HAW HAMBURG

BACHELORTHESIS
Tim Luca Muschke

Leistungsanalyse und Optimierung der Pixelformatkonvertierung im industriellen Kontext

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Tim Luca Muschke

Leistungsanalyse und Optimierung der Pixelformatkonvertierung im industriellen Kontext

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Franz Korf
Zweitgutachter: Prof. Dr. Andreas Meisel

Eingereicht am: 21. Februar 2022

Tim Luca Muschke

Thema der Arbeit

Leistungsanalyse und Optimierung der Pixelformatkonvertierung im industriellen Kontext

Stichworte

Leistung, Pixelformatkonvertierung, Optimierung, Industrie, Vektorisieren, SIMD, Parallelisieren, OpenMP, Intel IPP

Kurzzusammenfassung

Diese Ausarbeitung umfasst eine Code Analyse der Pixelformatkonvertierung der Basler AG und eine prototypische Implementation diverser Optimierungsansätze. Die Optimierungsansätze sind Cacheline Alignment, Vektorisieren und Parallisieren des Codes.

Tim Luca Muschke

Title of Thesis

Performance analyses and optimization of the pixelformat conversion within the industrial context

Keywords

Performance, Pixelformat, Conversion, Optimization, Industry, Vectorization, SIMD, Parallelize, OpenMP, Intel IPP

Abstract

This thesis deals with the code analysis of the 'Pixelformat Conversion' from 'Basler AG' and different implementations of optimization strategies. These strategies are cache lien alignment, vectorization and parallelization.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

CPI cycles per instruction.

CPU Central Processing Unit.

DDR double data rate.

GB Giga Byte.

GPU Graphics Processing Unit.

MB Mega Byte.

MP Mega Pixel.

RAM random access memory.

SIMD Single Instruction Multiple Data.

1 Einleitung

Die Einleitung verschafft einen groben Überblick über das Thema und die Motivation der Bachelorarbeit.

Motivation

Die Basler AG entwickelt und produziert Kameras für die unterschiedlichsten Märkte, unter anderem für Medizin, Verkehr und Industrie. Der Kameramarkt ist stark umkämpft. Die Kamerahersteller entwickeln daher stets neue Funktionen und neue Kameras, die bessere und größere Bilder liefern, und auch schneller sind. Von der Erfassung der Bilddaten in der Kamera bis zur Darstellung der Bilder auf dem PC-Bildschirm müssen die Bilddaten verschiedene Phasen durchlaufen. Dazu zählt auch die Pixelformatkonvertierung. Diese ist ein Teil der Software, die aus den aufgenommenen Pixeldaten ein Bild erzeugt, das der PC-Bildschirm anzeigen oder zusätzliche Software weiterverarbeiten kann. Das Kundenfeedback besagt, dass die Konvertierung zu langsam ist. Daraus ergibt sich, dass der entsprechende Software-Teil analysiert und überarbeitet werden muss, mit dem Fokus auf mehr Leistung. Wenn die Pixelformatkonvertierung nicht ausreichend Leistung erzielen kann, kann der Kunde dadurch in seiner Produktion gedrosselt werden. Die Bilder können nicht schnell genug an weitere Software zur automatisierten Analyse gegeben werden. Diese Software würde zum Beispiel Defekte im Produkt per Bildverarbeitung erkennen. Der Einfluss von der Performance der Pixelformatkonvertierung liegt daran, weil die Pixelformatkonvertierung ein Teil der Bildverarbeitungskette ist. Es gibt unterschiedliche Verfahren, um den bestehenden Code zu optimieren.

Zielsetzung

Das Ziel dieser wissenschaftlichen Abschlussarbeit ist es, herauszufinden, wie effizient ein Teil der entwickelten Software der Basler AG ist und gegebenenfalls Optimierungs-

möglichkeiten darzulegen. Zu diesem Zweck erfolgt eine Analyse des Codes, genauer der Pixelformatkonvertierung, und eine prototypische Umsetzung diverser Optimierungen. Diese Optimierungen werden auf ihre Effektivität getestet und anschließend bewertet.

Struktur der Arbeit

Die Struktur der Arbeit gibt einen kurzen Überblick über das was in jedem Kapitel behandelt wird. In Kapitel 2 werden die theoretischen Grundlagen beschrieben, die zum Verständnis dieser Ausarbeitung benötigt werden. Es werden unter anderem die Pixelformate definiert. In Kapitel 3 wird der Ist-Zustand beschrieben, der zu analysieren und optimieren ist. Dies wird die Basis für die anschließenden Optimierungen darstellen. Des weiteren wird ein finales Problemstatement verfasst. In Kapitel 4 werden einige Methoden zur Optimierung von Code erläutert. Aus diesen Methoden ergeben sich die Möglichkeiten zur Optimierung im Rahmen dieser Arbeit. Es werden final die ausgewählten Optimierungsmethoden dargestellt. In Kapitel 5 wird das Testverfahren erläutert und was bei den Testdurchführungen zu beachten ist. In Kapitel 6 werden die Referenzwerte und die Messergebnisse von den ausgewählten Optimierungsmethoden dargestellt. Außerdem werden die Umsetzung beschrieben und die Ergebnisse diskutiert. In Kapitel 7 wird ein Fazit und ein Schlusswort verfasst. Es wird auch ein Ausblick gegeben über welche weiteren Schritte möglich wären.

2 Grundlagen

Dieses Kapitel beschreibt die Grundlagen, welche für das Verständnis der Bachelorarbeit benötigt werden. Unter anderem werden die Pixelformate definiert.

2.1 Was ist ein Bild?

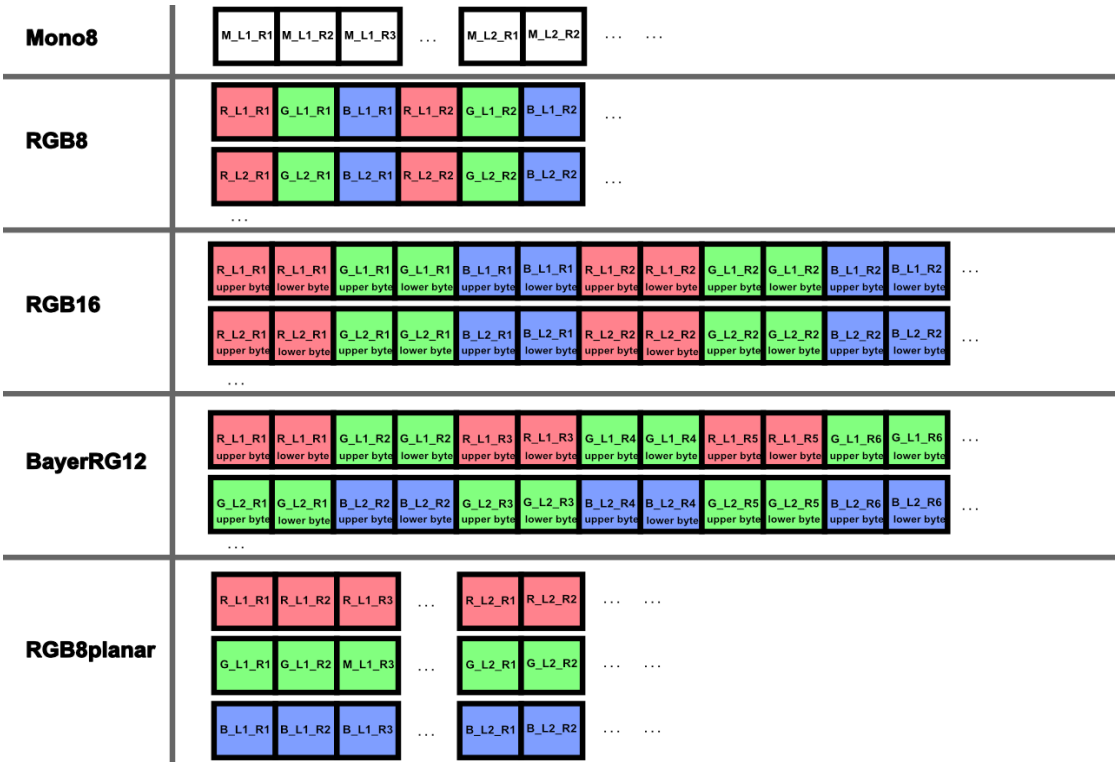
Ein Bild ist ein zwei dimensionales Abbild von realen Objekten, künstlerischen Darstellungen und anderen ähnlichen Dingen, welches von einem Monitor angezeigt werden kann. Beim Betrachten einer Bilddatei auf dem Computer fallen die typischen Dateiendungen '.jpeg', '.png' und einige weitere auf. Das sind lediglich die Dateiformate und sind nicht weiter wichtig für die Pixelformatkonvertierung in dieser Ausarbeitung.
Ein Bild besteht aus vielen sogenannten Pixeln, welche in Zeilen und Spalten, ähnlich einer Tabelle, aufgeteilt sind. Ein Pixel hält die Informationen zum Beispiel für die Rot, Grün und Blau Werte an einer expliziten Stelle eines Bildes. Aus den drei Werten kann dann eine Farbe erzeugt werden. Bei einem monochrom Bild benötigt ein Pixel lediglich die Helligkeit an einer Stelle im Bild.
Ein typischer Kamerasensor hat viele Sensorpixel, welche einen Farbfilter, zum Beispiel im BayerRG Muster, darüber liegen haben. So entsteht ein Bild direkt von der Kamera im BayerRG Pixelformat. Dies sieht eigenartig aus, würde es direkt vom Monitor angezeigt werden. Daher wird es typischerweise vor dem Anzeigen in ein RGB Pixelformat konvertiert.

2.1.1 Pixelformate

Die Pixelformate sind durch die PFNC standardisiert [5]. In Absprache mit dem Fach-Betreuer haben sich die folgenden Pixelformate als relevant herausgestellt. Für die Grafik, welche die Speicherbelegung der einzelnen Pixelformate darstellen, gilt:

- ein 'Kasten' repräsentiert ein Byte
- 'M' steht für einen Grau-Wert
- 'R' steht für einen Rot-Wert
- 'G' steht für einen Grün-Wert
- 'B' steht für einen Blau-Wert
- 'Lx'(x = Nummer) steht für die Zeile (Line) des Bildes.
- 'Rx'(x = Nummer) steht für die Spalte (Row) des Bildes

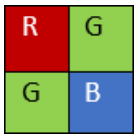
Abbildung 2.1: Pixelformate - Speicherbelegung



Für eine größere Darstellung siehe Anhang A.2.
Alle Pixelformate sind vorzeichenlos und unpacked. Das bedeutet, dass die Pixel keinen negativen Wert annehmen können und freie Bits in einem Byte nicht mit dem nächsten Pixel aufgefüllt sind.
Mono8: Dieses Pixelformat definiert ein 8-Bit, monochromes Format. Es sind 8-Bit pro

Pixel.
RGB8: Dieses Pixelformat definiert ein 8-Bit RGB (rot, grün, blau) Format. Es ergeben sich 24-Bit pro Pixel.
RGB16: Dieses Pixelformat definiert ein 16-Bit RGB (rot, grün. Blau) Format. Es ergeben sich 48-Bit pro Pixel.
BayerRG12: Dieses Pixelformat definiert ein 12-Bit, RGGB(rot, grün/grün, blau) Bayer Muster Format (siehe 2.2). Es ergeben sich 16-Bit pro Pixel. Die Helligkeitsinformation befindet sich in den 12 niederwertigsten Bits und die 4 höchstwertigen Bits sind auf Nullen gesetzt.

Abbildung 2.2: BayerRG Muster 2x2



RGB8planar: Dieses Pixelformat definiert ein 8-Bit RGB (rot, grün, blau) planares Format. Planar bedeutet, dass jede Farbe einen separaten Block an kontinuierlichem Speicher hat. Für dieses Pixelformat werden drei aufeinander folgende kontinuierliche Speicherblöcke verwendet. Es ergeben sich 24-Bit pro Pixel. Es werden ein Rot-Wert, zwei Grün-Werte und ein Blau-Wert verwendet, um ein Pixel eines anderen hier verwendeten Formats zu errechnen.

2.1.2 Anwendungsbeispiele

Um die Relevanz der Pixelformate und dessen Konvertierung deutlich zu machen werden im folgenden Anwendungsbeispiele aus der Industrie dargestellt.

Anwendungsbeispiel 1

Die meisten farbigen Kamerasensoren sind mit einer Art von Bayer-Farb-Filter versehen. Dieser Filter lässt pro Sensor Pixel lediglich die Lichtinformationen einer bestimmten Farbe entsprechend des Bayer-Musters durch. So entstehen Bilder, welche nicht besonders gut aussehen und die einen grün-Stich haben, weil doppelt so viele grüne Pixel vorhanden sind als rote oder blaue Pixel. Damit ein PC-Bildschirm die Bilder möglichst so anzeigen

kann, wie das menschliche Auge die Umwelt wahrnimmt, wird das Bayer-Muster-Bild in ein RGB-Bild konvertiert. Der PC-Bildschirm kann pro Bildschirmpixel einen roten, einen grünen und einen blauen Wert darstellen. Somit kann das RGB-Bild direkt angezeigt werden

Anwendungsbeispiel 2

Oft gilt, auf desto weniger Daten gerechnet werden muss, desto schneller ist die Anwendung. Das ist besonders der Fall bei der Bildverarbeitung. Daher kann es für Software, welche zum Beispiel Kanten in Bildern erkennt, sinnvoll sein die Eingangsdatenmenge möglichst klein zu halten. Es wäre möglich die Auflösung zu reduzieren oder sich auf einen Teilbildausschnitt zu beschränken. Aber gerade bei einem Farbbild kann die Datenmenge reduziert werden, indem eine Konvertierung auf ein Mono-Pixelformat gemacht wird.

2.2 Einfluss der Hardware auf die Geschwindigkeit der Konvertierung

Je schneller die Hardware, desto höher die allgemeine Leistung. Die Hardware, auf der ein Programm ausgeführt wird, hat einen entscheidenden Einfluss auf die resultierende Performance. Die Elemente der Hardware eines Computers mit dem meisten Einfluss auf die Leistung sind die CPU, der RAM und der Cache.

2.2.1 CPU

Die CPU ist die zentrale Recheneinheit eines Computers. Diese arbeitet die jeweiligen Instruktionen des Prozesses ab, welcher vom Scheduler die CPU zugewiesen bekommt. Moderne Computer haben oft CPUs mit mehreren Kernen. Ein Kern kann jeweils einen Prozess ausführen. Somit können mehrere Prozesse parallel von mehreren Kernen ausgeführt werden. Eine weitere Eigenschaft von modernen Computern ist es, das Hyperthreading. Hyperthreading ist eine Technologie, welche aus einem physikalischen Prozessor-Kern zwei virtuelle Kerne beziehungsweise zwei logischer Kerne macht.

Instruction Level Parallelism

Es gibt üblicherweise fünf Schritte, die jede CPU-Anweisung durchlaufen muss, bis diese vollständig abgearbeitet ist. Die Schritte sind Anweisung holen, Anweisung dekodieren, Anweisung ausführen, Daten aus dem Speicher holen und das Ergebnis in den Speicher zurückschreiben. Diese fünf Schritte für jede Anweisung sequentiell auszuführen würde erheblich länger dauern, als die Schritte gleichzeitig auszuführen. Also während eine Anweisung dekodiert wird, kann die nächste Anweisung bereits gleichzeitig eingeholt werden und so weiter. Dies wird auch 'Pipelining' genannt. Dadurch kann ein CPI Wert von theoretischen eine Instruktion pro Zyklus erreicht werden. Durch weitere CPU-Architektur Optimierungen kann ILP (Instruction Level Parallelism) umgesetzt werden. ILP bedeutet, dass von einem spezifischen Prozess mehrere Operationen parallel ausgeführt werden können.
Zu diesen Mechanismen zählen unter anderem 'Out-of-Order Execution', 'Speculative Execution', 'Branch Prediction' und 'Register Renaming'. Je niedriger der CPI-Wert ist, desto tedenziell effizienter ist der entsprechende Code. Ein hoher CPI-Wert kann unter anderem auf viele Branch mispredictions, Pipeline stalls, Cache misses oder andere Ursachen deuten. Es gibt

Date Level Parallelism

Jeder Prozessor-Kern hat eigene Register, welche direkt und ohne Verzögerung von der CPU benutzt werden können. Die modernen Computer besitzen, abhängig von ihrer Architektur, zusätzliche Vektorregister. Diese können genutzt werden, um spezielle Vektorbefehle auszuführen. Diese Vektorbefehle führen auf allen Werten im Vektorregister die gleiche Instruktion aus. Das Prinzip ist auch bekannt als SIMD. Dadurch kann auf mehreren Daten gleichzeitig die selbe Instruktion ausgeführt werden. Die CPU führt dabei nur eine einzelne Instruktion aus.
Die Rate mit der ein Prozessor Instruktionen ausführen kann wird durch seine Geschwindigkeit festgelegt. Diese wird heutzutage meist in Gigahertz(Ghz) angegeben. 1 Gigahertz entspricht 1000000000 Instruktionen pro Sekunde.

2.2.2 RAM

Der RAM(Hauptspeicher) ist Speicher im Computer, der von allen laufenden Programmen benutzt wird. RAM hat eine Kapazität die in Gigabyte angegeben wird. Der RAM hat ebenfalls eine Geschwindigkeit, welche in Megatransfers pro Sekunde angegeben wird. Daraus ergibt sich auch eine maximale Bandbreite, mit der Daten übertragen werden können. Für den, im Rahmen dieser Bachelorarbeit verwendeten, RAM (DDR4 2133) ergibt sich eine maximale Bandbreite von 17GBGB pro Sekunde[22].

2.2.3 Cache

Viele Informationen zum Thema 'Cache' kommen von dieser Quelle [3]. Die CPU ihre Berechnungen erledigen kann, müssen die zu verarbeitenden Daten in den Registern der CPU vorliegen. Diese jedes Mal aus dem RAM zu fetchen, würde im Vergleich zu der Taktung der CPU sehr lange dauern. Dies würde die Leistung erheblich mindern. Damit die Latenz möglichst gering ist, bis ein Wert im Register der CPU ist, gibt es unter Anderem den Cache. Der Cache hat in modernen Computern eine Hierarchie und ist in mehrere sogenannte Level unterteilt. Der Cache hat in den meisten Fällen drei Level und ist sehr CPU nah und sehr effizient. Der L1-Cache hat die niedrigste Latenz und somit die höchste Effizienz. Dafür kann L1-Cache am wenigsten Daten vorhalten. Der L3-Cache ist am langsamsten von den drei Cache Ebenen, dafür am größten. Der L3-Cache wird meistens von mehreren Kernen geteilt.
Latenz ist die Zeit von dem Anfordern der Daten, bis diese Daten verfügbar sind.

Cache hit und Cache miss

Ob Daten im Cache stehen und in welchem Cache-Level oder aus dem RAM geladen werden, ist für den Programmierer nicht direkt ersichtlich. Bevor ein Wert aus dem RAM geladen wird, prüft der Prozessor, ob der Wert bereits im Cache vorhanden ist. Wenn der Wert im Cache liegt, kann dieser mit einer sehr geringen Latenz in die CPU Register geladen werden. Das vorfinden von benötigten Daten im Cache wird auch Cache hit genannt. Das Gegenteil ist der Fall, wenn die Daten nicht im Cache vorliegen und erst aus dem RAM gefetcht werden müssen. Dies wird auch Cache miss genannt.

Cachelines

Auch wenn der Prozessor nur einen wenige Byte großen Wert für eine Berechnung fetcht, wird die gesamte Cacheline, in der dieser Wert steht, gelesen. Viele Prozessoren arbeiten mit Cachelines und deshalb wird auch beim Ändern eines einzelnen wenige Byte großen Wertes eine ganze Cacheline gelesen, ein Wert verändert und eine ganze Cacheline zurückgespeichert. Cache Speicher besteht aus Zeilen, auch Cachelines genannt.
Auf einem AMD Ryzen 7 3700x, der im Rahmen dieser Arbeit verwendet wird, ist eine Cacheline 64 Bytes groß.
Ein Nebeneffekt, der die Performance beeinflussen kann ist, dass das Lesen eines kleinen, zum Beispiel 4 Byte groß, Wertes dazu führen kann, dass zwei je 64 Byte große Cachelines gelesen werden müssen. Dies kann der Fall sein, wenn die 4 Byte ungünstig im Speicher liegen. Als Beispiel könnte die Anfangsadresse des 4 Byte großen Wertes bei 0x3E liegen. Damit der gesamte Wert in den Cache geladen werden kann, muss sowohl 0x00 bis 0x3F als auch 0x40 bis 0x7F gelesen werden.

Cache Arten

Der Cache besteht nicht nur aus mehreren Leveln, sondern kann auch unterschiedlich gemapped sein. Zum Einen kann der Cache ein sogenannter 'direct-mapped Cache' sein. Aus der Adresse im Hauptspeicher wird genau ein Adresse im Cache berechnet. Es kann bei der direct-mapped Cache Methode dazu kommen, dass zwei Adressen aus dem Hauptspeicher auf die selbe Adresse im Cache abgebildet werden. Bei abwechselndem Lesen der beiden Variablen kann es zu Cache misses kommen und es muss jedes Mal aus dem Hauptspeicher gelesen werden, statt den effizienteren Cache zu nutzen.
Eine weitere Cache Art ist der 'associative Cache'. Eine Adresse aus dem Hauptspeicher kann an mehrere möglichen Adressen im Cache gespeichert werden. Es wird auch von 'x-way associative Cache' gesprochen. Das 'x' steht für die Anzahl an möglichen Stellen im Cache, auf die eine gewisse Adresse gemapped werden kann.
Die letzte Variante ist der 'fully associative Cache'. Es kann ein Speicherblock an eine beliebe freie Stelle im Cache gespeichert werden.

3 Ist-Zustand

In diesem Kapitel wird der Ist-Zustand beschrieben. Dazu gehört eine Beschreibung der Struktur des Codes und eine Analyse auf Optimierungspotential. Die Rahmenbedingungen und ein finales Problemstatement werden hier definiert.
Der Ist-Zustand ist die Grundlage für eine qualifizierte Bewertung der möglichen Optimierungen für die Pixelformatkonvertierung bei Basler.

3.1 Basler: Bildverarbeitungspipeline

Die Bilddaten werden von der Kamera genauer von dem Sensor erfasst und an den Computer per Datenübertragungsmedium übertragen.

1. Erfassung der Bildinformationen im Sensor
2. Es werden Zeile für Zeile ausgelesen
3. Die Bilddaten werden über das Datenübertragungsmedium übertragen
4. Die Bilddaten werden auf dem Computer entgegengenommen und im Hauptspeicher gespeichert
5. Die Bilddaten aus dem Speicher lesen und in das entsprechende Pixelformat per ImageFormatConverter 3.3 umwandeln
6. Das umgewandelte Bild zurück in einen anderen Speicherbereich im Hauptspeicher schreiben
7. Das Bild durchläuft noch weitere Schritte in der „Bildverarbeitungskette“, welche nicht weiter relevant für diese Arbeit sind. Zum Beispiel Kantenerkennung, Machine Learning Anwendung und viele mehr.

3.2 Basler: Pixelformatkonvertierung

Die Pixelformatkonvertierung ist ein Schritt in der Bildverarbeitung, um unter anderem die rohen Bilddaten vom Sensor in ein Format zu konvertieren, welches von Bildverarbeitungssoftware besser verwenden kann. Die folgenden Pixelformate sind repräsentativ für den Hauptteil der Pixelformate und bildet die Basis für die Ist-Zustand Analyse und eine prototypische Implementation der möglichen Optimierungen.

