Universität Stuttgart
Institut für Parallele und Verteilte Systeme
Universitätsstraße 38
D-70569 Stuttgart

Studienarbeit Nr. 2342

Parallelisierung eines JPEG-LS Decoders und Visualisierung von Hochgeschwindigkeits-Videoaufnahmen

Yurij Gera

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. Sven Simon
Betreuer:	Dipl.-Math. techn. Philipp Werner
Beginn:	26.04.2011
Ende:	26.10.2011
CR-Classification:	D.1.3, I.4.2, I.4.0, E.4

Abstract

Verarbeitung und Visualisierung von Hochgeschwindigkeits-Videoaufnahmen ist mit hohen Datenaufkommen und rechenintensiven Verarbeitung verbunden. Die für das Projekt spezifizierten Video-Daten sind mit einem modifizierten JPEG-LS Verfahren codiert und liegen in einem besonderen Format vor. Zentrales Ziel dieser Arbeit ist es, einen leistungsfähigen Software-Decoder für die Visualisierung und anschließende Weiterverarbeitung der vorliegenden Video-Ströme zu entwickeln. Dafür soll ein Framework geschaffen werden, welches es dem Benutzer ermöglicht mit den Videoaufnahmen visuell zu arbeiten. Ferner soll Anbindung an externe Systeme, etwa durch Konvertierung der Videoströme in standardisierte Austauschformate oder die Bereitstellung der Framework-Funktionalität durch APIs, geschaffen werden.

Inhalt

1. Einleitung

1.1 Hochgeschwindigkeits-Aufnahmesysteme

Hochgeschwindigkeitskameras ermöglichen das Festhalten von zeitlich sehr schnell ablaufenden und dadurch von einem Menschen nicht mehr wahrnehmbaren Vorgängen. Aufnahmen dieser Vorgänge können anschließend in einem Postprocessing-Schritt analysiert, verarbeitet und dem Betrachter präsentiert werden. Diese Technik ermöglicht vielfältige industrielle und wissenschaftliche Anwendungsmöglichkeiten. Crashtest-Analyse [16] in der Automobilindustrie oder *Particle Image Velocimetry* [17] in der Grundlagenforschung sind nur einige davon. Selbst in der Unterhaltungsindustrie finden solche Kameras Einsatz – als bekanntestes Beispiel sei *Zeitlupe* für Effekt- oder Sportaufnahmen [18] genannt.

Erste grobe Unterscheidung der Hochgeschwindigkeitskameras erfolgt nach dem eingesetzten Aufnahmeverfahren. Mit *Analogen Verfahren* wurden bereits im Jahr 1986 Aufnahmen mit bis zu 200.000 Bildern pro Sekunde gezeigt [19]. *Digitale Verfahren* basieren auf CMOS CMOS Sensoren und liefern je nach Auflösung zwischen 1000 bis 60000 Bilder/s ([1] Abschnitt 3.2). Die schnellsten derzeit bekannten Ultra-Hochgeschwindigkeitskameras arbeiten nach dem *Elektronischen Verfahren*. Die PCO HSFC Pro soll z.B. bis zu 500 Millionen Bilder pro Sekunde erreichen ([1] Abschnitt 4.3). Die Aufnahmedauer ist bei diesen Kameras jedoch sehr beschränkt.

Hochgeschwindigkeitsaufnahmesysteme müssen vielen Anforderungen genügen, einige der wichtigsten davon sind Belichtung, Speicherung der Daten und Auslöser. Mit kleiner werdender Belichtungszeit werden stärkere Belichtungsquellen, etwa extrem leistungsstarke Blitzgeräte, benötigt. Der Aufwand für die Belichtung überwiegt manchmal den Aufwand für die eigentliche Aufnahmetechnik. Unerwünschte Faktoren wie Inhomogenität im CCD Chip, Wechselwirkungen der Umladevorgänge und zu geringe Sensitivität zur Detektion von sehr kleinen Photonenmengen pro Pixel verfälschen die aufgenommenen Luminanzwerte. Bei den Aufnahmen erfasste Datenmengen stellen enorme Anforderungen an das eingesetzte Speichersystem, sowohl hinsichtlich der Bandbreite als auch der Kapazität. Aus diesem Grund ist eine kontinuierliche Aufnahme über längere Zeit nicht immer möglich. Um ein Ereignis von der begrenzten Aufnahmedauer zu erfassen, werden zudem geeignete Auslösevorrichtungen benötigt.

1.2 Voraussetzungen

Als Ausgangspunkt für die Visualisierung von Hochgeschwindigkeitsaufnahmen dient ein, mit CMOS Sensor ausgestattetes, Kamera-System. Der 3 Megapixel auflösende Sensor erfasst 485 Graustufenbilder/s mit einer Luminanztiefe von 8 Bit pro Pixel.

Um die dabei entstehenden hohen Datenmengen effizient zu übertragen und zu speichern, werden diese in ihrer Größe reduziert. Dabei dürfen die Daten weder verfälscht noch geändert werden. Als verlustloses Kompressionsverfahren ist in der Kamera-Hardware modifiziertes JPEG-LS Encoder implementiert. Seine Besonderheit besteht darin, dass die Bilddaten in Segmente unterteilt und parallel kodiert werden. Die entstehenden Datenströme werden in Echtzeit zusammengefügt (verschränkt), paketweise über Ethernet-Schnittstelle zum PC übertragen und auf Massenspeicher abgelegt. Diese Verschränkung wird später bei der Parallelisierung eine wichtige Rolle spielen.

Für die anschließende Weiterverarbeitung und Visualisierung dieser Aufnahmen müssen die komprimierten Datenströme von Datenträgern geladen und dekodiert werden. Als kosteneffiziente Lösung bieten sich hierfür handelsübliche Rechner-Systeme. Für die nachfolgende Ausarbeitung wird daher ein PC-System als Hardware-Basis verwendet.

Es existiert hier eine große Vielfalt an verschiedenen Massenspeicher-Lösungen mit unterschiedlichen Kapazitäten und Bandbreiten, einige davon werden in nachfolgenden Kapiteln vorgestellt und zur Messungen eingesetzt. Zur Dekodierung wird klassischer Prozessor (*Central Processing Unit, CPU*) verwendet. Dadurch, dass moderne Prozessoren gleich mit mehreren Kernen ausgestattet sind, bieten sich für eine Parallelisierung des Decoders an.

Seit einigen Jahren lassen sich auch Grafikkarten zur Berechnungen einsetzten. Deren Verwendung wird als *General Purpose Computation on Graphics Processing Unit* (kurz *GPGPU*) bezeichnet. Für die Untersuchung der GPU Parallelisierung stehen zwei Grafikkarten mit unterschiedlichen Leistungsdaten zur Verfügung.

1.3 Ziele

Die wichtigsten Ziele der folgenden Betrachtungen ist die Erforschung der Möglichkeiten zur effizienten Speicherung, Bereitstellung und Visualisierung von Hochgeschwindigkeitsaufnahmen mit zugänglichen Mitteln, etwa mit einem PC System mit einem oder mehreren Mehrkern-Prozessoren und einer dedizierten Grafikkarte. Effizienz soll in erster Linie hinsichtlich der Maximierung des Durchsatzes entlang aller Verarbeitungsstufen erreicht werden. Gleichzeitig darf der Speicherplatzbedarf nicht außer Acht gelassen werden.

Bei der Bereitstellung von Videoaufnahmen sind gleich mehrere Komponenten eines PCs involviert: Festplatten, I/O Schnittstellen, Systemspeicher, Cache, Systembusse, CPU und GPU. Die Speicherkapazitäten, Bandbreiten und andere Leistungsdaten dieser Hardware müssen als gegeben betrachtet werden.

Auf der anderen Seite ist die, auf die Bilddaten angewendete, Encoder-Funktion eindeutig definiert. Damit ist auch die inverse Decoder-Funktion eindeutig bestimmt.

Sowohl Hardware als auch die Decoder-Funktion sind vorgegeben. Eine Durchsatzsteigerung kann nur durch effiziente Ausnutzung der Hardwareressourcen durch die Decoder-Funktion erreicht werden – dies stellt den einzigen Freiheitsgrad nachfolgender Betrachtungen. Das Ziel der Ausarbeitung ist also die Suche nach einer optimalen Decoder-Implementierung. Um das Ziel zu erreichen, werden unterschiedliche Optimierungsmaßnahmen und Parallelisierungs-Strategien untersucht. Ihre Wirkung wird dabei durch Messungen verifiziert.

2 Hardwarebasis

Zuerst werden die wichtigsten Daten des verwendeten Kamera-Systems kurz vorgestellt. Danach wird ein Blick auf unterschiedliche Komponenten eines PC-Systems hinsichtlich ihrer Kapazitäten und Bandbreiten geworfen. Aus den Bandbreiten und anfallenden Datengrößen einer Aufnahme lassen sich erste Abschätzungen über den erreichbaren Durchsatz treffen. Abschließend werden x86 und GPU Architekturen betrachtet. Ihre Funktionsweise ist für. effiziente Ressourcennutzung ausschlaggebend..

2.1 Kamerasystem

Das Herzstück des Kamera-Systems ist ein 3 Megapixel auflösendes High Speed CMOS Sensor LUPA 3000 des Herstellers Cypress Semiconductor Co. Die wichtigsten technischen Eckdaten [20] des Sensors:

- Auflösung: 1696 x 1710 Bildpunkte
- Digitale Graustufen- oder Farb-Ausgabe
- Bildrate: 485 Bilder/s
- 32 serielle LVDS Verbindungen mit je 412 MBit/s
- 369-pin µPGA Bauweise

Der CMOS Sensor ist über LVDS Verbindungen an einen Xilinx FPGA Board der Spartan-6 Familie angeschlossen. Die Bilddaten werden vom FPGA Board ausgelesen und komprimiert.

Für die Kompression wurde von Anto Joys Yesuadimai Michael in seiner Arbeit „Efficient Context Modelling and Segmentation for Parallel Lossless Image Compression“ [2] ein paralleles Kompressionsverfahren auf Basis von JPEG-LS entworfen und in VHDL implementiert. Um die vorhandenen FPGA Ressourcen effizient zu nutzen, hat Anto J.Y.M. mehre Modifikationen im JPEG-LS Kompressionsverfahren eingeführt. Dadurch wurde eine 25% höhere Betriebsfrequenz mit gleichzeitig geringeren Speicherplatzbedarf erzielt.

Die Bilder werden vom Encoder in 8 Segmente unterteilt (Abb. 1) und parallel kodiert. In jedem Segment entsteht pro Bildpunkt ein Codewort variabler Länge. Das modifizierte JPEG-LS verwendet einen Multiplexer-Baum um diese 8 Codewörter nacheinander in eine Warteschlange (FIFO Puffer) abzuspeichern (Abb. 2), von wo sie paketweise über Gigabit-Ethernet

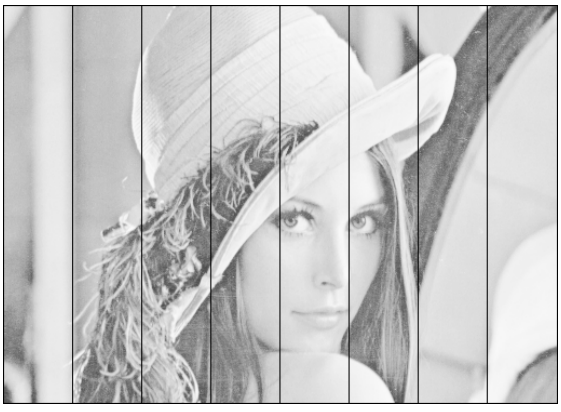


Abbildung 1: Bildsegmentierung im Encoder.

Schnittstelle zum Zielsystem (PC) transportiert werden. Dabei entsteht ein kontinuierlicher Datenstrom mit nacheinander verschränkten Codewörtern. Ein Ausschnitt aus diesem Datenstrom, welcher genau alle Codewörter eines Bilds enthält, wird nachfolgend *Block* genannt.

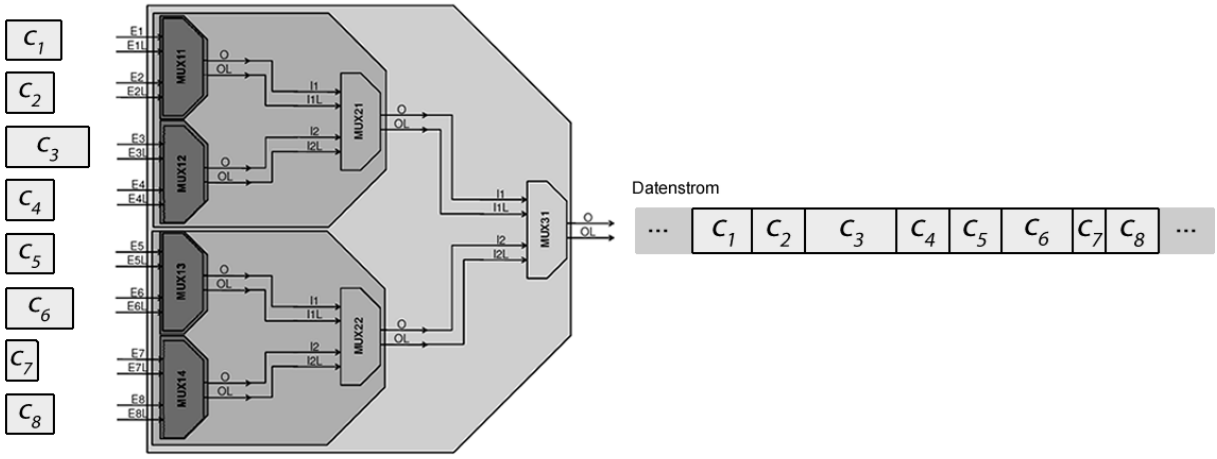


Abbildung 2: Zusammenfügen der Segment-Codewörter durch Multiplexer-Baum. Quelle: [2]

Genauere Einzelheiten zu der Funktionsweise des originalen JPEG-LS sowie zu den Modifikationen werden im Kapitel 3 vorgestellt.

2.2 Datenmenge und Datenrate

Die von der Hochgeschwindigkeitskamera gelieferten und vom Hardware-Encoder komprimierten Datenströme müssen zunächst zum geeigneten Zielsystem transportiert und gespeichert werden. Für die Verarbeitung und Visualisierung müssen die gespeicherten Datenblöcke von Datenträgern wieder geladen und dekodiert werden. Um die Anforderungen an die, im nächsten Abschnitt vorgestellten, I/O Subsysteme bzgl. Datenrate und -kapazität besser einzuordnen, werden zunächst einige Kenngrößen eingeführt und die anfallenden Datenmengen eingeschätzt.

Die Größe *L* eines einzelnen nicht komprimierten Graustufenbildes mit einer Auflösung *w x h* Bildpunkten und Luminanztiefe *bpp* (*bits per pixel*) ist:

$$L = w \cdot h \cdot bpp \qquad [Bit]$$

Wird eine Bildsequenz mit *U* Bildern in einem normierten Zeitintervall *t=1* Sekunde betrachtet, gilt für *Bitrate R*:

$$R = L \cdot U \qquad \left[\frac{Bit}{s}\right]$$

Weiterhin wird *Datengröße S* einer Bildsequenz mit *N* Bildern benötigt:

$$S = N \cdot L \quad [Bit]$$

Alternativ kann die Datengröße für eine Bildsequenz mit Bildrate R und Dauer T [s] angegeben werden:

$$S = L \cdot R \cdot T \quad [Bit]$$

Anwendung des Encoders auf Bild der Größe L liefert einen komprimierten Datenblock der Größe C. Als Kennzahl der Kompressionsstärke wird für Graustufenbilder oft die mittlere Anzahl der erzeugten Bits *cbpp* pro Eingabepixel (8 Bit) verwendet. Den Kehrwert von *cbpp* definieren wir als *Kompressionsfaktor E*:

$$E = \frac{8}{cbpp} \quad [1]$$

Sowohl *cbpp* als auch Kompressionsfaktor sind einheitslos.

Jedes erzeugte Codewort kann zwischen 1 und 16 Bit lang sein (siehe Kapitel 2). Entsprechend kann Kompressionsfaktor Werte im Bereich $\frac{8}{16} \leq E \leq \frac{8}{1}$ annehmen. Mit E kann die Datengröße S_E und Bitrate R_E für *komprimierte* Bildsequenz angegeben werden:

$$S_E = \left(\frac{L}{E}\right) \cdot R \cdot T \quad [Bit]$$

$$R_E = \frac{R}{E} \quad \left[\frac{Bit}{s}\right]$$

Mit den obigen Formeln lässt sich für das gegebene Aufnahmesystem (3 Megapixel-Kamera, 485 Bilder pro Sekunde) eine Tabelle der anfallenden Datenmengen aufstellen. Tabelle 1 zeigt Datengrößen S_E in Megabyte aufgestellt nach Aufnahmedauer und Kompressionsstärke E. Die Zeile für $E = 1$ gibt gleichzeitig die Datengrößen S der nicht komprimierten Bilddaten. Die erste Spalte (fett umrandet) entspricht genau der komprimierten Bitraten R_E, für $E = 1$ gibt sie die Bildrate R des nicht komprimierten Datenstroms.

	Dauer (s)	1	2	5	10	30	60	300	3600
	# Bilder	485	970	2425	4850	14550	29100	145500	1746000
cbpp	E								
1	8,00	182	364	909	1.819	5.456	10.913	54.563	654.750
2	4,00	364	728	1.819	3.638	10.913	21.825	109.125	1.309.500
3	2,67	546	1.091	2.728	5.456	16.369	32.738	163.688	1.964.250
4	2,00	728	1.455	3.638	7.275	21.825	43.650	218.250	2.619.000
4,78	**1,67**	**869**	**1.739**	**4.347**	**8.694**	**26.081**	**52.162**	**260.809**	**3.129.705**
5	1,60	909	1.819	4.547	9.094	27.281	54.563	272.813	3.273.750
6	1,33	1.091	2.183	5.456	10.913	32.738	65.475	327.375	3.928.500
7	1,14	1.273	2.546	6.366	12.731	38.194	76.388	381.938	4.583.250
8	1,00	1.455	2.910	7.275	14.550	43.650	87.300	436.500	5.238.000
9	0,89	1.637	3.274	8.184	16.369	49.106	98.213	491.063	5.892.750
10	0,80	1.819	3.638	9.094	18.188	54.563	109.125	545.625	6.547.500
11	0,73	2.001	4.001	10.003	20.006	60.019	120.038	600.188	7.202.250
12	0,67	2.183	4.365	10.913	21.825	65.475	130.950	654.750	7.857.000
13	0,62	2.364	4.729	11.822	23.644	70.931	141.863	709.313	8.511.750
14	0,57	2.546	5.093	12.731	25.463	76.388	152.775	763.875	9.166.500
15	0,53	2.728	5.456	13.641	27.281	81.844	163.688	818.438	9.821.250
16	0,50	2.910	5.820	14.550	29.100	87.300	174.600	873.000	10.476.000

Tabelle 1: Datengrößen [MByte] der 3 Megapixel Aufnahmen bei einer Bildrate *R=485* [1/s] aufgestellt nach Aufnahmedauer und Kompressionsfaktor. Erste Spalte gibt gleichzeitig die Datenrate [MB/s] an.

Messungen am parallelen Encoder mit realen Aufnahmen ergaben ([2] Kapitel 6.1) für *cbpp* einen Mittelwert von 4.78. Die nachfolgenden Betrachtungen werden sich nach diesem Wert richten.

Mit den Kenntnissen über die anfallenden Datenmengen, kann nun die Eignung und voraussichtliche Auslastung der involvierten Speicher und I/O Subsysteme besser eingeschätzt werden.

2.3 I/O- und Speichersysteme

Die komprimierten Daten müssen vom Aufnahmesystem zum Zielsystem transportiert werden, dies stellt an Transportsystem bestimmte Bandbreitenanforderungen. In Tabelle 2 sind einige verfügbare (Stand 2011) Schnittstellen mit ihren Bandbreiten aufgelistet.

Die Datenraten hierfür wurden den jeweiligen Spezifikationen ohne Berücksichtigung des Protokolloverheads entnommen. Ausschlaggebend für die Datenübertragung ist aber die Nutzdatenrate. Es lässt sich dennoch leicht erkennen, welche Schnittstellen für die vorliegenden Datenraten überhaupt erst in Frage kommen - diese sind in der Tabelle hervorgehoben. Im Gegensatz zum Transfer nicht komprimierter Bilddaten, ermöglich die eingesetzte Kompression eine größere Auswahl an Schnittstellen: PCIe 1.0 x4, PCIe 2.0 x2, PCIe 3.0 x1 und 10GBit Ethernet sind erst durch Kompression einsetzbar.

	max. Datenrate PCI Express [MB/s]			max. Bildrate (unkomprimiert) [1/s]			max. Bildrate (komprimiert) [1/s]		
	PCIe 1.0/1.1	PCIe 2.0/2.1	PCIe 3.0	PCIe 1.0/1.1	PCIe 2.0/2.1	PCIe 3.0	PCIe 1.0/1.1	PCIe 2.0/2.1	PCIe 3.0
PCIe x1	250	500	985	83	167	328	139	279	550
PCIe x2	500	1.000	1.969	167	333	656	279	558	1.098
PCIe x4	1.000	2.000	3.938	333	667	1.313	558	1.116	2.197
PCIe x8	2.000	4.000	7.877	667	1.333	2.626	1.116	2.232	4.394
PCIe x16	4.000	8.000	15.754	1.333	2.667	5.251	2.232	4.463	8.789
PCIe x32	8.000	6.000	31.508	2.667	2.000	10.503	4.463	3.347	17.578
PCI 2.0		133			44			74	
PCI 32 bit 2.1		266			89			148	
PCI 64 bit 2.1		533			178			297	
PCI 2.2, 2.3, 3.0		533			178			297	
USB Low-Speed		0,1875			0,06			0,1	
USB Full-Speed		1,5			0,5			1	
USB High-Speed		60			20			33	
USB Super-Speed		625			208			349	
FireWire 100		12			4			7	
FireWire 200		25			8			14	
FireWire 400		50			17			28	
FireWire 800		100			33			56	
FireWire S3200		400			133			223	
Ethernet 10MBit/s		1,25			0,42			1	
Ethernet 100MBit/s		12,5			4			7	
Ethernet 1GBit/s		125			42			70	
Ethernet 10GBit/s		1250			417			697	
Ethernet 100GBit/s		12500			4.167			6.974	

Tabelle 2: Maximale Schnittstellen-Bandbreiten und -durchsatz jeweils für nicht komprimierten und für komprimierten Fall (cbpp=4,78). Protokolloverhead nicht berücksichtigt.

Beim Einsatz von PCI-Express oder Ethernet werden die Daten zum Zielsystem in Paketen transportiert. Dort müssen Sie vom Gerätetreiber zusammengesetzt, gepuffert und persistent gespeichert werden. Wenn das Zielsystem ein PC ist, sind in diesem Prozess das Betriebssystem, die Central Processing Unit (CPU), der Systemspeicher (RAM) und der Massenspeicher involviert. Einflussfaktoren sind dabei die Leistung der CPU, die Größe und Bandbreite des System- und Massenspeichers sowie das Zusammenwirken der Betriebssystem-Subsysteme mit der Treiberschicht.

	max. Datenrate [MB/s]			max. Bildrate (unkomprimiert) [1/s]			max. Bildrate (komprimiert) [1/s]		
	pro Modul	Dual Channel	Triple Chnannel	pro Modul	Dual Channel	Triple Chnannel	pro Modul	Dual Channel	Triple Chnannel
DDR-200	1.600	3.200		533	1.067		893	1.785	
DDR-266	2.100	4.200		700	1.400		1.172	2.343	
DDR-333	2.700	5.400		900	1.800		1.506	3.013	
DDR-400	3.200	6.400		1.067	2.133		1.785	3.570	
DDR2-400	3.200	6.400		1.067	2.133		1.785	3.570	
DDR2-533	4.200	8.400		1.400	2.800		2.343	4.686	
DDR2-667	5.300	10.600		1.767	3.533		2.957	5.914	
DDR2-800	6.400	12.800		2.133	4.267		3.570	7.141	
DDR2-1066	8.500	17.000		2.833	5.667		4.742	9.484	
DDR3-800	6.400	12.800	19.200	2.133	4.267	6.400	3.570	7.141	10.711
DDR3-1066	8.500	17.000	25.500	2.833	5.667	8.500	4.742	9.484	14.226
DDR3-1333	10.600	21.200	31.800	3.533	7.067	10.600	5.914	11.827	17.741
DDR3-1600	12.800	25.600	38.400	4.267	8.533	12.800	7.141	14.282	21.423
DDR3-1866	14.900	29.800	44.700	4.967	9.933	14.900	8.312	16.625	24.937
DDR3-2133	17.000	34.000	51.000	5.667	11.333	17.000	9.484	18.968	28.452

Abbildung 3: Durchsatz verschiedener Speicherspezifikationen.

In Abb. 3 sind Durchsatzraten einiger verfügbarer Speicherspezifikationen gelistet. Offensichtlich stellt Systemspeicheranbindung ausreichend Bandbreite für den Transfer der anfallenden Datenmengen bereit und dürfte kein limitierender Faktor werden.

Ganz anders sieht es bei der Betrachtung der Massenspeichersysteme. Die Vielfalt existierender Speichertechnologien macht eine ausführliche Betrachtung sehr schwierig. Daher wird in der nachfolgenden Tabelle nur eine grobe Zusammenfassung vorgelegt.

	Sequentielles schreiben, MB/s	Kapazität, GB	Kosteneinstufung
Konventionelle Festplatte (HDD)	30 – 190	160 – 3.000	gering
HDD RAID-0	70 – 270	320 – 72.000	mittel
Solid State Drives (SSD)	140 – 230	32 – 512	mittel
SSD RAID-0	230 – 2000 [9]	64 – 6144	sehr hoch
RAM-DISK	430 – 6400 [8]	1 – 64	hoch

Tabelle 3: Übersicht verschiedener Massenspeicherlösungen.

