

Eberhard Karls Universität Tübingen  
Mathematisch-Naturwissenschaftliche Fakultät  
Wilhelm-Schickard-Institut für Informatik

## Masterarbeit Medieninformatik

# Analyse und plattformunabhängige Implementierung der hardwarebeschleunigten Auflösung des Bayer-Mosaiks

Florian Kellner

14.07.2023

### Betreuer

Prof. Dr. Thomas Walter (Informationsdienste) Wilhelm-Schickard-Institut für Informatik Universität Tübingen	Prof. Dr. Andreas Schilling (Visual Computing) Wilhelm-Schickard-Institut für Informatik Universität Tübingen
---	--

**Kellner, Florian:**

*Analyse und plattformunabhängige Implementierung der hardwarebeschleunigten Auflösung des Bayer-Mosaiks*

Masterarbeit Medieninformatik

Eberhard Karls Universität Tübingen

Bearbeitungszeitraum: 01.04.2023 - 01.09.2023

## Zusammenfassung

Digitale Kamerasensoren nehmen Farben nicht simultan wahr, sondern bestehen aus Helligkeitssensoren, vor die in einem nach seinem Erfinder benannten Bayer-Mosaik Farbfilter der Farben rot, grün und blau geschaltet sind. Mit welchem Algorithmus die jeweils fehlenden Farbwerte interpoliert werden, hat einen großen Einfluss auf die Schärfe und Detailtreue des resultierenden Farbbildes. Auf professionellen Kameras können die Sensordaten deswegen weitestgehend unverarbeitet gespeichert werden.

Mit Jeniffer2 wurde von Ljavin ([2020](#)) eine Java-Anwendung zur Verarbeitung von Bildern im offen standardisierten Adobe DNG (Digital Negative) Rohdatenformat geschaffen. Reiter ([2023](#)) erweiterte diese um einige aktuelle und teils recht komplexe Demosaicing-Algorithmen, zwischen denen frei gewählt werden kann. Diese Algorithmen sowie die Schritte zum Postprocessing stellten sich als teilweise sehr zeitintensiv heraus.

In dieser Arbeit wurden verschiedene Ansätze erkundet, die implementierten Berechnungen zu beschleunigen, wobei ein besonderer Fokus auf der Plattformunabhängigkeit, der Integration in Java und dem Erhalt der Les- und Wartbarkeit der Software lag. Nach einer Zusammenfassung der technischen Möglichkeiten wurde zunächst die Portierung eines Demosaicing-Algorithmus und des Postprocessing auf die Grafikkarte mittels OpenGL umgesetzt und in einem Feldtest auf den Endgeräten von 20 Teilnehmenden validiert.

Aufgrund der aufgetretenen Schwierigkeiten bei der Cross-Platform-Kompatibilität der GPU-Version wurden die bei der Portierung gewonnenen Kenntnisse über den Datenfluss innerhalb des Algorithmus für die Optimierung aller Algorithmen auf der CPU genutzt. So konnte für die drei komplexesten Algorithmen eine Beschleunigung um etwa das zehnfache erreicht werden, bei einer gleichzeitigen Verbesserung der Verständlichkeit des Quellcodes. Zusätzlich wurde eine Verarbeitung der Bilder in Kachelabschnitten getestet, was die Performance nochmals geringfügig verbessert, aber vor allen den Speicherbedarf der Anwendung reduziert. Ermittelt wurden diese Werte in extensiven Benchmarks sowohl auf einem etwas älteren Laptop mit x86-Architektur als auch auf einem modernen Mac Mini mit M1-Chiparchitektur. Aufgrund verschiedener Faktoren ist die finale CPU-Version fast so schnell wie die Version auf der GPU (sofern vorhanden).

Im Verlauf der Arbeit wurde die Codebasis von Jeniffer2 weiter gepflegt. Außerdem wurden mit der Pixellupe mit verschiedenen Vergleichsmodi und dem schnellen Konfigurationswechsler zwei neue Elemente hinzugefügt, die zum Erkunden und Vergleichen der Demosaicing-Algorithmen einladen.

# Danksagung

Zuallererst geht mein Dank natürlich an Prof. Dr. Walter, den geistigen Vater von Jeniffer, sowohl für die gutgelaunte Betreuung der Arbeit als auch für das Mentoring in Karrierefragen.

Ein weiteres Dankeschön geht an Chaz fürs Korrekturlesen und die konstruktive Kritik.

Mein Dank geht auch an Andreas Reiter, der mir immer für Fragen zu seiner Arbeit zur Verfügung stand und seinen Matlab-Code mit mir geteilt hat.

Auch wenn sie in der Literatur bereits erwähnt sind, soll hier nochmal explizit allen Autorinnen und Autoren der verwendeten Blogbeiträge, Wikis und Tutorials dafür gedankt werden, dass sie sich unentgeltlich die Zeit genommen haben, ihr Wissen für Fremde wie mich aufzubereiten und verfügbar zu machen.

Zu guter Letzt geht ein großes Dankeschön an meine Freunde und Familie, die mich auf der doch recht langen Zielgerade unterstützt haben und mir zur Seite gestanden sind.

Tübingen, 14.07.2023

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	vii
<b>Tabellenverzeichnis</b>	ix
<b>Abkürzungsverzeichnis</b>	xi
<b>1 Einleitung</b>	1
1.1 Motivation . . . . .	1
1.2 Ziele . . . . .	2
1.3 Überblick . . . . .	3
<b>2 Grundlagen Demosaicing und Jeniffer</b>	4
2.1 Offline-RAW-Processing und das DNG-Format . . . . .	4
2.2 Demosaicing-Algorithmen und Komplexität . . . . .	4
<b>3 Grundlagen Paralleles Programmieren</b>	9
3.1 Motivation . . . . .	9
3.2 Theorie . . . . .	10
3.2.1 Amdahl's Law . . . . .	10
3.2.2 Gustafson's Law . . . . .	10
3.2.3 Flynn's Taxonomie . . . . .	12
3.3 Parallelismus auf der CPU . . . . .	13
3.3.1 SIMD-Instruktionen . . . . .	13
3.3.2 Multithreading/Streams . . . . .	15
3.4 Parallelismus auf der GPU . . . . .	19
3.4.1 CUDA . . . . .	21
3.4.2 OpenCL . . . . .	22
3.4.3 OpenGL . . . . .	23
3.4.4 TornadoVM . . . . .	25
3.5 Vergleich CPU/GPU . . . . .	25
<b>4 Grundlagen Speicheroptimierung</b>	28
<b>5 Vorgehen</b>	29

5.1	Anforderungen an die Software . . . . .	29
5.2	Auswahl der Frameworks . . . . .	29
5.2.1	CPU . . . . .	29
5.2.2	GPU . . . . .	30
5.3	Empirie . . . . .	30
5.3.1	Feldtest . . . . .	30
5.3.2	Benchmarks . . . . .	31
5.4	Verworfene Ansätze . . . . .	32
5.4.1	Caching und Abbildungstabellen . . . . .	32
5.4.2	TornadoVM . . . . .	33
<b>6</b>	<b>Implementierung</b>	<b>37</b>
6.1	CLI-Modul . . . . .	37
6.2	Logging . . . . .	37
6.2.1	Interface . . . . .	38
6.2.2	Systeminformationen . . . . .	38
6.3	Regressionstests . . . . .	39
6.4	Distribution . . . . .	39
6.4.1	Java-Modulsystem . . . . .	40
6.4.2	Reflektion . . . . .	40
6.4.3	Azul Zulu und Warp-Packer . . . . .	41
6.5	Verarbeitungsschritte auf der Grafikkarte . . . . .	41
6.5.1	Grafikkontext . . . . .	43
6.5.2	Datenübertragung . . . . .	43
6.5.3	Datenverarbeitung . . . . .	44
6.5.4	Aufteilung in Kacheln - “Tiling” . . . . .	46
6.5.5	Tests . . . . .	47
6.6	CPU-Optimierung . . . . .	47
6.6.1	Datenflussanalyse . . . . .	49
6.6.2	Tiling auf der CPU . . . . .	52
6.6.3	Zusammenfassung von Operationen . . . . .	54
6.6.4	Edge-Handling-Strategie . . . . .	54
6.7	Reproduktion Qualitätsbenchmarks . . . . .	54

6.8	Grafische Benutzeroberfläche . . . . .	56
6.8.1	Wahl der Beschleunigungsstrategie . . . . .	56
6.8.2	Fehleranzeige . . . . .	58
6.8.3	Algorithmenvergleich . . . . .	58
6.8.4	Zoom und Pixellupe . . . . .	59
<b>7</b>	<b>Ergebnisse</b>	<b>61</b>
7.1	Reproduktion der qualitativen Metriken . . . . .	61
7.2	Feldtest . . . . .	66
7.2.1	Hardware der Stichprobe . . . . .	66
7.2.2	Bugs und Probleme . . . . .	66
7.2.3	Einfluss von Größe und Verarbeitungsreihenfolge . . . . .	69
7.2.4	Vergleich der Beschleunigung für die einzelnen Schritte .	71
7.3	Benchmarks . . . . .	72
7.3.1	Testgeräte . . . . .	73
7.3.2	Testbilder . . . . .	73
7.3.3	Kachelgröße und Multithreading-Strategie . . . . .	73
7.3.4	Übersicht nach Bildgröße und Gerät . . . . .	76
7.3.5	Optimale Kachelgröße und Strategie für große Bilder .	80
7.3.6	Optimale Kachelgröße und Strategie für kleine Bilder .	84
7.3.7	Pre- und Postprocessing . . . . .	84
7.3.8	Beste Strategie insgesamt . . . . .	85
7.4	Diskussion . . . . .	92
7.4.1	Nutzen der Verwendung der Grafikhardware . . . . .	92
7.4.2	Optimierungen auf der CPU . . . . .	93
<b>8</b>	<b>Fazit und Ausblick</b>	<b>95</b>
8.1	Fazit . . . . .	95
8.2	Ausblick . . . . .	97
8.2.1	Open-Sourcing . . . . .	97
8.2.2	Optimierungsansätze für die Zukunft . . . . .	97
8.2.3	Neuronale Netze und KI . . . . .	97
8.2.4	Benutzeroberfläche: Feinschliff oder Lehrbuch . . . . .	98



# Abbildungsverzeichnis

1	Sensor mit Bayer-Mosaik, Quelle: RawPedia (2023) . . . . .	1
2	DNG-Verarbeitungsschritte (grün implementiert in Jeniffer2), Quelle: Ljavin (2020)) . . . . .	5
3	Links: Aliasing-Artefakte bei Bilinear-Median-Demosaicing, Rechts: Zipper-Artefakte bei Nearest-Neighbour-Demosaicing .	7
4	Ausführungszeiten verschiedener Demosaicing-Algorithmen in Jeniffer2, Reiter (2023) . . . . .	7
5	Entwicklung der CPU-Frequenz laut Stanford CPU Database (Danowitz et al. (2012)) . . . . .	9
6	Vergleich Amdahl's Law (fixe Inputgröße) vs Gustafson's Law (fixe Ausführungszeit, paralleler Anteil O(1)) . . . . .	11
7	SISD vs SIMD-Architektur, Quelle: Engheim (2021) . . . . .	13
8	Performanz einer einfachen Addition von Werten in 2 Arrays mit unterschiedlichen Beschleunigungsstrategien, Quelle: Stypinski (2022a) . . . . .	16
9	Performanz einer einfachen Addition mit Multiplikation mit unterschiedlichen Beschleunigungsstrategien, Quelle: Stypinski (2022b) . . . . .	17
10	Unterschied der Transistoraufteilung CPU vs GPU (NVIDIA Corporation & Affiliates (2023)) . . . . .	20
11	Aufteilung von Thread-Blöcken auf Streaming Multiprocessors (NVIDIA Corporation & Affiliates (2023)) . . . . .	21
12	Die OpenGL Rendering Pipeline (programmierbare Teile in blau, Quelle: OpenGL Wiki contributors (2022)) . . . . .	24
13	Formel für die Gammakorrektur laut Ljavin (2020) . . . . .	32
14	Vergleich der Ausführungszeiten des RCD-Demosaicing mit Tor- nadoVM und Multithreading . . . . .	34
15	Ausschnitt TornadoVM Profiler-Ergebnis auf der integrierten Grafikkarte . . . . .	35
16	Vergleich DNG-Processing auf der CPU und auf der GPU . .	42
17	Bildverarbeitung Kachel für Kachel . . . . .	48

18	Bildverarbeitung alle Kacheln auf einmal . . . . .	49
19	Datenfluss Ratio Corrected Demosaicing auf der GPU . . . . .	50
20	Datenfluss Ratio Corrected Demosaicing (alte Implementation) . . . . .	51
21	Datenfluss Ratio Corrected Demosaicing (neue Implementation) . . . . .	51
22	Datenfluss DLMMSE Demosaicing . . . . .	52
23	Datenfluss DLMMSE+RCD Demosaicing . . . . .	53
24	Akkuratheit der reimplementierten Algorithmen anhand unterschiedlicher Metriken . . . . .	57
25	Konfigurationsmöglichkeiten in Jeniffer2 . . . . .	58
26	Konfigurations-Vergleich in der Toolbar . . . . .	59
27	Pixel-Lupe . . . . .	60
28	Farbkanal- und Vergleichsmodi der Pixel-Lupe . . . . .	60
29	Überblick über die Demosaicing-Algorithmen in Jeniffer2 anhand verschiedener Metriken . . . . .	62
30	Ergebnisse für Mean Square Error aufgeschlüsselt nach Datenset . . . . .	63
31	Ergebnisse für PSNR aufgeschlüsselt nach Datenset . . . . .	64
32	Ergebnisse für MSSIM aufgeschlüsselt nach Datenset . . . . .	65
33	Speedup verschiedener Verarbeitungsschritte nach Bildgröße und Verarbeitungsreihenfolge, n=13 . . . . .	70
34	3,3 Megapixel Testbild, entwickelt mit RCD . . . . .	74
35	47,4 Megapixel Testbild, entwickelt mit RCD und herunterskaliert . . . . .	75
36	61,1 Megapixel Testbild, entwickelt mit RCD und herunterskaliert . . . . .	75
37	Ideale Lebensdauer Pixel im Cache bei der Berechnung des PQ-Gradienten im RCD- und DLMMSE+RCD-Algorithmus bei einem 16px breiten Bild . . . . .	81
38	Vergleich gesamtes DNG Processing auf dem ThinkPad W530 nach Beschleunigungsstrategie (mit integrierter Grafikkarte) . . . . .	94
39	Beste Ausführungszeit Demosaicingalgorithmen (Durchschnitt Mac Mini M1/ThinkPad W530) vs. Speedup-Faktor gegenüber alter Version (nur ThinkPad), 47,4 MP Bild . . . . .	96

# Tabellenverzeichnis

1	Ausführungszeiten der einzelnen Verarbeitungsschritte in Jenifer2, Quelle: Ljavin (2020) . . . . .	6
2	Speicherbausteine und Latenzen für einen dem Testgerät ähnlichen Prozessor und einen aktuellen Prozessor, Quelle: Pavlov (2023) (eigene Zusammenstellung) . . . . .	28
3	Überblick über die Hardware der Stichprobe . . . . .	67
4	Zusammenfassung der aufgetretenen Fehler nach Art der GPU . . . . .	68
5	Durchläufe in der Stichprobe . . . . .	69
6	Durchläufe pro Person und Konfiguration . . . . .	69
7	Durchschnittszeiten Verarbeitung des größeren Bilds mit GPU Tiling . . . . .	71
7	Durchschnittszeiten Verarbeitung des größeren Bilds mit GPU Tiling . . . . .	72
8	Durchschnittszeiten Verarbeitung des größeren Bilds mit Multithreading . . . . .	72
9	Vergleich der Ausführungszeiten nach Beschleunigungsstrategie, Bildgröße und Gerät, Teil 1 . . . . .	78
10	Vergleich der Ausführungszeiten nach Beschleunigungsstrategie, Bildgröße und Gerät, Teil 2 . . . . .	79
11	Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 47,4 MP Bild, ThinkPad W530 . . . . .	82
12	Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 47,4 MP Bild, Mac Mini M1 . . . . .	83
13	Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 3,3 MP Bild, ThinkPad W530 . . . . .	86
14	Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 3,3 MP Bild, Mac Mini M1 . . . . .	87
15	Implementationsvorschlag ideale Beschleunigungsstrategie für kleine Bilder . . . . .	88
16	Implementationsvorschlag ideale Beschleunigungsstrategie für große Bilder (Trade-Off-Werte für mittleres Bild) . . . . .	89

17	Ausführungszeiten in allen Konfigurationen, Preprocessing . . .	90
18	Ausführungszeiten in allen Konfigurationen, Postprocessing . .	91
19	Implementationsvorschlag ideale Beschleunigungsstrategie für Pre- und Postprocessingschritte (Trade-Off-Werte großes Bild für 47MP Bild) . . . . .	92

## Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface
<b>CFA</b>	Color Filter Array
<b>CNN</b>	Convolutional Neural Network
<b>CPU</b>	Central Processing Unit
<b>DLMSE</b>	Directional Linear Minimum Mean-Square Error
<b>DNG</b>	Digital Negative
<b>FMA</b>	Fused Multiply Add
<b>FPGA</b>	Field Programmable Gate Array
<b>FLOPS</b>	Floating-Point Operations Per Second
<b>GLSL</b>	OpenGL Shading Language
<b>GPU</b>	Graphics Processing Unit
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>JDK</b>	Java Development Kit
<b>JENIFFER</b>	Java Extended NEF File Format Editor
<b>JIT-Compilation</b>	Just-In-Time Compilation
<b>JRE</b>	Java Runtime Environment
<b>JVM</b>	Java Virtual Machine
<b>KI</b>	Künstliche Intelligenz
<b>LLVM IR</b>	Low-Level Virtual Machine Intermediate Representation
<b>MIMD</b>	Multiple Instructions, Multiple Data
<b>MISD</b>	Multiple Instructions, Single Data
<b>MSSIM</b>	Mean Structural Similarity Index Measure
<b>NEF</b>	Nikon Electronic Format
<b>PNG</b>	Portable Network Graphics
<b>PPG</b>	Patterned Pixel Grouping
<b>PSNR</b>	Peak Signal to Noise Ratio
<b>RAM</b>	Random Access Memory
<b>RCD</b>	Ratio Corrected Demosaicing
<b>SIMD</b>	Single Instruction, Multiple Data
<b>SISD</b>	Single Instruction, Single Data
<b>SSE</b>	Streaming SIMD Extensions
<b>VM</b>	Virtual Machine



# 1 Einleitung

Bei digitalen Bildern setzt sich jedes Pixel aus drei Farbwerten zusammen: dem Rot-, Grün- und Blauanteil. Im Bildsensor einer Digitalkamera können diese Farbwerte aber nicht simultan an einer Stelle aufgenommen werden, denn er besteht aus Helligkeitssensoren, vor die jeweils entsprechende Farbfilter geschaltet sind. Die Farbfilter sind wie ein Mosaik in einem Farbfilter-Array (Color Filter Array, CFA) nebeneinander angeordnet - die bekannteste und verbreitetste Form ist das nach seinem Erfinder benannte Bayermosaik (Bayer (1976)), das in jeder 2x2-Pixel-Kachel je zwei Grün-, einen Rot- und einen Blausensor enthält (siehe Abb. 1). Ein weiterer Unterschied zu fertigen Bilddateien ist, dass der Kamerasensor oft einen höheren Kontrast registriert als in den üblichen 8 Bit pro Farbwert dargestellt werden kann.

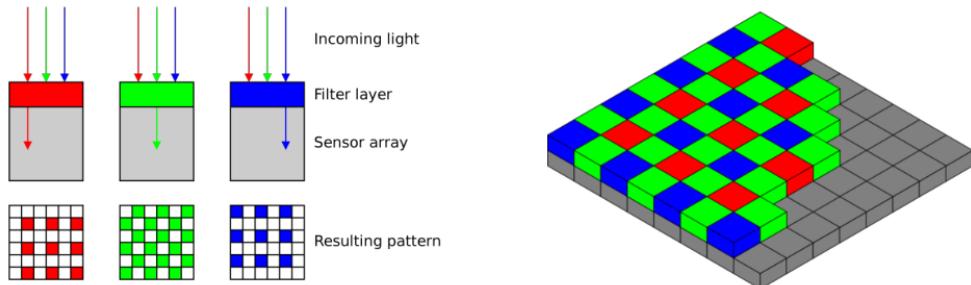


Abbildung 1: Sensor mit Bayer-Mosaik, Quelle: RawPedia (2023)

Während Verbraucherkameras direkt beim Speichern die fehlenden Farbwerte interpolieren, die Bittiefe reduzieren und die Daten in ein Endnutzerformat überführen, speichern professionelle Kameras die Rohdaten und erlauben so die manuelle Bildoptimierung mittels spezieller Software. Neben bekannten Produkten wie z.B. Adobe Lightroom oder Open-Source-Projekten wie Raw-Therapee und Darktable wurde JENIFFER an der Uni Tübingen speziell dafür entwickelt, mit unterschiedlichen Demosaicing-Algorithmen zu experimentieren und diese zu vergleichen. Die aktuelle Version Jeniffer2 ist in Java geschrieben und nimmt als Inputformat Dateien im offen standardisierten DNG-Format an (Ljavin (2020)). Die meisten proprietären Rohbildformate lassen sich mithilfe kostenfreier Programme zu DNG konvertieren.

## 1.1 Motivation

Jeniffer2 enthält fortgeschrittene Demosaicing-Algorithmen, die zum Teil viel Rechenzeit benötigen - in einem Ausmaß, das die Interaktivität der Anwendung

einschränkt. Deswegen bietet es sich an, verschiedene Optionen, die Berechnung zu beschleunigen, zu erkunden, unter anderem durch die Nutzung dedizierter Grafikhardware.

Bisherige Arbeiten zum Thema hardwarebeschleunigtes Demosaicing sind meist durch die Effizienzansprüche von Echtzeit- oder Fast-Echtzeit-Anwendungen wie Streaming oder Fotographie auf einem Smartphone motiviert. McGuire (2008) z.B. zitieren als Erfolgsmetrik ihrer Portierung des Algorithmus von Malvar, He, und Cutler (2004) auf die Grafikkarte, dass damit auf aktueller Hardware simultan bis zu 40 HD-1080p-Videostreams gefiltert werden könnten, und Goorts, Rogmans, and Bekaert (2012) erhöhen die Performanz weiter durch die Low-Level-Optimierung auf NVIDIA-Grafikkarten. Auch Zapryanov und Nikolova (2019) beschränken sich auf NVIDIA-Grafikkarten und vergleichen die Performanz verschiedener Algorithmen in Gigabyte pro Sekunde Durchsatz und Frames Per Second. Langseth et al. (2014) vergleichen Qualität und Latenz verschiedener Demosaicing-Algorithmen explizit für den Anwendungsfall, das Bild von Panoramakameras in einem Sportstadion zu streamen und aufzunehmen.

Aufgrund der Echtzeitanforderungen wird jedoch oft mit eher einfachen Algorithmen und lediglich 8 Bit Farbtiefe gearbeitet (so auch z.B. Wang, Guo, und Wei (2019)). Eine Ausnahme stellen Faruqi, Ino, und Hagihara (2012) mit ihrem fortgeschrittenen Variance of Colors-Algorithmus dar, mit dem sie 4K-Aufgelöste RAW-Daten verarbeiten, allerdings auch mit 8 Bit Farbtiefe.

Auch wenn das Ziel dieser Arbeit ist, die Bildverarbeitung zu beschleunigen, hat Jeniffer2 als RAW-Konverter keinen Echtzeitanspruch. Dafür sollen komplexere und experimentelle Demosaicing-Algorithmen, eine Bittiefe von bis zu 16 Bit und Bilder in einer Auflösung, die ein Vielfaches von HD-Videoframes beträgt, unterstützt werden.

Als weiteres herausstellendes Merkmal soll die in Java geschriebene und damit plattformunabhängige Software Jeniffer2 weiterhin auf möglichst vielen Endgeräten laufen, sei es mit NVIDIA-, AMD- oder eingebauter Intel-Grafikkarte oder mit modernen Chip-Architekturen wie Apple Silicon.

## 1.2 Ziele

Ziel dieser Arbeit ist es also einerseits, die Vorteile von Jeniffer2 zu erhalten. Dazu gehören unter anderem die bereits angesprochene Plattformunabhängigkeit, aber auch softwaretechnische Faktoren wie die klare Programmstruktur und die einfache Erweiterbarkeit durch neue Algorithmen, welche verglichen werden können.

Andererseits soll die Ausführungszeit reduziert und damit die Interaktivität der Anwendung gesteigert werden. Dafür werden verschiedene Optimierungs-

und Parallelisierungsoptionen sowohl zur effizienten Ausnutzung der CPU als auch zur Einbindung von Grafikhardware explorativ erkundet. Die Effektivität und Portabilität der Umsetzung auf der GPU wird in einem Feldtest verifiziert, während Benchmarks auf zwei unterschiedlich alten Testgeräten dabei helfen, die Optimierungen auf der CPU zu tunen und einzuordnen.

### 1.3 Überblick

In Kapitel 2 werden zunächst die Grundlagen des Demosaicing-Prozesses und der Verarbeitung von Kamera-RAW-Daten im DNG-Format durch Jeniffer2 zusammengefasst. Kapitel 3 gibt einen Überblick über Motivation, Theorie und Implementation paralleler Programmierung sowohl auf der CPU als auch auf der GPU. Hier wird auf bisherige Arbeiten eingegangen und zusätzlich ein Fokus auf die Möglichkeiten in der verwendeten Programmiersprache Java gelegt. Abgeschlossen wird der Theorieteil mit einer kurzen Anmerkung zum Thema Speicheroptimierung und Caches (Kapitel 4).

Welche Optionen experimentell verfolgt werden und warum, wird in Kapitel 5 dargelegt - inklusive einer Beschreibung der verworfenen Optionen. Hier wird auch ein Überblick über die später durchgeföhrten Tests und die Feldstudie gegeben. Kapitel 6 beschreibt schließlich im Detail die implementierten Änderungen und geht auf Herausforderungen der verschiedenen Ansätze ein. Dies umfasst sowohl Erfahrungen in der Cross-Plattform-Distribution als auch Implementationsentscheidungen in der Umsetzung auf der GPU und die Datenflussanalyse fortgeschrittenen Demosaicing-Algorithmen zur Optimierung auf der CPU.

Kapitel 7 beschreibt und diskutiert die Ergebnisse der Feldstudie und der Benchmarks. Abschließend werden in Kapitel 8 die gewonnenen Erkenntnisse zusammengefasst und ein Ausblick auf weitere Entwicklungsaufgaben sowie die Veröffentlichung des Quellcodes gegeben.

## 2 Grundlagen Demosaicing und Jeniffer

### 2.1 Offline-RAW-Processing und das DNG-Format

Um die Dateigrößen klein zu halten und den Endnutzern maximale Kontrolle über die Verarbeitungsschritte der Bilddaten zu lassen, lösen High-End-Kameras das Bayer-Mosaik nicht per Default selbst auf, sondern speichern die Rohdaten, die dann offline verarbeitet werden können.

Hierbei ist zu erwähnen, dass das Demosaicing zwar ein sehr wichtiger und potentiell rechenintensiver, aber nicht der einzige Verarbeitungsschritt von den Rohdaten zum endgültigen Bild ist. Weitere Schritte sind z.B. die Dunkelstromkompensation, der Weißabgleich, die Gammakorrektur und die Farbraumtransformation.

Diese Schritte benötigen Metainformationen von der Kamera, wie z.B. die Anordnung der Farben im Bayer-Mosaik und den Kamerafarbraum. Die Dateiformate dafür unterscheiden sich von Hersteller zu Hersteller. Das von Nikon verarbeitete Format heißt zum Beispiel NEF (Nikon Extended File Format). Der Name Jeniffer ist ursprünglich ein Akronym: Der **J**ava **E**nhaned **N**ef **I**mage **F**ile **F**ormat **E**dito**R** konnte in seiner ersten Version nur NEF-Dateien verarbeiten.

Doch es gibt inzwischen einen - im Gegensatz zu den Hersteller-Formaten - offen lizenzierten Standard, der auf dem Tagged Image Format (TIF) basiert: Den Adobe DNG (Digital Negative)-Standard (Adobe Inc. (2021)). Für die Konvertierung von Herstellerformaten in DNG gibt es sowohl ein kostenloses Tool der Firma Adobe als auch diverse frei verfügbare Dienste im Internet. Manche Kameras speichern inzwischen auch schon nativ im DNG-Format. Sogar das Apple ProRAW-Format, das von iPhone-Kameras gespeichert werden kann, basiert auf DNG und ist nominal mit dem Standard kompatibel (das Mosaik ist hier aber bereits aufgelöst).

Um Bilder möglichst vieler Kameras zu unterstützen, schrieb Ljavin (2020) bei der Neuimplementierung von Jeniffer ein Modul zur DNG-Verarbeitung, welches die wichtigsten Verarbeitungsschritte implementiert (Abb. 2). Dieses stellt die Grundlage der von ihm geschriebenen graphischen Benutzeroberfläche von Jeniffer2 dar. Als zeitraubendster Verarbeitungsschritt stellte sich an dieser Stelle die Farbraumtransformation heraus (siehe Tabelle 1).

### 2.2 Demosaicing-Algorithmen und Komplexität

Um das von der Kamera aufgenommene kontinuierliche Bildsignal für jeden Farbkanal an den fehlenden Stellen zu interpolieren, gibt es eine große Anzahl an unterschiedlichen Algorithmen, die einen großen Einfluss auf die Bildqualität

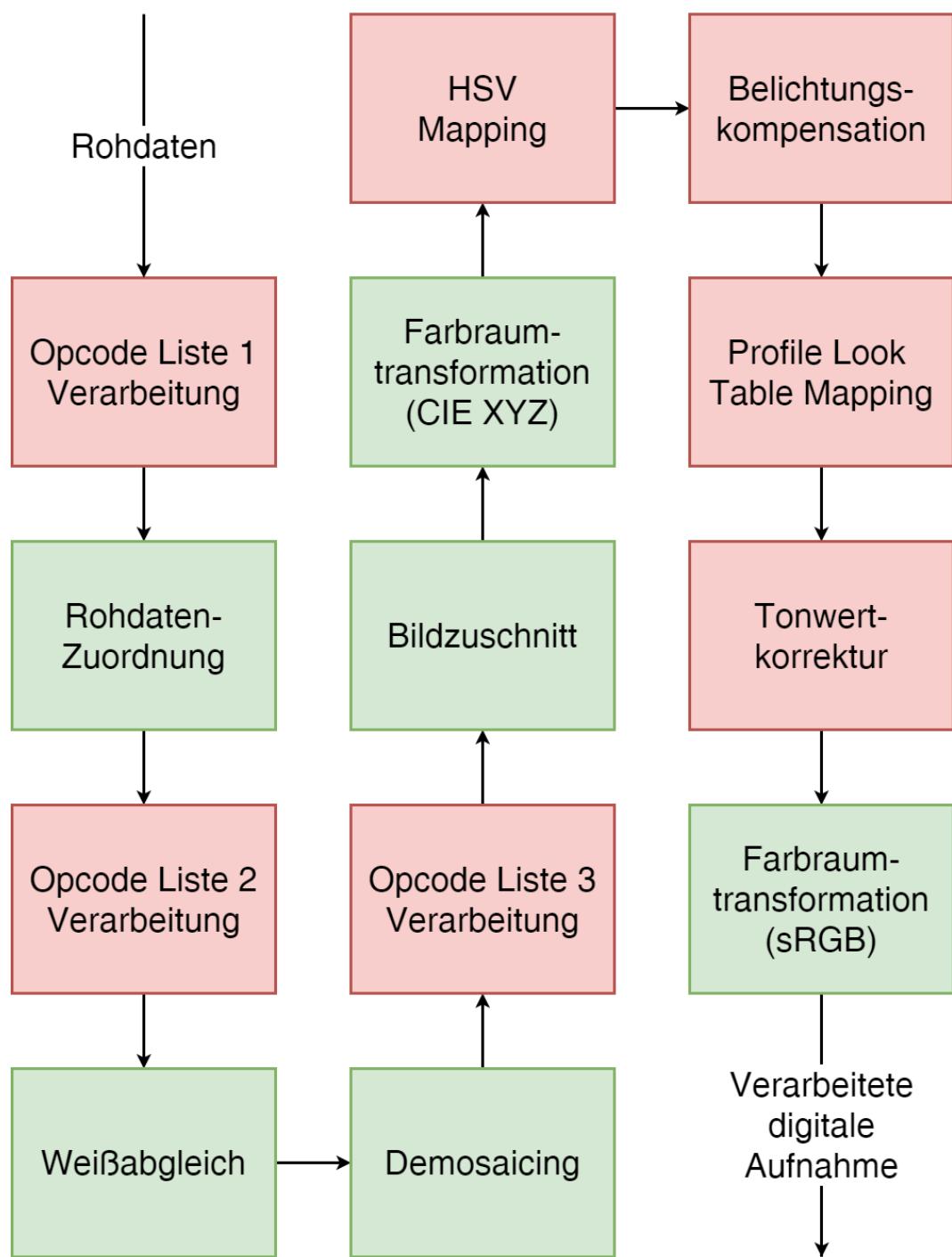


Abbildung 2: DNG-Verarbeitungsschritte (grün implementiert in Jeniffer2),  
Quelle: Ljavin (2020))

Tabelle 1: Ausführungszeiten der einzelnen Verarbeitungsschritte in Jeniffer2,  
Quelle: Ljavin (2020)

	Rohdaten-zuordnung [ms]	Weiß-abgleich [ms]	Demo-saicing [ms]	Farbraum-transformation (davon Gamma-korrektur) [ms]	Summe [ms]
<b>Pixel-wiederholung</b>	1375,3	525,3	761,7	4876,1 (4238,7)	7538,4
<b>Bilineare Interpolation</b>	1407,1	525	1117,7	4755,4 (4154,6)	7805,2
<b>Median Interpolation</b>	1373	521,8	2309,6	5153,2 (4549,8)	9357,6
<b>Bikubische Interpolation</b>	1361,9	520,4	3288,5	5137 (4525,9)	10307,8
<b>Mittelwert</b>	1381,1	523,1	-	4980,4 (4367,3)	-

haben können: Ein schlechter Demosaicing-Algorithmus kann sich z.B. durch Aliasing-Effekte, Zipper-Artefakte oder Falschfarben bemerkbar machen (siehe Abb. 3).

Reiter (2023) erweiterte Jeniffer2 um einige Algorithmen aus Literatur und Open-Source-Praxis. Neben den Algorithmen von Malvar, He, und Cutler (2004) und Hamilton und Adams (1997) wurden auch Patterned Pixel Grouping (PPG) nach Lin (2010), Ratio Corrected Demosaicing (RCD) nach Rodriguez (2017) und Directional Linear Minimum Mean-Square Error (DLMMSE) Demosaicing nach Zhang und Wu (2005) implementiert. Außerdem kombiniert er die existierenden DLMMSE- und RCD-Methoden zu einem neuen Demosaicing-Algorithmus.

Die verschiedenen Algorithmen vergleicht Reiter (2023) sowohl qualitativ als auch quantitativ anhand verschiedener Benchmarks mit großen Sätzen an Referenzbildern, wobei der neue DLMMSE+RCD im Durchschnitt die besten Ergebnisse liefert. Dies ist insbesondere auch deshalb interessant, da die mit DLMMSE+RCD erzeugten Bilder sehr ähnlich zum Resultat der proprietären Algorithmen von Adobe Lightroom und Capture One sind.

Allerdings scheint diese Qualität auch mit einer deutlich höheren Komplexität der Berechnung einherzugehen. So benötigen insbesondere RCD und dessen Kombination mit DLMMSE über die 30-fache Zeit der einfacheren Algorithmen (siehe Abb. 4). Andererseits benötigen weder proprietäre Programme, die ja in der Berechnung dem neuen DLMMSE+RCD-Algorithmus zu ähneln scheinen, noch Open-Source-Tools wie RawTherapee oder DarkTable, die DLMMSE verwenden, so lange wie Jeniffer2 zum Demosaicing. Dies legt nahe, dass die Implementation in Jeniffer2 noch Beschleunigungs- und Optimierungspotentiale



Abbildung 3: Links: Aliasing-Artefakte bei Bilinear-Median-Demosaicing, Rechts: Zipper-Artefakte bei Nearest-Neighbour-Demosaicing

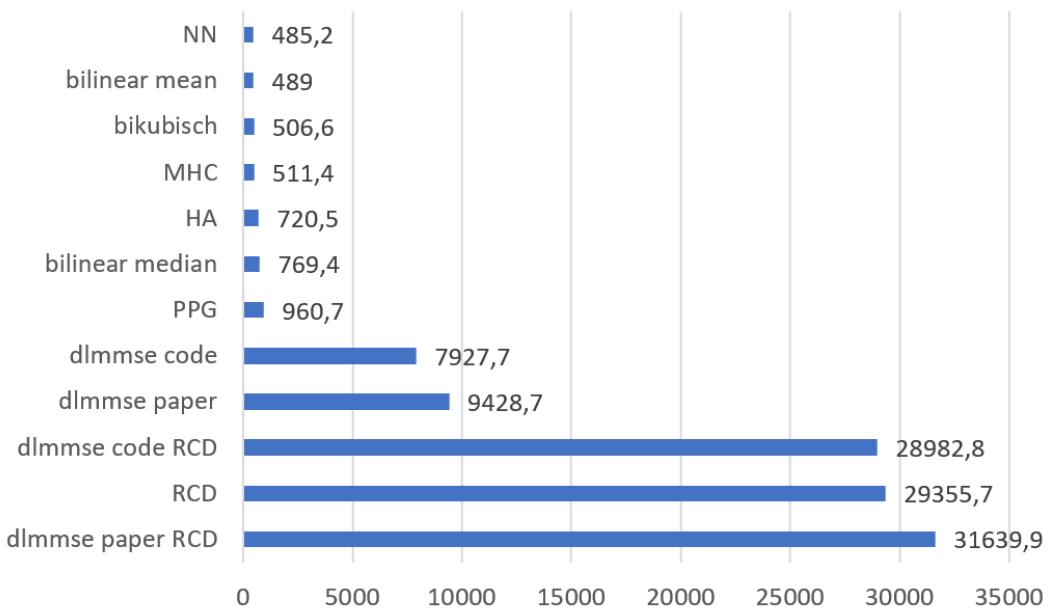


Abbildung 4: Ausführungszeiten verschiedener Demosaicing-Algorithmen in Jeniffer2, Reiter (2023)

ungenutzt lässt.

### 3 Grundlagen Paralleles Programmieren

#### 3.1 Motivation

“Moore’s Law” (Moore et al. (1965)) bezeichnet die seit 1965 gültige Voraussage, dass sich die Anzahl an Transistoren auf einem Computerchip alle 2 Jahre verdoppelt, da Fortschritte in der Fertigungstechnologie immer kleinere Komponenten ermöglichen. In der Vergangenheit übersetzte sich dieser Fortschritt direkt in eine Verdopplung der Frequenz von CPUs - was zur Folge hatte, dass Performanzoptimierung in der Softwareentwicklung einen geringen Wert hatte. (Sutter (2009)). Wie in den Daten aus der Stanford CPU Datenbank (Danowitz et al. (2012), Abb. 5) zu sehen, stagniert seit einigen Jahren allerdings die CPU-Frequenz, unter anderem aufgrund von Problemen mit der Wärmeentwicklung und dem Energieverbrauch bei hohen Frequenzen. Statt die Frequenz zu erhöhen, wird die Verkleinerung von Transistoren zur Erhöhung der Anzahl der Rechenkerne ausgenutzt. Um diese Parallelisierung effektiv in eine Beschleunigung der Berechnung umzusetzen, wird allerdings ein Eingreifen ins Programm nötig.