Input-Pixelformate

1. Mono8
2. BayerRG12
3. RGB8Planar

Output-Pixelformate

1. Mono8
2. RGB8
3. RGB16

Bildgrößen

1. 5MP
    - Beispiel Kamera: a2A2590-60ucBAS
    - Pixelanzahl Höhe: 1944 Pixel
    - Pixelanzahl Breite: 2592 Pixel
2. 24,4MP
    - Beispiel Kamera: a2A5328-15ucBAS
    - Pixelanzahl Höhe: 4608 Pixel

KonvertierungsID	Input-Pixelformat	Output-Pixelformat	Bildgröße
1	Mono8	Mono8	5MP
2	Mono8	RGB8	5MP
3	Mono8	RGB16	5MP
4	BayerRG12	Mono8	5MP
5	BayerRG12	RGB8	5MP
6	BayerRG12	RGB16	5MP
7	RGB8Planar	Mono8	5MP
8	RGB8Planar	RGB8	5MP
9	RGB8Planar	RGB16	5MP
10	Mono8	Mono8	24.4MP
11	Mono8	RGB8	24.4MP
12	Mono8	RGB16	24.4MP
13	BayerRG12	Mono8	24.4MP
14	BayerRG12	RGB8	24.4MP
15	BayerRG12	RGB16	24.4MP
16	RGB8Planar	Mono8	24.4MP
17	RGB8Planar	RGB8	24.4MP
18	RGB8Planar	RGB16	24.4MP

Tabelle 3.1: Konvertierungstabelle

- Pixelanzahl Breite: 5328 Pixel

Die Bildgrößen wurden gewählt, weil die a2A2590-60ucBAS Kamera die meist-verkaufte Kamera bei Basler ist und 24.4 MP der größte Sensor ist, welcher in den Kameras der Basler AG verbaut wird.

3.3 Basler: ImageformatConverter

Der ImageFormatConverter ist eine Software-Klasse, welche die Konvertierung von einem Ausgangsformat in ein Zielformat durchführt. Der ImageFormatConverter unterstützt 59 Input-Pixelformate und elf Output-Pixelformate. Die Konvertierung läuft auf einem Thread und ist linear abhängig von der zu konvertierenden Datenmenge.
Der ImageformatConverter ist Bestandteil der sogenannten 'pylon Camera Software Suit', kurz 'pylon'. 'pylon' umfasst SDKs, welche die Programmierschnittstelle zwischen Kamera und Code anbietet. [2]

ImageformatConverter: Eigenschaften

Der ImageformatConverter ist in C++ programmiert. Die Implementation ist durch templates sehr stark verallgemeinert. Alle verwendeten Bibliotheken und Features sind plattformunabhängig, damit die Software sowohl unter Windows als auch unter Linux verwendet werden kann. Die Pixelformatkonvertierung wird von lediglich einem Thread ausgeführt. Bereits angewandte Optimierungen sind das verwenden des 'O2' Compiler-Flags, welches dem Compiler erlaubt den Code so gut es der Compiler kann auf Performance zu optimieren.

Basler: Konvertierungsschritte

Bei der Konvertierung von einem Pixelformat in ein anderes Pixelformat gibt es drei Phasen. Die erste Phase ist die Vorverarbeitung. Die Vorverarbeitung sorgt dafür, dass das zu konvertierende Bild korrekt gelesen werden kann. Die zweite Phase ist das Konvertieren. Das Konvertieren führt die eigentliche Konvertierung durch. Dafür werden die Pixelinformationen Pixel-weise in den Zielbuffer an die entsprechende Stelle kopiert. Im Falle von einer Konvertierung von einem bunten Pixelformat in ein monochromes Pixelformat, wird noch zusätzlich die Luminanz Pixel-weise berechnet. Die dritte Phase ist das Planarisieren. Das Planarisieren überprüft, ob eine Planarisierung notwendig ist und führt diese gegebenenfalls durch. Das wäre der Fall, wenn das Zielformat ein planares Pixelformat ist.

Ebenen der Pixelformatkonvertierung

Der original Code lässt erkennen, dass die Konvertierung auf verschiedene Ebenen herunter gebrochen werden. Die Ebenen lassen sich wie folgt beschreiben:

1. Daten-Ebene: Es kann entscheidend sein, wie die Bilddaten gespeichert sind zum Beispiel ob sie aligned im Speicher sind oder nicht.

2. Bild-Ebene: Der Funktionsaufruf, der pro Bild getätigt wird. Hier wird über jede Zeile des Bildes iteriert und eine entsprechende LineConverter::Convert Methode aufgerufen.

3. Zeilen-Ebene: Die LineConverter::Convert Methode macht die finale Konvertierung. In den meisten Fällen wird über alle Pixel der Bildzeilen iteriert und Pixel für Pixel konvertiert. Eine Ausnahme stellt die Konvertierung von Mono8 zu Mono8 dar, welche ein zeilenweises memcpy macht.

4. Pixel-Ebene: Diese Ebene wird durch keine eigene Klasse repräsentiert. Optimierungen in dieser Ebene haben potentiell den größten Einfluss auf die Performance.

Basler: Konkrete Verfahren der Konvertierungen

Die folgende Auflistung beschreibt was der Code für die jeweilige Konvertierung hauptsächlich macht.

- Mono8 -> Mono8: zeilenweise memcpy (Sonderfall, weil Quell- und Zielpixelformat identisch sind)
- Mono8 -> RGB8: ein Mono8 Grauwert wird zu je ein Byte rot, grün und blau Grauwert im Zielbild pro Pixel
- Mono8 -> RGB16: ein Mono8 Grauwert wird zu je zwei Byte rot, grün und blau Grauwert im Zielbild pro Pixel
- BayerRG12 -> Mono8: pixelweise Interpolation von zwei grün Werten, Luminanz errechnen aus RGB-Wert,
- BayerRG12 -> RGB8: pixelweise Interpolation von zwei grün Werten,
- BayerRG12 -> RGB16:pixelweise Interpolation von zwei grün Werten,
- RGB8planar -> Mono8: aus drei ein-Byte RGB-Werten wird ein Grauwert (1 Byte) pro Pixel im Ziel
- RGB8planar -> RGB8: aus drei ein-Byte RGB-Werten werden identische drei ein-Byte RGB-Werte pro Pixel im Zielbild. Die Art, wie die Pixeldaten im Speicher stehen ändert sich dabei ausschließlich.
- RGB8planar -> RGB16: aus drei ein-Byte RGB-Werten werden drei zwei-Byte RGB-Werte pro Pixel im Zielbild.

Um die Luminanz der Mono8 Pixel zu berechnen wird die Formel '(5*green + blue + 2*red)/8' verwendet.

3.4 Potential Analyse

Um eine entsprechende Optimierung umsetzen zu können, muss der zu optimierende Code auf sein Optimierungspotential analysiert werden. Den ersten Schritt haben die Kunden der Basler AG gemacht, indem sie Feedback geben und rückmelden, das die Anwendung teilweise zu langsam ist. Dies ist nicht ausreichend. Es werden mehr Details für eine sinnvolle Optimierung benötigt. Unter anderem ist es wichtig die Funktionen oder Code Zeilen zu finden, die viel Zeit im Prozessor beanspruchen, auch 'Hot Path' genannt. Als nächstes wird die Performance vom Hot Path genauer betrachtet und die Parallelität bestimmt. Es gibt noch andere Möglichkeiten um potentiell Optimierungsbedarf festzustellen, die nicht weiter behandelt werden.

3.4.1 Hot Path Analyse

Per Analyse mit dem AMD uProf Profiler (5.3) kann bestimmt werden, welcher Aspekt bei der Konvertierung am meisten CPU-Zeit beansprucht. Es gibt zwei große potentielle Fälle für besonders ̈heißen ̈Code. Das sind Schleifen und häufig aufgerufene Funktionen. Es ist zu erwarten, dass am meisten CPU-Leistung von der LineConverter::Convert Methode des ImageFormatConverters beansprucht wird, weil diese Methode für jede Bildzeile, also sehr häufig, pro Konvertierung aufgerufen wird und dort die eigentliche Konvertierung stattfindet.
Um den Hot Path mit dem Profiler zu bestimmen wird der Messaufbau verwendet wie in 5.2 beschrieben ist und repräsentativ für alle Konvertierungsmöglichkeiten wird die Konvertierung von RGB8planar nach Mono8 mit 24,4MP Bildern analysiert. Dieses Hot-Path Analyseergebnis zeigt, dass die LineConverter::Convert Methode 75% der CPU Zeit während der Testdurchführung beansprucht. Dadurch kann die zuvor aufgestellte Hypothese bestätigt werden. Für diese Bachelorarbeit kann der Code für das Zeilenkonvertieren als Hauptfokus betrachtet werden. Diese Analyse und das Ergebnis weder bestätigen noch widerlegen eine konkrete Möglichkeit zur Optimierung, da nicht ersichtlich ist, ob der Code mit seiner maximal möglichen Effizienz läuft.

3.4.2 Performance Analyse

Die Performance kann mit dem Profiling Profil 'Assess Performance' vom AMD uProf Profiler analysiert werden.

Abbildung 3.1: Hot-Path Konvertierung RGB8planar nach Mono8 mit 24,4MP Bildern

Functions	Modules	CPU_TIME (s) ▼	
Pylon::CRGBConverter<class Pylon::ColorPlanarLineConvert	Project7.exe	75.33%	
Pylon::FillImage<unsigned char,1>(struct Pylon::IImage &,un	Project7.exe	10.84%	
PylonUtility_v6_3.dll!0x7ffd1f5959b0	PylonUtility_v6_	4.82%	
PylonUtility_v6_3.dll!0x7ffd1f5959b4	PylonUtility_v6_	2.87%	
PylonUtility_v6_3.dll!0x7ffd1f5959bb	PylonUtility_v6_	1.97%	
PylonUtility_v6_3.dll!0x7ffd1f5959b9	PylonUtility_v6_	1.37%	
PylonUtility_v6_3.dll!0x7ffd1f5959be	PylonUtility_v6_	0.70%	
PylonUtility_v6_3.dll!0x7ffd1f5959c0	PylonUtility_v6_	0.30%	
PylonUtility_v6_3.dll!0x7ffd1f5959c3	PylonUtility_v6_	0.24%	
PylonUtility_v6_3.dll!0x7ffd1f5959f0	PylonUtility_v6_	0.01%	
atcuf64.dll!0x7ffd2b2f484d	atcuf64.dll	0.00%	

Diese Analyse wird an der gleichen exemplarischen Konvertierung wie in der 'Hot Path Analyse' gemacht.

Abbildung 3.2: Performance Analyse Konvertierung RGB8planar nach Mono8 mit 24,4MP Bildern

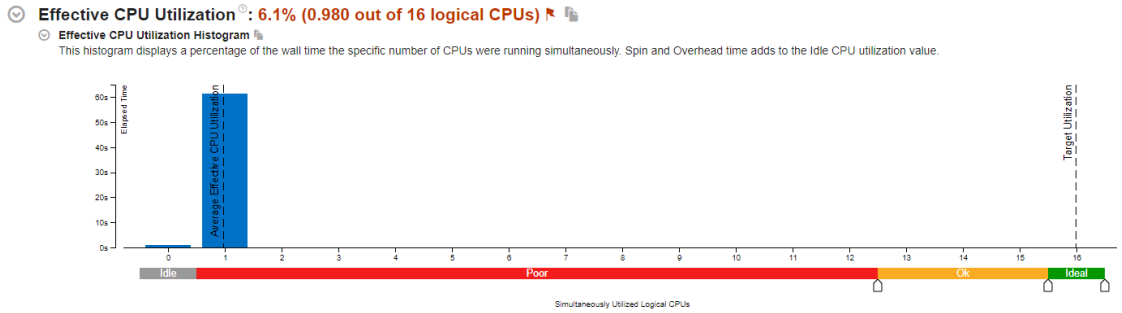
Functions	Modules	CYCLES_NOT_IN_HALT	RETIRED_INST	IPC	RETIRED_BRANCH_INSTRUCTION_MISPREDICTED_PTI	%RETIRED_BRANCH_INSTRUCTION_MISPREDICTED	DATA_CACHE_ACCESSES_PTI	MAB_ALLOC_PTI	%DATA_CACHE_MISS	MISALIGN_LOADS_PTI
Pylon::CRGBConverter<class Pylon::ColorPlanarLineConvert	Project7.exe	852903	2886273	3.38	0.08	0.11	276.36	2.43	0.88	
Pylon::FillImage<unsigned char,1>(struct Pylon::IImage &,un	Project7.exe	170563	360013	2.11	0.16	0.05	461.44	0.42	0.09	
PylonUtility_v6_3.dll!0x7ffcf8c459b0	PylonUtility_v6_	36856	77078	2.09	0.16	0.05	459.50	0.42	0.09	
PylonUtility_v6_3.dll!0x7ffcf8c459b4	PylonUtility_v6_	34637	72115	2.08	0.15	0.04	461.29	0.44	0.10	
PylonUtility_v6_3.dll!0x7ffcf8c459bb	PylonUtility_v6_	33218	69357	2.09	0.15	0.04	460.18	0.47	0.10	
PylonUtility_v6_3.dll!0x7ffcf8c459b9	PylonUtility_v6_	1523	2812	1.85	0.07	0.02	488.26	0.53	0.11	
PylonUtility_v6_3.dll!0x7ffcf8c459be	PylonUtility_v6_	1355	2852	2.10	0.14	0.04	433.03	0.39	0.09	
PylonUtility_v6_3.dll!0x7ffcf8c459c3	PylonUtility_v6_	678	1328	1.96	0.08	0.02	478.92	0.38	0.08	
PylonUtility_v6_3.dll!0x7ffcf8c459c0	PylonUtility_v6_	331	694	2.10			445.24	0.29	0.06	
PylonUtility_v6_3.dll!0x7ffcf8c459f0	PylonUtility_v6_	42	70	1.67			600.00	1.43	0.24	
PylonUtility_v6_3.dll!0x7ffcf8c459f9	PylonUtility_v6_	15	39	2.60			307.69			
PylonUtility_v6_3.dll!0x7ffcf8c456d0	PylonUtility_v6_	14	21	1.50			190.48	14.29	7.50	
RunTest([, ...)	Project7.exe	12	13	1.08			461.54			
PylonUtility_v6_3.dll!0x7ffcf8c440f8	PylonUtility_v6_	11								
PylonBase_v6_3.dll!0x7fffcf777f537	PylonBase_v6_3	11							10.00	
PylonUtility_v6_3.dll!0x7ffcf8c4430c	PylonUtility_v6_	10								
atcuf64.dll!0x7ffd2b2f49ae	atcuf64.dll	8								
PylonUtility_v6_3.dll!0x7ffcf8c45699	PylonUtility_v6_	8	3	0.38						
PylonUtility_v6_3.dll!0x7ffcf8c44358	PylonUtility_v6_	7	5	0.71			600.00			
PylonUtility_v6_3.dll!0x7ffcf8c459fb	PylonUtility_v6_	6	11	1.83			363.64			

Diese Analyse zeigt, dass der Original-Code keine auffälligen Performance Defizite aufweist, da der Profiler diese in roter Farbe markieren würde. Dies ist ein Anzeichen dafür, dass der Code bereits tendenziell effizient ist. Es kann dennoch möglich sein, den Code effizienter zu machen und schließt eine potentielle Optimierung nicht aus.

3.4.3 Parallelitätsanalyse

Um die Parallelität zu analysieren wird der Intel VTune Profiler 5.3 mit der 'Threading' Konfiguration verwendet. Diese Analyse wird an der gleichen exemplarischen Konvertierung wie in der 'Hot Path Analyse' und der 'Performance Analyse' gemacht. Dies wird einen groben Überblick über die Parallelität geben.

Abbildung 3.3: Parallelitätsanalyse Konvertierung RGB8planar nach Mono8 mit 24,4MP Bildern



Das Analyseergebnis zeigt, dass der Code in einem einzelnen Thread ausgeführt wird. Die verwendete Hardware bietet die Möglichkeit 16 Threads gleichzeitig auszuführen. Somit ist das Parallelisieren eine mögliche Optimierungsstrategie, welche umgesetzt, erneut analysiert und auf die Effektivität untersucht werden.

3.5 Rahmenbedingungen

Diese Ausarbeitung findet im Rahmen eines industriellen Kontextes statt. Daher ergeben sich typischerweise Anforderungen seitens des Unternehmens. Die Basler AG stellt folgende Rahmenbedingungen:

1. Der ImageFormatConverter muss auf auf allen von pylon unterstützten Prozessorplattformen und Betriebssystemen immer alle Formate konvertieren können.
2. Die Leistung (CPU Last ohne Multiprozessing Overhead) des Konverters darf nicht schlechter sein als im Stand pylon 6.2.
3. Die Leistung (CPU Last ohne Multiprozessing Overhead) des Konverters kann besser sein als im Stand pylon 6.2.

4. Der ImageFormatConverter kann, sofern vorhanden, eine GPU für die Konvertierung verwenden.
5. Die Laufzeit für das Konvertieren eines Bildes muss signifikant geringer sein als im Stand pylon 6.2.
6. Für bildverarbeitende Systeme mit Echtzeitanforderungen ist es wichtig für den Nutzer den Einsatz von CPU-Ressourcen (Anzahl Threads, optional Kern Zuordnung) steuern zu können.

3.6 Problemstatement

Das Ziel ist den bestehenden Code mit Fokus auf Leistung zu analysieren und Optimierung umzusetzen. Die Optimierungen müssen mindestens eine Steigerung der Leistung um den Faktor 2 erzielen, damit sich eine Optimierung für die Basler AG als lohnenswert darstellt. Durch diese beispielhafte Analyse des Original-Codes und einer prototypische Umsetzung von Optimierungen soll bestimmt werden, ob eine komplett neue Lösung konstruiert werden muss oder eine Überarbeitung im Sinne der Leistungsoptimierung am existierenden Code ausreichend ist.

4 Stand der Technik für Performanceoptimierungen

Es gibt in der Programmierung oft viele verschiedene Möglichkeiten das gleiche Ergebnis zu erzielen. Dabei können unterschiedliche Entwicklungsebenen die Performance beeinflussen. Es beginnt bei dem Software-Design und kann bis zu Implementationsentscheidungen von einzelnen Zeilen Code gehen. Der Fokus wird auf den Methoden zur Optimierung von Code liegen. Daher werden in diesem Kapitel gängige Optimierungsmethoden erläutert.

4.1 Single-Thread Optimierungsmethoden

Das Ziel der Single-Thread Optimierung ist, dass die Performance des aktuellen Codes, der in einem einzelnen Thread ausgeführt wird, soweit wie möglich optimiert wird, bevor dieser auf mehrere Threads verteilt werden kann. Somit wird eine möglichst hohe Verbesserung der Performance angestrebt.

4.1.1 Compiler Verwendung optimieren

Der Compiler erzeugt aus Code, den ein Programmierer geschrieben hat, Maschinencode, der von der CPU ausgeführt werden kann. Durch das setzten von Compiler-Flags kann der Programmierer den Compiliervorgang beeinflussen, zum Beispiel kann eine gewisse Tendenz gegeben werden, ob der Code schnell laufen oder wenig Speicherplatz verbrauchen soll. Der Compiler kann durch unterschiedlichste Methoden, wie Loop Unrolling, Branch Prediction, auto-Vektorisierung, Function Inlining und mehr, einen starken Performanceunterschied machen. Dabei haben die vielen unterschiedlichen Compiler jeweils

unterschiedliche Umsetzung dieser Optimierungsstrategien und unterschiedliche zusätzliche Features. Somit ist die Wahl des Compilers und die Konfiguration relevant, wenn die Performance wichtig ist. Für einen übersichtlichen Vergleich siehe [7, 76].

4.1.2 Bibliotheken Verwendung optimieren

Software Bibliotheken erleichtern nicht nur den work-flow oder machen Code übersichtlicher, sondern es können auch besonders effiziente Funktionen in Bibliotheken zusammen gefasst werden. Dies erleichtert vielen Programmierern leistungsstarken Code zu entwickeln ohne das notwendige Detail-Wissen für performanten Code. Bibliotheken, wie die C++-Standardbibliothek, sind für ein breites Anwendungsgebiet entwickelt worden. Dadurch sind diese Bibliotheken vielseitig einsetzbar, aber es entsteht Platz für Optimierungen. Speziell für einen Anwendungsfall entwickelter Code ist oft performanter, als die universellen Bibliotheken.

4.1.3 Verwendete Algorithmen optimieren

Verwendete Algorithmen können oft an vielen Stellen optimiert werden. Algorithmen werden mit der O-Notation in ihrem Laufzeitverhalten beschrieben. So ist ein einfacher Bubblesort mit einer O-Notation von O(n$\hat{2}$) deutlich langsamer als ein Timsort mit einer O-Notation von O(n log2 n). Dadurch ist bereits die Wahl des Algorithmus entscheidend wie schnell dieser laufen kann. Nicht nur das Laufzeitverhalten sollte betrachtet werden, da teilweise auch vermeintlich langsame Algorithmen bei kleinen Datenmengen sehr effektiv sein können. Es gibt weitere Möglichkeiten verwendete Algorithmen zu optimieren.[8, 91]

4.1.4 Speichermanagement optimieren

Speicher ist nicht nur eine begrenzte Ressource in Computern, sondern spielt auch eine essentielle Rolle beim Ausführen von Programmen. Dateien werden auf der Festplatte gespeichert und bei Bedarf von dort geladen. Die Dateien können von wenigen Kilobytes bis viele Hunderte Gigabytes groß sein. Zwischen dem Speicher auf der Festplatte und den Registern mit denen eine CPU direkt arbeiten kann, liegen mehrere Schichten verschiedener Speicher. Dazu gehören Der Hauptspeicher (RAM), Caches (meistens Level

1 bis Level 3) und final die Register. Die unterschiedlichen Speicher haben eine unterschiedliche Latenz, bis ein Datum gelesen wurde und in einem Register der CPU steht. Daher ist es sehr wichtig, wie mit Speicher und den Daten im Speicher umgegangen wird. Gerade das Allozieren und Freigeben von Speicher zur Laufzeit eines Programms ist mit Performance-Kosten verbunden.

Das leichtsinnige Verwenden von dynamisch allozierten Variablen ist einer der größte Leistungsminderer in C++ Programmen. Das bedeutet, dass es ein großes Potential für Optimierungen bietet sich dynamisch allozierte Variablen im Programm zu betrachten. C++ stellt eine menge Features bereit, die dynamisch allozierte Variablen verwenden, wie die zum Beispiel C++-Standardbibliothek container, smart pointer und strings. Diese erleichtern das produktive Entwickeln von Programmen. Wenn Leistung allerdings relevant ist, dann ist 'new' nicht immer sinnvoll. Das Ziel ist es nicht auf solche Features komplett zu verzichten, sondern Aufrufe am Memory Manager mit bedacht zu machen und überflüssige Aufrufe reduzieren.

