 147 & 230 & 103 & 140 & 245 & 192 & 232 & 127 & \cdots \\
99 & 197 & 37 & 99 & 248 & 85 & 145 & 8 & 202 & 40 & 57 & \cdots \\
24 & 128 & 114 & 112 & 10 & 163 & 22 & 6 & 22 & 178 & 230 & \cdots \\
217 & 40 & 246 & 224 & 89 & 66 & 95 & 249 & 173 & 89 & 93 & \cdots \\
94 & 19 & 71 & 78 & 15 & 32 & 8 & 128 & 227 & 28 & 135 & \cdots \\
52 & 58 & 11 & 30 & 86 & 17 & 38 & 6 & 4 & 126 & 154 & \cdots \\
231 & 47 & 65 & 190 & 38 & 84 & 94 & 116 & 116 & 73 & 124 & \cdots \\
37 & 112 & 197 & 88 & 235 & 99 & 83 & 1 & 61 & 42 & 53 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{bmatrix}
\end{array}
$$

Figure 4.4. Illustration of a section of a matrix corresponding to one color lay of an image represented by integers between 0 and 255. The section marked by the red rectangle is the first sub-matrix used in the block-based transform, corresponding to $(x, y) = (1, 1)$. The second block is marked with blue, corresponding to $(x, y) = (1, 2)$. When it is no longer possible to shift to the right, it starts over from the left shifted downwards by two, marked by the green rectangle, corresponding to $(x, y) = (2, 1)$.

4.3.1 Floating point operations

The number of operations are given by the size of the image. The number of block multiplications is calculated by

$$\mathrm{BB}_{\mathrm{bm}} = 4 \cdot \frac{M}{2} \cdot \frac{N}{2} = M \cdot N \tag{4.36}$$

where M and N is the dimension of the image matrix. A single block calculation is given by

$$\mathbf{\Theta}(x, y) = \sum_{h=1}^{9} \sum_{k=1}^{9} \left[\mathbf{\Gamma} \circ \mathbf{B}(2x - 1, 2y - 1)\right](h, k) \tag{4.37}$$

which consists of 9 multiplications and 8 summations per row. The summation of all row sums adds another 8 additions, giving a total of

$$\mathrm{BB}_{\mathrm{bflops}} = (9+8)\cdot 9 + 8 = 161 \text{ FLOP} \tag{4.38}$$

per block. Combining equation (4.36) and (4.38) we get a total operation count for all 3 layers of

$$\mathrm{BB}_{\mathrm{flop}}(M,N) = 3\cdot \mathrm{BB}_{\mathrm{bm}} \cdot \mathrm{BB}_{\mathrm{bflops}} = 3\cdot M\cdot N\cdot 161 \text{ FLOP}. \tag{4.39}$$

For the first decomposition level of a HD color image the total number of floating point operations stated by equation (4.39) is

$$\mathrm{BB}_{\mathrm{flop}}(1080,1920) = 3\cdot 1080\cdot 1920\cdot 161 \text{ FLOP} \approx 1\,\mathrm{GFLOP} \tag{4.40}$$

for the first decomposition level of a HD color image. Seen in operations per pixel the count is

$$\mathrm{BB}_{\mathrm{fpp}} = \frac{1\,\mathrm{GFLOP}}{(1080\cdot 1920)\mathrm{pixel}} = 483\,\mathrm{FLOP/pixel}. \tag{4.41}$$

For the second decomposition level, $\mathbf{\Theta}_{\mathrm{LL}} \in \mathbb{R}^{\frac{M}{2}\times\frac{N}{2}}$ is transformed in the same way as the image in the first level, thus equation (4.39) yields

$$\mathrm{BB}_{\mathrm{flop}}(540,960) = 3\cdot 540\cdot 960\cdot 160 \text{ FLOP} = 250.4\,\mathrm{MFLOP} \tag{4.42}$$

which is a quarter of the first decomposition. This is expected as the transform operates on a quarter of the original image. This scales in the same manor for the remaining levels of decompositions.

4.3.2 Memory usage

All numbers are represented in float32 resulting in the size of a single filter matrix to become

$$\mathrm{BB}_{\mathrm{filter}} = 9\cdot 9\cdot 4\,\mathrm{B} = 324\,\mathrm{B}. \tag{4.43}$$

The block based transform can not overwrite the image while doing the filtering, since it would overwrite necessary data for later use. Therefor it needs an output buffer of the same size as the input. In a highly parallel setup, where the three color layers are transformed at the same time, an output buffer of with the size as the image is required. The memory used for the image is known from (4.22) and the total memory usage adds up to

$$\mathrm{BB}_{\mathrm{mem}} = 4\cdot \mathrm{BB}_{\mathrm{filter}} + 2\cdot \mathrm{IMG} \tag{4.44}$$

$$= 4\cdot 324\,\mathrm{B} + 2\cdot 24.8\,\mathrm{MB} \approx 49.7\,\mathrm{MB}. \tag{4.45}$$

This is for the storage of the image, the wavelet transformed image, and filter matrices.

4.4 Analysis

This short analysis will compare the three algorithms described in this chapter, and one will be chosen for implementation. We are seeking the algorithm with the fewest memory operations, lowest memory usage, floating point operations, and good opportunities for parallelization. An overview of the three algorithms is seen in table 4.1.

Algorithm	MFLOP	Filter memory B	Total memory MB
Row-column	187.0	64	25
Line-based	187.0	64	25
Block-based	1001.5	1296	25

Table 4.1. Recap of the number of floating point operations and required memory for the first decomposition of a wavelet transform of a 1080×1920 HD image.

Looking at table 4.1 it is obvious that the block-based wavelet transform has a disadvantage over the two other transforms. It requires

$$1001.5 - 187 = 814.5\,\text{MFLOPs} \tag{4.46}$$

more per picture, which is more than a factor 5 of extra FLOPs for the first decomposition level. The extra FLOPs only contribute with more operation time. Since the implementation should run as fast as possible, this leave us with row-column and the line-based method. These two methods are very similar and a more detailed comparison follows.

RC and LB are basically doing the same operations, the only difference is the order in which they are performed. The LB algorithm has an advantage in embedded systems where the available memory is limited, as it operates on smaller chunks at the time. This advantage is not important for a GPU implementation in the same way, as the available global memory is large enough to hold several images at the same time. That said, the LB algorithm might benefit the GPU implementation in other ways, such as using the faster local memory.

For both RC and LB there are no dependency between rows or columns operation, and it is possible to perform both low and high pass filtering simultaneously on the same row or column, keeping the read/write operations to a minimum. Since all rows or columns can be processed independently, it is possible to parallelize the filtering such that all rows or columns are done simultaneously. However, all row-wise operation must finish before the column-wise operation can start and vise-versa. The extra overhead by parallelizing is 64 B per computation unit, this however, is under the assumption that both filters are stored in local memory and the image is accessed globally. One of the great advantages of the line-based DWT is that the compression and transfer of the wavelet coefficients do not have to wait until one complete decomposition level has been processed [14, 15]. The lines of the high frequency sub-bands can be compressed and transferred as they are generated. This process can then run in parallel with the decomposition of the next lines. This allows

a more flexible distribution of data transfer over time. Furthermore, every generated new line of the low pass part can be horizontally filtered right away, also running in parallel to the other processes.

We assume that an image is transfered to the global memory of the device as transferring the image in smaller parts would add additional lag due to transfer latency. Looking at a single row in the picture, each element (pixel) is used in the filter calculations more than once, it is therefor beneficial to load these elements into the device local memory where the access time is very low. The local device memory has a limited size, but for newer GPUs it should be possible to load a full line of 1920 pixels into the local memory. It might however not be possible to load all 9 lines which is required by the LB method, as it would use

$$1920 \cdot 9 \cdot 4\,\mathrm{B} = 69\,120\,\mathrm{B} \tag{4.47}$$

of local memory, which exceed the amount available on some older cards, see table 5.2. Filtering the columns first, does not solve the memory problem as it still require

$$1080 \cdot 9 \cdot 4\,\mathrm{B} = 38\,880\,\mathrm{B}. \tag{4.48}$$

GPUs are generally bes at performing Single Instruction Multiple Data (SIMD). The RC method fits this structure, as the same instructions can be used for all rows and afterwards on all columns. An implementation of the LB method is a more complex task and would require either synchronization barriers, or dividing the problem into small tasks. Therefore the RC method is chosen for implementation.

Part II

Implementation and benchmarking

5 Implementation

It has been chosen to both develop an implementation using filtering and lifting, for the purpose of comparison. In this chapter the platform used for testing and development is presented as the first thing. As the next thing the structure of the implementation is specified, providing an overview. After this the initial setup and basic functions are explained followed by the filtering and lifting implementation. After this it is explained how HD images are padded to obtain full decomposition for these. Finally to verify if the implementation computes the correct forward and reverse transfrom, the verification test used during the development and how the test images are generated is explained.

5.1 Test and development platform

The test and development platform is a high end consumer grade PC. To give an idea of the compute capabilities of this, and to make result reproducible, both the hardware and software setup is presented. The specifications for the hardware of the development and test platform is listed in table 5.1.

Hardware module	**Specifications**
CPU	Intel Core i7 970 6 cores of 3.20 GHz 12 MB cache Supports hyper threading
Host memory	24 GB
Graphics Cards (GPU)	GeForce 9800 GT MSI GeForce GTX 680

Table 5.1. Specification for the hardware modules of the test platform.

The specifications for the grapics card[1] are listed in table 5.2. The GTX 680 is a fairly new model, with up to date specifications, and is considered a high end consumer card.

[1] GTX 680: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications

Specifications \ Card	GeForce GTX 680
Graphics Clock (MHz)	1006
CUDA Cores	1536
Memory on card (MB)	2048
Memory bandwidth (GB/sec)	192.2
Memory interface width	256-bit
Memory type	GDDR5

Table 5.2. Specification for the two graphics cards used for development and testing.

The software and their versions installed on the test platform are listed in table 5.3.

Name	**Version**
Linux (OS)	3.3.6-1-ARCH
NVIDIA driver	295.53-1
OpenCL	295.53-1
Python	2.7.3
`pyopencl`	2011.2-1
`numpy`	1.6.1-1
MATLAB	7.14.0.739 (R2012a)

Table 5.3. Specification for the software of the test platform.

5.2 Implementation structure

The implementation is mainly written in Python, using `PyOpenCL` as API for controlling the GPU. The kernels running on the GPU are written in a language based on C99 with some limitations and additions. The software is structured with a module called `wavelet`, with two classes: `lifting` and `filtering`. An illustration of the structure is shown in figure 5.1. Each class independently handles initialization of the GPU, memory layout, and perform the forward and reverse DWT.

In the following sections, the implementation of the filtering and lifting are described.

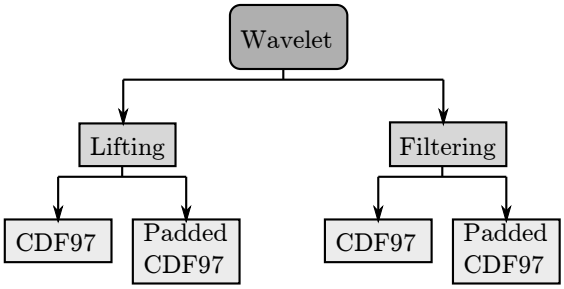


Figure 5.1. Ilustration of the code structure used through out the rest of the project.

5.3 Initial setup and basic functions

The general layout of a wavelet transform module is shown in listing 5.1.

```
1  class CDF97_lifting(object):
2      # Private initialization methods
3      def __init__(self, params):
4      def _getGPU(self, params):
5      def _allocateMem(self, params):
6      def _compileKernels(self, params):
7      def _setupSizes(self,params):
8      def __del__(self):
9
10     # Private methods
11     def _waveletDec(self, decomposition, levels):
12     def _waveletRec(self, image, levels):
13
14     # Public methods:
15     def decompose(self, params):
16     def reconstruct(self, params):
```

Listing 5.1. The structure of a wavelet transform module.

Using this layout makes it easy to initialize a DWT algorithm and obtain the decomposition and reconstruction. When the object is released, the destructor frees the memory and GPU.

When the user wants to perform a wavelet transform, he needs to perform the operations shown in listing 5.2.

```
1  import wavelet
2  import matplotlib.pyplot as plt
3
4  img = plt.imread('image.png') # Read the image
5  gpu = wavelet.lifting.CDF97_lifting(image_shape) # Initialize the GPU
6  imgD = gpu.decompose(img,levels) # Calculate decomposition
7  imgR = gpu.reconstruct(imgD,levels) # Calculate reconstruction
8  del gpu # Release the gpu object
```

Listing 5.2. The structure of a wavelet transform module.
Code: **stdscript.py**

The `plt.imread()` returns the image in float32.
generalt til psudokoden

5.3.1 Initialize a compute device

Initializing and finding the compute device is done in several steps. This all is part of the `_getGPU()` function in listing 5.1. The i device is initialized using `PyOpenCL`, imported as `cl`. OpenCL uses a handler called context to interface with computing device(s). In Python this can be done as shown on listing 5.3. If multiple devices are available on the platform, `device` is used to chose which compute device to use. If a specific device is chosen or wanted, `device` is set to this. In listing 5.3, line 2 through 4 shows the procedure. First `cl.get_platform()` finds the computation platform, where

after `platform.get_devices()` returns a list of known compute devices (Graphic-cards in most cases). The argument `device` is the index number to the wanted device. Lastly `cl.Context()` is used to created the OpenCL handler for the chosen device. If the compute device number is unknown, the `device` argument is not provided, or an optional pick is wanted, line 6 is used. The function `cl.create_some_context()` picks the default device (Graphic card if only one) or prints the list of available devices, and let the user pick the card.

At line 7, the `cl.CommandQueue(context)` creates a queue, used to schedule all handles to the GPU (kernels, data transfers, memory allocation and so on) between host and device.

```
1 if device:
2     platform = cl.get_platforms()[0]
3     device = platform.get_devices()[device]
4     ctx = cl.Context([device])
5 else:
6     ctx = cl.create_some_context()
7 queue = cl.CommandQueue(ctx)
```

Listing 5.3. Function used to retrieve and initialize a compute device.

Loading programs/kernels

Programs on the computing device are called kernels. A code example to prepare and compile a kernel is shown in listing 5.4. Firstly the kernel is loaded into python as a string. This is passed to the object `cl.Program(ctx, kernel)` in line 1 of listing 5.4. `Ctx` is the previous created context (line 4 or 6 in listing 5.3) and the kernel is the OpenCL-program as a string. This variable is then executed with `build()` in line 2.

```
1 program = cl.Program(context,kernel)
2 program.build()
```

Listing 5.4. Function used create the kernel program and compile it.

The variable `program`, now contains a list of executable programs on the compute device. Execution of these programs can be enqueued with `program.<kernel_name>()`. Additional arguments are needed in order to initiate and execute the kernel. Mandatory arguments are `queue, grid size and group size`, kernel related arguments are listed after these. The code example is shown in listing 5.5. Grid and group size are explained in section 3.4.

```
1 program.<kernel_name>(queue, grid size, group size, arg*)
```

Listing 5.5. Enqueue a kernel.

`program()` creates an event which has different methods, such as `wait()`, which stops the Python execution until the kernel executed has finished. Multiple kernels and data transfers can be enqueued at once.

Memory allocation and data transfer

Moving data and allocating memory on the device, can be done in a number of ways. In `PyOpenCL` this can be done with either `cl.Buffer` or the more `numpy` like array style `cl.array`. Buffers are initiated with `cl.Buffer(context, cl.memory_flags, data size / hostbuf=data)`. Depending the purpose of the data, different `cl.memory_flags` can be set:

- `READ_ONLY`
- `WRITE_ONLY`
- `READ_WRITE`
- `COPY_HOST_PTR`

These can be used to control data accessibility on the device. Setting the flag `COPY_HOST_PTR` initiates a data copy from host to device, the `hostbuf` is a pointer to the data on the host device. Otherwise, a buffer is initialized and the device only allocates the space, and no data is transfered. Alternatively, `cl.array(queue, sizes, dtype)` can be used, which is a subclass to `cl.Buffer`. This is very similar to `numpy`'s `ndarray`, with similar methods, e.g., `.zeros()` and `.zeros_like()`. But also other features such as memory flags for shape and size can be used by the host. This is very useful and help code readability. Transferring data to and from the compute device is easily done with a `.get()` and `.set()`.

Memory access

The global memory on a GPU has a flat layout, as mentioned in section 3.4 on page 39. This means that a two or three dimensional array is flattened out. To access the data, the dimensions of the original array must be know, so the original coordinates can be used. Listing 5.6 shows the basic idea to access the memory. The Python and C are languages are both row-major, meaning, that the last dimension always is contiguous. This means x is contiguous in a 2 dimensional array and z is contiguous in a 3 dimensional array.

```
1
2  #define GLOBAL_BUFFER(z,y,x) globalBuffer[z + ZDIM * (x + XDIM * y)]
3
4  __kernel void test(__global float *globalBuffer)
5  {
6    int Glox = get_global_id(0);
7    int Gloy = get_global_id(1);
8    int Gloz = get_global_id(2);
9
10   GLOBAL_BUFFER(Gloz,Gloy,Glox) = 3;
11 }
```

Listing 5.6. Enqueue a kernel.

The dimensions XDIM and ZDIM can either be provided as arguments, which makes the kernel flexible to operate on images of different dimensions, or these can be hard-coded using defines. Since the kernel code is read as a string in Python, these "hard-coded"

values can be substituted before the kernel is compiled. This makes it easy to change the dimensions and recompile the kernel, reducing the amount of arguments that need to be provided to the kernel itself when it is called.

5.4 Row-column filtering

As described in section 4.1 on page 45, the row-column method performs multiply-accumulation on 9 pixels at the time. Since accessing memory is the most time expensive operation, it is beneficial to keep these at a minimum. Looking at the filtering it is clear that each matrix element is used more than once when doing the decomposition. The time used on memory transfers can be reduced if the work group sizes are made in such a way, that it loads a larger part of a row or column into a shared array located in local memory. All work items in a work group shares access to the locale elements, thereby reducing the transfer operations to and from global memory and making the data faster accessible to the treads. Since it is the same 9 matrix elements that are used for both low and high pass filtering, the memory access can be reduced further by letting each work item do both filter operations, and thereby reduce the memory access by a factor 2.

The implemented row-column algorithm is divided in two steps for each decomposition level, each having its own GPU kernel. Step one filters all rows with both the low and high pass filters, and stores the output column wise thereby transposing the output. The output from step one is the input for step two. Transposing the image has two advantages. For one the same kernel code can be used for both steps, secondly the input for step two can be read row wise. Step two filters the transposed data, this corresponds to filtering the columns of the original image. Step two also transposes the output and the filtered image returns to its original orientation, and this completes a decomposition level.

5.4.1 Grid shape

The grid shape is set to be the same height and depth as the image, and half the width, as illustrated by figure 5.2. The result of this is half as many work items as there are elements in the input matrix. Since each work item downsamples the pixels and outputs both a low and a high pass element, there are always as many output elements as there are input elements. The grid is divided into two dimensional rectangular work group sizes, such that the width of the grid is an integer factor of the work group width and likewise with the height.

These two-dimensional work group sizes make it easy to map an element from the image onto a local array, which will be explained later. What needs to be noticed is that each work group has to operate on more elements than it contains to, cover the entire image. This is illustrated by figure 5.3, where it is shown that a work group should cover an area of the image of twice its own size. The overlap into neighboring work group areas is caused by the filter length.

