Fakultät für Informatik und Mathematik
Universität Passau

GPU-Implementierung von Filterbänken

Masterarbeit

Richard Maltan
Matrikel-Nummer 58941

Betreuer Prof. Dr. Tomas Sauer
Zweitprüfer Prof. Dr. Brigitte Forster-Heinlein

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CU	compute unit
DWT	Discrete Wavelet Transform
GPU	Graphics Processing Unit
PE	processing element
PGM	Portable Graymap
PR	Perfekte Rekonstruktion
PSNR	Peak signal-to-noise ratio
QMF	Quadrature Mirror Filter
SIMT	Single Instruction, Multiple Threads

1 Einleitung

Der Vorteil von GPUs (Graphics Processing Unit) bei bestimmten Berechnungen liegt darin, dass sie Aufgaben, die viele unabhängige Einzelberechnungen erfordern, wesentlich schneller bearbeiten können als CPUs. Ein Beispiel für eine solche Berechnung ist die Faltung zweier Vektoren. Dabei lässt sich jede Position des Ergebnisvektors unabhängig von den anderen Positionen berechnen, was parallele Verarbeitung ermöglicht.

Dies kann man sich bei der Implementierung von Filterbänken zunutze machen. Da die Faltung ein Beispiel für eine hochparallelisierbare Anwendung ist [11], kann die Verwendung von entsprechenden Prozessoren, wie zum Beispiel GPUs, Vorteile, wie zum Beispiel eine kürzere Ausführungszeit, bringen. Im Rahmen dieser Arbeit soll nun eine solche GPU-Implementierung mittels OpenCL erstellt und getestet werden.

Zunächst wird in Kapitel 2 auf die mathematischen Grundlagen eingegangen, wobei die Struktur von Filterbänken vorgestellt wird sowie einige mathematische Eigenschaften, zum Beispiel die perfekte Rekonstruktion von Daten, erklärt werden. Zusätzlich wird kurz mit der diskreten Wavelet-Transformation (DWT) und ihrer Verwendung im JPEG2000-Standard eine Anwendung vorgestellt. Zuletzt wird eine kurze Einführung in OpenCL und GPU-Computing gegeben.

In Kapitel 3 wird die Implementierung einer Filterbank in OpenCL und C++ vorgestellt. Dabei werden zunächst die Anforderungen, die an die Implementierung gestellt wurden, besprochen und dann die mathematische Theorie hinter der Implementierung erklärt. Des weiteren werden die Funktionen für Faltung und Sampling von eindimensionalen und zweidimensionalen Eingabedaten genauer erläutert und zuletzt auf den Ablauf einer Filterbank bei Ausführung der Implementierung eingegangen.

In Kapitel 4 wird die Implementierung dann bzgl. Laufzeit und Qualität untersucht. Da die Verwendung von OpenCL nicht auf eine GPU beschränkt ist, wird diese mit der CPU

verglichen, wobei hier sowohl der mit OpenCL parallelisierte Code verwendet wird als auch eine Test-Implementation ohne Parallelisierung.

2 Grundlagen

Vor dem eigentlichen Thema dieser Arbeit (die Implementierung einer Filterbank mit OpenCL) werden zunächst einige Grundlagen erläutert.

Dieses Kapite beinhaltet die Grundlagen sowie den Aufbau von Filterbänken und der diskreten Wavelet-Transformation (DWT) sowie ein Überblick über OpenCL und GPU-Computing gegeben.

2.1 Mathematische Grundlagen

Als Filterbänke werden eine Anzahl von parallelen Filtern bezeichnet die, zusammen mit einer Verschiebung des Signals, zur Zerlegung eines Signals in verschiedene Teilbänder oder Subbänder verwendet werden. Dazu wird das Signal jeweils mit den gegebenen Filtern gefaltet und danach eine Abtastung (Downsampling) darauf angewendet. Mithilfe einer weiteren Filterbank sowie Upsampling lassen sich diese Subbänder wieder zusammensetzen.

2.1.1 Z-Transformation

Um Filterbänke zu beschreiben wird häufig das Konzept der z-Transformation aus der Systemtheorie herangezogen [25]. Die z-Transformierte eines Signals $x(n)$ ist definiert als:

$$X(z) := \sum_{n \in \mathbb{Z}} x(n) z^{-n}, \qquad z \in \mathbb{C} / \{0\} \tag{2.1}$$

Eine Translation (Verschiebung) um l Stellen im Zeitbereich entspricht einer Multiplikation mit z^{-l} im z-Bereich. Zusätzlich lassen sich Über- und Unterabtastung (siehe Abschnitt 2.1.3) mit einer Potenz von z darstellen:

$$X\left(z^M\right) = \sum_{n \in \mathbb{Z}} x\left(nM\right) z^{-nM} \tag{2.2}$$

2.1.2 Diskrete Faltung

Die kontinuierliche Faltung zweier Funktionen f und g ist folgendermaßen definiert [20]:

$$\left(f * g\right)\left(n\right) = \int_{\mathbb{R}} f\left(n-k\right) g\left(k\right) dt \tag{2.3}$$

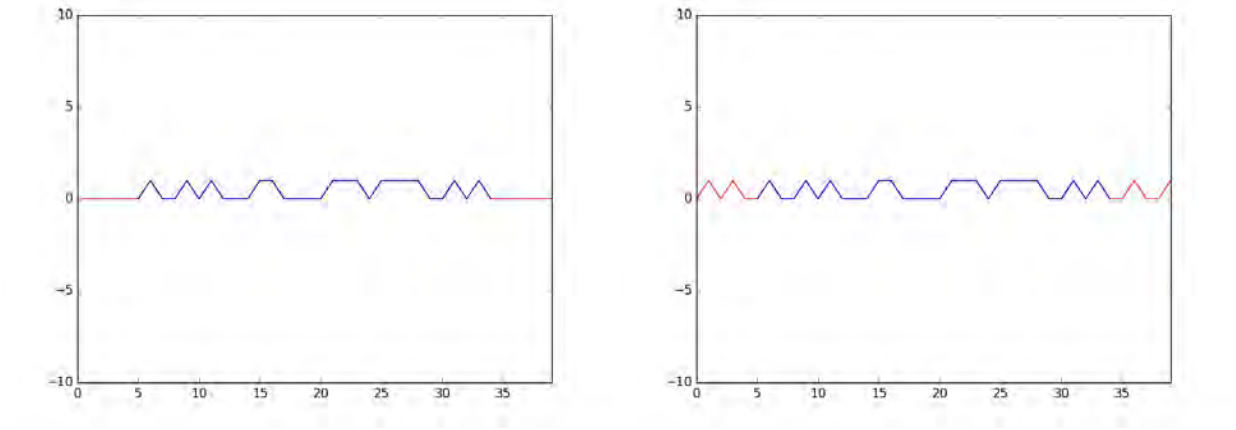
Da man bei Implementierungen in Software üblicherweise nicht auf Funktionen oder kontinuierlichen Signalen arbeitet, sondern die Werte als Vektoren c und d gegeben sind, verwendet man hier häufig die diskrete Faltung:

$$\left(c * d\right)\left(n\right) = \sum_{k \in \mathbb{Z}} c\left(n-k\right) d\left(k\right) \tag{2.4}$$

Weil im diskreten Fall oft endliche Signale behandelt werden, tritt früher oder später das Problem auf, dass die Faltung auf Werte zugreifen muss, die außerhalb des Signalvektors liegen. Da man allerdings keine Informationen über das Verhalten des Signals ausserhalb des gegebenen Bereichs hat, muss man an dieser Stelle Vermutungen anstellen. Es gibt nun zwei Möglichkeiten damit umzugehen [20].

1. Man setzt alle Werte außerhalb des Signals auf 0 (Abbildung 2.1(a)), oder
2. man setzt das Signal periodisch fort (zyklische Faltung, Abbildung 2.1(b))

Beide Methoden sind dabei nicht optimal, da an den Rändern des Ergebnisses der Faltung Artefakte auftreten können.



(a) Null (b) Zyklisch

Abbildung 2.1: Möglichkeiten ein Signal außerhalb seiner Grenzen zu behandeln. Der blau markierte Teil stellt dabei das gegebene (endliche) Signal dar.

2.1.3 Sampling

Als Sampling bezeichnet man das Abtasten eines Signals um einen Faktor $n \in \mathbb{N}$, was bedeutet, man verwendet jeden n-ten Wert des Signals um ein Teilsignal zu generieren (Downsampling, $\downarrow x$) oder um ein Signal mit Nullen "aufzufüllen" (Upsampling, $\uparrow x$) [20]:

- Downsampling:

$$\downarrow_n x = x\left(n\cdot\right), \qquad n \geq 2 \tag{2.5}$$

- Upsampling:

$$\uparrow_n x\left(j\right) = \begin{cases} x\left(j/n\right), & j \in n\mathbb{Z} \\ 0, & j \in \mathbb{Z}/n\mathbb{Z} \end{cases} \tag{2.6}$$

2.2 Aufbau einer Filterbank

Eine Filterbank setzt sich aus zwei Komponenten zusammen, zum einen eine Reihe von parallelen Filtern, die auf das Eingangssignal angewendet werden, zum anderen aus einer

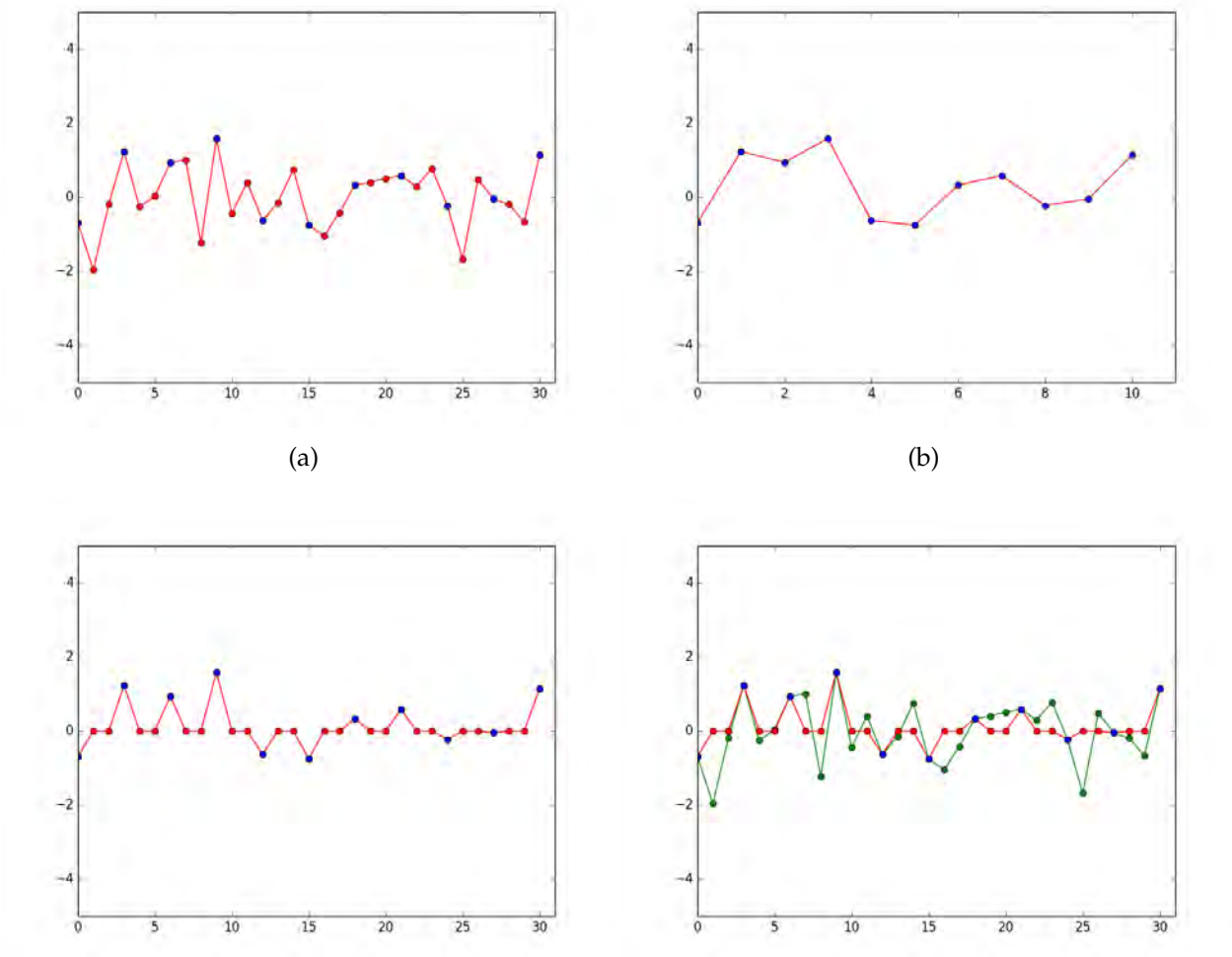


Abbildung 2.2: Ein zufälliges Signal vor (a) und nach dem Downsampling (b)und nach dem Upsampling (c). (d) zeigt den Vergleich zwischen dem ursprünglichen Signal (grün) und dem down- und wieder upgesampleten Signal (Rot). Die blauen Punkte in den Grafiken stellen dabei die Abtastpunkte dar. Zu beachten ist die Verkleinerung des downgesampleten Signals in (b).

Abtastung. Im weiteren Verlauf wird kurz auf den Aufbau einer Filterbank sowie der Eigenschaft der perfekten Rekonstruktion eingegangen, die es erlaubt die Subbänder eines Signals wieder zum ursprünglichen Signal zusammenzusetzen.

2.2.1 Subband Coding

Das Filtern und Zerlegen eines Signals wird als Subband Coding bezeichnet. Dabei wird mittels Filterung und Downsampling ein Eingangssignal in N Teilbänder zerlegt. Ein Teilsignal x_j ist dann folgendermaßen definiert [20]:

$$x_j = \downarrow_n F_j x, \qquad j \in \mathbb{Z}_n \tag{2.7}$$

wobei $F_j x$ die Faltung eines Signals mit dem j-ten Filter der Filterbank bezeichnet. Die einzelnen Teilsignale können dann mittels

$$x = \sum_{j \in \mathbb{Z}_n} F_j \uparrow_n x_j \qquad j \in \mathbb{Z}_n \tag{2.8}$$

wieder zusammengesetzt werden. Die Zerlegung liefert dabei

$$\begin{array}{ccccccccc}
\cdots & x(0) & x(n) & \cdots & x(2n) & x(3n) & \cdots & \leftarrow x_0 \\
\cdots & x(1) & x(n+1) & \cdots & x(2n+1) & x(3n+1) & \cdots & \leftarrow x_1 \\
 & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\
\cdots & x(n-1) & x(2n-1) & \cdots & x(3n-1) & x(4n-1) & \cdots & \leftarrow x_{n-1}
\end{array}$$

was sich wieder zu

$$\begin{array}{cccccccccc}
x_0 \rightarrow & 0 & x(0) & 0 & \cdots & 0 & x(n) & 0 & \cdots & \\
x_1 \rightarrow & 0 & 0 & x(1) & \cdots & 0 & 0 & x(n+1) & \cdots & \\
 & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \\
x_{n-1} \rightarrow & x(-1) & 0 & 0 & \cdots & x(n-1) & 0 & 0 & \cdots & \\
\hline
 & x(-1) & x(0) & x(1) & \cdots & x(n-1) & x(n) & x(n+1) & \cdots & \rightarrow x
\end{array}$$

zusammensetzen lässt. Wird an dieser Stelle statt eines Filters nur eine Verschiebung verwendet, so ergibt sich eine Zerlegung des Signals modulo n (Vergleiche auch [20]).

2.2.2 Analyse- und Synthese-Filterbank

Aufgebaut ist eine Filterbank aus zwei Komponenten, einem Analyseteil, der das Signal in die Subbänder zerlegt, und einem Syntheseteil, der die einzelnen Subbänder wieder zusammenfügt. Im folgenden wird der Analyseteil als Analyse-Filterbank und der Syntheseteil als Synthese-Filterbank bezeichnet. Beide ergeben zusammen die vollständige Filterbank (Abb. 2.3).

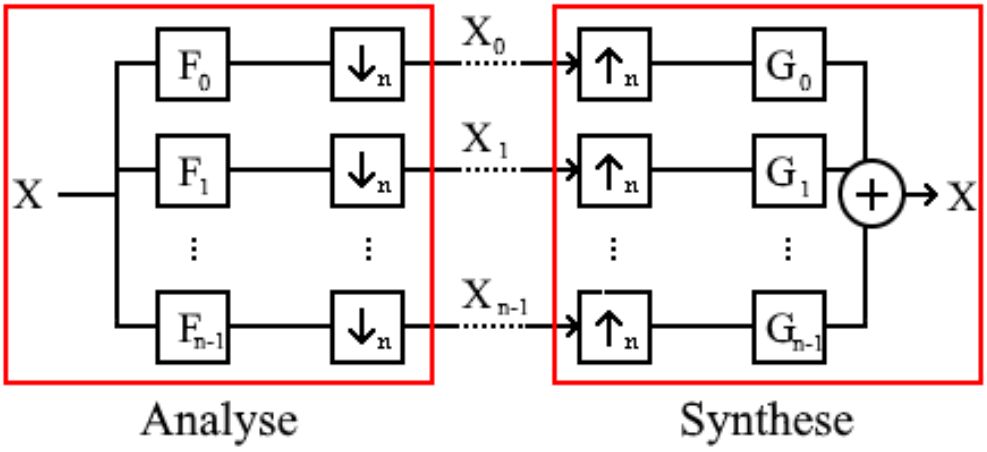


Abbildung 2.3: Schema einer Filterbank mit N Filtern. Zerlegung eines Signals X in N Subbänder X_i und anschließendes Zusammensetzen.

Ist der Faktor N der Abtastung ungleich der Anzahl der Filter spricht man von Unterabtastung ($N <$ #Filter) bzw. Überabtastung ($N >$ #Filter). Ist N gleich der Anzahl der Filter wird das als kritische Abtastung (*critically sampled*) bezeichnet.

2.2.3 Perfekte Rekonstruktion

Eine besonders interessante und wichtige Eigenschaft einer Filterbank ist die perfekte Rekonstruktion (PR). Diese wird nun im folgenden Anhand einer Zweikanal-Filterbank vorgestellt[1].

1 Das folgende Beispiel stammt zu großen Teilen aus [25] und [31] und wird hier nur verkürzt vorgestellt um die PR-Eigenschaft einer Filterbank zu beschreiben. Für genauere Informationen wird auf die Quellen verwiesen.

Nimmt man eine Zweikanal-Filterbank mit Analyse-Filtern $H_0(z)$,$H_1(z)$ und Synthese-Filtern $G_0(z)$,$G_1(z)$ sowie kritischer Abtastung, so erhält man für ein Signal $X(z)$ folgende Subbänder [25]:

$$\begin{aligned} X_0\left(z^2\right) &= \frac{1}{2}\left[H_0(z)\,X(z) + H_0(-z)\,X(-z)\right] \\ X_1\left(z^2\right) &= \frac{1}{2}\left[H_1(z)\,X(z) + H_1(-z)\,X(-z)\right] \end{aligned} \tag{2.9}$$

oder in Matrixform:

$$\begin{bmatrix} X_0\left(z^2\right) \\ X_1\left(z^2\right) \end{bmatrix} = \frac{1}{2}\begin{bmatrix} H_0(z) & H_0(-z) \\ H_1(z) & H_1(-z) \end{bmatrix}\begin{bmatrix} X(z) \\ X(-z) \end{bmatrix} \tag{2.10}$$

Mittels Synthese lassen sich die Subbänder $X_0(z)$,$X_1(z)$ wieder zusammensetzen, man erhält als Ergebnis das rekonstruierte Signal $X'(z)$:

$$\begin{aligned} X'(z) &= G_0(z)\,X_0\left(z^2\right) + G_1(z)\,X_1\left(z^2\right) \\ &= \begin{bmatrix} G_0(z) & G_1(z) \end{bmatrix}\begin{bmatrix} X_0\left(z^2\right) \\ X_1\left(z^2\right) \end{bmatrix} \end{aligned} \tag{2.11}$$

Setzt man nun Gleichung 2.10 in Gleichung 2.11 ein erhält man als Formel für die Filterbank:

$$X'(z) = \frac{1}{2}\begin{bmatrix} G_0(z) & G_1(z) \end{bmatrix}\begin{bmatrix} H_0(z) & H_0(-z) \\ H_1(z) & H_1(-z) \end{bmatrix}\begin{bmatrix} X(z) \\ X(-z) \end{bmatrix} \tag{2.12}$$

Perfekte Rekonstruktion bedeutet nun, dass das Ausgangssignal $X'(z)$ eine evtl. verzögerte und skalierte Form des Eingangssignals $X(z)$ ist. Es muss also gelten:

$$X'(z) = cz^{-k}X(z) \tag{2.13}$$

Anhand Gleichung 2.12 kann man jetzt erkennen, dass sich das rekonstruierte Signal aus zwei Komponenten zusammensetzt [25]:

$$\begin{aligned} X'(z) &= \frac{1}{2}\left[G_0(z)H_0(z) + G_1(z)H_1(z)\right]X(z) \\ &\quad + \frac{1}{2}\left[G_0(z)H_0(-z) + G_1(z)H_1(-z)\right]X(-z) \\ &= F_0(z)X(z) + F_1(z)X(-z) \end{aligned} \tag{2.14}$$

Um ein Signal also perfekt zu rekonstruieren sind zwei Bedingungen notwendig. Zum einen muss der Term $X(-z)$ eliminiert werden, da dieser eine Aliasingkomponente ist und Frequenzen enthält, die nicht Teil des Signals sind [25]. Dies lässt sich über die Bedingung $F_1(z) = 0$ erreichen. Die zweite Bedingung hängt mit der Definition von PR zusammen (Gleichung 2.13). $X'(z)$ darf also höchstens eine skalierte und verzögerte Form von $X(z)$ sein. Dies lässt sich durch $F_0(z) = z^{-l}$ erreichen. Man erhält also folgende Bedingungen für PR:

- Verzerrungsfreiheit
$$G_0(z)H_0(z) + G_1(z)H_1(z) = 2z^{-l} \tag{2.15}$$
- Aliasfreiheit
$$G_0(z)H_0(-z) + G_1(z)H_1(-z) = 0 \tag{2.16}$$

Anhand dieser Bedingungen lassen sich nun Filter generieren, die eine Filterbank mit perfekter Rekonstruktion bilden können.

Quadrature Mirror Filter

Ein Ansatz für Filter, die das PR-Kriterium erfüllen sind die sogenannten Quadrature Mirror Filter (QMF), wie sie zum Beispiel in [31] vorgestellt werden.

Die QMF-Lösung für die Bedingungen 2.15 und 2.16 lautet folgendermaßen [31]:

$$\begin{aligned} H_1(z) &= H_0(-z) \\ G_0(z) &= H_0(z) \\ G_1(z) &= -H_1(z) = -H_0(-z) \end{aligned} \tag{2.17}$$

Setzt man die Gleichungen aus 2.17 in Bedingung 2.16 ein, so erhält man

$$\begin{aligned} G_0(z) H_0(-z) + G_1(z) H_1(-z) &= H_0(z) H_0(-z) - H_0(z) H_0(-z) \\ &= 0 \end{aligned} \tag{2.18}$$

Die QMF erfüllen also die Bedingung der Aliasfreiheit. Zusätzlich erhält man für Bedingung 2.15 folgendes

$$H_0^2(z) - H_0^2(-z) = 2z^{-l} \tag{2.19}$$

Da $H_0(-z)$ die Spiegelung von $H_0(z)$ ist erklärt sich damit auch der Name Quadrature Mirror Filter, da sowohl Filter als auch Spiegelung quadriert werden [31].

Ein Beispiel für einen QMF ist das Haar-Wavelet mit [31]

$$H_0(z) = \frac{1}{\sqrt{2}}\left(1 + z^{-1}\right) \tag{2.20}$$

Eingesetzt in Gleichung 2.19 erhält man:

$$\frac{1}{2}\left(1 + 2z^{-1} + z^{-2}\right) - \frac{1}{2}\left(1 - 2z^{-1} + z^{-2}\right) = 2z^{-1} \tag{2.21}$$

2.3 Diskrete Wavelet-Transformation

Ein Beispiel für die Verwendung von Filterbänken ist die diskrete Wavelet-Transformation (DWT). Dabei wird eine Kaskadierung von Filterbänken unter Verwendung von sogenannten Wavelet-Filtern durchgeführt.

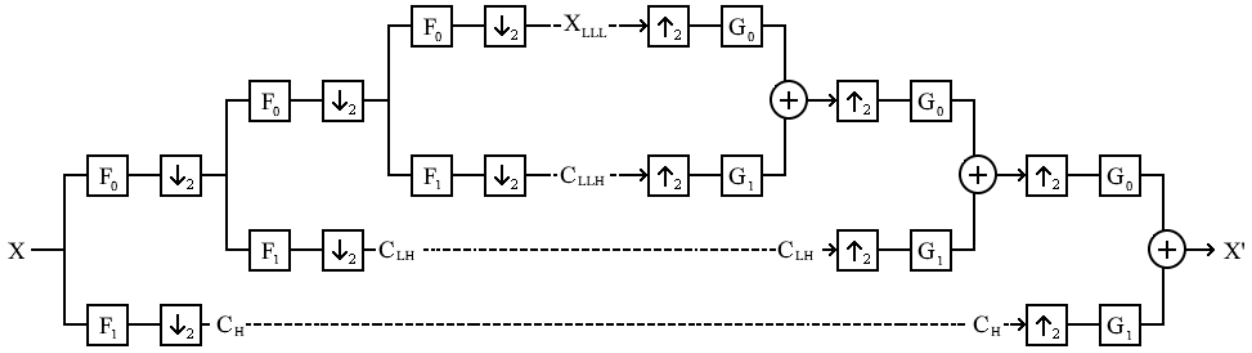


Abbildung 2.4: Filterbankkaskade der Tiefe 3

Bei der Kaskadierung von Filterbänken wird das Ergebnis des Tiefpassfilters einer Filterbank als Eingangssignal für die nächste verwendet, sodass man für ein Signal $x(n)$ folgende Zerlegung erhält [21]:

$$\begin{array}{ccccccccc} x = x^0 & \longrightarrow & x^1 & \longrightarrow & x^2 & \longrightarrow & \dots & \longrightarrow & x^n \\ & \searrow & & \searrow & & & & \searrow & \\ & & c^1 & & c^2 & & & & c^n \end{array}$$

Das Ergebnis des Hochpassfilters, die sog. Wavelet-Koeffizienten c^i werden gespeichert und das Ergebnis des Tiefpassfilters x^i als Eingabe für die $i+1$-te Stufe der Kaskade verwendet. Das Ergebnis einer Analysekaskade ist also eine Anzahl von Subbändern $\left(c^0, c^1, \cdots, c^n, x^n\right)$.

Die Subbänder x^n, c^i lassen sich nun mit einer Synthesekaskade wieder zusammensetzen (siehe Abbildung 2.4). Man erhält folgendes:

$$\begin{array}{ccccccccc} x^n & \longrightarrow & x^{n-1} & \longrightarrow & \ldots & \longrightarrow & x^1 & \longrightarrow & x^0 = x' \\ & \nearrow & & \nearrow & & & & \nearrow & \\ c^n & & c^{n-1} & & & & c^1 & & \end{array}$$

Unter der Voraussetzung von PR gilt dann auch hier für das Ergebnis $x'(n)$:

$$x'(n) = x(n) \tag{2.22}$$

man erhält also wieder das Eingangssignal $x(n)$.

2.3.1 DWT auf Bildern

Besonders interessant wird die DWT auf Bildern. Im ersten Schritt erfolgt die Filterung und Zerlegung zeilenweise, im zweiten spaltenweise (siehe Abb 2.5)[20]. Man erhält also vier Subbänder TT, TH, HT und HH. Der erste Buchstabe steht dabei für den auf die Spalten angewendeten Filter (Tiefpass T oder Hochpass H), der zweite für den jeweiligen Filter auf den Zeilen.