Dynamische Variablen und Einfluss auf die Performance

Die meisten C++ Statements kosten höchsten ein paar Speicherzugriffe, weil C++ direkt in Maschinen-Code übersetzt und vom Computer ausgeführt wird. Die Kosten um Speicher für dynamische Variablen zu allozieren werden in Tausenden von Speicherzugriffen gemessen.
Konzeptionell durchsucht eine Speicher-Allokationsfunktion eine Kollektion an freien Speicherblöcken um die Anfrage zu erfüllen. Wenn die Funktion einen Speicherblock der richtigen Größe findet, wird dieser aus der Kollektion entfernt und zurückgegeben. Falls ein Speicherblock gefunden wird, der viel größer als der angefragte Speicherblock ist, so kann dieser unterteilt werden und nur ein Teil zurückgegeben werden. Falls kein freier Speicher in der entsprechenden Größe vorhanden ist, macht die Funktion einen Aufruf am Betriebssystem Kernel um einen weiteren großen Speicherblock vom verfügbaren Systemspeicher zu erhalten. Dies ist sehr zeitaufwendig. Der neue Speicher kann noch nicht gecached sein und würde beim ersten Zugriff auf dein neuen Speicher für eine zusätzliche Verzögerung sorgen. Es kann aufwendig sein die Liste der freien Speicherblöcke zu durchsuchen, weil diese oft verstreut sind und weniger wahrscheinlich gecached sind, als Speicherblöcke, die dem laufenden Programm bekannt sind. Die Kollektion der verfügbaren Speicherblöcke ist eine geteilte Ressource, welche von allen Threads eines Programms

verwendet werden. Jegliche Änderungen an dieser Kollektion müssen Thread-safe sein. Im Fall, dass mehrere Threads häufig Speicher allozieren oder freigeben, kann es zu Streit um den Memory Manager kommen. Alle bis auf ein Thread müssen warten. [13, 118]

Pre-allozieren von dynamischen Variablen

Da Container wie der std::vector und der std::string, durch hinzufügen von Elementen ihren dynamischen internen Speicher erweitern müssen, wenn dieser voll ist, müssen teure Aufrufe am Memory Manager gemacht werden. Beide Container bieten die Funktion reserve(size_t n) an. Mit dieser Funktion kann genug Speicher zuvor alloziert werden um n-Einträge halten zu können. Dadurch wird das allozieren von neuem Speicher verhindert bis der container n-Einträge hält. Das funktioniert besonders gut, wenn vorher absehbar ist, wie viele Einträge in dem container gehalten werden. Ein weiterer Aspekt ist, dass dynamisch allozierte variablen nicht in Schleifen erzeugen, sondern davor. In der Schleife sollte zum Beispiel an einem std::string nur noch clear() aufgerufen werden. Diese Funktion setzt die interne Größe (size) auf 0, aber der allozierte Speicher (Kapazität) bleibt auf dem Wert zu vor. So kann pro Schleifendurchlauf das Allozieren von Speicher eingespart werden. [13, 127]

4.1.5 Weniger kopieren

Die klassische Definition von C [12] erlaubte ausschließlich primitive Typen (zum Beispiel int, char und pointer) direkt zugewiesen zu werden, die typischerweise in einen Maschinen-Register passten. Daher war jede Zuweisung a = b effizient und generierte nur ein bis zwei Instruktionen. In C++ sind solche Zuweisungen genau so effizient.
Es gibt einfach aussehende Zuweisungen, die nicht effizient sind in C++. Wenn a und b Instanzen von der Klasse BeispielKlasse sind, dann sorgt die Zuweisung a = b dafür, dass der Zuweisungsoperator der BeispielKlasse aufgerufen wird. BeispielKlasse kann dutzende an Instanzvariabeln haben, die kopiert werden müssen. Wenn unter den Instanzvariabeln auch dynamische Variablen sind, kann ein Kopiervorgang dazuführen, dass Aufrufe am Memory Manager gemacht werden müssen. Wenn BeispielKlasse ein Array, Vector oder anderen Container mit vielen Tausenden Einträgen hält, dann sind die Kosten einer Zuweisung signifikant und mindern die Performance.
Eine Deklaration und Initialisierung BeispielKlasse a = b ruft den CopyConstructor auf.

Dieser kann im Grunde das gleich wie der Zuweisungsoperator machen und zu der gleichen Situation führen.
Daher sollte bei besonders häufig aufgerufenem Code auf folgende Situationen geachtet werden [13]:

- Initialisierung (Constructor Aufruf)
- Zuweisung (Zuweisungsoperator Aufruf)
- Funktionsargumente (jeder Argument-Ausdruck wird entweder per move-Operator oder Copy-Constructor in das jeweilige formale Argument überführt)
- Funktionsrückgabewerte (Move oder Copy-Constructor Aufruf, eventuell sogar doppelt)
- Elemente in einen standard Bibliothek container einfügen (Elemente werden per move-Operator oder Copy-Constructor Aufruf eingefügt)
- Einfügen in einen Vector (Alle Elemente werden kopiert oder gemoved, wenn der Vector neuen internen Speicher alloziert)

4.1.6 Berechnungen entfernen

Die Berechnung mit der besten Performance ist die Berechnung, die nicht ausgeführt werden muss.
Es gibt viele Berechnungen, die die CPU stark beanspruchen und teuer sind. Diese Berechnungen auf ein Minimum zu reduzieren ist entsprechend wichtig. Es ist möglich Berechnungen vor den eigentlichen Hot-Path zu ziehen. Diese Berechnung kann entweder an einer Stelle vorher im Programm-Code, zur Link-Zeit, oder zur Compile-Zeit berechnet werden. Die Berechnungen dichter an der Stelle im Code machen, wo sie auch benutzt werden. So besteht die Chance, dass die Ergebnisse noch im Cache sind. [8, 147]

4.1.7 Verwenden von effizienteren Datenstrukturen

Datenstrukturen sind Abstraktionen von Daten. Dies kann in C++ in Form von structs oder Klassen erreicht werden. Eine Datenstruktur kann mehrere interne Variablen von unterschiedlichen Typen beinhalten. Dazu zählen die primitiven Typen und weitere Objekte von beliebigen Klassen. Die internen Variablen einer Klasse werden im Speicher

entsprechend angeordnet. Es kann sinnvoll sein, das durch ändern der Deklarierungsreihenfolge eine Leistungssteigerung erzielt wird. Zum Beispiel kann eine häufig gelesene Instanz-Variable als erstes in einer Klasse deklariert werden und dadurch Zeit sparen. [1]

4.1.8 Cacheline Alignment

Caches sind sehr klein verglichen mit dem Rest des Speichers in einem Computer. Dafür sind Caches deutlich schneller als der Hauptspeicher. Dieser ist in Cachelines aufgebaut 2.2.3. Um das Laden von zwei Cachelines bei Daten, die kleiner sind als eine Cacheline zu vermeiden, können die Daten-Adressen ausgerichtet werden. Dabei gilt, dass die jeweilige Adresse durch die Größe einer Cacheline teilbar sein muss. Der Performancegewinn ist besonders ersichtlich, wenn mit vielen kleinen Daten gerechnet wird, die jeweils mehr als eine Cachelinegröße auseinander im Speicher liegen. Bei großen Daten, die viel zusammenhängenden Speicher belegen, ist der Performancegewinn eher gering. Das Gleiche gilt auch für viele kleine Daten, die in zusammenhängenden Speicherblöcken sind. Die CPU kann beim Zugriff auf den nächsten Wert einen Cache hit fest stellen, da dieser Wert mit dem vorherigen zusammen in einer Cacheline in den Cache geladen wurde. Es kann die Performance theoretisch maximal verdoppelt werden durch Cacheline Alignment.

4.2 Vektorisieren

Vektorisieren ist eine Methode, die sowohl die CPU Architektur, den Compiler und den geschriebenen C++ Code betrifft. Zu der Methode gehört das Verwenden der SIMD (Single Instruction Multiple Data) Befehle. Dies kann nur gemacht werden, wenn die CPU Architektur dies unterstützt. Da es nicht nur eine Art von SIMD Befehlssätzen gibt, kommt es auf die CPU an, welche Befehlssätze verwendet werden können. Dazu kommt, dass der Compiler ebenfalls diese Befehlssätze kennen muss, sonst kann er keine entsprechenden Maschinen Befehle erzeugen. Als letztes muss der geschriebene C++ Code so strukturiert sein, dass dieser in SIMD Befehle umgewandelt werden kann. Dies geschieht typischer Weise in Schleifen. Dort darf es zwischen den einzelnen Schleifendurchläufen keine Daten-Abhängigkeiten geben. Zum Beispiel darf ein berechneter Wert nicht von einem anderen Wert aus dem Schleifendurchlauf zuvor abhängig sein. Für einen Überblick über konkreten die SIMD-Befehle siehe [11].

4.2.1 SIMD-Vektorbefehlssätze

Über die Zeit haben sich nicht die die Anzahl der Kerne oder die Geschwindigkeit der Prozessoren weiter entwickelt, sondern auch die Vektorregister, die dazugehörigen Befehlssätze und Verwendungsmöglichkeiten.

SSE und SSE2

SSE steht für 'Streaming SIMD Extensions'. Mit SSE können Vektoren mit bis zu vier Gleitkommawerten mit einfacher Genauigkeit berechnet werden. Skalare Gleitkommaanweisungen wurden entsprechend hinzugefügt.
Durch SSE2 wird die Berechnung mit 128-Bit Vektoren mit einfacher und doppelter Genauigkeit ermöglicht. Außerdem kann mit 1-, 2-, 4- und 8-Byte Ganzzahlwerten gerechnet werden. Es wurden Skalaranweiseungen mit doppelter Genauigkeit hinzugefügt. [17]

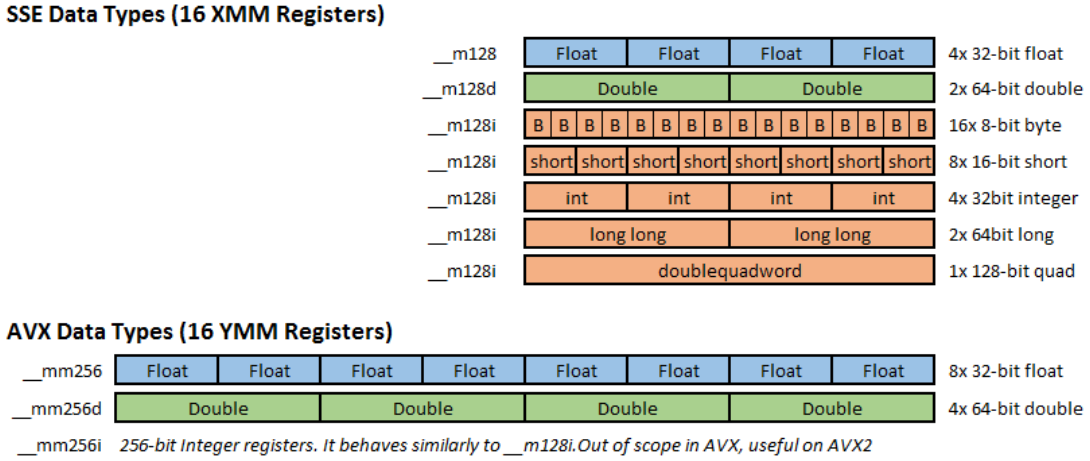
AVX, AVX2 und AVX512

AVX steht für 'Advanced Vector Instructions'. AVX führt eine alternative Anweisungscodierung für Vektor- und Gleitkomma-Skalaranweisungen ein, die Vektoren mit 128 Bits oder 256 Bits erlaubt und alle Vektorergebnisse auf die vollständige Vektorgröße erweitert.
AVX2 ermöglicht es FMA-Anweisungen (Fused Multiply-Add) zu verwenden und erweitert die meisten ganzzahligen Vorgänge auf 256-Bit Vektoren.
AVX512 für zusätzlich 512-Bit Vektoren ein und weitere Features. [16] Multiplikation und Additionen von Werten können vom Compiler mit FMA-Anweisungen zusammengefasst und optimiert werden.

SIMD Datentypen

SIMD Befehle nutzen spezielle SIMD-Register in der CPU, um mit einem Befehl auf mehreren Daten gleichzeitig zu rechnen. Dafür können im Code spezielle Datentypen verwendet werden, die auf diese Register zurück greifen. Die entsprechenden Datentypen sind in der folgenden Grafik dargestellt.

Abbildung 4.1: Datentypen AVX und SSE [26]



Es kann zu Performance Problemen kommen, wenn sowohl SSE als auch AVX Datentypen verwendet werden, weil die XMM und YMM Register überlappen.

4.2.2 Intel IPP

Intel IPP(Integrated Performance Primitives)[9] ist eine multi-threaded Software Bibliothek mit dem Ziel, die Leistung über unterschiedliche Plattformen hinweg zu zu steigern. Die Bibliothek wird von Intel entwickelt und Teil der Intel oneAPI.
Intel IPP hilft SIMD effizienter zu verwenden. Es ist besonders Leistung stark im Bereich von sehr aufwändigen Berechnungen. Im dem Bereich Computer Vision bietet Intell IPP circa 500 Funktionen (primitives) an.

4.3 Parallelität erhöhen

Die Parallelität erhöhen bezieht sich auf das Verteilen der Arbeit auf mehrere Threads und das Verteilen auf mehrere Kerne [7, 108]. Ein Kern kann nur eine bestimmte Leistung erbringen. Das Verteilen auf mehrere Threads ist insofern sinnvoll, da ein Kern zwei Threads sehr effektiv parallel ausführen kann. Bei zwei Threads pro Kern kann der Kern die Kontextinformationen lokal vorhalten. Die Wechsel zwischen den zwei parallelen Threads auf einem Kern sind dadurch sehr effizient.

Eine Aufgabe braucht Zeit t um von einem Kern bearbeitet zu werden. Die Zeit kann maximal auf 1/n bei n-Kernen reduziert werden.
Durch die richtige Anzahl an Threads kann der theoretische Performance Gewinn also gesteigert werden. Als Beispiel wäre, mit der ihm Rahmen dieser Ausarbeitung verwendeten Hardware 5.2.3, eine maximale Leistungssteigerung von 1600% möglich.
Bei einer Verwendung von mehr Threads würde sich die Leistung senken, im Vergleich zum Optimum, weil die Kontextwechsel einen höheren Overhead haben, als das sie Leistung bringen. Es gibt viele verschiedene Thread-Implementationen. Zu den bekannteren Varianten gehören OpenMP und std::thread.

4.3.1 OpenMP

OpenMP(Open Multi-Processing)[18] ist eine Multi-Plattform API für paralleles Programmieren in unter anderem C++. Zur Benutzung fügt der Programmierer pragmas in den Code ein, die das parallele Verhalten beschreiben. OpenMP ist ein Modell von Parallelität mit Fokus auf numerische Berechnungen. OpenMP entwickelt sicher teilweise Richtung GPU Programmierung. Es kann unter Windows und Linux verwendet werden. OpenMP vereinfacht es dem Entwickler for-Schleifen zu parallelisieren und übernimmt dabei die Entscheidung über die Anzahl der Threads, die dafür verwendet werden. Es ist möglich dies zu beeinflussen.

4.3.2 std::thread

Der std::thread[25] ist Teil der C++-Standardbibliothek und somit weit verbreitet in der C++Softwareentwicklungscommunity. Diese Thread-Implementation kann auf mehreren Plattformen verwendet werden. Ein std::thread Objekt bietet nicht von sich aus die Möglichkeit wiederverwendet zu werden. Es lässt dem Entwickler die Entscheidung, über die Anzahl der verwendeten Threads und die Verwendung.

4.4 Erwähnenswertes

Zusätzlich zu den bisher beschrieben Optimierungsmöglichkeiten gibt es viele weitere Optimierungsverfahren. Ein paar Weitere werden im Folgenden erwähnt.

4.4.1 Amdahl's Gesetz

Gene Amdahl war ein Computer Ingenieur und Vorreiter in seinem Feld. Amdahl's Gestz wurde nach ihm benannt und beschreibt wie viel Performance insgesamt Steigt, wenn ein Teil des Codes optimiert wird.

$$S_{T} = \frac{1}{(1-P) + \frac{P}{S_{P}}} \tag{4.1}$$

ST ist der Anteil der Verbesserung in der Gesamtlaufzeit der Anwendung durch eine Optimierung, P ist der Anteil der ursprünglichen Gesamtlaufzeit des zu optimierenden Codes und SP ist der Anteil der Verbesserung in dem optimierten Teil P. [8, 31]

4.4.2 Halide

Halide [20] ist eine Programmiersprache, welche speziell entwickelt wurde um Bildverarbeitung besonders performant zu machen. Anstatt eine stand-alone Programmiersprache zu sein, ist Halide wie ein Framework und kann in C++ wie eine Bibliothek importiert und benutzt werden. Es basiert auf einer speziellen Vorgehensweise. Es wird eine Pipeline definiert und Halide kann diese optimieren.
Halide ist besonders interessant im Bereich der Bildverarbeitung und ähnlichen Anwendungsfällen.

4.4.3 'ispc: A SPMD Compiler'

Der Intel SPMD Program Compiler (ispc) wird mit dem primären Fokus auf hohe CPU Performance entwickelt. SPMD steht für 'Single Program, Multiple Data' und ist ein Modell, welches einzelne Aspekte eines Programms auf eine SIMD Ausführungseinheit abbildet. Der ispc verwendet die Möglichkeiten der Parallelität von Multi-Kern Prozessoren und den SIMD-Vektor Befehlen. [19]

4.4.4 Mikroarchitektur-spezifische Optimierung

Es ist möglich gewisse Performance Eigenschaften der Mikroarchitektur der Hardware auszunutzen und den Code daraufhin zu optimieren. Dafür kann spezieller Assembler Code geschrieben werden. [6]

4.5 Fazit

es wurden verschiedene Optimierungsmethoden beschrieben. Es können im Rahmen dieser Bachelorarbeit nicht alle Methoden umgesetzt werden. Aus diesem Grund wird sich auf drei Methoden für eine prototypische Implementation entschieden.
Die erste Optimierungsmethode ist das 'Cacheline Alignment'. Diese Methode wird Implementiert, weil die Basler AG bereits vor einigen Jahren Cacheline Alignment verwendet hat. Dies hat, laut Angaben des Fachbetreuers, keinen Beitrag zu einer besseren Performance geleistet. Dies wird daher im Rahmen dieser Ausarbeitung überprüft und beurteilt.
Als zweite Optimierungsmethode wird 'Vektorisierung' ausgewählt. Vektorisierung ist sehr vielversprechend, was den maximalen Performancegewinn betrifft. Es werden im Rahmen der Vektorisierung zwei Varianten betrachtet. Zum Einen wird eine händische Vektorisierung exemplarisch umgesetzt und zum Anderen wird die Intel IPP API verwendet. Diese Varianten können schließend verglichen werden.
Als dritte Optimierungsmethode wird 'Parallelisierung' implementiert. Da der Code auf einem Multi-Kern Prozessor ausgeführt wird und nur ein Thread verwendet wird, bietet dies eine sehr gute Möglichkeit für mehr Performance. Es werden zwei Varianten der Parallelisierung umgesetzt. Die Erste ist eine Parallelisierung mit OpenMP und die Zweite ist eine händische Implementation mit std::thread.

5 Methodik

In diesem Kapitel wird die Struktur für die Versuchsdurchführungen beschrieben. Es wird auf die Messmethode eingegangen und der Versuchsaufbau festgelegt, sowie Hardware-Spezifikationen, die für diese Arbeit relevant sind, festgelegt.

5.1 Messmethode

Der verwendete Timer ist ein von der Basler AG programmierter Timer. Für eine möglichst akkurate Zeitmessung wird ein Timer vor der Konvertierung von mehreren Bildern gestartet und danach gestoppt. Das Messen von jeder einzelnen Konvertierung könnte die Laufzeit direkt beeinflussen, da das Konvertieren, der Timer und das Speichern der Messergebnisse im gleichen Prozess im gleichen Thread ausgeführt werden. Um möglichst wenig Fremdeinwirkung auf die Laufzeit der Konvertierung bei der Messung zu haben, ist es wichtig, dass:

- - Im Release gebaut wird
- - Die x64 Variante gebaut wird
- - Kein Debugger verwendet wird
- - Keine Printausgaben, Logs oder andere nicht notwendige IO-Operationen während der Messung gemacht werden
- - Der PC nicht im Energiesparmodus sonder auf einer gewissen Höchstleistungseinstellung läuft

5.2 Versuchsaufbau

Um eine geeignete Laufzeitmessung durch zu führen, werden in einem Beispiel Code 100 Bilder mit dem jeweiligen Input-Pixelformat generiert. Die Bilder haben eine Größe von 5MP und 24,4MP. Die Pixelwerte sind für die Konvertierung irrelevant und haben keinen Einfluss auf die Laufzeit. Die erzeugten Bilder sind somit vergleichbar. Dennoch werden die Bilder mit der Methode 'FillImage()' mit Werten gefüllt A.1. Nachdem die Bilder generiert wurden wird ein Timer gestartet, die 100 Bilder in das jeweilige Output-Pixelformat konvertiert, die Konvertierung 20 Mal durchgeführt und der Timer gestoppt. Die berechnete gesamte Datenmenge in Megabytes durch die gemessene Zeit geteilt, um die Leistung in Megabyte pro Sekunde zu erhalten. Die konkrete Berechnung ist im folgenden Code-Ausschnitt dargestellt.

```
1 double inputBytes = numberOfRepetitions * cCountImages * ComputeBufferSize(
    INPUT_PIXELTYPE, cWidth, cHeight);
2 double outputBytes = numberOfRepetitions * cCountImages * ComputeBufferSize(
    OUTPUT_PIXELTYPE, cWidth, cHeight);
3 double megaBytePerSecond = ((inputBytes + outputBytes) / 1000000.0) / watch.
    ElapsedTime();
```

Listing 5.1: Performance Berechnung

INPUT_PIXELTYPE und OUTPUT_PIXELTYPE sind jeweils abhängig von der Konvertierung. ComputeBufferSize() berechnet die Größe eines Bildes. Dafür wird das Pixelformat, die Höhe und die Breite übergeben. Es wird die Größe eines Bildes in Bytes zurückgegeben. Dies wird mit der Anzahl an Bildern und der Anzahl an Wiederholungen multipliziert um je die gesamte Input-Datenmenge und Output-Datenmenge zu berechnen. Anschließend werden die beiden Werte addiert und durch 1000000 geteilt, um die Gesamt-Datenmenge in Megabyte zu berechnen. Dieser Wert wird durch den return-Wert der Methode 'watch.ElapsedTime()' geteilt und das Ergebnis ist die finale Performance in Megabyte pro Sekunde.

5.2.1 Ablaufübersicht

1. 100 Bilder werden mit dem entsprechenden Input-Pixelformat und Größe erzeugt
2. Die 100 Bilder werden mit Werten gefüllt und die Output Speicher werden ein Mal beschrieben

3. Timer starten
4. 100 Bilder in jeweilige Output-Pixelformat konvertieren
5. Schritt 4 20 Mal wiederholen
6. Die Anzahl der konvertierten Bytes durch die gemessene Gesamtzeit teilen und Ergebnis anzeigen

Um Zufallsbefunde auszuschließen werden je Konvertierungskombination drei Durchläufe durchgeführt und der höchste Wert für die Auswertung verwendet.
Das ganze wird für jede Konvertierung 3.1 ausgeführt und die Ergebnisse in einem Excel Dokument dokumentiert.

5.2.2 Korrektheit der Konvertierung

Bei der Optimierung kann es zu Bildfehlern bei der Konvertierung kommen. Da die Korrektheit entsprechend sichergestellt werden muss, muss das Ergebnis der Konvertierung überprüft werden können.
Um die Korrektheit der Konvertierung zu prüfen, gibt es die CorrectnessCheck Methode, welche zwei Bilder als Input hat und einen Boolean zurück gibt, abhängig davon, ob die Konvertierung korrekt ist. Die Korrektheit wird überprüft, indem in der Methode jeweils die Pixel an der gleichen Position miteinander verglichen werden. Zusätzlich kann in einem extra Durchlauf, der nicht für die Performance Messung geeignet ist, das konvertierte Bild angezeigt und von der Person vor dem Monitor begutachtet werden.