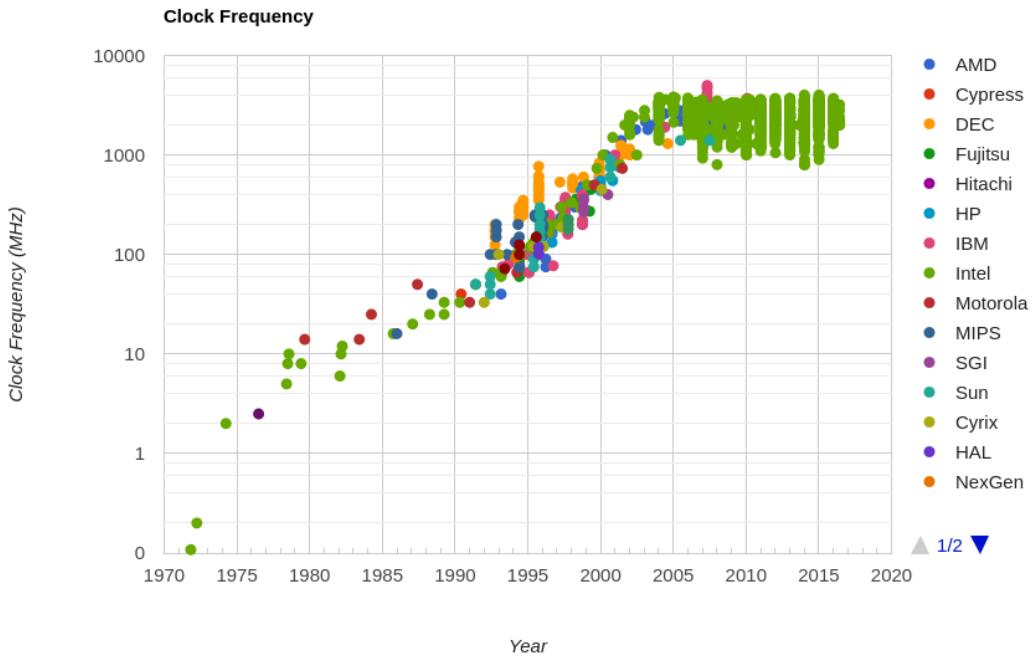


Abbildung 5: Entwicklung der CPU-Frequenz laut Stanford CPU Database (Danowitz et al. (2012))

## 3.2 Theorie

Eine grundsätzliche Begriffsunterscheidung im Bereich des Multithreading ist die zwischen Nebenläufigkeit (engl. Concurrency) und Parallelismus: Nebenläufigkeit ist nämlich auch auf einem einzelnen CPU-Kern möglich. Die Berechnungen werden dann nach verschiedenen Prinzipien abgewechselt, so können z.B. auch Latenzen im Speicherzugriff dazu genutzt werden, dass in der Zwischenzeit anderen Berechnungen der Vorrang gelassen wird. Selbst auf Mehrkern-Prozessoren führt ein Betriebssystem schon allein aufgrund seiner Software-Architektur eine Anzahl an Prozessen aus, die in keinem Vergleich zur Anzahl der Kerne steht.

Für die folgenden theoretischen Grundlagen werden diese praktischen Aspekte aber außer Acht gelassen, und es wird von tatsächlich unabhängig voneinander parallel ausgeführten Berechnungen ausgegangen.

### 3.2.1 Amdahl's Law

Unter dem Namen “Amdahl's Law” ist die Einsicht bekannt, dass der Speedup eines Programms exponentiell sinkt, je geringer der parallelisierbare Anteil ist. Wenn ein Programm also einen parallelisierbaren Anteil von  $0 \leq p \leq 1$  hat, so ist der Speedup gegenüber der nicht parallelisierten Version mit  $S = 1 / ((1 - p) + (p/N))$  zu beziffern, wobei  $N$  die Anzahl der parallelen Prozesse ist. Geht man davon aus, dass  $N$  gegen Unendlich und damit die Ausführungszeit des parallelisierten Anteils gegen null geht, bekommt man für den maximal theoretisch möglichen Speedup die Formel  $S = 1 / (1 - p)$  (Amdahl (1967)).

Die Anwendung von Amdahl's Gesetz auf das Problem der DNG-Verarbeitung ist zu einem großen Teil eine Definitionsfrage: Alle Operationen bis auf das Zuschneiden des Bildbereichs sind lokal für ein einzelnes Pixel oder ein gewisses Fenster an Pixeln definiert und somit theoretisch vollständig parallelisierbar. Allerdings gibt es Synchronisationsbarrieren vor und nach dem Demosaicing, und sowohl das Auslesen der DNG-Datei als auch das Übertragen der Daten an die GUI finden im seriellen Java-Host-Code statt.

### 3.2.2 Gustafson's Law

Gustafson (1988) argumentiert, dass Amdahl's Gesetz zu pessimistisch sei, da der Nutzen eines schnelleren Algorithmus meistens darin läge, ein größeres Problem, also mehr Daten, in der gleichen Zeit verarbeiten zu können. Davon ausgehend, dass die Ausführungszeit des sequentiellen Teils eines Algorithmus unabhängig von der Größe des Inputs konstant sei, berechnet er den Speedup  $S$  als  $S = (1 - p) + p \cdot N$ , wobei  $N$  die Anzahl der verfügbaren Prozessoren beziffert. Dies resultiert in einem linearen statt exponentiellen, und somit in dem Wertebereich von  $p$  stärkeren Speedup (siehe Abb. 6).

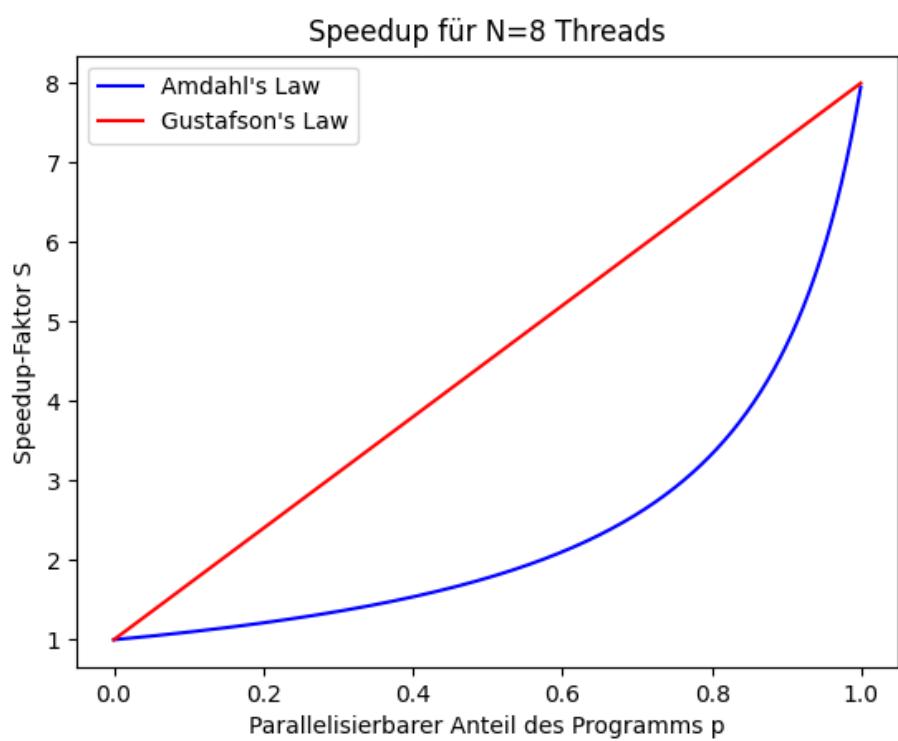


Abbildung 6: Vergleich Amdahl's Law (fixe Inputgröße) vs Gustafson's Law (fixe Ausführungszeit, paralleler Anteil O(1))

Verallgemeinert kann man also sagen, dass es bei der Parallelisierung von Algorithmen eine wichtige Rolle spielt, ob der sequentielle oder parallele Teil stärker mit der Größe des Inputs wächst. Sofern nicht beide Teile gleich stark wachsen, kann mit Amdahl's Gesetz immer nur eine Aussage für eine bestimmte Inputgröße getroffen werden.

Es hängt allerdings vom Anwendungsfall ab, ob eine Vergrößerung des verarbeiteten Inputs tatsächlich einen praktischen Nutzen darstellt. Im Methodenteil dieser Arbeit bleibt die Größe des Inputs die meiste Zeit konstant - es interessiert uns, ob das gleiche Bild schneller verarbeitet werden kann. Da aber die Auflösung von Kameratasensoren mit neuen Fertigungstechniken weiterhin im Wachstum ist, lohnt sich mit Blick auf die Zukunft auch ein Vergleich, ob sich der Speedup zwischen verschiedenen Bildgrößen verändert.

Die Daten von Wenninger (2012), der die Ausführungszeit von GPU-parallelisierten Demosaicing-Algorithmen mit verschiedenen Bildgrößen erhebt, lassen vermuten, dass der serielle Teil der Algorithmen zumindest nicht stärker als der parallele Teil von der Bildgröße abhängt. In der Praxis muss bei Algorithmen, die Daten im Speicher verarbeiten, an den Größengrenzen von Caches und Arbeitsspeicher selbstverständlich mit Stufeneffekten gerechnet werden, es ist aber nicht davon auszugehen, dass Camera-Raw-Daten in absehbarer Zukunft den RAM-Speicher handelsüblicher Verbraucherlaptops sprengen - je nachdem, wie sich die Lage entwickelt, könnten sie aber bald in den CPU-Cache passen (z.B. verfügt der Apple M2 Max Chip über einen L3-Cache von 48MB, während ein Intel i7-Prozessor von vor 10 Jahren noch einen L3-Cache von 6MB hat).

### 3.2.3 Flynn's Taxonomie

Flynn (1972) klassifiziert die Architektur von Prozessoren anhand der parallel ausgeführten Instruktionen und der parallel verarbeiteten Daten.

Der einfachste der 4 Fälle, SISD (Single Instruction, Single Data) bezeichnet den Standardfall, dass eine Instruktion bewirkt, dass eine Berechnung auf einem oder mehreren Elementen eines Datenstroms ausgeführt wird.

Heutzutage aber auch präsent ist das Modell SIMD (Single Instruction, Multiple Data, siehe Abb. 7), welches in seiner Ursprungsdefinition bedeutet, dass ein Programm auf mehrere Datenströme gleichzeitig angewendet wird. Auch wenn es ein Unterschied in der Prozessor-Architektur ist, könnte man mit Blick auf die Berechnung sagen, dass es sich bei SIMD um SISD mit einem größeren Datum handelt, es werden also z.B. Vektoren addiert statt einzelne Floating-Point-Werte. Deswegen werden SIMD-Instruktionen auch häufig als Vektorinstruktionen bezeichnet.

MIMD (Multiple Instruction, Multiple Data) bezeichnet letztendlich moderne

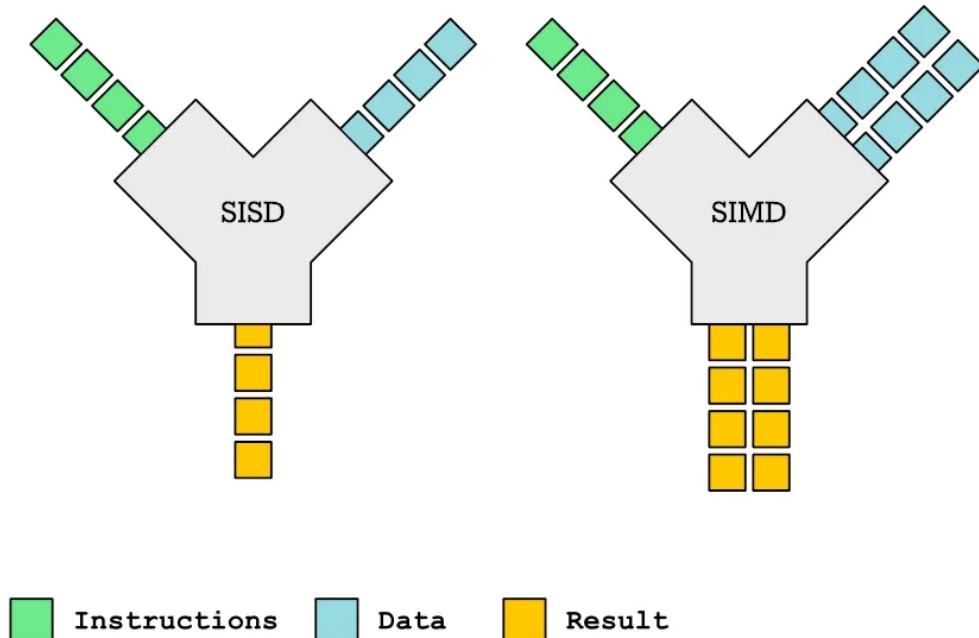


Abbildung 7: SISD vs SIMD-Architektur, Quelle: Engheim ([2021](#))

Multiprozessoren, die unterschiedliche Instruktionen auf unterschiedlichen Daten parallel ausführen. Die Taxonomie vervollständigend existiert auch noch MISD (Multiple Instruction, Single Data), also die Ausführung mehrerer Instruktionen auf dem selben Datum, was bei besonders fehlertoleranten Systemen Sinn ergibt.

### 3.3 Parallelismus auf der CPU

#### 3.3.1 SIMD-Instruktionen

Dass die gleiche Operation auf einem Vektor an Daten ausgeführt werden muss, ist schon lange ein Anwendungsfall in der Audio- und Bilddatenverarbeitung. Für solche nach Flynn ([1972](#)) als SIMD klassifizierten Operationen verfügen moderne CPU-Kerne über entsprechende spezialisierte Hardware, die z.B. mehrere Floating-Point-Werte in einem Vektor parallel addieren und multiplizieren kann. Hierfür werden im Bytecode sogenannte SIMD-Instruktionen zur Verfügung gestellt.

Darktable nutzt diese (zusammen mit anderen Optimierungen wie Multithreading) über das OpenMP-C/C++-Framework (Wikipedia contributors ([2023](#)), für eine Codestelle siehe z.B. Darktable Developers ([2023](#))). Auch RawTherapee nutzt speziell auf das Datenstreaming optimierte SIMD- Instruktionen

(sogenannte SSE2-Instruktionen, siehe Intel ([2023](#))), sofern diese verfügbar sind (siehe z.B. The RawTherapee Team ([2023](#))).

Im Gegensatz zu C und C++ wird Java-Code zunächst zu plattform-unabhängigem Bytecode kompiliert, welcher auf der Java Virtual Machine (JVM) ausgeführt wird. Standardmäßig wird der Code zunächst interpretiert, und für häufig ausgeführte Codestellen wird dann in sogenannter “Tiered Compilation” in mehreren Stufen zunehmend optimierter Maschinencode generiert - zunächst im “C1 Tier”, das auf einen schnellen Programmstart fokussiert ist, dann - falls nötig - im “C2 Tier”, bei dem Profiling-Informationen aus der C1-Version mit einfließen, um für lang laufende Prozesse noch optimiertere Versionen zu generieren.

Einige JVM-Implementationen führen bei der Kompilation eine sogenannte Auto-Vectorization durch, generieren also automatisch für angemessene Programmteile Maschinencode mit SIMD-Instruktionen dort. Hierbei die Korrektheit des Programms zu gewährleisten, ist allerdings nicht trivial, weshalb der Prozess oft erst im C2 Tier stattfindet.

Eine testweise Ausführung von Jeniffer2 auf dem Thinkpad W530 des Autors auf der GraalVM 22.1.0.1 ([OpenJDK 64-Bit Server VM GraalVM 22.1.0.1 \(build 17.0.3+7-jvmci-22.1-b06, mixed mode, sharing\)](#)) mit aktiviertem HotSpot-Disassembly-Plugin (z.B. erhältlich bei Newland ([2023](#))) ergab, dass keine Methode von Jeniffer2 im optimierten C2-Tier kompiliert wird. Das gilt sowohl für die Verarbeitung eines einzelnen Bildes auf der Kommandozeile als auch die Verarbeitung mehrerer Bilder hintereinander mit der graphischen Oberfläche. Es werden auch keine arithmetischen SIMD-Instruktionen wie z.B. `vmulpd` (“Vector Multiplication of Packed Double Precision Values”, also Multiplikation), `vaddpd` (Addition) produziert (Siehe Cloutier ([2023](#)) für eine Übersicht über die verfügbaren Instruktionen, Intel Corporation ([2023](#)) für die offizielle Dokumentation). Es werden lediglich die Versionen der Instruktionen verwendet, die auf den mehrere Werte breiten SIMD-Registern jeweils nur einzelne Werte verarbeiten und den Rest ignorieren, wie z.B. `vmulsd`.

Bei der Ausführung auf einem gemieteten Mac Mini M1 mit einem OpenJDK der Version 20 hingegen wurde das C2-Tier selbst bei kleinen Bildgrößen erreicht. Im vom M1-Chip verwendeten ARM-Instruktionsset lassen sich SIMD-Instruktionen an der Breite der verwendeten Register erkennen. Es ließen sich aber auch dort keine SIMD-Instruktionen im Maschinencode finden.

Unter Java gab es historisch keine native Kontrolle darüber, ob bei der Just-In-Time (JIT)-Compilation bestimmte Assembly-Instruktionen generiert werden, da keine Aussage über das verfügbare Set an Instruktionen getroffen werden kann - schließlich ist diese Hardwareunabhängigkeit ein großer Vorteil der Programmiersprache. Dank der zunehmenden Verfügbarkeit von SIMD-

oder Vektor-Instruktionen auf unterschiedlichen Hardware-Plattformen gibt es seit Anfang 2022 in Java 18 Support für den expliziten Einsatz von SIMD-Instruktionen sowohl auf 64-Bit-CPUs als auch auf aarch64-CPUs ((Sandoz 2023a), Sandoz (2023b) für das Release Date).

In ersten Benchmarks zeigt sich für den Einsatz von vektorisierter Addition ein Speedup um den Faktor 2 (siehe Abb. 8 aus Stypinski (2022a)), bei kombinierten Multiplikationen und Additionen bis zu Faktor 16 gegenüber unoptimiertem Code (siehe Abb. 9 aus Stypinski (2022b), Anmerkung: Diese Instruktion wird von der CPU des zum Disassembling verwendeten Computers noch nicht unterstützt). Der Speedup hängt allerdings von der Größe der verarbeiteten Arrays ab: Ab einer Größenordnung von  $10^7$  Elementen - einer realistischen Größe für Kameras, die in der Größenordnung von 40 Megapixeln Bilddaten liefern - bringt Vektorisierung in der Addition fast keinen Vorteil, und in der kombinierten Addition und Multiplikation nur noch ca. einen Faktor von 2. Es ist zu vermuten, dass bei der Verarbeitung von so großen Datenmengen die Speicherlatenz zum limitierenden Faktor wird.

Es gibt in den einzelnen Berechnungen, die im Rahmen des RAW-Processings durchgeführt werden, wie z.B. den Matrixmultiplikationen zur Berechnung der Farbraumtransformation, einige Stellen, an denen man Addition und Multiplikation kombinieren könnte. Die Verwendung der Java Vektor API macht den Code aber auch komplexer und kann die eigentliche Berechnung verschleiern (siehe Listing 1). Im Rahmen dieser Arbeit wurde daher noch auf die Verwendung der Java Vektor API verzichtet.

### 3.3.2 Multithreading/Streams

Während SIMD-Instruktionen Hardware-Parallelismus in einem einzelnen Kern nutzen, ermöglicht Multithreading das Aufteilen der Berechnung auf mehrere Prozessorkerne. Zusätzlich können Threads, wie bereits weiter oben erwähnt, auch nebenläufig ausgeführt werden und so durch Pipelining Lese- und Schreiblatenzen verborgen.

Dementsprechend werden die unterschiedlichen Berechnungsstränge, also Threads, in die sich ein Programm aufteilt, nicht zwingend gleichzeitig “im Lockschritt” ausgeführt. Das bedeutet aber auch, dass Threads letztendlich nicht ohne Weiteres Daten untereinander austauschen können. Um eine Berechnung aufzuteilen, müssen also vom Hauptthread nicht nur mehrere Threads gestartet werden, es muss auch gewartet werden, bis alle Threads mit ihrer Berechnung fertig sind, um fortfahren zu können.

Um Multithreading in Java manuell umzusetzen, gibt es das `Runnable`-Interface (Oracle (2023b)). Hier entspricht ein Thread der Instanz einer Klasse, welche die `run`-Methode implementiert.

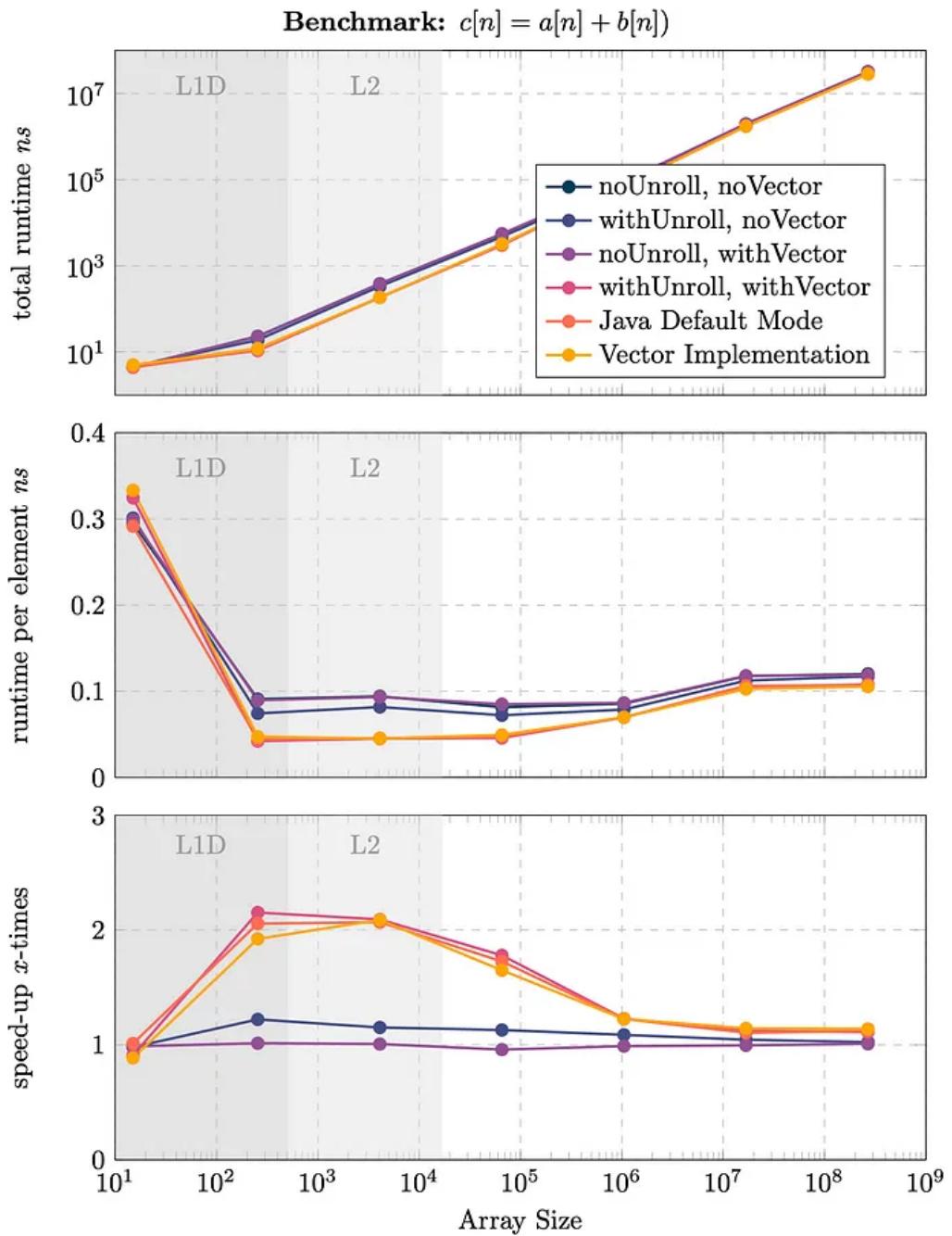


Abbildung 8: Performanz einer einfachen Addition von Werten in 2 Arrays mit unterschiedlichen Beschleunigungsstrategien, Quelle: Stypinski ([2022a](#))

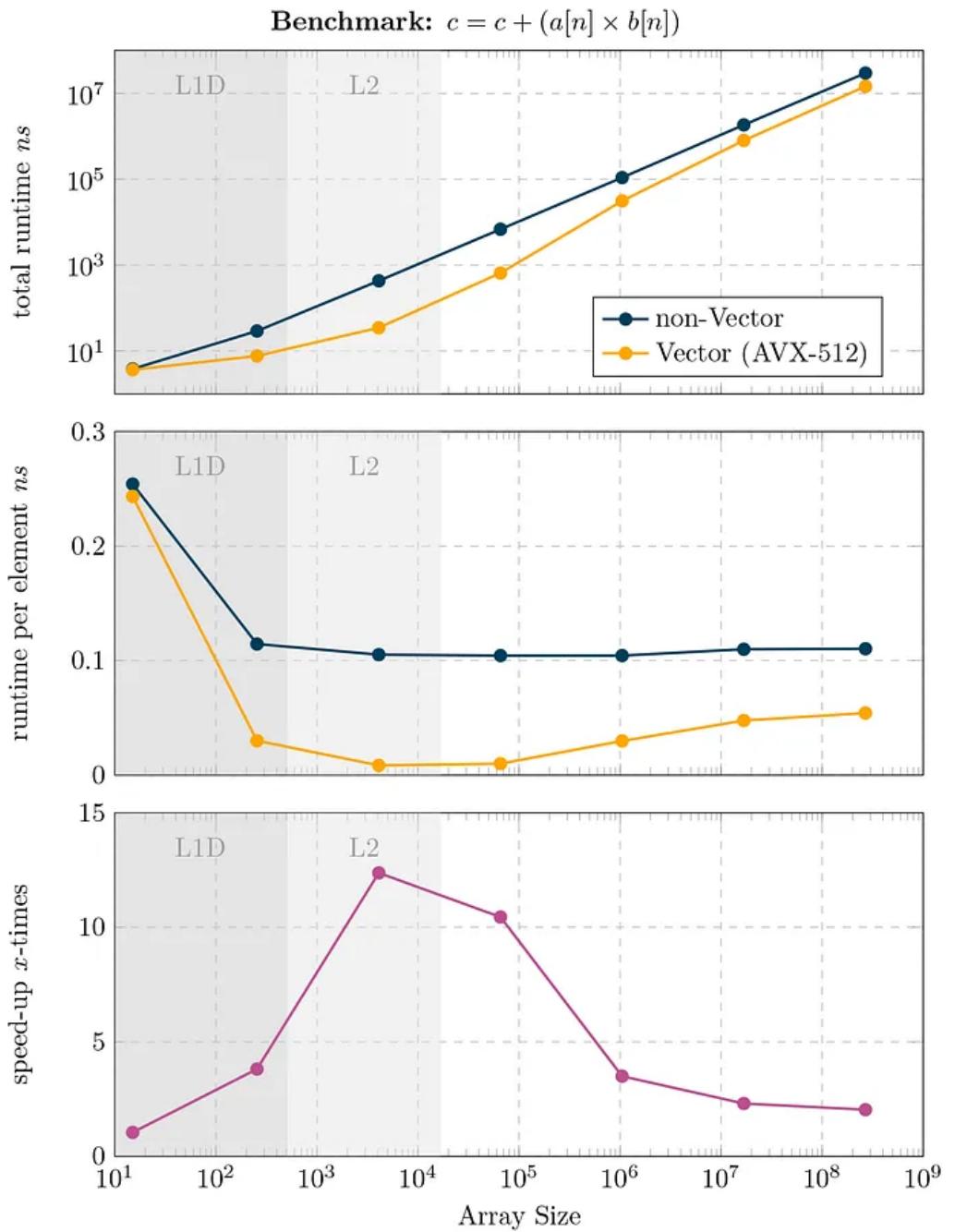


Abbildung 9: Performanz einer einfachen Addition mit Multiplikation mit unterschiedlichen Beschleunigungsstrategien, Quelle: Stypinski ([2022b](#))

```

1 // FMA: Fused Multiply Add: c = c + (a * b)
2 // Simple Version
3 public static float scalarFMA(float[] a, float[] b){
4     var c = 0.0f;
5
6     for(var i=0; i < a.length; i++){
7         c = Math.fma(a[i], b[i], c);
8     }
9     return c;
10}
11
12 // SIMD Version
13 private static final VectorSpecies<Float> SPECIES = FloatVector
14 .SPECIES_PREFERRED;
15
16 public static float vectorFMA(float[] a, float[] b){
17     var upperBound = SPECIES.loopBound(a.length);
18     var sum = FloatVector.zero(SPECIES);
19
20     var i = 0;
21     for (; i < upperBound; i += SPECIES.length()) {
22         // FloatVector va, vb, vc
23         var va = FloatVector.fromArray(SPECIES, a, i);
24         var vb = FloatVector.fromArray(SPECIES, b, i);
25         sum = va.fma(vb, sum);
26     }
27     var c = sum.reduceLanes(VectorOperators.ADD);
28
29     for (; i < a.length; i++) { // Cleanup loop
30         c += a[i] * b[i];
31     }
32     return c;
33 }
```

Listing 1: Vergleich unoptimierte und SIMD-Version kombinierte Addition und Multiplikation, Quelle: Stypinski (2022b)

Eine alternative Methode ist die Java Streams API (z.B. Oracle (2023a)). Einzig und allein durch den Aufruf der Methode `parallel()` auf einem Stream werden die in der Folge aufgerufenen Operationen auf mehrere Threads aufgeteilt. Die Auswahl der Anzahl an gestarteten Threads, die Verteilung der Arbeit auf die Threads und jegliche sonstige Threadverwaltung erfolgen automatisch. Wie in Listing 2 zu sehen, ist das Einbauen von Multithreading in Jeniffer2 somit sehr einfach.

```

1 // im Preprocessor vorher
2 for (int i = 0; i < samples.length; i++) {
3     samples[i] = (short) operation.process(samples[i] & 0xFFFF, i);
4 }
5 // Multithreading im Preprocessor
6 IntStream.rangeClosed(0, samples.length - 1).parallel().forEach
7     (i -> {
8     samples[i] = (short) operation.process(samples[i] & 0xFFFF, i);
9 });

```

Listing 2: Austausch von `for`-Schleifen durch IntStreams in Jeniffer2

In vereinfachenden Benchmarks<sup>1</sup> ergab sich kein signifikanter Unterschied in der Performanz von manuellem Multithreading und Multithreading via Streams. Beide Methoden bringen einen Performanzvorteil von etwas weniger als der Anzahl der verfügbaren Prozessorkerne mit sich. Dies ist aber auch zu erwarten, da Threads vom Betriebssystem gemanaged werden, was einen gewissen Ressourcenfußabdruck mit sich bringt. Außerdem beanspruchen Systemdienste wie die GUI eines Rechners ebenfalls Rechenressourcen, die vorher auf andere Kerne ausgelagert werden konnten.

Da der Einbau von Multithreading derart trivial und wirkungsvoll ist, wurde es u.a. in neuen rechenintensiven Demosaicing-Algorithmen wie RCD durch Reiter (2023) standardmäßig implementiert. In den folgenden Tests wurde Multithreading daher als Baseline verwendet, mit der die Beschleunigung auf der GPU verglichen werden muss.

### 3.4 Parallelismus auf der GPU

Während die CPU darauf optimiert ist, dutzende von Threads, die unterschiedliche Aufgaben haben, gleichzeitig möglichst schnell auszuführen, sind GPUs darauf ausgelegt, hunderte von Threads, die jeweils ähnliche oder gleiche Instruktionen beinhalten, gleichzeitig auszuführen. GPUs haben besonders viele Transistoren für Floating-Point-Operationen (siehe Abb. 10) und operieren zwar mit einer geringeren Frequenz, was sie aber durch den höheren Grad an Parallelisierung ausgleichen (NVIDIA Corporation & Affiliates (2023)). Außerdem

---

<sup>1</sup>Code auf Datenträger beigelegt

enthalten Grafikkarten speziellen, auf zweidimensionale Lokalität ausgelegten Texturspeicher und Hardware, welche Potenzoperationen auf Kosten der Genauigkeit deutlich beschleunigt (Lee et al. (2010)).

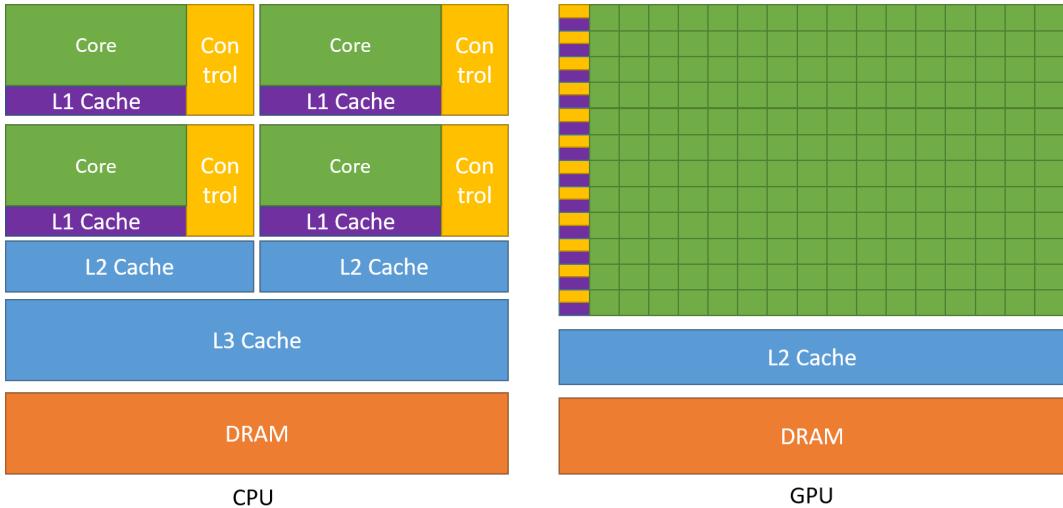


Abbildung 10: Unterschied der Transistoraufteilung CPU vs GPU (NVIDIA Corporation & Affiliates (2023))

Grundsätzlich bestehen moderne GPUs aus sogenannten “Streaming Multiprocessors” (NVidia) oder “Compute Units” (AMD), welche Blöcke an Threads untereinander aufteilen (Abb. 11). Diese Blöcke lassen sich wiederum in sogenannte Warps<sup>2</sup> aufteilen, deren Instruktionen ähnlich zu SIMD-Instruktionen im Gleichschritt ausgeführt werden. Wenn die Instruktionen eines Threads in einem Warp divergieren, z.B. wegen einer If-Abfrage, werden die jeweils inaktiven Threads pausiert. Um Latenzzeiten beim Datenzugriff zu verbergen, lässt der Warp-Scheduler auf dem Multiprozessor in der Zwischenzeit andere Warps laufen (NVIDIA Corporation & Affiliates (2023), TheBeard (2023)).

Die meisten Programmiermodelle für Grafikkarten gehen davon aus, dass ein Thread genau einen interessanten Datenpunkt (z.B. einen Pixel, oder einen Koordinatenpunkt) verarbeitet. Es werden dann so viele Threads gestartet wie Datenpunkte, und aus seiner Thread- und evtl. auch Block- und Warp-Id kann der Thread dann inferieren, auf welche Speicherstelle er zugreifen muss. Diese in vielen Threads gleichzeitig ausgeführten Programme werden auch Kernel genannt und sind meist nur wenige bis wenige dutzend Zeilen lang. Das liegt unter anderem auch daran, dass die Anzahl der verfügbaren Register begrenzt ist bzw. eine zu hohe Registerbelegung die Anzahl der gleichzeitig ausführbaren Threads verringert. Außerdem wirkt sich auch eine zu komplexe Logik mit zu

<sup>2</sup>“Thread” bedeutet übersetzt Faden, “Warp” heißt in der Übersetzung Kettfaden und spinnt somit die Metapher der Weberei weiter.



Abbildung 11: Aufteilung von Thread-Blöcken auf Streaming Multiprocessors (NVIDIA Corporation & Affiliates (2023))

vielen Verzweigungen negativ auf die Performance aus, da dann immer nur ein kleiner Anteil der Threads in einem Warp aktiv ist. Die Kernel werden meistens in einer eigenen domänen spezifischen Sprache geschrieben und zur Laufzeit vom Rahmenprogramm für die spezifische Grafikkarte des ausführenden Systems kompiliert.

Während sich auf mobilen und low-end-Geräten CPU und GPU den Hauptspeicher (RAM) teilen, haben dedizierte Grafikkarten meist ihren eigenen Speicher, welcher über einen BUS mit dem Hauptspeicher verbunden ist.

### 3.4.1 CUDA

Die vom Grafikkartenhersteller NVIDIA seit 2007 zur Verfügung gestellte CUDA-Bibliothek macht genau dieses eben beschriebene Modell vollständig transparent und bis ins kleinste Detail konfigurierbar. Außerdem gibt es graphische Profiling-Tools, welche die mit Datentransfer, Kernel-Kompilierung und Kernel-Ausführung verbrachte Zeit visualisieren (genutzt z.B. von Zapryanov und Nikolova (2019)). Deswegen wird CUDA in vielen Arbeiten zum Echtzeit-Demosaicing eingesetzt ( Wenninger (2012), Goorts, Rogmans, und Bekaert (2012), Faruqi, Ino, und Hagihara (2012), Langseth et al. (2014), Zapryanov and Nikolova (2019) ).

Im Kernel besteht Zugriff auf den aktuellen Thread-Index, aus dem sich berech-

nen lässt, für welche Daten - z.B. welches Array-Element - ein Thread zuständig ist. Da ein Thread auch für mehrere Elemente zuständig sein kann, lässt sich so z.B. auch Branching verringern, indem alle 4 Pixel eines Bayer-Mosaikbausteins auf einmal verarbeitet werden (Goorts, Rogmans, und Bekaert (2012)).

Da die Threads eines Warps über einen gemeinsamen Cache verfügen, auf den sie zugreifen können, können sie auch kooperativ Daten in ihren Speicher laden. Goorts, Rogmans, und Bekaert (2012) nutzen dies z.B., um die für einen 9x9-Box-Filter nötigen Datentransfers zu reduzieren.

Faruqi, Ino, und Hagihara (2012) nutzen die von CUDA zur Verfügung gestellte Kontrolle über das Scheduling, um Warps so in einer Welle (“Wavefront”) anzurufen, dass Datenabhängigkeiten zwischen den einzelnen Schritten ihres Algorithmus effizient genutzt werden.