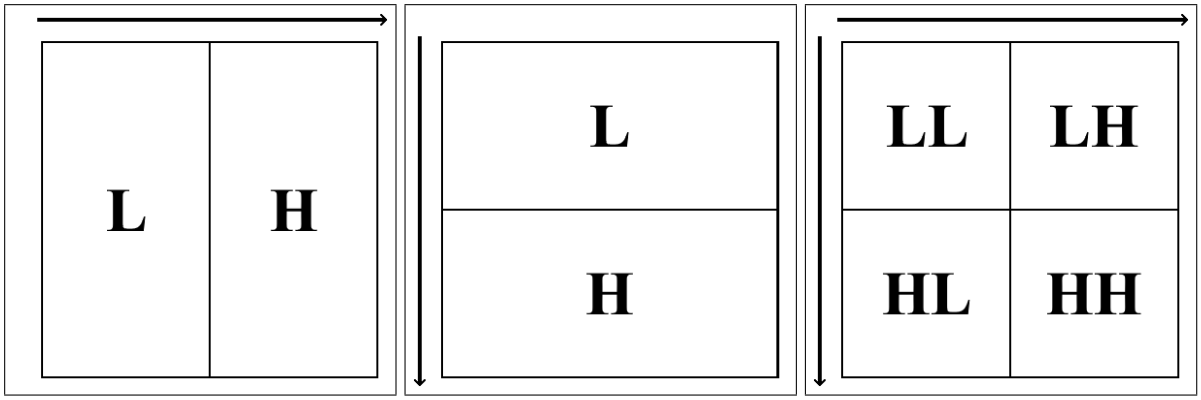


Abbildung 2.5: Zweidimensionale Zerlegung: Zeilenweise (a), Spaltenweise (b) und kombiniert (c). **H** bezeichnet dabei die Anwendung des Hochpass-Filters (High-Pass), **L** die des Tiefpass-Filters (Low-Pass)

Natürlich kann man die Filterbänke auch hier kaskadieren. Dazu wird das (doppelt) tiefpassgefilterte Teilsignal TT als Eingabe für die nächste Kaskadenstufe verwendet (siehe Abb. 2.6 und 2.7).

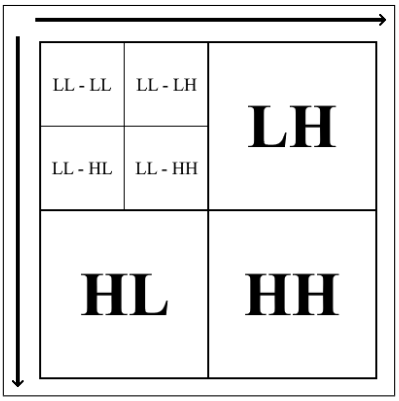


Abbildung 2.6: Kaskade mit Tiefe 2

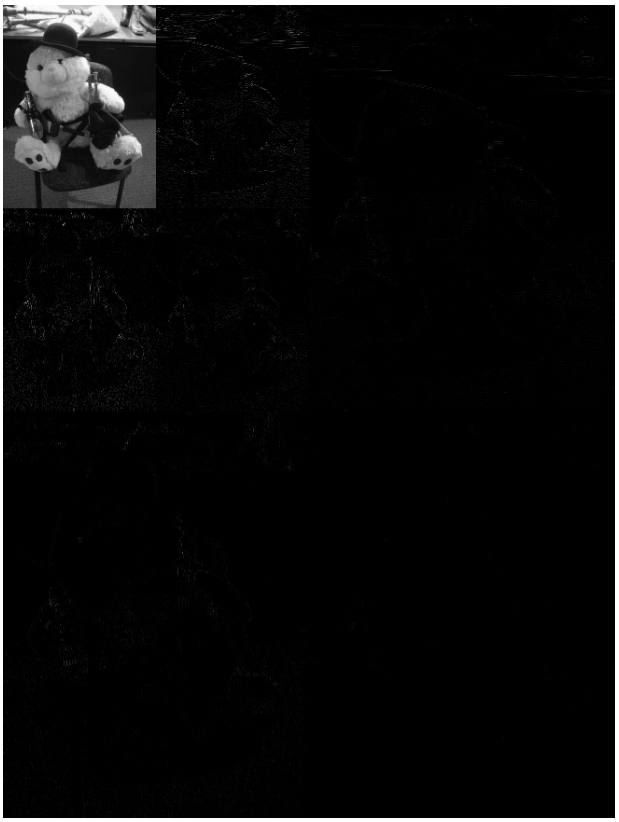


Abbildung 2.7: Beispiel für eine DWT der Tiefe 2 auf Testbild "Harvey"

2.3.2 Beispiel: JPEG2000-Standard

Ein Anwendungsbereich für die DWT ist zum Beispiel der JPEG2000 Standard zur Bildkompression. Dabei macht man sich zunutze, dass die Waveletkoeffizienten im Normalfall

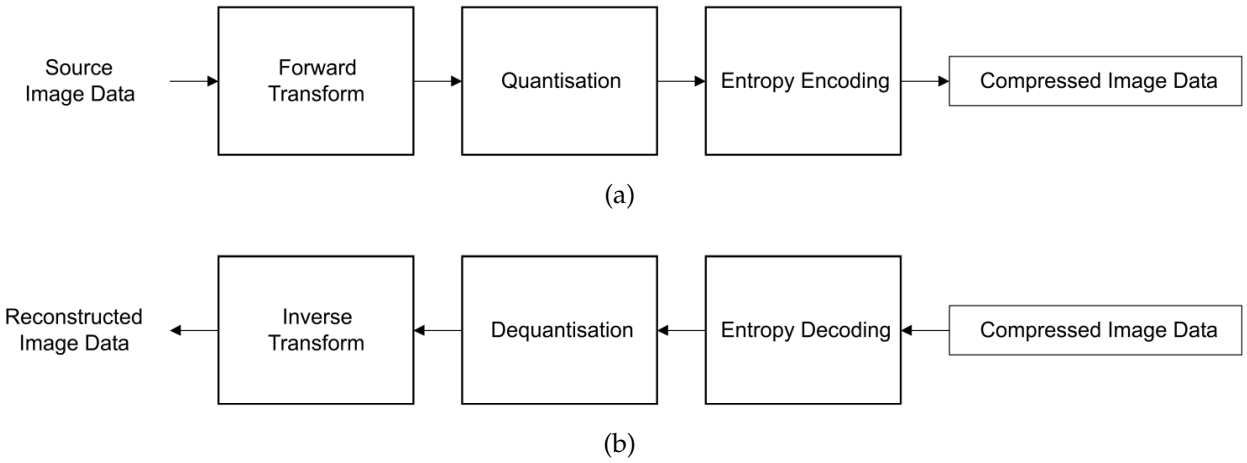


Abbildung 2.8: Blockdiagramm für JPEG2000 Codierung (a) und Decodierung (Quelle: [22])

dünn verteilt sind (Vergleiche zum Beispiel Abb. 2.7).

JPEG2000 besteht aus drei Stufen (Abb 2.8): DWT, Quantisierung und Codierung.
Zunächst wird das Eingangsbild in einzelne rechteckige Segmente, sogenannte Tiles unterteilt, die einzeln codiert werden. Auf die einzelnen Tiles wird im ersten Schritt eine DWT angewendet, wobei der Standard zwei verschiedene Wavelet-Filter vorsieht. Zum einen verwendet man das 9/7 Daubechies Wavelet, das aufgrund der Verwendung von Fließkommazahlen zur irreversiblen Transformation verwendet wird, zum anderen ein 5/3 LeGall Wavelet für reversible Transformation, da dieses aus Festkommazahlen besteht [22].

Die zweite Stufe ist eine Quantisierung der einzelnen Subbänder, ein Prozess bei dem die Genauigkeit der Waveletkoeffizienten reduziert wird [23]. Im Gegensatz zum 9/7-Wavelet ist diese Operation bei Verwendung des 5/3-Wavelets üblicherweise nicht verlustbehaftet [23] und wird aus diesem Grunde manchmal übersprungen [25].

Zuletzt werden die einzelnen Subbänder in Blöcke aufgeteilt und unabhängig voneinander mittels Entropieencoding codiert.

2.4 OpenCL-Grundlagen

OpenCL[2] (auch OpenCL C [11]) ist eine standardisierte Schnittstelle zur Entwicklung von parallelisierten Applikationen, die die Rechenkapazitäten von entsprechenden Prozessoren ausnutzen können. Das OpenCL Modell ist mit dem CUDA Modell vergleichbar, wobei anders als bei NVIDIA CUDA die Anwendung nicht nur auf GPUs eines einzelnen Herstellers beschränkt ist, sondern auch CPUs und andere Prozessortypen verwendet werden können.

2.4.1 Device-Architektur

Das OpenCL Modell setzt sich aus dem Host, der den allgemeinen Teil eines Programms ausführt, sowie ein oder mehrere Devices, die den parallelisierten OpenCL-Teil ausführen, zusammen. Ein Device (Abb. 2.9) besteht dabei aus einem oder mehren Prozessoren, sogenannten *compute units* (CUs), die wiederum aus einem oder mehreren *processing elements* (PEs) bestehen [11].

Die einzelnen PEs haben dabei Zugriff auf verschiedene Typen von Speicher. Der *Global Memory* dient zum Speichern von Daten, die bearbeitet werden sollen. Dieser ist der größte verfügbare Speicher, allerdings ist der Zugriff darauf im Vergleich zu den restlichen Speicherarten relativ langsam.

Der *Constant Memory* ist ebenfalls Teil des Global Memory. Dieser ist vor allem für Daten wie zum Beispiel Konstanten gedacht, die nicht verändert werden und auf die häufig zugegriffen werden muss. Aus diesem Grund werden Zugriffe auf den Constant Memory stark gecached, was im Schnitt für eine schnellere Zugriffszeit sorgt [11]

Local Memory und *Private Memory* befinden sich direkt im Device (im Gegensatz zu Global und Constant) und haben daher schnelle Zugriffszeiten. Variablen im Local Memory sind für alle PEs einer CU sichtbar, der Private Memory dient zur Speicherung von Werten für jede PE einzeln.

2 https://www.khronos.org/opencl/

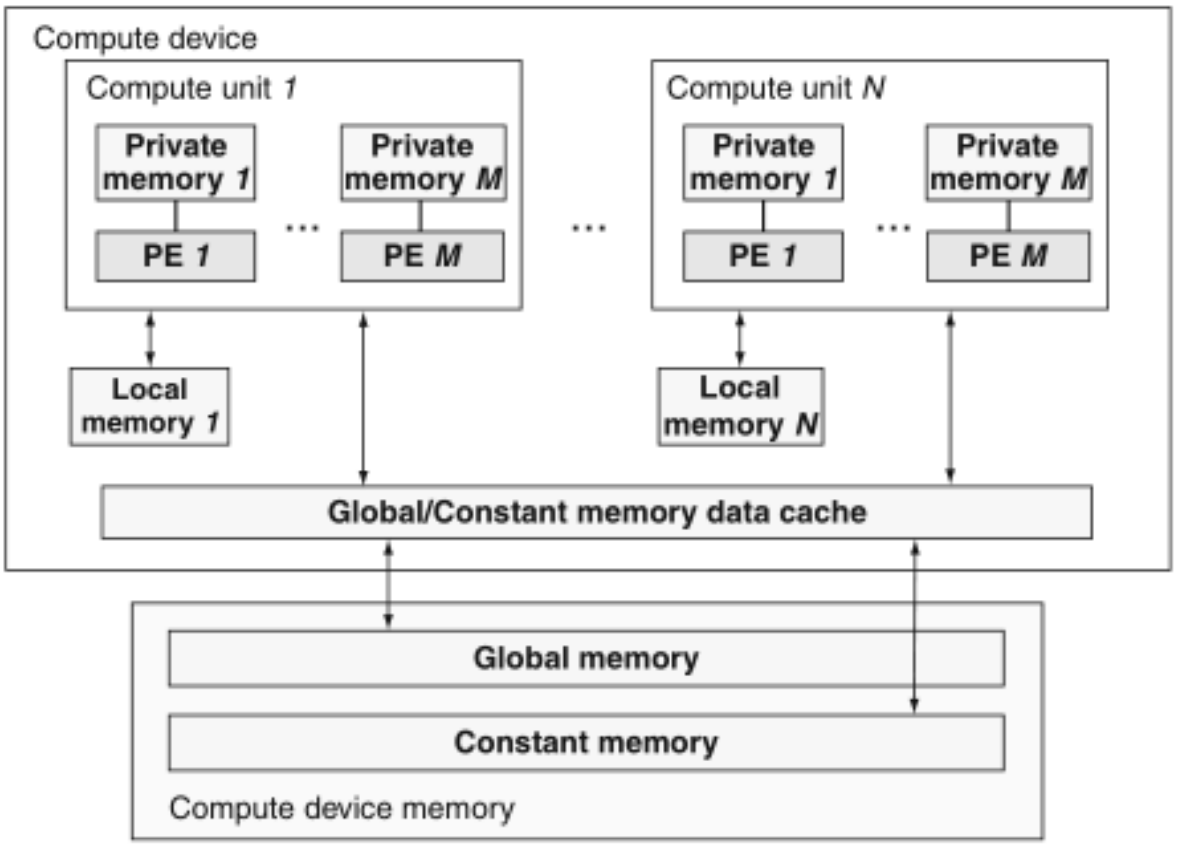


Abbildung 2.9: OpenCL Device-Architektur (Quelle: [11]).

2.4.2 Kernel

Die parallelisierten Teile eines OpenCL Programms werden als sogenannte Kernel implementiert. Dabei handelt es sich um Funktionen, die auf dem Device ausgeführt werden.

Zum Ausführen einer Kernel-Funktion muss zunächst festgelegt werden, wie viele *work items* (oder Threads) ihn ausführen sollen. Danach wird für jedes work item eine Instanz des Kernels angelegt und diese ausgeführt. Jedes work item führt dabei den gleichen Kernel (auf evtl. unterschiedlichen Daten) aus. Die work items lassen sich dabei zu *work groups* (Blöcken) zusammenfassen. Damit die einzelnen Kernel-Instanzen auf verschiedene Daten zugreifen können besitzt jedes work item und jede work group eine globale ID, die sich innerhalb der Kernel-Funktion abrufen lässt und die für den Zugriff auf verschiedene

Speicherbereiche verwendet werden kann.

Listing 2.1: Beispiel für einen OpenCL Kernel zur Vektoraddition

```
__kernel void vector_addition(__constant int *vector_a,
                              __constant int *vector_b,
                              __global int *result,
                              const int vector_size)
{
    int tid = get_global_id(0);

    if (tid < vector_size)
    {
        result[tid] = vector_a[tid] + vector_b[tid];
    }

    barrier();
}
```

Ein Beispiel für einen Kernel zur Vektor-Addition findet man in Listing 2.1. Dabei werden zwei Vektoren `vector_a` und `vector_b` der Länge `vector_size` elementweise addiert und in einem Vektor `result` gespeichert. Der Befehl `barrier()` dient zur Synchronisation der work items innerhalb einer work group.

2.4.3 OpenCL Context

OpenCL Devices werden als sogenannter *Context* verwaltet. Dieser muss vor Aufruf einer Kernelfunktion anhand der im System vorhandenen OpenCL Plattform (z.B. NVIDIA CUDA) festgelegt werden. Ist der Context festgelegt lassen sich Kernel-Funktionen auf einem Device ausführen.

Der Context ist ein Adressraum für den Speicher der im Context angegebenen Devices [11]. Aus dem Context lässt sich nun eine *Command Queue* für ein Device erstellen, in die Aufgaben für das Device eingereiht werden, zum Beispiel kopieren auf und vom Device und Kernel-Aufrufe (siehe Abb. 2.10).

2.4.4 OpenCL auf GPUs

Da der Schwerpunkt dieser Arbeit auf der Implementierung einer GPU-basierten Filterbank liegt, werden hier einige Informationen vorgestellt.

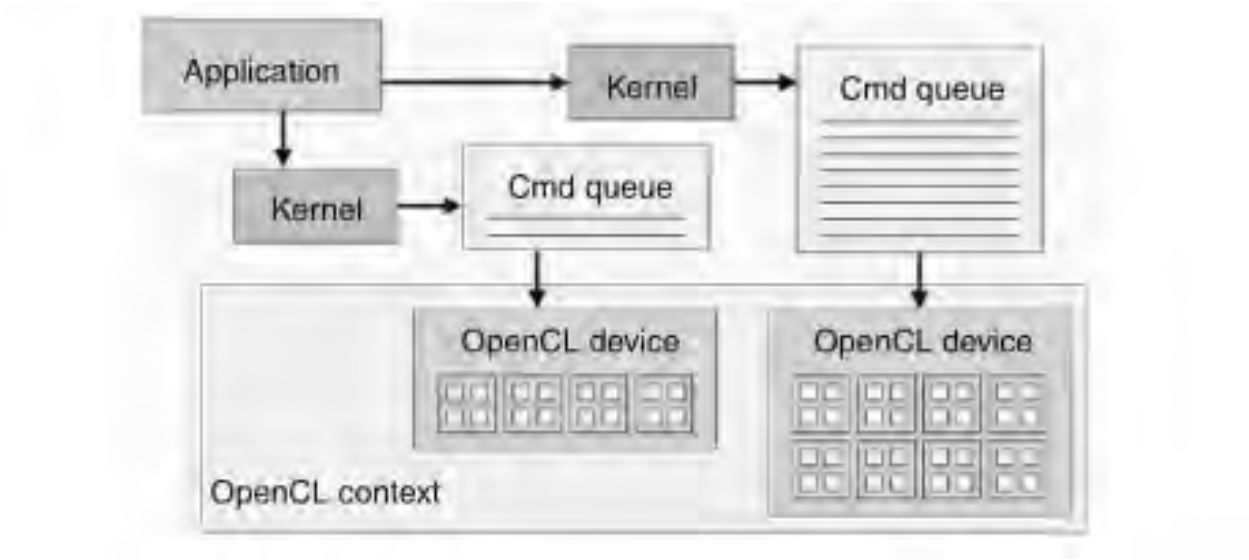


Abbildung 2.10: OpenCL Context zur Verwaltung von Devices (Quelle: [11]).

Der Aufbau einer GPU besteht aus einer Anzahl an Multiprozessoren, sog. *streaming multiprocessors*. Jeder Multiprozessor besteht aus einer Gruppe von einzelnen *streaming processors*, die sich Aufgabencache und Kontrolllogik teilen [11].

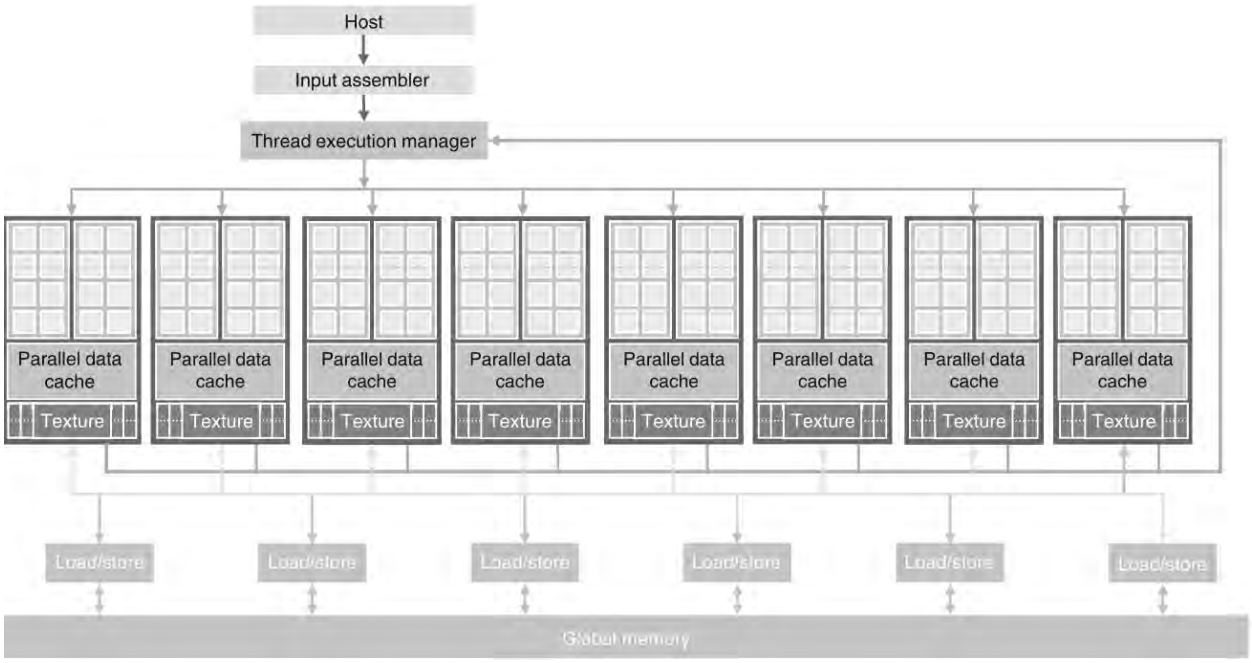


Abbildung 2.11: Aufbau einer GPU (Quelle: [11]).

Zur Ausführung werden die einzelnen work items zu sogenannten warps (NVIDIA) [16] oder wavefronts (AMD) [7] zusammengefasst. Jede Instruktion eines Kernels wird gleichzeitig auf alle work items des warps angewendet. Dieses Modell wird als *Single Instruction, Multiple Threads* (SIMT) bezeichnet [16].

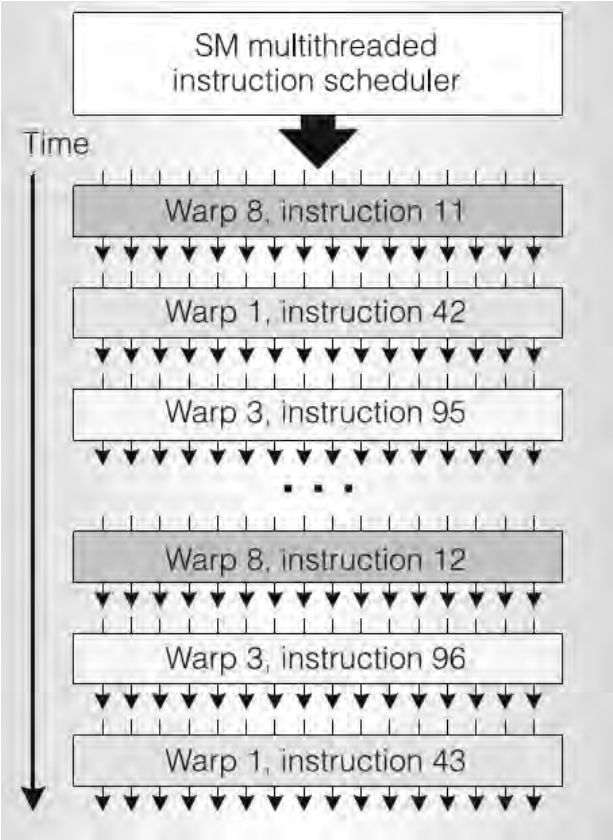


Abbildung 2.12: Ausführung von Thread-Warps (Quelle: [12]).

3 Implementierung mit OpenCL

Im folgenden Kapitel werden die Implementierungen von Filterbank-Strukturen für eindimensionale und zweidimensionale Daten besprochen. Zunächst werden die einzelnen Anforderungen an die Implementation sowie deren Auswirkungen vorgestellt und beschrieben. Der zweite Teil dient zur Erklärung des Aufbaus der Filter-Kernel sowie der Formeln für Sampling und Faltung. Im letzten Abschnitt beinhaltet die Implementierung der Formeln in die einzelnen Kernel-Funktionen und eine Beschreibung derselbigen, sowie den allgemeinen Aufbau der Filterbank und den Ablauf der Filterung anhand von Ablaufdiagrammen.

3.1 Anforderungen an die Implementierung

Im Rahmen dieser Arbeit wurden verschiedene Anforderungen an die Implementierung einer OpenCL-Filterbank gestellt, die diese zum Teil von anderen Implementierungen unterscheiden können.

3.1.1 Modifikation von Länge und Nullpunkt

Die wichtigste Anforderung ist eine Modifikation der Länge durch die Faltung und der damit verbundenen Verschiebung des Nullpunktes.

Im Rahmen der Implementierung werden sowohl Daten als auch Filter als Vektoren gegeben. Für den Filter kann ein Nullpunkt vorgegeben werden, der den Ausgangspunkt der Faltung festlegt, also diejenige Position im Datenvektor, um die herum die Werte für die Faltung verwendet werden. Bei einer Änderung des Nullpunktes verschiebt sich auch

der Bereich im Vektor, aus dem die Werte zur Faltung verwendet werden (siehe Abbildung 3.1).

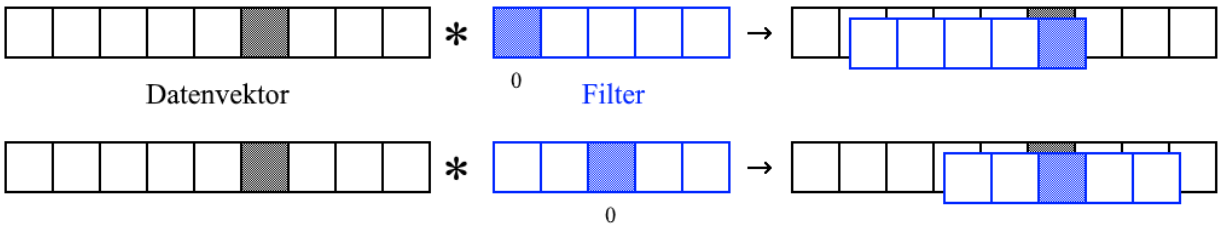


Abbildung 3.1: Änderung des Wertebereichs bei der Faltung durch Filter mit verschiedenen Nullpunkten. Die markierten Bereiche in Filter und Daten repräsentieren dabei die Nullpunkte.

Die einzelnen Daten-Positionen im Filter-Vektor bezeichnet man als Taps. Wie bereits im Abschnitt 2.1.2 besprochen ist es möglich, dass an den Rändern des Datenvektors einzelne Filter-Taps mit Werten gefaltet werden sollen, die nicht mehr Teil der Eingabedaten sind. Dies führt zu den beiden Grenzfällen, die im Abschnitt 2.1.2 besprochen wurden.
Um nun das Grenzverhalten besser nachvollziehen zu können, kann man die Werte, auf die außerhalb des Datenvektors zugegriffen wurde, in das Ergebnis aufnehmen, was zu einer Vergrößerung des vom Datenvektor gegebenen Bereichs führt. Die genaue Änderung der Vektorlänge lässt sich durch

$$len\left(\mathbf{v}_{neu}\right) = len\left(\mathbf{v}_{alt}\right) + \#Taps - 1 \tag{3.1}$$

berechnen, da ein Filter der Länge 1 keine Änderung an der Länge des Vektors hervorruft.

Im folgenden werden nun die Taps, deren Position im Filtervektor kleiner als die des Nullpunkts ist, als *negative Taps* (Taps mit negativem Index, ausgehend vom Nullpunkt) und deren Position größer als die des Nullpunkts ist als *positive Taps* bezeichnet. Offensichtlich gilt also, dass Änderung der Länge des Datenvektors am Anfang gleich der Anzahl der negativen Taps und am Ende gleich der Anzahl der positiven Taps ist (was zu einer Gesamtänderung der Länge um (#Taps - 1) führt).

Das hat zur Folge, dass sich die Positionen der Werte im Datenvektor (und damit der Nullpunkt des Datenvektors) im Vergleich zum ursprünglichen Vektor nach rechts verschieben

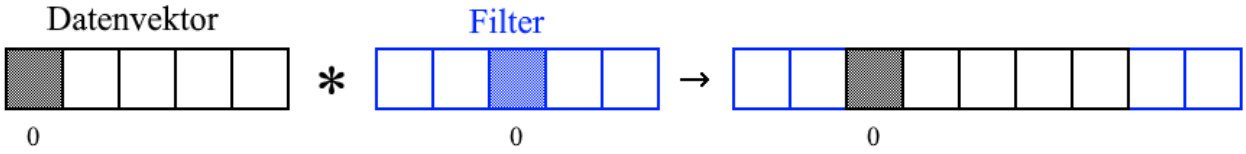


Abbildung 3.2: Änderung der Länge eines Vektors durch Faltung.

können (siehe Abbildung 3.2).

Im weiteren Verlauf wird damit nun zwischen dem **Positionsbereich** des Vektors (von 0 bis $len\,(\mathbf{v}-1)$) und dem **Indexbereich** (von $-Nullpunkt$ bis $len\,(\mathbf{v}-1)-Nullpunkt$) unterschieden. Damit gilt auch, dass der Nullpunkt den Index 0 hat, aber die Position des Nullpunktes ≥ 0 im Vektor ist. Kurz gesagt, der Positionsbereich ist der normale Bereich in dem ein Vektor arbeitet, der Indexbereich dient zur Repräsentation der Verschiebung durch die Faltung. Der Nullpunkt ist also die Position im Vektor an der der Index 0 ist.

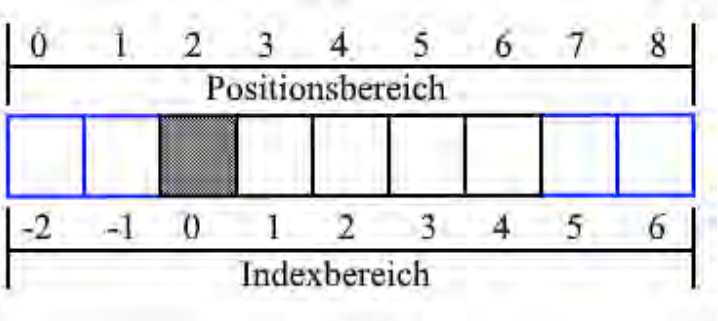


Abbildung 3.3: Positionsbereich und Indexbereich.