5.2.3 Spezifikationen des Host-Rechners

Da die Konvertierung auf einem Host-Rechner durchgeführt wird und die Zeit der Konvertierung von der entsprechenden Ausstattung des Host-Rechners abhängig ist, ist es relevant diese zu beschreiben. Es sind die relevanten Spezifikationen aufgeführt.

CPU

- Model: AMD Ryzen 7 3700X
- Taktung: 3,60 GHz
- Kerne: 8 (16 logische)

Physikalischer Speicher [23]

- Level 1 Data Cache Speichergröße : 32 KB pro Kern (8-way Cache)
- Level 1 Instruction Cache Speichergröße : 32 KB pro Kern (8-way Cache)
- Level 2 Cache Speichergröße : 512 KB pro Kern (8-way Cache)
- Level 3 Cache Speichergröße : 2 x 16 MB geteilt mit jeweils 4 Kernen (16-way Cache)
- RAM Speichergröße : 32 GB (DDR4 2133)

Betriebssystem

- Betriebssytem: Microsoft Windows 10 Pro
- Systemtyp: x64

5.3 Verwendete Softwaretools und Compilerkonfiguration

Es gibt eine Vielzahl an Softwaretools zum Performance Analysieren, Messen und Code Compilieren. Es wurde sich im Rahmen dieser Arbeit auf ein Softwaretool zum Analysieren und eins zum Messen der Zeit und einen Compiler beschränkt. Die verwendeten Softwaretools werden kurz beschrieben und wie diese konfiguriert sind für die Durchführung der Messungen.

Intel V-Tune

Intel bietet ein Analyse-Tool namens 'Intel V-Tune'[10] an. Intel V-Tune bietet eine Vielzahl an Möglichkeiten an um den Code zu analysieren. Darunter ein Profiler, welches im Rahmen dieser Analyse verwendet wird um den 'Hot-Path' zu ermitteln. Der Hot-Path zeigt die Stellen im Code auf, welche am meisten Potential für Performance-Verbesserungen haben. Ein weiteres Feature ist, dass der Intel V-Tune Profiler einen Einblick gibt in was Leistungseinbüßen sein könnten. Intel-CPU spezifische Features sind auf dem verwendeten AMD Prozessor nicht verfügbar, dennoch teilweise es für Analysen verwendbar. Der Profiler ist in der Intel oneAPI enthalten.

AMD uProf

Der AMD µProf[4] ist ein Profiling Tool, welches von AMD hauptsächlich für AMD-Prozessoren entwickelt wurde. Es bietet verschiedene Möglichkeiten zur Analyse von Code. Eines der wichtigsten Analyseoptionen ist die 'Assess Performance' Konfiguration. Diese bietet einen Überblick über potentielle Performance-Schwachstellen im Code. Diese Analyse-Software wird ein essentielles Werkzeug für die weitere Bachelorarbeit.

Compiler Konfiguration

Der Compiler macht heutzutage nicht mehr ausschließlich eine eins zu eins Übersetzung von Code in Maschinenbefehle, sonder spielt eine wichtige Rolle in der Optimierung. Zum Code Compilieren wird der Microsoft C++ (MSVC) Compiler mit der 'Visual Studio 2019 v142' Version. Für alle Compliliervorgänge, die Code zum Performance Messen erzeuge, wird folgende Liste an Compiler-Flags verwendet.
'/permissive- /GS /GL /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /sdl /Zc:inline /D "NDEBUG/D "_CONSOLE/D "_UNICODE/D ÜNICODE/errorReport:prompt /WX- /Zc:forScope /arch:AVX2 /Gd /Oi /MD /openmp /FC /EHsc /nologo /Ot /diagnostics:column '
Diese Liste ist für Reproduzierbarkeit der Ergebnisse in diesem Umfang angegeben. Das einzige Compiler-Flag für Performance Optimierung, welches die Basler AG zum Compilieren verwendet ist '/O2'[24].

Basler Timer Klasse: CStopWatch

Die CStopWatch-Klasse ist ein in C++ implementierter Timer, welcher innerhalb der Basler AG programmiert wurde. Diese Klasse hat den PerformanceCounter als zu Grunde liegenden Counter und dieser hat eine Auflösung von unter einer 1 Mikrosekunde.

6 Evaluation

In diesem Kapitel werden die Messergebnisse der jeweiligen Optimierungsthesen dargestellt und evaluiert entsprechend ihrer Effektivität. Aus den Optimierungsmethoden werden Optimierungsthesen definiert und jeweils anhand von relevanten Konvertierungsbeispielen getestet. Es werden für jede These Änderungen am existierenden Code gemacht und hier dokumentiert.

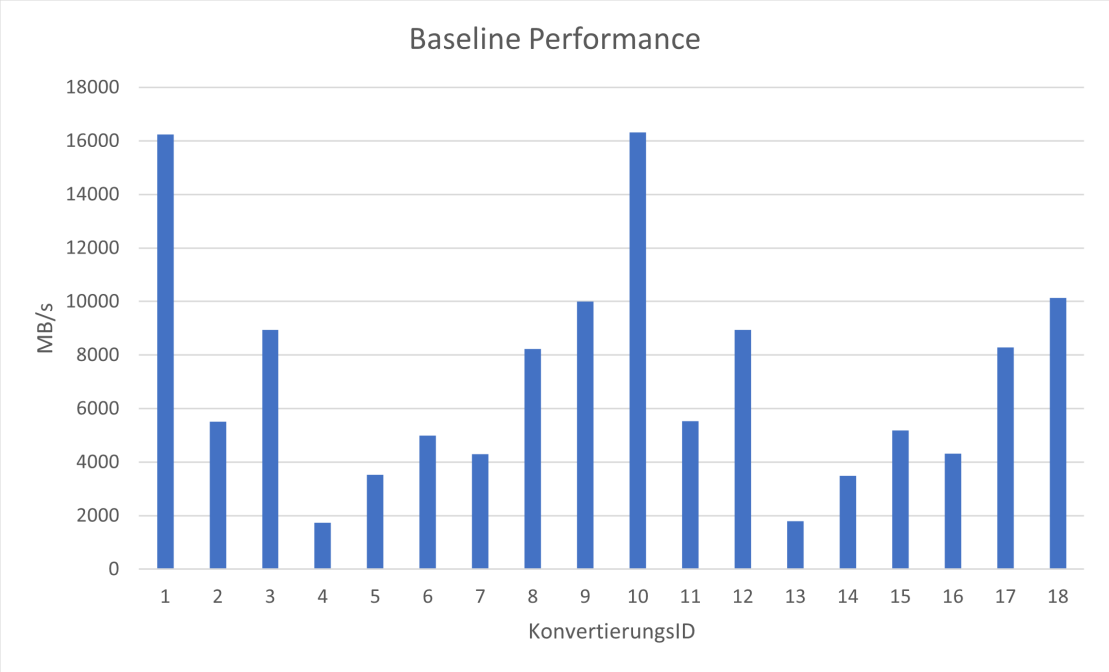
6.1 Baseline Messung

Es muss ein Performance-Ausgangspunkt gemessen werden, der die Performance des nicht-optimierten Codes repräsentiert. Diese sogenannte Baseline ist wichtig, um die Ergebnisse der verschiedenen Optimierungen qualifiziert bewerten zu können. Es wird die höchste Performance von drei Messdurchläufen in der folgenden Tabelle dokumentiert.

KonvertierungsID	Beste Performance aus drei Messungen
1	16250.3
2	5504.99
3	8943.58
4	1741.38
5	3529.75
6	4991.2
7	4304.88
8	8220.21
9	10007.8
10	16320.6
11	5536.93
12	8938.6
13	1796.82
14	3495.82
15	5193.09
16	4319.2
17	8283.71
18	10138

Tabelle 6.1: Baseline Messung: Werte in MB/s

Abbildung 6.1: Baseline Performance: Diagramm



Für die Konvertierungstabelle siehe 3.1. In dem Diagramm gibt es einige Auffälligkeiten. Die maximale Performance wird bei der Konvertierung von Mono8 zu Mono8, also beim Kopieren von Speicher mit memcpy(), erreicht. Die niedrigste Performance wird bei der Konvertierung von BayerRG12 zu Mono8 erreicht. Der Performance Unterschied zwischen den beiden Extremen ist ungefähr 14500 MB/s. Die KonvertierungsID 1 ist daher circa 9 Mal performanter. Auffällig ist, das die Bildgröße in der ursprünglichen Implementation einen linearen Einfluss auf die Performance hat. Also die Konvertierungszeit dauert proportional Länger wie die Bilder größer sind. Daher sind die Performance Werte der 5MP Bilder (Konvertierung 1 bis 9) fast deckungsgleich zu den Werten der 24,4MP Bilder (Konvertierung 10 bis 18).

6.2 These 1: Cacheline Alignment

'Das Cacheline Alignment der Bilder ergibt keine messbare Leistungssteigerung.'
Diese These ist für alle Konvertierungen von Relevanz und wird für alle Konvertierungen umgesetzt. Die Implementation ist dabei unabhängig von der Konvertierung, weil die Daten aligned werden und nichts an der konkreten Konvertierung verändert wird.

6.2.1 Umsetzung

Das Cachline-Alignment wird sowohl für die Input-Bilder als auch für die Output-Bilder umgesetzt, um den maximalen Effekt zu erzielen. Bei der Implementation werden die Bild-Speicher um 64 Byte größer gemacht, damit mit std::align die Startadresse des Bildes auf eine durch 64 teilbare Adresse verschoben werden kann. Dieser extra Aufwand durch das Alignment wird nicht in der Performance Messung vom Timer gemessen. Somit wird dieser sich in den Ergebnissen nicht negativ auswirken. Dies wird im folgenden Code Beispiel konkret dargestellt. Für mehr Code Kontext siehe A.6.

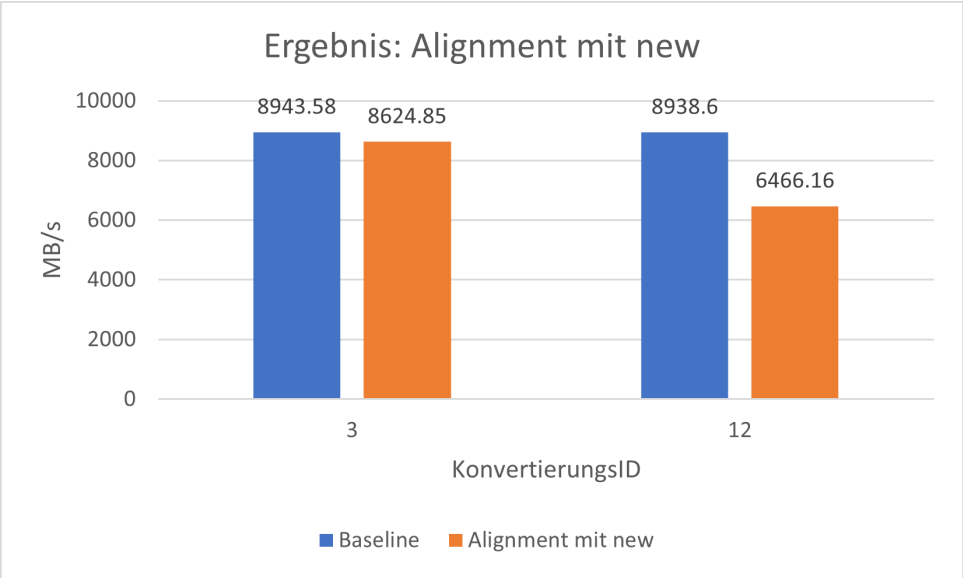
```
1 {...}
2 //create - output
3 for (size_t i = 0; i < cCountImages; ++i)
4 {
5     size_t bufferSizeIncPadding = constBufferSizeIncPaddingOut;
6     Pylon::CPylonImage& image = imagesOut[i];
7     void* var = new char[constBufferSizeIncPaddingOut];
8     std::align(64, bufferSizeOut, var, bufferSizeIncPadding);
```

```
9       image.AttachUserBuffer((char*)var, bufferSizeOut, PixelType_Mono8, cWidth
        , cHeight, 0);
10  }
11  {...}
```

Listing 6.1: C++ Code These 1 - Cacheline Alignment

Bei Messungen währen der Implementierungsphase ist ein unerwartetes Ergebnis herausgekommen. Dies wird exemplarisch anhand einer Konvertierung in folgender Grafik dargestellt.

Abbildung 6.2: Ergebnis: Cacheline Alignment mit new



Das Ergebnis bei der Konvertierung von Mono8 nach RGB16 mit 24,4MP Bildern, ist entgegen der Erwartung schlechter, als die Baseline Performance. Die Performance ist in diesem Fall von von 8938 MB/s auf 6466 MB/s gesunken. Dies entspricht einem Performanceverlust von circa 28Die folgenden zwei Zeilen Code ersetzen Zeile 7 in dem Code 'C++ Code These 1 - Cacheline Alignment', um diese Situation zu beheben.

```
1  buffersOut[i].resize(bufferSizeIncPadding);
2  void* var = buffersOut[i].data();
```

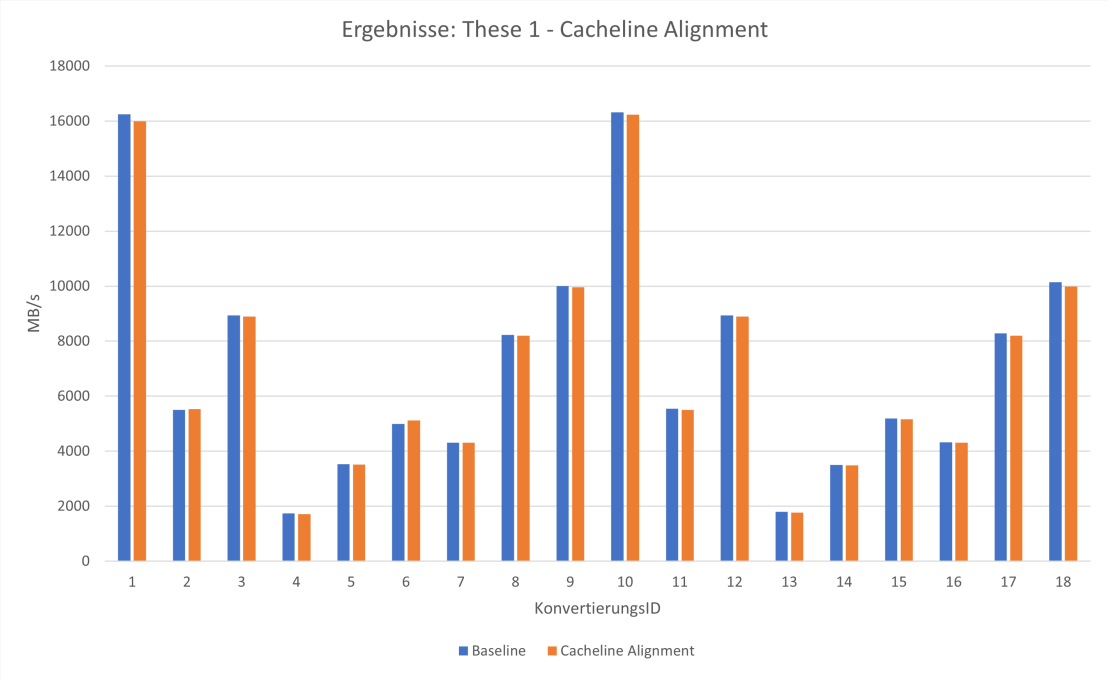
Listing 6.2: C++ Code These 1 - Cacheline Alignment - Implementationsfix

Das Erzeugen des Output-Buffers wird an Stelle von 'new char[]' mit einem 'std::vector' umgesetzt (siehe A.6. Diese Änderung führt dazu, dass die Performance Erwartungsgemäße Ergebnisse 6.3 liefert.

6.2.2 Ergebnis

Nach der Durchführung von den Performance Tests konnten folgende Ergebnisse gemessen werden.

Abbildung 6.3: Ergebnisse: These 1 - Cacheline Alignment



Die Grafik 6.3 zeigt die Messergebnisse nach der Implementation von der Optimierung aus These 1. Es ist zu sehen, dass die Performance nach dem Cacheline Alignment fast mit der Performance ohne Optimierungen übereinstimmt.

6.2.3 Diskussion

Die Diskussion der Ergebnisse dieser Optimierungsthese wird in zwei Abschnitte unterteilt. Zum einen wird auf das unerwartete zwischen Ergebnis eingegangen und genauer analysiert und anderen wird das endgültige Ergebnis diskutiert.

Zwischenergebnis bei der Umsetzung

Das Ergebnis gibt einen Anlass diesen Fall genauer zu untersuchen. Dafür wird der Profiler 5.3 mit der 'Assess Performance (extended)' Variante auf die entsprechende Konvertierung mit der 'new char[]' Code Zeile und der 'std::vector' Variante angewendet. Folgende Ergebnisse konnten gemessen werden.

Abbildung 6.4: Ergebnisse: Performance Profiling Analyse - KonvertierungsID 12 mit Cacheline Alignment 'new char[]'

Functions	Modules	CYCLES_NOT_IN_HALT ▼	RETIRED_INST	L1_DC_ACCESSES.ALL	L2_CACHE_ACCESS.FROM_L1_DC_MISS	CPI	IPC	L1_DC_ACCESS_RATE	L1_DC_MISS_RATE	L1_DC_MISS_RATIO
Pylon::CMonoConverter<class Pylon::MonoToColorLineCon	Project3_new.exe	237492	821680	405583	65956	0.29	3.46	0.49	0.01	0.02
main	Project3_new.exe	7220	39674	5778	533	0.18	5.50	0.15	0.00	0.01

Abbildung 6.5: Ergebnisse: Performance Profiling Analyse - KonvertierungsID 12 mit Cacheline Alignment 'std::vector'

Functions	Modules	CYCLES_NOT_IN_HALT ▼	RETIRED_INST	L1_DC_ACCESSES.ALL	L2_CACHE_ACCESS.FROM_L1_DC_MISS	CPI	IPC	L1_DC_ACCESS_RATE	L1_DC_MISS_RATE	L1_DC_MISS_RATIO
Pylon::CMonoConverter<class Pylon::MonoToColorLineCon	Project3_std_vec.exe	243717	830748	410221	74580	0.29	3.41	0.49	0.01	0.02
main	Project3_std_vec.exe	10573	42367	5555	553	0.25	4.01	0.13	0.00	0.01

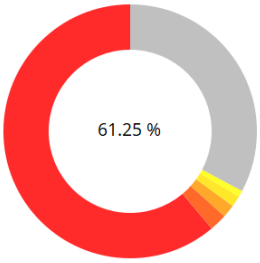
Die Grafiken zeigen aus dem Grund der Erkennbarkeit nur je einen Ausschnitt der Analyse. Es gibt keine Signifikanten Performance Unterschiede in der LineConverter::Convert Methode im Bezug auf diese Analyse und somit keinen Hinweis auf die Ursache für den Performance Unterschied. Außer der bereits beschrieben Zeilen Code sind die Konvertierungen identisch.
Nach weiterem Analysieren der Profiling Resultate ist Folgendes aufgefallen.

Abbildung 6.6: Summary: Performance Profiling Analyse - KonvertierungsID 12 mit Cacheline Alignment 'new char[]'

Profile Duration : 1m:18s

CYCLES_NOT_IN_HALT

Hot Functions



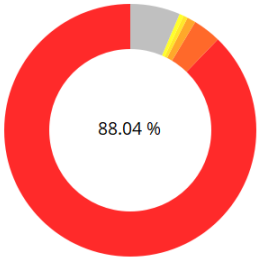
Function	CYCLES_NOT_IN_HALT [sample count]	CYCLES_NOT_IN_HALT [event count]	Module
Pylon::CMonoConverter::Conve ...>([, ...)	237492	59373000000	Project3_new.exe
ntoskrnl.exe!0xfffff8032529ec18	9148	2287000000	ntoskrnl.exe
main	7220	1805000000	Project3_new.exe
ntoskrnl.exe!0xfffff803253317e5	6341	1585250000	ntoskrnl.exe
ntoskrnl.exe!0xfffff803252a57dd	3861	965250000	ntoskrnl.exe
Others	123702	30925500000	N/A

Abbildung 6.7: Summary: Performance Profiling Analyse - KonvertierungsID 12 mit Cacheline Alignment 'std::vector'

Profile Duration : 53s

CYCLES_NOT_IN_HALT

Hot Functions



Function	CYCLES_NOT_IN_HALT [sample count]	CYCLES_NOT_IN_HALT [event count]	Module
Pylon::CMonoConverter::Conve ...>([, ...)	243717	60929250000	Project3_std_vec.exe
main	10573	2643250000	Project3_std_vec.exe
vcruntime140.dll!0x7ff92d231a77	3110	777500000	vcruntime140.dll
ntoskrnl.exe!0xfffff803252430d9	2292	573000000	ntoskrnl.exe
ntoskrnl.exe!0xfffff8032523efbc	1874	468500000	ntoskrnl.exe
Others	15256	3814000000	N/A

In diesen Grafiken ist zu sehen, dass die 'new char[]' Variante mit einer Laufzeit 1 Minute und 18 Sekunden länger braucht als die 'std::vector' Variante mit 53 Sekunden und somit auch beim Profiling einen messbaren Performance Unterschied aufzeigt. Der dunkel rote Anteil der Ringdiagramme und die Prozentzahl in der Mitte zeigen den Anteil der gesamten Zyklen, die die LineConverter::Convert Methode im Prozessor benötigt. Die 'new char[]' Variante weist einen wesentlich höheren grauen Anteil (als 'Others' benannt) auf. Unter den Aspekt 'Others' fasst der Profiler verschiedene Funktionen, die allein betrachtet nicht signifikant genug sind. Darunter können auch viele Kernel-Funktionen und mehr fallen. Dieser unterschiedlich große 'Others' Aspekt kann deshalb auf verschiedene Ursachen hindeuten.
Eine mögliche Erklärung wäre, dass das die Prozesse jeweils unterschiedlich viel Speicher in den jeweiligen so genannten 'working set' und 'virtual bytes' haben. Das Betriebs-

system muss den virtuellen Speicher eines Prozesses auf den physikalischen Speicher zuordnen, auch als 'paging' bekannt. Der Speicher ist in Speicherseiten(pages) je 4k Bytes organisiert. Dieser 'paging' Vorgang kann die Ursache sein. Es kann auch einen anderen Grund für den Performance Unterschied geben. Unabhängig von der genauen Ursache, wird die 'std::vector' Variante verwendet für die Umsetzung dieser These.[21]

Finale Ergebnisse

Die gemessenen Ergebnisse zeigen lediglich die Performance von der reinen Konvertierung, weil der extra Aufwand für das Cacheline Alignment nicht Teil der Messung ist und davor gemacht wird. Dies ist auch nicht weiter relevant, da es möglich wäre, dass die Bilder aus der Kamera bereits Cacheline-Aligned kommen könnten.
Es wird, wie bereits angenommen, keine Performance Steigerung gemessen. Es wird auf kontinuierlichen Speicher zugegriffen, der anschließend gecached werden kann.
Im Bezug auf das Problemstatement 3.6 und die Rahmenbedingungen 3.5 lässt sich schließen, dass eine Optimierung in Sinne eines Cacheline Alignments nicht sinnvoll ist und keine Option darstellt für die Basler AG.

6.3 These 2: Vektorisierung

'Durch das Vektorisieren des Codes, sofern möglich, kann die Leistung ein Vielfaches erreichen.'