RAM-Disks und SSD RAID-0 Verbunde würden ausreichend Durchsatz für eine Echtzeitspeicherung bereitstellen, leiden aber unter der begrenzten Speicherkapazität und sind mit hohen Kosten verbunden. Die Speicherkapazitäten reichen für kürzere Aufnahmen bis zu einer Gesamtdauer in der Größenordnung von 1 Minute. Speichersysteme mit konventionellen Festplatten bieten dagegen eine kosteneffiziente Möglichkeit zur Speicherung einer großen Anzahl von Aufnahmen mit einer Gesamtdauer in der Größenordnung von mehreren Stunden, ihr Durchsatz reicht aber für eine Echtzeitaufnahme nicht aus.

Für eine einzelne kurze Aufnahme bis ca. 10 Sekunden Dauer wäre entweder RAM-DISK oder die Reservierung eines ausreichend großer Puffers im Systemspeicher eine geeignete Lösung. Nach der Aufnahme muss in diesem Fall entschieden werden, ob die Aufnahme übernommen und auf größere Massendatenträger kopiert oder verworfen wird.

Für längere Aufnahmen wäre z.B. eine Hintereinanderschaltung verschiedener Speicherstufen denkbar. Die reservierten Speicherbereiche garantieren dabei den erforderlichen Durchsatz und dienen lediglich als Zwischenpuffer. Gleichzeitig werden dort gepufferten Daten auf schnelle und für eine Langzeitaufnahme ausreichend dimensionierte Arbeits-Massenspeicher weggeschrieben. Die Arbeits-Massenspeicher können dabei als großer persistenter Langzeit-Cache angesehen werden. Nach der Aufnahme können die aufgenommenen Daten auf langsamere aber deutlich größere Archiv-Datenträger verschoben werden.

Wie man sieht, sind der Transfer und vor allem die Speicherung mit großem Aufwand verbunden. Das resultiert aus der Notwendigkeit einen gewissen Datendurchsatz bei der Speicherung einzuhalten.

Bei der Bereitstellung von Aufnahmen ist das Problem etwas entschärft. Zwar wird eine Maximierung des Decoder-Durchsatzes angestrebt, strikte Mindestdurchsatzraten müssen hier nicht eingehalten werden. Z.B. kann das menschliche Auge die hohen Bildraten einer

Hochgeschwindigkeitsaufnahme nicht wahrnehmen, eine Visualisierung in der Zeitlupe oder mit Sprüngen (*frame skipping*) wäre damit auch von langsamen Datenträgern möglich. Für analytische Verarbeitung, um z.B. die Ergebnisse einer Auswertung früher zu erhalten, wäre ein möglichst hoher Durchsatz dagegen sinnvoll.

2.4 x86 Architektur

x86 ist eine Mikroprozessor-Architektur mit langjähriger Entwicklungsgeschichte (1978: erster 8086 Prozessor von Intel). Moderne x86 Prozessoren verwenden komplexen CISC Befehlssatz und besitzen einen oder mehrere Kerne. Jeder Kern ist als mehrstufige Pipeline aufgebaut, in der sich gleich mehrere Befehle in verschiedenen Stufen der Ausführung befinden können. In jedem Prozessorkern befinden sich mehrere Arithmetisch-Logische-Recheneinheiten (ALUs), Fließkomma-Recheneinheiten (FPUs) und ein begrenzter Satz von Registern mit einer Registerbreite von 8 bis 32 Bit (x86) oder 64 Bit (x64).

Die Architektur erlebte mehrere Befehlssatz-Erweiterungen (MMX, SSE, AVX), wodurch weitere Recheneinheiten und Register hinzugekommen sind. Die Erweiterungen beinhalten überwiegen SIMD Operationen (z.B. für Vektor-Berechnungen), können aber auch die Recheneinheiten zusammenschalten um Operationen auf bis zu 256 Bit breiten Datenwörtern durchzuführen.

Die Pipeline kann Maschinenbefehle mit unterschiedlichen Durchsatz und Latenz ausführen. Tabelle 4 listet einige der für den Decoder wichtigsten Maschinenbefehle.

		Intel P4		Intel Core 2		AMD K10	
		L32	T32	L32	T32	L32	T32
add, sub	r, ri	1	2,5	1	3	1	3
and, or, xor	r, r	1	2	1	3	1	3
inc	r	1	1	1	3	1	3
neg	r	1	2	1	3	1	3
not	r	1	1,7	1	3	1	3
imul	r, ri	10	1	3	1	3	1
div	r	80	1/34	40		45	1/45
shl, shr	r, i	1	1,7	1	2	1	3
cmp, test	r, i	1	2,5			1	3
mov movzx	r, r	1	2,5	1	3	1	3
movsx	r, r	2	1,5	1	3	1	3
bswap	r	1	2	4	1	1	3
bsr	r, r	16	1/2	2	1	4	1/3

Tabelle 4: Latenz (L32) und Durchsatz (T32) einiger 32bit Maschinenbefehle am Beispiel mit 3 Mikroprozessoren. Quelle: [10]

Während der Programmabarbeitung werden die benötigten Operanden vorher in die Register geladen. Da die Speicherzugriffe auf Systemspeicher teuer sind, werden mehrstufige Cache-Anordnungen zwischengeschaltet. Typische Cache-Größen sind in der Tabelle 5 angegeben.

	Größe
L1 Cache	32KB / 32KB
L2 Cache	256KB
L3 Cache	8MB
Systemspeicher	1 – 64 GB

Tabelle 5: Typische Cache- und Systemspeicher-Größen am Beispiel der Nehalem Architektur. Quelle: [11]

Wird der Operand in L1 Cache gefunden (*cache hit*), kostet die Ladeoperation ca. 4 Taktzyklen (siehe Tabelle 6). Andernfalls (*cache miss*) werden L2 und L3 Caches mit zunehmender Latenz nacheinander abgefragt. Verhältnismäßig teuer sind schließlich die Zugriffe auf Systemspeicher.

Um teure Zugriffe auf Systemspeicher zu vermeiden, müssen die Daten entweder komplett in den Cache passen oder der Zugriff darauf mit gewisser Lokalität erfolgen. Diese Gegebenheit wird als *Cache-Lokalität* bezeichnet.

	Latenz
L1 cache hit	~4 cycles
L2 cache hit	~10 cycles
L3 cache hit, line unshared	~40 cycles
L3 cache hit, shared line in another core	~65 cycles
L3 cache hit, modified in another core	~75 cycles
Remote L3 cache	~100-300 cycles
Local DRAM	~60 ns
Remote DRAM	~100 ns

Tabelle 6: Cache-Latenz der Nehalem Architektur. Quelle: [21]

Eine wichtige Bedeutung für die Ausführung des Programmcodes auf der Prozessor-Pipeline haben Schleifen und Verzweige. Bei einem Verzweig wird das Programm entweder an der gleichen Stelle fortgesetzt oder es wird ein Sprung zu einer anderen Stelle im Programmcode ausgeführt (*conditional jump*). Bei Verzweigen wird die CPU versuchen das Sprungziel durch einen *Branch-Predictor* vorherzusagen. Im Fall einer falschen Vorhersage sind alle in der Pipeline befindlichen und teils abgearbeiteten Befehle ungültig. Diese Befehle müssen verworfen und die Pipeline neu geladen werden, was zu einem Durchsatzverlust führen wird.

2.5 CUDA Architektur

Im Gegensatz zu multicore CPUs sind GPUs hochparallele Rechenprozessoren und werden daher als *manycore* bezeichnet. Abbildung 4 verdeutlicht dieses Verhältnis: in einer GPU sind zwar einfachere Recheneinheiten (ALUs) dafür aber in einer deutlich größeren Anzahl als bei CPU untergebracht.

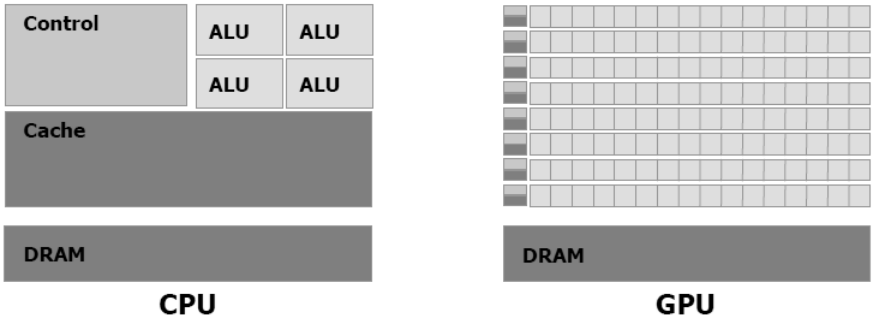


Abbildung 4: GPU verwendet mehr Transistoren für Datenverarbeitung. Quelle: [5]

Die CUDA Architektur baut auf einer flexiblen Anordnung von *Streaming Multiprozessoren* (*SM*). Der Programmcode wird in einem *Kernel* untergebracht und durch mehrere, in Blöcke organisierte, Threads auf Streaming Multiprozessoren parallel ausgeführt (siehe Abb. 5).

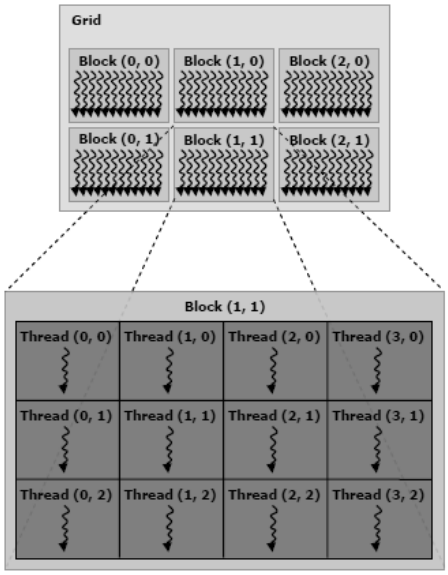


Abbildung 5: Organisation der CUDA Threads. Quelle: [5]

Zitat aus CUDA C Programming Guide ([5], S. 97):

„A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called SIMT (Single-Instruction, Multiple-Thread)… The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively with simultaneous hardware multithreading… Unlike CPU cores they are issued in order however and there is no branch prediction and no speculative execution".

Die Besonderheit der SIMT Ausführung ist die Partitionierung der Block-Threads in *Warps* durch den *warp scheduler*. Ein *warp* umfasst 32 parallele Threads und führt zu jeder Zeit für alle Threads identische Instruktion aus. Im Gegensatz zu SIMD, werden die Instruktionen nicht einfach auf unterschiedlichen Daten sondern im Kontext der unterschiedlichen Threads ausgeführt. Daraus resultiert eine sehr wichtige Eigenschaft der Programmausführung bei Zweigen und Schleifen, Zitat aus dem CUDA C Programming Guide ([5] S. 98):

„A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.“

D.h. bei Algorithmen mit vielen Verzweigen müssen mit hoher Wahrscheinlichkeit mehrere der möglichen Code-Pfade in einem Warp ausgeführt werden.

Eine weitere Besonderheit der CUDA Architektur ist die Unterscheidung verschiedener Speicherbereiche (*Memory Hierarchy*). Während es bei der CPU nur Register und einen globalen Speicherbereich gibt (Caches sind aus Programmsicht völlig transparent), differenziert man bei CUDA gleich 6 verschiedene Speicherbereiche mit völlig unterschiedlichen Eigenschaften (Tabelle 7).

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

†Cached only on devices of compute capability 2.x.

Tabelle 7: CUDA Speicherbereiche. Quelle: [15]

Vor der Kernelausführung müssen die Daten vom Systemspeicher (*Host*) zur Grafikkarte (*Device*) mittels *cudaMemcpy()* kopiert werden. Indem der Programmierer die vom Kernel verwendeten Daten in geeignete Speicherbereiche unterbringt, hat er direkten Einfluss auf Speicherzugriffslatenz.

Das Laden von einem Operand aus Register benötigt normalerweise keine extra Taktzyklen. Allerdings muss bei der Verwendung von Registern auf *read after write* Konflikte (*RAW*) geachtet werden. Zitat aus CUDA C Best Pracices Guide ([15], S. 51):

„Register dependencies arise when an instruction uses a result stored in a register written by an instruction before it. The latency on current CUDA-enabled GPUs is approximately 24 cycles… this latency can be completely hidden by the execution of threads in other warps.“

Zugriffe auf *off-chip* Speicher sind dagegen mit 400 bis 800 Taktzyklen sehr teuer. Um diese Speicherzugriffe zu „verstecken“ ist eine gewisse Mindestanzahl von *waprs* notwendig. Zitat aus CUDA C Best Pracices Guide ([15], S. 103):

„If some input operand resides in off-chip memory, the latency is much higher: 400 to 800 clock cycles. The number of warps required to keep the warp schedulers busy during such high latency periods depends on… the ratio of the number of instructions with no off-chip memory operands… to the number of instructions with off-chip memory operands“.

Damit die Zugriffe auf globalen Speicher zu einer einzigen Transaktion zusammengefasst werden können, müssen außerdem bestimmte Zugriffsmuster eingehalten werden (*coalesced access*). Es wird sich zeigen, dass der Decoder keinen Einfluss auf Speicherzugriffsmuster nehmen kann, daher wird eine ausführliche Beschreibung hierzu ([15] ab S. 30) ausgelassen.

2.6 Testsysteme

Für die Messungen der Decoder- und Anwendungsleistung werden zwei zur Verfügung stehende Tastkonfigurationen verwendet:

	Testkonfiguration 1	Testkonfiguration 2
Prozessor	Intel Core U7300	Intel Core 2 2600K
Taktfrequenz	1,3 GHz	3,92GHz
Physische Kerne	2	4
Logische Kerne	2	8
L1 Cache	32KB + 32KB	32KB + 32KB
L2 Cache	3MB	256KB je Kern
L3 Cache		8MB
Systemspeicher	4GB DDR3	4GB DDR3
Grafikkarte	NVIDIA GeForce G210M	NVIDIA GeForce GTX560Ti
GPU Cores	16	384
Compute Capability	1.2	2.1
Festplatte	320GB SATA HDD	128GB SATA SSD
Betriebssystem	Windows 7 32bit	Windows 7 64bit

Tabelle 8: Testkonfigurationen.

Die NVIDIA GeForce GTX560Ti basiert dabei auf der neusten (Stand 2011) „Fermi“ Architektur mit Compute Capability 2.1.

3 Kompressionsverfahren

Eine wissenschaftliche Analyse der aufgenommenen Bilddaten ist mit Verlust der Bildinformationen behaftet. Dies geschieht bereits während der Aufnahme, bedingt durch sehr viele Faktoren, etwa Verunreinigungen in Sensoren, zu geringer Ausleuchtung, Sampling usw. Ein weiterer Verlust an Informationen durch einen verlustbehafteten Kompressionsverfahren würde zu mehr Fehlern in der Analyse führen und soll daher mit der Wahl eines verlustlosen Verfahrens vermieden werden.

Aus einer Reihe von solchen Kompressionsverfahren wurde JPEG-LS gewählt. Es basiert auf dem von Hewlett-Packard entwickelten LOCO-I und wurde von der ISO und ITU im Jahr 1998 standardisiert [6]. Um eine effiziente Implementierung in Hardware zu ermöglichen versucht das Verfahren die Komplexität der Kodierung niedrig zu halten. Dadurch soll JPEG-LS sich besonders gut für den FPGA Hardwareencoder der gewählten Hochgeschwindigkeitskamera eignen [2].

3.1 JPEG-LS

In diesem Abschnitt werden nun die wesentlichen Bestandteile (Abb. 6) des JPEG-LS Kompressionsverfahrens beschrieben. Als Grundlage dient dabei die ITU-T Recommendation T.87 [3].

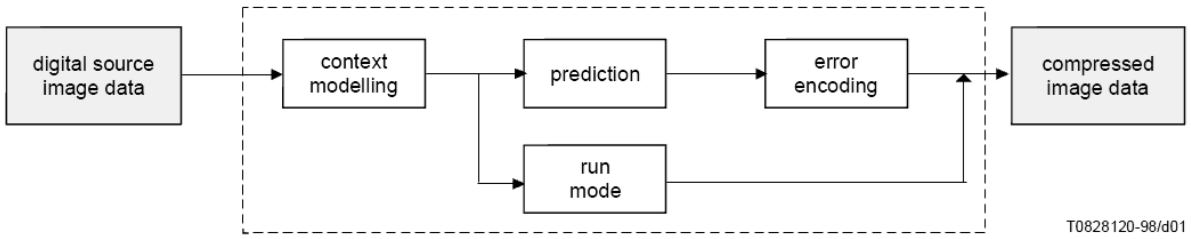


Abbildung 6: Vereinfachte Darstellung des JPEG-LS Encoders. Quelle [3]

Zunächst eine Kurzübersicht aller Kodierungsschritte:

- Kontextbestimmung (Context determination)
  - Gradienten-Berechnung (Local gradient computation)
  - Modus-Wahl (Mode selection)
  - Gradient-Quantisierung (Local gradient quantization)
  - Indexabbildung (Local gradient merging)
- Vorhersage (Prediction)
  - MED-Prädiktor (Edge-detecting predictor)
  - Vorhersage-Korrektur (Prediction correction)

    - Berechnung des Vorhersage-Fehlers (Prediction error computation)
    - Modulo-Reduktion des Fehlers (Modulo reduction of the prediction error)
    - Fehler-Abbildung auf positive Werte (Error mapping to non-negative values)
- Fehler-Kodierung (Prediction error encoding)
    - Bestimmung des Golomb-Parameters k (Golomb coding variable computation)
    - Fehler-Kodierung (Mapped-error encoding)
- Variablen-Aktualisierung (Variable update)
    - Kontextaktualisierung (Context-Update)
    - Fehlerausrichtung (Bias computation)
- Run Mode

Der Algorithmus verarbeitet das Eingabebild pixelweise entlang der *Scanline*. Bis zu dem Pixel, dass gerade betrachtet wird, sind alle vorhergehenden Pixelwerte definitiv bestimmt. Grundlegende Annahme des Algorithmus basiert darauf, dass statistisch gesehen eine starke Korrelation zwischen dem betrachteten Pixel und seinen Nachbarn bestehen muss.

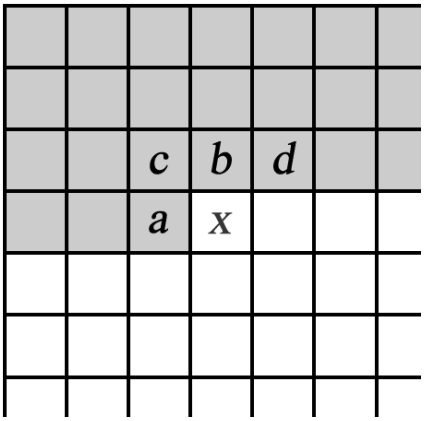


Abbildung 7: Bildausschnitt mit den betrachteten Pixeln

Siehe hierzu Abb. 7: wenn die Verarbeitung den Pixel *x* erreich, sind alle Pixelwerte oberhalb der Scanline von *x* und in seiner Scanline links von ihm definitiv bekannt. Zur Vereinfachung werden nur die definitiv bekannten direkten Nachbarn betrachtet, diese müssen die stärkste Korrelation zu x besitzen. Die Kernidee besteht darin, aus diesen Nachbarpixeln eine möglichst präzise Vorhersage für *x* zu treffen. Die definitiv bekannten direkten Nachbarn sind *c*, *b*, *d* und *a*. Als *Predictor* setzt JPEG-LS den *Median Edge Detector* (MED) ein, um aus den Pixelwerten direkter Nachbarn den Wert von *x* vorherzusagen. Der MED Prädiktor ist folgendermaßen definiert:

$$x_P = \begin{cases} \min(a,b) & wenn\ c\ \geq \max(a,b) \\ \max(a,b) & wenn\ c\ \leq \min(a,b) \\ a+b-c & sonst \end{cases}$$

Die Differenz zwischen dem tatsächlichen und vorhergesagten Wert von *x* wird als *Prediction Error* bezeichnet.

$$Err_{MED} = x - x_P$$

Dieser Wert soll betragsmäßig möglichst gering ausfallen. Eine ideale Vorhersage wird für Prediction Error daher den Wert 0 liefern.

Durch die Betrachtung von nur wenigen Pixeln, wird der Aufwand für die Vorhersage gering gehalten, allerdings hat der Prädiktor dadurch eine eingeschränkte lokale Sicht. Reale Bilder bestehen aber aus großflächigen homogenen Regionen und Mustern – Informationen, welche vom Prädiktor nicht berücksichtigt werden. Um die Vorhersagen des Prädiktors zu verbessern, wird versucht mittels *Context Modelling* diese Information in die Berechnung einfließen zu lassen. Der Kontext, in dem ein Pixel vorkommt, wird durch das umgebende Bildmuster bestimmt, dabei sind nicht die absoluten Werte der Bildpunkte interessant, sondern deren Veränderungen. Dafür werden aus den vier Nachbarpixeln drei Gradienten bestimmt:

$$Q_1 = d - b$$

$$Q_2 = b - c$$

$$Q_3 = c - a$$

Die Gradienten dienen direkt zur Identifikation des Verlauf-Musters, in welchem der Bildpunkt (*x*) vorkommt. Der Wertebereich eines Bildpunktes in einem 8bpp Graustufenbild ist [0…255]. Damit kann ein Gradient 511 mögliche Werte im Bereich [-255…255] annehmen. Mit 3 Gradienten wären $511^3 = 133.432.831$ Kontexte möglich. Um Anzahl der Kontexte zu reduzieren, wird Wertebereich der Gradienten durch *Quantisierung* von [-255…255] auf [-4…4] reduziert. Dies reduziert die Anzahl der Kontexte zunächst auf $9^3 = 729$. Die drei quantisierten Gradienten werden auf einen Indexwert abgebildet, welcher den Kontext eindeutig identifiziert. Diese Abbildung ist im JPEG-LS Standard nicht spezifiziert, es wird lediglich verlangt, dass die Abbildung eindeutig und umkehrbar ist. Betrachtet man zwei beliebige Kontexte, in denen alle Gradienten paarweise betragsmäßig gleich jedoch entgegengerichtet sind (unterschiedliche Vorzeichen), lässt sich beobachten, dass die Fehler in diesen Kontexten ebenfalls betragsmäßig gleich aber mit entgegengesetzten Vorzeichen vorkommen. Diese Eigenschaft wird dazu genutzt das Vorzeichen aus dem Kontext zu entnehmen, wodurch sich die Anzahl möglicher Indizes für einen Kontext auf insgesamt 365 halbiert. Die Berechnung des Kontext-Index *Q* und des Vorzeichens *Sign* wird *Context Determination* genannt und könnte folgendermaßen aussehen:

$$Q = |81 \cdot Q_1 + 9 \cdot Q_2 + Q_3|$$

$$Sign = sign(81 \cdot Q_1 + 9 \cdot Q_2 + Q_3)$$

Durch Context Modelling wird versucht das Verhalten des Vorhersagefehlers Err_{MED} während der Scanline-Verarbeitung zu verfolgen, und mit Hilfe dieser Information die

Vorhersage kontinuierlich zu verbessern. Als Maß für das Fehlerverhalten dient dabei der Mittelwert des Fehlers Err_{MED}, welcher für jeden Kontext separat bestimmt wird. Um diesen Wert zu bestimmen, muss die Anzahl der vorgekommenen Bildpunkte (*N*) sowie die Summe aller Vorhersagefehler Err_{MED} (S_E) für jeden Kontext akkumuliert werden:

$$S_E = \sum_{i=1}^{N} Err_{MED,i}$$

Der Mittelwert der Fehler Err_{MED} lässt sich daraus ganz einfach berechnen:

$$C = {}^{S_E}/_{N} \text{ mit } C \in \mathbb{R}$$

Um die teure Division zu vermeiden, wird diese durch Multiplikation in Integer-Arithmetik ersetzt:

$$S_E = C \cdot N + B \text{ mit } C \in \mathbb{N}$$

Statt S_E und *N* werden in jedem Kontext die Werte *C*, *N* und *B* geführt.

Der Fehler-Mittelwert Err_{MED} wird als *Correction Factor* (*C*) zusammen mit dem Vorzeichen *Sign* aus Correction Determination Schritt dazu verwenden, die Vorhersage des Prädiktors zu korrigieren.

$$X_{PC} = X_P + Sign \cdot C$$

Anschließend wird durch einfaches Abschneiden (engl. *clamping*) sichergestellt, dass der Wertebereich von X_{PC} im Bereich [0…255] liegt. Diese Schritte zusammengefasst werden *Adaptive Correction* genannt.

Jetzt kann der Fehler zwischen dem korrigierten vorhergesagten Wert und dem tatsächlichen Pixelwert berechnet werden:

$$Err = X - X_{PC}$$

Der Wertebereich für X und X_{PC} ist jeweils [0…255]. Damit bestimmt sich der Wertebereich von *Err* zu [-255…255]. Sowohl der Encoder als auch der Decoder arbeiten auf den gleichen Eingabepixeln und berechnen die gleichen Kontexte, daher werden beide dieselbe Vorhersage X_{PC} treffen. Der mögliche Wertebereich des Fehlers unterliegt damit folgender Ungleichung:

$$0 - X_{PC} \leq Err \leq 255 - X_{PC}$$

Err kann also für jedes gegebenes X_{PC} nur 256 mögliche Werte annehmen. X_{PC} gibt lediglich eine Verschiebung des Wertebereichs an. Der Fehler wird mittels *Modulo Reduction* in einen fixierten Wertebereich [-128…127] gebracht.

$$Err = \begin{cases} Err + 256 & falls\ Err < -128 \\ Err - 256 & falls\ Err > 127 \end{cases}$$

Die Fehlerwahrscheinlichkeit für reale fotographische Aufnahmen folgt dabei einer zweiseitigen Geometrischen Verteilung, dies ist ausgezeichnete Voraussetzung für die Datenreduktion mittels Entropiekodierung. JPEG-LS setzt hierzu den Golomb-Coder ein. Es kann sogar nachgewiesen werden, dass Golomb-Code für geometrisch verteilte Datenquellen optimal ist [7]. Der Golomb-Coder setzt allerdings einen Wertebereich mit positiven natürlichen Zahlen voraus, daher wird im *Error Mapping* Schritt der Wertebereich des Fehlers zunächst auf einen positiven Wertebereich [0…255] abgebildet. Die Verteilung vor und nach Error Mapping ist in der Abb. 8 dargestellt.