Es lassen sich neben der Anzahl der gestarteten Threads auch die Threads per Block einstellen. Ein Pipelining des Speicherzugriffs und das Verhindern des Auslagerns von zwischen gemapptem Speicher aus dem RAM ist ebenfalls möglich (Harris (2012)).

Neben der Standard-C++-API gibt es auch eine Java-Schicht (jcuda.org (2022)), die es theoretisch ermöglichen würde, CUDA in JENIFFER2 zu verwenden. Allerdings erfordert die Verwendung von CUDA eine Grafikkarte des Herstellers NVIDIA.

### 3.4.2 OpenCL

Man könnte die zwei Jahre später veröffentlichte OpenCL (Open Computing Language) gewissermaßen als Open-Source-Antwort auf CUDA bezeichnen, denn das Programmiermodell ist ähnlich: Auch in OpenCL werden Kernel geschrieben und dann als eine festgelegte Anzahl an Threads in bestimmten Workgroups instanziert (TheBeard (2020)).

Die verfügbaren Entwicklerwerkzeuge für OpenCL sowie die feingranularen Optimierungsoptionen reichen nicht ganz an die von CUDA heran. Dafür bietet es eine etwas einfachere API, und - als wichtigster Vorteil - die Ausführung nicht nur auf Grafikkarten verschiedener Hersteller, sondern auch auf Mehrkern-CPUs und vielen anderen Hardwareplattformen (The Khronos® Group Inc. (2023)). Wang, Guo, und Wei (2019) nutzen OpenCL z.B., um Demosaicing auf in die CPU integrierten Grafikkarten (Intel HD Graphics) auszuführen und nutzen dabei aus, dass sich diese den Speicher mit der CPU teilen.

Als Teil der u.a. von populären Videospielen verwendeten Lightweight Java Game Library gibt es auch Java-Bindings für OpenCL, die sich an der originalen C++-API orientieren (Lightweight Java Game Library (2023)).

### 3.4.3 OpenGL

Im Gegensatz zu CUDA und OpenCL ist das deutlich ältere, seit 1992 entwickelte OpenGL-Framework nicht anwendungsagnostisch, sondern für das Rendering von 3D-Daten und generelle Grafikverarbeitung konzipiert. Programmierer müssen sich keine Gedanken um Implementationsdetails wie die Speicherverwaltung und Gruppierung von Threads machen, sondern bedienen die OpenGL-Pipeline (Abb. 12), welche z.T. auch in Hardware implementiert ist.

Als Input werden zunächst Geometriedaten in Form von normierten Koordinatenpunkten (Vertices) benötigt. Diese können von speziellen Kerneln, den Vertex-, Tesselations- und Geometrie-Shadern, modifiziert werden, und es können Informationen wie Texturkoordinaten hinzugefügt werden. Im folgenden werden die Geometriedaten u.a. in den Bildschirmraum transformiert und nicht sichtbare Punkte entfernt. Bei der Rasterisierung werden dann die Datenpunkte so interpoliert, dass es für jeden Zielpixel einen interpolierten Punkt auf der aufgespannten Geometrie gibt. Diese Pixelfragmente bekommen dann in einem weiteren Kernel, dem Fragment Shader, einen Farbwert zugewiesen. Neben den interpolierten Daten kann der Fragment Shader z.B. auch Zugriff auf eine oder mehrere Texturen bekommen.

Das Ergebnis eines Durchlaufs der OpenGL Pipeline muss nicht zwingend auf dem Bildschirm angezeigt werden, es kann auch in eine weitere Textur gespeichert werden, welche direkt heruntergeladen werden kann. Dem Fragment Shader stehen die interpolierten Texturkoordinaten zur Verfügung, er kann aber auf beliebige Stellen einer Textur zugreifen.

Auch OpenGL wird häufig zur Implementation von Echtzeitdemosaicing eingesetzt, unter anderem von McGuire (2008) bei der Implementation von Malvar, He, und Cutler (2004) oder von Brady (2016), welche in ihrer Pipeline auch das Warping und Stitching der Bilder von vielen Einzelkameras zu einem Panorama in OpenGL beschleunigen.

Da ein Dialekt von OpenGL auch auf der Grafikhardware von Smartphones läuft, wird OpenGL auch für experimentelle Demosaicing-Algorithmen, die besonders auf Mobilgeräten Sinn ergeben, eingesetzt, wie z.B. eine Superresolution durch die Kombination von mehreren leicht verschobenen Aufnahmen (Fung (2007), weiterentwickelt von Wronski et al. (2019) und jetzt auf Smartphones der Firma Google im Einsatz).

Lightweight Java Game Library (2023) bietet auch für OpenGL eine an der C++-API orientierte Java-Schnittstelle für OpenGL an. Dass sich die Java-Schnittstelle an der C++-API orientiert ist sehr hilfreich, da so die vielfältig im Internet verfügbaren Tutorials, welche meist auf C++ ausgelegt sind, trotzdem genutzt werden können (z.B. Vries (2022)).

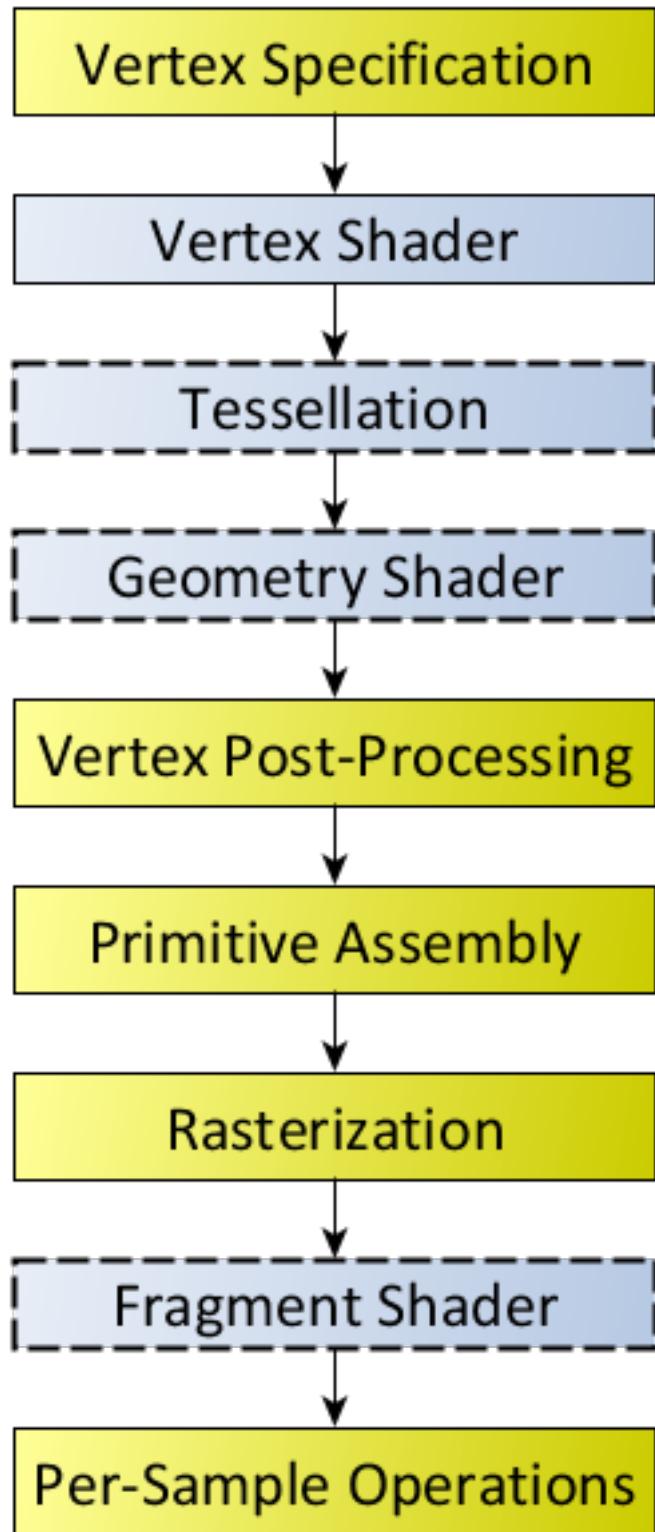


Abbildung 12: Die OpenGL Rendering Pipeline (programmierbare Teile in blau,  
Quelle: OpenGL Wiki contributors ([2022](#)))

### 3.4.4 TornadoVM

Ein relativ neues, ambitioniertes Projekt in aktiver Entwicklung ist TornadoVM (Fumero et al. (2019)): Ein Open-Source-Plugin für die JVM, welches Code auf Grafikkarten verschiedener Hersteller sowie integrierten GPUs und FPGAs ausführen kann.

Das besondere an TornadoVM ist, dass keine eigene Sprache wie CUDArt, OpenCL oder die GLSL (OpenGL Shader Language) zum Schreiben von Kernen erlernt werden muss, denn der auf der Grafikkarte ausgeführte Code wird in Java geschrieben. Es gibt neben dem Schreiben und Instanziieren von Kernen mit manueller Verwaltung der Anzahl der gestarteten Threads und Blockgrößen auch die Möglichkeit, in statischen Methoden `for`-Schleifen mit der `@Parallel`-Annotation zu versehen. Diese können dann in einem `TaskSchedule`, welcher auch die Übertragung der benötigten Daten spezifiziert, aneinandergereiht und ausgeführt werden (siehe Listing 3). Die `TaskSchedules` werden von TornadoVM in einen Graphen der Datenabhängigkeiten und daraus in ein eigenes Bytecodeformat übersetzt. Die VM orchestriert anhand dieser Informationen dann die Datenübertragung und führt den Code parallelisiert aus. Welche der verfügbaren CPUs und GPUs dafür ausgewählt wird, kann automatisiert oder nach bestimmten Optimierungsstrategien entschieden werden.

TornadoVM kann aus dem zugrundeliegenden Java-Bytecode sowohl OpenCL- als auch CUDA-Code und Code im der LLVM IR-ähnlichen Spir-V-Format produzieren.

## 3.5 Vergleich CPU/GPU

Es gibt also eine Vielfalt an Möglichkeiten, Code sowohl auf der CPU als auch auf der GPU parallelisiert auszuführen. Da in der Vergangenheit Paper öfter von einer Beschleunigung der Berechnung um den Faktor 50, 100 oder mehr auf der GPU gegenüber der CPU berichteten, verglichen Lee et al. (2010) die Ausführungszeit verschiedener Algorithmen gegeneinander, welche sie sowohl auf der CPU (mit Multithreading und SIMD-Instruktionen in C++) als auch auf der GPU (mit CUDA) optimiert hatten.

Als ein dem Demosaicing ähnlicher Algorithmus wurde eine Faltung mit festen Gewichten getestet. Ebenfalls einem Demosaicing-Algorithmus oder auch der Gammakorrektur ähnelte außerdem ein bilateraler, kantenerhaltender Glättungsfilter, dessen Implementation Potenzoperationen enthielt.

Während mit der Faltung nur eine gut zweifache Beschleunigung auf der GPU gegenüber optimiertem CPU-Code gelang, erreichte der bilaterale Filter eine mehr als fünffache Performanz. Als Grund dafür wird unter anderem die schnellere Hardwareimplementation von Potenzoperationen auf der GPU

```

1 public class Compute {
2     private static void mxmLoop(Matrix2DFloat A, Matrix2DFloat B,
3         Matrix2DFloat C, final int size) {
4         for (@Parallel int i = 0; i < size; i++) {
5             for (@Parallel int j = 0; j < size; j++) {
6                 float sum = 0.0f;
7                 for (int k = 0; k < size; k++) {
8                     sum += A.get(i, k) * B.get(k, j);
9                 }
10                C.set(i, j, sum);
11            }
12        }
13    }
14
15    public void run(Matrix2DFloat A, Matrix2DFloat B,
16        Matrix2DFloat C, final int size) {
17        TaskGraph taskGraph = new TaskGraph("s0")
18        // Transfer data from host to device only in the first
19        // execution
20        .transferToDevice(DataTransferMode.FIRST_EXECUTION, A, B)
21        // Each task points to an existing Java method
22        .task("t0", Compute::mxmLoop, A, B, C, size)
23        // Transfer data from device to host
24        .transferToHost(DataTransferMode.EVERY_EXECUTION, C);
25        // Create an immutable task-graph
26        ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot()
27        ;
28
29        // Create an execution plan from an immutable task-graph
30        TornadoExecutionPlan executionPlan = new
31        TornadoExecutionPlan(immutableTaskGraph);
32
33        // Execute the execution plan
34        TorandoExecutionResult executionResult = executionPlan.
35        execute();
36    }
37}

```

Listing 3: Codebeispiel TornadoVM, Quelle: TornadoVM Contributors ([2023](#))

genannt. Ein anderer Algorithmus zur Kollisionsdetektion erreichte den höchsten Beschleunigungsfaktor von 10 unter anderem durch die Nutzung des eingebauten Texturspeichers. Ob dieser für die Faltung ebenfalls verwendet wurde oder einen Vorteil gebracht hätte, wird nicht erwähnt. Grundsätzlich stehen diese Ergebnisse in Einklang mit dem tatsächlichen Leistungsunterschied in FLOPS (Floating-Point-Operationen pro Sekunde) zwischen der getesteten CPU und GPU (jeweils über alle Kerne summiert).

Zu beachten ist bei diesem Vergleich aber, dass es sich um eine Benchmark handelt. Nicht mit einbezogen ist somit, dass die CPU im normalen Betrieb eventuell stärker mit anderen zusätzlichen Aufgaben ausgelastet ist, wie z.B. in der praktischen Anwendung von Langseth et al. (2014) mit dem Encoding des verarbeiteten Videostreams.

## 4 Grundlagen Speicheroptimierung

Neben der Optimierung der Berechnungszeit von Algorithmen, die durch die Parallelisierung adressiert wird, ist auch eine Betrachtung des Speicherfußabdrucks wichtig.

Wie in Tabelle 2 zu sehen, dauert der Zugriff auf gespeicherte Daten ein Vielfaches der Zeit, die für eine einfache arithmetische Instruktion benötigt wird. Um diese Latenzen zu verbergen implementieren Prozessoren verschiedene Techniken:

Sofern ein Programm in seinem Datenzugriff einem vorhersagbaren Muster folgt, werden diese bereits geladen, bevor die entsprechenden Instruktionen ausgeführt werden (Pre-Fetching). Wie effektiv so die Latenz des Speicherzugriffs verborgen werden kann, hängt unter anderem auch davon ab, wie viel Berechnung für jeden gespeicherten Datenpunkt durchgeführt wird - wenn die Zeit für den Datenzugriff insgesamt länger dauert als die Berechnung, wird sie zum limitierenden Faktor der Performance.

Außerdem werden immer größere Caches mit einer niedrigeren Latenz zwischen CPU-Kerne und RAM gesetzt (siehe hierfür nochmals Abb. 10). Um von diesen zu profitieren, sollten Algorithmen das Set an Daten, welches wiederholt benötigt wird, möglichst klein halten. Es hilft auch, wenn Zwischenergebnisse zu dem Zeitpunkt, zu dem sie wieder gelesen werden, noch nicht aus dem Cache verdrängt worden sind.

	Intel i7-3770 (Ivy Bridge) (2012)		Apple M1 (2020) (kleine/große Kerne) <sup>1</sup>	
Speicher	Größe	Latenz	Größe	Latenz
L1 Instruction Cache	32 KB		128/192 KB	
L1 Data Cache	32 KB	4-5 Zyklen	64/128 KB	3-4 Zyklen
L2 Cache	256 KB	12 Zyklen	4/12 MB	18 Zyklen
L3 Cache	8 MB		8 MB	18 Zyklen + (10-15 ns) <sup>2</sup>
RAM	4 GB	30 Zyklen + 53 ns <sup>2</sup>	12 GB	18 Zyklen + 91 ns <sup>2</sup> <sup>3</sup>

<sup>1</sup>Enthält 4 "große"Kerne mit 3,2 GHz und 4 "kleine"Kerne mit ca 2 GHz

<sup>2</sup>Taktfrequenz >3 Ghz, also grob 3 Taktzyklen pro Nanosekunde <sup>3</sup> große Kerne

Tabelle 2: Speicherbausteine und Latenzen für einen dem Testgerät ähnlichen Prozessor und einen aktuellen Prozessor, Quelle: Pavlov (2023) (eigene Zusammenstellung)

## 5 Vorgehen

Im Folgenden werden die Entscheidungen, welche Beschleunigungsstrategien wie implementiert und getestet wurden, erläutert und das Vorgehen bei der Empirie beschrieben. Abschließend werden verworfene Ansätze dokumentiert.

### 5.1 Anforderungen an die Software

Wie bereits eingangs erwähnt, geht es in dieser Arbeit nicht nur um die reine Performance-Optimierung einer Anwendung. Jeniffer2 ist keine Software, die alltäglich von Profifotographen eingesetzt wird, und steht auch nicht in Konkurrenz zu anderen Open-Source-Projekten wie RawTherapee oder Darktable.

Jeniffer2 ist eine Lehrsoftware, die es ermöglicht, schnell und einfach verschiedene Demosaicing-Algorithmen auszuprobieren und zu vergleichen. Außerdem ist sie modular aufgebaut, wodurch für zukünftige Abschlussarbeiten leicht weitere Algorithmen oder DNG-Prozessschritte implementiert werden können. Als Anforderungen lassen sich also definieren:

- **Erhalt der Modularität & Erweiterbarkeit:** Prozessschritte im Demosaicing zu verstehen, zu verändern oder zu ergänzen, sollte durch klar abgegrenzte Schnittstellen und Verantwortlichkeiten auch weiterhin mit geringer Einarbeitung möglich sein.
- **Erhalt der Cross-Plattform-Kompatibilität:** Studierende sollten die Software auf ihrem eigenen Rechner installieren und ausführen können, unabhängig vom Betriebssystem oder Grafikkartenhersteller.
- **Optimierung der Effizienz:** Je schneller das RAW-Processing vonstatten geht, desto niederschwelliger lassen sich verschiedene Algorithmen testen. Eine Reduktion des Ressourcenverbrauchs sorgt außerdem dafür, dass das Programm auch auf älterer Hardware noch läuft, sodass auch weniger wohlhabenden Studierenden eine Teilhabe ermöglicht wird.

### 5.2 Auswahl der Frameworks

Anhand dieser Kriterien wurden aus den oben beschriebenen Optimierungsoptionen mehrere Punkte zur Implementation ausgewählt:

#### 5.2.1 CPU

Der zu erwartende Speedup aus dem Einsatz der Java Vector API ist bei einer Verarbeitung des ganzen Bilds sehr begrenzt, dafür kann die Verwendung der API das Programm weniger les- und wartbar machen.

Auch die Implementation von Multithreading über das Runnable-Interface würde das Verständnis des Programmflusses erschweren, ohne einen großen

Vorteil über die Implementation von Multithreading mit der Streams-API zu bieten.

Deswegen wurde für die Beschleunigung auf der CPU zunächst nur die Parallelisierung von Schleifen mithilfe der Streams-API implementiert. Bei der späteren algorithmischen Optimierung auf der CPU wurde durch die Verarbeitung in Kacheln zwar teilweise doch etwas Komplexität in den Programmfluss eingebracht, diese wurde aber erfolgreich vor der Implementation der Demosaicing-Algorithmen und anderen Operationen verborgen.

### 5.2.2 GPU

Das Kriterium der Cross-Plattform-Kompatibilität schließt den Einsatz von CUDA aus. Sowohl OpenCL als auch OpenGL benötigen ein fundiertes Grundlagenwissen über die Datenverarbeitung auf der Grafikkarte.

Die aus programmier-ergonomischer Sicht ideale Option wäre hier TornadoVM: Es muss keine neue Sprache gelernt werden, die Typsicherheit und der Editorsupport von Java ist gegeben, und aufgrund der verschiedenen Back-Ends wäre eine Implementation zukunftssicher für neue Technologien. Bei dem Versuch, einen Demosaicing-Algorithmus mithilfe von TornadoVM auf die Grafikkarte zu portieren, stellte sich allerdings heraus, dass das Projekt für diesen Fall noch nicht ganz anwendungsreif ist - Details werden weiter unten beschrieben. Außerdem ist das Framework inkompatibel mit dem vom Front-End verwendeten GUI-Framework.

Aufgrund der weiteren Verbreitung fiel die Wahl deshalb auf OpenGL. Ziel der Implementation war es in der Folge unter anderem, die Komplexität von OpenGL weitestgehend in einem abgeschlossenen Modul zu verbergen, sodass für eine neue Operation lediglich die Logik in der OpenGL Shader Language (GLSL) geschrieben werden muss.

## 5.3 Empirie

Die Tests zur Verifikation der Beschleunigungsstrategien teilen sich in zwei Abschnitte auf: Einen Feldtest mit 20 Teilnehmenden, und extensive Benchmarks, die auf dem ThinkPad W530 des Autors sowie zu einem Teil auch auf einem gemieteten Mac Mini M1 ausgeführt wurden.

### 5.3.1 Feldtest

Der Feldtest diente dazu, insbesondere die OpenGL-Implementierung auf heterogener Hardware in einem realistischen Setting zu testen. So können Aussagen zur plattformunabhängigen Lauffähigkeit und Effektivität getroffen werden.

Aufgrund zeitlicher Beschränkungen in der Probandenakquise wurde beispielhaft nur ein komplexer Demosaicing-Algorithmus, nämlich Ratio Corrected Demosaicing, in OpenGL umgesetzt.

Die Teilnehmenden erhielten Jeniffer2 in der GUI-Version und wurden angewiesen, zwei Testbilder (eines mit 3,3 Megapixeln und eines mit 47,4 Megapixeln) jeweils mit Multithreading und beiden Implementationsvarianten auf der GPU zu “entwickeln”. Bei der Ausführung auf den privaten Rechnern der Teilnehmenden wurden hierbei keine Vorgaben zu anderen gleichzeitig laufenden Programmen gemacht, um ein realistisches Setting abzubilden. Danach sollten sie die dabei generierten CSV-Logdateien sowie, falls zutreffend, eine Beschreibung aufgetretener Fehler übermitteln. Diese enthalten detaillierte Timing-Daten zu den einzelnen Verarbeitungsschritten und Debugging-Informationen zur verwendeten Hardware.

Die Ergebnisse dieses Feldtests übertrafen, wie in Abschnitt 7.2 beschrieben, die anhand der Literatur zu erwartende Beschleunigung in der Berechnung. Andererseits traten unerwartete Kompatibilitätsprobleme unter anderem mit OpenGL auf Apple-Silicon-Prozessoren zu Tage. Beides motivierte die tiefergreifende Auseinandersetzung mit der algorithmischen Optimierung des Programms auf der CPU.

### 5.3.2 Benchmarks

Die Optimierungen auf der CPU im plattformunabhängigen Java-Code wurden nach ihrer Umsetzung in lokalen Benchmarks getestet. Neben dem Vergleich mit der alten Implementierung dienen diese Tests auch dem Erkunden der besten Konfiguration anhand verschiedener Parameter, die sich bei der Implementation auftaten. Hierbei wurden alle Parameterkombinationen mit den beiden Testbildern aus dem Feldtest sowie einem Bild mit 60,5 Megapixeln mindestens 5 mal getestet.

Hierbei ist zu erwähnen, dass das verwendete Testgerät des Autors zwar ein High-End-Laptop, aber bereits 10 Jahre alt ist. Da die Effektivität mancher Optimierungen von der Anzahl der verfügbaren Prozessorkerne sowie den CPU-Cache-Größen abhängt, wurde deshalb zusätzlich ein Mac Mini Baujahr 2020 mit Apple M1-Prozessor angemietet. Auf diesem wurde zur Verifikation die neue Variante des Programms ausgeführt, die alte Vergleichsversion war dort leider nicht lauffähig.

## 5.4 Verworfene Ansätze

### 5.4.1 Caching und Abbildungstabellen

In der Anfangsphase dieser Arbeit kam die Idee auf, Berechnungsergebnisse zu cachen. Lookup-Tables können im DNG-Format in den Metadaten zur Verarbeitung der monochromen Sensordaten eingebettet werden, insofern ist die Verwendung einer abgewandelten Version für die weiteren Berechnungsschritte eine plausible Idee.

Bei der Neuentwicklung von Jeniffer2 fiel auf, dass die Gammakorrektur, welche im Rahmen der Farbraumtransformation stattfindet, die zeitintensivste Operation ist (Ljavin (2020), Abb. 13). Diese ist letztendlich nur eine abgewandelte Potenzoperation:

$$\begin{aligned} R'_{sRGB} &= \begin{cases} 12.92 \cdot R_{sRGB} & \text{für } R_{sRGB} \leq 0.00304 \\ 1.055 \cdot R_{sRGB}^{(1.0/2.4)} - 0.005 & \text{anderenfalls} \end{cases} \\ G'_{sRGB} &= \begin{cases} 12.92 \cdot G_{sRGB} & \text{für } G_{sRGB} \leq 0.00304 \\ 1.055 \cdot G_{sRGB}^{(1.0/2.4)} - 0.005 & \text{anderenfalls} \end{cases} \\ B'_{sRGB} &= \begin{cases} 12.92 \cdot B_{sRGB} & \text{für } B_{sRGB} \leq 0.00304 \\ 1.055 \cdot B_{sRGB}^{(1.0/2.4)} - 0.005 & \text{anderenfalls} \end{cases} \end{aligned} \quad (3.13)$$

Abbildung 13: Formel für die Gammakorrektur laut Ljavin (2020)

Im DNG-Format werden zurzeit maximal 16 Bit pro Farbe gespeichert. Eine Abbildungstabelle von jedem Farbwert auf sein umgewandeltes Ergebnis muss also nur  $2^{16}=65535$  Werte abdecken, nimmt also nur gut 131 Kilobyte an Platz ein. Das passt bei modernen Prozessoren in den L1-Cache, und selbst bei alten Modellen zumindest in den L2-Cache (Danowitz et al. (2012), Pavlov (2023)) - was einen Speicherzugriff dementsprechend potentiell schneller macht als die Berechnung der Potenz, welche hunderte Taktzyklen benötigt<sup>3</sup>. Konsequenterweise bietet sich die Vorberechnung und Verwendung eines Lookup-Tables an, was in einem Standalone-Experiment (siehe Codeausschnitt in Listing 4) für eine 12-bis 14-fache Beschleunigung sorgte.

In Jeniffer2 wird aber bereits das Zwischenergebnis der vorherigen Berechnung als 32-Bit-Integer weitergegeben. Das alleine sollte technisch noch lösbar sein,

---

<sup>3</sup><https://e2e.ti.com/support/processors-group/processors/f/processors-forum/258204/c674x-clocks-for-math-operations-sqrt-and-pow>

```

1 // ## pre-computed lookup table
2 private static char[] lookupGamma(char[] in, char[] lut) {
3     char[] out = new char[in.length];
4     for(int i = 0; i < in.length; i++) {
5         out[i] = lut[in[i]];
6     }
7     return out;
8 }
9 private static char[] computeLookupTable() {
10    char[] input = new char[Character.MAX_VALUE];
11    for(int i = 0; i < Character.MAX_VALUE; i++) {
12        input[i] = (char)i;
13    }
14    return naiveGamma(input);
15 }
```

Listing 4: Naive Lookup-Table Implementation der Gammakorrektur

da diese Integer auf einem Short-Buffer mit 16Bit basieren und im gleichen Wertebereich bleiben.

Problematisch ist, dass bei den meisten Bildern die Berechnung der Gammakorrektur im Rahmen der Farbraumkonvertierung von XD50 nach sRGB stattfindet. Hierbei beeinflussen sich die Farbkanäle gegenseitig, was den Wertebereich um den Faktor  $2^{16} \cdot 2^{16}$  vervielfacht. Wenn man die Gammakorrektur als einzelne Operation extrahiert, geht außerdem Präzision verloren, da die Zwischenergebnisse als Floating-Point-Werte (hier 64 Bit) gespeichert werden. Bereits ein Quell-Wertebereich, der alle möglichen 32-Bit-Floating-Point-Werte abdeckt, würde in einer Lookup-Tabelle resultieren, die um mehrere Größenordnungen mehr Speicher braucht als das eigentliche Bild.

Deswegen schied ein Caching von Berechnungsergebnissen, auch wenn es isoliert betrachtet vielversprechend scheint, schnell als Option aus. Ein reverse-Caching, das ein Array an Werten speichert, die die Untergrenze des auf den jeweiligen Index abgebildeten Wertebereichs darstellen, benötigt  $16 (\log_2(2^{16}))$  Vergleiche und brauchte in der Implementation damit sogar etwas länger als die reguläre Potenzoperation.

#### 5.4.2 TornadoVM

Bei der Testweisen Umsetzung eines Algorithmus mithilfe von TornadoVM stellte sich zunächst heraus, dass die nötigen Treiber auf dem ThinkPad W530 des Autors nur unter Windows verfügbar sind. Dementsprechend wurde auf der Windows-Partition getestet, obwohl Windows offiziell nur experimentell unterstützt wird.

Als zu testender Algorithmus wurde Ratio Corrected Demosaicing ausgewählt,

weil dieser Algorithmus zu den am längsten Laufenden zählt. Als erster Schritt musste die Implementation in eine statische Methode übersetzt werden, welche auf Arrays operiert. Dafür wurde sie aus dem existierenden pixelweisen Operations-Interface gelöst und stattdessen in eine neue Klasse verschoben, die ein ganzes Bild auf einmal verarbeitet. Allein diese Linearisierung des Codes und die Übertragung der Kontrolle über den Programmfluss an den Algorithmus beschleunigten die Ausführungszeiten deutlich, was unter anderem auch die weiter unten im Detail beschriebenen Optimierungen auf der CPU inspirierte.

Um die resultierende statische Methode mittels TornadoVM parallelisiert auszuführen, wird sie als Knoten in einem sogenannten “Task Graph” hinzugefügt. Ein erster Versuch, den Algorithmus so auszuführen, schlug fehl: Da anscheinend ein Knoten in einen (OpenCL) Kernel übersetzt wird, reichte der Programmspeicherplatz dafür nicht aus. Dementsprechend wurde die Berechnung jedes im Abschnitt der Datenflussanalyse genauer beschriebenen Zwischenergebnisses in eine eigene statische Methode ausgelagert.

Bei der Ausführung dieser Variante fiel ein Bug in der OpenCL-Code-Generation auf: In der zum Zeitpunkt der Umsetzung verfügbaren Version sorgten “Oder”-Verknüpfungen in `if`-Bedingungen für die Generation von Syntaxfehlern. Der Autor des Frameworks reagierte hier sehr schnell und lieferte eine Identifikation des Fehlers sowie einen Workaround (<https://github.com/beehive-lab/TornadoVM/issues/247>). Die Existenz solcher Fehler zeigt dennoch, dass es sich um ein Forschungsprojekt und keine Software für den produktiven Betrieb handelt.

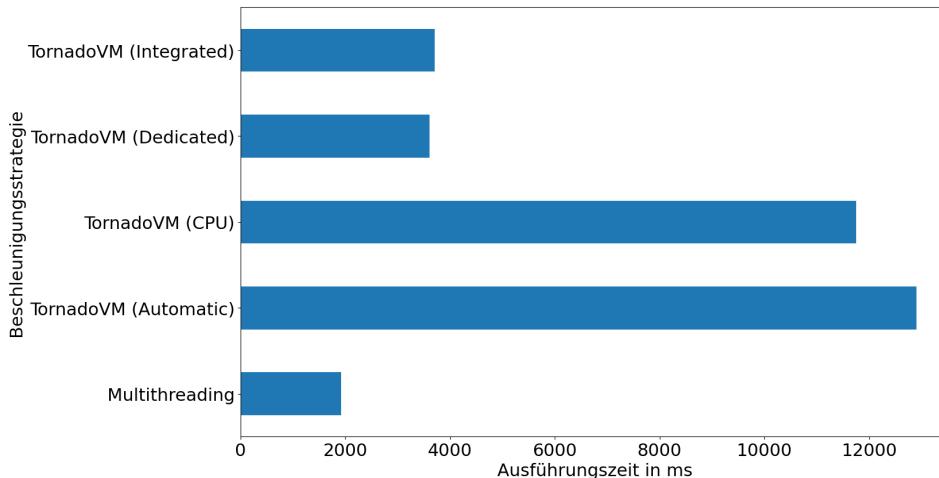


Abbildung 14: Vergleich der Ausführungszeiten des RCD-Demosaicing mit TornadoVM und Multithreading

Wie in Abb. 14 zu sehen, konnte die Verwendung von TornadoVM die Berechnung nicht beschleunigen. Interessant ist, dass durch die explizite Festlegung des zu verwendenden Geräts Zeit gespart werden kann, und automatisch wahrscheinlich nicht die schnellste Option, sondern die CPU ausgewählt wird. Die Verarbeitung konnte außerdem nur mit dem kleineren (3,3 Megapixel großen) Testbild getestet werden, da bei dem größeren (47,4 Megapixel großen) Testbild TornadoVM der Speicher (4GB Heap) ausging.

Eine Untersuchung mithilfe der in TornadoVM eingebauten Profiling-Tools (Abb. 15) zeigt, dass tatsächlich auch ein Großteil der verbrauchten Zeit für die eigentliche Berechnung aufgewendet wird. Hierbei braucht vor allem die Berechnung des XY-Gradienten viel Zeit. Für diese Berechnung werden 9 Pixel lange Zeilen und Spalten aus dem Bild miteinander korreliert und mit einer Matrix an Gewichten multipliziert. Schwer zu erklären ist, dass die Berechnung der PQ-Gradienten, welcher die gleiche Berechnung mit diagonalen Spalten durchführt, deutlich weniger Zeit benötigt. Bei der Inspektion des generierten OpenCL-Codes fällt jedoch auf, dass dieser deutlich länger ist als die manuelle Übersetzung in OpenGL, und weniger Operationen zusammenfasst.

```
{
    "TOTAL_BYTE_CODE_GENERATION": "12Mb",
    "TOTAL_KERNEL_TIME": "6836.5ms",
    "COPY_IN_TIME": "9.7ms",
    "TOTAL_DISPATCH_DATA_TRANSFERS_TIME": "6.6ms",
    "TOTAL_DISPATCH_KERNEL_TIME": "3.2ms",
    "TOTAL_TASK_GRAPH_TIME": "8129.7ms",
    "TOTAL_GRAAL_COMPILE_TIME": "468.2ms",
    "TOTAL_DRIVER_COMPILE_TIME": "518.2ms",
    "TOTAL_CODE_GENERATION_TIME": "48.8ms",
    "COPY_OUT_TIME": "14.0ms",
    "TOTAL_COPY_IN_SIZE_BYTES": "40Mb",
    "TOTAL_COPY_OUT_SIZE_BYTES": "40Mb"
}
```

Abbildung 15: Ausschnitt TornadoVM Profiler-Ergebnis auf der integrierten Grafikkarte

Die Tests wurden mit der Kommandozeilenvariante von Jeniffer2 ausgeführt. Auch wenn sich das Problem der langen Berechnung für eine spezifische Operation vielleicht durch genaueres Nachforschen noch hätte lösen lassen, stellte sich beim Versuch, die Version mit graphischer Benutzeroberfläche auszuführen, heraus, dass TornadoVM inkompatibel mit dem verwendeten GUI-Framework ist. Zwar lässt sich das TornadoVM-Plugin mit unterschiedlichen Java Development Kits kompilieren, intern wird aber der GraalVM-Compiler verwendet, um nativen Code zu erzeugen. Warum genau sich die Benutzeroberfläche damit

nicht kompilieren lässt, wird im nächsten Abschnitt im Detail beschrieben. Ein Austausch des verwendeten Frameworks ist jedenfalls nicht trivial und im Rahmen dieser Arbeit nicht mehr zu leisten gewesen.

## 6 Implementierung

Sowohl um die lokalen Benchmarks als auch die Empirie mit einer größeren Stichprobe zu ermöglichen, wurden ein paar mundäne Änderungen an Jeniffer2 vorgenommen, die hier der Vollständigkeit halber dokumentiert werden: Es wurde ein Kommandozeilen-Modul eingeführt, sowie strukturiertes Logging und Regressionstests. Die Erfahrungen bei der Cross-Plattform-Distribution der JavaFX-basierten graphischen App sind ebenfalls möglicherweise für andere Projekte interessant.

Bei der Implementation der Demosaicing-Algorithmen auf der Grafikkarte mit OpenGL offenbarte sich u.a. das nicht-triviale Problem der maximalen Texturgröße und des begrenzten Arbeitsspeichers. Bei der Performance-Optimierung auf der CPU hingegen liegt der Fokus weniger auf technischen Details als auf einer Datenflussanalyse der komplexeren Demosaicing-Algorithmen sowie Überlegungen zur Verwendung der Caches.

Abschließend wird auf die Änderungen an der graphischen Benutzeroberfläche eingegangen, die unter anderem durch die schnellere Verarbeitung der Bilder ermöglicht wurde.

### 6.1 CLI-Modul

Wie weiter unten ausführlicher beschrieben, gab es bei der Distribution von Jeniffer2 zwischenzeitlich Probleme mit dem JavaFX-basierten GUI-Framework. Um unterscheiden zu können, ob Fehler im GUI-Modul oder bereits im DNG-Prozessor entstehen, wurde eine kleine Kommandozeilenanwendung eingeführt, welche die Funktionalität des DNG-Prozessors ebenfalls abbildet. Zusätzlich zum in der graphischen Anwendung zur Verfügung gestellten Funktionsumfang kann je nach Bedarf die Logging-Strategie gewechselt werden.

Das CLI-Modul ist außerdem essentiell für Tests: Für die Performance-Tests können so hunderte Durchläufe mit verschiedenen Parametern geskriptet ausgeführt werden. Außerdem wurde die Möglichkeit eingebaut, nur den Demosaicing-Schritt auf einer PNG-Datei auszuführen. Dies ist ein Bestandteil der Reproduktion der Qualitätsmetriken aus Reiter (2023) mit verschiedenen Datensätzen.

### 6.2 Logging

Um die Performanz der einzelnen Verarbeitungsschritte nicht nur messen, sondern auch auswerten zu können, ist ein gutes Logging notwendig. Während für kleine Proof-Of-Concept-Tests die Ausgabe der benötigten Zeit auf die Konsole ausreicht, ist das Loggen in eine Datei in einem strukturierten Format unerlässlich für das Zusammenführen von Daten unterschiedlicher Testpersonen oder das Benchmarking unterschiedlicher Konfigurationen in mehreren Durchläufen.

Hierbei fiel die Entscheidung auf das CSV-Format, welches sich z.B. in relationale Datenbanken einlesen oder mit Python-Statistik-Frameworks wie Pandas (The pandas development team (2023)) verarbeiten lässt. Je nach benötigtem Anwendungsfall wurden verschiedene Logging-Klassen implementiert.

### 6.2.1 Interface

Ein **Timer**, wie das Logging-Interface genannt wird, muss im Prinzip nur zwei Dinge verwalten: Welches Bild gerade verarbeitet wird, und welche Schritte gerade ausgeführt werden. Es kann immer nur ein Bild gleichzeitig verarbeitet werden, Meta-Informationen zu diesem Bild werden dann jeder geloggten Zeile beigefügt.