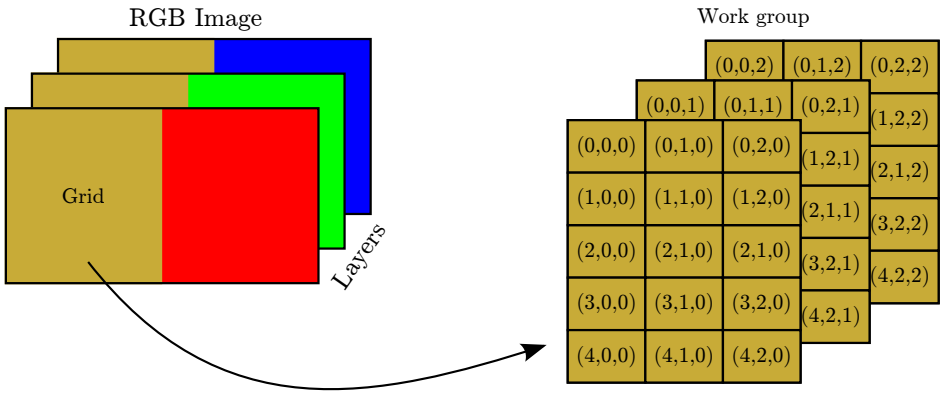


Figure 5.2. Illustration of the grid layout compared to the original image. On the left the grid is shown to cover all three color layers and to the right how the grid is divided into work groups, with work group indices.

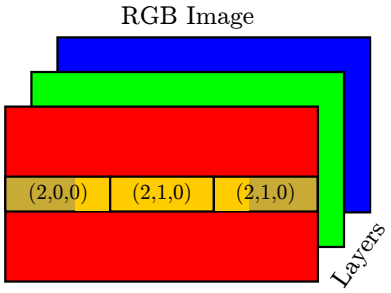


Figure 5.3. Illustration of the part of the image processed by work group $(2, 1, 0)$ and the overlap with nearby work groups (marked with orange).

5.4.2 Overhead

The number of FLOPs needed for a decomposition level for the algorithm in its raw form was introduced in section 4.1 on page 45. With the overlap between GPU work groups more operations are done. How many more will be calculated in the following. The number of work groups per row is

$$\mathrm{RC}_{\mathrm{workgroups}} = \frac{\mathrm{Image}_{\mathrm{width}}}{\mathrm{Group}_{\mathrm{width}}}. \tag{5.1}$$

The FLOP count for a single work group is

$$\mathrm{RC}_{\mathrm{FLOPs}} = 2 \cdot \frac{1}{2} \cdot \mathrm{Group}_{\mathrm{width}} \cdot (8 + 9) = \mathrm{Group}_{\mathrm{width}} \cdot 17 \text{ FLOP} \tag{5.2}$$

From the dimensions of the image and the work group width, the total amount of FLOPs for a single kernel can be calculated as

$$\begin{aligned} \mathrm{Kernel}_{\mathrm{FLOPs}} &= \mathrm{RC}_{\mathrm{workgroups}} \cdot \mathrm{RC}_{\mathrm{FLOPs}} \cdot \mathrm{Image}_{\mathrm{height}} \cdot \mathrm{Image}_{\mathrm{depth}} && (5.3) \\ &= 17 \text{ FLOP} \cdot \mathrm{Image}_{\mathrm{width}} \cdot \mathrm{Image}_{\mathrm{height}} \cdot \mathrm{Image}_{\mathrm{depth}}. && (5.4) \end{aligned}$$

Stated as a nice function, equation (5.4) becomes

$$\mathrm{FLOPs}_{\mathrm{total}}(M, N, Z) = M \cdot N \cdot Z \cdot 17 \text{ FLOP} \tag{5.5}$$

where M, N, Z is the image dimensions. To give an impression of the overhead and example is given for the first level of decomposition of a $1080 \times 1920 \times 3$ color image. The FLOP count for the algorithm is given by equation (4.10) as

$$R = 3 \cdot 1920 \cdot 1080 \cdot 30 \text{ FLOP} = 186.6 \text{ GFLOP}. \tag{5.6}$$

The flop count for the implementation using equation (5.5) becomes

$$C = 2 \cdot \text{FLOPs}_{\text{total}}(1080, 1920, 3) \tag{5.7}$$
$$= 2 \cdot 17 \text{ FLOP} \cdot 1080 \cdot 1920 \cdot 3 \tag{5.8}$$
$$= 211.5 \text{ GFLOP} \tag{5.9}$$

for execution of both kernels. The overhead is now calculated as

$$\text{Flop}_{\text{difference}} = C - R \tag{5.10}$$
$$= 211.5 \text{ GFLOP} - 186.6 \text{ GFLOP} \tag{5.11}$$
$$= 24.9 \text{ GFLOP} \tag{5.12}$$

additional floating point operations for the first level of decomposition. This corresponds to an increase of $\frac{24.9 \text{ GFLOP}}{186.6 \text{ GFLOP}} \cdot 100 = 13.3\%$.

As mentioned, the global indexes for each work item in a work group are mapped onto a local array. The local array is implemented in the kernel as a two-dimensional shared local array. Remember that the grid size is only half of the image, so each work item loads two floating point numbers from the global memory to the local array. The following is a thought example to explain the mapping between the global grid indexes and the shared array. Assume that the work group $(2, 1, 0)$ from figure 5.3 has dimensions 1×8. Then the local array shared between the work items of this work group has dimension 1×24, because the work group covers twice as many pixels as its own length plus filter overhead. This gives the work item indexes $[0; 7]$. Each work item loads two contiguous elements from the image into the array, such that the array is a segment of the image as illustrated on figure 5.4

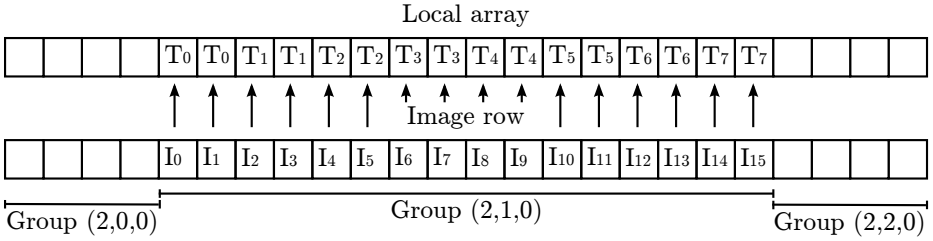


Figure 5.4. The T_x in the local array denotes the work item index. The I_x is denoted in this way only to emphasise that these are contiguous pixel elements. The values of the two array are equal.

When all work items have loaded two pixels into local memory, work items $[0; 3]$ each loads an element to the start and end of the array, as illustrated by figure 5.5.

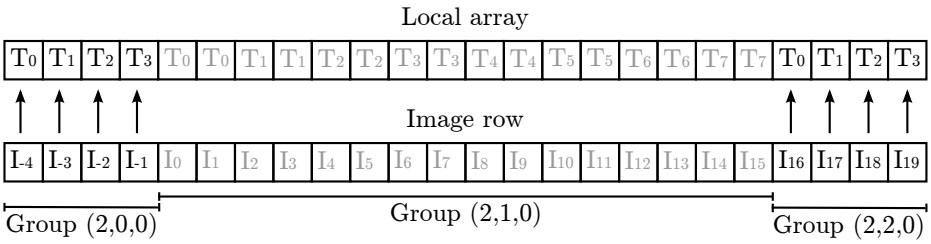


Figure 5.5. Each work item $T_{[0;3]}$ loads a value to the start and end of the local array. The local array is now equal to the part of the image that the work group should cover plus the extra values that the filter exceeds.

The work items in the work group now have fast access to the necessary values to perform 8 low pass and 8 high pass filtering operations. The rest is trivial multiply and accumulate, and will not be explained in more details. The reader is referred to the kernel code in appendix D for more information.

5.4.3 Device buffers

In order to keep the transfer between host and device and the allocation of new memory to a minimum the implementation uses the same two buffers for all decomposition levels. The first buffer has the same dimensions as the image. In this the image is stored at the beginning of a decomposition, and the final decomposed image is read from the same buffer. The second buffer has the same dimensions as the transposed image, and is used to store the transposed image between the two kernels. For the second decomposition the same buffers are used but only one fourth of the buffer is written to at this point because of the new dimensions of the already filtered image. This continues for all decomposition levels such that the buffer sizes remain but the part of the buffer used is divided by four for each level. This is roughly illustrated by figure 5.6.

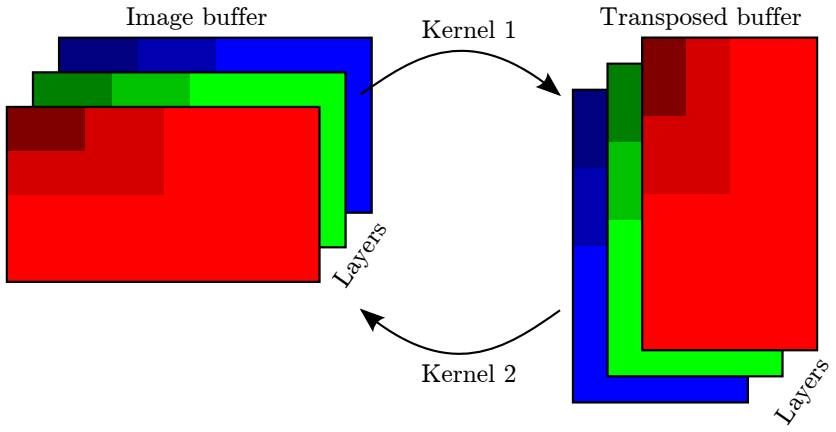


Figure 5.6. Illustration of the image buffer and the temporary buffer holding the transposed image before second filtering. The part of the buffer used for the second decomposition is marked with a slightly darker color and the darkest colors mark the buffer used in the third decomposition.

Since the image and transposed buffer is flat in memory it would make sense just to use the full width of the transposed buffer, it is however easier to keep track of the indexes and make simpler code by placing it in the same manner as the decomposition levels are placed naturally.

5.5 Lifting

One of the implemented wavelet transform algorithms is lifting. The theory is described in section 2.5. Lifting has the advantage over filtering that it only requires around half of the FLOPs [7]. The forward lifting equations for the CDF 9/7 (equation (2.114) to (2.119)), from section 2.6 are rewritten here:

$$d_0[n] = S[2n+1] \tag{5.13}$$
$$s_0[n] = S[2n] \tag{5.14}$$
$$d_1[n] = d_0[n] + \alpha \cdot (s_0[n] + s_0[n+1]) \tag{5.15}$$
$$s_1[n] = s_0[n] + \beta \cdot (d_1[n-1] + d_1[n]) \tag{5.16}$$
$$d_2[n] = d_1[n] + \sigma \cdot (s_1[n] + s_1[n+1]) \tag{5.17}$$
$$s_2[n] = s_1[n] + \delta \cdot (d_2[n-1] + d_2[n]) \tag{5.18}$$
$$s[n] = K \cdot s_2[n] \tag{5.19}$$
$$d[n] = K^{-1} \cdot d_2[n]. \tag{5.20}$$

The d steps are the predict steps, calculating the details (high pass part) of the signal, whereas the s steps calculate the mean values through the update steps. The final equations (5.19) and (5.20) scale the details and mean values such that the transform is biorthogonal. The input signal (the image) S is split up in the even and odd pixels, and the multiplication constants α, β, σ, δ and K can be found in table 2.5 in section 2.5.

5.5.1 Algorithm analysis

The lifting algorithm for the CDF 9/7 wavelet consists of 4 steps, two predict and two update, and a scaling in the end. The steps depend on values, created by preceding steps. This thus makes it necessary to perform the steps in the correct order, as listed in equations (5.13) to (5.20).

Sequential algorithm

A sequential algorithm is illustrated in figure 5.7. This algorithm performs the lifting steps in-place, meaning that the results are stored in the same array as the operands are read from.

When looking at figure 5.7 and equations (5.15) and (5.16), it becomes clear, that d_1 only operates on the odd indexes, whereas s_1 only operates on the even indexes. Since s_1 does not need any s_0 data from the odd indexes, d_1 can be stored in these. Subsequently s_1

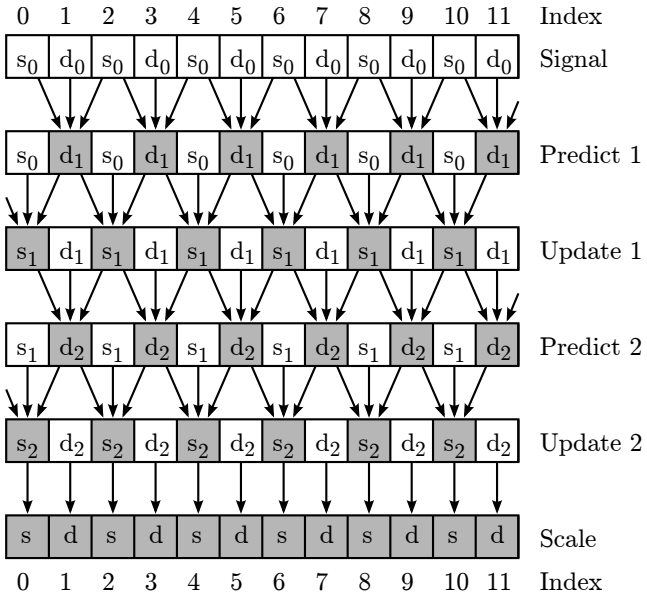


Figure 5.7. Sequential lifting algorithm illustrated as an in-place implementation. The gray colored fields are updated through the step. The numbers on top and in the bottom are the indexes, such that s_0 below number 2 corresponds to $s_0[1]$. The arrows coming from "outside" the figure illustrate the required values, from outside the signal.

can be stored in the even indexes. This can be done through all the lifting steps. An issue here is, that for every lifting step, a large number of read/write operations to the memory have to be performed, which can reduce the performance of the algorithm. Pseudo code for sequential lifting algorithm is shown in algorithm 5.1. The scaling steps are omitted here and the s and d data sets have to be separated after performing the algorithm. The advantage is, that no additional memory is needed to perform the algorithm.

The memory substitution can be illustrated as:

$$S[2n] \rightarrow s_0[n] \rightarrow s_1[n] \rightarrow s_2[n] \rightarrow s[n] \tag{5.21}$$

$$S[2n+1] \rightarrow d_0[n] \rightarrow d_1[n] \rightarrow d_2[n] \rightarrow d[n] \tag{5.22}$$

Algorithm 5.1 Sequential in-place lifting implementation - One step for each iteration.

for n in range$(N/2)$ **do** ▷ Length of image
  $S[2n+1] \leftarrow S[2n+1] + \alpha \cdot (S[2n] + S[2n+2])$ ▷ Calculate d_1
end for
for n in range$(N/2)$ **do**
  $S[2n] \leftarrow S[2n] + \beta \cdot (S[2n-1] + S[2n+1])$ ▷ Calculate s_1
end for
for n in range$(N/2)$ **do**
  $S[2n+1] \leftarrow S[2n+1] + \sigma \cdot (S[2n] + S[2n+2])$ ▷ Calculate d_2
end for
for n in range$(N/2)$ **do**
  $S[2n] \leftarrow S[2n] + \delta \, (S[2n-1] + S[2n+1])$ ▷ Calculate s_2
end for

On-the-fly algorithm

Another approach is to run the lifting steps "on-the-fly", as shown in figure 5.8. Here all lifting steps are applied subsequently for one pair of samples of the signal. In this case we have to observe the data dependencies for s_2, which is the last step in the procedure, and iterate backwards to see what data s_2 depends on. This is done by inserting equations (5.17) (5.16) and (5.15) into (5.18) to trace, which data has to be calculated in advance. The result is that the $s_2[n]$ value, calculated at index $2n$, requires results from 4 pixels to each side. For $s_2[2]$ and $d_2[2]$, located at indexes 4 and 5 respectively, ($s_2[0]$ and $s_2[1]$ are not calculated), calculations based on the original samples from index 0 to 8 are required. This is illustrated in figure 5.8 where the arrows from s_2 can be followed. This results in the pseudo code shown in algorithm 5.2.

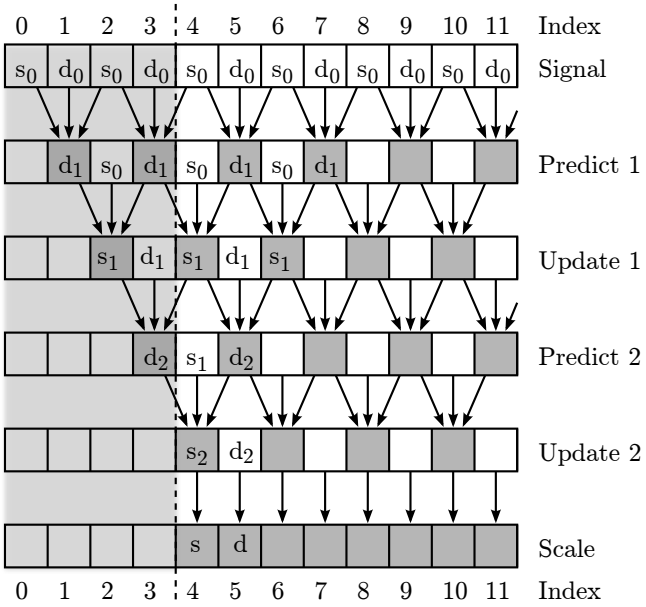


Figure 5.8. Illustration of an on-the-fly lifting implementation. The values at $n = 0$ to $n = 3$ have to be calculated before entering the loop. The numbers on top and in the bottom of the figure indicate the indexes of the samples, such that s_1 below index 2 is $s_1[1]$

Algorithm 5.2 Lifting implementation - On The Fly implementation.

$S[1] \leftarrow S[1] + \alpha \cdot (S[0] + S[2])$ ▷ Do calculations prior to the loop
$S[3] \leftarrow S[3] + \alpha \cdot (S[2] + S[4])$ ▷ Calculate d_1 in $S[1]$ and $S[3]$
$S[2] \leftarrow S[2] + \beta \cdot (S[1] + S[3])$ ▷ Calculate s_1 in $S[2]$
$S[3] \leftarrow S[3] + \sigma \cdot (S[2] + S[4])$ ▷ Calculate d_2 in $S[3]$
for n in range(2 to $N/2$) **do** ▷ Loop through the elements
  $S[2n+3] \leftarrow S[2n+3] + \alpha \cdot (S[2n+2] + S[2n+4])$ ▷ Calculate d_1 two steps ahead
  $S[2n+2] \leftarrow S[2n+2] + \beta \cdot (S[2n+1] + S[2n+3])$ ▷ Calculate s_1 one step ahead
  $S[2n+1] \leftarrow S[2n+1] + \sigma \cdot (S[2n] + S[2n+2])$ ▷ Calculate d_2 one step ahead
  $S[2n] \leftarrow S[2n] + \delta \cdot (S[2n-1] + S[2n+1])$ ▷ Calculate s_2
end for

This algorithm begins at $n = 2$, such that the first s_2 is stored in $S[4]$. Note that

the first steps of d_1 and s_1 have to be calculated prior to the `for loop`, whereas the last steps of $d_2[n]$ and $s_2[n]$ have to be calculated after the loop. The result calculated in $s_2[n]$ and $d_2[n]$ can be stored in the input array.