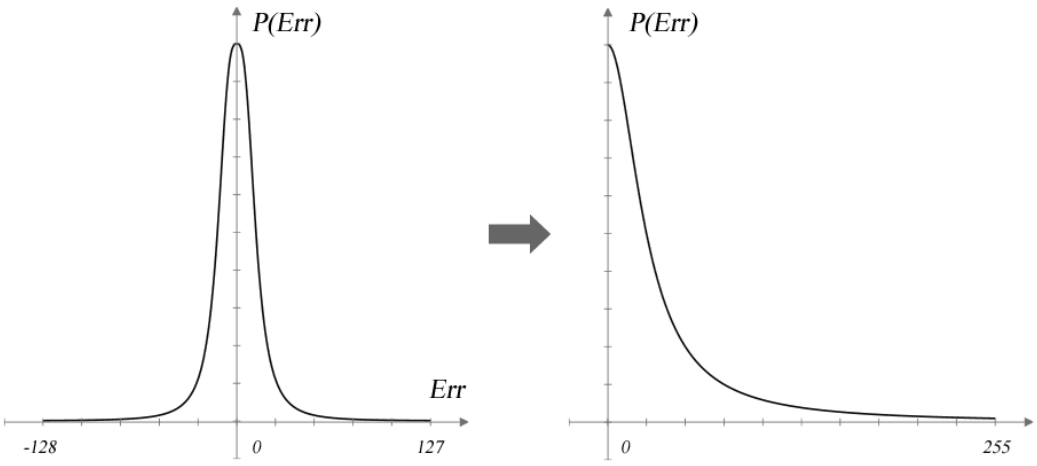


Abbildung 8 Zweiseitige geometrische Fehlerverteilung (links) abgebildet auf einseitige geometrische Verteilung (rechts).

Error Mapping bildet positive bzw. negative *Err* Werte auf gerade bzw. ungerade Zahlen ab:

$$MErr = \begin{cases} 2 \cdot Err & falls\ 0 \leq Err \\ (-2 \cdot Err) - 1 & sonst \end{cases}$$

Der Golomb-Coder zerlegt *MErr* in zwei Komponenten:

$$MErr = Q_d \cdot 2^k + R$$

Q_d ist dabei Quotient der Division von *MErr* durch 2^k, *R* - der Rest dieser Division. Anschließend wird Q_d mittels *Unary Coding* kodiert: Q_d „0“-Bits gefolgt von einer „1“. Daran wird die zweite Komponente angehängt: die *k* Bits von *R*. Das so entstehende Codewort hat variable Bit-Länge $L = Q_d + 1 + k$ Bit. Entscheidend dabei ist die Wahl des

Parameters k. Der Parameter k wird sowohl vom Encoder als auch vom Decoder aus dem Kontext berechnet und ist wegen den gleichen Kontexten für beide identisch. Dafür werden im Kontext zusätzlich die Absolut-Werte des Fehlers (*A*) aufsummiert. Die Berechnung des Parameters k erfolgt dann durch folgende Formel:

$$k = max\left(0, \left\lceil log_2\left(\frac{A}{N}\right)\right\rceil\right)$$

Wenn ein großer Fehler vorliegt und *k* zu klein gewählten ist (z.B. *k=0*), wird der Quotient Q_d entsprechend groß, was in einer langen „0"-Bit Sequenz resultiert. Eine Modifikation des Golomb-Coders beschränkt deshalb die maximale Länge des Codeworts: Erreicht der Quotient Q_d einen voreingestellten Limit *QMAX*, dann wird statt der regulären eine Sondersequenz erzeugt: Q_d wird auf QMAX gesetzt und unär kodiert, gefolgt von *qbpp* Bit langen Wert von $MErr - 1$. Für Graustufenbilder hat *qbpp* den Wert 8. Die Länge des Codeworts (und damit auch die maximale Länge) berechnet sich in diesem Fall zu:

$$L_{max} = QMAX + 1 + qbpp \quad [Bit]$$

Abschließend führt JPEG-LS nach jedem verarbeiteten Bildpunkt einen *Context Update* durch. Darin wird die Anzahl der betrachteten Bildpunkten (*N*) gezählt, der Fehler (ausgedrückt durch Correction Factor *C* und Restbetrag *B*) gemittelt, sowie Beträge des Fehlers für die Parameter k Berechnung aufsummiert (*A*). Folgender Pseudocode beschreibt die beiden Schritte für *Update* und *Bias Correction*:

Update

```
B ← B + Err
A ← A + abs(Err)
if ( N == RESET )
{
    A = A / 2
    if ( B ≥ 0)
        B = B / 2
    else
        B = - ((1 – B) / 2)
    N = N / 2
}
N = N + 1
```

Bias Correction

```
if ( B ≤ -N)
    B = B + N
    if ( C > MIN_C)
        C = C – 1
    if (B ≤ -N)
        B = -N + 1
else if ( B > 0)
    B = B - N
    if ( C < MAX_C)
        C = C + 1
    if (B > 0)
        B = 0
```

Akkumulation der Kontextparameter würde deren Wertebereiche abhängig von der Bildgröße nach oben treiben. Zudem verlieren die Einflüsse einzelner Bildpunkte zunehmend an Relevanz, je weiter sich die Scannline von diesen Bildpunkten entfernt. Daher werden die entsprechenden Kontext-Parameter regelmäßig halbiert, und zwar dann, wenn die Anzahl der

Bildpunkte *N* einen Schwellwert *RESET* (Wert 64 als Voreinstellung im JPEG-LS Standard) erreicht.

JPEG-LS besitzt zusätzlich einen Speziellen Kodierungs-Modus: *Run-Length Coding* (RLC). Wenn alle drei Gradienten Null sind (0, 0, 0), erkennt der Encoder einen homogenen Bildbereich und es wird in den *Run Mode* gewechselt. Dieser Modus wird hier nicht näher betrachtet.

Die Decoder-Schritte unterscheiden sich von Encoder-Schritten hauptsächlich nur durch inverse Abbildungen zu der Fehler-Kodierung (Golomb-Dekodierung) sowie zur Fehler-Abbildung auf positive Werte (Inverse Error Mapping).

3.2 JLX: JPEG-LS Modifikation

Obwohl JPEG-LS als ein Kompressionsformat niedriger Komplexität angesehen wird [6], muss es im Kontext der Hochgeschwindigkeitsaufnahmen noch einmal genauer analysiert werden. In der Arbeit von Anto J.Y.M.: „Efficient Context Modelling and Segmentation for Parallel Lossless Image Compression„ [2] wurde der ursprüngliche Standard darauf untersucht, inwieweit es sich für eine Encoder-Implementierung in FPGA Hardware eignet. Um den Anforderungen an Ressourceneffizienz und Zeitvorgaben für vorgegebene Bildraten zu genügen, wurde eine Reihe von Modifikationen am JPEG-LS Encoder eingeführt. Messungen mit realen Testdaten haben belegt, dass diese Modifikationen zu einer signifikanten Verbesserung der gewünschten Encoder-Eigenschaften bei einem vernachlässigbaren Verlust in der Kompressionsstärke führten.

Für weitere Betrachtungen sei diese modifizierte Codec-Variante als *JLX* referenziert.

Die erste Modifikation betrifft *Run Length* Modus. Während bei synthetischen Bildern größere homogene Regionen denkbar sind (etwa weißes Hintergrund bei Textgrafiken), kommen solche in photographischen Bildern vernachlässigbar selten bis gar nicht vor. Der Run Length Modus wurde aus dem Encoder daher komplett entfernt.

	Original	Modifiziert
MIN_C	-128	-32
MAX_C	127	31
T1	3	3
T2	7	7
T3	21	21
RESET	64	64
LIMIT	32	16
QMAX	23	7
qbpp	8	8
MAXVAL	255	255
RANGE	256	256
Startwert für A	4	256
Startwert für N	1	56

Tabelle 9: Parametereinstellungen des JLX Encoders.

Der ursprüngliche JPEG-LS Standard enthält eine Reihe von einstellbaren Konfigurationsparametern und definiert Marker, mit welchen diese Parameter während der Scannline-Verarbeitung an beliebigen stellen verändert werden können. Dies ermöglicht

das Einsetzen komplexerer Encoder-Algorithmen, welche durch aufwendigere Analyseverfahren bessere Parametereinstellungen finden und dadurch die Kompressionsstärke verbessern könnten. Wegen der begrenzten Ressourcen der Hardwareplattform, wurde auf dieser Option verzichtet. Die Konfigurationsparameter werden auf die, bei Messungen als optimal bestimmte, Werte fest voreingestellt (Tabelle 9).

Um den Platzbedarf für Kontextspeicher zu reduzieren wurde die Berechnung einiger Kontextparameter angepasst. Der Parameter N wurde auf konstanten Wert 56 gesetzt (Einzelheiten siehe [2], Abschnitte 3.1.3, 3.2.1 und 3.3.1). Die dadurch erreichten Verbesserungen im Kontext-Speicherplatzbedarf sind in der Abb. 9 dargestellt.

Parameter	Original		Modified	
	Range	Number of Bits	Range	Number of Bits
N	[1, 64]	6	-	0
A	[0, 8192]	14	[15, 2000]	11
B	[-63, 0]	6	[-55, 0]	6
C	[-128, 127]	8	[-32, 31]	6
Total	-	**34**	-	**23**

Abbildung 9: Kontextspeicherbedarf (JPEG-LS und JLX) Quelle: [2]

Um den Ressourcenbedarf (Registerbreite, Multiplexerbreite) für die Speicherungen der komprimierten Bitströme zu reduzieren, wurde zudem die maximale Länge des golomb-kodierten Codeworts von 32 auf 16 Bit beschränkt ([2] Abschnitt 5.2.5).

Betrachtet man die grundlegende Funktionsweise von JPEG-LS, wird schnell ersichtlich, dass die Kodierung eines Bildpunktes im Schritt *i* als Eingabe die Kontexte und Pixelwerte aus dem vorherigen Schritt *i-1* benötigt. Somit stellt die Implementierung des JPEG-LS zunächst ein rein *sequentielles Problem* dar.

Die wichtigste Neuerung im JLX betrifft einen Ansatz zur Parallelisierung des Encoders. Es wird zunächst versucht mittelst Bildsegmentierung Daten-Parallelität herzustellen. Dafür wurde die Möglichkeit untersucht, das zu kodierende Bild in unabhängige *Segmente* zu unterteilen. Die Segmentierung kann prinzipiell in vertikale oder horizontale Blöcke erfolgen. Jedes Segment kann dann *unabhängig* von allen anderen verarbeitet werden. Mit der Unabhängigkeit ist gemeint,

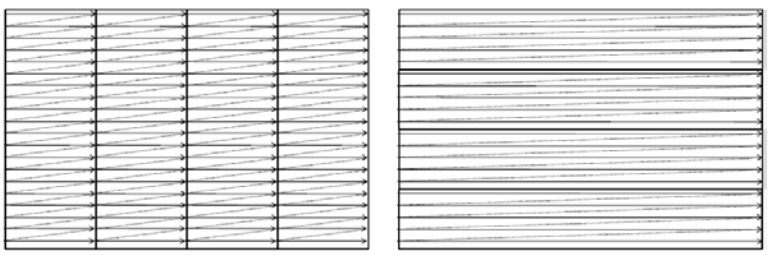


Abbildung 10: Mögliche Segmentierung: vertikale Blöcke (links), horizontale Blöcke (rechts). Quelle: [2]

dass jedes Segment einen individuellen Variablen-Satz (einschl. Kontext-Liste) besitzt, und jede auf einem Segment arbeitende Decoder-Einheit ausschließlich den Variablen-Satz des zugewiesenen Segments verwendet. Zwischen den Kodierungsströmen verschiedener Segmente findet kein Informationsaustausch statt.

Messungen ergaben ([2] Abschnitt 4.3.3), dass vertikale Segmentierung durchgehend besseren Kompressionsfaktor lieferte, weshalb sie als Basis für den Hardwareencoder gewählt wurde. Die vorgeschlagene VHDL Implementierung des Encoders instanziiert für jedes Segment eine dedizierte Kodierungseinheit im FPGA. Alle Encoder arbeiten parallel und liefern an ihren Ausgängen gleichzeitig je ein Codewort. Eine hierarchisch geschaltete Multiplexer-Schaltung in Form eines Baumes, fügt die Codewörter variabler Bit-Länge zusammen und legt sie in einen FIFO Puffer nacheinander ab. Die so zusammengefügten Codewörter liegen im Puffer verschränkt vor und werden über die externe Schnittstelle (Gigabit Ethernet) des FPGA Boards zur Speicherung transferiert. Das verschränkte Layout der Codewörter wurde bereits in der Abb. 2 gezeigt. Dieses Layout wird bei Parallelisierung, wie wir es später sehen, eine entscheidende Rolle spielen.

Weiterhin hat die Wahl der Segmentanzahl eine wichtige Bedeutung. Zu wenige Segmente schränken den erreichbaren Parallelisierungsgrad und damit den erreichbaren Durchsatz ein. Andererseits steigt mit Segmentanzahl auch der schaltungstechnische Ressourcenbedarf. Ferner wurde mit steigender Segmentzahl zunehmender Entropieverlust festgestellt. ([2] Abb. 4.6). JPEG-LS verwendet Schätzwerte für Nachbarpixel außerhalb des Bildabschnittes. Mit steigender Segmentzahl nimmt die Zahl der „inneren“ Bildpunkten schneller ab, als die Zahl der geschätzten „äußeren“ - damit wird die Vorhersage zunehmend mehr von geschätzten Pixelwerten verfälscht. Gleichzeitig hängt vom Segmentierungsgrad der Informationsgehalt der Kontexte ab. Mit kleiner werdenden Ausschnitt verbessert sich zunächst die Lokalität der Kontextinformation, weshalb die Messungen eine Verbesserung der Entropie am Anfang zeigen. Doch ab einem bestimmten Segmentierungsgrad und kleiner werdender Breite, nehmen die Lokalität und damit die Entropie wieder ab. Im eingesetzten Hardware-Encoder wurde 8-fache Segmentierung gewählt.

In nachfolgenden Ausarbeitungen zur Optimierung und Parallelisierung des JLX Decoders müssen die eingeführten Modifikationen, insbesondere die Verschränkung und der Segmentierungsgrad als gegebene Vorgaben angesehen werden. Sowohl Decoder-Implementierung als auch alle Messungen werden sich nach diesen richten.

4 Optimierung

Um später maximalen Durchsatz durch Parallelisierung erreichen zu können, muss vorher ein geeignetes Dateiformat geschaffen und der (sequentielle) Decoder für die vorliegende Hardware optimiert werden. In diesem Abschnitt wird eine Datenstruktur zur effizienten direkten Adressierung (*random access*) der Datenblöcke eingeführt. Um die Hardware-Ressourcen besser auszulasten und damit höheren Durchsatz zu erzielen, werden anschließend verschiedene Decoder-Optimierungen untersucht.

4.1 Dateiformat

Die Bereitstellung von Hochgeschwindigkeitsaufnahmen beginnt mit dem Laden der Blöcke von Datenträgern in den Systemspeicher. Dieser, als *Disk I/O* bezeichnete, Schritt ist, neben dem eigentlichen Decoder, einer der wichtigsten Bestandteile im gesamten Bereitstellungsprozess. Die verfügbaren Bandbreiten der I/O Systeme sind gegeben, darauf hat man i.d.R. keinen Einfluss, auf die Datenstruktur dagegen schon.

Die Daten erhalten ihre Struktur bereits während der Echtzeitspeicherung einer Aufnahme durch den Kamera-Gerätetreiber. Der vom Treiber empfangene komprimierte Datenstrom, weist ein besonderes Layout, in dem die einzelnen Codewörter variabler Bit-Länge ineinander verschränkt sind (Abb. 2). Diese Verschränkung koppelt die parallelen Datenströme der Bild-Segmente wieder zusammen und wird, wie wir es in Kapitel 5 sehen werden, die Parallelisierung beeinträchtigen. Um den verschränkten Datenstrom wieder in einzelne Segment-Ströme zu trennen, muss die Länge jedes einzelnen Codeworts bestimmt werden. Dafür müssen für das gesamte Bild alle Decoder-Schritte ausgeführt werden. Es wird sich herausstellen, dass der, für die Dekodierung und Aufteilung der Datenströme benötigter, Durchsatz auf der vorliegenden Hardware nicht erreicht wird.

Der Treiber wird also keine Echtzeitdekodierung während der Speicherung durchführen können. Die ankommenden Datenströme müssen ohne rechenintensive Verarbeitung vom Treiber direkt auf den Datenträger geschrieben werden.

Damit beim Laden der gespeicherten Daten direkter und effizienter Zugriff auf beliebige Bilder innerhalb der Bildsequenz möglich ist, braucht man für jeden einzelnen Block eine eindeutige Abgrenzung. Mechanismen zu dieser Abgrenzung müssen bereits im Kommunikationsprotokoll zwischen dem FPGA Board und dem Treiber implementiert sein.

Um später eine teure Transkodierung zu vermeiden, muss ein Dateiformat geschaffen werden, welcher sowohl effizienten Zugriff beim Lesen ermöglicht als auch den hohen Bandbreitenanforderungen während der Echtzeitspeicherung genügt.

Beim Dateiformat stehen zunächst zwei prinzipiell verschiedene Optionen zur Auswahl:

- Eine (durchnummerierte) Dateisequenz mit je einer Datei pro Bild.
- Eine einzelne Datei beinhaltet alle Datenblöcke einer Aufnahme.

Um die Wahl zu begründen, wurden unterschiedlich parametrisierte Schreibmessungen auf beiden Testsystemen durchgeführt (Tabelle 10).

Messung		Testkonfiguration 1	Testkonfiguration 2
M1	Erzeugen von 10.000 Dateien (ohne Daten)	1000-2000 Dateien/s - MB/s	4400-10000 Dateien/s - MB/s
M2	10.000 Dateien je 1KB	400-1600 Dateien/s 0,4-1,6 MB/s	3500-6500 Dataien/s 3,3-6,4 MB/s
M3	1.000 Dateien je 1MB	35 Dateien/s 35 MB/s	330 Dataien/s 330 MB/s
M4	1 Datei, 1.000.000 Speichervorgänge je 1KB	17-20 MB/s	290MB/s
M5	1 Datei, 1000 Speichervorgänge je 1MB	67 MB/s	370MB/s

Tabelle 10: Messungen des Dateisystems (Schreiboperationen).

Messungen M1 und M2 zeigen, dass alleine schon die Erzeugung von neuen Dateien mit gewissem Mehraufwand verbunden ist. Aus Messungen M3 bis M5 folgt, dass ein hoher Durchsatz bei der Speicherung großer Datenmengen mit einer einzelnen Datei zu erreichen ist. Dadurch wird der Overhead für die Erzeugung von 485 Dateien/s reduziert. Wenige Speichervorgänge mit größeren Datenblöcken reduzieren zudem den Overhead für die Schreiboperationen (*fwrite)* und führen schließlich zum optimalen I/O-Durchsatz.

Als Nachteil bei der Verwendung einer einzelnen Datei kann die Größenbeschränkung einiger Dateisysteme angesehen werden. So kann z.B. bei FAT32 eine einzelne Datei maximal 4 GB groß sein. Dieses Problem ist bei neueren Dateisystemen (z.B. NTFS: 16 TB) vernachlässigbar. Um unter 32bit Anwendungen Dateien jenseits der 4GB Grenze zu adressieren, müssen unter C++ z.B. spezielle Funktionen _fseeki64 und _ftelli64 auf __int64 Datentypen verwendet werden.

Für JLX lässt sich nun ein Dateiformat definieren, das auf einer einzelnen Datei basiert. Die Dateierweiterung wird auf *.jlx* festgelegt. Damit die Eigenschaften einer Aufnahme (Auflösung, Bildanzahl etc.) dem Decoder bekannt werden, werden diese Informationen im Dateikopf (engl. *Header*) gespeichert. Ein möglicher Header ist in der Tabelle 11 definiert: dieser beinhaltet eine Identifikation-Zeichenkette und die wichtigsten Parameter einer Bildsequenz.

Offset [Bytes]	Länge [Bytes]	Wertebereich	Parameterbezeichnung	Beschreibung
0	4	-	Identifikation	Vorgabe: „JLX1“
4	2	0 - 65535	Breite	Horizontale Auflösung der Bilddaten
6	2	0 - 65535	Höhe	Vertikale Auflösung der Bilddaten
8	4	0 - 4294967296	Bildanzahl	Anzahl Bilder in der Bildsequenz
12	1	0-255	Segmentanzahl	Anzahl Segmente in welche jedes Bild unterteilt wird

Tabelle 11: Headerdefinition des JLX-Formats.

Das definierte Header hat eine konstante Länge von 13 Bytes. Breite, Höhe und Bildanzahl werden im *Little Endian* Format gespeichert (niederwertiger Byte zuerst). Hiermit ergibt sich für JLX Aufnahme eine maximale Bildauflösung von 65535 x 65535 Bildpunkten und theoretische Aufnahmelänge bis zu 4 Milliarden Bildern. Bilder können dabei in bis zu 255 Segmente aufgeteilt werden.

Sofort nach dem Header werden die komprimierten Bilddatenblöcke variabler Länge abgelegt. Beim Auslesen eines zufälligen Datenblocks müsste die komplette Dekodierung aller davor stehender Blöcke durchgeführt werden. Um z.B. das letzte Block zu erhalten, muss die gesamte Bildsequenz dekodiert werden, Für direkte Block-Adressierung wird daher eine unterstützende Längenangabe eingeführt (Abb. 11).

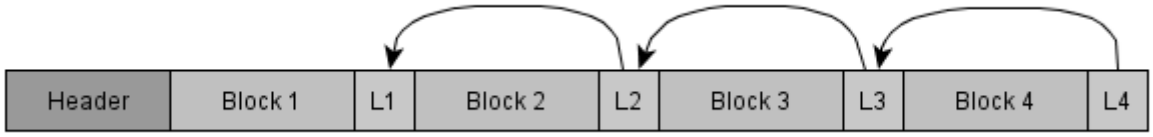


Abbildung 11: Blockspeicherung im JLX Format am Beispiel mit 4 Blöcken.

Nach jedem vollständigen Datenblock eines Bildes wird sofort ein 32 Bit Feld mit der Längenangabe (*L1*, *L2*, usw.) in Bytes angehängt. Die Längenangabe ist vorzeichenlos und ebenfalls *Little Endian* kodiert. Die maximale mögliche Länge eines Blocks berechnet sich zu $2^{32} - 1 = 4.294.967.295$ Bytes und ist selbst für sehr hochauflösende Bilder ausreichend.

Zu Beginn initialisiert der Treiber die Bildanzahl im Header auf 0 und verwendet während der Aufnahme einen Zähler, wo die Anzahl der gespeicherten Bilder akkumuliert wird. Erst nach Beendigung der Aufnahme wird die Anzahl der tatsächlich korrekt gespeicherten Bilder in den Header geschrieben. Dies soll den Decoder vor dem Einlesen inkonsistenter Daten schützen, welche z.B. beim plötzlichen Abbruch der der Aufnahme entstehen könnten. Denn, um maximale Leistung zu erzielen, wird der Decoder auf Schutzmechanismen, wie

Überprüfung auf Pufferüberläufe, verzichten. Während der Ausführung des Decoders auf inkonsistenten (beschädigten) Daten wurde unerwünschtes Verhalten beobachtet: die Bilder waren entweder teilweise beschädigt oder es kam zu Pufferüberläufen mit anschließenden Programmabsturz.

Beim Öffnen der JLX Datei wird zuerst der Header eingelesen. Mit Hilfe der Header-Angaben können für die Dekodierung benötigte Puffer im Systemspeicher dimensioniert werden. Pro Decoder-Thread werden ein Block-Puffer und ein Bild-Puffer benötigt.

Mit der Bildanzahl im entsprechenden Header-Feld und den Längenangaben kann eine Katalog-Tabelle (*Block-Index* genannt) für direkte Blockadressierung aufgebaut werden. Die Bildanzahl liefert sofort die Anzahl der benötigten Tabellen-Einträge. Die Anwendung springt zum Dateiende und liest das Längenfeld des letzten Blocks ein. Das Abziehen der Blocklänge und der 4 Byte des Längenfelds vom aktuellen Offset des Dateizeigers liefert Offset des Längenfeldes für das davorstehende Block. Beim iterativen Rückwärtseinlesen der Block-Offsets werden die Offset-Adressen (64 Bit) und Längen (32 Bit) der entsprechenden Blöcke ebenfalls rückwärts in die Katalog-Tabelle geschrieben. Am Ende der Katalogaufbau muss die Anzahl eingelesener Blöcke genau der Bildanzahl im Header entsprechen und der Dateizeiger muss auf das zehnte Byte im Header zeigen, andernfalls liegt ein Fehler vor. Block-Daten müssen bei Katalogaufbau zunächst noch nicht eingelesen werden, der Aufbau kostet damit *N* Leseoperationen von jeweils 4 Bytes für eine Blocksequenz mit *N* Blöcken (bzw. Bildern).

Das Anfügen der Längenangaben erst am Ende eines Blocks ermöglicht dem Treiber die Block-Datenströme häppchenweise in die Datei zu schreiben. Wenn der Treiber einen neuen Block anfängt, ist die Blocklänge möglicherweise noch nicht bekannt. Spätestens am Ende des Blocks ist die Länge dann bekannt und kann angehängt werden. Im Gegensatz zu Längenangabe vor dem Block, wird hier kein Dateizeigersprung benötigt. Das führt zu einer Serialisierung der Schreiboperationen und damit zu Overheadreduktion.

Damit ermöglicht Katalog-Tabelle der Anwendung eine direkte Adressierung jedes Blocks innerhalb der Blocksequenz. Wird ein Datenblock für ein bestimmtes Bild benötigt, liefert der dazugehörige Eintrag in der Katalog-Tabelle sowohl Position (Offset) des Blocks innerhalb der Datei als auch seine Länge. Das Einlesen eines Blocks kann somit mit einem einzigen Aufruf von *fread* erfolgen. Mit dieser Eigenschaft kann die Anwendung schnell und effizient beliebige Bilder innerhalb der Bildsequenz anfordern.