Es können aber, je nachdem, wie feingranular man loggt, mehrere Verarbeitungsschritte gleichzeitig stattfinden. Deswegen wird den entsprechenden Methoden `startTask` und `endTask` jeweils der Name des Schritts (Tasks) mitgegeben, anhand dessen die gespeicherte Startzeit identifiziert wird. So kann die Zeit auch für mehrere Schritte, die sich überlappen, gemessen werden. Wurde vergessen, `endTask` aufzurufen, so wird der Schritt mit dem Ende des aktuellen Durchlaufs (`endRun`) abgeschlossen.

### 6.2.2 Systeminformationen

Um später identifizieren zu können, welche Testläufe auf dem gleichen Gerät ausgeführt wurden, müssen beim Logging auch Informationen über die ausführende Hardware erhoben werden. Generell sind solche Meta-Informationen für die Empirie und zum Debugging nützlich.

Mithilfe der OSHI(Operating System & Hardware Information, OSHI Contributors (2023))-Bibliothek kann quasi-nativ auf verschiedenste Hardware-Informationen zugegriffen werden. Um eine eindeutige System-Id zu erstellen, wird die Hardware-UUID oder, falls diese nicht vorhanden oder zugreifbar ist, die Prozessor-ID verwendet. Zusätzlich werden Informationen zur CPU-Frequenz, Hersteller, Mikroarchitektur und verfügbaren Kernen sowie der verfügbare Arbeitsspeicher erhoben. Die später beschriebene Wrapper-Klasse für die Operationen auf der Grafikkarte erfasst außerdem Informationen u.a. zur OpenGL-Version, dem verfügbaren Grafikspeicher und dem Hersteller der Grafikkarte.

Da diese Informationen über die Programmausführung stabil bleiben, werden sie nicht mit dem Abschluss jedes Verarbeitungsschritts geloggt, sondern nur einmal in eine eigene Datei geschrieben. Um sie trotzdem zuordnen zu können, wird in beiden Dateien in jeder Zeile die System-ID geloggt.

## 6.3 Regressionstests

Da die Performance-Optimierung sowohl auf der Grafikkarte als auch auf der CPU das Ergebnis ja nicht verändern soll, und um spätere Änderungen zu erleichtern, wurden Regressionstests für die einzelnen DNG-Verarbeitungsschritte eingeführt.

Hierfür wurde eine Helper-Funktion geschaffen, welche ein Input-Bild aus einer PNG-Datei einliest, mit einem Demosaicing- oder Postprocessor verarbeitet und mit dem gewünschten Ergebnis aus einer PNG-Datei vergleicht.

Für die Konfiguration der Prozessoren werden zwar zum Teil trotzdem noch Meta-Informationen aus der DNG-Datei benötigt. Der Vergleich mit PNG-Dateien verhindert jedoch, dass für einen Test eine bestimmte Referenzimplementation gespeichert und immer wieder ausgeführt werden muss. Außerdem kann der Helper auch das tatsächliche Ergebnis als PNG-Datei abspeichern, was bei der Erstellung von Tests z.B. von neuen Demosaicing-Algorithmen hilfreich ist.

## 6.4 Distribution

Bislang wurde Jeniffer2 immer als Java-Archiv, also .jar-Datei mit Java-Bytecode und Ressourcen verteilt. Eine solche Datei benötigt nur eine auf dem Zielrechner installierte Java-Laufzeitumgebung (JRE) und kann von der Kommandozeile mit `java -jar Jeniffer2.jar` gestartet werden. Bereits vor der Einführung von Java-Bibliotheken mit nativen Komponenten wie OSHI zur Systeminformation enthielt das Archiv Bibliotheken für unterschiedliche Betriebssysteme: Da das Front-End-Framework JavaFX nicht mehr standardmäßig Teil der Java-Laufzeitumgebung (JRE) ist, müssen diese vom OpenJFX-Projekt (Oracle Corporation (2023b)) heruntergeladen und inkludiert werden. Das Vorgehen, zur Distribution Abhängigkeiten für unterschiedliche Betriebssysteme in einer Datei zu verpacken, ist umgangssprachlich auch als “Fat Jar” bekannt.

Für das Elternpaket der betriebssystemspezifischen Komponenten von OpenJFX schlägt der Buildprozess mithilfe von Apache Maven (The Apache Software Foundation (2023a)) allerdings fehl. Dies liegt daran, dass die Verantwortlichen in den Konfigurationsdateien ihrer Pakete für unterschiedliche Betriebssysteme die gleiche Id verwenden.<sup>4</sup>

Für die meisten Systeme war dies kein Problem, beim Test auf neueren Apple-Geräten konnte Jeniffer2 allerdings die mitgelieferten JavaFX-Komponenten

---

<sup>4</sup>Das Problem ist seit längerem bekannt, wurde aber in der internen Kommunikation wohl noch nicht weitergegeben: <https://bugs.openjdk.org/browse/JDK-8305167?jql=text%20~%20%22duplicate%20profile%20id%22%20ORDER%20BY%20lastViewed%20DESC>

nicht finden. Wegen dieser Probleme wurden verschiedene Arten der plattformspezifischen Distribution erkundet.

#### 6.4.1 Java-Modulsystem

Seit Java 9 gibt es abgesehen vom existierenden Classpath-Ansatz für Java ein Modulsystem. Dadurch, dass Bibliotheken und Dienste über eine `module-info.java`-Datei explizit festlegen, welche Methoden von außen zugreifbar sein dürfen, soll die Sicherheit und Stabilität verbessert werden. Außerdem sind auch alle nativen Java-Pakete als Module verfügbar. Dies ermöglicht eine bessere Eliminierung von “totem”, nicht verwendetem Code und ist die Voraussetzung für nativ ausführbare Dateien, die z.B. mit dem im JDK zur Verfügung gestellten `jlink`-Tool<sup>5</sup> oder dem GraalVM-Compilerprojekt (Oracle Corporation (2023a)) erstellt werden können.

Die Umstellung von Paketen auf Module ist momentan noch immer im Gange. Die verwendete Apache Commons Math-Bibliothek (The Apache Software Foundation (2023b)) bietet z.B. noch keine Distribution im Modul-Format an, sie muss als sogenanntes automatisches Modul importiert werden, was von `jlink` nicht unterstützt wird. Es gibt zwar bereits verschiedene Plug-Ins, die automatische Module in reguläre Module umwandeln, mit diesen wurde aber kein nachhaltiger Erfolg erzielt.

#### 6.4.2 Reflektion

Der GraalVM `native-image`-Compiler hingegen kann auch Java-Projekte mit automatischen Modulen zu nativen ausführbaren Dateien kompilieren. Im Gegensatz zu `jlink`, welches letztendlich nur eine auf das Projekt zugeschnittene Java-Laufzeitumgebung in die Datei integriert, führt `native-image` eine echte Ahead-Of-Time-Compilation (AoT-Compilation) durch.

Features wie Reflektion, also die Code-Selbst-Inspektion zur Laufzeit, und Zugriff auf native Funktionen müssen bei der AoT-Compilation explizit für die Klassen und Methoden, in denen sie verwendet werden, freigeschaltet werden. Die Dokumentation des der graphischen Benutzeroberfläche zugrundeliegenden Afterburner.FX-Frameworks (Bien (2016)) ist hierzu leider wenig aussagekräftig. Zwar stellt GraalVM Werkzeuge zur Verfügung, die benötigten Informationen durch Profiling zu ermitteln, mit diesen konnte aber kein nachhaltiger Erfolg erzielt werden. Grundsätzlich ist ein Austausch des schon seit Jahren nicht mehr weiterentwickelten Afterburner.FX-Frameworks zwar sinnvoll, dies war im Umfang dieser Arbeit aber nicht zu leisten.

---

<sup>5</sup><https://docs.oracle.com/en/java/javase/11/tools/jlink.html>

### 6.4.3 Azul Zulu und Warp-Packer

Die Lösung für das JavaFX-Problem war letztendlich, eine Distribution der Java-Laufzeitumgebung mit eingebautem JavaFX-Framework zu installieren, wie z.B. Azul Zulu (Azul Systems ([2023](#))). Generell hat die Mitlieferung des JRE in der ausgelieferten Datei den Vorteil, dass die Fehlerquelle, dass etwas mit einem bestimmten JRE nicht funktioniert wird, zumindest auf eine Version beschränkt wird.

Das kleine **Warp-Packer**-Tool (Giagio ([2019](#))) ermöglicht es, beliebige Dateien in ein selbstextrahierendes Archiv zu verpacken, welches nach dem extrahieren ein Skript ausführt, das z.B. eine Java-Anwendung startet. Diese layenfreundliche Distribution als eine einzelne ausführbare Datei koexistiert aktuell mit der leichtgewichtigeren Jar-Distribution.

## 6.5 Verarbeitungsschritte auf der Grafikkarte

Ein wichtiger Baustein von Jeniffer2 ist die generische `Pipeline`-Klasse, die eine Liste von Prozessor-Objekten zusammenfügt. Diese stellen dann eine Methode zur Verfügung, um Daten eines bestimmten Typs zu verarbeiten, und ein Objekt vom gleichen Typ zurückzugeben.

In der `DNGProcessor`-Klasse<sup>6</sup> wird dann je nach den im Bild enthaltenen Meta-Informationen die Pipeline zusammengebaut, und die Prozessor-Klassen der einzelnen Schritte werden mit den extrahierten Parametern instanziert. Der Datentyp, der zwischen den Prozessor-Operationen weitergereicht wird, ist hier ein `BufferedImage` (siehe Abb. 16 links).

Dieses Interface ist aber ungeeignet, um Daten zwischen Operationen, die auf der GPU ausgeführt werden, weiterzureichen, denn der Up- und Download der Daten aus und in ein `BufferedImage` ist sehr zeitintensiv. Deswegen wurde eine zweite `gpuPipeline` eingeführt, die eine entsprechende Wrapperklasse verwendet (siehe Abb. 16 rechts).

Die PreProcessor-Operationen verwenden einen LookUp-Table, was für eine komplexe Datenübergabe sorgen würde, und wurden deshalb zunächst nicht auf der GPU implementiert. Dementsprechend wird erst nach dem PreProcessor zur GPUPipeline gewechselt. Der Bildzuschnitt, der ursprünglich zwischen Demosaicing und Postprocessing stattfand, profitiert ebenfalls nicht von der Portierung auf die GPU und lässt sich ohne die Korrektheit der Berechnung zu gefährden ans Ende der Pipeline stellen.

---

<sup>6</sup>ehemals `Facade` genannt, nach dem Architekturmuster, das sie umsetzt

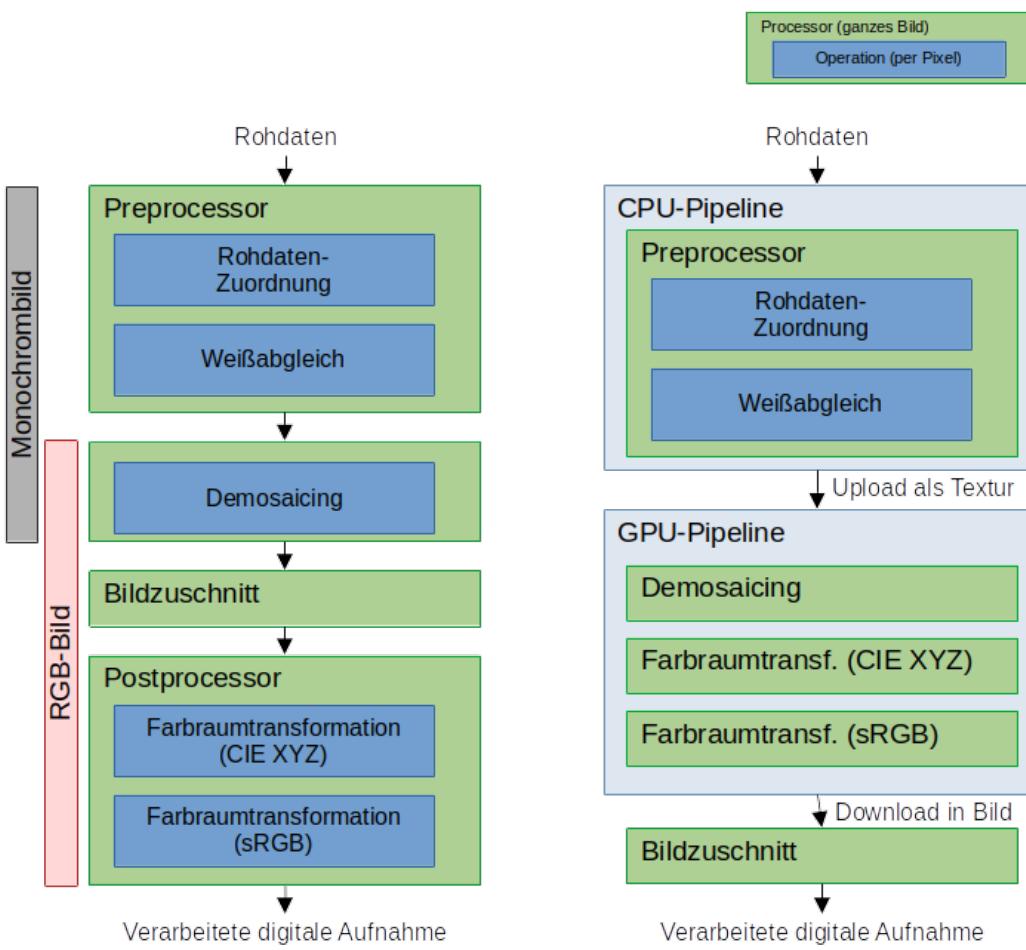


Abbildung 16: Vergleich DNG-Processing auf der CPU und auf der GPU

### 6.5.1 Grafikkontext

Um die OpenGL-API nutzen zu können, benötigt ein Programm einen sogenannten OpenGL-Kontext. Dieser wird Programmen zur Verfügung gestellt, die ein grafisches Fenster haben. Da Jeniffer2 in ein GUI-Modul und eine GUI-lose DNG-Verarbeitungsbibliothek aufgeteilt ist, steht den für die Bildverarbeitung zuständigen Klassen kein Fenster zur Verfügung. Um diese Limitation zu umgehen, wurde das `OpenGLContext`-Singleton-Objekt, welches ein für den Endnutzer unsichtbares Fenster öffnet und vorhält, eingeführt.

Um die Ressourcen freizugeben und Konflikte mit dem GUI zu verhindern, stellt dieses Objekt auch eine `delete`-Methode zur Verfügung, die nach dem Durchlauf einer Verarbeitungspipeline aufgerufen wird. Wird das Objekt gelöscht, schließt es das unsichtbare Fenster und gibt den OpenGL-Kontext wieder frei.

### 6.5.2 Datenübertragung

Das native Datenformat für Bilddaten, welches OpenGL zur Verfügung steht, sind Texturen. 2D-Texturen sind optimiert auf zweidimensional lokalen Datenzugriff (Lee et al. (2010)), wovon vor allem die Demosaicing-Operationen profitieren.

Im DNG-Format liegen die Pixeldaten als unsignierte Integer mit bis zu 16 Bit vor. Ein korrespondierendes Integer-Format gibt es in OpenGL-Texturen nicht. Allerdings gibt es bei der Umwandlung von 16-Bit-unsigned-Integern in das verfügbare IEEE 754 Single-Precision (32 Bit) Float-Format, welches aus 23 Bits zur Speicherung des Zahlenwerts, 8 zur Speicherung des Exponenten und einem Bit für das Vorzeichen besteht, keinen Präzisionsverlust.

Über einen OpenGL-Kontext können Bilddaten auf die GPU übertragen werden. Da wir nach der Datenübertragung von OpenGL nur eine Textur-ID zurückbekommen, wird diese auf Java-Seite in ein `OpenGLTexture`-Objekt verpackt, welches z.B. auch die Methode zum Download der Bilddaten zurück in ein `BufferedImage` enthält (für ein einfaches Anwendungsbeispiel siehe Listing 5).

```
1 // step 1: create OpenGL Context
2 OpenGLContext ogl = new OpenGLContext(ConsoleLogger.getInstance());
3 // step 2: upload image
4 GPUImage rawImageGPU = ogl.uploadImage(rawImage, 0);
5 // step 3: apply shaders
6 rawImageGPU.applyShaderInPlace(FragmentShaderExamples.ID_MONO);
7 // step 4: download image
8 rawImageGPU.downloadTo(rawImage);
9 // step 5: free resources
10 rawImageGPU.delete();
11 ogl.delete();
```

Listing 5: Beispielhafte Verwendung der OpenGL Wrapperklassen aus einem

Testfall

### 6.5.3 Datenverarbeitung

Die OpenGL-Pipeline (Abb. 12) erwartet grundsätzlich normierte 3D-Geometriedaten in Punktform (Vertices) als Input. Diese können mithilfe eines Vertex Shaders - eines Kernels, welcher auf Vertices operiert - noch verändert und mit zusätzlichen Daten, wie z.B. Texturkoordinaten angereichert werden:

```
1 #version 130
2 out vec2 TexCoord;
3
4 in vec3 aPos;
5 in vec2 aTexCoord;
6
7 void main()
8 {
9     gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
10    TexCoord = aTexCoord;
11 }
```

Listing 6: In Jeniffer2 verwendeter OpenGL Vertex Shader

Um zweidimensionale Bilddaten zu transformieren, wird in Jeniffer2 als Geometrie nur ein Rechteck aus vier Punkten, die jeweils an den Ecken unserer Textur liegen, benötigt:

```
1 /**
2 * Rectangle used for rendering texture to texture
3 */
4 private static float[] VERTICES_RENDERING = {
5     // positions [-1,1] // texture coordinates [0,1]
6     -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,
7     1.0f, -1.0f, 0.0f, 1.0f, 0.0f,
8     1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
9     -1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
10    -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,
11    1.0f, 1.0f, 0.0f, 1.0f, 1.0f
12};
```

Listing 7: Ein Quadrat bestehend aus zwei Dreiecken in XYZ-Raumkoordinaten und UV-Texturkoordinaten

Diese Informationen werden dann auf die einzelnen sichtbaren Pixel interpoliert. Die sichtbaren Pixel sind der Input für den Fragment Shader, der ebenfalls wieder parallelisiert und unabhängig ausgeführt wird. Dieser Kernel kann auf Bilddaten, die in Form einer Textur hochgeladen wurden, zugreifen. Eine einfache Inversion der Pixelfarbe sieht zum Beispiel so aus:

```

1 #version 130
2 out vec3 color;
3
4 in vec2 TexCoord;
5
6 uniform sampler2D ourTexture;
7
8 void main()
9 {
10     color = vec3(1.0f,1.0f,1.0f) - texture(ourTexture, TexCoord).rgb;
11 }

```

Listing 8: OpenGL Fragment Shader, der die Pixelfarbe invertiert

Für Demosaicing-Operationen muss die (parametrisierte) Position des Pixels im Bayer-Mosaik berechnet werden (Man beachte die Übergabe des Parameters über String-Interpolation in Zeile 10):

```

1 // assume square CFA pattern of side length 2. Positions are
2 // indexed as follows:
3 // +---+---+
4 // | 0 | 1 |
5 // +---+---+
6 // | 2 | 3 |
7 // +---+---+
8 int patternIdx = (int(floor(gl_FragCoord.x)) % 2) + 2 * (int(
9     floor(gl_FragCoord.y)) % 2);
10 // do things differently depending on position
11 // greenRedRowIdx == position of green pixel that is in a row
12 // with red pixels
13 if (patternIdx == "" + greenRedRowIdx + "") { ... }

```

Listing 9: GLSL-Code zur Berechnung der Pixelposition innerhalb des Bayer-Mosaiks

Da Positionen im Bild in Texturkoordinaten im Wertebereich zwischen 0 und 1 angegeben werden, muss vor dem Zugriff auf benachbarte Pixel die normierte Größe eines Pixels berechnet werden:

```

1 // calculate pixel dimensions in texture space
2 ivec2 textureSize2d = textureSize(tex,0);
3 float texelSizeX = 1.0 / float(textureSize2d.x);
4 float texelSizeY = 1.0 / float(textureSize2d.y);
5 // access some neighbouring pixels
6 float vertical =
7     (texture(tex, vec2(TexCoord.x, TexCoord.y + texelSizeY)).r +
8      texture(tex, vec2(TexCoord.x, TexCoord.y - texelSizeY)).r) /
9      2.0;

```

Listing 10: GLSL-Code zur Berechnung des Pixeloffsets innerhalb einer Textur

Das Ergebnis einer Ausführung dieser Pipeline sind RGB-Pixel, die entweder auf dem Bildschirm angezeigt oder - in unserem Fall - in einer neuen Textur gespeichert werden.

Der Fragment Shader enthält also den Kern eines Bildverarbeitungsschrittes. Deswegen stellt genau dieser in der C-ähnlichen GLSL (OpenGL Shading Language) geschriebene Quellcode als String letztendlich einen Prozessschritt dar, oder zumindest eine `for`-Schleife, die ein Zwischenergebnis berechnet.

Die meisten der im DNG-Standard definierten Verarbeitungsschritte benötigen nicht nur das Bild als Input, sondern auch verschiedene Parameter, wie zum Beispiel bei Demosaicing-Operationen, an welcher Stelle im Bayer-Mosaik sich welche Farbe befindet. OpenGL Shadern können programmatisch sogenannte "Uniform Parameter" übergeben werden, die für alle Kernel-Instanzen gleich sind. So können einmal kompilierte Shaderprogramme auch mit anderen Parametern wiederverwendet werden.

So eine Wiederverwendung wird aber im aktuellen Fall nicht benötigt, denn der OpenGL-Kontext wird nach jeder Pipeline-Ausführung wieder gelöscht. Frühe Zeitmessungen haben nämlich ergeben, dass die Kompilation von Shaderprogrammen keinen entscheidenden Einfluss auf die insgesamte Ausführungszeit hat und somit keine dringende Optimierung darstellt.

Für die Java-Wrapperklassen ist es dementsprechend am einfachsten, konstante Parameter als Strings im Quellcode zu übergeben. Unit-Tests zeigen, dass so keine Präzision verloren geht, und ein Großteil der DNG-Prozessschritte sich so abbilden lässt.

Eine Ausnahme sind Lookup-Table-Transformationen. Diese finden aber am Anfang statt und gehören zu den schnelleren Operationen. Daher wurden sie, um die Komplexität der OpenGL-Wrapperklassen zu begrenzen, zunächst zurückgestellt.

#### 6.5.4 Aufteilung in Kacheln - “Tiling”

Da OpenGL (und Grafikkarten im Allgemeinen) darauf ausgelegt sind, Daten auf dem Bildschirm darzustellen, ist der für Texturen verfügbare Grafikspeicher auf ein Vielfaches der Bildschirmauflösung begrenzt. In modernen Systemen ist das meistens 16384 mal 16384 Pixel, aber ältere Grafikkarten beschränken sich zum Teil auch auf 8192 Pixel in jede Richtung. Das bedeutet, dass es je nach System vorkommen kann, dass ein mit einer modernen Digitalkamera produziertes Bild nicht in eine OpenGL-Textur passt.

Zusätzlich hat OpenGL kein Konzept, um Daten "in-Place" zu bearbeiten, es wird im Grafikspeicher also im Prinzip der doppelte Platz einer zu transformierenden Textur benötigt. Dies ist bei dedizierten Grafikkarten, die über

einen eigenen Speicher verfügen, tendenziell eher kein Problem - integrierte Grafikkarten aber teilen sich den Hauptspeicher mit der CPU. Gerade bei Demosaicing-Algorithmen wie Ratio Corrected Demosaicing, die zusätzlich bis zu der 6-fachen Menge der ursprünglichen Daten als Zwischenergebnis in Texturen speichern müssen (mehr zur Datenflussanalyse in Abschnitt [6.6.1](#)), kann hier der Speicher knapp werden.

Deswegen ist es auf manchen Systemen nötig, dass größere Bilder abschnittsweise in Kacheln (engl. Tiles, im Folgenden deshalb auch “Tiling”) verarbeitet werden. Um dies zu unterstützen, wird in jeder Textur auch die Position der Daten im Ursprungsbild gespeichert, und ob es Randregionen gibt, welche nur für nichtlokale Filter (wie z.B. beim Demosaicing) benötigt werden und nicht zurückkopiert werden müssen.

Um sowohl die maximale Texturgröße als auch einen Mangel an Grafikspeicher zu adressieren, muss das Bild Kachel für Kachel hochgeladen, verarbeitet und wieder heruntergeladen werden. Diese Strategie wird im Folgenden auch “Tilewise” (im Code `GPU_TILE_WISE`) genannt und ist in Abb. [17](#) dargestellt.

Wird ein einfacherer Demosaicing-Algorithmus verwendet und ist somit lediglich die maximale Texturgröße ein potentielles Problem, können auch alle Bildstücke auf einmal hochgeladen, jeweils verarbeitet und wieder heruntergeladen werden (im Folgenden auch “Operation Wise” oder `GPU_OPERATION_WISE`, Abb. [18](#)). Dafür werden mehrere Texturen in einem `GPUImage` mit entsprechenden Methoden zusammengefasst.

Um Codeduplikation in den einzelnen Verarbeitungsschritten zu vermeiden, wurde das Interface `TransformableOnGPU` eingeführt, welches sowohl von der einzelnen `OpenGLTexture` als auch vom `GPUImage` implementiert wird.

### 6.5.5 Tests

Die `OpenGLContext`, `OpenGLTexture` und `GPUImage`-Wrapperklassen wurden zunächst unabhängig von der JENIFFER2-Codebasis entwickelt und getestet. Der Code wurde dann zusammen mit entsprechenden Unit-Tests in das Projekt übernommen.

## 6.6 CPU-Optimierung

Die Optimierung des Programms auf der CPU stützt sich neben dem trivialen Multithreading auf eine Datenflussanalyse der komplexeren Demosaicing-Algorithmen und ein paar zusätzliche Überlegungen zur Nutzung der CPU-Caches.

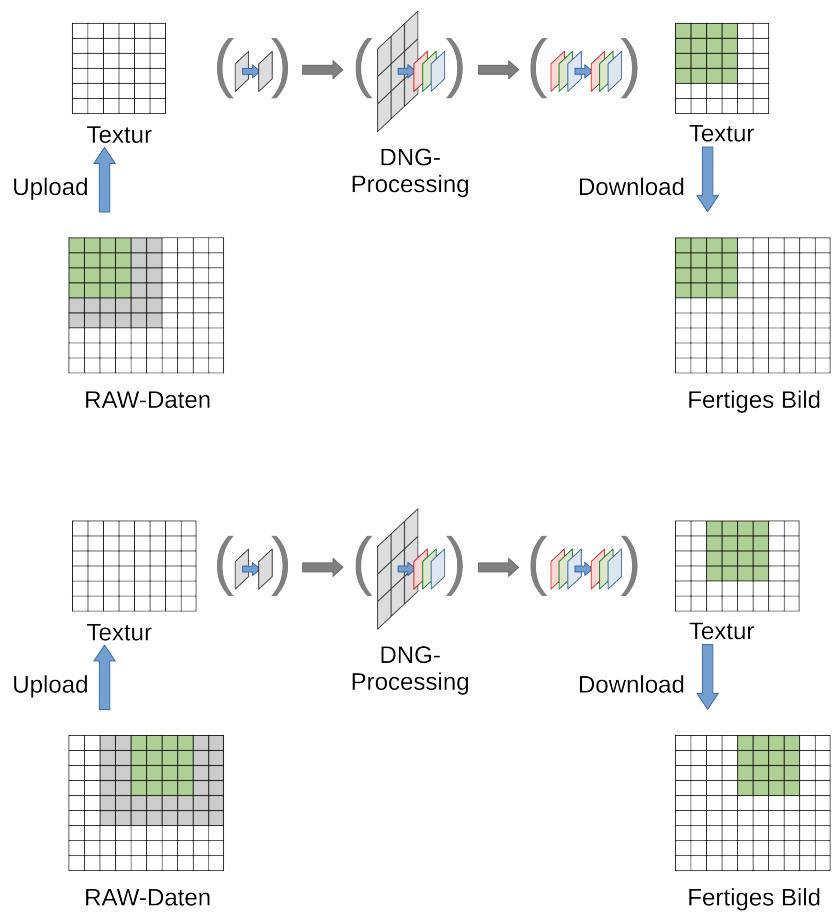


Abbildung 17: Bildverarbeitung Kachel für Kachel

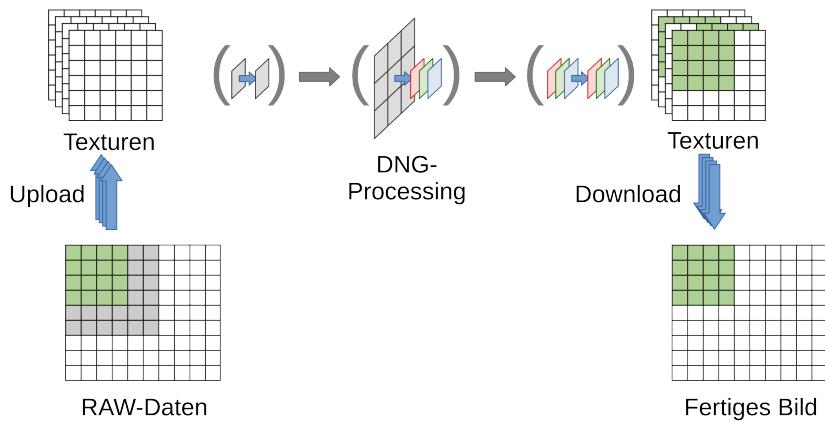


Abbildung 18: Bildverarbeitung alle Kacheln auf einmal

### 6.6.1 Datenflussanalyse

#### Hintergrund: Ratio Corrected Demosaicing

Der Ratio Corrected Demosaicing-Algorithmus berechnet, wie viele andere Algorithmen auch, zunächst die grünen Pixel für das ganze Bild und nutzt diese Information dann zur Interpolation der Rot- und Blauwerte. Allerdings werden auch verschiedene Zwischenergebnisse gespeichert, da sie mehrfach verwendet werden:

Zunächst wird der XY-Gradient, also das Verhältnis zwischen horizontaler und vertikaler Änderung der Helligkeitswerte für jeden Pixel ermittelt. Die fehlenden grünen Pixel werden dann vorläufig mit einem Tiefpassfilter aus den umliegenden Pixeln interpoliert. Daraus werden mithilfe von Richtungsgradienten eine horizontale und vertikale Schätzung ermittelt, welche über den XY-Gradient gewichtet werden. Für den XY-Gradienten wird hier das Maximum aus dem lokalen Wert und der direkten Umgebung herangezogen.

Als nächstes wird der PQ-Gradient als Verhältnis der diagonalen Änderungen berechnet. Dieser wird für die folgende Berechnung der Rot- und Blauwerte an den blauen und roten Stellen im Mosaik benötigt, um hier die diagonalen Schätzungen zu gewichten. Abschließende werden die fehlenden Farbwerte analog zu den grünen Pixeln aus den existierenden Werten und dem XY-Gradient errechnet.

Um Ratio Corrected Demosaicing (Rodriguez (2017)) auf der GPU umzusetzen,

musste der Algorithmus aufgeteilt werden: Jedes Zwischenergebnis wird mithilfe von einem Shader-Programm berechnet und in einer Textur gespeichert. In Abb. 19 ist die Lebensdauer dieser Texturen zu sehen, sowie die Anzahl der Pixel, die für jedes Pixel in einem Zwischenergebnis aus den vorherigen Ergebnissen benötigt wird.

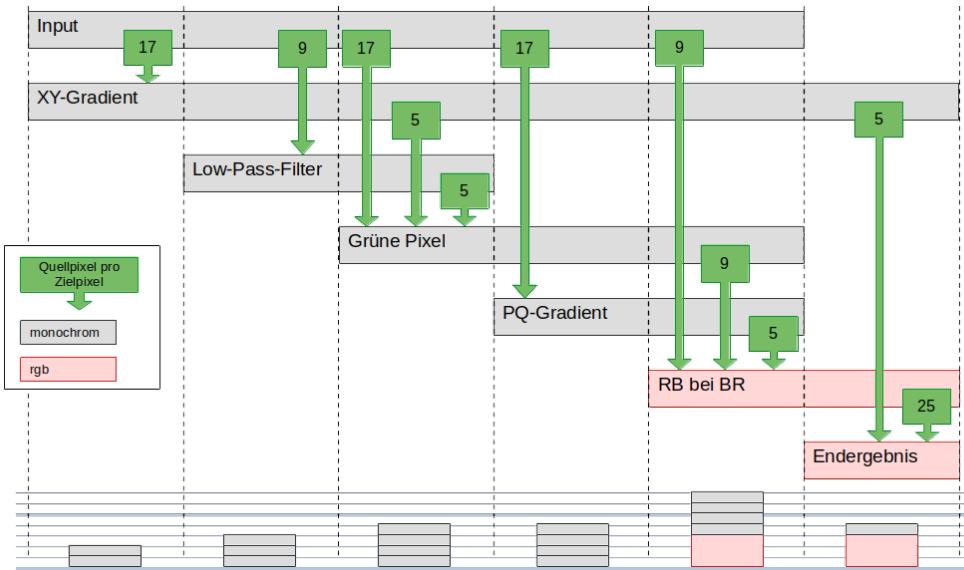


Abbildung 19: Datenfluss Ratio Corrected Demosaicing auf der GPU

Die existierende Implementation des RCD-Algorithmus war von der Programmstruktur, die auf deutlich einfachere Algorithmen ausgelegt war, geprägt: Demosaicing-Algorithmen waren als Operationen implementiert, die eine Pixelposition erhalten und das Resultat an dieser Stelle berechnen - die Kontrolle über den Programmfluss, also die Schleifen bzw. den parallelen Stream befand sich in der übergeordneten `DemosaicingProcessor`-Klasse. Das Operation-Interface war zwar schon dahingehend erweitert, dass zunächst die grünen Pixel als Zwischenergebnis berechnet wurden, und es einen Sonderfall gab, mithilfe dessen danach noch ein weiteres Zwischenergebnis gespeichert werden konnte. Dennoch wurden mit diesem Ansatz, wie in Abb. 20 zu sehen, die Zwischenergebnisse des Low-Pass-Filters, des XY-Gradienten und des PQ-Gradienten, die sonst von 5 Pixeln verwendet würden, jedes Mal neu berechnet. Dies resultierte zwar theoretisch aufgrund von Überlappungen nicht in einem 5-fachen Pixelfußabdruck, tatsächlich wurde aber die Funktion zum Auslesen des entsprechenden Pixels trotzdem 5 mal so oft aufgerufen.

Um dieses Problem zu beheben, wurde die Programmstruktur auf der CPU der auf der GPU angepasst. Auch die Struktur der DLMMSE- und DLMMSE+RCD-Algorithmen wurde entsprechend analysiert und die Algorithmen wurden reimplementiert.

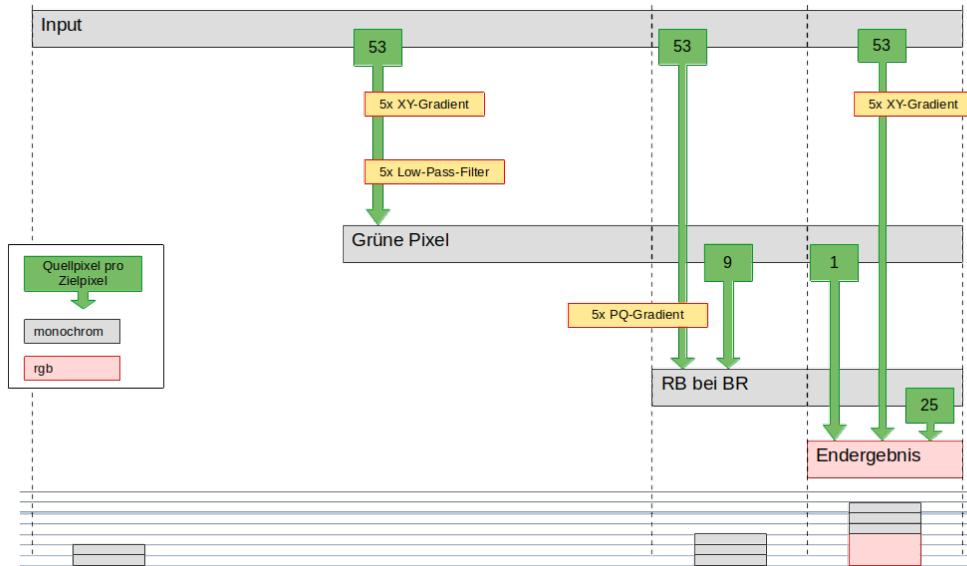


Abbildung 20: Datenfluss Ratio Corrected Demosaicing (alte Implementation)

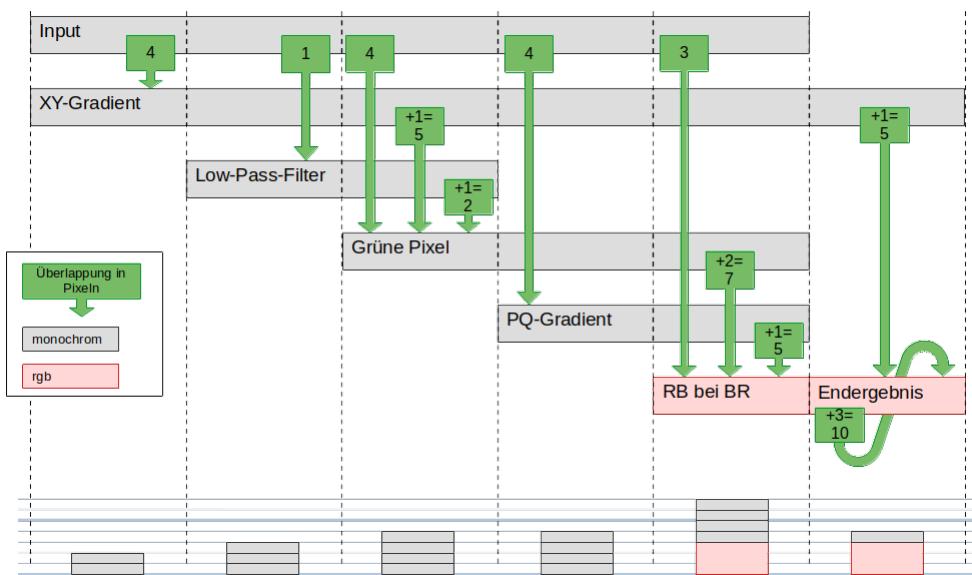


Abbildung 21: Datenfluss Ratio Corrected Demosaicing (neue Implementation)

Hierfür wurde der Demosaicing-Prozessor dahingehend geändert, dass er den Demosaicing-Operationen ein Array mit den Pixeldaten sowie Informationen zu den Bilddimensionen und zur Anordnung des Bayer-Mosaiks übergibt. Somit sind die Algorithmen in sich abgeschlossen und haben selber die Kontrolle über den Programmfluss und mögliche Zwischenergebnisse. Dies erhöht auch die

Lesbarkeit und Verständlichkeit der Algorithmen. So fiel durch das Umschreiben zum Beispiel auch ein Tippfehler in der Implementation von Patterned Pixel Grouping (Lin (2010)) auf.

Was die Datenflussanalyse der RCD (21), DLMMSE (22) und DLMMSE+RCD-Algorithmen (23) auf der CPU aber auch hervorbrachte, war, dass durch die Speicherung der Zwischenergebnisse der Speicherbedarf stark wächst.