Diese These ist für alle Konvertierungen relevant. Auf Grund der individuellen Implementierungen wird es keine allgemeine Umsetzung dieser These geben. Das bedeutet, dass die Ergebnisse nicht direkt untereinander vergleichbar sind. Das ist in Ordnung, da die Ergebnisse mit der Baseline verglichen werden und es nur darauf ankommt, ob die jeweilige Implementation der These 2 mehr Performance hat als die Baseline-Implementation und nicht, ob die Konvertierung x im Vergleich zu Konvertierung y irgendein Verhalten aufweist.
Der folgende Abschnitt ist von der Konvertierung von Mono8 zu Mono8, aber gilt identisch (Zeilen Angaben können abweichen) für alle Konvertierungen. Der Code wurde mit den Compiler Flags '/Qvec-report:2'[14] compiliert, um die Auto-Vektorisierung des Compilers zu überprüfen. Die Compiler Analyse ergab folgende Log-Ausgabe:

```
1>Conv1.cpp
1>--- Analyzing function: static void __cdecl Pylon::CMonoConverter<class
    Pylon::CopyLineConverterMono>::Convert<struct Pylon::SShiftValues<8,8,1>,
    class Pylon::CPlanarizerNoAction<struct Pylon::MonoPixel<unsigned char>
    >,class Pylon::CUnpackerNoAction<unsigned char,8>,class std::vector<
    unsigned char,class std::allocator<unsigned char> > >(class Pylon::
    CPlanarizerNoAction<struct Pylon::MonoPixel<unsigned char> > & __ptr64,
    class Pylon::CUnpackerNoAction<unsigned char,8> & __ptr64,class std::
    vector<unsigned char,class std::allocator<unsigned char> > const &
    __ptr64)
1>E:\projekte\visualstudio2019\BaCodeGit\Conv1.cpp(225) : info C5002: loop
    not vectorized due to reason '500'
1>E:\projekte\visualstudio2019\BaCodeGit\Conv1.cpp(401) : info C5002: loop
    not vectorized due to reason '1200'
```

Listing 6.3: Auto-Vektorisierungsanalyse des Compilers

Den Code mit den Compiler Flags '/Qvec-report:1' zu bauen erzeugt Log-Ausgaben, über alle Auto-Vektorisierung, die der Compiler gemacht hat. Dies ist in diesem Fall keine. Dieses Ergebnis zeigt, dass der Compiler keine der For-Schleifen vektorisiert hat und es die Möglichkeit bietet für eine Optimierung.
Der Compiler hat bei keiner der Konvertierungen eine Auto-Vektorisierung unternommen.

6.3.1 Umsetzung

Die Log-Ausgaben geben jeweils einen Code[15] an, weshalb eine Vektorisierung nicht möglich war. Es ist nicht möglich den Code perse so zu ändern, dass der Compiler eine Auto-Vektorisierung machen kann. Die interessante Code Stelle zum Optimieren ist in Zeile 401, die LineConverter::Convert Methode. Dies ist die Funktion, welche im Call-Stack ganz unten steht und somit am häufigsten aufgerufen wird beim Konvertieren der Pixelformate. Deshalb ist an der Stelle der Performance Gewinn potentiell am höchsten. Es ist nicht möglich im Rahmen dieser Ausarbeitung für jede Konvertierung eine vektorisierte Variante zu implementieren. Die Konvertierungen mit dem BayerRG12 Input-Pixelformat sind zu komplex für eine Vektorisierung. Aus diesem Grund werden diese Konvertierungen in Rahmen dieser These nicht weiter thematisiert. Es wird für die in Frage kommenden Konvertierungsarten eine individuelle Vektorisierung auf LineConverter Ebene umgesetzt. Die Details werden im Folgenden in die Abschnitte 'Eigene Vektorisierung' sowie 'Intel IPP' unterteilt und beschrieben.

Es ist zu erwarten, dass die Intel IPP Funktionen teilweise eindeutig mehr Performance haben, als die 'eigene Vektorisierung'.

Eigene Vektorisierung

Eine Möglichkeit der Vektorisierung ist eine Funktion durch die Verwendung von Intrinsics, den SIMD-Befehlen, eigenständig zu Vektorisieren. Dabei ist es dem Entwickler überlassen, welchen Befehlssatz dieser verwenden möchte. Der Entwickler kann durch sein Verständnis des Codes die Vektorisierung möglichst gut umsetzen. Es wird für vektorisierten Konvertierungen angenommen, dass es ausschließlich die zwei definierten Bildgrößen gibt. Wenn mit variierenden Bildgrößen zu rechnen ist, muss lediglich eine Sonderbehandlung für eine kleine Menge an Pixeln implementiert werden. Diese Sonderbehandlung muss dafür sorgen, dass die übrige Menge an Pixeln ein Vielfaches ist von der Anzahl an Pixeln, die pro Schleifendurchlauf konvertiert werden.
Eine Vektorisierung wird anhand der Konvertierung von Mono8 zu RGB8 beispielhaft dargestellt. Für die restlichen Implementationen siehe Anhang A.2.

```
1  //Vektorisierte Variante
2  __m128i vInput;
3
4  __m128i vMask0 = _mm_set_epi8(0x05, 0x04, 0x04, 0x04, 0x03, 0x03, 0x03, 0x02,
       0x02, 0x02, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00);
5  __m128i vMask1 = _mm_set_epi8(0x0a, 0x0a, 0x09, 0x09, 0x09, 0x08, 0x08, 0x08,
       0x07, 0x07, 0x07, 0x06, 0x06, 0x06, 0x05, 0x05);
6  __m128i vMask2 = _mm_set_epi8(0x0f, 0x0f, 0x0f, 0x0e, 0x0e, 0x0e, 0x0d, 0x0d,
       0x0d, 0x0c, 0x0c, 0x0c, 0x0b, 0x0b, 0x0b, 0x0a);
7
8  __m128i vTemp0, __m128i vTemp1, __m128i vTemp2;
9  for (; pIn < pInEnd; pIn+=16, pOut+=16)
10 {
11     vInput = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pIn));
12     vTemp0 = _mm_shuffle_epi8(vInput, vMask0);
13     vTemp1 = _mm_shuffle_epi8(vInput, vMask1);
14     vTemp2 = _mm_shuffle_epi8(vInput, vMask2);
15     _mm_store_si128((reinterpret_cast<__m128i*>(pOut)), vTemp0);
16     _mm_store_si128((reinterpret_cast<__m128i*>(pOut)) + 1, vTemp1);
17     _mm_store_si128((reinterpret_cast<__m128i*>(pOut)) + 2, vTemp2);
18 }
19 //Vektorisierte Variante Ende
20 //Originaler Code
21 for (; pIn < pInEnd; ++pIn, ++pOut)
```

```
22 {
23     typename OutT::value_type const shifted = ShiftChannel<ShiftValuesT, InT,
        typename OutT::value_type>( *pIn );
24     new (pOut) OutT( shifted, shifted, shifted );
25 }
26 //Originaler Code Ende
```

Listing 6.4: Eigene Vektorisierung: Mono8 zu RGB8

In dem Code Ausschnitt ist der vektorisierte Code(oberer Teil) und der originale Code(unterer Teil) aus der Methode 'MonoToColorLineConverter::Convert()' zu sehen. Der verwendete SIMD-Befehlssatz für diese Konvertierung ist der 'SSE3-Befehlssatz'. Es erstes werden die Vektoren definiert, welche die Maske für das Kopieren der Mono8 Werte an die richtige Stelle des RGB8 Output-Bildes bestimmen. Es werden Vektoren für die Zwischenergebnisse deklariert. Ein '__m128i'-Vektor bietet Platz für 16 Byte. Pro Schleifen Iteration werden 16 Byte gelesen und konvertiert. Das entspricht 16 Mono8 Pixeln. Bei dem Output-Pixelformat RGB8 müssen also alle Input-Werte verdreifacht werden. Aus diesem Grund werden nach dem korrekten Vervielfachen der Werte drei Vektor-Store Befehle ausgeführt. Diese Speichern den Inhalt eines Vektors an die entsprechende Stelle im Code.
Der Ablauf ist bei allen eigenen Vektorisierungen ähnlich. Zu erst werden Vektor-Masken definiert und die Vektoren zum Zwischenspeichern der Vektor-Operationensergebnisse, die zu konvertierenden Pixelwerte werden in Vektorregister geladen und die geladen Pixel werden gleichzeitig mit den entsprechenden Befehlen konvertiert. Das wird in einer for-Schleife zeilenweise ausgeführt. Dabei wird der Schleifenindex um die gleiche Anzahl inkrementiert, wie viele Pixel in einer Schleifeniteration konvertiert werden.

Intel IPP

Die Intel IPP Bibliothek bietet viele verschiedene vektorisierte Funktionen unter anderem im Bereich der Bildverarbeitung an. Darunter sind Funktionen zum Konvertieren von Pixelformaten. Es ist teilweise eingeschränkt in der Funktionalität. So kann die Konvertierung von RGB8planar Bildern nicht durchgeführt werden. Es bleibt das Konvertieren der Mono8 Input-Bilder in die drei Output-Pixelformate. Dies ist ausreichend, um eine grobe Bewertung machen zu können.
Die konkrete Verwendung wird im folgenden Code Ausschnitt gezeigt. Der Code wird ein Mal pro Konvertierung eines gesamten Bildes aufgerufen.

```
//Originaler Code
Convert(
    imageOut.GetBuffer(), // void* pDest,
    imageOut.GetImageSize(),// size_t sizeDest,
    imageIn.GetBuffer(), // const void* pSource,
    imageIn.GetPixelType(),// EPixelType pixelTypeSource,
    imageIn.GetWidth(),// uint32_t width,
    imageIn.GetWidth(),// uint32_t widthClipped,
    imageIn.GetHeight(),// uint32_t height,
    imageIn.GetHeight(),// uint32_t heightClipped,
    imageIn.GetPaddingX(),// size_t paddingXSource,
    imageOut.GetPixelType(),// EPixelType pixelTypeDestination,
    imageOut.GetPaddingX(),// size_t paddingXDestination,
    Pylon::InconvertibleEdgeHandling_Clip,// EInconvertibleEdgeHandling
    edgeHandleMethod,
    false,// bool flipY,
    lut8// const std::vector<uint8_t> & lut8
);
//Originaler Code Ende
//Mono8 zu Mono8
ippiCopy_8u_C1R((const Ipp8u*)imageIn.GetBuffer(), static_cast<int>(imageIn.
    GetWidth()), (Ipp8u*)imageOut.GetBuffer(), static_cast<int>(imageOut.
    GetWidth()), { static_cast<int>(imageIn.GetWidth()), static_cast<int>(
    imageIn.GetHeight()) });
//Mono8 zu Mono8 Ende
//Mono8 zu RGB8
ippiGrayToRGB_8u_C1C3R((const Ipp8u*)imageIn.GetBuffer(), static_cast<int>(
    imageIn.GetWidth()), (Ipp8u*)imageOut.GetBuffer(), static_cast<int>(
    imageOut.GetWidth()*3), { static_cast<int>(imageIn.GetWidth()),
    static_cast<int>(imageIn.GetHeight()) });
//Mono8 zu RGB8 Ende
//Mono8 zu RGB16 // tempBufSize ist definiert in Abhaengigkeit zu der image
    size
std::vector<Ipp16u> tempBuf(tempBufSize);
auto v = ippiConvert_8u16u_C1R((const Ipp8u*)imageIn.GetBuffer(), static_cast
    <int>(imageIn.GetWidth()),(Ipp16u*)tempBuf.data(), static_cast<int>(
    imageIn.GetWidth()*2), { static_cast<int>(imageIn.GetWidth()),
    static_cast<int>(imageIn.GetHeight()) });
ippiLShiftC_16u_C1IR(8, (Ipp16u*)tempBuf.data(), static_cast<int>(imageIn.
    GetWidth()*2), { static_cast<int>(imageIn.GetWidth()), static_cast<int>(
    imageIn.GetHeight()) });
auto w = ippiGrayToRGB_16u_C1C3R((const Ipp16u*)tempBuf.data(), static_cast<
    int>(imageIn.GetWidth()*2), (Ipp16u*)imageOut.GetBuffer(), static_cast<
```

```
    int>(imageOut.GetWidth()*6), { static_cast<int>(imageIn.GetWidth()),
    static_cast<int>(imageIn.GetHeight()) });
//Mono8 zu RGB16 Ende
```

Listing 6.5: Intel IPP Vektorisierung

Es ist sehr einfach mit Intel IPP Befehlen die Konvertierung von Mono8 zu Mono8 und von Mono8 zu RGB8 umzusetzen. Das Konvertieren von Mono8 zu RGB16 bedarf ein wenig extra Aufwand. Dafür muss zunächst das Input-Bild zu Mono16 konvertiert werden, damit es anschließend zu RGB16 konvertiert werden kann.

6.3.2 Ergebnis

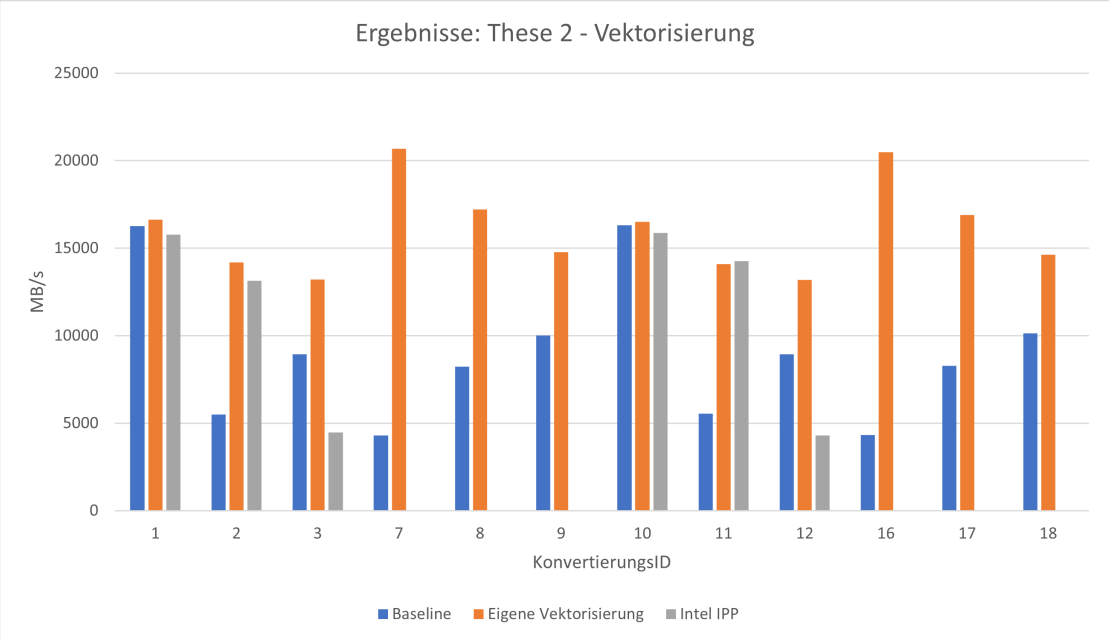
Nach der Durchführung von den Performance Tests konnten folgende Ergebnisse gemessen werden.

KonvertierungsID	Baseline	Eigene Vektorisierung	Intel IPP
1	16250.3	16624.9	15769.8
2	5504.99	14193.4	13127
3	8943.58	13219.3	4469.51
7	4304.88	20670.9	
8	8220.21	17205.5	
9	10007.8	14770.7	
10	16320.6	16514.4	15861.4
11	5536.93	14084.9	14261.9
12	8938.6	13194.3	4289.69
16	4319.2	20482.3	
17	8283.71	16902.9	
18	10138	14633.1	

Tabelle 6.2: Ergebnis-Tabelle: These 2; Werte in MB/s

Die Werte aus der obigen Tabelle sind in dem folgenden Diagramm visualisiert.

Abbildung 6.8: Ergebnisse: These 2 - Vektorisierung



Es ist zu erkennen, dass die Konvertierung von Mono8 zu Mono8(KonvertierugnsID 1 und 10), also das ausschließliche Kopieren von Speicher, vernachlässigbar geringe Unterschiede zum originalen Code aufzeigt. Die Konvertierung von Mono8 zu RGB8(KonvertierugnsID 2 und 11) zeigt, dass die 'eigene Vektorisierung' und die 'Intel IPP' Implementation wesentliche Performance Verbesserungen erbringen. In diesem Fall ist die Performance um mindestens einen Faktor von circa 230% gestiegen. Wird die Konvertierung von Mono8 zu RGB16(KonvertierugnsID 3 und 12) betrachtet, kann festgestellt werden, dass die 'eigene Vektorisierung' ungefähr 47% mehr Performance erbringt und 'Intel IPP' die Performance circa halbiert. Die höchste Performance Steigerung erbringt die 'eigene Vektorisierung' der Konvertierung von RGB8planar zu Mono8 mit dem mehr als dem 4,5 Fachen der Baseline Performance. Die Baseline Performance Steigt mit der größer werdenden Menge(KonvertierugnsID 7 - 9 und 16 - 18) an Bilddaten und die Performance der 'eigenen Vektorisierung' nimmt ab. Dennoch ist die 'eigene Vektorisierung' in jeder RGB8planar(KonvertierugnsID 7 - 9 und 16 - 18) Konvertierung besser als die Baseline. Die Ergebnisse zeigen gleiche Werte für die beiden Bildgrößen auf.

6.3.3 Diskussion

In dieser Diskussion der Ergebnisse der zweiten Optimierungsthese wird als erstes die Vektorisierung der Konvertierung von Mono8 zu RGB8 genauer Analysiert. Als letztes werden die Ergebnisse beurteilt.

Konvertierung von Mono8 zu RGB8

Die Performance Verbesserung der 'eigenen Vektorisierung' und der Vektorisierung mit Intel IPP bei der Konvertierung von Mono8 zu RGB8 sind sehr ähnlich. Damit ist dieser Fall ideal für eine detailliertere Analyse und einen Vergleich der beiden Implementationen. Dafür wird mit dem AMD uProf Profiler der Code analysiert. Da mit der Intel oneAPI keine Debug Dateien mitgekommen sind, ist es nicht möglich den erzeugten Assembler Code zu vergleichen.
Die Resultate der Performance Analysen sind in den zwei folgenden Grafiken zu sehen.

Abbildung 6.9: Performance Profiling Analyse: Konvertierung Mono8 zu RGB8 - 'eigene Vektorisierung'

Functions	Modules	CYCLES_NOT_IN_HALT	RETIRED_INST	DATA_CACHE_ACCESSES_PTI	%DATA_CACHE_MISS	MISALIGN_LOADS_PTI	IPC	RETIRED_BRANCH_INSTRUCTION_MISPREDICTED_PTI	%RETIRED_BRANCH_INSTRUCTION_MISPREDICTED	MAB_ALLOC_PTI
Pylon::CMonoConverter<class Pylon::MonoToC	vec.exe	77902	79673	319.14	16.10		1.02	0.30	0.38	51.38
main	vec.exe	10971	42763	127.91	1.00		3.90	0.13	0.05	1.28

Abbildung 6.10: Performance Profiling Analyse: Konvertierung Mono8 zu RGB8 - 'Intel IPP'

Functions	Modules	CYCLES_NOT_IN_HALT	RETIRED_INST	DATA_CACHE_ACCESSES_PTI	%DATA_CACHE_MISS	MISALIGN_LO	IPC	RETIRED_BRANCH_INSTRUCTION_MISPREDICTED_PTI	%RETIRED_BRANCH	MAB_ALLOC_PTI
ippccl9.dll!0x7ff8987d500c	ippccl9.dll	31879	48821	326.50	10.60		1.53	0.22	0.21	34.61
ippccl9.dll!0x7ff8987d5004	ippccl9.dll	20993	32807	304.72	11.03		1.56	0.23	0.24	33.62
main	ipp.exe	11125	42480	130.44	1.02		3.82	0.12	0.04	1.33
ippccl9.dll!0x7ff8987d501c	ippccl9.dll	10748	15717	308.65	10.92		1.46	0.31	0.30	33.69
ippccl9.dll!0x7ff8987d5011	ippccl9.dll	6641	9464	313.72	10.72		1.43	0.16	0.15	33.64
ippccl9.dll!0x7ff8987d5016	ippccl9.dll	3991	5785	298.36	10.79		1.45	0.12	0.12	32.19

Es ist sehr schwierig die Intel IPP Implementation zu Analysieren, denn es ist mit AMD uProf nicht ersichtlich, welche Methoden in der Übersicht angezeigt werden. Die blau markierten Zeilen in den beiden Grafiken zeigen den wahrscheinlich relevanten Anteil der Konvertierung. Es lassen sich nur grobe Aussagen treffen. Der IPC-Wert(Instructions per Cycle) ist das Gleiche wie 1 geteilt durch den CPI-Wert. Bei der Intel IPP Variante werden ungefähr 1,5 Befehle pro Zyklus ausgeführt und bei der 'eigenen Vektorisierung' wird ein Befehl pro Zyklus ausgeführt. Generell ist ein höherer IPC-Wert besser für die Leistung.

Beurteilung der Ergebnisse

Insgesamt ist die Vektorisierung eine effektive Optimierungsmethode. Dies ist an zwei offensichtlichen Ergebnissen zu erkennen. Zum einen konnte in einem Fall ein maximaler Performancegewinn von mehr als 4,5 Mal so viel, wie in dem originalen Code. Zum andern ist in allen Fällen eine signifikante Leistungssteigerung erkennbar, mit Ausnahme von der Konvertierung von Mono8 zu Mono8. Diese ist im Grunde einfaches memcpy(), welches bereits in der unoptimierten Variante sehr effizient vom Prozessor ausgeführt werden kann.
Die 'eigene Vektorisierung' der RGB8planar Konvertierungen zeigen vermeintlich einen Trend auf. Das ist nicht unbedingt der Fall, da die Implementation direkt von den Pixelformaten abhängig ist. Daher sind diese als individuelle Ergebnisse zu betrachten.
Besonders auffällig sind die Ergebnisse der Konvertierung 3/12 (Mono8 zu RGB8). Die 'eigene Vektorisierung' zeigt wie zu erwarten ist eine Verbesserung der Performance. Die Intel IPP Variante hingegen zeigt eine Verschlechterung der Leistung. Dies kann an der Art und Weise liegen, wie die Konvertierung mit den Intel IPP Funktionen implementiert ist. Da eine Konvertierung der Pixelformate und einer Erweiterung der Pixel-Bit-Tiefe nicht gleichzeitig möglich ist mit Intel IPP führt der extra zwischen Schritt zu mehr Leistungskosten, als es an Leistugnsgewinn bringt.
Im Bezug auf das Problemstatement 3.6 erzielt diese Optimierungsmethode die erforderliche Leistungssteigerung von mindestens einem Faktor 2. Auch wenn nicht jedes Ergebnis diesem Anspruch entspricht, so ist diese prototypische Umsetzung Beweis für das Potential dieser Optimierungsmethode.
Die Rahmenbedingungen miteinbezogen, so lässt sich sagen, dass die Intel IPP Bibliothek und die 'eigene Vektorisierung' beide für Windows, Linux und macOS verwendbar sind. Die Intel IPP Funktionen sind einfach zu verwenden und machen Vektorisierung einfach ohne das nötige Hintergrundwissen besitzen zu müssen. Auf der anderen Seite besteht eine Abhängigkeit zu den verfügbaren Funktionen. Spätestens, wenn eine spezielle Konvertierung nicht von Intel IPP konvertiert werden kann, kann dies erneut zu Performance Bottlenecks führen. Damit ist es nur bedingt für die Basler AG geeignet.
Die 'eigene Vektorisierung' setzt voraus, dass der Entwickler Ahnung von der Materie mitbringt. Außerdem, muss für jede Konvertierung eine individuelle vektorisierte Konvertierung programmiert werden. Dies ist bei den vielen unterstützten Input- und Output-Pixelformaten eine sehr zeitaufwändige Angelegenheit. Anderer Seits bietet diese Methode viele Möglichkeiten das Maximum an Performance herauszuholen. Alles in allem sind beide Varianten nicht ideal. Da die Performance der Pixelformatkonvertierung

eine sehr hohe Priorität hat, ist die 'eigene Vektorisierung' die bessere Variante und die empfohlene Optimierungsmethode.