Jeder Eintrag der Katalog-Tabelle belegt 12 Byte. Der Katalog-Speicherplatzbedarf für Aufnahmen mit N Bildern berechnet sich zu $12 \cdot N$ Byte. Das sind umgerechnet 5,8 Kilobyte für eine Sekunde Aufnahme – ein vertretbarer Aufwand.

An dieser Stelle sei angemerkt, dass das vorgestellte Container-Format nur ein sehr einfaches Modell ist, komplexere und effizientere Ausführungen sind durchaus denkbar. Dieser Abschnitt soll nur verdeutlichen, dass die Wahl der Dateistruktur einen großen Einfluss auf die Eigenschaften und Effizienz der Bereitstellung von Hochgeschwindigkeitsaufnahmen haben kann.

4.2 JLX Optimierung für x86 Architektur

Das angestrebte Ziel dieser Arbeit ist die Durchsatz-Maximierung bei der Bereitstellung von Hochgeschwindigkeitsaufnahmen, der Schwerpunkt wird dabei ganz klar auf Decoder gelegt. Die gesamte Bereitstellung lässt sich als eine Pipeline mit folgenden Verarbeitungsstufen auffassen:

- Einlesen der komprimierten Datenblöcke von Massenspeicher in den Systemspeicher
- Dekompression mittels CPU oder GPU
- Verarbeitung der Bilddaten (Rauschfilter, Particle-Tracing, etc.)
- Visualisierung für den Benutzer (*Rendering*)

In beiden nachfolgenden Abschnitten wird nur der Decoder betrachtet. Ferner wird angenommen, dass die Datenblöcke bereits im Systemspeicher vorliegen.

Parallelisierung ist ein vielversprechender Ansatz um die Leistung des Decoders zu steigern. Eine signifikante Leistungssteigerung kann aber bereits mit der Optimierung des zunächst sequentiellen Decoders für die vorliegende Hardware beginnen. Den Ausgangspunkt stellt eine Referenzimplementierung des JLX Decoders für x86 Architektur. Diese, nach JPEG-LS Spezifikationen [3] entworfene und zum JLX modifizierte, Implementierung enthält keine besonderen leistungssteigernden Maßnahmen und dient als Bezugspunkt für Leistungsbewertung der vorgestellten Optimierungen.

Auch wenn sich manche Konzepte auf andere Architekturen übertragen lassen, werden die eingeführten Optimierungen hiermit als *implementierungsspezifisch* für die betrachtete Architektur erklärt.

4.2.1 Der sequentielle Decoder im Überblick

Zunächst wird die sequentielle Implementierung des JLX Decoders betrachtet. Um einen Gesamtüberblick zu erhalten, wurde für den Decoder ein Variablen-Abhängigkeits-Graphs (Abb. 12) aufgestellt. Die Knoten stellen involvierte Variablen mit den jeweiligen

Wertebereichen dar. Die gerichteten Kanten zeigen von den verwendeten Quellvariablen hin zu den berechneten Zielvariablen. Der Graph visualisiert die Berechnung eines einzelnen Bildpunktes (innere Schleife des Decoder-Algorithmus).

Die Eingabevariablen sind die bekannten Pixelwerte (*Rd*, *Rb*, *Rc*, *Ra*), die Kontextliste, der Bitstrom mit komprimierten Daten und der Offset des aktuellen Codeworts.

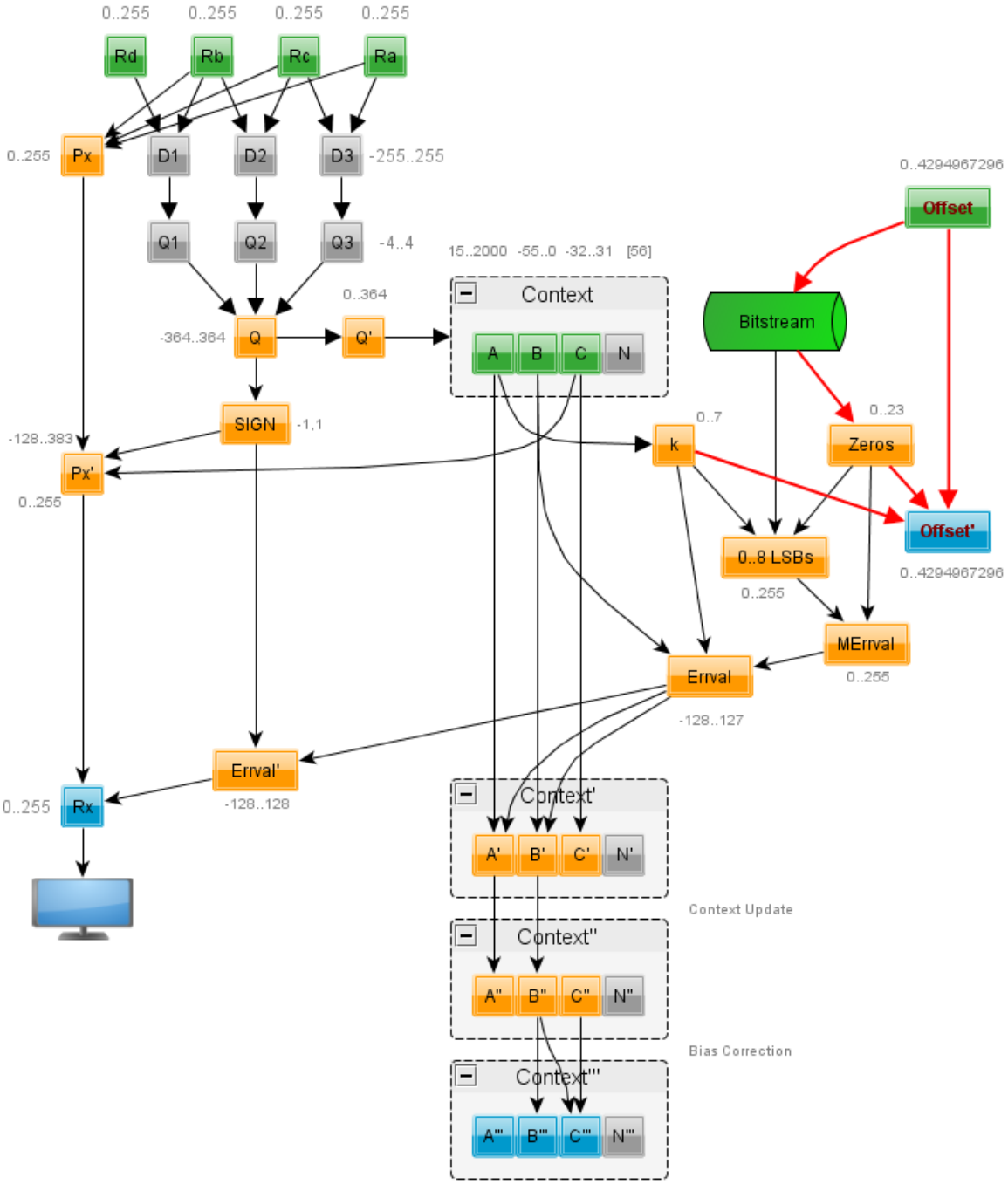


Abbildung 12: JLX Decoder-Graph mit Variablenabhängigkeiten und Wertebereichen.

Als Ausgabe wird der dekodierte Pixelwert *Px* und das Offset auf das nächste Codewort berechnet, gleichzeitig wird der gewählte Kontext aktualisiert.

Die drei Kontextparameter *A, B,* und C erfordern zusammen 23 Bits (*N* ist konstant). Für eine effiziente Verarbeitung unter x86 Architektur werden für sie folgende Basistypen verwendet: *short* (16 Bit) für A, und *char* (8 Bit) für *B* und *C*. Speicherplatzbedarf für alle 365 Kontexte berechnet sich zu 1460 Byte und ist 39% überdimensioniert. Dafür werden aufwendige Offset-Berechnungen, Maskierungen und Schiebeoperationen vermieden.

4.2.2 Optimierungsstrategie

Die Zugriffsreihenfolge auf die Codewörter und Bildpunkte kann vom Decoder nicht beeinfluss werden, sie ist vom Algorithmus eindeutig bestimmt. Allerdings erfolgt das Einlesen bereits sequentiell und die Kontexte passen komplett in den L1 Cache – damit weist der Decoder sehr hohe Cache-Lokalität auf (Kapitel 2.4). Im Zusammenhang mit Lookup-Tabellen muss Cache-Einfluss aber noch einmal genauer betrachtet werden.

Bei den nachfolgenden Optimierungen werden folgende Ziele verfolgt:

- Ersetzung von aufwendigen Berechnungen durch Looku-up Tabellen (*LUTs*)
- Vermeidung von teuren Maschinenbefehlen (insb. Division)
- Vermeidung von Verzweigen (*Conditional Jumps*)

Es muss darauf geachtet werden, dass keine Funktionsaufrufe (*call*) im Decoder-Rumpf vorkommen, da diese zur Registersicherung und Stapelaufbau führen. Bereits der Referenz-Decoder verzichtet daher auf Bibliotheken und APIs und implementiert die gesamte Funktionalität nur durch nativen C++ Code. Bei der Codeaufteilung in Unterfunktionen (zwecks Übersicht) werden stets die Compilerdirektiven *inline* oder *__forceinline* verwendet.

Moderne Compiler bieten einige Einstellungen zur Optimierung des erzeugten Codes. Teure Divisionen durch zweier Potenzen (bei Konstanten) werden durch diese z.B. erkannt und in schnelle Schiebeoperationen umgewandelt. Für optimierungszwecke müssen diese Einstellungen unbedingt gesetzt sein. Im Referenzdecoder ist das bereits erfolgt.

Ein vielversprechender aber auch aufwendiger Weg zur Optimierung ist die Verwendung der Inlineassemblierung, dabei werden Teile des Decoders direkt in Maschinen-Code (Assembler Sprache) geschrieben. Der Entwickler hat direkte Kontrolle über die Prozessorregister und Ausführungsreihenfolge. Zusätzlich können erweiterte Register der SSE und MMX Einheiten als Zwischenspeicher (Cache) mit besonders geringer Latenz eingesetzt werden. Allerdings wurde während der Implementierung festgestellt, dass kleine Assembler-Ausschnitte zu schlechteren Messergebnissen führen, offensichtlich wird dabei die Arbeit der

Compileroptimierung beeinflusst. Die Lösung wäre voraussichtlich eine vollständige Implementierung des Decoders in Assembler. Diese Möglichkeit wurde aber wegen des großen Aufwands aufgegeben.

Der Referenzdecoder benötigt für das gewählte Testbild 258ms Dekodierungszeit. Dieser Wert wird für die Bewertung der nachfolgenden Optimierungen als Bezugspunkt verwendet.

4.2.3 Lookup-Tabellen für Quantisierung

Lookup-Tabellen (*LUT*) sind eine vielversprechende Optimierungsmethode. Dabei werden für alle möglichen Eingabewerte einer Funktion die Ergebnisse vorberechnet. Die Eingabewerte werden auf einen Index abgebildet und das jeweilige Ergebnis unter diesem Index in der Tabelle abgelegt. Die Tabelle selbst kann entweder aus einer Datei geladen oder zur Laufzeit berechnet werden. Um das Ergebnis der Funktion auf einer bestimmten Eingabe zu erhalten, wird die Eingabe wieder zu einem Index abgebildet und das Ergebnis durch einen einzigen Tabellenzugriff (*lookup*) nachgeschlagen.

Lookup-Tabellen haben einen großen Nachteil – der Speicherplatzbedarf wächst exponentiell mit der Indexlänge. Eine optimistische Abschätzung der Indexlänge, um die gesamte Decoder-Funktion durch eine einzige Lookup-Tabelle zu ersetzten:

$$L_{LUT,Decoer} = 32\,Bit_{Nachbarpixel} + 23\,Bit_{Kontext} + 16\,Bit_{Codewort} = 71\,Bit$$

Tabelle mit $2^{71} \approx 2{,}36 \cdot 10^{21}$ Einträgen ist nicht implementierbar. Die LUTs können aber auch leicht auf beliebige Teilfunktionen angewendet werden.

Gradienten Berechnung mit Quantisierung (Abb. 13) ist ein großer zusammenhängender Programmblock im Decoder. Es werden pro Bildpunkt insgesamt 3 Quantisierungen mit je 8 möglichen Verzweigen durchgeführt, daher bietet sich dieser Teil besonders gut für eine Ersetzung durch LUT.

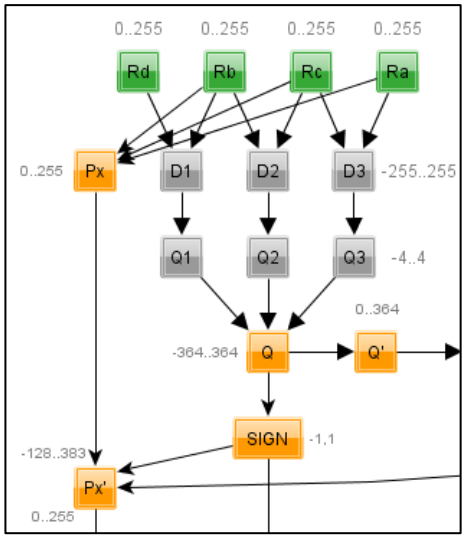


Abbildung 13: Ausschnitt für Gradienten Berechnung und Quantisierung.

Durch Aufstellung einer LUT für alle 4 Nachbarpixel könnten gleich das korrigierte *Px'*, Qunatisierungszeichen *SIGN* und das Kontextindex *Q'* als Ergebnis geliefert werden.

Die erforderliche LUT (Tabelle 12) ist noch zu groß, gleichzeitig ist die Gradienten-Berechnung mit 4 Eingabewerten und 3 einfachen Subtraktionen keine besonders teure Operation.

Eingabe	*Ra, Rb, Rc, Rd*	32 Bit
Ausgabe	*Px', SIGN, Q'*	24 Bit
Gesamtgröße		12 GB

Tabelle 12: LUT für Gradient-Berechnungen, Quantisierung und Indexabbildung.

Bei der Quantisierung wir aber jeder Gradient gegen 8 Schwellwerte verglichen, diese 8 teuren Verzweigungen müssen unbedingt eliminiert werden. Die LUT wird nun auf Quantisierung und Indexabbildung reduziert:

Eingabe	*D1, D2, D3*	27 Bit
Ausgabe	*Q', SIGN*	16 Bit
Gesamtgröße		268MB

Tabelle 13: LUT für Quantisierung und Indexabbildung.

LUT dieser Größe kann im Systemspeicher bereitgestellt werden, ist aber noch deutlich größer als der Cache. Da der LUT Zugriff nicht sequentiell sondern zufällig erfolgt, resultiert dies in vielen Cache-Verfehlungen (*cache miss*). Die LUT Größe für eine einzige Quantisierung (Tabelle 14) hat dagegen sehr geringen Speicherplatzbedarf:

Eingabe	D_i	9 Bit
Ausgabe	Q_i'	16 Bit
Gesamtgröße		1KB

Tabelle 14: LUT für eine einzige Quantisierung.

Diese LUT ersetzt 8 Verzweige einer einzelnen Quantisierung und passt vollständig in den schnellen L1 Cache, muss aber für alle Quantisierungen insgesamt 3 Mal abgefragt werden. Um auch die beiden nachfolgenden Multiplikationen der Indexabbildung zu sparen, können 3 verschiedene LUTs mit bereits vormultiplizierte Faktoren erzeugt werden. Der Speicherplatzbedarf steigt auf 3KB, das reicht aber um alle 3 LUTs im L1 Cache zu unterbringen. Die resultierende Quantisierung mit Indexabbildung ist im nachfolgenden Pseudocode dargestellt:

```
Q1 ← Q1LUT[255 + D1]
Q2 ← Q2LUT[255 + D2]
Q3 ← Q3LUT[255 + D3]
Q ← Q1 + Q2 + Q3
SIGN ← sign(Q);
Q' ← abs(Q);
```

Die 3 vormultiplizierte LUTs für Quantisierung reduzierten die Dekodierungszeit auf 180ms und steigerten somit den Durchsatz um 43,3%.

4.2.4 Berechnung des Golomb-Parameters *k*

Eine weitere Gelegenheit für die LUT Tabellen bietet die Berechnung des Golomb-Parameters *k*. Diese ist im JLX durch eine Schleife implementiert:

```
for (k = 0, (32 << k) < A, k++)
```

Parameter *k* hängt nur von Kontextparameter *A* mit Wertebereich [15…2000] ab ([2] S. 34). Die Schleife kann durch eine einfache LUT ersetzt werden:

Eingabe	*A*	1986 Einträge
Ausgabe	*K*	8 Bit
Gesamtgröße		2KB

Tabelle 15: LUT für die Berechung von k.

Diese Optimierung reduzierte Dekodierungszeit auf 242ms und brachte 6,6 % höheren Durchsatz.

4.2.5 Einlesen der Codewörtern variabler Länge

Sobald der Parameter k bekannt ist, muss aus dem komprimierten Datenstrom das Codewort für die Decodierung eingelesen werden. Der hierfür zuständiger Graph-Ausschnitt ist nochmal in Abb. 14 dargestellt. Das Problem dabei ist, dass die Kodewörter zunächst unbekannte (variable) Länge haben, entsprechend sind die Offset-Positionen der nachfolgenden Codewörter ebenfalls unbekannt. Anderseits erfolgt bei x86 die Adressierung immer an den Bytegrenzen (engl. *data alignment*).

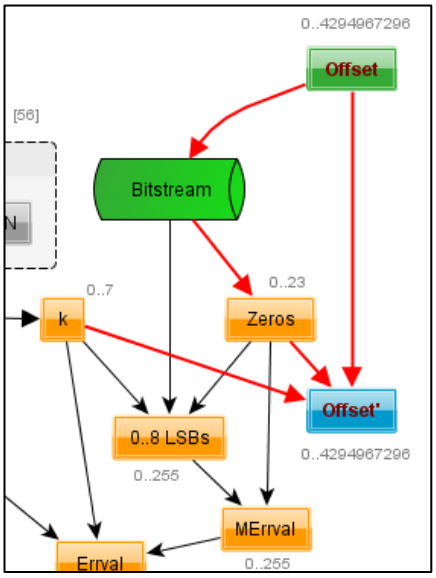


Abbildung 14: Golomb-Dekodierung und Offsetberechnung.

Die Adressierung wird anhand der Abb. 15 erklärt, alle Adressen sind dabei 32 Bit breit. Das *Offset [Bits]* ist bekannt und gibt die Position des nächsten Codeworts innerhalb des Blocks.

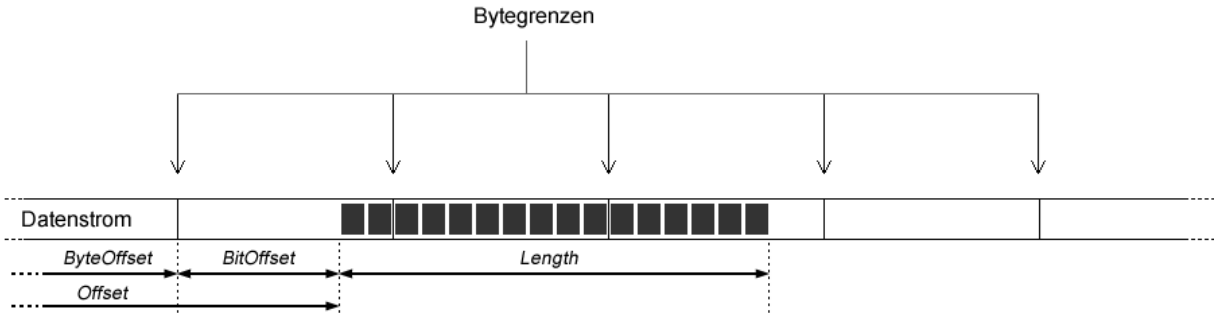


Abbildung 15: Codewort-Größe und Position im Bitstrom

Für den Lesevorgang wird allerdings die absolute Adresse R_A des ersten vom Codewort belegten Bytes benötigt. Ein einfaches Schieben des Offsets (*Offset*) um 3 Bits nach rechts entspricht der Division durch 8 und liefert genau den *ByteOffset* des ersten belegten Bytes relativ zum Blockbeginn:

$$ByteOffset \leftarrow Offset \gg 3$$

Mit der absoluten Blockadresse B_A ergibt sich die gesuchte Adresse R_A:

$$R_A \leftarrow B_A + (Offset \gg 3)$$

Um das erste Bit des Codewortes zu adressieren, muss noch das *BitOffset* relativ zur Bytegrenze berechnet werden. Dies geht mit einfacher UND-Maskierung der 3 niederwertigsten Bits (*LSBs*):

$$BitOffset \leftarrow Offset \wedge (0 \ldots 000111)_{BIN}$$

Bitweises Einlesen des Codewortes würde Schleifen mit bis zu 16 (im Schnitt 4,78) Offsetberechnungen, Leseoperationen, Vergleiche und Additionen erfordern. Mit der Eigenschaft, dass die maximale Länge eines Codeworts 16 Bit ist, kann das Einlesen in einem einzigen schleifenfreien Schritt erfolgen. Das Codewort kann zusammen mit Offset-Bits maximal 3 Bytes im Systemspeicher belegen – hinreichend um vollständig in einen 32 Bit Register untergebracht zu werden. Ab der Adresse R_A werden also insgesamt 4 Bytes mit einer einzigen Leseoperation in den Register geladen. Das Problem dabei ist, dass die x86 Architektur *Little Endian* Kodierung (LSB zuerst) verwendet wodurch die einzelnen Bytes aus dem *Big Endian* kodierten Datenstrom (MSB zuerst) in falscher Reihenfolge im Register angeordnet sind. Die x86 Operation *Byte Swap* (*BSWAP*) angewendet auf Register liefert wieder die richtige Byte-Reihenfolge:

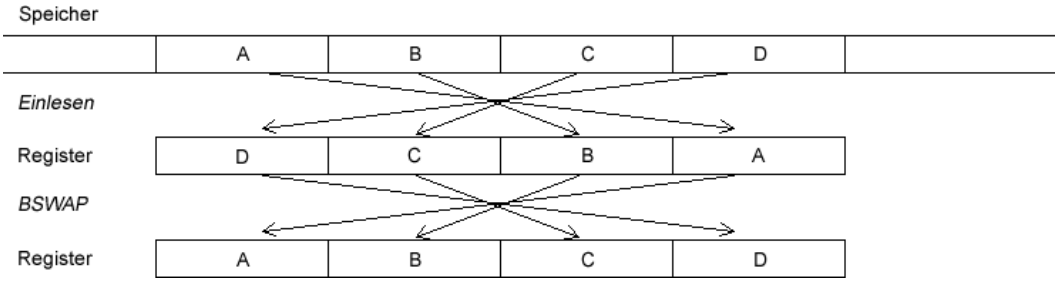


Abbildung 16: Byteanordnung des Datenworts beim Einlesen

Um die führenden Offset-Bits zu eliminieren, wird das Register um *BitOffset* Stellen nach links geschoben. Im nächsten Schritt müssen führende „0“ der unären Kodierung gezählt werden: hierfür bieten einige Architekturen (*ARM, PowerPC*) einen besonders geeigneten Operator: *Count Leading Zeros* (*CLZ*). Unter x86 ist nur ein vergleichbarer Operator verfügbar: *Bit Scan Reverse* (*BSR*) liefert die Position des höchstwertigen „1“-Bits, Bitpositionen werden dabei vom LSB (Position 0) zum MSB (Position 31) durchnummeriert. Wenn alle Bits „0“ sind, ist BSR undefiniert. Dieser Fall kann aber bei JLX nicht vorkommen, da spätestens nach 15 „0“ Bits eine „1“ folgen muss. Die Anzahl führender „0“ lasst sich mit BSR folgendermaßen berechnen:

$$Zeros \leftarrow 31 - bsr(x) \quad x \text{ ist der aktuelle Registerinhalt}$$

Abhängig von der Anzahl führender „0“ wird der Register anschließend nach rechts geschoben und seine k bzw. 8 niederwertigen Bits zur Berechnung des zu dekodierenden Fehlers verwendet. Die $\wedge$ Maskierungen entfernen dabei die nicht mehr benötigte unäre „1“ aus dem Registerwert.

Nachdem die Anzahl führender „0“ (*Zeros*) bekannt ist und der Registerinhalt wie oben beschrieben bearbeitet wurde, kann der Golomb-Decoder *MErr* berechnen.

Nachfolgend ist die Golomb-Decoder Funktion in Pseudocode beschrieben:

$$
\begin{aligned}
&if\ Zeros = QMAX \\
&\quad x \leftarrow (x \gg 16) \wedge (0 \ldots 011111111)_{BIN} \\
&\quad MErr \leftarrow x + 1 \\
&else \\
&\quad x \leftarrow \left(x \gg (32 - Zeros - 1 - k)\right) \wedge \overline{(1 \ll k)} \\
&\quad if\ (QMAX < Zeros) \\
&\qquad Zeros \leftarrow Zeros - 1 \\
&\quad MErr \leftarrow (Zeros \ll k) + x
\end{aligned}
$$

Für die Verwendung der vorgestellten *BSWAP* und *BSR* Operationen muss entweder eingebetteter Assemblercode (*inline assembly*) oder sog. Intrinsische Funktionen (*Intrinsics*) verwendet werden. Entsprechende Intrinsics heißen in VC++ z.B: `_byteswap_ulong` und `_BitScanReverse`.

Im Vergleich zum Referenzdecoder, bei welchem das Codewort durch eine Schleife bit-weise eingelesen wurde, reduziert der optimierte Golomb-Decoder die Dekodierungszeit auf 218ms und bringt somit einen 18,3 % höheren Durchsatz.

4.2.6 Zweigfreie *Error Sign Correction* und *Inverse Error Mapping*

Manche Verzweige lassen sich durch zweigfreie Implementierungen ersetzen, dadurch können Sprungvorhersagefehler in der CPU-Pipeline vermieden werden (siehe Abschnitt 2.4).