Ist nun von der Position eines Wertes im Vektor die Rede wird damit ein Wert an einer Stelle ($0 \leq$ Position $<$ **v**), also im Positionsbereich bezeichnet.

Für den Zusammenhang zwischen Indexbereich und Positionsbereich gilt offensichtlich, dass

$$Position = Index + Nullpunkt \tag{3.2}$$

Des weiteren geht man für einen ungefilterten Vektor (der nicht ein Ergebnis der Filterbank ist) davon aus, dass

$$Position = Index \tag{3.3}$$

und damit

$$Nullpunkt = \mathbf{v}[0] \tag{3.4}$$

Diese Anforderung hat zur Folge, dass beispielsweise die Speicherung des Ergebnisses bei der DWT nicht mehr in einer einzelnen Matrix möglich ist, da die einzelnen Subbänder unterschiedliche Längen haben und die klassische Aufteilung, wie zum Beispiel in Abbildung 2.7, nicht ohne zusätzliche Kopiervorgänge möglich ist.

3.1.2 Filterbänke beliebiger Größe

Ein weiterer Punkt ist die Zerlegung eines Signals in eine beliebige Anzahl an Subbändern. Im Rahmen der Implementierung werden die Filter nicht hart codiert, sondern vom Benutzer angegeben. Bei Erstellung einer Filterbank werden diese dann in den Speicher des Device kopiert und für die Zerlegung bzw. Synthese verwendet.

Das hat insbesondere Auswirkungen auf die zweidimensionale Filterbank, da dort horizontale und vertikale Filter getrennt angegeben werden und man so unterschiedliche Aufteilungen erhält.

3.2 Aufbau der Kernel-Funktionen

Die Kernel-Funktionen übernehmen in der Filterbank die Aufgabe von Sampling und Faltung. Sie werden in Analyse-Kernel und Synthese-Kernel unterteilt.
Jeder Kernel erhält als Input einen einzelnen Datenvektor sowie einen Filter und liefert als Ergebnis entweder ein gefiltertes und downgesampletes Subband (Analyse-Kernel)

oder addiert als Teil der Synthese ein upgesampletes und gefiltertes Subband auf einen Ergebnisvektor.

Bevor nun die Kernel im einzelnen besprochen werden zunächst einige Grundlagen.

3.2.1 Berechnung von Länge und Nullpunkt

Um mit einem Kernel aus einem Speicherbereich zu lesen bzw. in einen zu schreiben, muss dieser zunächst alloziert werden. Dazu wird zunächst die benötigte Größe des Speichers berechnet, was für Analyse und Synthese getrennt geschieht.

Analyse

Für einen Vektor $\mathbf{v}$ mit Länge $len\,(\mathbf{v})$ und Nullpunkt $z_{\mathbf{v}}$ sowie einen Filter $\mathbf{f}$ mit Länge $len\,(\mathbf{f})$ und Nullpunkt $z_{\mathbf{f}}$ lässt sich die Größe des Ergebnisvektors nach Faltung und Downsampling mit folgendem Ansatz berechnen:

Zuerst ermittelt man die Länge $len\,(\mathbf{v}_{conv})$ des Vektors nach der Faltung mit

$$len\,(\mathbf{v}_{conv}) = (len\,(\mathbf{v}) + len\,(\mathbf{f}) - 1) \tag{3.5}$$

Im zweiten Schritt zieht man die Anzahl der Elemente ab, die am Anfang des Vektors durch das Downsampling verloren gehen.

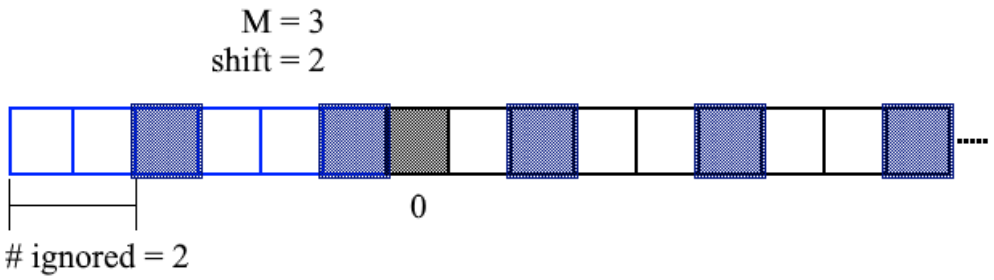


Abbildung 3.4: Darstellung der ignorierten Elemente beim Downsampling um den Faktor $M = 3$ mit $shift = 2$. Die Elemente an Position 0 und 1 sind beim Downsampling nicht Teil des Vektors, daher geht beim Upsampling dieser Teil verloren.

Dazu wird zunächst die Gesamtverschiebung des Nullpunktes aus bisherigem Nullpunkt und neuer Verschiebung durch den Filter berechnet und zum Ergebnis der $shift$-Wert für das Downsampling addiert. Modulo M erhält man so die Anzahl der Elemente, die beim Downsampling nicht beachtet werden.

$$\#ignored = \Big(z_v + z_f + shift\Big) \bmod M \tag{3.6}$$

Zieht man diesen Wert nun von der Länge des gefilterten Vektors $len\left(\mathbf{v}_{conv}\right)$ ab, so erhält man die Länge eines (theoretischen) Vektors, bei dem jedes M-te Element nach dem Downsampling im Ergebnisvektor vorhanden ist. Man kann die Länge nach dem Downsampling also mit

$$len\left(\mathbf{v}_{result}\right) = max\left(\left\lceil \frac{len\left(\mathbf{v}_{conv}\right) - \#ignored}{M} \right\rceil, 0\right) \tag{3.7}$$

berechnen. Die Überprüfung auf ≥ 0 ist notwendig, da bei sehr kleinen Vektoren $len\left(\mathbf{v}_{conv}\right) - \#ignored$ negativ werden kann. Da sich solche Vektoren mit $len\left(\mathbf{v}_{result}\right) = 0$ zum einen nicht vernünftig behandeln lassen und zum anderen ohnehin kein Subband entstehen würde, werden diese beim Ausführen der Filterbank ignoriert.

Der Nullpunkt z_{conv} lässt sich ähnlich berechnen. Dazu wird die Gesamtverschiebung berechnet und diese dann durch den Sampling-Faktor M geteilt und abgerundet (Vergleiche auch Gleichung 3.6):

$$z_{conv} = \left\lfloor \frac{z_v + z_f + shift}{M} \right\rfloor \tag{3.8}$$

Synthese

Für die Synthese muss man aus j gegebenen Subbändern $\mathbf{v}_j$, $j \in \mathbb{N}$
0 und Filtern $\mathbf{f}_j$ die Länge des synthetisierten Vektors berechnen. Dazu trennt man die Berechnung in zwei Teile auf, zum einen berechnet man die maximale Länge nach dem Upsampling der Teilvektoren $\mathbf{v}_{j,<0}$, die sich aus den Indices < 0 zusammensetzen, zum anderen berechnet man die maximale Länge der Teilvektoren $\mathbf{v}_{j,\geq,}$, bestehend aus den

jeweiligen Elementen mit Index ≥ 0.
Man erhält folgendes für die Länge des upgesampleten Vektors $\mathbf{v}_{result}$:

$$len\left(\mathbf{v}_{result}\right) = max\left(len\left(\mathbf{v}_{j,<0}\right)\right) + max\left(len\left(\mathbf{v}_{j,\geq 0}\right)\right) \tag{3.9}$$

Für die Teillänge $len\left(\mathbf{v}_{j,<0}\right)$ verwendet man das Maximum aller upgesampleten Teillängen der Subbänder mit Indices < 0. Es gilt für die Subbänder $\mathbf{v}_j$:

$$len\left(\mathbf{v}_{j,<0}\right) = max\left(\left(z_{v_j}M - shift\right), 0\right) + z_{f_j} \tag{3.10}$$

$\left(z_{v_j}M - shift\right)$ ist dabei die Länge des Teilvektors nach dem Upsampling. Da für kleine z_{v_j} (z.B. $z_{v_j} = 0$) dieser Term negativ werden kann, muss man zunächst wieder das Ergebnis auf ≥ 0 setzen. Mit z_{f_j} wird dann Veränderung der Größe nach Links dazu addiert.

Die upgesamplete Länge der Teilvektoren mit Indices ≥ 0 wird mit folgender Formel berechnet.

$$len\left(\mathbf{v}_{j,\geq 0}\right) = \#zeros + \#elements + \#pos\_taps + shift \tag{3.11}$$

Diese Formel setzt sich aus der Anzahl der Nullen zusammen, die dem Teilvektor $\mathbf{v}_{j,\geq 0}$ durch das Upsampling hinzugefügt wurden:

$$\#zeros = \left(len\left(\mathbf{v}_j\right) - z_{v_j} - 1\right)(M-1), \tag{3.12}$$

der Anzahl der Elemente in $\mathbf{v}_{j,\geq 0}$:

$$\#elements = len\left(\mathbf{v}_j\right) - z_{v_j}, \tag{3.13}$$

sowie dem Wert der Verschiebung durch das Upsampling $shift$ und der Anzahl der positiven Taps von Filter $\mathbf{f}_j$, was gleichbedeutend mit der Veränderung der Länge nach rechts ist:

$$\#pos\_taps = len\left(\mathbf{f}_j\right) - z_{f_j} - 1 \tag{3.14}$$

Die Gesamtlänge erhält man nun aus den maximalen Teillängen.

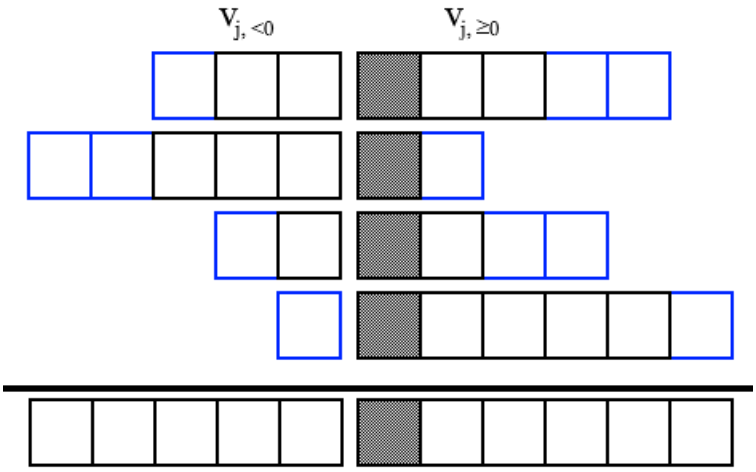


Abbildung 3.5: Berechnung der Länge des Synthese-Ergebnisses aus den maximalen upgesampleten Teilvektoren.

Die Berechnung des Nullpunktes geschieht mit folgender Formel:

$$z_{result} = max\left(max\left(\left(z_{v_j}M - shift\right), 0\right) + z_{f_j}\right) = max\left(len\left(\mathbf{v}_{j,<0}\right)\right) \tag{3.15}$$

3.2.2 Analyse-Kernel

Der Analyse-Kernel wird zur Zerlegung eines Eingabevektors in die einzelnen Subbänder verwendet. Dazu wird dieser nacheinander mit dem Eingabevektor und den verschiedenen Filtern und $shift$-Werten zum Downsampling aufgerufen, um das Subband für den gegebenen Filter zu generieren.

An dieser Stelle macht man sich die Tatsache zunutze, dass durch das Downsampling nur jede M-te Position im Subband wiederzufinden ist. Das ermöglicht es Rechenzeit bzw. work items zu sparen, indem man nur diese Positionen überhaupt bearbeitet.

Dazu geht man 'rückwärts' vor, indem man die jeweilige Position des Subbands auf die ursprüngliche Position im Eingabevektor zurückrechnet. Dabei muss man auch den für das Downsampling verwendeten $shift$ beachten.

Da man bei einem Sampling-Faktor von M jeden M-ten Wert verwendet, lässt sich die Position n des Ergebnisvektors mittels (Mn) auf die Position im Eingabevektor umrechnen. Hier muss man nun zusätzlich noch den $shift$-Wert beachten, welcher mittels eines $offset$s addiert wird und folgendermaßen definiert ist:

$$offset = \Big(\big(z_f + z_v + shift\big) \bmod M\Big) - z_f \tag{3.16}$$

Man erhält für eine Position n im Ergebnisvektor folgende Position im Eingabevektor:

$$n \leftarrow Mn + offset \tag{3.17}$$

Der $offset$ selbst setzt sich hauptsächlich aus $\big(z_f + shift\big) \pmod M$ zusammen. Die Addition des Nullpunktes z_v des Eingabevektors ist notwendig um eine bereits vorhandene Verschiebung (Beispielsweise durch Kaskadierung von Filterbänken verursacht) zu behandeln, da bei diesen ein Nullpunkt ≥ 0 möglich ist. Zuletzt wird noch der Nullpunkt des Filters abgezogen um die zusätzliche Verschiebung durch die Faltung zu beachten.
Es ergibt sich für die Faltung folgende Formel:

$$Subband = \sum_{k \in -z_f}^{len(\mathbf{f})-z_f-1} v\left[(Mn + offset) - k\right] d\left[k\right] \tag{3.18}$$

Damit bleibt noch die Behandlung der Grenzfälle der Faltung zu klären.
Wenn man bei der Faltung an die Definitionsgrenzen des Eingabevektors kommt, kann es passieren, dass man mit $(Mn + offset)$ auf Positionen zugreifen will, die nicht definiert sind. In diesem Fall muss man mittels einer Grenzfallprüfung eingreifen.

Zunächst wird überprüft, ob gilt $0 \leq (Mn + offset) < len\,(\mathbf{v})$. Falls $(Mn + offset) < 0$ oder $(Mn + offset) \geq len\,(\mathbf{v})$ hat man die Wahl zwischen der Fortsetzung mit Null oder der

zyklischen Fortsetzung.
Für die Nullfortsetzung werden die Werte folgendermaßen ausgewählt:

$$data = \begin{cases} v\left[i-k\right], & 0 \leq i-k < l_{in} \\ 0, & sonst \end{cases} \tag{3.19}$$

mit $i = Mn + offset$.

Im Fall der zyklischen Fortsetzung wird i einfach modulo der Länge des Vektors gerechnet, was zu den entsprechenden Positionen im Eingabevektor führt:

$$data = v\left[(i-k) \bmod len\left(\mathbf{v}\right)\right] \tag{3.20}$$

3.2.3 Synthese-Kernel

Der Kernel für die Synthese wird verwendet um die einzelnen Subbänder, die durch die Analyse generiert wurden, wieder zusammenzusetzen. Dazu werden die einzelnen Subbänder nacheinander zusammen mit dem zugehörigen Filter dem Kernel übergeben, welcher das upgesamplete Subband $\mathbf{v}_j$ mit dem Filter $\mathbf{f}_j$ faltet, und das Ergebnis dann elementweise auf den Ergebnisvektor addiert.

Da man das Ergebnis später in den Ergebnisvektor schreiben will, bietet es sich an, dessen Positionsbereich n als Ausgangsbasis zu verwenden. Das hat zur Folge, dass man jeden upgesampleten Eingabevektor mittels der Nullpunkte an diesen angleichen muss.

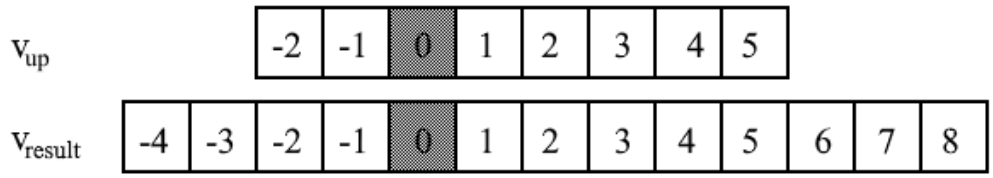


Abbildung 3.6: Angleichen von Eingabe- an Ergebnisindices anhand der Nullpunkte.

Dazu schränkt man den Bereich in dem n verwendet wird folgendermaßen ein:

$$z_{result} - z_{v_{up}} - z_f \leq n < z_{result} + \left(len\left(\mathbf{v}_{up}\right) - z_{v_{up}}\right) + \left(len\left(\mathbf{f}\right) - z_f - 1\right) \tag{3.21}$$

z_f ist hier die Modifikation der Länge nach links, $\left(len\left(\mathbf{f}\right) - z_f - 1\right)$ die Modifikation der Länge nach rechts, $z_{v_{up}}$ der upgesamplete Nullpunkt von $\mathbf{v}$, was gleichbedeutend mit der Anzahl der Elemente im upgesampleten Vektor mit Indices kleiner dem Index des Nullpunktes ist und $\left(len\left(\mathbf{v}_{up}\right) - z_{v_{up}}\right)$ die Anzahl der Elemente mit Indices größer Index des Nullpunktes.
Für die Position n_v als Position im upgesampleten Vektor gilt nun:

$$n_v = n - \left(z_{result} - z_{v_{up}}\right) \tag{3.22}$$

Der *index* für die Faltung ist damit folgendermaßen definiert:

$$index = (n - z_{result}) - k = i - k \tag{3.23}$$

Beim Upsampling um den Faktor M werden im Eingabevektor zwischen zwei Werten jeweils $M - 1$ Nullen platziert. Diese Information kann man sich ähnlich wie schon bei der Analyse zunutze machen, indem man anstatt den Vektor upzusamplen bei der Faltung überprüft, ob man auf eine Position zugreift, die Werte aus dem Eingabevektor enthält.

Dazu muss man zunächst anhand von $shift$, Nullpunkt und Sampling-Faktor die Positionen berechnen, welche die Werte im upgesampleten Vektor enthalten.
Es gilt für die Sampling-Position s folgendes:

$$s = \left(z_{v_{up}} + shift\right) \bmod M \tag{3.24}$$

mit $z_{v_{up}}$ als Nullpunkt des Eingabevektors nach dem Upsampling.

Bei der Faltung wird dann der *index* in die zugehörige Position im upgesampleten Vektor umgerechnet und das Ergebnis modulo M mit s verglichen:

$$\left(index + z_{v_{up}}\right) \bmod M == s \tag{3.25}$$

Wenn beide identisch sind (sich also Gleichung 3.25 zu `true` auswerten lässt), enthält *index* einen Wert aus dem Eingabevektor und man rechnet diesen dann auf die zugehörige Position im Eingabevektor um. Dazu verwendet man folgende Formel:

$$position = \frac{index - s + z_{v_{up}}}{M} \tag{3.26}$$

Falls gilt $-z_{v_{up}} \leq index < len\left(\mathbf{v}_{up}\right)$ und $\left(index + z_{v_{up}}\right) \neq s$ so zeigt *index* auf eine Position, die eine durch das Upsampling hinzugefügte 0 enthält. Es gilt also für *data*:

$$data = \begin{cases} \mathbf{v}\left[position\right], & 3.25\ wahr, \\ 0, & sonst \end{cases} \tag{3.27}$$

Falls *idnex* sich außerhalb des definierten Bereichs befindet, also $index < -z_{v_{up}}$ oder $index \geq len\left(\mathbf{v}_{up}\right)$, wird wieder eine Behandlung der Grenzfälle notwendig. Für die Nullfortsetzung wird dabei wie in Gleichung 3.19 vorgegangen, also *data* für jeden *index* in diesem Bereich auf 0 gesetzt.
Für die zyklische Fortsetzung wird *index* zunächst mit $\left(index + z_{v_{up}}\right)$ auf die Position umgerechnet und dann modulo $len\left(\mathbf{v}_{up}\right)$ auf den Positionsbereich des upgesampleten Eingabevektors umgerechnet. Zuletzt wird wieder mit Gleichung 3.25 überprüft, ob die Position einen Wert aus dem Eingabevektor enthält und, falls dem so ist, mit Gleichung 3.26 auf die zugehörige Position im Eingabevektor umgerechnet.
Zuletzt wird dann das Ergebnis der Faltung an die Stelle $\mathbf{v}_{result}\left[n\right]$ geschrieben.

3.3 Implementierung der Kernel-Funktionen

Im folgenden Abschnitt werden nun die Ergebnisse aus Abschnitt 3.2 dazu verwendet, die Kernel-Funktionen in OpenCL C zu implementieren. Zunächst werden die Kernel-Funktionen für eindimensionale Daten vorgestellt, welche dann auf zweidimensionale Daten erweitert werden.

3.3.1 Eindimensional - Analyse

Der Analyse-Kernel zur Zerlegung von eindimensionalen Daten generiert aus dem Eingabevektor, einem Filter sowie dem Sampling-Faktor und Shift-Wert ein gefiltertes, downgesampletes Subband.

Die Kernel-Funktion wird in N work items ausgeführt, wobei für N gilt $N \geq len\left(\mathbf{v}_{result}\right)$ sowie N ist Vielfaches der Größe eines *warp* (siehe Abschnitt 2.4.4) des ausführenden Device. Letztere Anforderung für N hängt mit der Art zusammen, wie eine GPU die einzelnen Threads bzw. work items bearbeitet [13].
In der Funktion selbst schränkt man die Anzahl der ausführenden work items ein, indem man überprüft, ob die ID des jeweiligen work items im Grid $< len\left(\mathbf{v}_{result}\right)$ ist. Man führt die Funktion also für jede Position im Ergebnisvektor aus.

Zusätzlich zu den bereits genannten wird noch ein Parameter für das Verhalten der Faltung an den Grenzen des Eingabevektors benötigt, der vom Benutzer festzulegen ist (0 für Nullfortsetzung bzw. 1 für Zyklische Fortsetzung). Der Funktionskopf des Analysekernels sieht damit folgendermaßen aus:

Listing 3.1: Funktionskopf des eindimensionalen Analysis-Kernels.

```
__kernel void filter1D_analysis(__global float *data,       //Eingabevektor
                                const int data_size,        //Länge Eingabe
                                const int data_zero,        //Nullpunkt Eingabe
                                __constant float *filter,   //Filtervektor
                                const int filter_size,      //Länge Filter
                                const int filter_zero,      //Nullpunkt Filter
                                __global float *result,     //Ergebnisvektor
                                const int result_size,      //Länge Ergebnis
                                const int border_type,      //Grenzverhalten (0:null, 1:zyklisch)
                                const int sampling_factor,  //Sampling-Faktor M
                                const int shift )           //Shift-Wert für Sampling
```

In der Funkion selbst muss zunächst die ID des work items (=Position im Ergebnisvektor) anhand von Gleichung 3.17 auf die entsprechende Position im Eingabevektor umgerechnet werden:

Listing 3.2: Berechnung der zu n gehörenden Position im Eingabevektor.

```
// data_pos = Mn + offset
int data_pos = (sampling_factor * tid_x)
                    + ((filter_zero + shift + data_zero) % sampling_factor)
                    - filter_zero;
```

Die Faltung selbst läuft im Analyse-Kernel folgendermaßen ab:

Listing 3.3: Faltung des Analyse-Kernels.

```
for ( int k = -filter_zero; k <= (filter_size - filter_zero - 1) ; k++ )
{
    pos = data_pos - k;

    /* Get data */
    /* c = ... */

    out += c * filter[k + filter_zero];
}
result[tid_x] = out;
```

Um an die Werte c zu kommen wird zunächst überprüft, ob pos sich innerhalb des Positionsbereichs des Eingabevektors befindet. Falls dies zutrifft kann man einfach mit $c = \mathbf{v}\,[pos]$ auf die Werte zugreifen.
Für den Fall, dass sich pos außerhalb des Positionsbereichs befindet, wird c anhand der Grenzfallbehandlung nach den Gleichungen 3.19 und 3.20 festgelegt. Die Implementierung sieht folgendermaßen aus:

Listing 3.4: Berechnung der Position von c im Synthese-Kernel.

```
if ( pos < data_size && pos >= 0)
{
    c = data[pos];
}
else
{
    if (border_type == CYCLIC)
    {
        c = data[ (pos + data_size) % data_size ];
    }
    else
    {
        c = 0;
    }
}
```

3.3.2 Eindimensional - Synthese

Ähnlich wie der Analyse-Kernel erhält auch der Synthese-Kernel als Parameter einen Eingabevektor (ein Subband), den zugehörigen Filter sowie den Ergebnisvektor, auf den

das upgesamplete, gefilterte Subband elementweise addiert wird. Der Funktionskopf der Synthese ist folgendermaßen aufgebaut:

Listing 3.5: Funktionskopf des eindimensionalen Synthese-Kernels.

```
__kernel void filter1D_synthesis(__global float *data,          //Eingabevektor
                                 const int data_size_upsampled,  //Länge EV nach Upsampling
                                 const int data_zero_upsampled,  //Nullpunkt EV nach Upsampling
                                 __constant float *filter,       //Filter
                                 const int filter_size,          //Länge Filter
                                 const int filter_zero,          //Nullpunkt Filter
                                 __global float *result,         //Ergebnisvektor
                                 const int result_size,          //Länge Ergebnisvektor
                                 const int result_zero,          //Nullpunkt Ergebnisvektor
                                 const int border_type,          //Variable für Grenzverhalten
                                 const int sampling_factor,      //Sampling-Faktor M
                                 const int sample_pos)           //Verschiebung der Position
                                                                 // der Werte nach dem Upsampling
```

Auch hier werden wieder N work items zur Ausführung des Kernels verwendet, mit $N \geq len\left(\mathbf{v}_{result}\right)$ und N Vielfaches der warp size des ausführenden Device.
Zusätzlich schränkt man die Anzahl der work items, die tatsächlich die Faltung ausführen sollen auf den Bereich ein, auf den das upgesamplete, gefilterte Subband addiert wird. Da man dieses anhand der Nullpunkte an den Ergebnisvektor angleicht, ergibt sich folgender ID-Bereich der work items (siehe auch Gleichung 3.21):

Listing 3.6: Einschränkung des ID-Bereiches der work items.

```
tid_x >= ( result_zero - data_zero_upsampled - filter_zero ) &&
tid_x < ( result_zero + ( data_size_upsampled - data_zero_upsampled )
                      + ( filter_size - filter_zero - 1 ) )
```

Die Faltung selbst läuft ähnlich ab wie bei der Analyse, der wesentliche Unterschied besteht dabei, wie man die Werte aus dem Eingabevektor erhält. Da man anders als beim Analyse-Kernel bei der Synthese einen downgesampleten Vektor als Eingabe erhält, muss man überprüfen, ob dieser nach dem Upsampling an der Position `pos` einen Wert aus dem Eingabevektor oder eine durch das Upsampling eingefügte Null enthält.

Dazu wird dem Kernel zusätzlich der Parameter `sample_pos` übergeben, der den Shift, der beim Sampling verwendet wird, auf eine Verschiebung von der Null-Position des Vektors umrechnet:

Listing 3.7: Faltung des Synthese-Kernels.

```
int data_index = tid_x - result_zero;

for ( int k = -filter_zero; k <= (filter_size - filter_zero - 1) ; k++ )
{
    int index = data_index - k;

    /* Get data */
    /* c = ... */

    out += c * filter[k + filter_zero];
}

result[tid_x] += out;
```

Zu beachten ist an dieser Stelle, dass, anders als im Analyse-Kernel, hier der Index anstatt der Position im Eingabevektor verwendet wird, da dieser die Beschreibung der Angleichung von upgesampleten Eingabevektor an den Ergebnisvektor erleichtert.

Für die Daten `c` wird zunächst überprüft, ob `index` im Indexbereich des upgesampleten Eingabevektors liegt. Falls dies zutrifft wird `index` daraufhin überprüft, ob an dieser Stelle ein Wert aus dem Subband oder eine upgesamplete Null steht (siehe Gleichung 3.25). Falls ja wird `index` auf die entsprechende Position im Eingabevektor umgerechnet, ansonsten wird `c` auf Null gesetzt:

Listing 3.8: Berechnung der Position von `c` im Synthese-Kernel (Teil 1).

```
// if in index range
if ( index >= (-data_zero_upsampled)
    && index < ( data_size_upsampled - data_zero_upsampled ) )
{
    int data_pos = index + data_zero_upsampled;
    if ( data_pos % sampling_factor == sample_pos )
    {
        int pos = (data_pos - sample_pos) / sampling_factor;
        c = data[pos];
    }
    else
    {
        c = 0;
    }
}
//...
```

Falls `index` nicht im Indexbereich liegt, wird abhängig vom gewünschten Verhalten entweder zyklisch oder mit Null fortgesetzt. Bei der zyklischen Fortsetzung wird `index` mittels

modulo auf den entsprechenden Index im Indexbereich gesetzt und es muss zusätzlich überprüft werden, ob dieser nach der Anpassung auf einen Wert des Eingabevektors oder eine upgesamplete Null enthält:

Listing 3.9: Grenzbehandlung bei Synthese-Faltung (Teil 2).