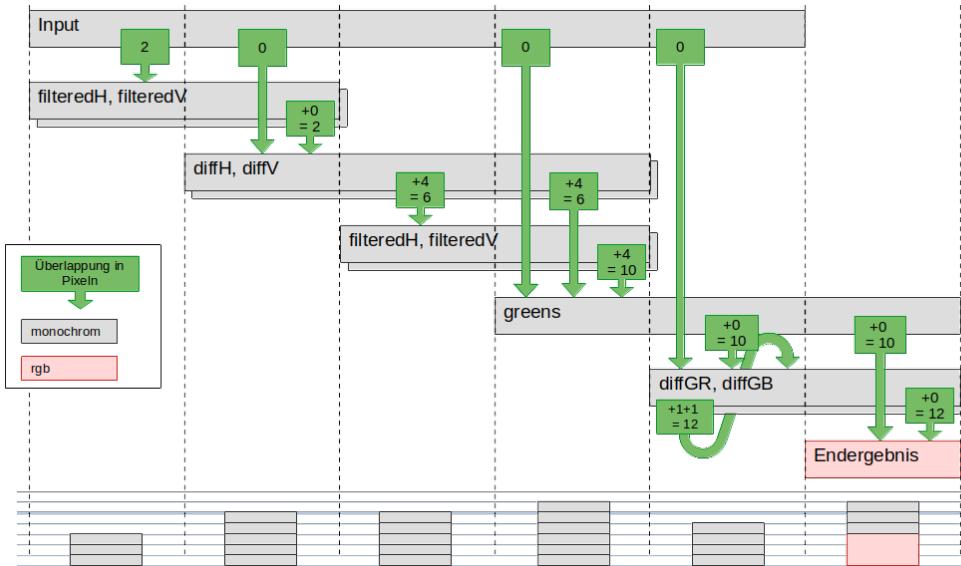


Abbildung 22: Datenfluss DLMMSE Demosaicing

### 6.6.2 Tiling auf der CPU

Wie in Abbildungen 21, 22 und 23 zu sehen, brauchen die optimierten Algorithmen teilweise bis zum 7-fachen der ursprünglichen, monochromen Bilddaten an Speicher. Zu beachten ist außerdem, dass durch die Speicherung in Arrays gleichzeitig noch ein monochromes und ein dreifarbiges BufferedImage im Speicher existieren. Da die Arrays die Daten als IEEE 754 Single-Precision Floating Point-Werte von 4 Byte speichern, sind das im Falle eines Bildes von 48 Megapixeln also bis zu  $4 \cdot 48 \cdot 7 = 1344$  Megabyte an Speicher allein für die Arrays. Der Faktor 7 geht davon aus, dass die nicht mehr benötigten Zwischenergebnisse sofort vom Garbage Collector erkannt und entfernt werden - ansonsten sind es bis zu  $4 \cdot 48 \cdot 13 = 2496$  Megabyte (mit DLMMSE). Neben dem Speicher, den das Betriebssystem und die Java Virtual Machine selbst belegen, und dem von anderen offenen Programmen wie z.B. einem Browser oder einem Code-Editor benötigten Speicher reichen hierfür 8 Gigabyte Arbeitsspeicher, eine bei Laptops immer noch häufige Konfiguration, manchmal nicht aus.

Um dieses Problem zu beheben, wurde im Demosaicing-Prozessor die Mög-

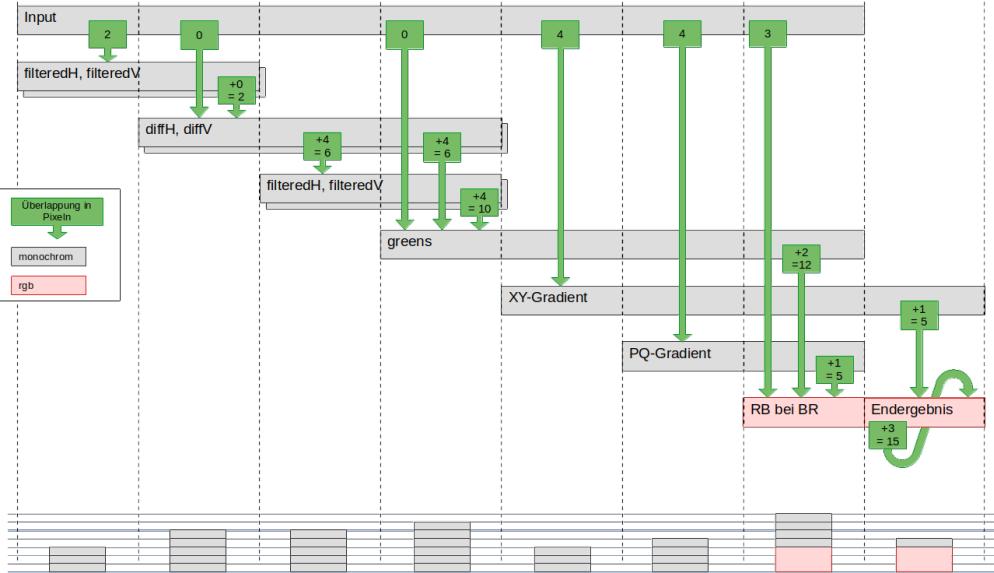


Abbildung 23: Datenfluss DLMMSE+RCD Demosaicing

lichkeit eingebaut, das zu verarbeitende Bild in Kacheln aufzuteilen, ähnlich den Kacheln auf der GPU. Die Abbildungen 21, 22 und 23 zeigen deswegen statt dem Pixelfußabdruck in verwendeten Pixeln die Anzahl der Pixel bezogen auf das Ursprungsbild an, die die Kacheln überlappen müssen, damit das Endergebnis korrekt bleibt.

Abgesehen davon, die Berechnung von großen Bildern auf Rechnern mit wenig Arbeitsspeicher überhaupt erst zu ermöglichen, bietet die Aufteilung des Bilds in Kacheln auch eine weitere Optimierungsmöglichkeit: Wenn die Zwischenergebnisse für eine Kachel alle in einen Prozessorcache passen, dessen Zugriff um Größenordnungen schneller ist als Zugriff auf den Hauptspeicher, so die Hypothese, beschleunigt dies die Berechnung. Andererseits findet dabei auch ein Trade-Off statt: Je kleiner die Kacheln im Verhältnis zum überlappenden Bereich sind, desto mehr Berechnung wird “verschwendet”.

Eine zusätzliche Dimension ist die Aufteilung des Multithreading: Sollen die Demosaicing-Algorithmen die Kacheln auf Threads verteilen, aber innerhalb der Threads mit einfachen `for`-Schleifen arbeiten? Soll die Aufteilung in Kacheln sequentiell stattfinden, aber die Berechnung innerhalb der Kacheln mit Multithreading? Alle vier Optionen, die sich ergeben, wurden implementiert und zusammen mit verschiedenen Kachelgrößen gebenchmarkt, um die optimale Konfiguration zu finden.

Auch wenn die Aufteilung in Kacheln mangels Zwischenergebnissen aus dem Gesichtspunkt der Speicher-Optimierung für die Pre- und Postprozessoren wenig Sinn ergibt, kann es trotzdem sein, dass sie z.B. den Overhead für

Multithreading verringert. Dementsprechend wurde auch dort Tiling in allen Varianten implementiert und getestet.

### 6.6.3 Zusammenfassung von Operationen

Während bei den Demosaicing-Algorithmen für die Berechnung eines Pixels eine Umgebung um den Pixel benötigt wird, muss für die Schritte im Pre- und Postprocessing jedes Pixel nur einmal gelesen und geschrieben werden, was an Ort und Stelle geschehen kann. Das bedeutet, dass die per-Pixel-Operationen auch tatsächlich so hintereinandergeschaltet werden können, dass nicht für jede Operation einmal neu über die Pixel iteriert wird, sondern jedes Pixel tatsächlich konzeptuell nur einmal gelesen und geschrieben wird. So finden die Speicherzugriffe, um die Zwischenergebnisse weiterzugeben, zeitlich deutlich näher beieinander statt und der Cache kann besser genutzt werden. Der einzige Nachteil dieses Verfahrens ist, dass bei der Zeitmessung nicht mehr unterschieden werden kann, welche Operation wieviel Zeit ausmacht.

Ein Vorteil ist allerdings im Postprozessor, dass alle Operationen intern mit Floating-Point-Daten rechnen (und einige davon im Wertebereich zwischen 0 und 1). Durch die Zusammenfassung der Operationen in einer Schleife muss die Konvertierung von und zu Integer-Werten nur noch einmal stattfinden.

### 6.6.4 Edge-Handling-Strategie

In der bisherigen, pixelweisen Implementation der Demosaicing-Algorithmen gab es eine Hilfsmethode für den Zugriff auf Pixel, welche die Koordinaten auf den Bildbereich einschränkte. Diese Methode, welche aus einer `Math.min` und einer `Math.max`-Operation pro Koordinate bestand, wurde durch eine Berechnung der Koordinaten direkt im Code des Algorithmus ersetzt. So kann das Wissen ausgenutzt werden, dass z.B. beim Zugriff auf ein Pixel rechts eines existierenden Pixels nur geprüft werden muss, ob es rechts über den Bildrand hinausgehen würde, was in nur noch einer Operation statt vier resultiert. Dafür nimmt die Lesbarkeit des Codes etwas ab - die Berechnung von Indizes in einem Array, das ein Bild in Row-Major-Order enthält, sollte aber allen, die sich mit Bildverarbeitung auseinandersetzen, hinreichend bekannt sein (siehe Listing 11 für einen beispielhafte Vergleich).

## 6.7 Reproduktion Qualitätsbenchmarks

Während die beschriebenen Refactorings bei den einfacheren Demosaicing-Algorithmen mithilfe der Regressionstests eine hundertprozentige Äquivalenz erreichten, blieb für die DLMMSE, RCD und DLMMSE+RCD-Algorithmen leider ein - wenn auch kleiner - Restfehler. Auch die Implementierung auf der GPU weist kleine Unterschiede auf.

```

1 // low-pass-filter vorher
2 public float getLPF(int x1, int y1) {
3
4     return 0.25f * getUndemosaicedSample(x1, y1) + 0.125f * (
5         getUndemosaicedSample(x1, y1 - 1) + getUndemosaicedSample(x1
6         , y1 + 1) + getUndemosaicedSample(x1 - 1, y1) +
7         getUndemosaicedSample(x1 + 1, y1)) + 0.0625f * (
8         getUndemosaicedSample(x1 - 1, y1 - 1) +
9         getUndemosaicedSample(x1 + 1, y1 - 1) +
10        getUndemosaicedSample(x1 - 1, y1 + 1) +
11        getUndemosaicedSample(x1 + 1, y1 + 1));
12
13 }
14 // low-pass-filter nachher
15 float middle = samples[x + y * width];
16
17 float top = samples[x + Math.max(0, y - 1) * width];
18 float bottom = samples[x + Math.min(height - 1, y + 1) *
19 width];
20 float left = samples[Math.max(0, x - 1) + y * width];
21 float right = samples[Math.min(width - 1, x + 1) + y *
22 width];
23
24 float topLeft = samples[Math.max(0, x - 1) + Math.max(0, y
25 - 1) * width];
26 float topRight = samples[Math.min(width - 1, x + 1) + Math.
27 max(0, y - 1) * width];
28 float bottomLeft = samples[Math.max(0, x - 1) + Math.min(
29 height - 1, y + 1) * width];
30 float bottomRight = samples[Math.min(width - 1, x + 1) + Math.
31 min(height - 1, y + 1) * width];
32
33 lowPassAtRB[x + y * width] = 0.25f * middle +
34 0.125f * (top + bottom + left + right) +
35 0.0625f * (topLeft + topRight + bottomLeft + bottomRight);

```

Listing 11: Vergleich Edge-Handling über Funktion und Inline

Um sicherzustellen, dass diese Differenz zur bisherigen Implementation für die Bildqualität nicht beeinträchtigend ist, und um eine Qualitätskontrolle für zukünftige Arbeiten zu vereinfachen, wurde im Jeniffer2-Repository das Vorgehen zur Reproduktion der Benchmarking-Ergebnisse von Reiter (2023) dokumentiert und geskriptet. Statt proprietärer Matlab-Funktionen wurden die in C geschriebenen und mit frei verfügbaren Mitteln kompilierbaren Kommandozeilenwerkzeuge zum Mosaicing und Bildvergleich von Getreuer (2011) integriert. Mithilfe eines Shell-Skripts können die Kommandozeilen-Version von Jeniffer2 auf den Bilddatensets aufgerufen und die Qualitätsmetriken im Vergleich mit den Referenzbildern in eine CSV-Datei gespeichert werden. Aus dieser können dann mithilfe eines Python-Skripts zusammenfassende Abbildungen generiert werden.

Wie in Abb. 24 zu sehen, entsprechen die umgeschriebenen Versionen in ihrem Ergebnis in der Qualitätsbenchmark dem Original. Die Behebung des Tippfehlers in der Implementation von Patterned Pixel Grouping verbessert die Performance zwar sichtbar, ändert aber nichts an der Einordnung im Gesamtbild. Genauer auf die Ergebnisse eingegangen wird in Abschnitt 7.1.

## 6.8 Grafische Benutzeroberfläche

Im Verlauf der Arbeit wurden verschiedene Erweiterungen und Usability-Verbesserungen der graphischen Benutzeroberfläche umgesetzt.

### 6.8.1 Wahl der Beschleunigungsstrategie

Um Feldtests zu ermöglichen, wurde in der graphischen Benutzeroberfläche eine Auswahl der Beschleunigungsstrategie - also z.B. Multithreading oder Tilewise-Verarbeitung auf der GPU - hinzugefügt (Abb. 25). Es ist geplant, anhand der Ergebnisse dieser Arbeit eine per Default ausgewählte Option "Optimal" hinzuzufügen, welche für den jeweiligen Schritt und Algorithmus die schnellste Beschleunigungsstrategie wählt.

Zum Debugging war zwischenzeitlich auch die Möglichkeit implementiert, den Demosaicing-Algorithmus bereits bei einem Zwischenergebnis, wie es in der Datenflussanalyse zu sehen ist, zu stoppen. Die Zwischenergebnisse sichtbar zu machen ist sicher auch für das Verständnis der Algorithmen interessant. Während dieses Feature im Zuge des Performance-Refactorings auf der CPU zunächst außen vor blieb, sollte eine Re-Implementation machbar sein und ist für die Zukunft geplant, sofern sich nicht herausstellt, dass sie die Performance stark beeinträchtigt.

## Überblick Akkurateit reimplementierter Algorithmen (n = 57)

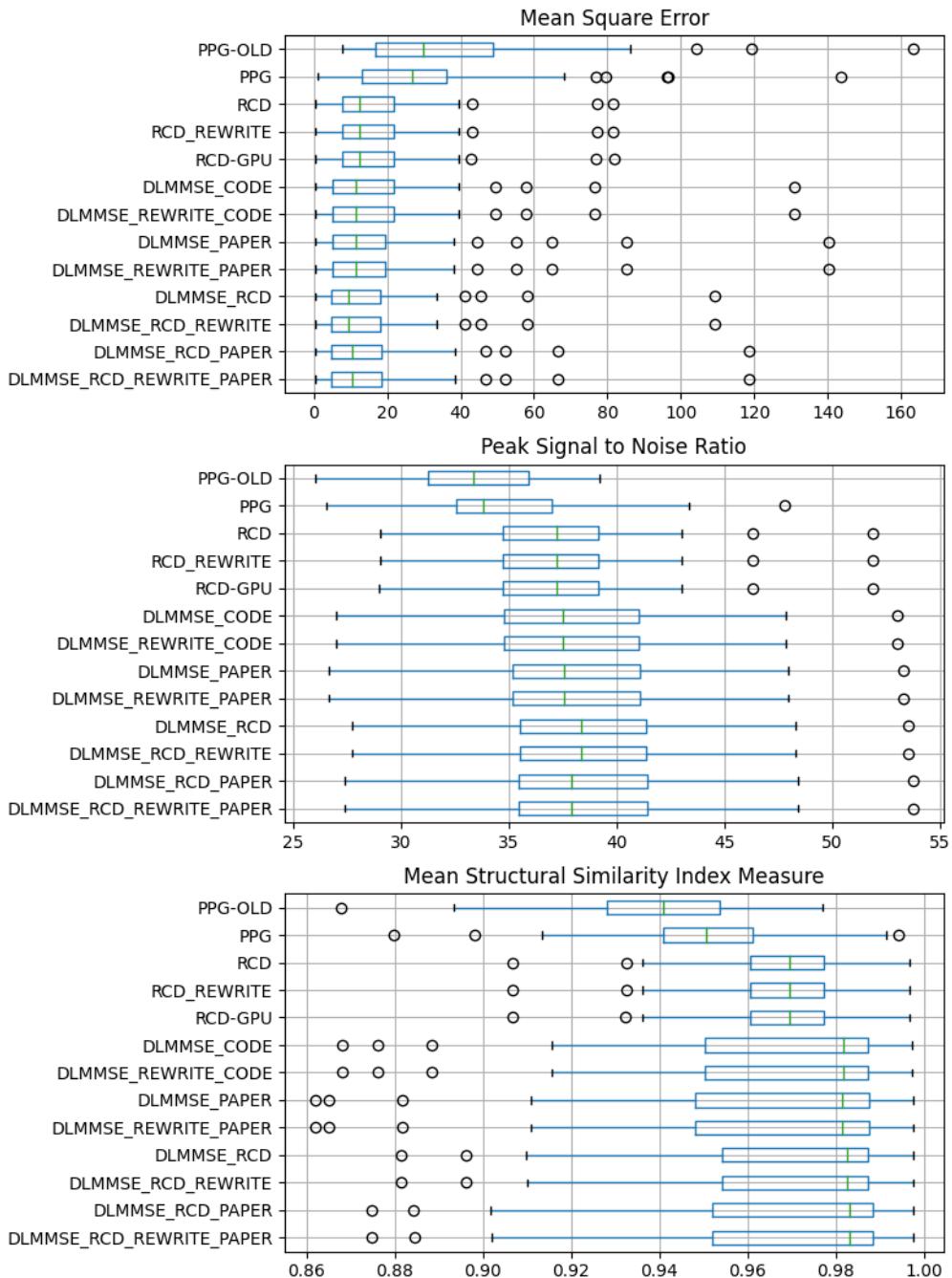


Abbildung 24: Akkurateit der reimplementierten Algorithmen anhand unterschiedlicher Metriken

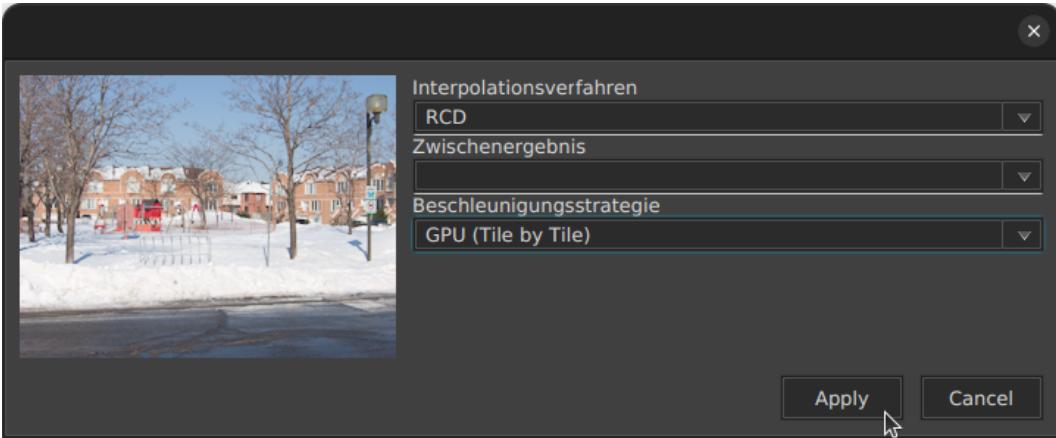


Abbildung 25: Konfigurationsmöglichkeiten in Jeniffer2

### 6.8.2 Fehleranzeige

Da das verwendete GUI-Framework die Komponenten der Benutzeroberfläche in Hilfsthreads initialisiert und dabei im Fehlerfall keinen Stacktrace ausgibt, war es in ersten Feldversuchen schwierig, Support zu leisten. In der GUI war teilweise gar nicht sichtbar, dass ein Fehler passiert war, da der Status der Berechnungsanzeige von der abgestürzten Komponente nicht mehr aktualisiert wurde und somit auf "in Arbeit" verblieb.

Um Fehler transparenter zu machen und den Support zu erleichtern, fangen die (Initialisierungs-)methoden aller Komponenten nun selbstständig mögliche Exceptions ab und geben im Zweifelsfall einen Stacktrace auf die Konsole aus. Zusätzlich wird ein Fehlerdialog angezeigt, der die Nutzenden informiert und einen Aufruf enthält, dem Entwicklungsteam Bescheid zu geben.

### 6.8.3 Algorithmenvergleich

Jeniffer2 ist ein Programm, dass zum Erkunden und Vergleichen verschiedener Demosaicing-Algorithmen einladen soll. Um dies zu erleichtern, wurden in der Toolbar über einem fertig verarbeiteten Bild Drop-Down-Menüs mit den unterschiedlichen Verarbeitungsoptionen eingeführt. Durch einen Klick auf "Compute" kann so das Bild durch das Ergebnis einer anderen Konfiguration ersetzt werden. Wurde das Bild schon in einer Konfiguration verarbeitet, so reicht die Auswahl des grün markierten Eintrags im Drop-Down-Menü, um die Ansicht zu wechseln. So kann schnell und einfach zwischen den Ergebnissen verschiedener Algorithmen gewechselt werden. Abb. 26 zeigt eine beispielhafte Navigation durch das Menü mit zwei Klicks. Diese Art der Navigation wird besonders dadurch ermöglicht, dass auch auf das Ergebnis komplexer Algorithmen nicht lange gewartet werden muss.

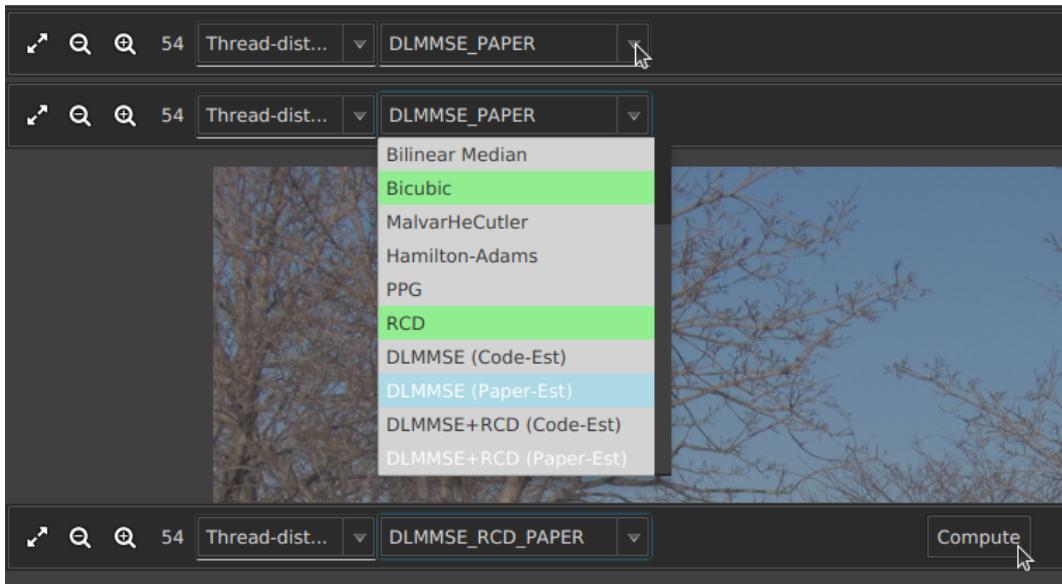


Abbildung 26: Konfigurations-Vergleich in der Toolbar

#### 6.8.4 Zoom und Pixellupe

Im Rahmen der Empirie traten einige Bugs mit der Bildvergrößerung zu Tage, die unter anderem auch im Zuge der Arbeit von Reiter (2023) behoben wurden. Als viel größeres Problem fiel allerdings auf, dass die verwendete Bildkomponente automatisch eine Interpolation der Pixel im Bild vornimmt und somit auch bei starkem Zoom das Resultat der Algorithmen nicht auf Pixelebene betrachtet und verglichen werden kann.

Eine im Zuge der Arbeit von Ljavin (2020) implementierte Lösung war ein Upsampling des Bilds beim Zoom-In. Dies resultierte aber in einem erhöhten Speicherbedarf und einer schlechten Performance, weswegen der entsprechende Code bereits deaktiviert war.

Im Zuge dieser Arbeit wurde die bereits vorhandene Komponente, welche die Farbwerte des Pixels unter dem Mauszeiger anzeigt, derart erweitert, dass sie in einem Canvas-Element vergrößert die Pixel-Umgebung um den Pixel darstellt (Abb. 27). Die Größe des in dieser ‘‘Lupe’’ dargestellten Bereichs sowie der angezeigte Farbkanal können dabei je nach Wunsch gewählt werden.

Zusätzlich kann in der Pixel-Lupe auch ein Vergleich des ausgewählten Bereichs mit anderen, bereits berechneten Konfigurationen vorgenommen werden (Abb. 28). So werden die Unterschiede in Demosaicing-Algorithmen besonders deutlich.

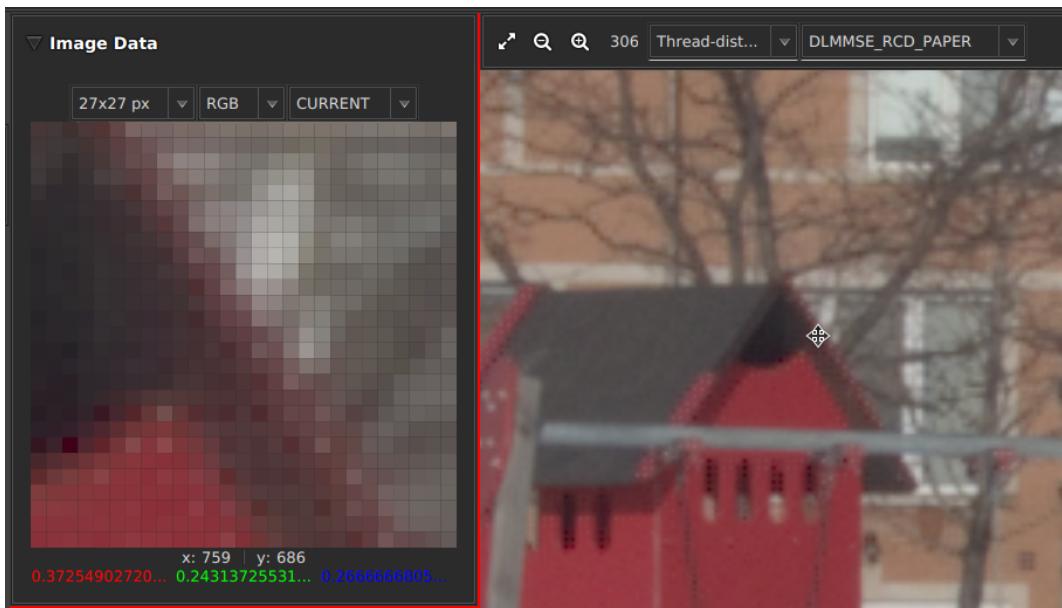


Abbildung 27: Pixel-Lupe

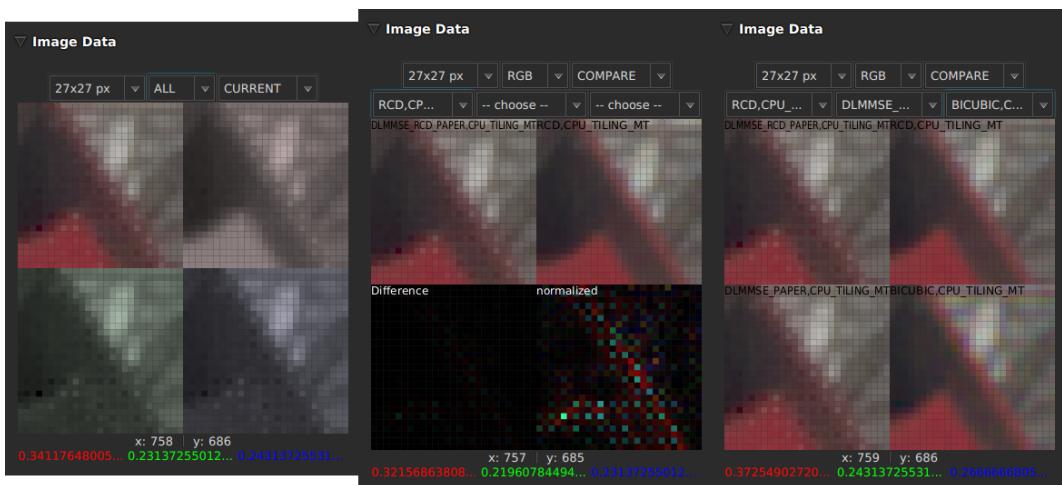


Abbildung 28: Farbkanal- und Vergleichsmodi der Pixel-Lupe

## 7 Ergebnisse

### 7.1 Reproduktion der qualitativen Metriken

Wie im Implementationsteil erwähnt, wurden im Rahmen dieser Arbeit die Messungen zur Akkuratheit der Demosaicing-Algorithmen aus Reiter (2023) reimplementiert, um die Qualität der neuen Versionen zu sichern. Es wurden die gleichen Datensets verwendet (Kodak Lossless True Color Image Suite<sup>7</sup>, McMaster Dataset<sup>8</sup>, Microsoft Research Cambridge Demosaicing Dataset<sup>9</sup>) und die Bilder wurden ebenfalls im 8-Bit-Farbraum verglichen. Der einzige Unterschied in der Messung ist, welcher Bildausschnitt mit dem Original verglichen wird: In der neuen Implementation wird der Randbereich der Bilder außer Acht gelassen, auf den die Randbehandlung - also z.B. die Wiederholung des äußersten Pixels oder das Auffüllen mit Grauwerten - einen Einfluss hat. Diese Information steht dank den Datenflussanalysen zur Implementation der Verarbeitung in Kacheln auf der CPU zur Verfügung.

Dementsprechend decken sich die Ergebnisse mit denen aus der reproduzierten Arbeit, auch wenn es natürlich durch die unterschiedliche Randbehandlung und Implementation kleine numerische Unterschiede gibt. Interessant ist aber die Darstellung der Ergebnisse in Boxplots, welche unter anderem auch hervorhebt, wie stabil oder voraussagbar die Performanz eines Demosaicing-Algorithmus ist.

Wie in Abb. 29 zu sehen, setzen sich die besten Algorithmen vor allem in der Quadratfehler-Metrik (Mean Square Error) als auch in der strukturellen Ähnlichkeit (Mean Structure Similarity Index) ab. Während die DLMMSE-Algorithmen die besten Ergebnisse liefern, fallen die Ergebnisse des RCD-Algorithmus auf durch eine vergleichsweise hohe Stabilität über die unterschiedlichen Bilder, sowohl was die Peak Signal to Noise Ratio als auch die strukturelle Ähnlichkeit angeht.

Schlüsselt man die einzelnen Metriken für die am besten abschneidenden Algorithmen nach den verwendeten Datensets auf (Abb. 30, 31, 32), zeigen sich starke Unterschiede. Insbesondere die Ergebnisse auf dem McMaster-Datenset unterscheiden sich oft von den anderen. So schneidet hier RCD am besten ab, und in der Implementation simple Algorithmen wie Hamilton-Adams und teilweise auch Malvar-He-Cutler übertreffen die verschiedenen deutlich komplexeren DLMMSE-Varianten. Dies ist insofern interessant, da das McMaster-Datenset Bilder mit höherer Sättigung und schärferen Kanten enthält als die anderen beiden Datensets (Reiter (2023)).

---

<sup>7</sup><http://r0k.us/graphics/kodak/> (Zugriff: 01.07.2023)

<sup>8</sup>[https://www4.comp.polyu.edu.hk/~cslzhang/CDM\\_Dataset.htm](https://www4.comp.polyu.edu.hk/~cslzhang/CDM_Dataset.htm) (Zugriff: 01.07.2023)

<sup>9</sup>Khashabi et al.(2023)

## Überblick Akkuratheit Algorithmen (n = 57)

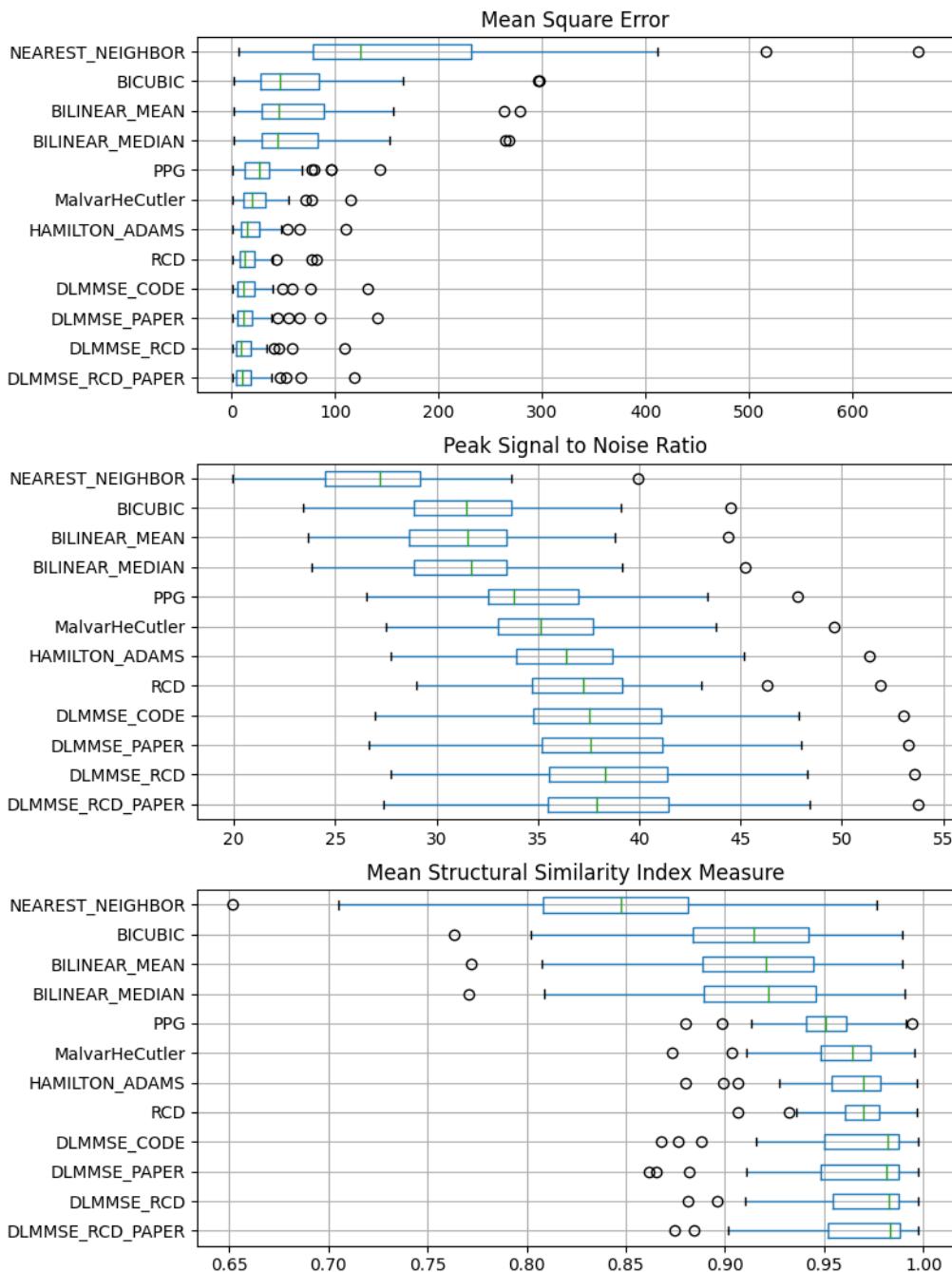


Abbildung 29: Überblick über die Demosaicing-Algorithmen in Jeniffer2 anhand verschiedener Metriken