Der nachfolgende Pseudocode für *Error sign correction* Schritt

$$\begin{array}{l} if\ Sign\ then \\ \quad Err \leftarrow -Err \end{array}$$

mit $Sign \in \{true, false\}$ lässt sich durch zweigfreie Variante ersetzten, wenn *Sign* als ein Zahlenwert implementiert ist ($Sign \in \{-1, 1\}$):

$$Err \leftarrow Sign \cdot Err$$

Ein weiterer Verzweig ist in *Inverse Error Mapping* vorhanden:

$$\begin{array}{l} if\ odd(MErr)\ then \\ \quad Err \leftarrow -(\frac{MErr+1}{2}) \\ else \\ \quad Err \leftarrow MErr/2 \end{array}$$

Für *Inverse Error Mapping* ließ sich folgender äquivalenter und zweigfreier Pseudocode herleiten:

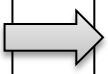
$$Err \leftarrow \Big(\Big(\big((MErr \wedge 1) \oplus 1\big) \ll 1\Big) - 1\Big) * \Big(MErr + \big((MErr \wedge 1) \oplus 0\big)\Big) \gg 1$$

Beide Optimierungen führen zu einer geringfügigen Verbesserung der Dekodierungszeit auf 248ms und steigern den Durchsatz um 4 %.

4.2.7 Das Entrollen der Schleifen

Abschließend betrachten wir die unbekannten Randpixel (*Ra, Rb, Rc, Rd)* außerhalb des Bildausschnitts. JPEG-LS / JLX Decoder definiert eindeutig wie diese Pixel bestimmt werden und verwendet für vertikalen und horizontalen Bildlauf zwei verschachtelte Schleifen, dabei muss durch Koordinatenvergleiche in der inneren Schleife für jeden Bildpunkt bestimmt werden ob und an welchem Rand dieser sich befindet. Um diese zweigbehafteten Sonderberechnungen bei der Berechnung jedes einzelnen Pixels zu eliminieren, können die Decoder-Schleifen entrollt werden. Das Prinzip wird mit dem nachfolgenden Pseudocode erklärt:

```
for (Y = 0; Y < Height; Y++)
{
   for (X = 0; X < Height; X++)
   {
      Sonderberechnung Ra, Rb, Rc, Rd
      Decoderschritte...
   }
}
```



```
Y = 0;
for (X = 0; X < Height; X++)
{
   Sonderfall Ra, Rb, Rc, Rd
   Decoderschritte...
}
for (Y = 1; Y < Height; Y++)
{
   X = 0;
   Sonderfall Ra, Rc
   Decoderschritte...
   for (X = 1; X < Height - 1; X++)
   {
      Decoderschritte...
   }
   X = Height - 1;
   Sonderfall Rd
   Decoderschritte...
}
```

Die innere Schleife (links, rot markiert) enthält Sonderberechnungen für die Bildpunkte am Rand. Der Code für Anfangs- und Endwerte von *X* und *Y* kann aus der inneren Schleife nach Außen gebracht werden. Nach diesem *Entrollen* sind in der inneren Schleife (recht, rot markiert) keine Sonderverzweige für die Randpixelberechnung vorhanden. Da diese Schleife für 99,7% der Bildpunkte ausgeführt wird, ist der Aufwand für die Randpixel auf 0,3% gesunken. Der Nachteil der Methode ist, dass der Maschinencode für die inneren Decoder-Schritte mehrmals repliziert wird (*code bloating*), was die *L1 Instruction Cache* Effizienz beeinflussen kann.

Das Entrollen der Schleifen reduzierte die Dekodierungszeit auf 239ms und führte zu 7,9% Durchsatzsteigerung.

4.2.8 Ergebnisse

Leistungsmessungen verwenden eine Testaufnahme im JLX Format mit 1293 Bildern und beziehen sich ausschließlich auf die sequentielle CPU-Dekodierung der Blöcke aus dem Systemspeicher, das Nachladen von Massenspeicher (*Disk-I/O*) wird nicht berücksichtigt. Jedes Bild ist in 8 Segmente unterteilt und hat eine Auflösung von 2304 x 1296 Bildpunkten. Kompressionsstärke der Aufnahme beträgt 1,71 (*cbpp = 4,66*). Die gemessenen Werte wurden mit der Testkonfiguration 2 erzielt.

Die Tabelle 16 listet nur die wichtigsten Optimierungen, andere sind im Referenzdecoder bereits enthalten. Es ist leicht erkennbar, dass mit gezielten Optimierungsmaßnahmen die Performance des Decoders deutlich verbessert werden kann. Alle Optimierungsmaßnahmen zusammen haben die Decoder-Leistung mehr als verdoppelt. Diese optimierte Version des Decoders wird im Kapitel 5 als Ausgangspunkt für die Parallelisierung verwendet.

	Dekodierungszeit	Durchsatz [Bilder/s]	Speedup
Referenzdecoder	258 ms	3,88	-
3 LUTs für Quantisierung	180 ms	5,56	+43,3 %
LUT für Golomb-Parameter k	242 ms	4,13	+6,6 %
Optimiertes Einlesen der Codewörter	218 ms	4,59	+18,3 %
Schleifen Ausrollen	239 ms	4,18	+7,9 %
Zweigfreies *Inverse Error Mapping*	248 ms	4,03	+4 %
Alle Optimierungen eingeschaltet	122 ms	8,20	+111,4 %

Tabelle 16: Messergebnisse der einzelnen Optimierungen

LUTs haben sich als besonders geeignet dafür gezeigt, aufwendige Berechnungen mit geringer Eingabebreite zu ersetzen. Werden aber mehrere LUT Tabellen verwendet, konkurrieren diese um den gemeinsamen Cache. Kontextupdates können im Decoder als weitere aufwendige Berechnung angesehen werden. Die hierfür notwendigen LUTs waren aber zu groß und konkurrierten mit den beiden anderen bereits eingeführten Look-up Tabellen. Der erzielte Gewinn durch Optimierungen bleibt somit bei 111,4%.

4.3 LX2: Optimierung des Problems

Bei der näheren Betrachtung von JLX und JPEG-LS stellt sich fest, dass die Kontextmodellierung und die hierfür notwendige Quantisierung den überwiegenden Anteil der Decoder-Berechnungen ausmachen. In diesem Abschnitt wird ein Versuch gewagt die Performance des Decoders über die Kompressionseffizienz zu stellen und mit dieser Vorgabe ein neues, auf die Bereitstellung von Hochgeschwindigkeitsaufnahmen gezieltes, Kompressionsverfahren zu modellieren. Aus JPEG-LS wird dabei nur das Grundprinzip der Vorhersage und Fehlerkodierung übernommen. Auch die Eigenschaft der Verlustlosigkeit wird beibehalten. Adaptivität durch Kontextmodellierung wird dagegen aufgegeben. Das neue modifizierte Codec sei im folgenden *LX2* genannt.

4.3.1 Vereinfachung zum kontextlosen Codec

Als größte Änderung werden beim LX2 alle 365 Kontexte und alle damit verbundene Berechnungen und Zwischenvariablen komplett gestrichen. Dadurch verliert die Fehlerberechnung an Adaptivität im Gradienten-Kontext.

Es bleiben nur die nachfolgenden Verarbeitungsschritte:

- Vorhersage (Prediction)
  - MED-Prädiktor (Edge-detecting predictor)
  - Berechnung des Vorhersage-Fehlers (Prediction error computation)
  - Fehler-Abbildung auf positive Werte (Error mapping to non-negative values)
- Fehler-Kodierung (Prediction error encoding)
  - Bestimmung des Golomb-Parameters k (Golomb coding variable computation)
  - Fehler-Kodierung (Mapped-error encoding)

Der vereinfachte Decoder-Graph ist in nachfolgender Abbildung dargestellt:

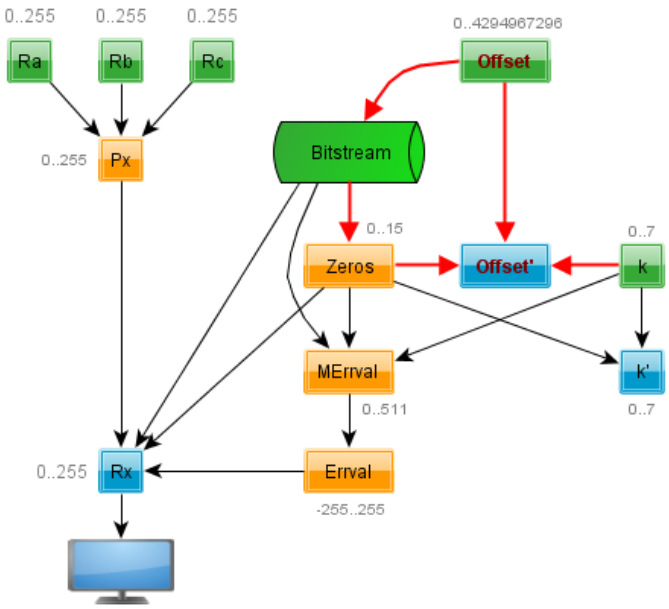


Abbildung 17: LX2 Decoder-Graph mit Variablenabhängigkeiten und Wertebereichen.

Das Entfallen der Kontexte reduziert den Speicherbedarf auf einige wenigen Variablen.

Um die Verzweige und *min*/*max* Berechnungen bei der Vorhersage zu eliminieren, wurde auch der MED Prädiktor vereinfacht:

$$Px \leftarrow CLAMP(Ra + Rb - Rc)$$

Dieser hängt nicht mehr von *Rd* ab, wodurch auch die Sonderbehandlung für Bildpunkte am rechten Bildrand entfällt. Da der L1 Cache jetzt frei von Kontexten ist, wird für das Abschneiden der Vorhersage (*clamping*) eine LUT mit 512 Einträgen (0,5KB) eingesetzt, was zwei weitere Verzweige spart. Mit der Vorhersage *Px* wird der Fehler *Err* berechnet und mittels *Error Mapping* (identisch zu JLEG-LS / JLX) auf positive natürliche Zahlen abgebildet:

$$
\begin{aligned}
&Err \leftarrow Px - Ix \\
&if\ 0 \leq Err \\
&\quad MErr \leftarrow 2 \cdot Err \\
&else \\
&\quad MErr \leftarrow -2 \cdot Err - 1
\end{aligned}
$$

Das abgebildete Fehler *MErr* wird mit Golomb-Coder kodiert. Der notwendige Parameter k wird beim JLX aus den Kontextparametern *A* und *N* berechnet. Beim LX2 wird ein anderer Weg eingeschlagen. Basieren auf Ideen der Microsoft Research Untersuchungen: „Adaptive Run-Length / Golomb-Rice Encoding of Quantized Generalized Gaussian Sources with Unknown Statistics“[4] wurde für die Bestimmung von *k* die Anzahl führender „0“ des vorher erzeugten Codeworts gewählt. Gleichzeitig wurde die maximale Länge des Codeworts auf 24 Bit (*QMAX* = 15) beschränkt. Begonnen wird mit einem Startwert von *k* = 1. Jedes Mal, wenn Golomb-Coder eine große Anzahl führender „0“ erzeugt, wird k erhöht. Dadurch wird gehofft, dass beim nächsten Bildpunkt die Division durch k einen noch deutlich kleineren Quotient und damit weniger führende „0“ liefert. Wenn gar keine „0“ entstehen, sieht LX2 das als Indiz dafür, dass der Fehler für das aktuelle *k* zu gering ist, und versucht durch langsame Reduktion (von *k*) sich an die minimale Gesamtlänge der Codewörter heranzutasten.

Die Veränderung Δk in Abhängigkeit von der Anzahl führender „0“ ($Zeros_{prev}$) des vorherigen Codeworts wurde für LX2 durch Messungen mit Testbildern näherungsweise bestimmt und zu einer LUT zusammengefasst.

$Zeros_{prev}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Δk	-1	0	+1	+1	+2	+2	+2	+2	+2	+2	+3	+3	+3	+3	+3	+3

Tabelle 17: Veränderung von k in Abhängigkeit von der Anzahl führender "0".

Das neue *k* berechnet sich dann zu:

$$k \leftarrow \max(0, \min(7, k + \Delta k))$$

Damit bleibt k stets im Wertebereich [0...7]. Das Abschneiden (*clamping*) wird wieder durch eine LUT mit 16 Einträgen (16 Byte) realisiert.

Anschließend wird für *MErr* der neue Quotient (*Zeros*) bestimmt. Ist $Zeros < QMAX$ wird *MErr*, wie bei JLX, unär kodiert. Im Fall von $Zeros \geq QMAX$ wird auch eine *Escape Sequence* und (im Unterschied zu JLX) die 8 Bits des tatsächlichen Pixelwerts *Ix* statt des

Fehlers *MErr* gespeichert. Diese letzte Modifikation wird beim Decoder alle Schritte für Vorhersage, Fehlerrücktransformation und Pixelwert-Berechnung eliminieren, allerdings nur für diesen Sonderfall.

4.3.2 Ergebnisse

Die ausführliche Betrachtung von LX2 entzieht sich den Rahmen dieser Arbeit. Es soll damit nur angedeutet werden, welche Wege aufgeschlagen werden können, um unter der x86 Architektur den Durchsatz bei der Bereitstellung von Hochgeschwindigkeitsaufnahmen noch viel weiter zu steigern.

LX2 wurde für die nachfolgenden Tests jeweils als Encoder und Decoder implementiert. Die Korrektheit des Algorithmus wurde mit Testbildsequenzen, unter anderen mit Rauschbildern, bestätigt. Wegen der geringen Komplexität konnte eine vollständig in Assembler geschriebene Decoder-Variante abgeleitet werden, welche auch getestet wurde.

Nachfolgend wird die Kompressionsstärke und Decoder-Leistung von LX2 und JLX verglichen. Für die Messungen wurde Testkonfiguration 2 mit drei Aufnahmen verwendet. Die erste Aufnahme ist rauscharm und erreicht hohen Kompressionsfaktor. Um verschiedene Kompressionsfaktoren zu untersuchen, wurden die beiden anderen Aufnahmen durch künstliches Hinzufügen vom weißen Rausch von der ersten abgeleitet. Aufnahme 2 entspricht weitgehend der gemessenen mittleren Kompressionsstärke von realen Aufnahmen in [2]. Gemessen wurde reine Decoder-Leistung ohne Disk-I/O.

	Komprimpierte Dateigröße und (cbpp)			Durchsatz [Bilder/s]		
	JLX	LX2	Verlust	JLX	LX2	Gewinn
Aufnahme 1 (rauscharm)	26,4% (2,10)	27% (2,16)	+0,6%	8,33	66,7	+700%
Aufnahme 2 +5,88% Rauschanteil	58,2% (4,66)	69,7% (5,58)	+11,5%	8,20	66,7	+713%
Aufnahme 2 +12,15% Rauschanteil	70,5% (5,63)	82,5% (6,59)	+12,0%	8,06	66,7	+726%

Tabelle 18: Vergleich der Kompressionsstärke und Decoder-Leistung zwischen LX2 und JLX.

Durch die starke Vereinfachung erreicht der LX2-Decoder 8-fachen Durchsatz im Vergleich zu JLX. Die Leistungssteigerung wird dabei durch den Verlust der Kompressionsstärke erkauft.

5 Parallelisierung

Bei der Parallelisierung wird die Dekodierungsaufgabe aufgeteilt und von mehreren Recheneinheiten gleichzeitig abgearbeitet. Dies reduziert die benötigte Zeit (geringere *Latenz*) und erlaubt mehr Daten pro Zeiteinheit zu verarbeiten (mehr *Durchsatz*). Die pro Pixel ausgeführten Decoder-Schritte benötigen in der inneren Decoder-Schleife zu wenig Zeit um eine *Task-Decomposition* anzuwenden, im Nachfolgenden wird daher *Domain-Decomposition* betrachtet. Die Aufnahmedaten liegen 3-dimensional vor: eine Dimension für zeitliche Auflösung der Bildsequenz, und zwei Dimensionen jeweils für horizontale und vertikale Bildauflösung jedes einzelnen Bildes.

Der JLX Encoder verarbeitet nicht nur jedes Bild individuell, d.h. *unabhängig* von den anderen, sondern teilt zusätzlich das Bild in vertikale Segmente. Jedes Segment wird als ein eigenständiges Bild betrachtet und mit einem eigenen Satz von Kontexten *unabhängig* von allen anderen Segmenten kodiert. Die erzeugten Codewörter werden über Multiplexer miteinander verschränkt und über einen kleinen FIFO Puffer zum Transfer weitergegeben (Abschnitt 2.1). Der so entstehende komprimierte Datenstrom lässt sich zweidimensional zerlegen (Abb. 18): die *unabhängigen* Datenblöcke können für temporäre Parallelisierung (*Interframe-Parallelization*) und *unabhängige* Segmente für segmentbasierte Parallelisierung (*Intraframe-Parallelization*) eingesetzt werden. Es wird sich herausstellen, dass die *Unabhängigkeit* der Segmente durch die Verschränkung nur zum Teil gegeben ist und eine *Synchronisation* erfordert.

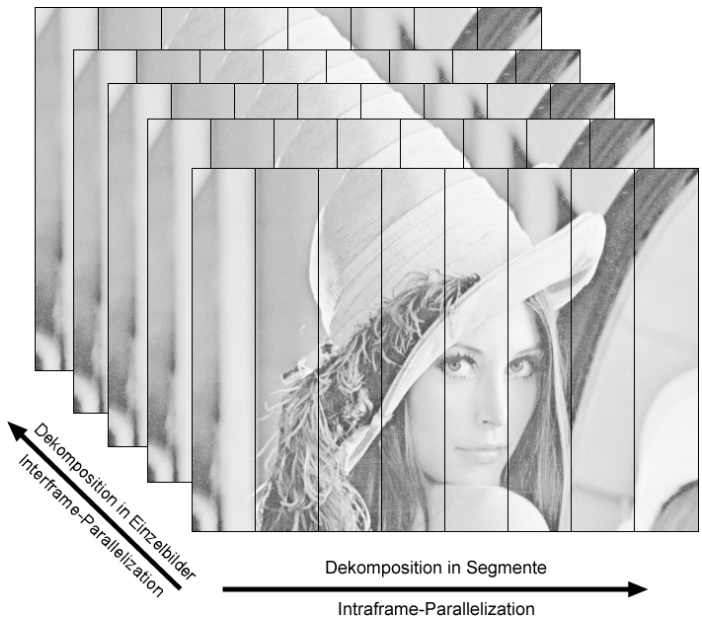


Abbildung 18: Domain-Dekomposition für JLX Parallelisierung.

5.1 Parallelisierungsgrad, Speed-up und Effizienz

Der Parallelisierungsgrad P eines Algorithmus ist die Anzahl parallel ausführbaren Operationen [12].

Im Fall von *Domain-Decomposition* entspricht Parallelisierungsgrad der Anzahl (unabhängiger) Komponenten, in welche die Daten zerlegt werden können. Beide Parallelisierungsstrategien können im vorliegenden Fall auch miteinander kombiniert werden, ihre Parallelisierungsgrade werden dabei miteinander multipliziert:

$$P_{ges} = P_{Temporär} \cdot P_{Segmentbasiert}$$

Die Leistungssteigerung durch Parallelisierung wird als *Speed-up S* [12] angegeben:

Es sei T_S die Zeit, die man mit dem schnellsten bekannten seriellen Algorithmus zur Lösung eines gegebenen Problems auf einem Prozessor benötigt und T_P die Zeit, die man zur Lösung des gleichen Problems auf dem Parallelrechner mit P solchen Prozessoren benötigt. Der Speed-up eines parallelen Algorithmus ist:

$$S = \frac{T_S}{T_P}$$

Ferner ist die Effizienz des parallelen Decoders interessant [12]:

Die Effizienz eines Parallelen Algorithmus bei einer Berechnung mit N_P Prozessoren ist

$$e = \frac{S}{N_P}$$

5.2 Multicore CPU

Nun wird eine allgemeine Anwendung zur Bereitstellung von Hochgeschwindigkeitsaufnahmen auf einem aus Massenspeicher, Systemspeicher und mehreren Prozessoren bestehenden x86 Rechensystem betrachtet. Der JLX Decoder ist nur ein Teil der Anwendung und führt ausschließlich Dekodierung auf komprimierten Eingabedaten aus. Wichtige Funktionen wie Steuerung, Dateiformat, Nachladen von Daten, Objekt- und Thread-Verwaltung, Verarbeitung sowie Rendering werden von anderen Anwendungskomponenten bereitgestellt. Einige dieser Komponenten haben direkten Einfluss auf die Gesamtperformance der Bereitstellung. Es wird daher versucht neben dem reinen Decoder-Durchsatz auch den Gesamtdurchsatz der relevanten Anwendungsteile zu bewerten.

5.2.1 Verarbeitungspipeline

Die wesentlichen Bestandteile einer Verarbeitungs-Pipeline sind das Laden von Blöcken aus dem Massespeichern in den Systemspeicher und das Dekodieren dieser Blöcke in die Bild-Puffer (Abb. 19).

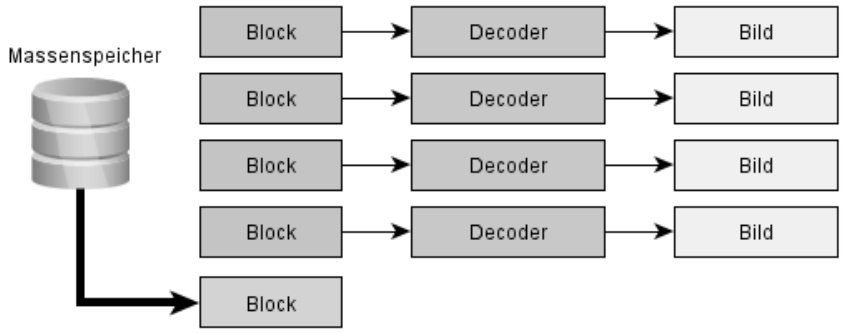


Abbildung 19 Sequentielles Disk-I/O und paralleles Decodieren in der Verarbeitungspipeline.

Während die Dekodierung von mehreren Recheneinheiten parallel (gleichzeitig) ausgeführt wird, erfolgt das Laden von Datenträgern (Disk-I/O) in der Regel nicht überlappend (seriell). Trägt man diese zwei prinzipiell verschiedenen Vorgänge zeitlich auf, dürfen sich die I/O Vorgänge nicht überschneiden. In Abb. 20 sind dabei zwei mögliche Szenarien für den Parallelisierungsgrad 4 dargestellt.

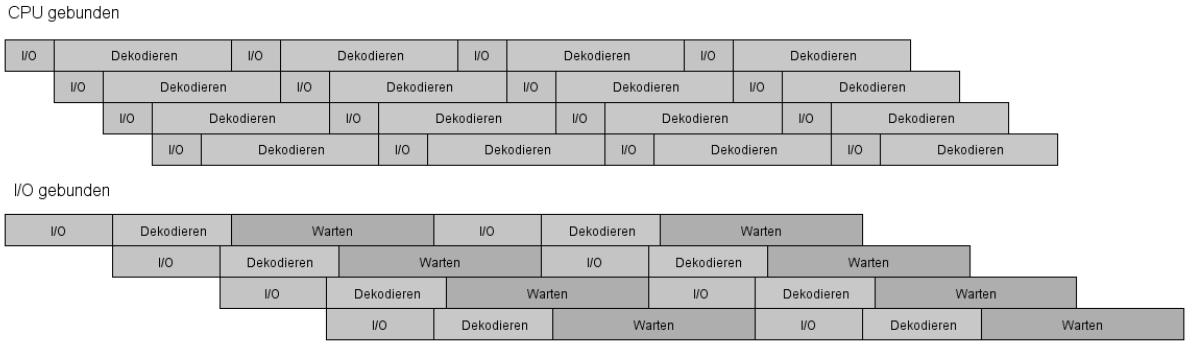


Abbildung 20: CPU gebundene (oben) und I/O gebunden Pipeline (unten).

Bei zu geringen I/O Zeiten (bzw. zu langen Dekodierungszeiten) müssen Datenträger warten bis Decodierung abgeschlossen ist und neue Daten angefordert werden – die Verarbeitungspipeline wird in diesem Fall als *CPU gebunden* bezeichnet. Im umgekehrten Fall müssen Recheneinheiten auf Daten warten – die Pipeline ist dann *I/O gebunden*. Für *N* parallele Decoder-Instanzen lässt sich das optimale Verhältnis zwischen I/O- und Decodierungszeit einfach angeben:

$$\frac{T_{I/O}}{T_{Decode}} = \frac{1}{N}$$

Mit dieser Metrik lässt sich ein Kompressionsverfahren für gegebene Hardware als I/O gebunden, CPU-gebunden oder ausbalanciert charakterisieren. Sowohl JLX als auch LX2 werden in nachfolgenden Kapiteln diesbezüglich bewertet.

Für I/O Vorgänge wird in der Regel DMA Transfer verwendet, die CPU ist während dieser Zeit nur selten durch verursachte Hardware Interrupts belegt. Um alle vorhandenen CPU Kerne auszulasten, muss die Anzahl der Decoder-Threads höher als die Anzahl der vorhandenen Kerne sein. Die CPU Zeit, welche während der I/O Vorgängen nicht benutzt wird, kann dann vom Betriebssystem den wartenden Decoder-Threads zugewiesen werden.

5.2.2 Anwendungsarchitektur

In der Visualisierungsanwendung findet das Dekodieren nicht alleine statt, sondern ist Teil einer Abfolge von mehreren Verarbeitungsschritten. Trägt man die Abfolge von Bildern auf eine Achse und die zu durchlaufenden Schritte für jedes Bild auf die andere, erhält man eine tabellarische Struktur in Abb. 21.