Selecting the algorithm

When considering parallel implementation on a GPU, algorithm 5.2 has a disadvantage, since the work items need to synchronize between each step to satisfy the data dependencies, as d_1, located in $S[2n+3]$, has to be ready before s_1 at $S[2n+2]$ can be calculated and so on. This data can be stored in registers while calculating the lifting steps on a signal, since the data is needed for at most 8 iterations. The issue here is to distribute instructions and data to the work groups, or make each work group perform one step, resulting in a Multiple Instruction Multiple Data (MIMD) technique which is not suited for GPU implementation [16].

The sequential implementation in algorithm 5.1 shows potential since it can be performed as SIMD, which generally gives good performance on a GPU[16]. Each step can be performed in parallel, distributed on different work groups. These need to write the results to the memory and synchronize before the next step can be performed. This can result in a large amount of context switching, which will reduce the performance of the GPU slightly. Since all work groups need to synchronize between each lifting step, a huge number of memory accesses are required. This has to be considered when implementing the algorithm. For these reasons, we continue with the sequential algorithm.

5.5.2 Performance considerations

The implementation of the sequential lifting algorithm is described in this section, and different considerations of the group and memory layout are presented and discussed.

In the implementation, it is chosen to load a part of the image into the local memory. An additional overhead of 4 pixels to each side of the window is loaded as well. This is illustrated in figure 5.9, where the additional overhead is indicated by the dashed lines. All lifting steps are then performed on this part of the image, where the work groups synchronize between each step. Finally, the data is written to the global memory and the next part of the image is loaded. The algorithm has to be executed twice; once horizontally and once vertically, resulting in one level of decomposition.

Algorithm 5.1 is performed in each of the work groups. When observing the steps of the algorithm, shown in figure 5.10, it becomes clear, that the pixels at the borders of each group, depend on values from outside the group, which are marked with a gray background. These additional values have to be available when performing the calculations. For this reason, an overhead of 4 pixels to each side is added. These will be used to calculate the d_1 and s_1 values, needed for d_2 and s_2, as shown in figure 5.10. This results in an overlap and the need for extra calculations since these calculations also are performed in the neighboring groups. These extra calculations can be performed in two ways:

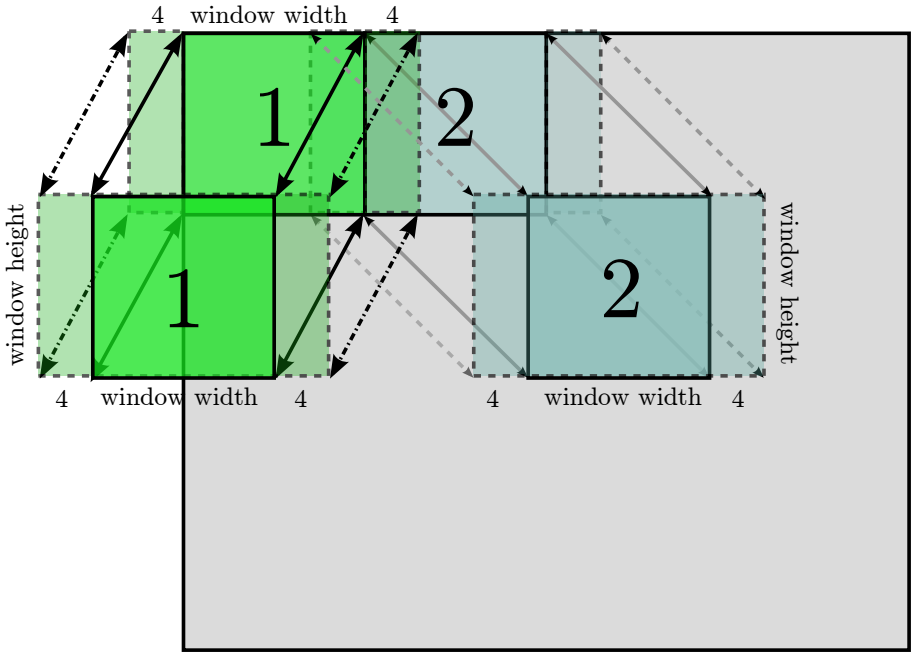


Figure 5.9. Two groups with overhead loaded into memory. The window width plus 8 pixels is twice the working group size. Te overhead is indicated by the dashed lines.

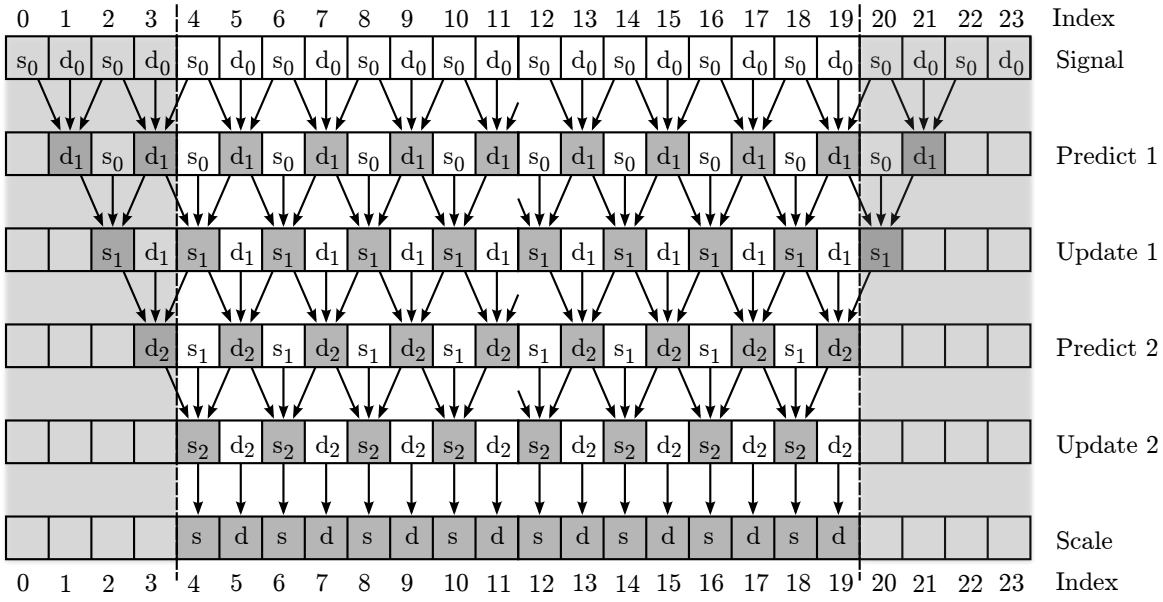


Figure 5.10. The algorithm performed on a working group. The overhead is colored gray. The numbers on top and in the bottom of the figure indicate the indexes, such that s_1 below 2 is $s_1[1]$

Layout 1: 4 work groups perform the extra calculations in the gray colored area in figure 5.10, while the others are idling. The work group width is in this case defined as:

$$\text{group\_width} = \frac{\text{window\_width}}{2} \tag{5.23}$$

such that every work group works on two pixels, and stores two pixels in the global memory when the calculations are finished. This means that the window_width is calculated as:

$$\text{window\_width} = 2 \cdot \text{group\_width}. \tag{5.24}$$

Some of the work groups are needed to fetch the last 8 pixels from the global memory before the calculations, and some work groups are needed to perform the necessary lifting steps on these pixels. This results in work group divergence, since not all work groups perform the same operations at the same time.

Layout 2: All work groups work on two pixels each. This results in the following working group width:

$$\text{group\_width} = \frac{\text{window\_width} + 8}{2}. \tag{5.25}$$

In this case, all work groups work on the same pixels, and no work groups are idling while others calculate. This reduces the work group divergence. The overhead is larger compared to the working group size, than in the first layout, since the working window is smaller. Work Group divergence is introduced when the results are stored to the global memory, since the first and last 4 pixels are discarded. The window_width can be calculated as:

$$\text{window\_width} = 2 \cdot \text{group\_width} - 8. \tag{5.26}$$

It is chosen to use **Layout 2**, since this avoids work group divergence during the calculations. Pseudo code for this implementation is shown in algorithm 5.3.

Algorithm 5.3 Lifting GPU implementation. S is a copy of the signal, and is located in the local memory of the GPU.

$n \leftarrow$ Local_id ▷ Get the local ID in the group
$S[2n+1] \leftarrow S[2n+1] + \alpha \cdot (S[2n] + S[2n+2])$ ▷ Calculate d_1
Sync_work_groups ▷ Wait for all work groups in group to reach point
$S[2n+2] \leftarrow S[2n+2] + \beta \cdot (S[2n+1] + S[2n+3])$ ▷ Calculate s_1
Sync_work_groups
$S[2n+3] \leftarrow S[2n+3] + \sigma \cdot (S[2n+2] + S[2n+4])$ ▷ Calculate d_2
Sync_work_groups
$S[2n+4] \leftarrow S[2n+4] + \delta \cdot (S[2n+3] + S[2n+5])$ ▷ Calculate s_2
Sync_work_groups

As seen, work group 0 moves from index 0 to 4 while performing the lifting steps, where the results from d_2 and s_2 are final results that can be stored in the output array. Since the indexing in arrays normally begins at 0, it is chosen to begin storing results from $S[4]$ and up.

In the first step, all work groups operate on an odd pixel, in the next on an even pixel and so on. For this reason each work group has to write an odd and even result to the global memory. It should be remembered that a total of 8 pixels; 4 to each side, have to be discarded due to the overlap. By observing algorithm 5.3, it is clear, that the results from the last 4 work groups can not be used, since they operate on non existing data. Additionally the results d and s, stored in the odd and even pixels respectively, have to be stored in the first and last half of the output array. This results in algorithm 5.4. Here the `if`-statement ensures, that the last 4 work groups do not write out their results. This

Algorithm 5.4 The algorithm to write out the results to global memory.

if $n <$ GROUP_SIZE $- 4$ **then**
  OUT(group_id$+n$) $\leftarrow S[2n+4]$ ▷ Even results in first half of image
  OUT(group_id + LENGTH$/2 + n$) $\leftarrow S[2n+5]$ ▷ Odd results in last half of image
end if

algorithm only performs the lifting steps in the horizontal direction, which means that the output matrix has to be transposed such that the same kernel can be executed again to obtain a two dimensional DWT. The OpenCL code for this algorithm can be found in **__CDF97Old.py** and the kernel in **__CDF97OldKernel.c**.

Grid shape

The computation grid must have the correct shape for the work groups to cover the entire image. In **layout 2**, the vertical size of the window is the same as the work group. The horizontal window shape is twice the group shape minus 8 pixels. The horizontal grid width can be calculated as:

$$x_{\text{grid}} = \frac{x_{\text{image}}}{x_{\text{window}}} \cdot x_{\text{group}} \tag{5.27}$$

where

$$x_{\text{window}} = 2 \cdot x_{\text{group}} - 8. \tag{5.28}$$

The grid for the first level of decomposition is shown on figure 5.11. For the next levels of decomposition, the image width and height correspond to the dimensions of the LL part of the image. The horizontal dimension of the image, x_{image}, is halved for every decomposition level, which means that the grid shape also is reduced for every level of decomposition. The shape of the image is 2×2 pixels at the final level of decomposition, which according to equation 5.27 means, that the grid should be smaller. Due to the overlap of 8 pixels, the width of the group must not go below 8, since calculations next to the pixel must be performed. For this reason the grid at the final iterations, where

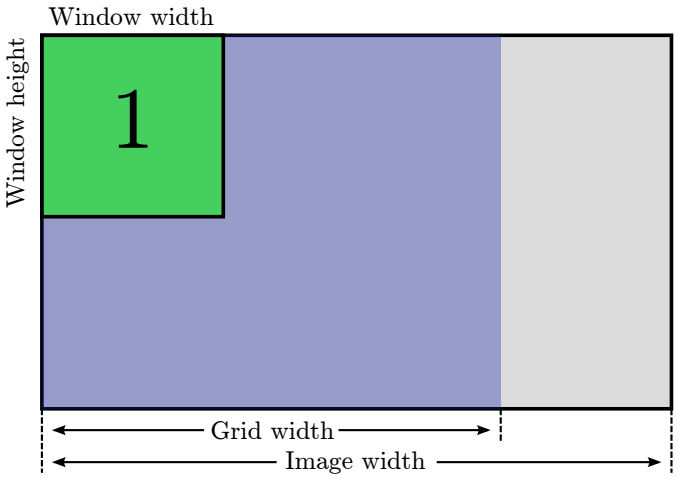


Figure 5.11. The grid shape illustrated on the image for the first level of decomposition.

the image gets below the chosen group shape, is set to the group shape. This results in additional calculations within the group, but does not affect the performance significantly, since these are performed in parallel with the necessary calculations. When the results are written back to the global memory, it must be assured that only pixels from inside the image are written. This is done by sending the image shape as an argument to the kernel, which checks whether the coordinates on the grid are inside the image. To ensure that the grid covers the image, and the number of work groups becomes an integer, equation (5.27) is modified to:

$$x_{\text{grid}} = \text{ceil}\left(\frac{x_{\text{image}}}{x_{\text{window}}}\right) \cdot x_{\text{group}} \tag{5.29}$$

The above mentioned solution makes the kernel more flexible to images of different resolutions, since the window shape and thereby the group shape do not necessarily have to fit the image exactly. This means that the block shape can be chosen almost arbitrarily, such that the one giving the best performance on the current GPU can be chosen. In this implementation the group shape is set to 32×32 pixels at default. This can result in some extra calculations on images of certain dimensions, since some on the work groups operate on pixels outside the image. This must be considered carefully when choosing the group shape.

Floating point operations required

Due to the overhead some additional floating point calculations are required, depending on the size of the windows. The amount of FLOPs needed for each lifting step is:

The number of FLOPs needed to perform the lifting algorithm on an image is found in equation (4.16) in section 4.1.1, and is rewritten here:

$$\text{RC}_{\text{layer}_{\text{lifting}}} = M \cdot N \cdot 14 \text{ FLOP} \tag{5.30}$$

where M is the number of rows in the image, N the number of columns, and n is the window_width. The $3N/n$ FLOP are the result of the overhead from each lifting and scaling step, shown as the gray colored fields in figure 5.10. Since there are 4 lifting steps

and 2 scaling steps, the extra overhead adds up to $18N/n$ FLOPS per row. This results in:

$$\mathrm{RC}_{\mathrm{FLOP}_{\mathrm{liftingBlock}}} = \mathrm{RC}_{\mathrm{layer}_{\mathrm{lifting}}} + 18\frac{N}{n} \tag{5.31}$$

$$= M \cdot N \cdot 14 + 18\frac{N}{n}\ \mathrm{FLOP} \tag{5.32}$$

Since a predict and update step always is performed only on half of the pixels per step, the number of lifting steps performed at each step per line is $N/2$. For the 4 lifting steps, this results in:

$$\mathrm{Lifting}_{\mathrm{FLOP}}(M,N,n) = 4 \cdot M \cdot \left(\frac{1}{2}N + 3 \cdot \frac{N}{n}\right) \tag{5.33}$$

$$= M \cdot \left(2N + 12 \cdot \frac{M}{n}\right) \tag{5.34}$$

$$\mathrm{Diff}_{\mathrm{FLOP}_{\mathrm{one\_step}}}(M,N,n) = \mathrm{TotalBlock}_{\mathrm{FLOP}} - \mathrm{RC}_{\mathrm{FLOP}_{\mathrm{lifting}}} \tag{5.35}$$

$$= 18 \cdot \frac{N}{n} \cdot M. \tag{5.36}$$

These calculations only account for the wavelet transform in the horizontal direction, meaning that the numbers are twice as large, for a 2D DWT, such that the overhead becomes:

$$\mathrm{Diff}_{\mathrm{FLOP}}(M,N,n) = 36 \cdot \frac{N}{n} \cdot M \tag{5.37}$$

per level of decomposition. It should be noted, that for every additional level of decomposition, both M and N got halved, reducing the number of needed calculations with $\frac{1}{4}$ for every level.

Improvements

The previously shown algorithms need to store the data twice for each level of decomposition; one for the horizontal and one for the vertical direction. This can be omitted by performing the decomposition in both directions in one pass, by expanding the algorithm illustrated in figure 5.9 to two dimensions. The modified windows and group shapes are shown in figure 5.12. In this case, the output LL, LH, HL and HH are generated at once. These have to be stored in different places in the global memory, as shown in figure 5.12.

This has the same overhead in the calculations as in the first algorithm, but only half of the reads and writes to the global memory. Additionally the need to transpose the matrix in between each kernel execution is eliminated. This gives an additional performance gain, since the reads and writes can be performed aligned to the memory layout.

Figure 5.13 shows the window from the image in one working group on the GPU. The overhead is however not shown. The block to the left shows the original image, the middle block shows the location of the results, s and d, after one horizontal pass, and the block to the right shows the result after one vertical pass. This results in the LL, LH, HL and HH, which are sorted into the output image, as shown in figure 5.12. The algorithm corresponds to the algorithm 5.3 executed twice; first horizontally and then vertically.

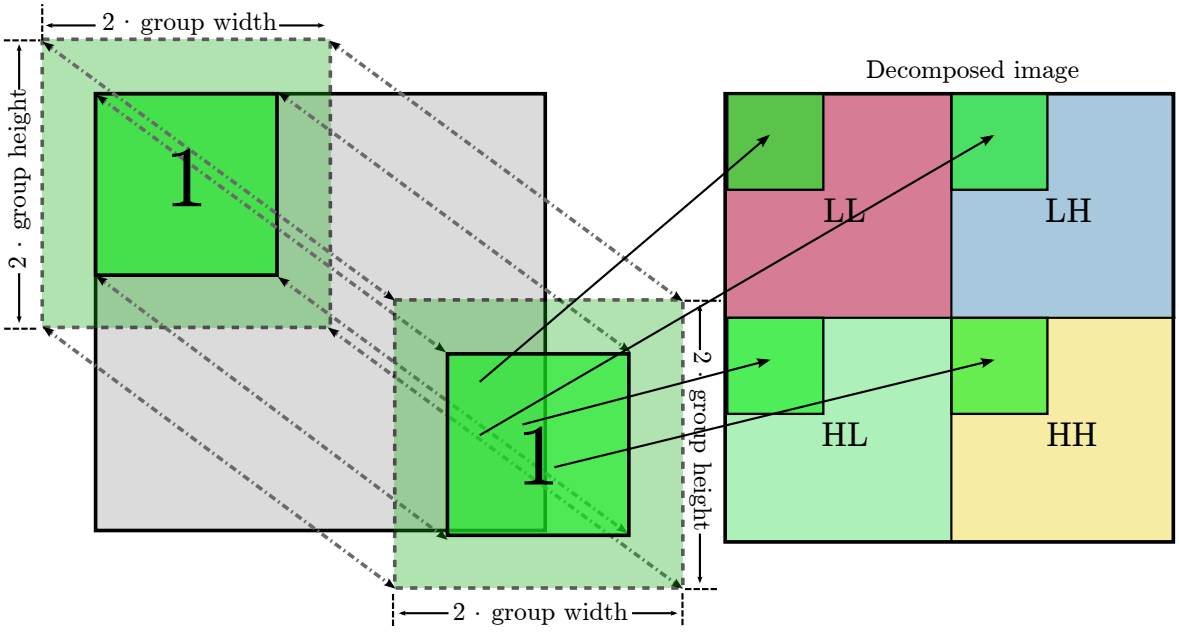


Figure 5.12. The group shapes and window sizes for the improved algorithm, as well as the mapping of the results to the output matrix.