### Mean Square Error

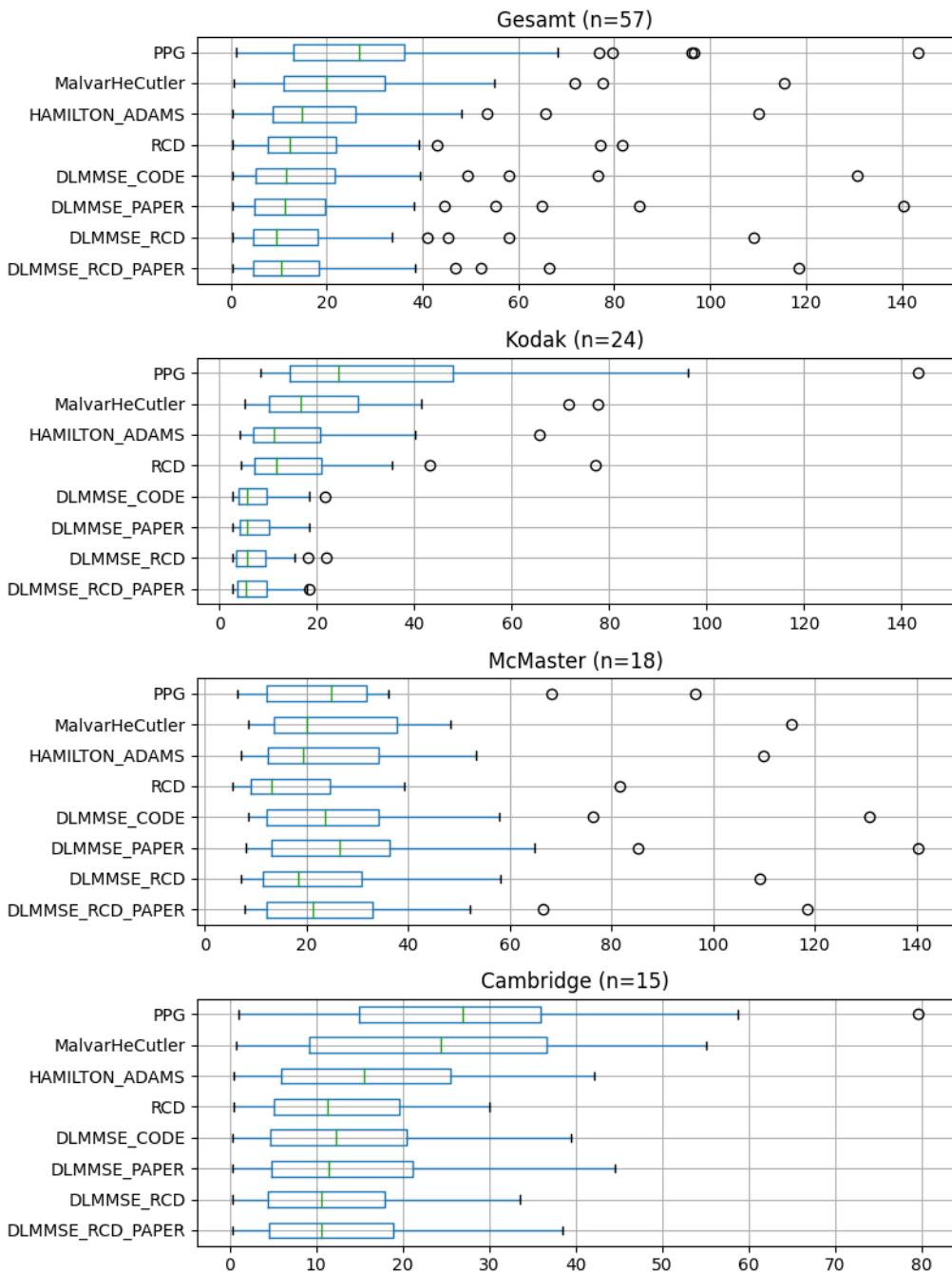


Abbildung 30: Ergebnisse für Mean Square Error aufgeschlüsselt nach Datenset

### Peak Signal to Noise Ratio

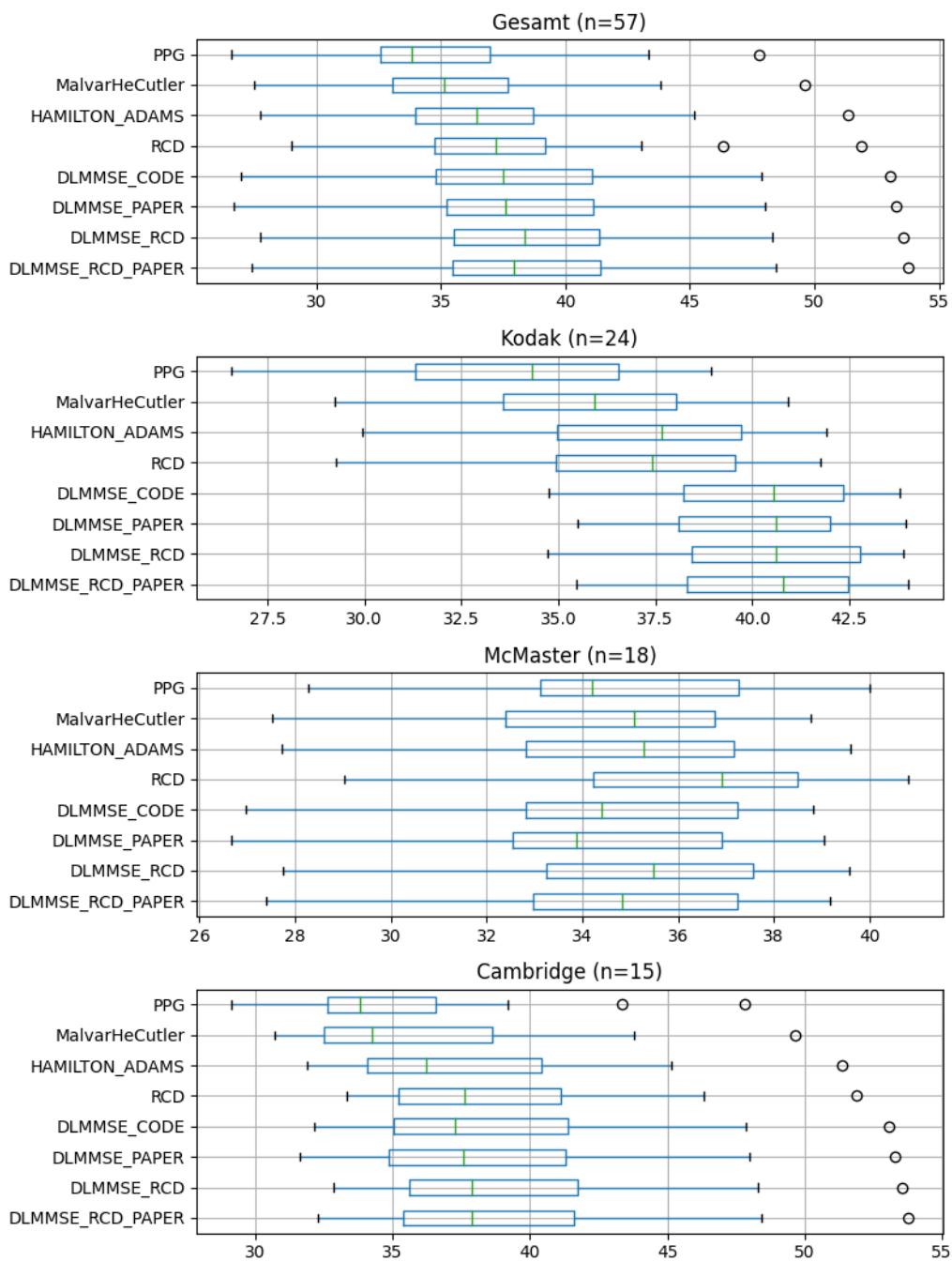


Abbildung 31: Ergebnisse für PSNR aufgeschlüsselt nach Datenset

### Mean Structural Similarity Index Measure

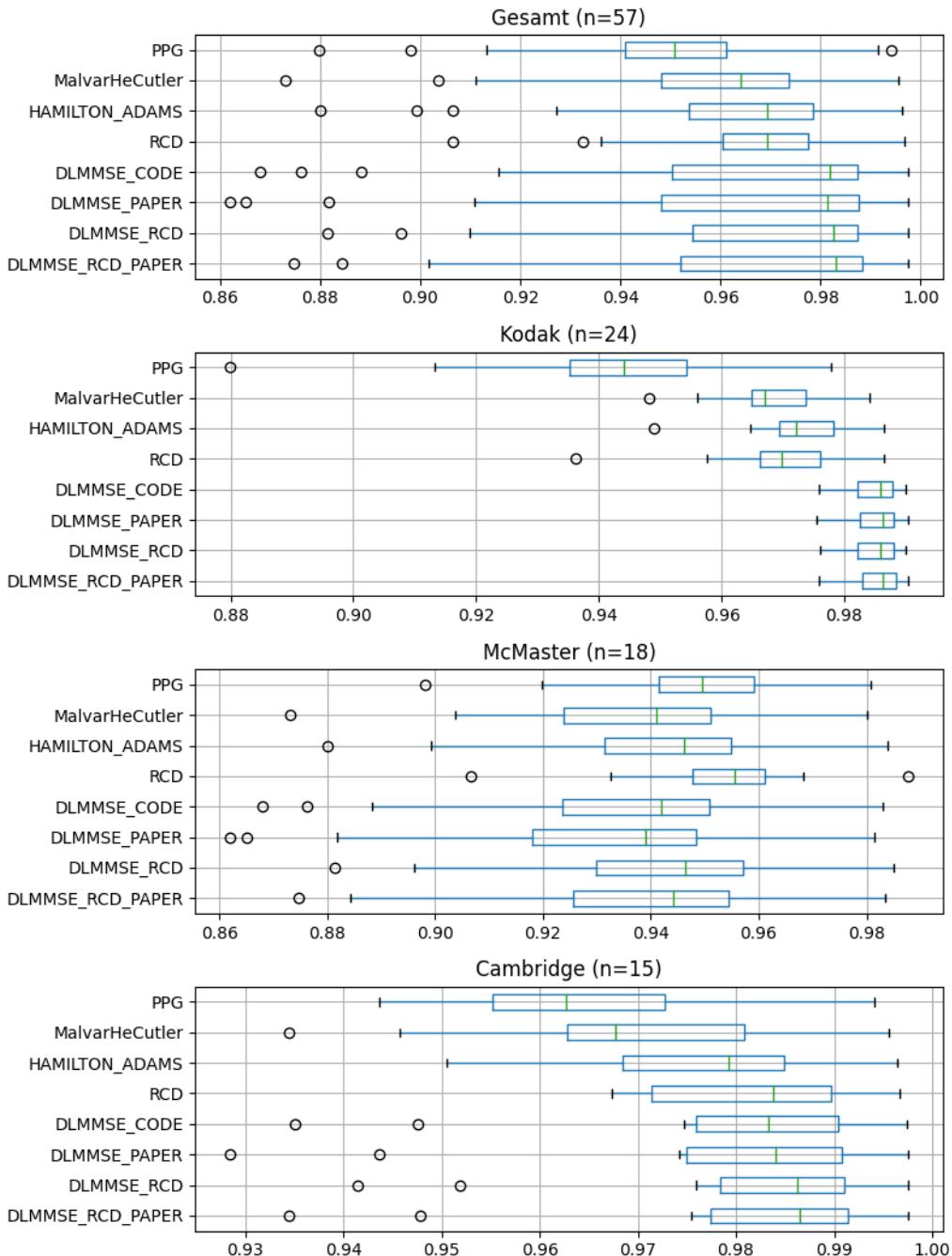


Abbildung 32: Ergebnisse für MSSIM aufgeschlüsselt nach Dataset

## 7.2 Feldtest

Wie weiter oben beschrieben, wurde für den Feldtest eine Entwicklungsversion von Jeniffer2 mit Instruktionen an Probanden verteilt. Insgesamt wurden von 25 Teilnehmenden Daten zurückgesendet. Allerdings verwendeten 3 die falsche Version, und 2 konnten keine Ergebnisse liefern.

### 7.2.1 Hardware der Stichprobe

Von den 20 Teilnehmenden, die brauchbare Daten lieferten, hatte eine Person die Daten über die Hardware aus der Logdatei entfernt. Teilweise konnte auch die Mikroarchitektur der CPU nicht ermittelt werden. Aus den restlichen Daten ergibt sich aber, wie in Tabelle 3 zu sehen, ein gutes Bild über die Stichprobe.

Das Spektrum des verbauten Arbeitsspeicher reicht von 6 bis 34 Gigabyte, wobei der hohe Anteil an Rechnern mit 16 Gigabyte sich vermutlich darauf zurückführen lässt, dass alle Teilnehmenden Studierende der Informatik oder verwandter Fächer sind. Es wurden am meisten Prozessoren mit 4 physischen bzw. 8 logischen Kernen getestet, die im Schnitt zum Zeitpunkt des Tests 4 Jahre alt waren.

Was die Grafikhardware anging, sind in etwa gleichem Maße Grafikkarten von NVIDIA und integrierte Grafikprozessoren von Intel vertreten, welche alle mindestens OpenGL 4.5 unterstützen. Da OpenGL backwards-kompatibel ist, war es also kein Problem, dass die Implementierung in OpenGL 3.0 geschrieben ist, um eines der Testgeräte des Autors zu unterstützen.

Auch die maximal unterstützte Textur-Seitenlänge der getesteten Grafikhardware reichte ohne Probleme, um auch das größere verwendete Testbild der Größe 8424 mal 5632 zu speichern. Da sich während der Implementation ein Performancevorteil durch die Begrenzung der Texturgröße auf die Hälfte der maximal möglichen Seitenlänge ergeben hatte, wurde so zumindest bei 10 von 20 Teilnehmenden die Aufteilung des Bilds in mehrere Kacheln aktiviert.

### 7.2.2 Bugs und Probleme

Bei den Tests ergaben sich auf vielen Geräten Probleme, unter anderem bei der Verarbeitung des größeren Bilds auf der GPU:

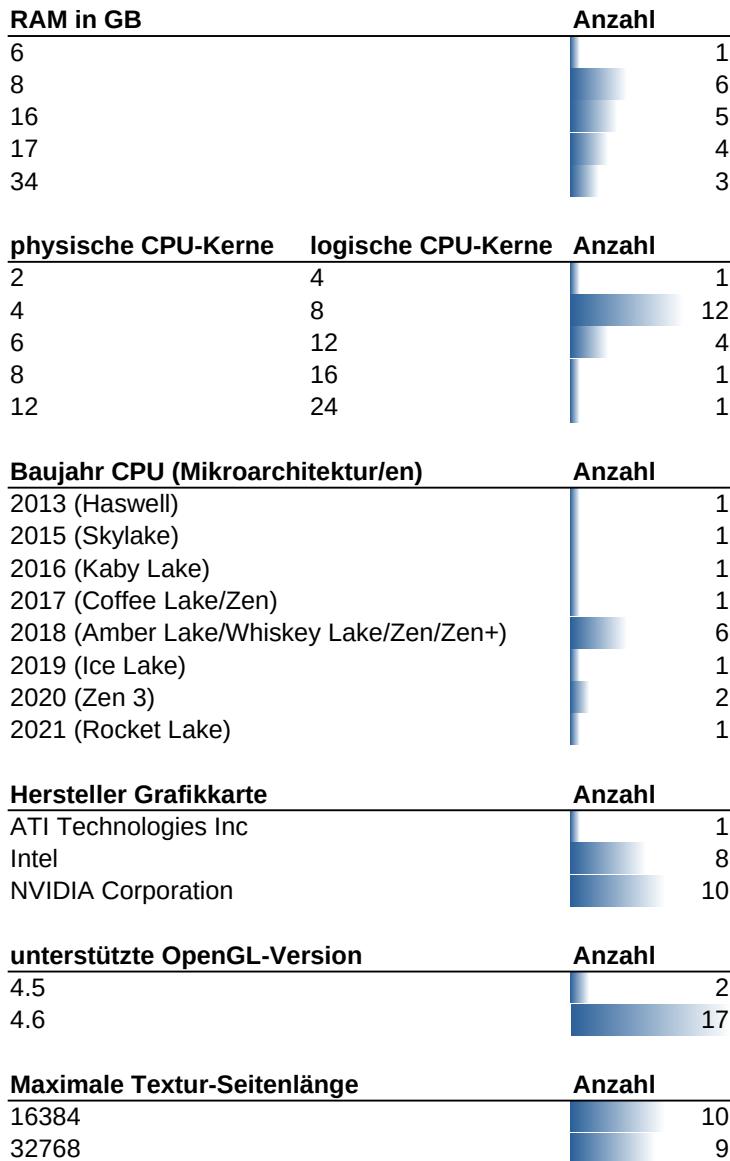


Tabelle 3: Überblick über die Hardware der Stichprobe

Tabelle 4: Zusammenfassung der aufgetretenen Fehler nach Art der GPU

Art der GPU	Fehlerkategorie	Häufigkeit
Grafikkarte	Jeniffer2 konnte nicht gestartet werden	1
Grafikkarte	JNI-Treiber vom Grafikframework nicht unterstützt	1
Grafikkarte	Speicher geht aus bei großem Bild	1
Grafikkarte	Kein Fehler	11
Integrierte Grafik	Ergebnis großes Bild auf der GPU ist schwarz	5
Integrierte Grafik	großes Bild konnte im Dateibrowser nicht gefunden werden	1
Integrierte Grafik	kein Fehler	2

Es stellte sich heraus, dass OpenGL auf Apple-Computern mit der Apple Silicon-Chiparchitektur nur in der “ES”-Version, also der Version für eingebettete Systeme und Smartphones, oder einer Version bis 2.1 unterstützt wird. Diese “ES”-Version unterscheidet sich grundlegend von der verwendeten OpenGL-Version für Computer, und die OpenGL-Pipeline gibt es in der oben beschriebenen programmierbaren Form auch erst ab Version 3. Die offizielle Dokumentation zur Unterstützung von OpenGL war diesbezüglich wenig aussagekräftig gewesen.

Dass der Java Native Interface (JNI)-Treiber vom Grafikframework auf einem System nicht unterstützt wurde, ist insofern verwunderlich, da dieses sehr weit verbreitet und produktiv eingesetzt wird, z.B. in populären Computerspielen wie Minecraft. Da es nicht geloggt wurde, konnte nicht ermittelt werden, ob das Problem auf eine Inkompatibilität der Hardware oder des Betriebssystems zurückgeht.

Bei dem System mit 6 Gigabyte Arbeitsspeicher ging bei der Verarbeitung des großen Bildes der Arbeitsspeicher aus. Den Probanden wurde zwar empfohlen, der JVM 4 Gigabyte Speicher zuzuweisen, das war auf diesem System aber vermutlich nicht erfolgreich.

Bei der Mehrheit der Rechner, die die integrierte Prozessorgrafik verwendeten, war das Ergebnis der Verarbeitung des größeren Testbilds auf der GPU ein schwarzes Bild. Dieser Fehler trat auch im Rahmen der Entwicklung häufiger bei der Anwendung der Operation-Wise-Strategie auf, welche deutlich mehr Texturen gleichzeitig speichert als die Tile-Wise-Strategie. Auch wenn OpenGL keinen entsprechenden Fehlercode zurückgibt, ist zu vermuten, dass dieses Verhalten auf mangelnden Texturspeicher zurückzuführen ist.

Der einmal aufgetretene Fehler, dass das große Bild nicht im Dateibrowser gefunden werden konnte, konnte leider nicht reproduziert werden.

Zusätzlich wurden in der Anwendungsoberfläche kleine Fehler gefunden, speziell in der Zoom-Komponente. Diese wurden in der Folge behoben.

Nach Ausschluss der fehlerhaften Läufe konnten insgesamt von 13 Teilnehmenden Zeitmessungen zur Verarbeitung des großen Bilds, und von 20 Teilnehmenden Zeitmessungen zur Verarbeitung des kleineren Bilds ausgewertet werden, wobei einige Teilnehmende mehr als einen Durchlauf pro Bild und Konfiguration durchgeführt hatten (siehe Tabellen 5 und 6).

Um jeden Rechner gleich zu werten, wurden die Werte gemittelt, falls es mehrere Durchläufe gab. Für den Vergleich zwischen kleinem und großen Bild wurden außerdem nur die Datensätze mit einbezogen, bei denen die Verarbeitung beider Bilder fehlerfrei ablief, um den Einfluss von Störfaktoren wie der Art der Grafikkarte auszuschließen.

Tabelle 5: Durchläufe in der Stichprobe

Erfolgreiche Durchläufe	Beschleunigungsstrategie
46	GPU_TILE
45	MT
42	GPU_OP

Tabelle 6: Durchläufe pro Person und Konfiguration

N	Datei	Beschleunigungsstrategie	avg runs p.P.
13	Bild_20222_01.DNG	GPU_OP	1.46
13	Bild_20222_01.DNG	GPU_TILE	1.46
13	Bild_20222_01.DNG	MT	1.23
20	RAW-CANON-S30.dng	GPU_OP	1.15
20	RAW-CANON-S30.dng	GPU_TILE	1.35
20	RAW-CANON-S30.dng	MT	1.10

### 7.2.3 Einfluss von Größe und Verarbeitungsreihenfolge

Wie in Abb. 33 zu sehen, wurde bei der Verwendung der Grafikhardware ein Speedup gegenüber der CPU-Version mit Multithreading erreicht, der in etwa konsistent ist mit den von Lee et al. (2010) berichteten Werten.

Auch wenn, wie oben beschrieben, Tiling auf der getesteten Hardware nur sehr begrenzt zum Einsatz kam, erreichte eine Kachel-für-Kachel-Verarbeitung

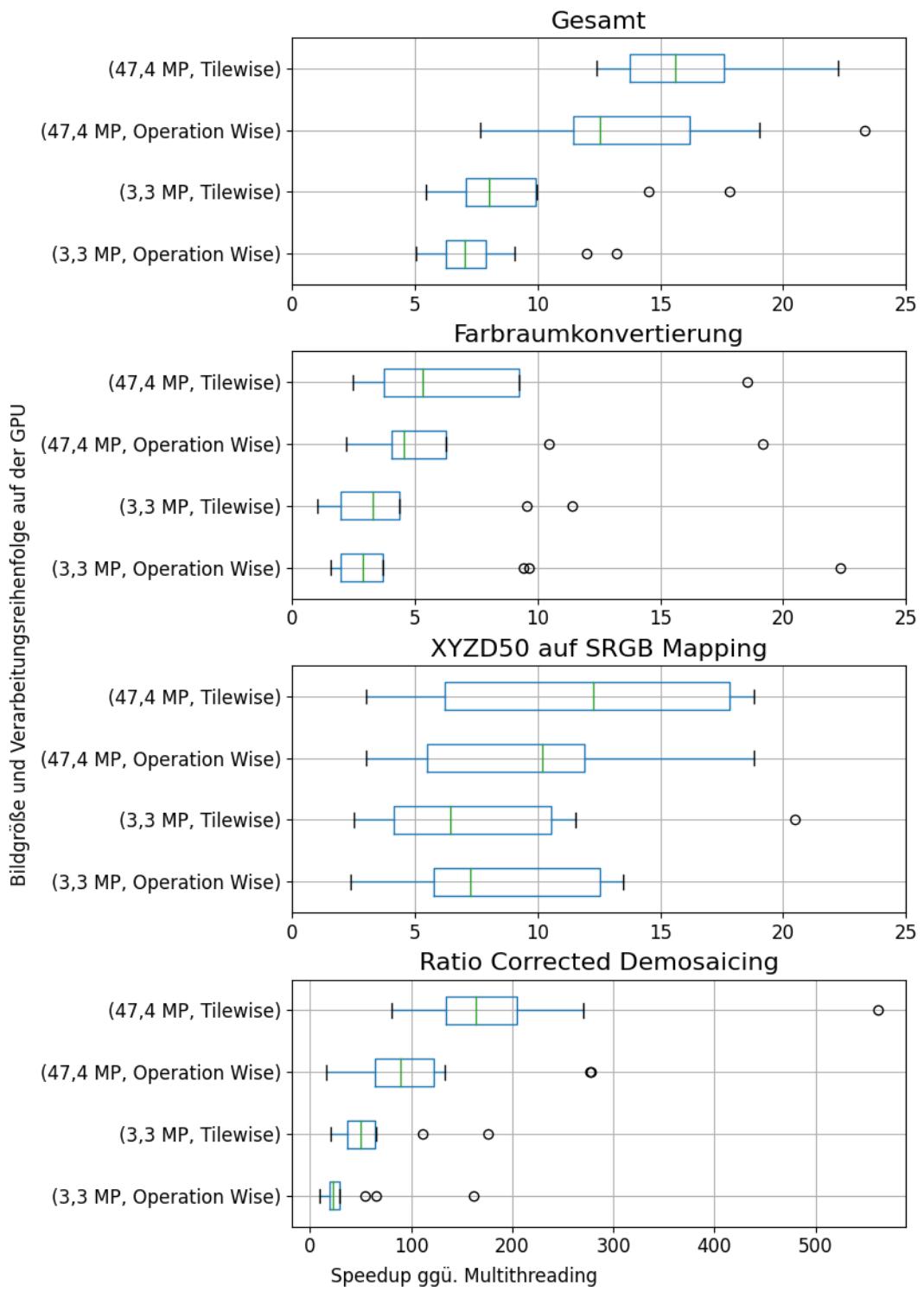


Abbildung 33: Speedup verschiedener Verarbeitungsschritte nach Bildgröße und Verarbeitungsreihenfolge, n=13

eine deutlich bessere Performance. Dies könnte unter anderem an der größeren Lokalität der Speicherzugriffe, aber auch dem geringeren Overhead durch die Allozierung von `GPUImage`-Objekten liegen.

Bei der Verarbeitung des größeren Bilds konnte ein größerer Speedup erreicht werden, was darauf hinweist, dass der parallele Teil des Programms stärker von der Größe der verarbeiteten Daten abhängt als der serielle Teil.

#### 7.2.4 Vergleich der Beschleunigung für die einzelnen Schritte

Beim Vergleich der Beschleunigung der einzelnen Verarbeitungsschritte fallen große Unterschiede auf: Bei der Farbraumtransformation fällt diese zum Beispiel geringer aus als bei der Transformation aus dem XYZD50- in das sRGB-Format. Dies lässt sich dadurch erklären, dass die erste Operation eine einfache Multiplikation des RGB-Vektors mit einer Matrix ist, während bei der zweiten Operation auch die Gammakorrektur, welche eine Potenzoperation enthält, durchgeführt wird. Gerade die Potenzoperation wird auf der GPU durch spezialisierte Hardware unterstützt.

Dass die Beschleunigung für den Schritt des Ratio Corrected Demosaicing derart groß ausfällt, liegt daran, dass der Algorithmus zu diesem Zeitpunkt noch nicht auf der CPU optimiert war - gerade dieses Ergebnis motivierte die weiter oben beschriebene Datenflussanalyse, welche den Unterschied schlussendlich erklärt. Im Verlauf der Arbeit konnte die Ausführungszeit auf der CPU um einen Faktor der Größenordnung 10 verringert werden, was die Ergebnisse konsistent mit der Beschleunigung der anderen Schritte macht.

Trotzdem bleibt die Frage, warum die Beschleunigung bezogen auf den Gesamtprozess hinter der Beschleunigung der einzelnen Schritte zurückbleibt. Um dies zu erklären, ist ein Blick auf die durchschnittliche Dauer der Verarbeitungsschritte je Strategie hilfreich: Nicht einbezogen in den Vergleich ist nämlich die Zeit, die benötigt wird, um das Bild in eine für die GPU verfügbare Textur umzuwandeln und wieder zurückzuverwandeln.

Tabelle 7: Durchschnittszeiten Verarbeitung des größeren Bilds mit GPU Tiling

Schritt	Beschreibung	ms
RawMapping		317.37
WhiteBalancing		134.42
PreProcessor		453.89
Initializing OpenGL Wrapper		81.68
Uploading image part	activating openGL context	25.74
Uploading image part	computing overlaps	0.00
Uploading image part	getting raster and runtime type-checking	0.05

Tabelle 7: Durchschnittszeiten Verarbeitung des größeren Bilds mit GPU Tiling

Schritt	Beschreibung	ms
Uploading image part	copying data into short array	198.32
Uploading image part	generating OpenGL Texture	0.95
Uploading image part	validating upload	1.89
Uploading image part	filling texture with data	193.68
uploading tile 0		390.53
RatioCorrectedDemosaicing		462.47
ColorSpaceTransformation		194.05
XYZD50ToSRGBMapping		180.37
downloading texture	sanity checks	46.68
downloading texture	allocating space and saving overlaps	39.32
downloading texture	downloading to buffer	1257.58
downloading texture	updating image contents	372.79
downloading tile 0		1694.21
uploading tile 1		165.50
downloading tile 1		142.00
ImageCroppingProcessor	after GPU	0.11
Total		4043.95

Tabelle 8: Durchschnittszeiten Verarbeitung des größeren Bilds mit Multithreading

Schritt	Beschreibung	ms
RawMapping		395.63
WhiteBalancing		157.19
PreProcessor		593.13
RatioCorrectedDemosaicing		59225.50
DemosaicingProcessor		59474.06
ImageCroppingProcessor		0.06
ColorSpaceTransformation		764.50
XYZD50ToSRGBMapping		1445.31
PostProcessor		2210.44
Total		62686.44

### 7.3 Benchmarks

Um die Optimierungen auf der CPU zu verifizieren und die idealen Kachelgrößen für die Tiling-Strategie zu ermitteln, wurden extensive Benchmarks auf

einem Testgerät des Autors sowie auf einem gemieteten Testgerät ausgeführt.

Hierfür wurde auf einem System ohne anderweitige Last und - im Falle des nicht online gemieteten Testgeräts - Internetverbindung die Kommandozeilenversion von Jeniffer2 mithilfe eines Skripts ausgeführt. Jede Konfiguration wurde mindestens 5 mal getestet, wobei sich die Konfigurationen immer abwechselten, um den Einfluss lokaler veränderlicher Faktoren wie der Umgebungstemperatur zu verringern. Diese Überlegung ist insofern relevant, da ein Durchlauf aller Konfigurationen ca. 2 Stunden dauerte.

### 7.3.1 Testgeräte

Beim bereits mehrfach erwähnten Testgerät des Autors handelt es sich um ein ThinkPad W530 mit einem i7-Prozessor der Ivy Bridge-Mikroarchitektur mit 4 physischen und 8 logischen Kernen und 8 Gigabyte Arbeitsspeicher. Das Gerät ist zwar aus dem Jahr 2013 und somit etwas älter, von der Größe des Arbeitsspeichers, der Prozessorfrequenz und der Anzahl der Kerne her liegt es aber im Vergleich mit den in der Empirie erhobenen Geräten noch im Mittelfeld. Neben einer integrierten Grafik vom Typ Intel HD Graphics 4000 verfügt es auch über eine externe NVIDIA-Quadro-K2000M-Grafikkarte mit 2GB Grafikspeicher. Sofern nicht anders beschrieben, wurden die Tests unter Ubuntu 22.04 mit der integrierten Grafikkarte durchgeführt. Auf der ansonsten verwendeten Windows-Partition läuft Windows 10 mit den neuesten Updates.

Beim online gemieteten Testgerät handelt es sich um einen Apple Mac Mini M1. Dieser verfügt ebenfalls über 8 Gigabyte Arbeitsspeicher, und einen Prozessor der Apple Silicon-Mikroarchitektur, welcher aus 4 niedrig getakteten und 4 höher getakteten physischen Kernen besteht. Als Betriebssystem läuft auf dem Gerät aus dem Jahr 2020 Apples macOS 12 Monterey.

Auf beiden Geräten wurde eine aktuelle Version des OpenJDK JREs verwendet.

### 7.3.2 Testbilder

Als Testbilder wurden sowohl die beiden Testbilder aus der Empirie mit 3,3 und 47,4 Megapixeln (Abb. 34 und 35) als auch ein noch größeres Testbild mit einer Auflösung von 60,1 Megapixeln (Abb. 36) verwendet.

### 7.3.3 Kachelgröße und Multithreading-Strategie

Um den Speicherfußabdruck des Programms zu reduzieren und die Berechnung durch die effiziente Verwendung von Caches zu beschleunigen, wurde die Möglichkeit getestet, die Bilder in Kacheln zu verarbeiten. Hierfür wurde für alle Demosaicing-Algorithmen die Aufteilung in Kacheln der Seitenlängen 2048, 1024, 512, 256, 128, 64 und 32 getestet. Für alle DLMMSE-Algorithmen



Abbildung 34: 3,3 Megapixel Testbild, entwickelt mit RCD



Abbildung 35: 47,4 Megapixel Testbild, entwickelt mit RCD und herunterskaliert

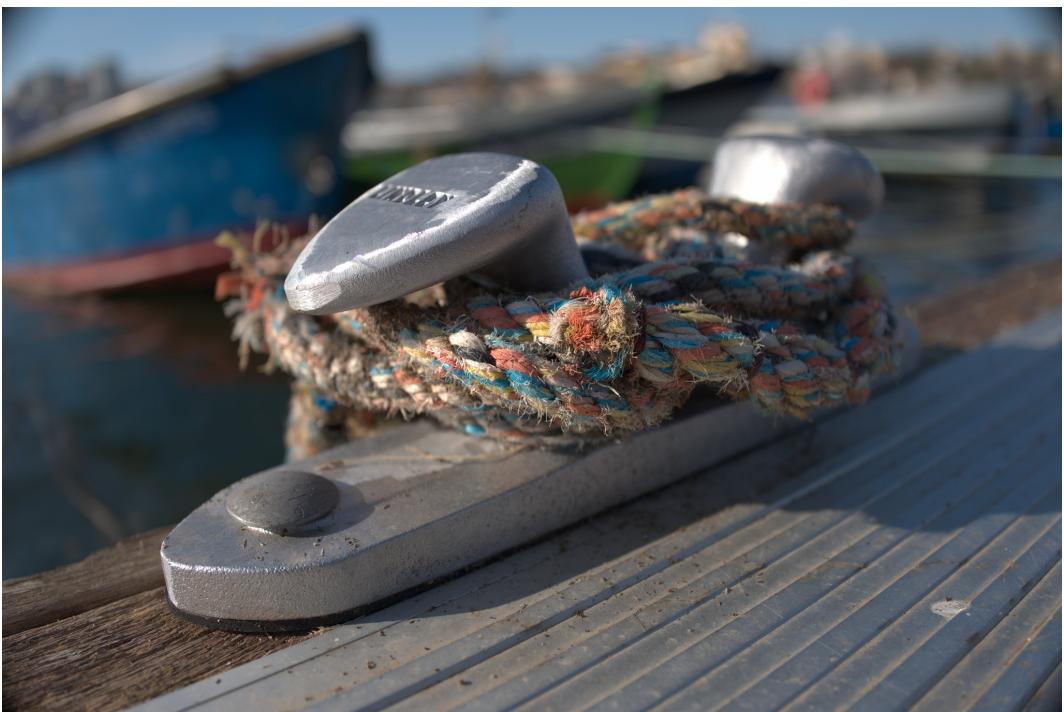


Abbildung 36: 61,1 Megapixel Testbild, entwickelt mit RCD und herunterskaliert

wurde die Kachelgröße von 32 nicht getestet, da hier ein Overhead von mehr als dem 800% entstehen würde und im Code eine Sonderbehandlung nötig wäre. So kann die optimale Balance zwischen dem Overhead, der durch die doppelten Berechnungen an den für die Korrektheit der Ergebnisse notwendigen Überlappungen der Kacheln entsteht, und dem erwarteten Speedup durch die größere Speicherlokalität ermittelt werden.

Zusätzlich wurde mit verschiedenen Multithreading-Strategien getestet: Es kann sowohl die Berechnung der einzelnen Kacheln auf Threads verteilt werden (“Thread-distributed CPU Tiling”) als auch bei der Verarbeitung innerhalb der Kacheln Multithreading angewendet werden (“with MT”). Da die Verteilung des Multithreadings auch für die Pre- und Postprocessor-Operationen interessant sein kann, wurden sie auch für diese Verarbeitungsschritte getestet - allerdings werden hier keine Kacheln, sondern die gleiche Menge an Pixeln als Abschnitte des als Array vorliegenden Bilds verarbeitet, da die zweidimensionale Lokalität hier keine Rolle spielt.

#### 7.3.4 Übersicht nach Bildgröße und Gerät

In den Tabellen 9 und 10 sind die durchschnittlichen Ausführungszeiten in Millisekunden für die verschiedenen Demosaicing-Algorithmen sowie die Pre- und Postprocessingschritte aufgeführt. Aufgeschlüsselt sind die Zeiten außerdem nach Beschleunigungsstrategie, Testgerät und Bildgröße, wobei die farblichen Schattierungen zum relativen Vergleich innerhalb der Daten für ein Bild und Gerät dienen. Bei den Strategien, bei denen unterschiedliche Kachelgrößen getestet wurden, wurde in dieser Tabelle die optimale Konfiguration dargestellt - diese ist jedoch nicht für alle Bildgrößen und vor allem nicht für beide Geräte gleich, wie später im Detail aufgeschlüsselt wird.

Interessant ist für die Einordnung zunächst, dass auf dem ThinkPad die Verarbeitungszeit im Preprocessor mit der alten Version nahe dem Durchschnitt des Feldtests liegt. Auch für RCD wird auf dem Gerät nur weniger als eine Sekunde mehr Zeit benötigt als der Durchschnitt, welcher bei knapp einer Minute liegt. Nur für den Postprocessing-Schritt wird mehr als die eineinhalbseitige Zeit benötigt. Eventuell hängt das damit zusammen, dass auf dem ThinkPad im Gegensatz zu den jüngeren Geräten der Stichprobe noch keine kombinierten Multiplikations-Additions-Instruktionen zur Verfügung stehen, welche bei den Matrixmultiplikationen an dieser Stelle von Vorteil sein könnten. Da die Benchmarks in einer kontrollierten Umgebung durchgeführt wurden, während beim Feldtest möglicherweise noch andere Programme sowie die grafische Benutzeroberfläche ausgeführt wurden, kann man das ThinkPad von der Leistung her im unteren Mittelfeld einordnen.

Der Mac Mini M1 ist bei den Demosaicing-Algorithmen grob dreimal so schnell wie das ThinkPad, allerdings haben bei den RCD-, DLMMSE- und

DLMMSE+RCD-Algorithmen die Beschleunigungsstrategien auf dem ThinkPad einen größeren Effekt: Mit einfachen `for`-Schleifen brauchen diese Algorithmen ungefähr das siebenfache der Zeit auf dem Mac Mini. Beim Preprocessing ist der Mac Mini nur doppelt so schnell und für das Postprocessing benötigt er sogar mehr als doppelt so lang wie das ThinkPad. Die testweise Verwendung eines anderen JDKs ändert daran auch nichts, es ist aber zu vermuten, dass die Java-interne `Math`-Bibliothek mit der hier verwendeten wichtigen Potenzfunktion noch nicht auf die ARM-Prozessorarchitektur des Mac Mini optimiert wurde.

Was bei den unoptimierten Versionen der RCD-, DLMMSE- und DLMMSE+RCD-Algorithmen ebenfalls auffällt, ist, dass das ThinkPad weniger Zeit braucht, sie auf dem größten Bild auszurechnen, als auf dem mittelgroßen. Auch wenn im Disassembly des auf dem ThinkPad generierten Codes keine C2-Kompilation nachgewiesen werden konnte, gibt es auch innerhalb des niedrigeren C1-Tiers verschiedene Optimierungsstufen, insofern ist davon auszugehen, dass für die Verarbeitung des größeren Inputs stärker optimierter Code generiert wurde.

Die wichtigste Erkenntnis aus dieser Übersicht ist aber die optimale Beschleunigungsstrategie. Für fast alle Demosaicing-Algorithmen sind hier Thread-distributed CPU Tiling für das mittlere und das große Bild und Thread-distributed CPU Tiling with MT für das kleine Bild am schnellsten.

Einiger Ausreißer auf dem ThinkPad ist der Bikubische Algorithmus, bei dem auch die mittleren und großen Bilder mit Multithreading in beiden Dimensionen am schnellsten verarbeitet werden. Auf dem Mac Mini sind mit dieser Strategie nicht nur der Bikubische, sondern auch die RCD- und DLMMSE+RCD-Algorithmen unabhängig von der Bildgröße am schnellsten.

Beim Postprocessing ist ebenfalls fast durchgehend Thread-distributed CPU Tiling with MT optimal, wobei der Unterschied zu normalem Multithreading auf dem Thinkpad minimal ist.

Was das Preprocessing angeht, unterscheiden sich die Ergebnisse zwischen den Geräten und Bildgrößen am meisten. Das Umordnen der Operationen in eine einzige Schleife hat sich bei der Verarbeitung des größten Bildes sogar leicht negativ auf die Performance ausgewirkt. Während auf dem Thinkpad in der Neuimplementation für das kleine Bild CPU Tiling with MT und ansonsten Thread-distributed CPU Tiling optimal sind, sind es auf dem Mac Mini respektive Thread-distributed CPU Tiling und Thread-distributed CPU Tiling with MT.