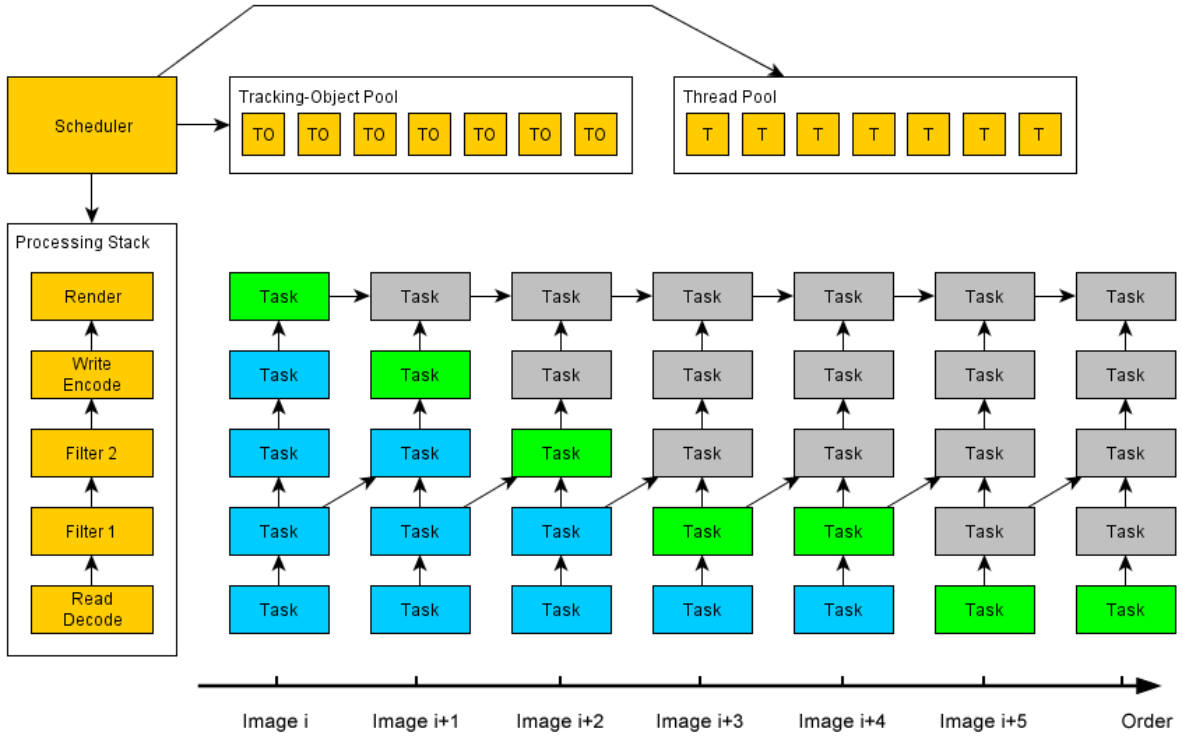


Abbildung 21: Berechnungsaufgaben aufgetragen nach Bildfolge und Verarbeitungsschritte, und die dafür benötigten Softwaremodule.

Jede Zelle in dieser Tabelle ist die Ausführung eines Verarbeitungsschrittes auf Daten aus vorherigen Schritten und ist als atomare Aufgabe (*Task*) aufgefasst. Jede Aufgabe kann unabhängig und parallel zu allen anderen ausgeführt werden, jedoch erst dann, wenn alle benötigten Eingabedaten bereitstehen. Für die Unterstützung dieser Verarbeitungsstruktur werden spezielle Module (z.B. als C++ Klassen implementierbar) eingeführt.

Um Entscheidungen (z.B. Instanziierung, Freigabe von Ressourcen) korrekt treffen zu können, muss mindestens ein Modul die *globale Sicht* besitzen. Diese Sicht hat in der betrachteten Anwendungsarchitektur der *Scheduler*. Wenn der Scheduler eine Aufgabe von der Hauptanwendung erhält (z.B. Abspielen einer Aufnahme aus Datei), werden von ihm *Processing Stack, Tracking-Object Pool* und *Thread-Pool* instanziiert. Im Processing Stack werden für jeden Verarbeitungsschritt verantwortliche Klassen instanziiert und in der Reihenfolge der Verarbeitung in eine Liste angeordnet. Am Anfang wird in der Regel eine Klasse für das Einlesen und Dekodieren von Aufnahmedateien stehen (*Quellenanbieter*). Es sind aber auch Generatoren für das Erzeugen von künstlichen Bildsequenzen (z.B. Rausch, Mandelbrot) denkbar. Nach dem Quellanbieter können mehrere Filter angeordnet werden. In Abb. 21 könnten z.B. *Filter 1* ein Rauschfilter und *Filter 2* ein Differenzbildfilter sein. Für die Speicherung oder Transkodierung in andere Formate kann im Stack optional eine Schreiber-Klasse eingeordnet werden. Ausgabe-Klasse (*Renderer*) präsentiert die Ergebnisse dem Benutzer visuell und schließt den Stack ab.

Ein *Thread Pool* erzeugt und verwaltet eine Menge von generischen *Threads*. Um den Overhead für die Thread-Erzeugung zu reduzieren werden Threads nur einmal angelegt und, wenn nicht mehr benötigt, für die spätere Verwendung suspendiert.

Während der Verarbeitung eines Bilds können in jeder Stufe Zwischenergebnisse erzeugt werden, welche wiederum als Eingabe für die nachfolgenden Verarbeitungsstufen möglicherweise benötigt werden. Das wichtigste Zwischenergebnis ist der Puffer mit Bilddaten selbst, es sind aber beliebig komplexe Datenstrukturen denkbar. Für die Instanziierung, Speicherung, Verwaltung und Bereitstellung der Zwischenergebnisse für ein einzelnes Bild wird ihm während seiner Lebenszeit ein generisches *Tracking-Object* zugeordnet. Insbesondere speichert jedes Objekt Informationen über den aktuellen Verarbeitungszustand seines Bildes. Die Menge der Tracking-Objekte wird im *Tracking-Object Pool* verwaltet.

Der Scheduler bestimmt aus dem Tracking-Object Pool ab, welche Aufgaben abgeschlossen sind und bestimmt gleichzeitig welche neuen Bilder hinzukommen sollen. Ferner teilen alle Verarbeitungsobjekte dem Scheduler mit, Zwischendaten welcher Stufe und von welchen Bildern sie als Eingabe benötigen. Aus allen diesen Informationen lässt sich ein Abhängigkeitsgraph ableiten. Um die nächsten Aufgaben zu bestimmen und nicht mehr benötigte *Tracking-Objects* freizugeben, muss der Scheduler ein graphen-theoretisches Problem effizient lösen (wird hier nicht betrachtet). Sobald die nächsten Tasks feststehen, werden die dazugehörigen Tracking-Objekte zusammen mit den zuständigen Einstiegsprozeduren der Verarbeitungsobjekte an den Thread-Pool übergeben. Der Thread-

pool sperrt daraufhin die Tracking-Objekte (*lock*) und führt die Einstiegsprozeduren auf den freien Threads aus. Ein Thread darf dabei nur in das ihm zugewiesene gesperrte Tracking-Objekt schreiben, alle anderen gesperrten Tracking-Objekte, die als Eingabe dienen, dürfen nur gelesen werden. Der Scheduler muss bei der Aufgabenverteilung sicherstellen, dass kein Wettlauf (*race condition*) auftritt: Tracking-Objekte in die geschrieben wird, können nicht als Eingabe für andere Aufgaben verwendet werden. Nach der Fertigstellung gibt jeder Thread seine gesperrten Tracking-Objekte frei, löst ein Fertigstellungsereignis aus und übergeht in suspendierten (wiederverwendbaren) Zustand.

Dies ist nur eine der möglichen Anwendungsarchitekturen. Die meiste Komplexität ist in den beiden Pools, dem Processing Stack und insbesondere im Scheduler enthalten. Dafür können die Verarbeitungs-Klassen einfach und frei von Threadverwaltung gehalten werden. Zwar ist eine interne Parallelisierung innerhalb der Klasse weiterhin möglich, aber auch ohne diese Maßnahme profitiert die Anwendung von der aufgabengranularen Gesamtparallelität des Schedulers.

Basierend auf der vorgestellten Anwendungsarchitektur wurde für die Untersuchung der Parallelität eine etwas einfachere Visualisierungsanwendung in C++ implementiert (Abb. 22). Die oberste Instanz mit globaler Sicht ist der *Graph*. Scheduler, Frame Pool und Thread Pool entspricht den obengenannten Konzepten, in Frame-Objekten werden jedoch neben dem Status nur Block- und Bild-Puffer verwaltet. Processing Stack wurde durch eine vereinfachte Anordnung aus *Reader*, *Writer* und *Renderer* ersetzt. Filter und komplexe Task-Abhängigkeiten wurden komplett ausgelassen.

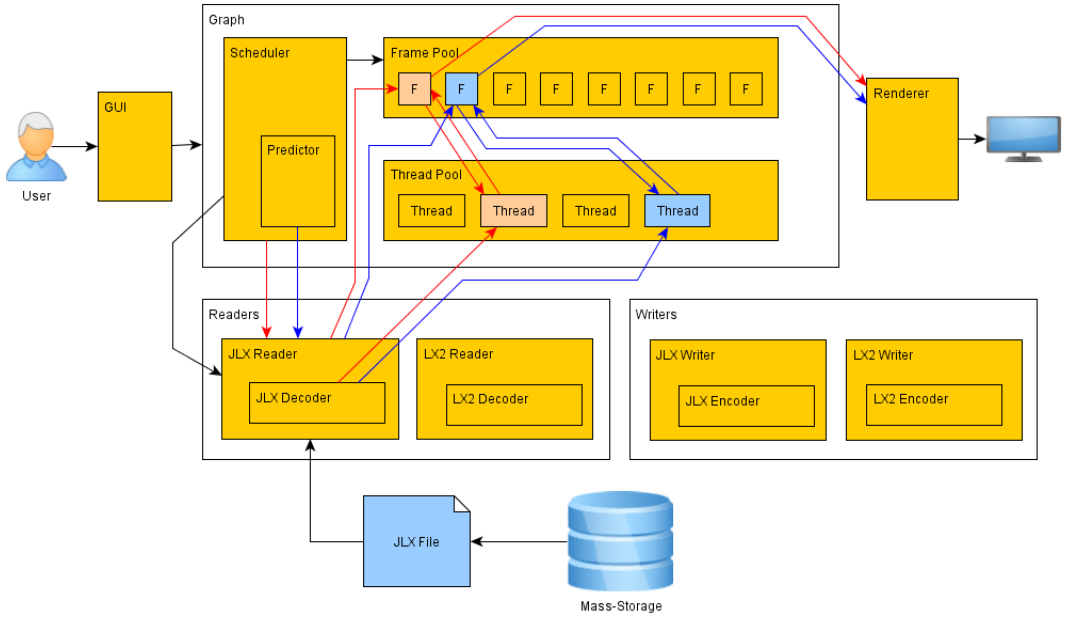


Abbildung 22: Aufbau der implementierten Testanwendung.

5.2.2.1 Prädiktor

Ein Frame-Prädiktor versucht vorauszusagen, welche Bilder als nächstes benötigt werden. Dies wird am Beispiel der Abb. 22 veranschaulicht. Der Benutzer öffnet über das GUI Interface eine JLX Datei und möchte diese abspielen. Der Graph erkennt Dateiendung, veranlasst den Scheduler einen passenden *Reader* und alle benötigten Pools anzulegen und fordert das erste Bild an. Die Bildanfragen an Scheduler können synchron und asynchron erfolgen. Im synchronen Fall ist die anfragende Funktion blockiert bis das angeforderte Bild vollständig verarbeitet ist. Im asynchronen Fall, wird die Verarbeitung bei der ersten Abfrage aufgenommen. Der Scheduler liefert auf alle asynchronen Anfragen sofort einen Statuscode der Nichtbereitschaft zurück. Erst wenn die Verarbeitung abgeschlossen ist wird ein Zeiger auf fertige Bilddaten zurückgegeben.

Ein *Timer* im Graph ruft regelmäßig eine Funktion zum Anfragen und Abspielen des nächsten Bildes auf. Ist das Bild noch nicht fertig, muss entschieden werden, ob das Bild verworfen (*frame drop*) oder darauf gewartet wird. Im letzten Fall kann die Anwendung Bilder nicht schneller abspielen als der Verarbeitungsdurchsatz es zulässt. Der Graph setzt dafür Selbstjustierung ein: der Timer-Intervall wird abhängig davon, ob bei einer Anfrage das Bild bereit war oder nicht, reduziert oder erhöht.

Der im Scheduler enthaltene Prädiktor betrachtet die Reihenfolge der Bildanforderungen und leitet davon ab, welche Bilder voraussichtlich als nächstes angefordert werden. Sind im Thread-Pool und Frame-Pool noch freie Ressourcen vorhanden, werden diese mit der Verarbeitung der vorhergesagten Bilder ausgelastet.

Aus Sicht der Anwendung kann ein Prädiktor völlig transparent arbeiten. Selbst wenn die Anwendung keine besondere Parallelisierungs-Logik enthält, kann der Prädiktor im Scheduler den Gesamtdurchsatz steigern.

5.2.3 Temporäre Parallelisierung

Eine Hochgeschwindigkeitsaufnahme besteht aus vielen unabhängigen Einzelbildern, die zugehörigen Datenblöcke lassen sich durch das vorgestellte Dateiformat mit Katalog-Tabelle (Abschnitt 4.1) direkt adressieren und in den Systemspeicher laden. Die einfache Idee der temporären Parallelisierung besteht darin, die unabhängigen Bilder von mehreren Threads gleichzeitig zu dekodieren. Jeder Decoder-Thread greift dabei auf separaten Datenblock eines Einzelbildes. Da es keine Daten-Abhängigkeiten zwischen den Blöcken gibt, kann jeder Thread unabhängig und parallel zu allen anderen arbeiten, es ist insbesondere keine Synchronisation zwischen den Decoder-Threads nötig.

Die Ausführung der 8 Decoder-Threads (ohne I/O) ist in Abb. 23 dargestellt. Jeder Thread führt mittels der sequentiellen und in Kapitel 4 optimierten Decoder-Funktion eine vollständige Bilddekodierung durch. Die Dekodierung (D) eines Bildpunktes verlangt Offset (O) des zugehörigen Codeworts als Eingabe. Erst nach Dekodierung ist das nächste Offset bekannt und die innere Decoder-Schleife noch einmal ausgeführt werden kann. Die gerichteten Kanten in der Abbildung zeigen Offset-Weitergabe zum nächsten Schleifendurchlauf. Insbesondere müssen die Threads nicht untereinander kommunizieren, eine Thread-Synchronisation ist damit nicht notwendig.

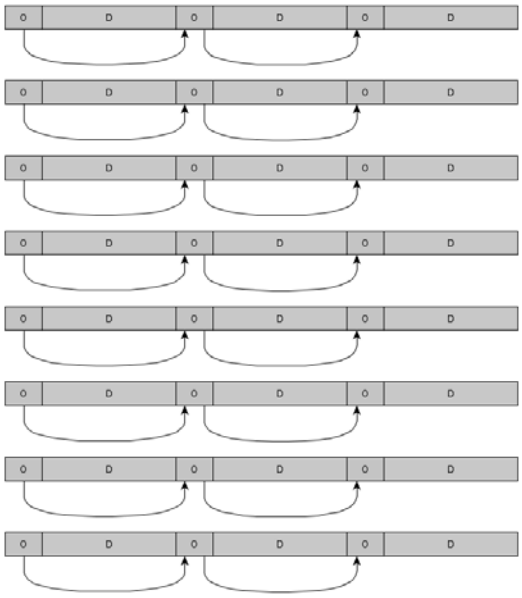


Abbildung 23: Abhängigkeitsfreie Ausführung der temporären Parallelisierung mit 8 Threads.

Der maximale theoretische Parallelisierungsgrad von temporären Parallelisierung entspricht genau der Bildanzahl der Aufnahme. Aufnahmen mit hunderten und mehr Bildern könnten Rechensysteme mit einer sehr großen Anzahl von Recheneinheiten auslasten. Das Problem dabei ist, dass die benötigte Transfer-Bandbreite linear mit dem Parallelisierungsgrad skaliert, wodurch immer schnellere I/O Busse notwendig sind. Mit der Pufferung der komprimierten Datenblöcke und der dekomprimierten Bilder im Systemspeicher steigt auch der Speicherplatzbedarf linear an.

Mit Parallelisierungsgrad *P*, Bilddatengröße *B* [MB] und der mittleren Anzahl komprimierten Bits pro Pixel *cbpp* lässt sich der mittlere Speicherplatzbedarf *S* bestimmen:

$$S = P \cdot B \cdot \left(1 + \frac{cbpp}{8}\right) \quad [MB]$$

Der mittlere Speicherplatzbedarf für eine Interframe-Parallelisierung vom Grad 16 einer 3 Megapixel Aufnahme mit *cbpp=4,78* ist beispielweise:

$$S = 16 \cdot 3\ [MB] \cdot \left(1 + \frac{4{,}78}{8}\right) = 76{,}68\ [MB]$$

Für aktuelle PC-Systeme mit mehreren GB Systemspeicher und vergleichsweise geringen Anzahl an Prozessorkernen dürfte dieser Aufwand vertretbar sein. Dies könnte sich mit *Manycore* CPUs aber ändern.

5.2.3.1 Ergebnisse

Für die nachfolgenden Messungen wurde die temporäre Parallelisierung über den vorgestellten Scheduler mit einem Thread-Pool implementiert (Kapitel 5.2.2).

Es kam wieder Testsystem 2 (S. 18) zum Einsatz (4 physikalische- bzw. 8 virtuelle *Hyperthreading*-Kerne). In einen Testlauf wurden alle 1293 Bilder der Aufnahme 2 mit 5,88% Rauschanteil (*cbpp=4,66*) einmal abgespielt und die verstrichene Gesamtzeit T_{ges} gemessen. Der Bildausgabe und die GUI wurden hierzu überbrückt, wodurch sich die Messungen nur auf das Nachladen (Disk-I/O), Dekodierung sowie Task-Verteilung des Schedulers beziehen. Die Testläufe wurden für JLX und LX2 mit jeweils unterschiedlichen Parallelisierungsgrad (Threadzahl) durchgeführt.

Durchsätze in Bildern/s und in MB/s berechnen sich nach folgenden Formeln:

$$D_{fps} = \frac{1293}{T_{ges}} \quad [Bilder/s]$$

$$D_{MB/s} = \frac{Filsize_{compressed}}{T_{ges}} \quad [MB/s]$$

Speed-up und *Effizienz* berechnen sich aus den gemessenen Zeiten T_S (sequentieller Fall) und T_P (Parallelisierungsgrad P) mit Formeln aus Abschnitt 5.1.

Die Ergebnisse der Messungen sind in der nachfolgenden Tabelle gelistet.

	JLX					LX2				
P	T_{ges} ms	Durchsatz Bilder/s	Durchsatz MB/s	Speed-up	Effizienz	T_{ges} ms	Durchsatz Bilder/s	Durchsatz MB/s	Speed-up	Effizienz
1	170414	7,6	13,19	1,00	1,00	60606	21,3	44,43	1,00	1,00
2	90683	14,3	24,79	1,88	0,94	30201	42,8	89,17	2,01	1,00
3	60575	21,3	37,11	2,81	0,94	20202	64,0	133,30	3,00	1,00
4	47814	27,0	47,02	3,56	0,89	15179	85,2	177,42	3,99	1,00
5	40513	31,9	55,49	4,21	0,84	13369	96,7	201,44	4,53	0,91
6	35319	36,6	63,65	4,82	0,80	13385	96,6	201,20	4,53	0,75
7	32136	40,2	69,95	5,30	0,76	13291	97,3	202,62	4,56	0,65
8	29780	43,4	75,49	5,72	0,72	13354	96,8	201,66	4,54	0,57
9	28798	44,9	78,06	5,92	0,66	13276	97,4	202,85	4,57	0,51
10	29453	43,9	76,32	5,79	0,58	13151	98,3	204,78	4,61	0,46
12	29484	43,9	76,24	5,78	0,48	13416	96,4	200,73	4,52	0,38

Tabelle 19: JLX und LX2 Decoder-Leistung einschl. I/O mit unterschiedlichen Parallelisierungsgrad.

Da der LX2 Decoder mit der eingesetzten SSD offensichtlich I/O gebunden ist, wurde eine virtuelle Festplatte im Systemspeicher (RAM-Disk) eingerichtet und eine zusätzliche Messreihe durchgeführt:

	LX2 mit RAM-Disk				
P	T_{ges} ms	Durchsatz Bilder/s	Durchsatz MB/s	Speed-up	Effizienz
1	40342	32,1	66,75	1,00	1,00
2	20186	64,1	133,41	2,00	1,00
3	13463	96,0	200,03	3,00	1,00
4	11248	115,0	239,42	3,59	0,90
5	10155	127,3	265,19	3,97	0,79
6	9251	139,8	291,10	4,36	0,73
7	8658	149,3	311,04	4,66	0,67
8	8034	160,9	335,20	5,02	0,63
9	7800	165,8	345,26	5,17	0,57
10	7893	163,8	341,19	5,11	0,51
12	7894	163,8	341,15	5,11	0,43

Tabelle 20: LX2 Decoder-Leistung mit RAM-Disk.

Für eine bessere Übersicht sind Ergebnisse beider Tabellen in den nachfolgenden Diagrammen grafisch abgebildet:

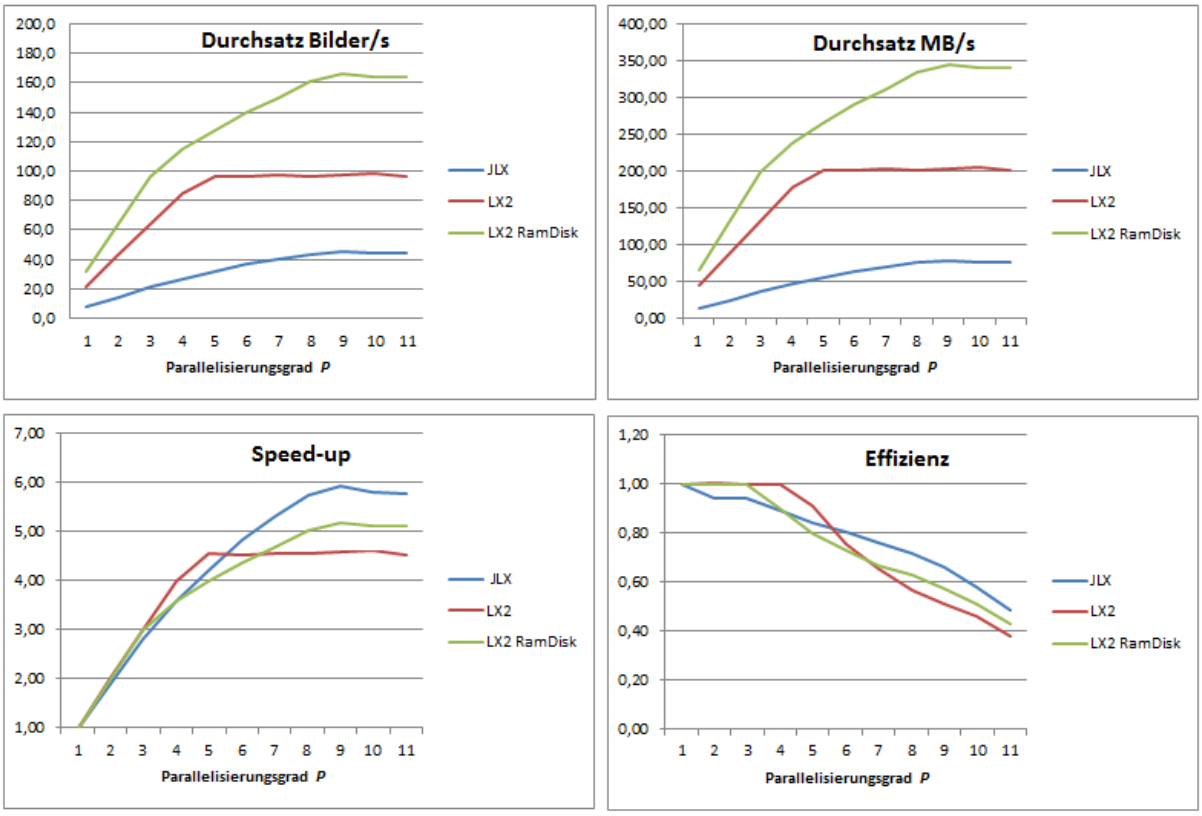


Abbildung 24: Diagramme der gemessenen Leistungswerte aus Tabellen 19 und 20.

Die temporäre Parallelisierung skaliert sowohl beim JLX als auch beim LX2 sehr gut mit steigenden Parallelisierungsgrad. JLX erreicht einen maximalen Speedup von 5,92. Angesichts dessen, dass die CPU nur 4 reale (8 virtuelle) Kerne besitzt, ist das ein

ausgezeichnetes Ergebnis. Allerdings wird I/O Bandbreite von JLX maximal zu 37% gefordert, damit ist JLX auf diesem Testsystem CPU gebunden. Erster Knick beim Übergang vom Parallelisierungsgrad 4 zu Parallelisierungsgrad 5 ist durch Kernvirtualisierung bedingt. Der maximale Durchsatz ist erst mit 9 Threads erreicht – ein Beleg dafür, dass ein zusätzlicher Decoder-Thread die unbenutzte CPU Zeit während der I/O Phasen mit Berechnungen füllen kann (Abschnitt 5.2.1). Doch mit steigenden Parallelisierungsgrad nimmt die Effizienz des Decoders erwartungsgemäß langsam ab. Ein Grund dafür könnte darin liegen, dass immer mehr Threads um gemeinsame Ressourcen (Systembusse, Cache, etc.) konkurrieren.

Auf der anderen Seite, forderte LX2 die eingesetzte SSD bereits ab dem Parallelisierungsgrad 5 an die Grenzen, dabei wurde 33%-ge Auslasung aller Prozessorkerne beobachtet. Damit ist LX2 stark I/O gebunden. Erst mit RAM-Disk konnte diese I/O Limitierung überweunden werden.