6.4 These 3: Parallelisierung

'Eine Parallelisierung der Konvertierung bringt eine Performance Steigerung bei großen Bilddaten und erzeugt extra Aufwand, der bei kleinen Bilddaten zu einer Performance Verschlechterung führt.'
Diese These ist für alle Konvertierungen relevant und wird für alle Konvertierungen umgesetzt. Die Implementation ist dabei unabhängig von der Konvertierung, weil die konkrete implementation der einzelnen Zeilen-Konvertierungen nicht verändert wird.
Der Code wurde mit den Compiler Flags '/Qpar /Qpar-report:2'[14] compiliert, um die Auto-Parallelisierung des Compilers zu überprüfen. Die Log-Ausgabe ergab folgende Analyse:

```
1 1>Conv1.cpp
2 1>--- Analyzing function: void __cdecl Pylon::CPlanarizerNoAction<struct
    Pylon::MonoPixel<unsigned char> >::ConversionDone(void) __ptr64
3 1>E:\projekte\visualstudio2019\BaCodeGit\Conv1.cpp(225) : info C5012: loop
    not parallelized due to reason '500'
4 1>--- Analyzing function: static void __cdecl Pylon::CMonoConverter<class
    Pylon::CopyLineConverterMono>::Convert<struct Pylon::SShiftValues<8,8,1>,
    class Pylon::CPlanarizerNoAction<struct Pylon::MonoPixel<unsigned char>
    >,class Pylon::CUnpackerNoAction<unsigned char,8>,class std::vector<
    unsigned char,class std::allocator<unsigned char> > >(class Pylon::
    CPlanarizerNoAction<struct Pylon::MonoPixel<unsigned char> > & __ptr64,
    class Pylon::CUnpackerNoAction<unsigned char,8> & __ptr64,class std::
    vector<unsigned char,class std::allocator<unsigned char> > const &
    __ptr64)
5 1>E:\projekte\visualstudio2019\BaCodeGit\Conv1.cpp(225) : info C5012: loop
    not parallelized due to reason '500'
6 1>E:\projekte\visualstudio2019\BaCodeGit\Conv1.cpp(401) : info C5012: loop
    not parallelized due to reason '1007'
```

Listing 6.6: Auto-Parallelisierungsanalyse des Compilers

Den Code mit den Compiler Flags '/Qpar /Qpar-report:1' zu bauen erzeugt Log-Ausgaben, über alle Auto-Parallelisierung, die der Compiler gemacht hat. Dies ist in diesem Fall keine.

Dieses Ergebnis zeigt, dass der Compiler keinen Code parallelisiert hat und es die Möglichkeit bietet für eine Optimierung.

6.4.1 Umsetzung

Für die Umsetzung der Parallelität werden eine OpenMP und eine std::thread Variante implementiert. Die beste Performance wird bei 16 Threads erwartet, da die zu Grunde liegende Hardware maximal 16 Threads gleichzeitig unterstützt. Potentiell liegt die Performance Steigerung bei einem Faktor von 16.

OpenMP

OpenMP ist ein Framework, welches die Parallelisierung übernimmt. Es bietet die Möglichkeit per '#pragma' for-Schleifen zu parallelisieren. OpenMP lässt zwar zu, dass die Anzahl der verwendeten Threads manuell eingestellt werden kann, aber OpenMP trifft die Entscheidungen ob parallelisiert wird und wie viele Threads verwendet werden zur Ausführungszeit. Dabei kommen mehrere Faktoren zum Tragen. [18, p. 78] Durch Testen konnte festgestellt werde, dass im Rahmen dieser Ausarbeitung das Festlegen der Threadanzahl problemlos funktioniert. Daher werden 2, 4, 8, 16 und 32 Threads auf Performance getestet.
Es wird für jede Konvertierung die for-Schleife, welche die LineConverter::Convert Methode aufruft mit dem pragma '#pragma omp parallel for shared(unpacker, planarizer)' parallelisiert.

std::thread

Es wird jedes Input-Bild in x (Anzahl) horizontale Streifen unterteilt. Es wird mit 2, 4, 8, 16 und 32 Streifen (Threads) getestet. Durch die Unterteilung in Streifen könnte sich die Anzahl an Context-Switches reduzieren, da ein Thread mit einer größeren Anzahl an Daten auf einmal beschäftigt ist. Für jeden Streifen wird dann in einem eigenen std::thread die Convert Methode aufgerufen. Nachdem alle Threads sind und alle Streifen konvertiert werden, werden diese wieder ge-joined. Es wird nicht das nächste Bild angefangen zu konvertieren, bevor nicht die bereits laufende Konvertierung fertig ist.

6.4.2 Ergebnis

Im folgenden Abschnitt werden alle Ergebnisse dieser These angezeigt. Anschließend werden die Ergebnisse zusammengefasst beschrieben.

Abbildung 6.11: Ergebnisse: These 3 - Mono8 zu Mono8'

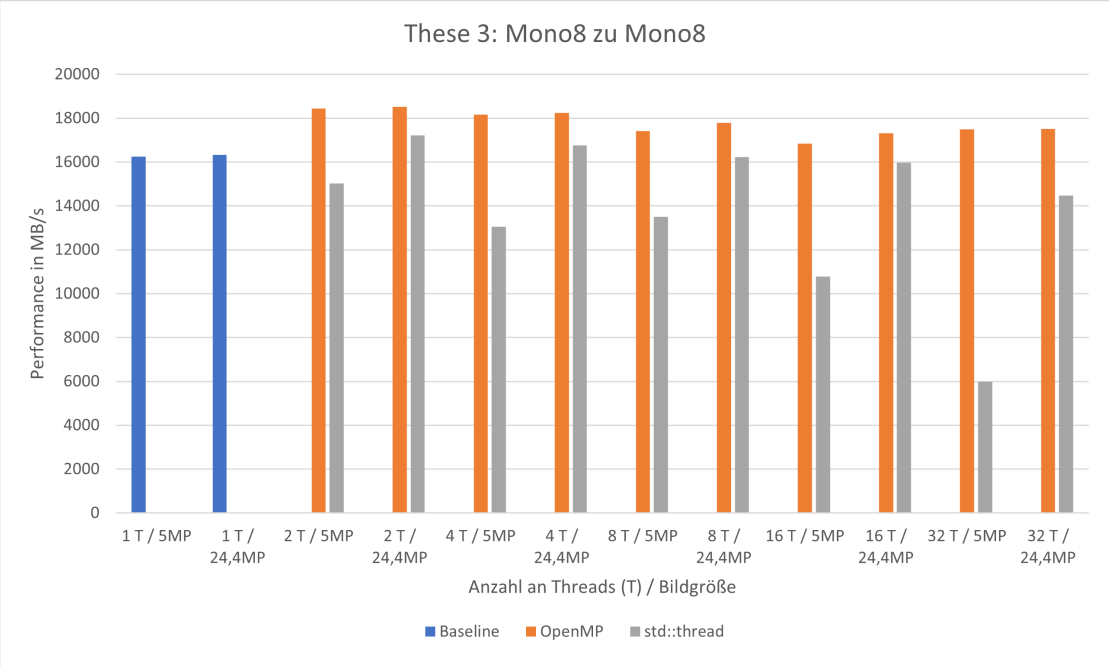


Abbildung 6.12: Ergebnisse: These 3 - Mono8 zu RGB8'

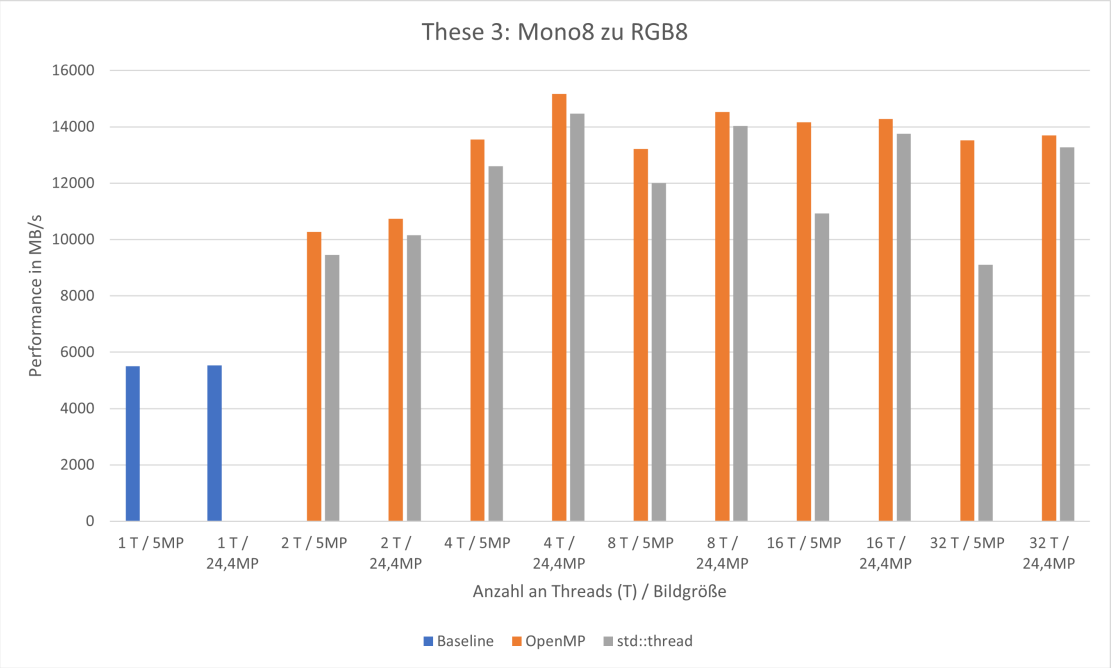


Abbildung 6.13: Ergebnisse: These 3 - Mono8 zu RGB16'

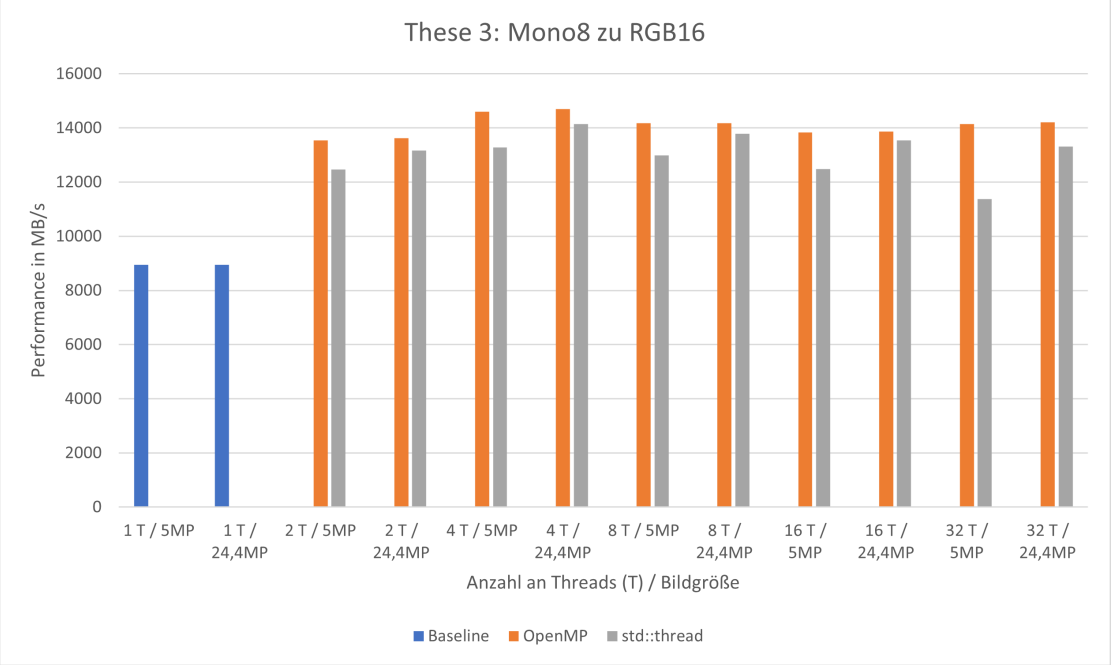


Abbildung 6.14: Ergebnisse: These 3 - BayerRG12 zu Mono8'

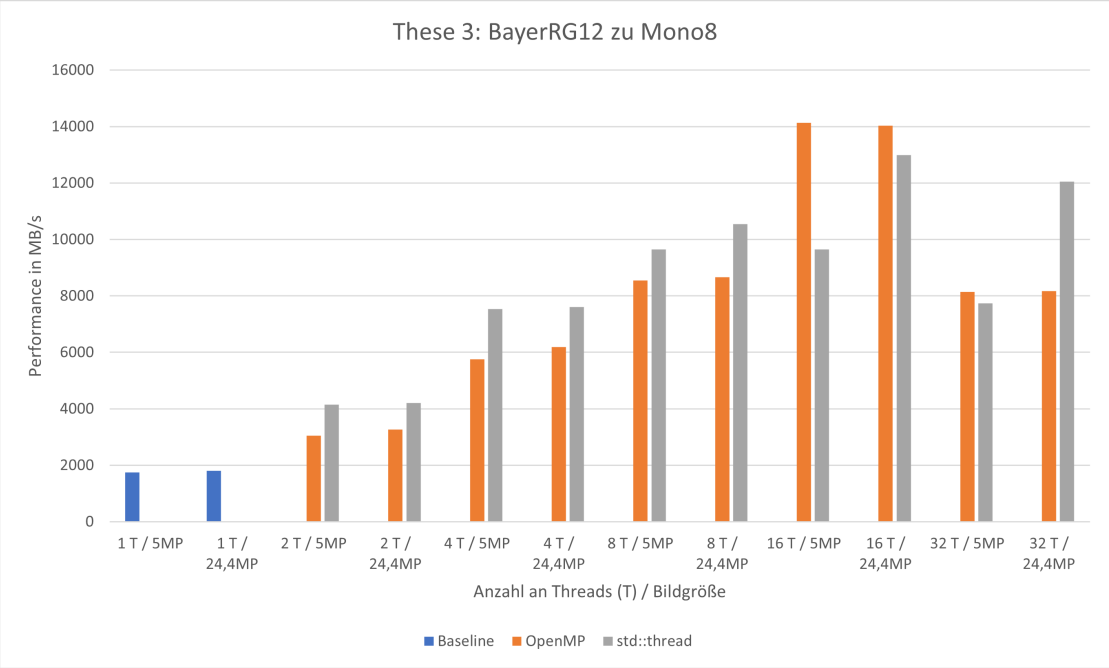


Abbildung 6.15: Ergebnisse: These 3 - BayerRG12 zu RGB8'

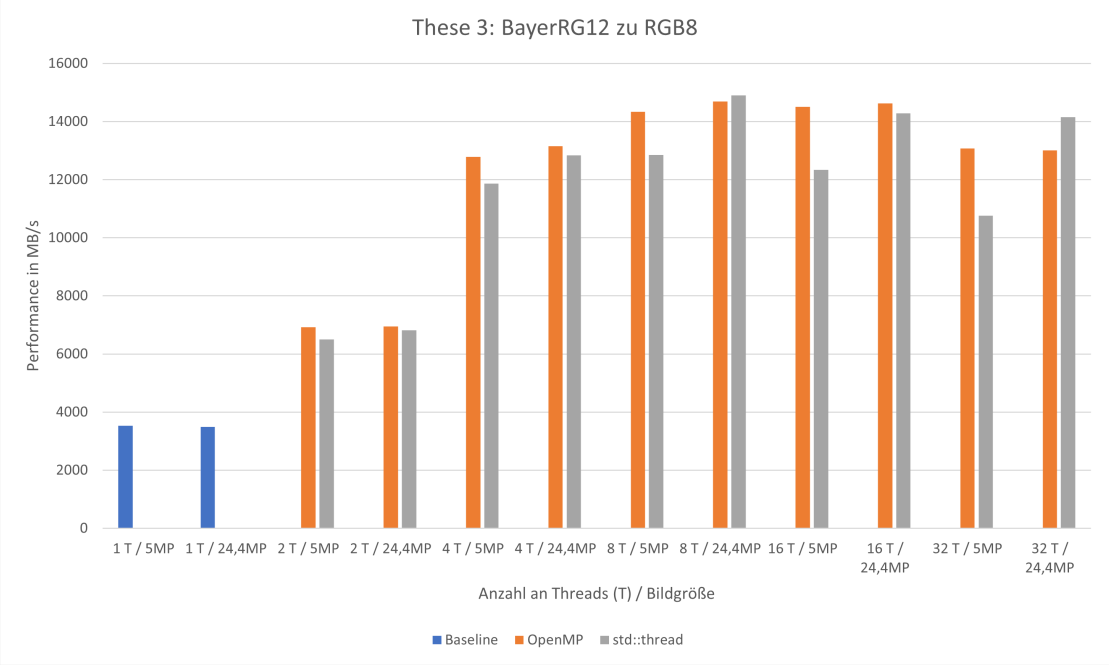


Abbildung 6.16: Ergebnisse: These 3 - BayerRG12 zu RGB16'

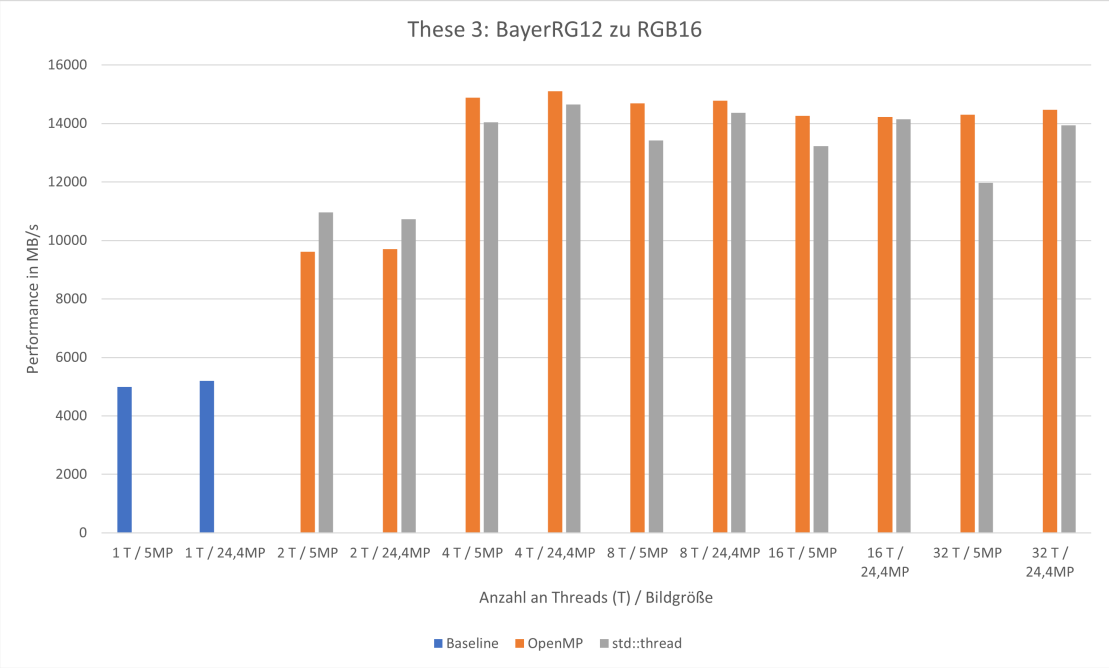


Abbildung 6.17: Ergebnisse: These 3 - RGB8planar zu Mono8'

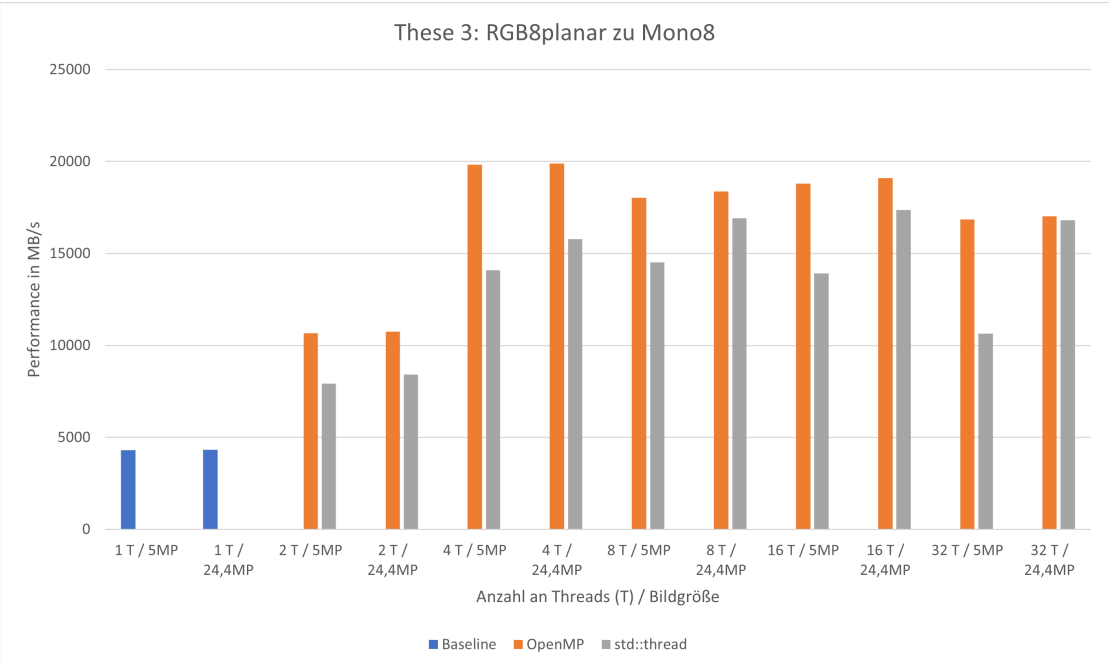


Abbildung 6.18: Ergebnisse: These 3 - RGB8planar zu RGB8'

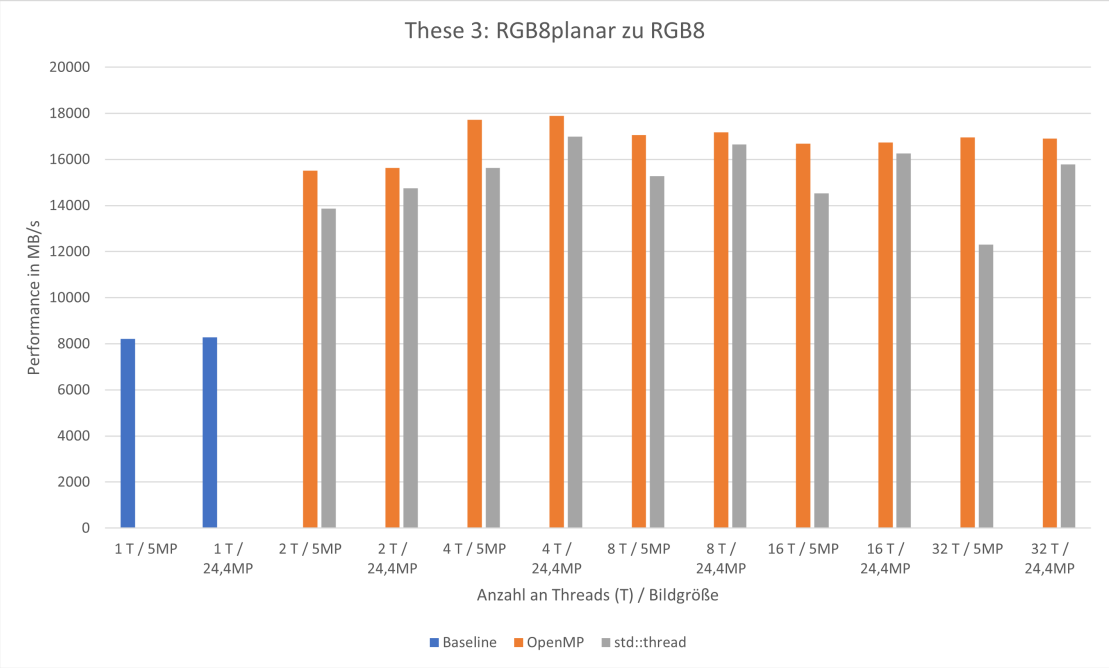
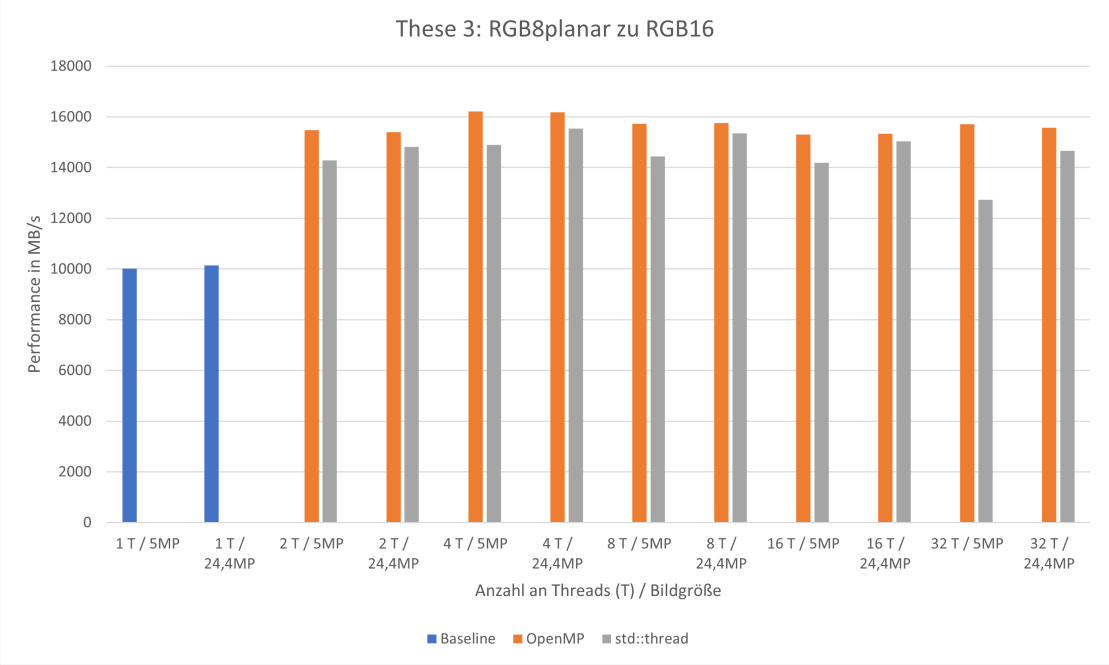


Abbildung 6.19: Ergebnisse: These 3 - RGB8planar zu RGB16'



Für eine Übersicht der konkreten Messwerte siehe A.3.

Zusammenfassung der Messergebnisse

Die Ergebnisse der Konvertierung Mono8 zu Mono8 stechen besonders heraus, weil diese als einziges Ergebnisse zeigt, welche eine niedrigere Performance haben, als die Baseline Performance. Außerdem ist ein starker Abwärtstrend zu erkennen bei den 5MP Bilder und der std::thread Implementation. Bei std::thread und 24,4MP Bildern ist dieser Abwärtstrend auch sichtbar, aber wesentlich flacher. Generell ist bei dieser Konvertierung selbst der Performance Gewinn mit OpenMP eher niedrig.
In der Grafik der Konvertierung von Mono8 zu RGB8 ist ein Performance Maximum bei 4 Threads zu erkennen. Danach fällt die Leistung Stück weise ab.
Die Ergebnisse der BayerRG12 zu Mono8 Konvertierung zeigen einen Performance Höchstwert bei 16 Threads für OpenMP. Die std::thread Variante hat ein Maximum bei 8 Threads bei 5MP Bildern und ein Maximum bei 16 Threads bei 24,4MP Bildern
Die, im Rahmen dieser Optimierungsthese, höchste gemessene Performance liegt bei 19900 MB/s und wird von der Konvertierung 'RGB8planar zu Mono8, 24,4MP, 4 Threads' erzielt.

6.4.3 Diskussion

Die bereits hohe Performance des original Codes in der Konvertierung von Mono8 zu Mono8 setzt einen hohen Maßstab und lässt nicht viel Raum für Leistungssteigerung. Die anderen Konvertierungen hingegen konnten eine Performancesteigerung von bis zu dem 8 fachen (BayerRG12 zu Mono8) der Baseline Performance erreichen.
Die Verwendung von std::thread mit 5MP Bildern zeigt in den meisten Fällen, dass die Performance stark abnimmt, nachdem das Performance Optimum erreicht wurde. Dieser Effekt tritt deutlich schwächer bei den Konvertierungen mit 24,4MP Bildern auf, mit Ausnahme von der Konvertierung BayerRG12 zu Mono8. Da nimmt die Leistung von OpenMp unerwartet stark bei 32 Therads ab. Allgemein ersichtlich ist, dass die OpenMP Implementation meistens ein wenig mehr Performance erbringt.
Die maximale Performance wird hingegen der Optimierungsthese überwiegend nicht bei 16 Threads erreicht. Die Performance hat teilweise bereits bei 4 Threads die beste Performance. Der Threading Overhead bei der Verwendung von std::thread macht sich wie

erwartet besonders bei den kleinen Bildern (Datenmengen) bemerkbar und deutlich weniger bei den größeren Bildern(Datenmengen). Die OpenMP Thread Implementation zeigt deutlich weniger Performanceverlust bei kleinen Bildern.
Wird OpenMP gewählt zur Parallelisierung, so wird eine gewisse Abhängigkeit zu dem OpenMP Framework hergestellt. Es kann damit auch das Problem einher gehen, dass der Kunde möglicherweise bereits OpenMP für andere Zwecke, wie zum Beispiel der Parallelisierung von weiteren Bilderverarbeitungsanwendungen, und dadurch die Umgebungsvariable zum Steuern der Thread Anzahl anders Konfiguriert ist, als für die Basler Anwendung benötigt wird. Auf der anderen Seite ist OpenMP sehr leicht zu verwenden und messbar effizient in der Parallelisierung.
Die std::thread Implementation hat einen messbar höheren Overhead. Dennoch ist die Verwendung der std::threads sehr flexibel gestaltbar. Das Erzeugen der std::threads und Starten einer Funktion ist nicht schwerer, als ein normaler Funktionsaufruf selbst. Der Entwickler sollte ein höheres Verständnis über den zu parallelisierenden Code haben, damit diese effektiv umgesetzt werden kann.
Im Bezug auf die Rahmenbedingungen 3.5 ist OpenMP für macOS, Windows und Linux verfügbar. Es erhöht die Performance wesentlich. Da OpenMP nicht zu 100% konfigurierbar ist, kann Rahmenbedingung Nr. 6 nich erfüllt werden.
Die std::thread Implementation bietet genug Flexibilität und Plattform Support, sowie bringt bei der korrekten Anwendung ein Vielfaches an Performance. Damit ist dies die Wahl zur Umsetzung der Parallelisierung.

7 Fazit

In diesem Kapitel werden die Resultate dieser Bachelorarbeit zusammengefasst und ein abschließender Ausblick gegeben.

7.1 Zusammenfassung

Das Ziel dieser Bachelorarbeit war es, durch die prototypische Implementation verschiedener Optimierungsmethoden festzustellen, inwiefern die Performance der Pixelformatkonvertierung effektiv gesteigert werden kann. Dafür wurden drei verschiedene Ansätze umgesetzt.
Der erste Ansatz ist das Cacheline Alignment. Dieser Optimierungsansatz hat sich als ineffizient herausgestellt. Es konnte keine Performance Verbesserung festgestellt werden.
Der zweite Ansatz ist das Vektorisieren. Dies wurde anhand von einer 'eigenen Vektorisierung' und mit Intel IPP umgesetzt. Es konnte festgestellt werden, dass durch eine Vektorisierung eine Performance Steigerung von bis zu 4,5 Mal möglich ist.
Als dritten Optimierungsansatz wurde eine Parallelisierung durchgeführt. Dabei wurden std::thread und OpenMP miteinander verglichen. Es kann durch das Parallelisieren die Performance um bis zu 8 Mal so viel gesteigert werden.

Abschließend ist festzuhalten, dass die prototypische Umsetzung von den drei Optimierungsvarianten zwei effektive Möglichkeiten der Optimierung aufzeigen kann. Damit kann bei der Basler AG die Pixelformatkonvertierung Optimiert werden und es ist nicht notwendig, dass eine komplett neue Lösung entwickelt werden muss.

7.2 Ausblick

Es ist fast unmöglich eine perfekte Optimierung zu erreichen. Es gibt keine 'Alles-in-Einem-Wunderlösung' für ein beliebiges Optimierungsproblem. Aus diesem Grund ist es wichtig das Performance Problem zur Hand möglichst gut zu analysieren und zu verstehen. Damit für den konkreten Grund für Performanceeinbußen eine sinnvolle effektive Lösung umgesetzt werden kann.
Das gleiche gilt auch für den Code des Pixelformatkonverters. Es gibt noch viele weitere Möglichkeiten zur Optimierung, die nicht im Rahmen dieser Bachelorarbeit behandelt worden sind. Außerdem dem können unterschiedliche Ansätze beliebig kombiniert werden, um die best mögliche Optimierung zu finden. Es wäre denkbar die Vektorisierung mit der Parallelisierung zu kombinieren. Es wäre im Bereich des angemessenen zu erwarten, dass diese Kombination bis zu 36 Mal mehr Performance bringen kann.
Eines der wichtigsten Lektionen aus dieser Ausarbeitung ist, dass Optimierung ein Prozess ist und auf einer besonders gründlichen Analyse aufbaut.

Literaturverzeichnis

[1] Alexandrescu, Andrei: Jan 2016. – URL https://www.youtube.com/watch?v=vrfYLlR8X8k. – Letzer Zugriff: 2021-10-14

[2] Basler AG: *pylon Camera Software Suite*. https://www.baslerweb.com/de/produkte/software/basler-pylon-camera-software-suite/. – Letzer Zugriff: 2022-02-12

[3] Börm, Steffen: *Hochleistungsrechnen*. No Publisher, 2019. – URL https://www.informatik.uni-kiel.de/~sb/data/Hochleistung.pdf

[4] Central, AMD D.: *AMD uProf*. – URL https://developer.amd.com/amd-uprof/. – Letzer Zugriff: 2022-02-08

[5] emva: *Pixel Format Naming Convention (PFNC)*. https://www.emva.org/wp-content/uploads/GenICam_PFNC_2_4.pdf. Juni 2021. – Letzer Zugriff: 2022-02-12

[6] Fog, Agner: *The microarchitecture of Intel, AMD, and VIA CPUs - An optimization guide for assembly programmers and compiler makers*. Technical University of Denmark, Aug 2021. – URL https://www.agner.org/optimize/microarchitecture.pdf. – Letzer Zugriff: 2022-02-12

[7] Fog, Agner: *Optimizing software in C++ - An optimization guide for Windows, Linux, and Mac platforms*. Technical University of Denmark, 2021. – URL https://www.agner.org/optimize/#manuals

[8] Guntheroth, K.: *Optimized C++: Proven Techniques for Heightened Performance*. O'Reilly Media, 2016. – URL https://books.google.de/books?id=V1kPDAAAQBAJ. – ISBN 9781491922033

[9] Intel. – URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/ipp.html#gs.etrc5m. – Letzer Zugriff: 2021-10-14

[10] Intel: *Intel® VTune™ Profiler*. – URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.i8kr3m. – Letzter Zugriff: 2022-02-08

[11] Intel: *Intel® Intrinsics Guide*. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. 2021. – Letzter Zugriff: 2022-02-12

[12] Kernighan, B.W. ; Ritchie, D.M. ; Tondo, C.L.: *The C Programming Language*. Prentice Hall, 1988 (Prentice-Hall software series). – URL https://books.google.de/books?id=OpJ_0zpF7jIC. – ISBN 9789688802052

[13] Meyers, S.: *Effektiv C++ programmieren: 55 Möglichkeiten, Ihre Programme und Entwürfe zu verbessern*. Pearson Deutschland, 2011 (Programmer's Choice). – URL https://books.google.de/books?id=jPLzshqOJ9gC. – ISBN 9783827330789

[14] Microsoft: *Auto-Parallelization and Auto-Vectorization*. August 2021. – URL https://docs.microsoft.com/en-us/cpp/parallel/auto-parallelization-and-auto-vectorization?view=msvc-170. – Letzer Zugriff: 2022-02-08

[15] Microsoft: *Vektorisierungs- und Parallelisierungsmeldungen*. January 2022. – URL https://docs.microsoft.com/de-de/cpp/error-messages/tool-errors/vectorizer-and-parallelizer-messages?view=msvc-170. – Letzer Zugriff: 2022-02-08

[16] */arch (x64)*. – URL https://docs.microsoft.com/de-de/cpp/build/reference/arch-x64?view=msvc-160. – Letzer Zugriff: 2021-10-14

[17] */arch (x86)*. – URL https://docs.microsoft.com/de-de/cpp/build/reference/arch-x86?view=msvc-160. – Letzer Zugriff: 2021-10-14

[18] OpenMP Architecture Review Board: *OpenMP Application Programming Interface*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. November 2018. – Letzer Zugriff: 2022-02-08

[19] Pharr, Matt ; Mark, William R.: ispc: A SPMD compiler for high-performance CPU programming. In: *2012 Innovative Parallel Computing (InPar)*, 2012, S. 1–13

[20] Ragan-Kelley, Jonathan. – URL https://halide-lang.org/. – Letzer Zugriff: 2021-10-14

[21] Sushovon Sinha: *Physical and Virtual Memory in Windows 10*. https://answers.microsoft.com/en-us/windows/forum/all/physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938l. Januar 2018. – Letzer Zugriff: 2022-02-12

[22] transcend: *Welche Transferraten erreichen DDR, DDR2, DDR3 und DDR4 ?* https://de.transcend-info.com/Support/FAQ-292. – Letzer Zugriff: 2022-02-12

[23] unbekannt: *AMD Ryzen 7 3700X specifications*. – URL https://www.cpu-world.com/CPUs/Zen/AMD-Ryzen%207%203700X.html. – Letzer Zugriff: 2022-02-08

[24] unbekannt: */O1, /O2 (Größe minimieren, Geschwindigkeit maximieren)*. – URL https://docs.microsoft.com/de-de/cpp/build/reference/o1-o2-minimize-size-maximize-speed?view=msvc-170. – Letzer Zugriff: 2022-02-08

[25] unbekannt: *std::thread*. https://en.cppreference.com/w/cpp/thread/thread. – Letzer Zugriff: 2022-02-12

[26] unbekannt: *What is SSE and AVX?*. – URL https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx. – Letzer Zugriff: 2022-02-08

A Anhang

A.1 Grafiken

Abbildung A.1: FillImage Beispielbild

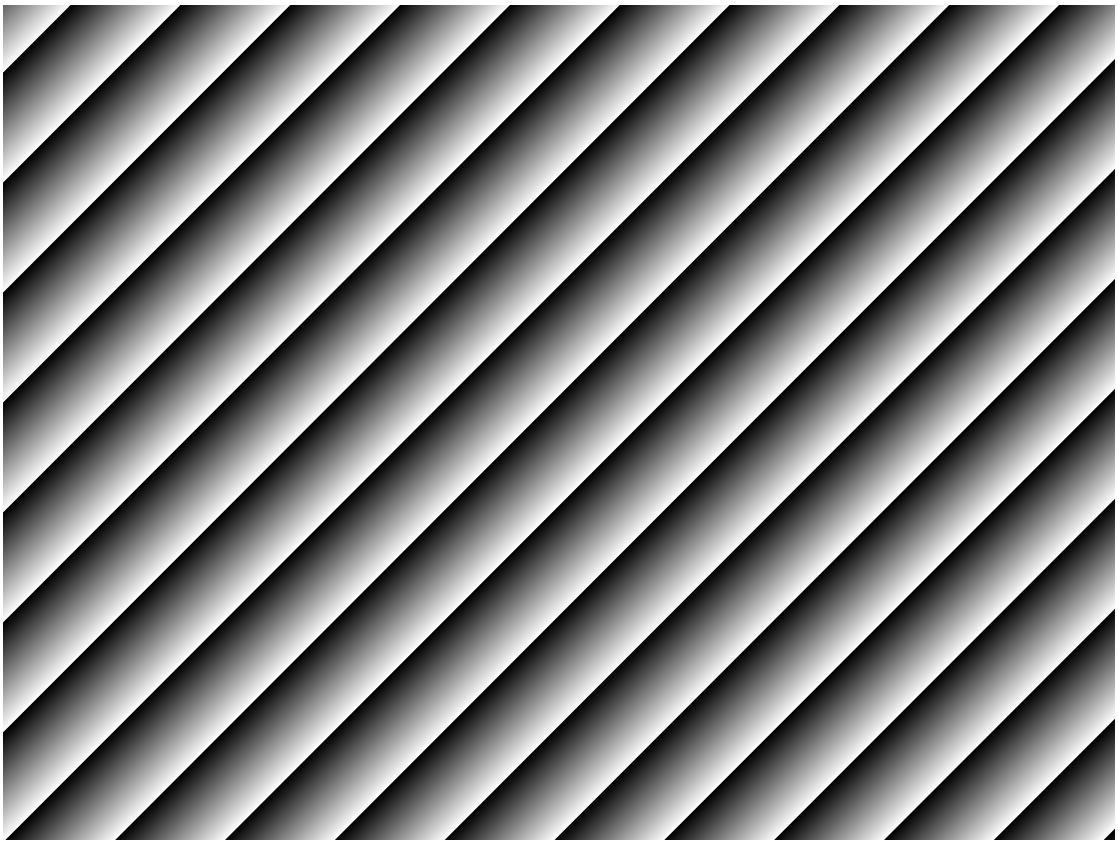
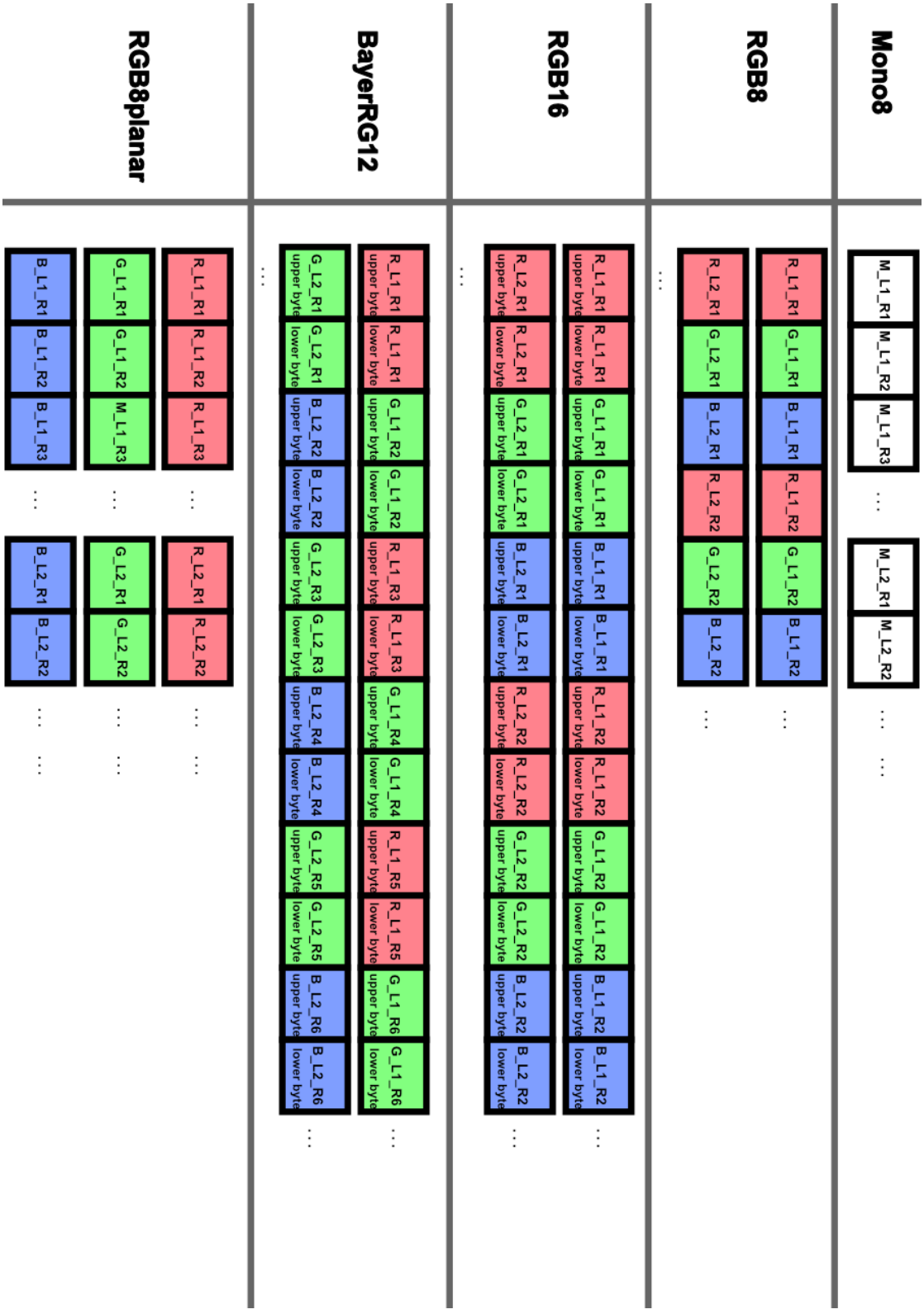


Abbildung A.2: Pixelformate - Speicherbelegung(quer)



A.2 Code

Konvertierung: Mono8 zu Mono8

```
//Vektorisierte Variante
// 32 Bytes each vector
__m256i v0;
__m256i v1;
__m256i v2;
__m256i v3;
for (const PixelT* p = pIn; p < pInEnd; p+=128, pOut+=128)
{
    v0 = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(p));
    v1 = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(p)+1);
    v2 = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(p)+2);
    v3 = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(p)+3);
    _mm256_store_si256(reinterpret_cast<__m256i*>(pOut), v0);
    _mm256_store_si256((reinterpret_cast<__m256i*>(pOut))+1, v1);
    _mm256_store_si256((reinterpret_cast<__m256i*>(pOut))+2, v2);
    _mm256_store_si256((reinterpret_cast<__m256i*>(pOut))+3, v3);
}
//Vektorisierte Variante Ende
//Originaler Code
//prepare copy
const uint8_t* start = reinterpret_cast<const uint8_t*>(pIn);
const uint8_t* end = reinterpret_cast<const uint8_t*>(pInEnd);

//check input as expected
PYLON_ASSERT(start <= end);

//copy, more efficient
memcpy(pOut, pIn, end - start);
//Originaler Code Ende
```

Listing A.1: Händische Vektorisierung: Mono8 zu Mono8

Konvertierung: Mono8 zu RGB16

```
//Vektorisierte Variante
__m128i vInput;

__m128i vMask0 = _mm_set_epi8(0x05, 0x04, 0x04, 0x04, 0x03, 0x03, 0x03, 0x02,
    0x02, 0x02, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00);
```

```
5  __m128i vMask1 = _mm_set_epi8(0x0a, 0x0a, 0x09, 0x09, 0x09, 0x08, 0x08, 0x08,
       0x07, 0x07, 0x07, 0x06, 0x06, 0x06, 0x05, 0x05);
6  __m128i vMask2 = _mm_set_epi8(0x0f, 0x0f, 0x0f, 0x0e, 0x0e, 0x0e, 0x0d, 0x0d,
       0x0d, 0x0c, 0x0c, 0x0c, 0x0b, 0x0b, 0x0b, 0x0a);
7  __m128i vMaskSwapBytes = _mm_set_epi8(0x0e, 0x0f, 0x0c, 0x0d, 0x0a, 0x0b, 0
      x08, 0x09, 0x06, 0x07, 0x04, 0x05, 0x02, 0x03, 0x00, 0x01);
8
9  __m256i vTemp0;
10 __m256i vTemp1;
11 __m256i vTemp2;
12
13 __m128i vRes0;
14 __m128i vRes1;
15 __m128i vRes2;
16 __m128i vRes3;
17 __m128i vRes4;
18 __m128i vRes5;
19
20 for (; pIn < pInEnd; pIn+=16, pOut+=16)
21 {
22     vInput = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pIn));
23     vTemp0 = _mm256_cvtepi8_epi16(_mm_shuffle_epi8(vInput, vMask0));
24     vRes0 = _mm_shuffle_epi8(_mm256_extractf128_si256(vTemp0, 0),
      vMaskSwapBytes);
25     vRes1 = _mm_shuffle_epi8(_mm256_extractf128_si256(vTemp0, 1),
      vMaskSwapBytes);
26     vTemp1 = _mm256_cvtepi8_epi16(_mm_shuffle_epi8(vInput, vMask1));
27     vRes2 = _mm_shuffle_epi8(_mm256_extractf128_si256(vTemp1, 0),
      vMaskSwapBytes);
28     vRes3 = _mm_shuffle_epi8(_mm256_extractf128_si256(vTemp1, 1),
      vMaskSwapBytes);
29     vTemp2 = _mm256_cvtepi8_epi16(_mm_shuffle_epi8(vInput, vMask2));
30     vRes4 = _mm_shuffle_epi8(_mm256_extractf128_si256(vTemp2, 0),
      vMaskSwapBytes);
31     vRes5 = _mm_shuffle_epi8(_mm256_extractf128_si256(vTemp2, 1),
      vMaskSwapBytes);
32     _mm_store_si128((reinterpret_cast<__m128i*>(pOut)), vRes0);
33     _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+1, vRes1);
34     _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+2, vRes2);
35     _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+3, vRes3);
36     _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+4, vRes4);
37     _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+5, vRes5);
38
39 }
```

```
40 //Vektorisierte Variante Ende
41 //Originaler Code
42 for (; pIn < pInEnd; ++pIn, ++pOut)
43 {
44     typename OutT::value_type const shifted = ShiftChannel<ShiftValuesT, InT,
       typename OutT::value_type>( *pIn );
45     new (pOut) OutT( shifted, shifted, shifted );
46 }
47 //Originaler Code Ende
```

Listing A.2: Händische Vektorisierung: Mono8 zu RGB16

Konvertierung: RGB8planar zu Mono8

```
1  //Vektorisierte Variante
2  // 16 Byte each vector
3  __m128i vR;
4  __m128i vG;
5  __m128i vB;
6
7  __m256i c5 = _mm256_set1_epi16(5);
8  __m256i wR;
9  __m256i wG;
10 __m256i wB;
11 __m256i res1;
12 __m256i res2;
13 __m256i res;
14 __m128i data;
15
16 for (; pInRed < pInRedEnd; pInRed += 16, pInGreen += 16, pInBlue += 16, pOut
       += 16)
17 {
18     vR = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInRed));
19     vG = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInGreen));
20     vB = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInBlue));
21     wR = _mm256_slli_epi16(_mm256_cvtepu8_epi16(vR), 1);
22     wG = _mm256_mullo_epi16(_mm256_cvtepu8_epi16(vG), c5);
23     wB = _mm256_cvtepu8_epi16(vB);
24     res1 = _mm256_add_epi16(wR, wG);
25     res2 = _mm256_add_epi16(res1, wB);
26     res = _mm256_srai_epi16(res2, 3);
27     data = _mm_packus_epi16(_mm256_extractf128_si256(res, 0),
       _mm256_extractf128_si256(res, 1));
```

```
28      _mm_store_si128((reinterpret_cast<__m128i*>(pOut)), data);
29  }
30  //Vektorisierte Variante Ende
31  //Originaler Code
32  for (; pInRed < pInRedEnd; ++pInRed, ++pInGreen, ++pInBlue, ++pOut)
33  {
34      //std::cout << "here" << std::endl;
35      new (pOut) OutT( ShiftChannel<ShiftValuesT, InT, OutT>( *pInRed ),
36                       ShiftChannel<ShiftValuesT, InT, OutT>( *pInGreen ),
37                       ShiftChannel<ShiftValuesT, InT, OutT>( *pInBlue ) );
38  }
39  //Originaler Code Ende
```

Listing A.3: Händische Vektorisierung: RGB8planar zu Mono8

Konvertierung: RGB8planar zu RGB8

```
1   //Vektorisierte Variante
2   // 16 Byte each vector
3   __m128i vR, vG, vB;
4
5   __m128i vMaskR0 = _mm_set_epi8(0x05, 0x80, 0x80, 0x04, 0x80, 0x80, 0x03, 0x80
        , 0x80, 0x02, 0x80, 0x80, 0x01, 0x80, 0x80, 0x00);
6   __m128i vMaskR1 = _mm_set_epi8(0x80, 0x0a, 0x80, 0x80, 0x09, 0x80, 0x80, 0x08
        , 0x80, 0x80, 0x07, 0x80, 0x80, 0x06, 0x80, 0x80);
7   __m128i vMaskR2 = _mm_set_epi8(0x80, 0x80, 0x0f, 0x80, 0x80, 0x0e, 0x80, 0x80
        , 0x0d, 0x80, 0x80, 0x0c, 0x80, 0x80, 0x0b, 0x80);
8
9   __m128i vMaskG0 = _mm_set_epi8(0x80, 0x80, 0x04, 0x80, 0x80, 0x03, 0x80, 0x80
        , 0x02, 0x80, 0x80, 0x01, 0x80, 0x80, 0x00, 0x80);
10  __m128i vMaskG1 = _mm_set_epi8(0x0a, 0x80, 0x80, 0x09, 0x80, 0x80, 0x08, 0x80
        , 0x80, 0x07, 0x80, 0x80, 0x06, 0x80, 0x80, 0x05);
11  __m128i vMaskG2 = _mm_set_epi8(0x80, 0x0f, 0x80, 0x80, 0x0e, 0x80, 0x80, 0x0d
        , 0x80, 0x80, 0x0c, 0x80, 0x80, 0x0b, 0x80, 0x80);
12
13  __m128i vMaskB0 = _mm_set_epi8(0x80, 0x04, 0x80, 0x80, 0x03, 0x80, 0x80, 0x02
        , 0x80, 0x80, 0x01, 0x80, 0x80, 0x00, 0x80, 0x80);
14  __m128i vMaskB1 = _mm_set_epi8(0x80, 0x80, 0x09, 0x80, 0x80, 0x08, 0x80, 0x80
        , 0x07, 0x80, 0x80, 0x06, 0x80, 0x80, 0x05, 0x80);
15  __m128i vMaskB2 = _mm_set_epi8(0x0f, 0x80, 0x80, 0x0e, 0x80, 0x80, 0x0d, 0x80
        , 0x80, 0x0c, 0x80, 0x80, 0x0b, 0x80, 0x80, 0x0a);
16
```

```
__m128i vBlendMask0 = _mm_set_epi8(0x80, 0x44, 0x00, 0x80, 0x44, 0x00, 0x80,
    0x44, 0x00, 0x80, 0x44, 0x00, 0x80, 0x44, 0x00, 0x80);
__m128i vBlendMask1 = _mm_set_epi8(0x00, 0x80, 0x44, 0x00, 0x80, 0x44, 0x00,
    0x80, 0x44, 0x00, 0x80, 0x44, 0x00, 0x80, 0x44, 0x00);
__m128i vBlendMask2 = _mm_set_epi8(0x44, 0x00, 0x80, 0x44, 0x00, 0x80, 0x44,
    0x00, 0x80, 0x44, 0x00, 0x80, 0x44, 0x00, 0x80, 0x44);

__m128i vTempR0, vTempR1, vTempR2, vTempG0, vTempG1, vTempG2, vTempB0,
    vTempB1, vTempB2, vOut0, vOut1, vOut2;
for (; pInRed < pInRedEnd; pInRed += 16, pInGreen += 16, pInBlue += 16, pOut
    += 16)
{
    vR = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInRed));
    vG = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInGreen));
    vB = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInBlue));

    vTempR0 = _mm_shuffle_epi8(vR, vMaskR0);
    vTempR1 = _mm_shuffle_epi8(vR, vMaskR1);
    vTempR2 = _mm_shuffle_epi8(vR, vMaskR2);
    vTempG0 = _mm_shuffle_epi8(vG, vMaskG0);
    vTempG1 = _mm_shuffle_epi8(vG, vMaskG1);
    vTempG2 = _mm_shuffle_epi8(vG, vMaskG2);
    vTempB0 = _mm_shuffle_epi8(vB, vMaskB0);
    vTempB1 = _mm_shuffle_epi8(vB, vMaskB1);
    vTempB2 = _mm_shuffle_epi8(vB, vMaskB2);
    vOut0 = _mm_blendv_epi8(_mm_blendv_epi8(vTempG0, vTempR0, vBlendMask0),
    vTempB0, vBlendMask1);
    vOut1 = _mm_blendv_epi8(_mm_blendv_epi8(vTempG1, vTempR1, vBlendMask1),
    vTempB1, vBlendMask2);
    vOut2 = _mm_blendv_epi8(_mm_blendv_epi8(vTempG2, vTempR2, vBlendMask2),
    vTempB2, vBlendMask0);
    _mm_store_si128((reinterpret_cast<__m128i*>(pOut)), vOut0);
    _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+1, vOut1);
    _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+2, vOut2);
}
//Vektorisierte Variante Ende
//Originaler Code
for (; pInRed < pInRedEnd; ++pInRed, ++pInGreen, ++pInBlue, ++pOut)
{
    new (pOut) OutT( ShiftChannel<ShiftValuesT, InT, OutT>( *pInRed ),
                     ShiftChannel<ShiftValuesT, InT, OutT>( *pInGreen ),
                     ShiftChannel<ShiftValuesT, InT, OutT>( *pInBlue ) );
}
```

```
52 //Originaler Code Ende
```

Listing A.4: Händische Vektorisierung: RGB8planar zu RGB8

Konvertierung: RGB8planar zu RGB16

```
1  //Vektorisierte Variante
2  // 16 Byte each vector
3  __m128i vR, vG, vB;
4  // Fuer die Initialisierung dieser Masken-Vektoren siehe 'Haendische
       Vektorisierung: RGB8planar zu RGB8'.
5  __m128i vMaskR0, vMaskR1, vMaskR2, vMaskG0, vMaskG1, vMaskG2, vMaskB0,
       vMaskB1, vMaskB2;
6  __m128i vBlendMask0, vBlendMask1, vBlendMask2;
7
8  __m128i vMaskSwapBytes = _mm_set_epi8(0x0e, 0x0f, 0x0c, 0x0d, 0x0a, 0x0b, 0
       x08, 0x09, 0x06, 0x07, 0x04, 0x05, 0x02, 0x03, 0x00, 0x01);
9
10 __m128i vTempR0, vTempR1, vTempR2, vTempG0, vTempG1, vTempG2, vTempB0,
       vTempB1, vTempB2;
11 __m256i vTemp0, vTemp1, vTemp2;
12 __m128i vOut0, vOut1, vOut2;
13 for (; pInRed < pInRedEnd; pInRed += 16, pInGreen += 16, pInBlue += 16, pOut
       += 16)
14 {
15     vR = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInRed));
16     vG = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInGreen));
17     vB = _mm_loadu_si128(reinterpret_cast<const __m128i*>(pInBlue));
18
19     vTempR0 = _mm_shuffle_epi8(vR, vMaskR0);
20     vTempR1 = _mm_shuffle_epi8(vR, vMaskR1);
21     vTempR2 = _mm_shuffle_epi8(vR, vMaskR2);
22     vTempG0 = _mm_shuffle_epi8(vG, vMaskG0);
23     vTempG1 = _mm_shuffle_epi8(vG, vMaskG1);
24     vTempG2 = _mm_shuffle_epi8(vG, vMaskG2);
25     vTempB0 = _mm_shuffle_epi8(vB, vMaskB0);
26     vTempB1 = _mm_shuffle_epi8(vB, vMaskB1);
27     vTempB2 = _mm_shuffle_epi8(vB, vMaskB2);
28     vOut0 = _mm_blendv_epi8(_mm_blendv_epi8(vTempG0, vTempR0, vBlendMask0),
       vTempB0, vBlendMask1);
29     vOut1 = _mm_blendv_epi8(_mm_blendv_epi8(vTempG1, vTempR1, vBlendMask1),
       vTempB1, vBlendMask2);
```

```
30      vOut2 = _mm_blendv_epi8(_mm_blendv_epi8(vTempG2, vTempR2, vBlendMask2),
        vTempB2, vBlendMask0);
31      vTemp0 = _mm256_cvtepi8_epi16(vOut0);
32      vTemp1 = _mm256_cvtepi8_epi16(vOut1);
33      vTemp2 = _mm256_cvtepi8_epi16(vOut2);
34      _mm_store_si128((reinterpret_cast<__m128i*>(pOut)), _mm_shuffle_epi8(
        _mm256_extractf128_si256(vTemp0, 0), vMaskSwapBytes));
35      _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+1, _mm_shuffle_epi8(
        _mm256_extractf128_si256(vTemp0, 1), vMaskSwapBytes));
36      _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+2, _mm_shuffle_epi8(
        _mm256_extractf128_si256(vTemp1, 0), vMaskSwapBytes));
37      _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+3, _mm_shuffle_epi8(
        _mm256_extractf128_si256(vTemp1, 1), vMaskSwapBytes));
38      _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+4, _mm_shuffle_epi8(
        _mm256_extractf128_si256(vTemp2, 0), vMaskSwapBytes));
39      _mm_store_si128((reinterpret_cast<__m128i*>(pOut))+5, _mm_shuffle_epi8(
        _mm256_extractf128_si256(vTemp2, 1), vMaskSwapBytes));
40  }
41  //Vektorisierte Variante Ende
42  //Originaler Code
43  for (; pInRed < pInRedEnd; ++pInRed, ++pInGreen, ++pInBlue, ++pOut)
44  {
45      new (pOut) OutT( ShiftChannel<ShiftValuesT, InT, OutT>( *pInRed ),
46                       ShiftChannel<ShiftValuesT, InT, OutT>( *pInGreen ),
47                       ShiftChannel<ShiftValuesT, InT, OutT>( *pInBlue ) );
48  }
49  //Originaler Code Ende
```

Listing A.5: Händische Vektorisierung: RGB8planar zu RGB16

```
1  //init variables
2  const size_t bufferSizeIn = cWidth * cHeight * bytePerPixel;
3  const size_t bufferSizeOut = cWidth * cHeight * bytePerPixel;
4  const size_t constBufferSizeIncPaddingIn = bufferSizeIn + 64; // + 64, fuer
       64 Byte Cache Line Alignment
5  const size_t constBufferSizeIncPaddingOut = bufferSizeOut + 64; // + 64, fuer
        64 Byte Cache Line Alignment
6  std::vector< std::vector<char> > buffersIn(cCountImages);
7  std::vector< std::vector<char> > buffersOut(cCountImages);
8  std::vector<Pylon::CPylonImage> imagesIn(cCountImages);
9  std::vector<Pylon::CPylonImage> imagesOut(cCountImages);
10
11 //create and fill - input
12 int offset = 0;
13 for (size_t i = 0; i < cCountImages; ++i)
```

```
14 {
15     size_t bufferSizeIncPadding = constBufferSizeIncPaddingIn;
16     Pylon::CPylonImage& image = imagesIn[i];
17     buffersIn[i].resize(bufferSizeIncPadding);
18     void* var = buffersIn[i].data();
19     std::align(64, bufferSizeIn, var, bufferSizeIncPadding);
20     image.AttachUserBuffer((char*)var, bufferSizeIn, PixelType_Mono8, cWidth,
       cHeight, 0);
21     FillImage<uint8_t, 1>(image, 256, offset++);
22 }
23 //create - output
24 for (size_t i = 0; i < cCountImages; ++i)
25 {
26     size_t bufferSizeIncPadding = constBufferSizeIncPaddingOut;
27     Pylon::CPylonImage& image = imagesOut[i];
28     void* var = new char[constBufferSizeIncPaddingOut];
29     std::align(64, bufferSizeOut, var, bufferSizeIncPadding);
30     image.AttachUserBuffer((char*)var, bufferSizeOut, PixelType_Mono8, cWidth
      , cHeight, 0);
31 }
```

Listing A.6: C++ Code These 1 - Cacheline Alignment

A.3 Tabellen

Beschriftung	Baseline	openMP	std_thread
1 T / 5MP	16250.3		
1 T / 24,4MP	16320.6		
2 T / 5MP		18439.2	15015.8
2 T / 24,4MP		18524.5	17222.8
4 T / 5MP		18167.2	13043.7
2 T / 24,4MP		18246.7	16753.7
8 T / 5MP		17421.5	13509.4
2 T / 24,4MP		17779.9	16225.4
16 T / 5MP		16843.6	10772.6
16 T / 24,4MP		17306.7	15971.3
32 T / 5MP		17498.1	5979.61
32 T / 24,4MP		17517.1	14472.6

Tabelle A.1: These3: Mono8 zu Mono8 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	5504.99		
1 T / 24,4MP	5536.93		
2 T / 5MP		10266.3	9458.04
2 T / 24,4MP		10741.2	10153
4 T / 5MP		13552.3	12599.1
2 T / 24,4MP		15170.2	14470.2
8 T / 5MP		13211.4	12006.9
2 T / 24,4MP		14526.4	14030.8
16 T / 5MP		14158.7	10932.4
16 T / 24,4MP		14281	13760.2
32 T / 5MP		13527.1	9102.96
32 T / 24,4MP		13698.5	13267.8

Tabelle A.2: These3: Mono8 zu RGB8 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	8943.58		
1 T / 24,4MP	8938.6		
2 T / 5MP		13535.4	12461.1
2 T / 24,4MP		13616.4	13169
4 T / 5MP		14599.3	13286.4
2 T / 24,4MP		14701.3	14139
8 T / 5MP		14180.2	12982.4
2 T / 24,4MP		14169.1	13786.5
16 T / 5MP		13841.2	12486
16 T / 24,4MP		13869.1	13542.6
32 T / 5MP		14136	11379
32 T / 24,4MP		14208.6	13310.8

Tabelle A.3: These3: Mono8 zu RGB16 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	1741.38		
1 T / 24,4MP	1796.82		
2 T / 5MP		3043.13	4147.68
2 T / 24,4MP		3269.73	4202.15
4 T / 5MP		5752.27	7537.9
2 T / 24,4MP		6186.15	7605.25
8 T / 5MP		8551.37	9645.97
2 T / 24,4MP		8656.67	10546.2
16 T / 5MP		14128.3	9647.66
16 T / 24,4MP		14035.6	12991.4
32 T / 5MP		8137.45	7733.54
32 T / 24,4MP		8163.68	12045.5

Tabelle A.4: These3: BayerRG12 zu Mono8 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	3529.75		
1 T / 24,4MP	3495.82		
2 T / 5MP		6923.92	6503.17
2 T / 24,4MP		6950.35	6820.79
4 T / 5MP		12779.6	11865.1
2 T / 24,4MP		13155.6	12839.6
8 T / 5MP		14334.1	12849.3
2 T / 24,4MP		14693.6	14900.1
16 T / 5MP		14507.1	12339.1
16 T / 24,4MP		14629.6	14276.5
32 T / 5MP		13073.5	10761
32 T / 24,4MP		13009.5	14149.6

Tabelle A.5: These3: BayerRG12 zu RGB8 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	4991.2		
1 T / 24,4MP	5193.09		
2 T / 5MP		9612.24	10962.1
2 T / 24,4MP		9697.94	10724.2
4 T / 5MP		14883.4	14042.2
2 T / 24,4MP		15105.5	14654.8
8 T / 5MP		14692.1	13415
2 T / 24,4MP		14782.4	14365.6
16 T / 5MP		14254.8	13223
16 T / 24,4MP		14226.3	14144.6
32 T / 5MP		14299.7	11971
32 T / 24,4MP		14468.7	13938.1

Tabelle A.6: These3: BayerRG12 zu RGB16 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	4304.88		
1 T / 24,4MP	4319.2		
2 T / 5MP		10661.9	7922.88
2 T / 24,4MP		10746.4	8408.03
4 T / 5MP		19830.8	14082.3
2 T / 24,4MP		19900.6	15775.7
8 T / 5MP		18025.2	14527.1
2 T / 24,4MP		18371.9	16914.9
16 T / 5MP		18797.8	13915.2
16 T / 24,4MP		19091.6	17361.9
32 T / 5MP		16860.9	10653
32 T / 24,4MP		17019	16803.1

Tabelle A.7: These3: RGB8planar zu Mono8 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	8220.21		
1 T / 24,4MP	8283.71		
2 T / 5MP		15515.3	13863
2 T / 24,4MP		15633.3	14757.3
4 T / 5MP		17727.4	15632.6
2 T / 24,4MP		17889.8	16988.8
8 T / 5MP		17062.7	15267.2
2 T / 24,4MP		17177.3	16650
16 T / 5MP		16686.8	14537.1
16 T / 24,4MP		16740	16263.8
32 T / 5MP		16964.8	12307.2
32 T / 24,4MP		16907	15788.1

Tabelle A.8: These3: RGB8planar zu RGB8 (T - Threadanzahl)

Beschriftung	Baseline	OpenMP	std::thread
1 T / 5MP	10007.8		
1 T / 24,4MP	10138		
2 T / 5MP		15478.4	14279.8
2 T / 24,4MP		15399.5	14812.8
4 T / 5MP		16212.6	14895.7
2 T / 24,4MP		16174.1	15541.7
8 T / 5MP		15723.9	14434.4
2 T / 24,4MP		15759.8	15346.2
16 T / 5MP		15298.3	14188.5
16 T / 24,4MP		15338.7	15039.5
32 T / 5MP		15711.1	12735.2
32 T / 24,4MP		15565.9	14658.3

Tabelle A.9: These3: RGB8planar zu RGB16 (T - Threadanzahl)

Glossar

Bit Kleinste Speichereinheit im Computer. Ein Bit kann zwei Zustände haben, entweder 0 oder 1..

Byte Besteht aus acht Bit. Ein Byte kann 256 Werte darstellen..

C++ Der verwendete C++ Standard bezieht sich auf den C++ 11 Standard [**?**]. Der Standard kommt aus dem Jahr 2011. Daher sind die meisten Quellen bis zum Jahr 2011 relevant. Teilweise sind sogar ältere Quellen verwendet, da C++ über viele Jahre hinweg sich entwickelt hat und somit auch ältere Elemente enthält. Teilweise sogar Aspekte aus der Programmiersprache C, welche noch älter sind..

Pixel Die Bildinformation an einer expliziten Position in einem digitalen Bild..

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original