Alg.	Beschleunigungsstrategie	ThinkPad W530			Mac Mini M1		
		3.3 MP	47.4 MP	60.5 MP	3.3 MP	47.4 MP	60.5 MP
NONE	Multithreading-OLD	162	852	1090			
	None	202	866	1091	32	199	272
	Multithreading	96	895	1061	28	187	248
	CPU Tiling	152	864	1170	40	308	375
	CPU Tiling with MT	92	655	996	28	185	252
	Thread-distributed CPU Tiling	128	379	453	33	110	136
	Thread-distributed CPU Tiling with MT	78	554	710	26	167	218
NEAREST NEIGHBOR	Multithreading-OLD	203	988	1280			
	None	223	1001	1304	41	244	333
	Multithreading	103	1057	1062	30	202	278
	CPU Tiling	219	1104	1432	50	357	461
	CPU Tiling with MT	102	793	1065	34	214	287
	Thread-distributed CPU Tiling	150	560	640	36	132	160
	Thread-distributed CPU Tiling with MT	97	644	811	29	190	241
BILINEAR MEAN	Multithreading-OLD	259	1334	1562			
	None	256	1257	1534	50	352	464
	Multithreading	116	1067	1240	38	251	326
	CPU Tiling	226	1288	1662	59	465	585
	CPU Tiling with MT	118	854	1098	37	255	343
	Thread-distributed CPU Tiling	162	558	642	48	163	198
	Thread-distributed CPU Tiling with MT	112	691	903	35	229	296
BILINEAR MEDIAN	Multithreading-OLD	311	1621	1994			
	None	345	1972	3369	131	1477	2068
	Multithreading	182	1246	1637	60	488	651
	CPU Tiling	360	2394	3174	135	1561	2106
	CPU Tiling with MT	172	1042	1464	64	465	644
	Thread-distributed CPU Tiling	298	938	1109	78	392	510
	Thread-distributed CPU Tiling with MT	168	939	1203	54	442	596
BICUBIC	Multithreading-OLD	538	2047	2473			
	None	604	2995	4978	209	1278	1607
	Multithreading	276	1648	2057	99	504	634
	CPU Tiling	710	4082	5180	231	1477	1871
	CPU Tiling with MT	284	1524	1837	97	508	645
	Thread-distributed CPU Tiling	783	1612	1858	218	553	638
	Thread-distributed CPU Tiling with MT	272	1341	1566	98	459	584
MALVAR HE CUTLER	Multithreading-OLD	460	1665	2110			
	None	394	1843	2308	71	565	757
	Multithreading	148	1198	1576	53	316	402
	CPU Tiling	262	1950	2430	80	757	955
	CPU Tiling with MT	151	1081	1442	48	309	424
	Thread-distributed CPU Tiling	243	772	928	63	260	303
	Thread-distributed CPU Tiling with MT	141	908	1116	45	284	364
HAMILTON ADAMS	Multithreading-OLD	510	2402	2997			
	None	415	2111	2531	88	715	934
	Multithreading	152	1381	1725	56	399	511
	CPU Tiling	292	2053	2592	92	834	1064
	CPU Tiling with MT	156	1163	1554	56	396	554
	Thread-distributed CPU Tiling	250	755	939	87	281	346
	Thread-distributed CPU Tiling with MT	138	1013	1310	49	368	474
PPG	Multithreading-OLD	657	2853	3458			
	None	568	3105	4271	151	1485	2035
	Multithreading	194	1665	2165	77	564	727
	CPU Tiling	460	3347	4348	162	1590	2171
	CPU Tiling with MT	203	1503	2048	74	560	768
	Thread-distributed CPU Tiling	349	1213	1383	115	438	540
	Thread-distributed CPU Tiling with MT	172	1338	1674	66	517	691

Tabelle 9: Vergleich der Ausführungszeiten nach Beschleunigungsstrategie, Bildgröße und Gerät, Teil 1

		ThinkPad W530			Mac Mini M1		
		3.3 MP	47.4 MP	60.5 MP	3.3 MP	47.4 MP	60.5 MP
Alg. Beschleunigungsstrategie	Multithreading-OLD	6564	59876	74179			
	None	4669	40234	34096	554	5694	7302
	Multithreading	511	4472	5467	176	1291	1598
	CPU Tiling	1608	13726	17741	565	4674	5990
	CPU Tiling with MT	515	4428	5616	177	1291	1730
	Thread-distributed CPU Tiling	840	<b>3815</b>	<b>4466</b>	253	1338	1650
	Thread-distributed CPU Tiling with MT	<b>414</b>	3899	4846	<b>142</b>	<b>1215</b>	<b>1548</b>
RCD DLMMSE CODE	Multithreading-OLD	3184	36018	47206			
	None	2123	22877	15860	266	3418	4390
	Multithreading	490	4647	5774	198	1395	1655
	CPU Tiling	698	7012	9084	273	2888	3751
	CPU Tiling with MT	480	4267	5835	178	1408	1907
	Thread-distributed CPU Tiling	626	<b>2486</b>	<b>2951</b>	174	<b>859</b>	<b>1065</b>
	Thread-distributed CPU Tiling with MT	<b>403</b>	3817	4944	<b>135</b>	1226	1569
DLMMSE PAPER	Multithreading-OLD	3043	35632	49060			
	None	2046	23304	15964	267	3394	4207
	Multithreading	498	4410	5834	194	1366	1675
	CPU Tiling	699	6870	9086	269	2887	3755
	CPU Tiling with MT	492	4234	5631	185	1457	1973
	Thread-distributed CPU Tiling	612	<b>2518</b>	<b>3012</b>	164	<b>882</b>	<b>1062</b>
	Thread-distributed CPU Tiling with MT	<b>373</b>	3847	5012	<b>130</b>	1214	1568
DLMMSE RCD CODE	Multithreading-OLD	7678	81050	105073			
	None	6057	67321	38664	572	7106	9121
	Multithreading	706	6581	7998	273	2046	2522
	CPU Tiling	1618	17436	21996	641	6391	8161
	CPU Tiling with MT	716	6540	8493	267	2070	2769
	Thread-distributed CPU Tiling	1239	<b>5303</b>	<b>6645</b>	388	1944	2460
	Thread-distributed CPU Tiling with MT	<b>581</b>	5987	7642	<b>193</b>	<b>1864</b>	<b>2347</b>
DLMMSE RCD PAPER	Multithreading-OLD	7453	81304	104902			
	None	4763	53141	38845	571	7104	9122
	Multithreading	691	6511	8472	269	2010	2591
	CPU Tiling	1566	17059	22328	642	6423	8251
	CPU Tiling with MT	714	6575	8457	271	2076	2832
	Thread-distributed CPU Tiling	1190	<b>5505</b>	<b>6242</b>	412	2004	2418
	Thread-distributed CPU Tiling with MT	<b>564</b>	5783	7230	<b>188</b>	<b>1853</b>	<b>2382</b>
Pre-processing	Multithreading-OLD	275	643	<b>740</b>			
	None	214	2334	3199	112	1397	1729
	Multithreading	180	626	796	100	321	388
	CPU Tiling	222	2401	3213	115	1384	1716
	CPU Tiling with MT	<b>133</b>	638	818	70	324	392
	Thread-distributed CPU Tiling	150	<b>601</b>	753	<b>66</b>	350	421
	Thread-distributed CPU Tiling with MT	176	627	799	102	<b>312</b>	<b>381</b>
Post-processing	Multithreading-OLD	506	3694	4538			
	None	563	6929	8872	1460	21529	27681
	Multithreading	<b>133</b>	1368	1752	250	3532	4577
	CPU Tiling	530	6815	8024	1440	21353	27280
	CPU Tiling with MT	136	1410	1822	250	3354	4335
	Thread-distributed CPU Tiling	202	1470	1843	272	3289	4234
	Thread-distributed CPU Tiling with MT	138	<b>1362</b>	<b>1732</b>	<b>244</b>	<b>3284</b>	<b>4221</b>

Tabelle 10: Vergleich der Ausführungszeiten nach Beschleunigungsstrategie, Bildgröße und Gerät, Teil 2

### 7.3.5 Optimale Kachelgröße und Strategie für große Bilder

Da sich die Ergebnisse für das große und das mittlere Bild stark ähneln, wurden für die Diskussion der optimalen Kachelgröße bei eher großen Bildern die Daten für das mittlere Bild ausgewählt.

Tabellen 11 und 12 zeigen die Ausführungszeit in Millisekunden abhängig von Algorithmus, Multithreadingstrategie und Kachelgröße. Zusätzlich zum Overhead, der durch die doppelte Berechnung von Werten an den Kachelrändern anfällt, ist in den Tabellen der für die Zwischenergebnisse einer Kachel benötigte Speicherplatz angegeben. Das Minimum sind hierbei 4 Werte pro Pixel: Der Quellwert nach der Umwandlung aus dem `BufferedImage` in ein Floating-Point-Format und die drei resultierenden Farbwerte. Die Minimalbelegung gibt an, auf wieviele Werte pro Pixel maximal gleichzeitig in einer Schleife bzw. einem Arbeitsschritt lesend oder schreibend zugegriffen werden muss. Die Maximalbelegung gibt an, wieviele Werte pro Pixel insgesamt in Arrays gespeichert werden, also wieviel Speicher belegt wird, wenn innerhalb der Berechnung keine Garbage Collection stattfindet. Um Platz zu sparen, wurden Algorithmen, bei denen diese Werte übereinstimmen, untereinander zusammengefasst. Für die Cache-Belegung ist das Minimum relevanter, weil Speicher nicht direkt im Cache alloziert wird, und unbenötigte Daten im Zweifel automatisch verdrängt werden.

Bei Thread-distributed CPU Tiling werden die Kacheln auf unterschiedliche CPU-Kerne verteilt, aber die Verarbeitung einer Kachel bleibt sequentiell auf dem gleichen Kern. Da jeder Kern über einen eigenen Level 2 Cache verfügt, schneidet für diese Strategie die Kachelgröße am besten ab, bei der die Zwischenergebnisse einer Kachel sicher in den L2-Cache passen, d.h. etwas Platz für z.B. den Programmcode selbst und andere Prozesse lassen. Die schnellere Zugriffszeit gleicht auf dem ThinkPad einen Overhead von bis zu 63,8% beim DLMSE+RCD-Algorithmus aus. Auf dem Mac Mini ist bei den einfacheren Algorithmen sogar noch eine Kachelgröße kleiner als die maximal passende Größe ideal, möglicherweise, weil auf einem Kern mehrere Threads ausgeführt werden.

Ein Zuschnitt der Kacheln auf den Level 1 Cache kann hingegen den Zuwachs an Overhead nicht mehr durch eine schnellere Zugriffszeit ausgleichen, oder die Speicherzugriffszeit ist an dieser Stelle nicht mehr der entscheidende Faktor.

Ein derartiger Fokus auf die Optimierung der Zeiten für den Speicherzugriff ist deswegen effektiv, weil diese Latenzen in den meisten Fällen nicht durch Pre-Fetching verborgen werden können, da die Berechnung pro Datenpunkt im Verhältnis nicht lange genug dauert. Die Implementation des bikubischen Demosaicing enthält aber im Verhältnis zu den anderen Algorithmen viel Logik, weshalb die Rechenleistung der Kerne anscheinend am effektivsten genutzt wird,

wenn auch die Pixel innerhalb der Kacheln auf Threads verteilt werden. Auch der RCD-Algorithmus enthält im Verhältnis zu den zugegriffenen Daten teils sehr rechenintensive Schritte. Der Unterschied in der Speicherlatenz zwischen den Caches und dem Hauptspeicher ist auf dem Mac Mini M1 anscheinend gering genug, dass auch hier der Einfluss der Zugriffszeit genügend abnimmt, dass Multithreading innerhalb einer Kachel etwas effektiver als die sequentielle Ausführung wird.

Wird Multithreading innerhalb der Kacheln betrieben, sind größere Kacheln deswegen vorteilhaft, da sich der Synchronisationsoverhead des Multithreadings auf eine größere Anzahl an Pixeln verteilt. Dass das Bild dann überhaupt noch in Kacheln aufgeteilt wird, ist aber trotzdem effektiv: Für den zweidimensionalen Zugriff auf ein Bild, welches in Row-Major-Ordnung gespeichert ist, ist nämlich ein weniger breites Bild vorteilhaft für die Lokalität der Datenzugriffe.

Wie in Abb. 37 dargestellt, wird z.B. für die Berechnung der XY- und PQ-Gradienten im RCD-Algorithmus auf Pixel bis zu 4 Reihen über und bis zu 4 Reihen unter dem Zielpixel zugegriffen. Da die Pixel horizontal auch nur bis zu 4 Pixel entfernt sind, ist es somit vorteilhaft, wenn 8 Reihen plus 9 Pixel gleichzeitig im Cache gehalten werden können. Bei dem mittelgroßen Bild, welches 8424 Pixel breit ist, sind das also  $(8424+9)*8*4 >= 269KB$  und damit mehr, als in den Level 1 Cache beider Systeme passt. Die bei einer Kachel-Seitenlänge von 2048 Pixeln benötigten  $(2048 + 9) * 8 * 4 >= 65KB$  passen hingegen sogar fast in den L1-Cache der kleineren im Mac Mini verbauten Kerne.

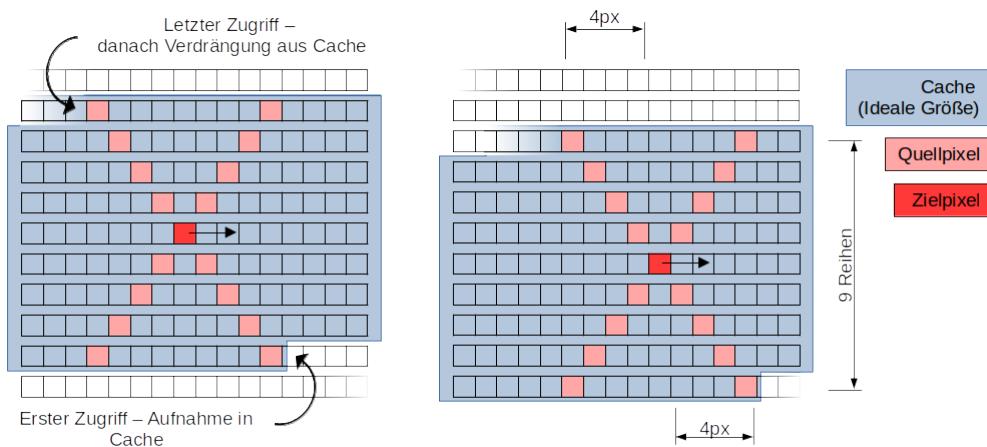


Abbildung 37: Ideale Lebensdauer Pixel im Cache bei der Berechnung des PQ-Gradienten im RCD- und DLMMSE+RCD-Algorithmus bei einem 16px breiten Bild

**ThinkPad W530 – 47,4 MP Bild**

Farben Cache-Größen:

too big L3-Cache: 1x 6MB L2-Cache: 4x 1MB L1-Cache: 4x 128KB

Alg.	Metrik/Kachelgröße	2048	1024	512	256	128	64	32
	Overlap: 0px -> Overhead (Prozent)	0	0	0	0	0	0	0
NONE	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	Thread-distributed CPU Tiling (ms)	1262	998	708	460	379	445	405
	Thread-distributed CPU Tiling with MT	642	636	636	554	559	644	696
NEAREST NEIGHBOR	Overlap: 2px -> Overhead (Prozent)	0.4	0.8	1.6	3.2	6.6	13.8	30.6
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	Thread-distributed CPU Tiling (ms)	1265	992	892	646	576	560	565
	Thread-distributed CPU Tiling with MT	700	678	750	644	693	781	892
BIL. MEAN	Thread-distributed CPU Tiling (ms)	1274	1039	900	610	558	559	627
	Thread-distributed CPU Tiling with MT	763	761	821	691	737	851	944
MHC MEDIAN	Thread-distributed CPU Tiling (ms)	1676	1433	1307	1056	938	978	1044
	Thread-distributed CPU Tiling with MT	1014	1061	1094	939	1026	1040	1211
	Thread-distributed CPU Tiling (ms)	1486	1222	1076	780	772	820	887
	Thread-distributed CPU Tiling with MT	999	980	1030	908	928	1143	1259
BICUBIC	Overlap: 4px -> Overhead (Prozent)	0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	Thread-distributed CPU Tiling (ms)	2543	2515	2362	2060	1612	1714	1928
	Thread-distributed CPU Tiling with MT	1380	1447	1487	1341	1425	1648	2048
HAMILTON ADAMS	Overlap: 4px -> Overhead (Prozent)	0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 5x	83.9 MB	21.0 MB	5.2 MB	1.3 MB	327.7 KB	81.9 KB	20.5 KB
	Thread-distributed CPU Tiling (ms)	1555	1345	1131	755	774	855	973
	Thread-distributed CPU Tiling with MT	1182	1112	1156	1013	1082	1417	1714
PPG	Thread-distributed CPU Tiling (ms)	1908	1764	1746	1306	1213	1237	1584
	Thread-distributed CPU Tiling with MT	1362	1378	1504	1338	1353	1748	2359
RCD	Overlap: 10px -> Overhead (Prozent)	2	4	8.3	17.7	40.5	111.6	611.1
	min. Belegung: 7x	117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
	max. Belegung: 8x	134.2 MB	33.6 MB	8.4 MB	2.1 MB	524.3 KB	131.1 KB	32.8 KB
	Thread-distributed CPU Tiling (ms)	7622	6765	5305	4138	3815	5192	15090
	Thread-distributed CPU Tiling with MT	3899	4049	4447	4200	5146	7268	27913
DLMSE CODE	Overlap: 12px -> Overhead (Prozent)	2.4	4.9	10.1	21.8	51.5	156	1500
	min. Belegung: 6x	100.7 MB	25.2 MB	6.3 MB	1.6 MB	393.2 KB	98.3 KB	24.6 KB
	max. Belegung: 13x	218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
	Thread-distributed CPU Tiling (ms)	4337	3915	3034	2661	2486	3461	
	Thread-distributed CPU Tiling with MT	3997	4371	3942	3817	4343	7753	
PAPER	Thread-distributed CPU Tiling (ms)	4407	3936	3141	2545	2518	3600	
	Thread-distributed CPU Tiling with MT	4145	4070	4234	3847	4679	7306	
DLMSE RCD CODE	Overlap: 14px -> Overhead (Prozent)	2.8	5.7	11.9	26.1	63.8	216	6300
	min. Belegung: 7x	117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
	max. Belegung: 13x	218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
	Thread-distributed CPU Tiling (ms)	8658	8180	6519	5303	5467	9034	
	Thread-distributed CPU Tiling with MT	5987	6384	6302	6208	7512	15753	
PAPER	Thread-distributed CPU Tiling (ms)	8909	8005	6439	5512	5505	9025	
	Thread-distributed CPU Tiling with MT	5783	5992	5832	6094	7838	17488	

Tabelle 11: Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 47,4 MP Bild, ThinkPad W530

Mac Mini M1 – 47,4 MP Bild		too big	L2-Cache (groß): 4x12MB		L2-Cache (klein): 4x 4MB		L1-Cache: (4x 64KB) + 4x128 KB		
Alg.	Metrik/Kachelgröße		2048	1024	512	256	128	64	32
NONE	Overlap: 0px -> Overhead (Prozent)		0	0	0	0	0	0	0
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	Thread-distributed CPU Tiling (ms)		287	186	150	126	110	160	204
NEAREST NEIGHBOR	Thread-distributed CPU Tiling with MT		172	186	187	167	191	209	238
	Overlap: 2px -> Overhead (Prozent)		0.4	0.8	1.6	3.2	6.6	13.8	30.6
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
BILINEAR	Thread-distributed CPU Tiling (ms)		277	221	170	138	132	166	206
	Thread-distributed CPU Tiling with MT		190	204	202	195	216	246	306
	Thread-distributed CPU Tiling (ms)		301	238	201	172	163	202	229
	Thread-distributed CPU Tiling with MT		236	242	249	229	259	292	364
MHC MEDIAN	Thread-distributed CPU Tiling (ms)		555	460	445	414	392	430	508
	Thread-distributed CPU Tiling with MT		442	470	453	454	470	518	626
	Thread-distributed CPU Tiling (ms)		360	309	284	260	264	292	330
	Thread-distributed CPU Tiling with MT		284	311	302	292	321	346	426
BICUBIC	Overlap: 4px -> Overhead (Prozent)		0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	Thread-distributed CPU Tiling (ms)		676	694	694	705	553	601	704
HAMILTON ADAMS	Thread-distributed CPU Tiling with MT		459	488	488	490	516	592	813
	Overlap: 4px -> Overhead (Prozent)		0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 5x		83.9 MB	21.0 MB	5.2 MB	1.3 MB	327.7 KB	81.9 KB	20.5 KB
RCD	Thread-distributed CPU Tiling (ms)		397	370	342	292	281	353	410
	Thread-distributed CPU Tiling with MT		368	391	407	375	412	490	712
	Thread-distributed CPU Tiling (ms)		581	514	499	459	438	540	653
	Thread-distributed CPU Tiling with MT		517	540	548	534	596	682	976
DLMMSE CODE	Overlap: 10px -> Overhead (Prozent)		2	4	8.3	17.7	40.5	111.6	611.1
	min. Belegung: 7x		117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
	max. Belegung: 8x		134.2 MB	33.6 MB	8.4 MB	2.1 MB	524.3 KB	131.1 KB	32.8 KB
	Thread-distributed CPU Tiling (ms)		2127	1645	1415	1368	1338	1920	5845
PAPER	Thread-distributed CPU Tiling with MT		1215	1272	1282	1341	1591	2395	8188
	Overlap: 12px -> Overhead (Prozent)		2.4	4.9	10.1	21.8	51.5	156	1500
	min. Belegung: 6x		100.7 MB	25.2 MB	6.3 MB	1.6 MB	393.2 KB	98.3 KB	24.6 KB
	max. Belegung: 13x		218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
DLMMSE RCD CODE	Thread-distributed CPU Tiling (ms)		1108	961	898	859	950	1414	
	Thread-distributed CPU Tiling with MT		1226	1292	1314	1375	1665	2684	
	Thread-distributed CPU Tiling (ms)		1097	2626	904	882	945	1418	
	Thread-distributed CPU Tiling with MT		1214	1254	1295	1332	1623	2701	
PAPER	Overlap: 14px -> Overhead (Prozent)		2.8	5.7	11.9	26.1	63.8	216	6300
	min. Belegung: 7x		117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
	max. Belegung: 13x		218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
	Thread-distributed CPU Tiling (ms)		2572	2079	2008	1944	2151	3714	
PAPER	Thread-distributed CPU Tiling with MT		1864	1936	1998	2124	2798	5130	
	Thread-distributed CPU Tiling (ms)		2534	2127	2058	2004	2141	3706	
	Thread-distributed CPU Tiling with MT		1853	1920	1997	2142	2764	5034	

Tabelle 12: Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 47,4 MP Bild, Mac Mini M1

### 7.3.6 Optimale Kachelgröße und Strategie für kleine Bilder

Für das kleinste Bild waren Strategien mit Multithreading innerhalb der Kacheln am erfolgreichsten. Auch wenn sich der erreichte Compilation Tier der JVM auf beiden Geräten nicht von dem mit größeren Bildern erreichten unterscheidet (C2 auf dem Mac Mini, C1 auf dem ThinkPad), gibt es innerhalb der Tiers unterschiedlich starke Optimierungsstufen. Es ist auch möglich, dass die JVM aufgrund der geringeren Datenmenge weniger Zeit in Profiling investieren konnte, oder dass andere Faktoren wie der Branch-Predictor der CPU oder das Pre-Fetching der Daten eine gewisse “Aufwärmzeit” benötigen.

Wie in Tabellen 13 und 14 zu sehen, stimmt die ideale Kachelgröße auf den Geräten nur bei den komplexeren RCD, DLMMSE und DLMMSE+RCD-Algorithmen überein. Da das kleine Bild nur 2144x1560 Pixel misst, sind die idealen 1024 Pixel Seitenlänge die größte Größe, bei der tatsächlich auch 4 volle Kacheln entstehen.

Bei den anderen Algorithmen hingegen ist auf dem Mac Mini fast immer die Aufteilung in 2 Teile am schnellsten, während auf dem ThinkPad teilweise auch kleinere Stücke ideal sind, wobei hier die ideale Größe keiner klaren Logik zu folgen scheint. Ein Rückblick auf die Übersichtstabelle 9 zeigt aber, dass die Kachelaufteilung auf beiden Geräten nur einen geringen Vorteil bietet. Das liegt wohl auch daran, dass bei diesen Algorithmen nur ein kleinerer Bereich um das Zielpixel benötigt wird, und so eine ideale Zugriffslokalität in einer Schleife bereits mit der ursprünglichen Bildbreite gegeben ist. Zu beachten ist außerdem, dass beim Multithreading innerhalb einer Kachel keine verlässliche Aussage getroffen werden kann, ob die Threads auf dem gleichen (physischen) CPU-Kern ausgeführt werden.

### 7.3.7 Pre- und Postprocessing

Für die Beschleunigung der Pre- und Postprocessing-Schritte ist die zweidimensionale Lokalität uninteressant, und es werden auch keine Zwischenergebnisse gespeichert, deren Platzbedarf durch die Bild- oder Kachelgröße beeinflusst werden könnte.

Insofern war zu erwarten, dass die Ausführungszeit der auf Abschnitte verteilten Versionen höchstens marginal besser ist. Wie in Tabellen 17 und 18 zu sehen, trifft dies auch oft zu.

Beim Postprocessing auf dem Mac Mini ist es allerdings so, dass bei den größeren Bildern die Aufteilung in besonders kleine Abschnitte einen merklichen Vorteil bringt. Möglicherweise sorgt die stärkere Strukturierung des Multithreading für einen geringeren Thread-Synchronisationsoverhead zwischen den unterschiedlich schnellen Kernen.

Dass beim Preprocessing des kleinsten Bilds auf beiden Geräten eine Aufteilungsstrategie mit Multithreading in nur einer Dimension ideal ist, könnte daran liegen, dass so der Overhead der Verteilung der Arbeit auf die einzelnen Threads reduziert wird. Interessant ist, dass beim größten Bild die alte Version ohne Re-Ordering der Operationen am schnellsten ist. Möglicherweise liegt das am Speicherbedarf der Lookup-Tabelle im Raw-Mapping, welche dann nicht zusammen mit den Parametern des Weißabgleichs für dieses Bild in einen bestimmten Cache passt.

### 7.3.8 Beste Strategie insgesamt

Auch wenn die optimale Beschleunigungsstrategie und gegebenenfalls Kachelgröße für die einzelnen Algorithmen sich je nach Bildgröße und Gerät unterscheiden, lassen sich doch einige Heuristiken ableiten, die im Code umgesetzt werden können.

**7.3.8.1 Kleine Bilder** Bei kleinen Bildern und Algorithmen mit wenigen Zwischenergebnissen unterscheiden sich die optimalen Kachelgrößen zum Teil in einem Maße, dass die ideale Kachelgröße für ein Gerät auf dem anderen langsamer ist als überhaupt keine Aufteilung. Da der Vorteil der Aufteilung selbst mit der besten Konfiguration gering ist, bietet sich hier per Default die Anwendung von regulärem Multithreading an.

Bei den RCD, DLMMSE und DLMMSE+RCD-Algorithmen ist bei der Verarbeitung des kleinen Bilds hingegen eindeutig eine Kachelgröße von 1024x1024 Pixeln am schnellsten und reduziert die Ausführungszeit um ungefähr 25%. Da sich dieser Wert klar damit begründen lässt, dass mit ihm das Bild in mehrere volle Kacheln aufgeteilt wird, ist davon auszugehen, dass er auch auf anderen Geräten ideal ist.

ThinkPad W530 – 3,3 MP Bild								
Farben Cache-Größen:								
Alg.	Metrik/Kachelgröße	2048	1024	512	256	128	64	32
NONE	Overlap: 0px -> Overhead (Prozent)	0	0	0	0	0	0	0
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	CPU Tiling with MT (ms)	95	92	103	102	135	152	289
	Thread-distributed CPU Tiling with MT	98	99	78	92	105	97	137
	Overlap: 2px -> Overhead (Prozent)	0.4	0.8	1.6	3.2	6.6	13.8	30.6
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
NEAREST NEIGHBOR	max. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	CPU Tiling with MT (ms)	102	109	109	118	142	181	371
	Thread-distributed CPU Tiling with MT	110	100	103	97	115	117	157
	CPU Tiling with MT (ms)	118	119	124	120	156	182	380
	Thread-distributed CPU Tiling with MT	115	119	124	112	136	127	163
	CPU Tiling with MT (ms)	197	175	172	186	214	248	442
	Thread-distributed CPU Tiling with MT	174	168	182	179	189	181	267
MHC MEDIAN MEAN	CPU Tiling with MT (ms)	155	151	154	159	190	230	424
	Thread-distributed CPU Tiling with MT	150	145	141	144	171	157	212
	Overlap: 4px -> Overhead (Prozent)	0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	CPU Tiling with MT (ms)	302	323	284	317	298	376	673
	Thread-distributed CPU Tiling with MT	287	272	300	288	290	314	392
BICUBIC	Overlap: 4px -> Overhead (Prozent)	0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	CPU Tiling with MT (ms)	162	156	160	187	209	268	665
	Thread-distributed CPU Tiling with MT	148	138	147	161	166	174	253
	CPU Tiling with MT (ms)	214	203	212	215	250	307	725
	Thread-distributed CPU Tiling with MT	201	190	172	204	204	202	300
HAMILTON ADAMS	Overlap: 4px -> Overhead (Prozent)	0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x	67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 5x	83.9 MB	21.0 MB	5.2 MB	1.3 MB	327.7 KB	81.9 KB	20.5 KB
	CPU Tiling with MT (ms)	162	156	160	187	209	268	665
	Thread-distributed CPU Tiling with MT	148	138	147	161	166	174	253
	CPU Tiling with MT (ms)	214	203	212	215	250	307	725
	Thread-distributed CPU Tiling with MT	201	190	172	204	204	202	300
PPG RCD	Overlap: 10px -> Overhead (Prozent)	2	4	8.3	17.7	40.5	111.6	611.1
	min. Belegung: 7x	117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
	max. Belegung: 8x	134.2 MB	33.6 MB	8.4 MB	2.1 MB	524.3 KB	131.1 KB	32.8 KB
	CPU Tiling with MT (ms)	522	515	563	517	603	1021	5326
	Thread-distributed CPU Tiling with MT	469	414	439	489	556	691	2008
	Overlap: 12px -> Overhead (Prozent)	2.4	4.9	10.1	21.8	51.5	156	1500
	min. Belegung: 6x	100.7 MB	25.2 MB	6.3 MB	1.6 MB	393.2 KB	98.3 KB	24.6 KB
DLMMSE CODE	max. Belegung: 13x	218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
	CPU Tiling with MT (ms)	492	480	519	543	643	1382	
	Thread-distributed CPU Tiling with MT	452	403	414	481	494	733	
	CPU Tiling with MT (ms)	523	492	512	525	570	1367	
	Thread-distributed CPU Tiling with MT	420	373	398	476	500	719	
	Overlap: 14px -> Overhead (Prozent)	2.8	5.7	11.9	26.1	63.8	216	6300
	min. Belegung: 7x	117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
PAPER ER	max. Belegung: 13x	218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
	CPU Tiling with MT (ms)	717	716	735	752	886	2000	
	Thread-distributed CPU Tiling with MT	634	581	609	705	859	1280	
	CPU Tiling with MT (ms)	750	714	726	756	890	1998	
	Thread-distributed CPU Tiling with MT	641	564	629	716	795	1216	

Tabelle 13: Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 3,3 MP Bild, ThinkPad W530

Mac Mini M1 – 3,3MP Bild		too big	L2-Cache (groß): 4x12MB		L2-Cache (klein): 4x 4MB		L1-Cache: (4x 64KB) + 4x128 KB		
Alg.	Metrik/Kachelgröße		2048	1024	512	256	128	64	32
NONE	Overlap: 0px -> Overhead (Prozent)		0	0	0	0	0	0	0
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	CPU Tiling with MT (ms)		28	39	36	46	61	72	131
NEAREST NEIGHBOR	Thread-distributed CPU Tiling with MT		26	30	31	27	38	49	83
	Overlap: 2px -> Overhead (Prozent)		0.4	0.8	1.6	3.2	6.6	13.8	30.6
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
BIL. MEAN	CPU Tiling with MT (ms)		34	41	41	52	68	84	172
	Thread-distributed CPU Tiling with MT		29	34	32	33	39	51	98
	CPU Tiling with MT (ms)		37	45	46	56	73	92	189
	Thread-distributed CPU Tiling with MT		36	35	38	37	41	56	95
MHC MEDIAN	CPU Tiling with MT (ms)		64	70	70	80	102	137	270
	Thread-distributed CPU Tiling with MT		57	54	59	63	65	78	112
	CPU Tiling with MT (ms)		48	58	62	68	88	109	209
	Thread-distributed CPU Tiling with MT		45	48	50	51	60	64	96
BICUBIC	Overlap: 4px -> Overhead (Prozent)		0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	CPU Tiling with MT (ms)		97	111	122	127	133	189	386
HAMILTON ADAMS	Thread-distributed CPU Tiling with MT		98	102	99	100	106	103	158
	Overlap: 4px -> Overhead (Prozent)		0.8	1.6	3.2	6.6	13.8	30.6	77.8
	min. Belegung: 4x		67.1 MB	16.8 MB	4.2 MB	1.0 MB	262.1 KB	65.5 KB	16.4 KB
	max. Belegung: 5x		83.9 MB	21.0 MB	5.2 MB	1.3 MB	327.7 KB	81.9 KB	20.5 KB
PPG	CPU Tiling with MT (ms)		56	61	63	79	106	147	366
	Thread-distributed CPU Tiling with MT		49	53	56	58	62	76	150
	CPU Tiling with MT (ms)		74	78	86	95	130	197	414
	Thread-distributed CPU Tiling with MT		66	68	70	73	78	90	137
RCD	Overlap: 10px -> Overhead (Prozent)		2	4	8.3	17.7	40.5	111.6	611.1
	min. Belegung: 7x		117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
	max. Belegung: 8x		134.2 MB	33.6 MB	8.4 MB	2.1 MB	524.3 KB	131.1 KB	32.8 KB
	CPU Tiling with MT (ms)		177	187	188	216	303	572	3120
DLMSE CODE	Thread-distributed CPU Tiling with MT		150	142	162	170	186	242	709
	Overlap: 12px -> Overhead (Prozent)		2.4	4.9	10.1	21.8	51.5	156	1500
	min. Belegung: 6x		100.7 MB	25.2 MB	6.3 MB	1.6 MB	393.2 KB	98.3 KB	24.6 KB
	max. Belegung: 13x		218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
PAPER	CPU Tiling with MT (ms)		178	186	200	236	374	740	
	Thread-distributed CPU Tiling with MT		139	135	148	165	192	267	
	CPU Tiling with MT (ms)		188	185	196	246	372	748	
	Thread-distributed CPU Tiling with MT		142	130	149	160	195	266	
DLMSE RCD CODE	Overlap: 14px -> Overhead (Prozent)		2.8	5.7	11.9	26.1	63.8	216	6300
	min. Belegung: 7x		117.4 MB	29.4 MB	7.3 MB	1.8 MB	458.8 KB	114.7 KB	28.7 KB
	max. Belegung: 13x		218.1 MB	54.5 MB	13.6 MB	3.4 MB	852.0 KB	213.0 KB	53.2 KB
	CPU Tiling with MT (ms)		267	281	294	334	519	1193	
PAPER	Thread-distributed CPU Tiling with MT		212	193	219	258	291	459	
	CPU Tiling with MT (ms)		272	271	280	332	518	1196	
	Thread-distributed CPU Tiling with MT		206	188	218	257	293	452	

Tabelle 14: Ausführungszeiten nach Multithreadingverteilung und Kachelgröße, 3,3 MP Bild, Mac Mini M1

Tabelle 15: Implementationsvorschlag ideale Beschleunigungsstrategie für kleine Bilder

Demosaicing-Algorithmus	Gewählte Strategie	Trade-Off ThinkPad	Trade-Off Mac Mini
Kein Demosaicing	Multithreading	23%	7%
Nearest Neighbour	Multithreading	6%	3%
Bilinear Mean	Multithreading	3%	8%
Bilinear Median	Multithreading	8%	11%
Bicubic	Multithreading	1%	1%
Malvar-He-Cutler	Multithreading	4%	17%
Hamilton-Adams	Multithreading	10%	14%
Patterned Pixel Grouping	Multithreading	12%	16%
RCD	Thread-dist. Tiling with MT (1024px)	kein	kein
DLMMSSE	Thread-dist. Tiling with MT (1024px)	kein	kein
DLMMSSE+RCD	Thread-dist. Tiling with MT (1024px)	kein	kein

Unklar ist nur, wo die Grenze zwischen kleinen und großen Bildern liegt. Sie kann zwar durch weitere Benchmarks ermittelt werden, der Wert ist aber vermutlich stark vom verwendeten Gerät abhängig.

**7.3.8.2 Große Bilder** Für die Verarbeitung von großen Bildern ist bei den meisten Algorithmen auf beiden Geräten Thread-distributed CPU-Tiling mit Kacheln der Seitenlänge 128px ideal, trotz der teils unterschiedlichen Cache-Größen. Das Optimum für das jeweilige Gerät ist wahrscheinlich etwas größer oder kleiner, unterscheidet sich dann aber vermutlich. Deshalb ist der Wert, es zu ermitteln, gering. Da die beiden Geräte in der Herstellung 7 Jahre auseinanderliegen, ist es gut möglich, dass diese Konfiguration auch für viele weitere und zukünftige CPUs gut funktioniert und im Zweifelsfall in der Zukunft manuell angepasst werden kann. Alternativ könnte die Cachegröße mithilfe des OSHI-Frameworks ermittelt werden, um Heuristiken wie “die kleinste Kachelgröße, die nicht mehr in den L1-Cache passt” zu implementieren.

Beim bikubischen Demosaicing stimmt die beste Strategie für die Geräte überein (Thread-distributed CPU Tiling with MT), aber nicht die Kachelgröße: Die ideale Größe für den Mac Mini M1 braucht auf dem ThinkPad W530 39ms länger, umgekehrt sind es 31ms Unterschied. Da es ein im Verhältnis zur Ausführungszeit kleiner Unterschied ist, ist es an dieser Stelle vermutlich sinnvoller, sich am neueren Gerät zu orientieren. Der Wert - 2048x2048px - lässt

sich nämlich nicht durch ein ins-Verhältnis-setzen der Zwischenergebnisse und des verfügbaren Caches ermitteln. Ähnlich lässt sich beim RCD-Algorithmus argumentieren, da hier die für den Mac Mini M1 die gleiche Konfiguration ideal ist, und auf dem ThinkPad nur 84ms länger als die Bestzeit mit Thread-distributed CPU Tiling (3815 ms) braucht. Der Unterschied beim Mac Mini ist hier, trotz der insgesamt kürzeren Verarbeitungszeit, etwas größer.

Schwieriger ist die Entscheidung beim DLMMSE+RCD-Algorithmus, da hier der Unterschied zu den jeweils auf dem anderen Gerät optimalen Konfigurationen je nach Version zwischen 5 und 10% der Ausführungszeit beträgt. Letztendlich bedeutet das aber auch nur, dass der Trade-Off für den Fokus auf neuere Geräte etwas größer ist.

Tabelle 16: Implementationsvorschlag ideale Beschleunigungsstrategie für große Bilder (Trade-Off-Werte für mittleres Bild)

Demosaicing-Algorithmus	Gewählte Strategie	Trade-Off ThinkPad	Trade-Off Mac Mini
Kein Demosaicing	Thread-dist. Tiling (128px, größer L1-Cache)	kein	kein
Nearest Neighbour	Thread-dist. Tiling (128px, größer L1-Cache)	2%	kein
Bilinear Mean	Thread-dist. Tiling (128px, größer L1-Cache)	kein	kein
Bilinear Median	Thread-dist. Tiling (128px, größer L1-Cache)	kein	kein
Bicubic	Thread-dist. Tiling with MT (2048px)	2%	kein
Malvar-He-Cutler	Thread-dist. Tiling (128px, größer L1-Cache)	kein	1%
Hamilton-Adams	Thread-dist. Tiling (128px, größer L1-Cache)	2%	kein
Patterned Pixel Grouping	Thread-dist. Tiling (128px, größer L1-Cache)	kein	kein
RCD	Thread-dist. Tiling with MT (2048px)	2%	kein
DLMMSE Code/Paper	Thread-dist. Tiling (256px)	7% / 1%	kein
DLMMSE+RCD Code/Paper	Thread-dist. Tiling with MT (2048px)	12% / 5%	kein

### Algorithmus: Preprocessing

		ThinkPad W530   Cache-Größen		too big	L3-Cache: 1x 6MB		L2-Cache: 4x 1MB		L1-Cache: 4x 128KB	
Bild	Metrik/Kachelgröße	2048	1024	512	256	128	64	32		
3,3 MP	Belegung: 2x	33.6 MB	8.4 MB	2.1 MB	524.3 KB	131.1 KB	32.8 KB	8.2 KB		
	Multithreading-OLD	275								
	None	214								
	Multithreading	180								
	CPU Tiling (ms)	222	273	276	263	230	233	231		
	CPU Tiling with MT (ms)	183	172	144	133	140	174	245		
	Thread-distributed CPU Tiling (ms)	214	150	200	263	257	254	269		
47,4 MP	Thread-distributed CPU Tiling with MT	180	176	192	194	201	233	252		
	Multithreading-OLD	643								
	None	2334								
	Multithreading	626								
	CPU Tiling (ms)	2438	2455	2466	2486	2401	2416	2414		
	CPU Tiling with MT (ms)	638	643	660	706	771	987	1676		
	Thread-distributed CPU Tiling (ms)	667	603	725	639	631	617	601		
61,2 MP	Thread-distributed CPU Tiling with MT	627	631	637	666	707	766	837		
	Multithreading-OLD	740								
	None	3199								
	Multithreading	796								
	CPU Tiling (ms)	3213	3272	3274	3265	3213	3220	3230		
	CPU Tiling with MT (ms)	818	818	850	896	965	1232	2145		
	Thread-distributed CPU Tiling (ms)	780	769	862	775	777	758	753		
	Thread-distributed CPU Tiling with MT	803	799	819	848	864	913	996		
Mac Mini M1   Cache-Größen		too big		L2-Cache (groß): 4x12MB		L2-Cache (klein): 4x 4MB		L1-Cache: (4x 64KB) + 4x128 KB		
Bild	Metrik/Kachelgröße	2048	1024	512	256	128	64	32		
3,3 MP	Belegung: 2x	33.6 MB	8.4 MB	2.1 MB	524.3 KB	131.1 KB	32.8 KB	8.2 KB		
	None	112								
	Multithreading	100								
	CPU Tiling (ms)	115	123	121	120	116	116	115		
	CPU Tiling with MT (ms)	103	101	78	70	82	106	141		
	Thread-distributed CPU Tiling (ms)	116	66	92	169	164	178	186		
	Thread-distributed CPU Tiling with MT	107	102	106	110	111	132	173		
47,4 MP	None	1397								
	Multithreading	321								
	CPU Tiling (ms)	1408	1422	1404	1391	1415	1393	1384		
	CPU Tiling with MT (ms)	327	324	331	380	558	921	1143		
	Thread-distributed CPU Tiling (ms)	410	350	363	385	359	353	365		
	Thread-distributed CPU Tiling with MT	318	317	312	333	367	402	452		
	None	1729								
61,2 MP	Multithreading	388								
	CPU Tiling (ms)	1739	1776	1720	1716	1735	1716	1744		
	CPU Tiling with MT (ms)	393	392	409	459	684	1118	1416		
	Thread-distributed CPU Tiling (ms)	455	425	442	456	426	421	431		
	Thread-distributed CPU Tiling with MT	383	381	387	408	428	472	532		

Tabelle 17: Ausführungszeiten in allen Konfigurationen, Preprocessing

### Algorithmus: Postprocessing

ThinkPad W530   Cache-Größen		too big	L3-Cache: 1x 6MB		L2-Cache: 4x 1MB		L1-Cache: 4x 128KB		
Bild	Metrik/Kachelgröße		2048	1024	512	256	128	64	32
3,3 MP	Belegung: 6x		100.7 MB	25.2 MB	6.3 MB	1.6 MB	393.2 KB	98.3 KB	24.6 KB
	Multithreading-OLD		506						
	None		563						
	Multithreading		133						
	CPU Tiling (ms)		530	578	582	552	542	540	535
	CPU Tiling with MT (ms)		136	137	140	150	140	157	203
	Thread-distributed CPU Tiling (ms)		553	243	223	255	221	206	202
47,4 MP	Thread-distributed CPU Tiling with MT		139	139	138	148	149	160	173
	Multithreading-OLD		3694						
	None		6929						
	Multithreading		1368						
	CPU Tiling (ms)		7095	7094	7111	6919	6883	6864	6815
	CPU Tiling with MT (ms)		1410	1635	1574	1587	1624	1918	2610
	Thread-distributed CPU Tiling (ms)		1712	1499	1540	1532	1494	1477	1470
61,2 MP	Thread-distributed CPU Tiling with MT		1382	1370	1362	1380	1406	1491	1423
	Multithreading-OLD		4538						
	None		8872						
	Multithreading		1752						
	CPU Tiling (ms)		9000	9083	9211	8617	8289	8024	8033
	CPU Tiling with MT (ms)		1822	1955	1883	1855	1960	2237	3081
	Thread-distributed CPU Tiling (ms)		1950	1942	1912	1906	1847	1843	1900
Mac Mini M1   Cache-Größen	Thread-distributed CPU Tiling with MT		1732	1749	1765	1751	1790	1835	1776
Mac Mini M1   Cache-Größen		too big	L2-Cache (groß): 4x12MB		L2-Cache (klein): 4x 4MB		L1-Cache: (4x 64KB) + 4x128 KB		
Bild	Metrik/Kachelgröße		2048	1024	512	256	128	64	32
3,3 MP	Belegung: 6x		100.7 MB	25.2 MB	6.3 MB	1.6 MB	393.2 KB	98.3 KB	24.6 KB
	None		1460						
	Multithreading		250						
	CPU Tiling (ms)		1463	1461	1460	1461	1446	1445	1440
	CPU Tiling with MT (ms)		250	255	254	255	266	320	542
	Thread-distributed CPU Tiling (ms)		1466	487	309	284	291	273	272
	Thread-distributed CPU Tiling with MT		251	253	248	244	247	248	246
47,4 MP	None		21529						
	Multithreading		3532						
	CPU Tiling (ms)		21631	21637	21720	21719	21411	21451	21353
	CPU Tiling with MT (ms)		3354	3418	3421	3480	3677	4479	7641
	Thread-distributed CPU Tiling (ms)		4336	3731	3432	3374	3366	3356	3289
	Thread-distributed CPU Tiling with MT		3384	3326	3320	3316	3316	3332	3284
	None		27681						
61,2 MP	Multithreading		4577						
	CPU Tiling (ms)		27718	27764	27873	27783	27408	27404	27280
	CPU Tiling with MT (ms)		4335	4419	4430	4472	4715	5736	9775
	Thread-distributed CPU Tiling (ms)		4665	4629	4394	4331	4329	4306	4234
	Thread-distributed CPU Tiling with MT		4331	4276	4269	4263	4270	4280	4221

Tabelle 18: Ausführungszeiten in allen Konfigurationen, Postprocessing

**7.3.8.3 Pre- und Postprocessing** Beim Postprocessing großer Bilder ist die ideale Verteilung von Multithreading geräteunabhängig, und die Zeiten für die verschiedenen Abschnittsgrößen unterscheiden sich nur geringfügig. Da hier kein Zusammenhang mit Cachegrößen hergestellt werden kann, bietet sich an, per Default die für den Mac Mini beste Konfiguration mit den kleinstmöglichen Abschnitten zu wählen.

Beim kleinsten Bild hingegen ist die auf dem ThinkPad schnellste Strategie - Multithreading ohne Abschnitte - auch auf dem Mac Mini nur gut 2% langsamer als die schnellste Konfiguration, während umgekehrt der Unterschied deutlich größer ist.

Beim Preprocessing großer Bilder sind die Unterschiede zwischen den Geräten verhältnismäßig am größten. Beide Geräte haben aber gemein, dass die beste Konfiguration nur marginal besser ist als keine Aufteilung in Abschnitte, weshalb sich diese Strategie als Kompromiss anbietet. Beim kleinen Bild ist der Unterschied zum Multithreading größer, und auch wenn sich die idealen Strategien unterscheiden, kostet die Wahl der für das neuere Gerät besseren Strategie das ältere Gerät nur wenige Millisekunden.

Tabelle 19: Implementationsvorschlag ideale Beschleunigungsstrategie für Pre- und Postprocessingschritte (Trade-Off-Werte großes Bild für 47MP Bild)

Verarbeitungsschritt	Gewählte Strategie	Trade-Off ThinkPad	Trade-Off Mac Mini
Preprocessing (kleines B.)	Thread-dist. Tiling (1024px)	12%	kein
Preprocessing (großes B.)	Multithreading	4%	2%
Postprocessing (kleines B.)	Multithreading	kein	2%
Postprocessing (großes B.)	Thread-dist. Tiling with MT (32px)	4%	kein

## 7.4 Diskussion

### 7.4.1 Nutzen der Verwendung der Grafikhardware

Die Erfahrungen des Feldtests zeigen: Heterogene Grafikhardware zu unterstützen ist mit viel Aufwand verbunden, und keine Technologie kann alle Geräte abdecken. Die Parallelisierung von Berechnungen auf der GPU kann deren Ausführungszeit zwar um einen Faktor von 5-10 gegenüber Multithreading auf der CPU beschleunigen. Ein abschließender Test auf dem ThinkPad W530 (Abb. 38) zeigt aber, dass aufgrund diverser Overheads davon bei den Endnutzenden

wenig ankommt - bei einem einfachen Algorithmus wie Bilinear Mean ist die GPU-Version sogar langsamer.

Ein großer bremsender Faktor ist das Zurückspielen der Bilddaten in die Kontrolle der CPU. Dieser Schritt würde wegfallen, wenn man direkt OpenGL zur Darstellung des Bilds auf dem Bildschirm nutzt. Das würde aber einen kompletten Umbau der Anwendung mit einer viel engeren Verzahnung zwischen DNGProcessor und Benutzeroberfläche bedeuten.

#### 7.4.2 Optimierungen auf der CPU

Was sich in den Benchmarks und auch im abschließenden Test zeigt, ist, dass gerade Optimierungen, die sich auf Details wie die Größen der CPU-Caches beziehen, letztendlich sehr gerätespezifisch sind. So ist zum Beispiel die weiter oben beschriebene Heuristik für kleine Bilder auf dem ThinkPad teilweise langsamer als die Verwendung von regulärem Multithreading.

Gerade in einer inhärent von der Hardware abstrahierten Sprache wie Java wird die Performanz außerdem von vielen Faktoren beeinflusst, die nicht unter der Kontrolle des Programms stehen: Neben dem JIT-Compiler der JVM spielt z.B. auch die Implementation des Multithreadings in der Java Streams-API eine große Rolle. Wie der geringe Unterschied zwischen der mit CPU-Tiling optimierten Zeit zeigt, sind diese eingebauten Optimierungsfeatures ziemlich stark und es kann in vielen Fällen ausreichend sein, sich darauf zu verlassen.

Allerdings ist es trotzdem nicht zwecklos, sich Gedanken über die Effizienz von Algorithmen zu machen. Die Verbesserungen bei den RCD, DLMMSE und DLMMSE+RCD-Algorithmen zeigen, was zum Beispiel durch eine Datenflussanalyse und gutes Softwaredesign erreicht werden kann.

Des weiteren liegt der Erfolg der Tiling-Strategien auf der CPU nicht nur in einer schnelleren Ausführung des Programms, sondern auch in der Reduktion des Speicherfußabdrucks der komplexeren Algorithmen. Auf manchen älteren Geräten wird so erst die Anwendung dieser Algorithmen auf große Bilder ermöglicht, und insgesamt wird so die theoretische Verarbeitung von beliebig großen RAW-Dateien unterstützt.

Es ergibt also durchaus Sinn, den Nutzenden von Jeniffer2 die Möglichkeit zu geben, aus verschiedenen Ausführungsstrategien auf der CPU zu wählen. Allerdings sollten die Bezeichnungen angepasst werden, um auch Laien eine informierte Auswahl zu ermöglichen, mit Begriffen wie "geringer Speicherfußabdruck" oder "ältere Hardware".

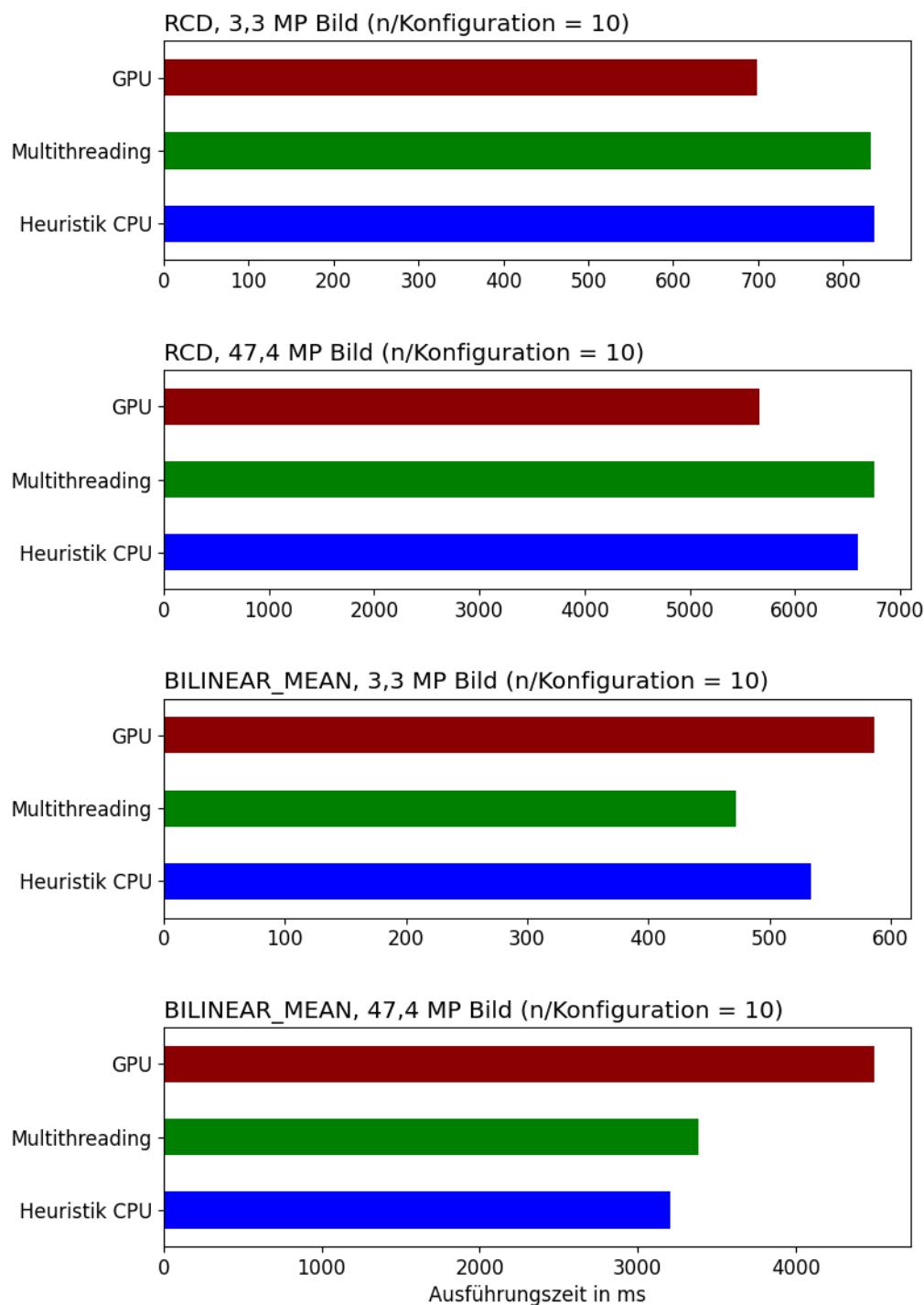


Abbildung 38: Vergleich gesamtes DNG Processing auf dem ThinkPad W530 nach Beschleunigungsstrategie (mit integrierter Grafikkarte)

## 8 Fazit und Ausblick

### 8.1 Fazit

Es gibt aktuell eine Vielzahl an Möglichkeiten, Java-Programme durch eine Parallelisierung und effiziente Nutzung der Hardware zu beschleunigen.

Die Verwendung von Grafikhardware hat das Potential, Anwendungen um Größenordnungen schneller zu machen. Die partielle Implementation in OpenGL zeigt, dass der Preis dieser Beschleunigung eine deutlich höhere Komplexität des Programms ist. Um den Code zu verstehen, ist Fachwissen über die unterliegende Hardware nötig, und die universelle Unterstützung einer heterogenen Gerätekulisse ist, wie der Feldtest beweist, so gut wie unmöglich. Die Details dieser Probleme von der Programmlogik zu abstrahieren ist ein aktives Forschungsfeld, in dem sich in den nächsten Jahren sicherlich noch einiges tun wird - ein Beispiel ist hierfür TornadoVM.

Bis solche Projekte eine gewisse Reife erreichen, ist der Nutzen der Einbindung der GPU in Jeniffer2 aber noch zu gering, um eine programmweite Umstellung zu rechtfertigen. Die Beschleunigung der RCD, DLMMSE und DLMMSE+RCD-Algorithmen um einen Faktor von ca. 10 nur durch eine Optimierung der Programmstruktur zeigt, dass Überlegungen zu Aspekten wie dem Datenfluss innerhalb einer Anwendung mindestens genauso wichtig sind wie die Nutzung von Parallelisierungsfeatures.

Außerdem sind auch die in Java eingebauten Möglichkeiten zum Multithreading auf der CPU bereits sehr effizient. Moderne CPUs sind auch schon sehr weit, was die automatische Reduktion von Latenzen im Speicherzugriff angeht. Aus expliziten Überlegungen zur Speicherhierarchie entstandene Ansätze wie die Verarbeitung des Bilds in Kacheln sorgen also nur noch für eine geringe und nur bedingt portable Verbesserung, machen das Programm aber leichtgewichtiger, was den benötigten Speicherplatz angeht.

Wie in Abb. 39 zu sehen, konnte insgesamt für alle Demosaicing-Algorithmen ein Speedup erreicht werden und die komplexeren Algorithmen stellen nun keine so großen Ausreißer mehr dar.

Der Benutzeroberfläche wurden mit der Pixellupe mit ihren unterschiedlichen Modi und dem schnellen Konfigurationswechsel über das Dropdown-Menü zwei neue Features hinzugefügt, von denen vor allem zweiteres von der schnelleren Verarbeitungszeit profitiert.

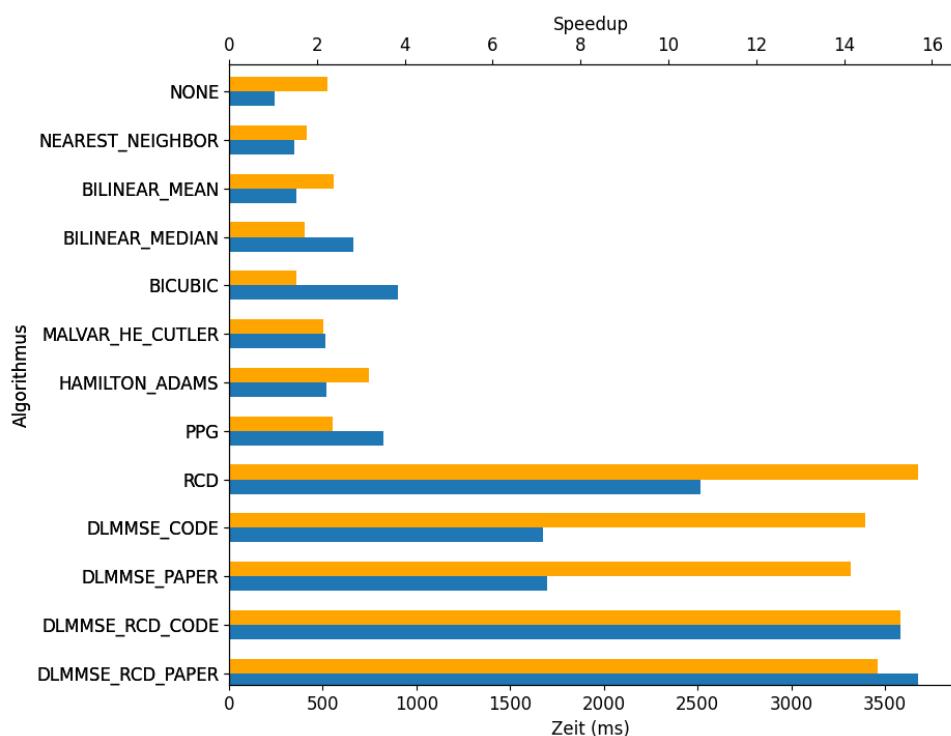


Abbildung 39: Beste Ausführungszeit Demosaicingalgorithmen (Durchschnitt Mac Mini M1/ThinkPad W530) vs. Speedup-Faktor gegenüber alter Version (nur ThinkPad), 47,4 MP Bild

## 8.2 Ausblick

### 8.2.1 Open-Sourcing

Ein großer Vorteil von Jeniffer2 ist die Verwendung einer höher-abstrahierten Sprache wie Java. Verbunden mit der klaren Programmstruktur können so auch Menschen mit wenig Programmiererfahrung zum Beispiel eigene Demosaicing-Algorithmen hinzufügen und die Resultate vergleichen.

Um diese Vision einer Lehr- und Experimentiersoftware zu realisieren, wird der Quellcode von Jeniffer2 nach dem Abschluss dieser Arbeit öffentlich zur Verfügung gestellt. Hierbei wird mit der “GNU General Public License v3”<sup>10</sup> eine Lizenz gewählt, die sicherstellt, dass Änderungen und Erweiterungen der Software ebenfalls der Allgemeinheit zu Gute kommen.

### 8.2.2 Optimierungsansätze für die Zukunft

Am Anfang dieser Arbeit wurde die Verwendung von SIMD-Instruktionen mithilfe der Java Vector API deshalb ausgeschlossen, weil Experimente bei der Verarbeitung großer Datenmengen nur einen geringen Vorteil attestierten. Durch die Aufteilung des Bilds in Kacheln nimmt die von einer potentiell vektorisierten Operation verarbeitete Datenmenge ab, und so könnte sich hier der Einsatz von Vektorinstruktionen doch wieder lohnen.

Allerdings steht die Verwendung im Gegensatz zur Zielsetzung, neue Verarbeitungsschritte in Jeniffer2 leicht implementierbar und den Code lesbar zu halten. Die von TornadoVM bereitgestellte API lässt sich hingegen fast nahtlos in existierenden Code einbetten. Mit Blick auf die Zukunft ist es also vielleicht sinnvoller, die Entwicklung dieses Projekts weiter zu verfolgen: Wenn sich hier der Kompilations-Overhead reduziert und Kinderkrankheiten ausgebügelt werden, kann es sich lohnen, noch einmal das Problem im aktuell am längsten dauernden Prozessschritt zu suchen und das GUI-Framework der Benutzeroberfläche durch eine kompatible Lösung auszutauschen.

Als letzter Punkt lag der Fokus dieser Arbeit auf einer Beschleunigung der Berechnung bei gleichbleibendem Ergebnis. Gerade beim Schritt der Gamma-korrektur und Farbraumtransformation könnte aber auch untersucht werden, wie sich weniger rechenintensive Annäherungen der verwendeten Formeln auf die Bildqualität auswirken. Mit dem hinzugefügten Code zum Qualitätsbenchmarking ist der Grundstein dafür bereits gelegt.

### 8.2.3 Neuronale Netze und KI

Die Verwendung von sogenannter künstlicher Intelligenz (KI) in Form von neuronalen Netzen ist im Bereich der Bildverarbeitung weit verbreitet. Gharbi

---

<sup>10</sup><https://www.gnu.org/licenses/gpl-3.0.en.html>

et al. (2019) experimentieren zum Beispiel damit, Denoising und Demosaicing in einem Schritt über ein Convolutional Neural Network (CNN) zu implementieren. Gerade für die an Matrixmultiplikationen intensive Implementation neuronaler Netze in Java sind die in dieser Arbeit erkundeten Beschleunigungsstrategien sicher interessant.

Dank der Modularität von Jeniffer2 wäre es aber auch denkbar, eine KI-Implementation von z.B. Demosaicing in externen Code in einer für diese Anwendung besser geeigneten Sprache wie Python auszulagern, aber die Infrastruktur zum Lesen des DNG-Formats und die GUI in Java zu belassen. Ähnlich zu TornadoVM gibt es aktuell mit Mojo<sup>11</sup> ein Projekt, welches sich mit der Ausführung von Python auf Grafikhardware beschäftigt.

#### 8.2.4 Benutzeroberfläche: Feinschliff oder Lehrbuch

Die Möglichkeiten zum Vergleichen der Ergebnisse verschiedener Algorithmen wurden im Rahmen dieser Arbeit deutlich verbessert. Trotzdem gibt es in der Benutzeroberfläche noch einiges an Potential: Die Auswahl der Beschleunigungsstrategie ist zum Beispiel vermutlich sinnvoller als globale Option, die sich über ein Menü oder in einer Fußeiste einstellen lässt. Außerdem wäre es schön, mehrere Bilder gleichzeitig offen zu haben, und Funktionalitäten wie ein Histogramm könnten reaktiviert werden.

Eine etwas ambitioniertere Idee wäre, Jeniffer2 komplett als Lehrsoftware auszurichten und zu jedem Demosaicing-Algorithmus eine vielleicht sogar interaktive Beschreibung mit Grafiken und Beispielen zu integrieren. Auch die anderen DNG-Processing-Schritte könnten erklärt und ihre Zwischenergebnisse sichtbar gemacht werden.

---

<sup>11</sup><https://www.modular.com/mojo>

# Literatur

- Adobe Inc. 2021. “Digital Negative (Dng) Specification, Version 1.6.0.0.” [https://helpx.adobe.com/content/dam/help/en/photoshop/pdf/dng\\_spec\\_1\\_6\\_0\\_0.pdf](https://helpx.adobe.com/content/dam/help/en/photoshop/pdf/dng_spec_1_6_0_0.pdf). (Zugriff: 12.07.2023)
- Amdahl, Gene M. 1967. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.” In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 483–85.
- Azul Systems. 2023. “Azul Zulu Openjdk Distribution.” [https://www\\_azul\\_com\\_downloads/?package=jdk#zulu](https://www_azul_com_downloads/?package=jdk#zulu). (Zugriff: 12.07.2023)
- Bayer, Bryce E. 1976. “Color Imaging Array.” *US Patent 3,971,065*.
- Bien, Adam. 2016. “Afterburner.fx.” *Github Repository*. GitHub. <https://github.com/AdamBien/afterburner.fx>. (Commit-Id: a24f8cb)
- Brady, David J. 2016. “AWARE Wide Field View.” Duke University Durham United States.
- Cloutier, Félix. 2023. “X86 and Amd64 Instruction Reference.” [https://www\\_felixcloutier\\_com/x86/](https://www_felixcloutier_com/x86/). (Zugriff: 12.07.2023)
- Danowitz, Andrew, Kyle Kelley, James Mao, John P. Stevenson, und Mark Horowitz. 2012. “CPU Db: Recording Microprocessor History.” *Commun. ACM* 55 (4): 55–63. <https://doi.org/10.1145/2133806.2133822>.
- Darktable Developers. 2023. “Darktable (Gamma Correction).” *Github Repository*. GitHub. <https://github.com/darktable-org/darktable/blob/master/src/iop/gamma.c#L62>. (Commit-Id: 86d1d20)
- Engheim, Erik. 2021. “ARMv9: What Is the Big Deal?” <https://levelup.gitconnected.com/armv9-what-is-the-big-deal-4528f20f78f3>. (Zugriff: 12.07.2023)
- Faruqi, Muhammad Ismail, Fumihiko Ino, und Kenichi Hagihara. 2012. “Acceleration of Variance of Color Differences-Based Demosaicing Using Cuda.” In *2012 International Conference on High Performance Computing & Simulation (Hpcs)*, 503–10. <https://doi.org/10.1109/HPCSim.2012.6266965>.
- Flynn, Michael J. 1972. “Some Computer Organizations and Their Effectiveness.” *IEEE Transactions on Computers* 100 (9): 948–60.
- Fumero, Juan, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, und Christos Kotselidis. 2019. “Dynamic Application Reconfiguration on Heterogeneous Hardware.” In *Proceedings of the 15th Acm Sigplan/Sigops International Conference on Virtual Execution Environments*, 165–78. VEE 2019. New York, NY, USA: Association for Computing

- Machinery. <https://doi.org/10.1145/3313808.3313819>.
- Fung, James. 2007. “Parallel General Purpose Computing Across Multiple Computer Graphics Devices”. University of Toronto.
- Gharbi, Michaël, Gaurav Chaurasia, Sylvain Paris, und Frédo Durand. 2016. “Deep joint demosaicking and denoising”. *ACM TOG 35, 6 (2016)*, 191
- Getreuer, Pascal. 2011. “Zhang–Wu Directional LMMSE Image Demosaicking.” *Image Processing on Line*. [https://doi.org/10.5201/ipol.2011.g\\_zwld](https://doi.org/10.5201/ipol.2011.g_zwld).
- Giagio, Diego. 2019. “Warp.” *GitHub Repository*. GitHub. <https://github.com/dgiagio/warp>. (Commit-Id: b82c90b)
- Goorts, Patrik, Sammy Rogmans, und Philippe Bekaert. 2012. “Raw Camera Image Demosaicing Using Finite Impulse Response Filtering on Commodity Gpu Hardware Using Cuda.” In *SIGMAP*, 96–101.
- Gustafson, John L. 1988. “Reevaluating Amdahl’s Law.” *Communications of the ACM* 31 (5): 532–33.
- Harris, Mark. 2012. “How to Optimize Data Transfers in Cuda c/C++.” *NVIDIA Developer Technical Blog*. NVIDIA Corporation & Affiliates. <https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/>. (Zugriff: 12.07.2023)
- Intel. 2023. “Intel® Instruction-Set-Extensions-Technik.” <https://www.intel.de/content/www/de/de/support/articles/000005779/processors.html>. (Zugriff: 12.07.2023)
- Intel Corporation. 2023. “Intel® Intrinsics Guide.” <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=vmulpd>. (Zugriff: 12.07.2023)
- jcuda.org. 2022. “Java Bindings for Cuda.” <http://javagl.de/jcuda.org/>. (Zugriff: 12.07.2023)
- John F. Hamilton Jr. und James E. Adams Jr. 1997. “Adaptive Color Plane Interpolation in Single Sensor Color Electronic Camera.” *US Patent 5,652,621*.
- Khashabi, Daniel, Sebastian Nowozin, Jeremy Jancsary und Andrew W Fitzgibbon. 2023. “Microsoft Research Cambridge Demosaicing Dataset”. <https://www.microsoft.com/en-us/download/details.aspx?id=52535>. (Zugriff: 01.07.2023)
- Langseth, Ragnar, Vamsidhar Reddy Gaddam, Håkon Kvale Stensland, Carsten Griwodz, und Pål Halvorsen. 2014. “An Evaluation of Debayering Algorithms on Gpu for Real-Time Panoramic Video Recording.” In *2014 IEEE*

*International Symposium on Multimedia*, 110–15. IEEE.

- Lee, Victor W., Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, et al. 2010. “Debunking the 100X Gpu Vs. CPU Myth: An Evaluation of Throughput Computing on Cpu and Gpu.” In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 451–60. ISCA ’10. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/1815961.1816021>.
- Lightweight Java Game Library. 2023. “LWJGL Website.” <https://www.lwjgl.org>. (Zugriff: 12.07.2023)
- Lin, Chuan-Kai. 2010. “Demosaic.” <https://web.archive.org/web/20160923211135/https://sites.google.com/site/chklin/demosaic/>. (Zugriff: 12.07.2023)
- Ljavin, Eugen. 2020. “JENIFFER2: Ein Raw-Processor Mit Wählbaren Demosaicing-Algorithmen Für Das Universelle Raw-Format Dng.” Masterarbeit, Universität Tübingen.
- Malvar, Henrique S, Li-wei He, und Ross Cutler. 2004. “High-Quality Linear Interpolation for Demosaicing of Bayer-Patterned Color Images.” In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, 3:iii–485. IEEE.
- McGuire, Morgan. 2008. “Efficient, High-Quality Bayer Demosaic Filtering on Gpus.” *Journal of Graphics Tools* 13 (4): 1–16. <https://doi.org/10.1080/2151237X.2008.10129267>.
- Moore, Gordon E, et al. 1965. “Cramming More Components onto Integrated Circuits.” McGraw-Hill New York.
- Newland, Chris. 2023. “OpenJDK Hsdis (Hotspot Disassembly Plugin) Downloads.” <https://chriswhocodes.com/hsdis/>. (Zugriff: 12.07.2023)
- NVIDIA Corporation & Affiliates. 2023. “CUDA C++ Programming Guide.” <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. (Zugriff: 12.07.2023)
- Oracle. 2023a. “IntStream Interface (Java Se 17 & Jdk 17).” <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/IntStream.html>. (Zugriff: 12.07.2023)
- . 2023b. “Runnable Interface (Java Se 17 & Jdk 17).” <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html>. (Zugriff: 12.07.2023)
- Oracle Coporation. 2023a. “GraalVM.” <https://www.graalvm.org/>. (Zugriff: 12.07.2023)

- . 2023b. “OpenJFX Project.” <https://openjdk.org/projects/openjfx/>. (Zugriff: 12.07.2023)
- OSHI Contributors. 2023. “OSHI Operation System & Hardware Information.” *GitHub Repository*. GitHub. <https://github.com/oshi/oshi>. (Commit-Id: a2a6dab)
- Pavlov, Igor. 2023. “7cpu Benchmark.” <https://www.7-cpu.com/>. (Zugriff: 25.05.2023)
- RawPedia. 2023. “Demosaicing.” <https://rawpedia.rawtherapee.com/Demosaicing>. (Zugriff: 25.05.2023)
- Reiter, Andreas. 2023. “Analyse Und Plattformunabhängige Implementierung Zum Effizienten Auflösen Des Bayer-Mosaiks.” Masterarbeit, Universität Tübingen.
- Rodriguez, Luis Sanz. 2017. “Ratio Corrected Demosaicing.” *GitHub Repository*. GitHub. <https://github.com/LuisSR/RCD-Demosaicing>. (Commit-Id: 0aa9d6e)
- Sandoz, Paul. 2023a. “JEP 417: Vector Api (Third Incubator).” *JDK Extension Proposals*. Oracle Corporation. <https://openjdk.org/jeps/417>. (Zugriff: 18.05.2023)
- . 2023b. “JEP 417: Vector Api (Third Incubator).” *JDK Bug System*. Oracle Corporation. <https://bugs.openjdk.org/browse/JDK-8269306>. (Zugriff: 18.05.2023)
- Stypinski, Martin. 2022a. “Auto-Vectorization: How to Get Beaten by Compiler Optimization — Java Jit!” <https://itnext.io/auto-vectorization-how-to-get-beaten-by-compiler-optimization-java-jit-vector-api-92c72b97fba3>. (Zugriff: 10.06.2023)
- . 2022b. “Java 18: Vector Api — Do We Get Free Speed-up?” <https://medium.com/@Styp/java-18-vector-api-do-we-get-free-speed-up-c4510eda50d2>. (Zugriff: 10.06.2023)
- Sutter, Herb. 2009. “The Free Lunch Is over: A Fundamental Turn Toward Concurrency in Software.” <http://www.gotw.ca/publications/concurrency-ddj.htm>.
- The pandas development team. 2023. *Pandas-Dev/Pandas: Pandas* (version v2.0.3). Zenodo. <https://doi.org/10.5281/zenodo.8092754>.
- The Apache Software Foundation. 2023a. “Apache Maven Project.” <https://maven.apache.org/>. (Zugriff: 11.07.2023)

- . 2023b. “Commons Math: The Apache Commons Mathematics Library.” <https://commons.apache.org/proper/commons-math/>. (Zugriff: 11.07.2023)
- TheBeard. 2020. “OpenCL – Platform and Execution Model.” <http://thebeardsage.com/opencl-platform-and-execution-model/>. (Zugriff: 19.05.2023)
- . 2023. “CUDA – Streaming Multiprocessors.” <http://thebeardsage.com/cuda-streaming-multiprocessors/>. (Zugriff: 19.05.2023)
- The Khronos® Group Inc. 2023. “OpenCL Website.” <https://www.khronos.org/opencl/>. (Zugriff: 19.05.2023)
- The RawTherapee Team. 2023. “RawTherapee (Color Helpers).” *GitHub Repository*. GitHub. <https://github.com/Beep6581/RawTherapee/blob/dev/rtengine/color.cc#L1606>. (Commit-Id: 7afdfc1)
- TornadoVM Contributors. 2023. “TornadoVM.” *GitHub Repository*. GitHub. <https://github.com/beehive-lab/TornadoVM>. (Commit-Id: 292128a)
- Vries, Joey de. 2022. “Learn Opengl (Website).” <https://learnopengl.com>. (Zugriff: 19.05.2023)
- Wang, Tongli, Wei Guo, und Jizeng Wei. 2019. “An Optimization Scheme for Demosaicing Algorithm on Gpu Using Opencl.” In *Computer Engineering and Technology*, Hrsg.: Weixia Xu, Liquan Xiao, Jinwen Li, und Zhenzhen Zhu, 142–52. Singapore: Springer Singapore.
- Wenninger, Marc. 2012. “Parallel Image Processing on Graphics Processing Units.” Bachelorarbeit, Hochschule Rosenheim.
- OpenGL Wiki contributors. 2022. “Rendering Pipeline Overview” *OpenGL Wiki* [http://www.khronos.org/opengl/wiki\\_opengl/index.php?title=Rendering\\_Pipeline\\_Overview&oldid=14914](http://www.khronos.org/opengl/wiki_opengl/index.php?title=Rendering_Pipeline_Overview&oldid=14914). (Zugriff: 19.05.2023)
- Wikipedia contributors. 2023. “OpenMP” *Wikipedia, the Free Encyclopedia*. <https://en.wikipedia.org/w/index.php?title=OpenMP&oldid=1154665236>. (Zugriff: 18.05.2023)
- Wronski, Bartłomiej, Ignacio Garcia-Dorado, Manfred Ernst, Damien Kelly, Michael Krainin, Chia-Kai Liang, Marc Levoy, und Peyman Milanfar. 2019. “Handheld Multi-Frame Super-Resolution.” *ACM Trans. Graph.* 38 (4). <https://doi.org/10.1145/3306346.3323024>.
- Zapryanov, Georgi, und Iva Nikolova. 2019. “An Experimental Comparative Performance Study of Demosaicing Algorithms on General-Purpose Gpus.” In *Proceedings of the 9th Balkan Conference on Informatics*. BCI’19. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/3351556.3351561>.

Zhang, Lei, und Xiaolin Wu. 2005. "Color Demosaicking via Directional Linear Minimum Mean Square-Error Estimation." *IEEE Transactions on Image Processing* 14 (12): 2167–78.

## **Selbständigkeitserklärung**

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Unterschrift