5.2.4 Segmentbasierte Parallelisierung

Einen anderen Weg zur Parallelisierung eröffnet die Aufteilung der Bilder in Segmente. Die Segmente werden jeder für sich individuell und unabhängig von allen anderen kodiert. Die resultierenden Codewörter sind von unterschiedlicher (variabler) Länge und werden von einer mehrstufigen Multiplexer-Anordnung nacheinander in FIFO Puffer abgelegt, von wo sie zum Zielsystem transportiert und im geeigneten Format auf Datenträgern abgelegt werden (Abschnitt 2.1).

Vor dem Dekodieren eines Bilds wird der entsprechende Block zunächst in einen Puffer im Systemspeicher geladen. Die Idee der segmentbasierten Parallelisierung besteht nun darin, eine Menge von Decoder-Threads zu erzeugen und die Dekodierungsarbeit unter diesen zu verteilen, dabei muss die Threadanzahl nicht zwangsweise der Segmentanzahl entsprechen. Daraus ergibt sich folgende Eigenschaft: ist Threadanzahl gleich der Segmentanzahl, arbeitet jeder Thread immer auf demselben Segment, andernfalls liegt eine Verschiebung vor und jeder Thread wird an unterschiedlichen Segmenten arbeiten.

Da jedes Segment eigene Kontext-Liste mit je 365 Elementen hat, muss eine entsprechende Datenstruktur angelegt werden (z.B.: ein zweidimensionales Array). Für die Ausgabe wird zusätzlich ein Bild-Puffer benötigt. Die Threads erhalten als Startparameter alle die gleichen Zeiger auf Block-Puffer, Bild-Puffer und Kontext-Struktur. Der Bildpuffer muss von allen Threads beschrieben werden können, unter C++ muss er dafür als *volatile* deklariert sein. Als zusätzlichen Startparameter erhält jeder Thread eine eindeutige fortlaufende Nummer (*TID*) und die Anzahl der erzeugten Threads. Aus diesen beiden Werten kann sich jeder Thread identifizieren und ableiten, welche Bildpunkte von ihm zu dekodieren sind. Dabei darf jeder

Bildpunkt nur von einem bestimmten Thread genau einmal Dekodiert werden. Eine solche Zuordnung kann unterschiedlich gewählt werden, musst aber eindeutig sein.

Wenn die Decoder-Threads alle gestartet werden, erreichen sie nach einer kurzen Initialisierungsphase die innere Schleife des Decoder-Algorithmus. Dort können sie Schritte für Vorhersage, Quantisierung, Kontextbestimmung und Berechnung des Parameters *k* durchführen, müssen aber spätestens beim Dekodieren von *MErr* das entsprechende Codewort einlesen. An dieser Stelle ist das Offset nur dem ersten Thread bekannt. Alle anderen Threads müssen zunächst warten.

Sobald der erste Thread den Golomb-Code eingelesen und dekodiert hat, ist dessen Länge bekannt. Die dafür mindestens notwendige Schritte sind im Dekoder-Ausschnitt (Abb. 25) als rote Pfeile eingezeichnet. Das Offset des nächsten Codeworts ergibt sich durch einfache Addition dieser Länge auf den bekannten Offset des Vergängers. Der neue Offset muss schnellstmöglich an den, für das nächste Codewort zuständigen, Thread weitergegeben werden. Für diesen Zweck wird eine *Offset-Liste* mit je einem Offset-Element pro Thread angelegt. Während der Initialisierung setzt jeder Thread den Wert des eigenen Offset-Elements auf 0 und bestimmt das Offset-Element des Nachfolgers.

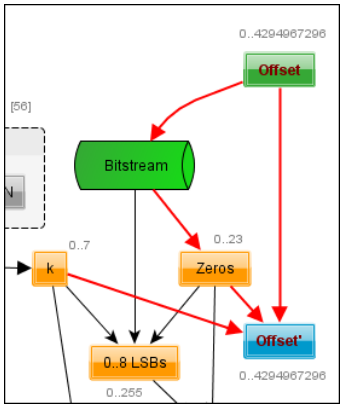


Abbildung 25: Decoder-Ausschnitt für Offsetberechnung.

TID des Nachfolger lässt sich mit folgender Formel bestimmen (N_T ist die Threananzahl):

$$TID_{i+1} = (TID_i + 1)\ mod\ N_T$$

Zusätzlich verwendet jeder Thread eine interne Offset-Variable. Beim ersten Thread wird diese am Anfang zu 1 und bei allen anderen zu 0 gesetzt. Das Vorgehen wird anhand der Abb. 26 erklärt. Um festzustellen, ob ein neuer Offset vom Vorgänger-Thread in der Offset-Liste bereitliegt, vergleicht jeder Thread ständig den intern gespeicherten Offset mit dem Wert seines Listeneintrags:

```
while (InternalOffset <= ListOffset) { do nothing }
```

Diese Konstruktion ist ein sogenannter *waiting spin loop*. Zu Beginn kann nur der erste Thread *spin loop* passieren. Daraufhin wird der übegebene Offset aus der Liste in die interne Offset-Variable übernommen (1) und das Codewort an diesem Offset dekodiert (2). Sobald das nächste Offset bekannt ist, wird dieser in die Offset-Liste für den nächsten Thread gespeichert (3) und mit anderen Dekodierungsschritten weitergemacht. Der nächste Thread

ließt aus der Offset-Liste jetzt einen größeren Offset als das intern gespeicherte. Er kann nun das *spin loop* auch passieren, Offset übernehmen (4) und das nächste Codewort einlesen (5). Dieses Vorgehen wird fortgesetzt bis das gesammte Bild dekodiert ist.

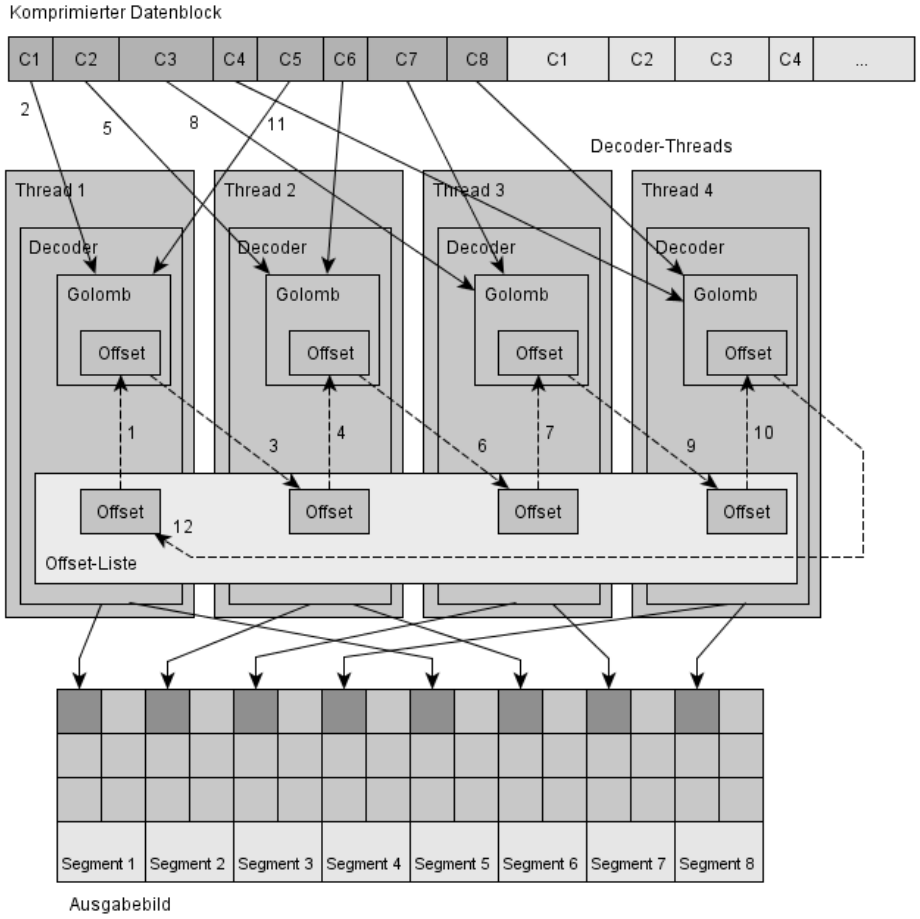


Abbildung 26: Prinzip der segmentbasierten Dekodierung.

Nach der Übernahme des Offsets in die interne Variable und dem Einlesen des Codeworts wird die Codewort-Länge auf den internen Offset draufaddiert und dieser Wert an den nächsten Thread weitergegeben. Ab diesem Zeitpunkt ist die Bedingung für *spin loop* solange erfüllt, bis ein größerer Listen-Offset als das intern gespeicherte eingelesen wird. Das kann nur dann passieren, wenn ein *Dekodierungs-Zyklus* über alle Nachfolgerthreads abgeschlossen ist und der Vorgänger das korrekte Offset für das aktuelle Codewort in die Offset-Liste schreibt. Insbesondere können Werte der Offset-Liste nur zunehmen, und zwar genau einmal pro inneren Schleifendurchlauf.

Wenn man die getrennten Phasen der Offset-Verarbeitung, Dekodierung (restliche Schritte) und des Spinlock-Wartens für alle Decoder-Threads zeitlich einzeichnet entsteht eine Pipeline (Abb. 27). Die gerichteten Kanten zeigen Richtung der Offsetweitergabe. Die Offsets-Werte müssen von einem Threads zum nächsten weitergereicht werden. Ein Thread kann die

Dekodierung eines Bildpunktes nicht abschließen, bis er das Offset vom Vorgänger bekommt. Dies führt dazu, dass die Threads gewisse Zeit im aktiven wartenden Zustand verbringen können. Ob das passiert und wie lange das Warten andauert muss von der Dauer der Offsetberechnung und Offsetweitergabe abhängen. Die Offsetphase (O) ist der kritische Abschnitt der Pipeline, sie kann zu jeder Zeit nur in einem Thread ausgeführt werden – es ist keine Überlappung möglich. Je kleiner diese Phase in Relation zu Dekodierung (D) ist, desto kleiner dürfte die Wahrscheinlichkeit und Dauer des *spin loop* Wartens ausfallen.

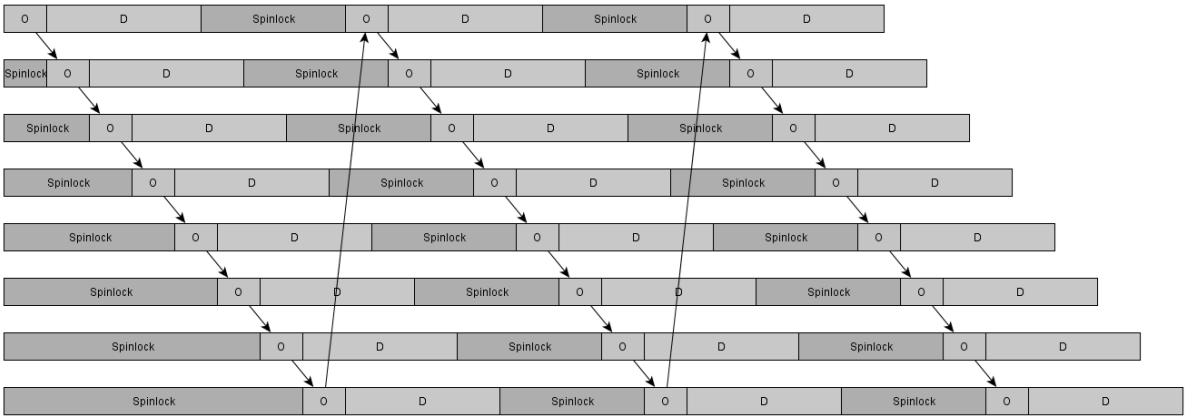


Abbildung 27: Segmentbasierte Dekodierungs-Pipeline.

Die Offsetweitergabe ist der einzige Synchronisationspunk des Algorithmus (*single point of synchronization*), diese Synchronisation muss pro Bildpunkt (bzw. pro Codewort) durchgeführt werden. Wäre der Decoder in Hardware implementiert, mit je einer dedizierten Decoder-Pipeline pro Segment, könnten die Offsets sofort über Direktverbindungen zwischen den Pipelines weitergereicht werden.

Unter der x86 Architektur muss der Thread entweder auf Offset warten (*busy waiting*) oder die Ausführung anhalten (*suspend*). Im letzten Fall werden zwar die Ressourcen für andere Threads freigegeben, ein Aufwecken aus dem suspendierten Zustand durch Betriebssystem-Scheduler würde aber zu lange dauern. Das aktive Warten ist daher die einzige sinnvolle Möglichkeit für derart kurze Durchlaufzeiten zwischen den Offsetsynchronisationen.

Interessanterweise entsteht in der vorgestellten Offset-Weitergabe keine Wettlaufsituation (*race condition*). Jedes Offset in der Offset-Liste wird nur von einem bestimmten Thread gelesen und vom andern bestimmten Thread geschrieben. Die Offset-Weitergabe erfolgt in einer eindeutigen (deterministischen) Weise selbst dann, wenn Lese- und Schreiboperationen der parallelen Threads in verschiedener Reihenfolge stattfinden. Wegen dieser Eigenschaft sind weder kritische Bereiche (*critical* sections) noch das Sperren (*lock*) der Offset-Liste notwendig. Auf Cache-basierten Multiprozessoren sind *spin loops* die schnellste bekannte [13] Methode zur Synchronisation von zeitlich schnell ablaufenden Vorgängen und ist im

vorliegenden Fall die einzig sinnvolle Lösung. Die Threads müssen also in einer Warteschleife (*busy waiting loop*) warten und kontinuierlich neues Offset abfragen. Da die Offset-Liste im Systemspeicher liegt, wird die Synchronisationsdauer vor allem durch Speicherzugriffslatenz und Cache-Organisation bestimmt.

5.2.4.1 Ergebnisse

Um die Effizienz der segmentbasierten Parallelisierung zu bewerten, werden wieder Messungen mit unterschiedlichen Parallelisierungsgraden durchgeführt. Im Unterschied zur temporären, ist segmentbasierte Parallelisierung intern im Decoder implementiert. Dabei wurde temporäres Scheduling durch kritische Bereiche (*critical section*) effektiv serialisiert. Das Testsystem und die Aufnahme entsprechen denjenigen aus der temporären Parallelisierung. Mit dem gleichen Testsystem und Daten, wie auch bei der temporären Parallelisierung, wurden nachfolgenden Ergebnisse erzielt:

	JLX segmentbasiert				
P	T_{ges} ms	Durchsatz Bilder/s	Durchsatz MB/s	Speed-up	Effizienz
1	165579	7,8	13,58	1,00	1,00
2	217371	5,9	10,34	0,76	0,38
3	303546	4,3	7,41	0,55	0,18
4	464069	2,8	4,84	0,36	0,09
5	621157	2,1	3,62	0,27	0,05
6	726148	1,8	3,10	0,23	0,04
7	790798	1,6	2,84	0,21	0,03
8	1000523	1,3	2,25	0,17	0,02

Tabelle 21: Messergebnisse der segmentbasierten JLX Parallelisierung.

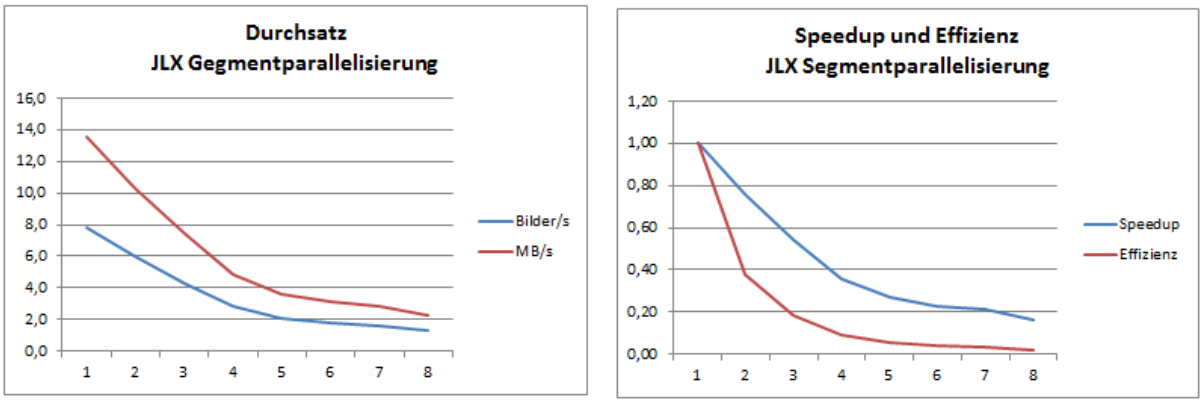


Abbildung 28: Leistungsdiagramme der segmentbasierten JLX Parallelisierung.

Die segmentbasierte Parallelisierung der verschränkten Bilddaten brachte keinen Speed-up. In Gegenteil, die Leistung sinkt mit steigenden Parallelisierungsgrad kontinuierlich ab. Die Ursache dafür muss zu einem in der Synchronisation selbst zu finden sein. Untersuchungen zu Spin-Locks [13][14] haben gezeigt, dass beim Beschreiben der Synchronisationsvariablen

ihre Cache-Kopien verworfen werden (*cache line invalidation*) und es mehrere Buszyklen dauert bis die Änderungen an alle Kerne weitergegeben werden. Um den Synchronisationsoverhead zu reduzieren, wurden in beiden Untersuchungen verschiedene Optimierungsmethoden für *spin locks* vorgestellt. Die Decoder-Pipeline hat aber noch ein weiteres Problem: die Verarbeitungsreihenfolge der Bildpunkte bzw. Codewörter ist strikt verkettet. Wird ein Decoder in dieser Kette angehalten, können alle nachfolgenden Decoder-Threads den *spin loop* solange nicht passieren, bis die Ausführung des angehaltenen Threads nicht fortgesetzt wird. Die Ursache für das Anhalten eines Threads ist das Prozess-Scheduling des Betriebssystems. Die Threads werden dabei auf der CPU für eine vorgegebene Dauer ausgeführt, dann angehalten und die nächsten Threads für die Ausführung bestimmt. Neben den Decoder-Threads muss das Betriebssystem noch andere Dienste und Hintergrundprozesse bedienen. Auch das Scheduling selbst dauert gewisse Zeit. Ferner müssen vom Prozessor Hardware- und Software-Unterbrechungen (*Interrupts*) mit höherer Priorität bedient werden. Typische Werte liegen bei 100 bis 1000 Unterbrechungen pro Sekunde (Quelle: Microsoft TechNet). Mit steigender Anzahl der Decoder-Threads nimmt Wahrscheinlichkeit und Dauer der Pipeline-Unterbrechungen stark zu.

Während der Dekodierung eines einzigen 3 Megapixel Bildes müssen 3 Millionen Synchronisationen durchgeführt werden. Im Fall von Parallelisierungsgrad 2 führen Decoder-Threads bereits 17,7 Millionen Synchronisationen pro Sekunde durch. Mit zunehmenden Parallelisierungsgrad steigt der Synchronisationsaufwand an und die Ausführung wird immer öfter und länger angehalten. Offensichtlich ist der erforderliche Synchronisationsaufwand in Relation zu den restlichen Dekodierungsschritten zu groß.

Um die Problematik der Pipeline-Anhaltung zu verdeutlichen wurde eine spezielle Zusatzmessung durchgeführt: Die Anwendung wurde hierbei mittels *process affinity* auf 7 Prozessorkerne limitiert und das Dekodieren eines einzigen Bildes mit Parallelisierungsgrad 8 durchgeführt. Die Dekodierungszeit brach dabei zusammen (nach 10 Minuten war das Bild nicht fertig, die Messung musste abgebrochen werden). Da keine 8 Threads auf 7 CPU-Kernen gleichzeitig ausgeführt werden können, war zu jeder Zeit mindestens 1 Thread angehalten. Als Folge befindet sich die Pipeline fast die gesamte Zeit im *spin loop*. Der segmentbasierte Parallelisierungsgrad darf also die Anzahl der Prozessoren nicht überschreiten. Temporäre Parallelisierung war in dieser Hinsicht nicht beschränkt.

Zu den wenigen Stärken der (verschränkten) segmentbasierten Parallelisierung zählt lediglich der Speicheraufwand. Bei der temporären Parallelisierung steigt der Speicherbedarf für Block- und Bild-Puffer linear mit steigenden Parallelisierungsgrad. Bei der segmentbasierten

Parallelisierung steigt nur der Speicherbedarf für Kontexte, diese sind aber im Vergleich zu Bild- und Blockpuffer vernachlässigbar klein.

5.3 GPGPU (CUDA)

Für GPU bieten sich zunächst beide vorgestellten Parallelisierungsstrategien. Um die vielen CUDA-Kerne (bis zu 512 bei Fermi) der GPU auszulasten und Speicherzugriffslatenz zu verstecken, muss der Decoder einen sehr hohen Parallelisierungsgrad erreichen. (Siehe Abschnitt 2.5). Die segmentbasierte Parallelisierung mit Parallelisierungsgrad 8 kann diese Bedingung von Anfang an nicht erfüllen, könnte aber mit der temporären Parallelisierung (z.B. vom Grad 50) kombiniert werden, um einen deutlich höheren Gesamtparallelisierungsgrad zu erreichen.

5.3.1 Decoder-Kernel

Zunächst wird segmentbasierte Parallelisierung betrachtet und CPU Implementierung der Decoder-Funktion in einen CUDA-Kernel überführt. Die Daten müssen hierbei im GPU RAM untergebracht werden. Die benötigten Puffer im GPU Speicher werden daher vor der Kernelausführung angelegt und die zu dekodierende Datenblöcke via *cudaMemcpy* hineinkopiert. Nach der Kernelausführung können Puffer mit fertigen Bilddaten zurück zum *Host* kopiert oder aber von anderen Kernel direkt weiterverwendet werden. Die aktuelle CUDA 4.0 API erlaubt einen Datenaustausch mit OpenGL (z.B. via *Pixel Buffer Objects*), womit ein direktes Rendering der Bildsequenzen ohne Umweg über den Systemspeicher möglich wäre.

Als Ausgangsbasis dient eine, vom segmentbasierten parallelen CPU Decoder abgeleitete, „naive“ Referenzimplementierung des Kernels. Der Kernel entspricht bis auf die Offsetweitergabe weitgehend der CPU Implementierung.

5.3.2 Optimierung

Um einen leistungsfähigen Decoder zu erhalten, muss dieser für GPU Hardware zunächst optimiert werden. Um die Wirkung der Optimierungen besser zu verfolgen, wird dieser Schritt ohne Parallelisierung mit einem sequentiellen Decoder durchgeführt. Als Testsystem kommt zunächst das langsamere Testsystem 1 (GeForce G210M, 16 Computing-Kerne) zum Einsatz. Gemessen wird Kernel-Dekodierungszeit (ohne I/O und Datentransfer zwischen Host und GPU) mit einem einzigen CUDA-Thread.

Die sequentielle Dekodierung mit Referenzdecoder benötigt hierzu 22833ms. Diese Latenz ist eindeutig zu hoch und muss unbedingt verbessert werden.

Als erstes bieten sich wieder die zweigbehafteten Gradienten-Berechnungen zur Optimierung. Die Referenzimplementierung verwendet bereits die bewährten Look-Up Tabellen (Abschnitt 4.2.3), allerdings liegen diese in Global Memory und profitiert damit nicht vom on-chip Cache. Die drei Quantisierungs-LUTs sind zusammen 3KB groß und naturgemäß nur lesbar (*read-only*). Für sie eignet sich besonders Constant Memory, welcher von on-chip Cache unterstützt wird.

Wenn die Quantisierungs-LUTs vom Host in den Constant Memory verlegt werden, reduziert sich die Kernelausführungszeit um 9,2% auf 20736ms.

Ein weiterer Datenblock, dass vom Decoder nur gelesen wird, ist der Block-Puffer mit komprimierten Bilddaten. Dieser ist für Constant-Memory (65KB) zu groß, hier bietet sich nur Textur-Speicher (*Texture Memory*) an. Die Texture Memory wird ebenfalls von on-chip Cache unterstützt und da die Codewörter linear eingelesen werden, ist hier besonders gute Cache-Lokalität gegeben. Die Zugriffe auf Texturspeicher müssen dabei über die Shader-Funktion *tex1Dfetch* erfolgen. Das Codewort kann zusammen mit BitOffset bis zu 3 Byte im Texturspeicher belegen – es werden insgesamt 3 Textur-Ladeoperationen benötigt. Um Offsetumrechnungen in 2D Texturkoordinaten zu vermeiden, wird eine 1D Textur verwendet.

Durch die Verlegung der Block-Puffer in den Texturspeicher dauert Kernelausführung nur noch 16263ms (21,5% weniger).

Ein besonders wichtiger Bestandteil des Decoders ist das Einlesen und Dekodieren der Codewörter. Es ist Teil der Offset-Weitergabe und beeinflusst im höchsten Maß die Wartezeit (*idle*) der Synchronisation. Um das Einlesen zu beschleunigen, wird das Wort komplett in Register eingelesen, um *BitOffset* Stellen nach links verschoben und mit einer speziellen Funktion *__clz()* die Anzahl führender „0“ bestimmt. Diese Funktion ist ähnlich zu *BitScanReverse*, liefert aber sofort die gesuchte Bitanzahl. *__clz()* ist erst in Grafikkarten mit Computing Shade Capability 2.0 und höher in Hardware implementiert, aber auch als Softwareimplementierung liefert sie bessere Performance als das Einzelbit-Einlesen des Referenzdecoders. Schließlich wird Registerinhalt nach rechts verschoben und *MErr* wie bei der CPU Implementierung berechnet.

Optimiertes Einlesen der Codewärter mit __clz() verbessert die Kernelausführungszeit um 22% auf 12686ms.

Das Optimierungspotenzial nimmt ab hier stark ab. Die Berechnung der *min* und *max* Werte vor dem Verzweig im MED Prädiktor bringt nur 0,65% Verbesserung.

Die Berechnung des Golomb-Parameters *k* mit einer Schleife ist nicht Optimal. Dafür lässt sich eine äquivalente schleifenfreie Variante herleiten:

```
k = log2(((A + 31) >> 5) - 1);
```

Auch die, in CUDA Bibliotheken enthaltene, log2 Funktion lässt sich speziell für die Ganzzahlarithmetik des 32 Bit Registers geringfügig Optimieren:

```
static __device__ __forceinline__ int log2(unsigned int a)
{
  return (a) ? (__float_as_int(__uint2float_rz(a)) >> 23) - 126 : 0;
}
```

Diese optimierte Parameter-*k*-Berechnung reduzierte die Kernelzeit immerhin um 5,6% auf 11894ms.