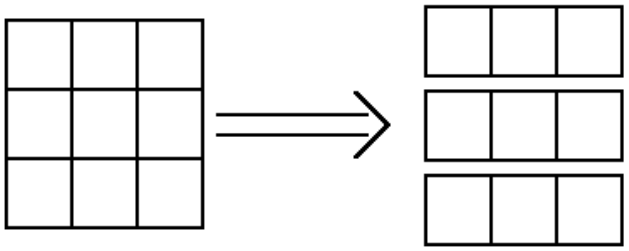
```
//...
else
{
    // recalculate index to index range
    int data_pos = mod(index + data_zero_upsampled, data_size_upsampled);

    // if cyclic continuation
    if ( border_type == CYCLIC
         && data_pos % sampling_factor == sample_pos )
    {
        // calculate position in data vector
        int pos = (data_pos - sample_pos) / sampling_factor;
        c = data[pos];
    }
    // if zero continuation
    else
    {
        c = 0;
    }
}
```

3.3.3 Erweiterung auf zweidimensionale Daten

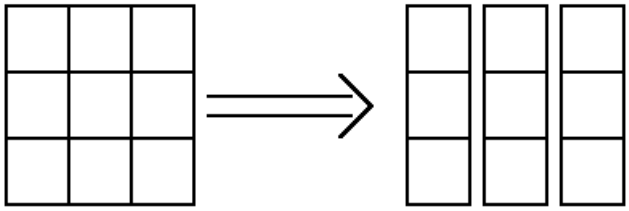
Für die Bearbeitung von zweidimensionalen Daten, zum Beispiel bei der DWT (siehe Abschnitt 2.3) muss man die einzelnen Kernel-Funktionen so erweitern, dass sie die Faltung auch horizontal oder vertikal anwenden können. Das lässt sich recht einfach erreichen, wenn man sich in Erinnerung ruft, dass man die einzelnen Zeilen und Spalten als Vektoren interpretieren kann, die aneinandergereiht wurden.

Da die Daten in Form eines Vektors vorliegen muss man zusätzlich noch eine Verschiebung `data_shift` zum Zugriff auf die Daten verwenden.

Im folgenden wird nun auf die Erweiterung von Analyse- und Synthese-Kernel auf die horizontale (zeilenweise) und vertikale (spaltenweise) Faltung von zweidimensionalen Daten eingegangen.



(a) horizontal



(b) vertikal

Abbildung 3.7: Zerlegung von zweidimensionalen Daten in Vektoren.

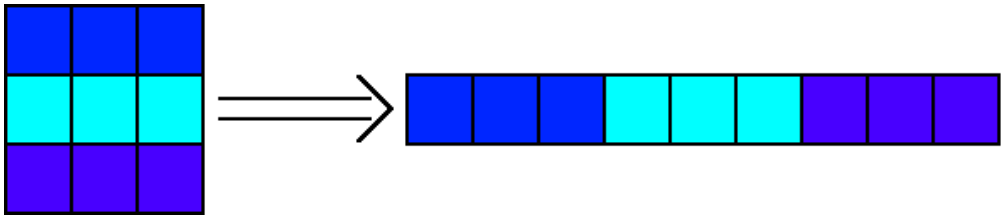


Abbildung 3.8: Interne Repräsentation von zweidimensionalen Daten (horizontal).

Analyse

Zusätzlich zu den bisherigen Berechnungen für eindimensionale Daten müssen außerdem die Verschiebungen in Eingabe und Ergebnisvektor berechnet werden. Da man nun ein zweidimensionales Grid verwendet, lässt sich die Datenposition weiterhin über `tid_x` festlegen. Die Verschiebung hingegen wird über die Y-ID `tid_y` berechnet.

Listing 3.10: Verschiebung in Eingabe- und Ergebnisvektor bei zeilenweise Analyse.

```
1 int result_shift = tid_y * result_size_x;
2 int data_shift = tid_y * data_size_x;
```

Für den Datenzugriff werden diese Verschiebungswerte dann zu `pos` addiert um Position in der Datenmatrix zu erhalten.

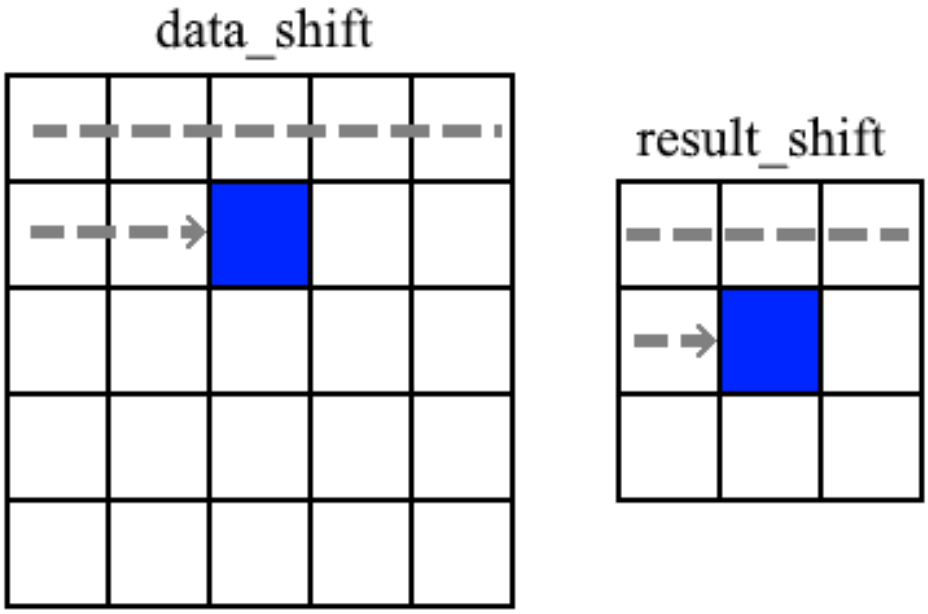


Abbildung 3.9: Verschiebung der Positionen in der Datenmatrix.

Listing 3.11: Zugriff auf Eingabevektor und Ergebnisvektor.

```
1 c = data[pos + data_shift];
2 // calculate convolution
3 result[tid_x + result_shift] = out;
```

Für die vertikale Faltung lassen sich in den meisten Fällen `tid_x` und `tid_y` vertauschen, da man nun statt dem zeilenweisen Zugriff spaltenweise auf die Daten zugreifen will. Da die Spalten keine zusammenhängenden Vektoren sind, muss man hier den Verschiebungswert als Sprung im Eingabevektor betrachten.

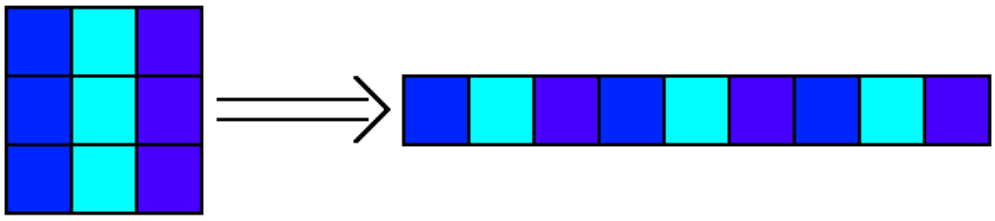


Abbildung 3.10: Interne Repräsentation von zweidimensionalen Daten (vertikal)

Zudem verwendet man statt eines festen Wertes `data_shift` an dieser Stelle `pos`, da man auf aufeinanderfolgende Werte in der Spalte zugreifen will. So erhält man folgendes für

den Zugriff auf den Eingabevektor:

Listing 3.12: Zugriff auf Eingabevektor im vertikalen Analyse-Kernel.

```
1 c = data[tid_x + pos * data_size_x]
```

Synthese

Die Faltung selbst läuft in den zweidimensionalen Synthese-Kernel analog zur Analyse und eindimensionalen Synthese ab. Der einzige wesentliche Unterschied besteht darin, dass, bedingt durch die Art der Implementierung der Filterbank im vertikalen Synthese-Kernel, eine Verschiebung beim Schreiben in den Ergebnisvektor verwendet wird (siehe dazu Abschnitt 3.4.1).

Dies ist aufgrund des Aufbaus der zweidimensionalen Synthese notwendig. Zur Minimierung der Kernelaufrufe werden alle Eingabedaten, die für die horizontale Synthese den gleichen Filter verwenden, zunächst vertikal gefiltert und aufaddiert und das Ergebnis im zweiten Schritt horizontal gefiltert. Der Ablauf ist damit umgekehrt zur Analyse-Filterbank, bei der die Eingabedaten zunächst horizontal gefiltert werden und die einzelnen Ergebnisse danach vertikal um die Subbänder zu erhalten.
Das hat den Vorteil, dass dabei Aufrufe des horizontalen Kernels eingespart werden können, allerdings lassen sich horizontale und vertikale Filterung nicht mehr vertauschen.

3.4 Programmstruktur

Nach der Beschreibung der Kernel-Funktionen werden als nächstes die einzelnen Komponenten der Implementierung kurz vorgestellt und auf den Ablauf einer Filterbank-Ausführung eingegangen.

3.4.1 Komponenten

Die Implementierung besteht aus Klassen für die Filter, die Filterbank sowie für die Kaskade, wobei jede Klasse sich in eindimensional und zweidimensional aufteilt.

SamplingFilter

Die Klassen `SamplingFilter1D` und `SamplingFilter2D` sind für das Initialisieren und den Aufruf der Kernel-Funktionen zuständig. Beide Klassen erben dabei von einer abstrakten Oberklasse `abstract_Filter`, welche die Funktionen zum Einlesen des OpenCL-Quellcodes, zum Kompilieren des selbigen sowie zur Überprüfung auf Kompilierfehler enthält.

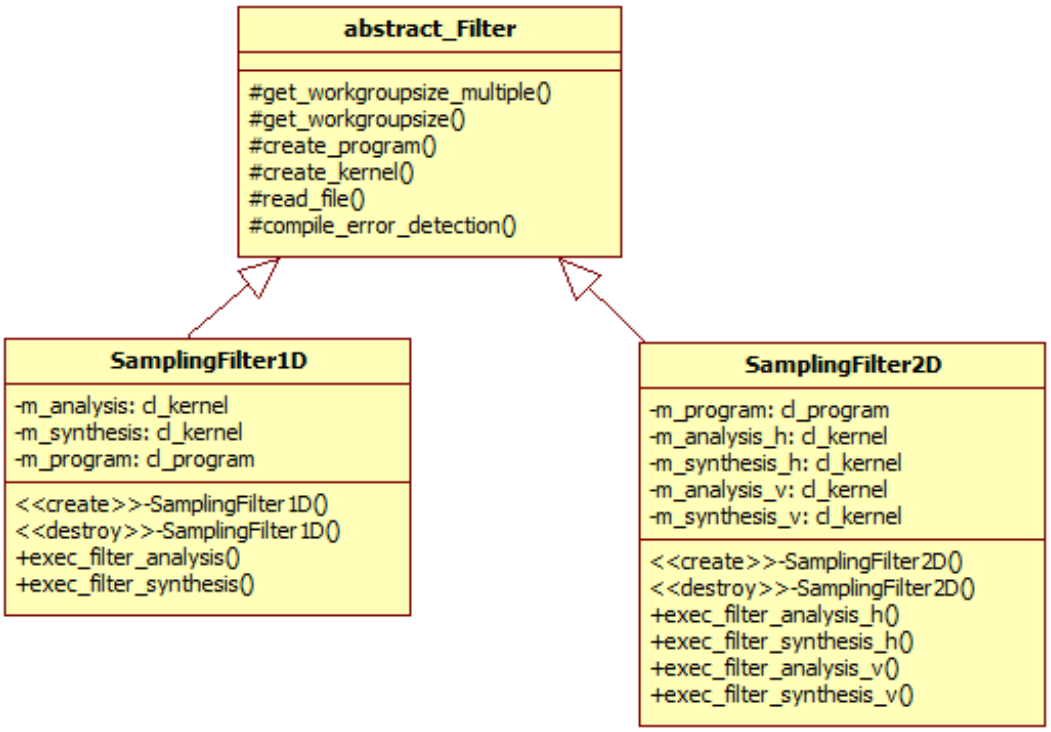


Abbildung 3.11: Klassendiagramm der Filterklassen. Beide Filter erben die Grundfunktionen zum Einlesen und Kompilieren des Kernel-Quellcodes von der abstrakten Oberklasse `abstract_filter`.

Bei Erstellung eines Filterobjekts wird zunächst der OpenCL-Code eingelesen und aus diesem mit `clCreateProgramWithSource()` ein OpenCL-Programm für den gegebenen Context und Device erstellt, welches dann mit `clBuildProgram()` kompiliert wird. Mit diesem Programm werden anschließend die Kernels erstellt (`clCreateKernel()`).

Wird nun eine Kernel-Funktion aufgerufen, setzt der Filter mit `clSetKernelArg()` die Parameter und erstellt anhand der Größe des Ergebnisvektors mit folgendem Code das

ausführende Grid auf dem Device:

Listing 3.13: Festlegung der Grid-Größe in `SamplingFilter2D`.

```
#define BLOCK_X 32 // oder variabel berechnet anhand der warp size
#define BLOCK_Y 32 // oder variabel berechnet anhand der warp size

// size of a work group
size_t block_x = BLOCK_X;
size_t block_y = BLOCK_Y; // = 1 in SamplingFilter1D

// grid size
size_t grid_x = (result.size_x + BLOCK_X - 1) / BLOCK_X ;
size_t grid_y = (result.size_y + BLOCK_Y - 1) / BLOCK_Y;  // = 1 in SamplingFilter1D

size_t gridDim[2] = {grid_x * block_x, grid_y * block_y};
size_t blockDim[2] = {block_x,block_y};
```

Filterbank

Die Klassen `Filterbank1D` und `Filterbank2D` stellen die eigentliche Filterbank, wie sie in Abschnitt 2.2 beschrieben wird, dar. Der Konstruktor dieser Klassen erhält als Parameter die erstellten Context und Device sowie eine Anzahl an Filtervektoren und optional ein bereits erstelltes `SamplingFilterXD`-Objekt mit den benötigten Kernel-Funktionen. Falls dieses nicht angegeben wird, erstellt der Konstruktor der Filterbank ein `SamplingFilter-`

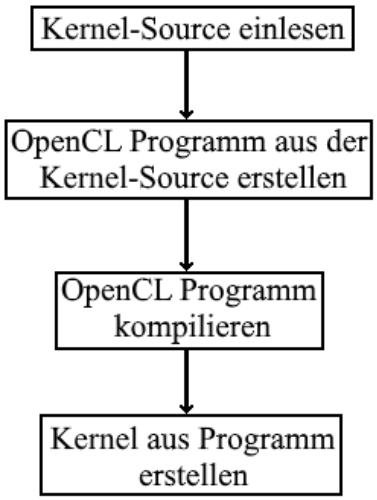


Abbildung 3.12: Ablauf zur Erstellung der Filterkernel aus der Source.

Objekt, welches in diesem Fall am Ende der Lebenszeit des Filterbank-Objekts zerstört wird.

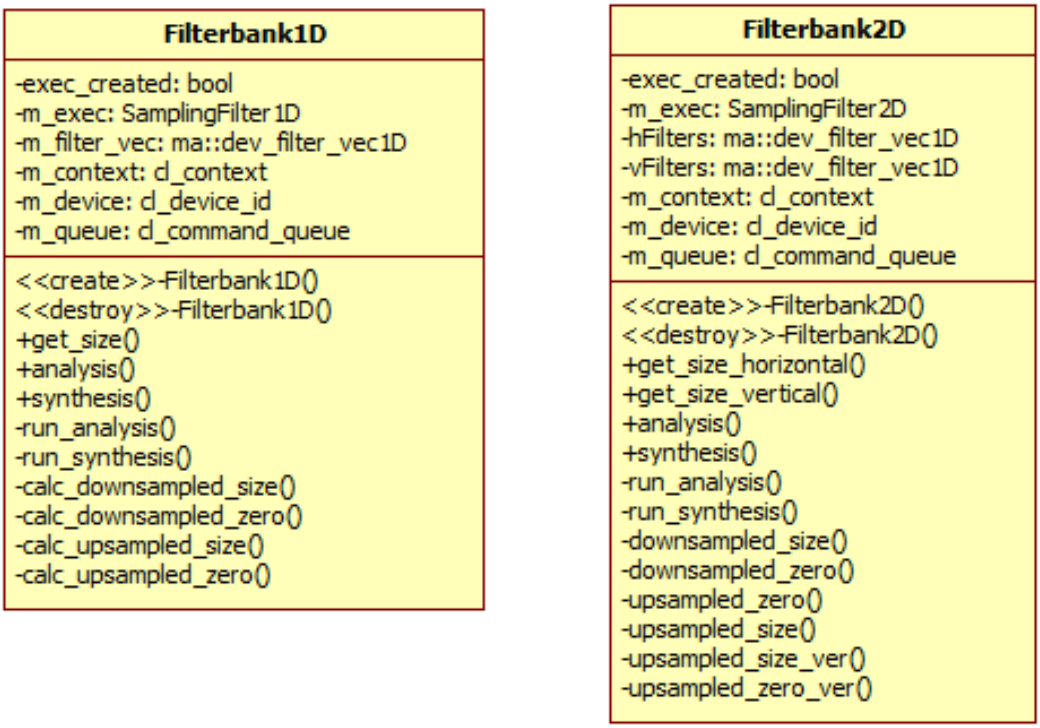


Abbildung 3.13: Klassendiagramm der Filterbankklassen.

Für jedes Filterbank-Objekt werden die Filter fest vorgegeben und können für die gesamte Lebenszeit nicht geändert werden. Übergeben werden die Filter als Vektoren, die während der Ausführung des Konstruktors in den Global Memory kopiert und in der Filterbank selbst als Vektor von Memory Objekten gespeichert werden. Bei Aufruf von Analyse- oder Synthese-Funktion des `SamplingFilter`-Objekts wird dann der gewünschte Filter als Parameter übergeben. Jedes Filterbank-Objekt enthält damit genau ein Set an Filtern, die sowohl für Analyse als auch für Synthese verwendet werden. Die Filterbank selbst enthält nur Funktionen zur Berechnung der Größe der jeweiligen Ergebnisvektoren nach dem Upsampling bzw. Downsampling sowie die Logik für den Ablauf der Analyse bzw. Synthese.

Die Reihenfolge der Faltung bei der Analyse in `Filterbank2D` ist so festgelegt, dass die Eingabedaten zuerst horizontal und danach vertikal gefaltet werden, sodass die Ausgabe-

daten für N horizontale und M vertikale Filter die Reihenfolge $((hor0, ver0), (hor0, ver1), \cdots, (horN, ver(M-1)), (horN, verM))$ haben.

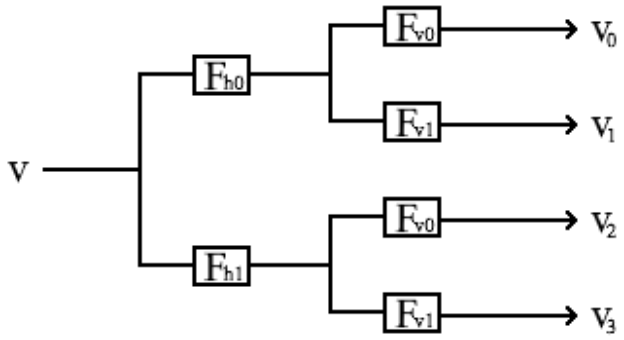


Abbildung 3.14: Schema der Analyse der zweidimensionalen Filterbank.

Will man diese Daten mit der Synthese wieder zusammensetzen, so muss die Reihenfolge erhalten bleiben, da die Rekonstruktion andernfalls keine perfekte Rekonstruktion (bei geeigneten Filtern, siehe Abschnitt 2.2.3) liefern kann. Sind bei der Synthese nicht genug Eingabevektoren vorhanden um alle Filter einer Filterbank zu verwenden, werden die fehlenden Elemente durch *Nullelemente* ersetzt, welche die Länge 1 und den Nullpunkt 0 besitzen und als Wert nur 0 beinhalten. Diese Elemente können sich zwar auf die Länge und Nullpunkt des Ergebnisses auswirken, werden allerdings nicht in der Filterbank selbst ausgeführt, da sie keinen Einfluss auf die Werte des Ergebnisses haben.

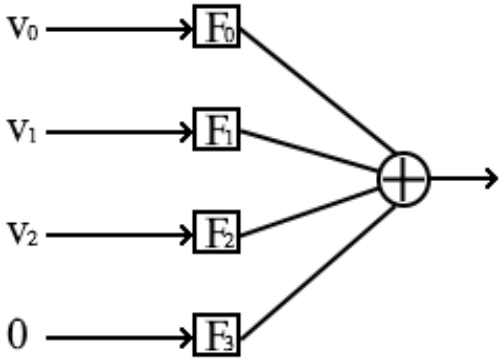


Abbildung 3.15: Sind zur Ausführung einer Filterbank nicht genug Daten vorhanden wird ein zusätzliches Nullelement mit Länge 1 und Nullpunkt 0 dem Eingabevektor mit den Subbändern hinzugefügt.

Bei der zweidimensionalen Synthese werden zunächst alle Eingabevektoren, auf die der

gleiche horizontale Filter angewendet werden soll, vertikal gefiltert und zusammenaddiert. Auf das Zwischenergebnis wird dann der horizontale Filter angewendet und das Ergebnis auf den Ergebnisvektor addiert.

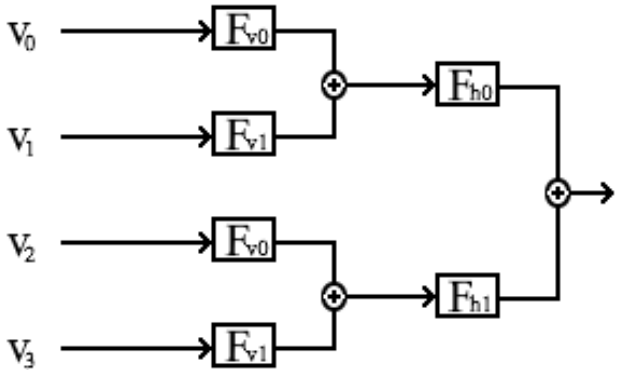


Abbildung 3.16: Schema der Synthese der zweidimensionalen Filterbank.

Cascade

Die Klassen `CascadeXD` dienen zur Kaskadierung von Filterbänken. Sie teilen sich zusätzlich zu eindimensional und zweidimensional in normale Kaskade `CascadeXD` und erweitere Kaskade `ExtCascadeXD` auf.

Die normale Kaskade erhält als Parameter einen (eindimensional) bzw. zwei (zweidimensional) Vektoren mit den Filtervektoren als Parameter. Zusätzlich kann, wie schon bei den `Filterbank`-Klassen ein `SamplingFilter`-Objekt mit den Kernel-Funktionen übergeben werden. Wird dieses nicht übergeben, so erstellt die Kaskade für die Lebenszeit ein temporäres Objekt und verwendet dieses bei der Erstellung des Filterbank-Objekts.
Ein normales `Cascade`-Objekt enthält nur eine einzelne Filterbank, die bei der Analyse oder Synthese mehrfach hintereinander ausgeführt wird.

Die erweiterte Kaskade ermöglicht es verschiedene Filterbänke zu verwenden. Dazu werden mehrere Vektoren mit Filtervektoren übergeben um die einzelnen Filterbänke zu erstellen. Auch hier lässt sich wieder optional ein `SamplingFilter`-Objekt angeben, das zur Erstellung der Filterbänke verwendet wird. Falls dieses fehlt wird ebenfalls ein temporäres Objekt erstellt und mit diesem die Filterbänke aufgerufen. Der Vorteil hierbei ist, dass die Kernel-Funktionen nicht mehrfach eingelesen und kompiliert werden müssen,

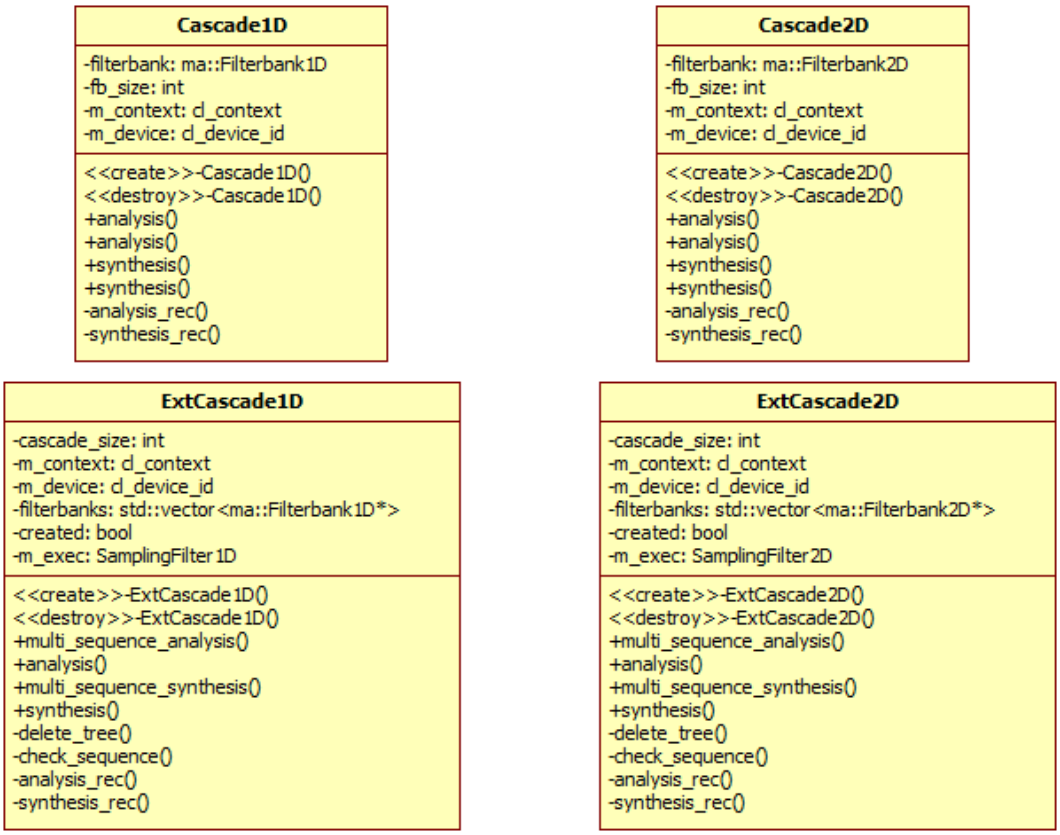


Abbildung 3.17: Klassendiagramm der Kaskadeklassen.

was die Initialisierung der Kaskade beschleunigt.
Bei Aufruf von Analyse oder Synthese wird als zusätzlicher Parameter anstelle der Kaskadentiefe eine Sequenz von Filterbänken angegeben, die auf den Daten ausgeführt werden soll. Die Kaskade wendet dann bei jedem Schritt die entsprechende Filterbank an.

Zusätzlich bietet die erweiterte Kaskade eine Funktion an, die es erlaubt auch mehrere Sequenzen von Filterbänken auf die gleichen Daten anzuwenden. Um Berechnungen zu sparen werden dabei die einzelnen Zwischenergebnisse in einer Baumstruktur gespeichert, sodass die Berechnung einer Filterbank für die gleichen Eingabedaten nicht mehrfach hintereinander ausgeführt wird.

Die Baumstruktur der erweiterten Kaskade ist dabei aus Objekten vom Typ `Node` zusammengesetzt. Jede `Node` im Baum enthält dabei ein Memory Objekt mit dem Ergebnis einer

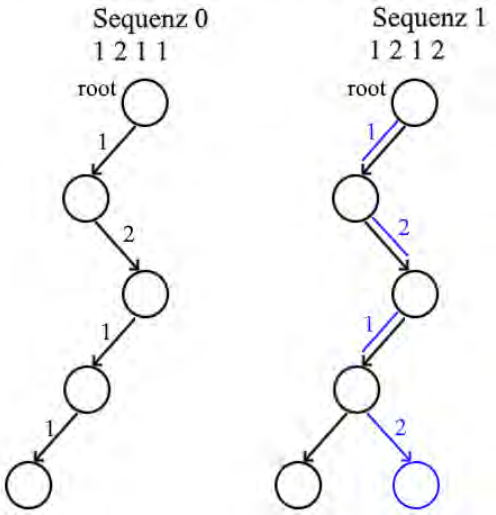


Abbildung 3.18: Bei der Multisequenzkaskadierung werden die einzelnen Sequenzen nacheinander abgearbeitet. Die Ergebnisse werden dabei in einem Baum gespeichert. Gibt es bei den Sequenzen am Anfang Überschneidungen so wird das Ergebnis aus dem Baum verwendet anstatt dieses nochmals neu zu berechnen. Nach Beendigung der Kaskade wird der Baum gelöscht und alle nicht im Ergebnis verwendeten Elemente freigegeben.

Filterbank für eine bestimme Eingabe. Wird dieses Ergebnis an den Nutzer zurückgegeben wird die Node als markiert gesetzt, was dafür sorgt, dass der Speicher des Ergebnisses, welches in dieser Node gespeichert ist, nicht freigegeben wird.
Des weiteren wird ein `boolean`-Array verwendet, das anzeigt, für welche Filterbank der Kaskade für diesen Input bereits ein Ergebnis berechnet und dem Baum hinzugefügt wurde. Die Kind-`Nodes` werden dann in einem Vektor von `Node`-Vektoren gespeichert. Der Index des äußeren Vektor dient zum Zugriff auf das Ergebnis der Filterbank mit gleichem Index, die inneren Vektoren enthalten die einzelnen Memory Objekte, die bei Analyse oder Synthese entstanden sind.

3.4.2 Programmablauf

Nach Vorstellung des Aufbaus der Implementierung wird als nächstes genauer auf den Ablauf einer Filterbank-Ausführung eingegangen.

Initialisierung

Der erste Schritt zur Ausführung der OpenCL-Filterbank ist es, den benötigten Context und das gewünschte Device festzulegen[3]. Dazu wird zunächst die verwendete OpenCL-Plattform ausgewählt:

Listing 3.14: Auswählen der gewünschten OpenCL-Platform.

```
cl_uint num_platforms;
cl_platform_id platforms[10];
clGetPlatformIDs(10, platforms, &num_platforms);
cl_platform_id default_platform = platforms[DEFAULT_PLATFORM];
```

Aus der gewählten Plattform wird im nächsten Schritt das Device ausgewählt:

Listing 3.15: Auswählen des gewünschten OpenCL-Device.

```
cl_uint num_devices;    // Anzahl an gefundenen Devices
cl_device_id devices[10];  //Array zum Speichern der gefundenen Devices
clGetDeviceIDs(default_platform, DEVICE_TYPE, 10, devices, &num_devices);
cl_device_id device = devices[DEFAULT_DEVICE];
```

Zuletzt wird mit dem Device der OpenCL Context festgelegt:

Listing 3.16: Erstellen des OpenCL Context mit dem ausgewählten Device.

```
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
```

Mit `device` und `context` lassen sich nun sämtliche Funktionen verwenden, zum Beispiel das Kopieren auf oder vom Device oder das Erstellen der einzelnen Objekte.

Filterbank-Ablauf

Nachdem die OpenCL-Umgebung initialisiert wurde müssen zunächst die Eingabedaten auf das Device kopiert werden. Die Implementierung bietet dazu den Datentyp `dataXD`, der einen Pointer auf die Eingabedaten enthält, sowie Länge(n) und Nullpunkt(e). Die Daten lassen sich beispielsweise mit der Utility-Funktion `copy_to_device()` als `dev_dataXD` auf das Device kopieren.

3 Für genauere Informationen zur Erstellung einer OpenCL Umgebung siehe beispielsweise [9] oder [15]

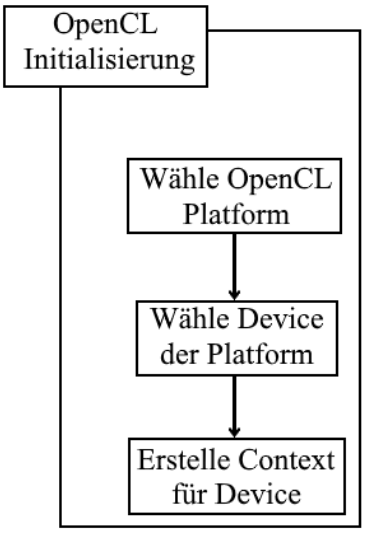


Abbildung 3.19: Ablauf der Initialisierung der OpenCL-Umgebung.

Listing 3.17: Erstellen der Eingabedaten und kopieren auf das Device.

```
1 /* Erstellung der Eimgabedaten auf dem Host */
2 ma::data2D data_in = {input_data, size_x, size_y, zero_x, zero_y};
3
4 /* Kopieren in den Device Memory */
5 ma::dev_data2D dev_in = ma::copy_to_device(context, device, data_in);
```

Zusätzlich muss ein Filterbank-Objekt erstellt werden. Dazu werden zunächst die Filter festgelegt und als Vektor(en) gespeichert. Im nächsten Schritt erstellt man ein Filterbank-Objekt aus `context`, `device` und den Filtervektoren.

Will man mit der erstellten Filterbank eine Analyse der Eingabedaten durchführen, wird dazu die Funktion `analysis()` verwendet. Dieser Funktion übergibt man als Parameter die auf das Device kopierten Daten sowie den gewünschten Sampling-Faktor und einen Wert, der das Grenzverhalten der Faltung festlegt.

Die Filterbank überprüft zunächst die einzelnen Parameter (Länge und Nullpunkt, sowie Sampling-Faktor(en)) und ruft danach eine interne Funktion zur Ausführung der Analyse auf. Für eine eindimensionale Filterbank führt diese in einer Schleife für jeden Filter folgende Aufgaben aus:

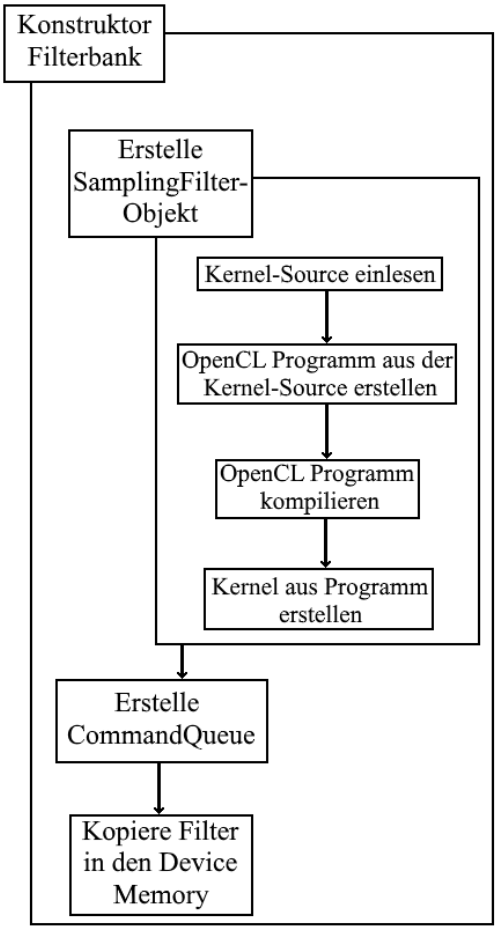


Abbildung 3.20: Ablaufdiagramm Filterbank-Konstruktor.

1. *Berechne Größe und Nullpunkt nach Downsampling*

2. *Alloziere Speicher im Global Memory*
   Anhand der im vorhergehenden Schritt berechneten Größe des Ergebnisses wird nun der Speicher für das Ergebnis alloziert.

3. *Führe Analyse-Kernel aus*
   Der Filter-Kernel wird ausgeführt und faltet die Eingabedaten mit dem in dieser Iteration gewählten Filter. Das Ergebnis wird in das Speicherobjekt geschrieben, das im vorhergehenden Schritt erstellt wurde.

4. *Füge Ergebnis dem Ergebnisvektor hinzu*

Wurde die Schleife für jedes Ergebnis ausgeführt, wird der Ergebnisvektor zurückgegeben.

Dieser enthält dann die einzelnen Subbänder in der Reihenfolge der vorgegebenen Filter.

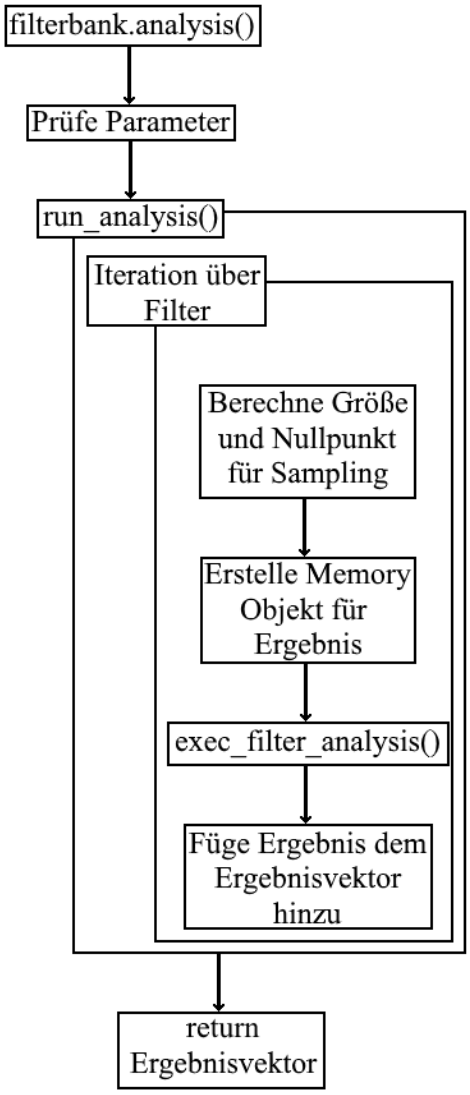


Abbildung 3.21: Ablaufdiagramm eindimensionale Analyse.

Bei der zweidimensionalen Filterbank läuft die Analyse ähnlich ab, es wird allerdings zusätzlich zur horizontalen Filterung noch eine vertikale durchgeführt (siehe Abbildung 3.22). Bei Aufruf der Analyse-Funktion werden auch hier zunächst die Parameter überprüft und dann die eigentliche Analyse-Funktion aufgerufen. Diese läuft nun in zwei verschachtelten Schleifen jeweils für die horizontalen und vertikalen Filter ab. Die äußere Schleife legt dabei den horizontalen Filter fest:

1. *Berechne Größe und Nullpunkt für Ergebnis des horizontalen Downsampling und Faltung*
   Zunächst werden die Eingabedaten mit den horizontalen Filter gefaltet. Dabei wird berechnet wie groß das Ergebnis nach horizontaler Faltung und Downsampling ist. In diesem Schritt werden horizontale Größe und Nullpunkt des Endergebnisses berechnet, vertikale Größe und Nullpunkt werden von den Eingabedaten übernommen.

2. *Alloziere Speicher im Global Memory für das Zwischenergebnis*
   Es wird ein temporäres Memory Objekt im Global Memory angelegt, in dem das Ergebnis der horizontalen Faltung gespeichert wird.

3. *Führe horizontalen Analyse-Kernel aus*
   Der nächste Schritt ist die horizontale Faltung der Eingabedaten mit dem gegebenen Filter. Das Ergebnis von Faltung und Downsampling wird in dem Memory Objekt gespeichert, das im vorhergehenden Schritt erstellt wurde.

4. *Führe Schleife für vertikale Filter aus*
   Nachdem der horizontale Filter angewendet wurde, wird das temporäre Ergebnis vertikal gefiltert. Dazu wird die innere Schleife über die vertikalen Filter aufgerufen:

   a) *Berechne Größe des Ergebnisses*
      Ähnlich wie bei Schritt 1 der äußeren Schleife werden Größe und Nullpunkt nach dem vertikalen Sampling berechnet. Hier werden nun vertikale Größe und Nullpunkt des Endergebnisses berechnet.

   b) *Alloziere Speicher für das Ergebnis im Global Memory*

   c) *Führe vertikalen Filter aus*
      Als Eingabedaten für den vertikalen Filter wird dazu das Zwischenergebnis des horizontalen Filters verwendet. Das Ergebnis ist dann das Endergebnis für diese Filterkombination.

   d) *Füge Ergebnis zum Ergebnisvektor hinzu*

Zuletzt wird der Vektor mit den Ergebnissen zurückgegeben.

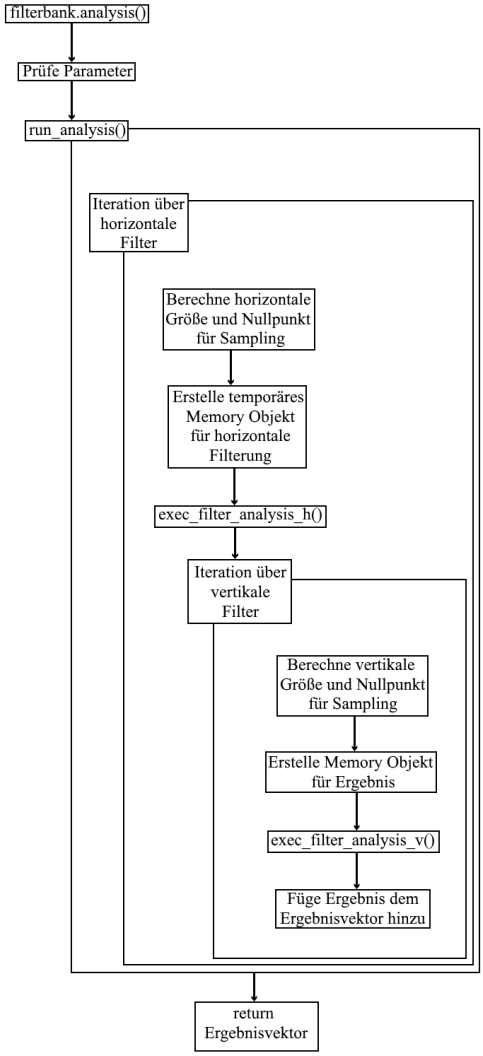


Abbildung 3.22: Ablaufdiagramm zweidimensionale Analyse.

Zum Aufruf der Synthese wird die `synthesis()`-Funktion der Filterbank verwendet. Dieser werden nun ein Vektor mit den Eingabedaten (Subbändern) sowie der /die Sampling-Faktor(en) und die Variable `border` für das Grenzverhalten der Faltung übergeben. Nach Überprüfung der Parameter wird die Synthese ausgeführt.

Im eindimensionalen Fall wird bei der Synthese zunächst die Größe und Nullpunkt des Endergebnisses berechnet und dann ein Memory Objekt im Device Memory erstellt. Zusätzlich muss dieses mit Nullen initialisiert werden, da die einzelnen Subbänder nach Upsampling und Faltung auf dieses Memory Objekt addiert werden. Zuletzt wird der Synthese-Kernel aufgerufen und diesem das jeweilige Subband und der zugehörige Filter

übergeben. Nach Abschluss der Schleifeniteration erfolgt die Rückgabe des Ergebnisses.

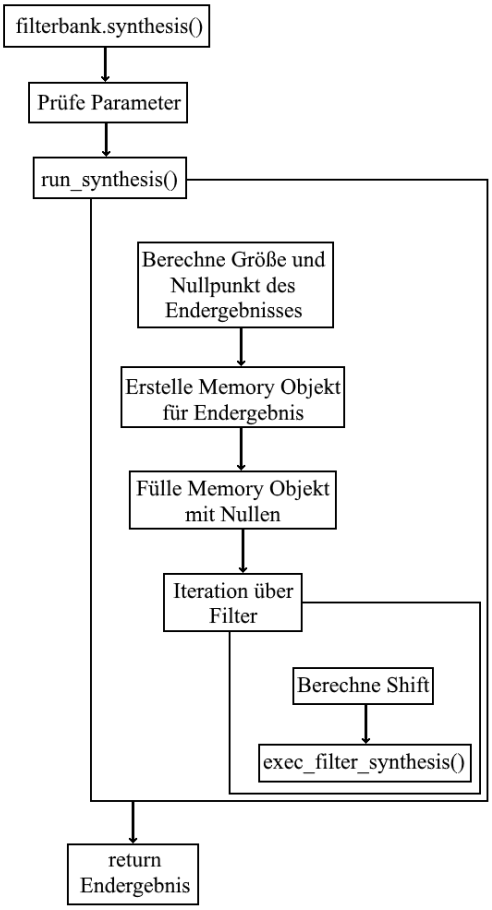


Abbildung 3.23: Ablaufdiagramm eindimensionale Synthese.

Auch bei der zweidimensionalen Synthese werden zunächst die Größen und Nullpunkte des Ergebnisses berechnet und ein Memory Objekt erstellt und mit Nullen gefüllt. Der weitere Verlauf der Synthese wird wieder in zwei verschachtelten Schleifen bearbeitet. Die äußere Schleife iteriert dabei über die horizontalen Filter und führt folgende Aufgaben aus:

1. *Erstelle einen Subvektor aus allen Daten, die den aktuellen Filter als horizontalen Filter verwenden*
   Zunächst werden alle Subbänder aus den Eingabedaten ausgewählt, die bei der horizontalen Filterung den gleichen Filter verwenden würden und zu einem Sub-

vektor zusammengefasst, der alle ausgewählten Subbänder enthält. Die Größe des Subvektors ist normalerweise gleich der Anzahl der vertikalen Filter.

2. *Berechne Größen und Nullpunkte eines temporären Zwischenergebnisses*
   Aus den Subbändern des Subvektors werden die Größen und Nullpunkte eines temporären Zwischenergebnisses berechnet, in dem das Ergebnis der vertikalen Filterung und Addition dieser Subbänder gespeichert wird.

3. *Fülle temporäres Memory Objekt mit Nullen*

4. *Schleife über die vertikalen Filter*
   In der inneren Schleife über die vertikalen Filter werden die Elemente des Subvektors vertikal gefiltert und auf das Zwischenergebnis addiert.

5. *Horizontale Filterung des Zwischenergebnisses*
   Das Zwischenergebnis wird zuletzt horizontal gefiltert und auf das Endergebnis addiert.

Zuletzt wird auch hier das Endergebnis zurückgegeben.

Kaskade-Ablauf

Bei der Ausführung laufen eindimensionale und zweidimensionale Kaskade gleich ab, der Unterschied liegt in den Datentypen und den verwendeten Filterbank-Klassen. Daher werden hier nur die Abläufe von normaler und erweiterter Kaskade vorgestellt.

Bei der Analyse-Funktion der normalen Kaskade werden wieder die einzelnen Parameter auf Validität überprüft und die Analyse dann als rekursive Funktion mit der Kaskadentiefe `depth` als Kontrollvariable, die in jedem Rekursionsschritt um 1 dekrementiert wird, ausgeführt. Dabei wird zuerst der Input mit einer Filterbank-Analyse zerlegt. Im nächsten Schritt wird geprüft, ob die maximale Kaskadentiefe bereits erreicht (also ob `depth == 0`) und in diesem Fall der Vektor mit den Subbändern des aktuellen Rekursionsschritts zurückgegeben. Falls die maximale Tiefe noch nicht erreicht wurde wird die Rekursionsfunktion erneut aufgerufen, wobei als Input das Subband an Position 0 im Vektor verwendet wird und `depth-1` als Kontrollvariable.

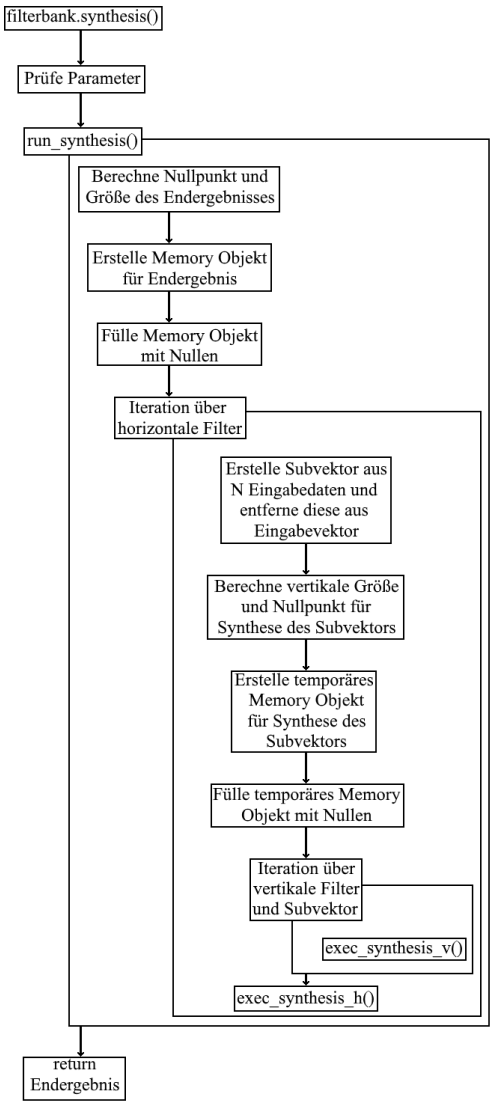


Abbildung 3.24: Ablaufdiagramm zweidimensionale Synthese.

Im nächsten Schritt werden dann die Ergebnisse zu einem Vektor bestehend aus den Ergebnissen des nächsten Rekursionsschritts und dem Ergebnis des aktuellen Schrittes ohne das Element an Position 0 zusammengefasst und zurückgegeben.

Bei der Rekursionsfunktion der Synthese wird zunächst überprüft, ob die Kontrollvariable `depth` gleich Null ist, also die maximale vorgegebene Tiefe bereits erreicht ist. Falls dies nicht der Fall ist wird aus dem Vektor der Eingabedaten ein Subvektor erstellt, dessen Größe kleiner oder gleich der Größe der Filterbank der Kaskade ist. Dieser Subvektor wird als Input für die Filterbank verwendet, das Ergebnis zusammen mit den nicht verwendeten

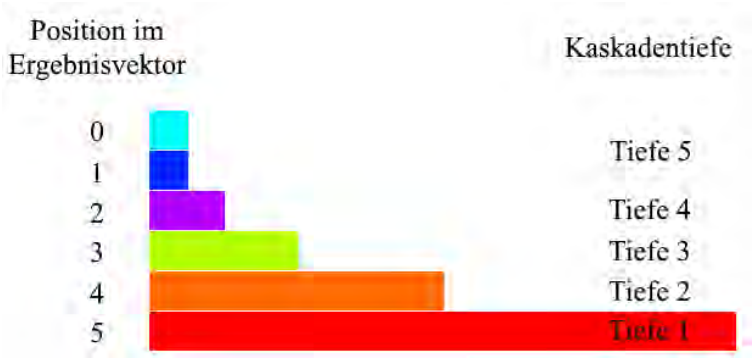


Abbildung 3.25: Pyramidenschema des Kaskadenergebnisses und Position der Subbänder im Ergebnisvektor der Kaskaden-Analyse.

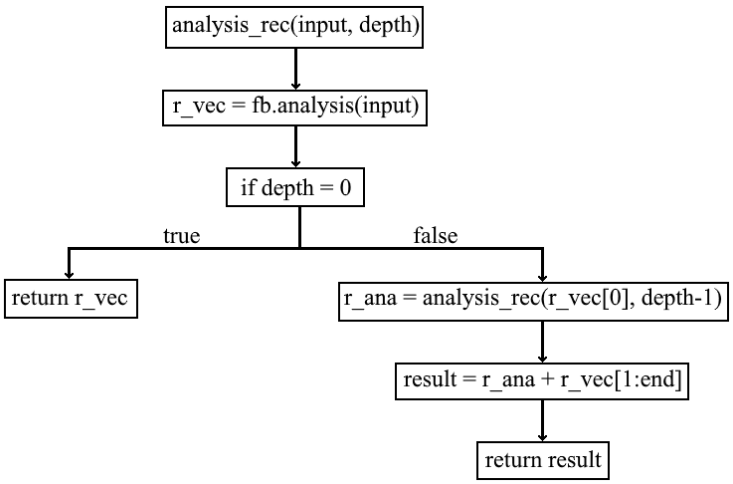


Abbildung 3.26: Ablaufdiagramm Rekursionsfunktion der Analysis-Kaskade.

Daten als neuer Vektor zusammengefasst und als Input für den nächsten Rekursionsschritt verwendet.
Falls die maximale Tiefe erreicht wurde, wird stattdessen das Element an der Stelle 0 im Input des Rekursionsschritts zurückgegeben. Dadurch lassen sich Daten auch nur teilweise synthetisieren.

Die Rekursionsfunktionen der erweiterten Kaskade erhalten als Parameter keine Eingabe-Daten sondern `Nodes` der Baumstruktur, die die jeweiligen Eingabe-Daten enthalten. Zusätzlich wird eine Sequenz der Filterbänke und die aktuelle Position in dieser Sequenz übergeben.
Bei der Analyse-Rekursionsfunktion der erweiterten Kaskade wird zunächst geprüft,

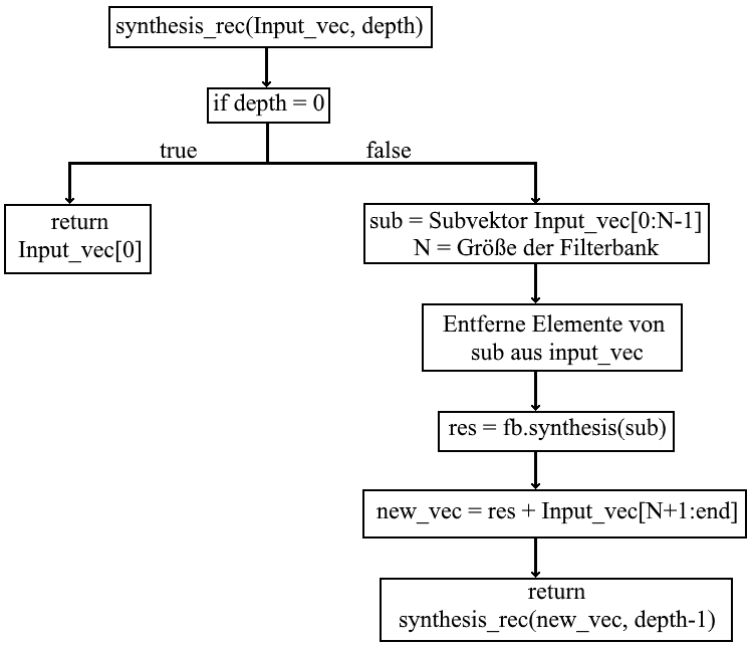


Abbildung 3.27: Ablaufdiagramm Rekursionsfunktion der Synthese-Kaskade.

ob man die letzte Position der angegebenen Sequenz erreicht hat. In diesem Fall wird der Inhalt der aktuellen `Node` zurückgegeben. Ansonsten wird überprüft, ob im Baum ein Ergebnis für diesen Input und diese Filterbank existiert und gegebenenfalls eine Filterbank-Analyse ausgeführt, deren Ergebnisse dann in den Baum eingefügt werden. Im nächsten Schritt wird dann für die Ergebnis-Node an Position 0 im Kind-Node-Vektor die Rekursion fortgesetzt mit der gleichen Sequenz und der Position um 1 inkrementiert und das Ergebnis in einen temporären Vektor gespeichert. Dieser wird zusammen mit den Daten der restlichen Kind-Nodes im Ergebnisvektor gespeichert und dieser zurückgegeben. Das Ergebnis ist ähnlich wie bei der Analyse der normalen Kaskade ein Vektor, der die einzelnen Subbänder enthält.

Die Rekursion der Synthese der erweiterten Kaskade läuft ähnlich zur Synthese ab, mit dem Unterschied, dass man zusätzlich zu einer `Node` den Vektor mit den Subbändern übergibt. Zunächst wird wieder überprüft ob man die letzte Position der Sequenz erreicht hat und gibt in diesem Fall den Inhalt der aktuellen `Node` zurück.
Falls man noch nicht die letzte Position erreicht hat wird aus dem Subband-Vektor ein Subvektor von der Größe der aktuellen Filterbank erstellt und überprüft, ob bereits ein Synthese-Ergebnis für die aktuelle Filterbank vorliegt. Falls nicht wird die Synthese

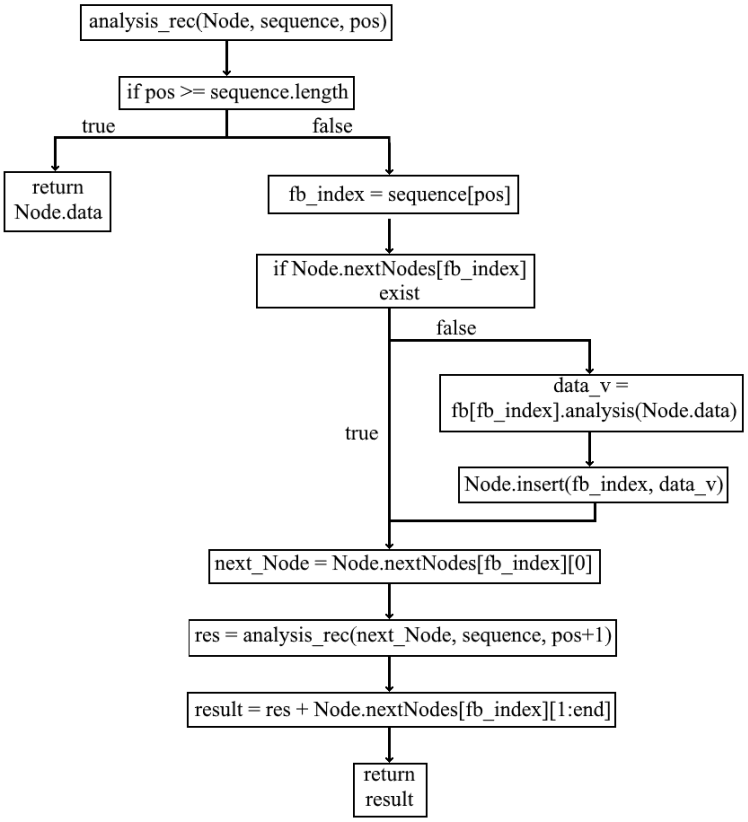


Abbildung 3.28: Ablaufdiagramm Rekursionsfunktion der erweiterten Analysis-Kaskade.

ausgeführt und das Ergebnis dem Baum hinzugefügt. Mit der Synthese des Subvektors und den restlichen Werten des Input wird dann ein neuer Input-Vektor erstellt, der für den nächsten Rekursionsschritt verwendet wird.

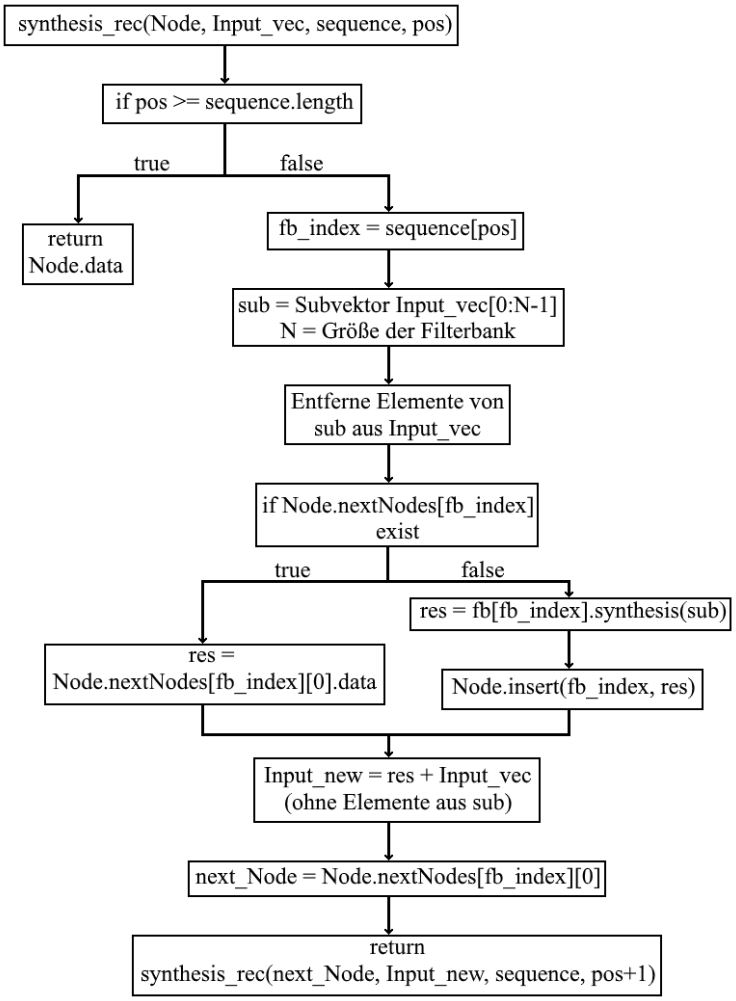


Abbildung 3.29: Ablaufdiagramm Rekursionsfunktion der erweiterten Synthese-Kaskade.

4 Tests

Für Filterbank und Kaskade sind zwei Faktoren wichtig, zum einen die Laufzeit und zum anderen die Qualität des Ergebnisses. Die Laufzeit der Filterbank ist die Zeitspanne, die benötigt wird um ein Signal zu zerlegen bzw. zu synthetisieren. Zusätzlich kommt bei der Verwendung von OpenCL noch die Zeit hinzu, die das Kopieren auf und vom Device benötigt. Der andere Faktor,die Qualität des Ergebnisses, muss deswegen beachtet werden, da man üblicherweise mit Filtern arbeitet, die aus `float` oder `double`-Werten bestehen. Unter Qualität wird hier das Maß der Übereinstimmung von Eingabe und Ausgabe der Filterbank verstanden. Da diese bei der internen Repräsentation beschränkt sind, kann es bei Anwendung der Filter zu Rundungsfehlern kommen, sodass sich ein Signal nicht mehr perfekt zusammensetzen lässt. In diesem Kapitel wird die Implementierung auf diese beiden Faktoren getestet.

4.1 Aufbau Testumgebung

Jeder Test wird mit drei Konfigurationen durchgeführt:

- GPU (GPU)
- CPU unter Verwendung von OpenCL (OpenCL-CPU)
- CPU ohne OpenCL (CPU)

Für die Tests mit GPU und CPU mit OpenCL wird dabei die vorgestellte Implementierung verwendet. Für die CPU ohne OpenCL wird eine angepasste Version des Kernels und des Hostprogramms verwendet. Dabei werden die Kernel-Funktionen als normale C++-Funktionen implementiert, mit dem Unterschied, dass man statt der Thread-Indices `tid_x` und `tid_y` eine Schleife benutzt (Beispiel-Listing 4.1).

Listing 4.1: Verwendung von Schleifen anstelle von Threads.

```
for (unsigned int tid_y = 0; tid_y < result.size_y; tid_y++)
{
    for (unsigned int tid_x = 0; tid_x < result.size_x; tid_x++)
    {
        /* kernel code */
    }
}
```

Als Compiler für den Host-Code wurde der *GCC in Version 5.3.0 des MinGW-64 Projekts* under *Windows 10 Pro 64-bit* verwendet [14]. Kompiliert wird mit dem G++ unter Verwendung der folgenden Parameter:

```
-std=c++14 -static -O2
```

4.1.1 Verwendete Hardware

Das Testsystem besteht aus einer *NVIDIA GeForce GTX 780 Ti* als GPU und einem *Intel Xeon E3 1230v2* als CPU. Das Testsystem ist mit *16GB RAM* ausgestattet. Die CPU wird dabei sowohl für die OpenCL-Implementierung als auch für die Vergleichsimplementierung aus Listing 4.1 verwendet.

4.1.2 Verwendete Tools

Die Berechnung der Laufzeit erfolgt im Code selbst unter Verwendung der `chrono`-Funktionen der C++14-Standardbibliothek *chrono* (Siehe Listing 4.2).

Listing 4.2: Laufzeitmessung mit `chrono`.

```
#include <chrono>

auto start = std::chrono::steady_clock::now();
/* execute analysis or synthesis */
auto stop = std::chrono::steady_clock::now();

auto time_in_ns = chrono::duration_cast<chrono::nanoseconds>(stop - start).count();
```

Die Berechnung der Qualität eines Filterbank- oder Kaskadendurchlaufs (Analyse und Synthese) wird mittels des `compare`-Tools der Softwarebibliothek ImageMagick[4] durchgeführt.

4 https://www.imagemagick.org/

Mit diesem lassen sich zwei Bilder vergleichen und sowohl die Unterschiede visualisieren als auch verschiedene Qualitätsmaße berechnen.

4.1.3 Qualitätsmaße

Bei Verwendung von geeigneten Filtern sollte bei einer Filterbank das Ergebnis identisch zur Eingabe sein. Dies lässt sich auch auf die Kaskade übertragen. Da allerdings bei der internen Repräsentation von Fließkommazahlen Rundungsfehler auftreten können, kann es selbst bei einer perfekt rekonstruierenden Filterbank möglich sein, dass das Ergebnis Abweichungen zur Eingabe aufweist.

Als Maß für die Qualität des Ergebnisses einer Filterbank wird die *Peak signal-to-noise ratio* (PSNR) von Bild und Urbild verwendet. Für die hier verwendeten 8-bit Graustufenbilder ist die PSNR zweier Bilder A und B der Größen $M \times N$ folgendermaßen definiert [6]:

$$PSNR(A,B) = 10 \log_{10} \left(\frac{255^2}{MSE(A,B)} \right) \tag{4.1}$$

mit

$$MSE(A,B) = \frac{1}{MN} \sum_{i=0}^{M} \sum_{j=0}^{N} \left(A_{ij} - B_{ij} \right)^2 \tag{4.2}$$

Je höher dieser Wert, desto geringer sind die Unterschiede zwischen Bild und Urbild. Bei identischen Bildern (ohne Abweichungen) ist die PSNR $+\infty$. Zum Vergleichen der Bilder wird das Ergebnis der Synthese auf die Größe des Eingabebildes zurecht geschnitten, da bei Nullfortsetzung alle Werte außerhalb dieses Bereichs Null sind.

4.1.4 Testdaten und verwendete Filter

Um das Laufzeit-Verhalten der Implementierung nachzuvollziehen werden sowohl für eindimensionale als auch für zweidimensionale Filterbank und Kaskade verschiedene

Datengrößen verwendet. Für die Tests der eindimensionalen Klassen sind das im einzelnen:

- 600000 Elemente (600k)
- 2 Millionen Elemente (2M)
- 4 Millionen Elemente (4M)
- 6 Millionen Elemente (6M)
- 8 Millionen Elemente (8M)

Um später die Qualität der Rekonstruktion zu überprüfen werden diese Werte als Integer-Werte gegeben. In Klammern steht dabei die Bezeichnung, die für diesen Testfall im Text verwendet wird.

Für die zweidimensionalen Tests werden mehrere Bilder verschiedener Größen verwendet:

- 653x871 (harvey)
- 1920x1080 (test_1920)
- 2560x1600 (test_2560)
- 3675x2175 (test_3675)

Die Bilder liegen dabei als Graustufenbilder im Portable Graymap-Format (PGM) vor. Das PGM-Format besteht dabei aus einem Header, der das Format (hier P2 für ASCII-Darstellung der Werte) angibt, die Länge und die Breite des Bildes und den maximalen Wert, den die Pixel erreichen können (Helligkeitswert). Die Pixel werden dann als Integer-Werte zwischen 0 und 255 gespeichert:

```
P2
# Harvey.pgm
653 871
255
37  35  33  32  32  32  ...
...
```

Als Filter wird der 5/3-LeGall-Filter von JPEG2000 verwendet (siehe Abschnitt 2.3.2 und Tabelle A.2). Dieser bietet sich deswegen an, da er mit Festkomma-Zahlen arbeitet und perfekte Rekonstruktion ermöglicht. Das hat zur Folge, dass die Integer-Werte des PGM-Formats nach Analyse und Synthese erhalten bleiben, und Abweichungen mit ziemlicher Sicherheit auf interne Rundungsfehler zurückgeführt werden können.

4.2 Laufzeit und Bildqualität

Die Testläufe werden in drei Teile untergliedert. Im ersten Abschnitt wird die Laufzeit von Filterbank-Analyse und Filterbank-Synthese mit und ohne kopieren auf bzw. vom Device durchgeführt. Wichtig ist an dieser Stelle die Laufzeit der einzelnen Devices bei verschiedenen Datengrößen.

Die zweite Testreihe untersucht die Qualität der Kaskade bei verschiedenen Kaskadentiefen. Da der 5/3-LeGall-Filter verwendet wird, sollte im besten Fall für jede Tiefe eine perfekte Rekonstruktion möglich sein. Aufgrund interner Rundungsfehler kann es allerdings dazu kommen, dass dies nicht möglich ist.

Im letzten Testlauf wird die Verwendung von `double` anstelle von `float` untersucht. Dabei werden Laufzeit und Qualität von Filterbank und Kaskade mit den vorherigen Ergebnissen verglichen. Die Verwendung von `double` sollte dabei für größere Kaskadentiefen bessere Qualität liefern, im Gegenzug dafür aber langsamer sein.

4.2.1 Laufzeit

Für die Laufzeit wurden Analyse- und Synthese-Filterbank zunächst mit den Kopiervorgängen auf und vom Device gemessen und dann ohne. Für jede Messung wurde dabei der Durchschnittswert aus 60 Testläufen verwendet. Zu erwarten ist hierbei, dass GPU und OpenCL-CPU die CPU bzgl. der Laufzeit abhängen, da erstere von der Parallelisierung profitieren sollten.

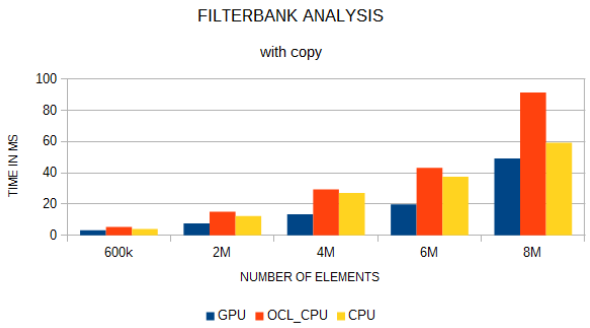
Eindimensional

Abbildung 4.1 zeigt die Laufzeiten für die eindimensionale Analyse-Filterbank in Abhängigkeit der Länge der Eingabedaten mit (Abb. 4.1(a)) und ohne (Abb. 4.1(b)) Kopieren der Daten. Auffällig ist dabei, dass die Laufzeit mit Kopieren für die OpenCL-CPU höher ist als für die normale CPU. Die Laufzeit ohne Kopieren liegt im erwarteten Bereich, da durch die Verwendung von Threads die einzelnen Subbänder schnell generiert werden können. Beim Verwendung der OpenCL-CPU werden die Daten innerhalb des Arbeitsspeichers kopiert, was vergleichsweise ineffizient ist. Die Laufzeit mit Kopiervorgängen der GPU ist im Vergleich zur Laufzeit ohne Kopieren ebenfalls erheblich höher, allerdings kommt hier der Vorteil der GPU beim parallelen Verarbeiten der Daten zum Tragen, sodass die Gesamtlaufzeit hier besser als für CPU und OpenCL-CPU ist.
Anders als bei der Analyse werden bei der Synthese in jedem Durchlauf eine hohe Anzahl an Threads verwendet, sodass sich OpenCL-CPU und GPU von der CPU absetzen können. Dieser Vorteil ist dabei groß genug, dass beide Devices auch bei Miteinbeziehung der Kopierzeiten von und auf das Device durchgehend schneller sind als die CPU.

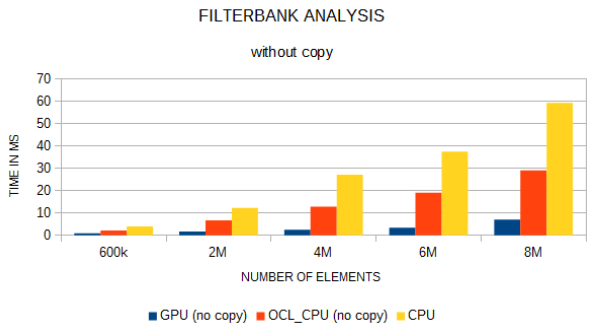
Betrachtet man nur die Laufzeiten der GPU mit und ohne Kopieren, fällt auf, dass mit zunehmender Länge der Daten die Zeit für das Kopieren wesentlich stärker ansteigt als die Zeit für die Berechnung (Abb. 4.2).

Ein ähnliches Bild erhält man bei der eindimensionalen Kaskade. Abbildung 4.3 zeigt dabei die Laufzeiten für die Analyse-Kaskade der Tiefen 2, 5 und 9 mit (Abb. 4.3(a), (c), (e)) und ohne Kopieren der Daten (Abb. 4.3(b),(d),(f)).

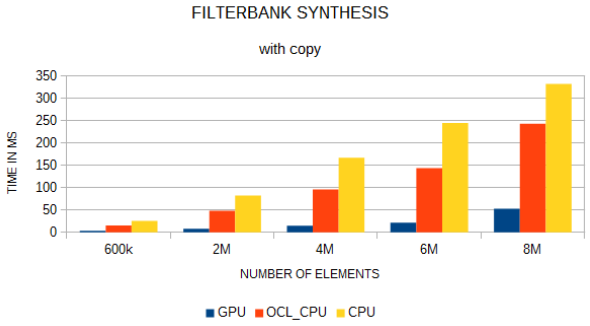
Bei der Analyse-Kaskade sieht man, dass bis zu den Testdaten der Länge 6M die OpenCL-CPU schneller ist als die CPU, für die Länge 8M allerdings die Kopierzeit vom und auf das Device den Vorteil bei der Rechenzeit zunichte macht. Interessant ist dabei, dass die Laufzeit zwischen Tiefe 5 und Tiefe 9 nur leicht ansteigt, die Kopierzeit allerdings stark. Hier kommt vor allem zu tragen, dass viele kleine Subbänder kopiert werden müssen, deren Berechnung schnell erledigt ist, das Kopieren aber vergleichsweise viel Zeit in Anspruch nimmt.
Bei der Synthese-Kaskade (Abb. 4.4) erhält man das gleiche Ergebnis wie schon bei der Synthese-Filterbank. Zwar sind auch hier die Kopierzeiten relativ hoch, allerdings wiegen



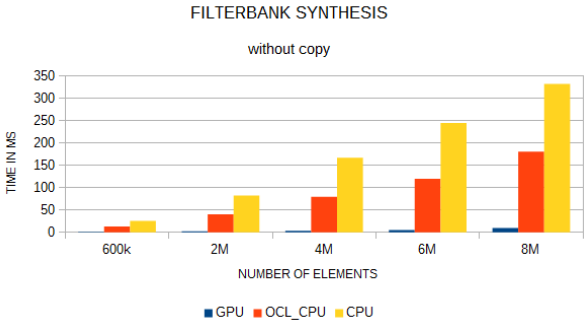
(a) Analyse



(b) Analyse ohne Kopieren

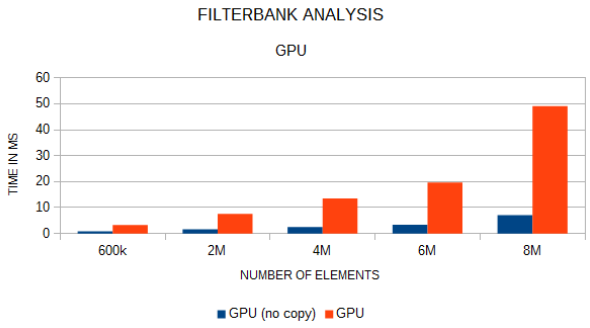


(c) Synthese

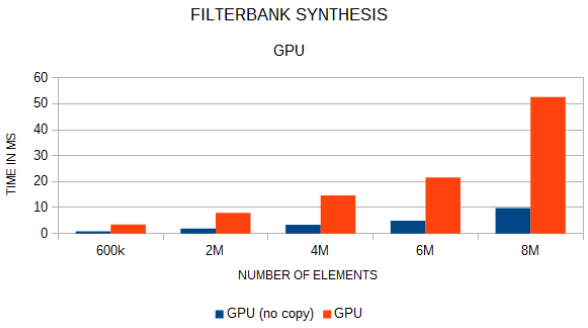


(d) Synthese ohne Kopieren

Abbildung 4.1: Laufzeit für die 1D-Filterbank mit und ohne Kopieren.

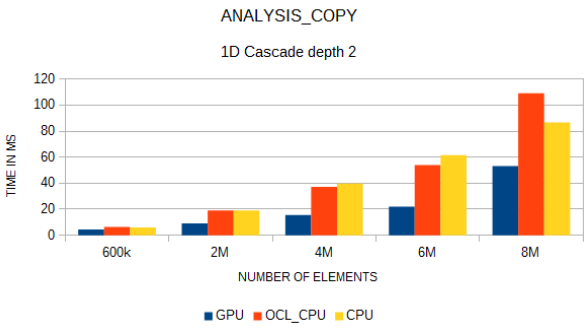


(a) Analyse

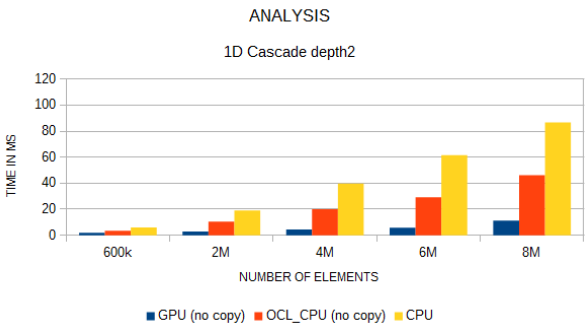


(b) Synthese

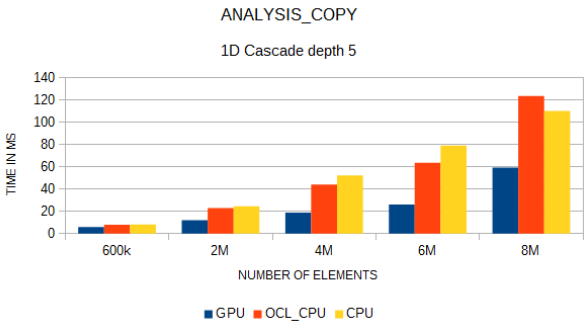
Abbildung 4.2: Laufzeiten 1D-Filterbank (nur GPU).



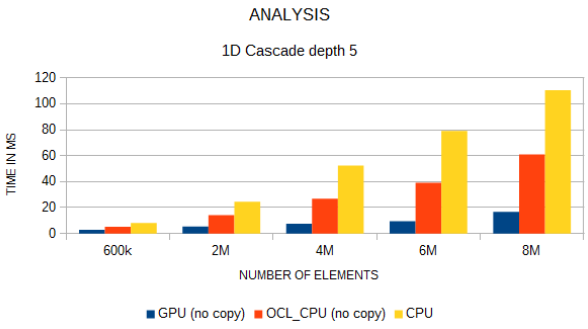
(a) Tiefe 2



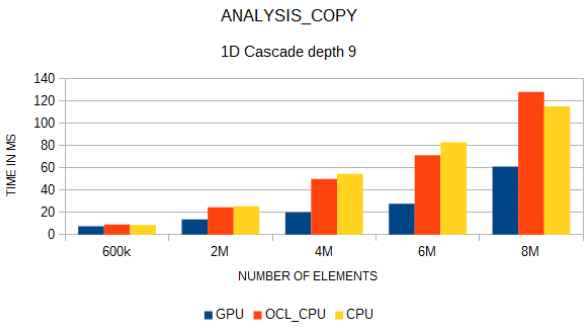
(b) Tiefe 2 (ohne Kopieren)



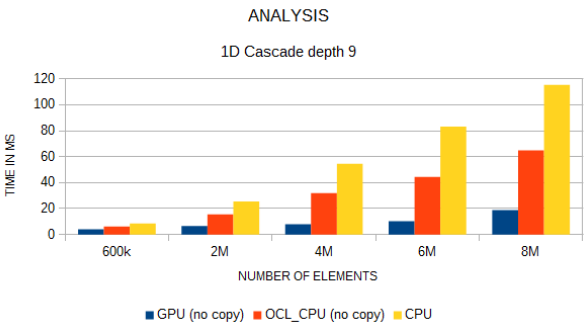
(c) Tiefe 5



(d) Tiefe 5 (ohne Kopieren)



(e) Tiefe 9



(f) Tiefe 9 (ohne Kopieren)

Abbildung 4.3: Laufzeiten der eindimensionalen Analyse-Kaskade für die Kaskadentiefen 2, 5 und 9. Links die Laufzeiten mit Kopieren, Rechts ohne Kopieren. Bei Verwendung der normalen CPU sind keine Kopieroperationen notwendig.

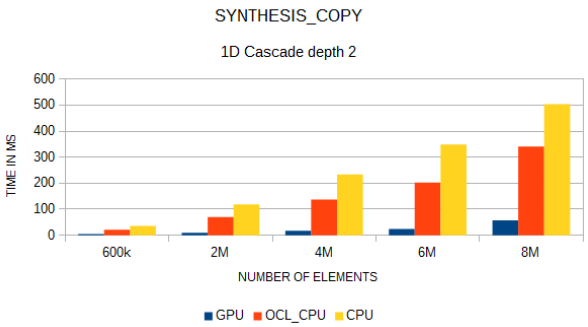
hier die Vorteile durch die Parallelisierung stärker, so dass die Gesamtlaufzeit für GPU und OpenCL-CPU kürzer ist als bei Verwendung der CPU.

Zweidimensional

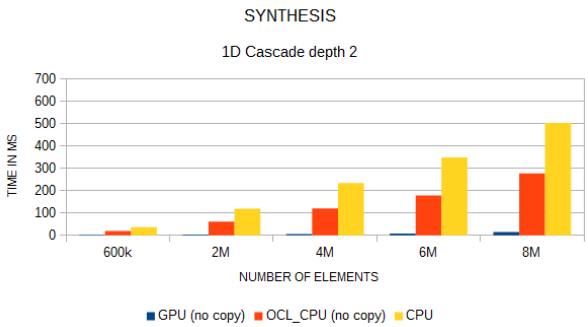
Die Ergebnisse der Tests der zweidimensionalen Synthese-Filterbank (Abb. 4.5(c) und 4.5(d)) sind zu großen Teilen mit denen der eindimensionalen vergleichbar. Auch hier zeigt sich wieder, dass die parallelisierte Implementierung auf GPU und OpenCL-CPU wesentlich schneller ist als die Vergleichsimplementierung ohne Parallelisierung. Mit zunehmender Größe der Daten steigt dabei der Vorsprung von GPU und OpenCL-CPU auf die CPU. Dies hängt auch damit zusammen, dass hierbei die Faltung doppelt durchgeführt wird, einmal horizontal und einmal vertikal für jeden Filter.

Dies sieht man ebenfalls bei der zweidimensionalen Analyse-Filterbank (Abb. 4.5(a) und 4.5(b)). Da die Filterung selbst mehr Rechenleistung benötigt steigt die Laufzeit der CPU stärker an als die Laufzeit der OpenCL-CPU, was zur Folge hat, dass die Kopierzeit anders als bei der eindimensionalen Analyse-Filterbank einen geringeren Einfluss auf die Gesamtausführungszeit hat und damit die OpenCL-CPU in allen Testfällen schneller ist als die CPU.

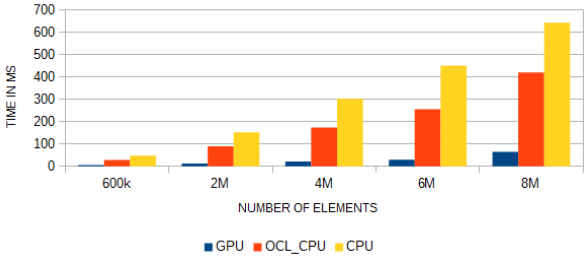
Bei der Analyse-Kaskade ist hingegen auffällig, dass bei zunehmender Tiefe der Kaskade die Laufzeit der GPU stärker steigt als bei OpenCL-CPU oder CPU (Abb. 4.6). Um dieses Verhalten nachzuvollziehen muss man sich die Laufzeiten der einzelnen Kaskadenschritte ansehen. Jeder Kaskadenschritt besteht aus 2 horizontalen und 4 vertikalen Filterungen (vgl. Abb. 3.22), wobei das Ergebnis der ersten horizontalen und vertikalen Filterung als Input für den nächsten Schritt verwendet wird. Wie man anhand der Messungen aus Abb. 4.7 sieht, bremsen in den höheren Kaskadenschritten vor allem die vertikalen Filterungen. Dies könnte damit zusammenhängen, dass bei der vertikalen Filterung die benötigten Daten für die Faltung nicht nebeneinander im Speicher liegen und aus diesem Grund die Speicherzugriffe weniger performant sind als bei der horizontalen Filterung. Die Ursache hierfür liegt vermutlich in der Speicherverwaltung der GPU, die Speicherzugriffe bündeln kann, um sie zu beschleunigen [11]. Dazu kommt, dass bei größerer Kaskadentiefe die Eingabedaten immer kleiner werden, was zur Folge hat, dass die Verwendung der GPU weniger effizient wird, da für vergleichsweise wenige Daten ein Kernel gestartet werden muss.



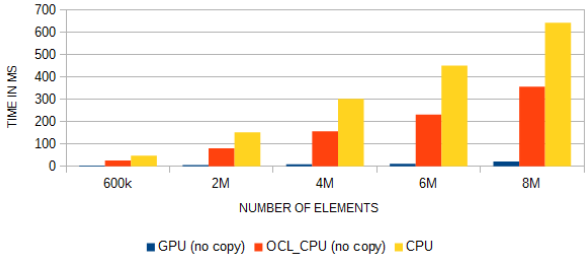
(a) Tiefe 2



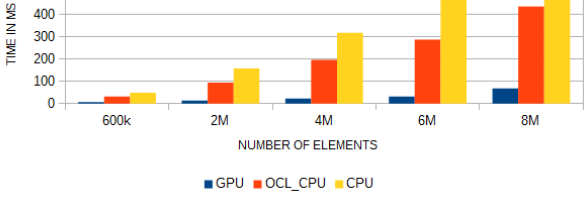
(b) Tiefe 2 (ohne Kopieren)



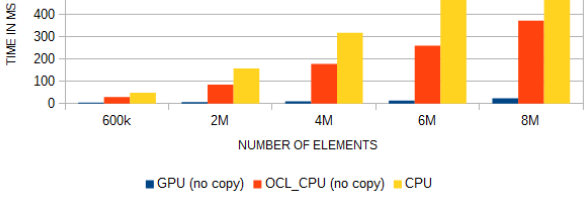
(c) Tiefe 5



(d) Tiefe 5 (ohne Kopieren)

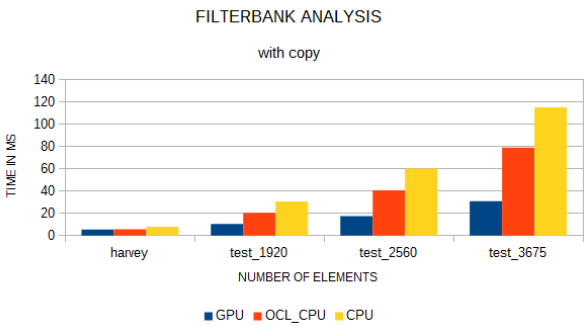


(e) Tiefe 9

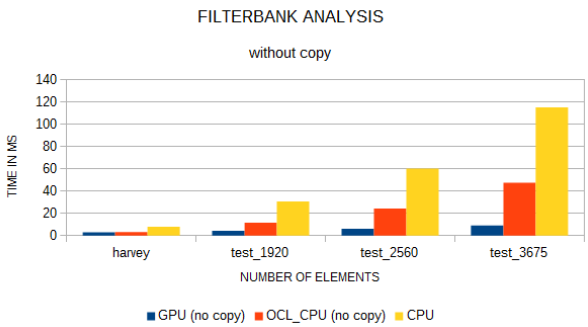


(f) Tiefe 9 (ohne Kopieren)

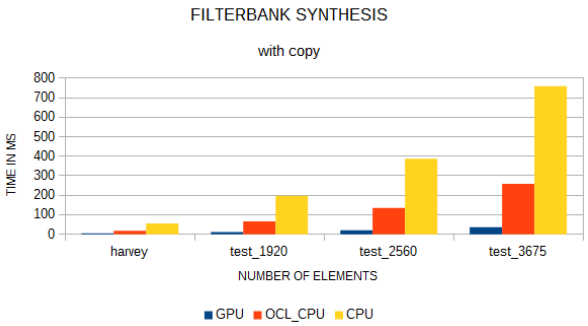
Abbildung 4.4: Laufzeiten der eindimensionalen Synthese-Kaskade für die Kaskadentiefen 2, 5 und 9.



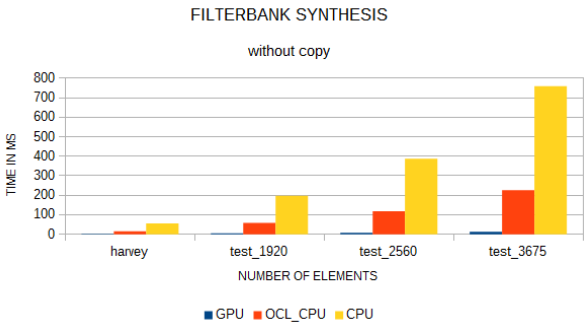
(a) Analyse



(b) Analyse ohne Kopieren

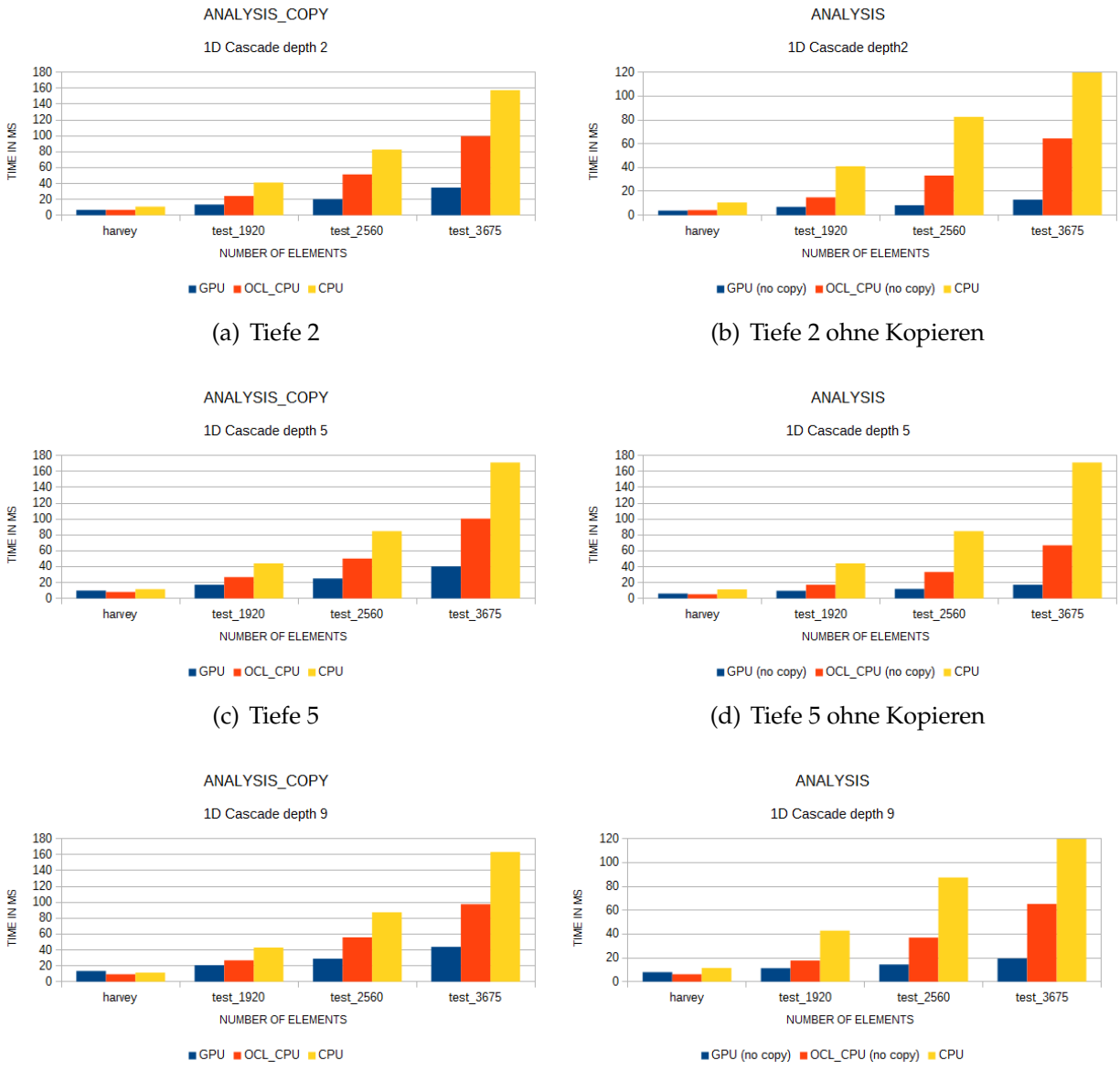


(c) Synthese



(d) Synthese ohne Kopieren

Abbildung 4.5: Laufzeiten 2D-Filterbank mit und ohne Kopieren der Daten.



(a) Tiefe 2

(b) Tiefe 2 ohne Kopieren

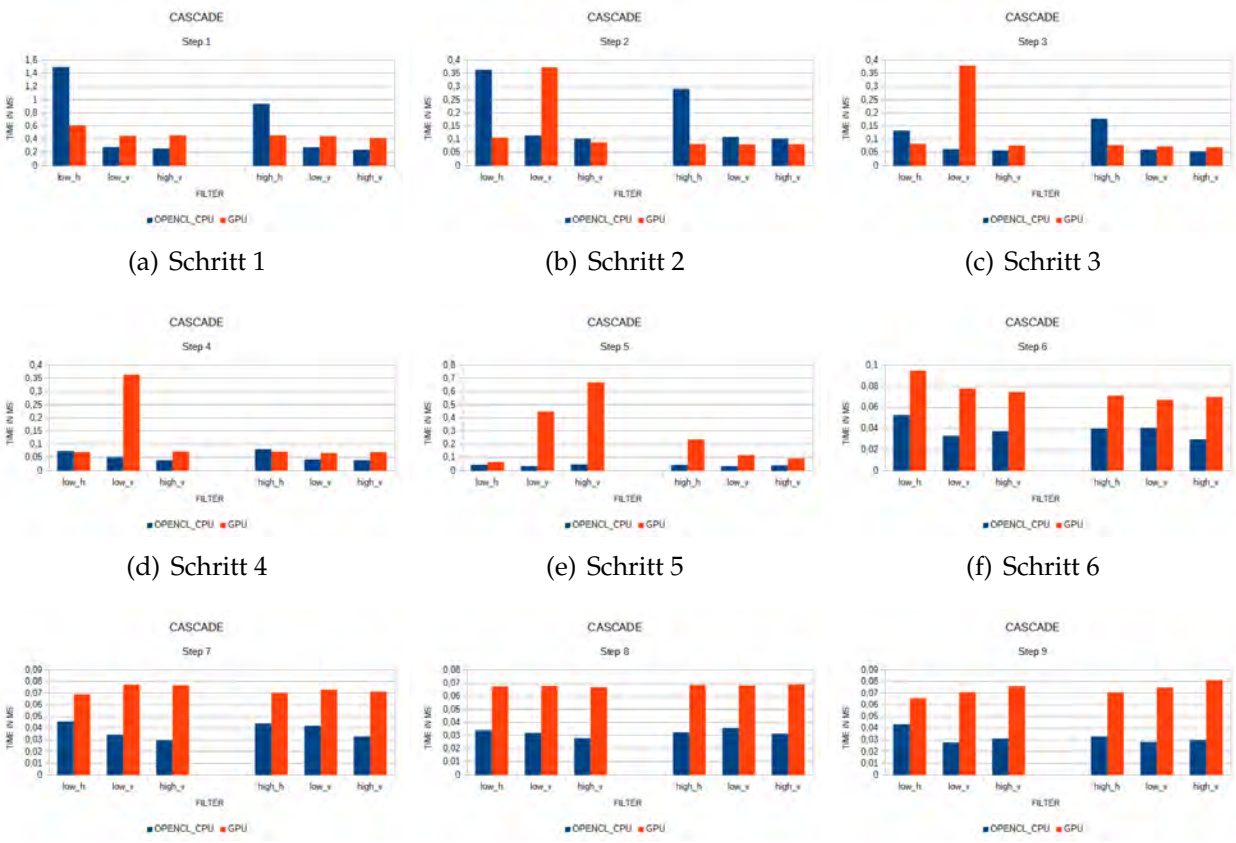
(c) Tiefe 5

(d) Tiefe 5 ohne Kopieren

(e) Tiefe 9

(f) Tiefe 9 ohne Kopieren

Abbildung 4.6: Laufzeiten 2D-Analyse-Kaskade.



(a) Schritt 1 (b) Schritt 2 (c) Schritt 3

(d) Schritt 4 (e) Schritt 5 (f) Schritt 6

(g) Schritt 7 (h) Schritt 8 (i) Schritt 9

Abbildung 4.7: Laufzeiten Kaskadenstufen für Testdaten *harvey*.

Wenn man sich die Laufzeiten für verschiedene Datengrößen ansieht (Abb. 4.8) so erkennt man, dass die GPU bei der 2D-Analyse-Kaskade mit zunehmender Größe der Daten effizienter wird, da mehr Threads zur Berechnung verwendet werden.

Bei der Synthese-Kaskade erhält man wieder das erwartete Ergebnis. Durch die Parallelisierung und die Verwendung einer großen Anzahl an Threads hängen GPU und OpenCL-CPU die CPU ab. Zusätzlich sieht man hier auch, dass bei zunehmender Größe der Daten die Laufzeit der GPU weniger stark zunimmt als die der OpenCL-CPU.
Das Verhalten der Analyse bei kleinen Daten wiederholt sich hier nicht, da die Synthese bedingt durch den Aufbau des Grids eine höhere Anzahl an Threads verwendet, was die GPU besser verarbeiten kann als die OpenCL-CPU.

Zusammenfassung: Laufzeit

Die hier gemachten Messungen legen nahe, dass die Analyse aufgrund der vergleichsweise wenigen verwendeten Threads für die GPU nicht optimal sind, vor allem bei kleinen Datengrößen. Bei der zweidimensionalen Analyse kommt noch dazu, dass die vertikale Faltung nicht von den Optimierungen der GPU-Speicherzugriffe profitieren kann, da die benötigten Daten nicht nebeneinander im Grafikspeicher liegen. Ein weiteres Problem bei der Analyse sind die Kopierzeiten, da man die Daten zunächst auf das Device kopieren muss, um sie dort zu bearbeiten und danach wieder auf den Host. Besonders bei kleinen Daten, wo CPU und OpenCL-Devices ohnehin nahe beieinander liegen, wirkt sich dies auf die Laufzeit aus.
Bei der Synthese hingegen sieht man bei zunehmender Datengröße, dass trotz der Kopierzeiten die OpenCL-Devices die CPU in allen Testfällen abhängen. Hier kommen mehr Threads zum Einsatz, was vor allem der GPU zugute kommt.

4.2.2 Qualität

Da während der Tests das 5/3-LeGall Wavelet verwendet wurde, sollte das Ergebnis eines kompletten Filterbank- oder Kaskadendurchlaufs (Analyse und Synthese) bei beliebigen Kaskadentiefen identisch zur Eingabe sein, wenn man die implementierungsbedingte Erweiterung an den Randbereichen durch Nullen außer Acht lässt. Durch die begrenze Genauigkeit des verwendeten Datentyps `float` und der Verwendung von Festkomma-Werten

beim Filter kann es allerdings vor allem bei größeren Kaskadentiefen zu Rundungsfehlern kommen.

Da die zweidimensionale Kaskade die Faltung zweimal ausführt, ist sie wesentlich anfälliger für solche Rundungsfehler. Betrachtet man die PSNR für das Testbild *harvey* für verschiedene Kaskadentiefen (Abb. 4.10), so sieht man, dass die Kaskade bis Tiefe 2 perfekt rekonstruieren kann. Ab Tiefe 3 nimmt die Qualität langsam ab.

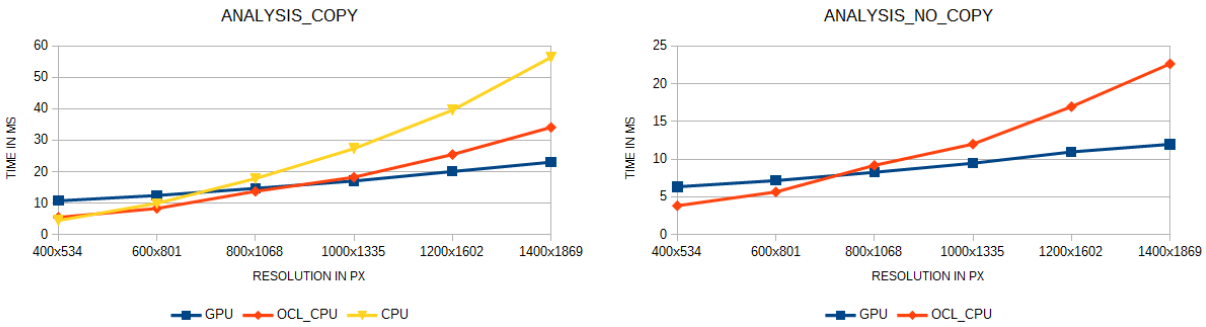
Die Tests der eindimensionalen Kaskade ergeben erst ab Tiefe 11 überhaupt eine Differenz zwischen Eingabe und Ausgabe und werden daher an dieser Stelle nicht weiter beachtet.

4.2.3 Vergleich `float` und `double`

Um den Rundungsfehlern etwas entgegenzuwirken kann man statt dem Datentypen `float` mit einfacher Genauigkeit den Datentyp `double` verwenden, welcher doppelte Genaugigkeit bietet. Der Vorteil hierbei ist, dass Rundungsfehler sich erst in höheren Kaskadentiefen bemerkbar machen, allerdings kann die Ausführung von Berechnungen mit doppelter Genauigkeit, abhängig vom verwendeten Device, etwas langsamer sein [11].

Bei den Laufzeiten der zweidimensionalen Filterbank (Abb. 4.11) sieht man, dass bei Verwendung von `double` sowohl die Zeit steigt, die zur Berechnung benötigt wird, als auch die Kopierzeit, da `double` mit 8 Bit doppelt so groß ist wie `float` mit 4 Bit (auf dem verwendeten Testsystem).

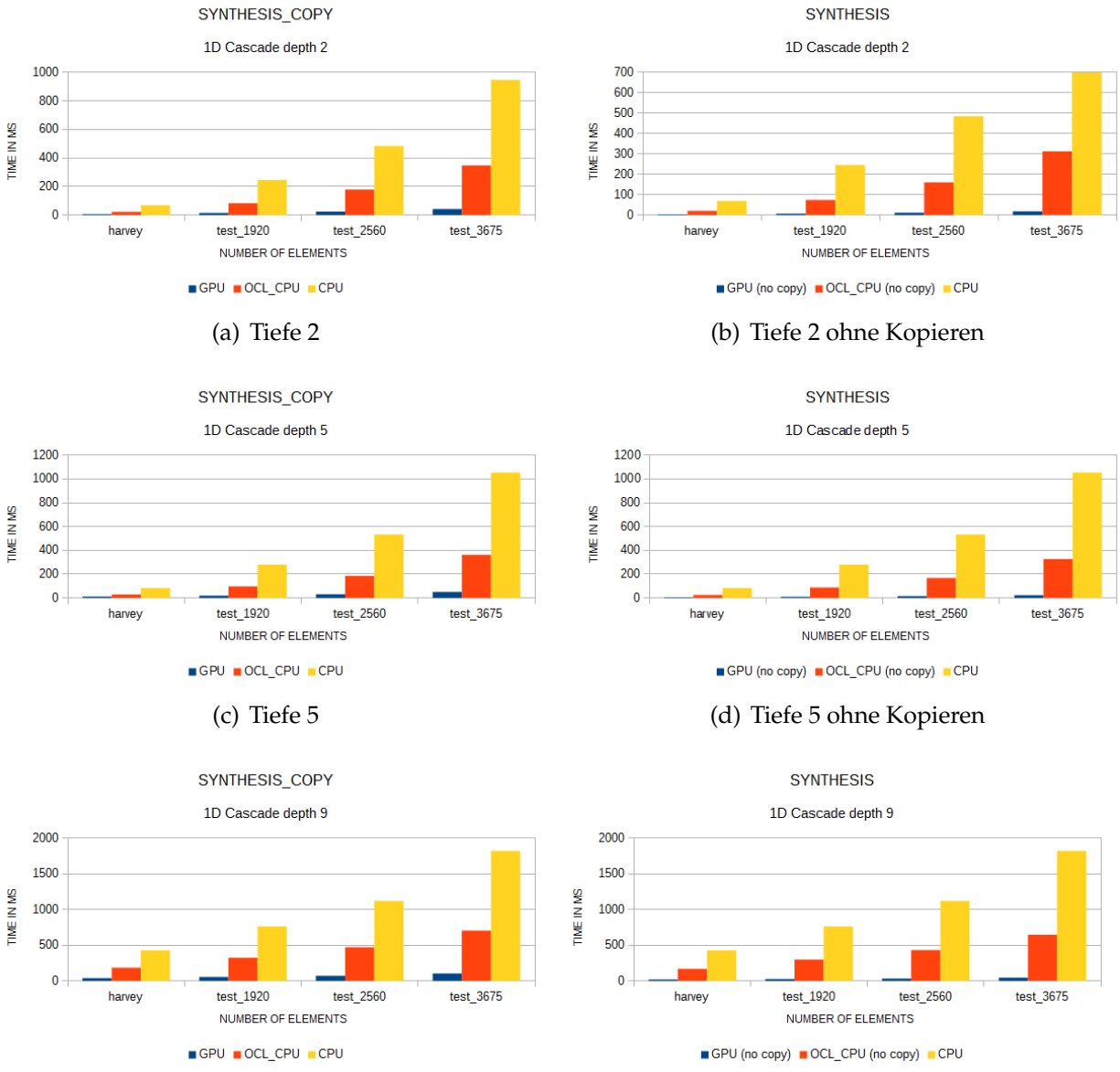
Bei der Bildqualität bietet `double` eine große Verbesserung gegenüber `float`. Für das Testbild *harvey* wird nun bis Kaskadentiefe 7 perfekt rekonstruiert, erst danach fällt die PSNR durch Rundungsfehler ab (Abb. 4.12).



(a) mit Kopieren

(b) ohne Kopieren

Abbildung 4.8: Laufzeiten der Analyse-Kaskade auf der GPU für verschiedene Datengrößen.



(a) Tiefe 2

(b) Tiefe 2 ohne Kopieren

(c) Tiefe 5

(d) Tiefe 5 ohne Kopieren

(e) Tiefe 9

(f) Tiefe 9 ohne Kopieren

Abbildung 4.9: Laufzeiten 2D-Synthese-Kaskade.

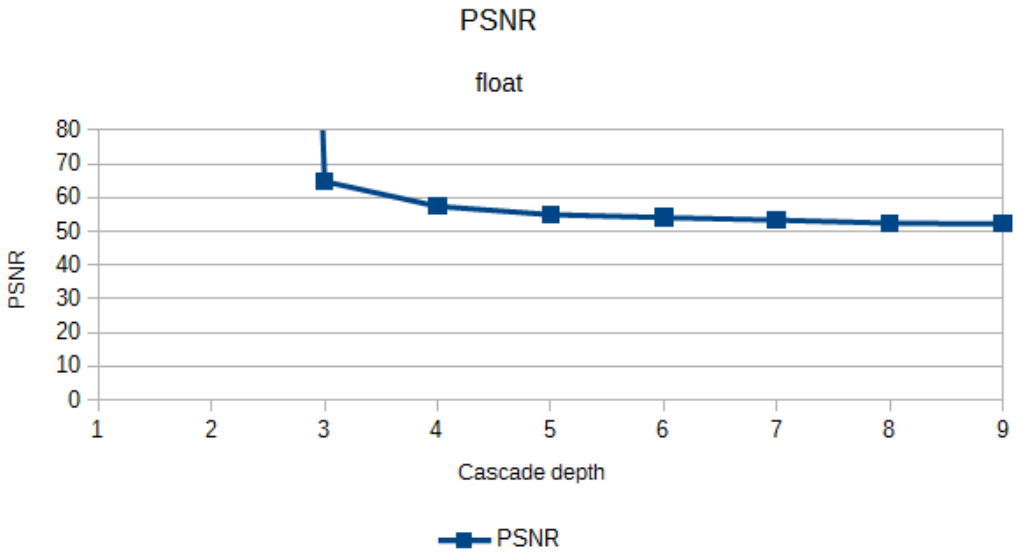
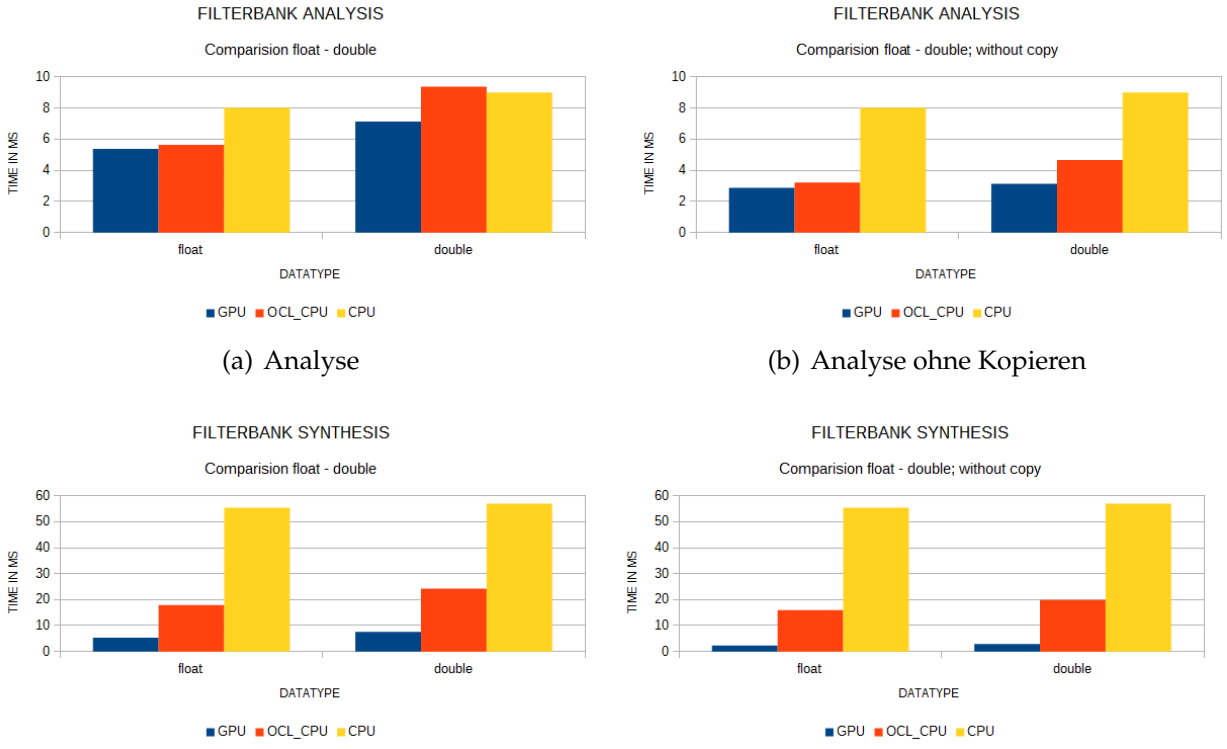


Abbildung 4.10: Verlauf PSNR für verschiedene Kaskadentiefen für Bild *harvey*.



(a) Analyse

(b) Analyse ohne Kopieren

(c) Synthese

(d) Synthese ohne Kopieren

Abbildung 4.11: Laufzeiten 2D-Kaskade für `double` und `float`.

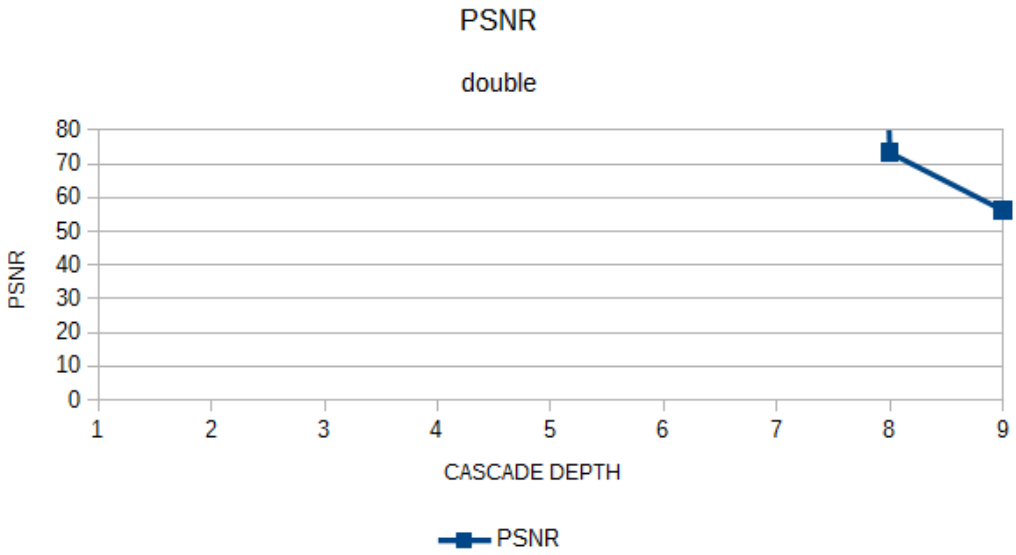


Abbildung 4.12: Verlauf PSNR für verschiedene Kaskadentiefen für Bild *harvey* unter Verwendung von `double` anstelle von `float`. Zu Beachten: Bis einschließlich Kaskadentiefe 7 ist die PSNR ∞, Bild und Ergebnis sind also identisch.

5 Zusammenfassung

5.1 Zusammenfassung

Die Faltung ist ein Paradebeispiel für eine Anwendung, die sich hochgradig parallelisieren lässt, da jede einzelne Berechnung nur von den Eingabedaten abhängt und keine Zwischenergebnisse von vorherigen Berechnungsschritten benötigt werden. Dies ermöglicht es digitale Filter parallelisiert zu implementieren, da diese nichts anderes als die Faltung eines Signals mit einem Filter sind. Diese Filter lassen sich zu Filterbänken zusammenfassen, die mehrere Filter auf ein Signal mit unterschiedlicher Verschiebung anwenden.

Im Rahmen dieser Arbeit wurde eine solche Filterbank mittels OpenCL so implementiert, dass sie sowohl auf der GPU als auch auf der CPU parallelisiert ausgeführt werden kann. Dazu wurden verschiedene Kernel-Funktionen für eindimensionale und zweidimensionale Analyse und Synthese erstellt. Zur Ausführung dieser Kernel-Funktionen wurden mehrere Klassen für 1D- und 2D-Filterbänke sowie für die Kaskadierung derselbigen erstellt und abschließend mit einer nicht-parallelisierten Implementierung auf der CPU verglichen.

Die Testreihen zeigen relativ deutlich, dass die GPU bei Ausführung dieser Implementierung verglichen mit der CPU vor allem bei Daten mit vielen Elementen Vorteile hat, da mehr Elemente parallel abgearbeitet werden können. Zusätzlich wurde deutlich, dass das Kopieren der Daten auf und von der GPU bei der Laufzeit einen großen Einfluss hat. Des weiteren wurde der Einfluss von doppelt-genauen Gleitkommazahlen (`double`) auf die Qualität von Zerlegung und Rekonstruktion im Vergleich zu einfacher Genauigkeit (`float`) überprüft, da vor allem bei tiefer Kaskadierung von Filterbänken Rundungsfehler entstehen können. Das Ergebnis war hier, dass die Verwendung von `double` bei den Testdaten eine Verbesserung darstellt, bei gleichzeitiger Verschlechterung der Laufzeit.

Daher eignet sich die Verwendung von `double` vor allem für Anwendungen bei denen der Qualitätsverlust durch Rundungsfehler möglichst eliminiert werden soll. Für Laufzeitkritische Anwendungen eignet sich `float` wegen der geringeren Laufzeit eher, wobei sich die Laufzeitzunahme bei dem verwendeten Testsystem im Bereich von wenigen Millisekunden befindet.

5.2 Ausblick

Im Rahmen dieser Arbeit wurde die Implementierung nur auf einem einzelnen Testsystem mit einer relativ kleinen Datenmenge getestet, sodass die Aussagen sich zwar auf andere Systeme übertragen lassen sollten, aber abhängig vom Typ und Hersteller der Devices auch Unterschiede bei der Laufzeit auftreten können.

Die Implementierung selbst scheint bei der Synthese sehr stark von der Thread-Kapazität der GPU zu profitieren, hier könnte man das verwendete Grid so überarbeiten, dass nur noch die Anzahl der Threads gestartet wird, die tatsächlich für die Faltung und Addition des Subbands benötigt werden, anstatt wie im Moment für jedes Element des Ergebnisses einen Thread zu starten. Dies sollte dafür sorgen, dass der Abstand zwischen CPU und GPU bei der Verwendung von OpenCL kleiner wird. Außerdem sollte sich dadurch die Laufzeit minimal verbessern, da weniger Threads gestartet werden, die keine Aufgaben ausführen. Eine weitere Optimierung der Kernel und des Host-Codes könnten zu einer verbesserten Laufzeit führen, was durch weitere Tests auf einem größeren Datenset überprüft werden sollte.
Eine Möglichkeit zur Verbesserung der Implementierung wäre beispielsweise, das Rückschreiben in den Speicher bei der horizontalen Faltung so zu vertauschen, dass die vertikale Faltung ebenfalls zeilenweise auf den Speicher zugreifen kann, womit man sich die optimierten Speicherzugriffe der GPU zunutze machen könnte.

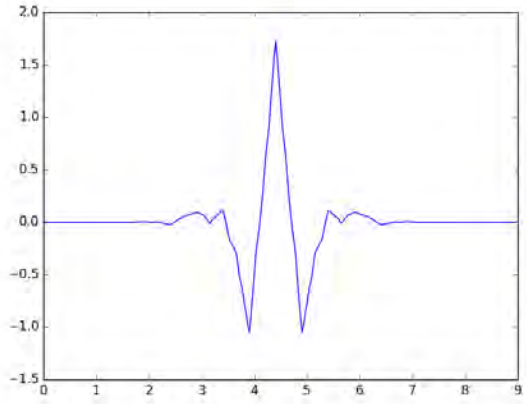
A Anhang

i	Analysefilter		Synthesefilter	
	Tiefpass	Hochpass	Tiefpass	Hochpass
0	0.602949018236	1.115087052457	1.115087052457	0.602949018236
±1	0.266864118443	−0.591271763114	0.591271763114	−0.266864118443
±2	−0.078223266529	−0.057543526228	−0.057543526228	−0.078223266529
±3	−0.016864118443	0.091271763114	−0.091271763114	0.016864118443
±4	0.026748757411			0.026748757411

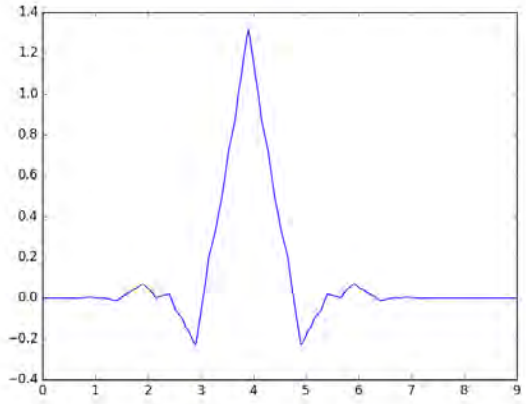
Tabelle A.1: 9/7-Waveletkoeffizienten für Analyse- und Synthesefilter (Quelle: ([17]))

i	Analysefilter		Synthesefilter	
	Tiefpass	Hochpass	Tiefpass	Hochpass
0	0.75	1	1	0.75
±1	0.25	−0.5	0.5	−0.25
±2	−0.125			−0.125

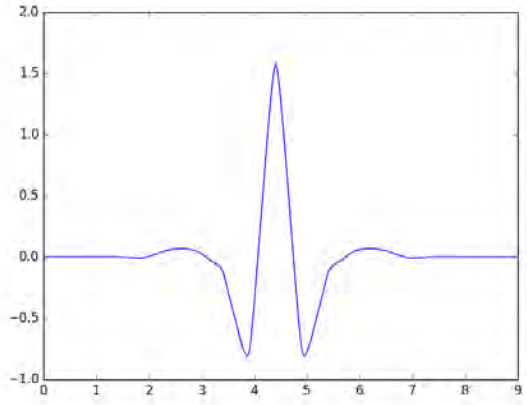
Tabelle A.2: 5/3-Waveletkoeffizienten für Analyse- und Synthesefilter (Quelle: ([22]))



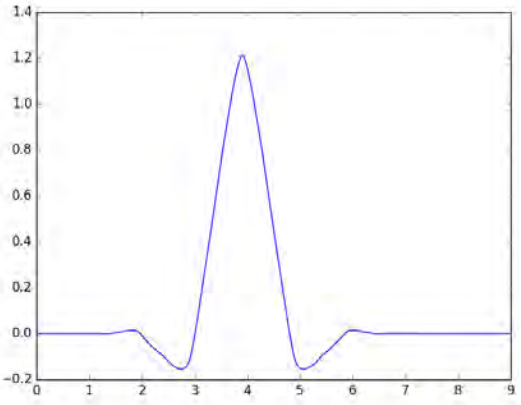
(a) Analyse Tiefpass



(b) Analyse Hochpass

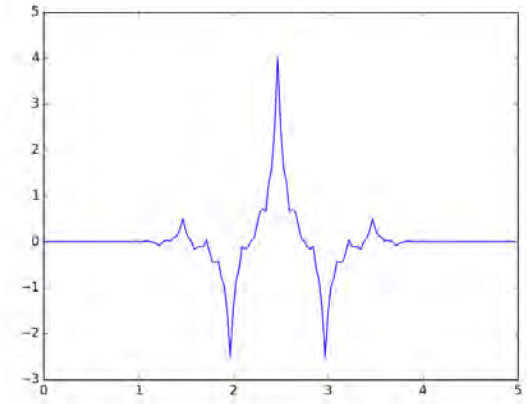


(c) Synthese Tiefpass

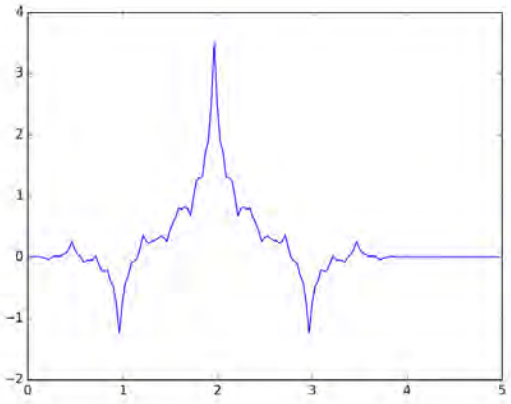


(d) Synthese Hochpass

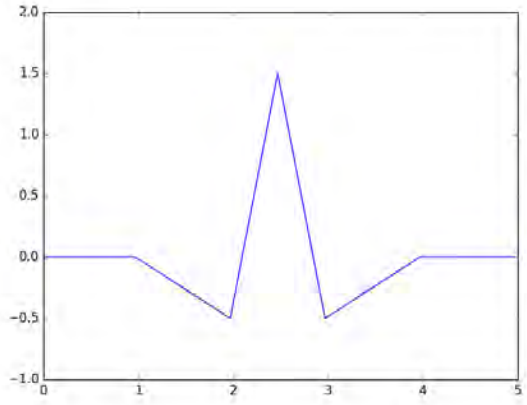
Abbildung A.1: Analysefilter (a),(b) und Synthesefilter (c),(d) für das JPEG2000-9/7-Wavelet (Daubechies) (Generiert mit PyWavelets [19])



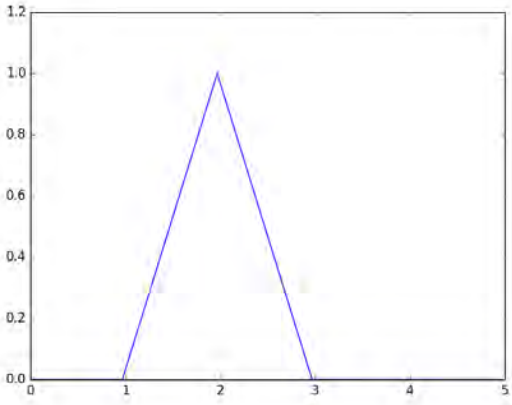
(a) Analyse Tiefpass



(b) Analyse Hochpass



(c) Synthese Tiefpass



(d) Synthese Hochpass

Abbildung A.2: Analysefilter a),(b) und Synthesefilter (c),(d) für das JPEG2000-5/3-Wavelet (LeGall) (Generiert mit PyWavelets [19])

Listing A.1: Code für eindimensionalen Analyse-Kernel.

```
#define CYCLIC 1
#define MOD_LEFT (filter_zero)
#define MOD_RIGHT (filter_size - filter_zero - 1)
#define TID_X (get_global_id(0))

__kernel void filter1D_analysis(__global float *data,
                                const int data_size,
                                const int data_zero,
                                __constant float *filter,
                                const int filter_size,
                                const int filter_zero,
                                __global float *result,
                                const int result_size,
                                const int border_type,
                                const int sampling_factor,
                                const int shift )
{
    int data_pos = (TID_X * sampling_factor)
                    + ((filter_zero + shift + data_zero) % sampling_factor)
                    - filter_zero;

    int pos;
    float out = 0;
    float c = 0;

    if ( TID_X < result_size )
    {

        for ( int k = -MOD_LEFT; k <= MOD_RIGHT; k++ )
        {

            pos = data_pos - k;

            if ( pos < data_size && pos >= 0)
            {

                c = data[pos];
            }
            else
            {

                if (border_type == CYCLIC)
                {

                    c = data[ (pos + data_size) % data_size ];
                }
                else
                {

                    c = 0;
                }
```

```
52
53              }
54
55              out += c * filter[k + filter_zero];
56          }
57
58          result[TID_X] = out;
59      }
60
61      barrier(0);
62  }
```

Listing A.2: Code für eindimensionalen Synthese-Kernel.

```
#define CYCLIC 1
#define MOD_LEFT (filter_zero)
#define MOD_RIGHT (filter_size - filter_zero - 1)
#define TID_X (get_global_id(0))

int mod( int a, int mod )
{
    while (a < 0)
    {
        a += mod;
    }

    return a % mod;
}

__kernel void filter1D_synthesis(__global float *data,
                                 const int data_size_upsampled,
                                 const int data_zero_upsampled,
                                 __constant float *filter,
                                 const int filter_size,
                                 const int filter_zero,
                                 __global float *result,
                                 const int result_size,
                                 const int result_zero,
                                 const int border_type,
                                 const int sampling_factor,
                                 const int sample_pos)
{
    if ( TID_X < result_size
            && TID_X >= (result_zero - data_zero_upsampled - filter_zero )
            && TID_X < (result_zero + data_size_upsampled - data_zero_upsampled + MOD_RIGHT) )
    {
        float c = 0;
        float out = 0;

        int data_index = TID_X - result_zero;

        for ( int k = -MOD_LEFT; k <= MOD_RIGHT; k++ )
        {
            int index = data_index - k;

            if ( index >= (-data_zero_upsampled)
                && index < ( data_size_upsampled - data_zero_upsampled ) )
            {
                int data_pos = index + data_zero_upsampled;

                if ( data_pos % sampling_factor == sample_pos )
                {
                    int pos = (data_pos - sample_pos) / sampling_factor;
                    c = data[pos];
                }
```

```
52                  else
53                  {
54                      c = 0;
55                  }
56              }
57              else
58              {
59                  int data_pos = mod(index + data_zero_upsampled, data_size_upsampled);
60
61                  if ( border_type == CYCLIC
62                       && data_pos % sampling_factor == sample_pos )
63                  {
64                      int pos = (data_pos - sample_pos) / sampling_factor;
65                      c = data[pos];
66                  }
67                  else
68                  {
69                      c = 0;
70                  }
71              }
72
73              out += c * filter[k + filter_zero];
74          }
75
76          result[TID_X] += out;
77      }
78
79      barrier(0);
80  }
```

Listing A.3: Code für zweidimensionalen Analyse-Kernel (horizontal).

```
#define CYCLIC 1
#define MOD_LEFT (filter_zero)
#define MOD_RIGHT (filter_size - filter_zero - 1)
#define TID_X (get_global_id(0))
#define TID_Y (get_global_id(1))

__kernel void filter2d_analysis_h( __global float *data,
                                   const int data_size_x,
                                   const int data_size_y,
                                   const int data_zero_x,
                                   __constant float *filter,
                                   const int filter_size,
                                   const int filter_zero,
                                   __global float *result,
                                   const int result_size_x,
                                   const int border_type,
                                   const int sampling_factor,
                                   const int shift )
{
    int data_pos = (TID_X * sampling_factor)
                   + ((MOD_LEFT + shift + data_zero_x) % sampling_factor)
                   - MOD_LEFT;

    int pos;
    float out = 0;
    float c = 0;
    if ( TID_X < result_size_x && TID_Y < data_size_y )
    {

        for ( int k = -MOD_LEFT; k <= MOD_RIGHT; k++ )
        {
            pos = data_pos - k;

            if ( pos >= 0 && pos < data_size_x)
            {
                c = data[pos + data_shift];
            }
            else
            {
                if (border_type == CYCLIC)
                {
                    int index_tmp = (pos + data_size_x) % data_size_x;
                    c = data[ index_tmp + data_shift ];
                }
                else
                {
                    c = 0;
                }
            }

            out += c * filter[k + filter_zero];
```

```
52          }
53
54          result[TID_X + result_shift] = out;
55      }
56
57      barrier(0);
58  }
```

Listing A.4: Code für zweidimensionalen Analyse-Kernel (vertikal).

```
#define CYCLIC 1
#define MOD_TOP (filter_zero)
#define MOD_BOTTOM (filter_size - filter_zero - 1)
#define TID_X (get_global_id(0))
#define TID_Y (get_global_id(1))

__kernel void filter2d_analysis_v( __global float *data,
                                   const int data_size_x,
                                   const int data_size_y,
                                   const int data_zero_y,
                                   __constant float *filter,
                                   const int filter_size,
                                   const int filter_zero,
                                   __global float *result,
                                   const int result_size_x,
                                   const int result_size_y,
                                   const int border_type,
                                   const int sampling_factor,
                                   const int shift )
{
    int result_shift = TID_Y * result_size_x;

    int data_pos = (TID_Y * sampling_factor)
                   + ((MOD_TOP + shift + data_zero_y) % sampling_factor)
                   - MOD_TOP;

    int pos;
    float out = 0;
    float c = 0;
    if ( TID_X < result_size_x && TID_Y < result_size_y )
    {
        for ( int k = -MOD_TOP; k <= MOD_BOTTOM; k++ )
        {
            pos = data_pos - k;

            if ( pos >= 0 && pos < data_size_y)
            {
                c = data[ TID_X + pos * data_size_x ];
            }
            else
            {
                if (border_type == CYCLIC)
                {
                    int pos_tmp = ( pos + data_size_y ) % data_size_y;
                    c = data[ TID_X + pos_tmp * data_size_x  ];
                }
                else
                {
                    c = 0;
                }
            }
```

```
52
53              out += c * filter[k + filter_zero];
54          }
55
56          result[TID_X + result_shift] = out;
57      }
58
59      barrier(0);
60  }
```

Listing A.5: Code für zweidimensionalen Synthese-Kernel (horizontal).

```
#define CYCLIC 1

#define MOD_LEFT (filter_zero)
#define MOD_RIGHT (filter_size - filter_zero - 1)

#define TID_X (get_global_id(0))
#define TID_Y (get_global_id(1))

int mod( int a, int mod )
{
    while (a < 0)
    {
        a += mod;
    }

    return a % mod;
}

__kernel void filter2d_synthesis_h(__global float *data,
                                   const int data_size_x,
                                   const int data_size_y,
                                   const int data_size_up_x,
                                   const int data_zero_y,
                                   const int data_zero_up_x,
                                   __constant float *filter,
                                   const int filter_size,
                                   const int filter_zero,
                                   __global float *result,
                                   const int result_size_x,
                                   const int result_zero_x,
                                   const int result_zero_y,
                                   const int border_type,
                                   const int sampling_factor,
                                   const int smpl_pos )
{
    #define INDEX_POS_X (index + data_zero_up_x)

    int data_shift = (TID_Y - ( result_zero_y - data_zero_y )) * data_size_x;
    int result_shift = TID_Y * result_size_x;


    if ( TID_X < result_size_x
            && TID_Y < data_size_y + ( result_zero_y - data_zero_y )
            && TID_Y >=  ( result_zero_y - data_zero_y )
            && TID_X >= (result_zero_x - data_zero_up_x - MOD_LEFT )
            && TID_X < (result_zero_x + data_size_up_x
                        - data_zero_up_x + MOD_RIGHT) )
    {
        int data_index = TID_X - result_zero_x;

        float c = 0;
```

```
52          float out = 0;
53          for ( int k = -MOD_LEFT; k <= MOD_RIGHT; k++ )
54          {
55              int index = data_index - k;
56
57              if ( index >= (-data_zero_up_x)
58                  && index < ( data_size_up_x - data_zero_up_x ) )
59              {
60                  if ( (INDEX_POS_X) % sampling_factor == smpl_pos )
61                  {
62                      int pos = ( INDEX_POS_X - smpl_pos ) / sampling_factor;
63                      c = data[pos + data_shift];
64                  }
65                  else
66                  {
67                      c = 0;
68                  }
69              }
70              else
71              {
72                  if ( border_type == CYCLIC
73                      && mod(INDEX_POS_X, data_size_up_x) % sampling_factor == smpl_pos )
74                  {
75                      int pos = (mod(INDEX_POS_X, data_size_up_x) - smpl_pos) / sampling_factor;
76                      c = data[pos + data_shift];
77                  }
78                  else
79                  {
80                      c = 0;
81                  }
82
83              }
84              out += c * filter[k + filter_zero];
85          }
86
87          result[TID_X + result_shift] += out;
88      }
89
90      barrier(0);
91  }
```

Listing A.6: Code für zweidimensionalen Synthese-Kernel (vertikal).

```
#define CYCLIC 1
#define MOD_TOP (filter_zero)
#define MOD_BOTTOM (filter_size - filter_zero - 1)
#define TID_X (get_global_id(0))
#define TID_Y (get_global_id(1))

int mod( int a, int mod )
{
    while (a < 0)
    {
        a += mod;
    }

    return a % mod;
}

__kernel void filter2d_synthesis_v(__global float *data,
                                   const int data_size_x,
                                   const int data_size_up_y,
                                   const int data_zero_x,
                                   const int data_zero_up_y,
                                   __constant float *filter,
                                   const int filter_size,
                                   const int filter_zero,
                                   __global float *result,
                                   const int result_size_x,
                                   const int result_zero_x,
                                   const int result_zero_y,
                                   const int border_type,
                                   const int sampling_factor,
                                   const int smpl_pos )
{
    #define INDEX_POS_Y (index + data_zero_up_y)

    int shift = TID_Y * result_size_x;

    if ( TID_X < data_size_x
            && TID_Y >= ( result_zero_y - data_zero_up_y - filter_zero )
            && TID_Y < ( result_zero_y + data_size_up_y - data_zero_up_y + MOD_RIGHT ) )
    {
        int data_index = TID_Y - result_zero_y;

        float c = 0;
        float out = 0;

        for ( int k = -MOD_LEFT; k <= MOD_RIGHT; k++ )
        {
            int index = data_index - k;

            if ( index >= (-data_zero_up_y) && index < ( data_size_up_y - data_zero_up_y ) )
            {
```

```
52                  if ( (INDEX_POS_Y) % sampling_factor == smpl_pos )
53                  {
54                      int pos = ( INDEX_POS_Y - smpl_pos ) / sampling_factor;
55                      c = data[TID_X + pos * data_size_x];
56                  }
57                  else
58                  {
59                      c = 0;
60                  }
61              }
62              else
63              {
64                  if ( border_type == CYCLIC
65                      && mod(INDEX_POS_Y, data_size_up_y) % sampling_factor == smpl_pos )
66                  {
67                      int pos = (mod(INDEX_POS_Y, data_size_up_y) - smpl_pos) / sampling_factor;
68                      c = data[TID_X + pos * data_size_x];
69                  }
70                  else
71                  {
72                      c = 0;
73                  }
74              }
75
76              out += c * filter[k + filter_zero];
77          }
78
79          result[TID_X + ( result_zero_x - data_zero_x ) + shift] += out;
80      }
81
82      barrier(0);
83  }
```

Listing A.7: Aufbau von `Node` für die erweiterte 2D-Kaskade. `Node` für die erweiterte 1D-Kaskade ist analog aufgebaut mit `dev_data1D` anstelle von `dev_data2D`

```
struct ExtCascade2D::Node
{
    dev_data2D data;

    bool is_used;

    bool *next_calc;

    vector<vector<pNode>> next_nodes;

    Node( dev_data2D data, const unsigned int c_size )
    {
        this->data = data;
        this->is_used = false;

        this->next_calc = new bool[c_size];

        for (unsigned int i = 0; i < c_size; i++)
            this->next_calc[i] = false;

        this->next_nodes = vector<vector<pNode>>(c_size);
    }

    ~Node()
    {
        delete[] this->next_calc;

        if (this->is_used == false)
        {
            release_dev_data( this->data );
        }
    }

    void insert(dev_data_vec2D dev_datas, const unsigned int index, const unsigned int c_size )
    {
        vector<pNode> new_nodes;
        new_nodes.reserve(dev_datas.size());
        for (unsigned int i = 0; i < dev_datas.size(); i++)
        {
            pNode node = new Node( dev_datas[i], c_size );
            new_nodes.push_back( node );
        }

        this->next_nodes[index] = new_nodes;
        this->next_calc[index] = true;
    }
};
```

Literatur

[1] Michael D Adams. *Multiresolution Signal and Geometry Processing: Filter Banks, Wavelets, and Subdivision (Version: 2013-09-26)*. Michael Adams, 2013.

[2] Marc Antonini u. a. „Image coding using wavelet transform“. In: *Image Processing, IEEE Transactions on* 1.2 (1992), S. 205–220.

[3] Charilaos Christopoulos, Athanassios Skodras und Touradj Ebrahimi. „The JPEG2000 still image coding system: an overview“. In: *Consumer Electronics, IEEE Transactions on* 46.4 (2000), S. 1103–1127.

[4] Peter L Chu. „Quadrature mirror filter design for an arbitrary number of equal bandwidth channels“. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 33.1 (1985), S. 203–218.

[5] Didier Le Gall und Ali Tabatabai. „Sub-band coding of digital images using symmetric short kernel filters and arithmetic coding techniques“. In: *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*. IEEE. 1988, S. 761–764.

[6] Alain Hore und Djemel Ziou. „Image quality metrics: PSNR vs. SSIM“. In: *Pattern recognition (icpr), 2010 20th international conference on*. IEEE. 2010, S. 2366–2369.

[7] Mike Houston. „AMD and OpenCL“. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, S. 1–25.

[8] *ImageMagick*. `https://www.imagemagick.org/`. Accessed: 19.7.2016.

[9] The Khronos Group Inc. *OpenCL 1.2 Reference Pages*. `https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`. Accessed: 10.7.2016.

[10] Vijay K Jain und Ronald E Crochiere. „Quadrature mirror filter design in the time domain“. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 32.2 (1984), S. 353–361.

[11] David B Kirk und W Hwu Wen-mei. *Programming Massively Parallel Processors (Second Edition)*. Second Edition. Morgan Kaufmann, 2013.

[12] Erik Lindholm u. a. „NVIDIA Tesla: A unified graphics and computing architecture“. In: *IEEE micro* 28.2 (2008), S. 39–55.

[13] Richard Maltan. „Effiziente Berechnung der FWT auf Grafikkarten“. Bachelor Thesis. Universität Passau, 2014.

[14] *mingw-w64 GCC for Windows 64 & 32 bits*. `http://mingw-w64.org/doku.php/start`. Accessed: 28.7.2016.

[15] Aaftab Munshi. *The OpenCL Specification*. Version: 1.2. Document Revision: 19. Khronos OpenCL Working Group.

[16] *OpenCl Programming for the CUDA Architecture*. v2.3Version 2.3. NVIDIA. Aug. 2009.

[17] *OpenJPEG*. `http://www.openjpeg.org/`. Accessed: 2016-06.23.

[18] Boaz Porat. *A course in digital signal processing*. Bd. 1. Wiley New York, 1997.

[19] *PyWavelets - Discrete Wavelet Transform in Python*. `https://pywavelets.readthedocs.io/en/latest/index.html`. Accessed: 2016.06.23.

[20] Tomas Sauer. *Einführung in die Signal- und Bildverarbeitung*. Vorlesungsskript Universität Passau. 2012.

[21] Tomas Sauer. *Wavelets*. Vorlesungsskript Universität Passau. 2011/2012.

[22] Athanassios N Skodras, Charilaos A Christopoulos und Touradj Ebrahimi. „JPEG2000: The upcoming still image compression standard“. In: *Pattern Recognition Letters* 22.12 (2001), S. 1337–1345.

[23] Athanassios Skodras, Charilaos Christopoulos und Touradj Ebrahimi. „The JPEG 2000 still image compression standard“. In: *Signal Processing Magazine, IEEE* 18.5 (2001), S. 36–58.

[24] Mark JT Smith und Thomas P Barnwell III. „A new filter bank theory for time-frequency representation“. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 35.3 (1987), S. 314–327.

[25] Tilo Strutz. *Bilddatenkompression: Grundlagen, Codierung, Wavelets, JPEG, MPEG*. Bd. 264. Springer-Verlag, 2009.

[26] Michael Unser und Thierry Blu. „Mathematical properties of the JPEG2000 wavelet filters“. In: *Image Processing, IEEE Transactions on* 12.9 (2003), S. 1080–1090.

[27] Palghat Vaidyanathan. „Quadrature mirror filter banks, M-band extensions and perfect-reconstruction techniques“. In: *ASSP Magazine, IEEE* 4.3 (1987), S. 4–20.

[28] PP Vaidyanathan. „Theory and design of M-channel maximally decimated quadrature mirror filters with arbitrary M, having the perfect-reconstruction property“. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 35.4 (1987), S. 476–492.

[29] Martin Vetterli. „A theory of multirate filter banks“. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 35.3 (1987), S. 356–372.

[30] Martin Vetterli und Cormac Herley. „Wavelets and filter banks: Theory and design“. In: *Signal Processing, IEEE Transactions on* 40.9 (1992), S. 2207–2232.

[31] Martin Vetterli und Jelena Kovacevic. *Wavelets and subband coding*. LCAV-BOOK-1995-001. Prentice-hall, 1995.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum Unterschrift