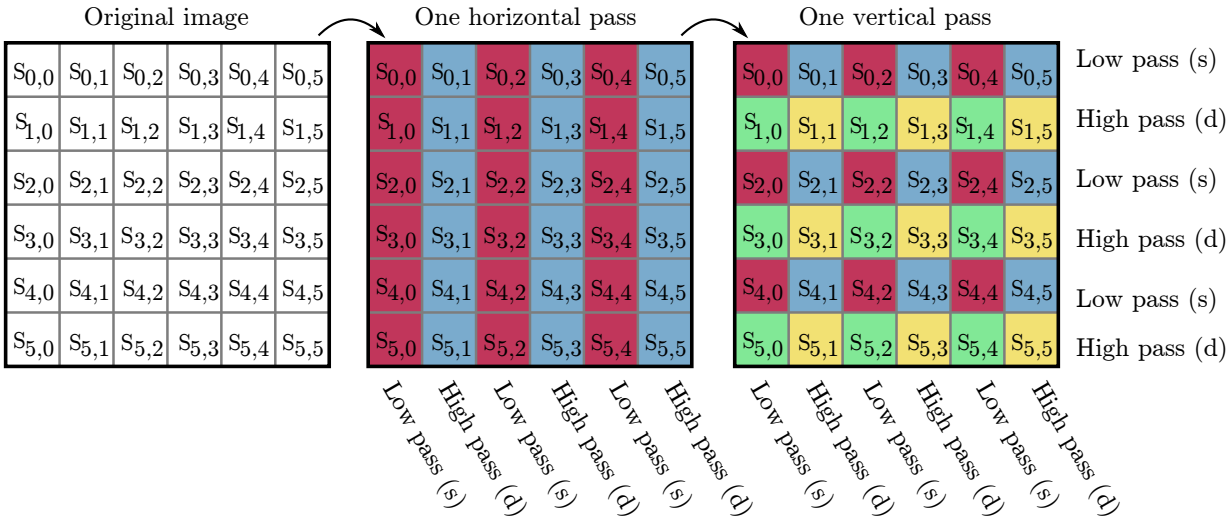


Figure 5.13. The location of the pixels when the lifting i performed in-place. The colors indicate whether it is LL, LH, HL or HH.

Reconstruction

The reconstruction using lifting is also implemented using both the improved and non-improved code. The reads and writes to and from the global memory are reversed, since the data is distributed in the four subbands of the image, as shown on the right side of figure 5.12. The algorithm itself is based on a sequential implementation of the inverse lifting steps shown in equations (2.122) to (2.127) from section 2.5.

During the implementation, it showed, that `matplotlib.pyplot` has issues when showing images with values outside the range 0 to 1. For this reason, it is chosen to truncate all values above 1 to 1 and below 0 to 0. This is done within the kernel, and is only done on the final level of reconstruction.

Performance

The chosen algorithm and the improved version of it are implemented and tested against each other on different sizes of gray scale images. This is done using the python script **testlift.py**. The resulting execution times for the decomposition and reconstruction from both algorithms are shown in figure 5.14. These are measured over 200 iterations, performing a full decomposition and reconstruction. The figure shows, that the improved algorithm performs significantly faster, and that the lead increases as the image size increases. The decomposition algorithms perform slightly better than the reconstruction.

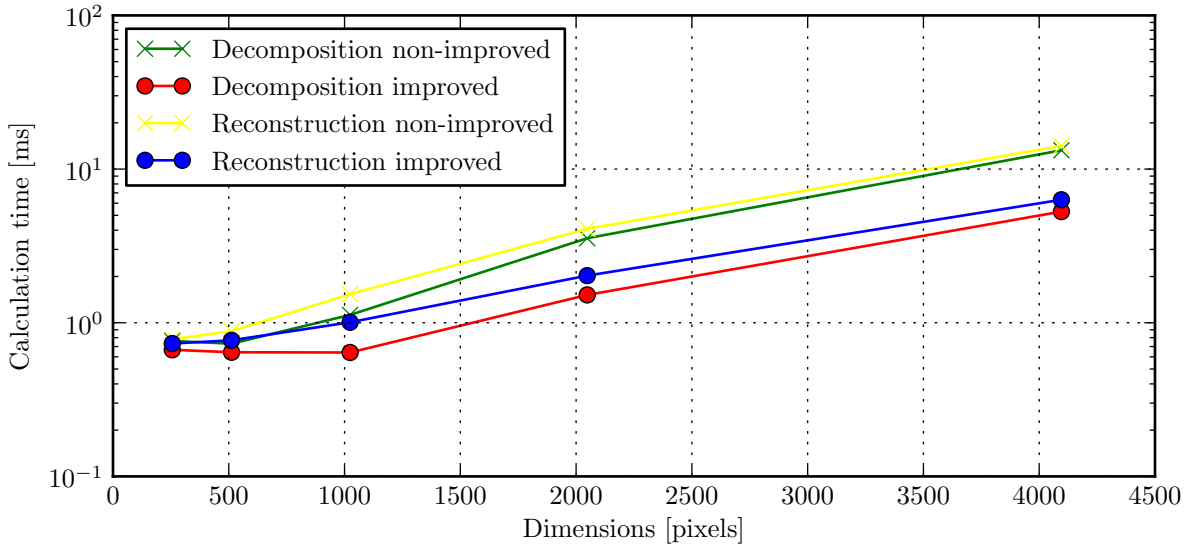


Figure 5.14. Average computation times for both decomposition and reconstruction using both lifting algorithms at different square resolutions with the necessary data available on the GPU.

This can have different explanations, such as the memory read and write patterns, these differ from the algorithms and can have a huge impact on GPU performance. The truncation of the pixel values also adds additional execution time. The number of floating point operations are exactly the same, since the reconstruction algorithm uses the reverted lifting steps. This means, that these should perform the same. The times are measured with the data available on the GPU and are raw calculation times.

Figure 5.15 shows the total computation times, where the transfer to (HtoD) and from (DtoH) the GPU memory are included. The time needed is much higher than the raw computation time, shown in figure 5.14. This indicates the transfer time between the CPU and GPU has the largest impact on the performance. It is also clear, that the memory transfer speed is the same for both algorithms, which also should be the case since they both use the same transfer methods.

Summary

The test shows, that the improved algorithm performs significantly better than the first implemented algorithm. The main reasons are the reduction of reads/writes to and from

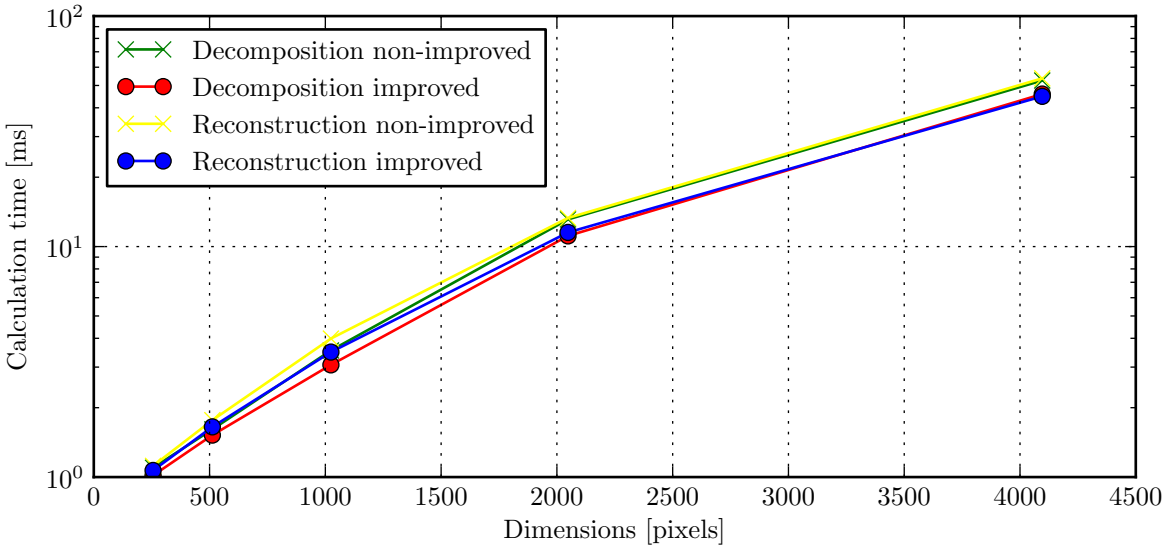


Figure 5.15. Total computation times inclusive transfer to and from the memory for decomposition and reconstruction using both lifting algorithms at different square resolutions including the time needed to transfer the data to and from the GPU.

the global memory as well as a better memory access pattern, since most reads and writes can be performed almost aligned to the memory.

Color images

The improved algorithm is expanded to feature three dimensional arrays, upgrading it to perform a DWT on color images. This is simply done, by adding a third dimension to the grid. The working groups still operate on two dimensions. The code for this implementation can be found in **__CDF97Color.py** and the kernel in **CDF97ColorKernel.c**

5.6 Padding HD images

When decomposing images, a full decomposition can be obtained when the LL part of the image is stored in one pixel. To obtain this, it is required that the dimensions of the image are square and a power of two. When decomposing High Definition (HD) images, the dimensions do not equal and they are not a power of two. Due to this it is often only possible to obtain 3 or 4 levels of decomposition.

One way to obtain a full decomposition of a HD or other odd shaped image, is to pad the image up to the nearest square dimension. This introduces a lot of extra calculations, since the entire image has to be decomposed. Here it is chosen to pad the images with zeros, since these contain no energy. When a full decomposition of a 1280×720 or 1920×1080 HD image is needed these have to be padded up to 2048 x 2048 pixels. A lot of extra overhead is added if the padding is done before the image is transferred to the GPU. For this reason, it is chosen to pad the image on the GPU. When a decomposition is performed on a padded image, the image is transferred to a buffer, the same size of

the image on the GPU. After this the padding kernel is called, which copies the image to a square buffer, the size of the padded image. All pixels outside the image region are replaced by zeros. The kernel code is shown in listing 5.7.

```
1  /*Defines to access the flat memory with three coordinates*/
2  #define PADDED(z,y,x) padded[ZDIM * (x + offset_x + padded_xdim * (y +
       offset_y)) + z]
3  #define IMAGE(z,y,x)  image[ZDIM * (x + XDIM * y) + z]
4
5  __kernel void padding_color(__global float *image, __global float
       *padded, unsigned offset_x, unsigned offset_y, unsigned padded_xdim )
6  {
7    /*Get global grid location*/
8    int Glox = get_global_id(0);
9    int Gloy = get_global_id(1);
10   int Gloz = get_global_id(2);
11
12   /*If location is inside image and offset, copy image, else write 0*/
13   if( Glox < XDIM && Gloy < YDIM )
14   {
15     PADDED(Gloz,Gloy,Glox) = IMAGE(Gloz,Gloy,Glox);
16   }
17   else
18   {
19     PADDED(Gloz,Gloy,Glox) = 0;
20   }
21 }
```

Listing 5.7. padding kernel placed in kernel file padding.cl. X/ZDIM are the original image size.

This results in some additional computation time on the GPU since the extra kernel has to be executed, but it reduces the Device to Host (DtoH) transfer time significantly compared to padding the image before it got transferred to the GPU. A similar kernel is used to reconstruct a padded image, so only the image part is fetched from the GPU removing the overhead created by the padding.

Implementation of padding

The padded DWT is implemented as sub-modules of the lifting and filtering implementations. It overrides the `__init__()` method and the `decompose()` and `reconstruct()` methods by adding the padding kernel to the queue. The implementations using padding can be found in **lifting/__CDF97Padded.py** and **filtering/__CDF97Padded.py**, where the kernel is located in **padding/padding.cl**

5.7 Generating test images

To obtain throughout test results, various image sizes must be used. A few images have been chosen and resized to the desired resolutions. A list of the images used for testing is found in appendix A. The tests uses both square and HD size images, and therefore two sets of images are found.

For the purpose of resizing the images, the Python script **generateImages.py** is

used. This uses the module `Image`, and creates output images of the file type PNG[2], in the resolutions specified in appendix A. The square sizes are 2^n, and are from 256×256 to 8192×8192. The HD sizes have the 16:9 format and, are from 852×480 to 7680×4320. If the image dimensions (width and height) do not have the same scale ratio as the desired resolutions, the output image is stretched. This makes it possible to also use odd size images as the test images are always generated in the desired resolutions.

5.8 Verification

During the implementation and prior to the benchmark, it is necessary to check whether the implemented algorithms provide the correct DWT. In this it is case insufficient to only check the error between the reconstruction and the original image, since a wrongly implemented algorithm with a matching reconstruction and decomposition, can provide a proper image. For this reason it is chosen to compare the decompositions to a MATLAB implementation of the CDF 9/7 by Pascal Getreuer[3]. This implementation is based on the lifting scheme, and uses the same scaling coefficients as used in this report [17].

The verification of the algorithms is made using a Python module, linking the matlab implementation to a Python interface. This generates all levels of decomposition for a given image and stores them in a HDF5 file. This file is automatically generated if it does not exist, and has the same file name as the image with the "h5" extension. The module can be found in **__imgtest.py**.

Using the verification module

When the verification module is initialized with an image, it is possible to fetch the image using the method `getImage()`. This assures that we operate on the same image as MATLAB used for the decompositions. It is also possible to fetch a specific decomposition level using `getDecomposition(level)`.

When decompositions have to be compared, the method `compareDecomposition(X,level)` is used. Here X is the decomposition of the image, and level is the levels of decomposition. This returns the Peak Signal to Noise Ratio (PSNR) of X compared to the decomposition at the specified level, as shown in listing 5.8.

```
1  > tester.compareDecomposition(X,3)
2
3  Comparing decomposition level 3 PSNR = 35.488544 dB
4  Block LL                              PSNR = 32.048492 dB
5  Block LL without border (5 pixels)    PSNR = 119.591832 dB
6  Block LH                              PSNR = 20.327369 dB
7  Block LH without border (5 pixels)    PSNR = 124.722319 dB
8  Block HL                              PSNR = 29.739085 dB
9  Block HL without border (5 pixels)    PSNR = 125.046394 dB
10 Block HH                              PSNR = 44.914305 dB
11 Block HH without border (5 pixels)    PSNR = 127.697789 dB
```

[2]PNG: Portable Network Graphics
[3]http://www.getreuer.info/home/waveletcdf97

Listing 5.8. Example of the output of `compareDecomposition(X,level)`.

In this case, the comparison is made for a three level decomposition of **grasshopper1920_1080.png**, which can be obtained by running the file **__CDF97Color.py**. The first PSNR in listing 5.8 is measured for the entire image. 35.5 dB is relatively low compared to the MATLAB reference implementation [18]. The next 8 values are for the different sub bands of the decomposition, with and without a border of 5 pixels. Here it shows, that the PSNR increases when the border is excluded from the calculations. This is due to the implemented border handling method, where we use periodicity by repeating the image at the borders, the Pascal Getruer implementation uses a form of extrapolation at the boundaries.

The reconstructions can be compared using the `compareReconstruction(Y)`, where Y is the reconstructed image. This results in the output shown in listing 5.9.

```
1  > tester.compareReconstruction(Y)
2
3  Comparing reconstruction      PSNR = 45.420947 dB
4  Block width 5                 PSNR = 59.544693 dB
5  Border width 10               PSNR = 66.061262 dB
6  Border width 15               PSNR = 85.483983 dB
7  Border width 20               PSNR = 113.471014 dB
```

Listing 5.9. Example of the output of `compareReconstruction(Y)`.

It seen how the border handling, affects the image, since the PSNR increases as we exclude larger parts the border of the image, and discard it.

6 Benchmarking

To test how well the implemented algorithms perform, a benchmark is performed. For the results to be reproducible, the benchmark is made as automatic as possible. The main script is written in Python (**benchImage.py**), and these are static in the sense that nothing is to be changed within them. To benchmark a desired setup, other scripts containing the test parameters (**benchLifting.py benchFiltering.py**), calls the functions of these.

The benchmarks are performed to find the fastest implementation, and the time needed to compress different resolutions with, and without padding. These are finally compared to a CPU implementation[1], and the speedup is calculated. The benchmarks on the GPU are performed on 3-layer color images, whereas the CPU benchmark is performed on randomly generated data in a 2D matrix. Due to this, the performance for color images on the CPU is estimated by multiplying the resulting times for a 2D matrix by 3.

When benchmarking, the internal speeds of the transform kernels and data transfer is measured. These results are then the effective transform speeds. To obtain a thorough benchmark various image sizes are used. The tests are run for both lifting and filtering, and the results are compared in the end.

6.1 Considerations

Certain precautions are necessary to ensure reproducible and valid results. These are explained in this section.

Warm up and number of runs

When executing an implementation on a complex system, like a PC, the performance may vary much due to e.g. the priority in the execution queue, other running applications and initializations of host and device. The result from the first run is therefore not a valid benchmark result. To avoid this "warm up runs", both for decomposition and reconstruction are performed for each resolution before the actual benchmarking starts and results are stored. Every test is repeated multiple times to average over possible variations in the performance.

Error checking

When benchmarking it is time consuming to check manually if every transform was performed without errors. The implemented algorithms are verified prior to the benchmark. Since the benchmark focuses on raw performance, no verification of the individual decompositions and reconstructions is performed while benchmarking. Instead

[1] DWT97.cpp by Gregoire Pau and Tobias Lindstrøm Jensen

the sums of all the pixel values are stored, and in the end it is checked, whether the variance of these is zero, on which it can be assumed, that the results are the same.

Switching the image

If an error occurs when e.g. the image is transferred to the device, the old data (of previous run) could be used for the transform, producing false time results. This would not be visible in the sum of pixel values if the same image is used, and might not be discovered. Therefore when performing the decomposition, the script will in between each run switch to a new image. This ensures that new data is used for the transform each time, and thereby no false results. The possibility that such an error is discovered in the sum is thereby also larger, as the images are different from each other.

6.2 Image benchmarking

For the image benchmark the main script **benchImage.py** containing the function `benchmark()` has been written. The function takes the five arguments: `testImages`, `res`, `runs` and `method padding`. The argument `method` specifies whether the lifting or filtering algorithm is used. If `padding` is set to `True`, the padded implementation of the selected method is used. For the choice given in `method`, the decomposition and reconstruction is benchmarked for all possible levels of the specified resolutions in (`res`).

As explained earlier in the considerations, the test must be repeated several times for each decomposition and reconstruction. This is specified by the argument `runs`, and it is chosen to perform one thousand runs. The arguments `testImages` and `res` are explained later. The results of the benchmark are stored in a HDF5 file using the Python module `h5py`. The pseudo code for the benchmarking is seen in algorithm 6.1.

6.2.1 Execution times

The internal times of the different steps in the implementation are measured. The timing must be done carefully, as the actual wavelet transform calculations are (only) a part of a larger Python implementation. Therefore the timing functionality is implemented as a part of the modules `lifting` and `filtering`. The times are returned when the decomposition and reconstruction have finished.

The times are measured using the Python `time.time()` function from the `time` module, and are implemented in the DWT modules. The timing is enabled by adding the parameter `timer = True` when the modules are initialized. The following times are measured and returned:

- **HtoD**: Transfer time from host to device.
- **DtoH**: Transfer time from device to host.
- **calc**: Time used for calculating on device.

Algorithm 6.1 Algorithm for **benchImage.py**.

```
for All resolutions do
    Load images.
    Initialize GPU.
    for All possible levels do

        3 warm up decompose runs.
        for Number of runs do                          ▷ Benchmark decomposition.
            Decompose.
            Get internal times and store in temp array.
            Calculate and store sum of decomposed pixel values in temp array.
        end for
        Store results in file.

        Create and temporary decomposition of each image.  ▷ Used for reconstruction.
        3 warm up reconstruct runs.
        for Number of runs do                          ▷ Benchmark reconstruction.
            Reconstruct.
            Get internal times and store in temp array.
            Calculate and store sum of reconstructed pixel values in temp array.
        end for
        Store results in file.

    end for
    Release GPU memory.
end for
```

- **total**: Total time.

6.2.2 Images and resolutions

A requirement from chapter 1 is that the implementation must run on full HD, 1920×1080 pixel, color images. To obtain detailed benchmark results it is not enough only to test using this resolution. It is chosen to run on six different HD resolutions, to see how the performance scales dependent on the this. The resolutions vary from 852×480 to 7680×4320 pixels, and are all in the 16:9 format. To benchmark with HD resolutions, the script **benchHD.py** is used.

An issue with HD resolutions is, that a full decomposition cannot be obtained. This might result in poor data compression, and it could therefore be desired to pad the image to a size that results in full decomposition. Due to this, the implementation is also benchmarked for padded HD resolution images.

The padding implementation requires additional memory on the GPU equal to the size of the image. This has to be added to the memory, used by the lifting and filtering implementations. When using large images at the resolution of 4520×2540, this results in a total memory consumption of:

$$\mathrm{PAD}_{\mathrm{mem}}(4520, 2540, 8192) = (3 \cdot 4520 \cdot 2540) \cdot 4 + 2 \cdot (3 \cdot 8192 \cdot 8192) \cdot 4 \approx 1748\,\mathrm{MB}. \tag{6.1}$$

When an image of the resolution 7680×4320 is used, the required memory is approximately 2009 MB for the data only. To this some additional overhead must be added, which exceeds the 2048 MB available on the GPU (see section 5.1. Therefore the resolutions for padded HD images are from 852×480 to 4520×2540. This is benchmarked using the script **benchHDpad.py**.

In most cases where implementations are benchmarked, square images are used. Results for an implementation may vary much dependent on the format of the images, and to obtain results for easy comparison with other implementations, square power of two (2^n) images are also benchmarked. The resolutions of these vary from 256×256 to 8192×8192 pixels. The test script used for this is called **benchSquare.py**.

The desired resolutions are given in the argument `res` as a list of tuples, each containing the width and height of the image. The images used for testing are given as a list of strings in the argument `testImages`. These are all generated using the script **generateImages.py**, which is explained in section 5.7

As an example: If `res` = [(256,256),(512,512)] and `testImages` = ["test"] the images used in the benchmark are **test256.png** and **test512.png**. For HD resolutions i.e. `res` = [(1920,1080)], the image used is **test1920_1080.png**. The images and the different resolutions used for testing are specified in appendix A.

6.3 Image results

As described earlier, is the benchmark performed on a series of images with different formats. The three image formats tested are: HD, padded HD and squared images. Shapes are specified in appendix A.

HD image test results

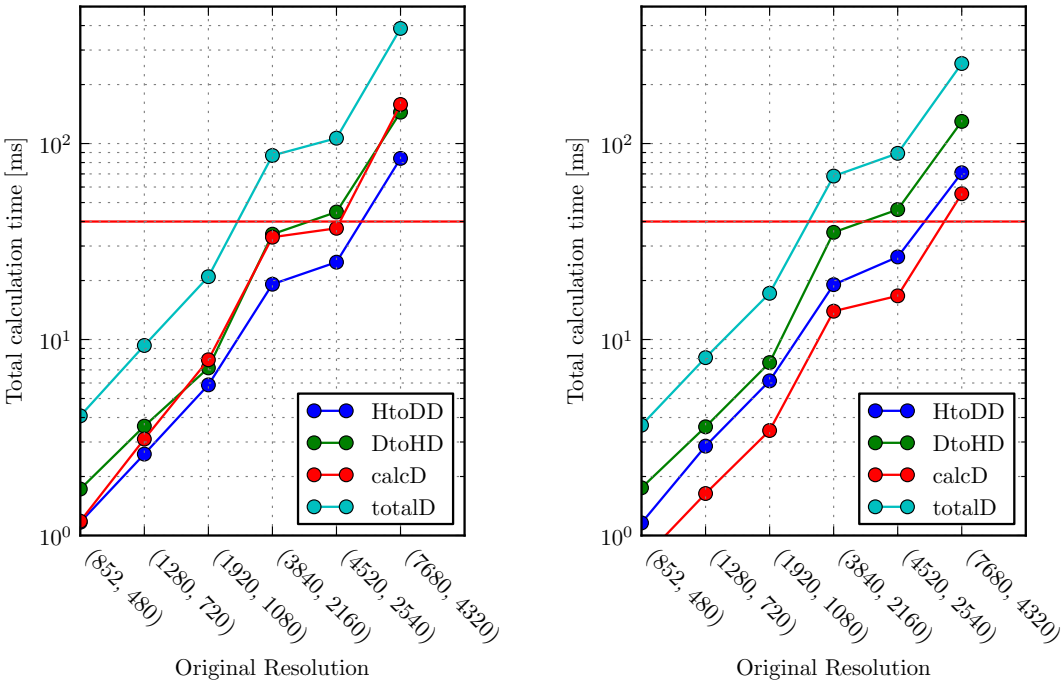
When using HD images for the decomposition, only a limited amount of decomposition levels are performed, according to section 5.6. The decomposition levels achieved for the different resolutions are shown in table 6.1.

Resolution	Levels
852×480	2
1280×720	4
1920×1080	3
3840×2160	4
4520×2540	2

Table 6.1. Decomposition levels for the respective HD resolutions.

On figure 6.1(a) and 6.1(b) the benchmark results for HD pictures without padding are shown. Histograms for the the total times can be found in appendix B. The requirement of at most 40 ms is marked in the plot with a horizontal red line. Here it is seen that the

data transfers Host to Device (HtoD) and DtoH do not differ for the two algorithms as expected, since the same amount of data is used. The calculation time (calc) does, however, vary for the two implementations. As explained in the lifting implementation in section 2.5, does the lifting algorithm only need about half the floating point operations per pixel. As a result of this, the raw calculation time for the lifting algorithm is much lower that filtering, and the total calculation times are, therefore, also higher for the filtering. According to the requirement, both implementations are able to perform the DWT on a HD resolution at 1920×1080 pixels below the stated 40 ms. Above this resolutions both implementations exceed the requirement of 40 ms, mostly because the data transfer times scale exponential at higher resolution.



(a) Filtering

(b) Lifting

Figure 6.1. Benchmark for HD images with (a) Filtering and (b) lifting. The requirement of 40 ms is marked with a horizontal red line.

Padded HD images

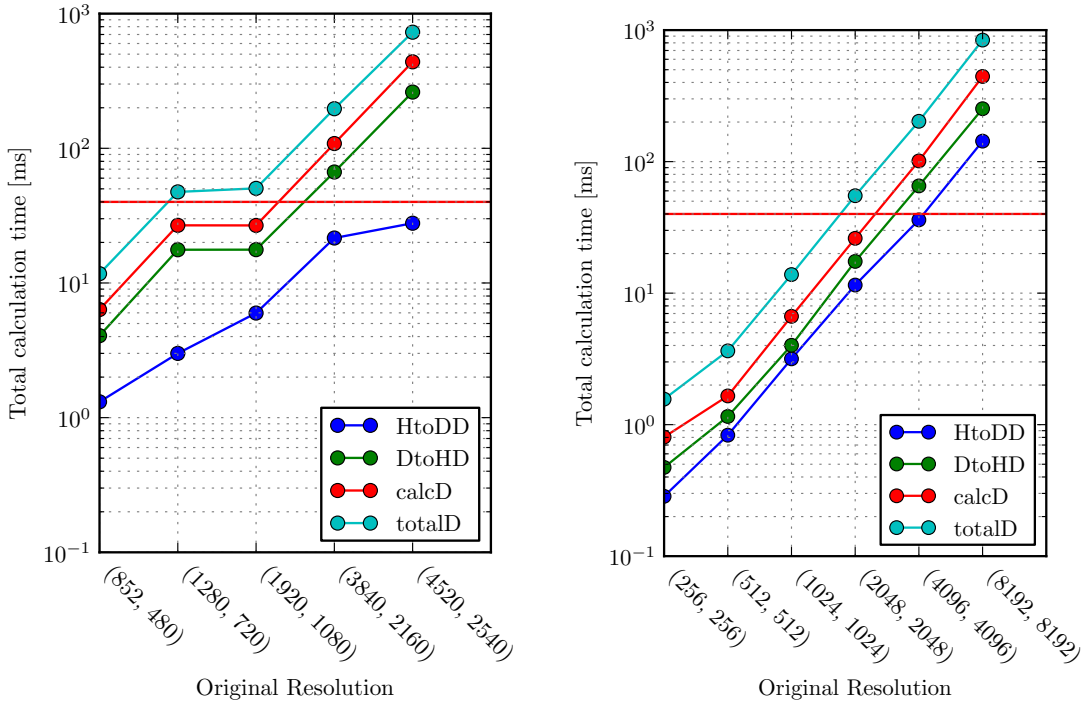
Figures 6.2(a) and 6.2(b) show the benchmark results for filtering on padded and squared images, respectively. Figures 6.3(a) and 6.3(b) show these for lifting. The padding operation is included in the calculation time, since it is performed on the GPU. The resolutions are padded to the closest 2^n resolution (see table 6.2 for reference). The padded images are in these cases very sparse, which result in a significant amount of unnecessary calculations in order to achieve a full decomposition. Since both implementations use the same padding algorithm, the result can still compared.

The HtoD transfer times are, as seen on figures 6.2(a) and 6.2(b), increasing for each resolution, but the DtoH transfer time is the same for (1280×720) and (1920×1080)

Resolution	Padded resolution
852×480	1024×1024
1280×720	2048×2048
1920×1080	2048×2048
3840×2160	4096×4096
4520×2540	8192×8192

Table 6.2. Decomposition levels for the respective resolutions.

pixels, since these are padded to the same resolution of 2048×2048 pixels. The calculation times remain the same for these as well. The total time does, however, increase with the increased HtoD transfer time. Likewise, on figure 6.3(a) and 6.3(b), this increase in the execution times is seen.

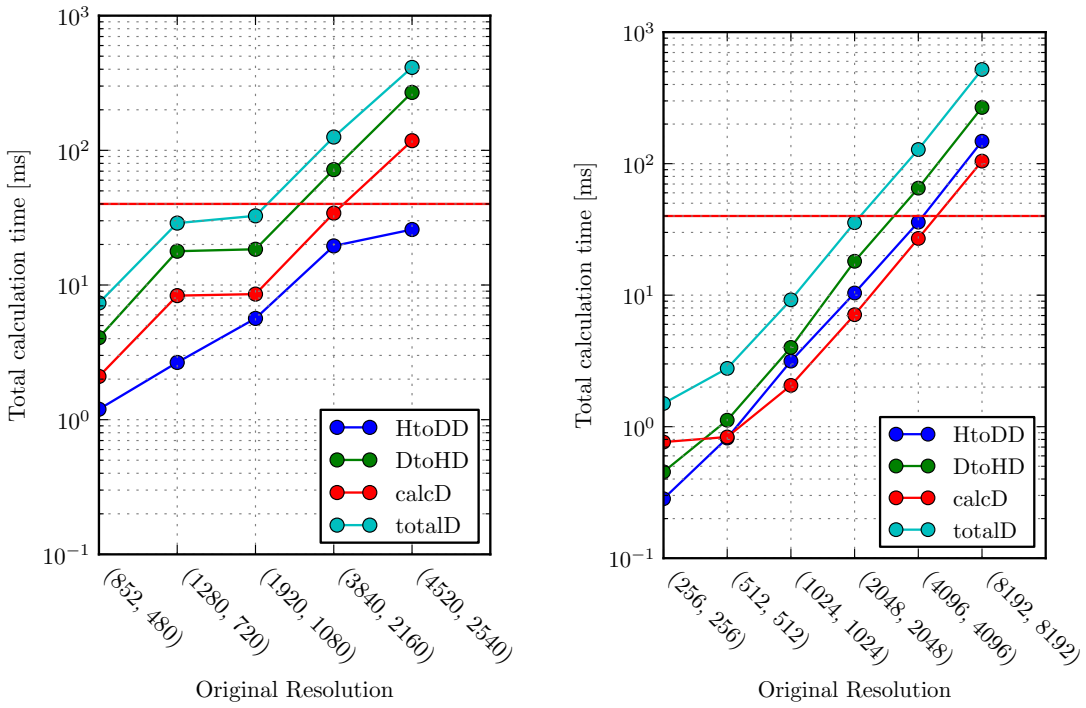


(a) HD Padded

(b) Square

Figure 6.2. Benchmark results for filtering a (a) HD padded images and (b) Squared image. The requirement of 40 ms is marked with the horizontal line.

Both filterings 6.2(b) and liftings 6.3(b) execution time increase exponentially each time the resolution is doubled for squared images, due to an exponential increase in the total amount of pixels. The main thing to note here, is that with the lifting implementation it is possible to perform a full decomposition on resolutions up to 2048×2048 pixels, whereas the filtering algorithm exceeds the 40 ms requirement at resolutions above 1024×1024 pixels, limiting it to a HD padded resolution of at most 852×480 pixels.



(a) HD padded

(b) Square

Figure 6.3. Benchmark results for lifting a (a) HD padded images and (b) Squared images. The requirement of 40 ms is marked with the horizontal line.

CPU implementation versus GPU implementation

On figure 6.4 and 6.5 the benchmarks for the maximum achievable decomposition levels versus dimensions are shown for HD and squared dimensions, respectively. The total and calculation times are plotted for the two GPU implementations. The CPU implementation is only benchmarked on gray scale images. This is because the implementation is limited to two dimensions, but since the implementation is sequential it is estimated that the calculation time needed for all color layers is around three times this. In practice this might differ slightly. The estimated calculation times for color are also plotted for reference. The average time consumed and speedup, compared to a color decomposition on the CPU, are shown in table 6.3. Here it is seen, that the speedup for the GPU generally increases as the resolution increases. The average speedup for lifting is 9.82 times, and for filtering 7.80 times.

Resolution height	480	720	1080	2160	2540	4320	**[pixels]**
Resolution width	852	1280	1920	3840	4520	7680	**[pixels]**
CPU gray scale	7.15	19.72	56.09	207.70	374.08	1214.04	**[ms]**
CPU color	21.46	59.16	168.28	623.11	1122.24	3642.12	**[ms]**
Lifting	3.66	8.08	17.21	68.23	89.14	256.09	**[ms]**
Filtering	4.08	9.32	20.92	86.91	106.50	386.82	**[ms]**
Speedup lifting	5.87	7.32	9.78	9.13	12.59	14.22	**X**
Speedup filtering	5.26	6.35	8.04	7.17	10.54	9.42	**X**

Table 6.3. Average times for the CPU and GPU algorithms including memory transfer for HD resolutions.

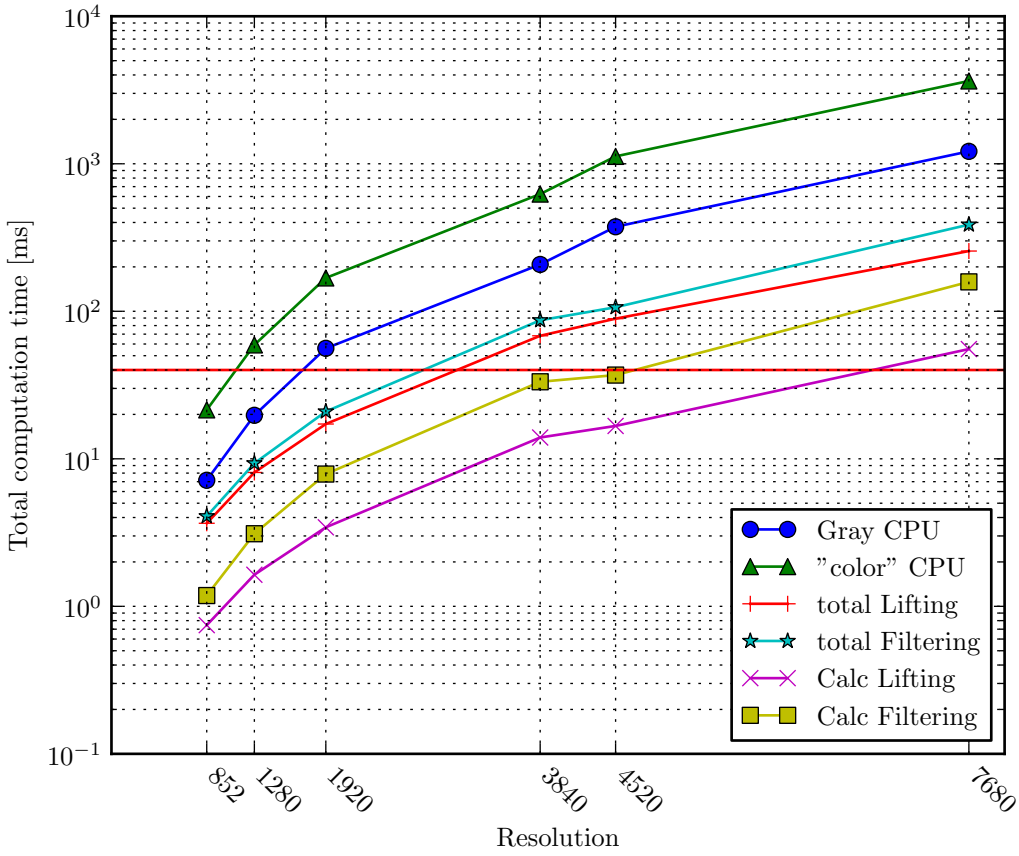


Figure 6.4. Total time and calculation time for the two implemented algorithms and total time for CPU implementation on HD images. The requirement (40 ms) is marked with the horizontal line.

Table 6.4 shows the raw calculation times for the GPU compared to the CPU implementation. Here it is seen, that the GPU is significantly faster. The lifting algorithm is on average 48.58 times faster than the CPU on color images, whereas filtering is 21.77 times faster. It is also seen, that lifting on average is 2.23 times faster than filtering, when only considering the raw calculation times.

Resolution height	480	720	1080	2160	2540	4320	**[pixels]**
Resolution width	852	1280	1920	3840	4520	7680	**[pixels]**
CPU color	21.46	59.16	168.28	623.11	1122.24	3642.12	**[ms]**
Lifting	0.75	1.64	3.43	13.95	16.70	55.45	**[ms]**
Filtering	1.18	3.11	7.88	33.28	36.95	158.29	**[ms]**
Speedup lifting	28.79	36.13	49.02	44.68	67.18	65.68	**X**
Speedup filtering	18.14	19.05	21.35	18.72	30.37	23.01	**X**

Table 6.4. Average raw calculation times for the CPU and GPU algorithms without memory transfer for HD resolutions.

As seen on figure 6.4 are the two implementation close together and the CPU time for gray scaled not far from them. At the resolution 720×1280 pixels are both implementations benchmarked faster than the CPU implementation on the same

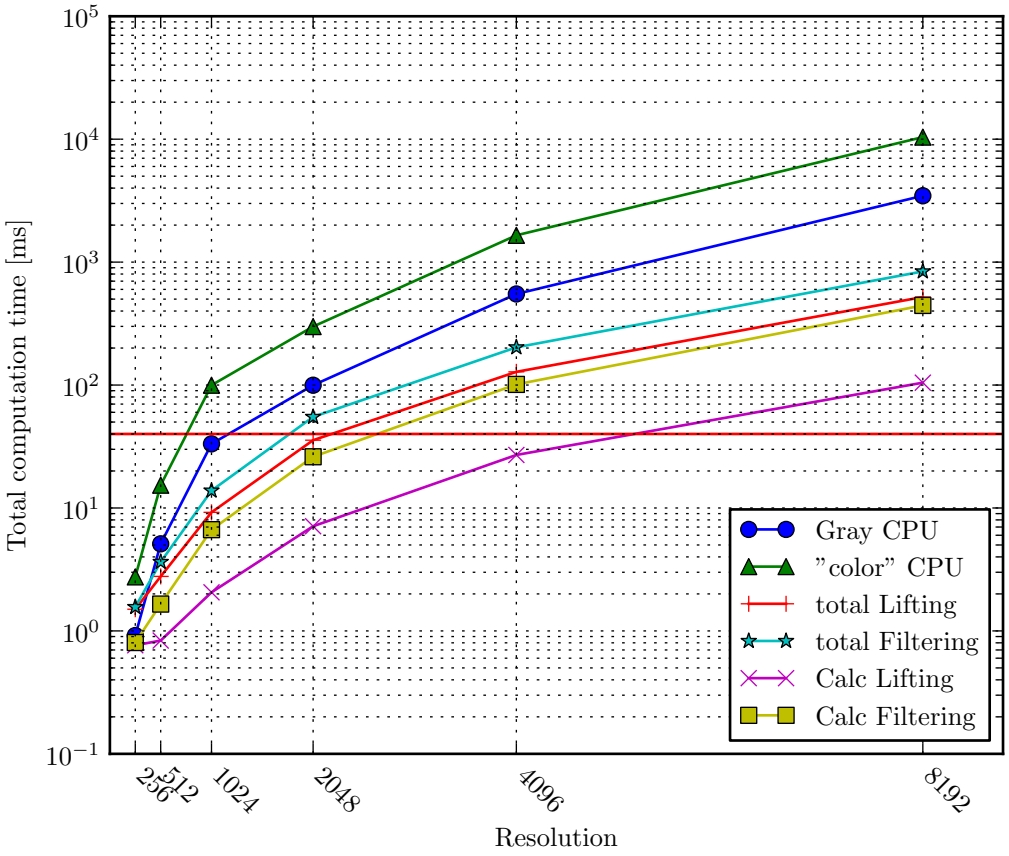


Figure 6.5. Total time and calculation time for the two implemented algorithms and total time for CPU implementation on squared images. The requirement (40 ms) is marked with the horizontal line.

resolution. After 1080×1920 pixels do both GPU implementations used around half the total time of the CPU implementation. This is the case for both HD images 6.4 and squared images 6.5. Another important thing to notice is, as the only wavelet implementation, lifting is able to transform a full DWT on 2048×2048 pixels, which is the padded resolution corresponding to a full HD resolution.

Resolution height	256	512	1024	2048	4096	8192	**[pixels]**
Resolution width	256	512	1024	2048	4096	8192	**[pixels]**
CPU gray scale	0.92	5.11	33.24	99.69	550.62	3464.90	**[ms]**
CPU color	2.75	15.33	99.71	299.08	1651.86	10394.70	**[ms]**
Lifting	1.50	2.78	9.22	35.60	128.07	520.19	**[ms]**
Filtering	1.56	3.64	13.84	55.06	202.65	840.06	**[ms]**
Speedup lifting	1.83	5.52	10.82	8.4	12.9	19.98	**X**
Speedup filtering	1.76	4.21	7.20	5.43	8.15	12.37	**X**

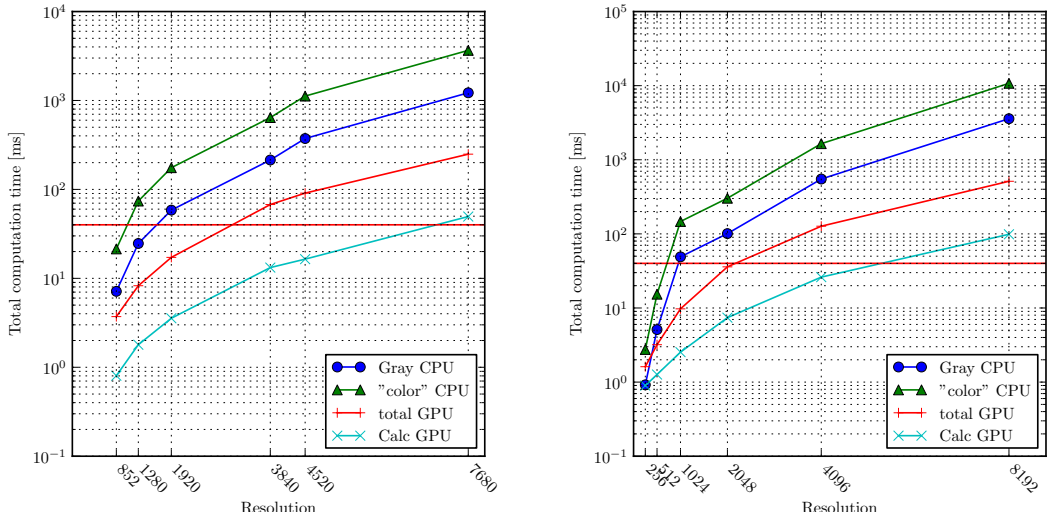
Table 6.5. Average times for the CPU and GPU algorithms including memory transfer for square image resolutions.

The total computing times for square image resolutions are shown in table 6.5. At the lowest resolution of 256×256 pixels, the speedup is only 1.76 and 1.83 for filtering and

lifting, respectively. When the resolution increases, the speedup follows the same pattern as for HD resolution images, seen in table 6.3. The average speedup for lifting is 9.91 times for lifting and 6.52 times for filtering, which is similar to the results from the HD resolutions.

6.3.1 Reconstruction

Since only one reconstruction algorithm is implemented, this is tested and compared to the CPU implementation in figure 6.6(a) and figure 6.6(b) for HD and squared images, respectively. The GPU implementation is, as seen on the figures, faster than the CPU implementation on both HD and squared images. The average speedup is 10.24 on HD images and 10.66 for square images for the GPU implementation. The detailed results can be found in appendix B.



(a) HD (b) Square

Figure 6.6. Total and calculation times for the implemented reconstruction algorithm and total time for CPU implementation on (a) HD images and (b) Squared images. The requirement (40 ms) is marked with the horizontal line.

6.4 Discussion

The benchmarks showed that the GPU is significantly faster than the CPU implementation. Average speedups of around 10 times are achieved when using the lifting implementation and the transfer times are included. Filtering is slightly slower, gaining a speedup of around 7 times. When only considering the raw calculation times, the GPU implementations are significantly faster, gaining speedups of 48.58 times for decomposition using lifting and 21.77 for filtering. According to the analysis in chapter 4, lifting should require less than half the FLOPs of filtering. The lifting implementation is furthermore

optimized, by only requiring half the reads/writes to the global memory compared to filtering, which further enhances the performance. The speedup of the GPU compared to the CPU increases as the resolution increases.

It should be noted, when considering the CPU implementation, that the CPU implementation is limited to one CPU core and could, when multi-threaded, achieve a theoretical speedup of 6 times on the Intel Core i7 970. This is though still slower than the GPU implementations.

Part III

Closure

7 Acceptance test

The purpose of the acceptance test is to verify that the requirements specified in chapter 1 on page 1 is fulfilled. As described in chapter 5 the implementation is developed to transform color images, and from the benchmark of chapter 6 and the verification described in section 5.8 it is clear that the CDF 9/7 transform function for 1920×1080 HD images. By this the first demand is fulfilled.

The second and third requirement specify that the wavelet transform must be performed at 25 fps (within 40 ms), as the purpose is video coding. The implementation performs the wavelet transform on one image at a time, and returns the result. Therefore, the performance concerning this is testes by invoking the transform with a sequence of images (video frames). The result from this are the outer times, with the overhead introduced by Python.

7.1 Video script

Instead of developing a camera interface that collects the video frames, this is simulated using a list of 2D `numpy` arrays, stored in host memory. For this test the HD resolution 1920×1080 is used, as this is a requirement from chapter 1 on page 1. No padding is used allowing three levels of decomposition.

The script written for the video test is called **acceptVideo.py**. The function of the script is called `acceptVideo()` and takes the arguments `frameFile`, `method`, `level`, `padding`, `maxFrames` and `runs`. The argument `method`, specifies if lifting or filtering is used, and `runs` the number of times the decomposition and reconstruction is performed for the list of video frames. This is chosen to be one hundred. `padding` specifies if padding is used, which it is not, as this is not a requirement. `level` specifies how many decomposition levels to perform. As it is possible to perform three levels of decomposition without padding, it is chosen to do so. The argument `frameFile` is a string containing the file name in which the video frames are stored. `maxFrames` is the maximum number of frames that will be loaded from the frame file and used for the test. This makes it possible to adjust the number of frames according to the host memory of the test platform. The pseudo code for the video test is seen in algorithm 7.1.

7.1.1 Video frames

To create the array of video frames a MATLAB function called `saveVideoFrames()` contained by the script **saveVideoFrames.m** is written. This uses the MATLAB function `VideoReader()` to convert the encoded data of a video into complete "image" frames. These frames are saved in a HDF5 file using the MATLAB function `hdf5write()`. To extract the frames from a video and save them, the function is called with the file name

Algorithm 7.1 Algorithm for **acceptVideo.py**.

```
Load video frames from file into host memory.
Initialize GPU.

for Number of frames do                          ▷ Decomposition warm up runs.
    Decompose.
end for
for Number of runs do
    for Number of frames do                      ▷ Decomposition test.
        Time start.
        Decompose.
        Time stop.
        Store times in temp array.
    end for
end for
Store results in file.

for Number of frames do                          ▷ Reconstruction warm up runs.
    Reconstruct.
end for
for Number of runs do
    for Number of frames do                      ▷ Reconstruction test.
        Time start.
        Reconstruct.
        Time stop.
        Store times in temp array.
    end for
end for
Store results in file.
```

of the video.

One image saved in the HDF5 file require:

$$(3 \cdot 1920 \cdot 1080) \cdot 4 \approx 25\,\text{MB}. \tag{7.1}$$

As it is desired to load all video frames into host memory, which is (only) 24 GB, the number of frames must be limited. If all the host memory is filled with video frames approximately

$$\frac{24 \cdot 10^9}{3 \cdot 1920 \cdot 1080 \cdot 4} \approx 964\,\text{frames} \tag{7.2}$$

can be contained.

It is desired that the test platform contains the generated HDF5 file (in static memory). For long videos the HDF5 files tend to grow large in size, and this is inconvenient to handle. If one thousand frames are saved from the video file this results in approximately 25 GB, and when using compression this is limited to approximately 5.7 GB. Therefore a maximum limit of one thousand frames are extracted from the video file. The actual number of frames used in the test is smaller, due to the limit of the host memory, and as many frames as the host memory allows are used. When testing on the reconstruction,

the decomposed frames are needed. Therefore the decomposed frames are stored in local memory and then used for the reconstruction afterwards. Because of this twice the amount of memory is needed for storing the frames. Therefore only 400 frames are used corresponding to:

$$(3 \cdot 1920 \cdot 1080) \cdot 4 \cdot 400 \cdot 2 \approx 20\,\text{GB}. \tag{7.3}$$

Initial test runs have shoved that if more frames are filled into host memory, the performance drops significantly. This might be due to cached data, stored in the remaining memory. As videos often have 20 to 30 frames per second, and neighbouring frames in time are very similar, the thousand extracted frames are spread equally throughout the video, to obtain more variation.

Video file

For the acceptance test the trailer for a video called "Bick Buck Bunny"[1] is used. This is a free video and can be used for various purposes. It is a full HD (1920×1080) color video with approximately a size of 30 MB. It is downloadable in the formats MOV and OGG. As OGG might not be supported for `VideoReader()` on some systems[2] the MOV format is chosen. The video contains 812 frames, and thereby all the frames are saved in the HDF5 file.

7.2 Results

The results of the acceptance test are listed in table 7.1. Histograms for the measured times can be found in appendix C. The average time per frame and frame rate is calculated by knowing that 400 frames have been used for 1000 runs.

	Filtering	**Lifting**	
	Decomposition	Decomposition	Reconstruction
Average time per frame	20.91 ms	16.42 ms	16.23 ms
Average frame rate	47.80 fps	60.91 fps	61.63 fps
Maximum time	24.35 ms	20.06 ms	20.78 ms
Minimum time	16.98 ms	12.96 ms	12.72 ms

Table 7.1. Result from the acceptance test.

7.3 Conclusions

The requirements from chapter 1 specify that the decomposition and reconstruction must each run with a frame rate of 25 fps (within 40 ms). From the frame rates in the table 7.1

[1] (c) copyright 2008, Blender Foundation / www.bigbuckbunny.org
[2] http://www.mathworks.se/help/techdoc/ref/videoreaderclass.html

it is clear that both the filtering and lifting fulfil this. These results however only give an idea of the performance if the implementation is used for real-time video coding. If the implementation instead only were feed with one frame each 40 ms, the result might be different, e.g. due to automatic clocking of the GPU.

The last requirement of chapter 1 states that the implementation must run on a consumer grade PC, and still fulfil the other requirements. The acceptance test was performed on the testing and development platform presented in section 5.1. This platform must be considered as a high end consumer PC. It can therefore be discussed if the last requirement is fulfilled.

8 Conclusions

The purpose of this project is to implement the Cohen-Daubechies-Feauveau (CDF) 9/7 wavelet transform on a Graphics Processing Unit (GPU), and test the potential to perform the wavelet transform in real-time at 25 FPS.

The basic theory behind the wavelet transform is analyzed, and the major wavelet families and their requirements and abilities are explained. After this, lifting equations are analyzed and derived for the CDF 9/7 wavelet transform.

Based on studies of the GPU architecture, different algorithms to perform the wavelet transform are investigated, including Row-Column (RC), Line-Based (LB), and Block-Based (BB). The computational and memory requirements are estimated, resulting in a discussion on parallelism and which algorithms fit the GPU architecture. It is chosen to base the implementation on the Row-Column method, since this is best suited for a GPU

In the next part, the software structure and benchmarks are specified and described. It is furthermore described how to verify the implemented algorithms and the obtained results. This is done using a MATLAB[1] implementation, based on the CDF 9/7 wavelet transform.

One filtering and two lifting algorithms are implemented. One of the lifting algorithms is only used throughout the development phase, and was dismissed after it was outperformed by an improved algorithm. The memory layout and utilization of the GPU have been investigated for the implementations to optimize the performance. The algorithms are implemented as Python modules in a package, making it easy to use. To perform full decompositions on HD resolution images, a function to pad images to square power of two resolutions is implemented on the GPU. Finally, a reconstruction algorithm, based on lifting, is implemented.

The platform used for testing is based on an Intel Core i7 970 CPU and a NVIDIA Geforce GTX680 GPU. The implementations are benchmarked against each other to find the fastest algorithm on the GPU. Here it shows that the lifting based algorithm requires less than half the computation time compared to the filter based implementation. This matches the estimation, that lifting needed less than half the FLOPs compared to filtering. The lifting implementation has the additional advantage, that it can perform a complete level of decomposition at once, whereas the filtering implementation needs to transpose the image and execute the same kernel twice. It should though be noted, that these execution times are measured with the data available on the GPU. The time needed to transfer the data to and from the GPU is generally higher than the computation time needed.

The benchmarks show, that we are able to perform 3 levels of decomposition on a 1080×1920 HD image within 17.21 ms using lifting. The performance of the GPU

[1] http://www.getreuer.info/home/waveletcdf97

is compared to a reference CPU implementation[2]. Since this implementation performs wavelet transforms on 2D arrays, the performance on color images is estimated by multiplying the obtained results by three. The lifting based GPU implementation is on average 10.03 times faster at a color image than the CPU estimation, when the transfer times to and from the memory also are taken into account. When only the raw computation time is considered, the lifting algorithm is on average 48.58 times faster and the filtering 21.77.

Finally the GPU implementations are tested in a video acceptance test, where 400 frames of a HD video have been decomposed and reconstructed. Here the lifting implementation was able to decompose the video stream at 60.91 fps and reconstruct at 61.63 FPS. Filtering is slightly slower, running at 47.80 FPS. All these results are better than the requirements of 25 FPS.

In the end, we can conclude, that there is a big potential in performing the wavelet transform on a GPU in real-time. It should though be noted, that the measurements are obtained using the fastest consumer grade hardware available on the marked today. This means that a midrange consumer PCs might not be able to achieve the same performance and fulfill the requirements.

[2]DWT97.cpp by Gregoire Pau and Tobias Lindstrøm Jensen

9 Perspective

The implementation has only been developed to either perform the forward or reverse CDF 9/7 at a time. In a setup where two musicians are interacting, both the decomposition and reconstruction must be performed on the same computer in real-time. This is theoretically still possible, as the implementation should be able to handle both operations at about 30 fps, which is still better than the specified requirements. If the memory layout of the implementation is also optimized, the performance should also improve.

In the developed implementation all three color layers are transferred to the GPU and processed together. If instead an asynchronous data transfer is used, and the transforming of the color layers are separated, the performance might be improved. Using three buffers, one for each color layer, and queuing the transform kernels, the latency of the data transfer could be hidden.

The used test platform uses the PCI-e 2.0 bus for the GPU interface. If instead a PC supporting PCI-e 3.0, theoretically having twice the bandwidth, is used, thereby lowering the transfer time between host and device.

The optimal dimensions for the work groups differs between GPUs, and a start-up script finding the best suited set-up, could automate the process, making it easy for a potential user to get the best performance possible on the hardware at hand.

Concerning the numerical precision, the implementation uses 32 bit floating point. If better numerical stability is desired, e.g. for scientific tasks, the implementation must be modified to feature this.

Bibliography

[1] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthr, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, 2003.

[2] P. S. Addison, *The Illustrated Wavelet Transform Handbook.* Institute of Physics Publishing, 2002.

[3] Khronos. (2012, Feb.) OpenCL - The open standard for parallel programming of heterogeneous systems. Online. Khronos Group. Accessed: 28-02-2012. [Online]. Available: http://www.khronos.org/opencl/

[4] Nvidia. (2012, Feb.) CUDA - Parallel programming made easy. Online. Nvidia. Accessed: 28-02-2012. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[5] P. J. Van Fleet, *Discrete Wavelet Transformations - An Elementary Approach with Applications.* Wiley-Interscience, 2008.

[6] F. Wasilewski. (2012) Wavelet browser by pywavelets. Accessed: 29-05-2012. [Online]. Available: http://wavelets.pybytes.com/wavelet/bior4.4/

[7] A. Jensen and A. La Cour-Horbo, *Ripples in Mathematics - The Discrete Wavelet Transform.* Springer, 2000.

[8] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier Analysis and Applications*, vol. 4, pp. 247–269, 1998, 10.1007/BF02476026. [Online]. Available: http://dx.doi.org/10.1007/BF02476026

[9] T. Larsen, *Behavioral Simulation and Computing.* AAU, 2012.

[10] ——, "Scientific computing - lecture 3," slides, Feb. 2012.

[11] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors.* Morgan Kaufman, 2000.

[12] Nvidia. (2012) Cuda c programming guide. Accessed 30-05-2012. [Online]. Available: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[13] W. van der Laan, A. Jalba, and J. Roerdink, "Accelerating Wavelet Lifting on Graphics Hardware Using CUDA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 132 –146, Jan. 2011.

[14] J. Oliver, E. Oliver, and M. P. Malumbres, "Fast integer-to-integer reversible lifting transform with reduced memory consumption," *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, pp. 289 –294, Dec. 2005.

[15] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Transactions on Image Processing*, vol. 9, no. 3, pp. 378 –389, Mar. 2000.

[16] O.-M. I. of Quantitative Finance. Gpu parallelizable methods. Accessed 20-05-2012. [Online]. Available: http://www.oxford-man.ox.ac.uk/gpuss/simd.html

[17] P. Getreuer. Wavelet cdf 9/7 implementation. Accessed 20-05-2012. [Online]. Available: http://www.getreuer.info/home/waveletcdf97

[18] Wikipedia. (2011, May) Peak signal-to-noise ratio. Accessed 20-05-2012. [Online]. Available: http://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

Part IV

Appendix

A Test images

Image	Resolution	Used for testing
bird.jpg	9000×9000	Square
lenna.jpg	936×1703 (936×936 cut)	Square
travel.jpg	7106×5715	Square
grasshopper.jpg	1920×1080	HD
insect.jpg	3840×2160	HD
waterfall.bmp	1920×1080	HD

Table A.1. Original images used for testing square and HD resolutions.

Square Resolutions	HD Resolutions
256×256	852×480
512×512	1280×720
1024×1024	1920×1080
2048×2048	3840×2160
4096×4096	4520×2540
8192×8192	7680×4320

Table A.2. Resolutions, that the images have been resized to using the Python module `Image`.

Below are the sources to the images.

bird.jpg:
http://fc04.deviantart.net/fs28/f/2008/077/2/3/Innocent_Striped_Beauty_by_satishverma.jpg
lenna.jpg:
http://kmlinux.fjfi.cvut.cz/ pavelro1/school/roz1/lenna_full.jpg
travel.jpg:
http://www.hipstertravelguide.com/wp-content/uploads/2011/05/21.jpg
grasshopper.jpg:
http://www.animalswallpapers.org/wp-content/uploads/2011/04/GrasshopperHDWallpaper.jpg
insect.jpg:
http://wallpaperswide.com/colors_define_beauty-wallpapers.html
waterfall.png:
http://t.wallpaperweb.org/wallpaper/nature/1920x1080/mmdght.bmp

B Benchmark results

CPU implementation

The CPU implementation [1] benchmark for the decomposition is shown on figure B.1(a) and reconstruction on B.1(b).

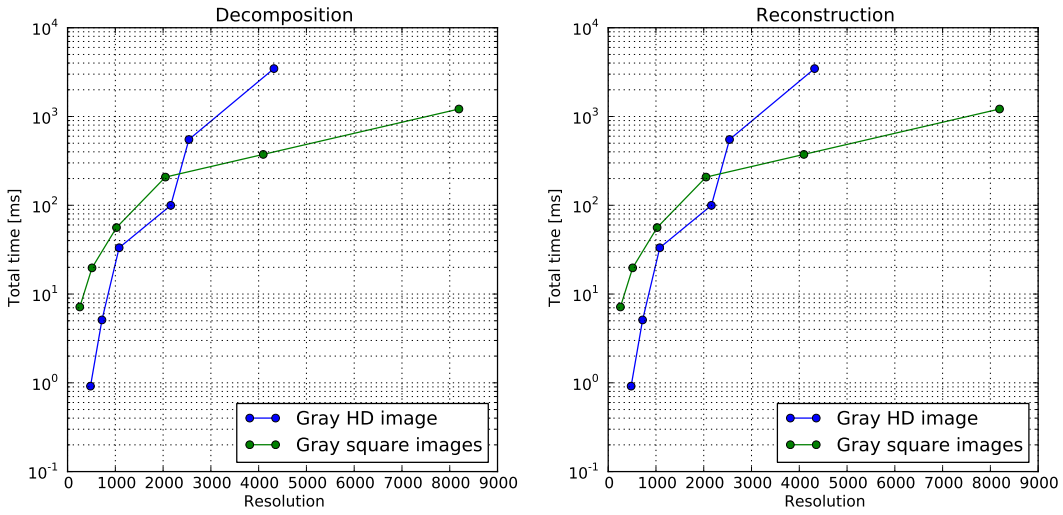


Figure B.1. Benchmark of the CPU implementation where figure (a) shows the decomposition and (b) shows reconstruction. This implementation is only tested on gray-scaled images.

Histograms

Histogram over to total execution time for HD resolution 1920×1080, using lifting B.2(a) and filtering B.2(b).

[1] DWT97.cpp by Gregoire Pau and Tobias Lindstrøm Jensen

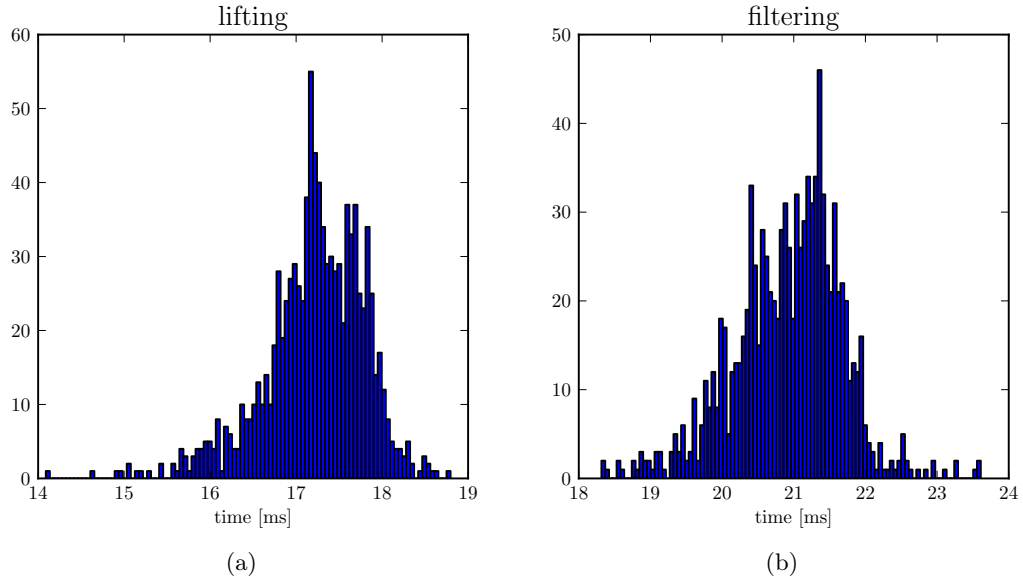
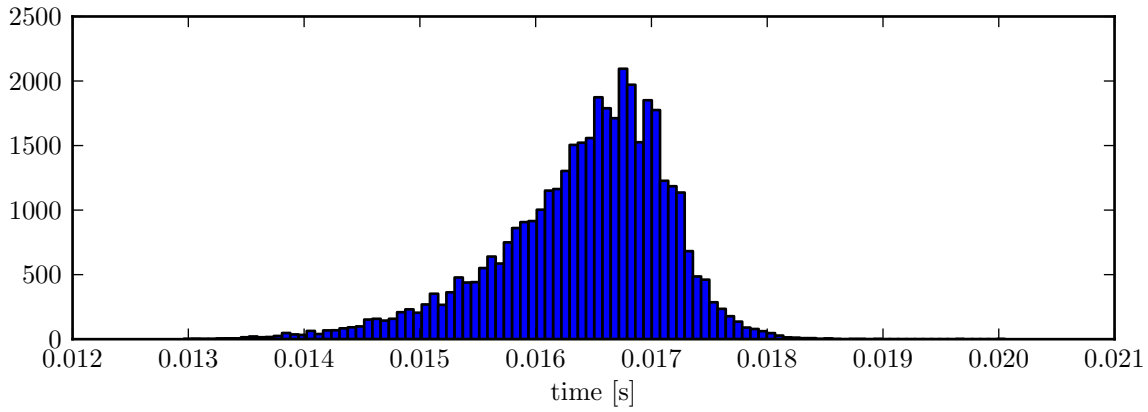
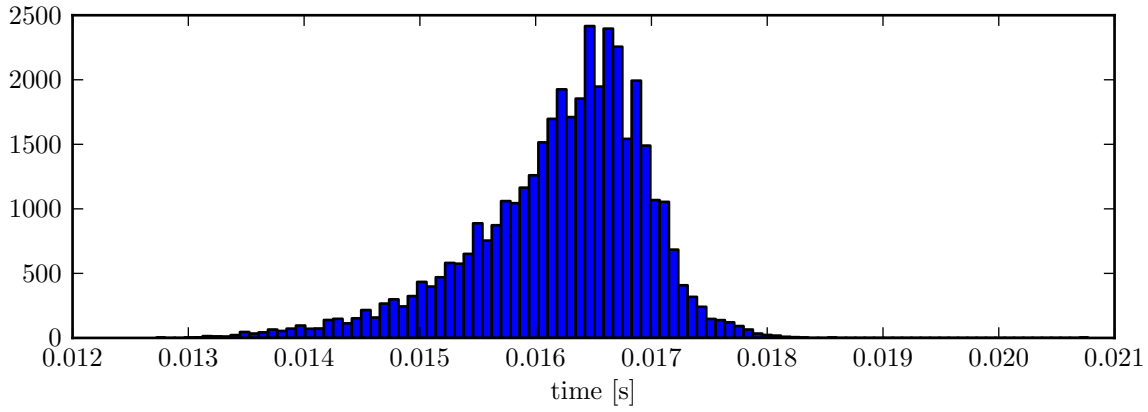


Figure B.2. Histogram showing total calculation time for the two algorithms (a) Lifting and (b) filtering.

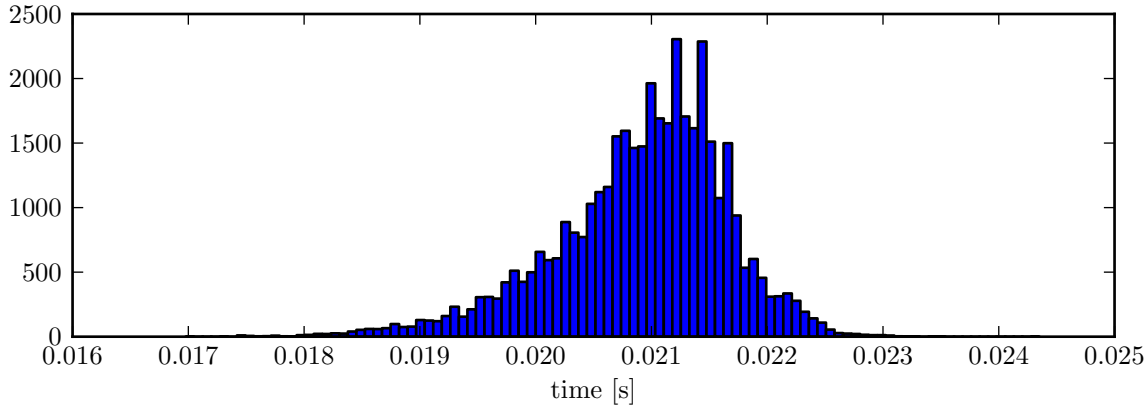
C Video result histograms



(a)



(b)



(c)

Figure C.1. Histograms of the times for the video benchmark for HD 1920 × 1080 frames. (a) Lifting decomposition. (b) Lifting reconstruction. (c) Filtering decomposition. Code: **videoPlot.py**.

D Kernel codes

Filtering kernel

```
#define OFFSET_READ %(offset_read)i
#define OFFSET_WRITE %(offset_write)i
#define WIDTH %(width)i
#define HEIGHT %(height)i
#define DEPTH %(depth)i
#define BLOCK %(block)i
#define ID_LP(y, x, z) (DEPTH * (x * HEIGHT + y) + z + DEPTH * x * OFFSET_WRITE)
#define ID_HP(base) (base + DEPTH * HEIGHT * WIDTH / 2 + DEPTH * WIDTH / 2 * OFFSET_WRITE)
#define READ_FROM_ID(y, x, z, n) (DEPTH * (y * WIDTH + 2 * x + n) + z + DEPTH * y * OFFSET_READ)

__kernel void cdf(
    __global const float *filterLP,
    __global const float *filterHP,
    __global float *img,
    __global float *out)
{
    int gidy = get_global_id(0);
    int gidx = get_global_id(1);
    int gidz = get_global_id(2);
    int lidx = get_local_id(1);
    int lidy = get_local_id(0);
    int bidx = get_group_id(1);

    __local float pixels[%(array_height)i][%(array_length)i];

    for(char x=0; x<2; x++) {
        int globalImgIndex = READ_FROM_ID(gidy, gidx, gidz, x);
        pixels[lidy][lidx * 2 + x + 4] = img[globalImgIndex];
    }

    if (lidx < 4) {
        int indexLeft = DEPTH * (gidy * WIDTH + ((bidx * 2 * BLOCK + lidx - 4) %(mod)c (WIDTH))) + gidz + DEPTH * gidy * OFFSET_READ;
        int indexRight = DEPTH * (gidy * WIDTH + ((bidx * 2 * BLOCK + 2 * BLOCK + lidx ) %(mod)c (WIDTH))) + gidz + DEPTH * gidy * OFFSET_READ;
        pixels[lidy][lidx] = img[indexLeft];
        pixels[lidy][lidx + 2 * BLOCK + 4] = img[indexRight];
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    float px[2] = {0, 0};

    for(char x=0; x<9; x++) {
        px[0] += pixels[lidy][lidx * 2 + x] * filterLP[x];
        px[1] += pixels[lidy][lidx * 2 + x] * filterHP[x];
    }
```

```
46
47      int lowPass = ID_LP(gidy, gidx, gidz);
48      int highPass = ID_HP(lowPass);
49
50      out[lowPass] = px[0];
51      out[highPass] = px[1];
52
53  }
```

Lifting kernel

```
1   /*
2   Wavelet lifting  approach with local memory line and block based combined
3   */
4
5
6
7   /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Decompose kernel
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
8   #define BLOCK_SIZEX %(BLOCK_SIZEX)i
9   #define BLOCK_SIZEY %(BLOCK_SIZEY)i
10  #define BLOCK_SIZEXG  (BLOCK_SIZEX - 4)
11  #define BLOCK_SIZEYG  (BLOCK_SIZEY - 4)
12  #define XDIM %(xdim)i
13  #define YDIM %(ydim)i
14  #define ZDIM %(zdim)i
15  #define OUTLL(z,y,x) out[z + ZDIM*((y)*XDIM + (x))]
16  #define OUTHL(z,y,x) out[z + ZDIM*((y)*XDIM + (windowdimX/2) + x)]
17  #define OUTLH(z,y,x) out[z + ZDIM*((y)*XDIM + (XDIM*windowdimY/2) + x)]
18  #define OUTHH(z,y,x) out[z + ZDIM*((y)*XDIM + (XDIM*windowdimY/2) +
        windowdimX/2 + x)]
19  #define MATRIX1(z,y,x) inm[z + ZDIM*((y)*XDIM+x)]
20
21  #define A (float) -1.586134342
22  #define B (float) -0.05298011858
23  #define C (float) 0.8829110753
24  #define D (float) 0.4435068521
25
26  #define K (float) 1.1496043989790903
27
28
29  __kernel void CDF97DEC( __global float *out, __global float *inm, int
        windowdimX, int windowdimY)
30  {
31
32    /* windowdimX *= 2; windowdimY *= 2; */
33
34    int gx = get_group_id(0); int gy = get_group_id(1);
35    int lx = get_local_id(0); int ly = get_local_id(1);
36
37    int x = lx + gx * BLOCK_SIZEXG; // global X is threadsize - 8 due to
          overlapping blocks.
38    int y = ly + gy * BLOCK_SIZEYG;
39    int z = get_global_id(2);
40
41    signed int yy, xx;
```

```
42
43    int lx2 = 2*lx; int ly2 = 2*ly;
44
45    __local float in[BLOCK_SIZEY*2+4][BLOCK_SIZEX*2+4];
46
47
48    xx = (x*2-4) %(MOD)c windowdimX; // MOD got replaced
49
50    while (xx < 0)
51    {
52      xx = (windowdimX - xx) %(MOD)c windowdimX;
53    }
54
55    yy = (y*2-4) %(MOD)c windowdimY;
56
57    while (yy < 0)
58    {
59      yy = (windowdimY - yy) %(MOD)c windowdimY;
60    }
61
62    // get data from global mem. Two pixels pr. thread
63    in[ly2][lx2] = MATRIX1(z,yy,xx);
64    in[ly2+1][lx2] = MATRIX1(z,yy+1,xx);
65
66    in[ly2][lx2+1] = MATRIX1(z,yy,xx+1);
67    in[ly2+1][lx2+1] = MATRIX1(z,yy+1,xx+1);
68
69    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
70
71    //if (lx < BLOCK_SIZEX - 1)
72    in[ly2][lx2+1] += A * (in[ly2][lx2] + in[ly2][lx2+2]);  // d1
73    in[ly2+1][lx2+1] += A * (in[ly2+1][lx2] + in[ly2+1][lx2+2]);  // d1
74
75    barrier(CLK_LOCAL_MEM_FENCE);
76
77    // if (lx < BLOCK_SIZEX - 2)
78    in[ly2][lx2+2] += B * (in[ly2][lx2+1] + in[ly2][lx2+3]); // s1
79    in[ly2+1][lx2+2] += B * (in[ly2+1][lx2+1] + in[ly2+1][lx2+3]); // s1
80
81    barrier(CLK_LOCAL_MEM_FENCE);
82
83    //if (lx < BLOCK_SIZEX - 3)
84    in[ly2][lx2+3] += C * (in[ly2][lx2+2] + in[ly2][lx2+4]); // d2
85    in[ly2+1][lx2+3] += C * (in[ly2+1][lx2+2] + in[ly2+1][lx2+4]); // d2
86
87    barrier(CLK_LOCAL_MEM_FENCE);
88
89    // if (lx < BLOCK_SIZEX - 4)
90    in[ly2][lx2+4] = (in[ly2][lx2+4] + D * (in[ly2][lx2+3] +
          in[ly2][lx2+5])) * K; // s2 and s
91    in[ly2+1][lx2+4] = (in[ly2+1][lx2+4] + D * (in[ly2+1][lx2+3] +
          in[ly2+1][lx2+5])) * K; // s2 and s
92    // Sort data in tmp array
93
94    barrier(CLK_LOCAL_MEM_FENCE);
95
96    in[ly2][lx2+5] /= K; // d
97    in[ly2+1][lx2+5] /= K; // d
98
```

```
99    barrier(CLK_LOCAL_MEM_FENCE);
100
101
102   // Second pass
103
104
105   //if (lx < BLOCK_SIZEX - 1)
106   in[ly2+1][lx2] += A * (in[ly2][lx2] + in[ly2+2][lx2]);  // d1
107   in[ly2+1][lx2+1] += A * (in[ly2][lx2+1] + in[ly2+2][lx2+1]);  // d1
108
109   barrier(CLK_LOCAL_MEM_FENCE);
110
111   // if (lx < BLOCK_SIZEX - 2)
112   in[ly2+2][lx2] += B * (in[ly2+1][lx2] + in[ly2+3][lx2]); // s1
113   in[ly2+2][lx2+1] += B * (in[ly2+1][lx2+1] + in[ly2+3][lx2+1]); // s1
114
115   barrier(CLK_LOCAL_MEM_FENCE);
116
117   //if (lx < BLOCK_SIZEX - 3)
118   in[ly2+3][lx2] += C * (in[ly2+2][lx2] + in[ly2+4][lx2]); // d2
119   in[ly2+3][lx2+1] += C * (in[ly2+2][lx2+1] + in[ly2+4][lx2+1]); // d2
120
121   barrier(CLK_LOCAL_MEM_FENCE);
122
123   // if (lx < BLOCK_SIZEX - 4)
124   in[ly2+4][lx2] = (in[ly2+4][lx2] + D * (in[ly2+3][lx2] +
          in[ly2+5][lx2])) * K; // s2 and s
125   in[ly2+4][lx2+1] = (in[ly2+4][lx2+1] + D * (in[ly2+3][lx2+1] +
          in[ly2+5][lx2+1])) * K; // s2 and s
126
127   barrier(CLK_LOCAL_MEM_FENCE);
128
129   // Sort data in in array
130   in[ly2+5][lx2] /= K; // d
131   in[ly2+5][lx2+1] /= K; //d
132
133   barrier(CLK_LOCAL_MEM_FENCE| CLK_GLOBAL_MEM_FENCE);
134
135   // Store data in global Mem.
136   if (x < windowdimX/2 )
137   {
138     if (y < windowdimY/2 )
139     {
140       if (ly < BLOCK_SIZEY - 4 )
141       {
142         if (lx < BLOCK_SIZEX - 4 )
143         {
144           OUTLL(z,y,x) = in[ly2+4][lx2+4];
145           OUTLH(z,y,x) = in[ly2+5][lx2+4];
146           OUTHL(z,y,x) = in[ly2+4][lx2+5];
147           OUTHH(z,y,x) = in[ly2+5][lx2+5];
148         }
149       }
150     }
151   }
152 }
153
154
155
```

```
156 inline float truncatePixel(float pixel)
157 {
158
159   if (pixel < 0)
160     pixel = 0;
161
162   if (pixel > 1)
163     pixel = 1;
164
165   return pixel;
166
167 }
168
169 /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Reconstruct kernel
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
170
171 #define MATRIXRECEVEN(z,y,x) out[z + ZDIM*((y)*XDIM + 2*(x))]
172 #define MATRIXRECODD(z,y,x) out[z + ZDIM*((y)*XDIM + 2*(x) + 1)]
173 #define MATRIXREC(z,y,x) out[z + ZDIM*((y)*XDIM + (x))]
174 #define BLOCK_SIZEXR %(BLOCK_SIZEXR)d
175 #define BLOCK_SIZEYR %(BLOCK_SIZEYR)d
176 #define BLOCK_SIZEXGR (BLOCK_SIZEXR - 4)
177 #define BLOCK_SIZEYGR (BLOCK_SIZEYR - 4)
178 #define INLL(z,y,x) inm[z + ZDIM*((y)*XDIM + x)]
179 #define INHL(z,y,x) inm[z + ZDIM*((y)*XDIM + windowdimX + x)]
180 #define INLH(z,y,x) inm[z + ZDIM*((y)*XDIM + (XDIM*windowdimY) + x)]
181 #define INHH(z,y,x) inm[z + ZDIM*((y)*XDIM + (XDIM*windowdimY) +
       windowdimX + x)]
182
183 #define LOCALSHAPEY BLOCK_SIZEY*2+7
184 #define LOCALSHAPEX BLOCK_SIZEX*2+7
185
186 __kernel void CDF97REC( __global float *out, __global float *inm, int
       windowdimX, int windowdimY, int truncate)
187 {
188
189
190   int lx = get_local_id(0); volatile int ly = get_local_id(1);
191
192   int x = get_group_id(0)*BLOCK_SIZEXGR + lx;
193   int y = get_group_id(1)*BLOCK_SIZEYGR + ly;
194   int z = get_global_id(2);
195
196   __local float in[LOCALSHAPEY][LOCALSHAPEX];
197
198
199   /* int lxb = lx + BLOCK_SIZEXR;    */
200   /* int lyb = ly + BLOCK_SIZEYR;    */
201
202   int lx2 = lx*2; int ly2 = ly*2;
203
204   signed int xx,yy;
205
206   xx = (x-2) %(MOD)c windowdimX; // MOD got replaced
207
208   while (xx < 0)
209   {
210     xx = (windowdimX - xx) %(MOD)c windowdimX;
211   }
```

```
212
213     yy = (y-2) %(MOD)c windowdimY;
214
215     while (yy < 0)
216     {
217       yy = (windowdimY - yy) %(MOD)c windowdimY;
218     }
219
220     // get data from global mem. Two pixels pr. thread
221     in[ly2][lx2] = INLL(z,yy,xx); /* Scaling s2 */
222     in[ly2+1][lx2] = INLH(z,yy,xx); /* Scaling s2 */
223     in[ly2][lx2+1] = INHL(z,yy,xx); /* Scaling d2 */
224     in[ly2+1][lx2+1] = INHH(z,yy,xx); /* Scaling d2 */
225
226     in[ly2][lx2] /= K;
227     in[ly2+1][lx2] /= K;
228     in[ly2][lx2+1] *= K;
229     in[ly2+1][lx2+1] *= K;
230
231     /* Compute first pass reconstruction horizontally */
232     barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
233
234     in[ly2][lx2+2] -= D * (in[ly2][lx2+1] + in[ly2][lx2+3]);  // s1
235     in[ly2+1][lx2+2] -= D * (in[ly2+1][lx2+1] + in[ly2+1][lx2+3]);  // s1
236
237     barrier(CLK_LOCAL_MEM_FENCE );
238
239     in[ly2][lx2+3] -= C * (in[ly2][lx2+2] + in[ly2][lx2+4]); // d1
240     in[ly2+1][lx2+3] -= C * (in[ly2+1][lx2+2] + in[ly2+1][lx2+4]); // d1
241
242     barrier(CLK_LOCAL_MEM_FENCE );
243
244     in[ly2][lx2+4] -= B * (in[ly2][lx2+3] + in[ly2][lx2+5]); // s0
245     in[ly2+1][lx2+4] -= B * (in[ly2+1][lx2+3] + in[ly2+1][lx2+5]); // s0
246
247     barrier(CLK_LOCAL_MEM_FENCE );
248
249     in[ly2][lx2+5] -=  A * (in[ly2][lx2+4] + in[ly2][lx2+6]); // d0 and xo L
250     in[ly2+1][lx2+5] -= A * (in[ly2+1][lx2+4] + in[ly2+1][lx2+6]); // d0
        and x0 H
251
252     barrier(CLK_LOCAL_MEM_FENCE );
253
254     /* Compute second pass reconstruction vertically */
255
256     in[ly2][lx2] /= K;
257     in[ly2+1][lx2] *= K;
258
259     in[ly2][lx2+1] /= K; // xe L
260     in[ly2+1][lx2+1] *= K; // xe H
261
262
263     barrier(CLK_LOCAL_MEM_FENCE );
264
265     in[ly2+2][lx2] -= D * (in[ly2+1][lx2] + in[ly2+3][lx2]);  // s1
266     in[ly2+2][lx2+1] -= D * (in[ly2+1][lx2+1] + in[ly2+3][lx2+1]);  // s1
267
268     barrier(CLK_LOCAL_MEM_FENCE );
269
```

```
270   in[ly2+3][lx2] -= C * (in[ly2+2][lx2] + in[ly2+4][lx2]); // d1
271   in[ly2+3][lx2+1] -= C * (in[ly2+2][lx2+1] + in[ly2+4][lx2+1]); // d1
272
273   barrier(CLK_LOCAL_MEM_FENCE );
274
275   in[ly2+4][lx2] -= B * (in[ly2+3][lx2] + in[ly2+5][lx2]); //s0
276   in[ly2+4][lx2+1] -= B * (in[ly2+3][lx2+1] + in[ly2+5][lx2+1]); // s0
277
278   barrier(CLK_LOCAL_MEM_FENCE );
279
280   in[ly2+5][lx2] -= A * (in[ly2+4][lx2] + in[ly2+6][lx2]); // d0
281   in[ly2+5][lx2+1] -= A * (in[ly2+4][lx2+1] + in[ly2+6][lx2+1]); // d0
282
283   barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
284
285   if (truncate)
286   {
287     in[ly2+4][lx2] = truncatePixel(in[ly2+4][lx2]);
288     in[ly2+4][lx2+1] = truncatePixel(in[ly2+4][lx2+1]);
289     in[ly2+5][lx2] = truncatePixel(in[ly2+5][lx2]);
290     in[ly2+5][lx2+1] = truncatePixel(in[ly2+5][lx2+1]);
291
292   }
293
294
295   if (x < windowdimX )
296   {
297     if (y < windowdimY )
298     {
299       if (ly < BLOCK_SIZEYR - 4 )
300       {
301         if (lx < BLOCK_SIZEXR - 4 )
302         {
303           MATRIXREC(z,2*y,2*x) = in[ly2+4][lx2+4];
304           MATRIXREC(z,2*y+1,2*x) = in[ly2+5][lx2+4];
305           MATRIXREC(z,2*y,2*x+1) = in[ly2+4][lx2+5];
306           MATRIXREC(z,2*y+1,2*x+1) = in[ly2+5][lx2+5];
307
308
309
310         }
311       }
312     }
313   }
314 }
315
316
317 /* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Copy kernel
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
318
319
320 #define IN(z,y,x) in[z + ZDIM*(((y)*XDIM)+x)]
321 #define OUT(z,y,x) out[z + ZDIM*(((y)*XDIM)+x)]
322
323 __kernel void Copy(__global float *out, __global float *in)
324 {
325
326   int x = get_global_id(0); int y = get_global_id(1); int z =
        get_global_id(2);
```

```
327
328   OUT(z,y,x) = IN(z,y,x);
329
330   barrier(CLK_LOCAL_MEM_FENCE);
331 }
```

Discarded lifting kernel

```
1  /*
2  Wavelet liftion approach with local memory. Linear one dimensional
3  */
4  #define BLOCK_SIZEX %(BLOCK_SIZEX)i
5  #define BLOCK_SIZEY %(BLOCK_SIZEY)i
6  #define BLOCK_SIZEXG  (BLOCK_SIZEX - 4)
7  #define XDIM %(xdim)i
8  #define OUTL(y,x) out[(y*XDIM)+x]
9  #define OUTH(y,x) out[(y*XDIM)+(XDIM*windowdim/2)+x]
10 #define MATRIX1(y,x) matrix1[((y*XDIM)+x)]
11
12 #define A (float) -1.586134342
13 #define B (float) -0.05298011858
14 #define C (float) 0.8829110753
15 #define D (float) 0.4435068521
16
17 #define K (float) 1.1496043989790903
18
19
20 __kernel void CDF97DEC( __global float *out, __global float *matrix1, int
       windowdim)
21 {
22
23   int gx = get_group_id(0); int gy = get_group_id(1);
24   int lx = get_local_id(0); int ly = get_local_id(1);
25
26   int x = lx + gx * BLOCK_SIZEXG; // global X is threadsize - 8 due to
         overlapping blocks.
27   int y = ly + gy * BLOCK_SIZEY;
28   int ii;
29
30   int lx2 = 2*lx;
31
32   __local float in[BLOCK_SIZEY][BLOCK_SIZEX*2+4];
33
34
35   ii = (x*2-4) %(MOD)c windowdim; // MOD got replaced
36
37   while (ii < 0)
38   {
39     ii = (windowdim - ii) %(MOD)c windowdim;
40   }
41
42   // get data from global mem. Two pixels pr. thread
43   in[ly][lx2] = MATRIX1(y,ii);
44   in[ly][lx2+1] = MATRIX1(y,ii+1);
45
46   barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
```

```
47
48    //if (lx < BLOCK_SIZEX - 1)
49      in[ly][lx2+1] += A * (in[ly][lx2] + in[ly][lx2+2]);  // d1
50
51    barrier(CLK_LOCAL_MEM_FENCE);
52
53    // if (lx < BLOCK_SIZEX - 2)
54      in[ly][lx2+2] += B * (in[ly][lx2+1] + in[ly][lx2+3]); // s1
55
56    barrier(CLK_LOCAL_MEM_FENCE);
57
58    //if (lx < BLOCK_SIZEX - 3)
59      in[ly][lx2+3] += C * (in[ly][lx2+2] + in[ly][lx2+4]); // d2
60
61    barrier(CLK_LOCAL_MEM_FENCE);
62
63    // if (lx < BLOCK_SIZEX - 4)
64      in[ly][lx2+4] += D * (in[ly][lx2+3] + in[ly][lx2+5]); // s2
65
66    // Store data in global Mem.
67      if (x < windowdim/2 )
68    {
69
70      if (lx < BLOCK_SIZEX-4 )
71      {
72        OUTL(x,y) = in[ly][lx2+4] * K; // scale to s
73        OUTH(x,y) = in[ly][lx2+5] / K; // scale to d
74      }
75    }
76  }
77
78
79  #define MATRIXRECEVEN(y,x) out[(y*XDIM*2) + x]
80  #define MATRIXRECODD(y,x) out[(y*XDIM*2+XDIM) + x]
81  #define INL(y,x) matrix1[(y*XDIM)+x]
82  #define INH(y,x) matrix1[(y*XDIM)+windowdim + x]
83  #define BLOCK_SIZEXGR (BLOCK_SIZEX - 4)
84
85  __kernel void CDF97REC( __global float *out, __global float *matrix1, int
        windowdim)
86  {
87
88    int gx = get_group_id(0); int gy = get_group_id(1);
89    int lx = get_local_id(0); int ly = get_local_id(1);
90
91    int x = gx*BLOCK_SIZEXGR + lx; int y = gy*BLOCK_SIZEY + ly;
92
93    //__local float inL[BLOCK_SIZEY][BLOCK_SIZEX];
94    //__local float inH[BLOCK_SIZEY][BLOCK_SIZEX];
95
96
97    __local float in[BLOCK_SIZEY][BLOCK_SIZEX*2+7];
98
99    int ii = 0;
100   int lx2 = lx*2;
101
102   ii = (x-2) %(MOD)c windowdim; // MOD got replaced
103
104   while (ii < 0)
```

```
105   {
106     ii = (windowdim - ii) %(MOD)c windowdim;
107   }
108
109   in[ly][lx*2] = INL(y,ii); // s
110   in[ly][lx*2+1] = INH(y,ii); // d
111
112   in[ly][lx2] /= K; // Scaling  s2
113   in[ly][lx2+1] *= K;  // Scaling d2
114
115   barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
116
117   if (lx < BLOCK_SIZEX - 1)
118     in[ly][lx2+2] -= D * (in[ly][lx2+1] + in[ly][lx2+3]);  // s1
119
120   barrier(CLK_LOCAL_MEM_FENCE);
121
122   if (lx < BLOCK_SIZEX - 2)
123     in[ly][lx2+3] -= C * (in[ly][lx2+2] + in[ly][lx2+4]); // d1
124
125   barrier(CLK_LOCAL_MEM_FENCE);
126
127   if (lx < BLOCK_SIZEX - 3)
128     in[ly][lx2+4] -= B * (in[ly][lx2+3] + in[ly][lx2+5]); // s0
129
130   barrier(CLK_LOCAL_MEM_FENCE);
131
132   if (lx < BLOCK_SIZEX - 4)
133     in[ly][lx2+5] -= A * (in[ly][lx2+4] + in[ly][lx2+6]); // d0
134
135
136   // Write reconstruction
137   if (x < windowdim)
138   {
139
140     if (lx < BLOCK_SIZEXGR )
141     {
142       MATRIXRECEVEN(x,y) = in[ly][2*lx+4];
143       MATRIXRECODD(x,y) =  in[ly][2*lx+5];
144     }
145   }
146 }
```

Padding kernel

```
1  #define XDIM %(xdim)i
2  #define YDIM %(ydim)i
3  #define ZDIM %(zdim)i
4
5  __kernel void padding_two_d(__global float *image, __global float *padded,
6                      unsigned offset_x, unsigned offset_y,
7                      unsigned padded_xdim )
8  {
9      int Glox = get_global_id(0);
10     int Gloy = get_global_id(1);
11
```

```
12      if (Glox < XDIM && Gloy < YDIM ) {
13          padded[Glox + offset_x + (Gloy + offset_y )* padded_xdim] =
                image[Glox + Gloy * XDIM ];
14      }
15      else
16      {
17          padded[Glox + offset_x + (Gloy + offset_y ) * padded_xdim] = 0;
18      }
19  }
20
21  __kernel void padding_color(__global float *image, __global float *padded,
22                      unsigned offset_x, unsigned offset_y,
23                      unsigned padded_xdim )
24  {
25      int Glox = get_global_id(0);
26      int Gloy = get_global_id(1);
27      int Gloz = get_global_id(2);
28
29      if( Glox < XDIM && Gloy < YDIM )
30      {
31          padded[ZDIM * Glox + ZDIM * offset_x + ZDIM * padded_xdim * (Gloy
                + offset_y) + Gloz] =
32              image[ZDIM * Glox + ZDIM * XDIM * Gloy + Gloz] ;
33      }
34      else
35      {
36          padded[ZDIM * Glox + ZDIM * offset_x + ZDIM * padded_xdim * (Gloy
                + offset_y) + Gloz] = 0;
37      }
38  }
39
40  __kernel void copy_d(__global float *image, __global float *padded)
41  {
42      int Glox = get_global_id(0);
43      int Gloy = get_global_id(1);
44      padded[Glox  + XDIM * Gloy ] = image[Glox + XDIM * Gloy ];
45  }
46
47  __kernel void copy_color(__global float *image, __global float *padded,
        uint xdimP)
48  {
49      int Glox = get_global_id(0);
50      int Gloy = get_global_id(1);
51      int Gloz = get_global_id(2);
52      padded[ZDIM * Glox  + ZDIM * XDIM * Gloy + Gloz] = image[ZDIM * Glox
            + ZDIM * xdimP * Gloy + Gloz] ;
53  }
54
55
56
57  // Version 1. This work
58  // __kernel void copy_3d(__global float *image, __global float *padded)
59  // {
60  //  int Gloz = get_global_id(0);
61  //  int Gloy = get_global_id(1);
62  //  int Glox = get_global_id(2);
63  //
64  // //       padded[Glox  + XDIM * Gloy + XDIM * YDIM * Gloz ] =
        image[Glox + XDIM * Gloy + XDIM * YDIM * Gloz];
```

```
65 // //    if (Gloz == 2)
66 //  {
67 // //        padded[Glox  + XDIM * Gloy] = image[Glox + xdim * Gloy ];
68 //
69 //       padded[3*Gloz  + 3 * XDIM * Gloy + Glox] = image[3*Gloz + 3 *
       xdim * Gloy + Glox] ;
70 //  }
71 // }
```