Ersetzung der inversen Fehlerabbildung durch LUT in Constant Memory und Einsatz der *min*, *max* Funktionen im Kontext-Update eliminieren zwei weitere Verzweige und verbessern das Ergebnis schließlich auf 11673ms (1,9% besser).

Die Ergebnisse aller durchgeführten Optimierungsmaßnahmen sind in der nachfolgenden Tabelle zusammengefasst.

	Kernelzeit (ms)	Verbesserung
Referenzimplementierung	22833	
Local gradient computation ersetzt durch LUT in __constant__ memory. Ausnutzung des on-chip Caches für LUT.	20736	+9,2%
Komprimierte Eingabedaten abgebildet auf 1D Textur, Datenzugriffe über tex1Dfetch. Ausnutzung des on-chip Caches.	16263	+21,5%
Hardwareoptimierung des Unary-Code Readers.	12686	+22,0%
Prädiktor Optimierung	12604	+0,65%
Schleifenfreie Berechnung des Parameters k. Optimierung der log2 Funktion für unsigned int.	11894	+5,6%
Inverse error mapping ersetzt durch LUT, Zweigfrei.	11790	+0,9%
Optimierte Kontextaktualisierung	11673	+1,0%

Tabelle 22: Optimierungen des sequentiellen JLX Decoders in CUDA.

Obwohl hier eine Einsteigergrafikkarte mit einem leistungsfähigen Prozessor verglichen wird, wirft das Ergebnis der sequentiellen CUDA Dekodierung Fragen auf. Zur Erinnerung: der sequentielle CPU Decoder brauchte lediglich 122ms (95-mal weniger) Zeit. Einfluss können gleich mehrere Faktoren haben. Offensichtlich ist ein einzelner Streaming Multiprozessor einfacher aufgebaut und erreicht im Gegensatz zu einem komplexen CPU-Kern mit mehrstufiger Pipeline deutlich geringeren sequentiellen Durchsatz. Die fehlende

Sprungvorhersage-Einheit, Cache-Effizienz, hohe Speicherzugriffslatenz und die Art, wie eine SIMT Einheit bedingte Verzweige ausführt, könnten nur einige der möglichen Gründe sein.

Die sequentiellen Ergebnisse müssen an dieser Stelle als gegeben betrachtet werden. Selbst nach der Optimierung ist die Decoder-Latenz noch deutlich zu hoch. Es wird nun untersucht, inwieweit eine segmentbasierte Parallelisierung das verbessern kann.

5.3.3 Segmentbasierte Parallelisierung

Eine segmentbasierte Parallelisierung auf der GPU bedarf im Wesentlichen drei Veränderungen am sequentiellen Decoder:

- Der Kernel wird gleich auf mehreren CUDA-Threads ausgeführt.
- Jedem Thread muss ein Segment mit Kontexten eindeutig zugeordnet werden.
- Synchronisation: die Threads müssen Offsetwerte untereinander austauschen.

Die einfachste Art der Thread-zu-Segment-Zuordnung ist die 1:1 Abbildung. Weniger Threads würden Parallelisierungsgrad einschränken. Anderseits können wegen der Datenabhängigkeiten in der verketteten Offsetweitergabe nicht mehr Threads als die Anzahl der Segmente erzeugt werden. Für die Testaufnahme mit 8 Segmenten werden daher genau 8 Threads erzeugt. Der Kernelaufruf wird mit der Anzahl der Blöcke, Anzahl der Threads und der Größe des Shared Memor Speichers konfiguriert und erhält zusätzlich eine Datenstruktur als Startparameter:

```
DecoderKernel<<< Blocks, Threads, SharedMemorySize >>>(Parameter);
```

Auch unter GPU müssen die Offsets für die Codewörter weitergegeben werden. Zur Offsetweitergabe wird ein gemeinsamer Speicherbereich mit möglichst geringer Latenz benötigt. Der schnellste gemeinsame Speicher ist die Shared Memory. Um Zugriff darauf zu erhalten, müssen alle Threads innerhalb eines Blocks ausgeführt werden. Für die Dekodierung der 8 Segmente eines einzelnen Bilds wird daher 1 Block mit 8 Threads benötigt. Der Speicherplatzbedarf für Shared Memory berechnet sich mit 8 Kontextlisten (je 1460 Bytes, Abschnitt 4.2.1) und einer Offsetweitergabe-Variable (4 Byte, *unsigned int*) zu:

$$8 \cdot 1460 + 4 = 11684 \quad \text{[Byte]}$$

Als Startparameter erhält der Kernel eine komplexe Struktur mit Zeigern auf die, im *device* vorher instanziierten, Block- und Bildpuffer sowie Angaben zur Bildauflösung und der Segmentanzahl. Bei Zeigerverwendung muss stets zwischen *device pointers* und *host pointers* unterschieden werden. Der Versuch einen *device pointer* im Host zu referenzieren führt zum

Zugriff auf falsche Speicherbereiche und mit hoher Wahrscheinlichkeit zur Speicherzugriffsverletzung (*memory access violation*). Ähnliches gilt auch für den umgekehrten Fall.

Der, in der CPU verwendete, *spin loop* lässt sich unter einer SIMT-Architektur nicht mehr einsetzen. Die Weitergabe der Offsets über Shared Memory erfordert CUDA Threadsynchronisation mittels `__syncthreads`, diese lässt sich aber mit divergierenden Ausführungspfaden der voneinander abhängigen Threads nicht durchführen. Der Versuch solche *spin loops* auf der GPU auszuführen führt zur sofortigen Verklemmung (*dead lock*).

Bei der Suche nach einer geeigneten Synchronisationsmöglichkeit wurde Serialisierung als einzige Lösung gefunden. Dabei werden *spin loops* durch eine synchronisierte *for* Schleife ersetzt. In einem Schleifendurchlauf identifiziert sich jeder Thread entweder als für die Offset-Berechnung zuständig oder nicht. Der zuständige Thread liest sein Offset aus dem Shared Memory, dann das Codewort, dekodiert diesen und speichert das neue Offset für den nächsten Thread zurück in den Shared Memory. Alle anderen Threads passieren diesen Schleifendurchlauf ohne Berechnungen. Nach jedem Schleifendurchlauf muss mit `__syncthreads` sichergestellt werden, dass der Nachfolgethread zuerst eine konsistente Sicht auf Shared Memory erhält und erst dann das neue Offset einliest. Dadurch werden Offsetwerte in einer deterministischen Reihenfolge weitergereicht. Für die Offsetübergabe reicht eine einzige Offset-Variable in Shared Memory. Offset-Liste, wie in der CPU Pipeline, wird hier nicht benötigt.

Diese Serialisierung der Offsetweitergabe verändert die parallele Decoder-Pipeline auf eine ungünstige weise (Abb. 29).

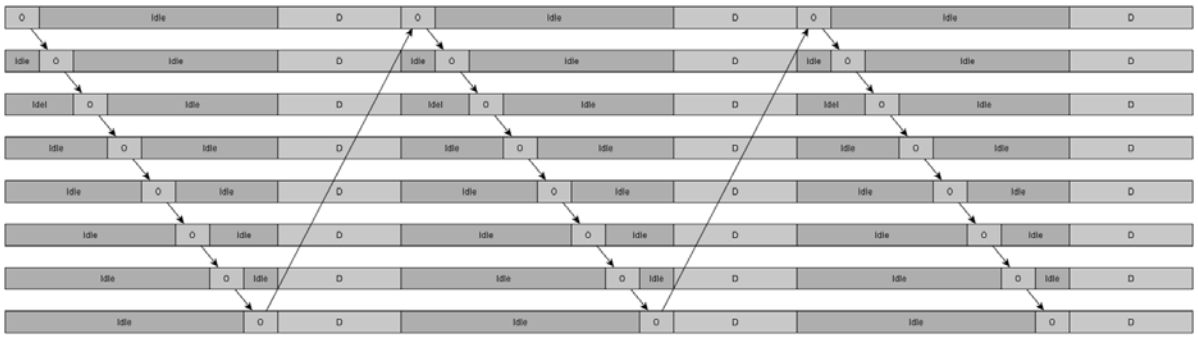


Abbildung 29: Segmentbasierte JLX Decoder-Pipeline auf GPU.

Der Gesamtaufwand für die parallele Pipeline wird durch Serialisierung steigen, da jeder Thread nun das 7-fache der mittleren Offsetberechnungszeit im Wartezustand (*idle*) verbringen muss. Danach ist die Dekodierung bis zur nächsten Offsetweitergabe unabhängig

(*dependency free*). Außer der Offsetsynchronisation sind deshalb keine weiteren Thread-Synchronisationen notwendig.

Da die horizontale Bildauflösung nicht zwangsweise ein Vielfaches der Segmentbreite sein muss, wird der letzte Thread unter anderem auf nicht vorhandenen Bildpunkten außerhalb des Bilds ausgeführt. Für diesen Fall ist eine Sonderbehandlung im Kernel notwendig. Für jeden nicht vorhandenen Bildpunkt muss der letzte Thread alle Decoder-Schritte in der inneren Schleife überspringen, dabei darf die Verkettung der Offsetweitergabe nicht unterbrochen werden. Als Lösung für dieses Problem wird der letzte Thread beim Überspringen eines Schleifen-Durchlaufs nur die synchronisierte *for* Schleife ausführen. In der *for* Schleife wird aber kein Codewort eingelesen und seine Länge auf das weitergereichte Offset aufaddiert, stattdessen wird der Thread diese Schleife tatenlos passieren (zusätzlicher Verzweig) und lediglich `__syncthreads` gemeinsam mit allen anderen Threads am Ende aufrufen. Der nächste Thread liest daraufhin das nicht veränderte Offset aus Shared Memory regulär ein – für ihn ist die Sonderbehandlung völlig transparent. Mit diesem Vorgehen wird eine konsistente und korrekte Reihenfolge der Codewortverarbeitung sichergestellt.

5.3.4 Ergebnisse

Das so angepasste Decoder-Kernel wird mit maximalen Parallelisierungsgrad 8 (limitiert durch 8 Segmente) getestet und mit dem sequentiellen Kernel verglichen. Diesmal findet der Vergleich auf beiden Testsystemen mit unterschiedlichen Grafikkarten statt. Gemessen wird die, für die Dekodierung eines einzelnen Bilds benötigte, mittlere Ausführungszeit des Kernels (ohne I/O). Die Ergebnisse sind in der nachfolgenden Tabelle zusammengefasst.

	Dekodierungsdauer [ms]		Speed-up
	1 Thread	8 Threads	
GeForce G210M	11673	4373	2,67
GeForce GTX560	4774	2706	1,76

Tabelle 23: Dekodierungsdauer mit verschränkter segmentbasierter Parallelisierung (Parallelisierungsgrade 1 und 8) für G210M und GTX560Ti.

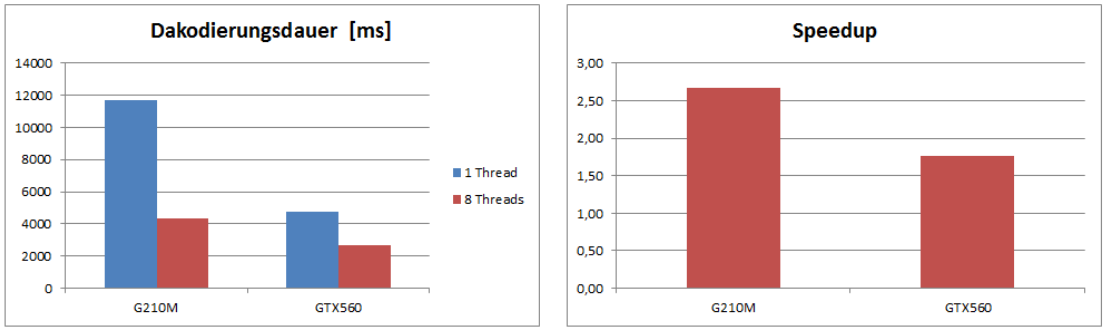


Abbildung 30: Dekodierungsdauer mit verschränkter segmentbasierter Parallelisierung (Parallelisierungsgrade 1 und 8) für G210M und GTX560Ti.

Während mit der CPU Speed-up Abfall gemessen wurde, erreicht GPU immerhin eine Leistungssteigerung durch Parallelisierung. Offensichtlich ist die Offset-Synchronisation über den schnellen on-chip Shared Memory deutlich effizienter als über Systemspeicher. Auch müssen die CUDA Threads nicht mit anderen Prozessen und Hardwareinterrupts um die gemeinsamen Ressourcen konkurrieren. Dadurch wird die Pipeline seltener unterbrochen, als das der Fall bei der CPU war. Doch der erreichte Speed-up zwischen 1.76 und 2,67 ist für Parallelisierungsgrad 8 zu gering. Die Notwendigkeit der Offset-Synchronisation pro Bildpunkt limitiert sowohl CPU als auch die GPU Implementierung deutlich.

Bei der GPU kommt noch ein weiteres Problem hinzu. Zitat aus NVIDIA Parallel Nsight:

„The Microsoft Windows Display Driver Model (WDDM) will reset the NVIDIA display driver if the GPU is unresponsive for over 2 seconds by default. Unfortunately, this means stopping at a GPU breakpoint causes the driver to reset… This feature is called Timeout Detection and Recovery (TDR)…"

Die parallele Dekodierung mit 8 Threads auf einer Fermi Grafikkarte dauert pro Bild 2,7 Sekunden und überschreitet damit die TDR Dauer von 2 Sekunden. Als Folge findet ein *reset* der Grafikkarte durch Betriebssystem statt, was zum Verlust aller Daten im Kernelkontext sowie der Grafikkartenbindung führt. Um die vollständige Dekodierung überhaupt erst möglich zu machen, musste die TDR Schutzvorrichtung des Betriebssystems komplett abgeschaltet werden (z.B. durch Parallel Nsight Tool).

5.3.5 Steigerung des Parallelisierungsgrades

Um die vielen Recheneinheiten der GPU auszulasten, ist für eine CUDA Implementierung ein deutlich höherer Parallelisierungsgrad als 8 notwendig. Zitat aus NVIDIA CUDA C Best Practices Guide [4]:

"The device is ideally suited for computations that can be run on numerous data elements simultaneously in parallel… where the same operation can be performed across thousands, if not millions, of elements at the same time. This is a requirement for good performance on CUDA: the software must use a large number of threads. The support for running numerous threads in parallel derives from the CUDA architecture's use of a lightweight threading model."

Der Gesamtdurchsatz [Bilder/s] könnte durch gleichzeitige temporäre Parallelisierung gesteigert werden. Ca. 960MB von 1024MB VRAM der Test-Grafikkarte sind für die Kernelausführung verfügbar und können für Block und Bild-Puffer verwendet werden. Der maximale Parallelisierungsgrad lässt sich mit Formeln aus Kapitel 5.2.3 berechnen:

$$P_{max} = \frac{960\ [MB]}{3\ [MB] \cdot \left(1 + \frac{4{,}78}{8}\right)} = 200$$

Dadurch können gleich 200 Bilder durch einen einzigen Kernelaufruf dekodiert werden. 200 Bilder mit je 8 Segmenten führen zu einem Gesamtparallelisierungsgrad von 1600. Der hohe Parallelisierungsgrad würde die Speicherzugriffslatenz verstecken und die Streaming-Multiprozessoren besser auslasten. Als Ergebnis ist eine deutliche Steigerung des Gesamtdurchsatzes zu erwarten. Doch die Kombination mit der temporären Parallelisierung ändert nichts am Hauptproblem der GPU Implementierung: die Latenz für die Dekodierung eines Bilds kann dadurch nicht unter 2,7 Sekunden sinken – die TDR Dauer bleibt überschritten. Aber auch ohne Betrachtung der TDR Dauer ist die Latenz zu groß: so wird die Anwendung bzw. der Benutzer nach einem Sprung (*seek*) in der Aufnahme 2,7 Sekunden warten müssen, bis das erste Bild angezeigt wird. Die CPU liefert das dekodierte Bild bereits nach 122ms (Tabelle 16).

Auf einer zusätzlichen temporären GPU Parallelisierung wird hier daher verzichtet. Zuerst müssen die Hauptprobleme des Kompressionsverfahrens in weiteren Untersuchungen gelöst werden. Das sind im Wesentlichen zwei:

- Notwendigkeit der Synchronisation durch die Segmentverschränkung
- Zu niedriger Segmentierungs- und damit Parallelisierungsgrad für eine GPU

Die Segmentverschränkung führt sowohl unter CPU als auch unter GPU zur Notwendigkeit der teuren Synchronisation. Der niedrige Segmentierungs- bzw. Parallelisierungsgrad resultiert zusammen mit Synchronisation in einer nicht akzeptablen Decoder-Latenz der GPU-Implementierung. Dabei ist eine Steigerung des Segmentierungsgrades nicht trivial. Vor „Fermi“ Architektur verfügte jeder Streaming Multiprozessor 16KB Shared Memory, erst mit „Fermi“ ist diese Größe auf 64KB gestiegen [22]. Der aktuelle Speicherplatzbedarf für Kontexte (11,6KB bei 8 Segmenten) liegt bereits sehr nahe an diesen Grenzwerten. Eine Verdopplung oder Vervierfachung des Segmentierungsgrades wäre erst mit „Fermi“ Architektur ohne weiteres möglich.

Abschließend noch eine Überlegung zum gleichzeitigen Einsatz der CPU und GPU Decoder. Wenn die GPU einen hohen Durchsatz erreichen sollte, die Latenz aber in Vergleich zu CPU immer noch zu hoch ist, könnten die Stärken der beiden Decoder kombiniert werden. Die CPU kann das angeforderte Bild und einige Nachfolger schon sehr früh bereitstellen, während die GPU die darauffolgenden Bilder mit hohem Durchsatz in Voraus berechnet. Dadurch würde CPU Decoder die hohe Latenz des GPU Decoders verstecken. Die Durchsätze der beiden Decoder ließen sich so zu einem höheren Gesamtdurchsatz zusammenschließen.

6 Zusammenfassung

Das Ziel dieser Ausarbeitung ist die Maximierung des Decoder-Durchsatzes für eine effiziente Bereitstellung von Hochgeschwindigkeitsaufnahmen. Der Schwerpunkt liegt auf Aufnahmen mit 3 Megapixeln Auflösung und einer Bildrate von 485 Bilder/s, welche bedingt durch den Hardware-Encoder des Kamera-Systems in einem modifizierten JPEG-LS Verfahren komprimiert sind. Die Bilder werden dabei in 8 Segmente aufgeteilt und jedes Segment separat kodiert. Allerdings werden die Codewörter der 8 resultierenden Datenströme durch mehrstufige Multiplexer-Anordnungen zu einem Gesamtdatenstrom miteinander verschränkt.

Nach einer kurzen Einführung in die JPEG-LS Modifikationen und die vorliegenden Hardware-Architekturen, wurde in Kapitel 4 die Wahl einer effizienten Speicherungsmethode behandelt. Im Gegensatz zu Dateisequenzen, ist der Mehraufwand bei einer einzelnen Datei deutlich geringer. Die eingeführte Blockindizierung erlaubt dabei direkten und wahlfreien Bildzugriff.

Im gleichen und im nachfolgenden Kapitel wurden jeweils für CPU und GPU hardwarespezifische Optimierungsmaßnahmen untersucht. Vor allem durch den Einsatz von Look-up Tabellen, zweigfreien Ausführungspfaden, speziellen Hardware-Operationen sowie durch die Ausnutzung der Cache-Lokalität konnte der Durchsatz mehr als verdoppelt werden. Dies war eine wichtige Vorarbeit, um auch bei der nachfolgenden Parallelisierung einen möglichst hohen Durchsatz zu erreichen.

In Kapitel 5 wurden zwei verschiedene Strategien der Parallelisierung verfolgt. Bedingt durch die Datenunabhängigkeit und einen geringen Verwaltungsaufwand, erreicht die temporäre Parallelisierung einen Durchsatz von 44,9 Bilder/s bei einem, für das Testsystem nahezu optimalen, Speed-up von 5.92. Der benötigte Speicherplatzbedarf für die Puffer erweist sich für aktuelle Rechnersysteme als vertretbar.

Mit segmentbasierten Parallelisierung konnte kein Speed-up auf der CPU erreicht werden. Die Leistung bricht mit steigenden Parallelisierungsgrad ein. Dabei wurde beobachtet, dass die gesamte Dekodierungs-Pipeline durch Hardware- und Software-Unterbrechungen oft angehalten wird.

Auf der GPU kann, je nach Grafikkarte, lediglich ein moderater Speed-up zwischen 1,76 und 2,67 erreicht werden, dabei ist die benötigte Kernel-Dekodierungszeit mit 2,7 Sekunden für die vorliegende Auflösung zu hoch. Um die GPU Implementierung um temporäre Parallelisierung zu erweitern, müssen zunächst die Probleme der segmentbasierten

Parallelisierung gelöst werden. Die temporäre Parallelisierung alleine hat mit einer inakzeptablen Latenz von 4,7 Sekunden pro Bild zu kämpfen.

Die Probleme der segmentbasierten Parallelisierung entstehen in erster Linie durch Verschränkung der eigentlich unabhängigen Bitströme der Segmente. Das führt zur Notwendigkeit einer Synchronisation pro Codewort bzw. Bildpunkt – der Mehraufwand für über 10 Mio. Synchronisationen pro Sekunde erweist sich dabei als zu hoch. Für dieses Problem muss unbedingt eine Lösung gefunden werden.

Außerdem ist Parallelisierungsgrad 8 bei der segmentbasierten Parallelisierung für eine GPU Architektur zu niedrig. Um die hohe Latenz auf annehmbare Werte zu senken, bedarf es kleinere Segmente und damit einen höheren Segmentierungsgrad. Um Parallelisierungsgrad und Gesamtdurchsatz weiter zu steigern, kann temporäre Parallelisierung hinzugenommen werden, dies ist aber erst dann sinnvoll, wenn die Latenz der segmentbasierten Parallelisierung auf deutlich unter 2 Sekunden gesunken ist.

Betrachtet man das gewählte JPEG-LS Verfahren hinsichtlich der Visualisierung von Hochgeschwindigkeitsaufnahmen aus der Vogelperspektive, so stellt sich fest, dass der Algorithmus, selbst nach Optimierung und Parallelisierung, unter der verwendeten Hardware stark Rechengebunden (*CPU bound*) ist. Im Kapitel 4.3 wurde daher ein Versuch gewagt das Kompressionsverfahren konsequent zu vereinfachen. Durch Wegfall der Kontext-Berechnungen und andere Modifikationen verliert das neue Verfahren (LX2) je nach Bild zwischen 0,6% und 12% an Kompressionsstärke, erreicht aber mit bis zu 165,8 Bilder/s einen um Faktor 3,75 höheren Durchsatz im Vergleich zum modifizierten JPEG-LS. Betrachtet man nur die reine Rechenzeit (ohne I/O und Scheduling) ist der Durchsatz sogar um Faktor 8 höher. In Fällen, wo Durchsatz wichtiger als Kompressionsstärke ist, können also durchaus neue Wege gegangen werden. Deren Erforschung und Verifizierung könnte für die zukünftigen Arbeiten ein spannendes Thema werden.

Literaturverzeichnis

[1] Joachim Holzfuss: Analoge und Digitale Hochgeschwindigkeitskinematographie, Technische Universit□at Darmstadt (2004)

[2] Anto Joys Yesuadimai Michael: Efficient Context Modelling and Segmentation for Parallel Lossless Image Compression, Master Thesis, Universität Stuttgart - IPVS (2011)

[3] Information technology – Lossless and near-lossless compression of continuous-tone still images – Baseline, ISO-14495-1 / ITU-T Recommendation T.87 (1998)

[4] Henrique S. Malvar: Adaptive Run-Length / Golomb-Rice Encoding of Quantized Generalized Gaussian Sources with Unknown Statistics, Microsoft Research

[5] NVIDIA CUDA C Programming Guide, Version 4.0, 5/6/2011 http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[6] Marcelo W., Gadiel S., Guillermo S. Hewlett-Packard Computer Systems Laboratory: The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS (1998)

[7] Neri Merhav: Optimal Prefix Codes for Sources with Two-Sided Geometric Distributions, Electrical Engineering Department Technion (1996)

[8] Kind, Tobias: RAMDISK Benchmarks, University of California (2011)

[9] Tom's-Hardware Online-Magazin: Performance Charts Festplatten (2011)

[10] Torbjörn Granlund: Instruction latencies and throughput for AMD and Intel x86 processors (2011)

[11] Trent Rolf: Cache Organization and Memory Management of the Intel Nehalem Computer Architecture, University of Utah Computer Engineering (2009)

[12] G. Alefeld, I. Lenhardt, H. Obermaier: Parallele Numerische Verfahren, Springer Verlag

[13] Thomas E. Anderson: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 1990

[14] Vladimir Cakarevic u. a.: Understanding the overhead of the spin-lock loop in CMT architectures. Barcelona Supercomputing Center (BSC).

[15] NVIDIA: CUDA C Best Practices Guide, (2011)

[16] Juergen Gall u.a.: Model-based Motion Capture for Crash Test Video Analysis, Max-Planck-Institute for Computer Science

[17] Z.Deng u.a.: Study of Fish Response Using Particle Image Velocimetry and High-Speed, High-Resolution Imaging, U.S: Department of Energy (2004)

[18] Southern Vision Systems, Inc.: Using High-Speed Cameras for Sports Analysis

[19] W. Lauterborn: Acoustic Turbulence. Frontiers in Physical Acoustics, D. Sette, Ed., North Holland, Amsterdam (1986).

[20] Technischer Datenblatt: LUPA 3000: 3-MegaPixel High Speed CMOS Sensor, Cypress Semiconductor Corporation (2009)

[21] Dr. David Levinthal: Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors, Intel Corporation

[22] NVIDIA Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf