Universität Bielefeld
Technische Fakultät
AG Technische Informatik

Bachelor-Arbeit

Beschleunigte Implementierung der Berechnung des Scale Plane Stacks beim Min-Warping-Verfahren

im Studiengang
Kognitive Informatik

Moritz Pflanzer

8. August 2014

Betreuer
Prof. Dr. Ralf Möller
M.Sc. David Fleer

Zusammenfassung

Min-Warping (Möller et al., 2010) ist ein Verfahren zur visuellen Navigation mobiler Roboter. Gerade auf diesen stehen in der Praxis nur beschränkte Rechenkapazitäten zur Verfügung, wodurch das Thema dieser Arbeit, die durchgeführten Berechnungen zu beschleunigen, motiviert ist. Dies soll mittels des neuen Befehls `_mm_sad_epu8` zur Berechnung des Bildabstandsmaßes und die daraus resultierende, höhere Parallelität erreicht werden. Im Rahmen der Arbeit wird zu Beginn eine kurze Einführung in die theoretischen Grundlagen des Verfahrens und der genutzten SIMD-Befehle gegeben. Anschließend werden die Abläufe der bisherigen und der angepassten Version miteinander verglichen, wobei insbesondere auf die Änderungen am Programmcode eingegangen wird. Schlussendlich werden die Laufzeit und die Qualität des Verfahrens bestimmt und ausgewertet.

Diese Bachelor-Arbeit wurde von Prof. Dr. Ralf Möller betreut und begutachtet. Zweiter Gutachter war M.Sc. David Fleer.

Besonders hervorzuheben sind die Unterstützung und das Engagement von Prof. Dr. Ralf Möller. Sein großes Interesse und die Relevanz des Themas dieser Arbeit, haben zu einer gelungenen Bearbeitung beigetragen. Außerdem danke ich meinen Eltern und meiner Freundin für ihre Mitarbeit und Mühe bei der Korrektur der Ausarbeitung.

Hiermit versichere ich, dass ich diese Bachelor-Arbeit selbständig bearbeitet habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und entsprechende Zitate kenntlich gemacht.

Bielefeld, den 8. August 2014

Moritz Pflanzer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programmauszugsverzeichnis

1

Einleitung

Das in dieser Arbeit verwendete *Min-Warping-Verfahren* (Möller et al., 2010) dient der visuellen Navigation von Robotern anhand zweier Panoramabilder. Da in der Echtzeitanwendung nur wenig Zeit pro Bildvergleich zur Verfügung steht, ist das Ziel dieser Arbeit, die Berechnungen auf diesen Bildern zu beschleunigen. Inbesondere soll das Verfahren auf den Befehl `_mm_sad_epu8` zur Berechnung des Abstandsmaßes angepasst werden.

Im folgenden Kapitel wird zunächst eine Einführung in das *Min-Warping-Verfahren* gegeben und auf bereits verwendete Abstandsmaße hingewiesen. Anschließend wird der theoretische Hintergrund der Vektorerweiterungen betrachtet und im Speziellen auf die zur Implementierung verwendeten *Streaming SIMD Extensions (SSE)* eingegangen. Darauffolgend werden die Details der bisherigen Umsetzung des Min-Warping-Algorithmus erläutert sowie der neu einzubringende Befehl `_mm_sad_epu8` und die Konsequenzen aus der Verwendung dargestellt. Das Kapitel 3 dient der Beschreibung der umgesetzten Änderungen und der Erörterung der Strategien, die bei der Umsetzung verfolgt wurden. Die Kapitel 4 und 5 umfassen die Testumgebung inklusive der verwendeten Hard- und Software sowie die Präsentation der gewonnen Ergebnisse aus Zeit- und Gütemessungen. Aus den Ergebnissen gewonnene Erkenntnisse und Ansätze für weitere Untersuchungen werden in Kapitel 6 diskutiert. Das abschließende Fazit und ein Ausblick finden sich in Kapitel 7.

2

Grundlagen

In diesem Kapitel werden die verschiedenen Grundlagen vorgestellt, auf denen die weitere Arbeit beruht. Darunter fallen das Min-Warping-Verfahren allgemein (Möller et al., 2010), die für das Verfahren geeigneten Abstandsmaße (Möller et al., 2014) und generelle Informationen über das SIMD-Konzept (Hennessy und Patterson, 2011).

2.1 Min-Warping

Das *Min-Warping-Verfahren* gehört zur Gruppe der *2D-Warping-Verfahren*, die zur visuellen Navigation mobiler Roboter eingesetzt werden (Möller et al., 2010). Anhand zweier Panoramabilder (*Current View* und *Snapshot*) bestimmen diese Verfahren die Bewegungsrichtung (*Homevektor*), die vom aktuellen Standpunkt zum Ziel führt. Dazu wird eines der Bilder verzerrt und die beste Passung gegenüber dem zweiten Bild gesucht. Aus der Verzerrung können anschließend die Bewegungsparameter α und ψ gewonnen werden. Die Bewegungsrichtung relativ zur Roboterorientierung wird durch α bestimmt und die Verdrehung des Roboters zwischen den beiden Bildern selbst durch ψ.

Grundsätzlich gilt für diese Art von Verfahren die *Equal-Distance Assumption*, die besagt, dass sich alle Objekte in der gleichen Entfernung zum Roboter befinden. Das in dieser Arbeit verwendete *Min-Warping-Verfahren* hebt diese Annahme größtenteils auf, sodass nur noch für Objekte innerhalb einer Bildspalte die gleiche Distanz zum Roboter angenommen werden muss. Als Folge kann die relative Distanz ρ des Roboters zu allen Landmarken nicht mehr für das gesamte Bild angegeben werden. Die systematische Suche über diskrete Werte des Parameters ρ entfällt daher und es wird stattdessen pro

Bildspalte die wahrscheinlichste Distanz ermittelt. Würde die *Equal-Distance Assumption* auch innerhalb der Spalten aufgehoben, wäre die derzeitige Vorausberechnung der spaltenweisen Bildabstände vor der eigentlichen Suche nicht mehr möglich.

Das Verfahren ist in zwei Phasen unterteilt: In der ersten Phase werden die Bildabstände für verschiedene Vergrößerungsstufen berechnet, in die sogenannten *Scale Planes* eingetragen und zum *Scale Plane Stack (SPS)* zusammengestellt. In der zweiten Phase werden per Minimumssuche über allen *Scale Planes* die besten Bewegungsparameter bestimmt.

Das Vergrößern der Bilder in der ersten Phase simuliert den Effekt des unterschiedlichen räumlichen Abstandes zu Landmarken nach einer Bewegung des Roboters. Die Panoramabilder werden dabei vertikal um den Horizont herum vergrößert. Ein Verkleinern der Bilder findet nicht statt, da dadurch die Höhe der Bilder verringert würde und der Vergleich zwischen zwei Bildern somit ohne Normalisierung nicht mehr möglich wäre (Möller et al., 2010). Anstelle dessen wird eine Vergrößerung des anderen Bildes um den inversen Skalierungsfaktor verwendet. Für jede Vergrößerungsstufe werden spaltenweise die Bildabstände zwischen *Current View* und *Snapshot* berechnet und in die jeweilige *Scale Plane* eingetragen. Im *Scale Plane Stack* sind demnach jeder Vergrößerung die entsprechenden Bildabstände zugeordnet.

Die systematische Suche nach den besten Bewegungsparametern in der zweiten Phase wird gemäß des sogenannten *Scale Templates* durchgeführt. Dieses schränkt die Suche auf die theoretisch möglichen Parameterkombinationen ein und bestimmt, welche *Scale Planes* konsultiert werden müssen, um für eine bestimmte Parameterkombination den Bildabstand für das gesamte Bild zu ermitteln. Gesucht wird dabei über diskrete Werte der Parameter α und ψ sowie indirekt über die relative Distanz zwischen Roboter und Landmarken. Letztere ist durch den Skalierungsfaktor der beiden Bildspalten mit dem kleinsten Abstand zueinander gegeben. Die Abstände werden pro Parameterkombination über alle Bildspalten aufsummiert, sodass schlussendlich die Kombination mit dem geringsten Gesamtabstand als durchzuführende Bewegung verwendet wird. Durch Erweiterungen wie zum Beispiel eine Kompass-Schätzung kann die zweite Phase weiter beschleunigt werden. Da das Thema dieser Arbeit sich jedoch ausschließlich auf die erste Phase bezieht, wird an dieser Stelle nicht weiter darauf eingegangen.

2.2 Abstandsmaße

Neben der in dieser Arbeit verwendeten *Normalised Sum of Absolute Differences* zur Bestimmung des Abstandes zwischen zwei Bildern sind weitere Abstandsmaße entwickelt und evaluiert worden (Möller et al., 2014). Das Hauptaugenmerk liegt dabei auf einer möglichst guten Toleranz gegenüber unterschiedlichen Beleuchtungsverhältnissen.

Die zunächst verwendete *Sum of Squared Differences (SSD)* (Euklidische Distanz) ist nur sehr eingeschränkt invariant gegenüber Beleuchtungsänderungen. Daher wurde eine gegen *multiplikative* Intensitätsschwankungen invariante Form auf Basis der SSD entwickelt und daraus später eine Art der *Normalised Cross Correlation (NCC)* abgeleitet.

Einen anderen Ansatz verfolgt die *Sequential Correlation (SC)*, die die räumliche Beziehung zwischen Bildpixeln berücksichtigt. Dies wird durch die Anwendung eines Kantenfilters erreicht. Das aus den Richtungstermen der einzelnen Kanten gewonnene Abstandsmaß ist invariant gegenüber *additiven* Intensitätsänderungen. Davon abgeleitet wird die *Approximated Sequential Correlation (ASC)*, die eine effizientere Umsetzung mittels Ganzzahlenarithmetik ermöglicht.

Die *Normalised Sum of Absolute Differences (NSAD)* setzt sich aus der *Sum of Absolute Differences (SAD)* und einem Normalisierungsterm zusammen. Die SAD berechnet blockweise (beim Min-Warping-Verfahren über die Bildspalten) die Differenz zwischen den Intensitätswerten und summiert über die Absolutwerte der Differenzen, um so eine Metrik für ähnliche Blöcke innerhalb der Bilder zu erhalten (Wikipedia, 2014). Diese Summe wird mittels der beiden Summen über die absoluten Intensitäten beider Bilder normalisiert, um auch hier eine Form der Beleuchtungsinvarianz zu erhalten. Dadurch ergibt sich folgende Definition der NSAD:

$$J_{\mathrm{NSAD}}(\vec{a}, \vec{b}) = \frac{\sum\limits_{i} |a_i - b_i|}{\sum\limits_{i} |a_i| + \sum\limits_{i} |b_i|} \in [0, 1]$$

Ein Wert von 0 entspricht dabei einer perfekten Übereinstimmung der beiden Bildblöcke, während ein Wert von 1 den gegenteiligen Fall ausdrückt. Im Min-Warping-Verfahren wird die NSAD ebenfalls auf kantengefilterte Bilder angewendet. Bei der auf Ganzzahlen basierenden Implementierung des Min-Warping-Algorithmus wird im Nenner eine 1 addiert, um Divisionen durch Null zu vermeiden.

2.3 SIMD Konzept

Eine einfache Klassifikation der verschiedenen Parallelrechnerarchitekturen basiert auf den vier von Flynn (1966) vorgestellten Grundtypen der Datenverarbeitung: *SISD*, *SIMD*, *MIMD* und *MISD*.

SISD	**S**ingle **I**nstruction **S**ingle **D**ata
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
MIMD	**M**ultiple **I**nstructions **M**ultiple **D**ata
MISD	**M**ultiple **I**nstructions **S**ingle **D**ata

Tabelle 2.1: Klassifikation von Parallelrechnerarchitekturen

Der klassische *Von-Neumann-Rechner* verarbeitet mit jedem Befehl nur ein Datenelement und fällt somit in die *SISD*-Gruppe. Hierbei kann Parallelität nur auf Instruktionsebene, zum Beispiel durch *Pipelining-Techniken*, erreicht werden. Parallelität auf Datenebene ist in den Kategorien *SIMD* und *MIMD* möglich. *SIMD*-Architekturen zeichnen sich durch eine zentrale Befehlsverwaltung aus, die die separaten Prozessoren mit jeweils eigenem Datenspeicher gemeinsam ansteuert. Im Gegensatz dazu existiert bei *MIMD*-Architekturen auch ein paralleler Befehlsfluss, der eine zusätzliche Synchronisation erforderlich macht und die Programmierung solcher Rechner deutlich erschwert. Die vierte Gruppe ist derzeit faktisch leer, da noch keine sinnvolle Anwendung für mehrere Befehle pro Datenelement gezeigt werden konnte, und wurde nur der Vollständigkeit halber mit eingeführt.

Der Begriff *SIMD* wird allerdings nicht mehr nur für Vektorrechner mit mehreren unabhängigen Prozessoren verwendet, sondern hat auch bei Ein-Prozessor-Maschinen Einzug gehalten. Diese stellen bestimmte Vektorregister zur Verfügung, die mehrere Elemente eines Grunddatentyps enthalten. Spezielle Vektorbefehle können dadurch parallel auf alle Datenelemente des Register angewendet werden (Abb. 2.1). Verglichen mit echten Vektorrechnern sind die Register momentan noch deutlich kleiner und auf eine fest definierte Anzahl von Elementen beschränkt. Weitere Details zu den Parallelrechnerarchitekturen stellen Hennessy und Patterson (2011) in Kapitel 4 ihres Buches vor.

2.3.1 Streaming SIMD Extensions

Als Vorgänger der *Streaming SIMD Extensions (SSE)* wurde 1996 der *MMX*-Befehlssatz von Intel eingeführt, um die Verarbeitung von Multimediadaten zu beschleunigen. Da beispielsweise Bild- oder Audiomaterial meist mit 8

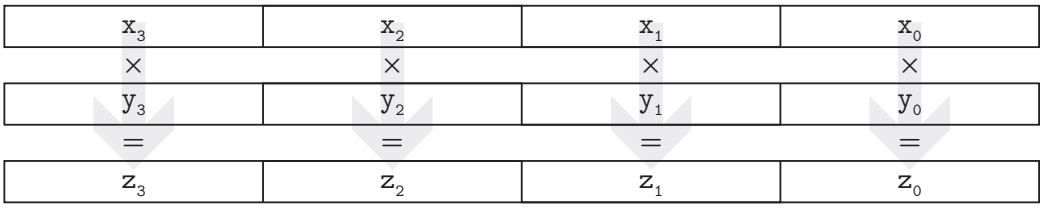


Abbildung 2.1: Parallele Verarbeitung von Daten mittels Vektorregistern

oder 16 Bit codiert wird, sind die gewöhnlichen 32-Bit-Befehle zur Datenverarbeitung nicht optimal. Hingegen bietet sich gerade bei Bildmaterial eine parallele Verarbeitung an, da dabei oftmals keine Abhängigkeiten zwischen einzelnen Operationen bestehen. Die zu verarbeitenden Daten müssen zur Verwendung der *MMX*-Befehle in den unteren 64 Bit der vorhandenen 80-Bit-Gleitkommaregister bereitgestellt werden. Dadurch können bis zu acht ganzzahlige Elemente gleichzeitig verarbeitet werden.

Mit der ersten Generation der *SSE*-Befehle wurden 1999 auch eigene 128-Bit-Register eingeführt. Zunächst waren mit diesen jedoch nur Operationen auf vier Gleitkommazahlen einfacher Genauigkeit möglich. Doppelte Genauigkeit sowie Befehle auf Ganzzahlen wurden unter anderem 2001 mit der Bezeichnung *SSE2* definiert. Bis 2007 folgten die Erweiterungen *SSE3*, *SSSE3*[1] und *SSE4*[2], mit denen neue Befehle eingeführt wurden.

Seit 2010 arbeitet Intel an den *Advanced Vector Extensions (AVX)*, die bisher schon eine Verdopplung der Registergröße auf 256 Bit gebracht haben und in Zukunft auch Register mit 512 Bit oder 1024 Bit nutzen können sollen. Weitere Informationen zur Programmierung von Vektorrechnern sind ebenfalls bei Hennessy und Patterson (2011) in Kapitel 4 zu finden.

Zusammen mit den neuen *SIMD*-Assemblerbefehlen hat Intel *Intrinsics*[3] definiert, die die direkte Nutzung der Vektorarithmetik im C/C++ Code ermöglichen. Dadurch sinkt vor allem die Gefahr, die automatischen Optimierungen des Compilers durch *Inline-Assembler* Abschnitte im Programmcode zu blockieren. Außerdem bieten *Intrinsics* die Möglichkeit, begrenzt hardwareunabhängigen Code zu schreiben, da sie je nach Verfügbarkeit in verschiedene Maschinenbefehle übersetzt werden können. Die Benennung der SSE-Intrinsics erfolgt nach dem Schema `_mm_<intrin_op>_<suffix>`, wobei

[1] *Supplemental Streaming SIMD Extensions 3*

[2] Der *SSE4* Befehlssatz ist in die Gruppen *SSE4.1* (47 Instruktionen) und *SSE4.2* (7 Instruktionen) unterteilt (Intel, 2014b).

[3] *Intrinsics* verhalten sich generell wie *Inline-Funktionen*. Jedoch hat der Compiler spezielles Wissen über die Funktionsweise und kann sie somit kontextabhängig umsetzen und effizienter optimieren.

s p	s d	
s	i	128
ep	i u	8 16 32 64
s – scalar *p – packed* *ep – extended packed*	*s – single-precision floating point* *d – double-precision floating point* *i – signed integer* *u – unsigned integer*	*Bit pro Element*

Table 2.2: Zusammensetzung des Suffix der Intel SSE-Intrinsics

`<intrin_op>` den eigentlichen Befehl charakterisiert und durch `<suffix>` bestimmt wird, wie der Inhalt des Registers interpretiert werden soll (Intel, 2007). Die Zusammensetzung möglicher Suffixe ist Tabelle 2.2 gegeben.

2.3.2 GCC Vector Extensions

Mit der Version 3.1[4] wurde 2002 auch für die *GNU Compiler Collection (GCC)*[5] die Unterstützung der *MMX*, *SSE* und *SSE2*[6] Befehle angekündigt. Definiert als Erweiterung der Programmiersprache C – zunächst nicht C++ – sollten diese sogenannten *Vector Extensions* kompatibel zu den von Intel vorgestellten *Intrinsics* sein, um geschriebenen Code möglichst portabel zu halten. Seit der Einführung wurden die *Vector Extensions* laufend verbessert und haben an Benutzerfreundlichkeit und Umfang zugenommen. In der aktuellsten Version der *GCC* umfassen sie weit mehr als nur die von Intel eingeführten *Intrinsics*. Sie ermöglichen zunehmend die von C/C++ bekannte Notation arithmetischer Operationen auch für die speziellen Vektordatentypen oder einen indexbasierten Zugriff auf einzelne Vektorkomponenten. Eine kurze Übersicht der wichtigsten, jeweils neu hinzugefügten, Eigenschaften bietet Tabelle B.1.

[4] `https://gcc.gnu.org/gcc-3.1/changes.html`; letzter Zugriff: 8. August 2014.
[5] Bis dahin *GNU C Compiler* genannt.
[6] *Intrinsics* für *SSE2* wurden erst in Version 3.3 implementiert.

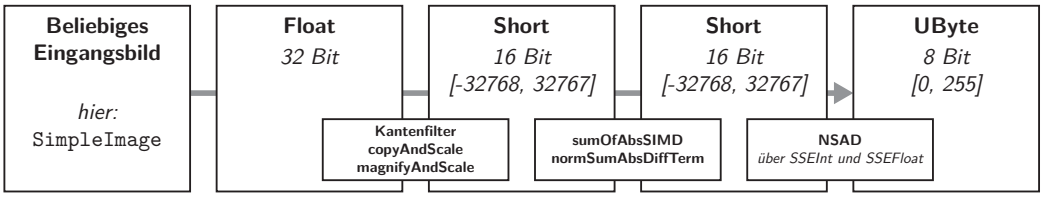


Abbildung 2.2: Wertebereiche der Bilder im Verlauf der ersten Phase des ursprünglichen Min-Warping-Verfahrens

2.4 Umsetzung der *Normalised Sum of Absolute Differences*

Nach der Betrachtung verschiedener Abstandsmaße in Abschnitt 2.2 beschreibt dieser Abschnitt die konkrete Implementierung der *Normalised Sum of Absolute Differences*, wie sie in der ersten Phase des Min-Warping-Verfahrens verwendet wird. Dabei wird zunächst die bestehende Version beschrieben und analysiert, um ihr anschließend das neue Konzept gegenüberstellen zu können.

2.4.1 Bisherige Implementierung

Der Min-Warping-Algorithmus kann prinzipiell auf Eingangsbildern beliebiger Pixeldatentypen arbeiten. Um jedoch die Berechnungen unabhängig vom ursprünglichen Datentyp durchführen zu können, findet in einem ersten Schritt eine Konvertierung der Intensitäten in eine einheitliche 32-Bit-Gleitkommarepräsentation[7] statt (Abb. 2.2). Der zweite, wichtigste Schritt der Vorverarbeitung besteht aus der Anwendung eines vertikalen Kantenfilters, aus dem die Beleuchtungsinvarianz des Verfahrens resultiert (Möller et al., 2014). Im Anschluss ist eine weitere Umwandlung der Gleitkommawerte notwendig, da die Berechnung des Zählers und Nenners der *Normalised Sum of Absolute Differences* aus Performanzgründen für vorzeichenbehaftete 16-Bit-Ganzzahlen ($\{x \in \mathbb{Z} \mid -32\,768 \leq x \leq 32\,767\}$) getrennt implementiert ist. Damit der neue Wertebereich nicht unter- oder überschritten, aber trotzdem ideal ausgenutzt wird, erfolgt eine Skalierung der Intensitätswerte[8] (Abschn. 3.1). Dabei ist zu beachten, dass der Kantenfilter richtungssensitiv

[7] Es wird angenommen, dass der Wertebereich von $1.5 \cdot 10^{-45}$ bis $3.4 \cdot 10^{38}$ ausreichend ist, um beliebige Intensitätswerte aufnehmen zu können. In Fällen, in denen diese Annahme nicht zutrifft, muss bereits an dieser Stelle passend skaliert werden.

[8] Streng genommen handelt es sich nach der Anwendung des Kantenfilters nicht mehr um Intensitätswerte. Der Einfachheit halber und da dies kein zentraler Bestandteil dieser Arbeit ist, wird im Weiteren dennoch diese Bezeichnung beibehalten.

```
double idealPixelScale =
    static_cast<double>(POT2(15) - 1) /
    (2.0 * static_cast<double>(h) *
    static_cast<double>(maxAbsSumPerPixel));
```

Programmauszug 2.1: Berechnung des idealen Skalierungsfaktors nach der Anwendung des Kantenfilters. Die Variable `idealPixelScale` speichert den Skalierungsfaktor, `h` ist die Höhe der kantengefilterten Eingangsbilder und `maxAbsSumPerPixel` ist die maximale Spaltensumme aus allen kantengefilterten Eingangsbildern dividiert durch die Höhe des jeweiligen Bildes. Der Faktor *2* resultiert daraus, dass im Nenner der NSAD zwei 16-Bit-Summen addiert werden, wobei auch dort der vorzeichenbehaftete 16-Bit-Wertebereich ($\{x \in \mathbb{Z} \mid -2^{15} \leq x \leq 2^{15} - 1\}$) nicht überschritten werden darf. Die Rechnung resultiert aus der Zielgleichung: $2 \cdot h \cdot \mathrm{maxAbsSumPerPixel} \leq 2^{15} - 1$.

arbeitet und sich somit der darzustellende Wertebereich im Vergleich zum Eingangsbild potentiell verdoppelt.

In der vorliegenden Programmversion sind die Eingangsbilder beispielsweise vom Typ `SimpleImage` mit vorzeichenlosen 8-Bit-Pixeln, wodurch die Intensitäten auf den Bereich $\{x \in \mathbb{N}_0 \mid 0 \leq x \leq 255\}$ festgelegt sind. Nach dem Kantenfilter auf den Gleitkommazahlen ergibt sich damit ein theoretischer Bereich von $\{x \in \mathbb{Z} \mid -255 \leq x \leq 255\}$. Hier dient die Skalierung folglich der optimalen Ausnutzung des 16-Bit-Wertebereichs, und der Skalierungsfaktor ergibt sich gemäß Programmauszug 2.1 zu rund 25.5[9].

Zur anschließenden Berechnung des Zählerterms der NSAD

$$\sum_i |a_i - b_i|$$

muss die *Sum of Absolute Differences* zwischen zwei kantengefilterten Bildern bestimmt werden. Die notwendigen Rechenoperationen sind durch die drei Intrinsics `_mm_abs_epi16`, `_mm_subs_epi16` sowie `_mm_adds_epi16` umgesetzt. Damit hat die Berechnung des Zählers eine theoretische *Latenz* von 3 Zyklen und einen *Throughput*[10] von 1.5 Zyklen (Tab. 2.3).

Bei `_mm_adds_epi16` und `_mm_subs_epi16` handelt es sich um *saturierte Arithmetik*, bei der Werte unter- und oberhalb des Wertebereichs – hier

[9] Bestimmt auf Basis der *living3_day_day_Hh288sh_0.10* Datenbank.

[10] Intel listet unter der Bezeichnung *Throughput* nicht den eigentlichen *Durchsatz*, d. h. wie viele Instruktionen pro Zeiteinheit verarbeitet werden können, sondern die Anzahl an Zeiteinheiten, während deren eine Instruktion die Ausführungseinheit blockiert. Siehe auch `https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput` (letzter Zugriff: 8. August 2014) oder (Fog, 2014a).

Befehl	**Latenz**		**Throughput**	
	Zähler	Nenner	Zähler	Nenner
`_mm_subs_epi16`	1.0	—	0.5	—
`_mm_abs_epi16`	1.0	1.0 (2.0)	0.5	0.5 (1.0)
`_mm_adds_epi16`	1.0	1.0 (2.0)	0.5	0.5 (1.0)
Gesamt	3.0	2.0 (4.0)	1.5	1.0 (2.0)

Tabelle 2.3: Theoretische Werte für Latenz und Throughput bei der Berechnung der NSAD. Diese sind entnommen aus dem Intel Intrinsics Guide (Intel, 2014d) für die Sandy Bridge Architektur und dienen daher nur als Richtwerte. Im tatsächlichen Programmablauf können die Werte durch das Zusammenspiel mit anderen Befehlen und durch die Umsetzung des Compilers variieren. In der Spalte des Nenners stehen die Zahlen in Klammern für den Aufwand der Berechnung beider Teilsummen und sind daher doppelt so groß.

vorzeichenbehaftet 16-Bit – abgeschnitten werden. Dadurch wird verhindert, dass es zu Unter- bzw. Überläufen bei der Berechnung kommt (Intel, 2014b, Abschn. 9.3). Die Funktionen entsprechen den mathematischen Definitionen 2.1 und 2.2.

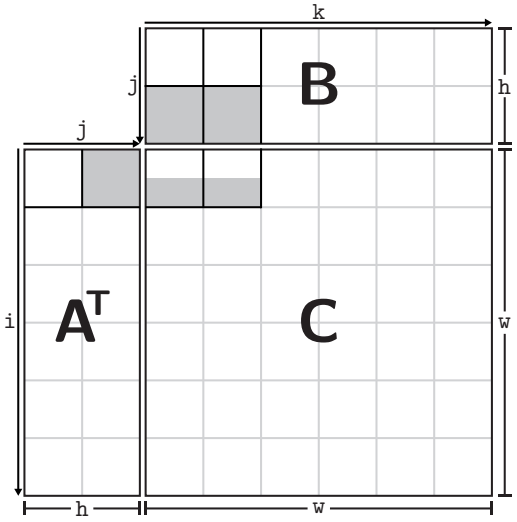
$$\operatorname{adds}(x,y) = \begin{cases} 32\,767 & \text{falls } x+y > 32\,767 \\ -32\,768 & \text{falls } x+y < -32\,768 \\ x+y & \text{sonst} \end{cases} \tag{2.1}$$

$$\operatorname{subs}(x,y) = \begin{cases} 32\,767 & \text{falls } x-y > 32\,767 \\ -32\,768 & \text{falls } x-y < -32\,768 \\ x-y & \text{sonst} \end{cases} \tag{2.2}$$

Die Saturierung dient allerdings nur als Absicherung, denn die Skalierung der Werte erfolgt, wie in Abschnitt 3.1 beschrieben, derart, dass die Wertebereiche ohnehin eingehalten werden.

Die Ergebnismatrix der einzelnen Summen über die Absolutdifferenzen wird dabei nicht elementweise, sondern wie in Abbildung 2.3 dargestellt, *partiell zeilenweise* berechnet. Dazu wird in einem der beiden Bilder das erste Element[11] der ersten Spalte ausgewählt, von dem zunächst alle Elemente der entsprechenden Zeile des anderen Bildes subtrahiert werden. Die Beträge dieser Differenzen werden zu den jeweiligen Elementen der Ergebnismatrix

[11] In der Umsetzung mittels Intrinsics entspricht ein Element einem Zeilenvektor mit acht Komponenten.



(a) Visualisierung

```
C[,] = 0

for i = 0 to (w - 1) step 1
   for j = 0 to (h - 1) step 1
      for k = 0 to (w - 1) step 1
         C[i,k] += ABS(A[i,j] -
                       B[j,k])
      endfor
   endfor
endfor
```

(b) Schleifenreihenfolge

Abbildung 2.3: Berechnung der Ergebnismatrix der *Sum of Absolute Differences*. Das Bild A^T in (a) ist lediglich zur besseren Visualisierung transponiert dargestellt. Beide Bilder liegen jedoch in gleicher Weise ($w \times h$, *row-major order*) im Speicher. Jeder der hier gezeigten Blöcke entspricht einem Zeilenvektor aus acht Elementen, der in ein `xmm`-Register geladen und parallel verarbeitet werden kann. Die Schleifen in (b) iterieren über ganze Blöcke und nicht über einzelne Elemente.

hinzuaddiert. Anschließend wird die Berechnung mit der nächsten Zeile[12] des ersten Bildes fortgesetzt. Dieser Vorgang wird für alle Zeilen wiederholt, sodass schließlich die *Sum of Absolute Differences* für die erste Zeile der Ergebnismatrix vollständig berechnet ist. In gleicher Weise wird mit den restlichen Spalten des ersten Bildes und entsprechend den Zeilen der Ergebnismatrix verfahren.

Diese Reihenfolge der Berechnung hat sich aufgrund der zeilenweisen Organisation der Bilder im Speicher (engl. *row-major order*) als die effizienteste herausgestellt. Dass im zweiten Bild und in der Ergebnismatrix lange Zeit immer nur eine Zeile gelesen beziehungsweise geschrieben wird und die Elemente einer Zeile im Speicher hintereinander stehen, ermöglicht es der CPU nur diese Bereiche zu cachen, um so einen schnellen Zugriff zu gewährleisten.

Der Nenner der NSAD

$$\sum_i |a_i| + \sum_i |b_i|$$

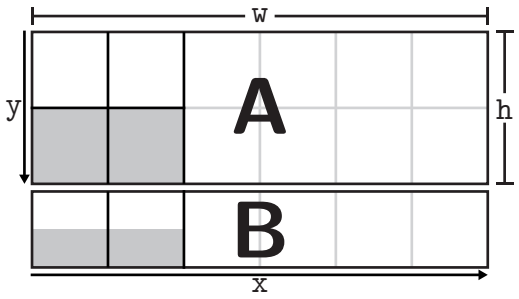
wird wiederum in zwei Summen aufgeteilt, sodass sich die Einzelterme jeweils ähnlich zum Zähler berechnen lassen: Es fällt lediglich die Subtraktion weg. Erst vor der Division werden die beiden Teilsummen schließlich addiert. Latenz und Throughput sind ebenso in Tabelle 2.3 aufgeschlüsselt und belaufen sich pro Teilsumme auf 2 Zyklen Latenz und 1 Zyklus Throughput beziehungsweise 4 Zyklen Latenz und 2 Zyklen Throughput für den kombinierten Nenner.

Die Schleifenreihenfolge zur Berechnung einer der Summen unterscheidet sich insofern von der des Zählers, als dass erst vertikal die Zeilen addiert werden und dann die Spalte[13] gewechselt wird (Abb. 2.4). Bei dieser direkt vollständigen Bildung der Spaltensumme können die Zwischenergebnisse in Registern gehalten werden, was gegenüber wiederholten Speicherzugriffen, die bei umgekehrter Reihenfolge notwendig wären, ebenfalls Geschwindigkeitsvorteile bringt.

Abschließend werden die vorher berechneten Zähler- und Nennerterme zum NSAD-Abstandsmaß kombiniert. Die Ergebnismatrix der *Normalised Sum of Absolute Differences* wird, wie bei der Berechnung des Zählers, zeilenweise ausgerechnet (Abb. 2.5). Für jede Zeile der NSAD-Matrix wird zuerst die entsprechende Komponente aus einem der beiden Spaltensummenvektoren verachtfacht und in die 16-Bit-Segmente eines `xmm`-Registers geladen. Der andere Vektor wird in Blöcken zu jeweils acht Komponenten durchlaufen.

[12] In Bezug auf Abbildung 2.3 müsste es korrekterweise *Spalte* heißen. Da das Bild in der Abbildung nur zur besseren Visualisierung transponiert wurde, bezieht sich der Begriff *Zeile* auf die Art, wie das Bild im Speicher liegt (*Breite* × *Höhe, row-major order*).

[13] Eine Spalte umfasst genau wie beim Zähler acht Komponenten, die parallel verarbeitet werden.



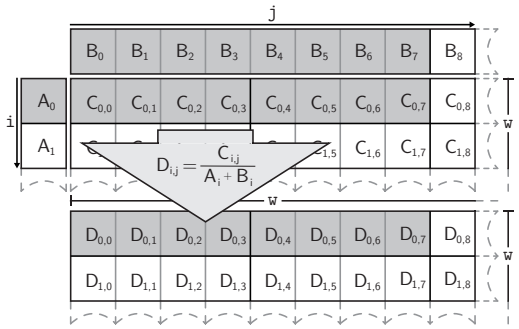
(a) Visualisierung

```
B[] = 0

for x = 0 to (w - 1) step 1
   for y = 0 to (h - 1) step 1
      B[x] += ABS(A[y,x])
   endfor
endfor
```

(b) Schleifenreihenfolge

Abbildung 2.4: Berechnung eines Spaltensummenvektors. Jeder der hier gezeigten Blöcke entspricht einem Vektor aus acht Komponenten, der in ein `xmm`-Register geladen und parallel verarbeitet werden kann. Die Schleifen in (b) iterieren über ganze Blöcke und nicht über einzelne Elemente.



(a) Visualisierung

```
for i = 0 to (w - 1) step 1
   T[0:7] = A[i]

   for j = 0 to (w - 1) step 8
      D[i,j:(j+8)] =
         C[i,j:(j+8)] /
         (T + B[i,j:(j+8)])
   endfor
endfor
```

(b) Schleifenreihenfolge

Abbildung 2.5: Berechnung der Ergebnismatrix der *Normalised Sum of Absolute Differences*. Der linke Spaltensummenvektor A^T ist lediglich zur besseren Visualisierung transponiert dargestellt. Beide Spaltensummenvektoren liegen jedoch in gleicher Weise ($1 \times w$, *row-major order*) im Speicher. Bei der Berechnung einer Zeile der NSAD-Matrix werden zum einen die acht Segmente eines `xmm`-Registers mit der Komponente der entsprechenden Zeile aus A gefüllt. Zum anderen werden wiederholt acht Komponenten aus B und aus C in zwei weitere Vektorregister geladen. Aufgrund der 32-Bit-Gleitkommadarstellung des Quotienten muss dieser jedoch in zwei Teilen aus Vektoren mit jeweils vier Komponenten gebildet werden.

```
__m128i C = _MM_4X4FLOAT_TO_16UBYTE(_MM_NDIST_PS(DfLL, LfLL, postScale4),
                                    _MM_NDIST_PS(DfLH, LfLH, postScale4),
                                    _MM_NDIST_PS(DfHL, LfHL, postScale4),
                                    _MM_NDIST_PS(DfHH, LfHH, postScale4));
_mm_store_si128((__m128i*)Cstore, C);

unsigned int *modulo2woffmj = modulo2woff - j;

for(int k = 0; k < 16; k++)
{
    *(destP + shufflepj[k] + *modulo2woffmj) = Cstore[k];
    modulo2woffmj--;
}
```

Programmauszug 2.2: Kombination aus Zähler- und Nennerterm des NSAD-Abstandsmaßes. Die Variablen `Dfxx` beinhalten die vier Zählerterme, die Variablen `Lfxx` die Nennerterme. Der Faktor `postScale4` dient der idealen Skalierung hinsichtlich der Wertebereiche in der zweiten Phase des Min-Warpings. Die Quotienten werden erst in das 16-elementige Array `Cstore` geschrieben, um dann die *Scale Planes geshuffled* erstellen zu können. Die verwendeten Makros sind in Anhang C enthalten.

Aus der Addition jedes Blocks mit der vervielfachten Komponente ergibt sich der Nenner für acht Elemente der jeweiligen Zeile der Ergebnismatrix. Ebenso werden acht Elemente der entsprechenden Zeile und der entsprechenden Spalten aus der SAD-Matrix geladen. Damit die Division durchgeführt werden kann, müssen die vorliegenden 16-Bit-Ganzzahlen zuvor in 32-Bit-Gleitkommazahlen umgewandelt werden. Dadurch reduziert sich die bisherige 8-fache vorübergehend auf eine 4-fache Parallelität und der Quotient muss in zwei Teilen mit jeweils vier Elementen gebildet werden. Die normalisierten Werte werden abschließend zur Vorbereitung auf die zweite Phase des Min-Warping-Verfahrens skaliert, in den vorzeichenlosen ganzzahligen 8-Bit-Wertebereich ($\{x \in \mathbb{N}_0 \mid 0 \leq x \leq 255\}$) konvertiert und als sogenannte *Scale Plane geshuffled* im Speicher abgelegt (Progr.-ausz. 2.2). Bei der Skalierung ist neben der Beschränkung durch den Wertebereich auch zu berücksichtigen, dass in der zweiten Phase die Summe über die Spalten der NSAD-Matrix den vorzeichenlosen ganzzahligen 16-Bit-Wertebereich ($\{x \in \mathbb{N}_0 \mid 0 \leq x \leq 65\,535\}$) nicht überschreiten darf. Für die auf das Intervall $[0, 1]$ normalisierten Werte der *Normalised Sum of Absolute Differences* ergibt sich damit bei einer $w \times w$ NSAD-Matrix die optimale Skalierung $p = \min\{255, \frac{65\,535}{w}\}$. Um beim Speichern der Werte immer effiziente *aligned* Zugriffe zu haben, werden pro Speicheroperation direkt die Werte für zwei 8er-Blöcke berechnet, gemeinsam in ein Register eingetragen und in den Speicher geschrieben. Durch das Um-

sortieren der Werte (*Shufflen*) beim Speichern ist in der zweiten Phase ein effizientes Auslesen möglich.

2.4.2 Spezielle Instruktion zur Berechnung summierter Absolutdifferenzen

Mit *SSE2* wurde der spezielle Befehl `_mm_sad_epu8` zur Berechnung der *Sum of Absolute Differences* implementiert. Daher bietet es sich an, auch bei der Berechnung der beim Min-Warping verwendeten *normalisierten* Summe über die Absolutdifferenzen, die bisherigen drei Befehle `_mm_subs_epi16`, `_mm_abs_epi16` und `_mm_adds_epi16` zu ersetzen.

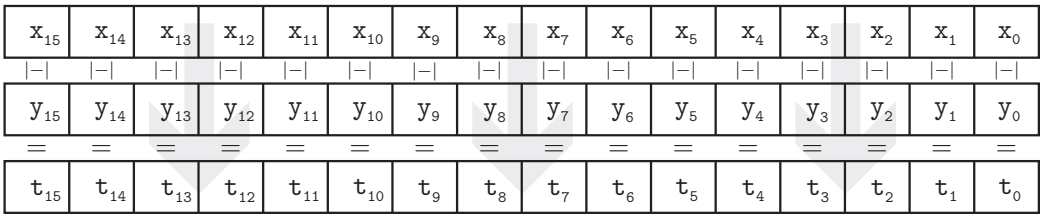
Im Gegensatz zu den meisten anderen Befehlen hat der Befehl `_mm_sad_epu8` neben der *vertikalen* Subtraktion durch die Summation über die Differenzen auch eine *horizontale* Komponente. Als *vertikal* klassifiziert Intel alle Operationen, die parallel zwischen den Elementen zweier Register durchgeführt werden, wohingegen *horizontale* Operationen die Elemente eines Registers betreffen. Das hat zur Folge, dass in vielen Fällen die *vertikale* Abarbeitung effizienter abläuft als die *horizontale* (Intel, 2014a, Abschn. 6.5.1.1). Um jedoch nicht immer die Repräsentation der Daten ändern zu müssen oder in Fällen wie diesem, wo beide Strategien zusammen verwendet werden müssen, existieren auch die *horizontalen* Instruktionen.

Der Befehl `_mm_sad_epu8` arbeitet auf gepackten vorzeichenlosen 8-Bit-Ganzzahlen, die in zwei 128-Bit-Registern übergeben werden. Im ersten Schritt bildet dieser die elementweisen Absolutwerte der Differenzen zwischen den Registern (Abb. 2.6a) und im zweiten Schritt wird über jeweils acht Elemente die Summe gebildet (Abb. 2.6b). Diese beiden ebenfalls vorzeichenlosen 16-Bit-Teilsummen werden in den unteren und oberen 64 Bit des Ergebnisregisters abgelegt (Intel, 2007).

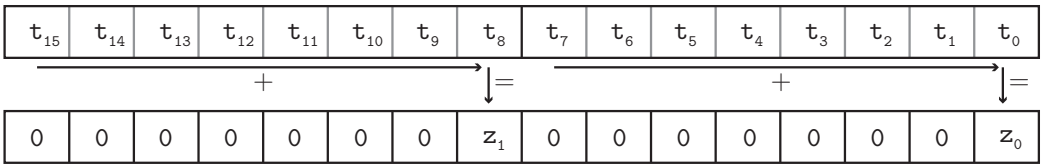
Im Vergleich zu den ursprünglichen drei Befehlen hat das Intrinsic `_mm_sad_epu8` mit 5 Zyklen eine geringfügig höhere Latenz. Der Throughput (1 Zyklus) ist jedoch einen halben Zyklus um geringer und zudem wird, bedingt durch den kleineren, vorzeichenlosen 8-Bit-Wertebereich ($\{x \in \mathbb{N}_0 \mid 0 \leq x \leq 255\}$), eine doppelt so hohe Parallelität (16-fach) erreicht.[14]

Bevor der Befehl für den Min-Warping-Algorithmus genutzt werden kann, sind noch zwei Bedingungen zu beachten: Der Befehl existiert ausschließlich für vorzeichenlose 8-Bit-Ganzzahlen und die Bilder müssen spaltenweise im Speicher abgelegt werden. Die erste Einschränkung macht es erforderlich, die

[14] Die Angaben zu Latenz und Throughput stammen ebenfalls aus dem Intel Intrinsics Guide (Intel, 2014d) und sind daher Richtwerte, die jedoch durch den Programmablauf beeinflusst werden können.



(a) Schritt 1: Absolute Differenzen



(b) Schritt 2: Bildung der Teilsummen

Abbildung 2.6: Darstellung der Funktionsweise des Intrinsics `_mm_sad_epu8`

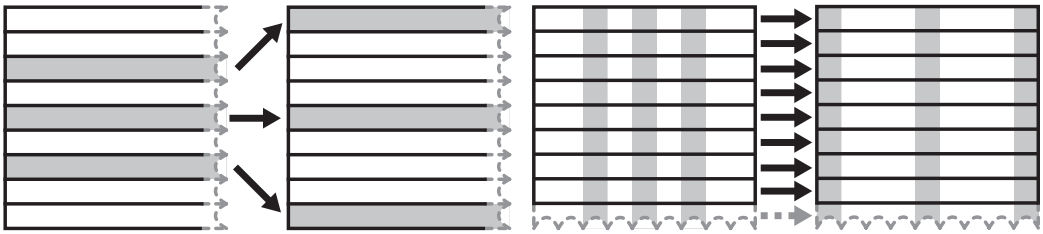
aus der Anwendung des Kantenfilters resultierenden, vorzeichenbehafteten Werte durch einen ausreichend großen Bias in den positiven Bereich zu verschieben. Aus der Überlegung, den Wertebereich weiterhin symmetrisch zu halten, ergibt sich der Bias $B = 128$, durch welchen der effektive Wertebereich nach dem Kantenfilter auf $\{x \in \mathbb{Z} \mid -128 \leq x \leq 127\}$ eingeschränkt ist.

Die Funktion `_mm_sad_epu8` mit solchen biascodierten Eingaben zu verwenden ist ohne Probleme möglich, da die additive Verschiebung (Gl. 2.3) durch die Differenzenbildung eliminiert wird (Gl. 2.4). Daher sind keine weiteren Nachverarbeitungsschritte notwendig.

$$\forall i \colon a_i^B = a_i + B,\ b_i^B = b_i + B \tag{2.3}$$

$$\begin{aligned} \sum_i |a_i^B - b_i^B| &= \sum_i |(a_i + B) - (b_i + B)| \\ &= \sum_i |(a_i - b_i) + (\underbrace{B - B}_{=0})| \\ &= \sum_i |a_i - b_i| \end{aligned} \tag{2.4}$$

Das zweite Problem betrifft das Layout der Daten im Speicher. Einerseits ist mit der bisherigen, zeilenweisen Speicherung der Bilder keine effiziente Verwendung des neuen Befehls möglich. Dies resultiert daraus, dass die Elemente einer Bildspalte erst in aufeinanderfolgende Speicherbereiche kopiert werden müssten, um sie an die Funktion `_mm_sad_epu8` übergeben zu können. Andererseits ist jedoch die vertikale Skalierung der Bilder insbesondere beim zeilenweisen Layout am besten realisierbar. Prinzipiell können dabei stets im



(a) row-major order **(b)** column-major order

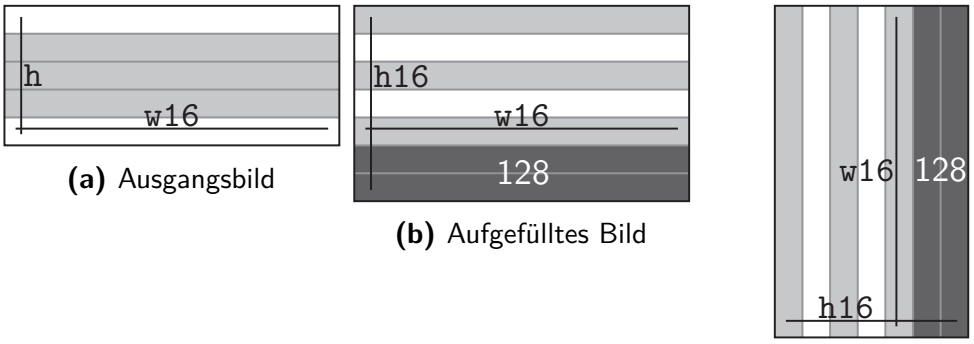
Abbildung 2.7: Skalierung der Bilder bei unterschiedlichen Speicherlayouts. In (a) können problemlos komplette Zeilen verschoben werden, wohingegen bei (b) nur einzelne Teile jeder Zeile übernommen werden könnten.

Speicher zusammenhängende Zeilen kopiert werden (Abb. 2.7a), wohingegen bei einer spaltenweisen Anordnung die einzelnen Elemente der Spalte separat übertragen werden müssten (Abb. 2.7b). Da bei einem sequentiellen Kopieren aller Elemente große Geschwindigkeitseinbußen zu erwarten sind, bietet es sich als Kompromiss an, die Bilder nach dem Skalieren zu transponieren, sodass die Daten für alle weiteren Schritte in *column-major order* vorliegen. Zum einen sollte dieser einmalige Aufwand des Transponierens, bei geschickter Implementierung (Abschn. 3.2.3), geringer ausfallen als der eines aufwändigen Skalierens. Zum anderen wird dieser voraussichtlich gegenüber den Geschwindigkeitsvorteilen durch die höhere Parallelität des neuen Befehls vernachlässigbar sein.

Schließlich erfordert es der Befehl `_mm_sad_epu8`, dass neben der Breite nun auch die Höhe der Bilder auf ganzzahlige Vielfache von 16 beschränkt wird. Andernfalls müsste der letzte Berechnungsschritt gesondert behandelt werden, falls keine 16 Werte für eine parallele Verarbeitung mittels des Befehls zur Verfügung stehen. Auch das Transponieren profitiert von der eingeschränkten Anzahl an möglichen Bildgrößen, da dadurch eine blockweise Umsetzung möglich ist, wie sie in Abschnitt 3.2.3 beschrieben wird.

Um die Einschränkung der Bildhöhe zu erreichen, wird die nächstgrößere durch 16 teilbare Höhe der Bilder berechnet und der Zielspeicherbereich für die skalierten Werte mit dieser neuen Höhe und der ohnehin eingeschränkten Breite angefordert. Damit ist gewährleistet, dass in den weiteren Schritten des Algorithmus sowohl Breite als auch Höhe der Bilder ein Vielfaches von 16 sind und das Resultat ein *column padded*[15] Bild ist. Dessen Elemente werden

[15] Nach dem Transponieren handelt es sich folglich um *row padded* Bilder. Da das Transponieren allerdings nur für eine effiziente Implementierung notwendig ist, jedoch keinen Einfluss auf den Algorithmus an sich hat, wird weiterhin die Bezeichnung des *column padded* Bildes verwendet.



(a) Ausgangsbild

(b) Aufgefülltes Bild

(c) Aufgefülltes, transponiertes Bild

Abbildung 2.8: Ablauf des Auffüllens und Transponierens der Eingangsbilder. Nach dem Padding ist sichergestellt, dass die Höhe des Bildes ein ganzzahliges Vielfaches von 16 ist und die zusätzlichen Elemente durch die Initialisierung mit 128 keinen Einfluss auf die späteren Rechnungen haben. Durch das Transponieren wird aus dem *Column Padding* ein *Row Padding*. Beides hat jedoch keinen Einfluss auf die Skalierung.

mit dem Wert 128 initialisiert, welcher sich nach Abzug des Bias zu 0 ergibt. Damit ist sichergestellt, dass alle zusätzlich hinzugefügten Elemente keinen Einfluss auf die spätere Summation haben. Die Vergrößerung wird weiterhin mit der originalen Bildhöhe, ohne Berücksichtigung des größeren Zielspeicherplatzes, durchgeführt, da die zusätzlichen Zeilen nur die Verwendung des Intrinsics ermöglichen sollen. Das Ergebnis des Algorithmus darf durch sie nicht beeinflusst werden. Im Schritt des Transponierens ist der zusätzliche Speicher erstmals für eine durchgängig blockweise Verarbeitung erforderlich. Die genannten drei Schritte sind in Abbildung 2.8 veranschaulicht.

16-Bit-Version

Der Ablauf des Min-Warping-Algorithmus ändert sich durch die Verwendung des Befehls `_mm_sad_epu8` nur geringfügig (Abb. 2.9). Die wichtigste Anpassung ist die zwischenzeitliche Reduktion des Wertebereichs auf 8 Bit, denn sowohl das Auffüllen der Bildhöhen als auch das Transponieren haben keinen Einfluss auf das eigentliche Verfahren. Die Spaltensummenvektoren und die SAD-Matrix haben wieder den gleichen 16-Bit-Wertebereich wie bei der ursprünglichen Version, sodass an der Division zur eigentlichen Berechnung des NSAD-Abstandsmaßes keine Änderungen vorzunehmen sind. Die weiteren Details bezüglich der Anpassungen sind nach Funktionen gegliedert in Kapitel 3 beschrieben.

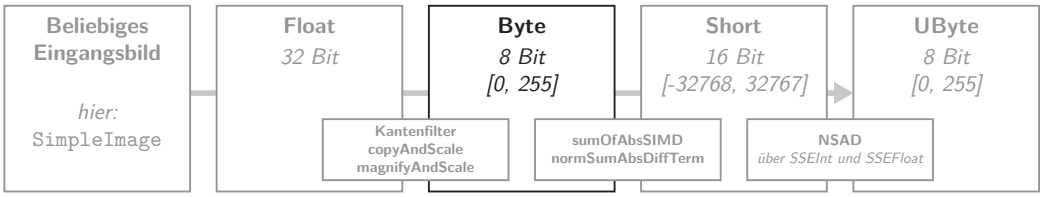


Abbildung 2.9: Wertebereiche der Bilder im Verlauf der ersten Phase des für `_mm_sad_epu8` adaptierten Min-Warping-Verfahrens

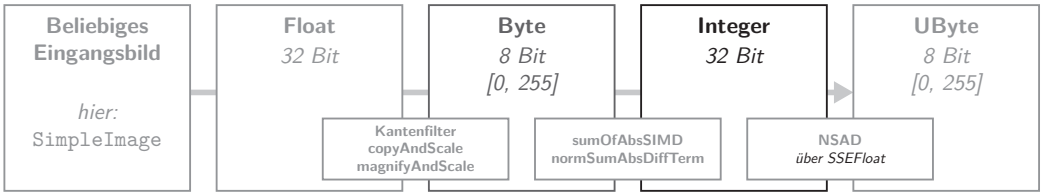


Abbildung 2.10: Wertebereiche der Bilder im Verlauf der ersten Phase des für `_mm_sad_epu8` und 32-Bit-Wertebereich adaptierten Min-Warping-Verfahrens

32-Bit-Version

Zusätzlich zu der angepassten 16-Bit-Version wird eine weitere Variante vorgestellt, die die beiden aus dem Befehl `_mm_sad_epu8` resultierenden Teilsummen als vorzeichenlose 32-Bit-Ganzzahlen interpretiert (Abb. 2.10). Dadurch vereinfacht sich einerseits die Konvertierung zu Gleitkommazahlen einfacher Genauigkeit vor der abschließenden Division in der *Normalised Sum of Absolute Differences*. In der 16-Bit-Version müssen die Werte von Typ `SSEShort` erst nach `SSEInt` konvertiert werden, bevor eine Umwandlung in Gleitkommazahlen möglich ist (Progr.-ausz. C.1). In der 32-Bit-Version fällt der erste Schritt weg, da die Werte direkt als 32-Bit-Ganzzahlen vorliegen. Andererseits verringert sich auch der Berechnungsaufwand des idealen Skalierungsfaktors. Dies resultiert aus der Annahme, dass der 32-Bit-Wertebereich ($\{x \in \mathbb{N}_0 \mid 0 \leq x \leq 4\,294\,967\,295\}$) bei gängigen Bildgrößen nicht überläuft. Die genauen Änderungen sind ebenfalls in Kapitel 3 erläutert.

3

Adaptierte Methoden

In diesem Kapitel werden alle am Programmcode vorgenommen Änderungen beschrieben und erläutert. Diese sind nach Funktionen gegliedert und soweit möglich nach ihrem Auftreten im Programmablauf geordnet. Im Laufe der Bearbeitung dieses Themas wurden teilweise verschiedene Versionen der einzelnen Funktionen entwickelt. Nachfolgend werden jedoch nur jene Varianten erwähnt, die entweder für den weiteren Verlauf wichtige Erkenntnisse gebracht haben oder alternativ verwendet werden können. Andernfalls findet sich lediglich ein Hinweis auf die entsprechenden Programmauszüge in Anhang C.

3.1 Berechnung der idealen Skalierungsfaktoren

Die Funktion zur Berechnung der idealen Skalierungsfaktoren der Intensitäts- und NSAD-Werte wird im eigentlichen Min-Warping-Algorithmus nicht aufgerufen. Sie dient nur der initialen Bestimmung dieser Faktoren.

Durch die Funktion wird sowohl die Skalierung der Intensitätswerte nach dem Kantenfilter (`pixelScale`) als auch die Skalierung der NSAD-Werte (`postScale`) bestimmt. Auf letztere wird allerdings nicht weiter eingegangen, da die Werte der *Normalised Sum of Absolute Differences* unverändert im Intervall $[0, 1]$ liegen und somit die Skalierung für die zweite Phase des Min-Warping-Verfahrens nicht angepasst werden musste.

16-Bit-Version

In dieser Variante werden die Ergebnisse des Befehls `_mm_sad_epu8` als 16-Bit-Werte interpretiert, sodass sich ab da die gleichen Wertebereiche wie in

```
double byteLimit = 127.0 / (static_cast<double>(maxValuePerInputPixel) -
                            static_cast<double>(minValuePerInputPixel));

double colSumLimit = static_cast<double>(POT2(15) - 1) /
                     (2.0 * static_cast<double>(h) *
                      static_cast<double>(maxAbsSumPerPixel));

idealPixelScale = min(byteLimit, colSumLimit);
```

Programmauszug 3.1: Berechnung der idealen Skalierung in der neuen 16-Bit-Version. Diese ergibt sich aus dem Minimum der bisherigen Skalierung (`colSumLimit`) und der Beschränkung auf den 8-Bit-Wertebereich (`byteLimit`) vor der Berechnung des NSAD-Maßes. Zusätzlich erforderlich für die Berechnung ist das Wissen über den Wertebereich der Eingangsbilder (`minValuePerInputPixel` und `maxValuePerInputPixel`).

der bisherigen Implementierung ergeben. Deswegen muss der bestehende Programmcode lediglich um eine Begrenzung bezüglich des vorzeichenlosen 8-Bit-Wertebereichs vor der Berechnung der *Sum of Absolute Differences* erweitert werden (Progr.-ausz. 3.1). Notwendig dafür ist die Kenntnis über die kleinste (minValue) und größte Intensität (maxValue) der in die Gleitkommadarstellung konvertierten Eingangsbilder. Durch die Skalierung $s = \frac{127}{\text{maxValue}-\text{minValue}}$ ist sichergestellt, dass die größte Intensität nach der Anwendung des Kantenfilters nicht die obere Grenze des geforderten Intervalls von $[-128, 127]$ überschreitet. Die untere Grenze muss nicht betrachtet werden, da sie betragsmäßig größer als die obere ist. Das Minimum aus der bisherigen und der neu hinzugefügten Skalierung wird als gewünschter Faktor mithilfe der Funktion `idealScalesNormSumAbsDiffColPadByte` ermittelt.

32-Bit-Version

Die Berechnung der Beschränkung aufgrund des kleineren 8-Bit-Wertebereichs kann aus der 16-Bit-Version übernommen werden, durch die Erweiterung der Zwischenergebnisse auf 32-Bit fällt jedoch deren Einfluss auf die Skalierung weg. Wie im Folgenden beschrieben wird, kann angenommen werden, dass bei derzeit gängigen Bildgrößen kein Überlaufen des 32-Bit-Wertebereichs zu erwarten ist.

Maximale Bildhöhen in der 16- und 32-Bit-Version Bei der Berechnung des NSAD-Maßes hängt sowohl der Zähler (Gl. 3.1) als auch der Nenner (Gl. 3.3) von der Höhe der Eingangsbilder ab, da beide Male über die Spalten summiert wird. Es ist aber eine getrennte Betrachtung beider Terme möglich,

sodass am Ende lediglich das Minimum aus den beiden bestimmten, maximalen Bildhöhen gewählt werden muss (Gl. 3.5). Des Weiteren kann vom Wertebereich der Eingangsbilder abstrahiert werden, da die Intensitätswerte vor der Berechnung der *Sum of Absolute Differences* ohnehin auf einen 8-Bit-Wertebereich ($\{\forall i\colon a_i, b_i \in \mathbb{Z} \mid -128 \leq a_i, b_i \leq 127\}$)[16] skaliert werden müssen.

$$\sum_{i=1}^{h} |a_i - b_i| \leq \sum_{i=1}^{h} \max_i \{|a_i - b_i|\} \tag{3.1}$$

$$= \sum_{i=1}^{h} 255 = h \cdot 255 \leq 2^{N-1} - 1$$

$$\Rightarrow h_{\max,N,\text{Num}} \leq \frac{2^{N-1} - 1}{255} \tag{3.2}$$

$$\sum_{i=1}^{h} |a_i| + \sum_{i=1}^{h} |b_i| \leq 2 \cdot \sum_{i=1}^{h} \max_i \{|a_i|\} \qquad \text{o. B. d. A.} \tag{3.3}$$

$$= 2 \cdot \sum_{i=1}^{h} 128 = 2 \cdot h \cdot 128 \leq 2^{N-1} - 1$$

$$\Rightarrow h_{\max,N,\text{Denom}} \leq \frac{2^{N-1} - 1}{256} \tag{3.4}$$

$$\overset{(3.2)}{\underset{(3.4)}{\Rightarrow}} h_{\max,N} = \lfloor \min\{h_{\max,N,\text{Num}}, h_{\max,N,\text{Denom}}\} \rfloor = \lfloor \frac{2^{N-1} - 1}{256} \rfloor \tag{3.5}$$

Für die 16-Bit-Version ($N = 16$) ergibt sich demnach nur eine maximale Bildhöhe von $h_{\max,16} = 127$ Pixeln. Bei der 32-Bit-Variante ($N = 32$) beträgt die maximale Bildhöhe allerdings bereits $h_{\max,32} = 8\,388\,607$ Pixel, sodass kein Überlauf des Wertebereichs zu erwarten ist.

[16] Im Programmcode selber handelt es sich um den *vorzeichenlosen* 8-Bit-Wertebereich. In Abschnitt 2.4.2 wurde gezeigt, dass die Biasverschiebung durch die Berechnung der *Sum of Absolute Differences* wieder eliminiert wird und die Zwischenergebnisse danach tatsächlich im *vorzeichenbehafteten* 8-Bit-Wertebereich liegen. Daher wird der Übersichtlichkeit halber an dieser Stelle darauf verzichtet, den Bias explizit mit in die Rechnung aufzunehmen.

3.2 Kopieren, Vergrößern und Skalieren der kantengefilterten Eingangsbilder

Nachdem die Eingangsbilder durch den Kantenfilter verarbeitet wurden, müssen sie zur Simulation des unterschiedlichen räumlichen Abstandes des Roboters von Landmarken vergrößert und die Intensitätswerte gemäß `idealPixelScale` (Abschn. 3.1) skaliert werden. Zusätzlich müssen aufgrund des Befehls `_mm_sad_epu8` in der neu entwickelten Version die Bilder transponiert werden.

3.2.1 Kopieren und Skalieren

Die Funktion `copyAndScaleSIMDFloat2ColPadByteImage` erwartet ein Bild vom Typ `SSEFloat` als Eingabe, konvertiert es inklusive der erforderlichen Skalierung in ein Ausgangsbild vom Typ `SSEByte` und schreibt dieses anschließend transponiert in den Speicher (Progr.-ausz. 3.2).

Das Eingangsbild wird dabei wie in der bisherigen Implementierung zeilenweisen sequentiell durchlaufen. Statt zwei Zeilenvektoren mit jeweils vier Gleitkommazahlen einfacher Genauigkeit einzulesen und parallel zu konvertieren, werden in der neuen Umsetzung aufgrund des kleineren 8-Bit-Wertebereichs vier `xmm`-Register mit Werten aus dem Speicher geladen. Dadurch kann bei der Konvertierung weiterhin die volle Registerbreite ausgenutzt werden. Die Variable `scale4` enthält demnach viermal den idealen Skalierungsfaktor, sodass eine Multiplikation mit den eingelesenen Gleitkommazahlen möglich ist. Anschließend werden innerhalb des Makros `_MM_4X4FLOAT_TO_16BYTE` (Progr.-ausz. C.4) die Produkte zunächst auf 16-Bit-Ganzzahlen reduziert, zu denen der Biasvektor `shortVec128` addiert wird. Außerdem erfolgt eine Umwandlung in den 8-Bit-Wertebereich, sodass in `byte16` die 16 konvertierten Werte zwischengespeichert werden können. Das Transponieren ist hier für jeden der eingelesenen Blöcke elementweise umgesetzt, da die Zielspeicherstellen im Speicher nicht zusammenhängen und somit der Block nicht als Ganzes zurückgeschrieben werden kann.

In der inneren Schleife über die Variable `x` (Progr.-ausz. 3.2, Zeile 17) wird blockweise eine komplette Zeile eingelesen und als Spalte wieder im Speicher abgelegt (Progr.-ausz. 3.2, Zeile 26). Dadurch wandert die Leseposition `src` immer um 16 Werte weiter, die Schreibposition `dst` jedoch immer um eine komplette Zeile des Ausgangsbildes (`w16Output`). Die äußere Schleife über die Variable `y` (Progr.-ausz. 3.2, Zeile 12) verschiebt die Leseposition in die nächste Zeile sowie die Schreibposition in die nächste Spalte (Progr.-ausz. 3.2, Zeilen 14 und 15).

```
const SSEFloat *src;
SSEByte *dst;
union
{
    __m128i r;
    SSEByte b[16];
} byte16;

__m128 scale4 = _mm_set1_ps(multiplyScale);
__m128i shortVec128 = _mm_set1_epi16(128);

for(int y = 0; y < hInput; ++y)
{
    src = img + w16Input * y;
    dst = scaledImage + y;

    for(int x = 0; x < w16Input; x += 16)
    {
        byte16.r = _MM_4X4FLOAT_TO_16BYTE(
                    _mm_mul_ps(*(__m128*) src, scale4),
                    _mm_mul_ps(*(__m128*) (src + 4), scale4),
                    _mm_mul_ps(*(__m128*) (src + 8), scale4),
                    _mm_mul_ps(*(__m128*) (src + 12), scale4),
                    shortVec128);

        for(int i = 0; i < 16; ++i)
        {
            dst[i * w16Output] = byte16.b[i];
        }

        src += 16; dst += 16 * w16Output;
    }
}
```

Programmauszug 3.2: Skalieren und Transponieren der kantengefilterten Eingangsbilder. Die Variable `scale4` repräsentiert ein Gleitkommaregister, das zur parallelen Verarbeitung viermal den idealen Skalierungsfaktor enthält. Die Biasverschiebung in den positiven Bereich wird durch die Variable `shortVec128` realisiert. Das Makro `_MM_4X4FLOAT_TO_16BYTE` setzt die Konvertierung um und wird im Programmauszug C.4 näher erläutert. Abschließend werden die zeilenweise eingelesenen Werte spaltenweise in den Speicher zurückgeschrieben, indem mittels einer Union auf die einzelnen 8-Bit-Elemente des `xmm`-Registers zugegriffen wird.

3.2.2 Vergrößern und Skalieren

Die Funktion `magnifyAndScaleSIMDFloat2ColPadByteImage` stellt zum Vergrößern der Bilder zwei verschiedene Methoden der Interpolation zur Verfügung: *Nearest Neighbour Interpolation* und *Linear Interpolation*.

Die Änderungen an der *Nearest Neighbour Interpolation* sind identisch zu denen aus Abschnitt 3.2.1, da die beiden Funktionen sich kaum voneinander unterscheiden: Durch das Vergrößern werden lediglich, wie auch in der bestehenden Implementierung, ausgewählte Zeilen kopiert. Die *Lineare Interpolation* benötigt zwar zusätzliche Rechenschritte, konnte jedoch auch analog erweitert werden, indem die Blockbreite von 8 auf 16 erhöht und in der inneren Schleife spaltenweises Speichern umgesetzt wurde. Dadurch bedingt werden in der neuen Version immer 16 Werte im Voraus interpoliert, die darauffolgend wie bereits beschrieben skaliert, konvertiert und transponiert werden. Beispielhaft sind die Änderungen an der linearen Interpolation in Programmauszug C.5 dargestellt.

3.2.3 Blockweise Matrixtransponierung

Alternativ zu der vorgestellten Methode, die Bilder elementweise zu transponieren, existiert die Möglichkeit des blockweisen Transponierens. Mathematisch gesehen besteht zwischen den beiden Methoden kein Unterschied: Das Ergebnis ist in beiden Fällen eine vollständig transponierte Matrix. Durch eine geringere Anzahl von Speicherzugriffen und einen höheren Grad an Parallelität ergibt sich für die blockweise Implementierung eine erheblich bessere Performanz als bei der sequentiellen Variante.

Generell lässt sich blockweises Transponieren rekursiv definieren (Abb. 3.1): Die Matrix muss in mehrere Untermatrizen unterteilt und basierend auf diesen als atomaren Elementen transponiert werden. Diese beiden Schritte sind für jede der Untermatrizen zu wiederholen, bis diese schließlich aus einem einzigen Element bestehen, das transponiert identisch zu sich selbst ist und deshalb direkt übernommen werden kann. Die Anzahl der Unterteilungen und die daraus resultierende Größe der Untermatrizen kann beliebig gewählt werden. Auch ist das Verfahren nicht auf quadratische Matrizen beschränkt.

Der rekursive Ansatz des Transponierens wird in dieser Arbeit durch die äquivalente iterative Strategie ersetzt. Statt einer wiederholten rekursiven Teilung der Matrix erfolgt eine einmalige Aufteilung in 16×16 Untermatrizen. Diese Blockgröße bietet sich an, da im Min-Warping-Algorithmus zumindest die Breite der Eingangsbilder auf ein ganzzahliges Vielfaches von 16 festgelegt ist und die vorhandenen Intrinsics 16 8-Bit-Werte parallel verarbeiten können. Anders als im allgemeinen Fall sind an dieser Stelle aus Symmetriegründen der

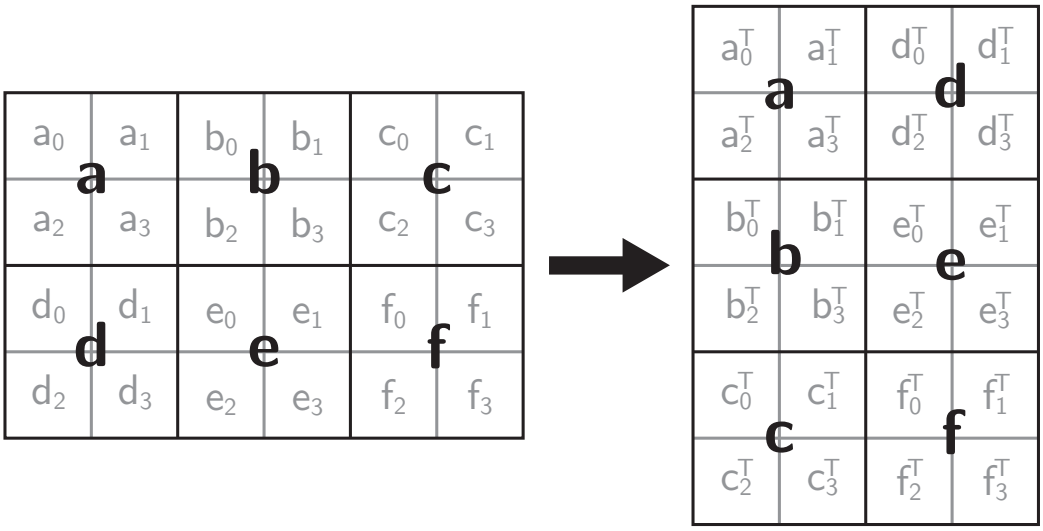


Abbildung 3.1: Blockweise rekursives Transponieren einer Matrix. Dazu wird diese in mehrere Untermatrizen unterteilt und basierend auf diesen als atomaren Elementen transponiert. Die Schritte sind für alle Untermatrizen zu wiederholen.

verwendeten Befehle quadratische Blöcke erforderlich. Zudem ändert sich so die Größe der Untermatrix beim Transponieren nicht, wodurch in jedem Schritt die für die eingesetzten Befehle ideale Anzahl von 16 Elementen pro Zeile genutzt werden kann. Jede Untermatrix wird unabhängig von den anderen transponiert und in den Speicher zurückgeschrieben. Die Zielspeicherstelle ergibt sich aus einer theoretisch durchgeführten Transponierung der Ausgangsmatrix mit den festgelegten 16×16 Blöcken als atomaren Elementen.

Die Methoden zum Kopieren, Skalieren und Transponieren müssen dahingehend geändert werden, dass immer 16 Zeilen im Voraus ausgewählt oder interpoliert und in ein Array aus Registern geladen werden. Dass die Höhe der Eingangsbilder nicht auf ganzzahlige Vielfache von 16 beschränkt ist, macht es erforderlich, entweder verbleibende Zeilen separat zu behandeln (Progr.-ausz. C.6) oder die Höhe der Bilder schon vor dem Transponieren auf ein ganzzahliges Vielfaches von 16 aufzufüllen (Progr.-ausz. C.7). Im Rahmen dieser Arbeit wurden beide Varianten implementiert, letztere zusätzlich mit umgekehrter Schleifenreihenfolge (Progr.-ausz. C.8), sodass die Performanz verglichen werden kann.

Um das Verständnis der nachfolgenden Beschreibungen der Methoden zur Transponierung eines einzelnen Blocks zu erleichtern und die Erläuterungen nicht durch Begriffserklärungen zu unterbrechen, werden wichtige Begriffe bereits an dieser Stelle im Voraus definiert.

Zeilen- und Spaltennummer Die Zeilen und Spalten einer $m \times n$ Matrix sind von 0 bis $m-1$ bzw. $n-1$ nummeriert.

Elementnummer Die Nummer eines Elements bezieht sich auf eine initiale zeilenweise Nummerierung aller Elemente der Ausgangsmatrix. Die Ecken einer $m \times n$ Matrix erhalten dadurch die folgen Nummern: 0 (oben links), $n-1$ (oben rechts), $(m-1) \cdot n$ (unten links) und $(m \cdot n)-1$ (unten rechts). Auch wenn ein Element an eine andere Position innerhalb der Matrix verschoben wird, behält es die einmal zugewiesene Nummer. Somit ist durch die Nummer eines Elements eindeutig die Position in der Ausgangsmatrix definiert.

Abstand Der Abstand zwischen zwei Elementen ist als der Absolutwert der Differenzen der beiden Elementnummern definiert. In einer $m \times n$ Ausgangsmatrix haben damit alle Elemente einer Zeile den Abstand 1 und alle Elemente einer Spalte den Abstand n.

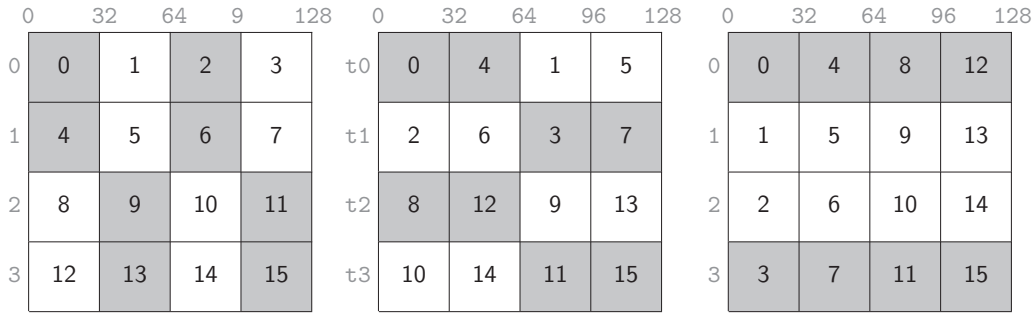
Zielabstand Der Zielabstand bezeichnet den Abstand, den die Elemente in einer vollständig transponierten Matrix zueinander haben. Bei einer $m \times n$ Ausgangsmatrix ergibt sich für die Elemente der transponierten Matrix ein Abstand von n innerhalb einer Zeile und ein Abstand von 1 innerhalb einer Spalte.

Gruppe Eine Gruppe umfasst alle benachbarten Elemente einer Zeile, die bereits den Zielabstand haben.

Folge Eine Folge umfasst alle benachbarten Elemente einer Zeile, die den gleichen Abstand zueinander haben. Dabei muss es sich nicht um den Zielabstand handeln.

Schrittweite Die Schrittweite gibt an, welche Zeilen der Matrix miteinander zu kombinieren sind. Eine Schrittweite von 1 bedeutet, dass benachbarte Zeilen paarweise kombiniert werden: zum Beispiel Zeile 0 mit Zeile 1 und Zeile 2 mit Zeile 3 etc. Bei einer Schrittweite von 2 würden stattdessen die Zeilen 0 und 2 und die Zeilen 1 und 3 kombiniert. In einer Kombinationsphase kommt es nicht vor, dass eine Zeile zweimal verwendet wird. Der Fall, Zeile 0 mit Zeile 1 und Zeile 1 mit Zeile 2 zu kombinieren, tritt also niemals auf.

Bereich/Bereichsgröße Ein Bereich umfasst einen Teil der Zeilen einer Matrix und gruppiert diese. Die Bereichsgröße gibt die Anzahl der Zeilen an, die der Bereich umfasst.



(a) Ausgangsmatrix **(b)** `_mm_unpackxx_ps` **(c)** `_mm_movexx_ps`

Abbildung 3.2: Blockweises Transponieren mittels des Makros `_MM_TRANSPOSE4_PS`. In (b) wurden durch die `_mm_unpackxx_ps`-Befehle Gruppen der Größe 2 gebildet. In (c) wurden diese Gruppe mithilfe der `_mm_movexx_ps`-Befehle zur vollständig transponierten Matrix umsortiert.

Mithilfe des von Intel definierten Makros `_MM_TRANSPOSE4_PS` (Progr.-ausz. C.9) lassen sich 4×4 Matrizen bestehend aus 16 32-Bit-Gleitkommazahlen transponieren (Abb. 3.2). Dazu wird die Matrix zeilenweise in vier Register geladen, in denen die Reihenfolge der Elemente vertauscht werden kann. Im ersten Schritt werden mittels der Befehle `_mm_unpacklo_ps` und `_mm_unpackhi_ps` alle Zeilen der Schrittweite 1 paarweise kombiniert. Das abwechselnde Ineinanderfügen der Elemente aus zwei Registern hat den gleichen Effekt wie eine Unterteilung der Matrix in 2×1 Spaltenvektoren und eine separate Transponierung dieser Vektoren. Dadurch bilden sich Gruppen der Größe 2, die im zweiten Schritt umsortiert werden müssen. Die effizienteste Möglichkeit bieten die Befehle `_mm_movelh_ps` und `_mm_movehl_ps`, mit denen zwei Low- bzw. zwei High-Teile unterschiedlicher Register in einem Register zusammengefügt werden können. Aus der paarweisen Kombination aller Zeilen der Schrittweite 2 resultieren Gruppen der Größe 4, sodass die Matrix nach diesem zweiten Schritt vollständig transponiert ist. Das Umsortieren der Gruppen der Größe 2 könnte auch über die Befehle `_mm_unpacklo_pd` und `_mm_unpackhi_pd` realisiert werden. Diese sind jedoch nicht so performant wie die `_mm_movexx_ps`-Befehle.

Aus diesem Makro lassen sich die zwei im Folgenden beschriebenen Algorithmen zum Transponieren von quadratischen $2^n \times 2^n$ Matrizen ableiten: Der *Bottom-Up*-Ansatz (Paul R, 2010) ist eine direkte Erweiterung des Makros auf die größeren Matrizen, wohingegen der *Top-Down*-Ansatz (nick_1234, 2012) eine andere Befehlsabfolge erfordert, dadurch aber ausschließlich mit `_mm_unpackxx_epixx`-Befehlen eines Wertebereichs auskommt. Bei beiden Ansätzen muss die Matrix vorab zeilenweise in Register geladen werden,

innerhalb derer die Elemente umsortiert werden. Abschließend können die Registerinhalte in den Speicher zurückgeschrieben werden.

Blockweises Transponieren mittels *Bottom-Up*-Ansatz

Das Ziel des *Bottom-Up*-Ansatzes ist es, die Gruppengröße in jedem Schritt zu verdoppeln, sodass nach dem letzten Schritt jede Zeile der Matrix aus einer einzigen Gruppe besteht und die Matrix transponiert ist. Die Verdopplung der Gruppengröße wird durch ein Umsortieren der bestehenden Gruppen innerhalb der Matrix erreicht. Um die Gruppen immer als atomare Einheiten verschieben zu können, sind `_mm_unpackxx_epixx`-Befehle für verschiedene Wertebereiche notwendig: Im ersten Schritt müssen einzelne Elemente verschoben werden, weshalb die `_mm_unpackxx_epixx`-Befehle des entsprechenden Wertebereichs der Elemente verwendet werden müssen. Zusammen mit der Gruppengröße verdoppelt sich anschließend in jedem Schritt auch der theoretische Wertebereich einer Gruppe, bis im letzten Schritt `_mm_unpackxx_epi64`-Befehle verwendet werden, die zwei Gruppen der halben Zeilenlänge zu einer kompletten verschmelzen.

Der genaue Ablauf lässt sich durch eine *Tail Recursion* beschreiben (Progr.-ausz. 3.3), bei der die Zeilen der Matrix zu verschiedenen Bereichen zusammengefasst und innerhalb dieser kombiniert werden. Die Bereiche dienen dabei ausschließlich der Auswahl der zu kombinierenden Zeilen, sodass der rekursive Aufruf als letzter Befehl der Funktion mit der gesamten Matrix ausgeführt wird. Der initiale Aufruf für eine $2^n \times 2^n$ Matrix erfolgt mit einer Bereichsgröße von 2^n und einer Schrittweite von 1. Die Bereichsgröße wird zu Beginn jedes Rekursionsschritts halbiert und die Schrittweite wird unmittelbar vor dem rekursiven Aufruf verdoppelt. Dazwischen werden alle Bereiche durchlaufen und für jeden werden die Zeilen im Abstand der Schrittweite paarweise kombiniert. Der theoretische Wertebereich und damit auch die zu verwendenden Versionen der Intrinsics lassen sich aus der Bereichsgröße berechnen. So können bei jeder Kombination die bestehenden Gruppen der jeweiligen Zeilen als atomare Elemente verschoben und zu Gruppen der doppelten Größe verschmolzen werden. Der Rekursionsabbruch erfolgt, sobald sich durch Halbierung eine Bereichsgröße kleiner 1 ergibt. Zu dem Zeitpunkt haben alle Elemente innerhalb der Zeilen den Zielabstand. Lediglich die Zeilen an sich sind noch in einer falschen Reihenfolge. Diese kann aber ohne Mehraufwand beim Zurückschreiben in den Speicher korrigiert werden. In Abbildung A.1 ist der Ablauf des Algorithmus zum besseren Verständnis schrittweise am Beispiel einer 8×8 Matrix gezeigt. Das in dieser Arbeit eingesetzte C-Präprozessor-Makro ist in Programmauszug C.10 dargestellt.

```
function transpose_bottom_up(ROWS, SECTIONS, STEPSIZE)
    SECTIONS = floor(SECTIONS / 2)

    if SECTIONS < 1
        return ROWS
    endif

    BIT = 64 / SECTIONS

    for i = 0 to (SECTIONS - 1) step 1
        SECTION = (2 * i * STEPSIZE)

        for j = 0 to (STEPSIZE - 1) step 1
            TMP = ROWS[SECTION + j]
            ROWS[SECTION + j] =
                _mm_unpacklo_epi@{BIT}(TMP, ROWS[SECTION + j + STEPSIZE])
            ROWS[SECTION + j + STEPSIZE] =
                _mm_unpackhi_epi@{BIT}(TMP, ROWS[SECTION + j + STEPSIZE])
        endfor
    endfor

    STEPSIZE = 2 * STEPSIZE

    transpose_bottom_up(ROWS, SECTIONS, STEPSIZE)
endfunction
```

Programmauszug 3.3: Ablauf des blockweisen Transponierens mittels *Bottom-Up*-Ansatz. Die Matrix wird zeilenweise in dem Array `ROWS` übergeben. Die Variablen `SECTIONS` und `STEPSIZE` geben die Anzahl der Bereiche, in die die Zeilen untergliedert werden, und die Schrittweite des jeweiligen Rekursionsschritts an. Die Anzahl der Bereiche wird direkt zu Beginn der Funktion halbiert und dient als Abbruchkriterium, sobald sie unter 1 gelangt. Über `BIT` wird festgelegt, für welchen Wertebereich die `_mm_unpackxx_epixx`-Befehle zu verwenden sind. In den beiden Schleifen werden in allen Bereichen die Zeilen der entsprechenden Schrittweite paarweise kombiniert. Es ist ein Register als temporärer Zwischenspeicher notwendig, da ansonsten benötigte Werte überschrieben würden. Vor dem rekursiven Aufruf wird die Schrittweite verdoppelt.

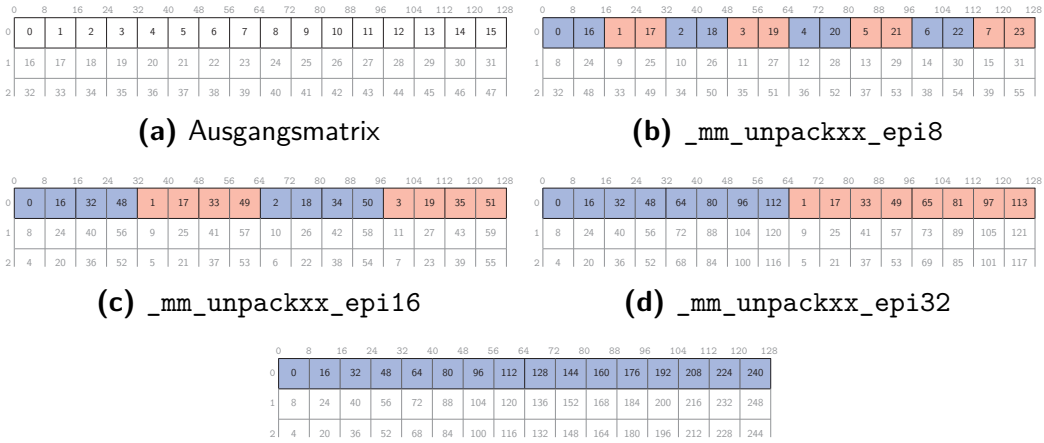


Abbildung 3.3: Blockweises Transponieren einer 16×16 Matrix mittels *Bottom-Up*-Ansatz am Beispiel der ersten Zeile. Von links oben nach rechts unten verdoppelt sich in jedem Schritt die Gruppen- und Bereichsgröße. Bild (e) zeigt zusätzlich, dass die Zeilen an sich noch nicht in der richtigen Reihenfolge sind, sondern diese erst beim Zurückschreiben in den Speicher erzielt wird. Jeder Schritt besteht aus einem `_mm_unpacklo_epixx`- und einem `_mm_unpackhi_epixx`-Befehl, welche die einzelnen Gruppen vereinen. Der Wertebereich der Befehle richtet sich dabei nach der Gruppengröße.

Um die im Min-Warping-Algorithmus vorkommenden 16×16 8-Bit-Matrizen zu transponieren, wären demnach bis zur Terminierung fünf rekursive Aufrufe notwendig. Statt der rekursiven Form mit den Schleifen in jedem Rekursionsschritt ist die Implementierung jedoch explizit für die gegebene Größe der Matrizen erfolgt, um den zusätzlichen Aufwand zur Ausführung der Schleifen zu vermeiden. Im gesamten Verlauf werden die `_mm_unpackxx_xx`-Befehle mit den Suffixen `_epi8`, `_epi16`, `_epi32` und `_epi64` verwendet. Abbildung 3.3 zeigt den Ablauf der Transponierung in diesem konkreten Fall am Beispiel der ersten Zeile.

Blockweises Transponieren mittels *Top-Down*-Ansatz

Während des rekursiven Transponierens mittels des *Top-Down*-Ansatzes wird in jedem Schritt die Folgenlänge verdoppelt und der Abstand zwischen den Elementen der Folge halbiert, wodurch im letzten Schritt der Zielabstand erreicht wird und die Folgen zu einer einzigen Gruppe pro Zeile verschmolzen werden. Um den Abstand in jedem Schritt zu halbieren, werden die Zeilen paarweise verzahnt. Das Einfügen neuer Elemente in die bestehenden Folgen bewirkt indirekt, dass die Elemente innerhalb einer Zeile zum Teil an ihre

```
function transpose_top_down(ROWS, STEPSIZE, BIT)
    STEPSIZE = floor(STEPSIZE / 2)

    if STEPSIZE < 1
        return ROWS
    endif

    for i = 0 to (STEPSIZE - 1) step 1
        TMP = ROWS[i];
        ROWS[i]             = _mm_unpacklo_epi@{BIT}(TMP, ROWS[i + STEPSIZE])
        ROWS[i + STEPSIZE] = _mm_unpackhi_epi@{BIT}(TMP, ROWS[i + STEPSIZE])
    endfor

    ROWS[0:(STEPSIZE - 1)] = transpose_top_down(ROWS[0:(STEPSIZE - 1)],
                                                STEPSIZE, BIT)
    ROWS[STEPSIZE:end]     = transpose_top_down(ROWS[STEPSIZE:end],
                                                STEPSIZE, BIT)
endfunction
```

Programmauszug 3.4: Ablauf des blockweisen Transponierens mittels *Top-Down*-Ansatz. Die Matrix wird zeilenweise in dem Array `ROWS` übergeben. Die Variable `STEPSIZE` gibt die Schrittweite des jeweiligen Rekursionsschritts an und über die Variable `BIT` wird festgelegt, für welchen Wertebereich die `_mm_unpackxx_epixx`-Befehle zu verwenden sind. Dieser ergibt sich aus der Größe der Ausgangsmatrix und ändert sich während der Rekursion nicht. Die Schrittweite wird direkt zu Beginn der Funktion halbiert und dient als Abbruchkriterium, sobald sie unter 1 gelangt. In der Schleife werden die Zeilen der entsprechenden Schrittweite paarweise kombiniert. Es ist ein Register als temporärer Zwischenspeicher notwendig, da ansonsten benötigte Werte überschrieben würden. Beim rekursiven Aufruf werden die Zeilen gemäß der *Divide and Conquer* Strategie in zwei gleich große Bereiche unterteilt.

Zielpositionen verschoben werden, anstatt dass alle Elemente im Wechsel mit anderen vertauscht werden. Da nicht die Positionen von Folgen als Ganzes, sondern nur die einzelner Elemente getauscht werden, sind nur `_mm_unpackxx_epixx`-Befehle eines Wertebereichs notwendig. Dieser richtet sich nach dem Wertebereich der Elemente.

Die kaskadenförmige Rekursion wird gemäß der *Divide and Conquer*-Strategie ausgeführt (Progr.-ausz. 3.4): In jedem Schritt der Rekursion werden die Zeilen in zwei Bereiche unterteilt, mit denen der rekursive Aufruf stattfindet. Die Teilergebnisse werden abschließend zusammengesetzt und ergeben die vollständig transponierte Matrix. Für eine $2^n \times 2^n$ Matrix wird die Rekursion mit allen Zeilen der Matrix und einer Schrittweite von 2^n gestartet. Diese wird direkt zu Beginn der Funktion halbiert und die Rekursion endet, wenn ein Wert kleiner 1 erreicht ist. Anschließend werden paarweise alle Zeilen im Abstand der Schrittweite kombiniert, wodurch sich zum einen die Anzahl der

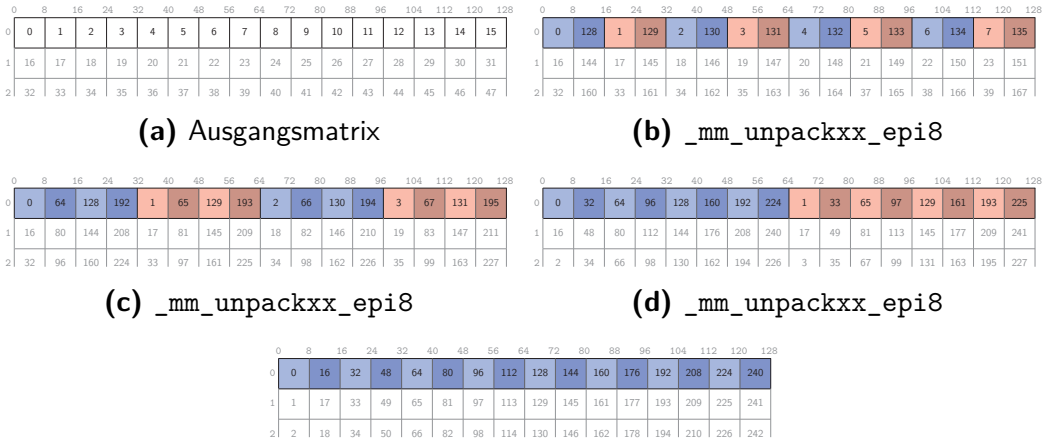
Folgen pro Zeile und der Abstand zwischen den Elementen einer Folge halbiert und sich zum anderen die Folgenlänge verdoppelt. Die initiale Schrittweite ist so gewählt, dass die anfänglichen Folgen bereits aus dem ersten und letzten Element der endgültigen Gruppen bestehen, sodass die Folgen durch einfaches Einfügen von fehlenden Elementen zu Gruppen vervollständigt werden können. Alle Elemente, die bereits in eine Folge integriert sind, werden durch das Einfügen neuer Elemente in Richtung ihrer Zielposition innerhalb der Zeile verschoben. Die beiden rekursiven Aufrufe werden jeweils auf der Hälfte der Zeilen durchgeführt. Nachdem alle Teilergebnisse zur vollständigen Matrix kombiniert wurden, haben alle Elemente den Zielabstand erreicht. Auch die Zeilen sind bereits in der richtigen Reihenfolge und können linear in den Speicher zurückgeschrieben werden. Zur Veranschaulichung des Algorithmus ist dieser am Beispiel einer 8×8 Matrix in Abbildung A.2 dargestellt. Die Umsetzung als C-Präprozessor-Makro befindet sich in Programmauszug C.11.

Ursprünglich lag der Algorithmus in Form von Inline-Assembler Befehlen vor, mit der eine Umsetzung mit weniger `movdqa`-Instruktionen möglich ist. Um jedoch in keinem Fall die Optimierungen des Compilers zu blockieren, ist der Algorithmus im Rahmen dieser Arbeit auf C-Präprozessor-Makros und Intrinsics umgeschrieben worden. Des Weiteren ist die Reihenfolge der rekursiven Aufrufe im Vergleich zur theoretischen Beschreibung leicht verändert und die `_mm_unpackxx_epi8`-Befehle sind passend zu den verwendeten 16×16 8-Bit-Matrizen fest vorgegeben. Da die Makros bereits durch den Compiler aufgelöst werden, entsteht zur Laufzeit kein zusätzlicher Aufwand für die Verwaltung der rekursiven Funktionsaufrufe, sodass keine explizite Implementierung notwendig ist. Für den konkreten Fall der Matrizen im Min-Warping-Verfahren ist der Ablauf in Abbildung 3.4 beispielhaft für die erste Zeile dargestellt.

Aufwand des blockweisen Transponierens

Bei beiden Methoden handelt es sich um sogenannte *In-Place*-Algorithmen[17], die die transponierte Matrix in die gleichen Register schreiben, in denen auch die Ausgangsmatrix bereitgestellt worden ist. Außerdem benötigen sie während des gesamten Ablaufs nur eine konstante Menge an zusätzlichem Speicher. Darin inbegriffen ist sowohl der benötigte Platz für Zwischenergebnisse als auch der Speicher zum Verwalten von Funktionsaufrufen. Auf unterster Ebene nutzen beide Methoden ein zusätzliches `xmm`-Register zum Zwischenspeichern

[17] Die Definition eine *In-Place*-Algorithmus erfolgt gemäß des gleichlautenden Wikipedia-Artikels: `http://en.wikipedia.org/wiki/In-place_algorithm`; letzter Zugriff: 8. August 2014.



(a) Ausgangsmatrix

(b) `_mm_unpackxx_epi8`

(c) `_mm_unpackxx_epi8`

(d) `_mm_unpackxx_epi8`

(e) `_mm_unpackxx_epi8`

Abbildung 3.4: Blockweises Transponieren einer 16×16 Matrix mittels *Top-Down*-Ansatz am Beispiel der ersten Zeile. Von links oben nach rechts unten verdoppelt sich in jedem Schritt die Folgenlänge und der Abstand zwischen den Elementen einer Folge halbiert sich. Jeder Schritt besteht aus einem `_mm_unpacklo_epi8`- und einem `_mm_unpackhi_epi8`-Befehl, durch die zwei Zeilen paarweise miteinander verzahnt werden.

einer Zeile der Matrix, da diese sonst zu früh überschrieben würde. Die *Tail Recursion* des *Bottom-Up*-Algorithmus lässt sich zudem unmittelbar in eine iterative Darstellung überführen, aus der ersichtlich ist, dass nur eine konstante Menge an zusätzlichem Speicher erforderlich ist. Aufgrund der *Divide and Conquer*-Strategie des *Top-Down*-Ansatzes ist dessen Umformung aufwändiger, resultiert aber auch in einer iterativen Form (Progr.-ausz. C.12), bei der der zusätzliche Speicherplatz nicht von der Größe der Ausgangsmatrix abhängt.

Neben der Menge des benötigten Speicherplatzes ist weiterhin die Anzahl der auszuführenden Befehle von Bedeutung. Im Folgenden werden dabei die Aufrufe der `_mm_unpackxx_epixx`-Befehle betrachtet. Der Gesamtaufwand wird an dieser Stelle nur am Beispiel der iterativen Form des *Top-Down*-Algorithmus analysiert, da die Herleitung für den *Bottom-Up*-Algorithmus wegen des ähnlichen Aufbaus der Schleifen analog erfolgen kann. Die äußerste Schleife (Progr.-ausz. C.12, Zeile 6) hängt direkt von der Größe der $2^n \times 2^n$ Ausgangsmatrix ab und wird aufgrund der initialen Schrittweite von 2^n und der Halbierung der Schrittweite in jeder Iteration $\log_2 2^n$ Mal durchlaufen. Die mittlere Schleife (Progr.-ausz. C.12, Zeile 9) hängt von der Anzahl der Bereiche ab, die sich wiederum in Abhängigkeit der Dimension der Matrix und der Schrittweite ausdrücken lässt (Gl. 3.6). Die innere Schleife

(Progr.-ausz. C.12, Zeile 12) ist direkt von der jeweiligen Schrittweite abhängig und wird dieser entsprechend häufig wiederholt. Zusätzlich besteht jede Operation aus zwei `_mm_unpackxx_epixx`-Befehlen. Aus diesen Beobachtungen lässt sich die Folgenden mathematische Betrachtung des Aufwandes ableiten.

$$\begin{aligned} 2^n &= \text{sections} \cdot 2 \cdot \text{stepsize} \\ \Leftrightarrow \text{sections} &= \frac{2^n}{2 \cdot \text{stepsize}} \end{aligned} \tag{3.6}$$

$$C_{\text{unpack}}(n) = \underbrace{\text{stepsize}}_{\substack{\text{innere}\\ \text{Schleife}}} \cdot \underbrace{\text{sections}}_{\substack{\text{mittlere}\\ \text{Schleife}}} \cdot \underbrace{\log_2(2^n)}_{\substack{\text{äußere}\\ \text{Schleife}}} \cdot \underbrace{2}_{\substack{\text{unpack}\\ \text{Befehle}}} \tag{3.7}$$

$$\overset{(3.6)}{=} \text{stepsize} \cdot \frac{2^n}{2 \cdot \text{stepsize}} \cdot \log_2(2^n) \cdot 2 \tag{3.8}$$

$$= 2^n \cdot \log_2(2^n) \tag{3.9}$$

Das Transponieren einer $m \times m$ Matrix erfordert demnach $m \cdot \log_2(m)$ `_mm_unpackxx_epixx`-Operationen. Ferner resultiert aus der logischen Betrachtung, dass jeder `_mm_unpackxx_epixx`-Befehl immer die Hälfte aller Elemente einer Zeile ($\log_2(m)$) umsortiert und für jede Zeile ein Befehlsaufruf erforderlich ist (m), derselbe Aufwand. Alleine mit `_mm_unpackxx_epixx`-Aufrufen ist somit kein Ablauf mit weniger Befehlen möglich.

3.3 Berechnung der Summe über Absolutwerte

Die Berechnung der beiden Teilsummen des Nenners der *Normalised Sum of Absolute Differences* ist in der angepassten Version des Min-Warping-Algorithmus ebenfalls durch das Intrinsic `_mm_sad_epu8` umgesetzt. Die neue Funktionalität wird durch die Funktion `sumOfAbsSIMDColPadByte` beziehungsweise ihre 32-Bit-Version (Suffix `32bit`) bereitgestellt. Ergänzend zu verschiedenen Varianten dieser beiden Funktionen werden in diesem Abschnitt verschiedene Ansätze zur Addition der zwei 64-Bit-Segmente nach der Anwendung des Befehls `_mm_sad_epu8` vorgestellt. Für das Verständnis wichtig ist außerdem, dass die Funktionen nur auf transponierten Bildern verwendet werden und sich daher in diesem Abschnitt die Begriffe *Breite* und *Höhe*, wenn nicht anders angegeben, ebenfalls auf die Dimension der transponierten Bilder und nicht die der Eingangsbilder beziehen.

```
for(int y = 0; y < h16; ++y, ++dst)                                          1
{                                                                            2
    as = _mm_setzero_si128();                                                3
                                                                             4
    for(int x = 0; x < w16; x += 16, src += 16)                              5
    {                                                                        6
        as = _mm_adds_epi16(as, _mm_sad_epu8(*(__m128i*)src, byteVec128));   7
    }                                                                        8
                                                                             9
    *dst = (SSEShort)_mm_cvtsi128_si32(as) +                                 10
           (SSEShort)_mm_cvtsi128_si32(_mm_srli_si128(as, 8));               11
}                                                                            12
```

Programmauszug 3.5: Direkte Umsetzung der Summe über Absolutwerte mittels des Befehls `_mm_sad_epu8`. Während der Berechnung wird der Bias `byteVec128` von den Intensitätswerten subtrahiert und die Spaltensumme ist aufgrund des Befehls `_mm_sad_epu8` zunächst in zwei getrennten Teilen des Registers `as` enthalten, die vor dem Abspeichern extrahiert und addiert werden.

3.3.1 Implementierte Schleifenreihenfolgen und Optimierungen

Die Umstellung auf den neuen Befehl erfordert bei einer direkten Umsetzung (Progr.-ausz. 3.5) nur eine Umkehr der Schleifenreihenfolge, weil die Bilder durch das Transponieren spalten- anstatt zeilenweise im Speicher liegen. Bedingt durch den Befehl `_mm_sad_epu8` muss von den Intensitätswerten `src` der Bias `byteVec128` subtrahiert werden (Progr.-ausz. 3.5, Zeile 7). Außerdem ist die Spaltensumme[18] `dst` zunächst auf die beiden 64-Bit-Segmente des Akkumulationsregisters `as` aufgeteilt. Die beiden Teilsummen werden einmalig, vor dem Schreiben des Ergebnisses in den Speicher, aus dem Register extrahiert und addiert (Progr.-ausz. 3.5, Zeile 10). Um den Einfluss der Schleifenvariablen `x` und `y` auf die Geschwindigkeit zu testen, wurde eine zeigerbasierte Version implementiert, welche die Lese- und Schreibposition direkt verschiebt und somit ohne zusätzliche Zählvariablen auskommt (Progr.-ausz. 3.6). Stattdessen sind allerdings zusätzliche Variablen notwendig, in denen die Abbruchbedingungen der Schleifen gespeichert werden.

Um die Summenbildung weiter zu beschleunigen, wurde für die direkte Umsetzung auch eine Form des *Array Preloadings* implementiert. Allgemein ist dabei das Ziel, den Zugriff auf lokal vom Algorithmus benötigte Daten

[18] Gemeint ist die Spaltensumme über den Eingangsbildern. Auf den transponierten Bildern wird sie durch das *zeilenweise* Aufaddieren der Elemente berechnet, da diese *spaltenweise* im Speicher abgelegt sind.

```
for(SSEShort *dst = absSum; dst < absSum + h16; ++dst)
{
    as = _mm_setzero_si128();

    for(const SSEByte *srcEnd = src + w16; src < srcEnd; src += 16)
    {
        as = _mm_adds_epi16(as, _mm_sad_epu8(*(__m128i*)src, byteVec128));
    }

    *dst = (SSEShort)_mm_cvtsi128_si32(as) +
           (SSEShort)_mm_cvtsi128_si32(_mm_srli_si128(as, 8));
}
```

Programmauszug 3.6: Zeigerbasierte Umsetzung der Summe über Absolutwerte mittels des Befehls `_mm_sad_epu8`

zu optimieren. Bei der Berechnung der Summe über Absolutwerte dient es dazu, die Lade- und Rechenoperationen zu entzerren. Deswegen werden alle Intensitätswerte, über die summiert werden soll, bereits vor Beginn der Rechnung in Register geladen. Dadurch treten *Cache Misses* unmittelbar hintereinander auf, sodass der Prozessor eher zum *Cachen* der Speicherbereiche angeregt wird, als wenn zwischen den *Cache Misses* immer wieder aus Sicht der Zugriffe erfolgreiche Berechnungen ablaufen. Zu bedenken ist allerdings, dass ein zusätzlicher Aufwand für das Vorausladen der Werte entsteht. Gerade in diesem Fall kann der Aufwand leicht den Nutzen, den die bessere Erreichbarkeit der Daten mit sich bringt, übersteigen, da die vorausgeladenen Werte nur ein einziges Mal benutzt werden. Bei der Berechnung der *Sum of Absolute Differences* als Zähler der *Normalised Sum of Absolute Differences* sind deutlichere Vorteile des *Array Preloadings* zu erwarten. Dort werden die Werte für mehrere Berechnungen im Voraus geladen, sodass sich die Anzahl der Speicherzugriffe erheblich verringert.

Für die Berechnungen im Nenner sind zwei unterschiedliche Varianten des *Array Preloadings* implementiert: Eine mit statischer (Progr.-ausz. C.13) und eine mit dynamischer Arraygröße (Progr.-ausz. C.14). Die statische Implementierung nutzt ein Array fester Größe, sodass bei kleinen Bildern oder solchen, bei denen die Breite kein ganzzahliges Vielfaches der im Voraus ladbaren Elemente ist, ein Teil der Register ungenutzt bleibt, aber auch an anderer Stelle nicht zur Verfügung steht. Das dynamische Array enthält immer die minimal benötigte Anzahl an Registern und ist daher ressourcenschonender.

Zwischen der 16- und 32-Bit-Version sind die einzigen beiden Unterschiede, dass beim Akkumulieren des Registers `as` die 32-Bit-Version des Intrinsics genutzt wird (`_mm_add_epi32`, saturierte Arithmetik steht nur bis 16-Bit zur

Verfügung) und dass die Teilsummen vor der letzten Addition zu `SSEInt` anstatt zu `SSEShort` gecastet werden. Alle weiteren Befehle können übernommen werden, da der Befehl `_mm_sad_epu8` alle nicht genutzten Teile des Registers mit Nullen belegt.

3.3.2 Horizontale Addition von High- und Low-Teil eines 128-Bit-`xmm`-Registers

Das Intrinsic `_mm_sad_epu8` summiert nicht direkt über alle Werte, sondern legt wie bereits beschrieben jeweils die Summe aus 8 Werten in den oberen und unteren 64 Bit des Registers ab. Diese beiden Werte müssen abschließend horizontal im Register addiert werden, um die gesamte Spaltensumme zu erhalten. Für die Addition steht dabei eine Reihe von Befehlen zur Verfügung.

Sequentielle Addition

Anstatt die Teilsummen noch im Register zu kombinieren, besteht die Möglichkeit, sie zuerst zu extrahieren und dann als skalare Größen zu addieren. Generell kann dazu der Zugriff über eine *Union* (Progr.-ausz. 3.7) oder spezielle Intrinsics (Progr.-ausz. 3.8) genutzt werden. Mit den GCC Vector Extensions (Abschn. 2.3.2) ist zudem der indexbasierte Zugriff auf die 64-Bit-Segmente möglich (Progr.-ausz. 3.9).

Union Durch die Union ist lediglich ein Intrinsic zum Schreiben des Registerinhalts in den Speicher notwendig, um die Teilsummen aus dem Register zu extrahieren. Diese können anschließend per indirekter Adressierung in zwei Allzweckregister geladen und addiert werden. Der Wertebereich des Ergebnisses resultiert dabei aus den verwendeten Segmenten der Register (z. B. `r10d` als *Low-DWord* von `r10`). Durch die Schreib- und Leseoperationen sind jedoch vergleichsweise viele Speicherzugriffe erforderlich.

Intrinsics Intel stellt mit `_mm_extract_epi16` zwar einen dedizierten Befehl zum Extrahieren von 16-Bit-Werten zur Verfügung, für 32-Bit-Werte ist jedoch bis zur Version SSSE3 kein Intrinsic definiert. Zudem ergeben sich in der Theorie für die Kombination aus `_mm_srli_si128` (Verschiebung) und `_mm_cvtsi128_si32` (Konvertierung) eine bessere Latenz und ein geringerer Throughput. Da das Ergebnisregister durch den Befehl `_mm_sad_epu8` bis auf die beiden Teilsummen komplett mit Nullen aufgefüllt wurde, kann die 32-Bit-Konvertierung auch für die 16-Bit-Variante verwendet werden. Das Verschieben ist außerdem nur beim Zugriff auf die oberen 64 Bit des Registers

```
union
{
    __m128i r;
    uint16_t w[8];
} as;

as.r = _mm_adds_epi16(as.r, ...);
*dst = as.w[0] + as.w[4]
```

(a) C-Code

```
movaps XMMWORD PTR [rsp-24], xmm13
mov    rax, QWORD PTR [rsp-24]
mov    r10, QWORD PTR [rsp-16]
add    eax, r10d
mov    WORD PTR [rcx-2], ax
```

(b) Assemblercode

Programmauszug 3.7: Registerzugriff mittels einer Union. Mithilfe der Komponente `r` der Union kann das Register des Typs `__m128i` an die Funktion `_mm_adds_epi16` übergeben werden. Die Teilsummen werden abschließend über die Elemente `w[i]` des Word-Arrays ausgelesen und addiert. Im dazugehörigen Assemblercode wird das `xmm`-Register vor dem Auslesen in den Speicher geschrieben, sodass die Teilsummen an den um 8 Byte verschiedenen Speicherstellen erreichbar sind. Das *Casten* wird durch die unterschiedlichen Größen der Allzweckregister umgesetzt. Den Befehl `movaps` statt dem für Ganzzahlen vorgesehenen `movdqa` zu verwenden, ist eine kontextabhängige Entscheidung des Compilers; beide Instruktionen sind von der Funktion her identisch, können jedoch unterschiedliche Auswirkungen auf die Performanz haben (Fog, 2014b).

notwendig, da bei den unteren die einfache Konvertierung ausreicht. Die durchgängige Verwendung von Registern macht den Speicherzugriff erst beim Schreiben der Werte erforderlich.

Indexbasierter Zugriff Die GCC Vector Extensions definieren die 128-Bit-`xmm`-Register als zweielementige 64-Bit-Vektoren, was es in diesem Fall möglich macht, die beiden Teilsummen direkt extrahieren, casten und addieren zu können. Daraus resultiert der einfachste und gemäß den theoretischen Werten effizienteste Ansatz für die Addition.

Horizontales Addieren per Intrinsic

Eine andere Möglichkeit, die beiden Teilsummen zu addieren, bietet der *horizontale* Befehl `_mm_hadd_epi32`, der in den beiden Quellregistern jeweils 2 benachbarte 32-Bit-Elemente addiert und die vier Summen im Zielregister speichert. Da die Teilsummen jedoch zunächst nicht benachbart sind, muss der Befehl entweder zweimal hintereinander angewendet werden (Progr.-ausz. 3.10a) oder die Elemente werden zuvor im Register umsortiert (Progr.-ausz. 3.10b). Im Gegensatz zu den vorherigen Ansätzen ist hierbei nur eine Extraktion notwendig.

```
*dst =
 (SSEShort)_mm_cvtsi128_si32(as) +
 (SSEShort)_mm_cvtsi128_si32(
     _mm_srli_si128(as, 8));
```

(a) C-Code

```
movdqa xmm1, xmm0
movd   r10d, xmm1
psrldq xmm0, 8
movd   r11d, xmm0
add    r10d, r11d
mov    WORD PTR [rcx-2], r10w
```

(b) Assemblercode

Programmauszug 3.8: Registerzugriff mittels Intrinsics. Die unteren 64 Bit können direkt in eine 32-Bit-Ganzzahl konvertiert und zu `SSEShort` gecastet werden. Bei den oberen 64 Bit ist vorher eine Verschiebung um 8 Byte notwendig. Der erzeugte Assemblercode entspricht nahezu einer direkten Umsetzung der Intrinsics: Über die Instruktion `movd` erfolgt das Kopieren der Doppelwörter und mittels `psrldq` erfolgt im Fall der oberen 64 Bit die Verschiebung. Lediglich das Duplizieren des `xmm`-Registers zu Beginn ist ein zusätzliche Instruktion und dient vermutlich einer effizienteren Parallelisierung, da dadurch auf zwei Registern gearbeitet werden kann.

```
*dst = (SSEShort)as[0] +
       (SSEShort)as[1];
```

(a) C-Code

```
movhlps xmm5, xmm0
movq    r11, xmm0
movq    r10, xmm5
add r11d, r10d
mov WORD PTR [rcx-2], r11w
```

(b) Assemblercode

Programmauszug 3.9: Registerzugriff mittels Index. Die GCC Vextor Extensions machen es möglich, die beiden 64-Bit-Segmente des 128-Bit-Registers direkt anzusprechen und zu addieren. Die oberen 64 Bit werden dabei, wie der Assemblercode zeigt, in den unteren Teil eines zweiten Registers kopiert. So können danach beide Teilsummen in Allzweckregister geladen und addiert werden. Die Typisierung erfolgt durch die Zugriffsart auf die Allzweckregister (z. B. `r10d` als *Low-DWord* von `r10`).

```
as = _mm_hadd_epi32(as, as);
as = _mm_hadd_epi32(as, as);
*dst =
 (SSEShort)_mm_cvtsi128_si32(as);
```

(a) Doppelter Aufruf

```
as = _mm_shuffle_epi32(as, 0x20);
as = _mm_hadd_epi32(as, as);
*dst =
 (SSEShort)_mm_cvtsi128_si32(as);
```

(b) Vorheriges Umsortieren

Programmauszug 3.10: Horizontales Addieren per Intrinsic. In (a) bewirkt der erste Aufruf des Intrinsics `_mm_hadd_epi32` eine Verschiebung der oberen Teilsumme in den unteren Teil des Registers, in (b) wird dies durch `_mm_shuffle_epi32` mit einer entsprechenden Bitmaske erreicht.

Doppelte Anwendung Der erste Aufruf des Befehls `_mm_hadd_epi32` trägt noch nicht zur Addition bei, sondern bewirkt, dass die beiden Teilsummen sich anschließend benachbart in den unteren 64 Bit eines Registers befinden. Beim zweiten Mal werden sie addiert und das Ergebnis kann extrahiert und in den Speicher geschrieben werden. Die prinzipielle zweifache Summenbildung hat keinen Einfluss auf das Ergebnis, da alle irrelevanten Elemente des Registers durch den Befehl `_mm_sad_epu8` auf Null gesetzt wurden.

Vorheriges Umsortieren Auch der Befehl `_mm_shuffle_epi32` lässt sich verwenden, um die beiden Teilsummen an benachbarte Positionen im Register zu bringen. Dabei ist die Bitmaske so gewählt, dass die obere Teilsumme ebenfalls in die unteren 64 Bit verschoben wird und die untere Teilsumme sich nicht ändert. Das obere Segment des Zielregisters spielt für die anschließende Addition keine Rolle, wodurch die Bitmaske in diesem Bereich beliebig definiert sein kann. Laut Intel Intrinsics Guide (Intel, 2014d) ergeben sich für diese Art der Verschiebung bessere Werte für Latenz und Throughput.

3.4 Berechnung der *Sum of Absolute Differences*

Die Berechnung der *Sum of Absolute Differences* ist im Rahmen dieser Arbeit ähnlich wie die Umsetzung der *Summe über Absolutwerte* (Abschn. 3.3) auch auf den Befehl `_mm_sad_epu8` umgestellt worden. Der Unterschied besteht darin, dass an dieser Stelle über zwei Bilder iteriert und die Differenz zwischen den Elementen gebildet wird, anstatt von einem Eingangsbild den Bias zu subtrahieren. So existieren auch für die Funktion `normSumAbsDiffTermColPadByte` sowohl zählerbasierte als auch zeigerbasierte Versionen, die optional mit statischem *Array Preloading* (Progr.-ausz. 3.11) verwendet werden können. Anders als bei der Berechnung des Nenners dient hier das *Array Preloading* nicht primär der Trennung von Lade- und Rechenoperationen, sondern es werden mehrfach verwendete Daten in möglichst großen Blöcken eingelesen. Das Auslesen eines großen Blocks sollte ein effizienteres *Caching* erwirken, als ein wiederholtes Lesen von kürzeren Abschnitten. Das *Preloading* (Progr.-ausz. 3.11, Zeile 21) wird nur auf die Zeilen eines der beiden Bilder angewendet (Progr.-ausz. 3.11, Zeile 11), da diese einmal geladenen Werte für alle Zeilen des anderen Bildes (Progr.-ausz. 3.11, Zeile 24) zur Bildung der Differenzen benötigt werden. Da nicht sichergestellt werden kann, dass eine komplette Zeile des Bildes vorausgeladen werden kann, müssen bei großen Bildbreiten mehrere Blöcke geladen werden (Progr.-ausz. 3.11, Zeile 15), bis eine komplette Zeile verarbeitet ist.

```
const SSEByte *src1 = l1;                                                    1
const SSEByte *src2 = l2;                                                    2
SSEShort *dst = dir;                                                         3
                                                                             4
__m128i as;                                                                  5
__m128i preload[8];                                                          6
int max = (w16 > 128 ? 128 : w16) >> 4;                                      7
                                                                             8
memset(dir, 0x00, h16 * h16 * sizeof(SSEShort));                             9
                                                                             10
for(int y1 = 0; y1 < h16; ++y1)                                              11
{                                                                            12
    src1 = l1 + w16 * y1;                                                    13
                                                                             14
    for(int base = 0; base < w16; base += 128)                               15
    {                                                                        16
        dst = dir + h16 * y1;                                                17
                                                                             18
        for(int i = 0; i < max; ++i)                                         19
        {                                                                    20
            preload[i] = *((__m128i*)(src1 + base + i * 16));                21
        }                                                                    22
                                                                             23
        for(int y2 = 0; y2 < h16; ++y2)                                      24
        {                                                                    25
            src2 = l2 + w16 * y2;                                            26
            as = _mm_setzero_si128();                                        27
                                                                             28
            for(int i = 0; i < max; ++i)                                     29
            {                                                                30
                as = _mm_adds_epi16(                                         31
                        as,                                                  32
                        _mm_sad_epu8(preload[i],                             33
                                     *((__m128i*)(src2 + base + i*16))));    34
            }                                                                35
                                                                             36
            as = _mm_shuffle_epi32(as, 0x0020);                              37
            as = _mm_hadd_epi32(as, as);                                     38
            *dst += (SSEShort)_mm_cvtsi128_si32(as);                         39
                                                                             40
            ++dst;                                                           41
        }                                                                    42
    }                                                                        43
}                                                                            44
```

Programmauszug 3.11: Direkte Umsetzung der *Sum of Absolute Differences* mittels des Befehls `_mm_sad_epu8` inklusive *Array Preloading* Optimierung. Das *Array Preloading* ist für die Elemente des ersten Bildes umgesetzt, die bei der Berechnung der SAD mehrfach mit Elementen aus dem zweiten Bild kombiniert werden müssen. Es wird jeweils ein Block aus dem ersten Bild geladen (Zeile 21) und mit allen Zeilen des zweiten Bildes kombiniert (Zeile 24). Da nicht garantiert ist, dass komplette Zeilen im Voraus geladen werden können, ist eine weitere Schleife notwendig (Zeile 15).

Die Schleifenreihenfolge ist dabei identisch zu der des bestehenden Algorithmus, sodass die Elemente partiell berechnet werden (Abschn. 2.4.1). Das bedeutet, dass für jede Zeile des ersten Bildes die SAD-Werte gleichzeitig für alle Zeilen des zweiten Bildes schrittweise im Speicher akkumuliert werden. Diese sind solange unvollständig, bis alle Spalten der Zeile des ersten Bildes durchlaufen wurden. Zusätzlich ist die *natürliche* Reihenfolge implementiert, bei der direkt zwei vollständige Zeilen zur Berechnung einer SAD genutzt werden (Progr.-ausz. C.15). Dabei werden die beiden Zeilen parallel durchlaufen und die Differenzen aus zwei Elementen direkt in einem Register summiert. Somit wird die endgültige SAD einmalig in den Speicher geschrieben.

Ebenso wie bei der in Abschnitt 3.3 beschriebenen Funktion `sumOfAbsSIMDColPadByte`, sind alle vorgestellten Methoden zur horizontalen Addition der beiden Registerteile implementiert. Aufgrund des anderen Kontexts und der zusätzlichen Berechnungen ist es denkbar, dass in dieser Funktion eine andere Art der Addition effizienter abläuft.

3.5 Berechnung der *Normalised Sum of Absolute Differences*

Die Funktion `calculateNormSumAbsDiffSPS` kombiniert die zuvor berechneten Komponenten zur *Normalised Sum of Absolute Differences*, skaliert diese passend für die weitere Verwendung in der zweiten Phase und erstellt den *Scale Plane Stack*. Die 16-Bit-Version (Progr.-ausz. C.16) entspricht der bisherigen Implementierung. Bei der 32-Bit-Version bleiben die Konvertierung der Eingangsbilder, die Anwendung des Kantenfilters und das Kopieren, Vergrößern und Skalieren ebenfalls bestehen. Zur Berechnung des Zählers und des Nenners werden die 32-Bit-Varianten der Funktionen verwendet, deren Veränderungen im Vergleich zur bisherigen oder zur 16-Bit-Version aus den jeweiligen Abschnitten dieses Kapitels ersichtlich sind. Zudem hat sich die Vorbereitung der Daten auf die Division vereinfacht: Bisher mussten die Werte vom Typ `SSEShort` über das Makro `_MM_8SHORT_TO_2X4FLOAT` (Progr.-ausz. C.1) in Gleitkommazahlen umgewandelt werden. Darin ist jedoch auch eine zeitweise Konvertierung der Werte in den Typ `SSEInt` enthalten, sodass dieser Zwischenschritt in der 32-Bit-Variante wegfällt und direkt die Funktion `_mm_cvtepi32_ps` verwendet wird.

4

Entwicklungs- und Testumgebung

In diesem Kapitel wird zunächst kurz die für die Zeitmessung verwendete Hardware beschrieben und auf die Unterschiede in den Spezifikationen hingewiesen. Im zweiten Abschnitt wird auf die verfügbaren Compiler eingegangen und anschließend weitere Software erwähnt, die zur Effizienzanalyse genutzt werden kann. Der letzte Abschnitt charakterisiert die durchgeführten Zeitmessungen und nennt die statistischen Parameter.

4.1 Hardware

Zum Evaluieren der Funktionalität und der Effizienz kamen drei verschiedene Prozessoren zum Einsatz: Zweimal ein *Intel® Core™ i7* und ein *Intel® Atom™*. Die Hauptunterscheidungskriterien zwischen diesen sind die Taktrate, die Cachegröße und die Architektur. Eine detailierte Übersicht über die jeweiligen Kennzahlen bietet Tabelle 4.1.

4.2 Software

Im Folgenden Abschnitt werden zuerst die eingesetzten Compilerversionen genannt sowie die verwendeten Aufrufparameter kurz erläutert. Der zweite Abschnitt umreißt die Methoden der Programmcode- und Laufzeitanalyse und stellt geeignete Programme für beide Anwendungsfälle dar.

	Atom	*i7_2ndGen*	*i7_3rdGen*
Bezeichnung	Intel® Atom™ CPU N2600	Intel® Core™ i7-2600	Intel® Core™ i7-3770
Architektur	Cedarview	Sandy Bridge	Ivy Bridge
Kerne	2	4	4
Taktrate	1.60 GHz	3.40 GHz	3.40 GHz
Cachegröße	512 kB	8 192 kB	8 192 kB

Tabelle 4.1: Darstellung der wesentlichen Kennzahlen der drei verwendeten Intel Prozessoren

4.2.1 Compiler

Auf den Computern mit dem *Intel® Core™ i7* der 2. Generation und dem *Intel® Atom™* Prozessor ist unter Ubuntu 12.04 standardmäßig die Version 4.6.3 der GCC installiert. Seit der Einführung im März 2012 wurde die SSE-Unterstützung jedoch deutlich verbessert und um neue Eigenschaften ergänzt. Daher ist es im Hinblick auf eine eventuelle Aktualisierung des Compilers sinnvoll, alle implementierten Neuerungen sowie das gesamte Min-Warping-Verfahren ebenfalls mit der zum Beginn dieser Arbeit aktuellsten Version 4.9.0[19] zu kompilieren. Zur Optimierung werden die Parameter `-O3`, `-mfpmath=sse` und `-funroll-loops` an den Compiler übergeben. Diese aktivieren die höchste Stufe der automatischen Optimierungen und geben an, die gesamte Gleitkommaarithmetik durch die skalaren SSE-Instruktionen umzusetzen und den Programmcode für alle Schleifen mit einer festen Anzahl an Iterationen zu duplizieren, anstatt ihn wiederholt zu durchlaufen. Für die letzten beiden Parameter hängt es wesentlich von der Hardware und dem Kontext des Programms ab, ob die Effizienz gesteigert werden kann. Zusätzliche werden über die Option `-mssse3` SSE-Befehle bis einschließlich Version SSSE3 eingeschaltet. Die beiden Intel® Core™ i7 Prozessoren verfügen auch über SSE4-Unterstützung, der Intel® Atom™ Prozessor hingegen nicht.

[19] Am 16. Juli ist die Aktualisierung auf Version 4.9.1 veröffentlicht worden. Aus Zeitgründen sind jedoch keine erneuten Messungen möglich.

4.2.2 Analyseprogramme und Profiler

Zeitmessungen geben meistens nur einen groben Überblick über die Gesamtlaufzeit eines Algorithmus oder die erlangten Verbesserungen. Zur gezielten Suche nach Flaschenhälsen im Programmablauf oder den Ursachen für eine Steigerung der Performanz sind spezielle Analyseprogramme besser geeignet. Diese lassen sich in die Gruppen *statisch* und *echtzeitfähig* unterteilen.

Als statisches Werkzeug stellt Intel den *Intel® Architecture Code Analyzer (IACA)* kostenfrei zur Verfügung. Mit diesem ist es möglich, die Latenz und den Throughput von Codesequenzen zu analysieren. Da es sich allerdings um eine statische Methode handelt, können bedingte Ausdrücke und Schleifen nicht berücksichtigt werden und es muss die gewünschte Zielarchitektur manuell ausgewählt werden. Der Intel® Atom™ Prozessortyp wird dabei nicht unterstützt. Daher konnte das Programm zwar für theoretische Betrachtungen während der Entwicklung der Anpassungen verwendet werden, eine vergleichende Auswertung für die verschiedenen verwendeten Systeme ist jedoch nicht möglich.

Die Messung von Parametern wie *Cache Hits* oder *Cache Misses*, verursachte *Pipeline Stalls* oder die tatsächliche *Belegung der einzelnen Ausführungseinheiten der CPU* ist innerhalb der Gruppe der sogenannten *Profiler* möglich. Auch hier bietet Intel mit dem *Intel® VTune™ Amplifier XE 2013* eine – diesmal kostenpflichtige – Lösung an. Als kostenfreie Alternativen wurden im Rahmen dieser Arbeit die für Unix Systeme verfügbaren Programme *perf*[20] und *gprof*[21] sowie das *HPCToolkit*[22] getestet. Um diese komplexeren Programme zielgerichtet nutzen und die Ergebnisse auswerten zu können, wäre eine umfangreiche Einarbeitung notwendig gewesen, für die im Rahmen dieser Arbeit keine Zeit vorgesehen war. Daher bleibt die Analyse des Min-Warping-Verfahrens mittels der verschiedenen Profiler als offener Punkt bestehen. Weitere ungetestete Möglichkeiten zur Analyse bieten die von Agner Fog bereitgestellten Programme[23], *valgrind*[24], der *Intel® Performance Counter Monitor (PCM)*[25] und die eigenständige Nutzung der verfügbaren *Performance Monitoring Counters* (Intel, 2014b, Kapitel 23).

[20] `http://en.wikipedia.org/wiki/Perf_(Linux)`; letzter Zugriff: 8. August 2014.
[21] `http://en.wikipedia.org/wiki/Gprof`; letzter Zugriff: 8. August 2014.
[22] `http://hpctoolkit.org`; letzter Zugriff: 8. August 2014.
[23] `http://www.agner.org/optimize/testp.zip`; letzter Zugriff: 8. August 2014.
[24] `http://valgrind.org`; letzter Zugriff: 8. August 2014.
[25] `http://www.intel.com/software/pcm`; letzter Zugriff: 8. August 2014.

4.3 Durchführung der (Zeit-)Messungen

Die Zeitmessungen können dank einer zu diesem Zweck geschriebenen Klasse vollautomatisiert durchgeführt werden. Diese erbt von der Klasse des Min-Warping-Verfahrens, sodass die neuen Testmethoden Zugriff auf die geschützten Methoden des Min-Warping-Algorithmus erhalten. Die Testmethoden selber sind öffentlich verfügbar und bündeln die verschiedenen Versionen der neu implementierten Methoden des Min-Warping-Verfahrens. Pro Version wird auf den Intel® Core™ i7 Prozessoren der Mittelwert über 100 Zeitmessungen und auf dem langsamerem Intel® Atom™ der Mittelwert über 10 Zeitmessungen gebildet. Die Varianzen der Messungen unterscheiden sich dadurch nicht signifikant, sodass von einer Vergleichbarkeit der Ergebnisse ausgegangen werden kann. Bei jeder Zeitmessung wird die Gesamtlaufzeit einer Schleife über den Aufruf der jeweiligen Methode gemessen (Progr.-ausz. C.17). Die Laufzeit eines einzelnen Aufrufs ist zu gering, als das repräsentative Aussagen getroffen werden können. Zusammenhängende Auswertungen werden per Shellscript hintereinander ausgeführt, wobei vor den protokollierten Messungen jeder Methode 1 zusätzlicher Aufruf getätigt wird, damit sich die CPU nicht mehr in einem heruntergetakteten Energiesparzustand befindet, wenn die Laufzeiten bestimmt werden.

Die abschließenden Zeitmessungen auf dem gesamten Min-Warping-Verfahren werden mittels des bestehenden Programms `warpingSIMD2Test` (mit Parameter `s`) durchgeführt. Diese Messungen wurden für alle Prozessoren über 10 Durchläufe gemittelt. Des Weiteren beinhalten die Ergebnisse des Programms `warpingSIMD2Test` die mittleren Winkelfehler für die Parameter ψ und β. Dadurch kann auch die Qualität der angepassten Min-Warping-Implementierung mit der bestehenden verglichen werden. Als *Snapshot* wurde sowohl in der *living3_day_day_Hh288sh_0.10* als auch in der *living3_day_night_Hh288sh_0.10* Datenbank das Bild mit den Koordinaten $(10, 4)$ gewählt.

5

Ergebnisse

In diesem Kapitel sind die Ergebnisse der durchgeführten Zeitmessungen zusammengefasst. Bei allen Messungen wurden die sechs möglichen Kombinationen aus Prozessortyp und Compilerversion getestet, sodass ein umfassender Vergleich durchführbar ist. Um einen guten Überblick zu geben, ist jeweils die schnellste Zeit einer Konstellation in den Tabellen hervorgehoben. Mit Ausnahme des ersten Abschnitts und der abschließenden Zeitmessung für das gesamte Min-Warping-Verfahren können alle Laufzeiten direkt miteinander verglichen werden. Dadurch ist eine Auswertung hinsichtlich der Lastverteilung innerhalb des Min-Warping-Algorithmus möglich. Die Messwerte in diesen Abschnitten ergeben sich aus dem Mittelwert über 100 Zeitmessungen für die Intel® Core™ i7 Prozessoren und über 10 Zeitmessungen für den Intel® Atom™ Prozessor. Jede Zeitmessung umfasst die Gesamtlaufzeit einer Schleife über 100 000 Aufrufe der jeweiligen zu testenden Methode.

5.1 Laufzeiten des Kopierens, Vergrößerns und Skalierens der kantengefilterten Eingangsbilder

Dieser Abschnitt fasst die Ergebnisse der Zeitmessungen für die Verarbeitung der kantengefilterten Eingangsbilder zusammen. In Tabelle 5.1 sind die Zeiten für ein Kopieren der Bilder ohne Vergrößerung und in Tabelle 5.2 die Ergebnisse für das Kopieren inklusive einer Vergrößerung mittels Interpolation der Werte im Vergleich zur bestehenden Implementierung dargestellt. Die Lauf-

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
cassf_short	1 477	1 475	108	99	101	91
cassf_st	2 928	2 878	412	468	385	387
cassf_bt	*1 714*	1 855	*152*	*161*	*141*	*141*
cassf_btmemxy	1 973	2 021	168	168	158	155
cassf_btmemyx	1 872	1 935	178	175	168	165
cassf_bt2	1 735	*1 824*	164	165	158	147
cassf_bt2memxy	1 950	2 030	180	175	170	158
cassf_bt2memyx	1 882	1 922	187	181	178	170

Tabelle 5.1: Laufzeiten des Kopierens und Skalierens der kantengefilterten Eingangsbilder. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *cassf* steht für die Methode `copyAndScaleSIMDFloat`. Bei den Suffixen gibt *short* die bisherige Implementierung an, *st* die Version mit sequentiellem Transponieren und *bt* sowie *bt2* kennzeichnen die beiden unterschiedlichen Methoden der Blocktransponierung. Letztere werden mit *memxy* und *memyx* durch zwei Schleifenreihenfolgen für die Umsetzungen mit größerem Speicherplatz (Progr.-ausz. C.7 und C.7) ergänzt.

zeiten sind dabei für alle Arten und Schleifenreihenfolgen des Transponierens ermittelt worden.

Aus den Tabellen ist ersichtlich, dass alle Versionen der neuen Implementierung langsamer als die bestehende sind. Des Weiteren ist ein blockweises Matrixtransponieren in beiden Fällen effizienter als ein sequentielles. Zwischen den zwei Varianten der blockweisen Transponierung existieren nur geringe Unterschiede.

5.2 Laufzeit von kontextunabhängigen horizontalen Additionen

In Abschnitt 3.3.2 wurden verschiedene Methoden vorgestellt und erläutert, mit denen es möglich ist, den High- und Low-Teil eines 128-Bit-`xmm`-Registers zu addieren. Tabelle 5.3 listet die Laufzeiten einer wiederholten Addition, die unabhängig von der eigentlichen Min-Warping-Implementierung durchgeführt wurde. Die in Millisekunden angegebenen Zeiten resultieren für die Intel® Core™ i7 Prozessoren aus dem Mittelwert über 100 Durchläufe, für den Intel® Atom™ Prozessor aus dem Mittelwert über 10 Durchläufe. Jeder

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
massf_short	2 630	2 595	301	294	275	270
massf_st	4 128	4 066	579	689	533	575
massf_bt	*2 922*	*3 009*	*348*	*353*	*314*	*320*
massf_btmemxy	3 061	3 160	354	369	326	346
massf_btmemyx	3 205	3 141	367	367	337	338
massf_bt2	2 959	3 032	350	358	323	327
massf_bt2memxy	3 111	3 175	361	374	336	348
massf_bt2memyx	3 184	3 104	385	373	354	346

Tabelle 5.2: Laufzeiten des Vergrößerns und Skalierens der kantengefilterten Eingangsbilder. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *massf* steht für die Methode `magnifyAndScaleSIMDFloat`. Bei den Suffixen gibt *short* die bisherige Implementierung an, *st* die Version mit sequentiellem Transponieren und *bt* sowie *bt2* kennzeichnen die beiden unterschiedlichen Methoden der Blocktransponierung. Letztere werden mit *memxy* und *memyx* durch zwei Schleifenreihenfolgen für die Umsetzungen mit größerem Speicherplatz (Progr.-ausz. C.7 und C.8) ergänzt.

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
extract	1 836	1 828	251	250	244	241
cvtsi128	2 097	1 773	269	251	256	243
index	—	*1 397*	—	237	—	*234*
union	*1 397*	*1 397*	*244*	*234*	*234*	237
2hadd	2 291	1 899	300	279	291	272
shuffle	2 103	1 711	269	247	258	244

Tabelle 5.3: Laufzeiten einer wiederholten horizontalen Addition unabhängig vom Min-Warping-Algorithmus. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 Millionen Additionen. Die Bezeichner stehen in der Reihenfolge der Zeilen für das Extrahieren der Teilsummen mittels `_mm_extract_epi16`, den Zugriff mittels der Intrinsics `_mm_cvtsi128_si32` und `_mm_srli_si128`, den indexbasierten Zugriff, den Zugriff über eine Union, die zweimalige Verwendung des Befehls `_mm_hadd_epi32` und das Umsortieren der Werte vor dem Aufruf des Befehls `_mm_hadd_epi32`. Der indexbasierte Zugriff wird von der GCC in Version 4.6.3 noch nicht unterstützt, wodurch damit für die Methode keine Laufzeiten ermittelt werden konnten.

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
soas_extract	617	581	79	80	62	61
soas_cvtsi128	635	599	87	82	62	61
soas_index	—	545	—	86	—	58
soas_union	617	671	86	82	62	62
soas_2hadd	690	618	80	81	71	70
soas_shuffle	*559*	*508*	*62*	*56*	*60*	*54*

Tabelle 5.4: Laufzeiten der Summe über Absolutwerte in Bezug auf die horizontale Addition. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *soas* steht für die Methode `sumOfAbsSIMD`. Die Suffixe stehen in der Reihenfolge der Zeilen für das Extrahieren der Teilsummen mittels `_mm_extract_epi16`, den Zugriff mittels der Intrinsics `_mm_cvtsi128_si32` und `_mm_srli_si128`, den indexbasierten Zugriff, den Zugriff über eine Union, die zweimalige Verwendung des Befehls `_mm_hadd_epi32` und das Umsortieren der Werte vor dem Aufruf des Befehls `_mm_hadd_epi32`. Der indexbasierte Zugriff wird von der GCC in Version 4.6.3 noch nicht unterstützt, wodurch damit für die Methode keine Laufzeiten ermittelt werden konnten. Alle Methoden wurden jeweils in die zählerbasierte 16-Bit-Schleifenvariante eingebunden.

Durchlauf bestand wiederum aus 100 Millionen hintereinander ausgeführten Additionen. Laut den Messungen ist je nach Architektur und Compilerversion der indexbasierte Zugriff oder die Verwendung einer Union die schnellste Umsetzung.

5.3 Laufzeit der Summe über Absolutwerte

Da zu vermuten ist, dass der Kontext, in dem die Additionen ausgeführt werden auch eine (erhebliche) Rolle spielt, wird die Messung über alle vorgestellten Methoden wiederholt. Für diese zweiten Messungen (Tab. 5.4) sind die Additionen bereits an der endgültigen Stelle in den Min-Warping-Algorithmus eingebunden und bei allen Methoden wurde die zählerbasierte 16-Bit-Schleifenvariante (Progr.-ausz. 3.5) zur Evaluation genutzt. Die angegebenen Laufzeiten beinhalten demnach nicht mehr nur die für die Addition benötigte Zeit, sondern sind für den gesamten Ablauf der Funktion gemessen worden. In diesem Fall ist die Methode des Umsortierens der Datenelemente und eine anschließende horizontale Addition innerhalb des Registers für alle Konfigurationen am effizientesten.

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
soas_lc	*599*	508	*61*	*57*	*58*	*55*
soas_lcps	890	*492*	148	131	105	62
soas_lcpd	891	1 487	154	155	94	127
soas_lp	780	671	109	108	81	84

Tabelle 5.5: Laufzeiten der Summe über Absolutwerte in Bezug auf die 16-Bit-Schleifenvarianten. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *soas* steht für die Methode `sumOfAbsSIMD`. Das Suffix *lc* steht für die zählerbasierten Schleifen und wird mit *ps* und *pd* für das statische und dynamische Array Preloading vervollständigt. Die zeigerbasierte Schleife ist durch *lp* gekennzeichnet. Die Addition bei allen Versionen wurde mittels der *shuffle*-Methode durchgeführt.

Die weitere Messung (Tab. 5.5) dient dem Vergleich der verschiedenen implementierten 16-Bit-Schleifenvarianten. Getestet wurden die zählerbasierte Schleife, deren Versionen mit statischem oder dynamischem Preloading und eine zeigerbasierte Variante. Bei allen Tests wurde zur Addition der Werte die *shuffle*-Methode verwendet. Gemäß den gemessenen Laufzeiten ist auf allen Prozessortypen die zählerbasierte Schleife effizienter. Beim Intel® Atom™ bringt das statische Preloading zusätzlich noch weitere Geschwindigkeitsvorteile.

Abschließend ist für jede Konfiguration aus Architektur und Compiler die Laufzeit der besten Kombination aus Addition und Schleifenvariante im Vergleich zur bisherigen Implementierung gemessen worden. Die genauen Zusammensetzungen und Ergebnisse sind in Tabelle 5.6 dargestellt. Diese zeigt, dass auf keinem der Prozessoren eine Steigerung der Geschwindigkeit im Vergleich zur bestehenden Implementierung erzielt werden konnte.

5.4 Laufzeit der *Sum of Absolute Differences*

Analog zu den ersten beiden Messungen des vorherigen Abschnitts sind auch für die *Sum of Absolute Differences* die effizienteste Addition (Tab. 5.7) und die schnellste 16-Bit-Schleifenvariante (Tab. 5.8) ermittelt worden. Zusätzlich wurde sowohl für die zähler- als auch die zeigerbasierte Schleifenvariante eine *natürliche* Reihenfolge implementiert und getestet, bei der die SAD direkt über die gesamten Spalten berechnet wird. Außerdem steht die zeigerbasierte Version ebenfalls mit statischem Array Preloading zur Verfügung.

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
soas	305	370	48	44	42	41
soas_16bit	599	492	61	*57*	58	55
soas_32bit	*509*	*418*	*57*	*57*	*55*	*52*
Methoden	shuffle, lc	shuffle, lcps	shuffle, lc	shuffle, lc	shuffle, lc	shuffle, lc

Tabelle 5.6: Laufzeiten der Summe über Absolutwerte in Bezug auf die beste Konfiguration. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *soas* steht für die Methode `sumOfAbsSIMD`. Die erste Zeile ohne Suffix ist der Referenzwert, gemessen mit der bisherigen Implementierung. Die weiteren beiden Zeilen geben die Laufzeiten für die 16- und 32-Bit-Varianten der neuen Implementierung an. Für die Messung der neuen Implementierungen wurde jeweils die angegebene Konfiguration aus Additionsmethode und Schleifenvariante verwendet.

Die Geschwindigkeit der Addition variiert stark mit der Kombination aus Prozessortyp und Compilerversion, sodass keine eindeutig beste Methode existiert. Bei den getesteten Schleifen ergeben sich auf den Intel® Core™ i7 Prozessoren die kürzesten Laufzeiten für die zeigerbasierte, akkumulierende Schleifenversion, wohingegen auf dem Intel® Atom™ Prozessor die zählerbasierte Variante (inklusive statischem Array Preloading) schneller ist.

Im Anschluss an die beiden Messungen wurden wieder die effizientesten Konfigurationen mit der bestehenden Implementierung verglichen (Tab. 5.9). Der Tabelle kann entnommen werden, dass mit Version 4.9.0 der GCC mit Ausnahme des Intel® Core™ i7 Prozessors der 3. Generation für alle Prozessoren eine Steigerung der Geschwindigkeit im Vergleich zur bisherigen Umsetzung möglich ist.

5.5 Laufzeit der *Normalised Sum of Absolute Differences*

Die in diesem Abschnitt beschriebenen Zeiten (Tab. 5.10) beziehen sich auf die reine Berechnung des Quotienten der *Normalised Sum of Absolute Differences* und die dafür notwendigen Konvertierungen der Daten. Die Berechnung der einzelnen Komponenten ist in den vorherigen Abschnitten separat beschrieben worden. Beispielsweise wird für die implementierte 16-Bit-Version die Laufzeit

	Atom		**i7_2ndGen**		**i7_3rdGen**	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
nsadt_extract	185 523	114 860	*23 733*	14 447	18 848	12 653
nsadt_cvtsi128	217 884	139 512	24 284	18 241	19 449	16 347
nsadt_index	—	104 528	—	*13 130*	—	*11 587*
nsadt_union	176 229	207 604	26 823	19 045	20 456	17 737
nsadt_2hadd	197 077	144 873	25 292	18 396	23 060	16 739
nsadt_shuffle	*170 861*	*113 590*	24 266	13 440	*17 933*	12 444

Tabelle 5.7: Laufzeiten der *Sum of Absolute Differences* in Bezug auf die horizontale Addition. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *nsadt* steht für die Methode `normSumAbsDiffTerm`. Die Bezeichner stehen in der Reihenfolge der Zeilen für das Extrahieren der Teilsummen mittels `_mm_extract_epi16`, den Zugriff mittels der Intrinsics `_mm_cvtsi128_si32` und `_mm_srli_si128`, den indexbasierten Zugriff, den Zugriff über eine Union, die zweimalige Verwendung des Befehls `_mm_hadd_epi32` und das Umsortieren der Werte vor dem Aufruf des Befehls `_mm_hadd_epi32`. Der indexbasierte Zugriff wird von der GCC in Version 4.6.3 noch nicht unterstützt, wodurch damit für die Methode keine Laufzeiten ermittelt werden konnten. Alle Methoden wurden jeweils in die zählerbasierte 16-Bit-Schleifenvariante inklusive statischem Preloading eingebunden.

	Atom		**i7_2ndGen**		**i7_3rdGen**	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
nsadt_lc	*167 387*	141 251	21 961	15 698	16 134	15 324
nsadt_lca	184 903	122 048	17 596	12 797	16 612	12 554
nsadt_lcaps	170 895	*113 517*	18 944	13 440	18 545	12 422
nsadt_lp	172 533	151 532	18 142	18 567	16 573	17 427
nsadt_lpa	187 007	124 524	*16 753*	*12 505*	*15 677*	*11 931*
nsadt_lpaps	217 796	129 128	24 894	17 032	21 393	14 646

Tabelle 5.8: Laufzeiten der *Sum of Absolute Differences* in Bezug auf die 16-Bit-Schleifenvarianten. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Suffix *lc* steht für die zählerbasierten und das Suffix *lp* für die zeigerbasierten Schleifen, jeweils in den *natürlichen* Umsetzungen. Ein folgendes *a* kennzeichnet die Versionen mit akkumulierender Schleifenumsetzung und *ps* steht für Varianten mit statischem Array Preloading. Die Addition bei allen Versionen wurde mittels der *shuffle*-Methode durchgeführt.

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
nsadt	122 142	116 494	17 814	19 347	15 574	15 622
nsadt_16bit	*167 387*	*113 517*	*18 165*	*16 426*	*15 649*	*15 881*
nsadt_32bit	173 242	141 722	22 609	17 164	16 279	16 175
Methoden	shuffle, lc	shuffle, lcaps	extract*, lpa	index, lpa	shuffle, lpa	index, lpa

Tabelle 5.9: Laufzeiten der Summe über Absolutwerte in Bezug auf die beste Konfiguration. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *nsadt* steht für die Methode `normSumAbsDiffTerm`. Die erste Zeile ohne Suffix ist der Referenzwert, gemessen mit der bisherigen Implementierung. Die weiteren beiden Zeilen geben die Laufzeiten für die 16- und 32-Bit-Varianten der neuen Implementierung an. Für die Messung der neuen Implementierungen wurde jeweils die angegebene Konfiguration aus Additionsmethode und Schleifenvariante verwendet.

*Für die 32-Bit-Version steht unter SSSE3 kein `_mm_extract_epi32`-Befehl zur Verfügung. Dieser ist durch das von Intel (2007) vorgeschlagene Makro (Progr.-ausz. C.18) ersetzt worden.

einer wiederholten Ausführung der Zeilen 77 bis 126 aus Programmauszug C.16 gemessen.

Aus der Tabelle ist ersichtlich, dass mit der 32-Bit-Version vor allem auf dem Intel® Atom™ Prozessor eine schnellere Berechnung möglich ist als mit der bestehenden Implementierung.

5.6 Laufzeit und Qualität des Min-Warping-Algorithmus

Die Zeitmessungen dieses letzten Abschnitts umfassen die gesamte erste Phase des Min-Warping-Algorithmus und vergleichen die bestehende Implementierung mit den jeweils effizientesten neuen Varianten. Dazu wurden für jede Kombination aus Prozessor und Compiler die schnellsten Teilkomponenten ausgewählt und kombiniert.

Wie die Tabelle 5.11 zeigt, hängen die erzielten Verbesserungen dabei sowohl von der Kombination aus Prozessor und Compiler als auch von der implementierten 16- oder 32-Bit-Variante ab.

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
nsad	137 718	138 555	14 174	14 554	13 594	13 463
nsad_16bit	137 628	137 704	*14 195*	14 557	*13 588*	*13 460*
nsad_32bit	*133 351*	*130 007*	14 541	*14 475*	13 892	13 771

Tabelle 5.10: Laufzeiten der *Normalised Sum of Absolute Differences* in Bezug auf die reine Rechnung und notwendige Konvertierungen. Die Berechnung der einzelnen Komponenten der NSAD ist in der Zeitmessung nicht berücksichtigt. Die Zeiten [ms] resultieren aus dem Mittelwert über 100 (i7) bzw. 10 (Atom) Durchläufe. Jeder Durchlauf besteht aus 100 000 Funktionsaufrufen. Das Präfix *nsad* steht für die Methode `calculateNormSumAbsDiffSPS`. Die erste Zeile ohne Suffix ist der Referenzwert, gemessen mit der bisherigen Implementierung. Die weiteren beiden Zeilen geben die Laufzeiten für die 16- und 32-Bit-Varianten der neuen Implementierung an.

	Atom		i7_2ndGen		i7_3rdGen	
	4.6.3	4.9.0	4.6.3	4.9.0	4.6.3	4.9.0
short	*28.859*	28.284	*3.000*	3.269	2.805	*2.803*
colpad_16bit	32.976	*28.088*	3.123	*2.979*	*2.801*	2.813
colpad_32bit	33.305	28.253	3.536	3.269	2.878	2.863

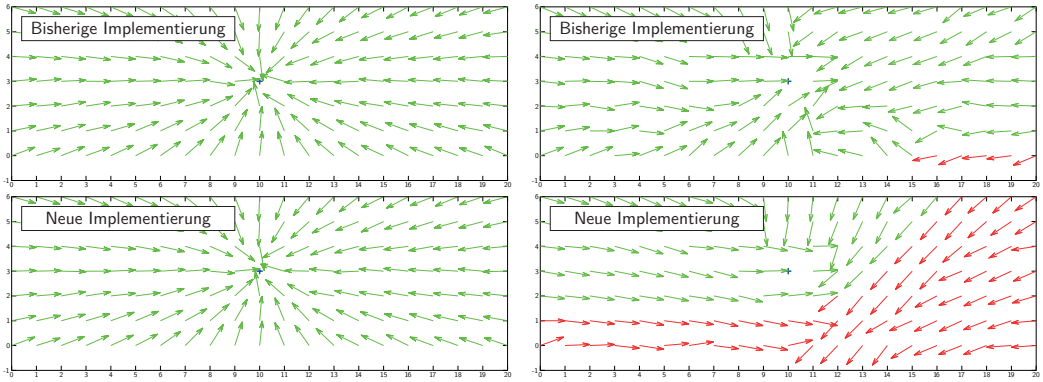
Tabelle 5.11: Laufzeiten der ersten Phase des Min-Warping-Algorithmus. Für die Messungen der neuen Varianten wurden jeweils die schnellsten Teilkomponenten kombiniert. Die Zeiten [ms] resultieren aus dem Mittelwert über 10 Durchläufe der vorliegenden Standalone-Version des Min-Warping-Verfahrens. Dabei wurde der Snapshot $(10, 4)$ mit allen Current Views der living3_day_day_Hh288sh_0.10 Datenbank verglichen. Die bisherige Implementierung ist unter der Bezeichnung *short* eingetragen. Das Präfix *colpad* kennzeichnet die neue Implementierung und das Suffix gibt die jeweilige Version an.

	Gleiche Beleuchtung		Unterschiedliche Beleuchtung	
	$\bar{\beta}$	$\bar{\psi}$	$\bar{\beta}$	$\bar{\psi}$
Bisher	0.047 682 9	0.022 862 6	0.291 678	0.098 922
Neu	0.045 326 7	0.021 667 2	0.569 414	0.391 653

Tabelle 5.12: Vergleich der durchschnittlichen Winkelfehler zwischen der bestehenden und der neuen Implementierung. Die Fehler wurden bei den jeweils idealen Skalierungen von $s_{\text{bisher}} = 25$ und $s_{\text{neu}} = 0.5$ bestimmt.

Zusätzlich zur Zeitmessung gibt das Testprogramm die durchschnittlichen Winkelfehler für die Parameter β und ψ an, die eine Metrik für das Min-Warping-Verfahren sind. Auf den gegebenen Testdatenbanken hängt die Abweichung der Fehler zwischen der alten und neuen Version stark von den Beleuchtungsverhältnissen ab (Tab. 5.12). Für Bilder gleicher Beleuchtung haben diese sich im Vergleich zur bisherigen Implementierung leicht verbessert und fallen rund 5 % geringer aus. Die Fehler bei unterschiedlicher Beleuchtung sind hingegen mit 100 % Abweichung für $\bar{\beta}$ und 300 % Abweichung für $\bar{\psi}$ deutlich größer geworden. Die Fehler wurden bei den jeweils idealen Skalierungen von $s_{\text{bisher}} = 25$ und $s_{\text{neu}} = 0.5$ bestimmt.

Zur Visualisierung des Aktionsraums des Roboters plottet das Testprogramm abschließend die bestimmten Homevektoren (Abb. 5.1). Bei gleicher Beleuchtung ist das Feld der Homevektoren im Vergleich der alten und neuen Version nahezu identisch. In der neuen Version zeigen lediglich die Vektoren im Nahbereich rechts um den Snapshot etwas deutlicher die korrekte Richtung an. Bei unterschiedlicher Beleuchtung verkleinert sich mit der neuen Version der Bereich aus dem zum Snapshot zurückgefunden wird (*Catchment Area*) jedoch erheblich. Fast von keinem der Punkte rechts des Snapshot kann die korrekte Richtung bestimmt werden.



(a) Gleiche Beleuchtung **(b)** Unterschiedliche Beleuchtung

Abbildung 5.1: Vergleichende Darstellung der generierten Homevektoren. Die aus dem bisherigen Verfahren resultierenden Vektoren sind jeweils oben dargestellt und die der in dieser Arbeit angepassten Version unten. Die Position des Snapshots ist mit einem blauen Kreuz markiert. Die grünen Homevektoren bilden die *Catchment Area*, während rote Homevektoren solche Positionen kennzeichnen, von denen der Roboter nicht zum Ziel zurückgefunden hätte. Die Homevektoren in (a) wurden auf Basis der living3_day_day_Hh288sh_0.10 Datenbank, die in (b) auf Basis der living3_day_night_Hh288sh_0.10 Datenbank bestimmt.

6

Diskussion

In diesem Kapitel wird noch einmal näher auf die erzielten Ergebnisse eingegangen und deren Aussage hinsichtlich der Themenstellung dieser Arbeit interpretiert. Der erste Abschnitt betrachtet den Nutzen von kontextunabhängigen Messungen. Danach folgen zwei Abschnitte, in denen die getesteten Konfigurationen miteinander verglichen werden, und im letzten Abschnitt werden die bestehende Implementierung und die beiden neuen Versionen in Bezug auf die Effizienz und die Qualität des Min-Warping-Verfahrens ausgewertet.

6.1 Vergleich zwischen kontextabhängigen und kontextunabhängigen Messungen

Im Vergleich von den kontextunabhängigen Messungen der horizontalen Addition (Tab. 5.3) zu denen aus der Min-Warping-Implementierung (Tab. 5.4, Tab. 5.7), sind keine Überschneidungen der jeweils schnellsten Methoden festzustellen. Werden nur die Additionen wiederholt hintereinander ausgeführt, ist vor allem auf dem Intel® Atom™ Prozessor die Union mit mehr als 20 % deutlich schneller als alle anderen vorgestellten Methoden. Da in diesem Fall der alleinigen Addition die gleichen Instruktionen immer direkt hintereinander wiederholt werden, ist ein Blockieren bestimmter Ausführungseinheiten sehr wahrscheinlich. Somit profitieren die Methoden, deren Befehle sich gut auf die verschiedenen Ports der CPU verteilen lassen und eine geringe Laufzeit aufweisen. Die Union-Methode benötigt nur ein einziges Intrinsic und drei `mov`-Befehle, die gut durch die CPU parallelisiert werden können (Intel, 2014a; Fog,

2014a). Alle anderen Varianten setzen sich aus aufwändigeren Instruktionen zusammen.

Daraus resultiert, dass Zeitmessungen am besten direkt an der Stelle durchgeführt werden, an der die Befehle im späteren Ablauf verwendet werden oder zumindest ein *genügend* großer Kontext in die Messung mit einbezogen wird. Weiterhin wird auch das Problem einer statischen Programmcodeanalyse (z. B. mittels *IACA*; Abschn. 4.2.2) erkennbar: Dort können keine Bedingungen und Schleifen berücksichtigt werden. In der Konsequenz entstehen falsche Abhängigkeiten zwischen vermeintlich aufeinanderfolgenden Befehlen, die das Ergebnis der Analyse negativ beeinflussen.

6.2 Vergleich der Laufzeiten zwischen den beiden Compilerversionen

Relativ unabhängig vom verwendeten Prozessor erzeugt die GCC in der Version 4.9.0 ebenso schnellen oder sogar schnelleren Programmcode als die Version 4.6.3. Die Sequenzen des Transponierens werden mit der neuen Version tendenziell allerdings etwas langsamer, fallen aber gegenüber den anderen Zeiten nicht gravierend ins Gewicht. Deutlich langsamer wird auf dem Intel® Atom™ Prozessor hingegen die Addition unter Verwendung der Union: Im Vergleich zur schnellsten Methode erhöht sich die Laufzeit um 10 % bzw. 20 % (Tab. 5.4, Tab. 5.7). Eine Erklärung für diese Geschwindigkeitseinbußen konnte im Rahmen dieser Arbeit nicht gefunden werden. Eine Analyse des erzeugten Assemblercodes und die jeweiligen Release Notes der GCC könnten Ansätze für weitere Nachforschungen sein. Da die beiden langsamen Programmteile jedoch nicht in der endgültigen Version des Algorithmus verwendet werden, wird an dieser Stelle aus Zeitgründen darauf verzichtet.

Die höhere Performanz bestimmter Methoden ist dabei nicht ausschließlich das Ergebnis einer effizienteren Umsetzung der Befehle durch die neuere Compilerversion im Vergleich zur alten Version. Teilweise ergeben mit der neuen Version auch andere Schleifenreihenfolgen oder eine andere Methode der Addition schnellere Zeiten (Tab. 5.9). Dadurch ist es schwierig, alleine anhand des Programmcodes abzuschätzen, welche Sequenzen effizient vom Compiler umgesetzt werden können, und häufig bleiben nur die Laufzeitmessungen, um eine verlässliche Aussage treffen zu können.

Auch wenn nicht alle Teile des Algorithmus mit Version 4.9.0 der GCC schneller geworden sind, empfiehlt sich dennoch der Wechsel hin zu dieser. Zum einen werden die deutlich langsameren Programmabschnitte, wie zum Beispiel die horizontale Addition mittels einer Union, im finalen Algorithmus

nicht verwendet und haben somit keinen Einfluss auf die Laufzeit. Zum anderen überwiegen insgesamt betrachtet zumindest auf dem Intel® Atom™ Prozessor als Zielplattform und auf dem Intel® Core™ i7 der 2. Generation die Effizienzsteigerungen und gleichen die geringen Verluste aus (Tab. 5.11).

6.3 Vergleich der Laufzeiten zwischen den drei Prozessortypen

Der Intel® Atom™ Prozessor ist über alle Messungen hinweg etwa zehnmal langsamer als die beiden Intel® Core™ i7 Prozessoren. Grund dafür ist zum einen die geringere Taktrate und zum anderen jedoch auch der deutlich kleiner ausfallende Cache. Dadurch müssen häufiger Inhalte umgeladen werden und es treten vermutlich vermehrt Cache Misses auf. Der Intel® Core™ i7 Prozessor der 3. Generation ist zudem unwesentlich schneller als der der 2. Generation. Eine genaue Analyse der Performanzunterschiede kann zu weiteren Ansätzen der Optimierung führen, erfordert aber die Verwendung der in Abschnitt 4.2.2 vorgestellten Programme und würde daher den Umfang dieser Arbeit überschreiten.

Trotz der verschiedenen Spezifikationen der Prozessoren ist häufig eine bestimmte Methode oder eine ganze Kombination aus Methoden auf allen Prozessoren am effizientesten (Tab. 5.6). Der Intel® Atom™ Prozessor weicht stellenweise von dieser Beobachtung ab, was jedoch auf die deutlich größeren Unterschiede zwischen dem Intel® Atom™ und dem Intel ® Core™ i7 Prozessor als zwischen den beiden Prozessoren letzteren Typs zurückzuführen ist.

6.4 Vergleich der Laufzeiten und der Qualität des Min-Warping-Algorithmus

Die Zeitmessung über die gesamte erste Phase des Min-Warping-Algorithmus (Tab. 5.11) hat die vorher theoretisch auf Basis der besten Kombinationen abgeschätzten Laufzeiten bestätigt. Die meisten Kombinationen, bei denen die Effizienzsteigerung der einzelnen Teile in der Summe größer war als die Einbußen, haben auch zu einer Verringerung der Gesamtlaufzeit gegenüber der bisherigen Implementierung um bis zu 3 % geführt. Einzig auf dem Intel® Core™ i7 Prozessor der 3. Generation konnte die Geschwindigkeit nicht signifikant gesteigert werden. Nicht außer Acht gelassen werden darf allerdings die Laufzeit der zweiten Phase (Daten nicht gezeigt): Auf dem Intel® Atom™

Prozessor und dem Intel® Core™ i7 Prozessor der 2. Generation wird diese bei Verwendung der Version 4.9.0 der GCC verglichen mit der Version 4.6.3 um bis zu *50 %* langsamer. Vor einem Wechsel des Compilers muss diese somit auch analysiert und optimiert werden, um eventuelle Flaschenhälse zu beseitigen.

Der neue Befehl `_mm_sad_epu8` hat nur teilweise Effizienzsteigerungen hervorgerufen. In der Berechnung der Nennerterme konnten keine besseren Zeiten erreicht werden (Tab. 5.6). Doch bei der komplexeren Berechnung des Zählers sind mit Version 4.9.0 der GCC auf dem Intel® Atom™ Prozessor und dem Intel® Core™ i7 der 2. Generation erhebliche Verbesserungen erzielt worden (Tab. 5.9). Letztere Rechnung schöpft die Funktion des Befehls auch in vollem Umfang aus, wohingegen er bei der Berechnung des Nenners in der neuen Version nur aufgrund des spaltenweisen Layouts der Bilder im Speicher verwendet werden muss. Die Performanzverluste bei der 32-Bit-Version sind vermutlich durch eine schlechtere Ausnutzung des Caches aufgrund des größeren Wertebereichs zu erklären. Auch hier übersteigt eine genaue Analyse den Rahmen dieser Arbeit.

Dennoch hat insgesamt die Umstellung auf 32-Bit-Zwischenergebnisse auf dem Intel® Atom™ Prozessor um 3–6 % schnellere Laufzeiten für die Berechnung der *Normalised Sum of Absolute Differences* ergeben (Tab. 5.10). Die von Intel empfohlene, approximierte Division mittels der Instruktionen `rcpps` und `mulps` (Intel, 2014a, Abschn. 11.12.1) hat entgegen der Erwartungen – laut Intel (2014a, Tabelle 11.4) verringert sich der Aufwand um das 6-Fache – keine Steigerung der Performanz ergeben (Daten nicht gezeigt). Der Flaschenhals liegt an dieser Stelle vermutlich im *Shufflen* der schreibenden Speicherzugriffe. Eine zu Laufzeitmessungen testweise Umstellungen auf sequentielles Schreiben hat die Berechnung um weitere 25 % (gemessen auf dem Intel® Core™ Prozessor der 3. Generation mit GCC Version 4.9.0) beschleunigt.

Anhand der nahezu unveränderten Winkelfehler bei gleicher Beleuchtung der Bilder und einem kaum abweichenden Homevektorfeld (Abb. 5.1a) lässt sich schließen, dass die zwischenzeitliche Halbierung des Wertebereichs keinen Einfluss auf die Qualität der Ergebnisse des Min-Warping-Verfahrens hat. Die geringen Abweichungen kommen durch skalierungsbedingte Rundungsunterschiede zustande, sind aber vernachlässigbar. Die großen Fehler bei unterschiedlichen Beleuchtungsverhältnissen und die zu etwa einem Drittel fehlerhaften Homevektoren (Abb. 5.1b) machen eine Verwendung der angepassten Version jedoch sehr unwahrscheinlich. Die Halbierung des Wertebereichs führt scheinbar zu großen Einschränkungen bezüglich der Beleuchtungsinvarianz.

7

Fazit und Ausblick

Zusammenfassend lässt sich feststellen, dass das Ziel dieser Arbeit, eine beschleunigte Version der ersten Phase des Min-Warping-Algorithmus zu implementieren, erreicht wurde. Zudem haben sich viele neue Erkenntnisse ergeben, die zur weiteren Optimierung und zum Entwurf neuer Strategien genutzt werden können.

Der neu eingeführte Befehl `_mm_sad_epu8` hat auf zwei der drei Plattformen zu einer Verringerung der Laufzeit geführt, jedoch indirekt auch die Toleranz des Verfahrens gegenüber Beleuchtungsunterschieden negativ beeinflusst. Daher ist abzuwägen, inwiefern er für weitere Optimierungen verwendet werden kann. Eine weiterführende Idee ist, die Halbierung des Wertebereichs durch eine *nichtlineare* Skalierung der Werte vor der Berechnung der *Normalised Sum of Absolute Differences* auszugleichen. Ziel wäre es, den Abstand zwischen kleinen Werten zu erhöhen, um im Bereich des Minimums bessere Kontraste zu bieten. Da das Verfahren nur nach dem Minimum sucht, sollte ein Zusammenschieben der hohen Werte keinen nachteiligen Einfluss haben.

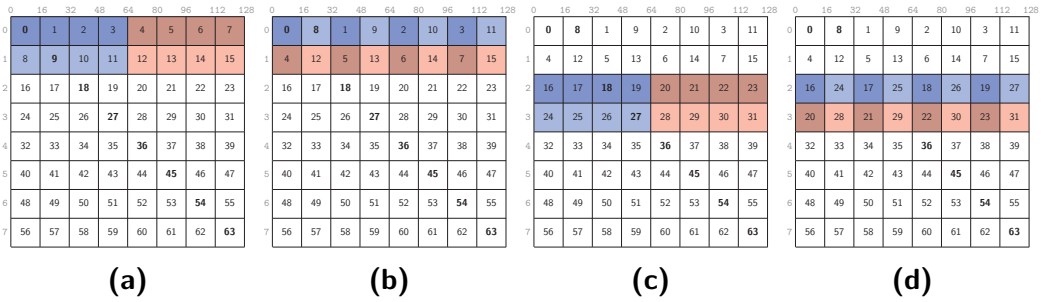
Des Weiteren sollten alle kritischen Speicheroperationen mittels geeigneter Analyseprogramme hinsichtlich ihrer Effizienz ausgewertet werden. Durch den gezielten Einsatz von *Cache Blocking* Techniken können dann weitere Flaschenhälse eliminiert werden. Besonders im Fall des Schreibens der Scale Planes ist über eine komplette Umstrukturierung nachzudenken, um ein lineares Schreiben zu ermöglichen und das *Shufflen* zu vermeiden. Anschließend sollte die Berechnung erneut auf einen Performanzgewinn durch eine approximierte Berechnung des Quotienten überprüft werden.

Vor dem Umstieg auf eine aktuellere Compilerversion muss schließlich auch die zweite Phase eingehend analysiert werden, um die Ursache für den

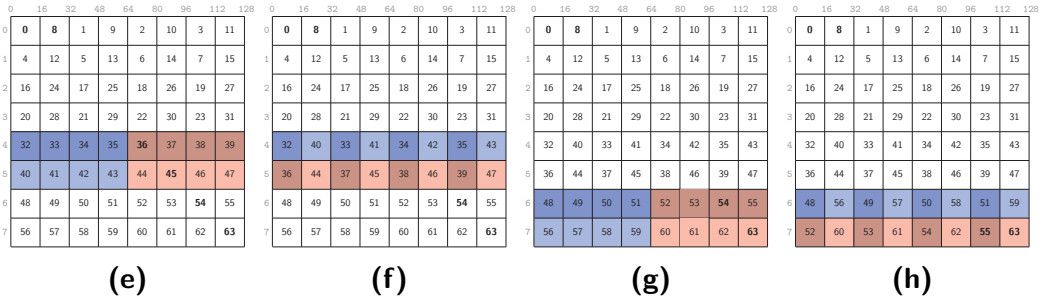
Geschwindkeitsverlust ausfindig zu machen. Generell sollten auch dort bessere Laufzeiten durch den Einsatz eines neueren Compilers zu erwarten sein. Im Rahmen dieser Arbeit wurde durch den Vergleich von zwei Compilerversionen auf drei verschiedenen Prozessortypen ebenfalls erkenntlich, dass Optimierungen teilweise nur für eine bestimmte Konfiguration Vorteile bringen und auf anderen eventuell sogar nachteilig sind. Die Vergleiche könnten auf die *Intel® C and C++ Compiler* ausgeweitet werden. Durch die Analyse des erzeugten Programmcodes lässt sich gegebenenfalls auf weitere Verbesserungen schließen.

A

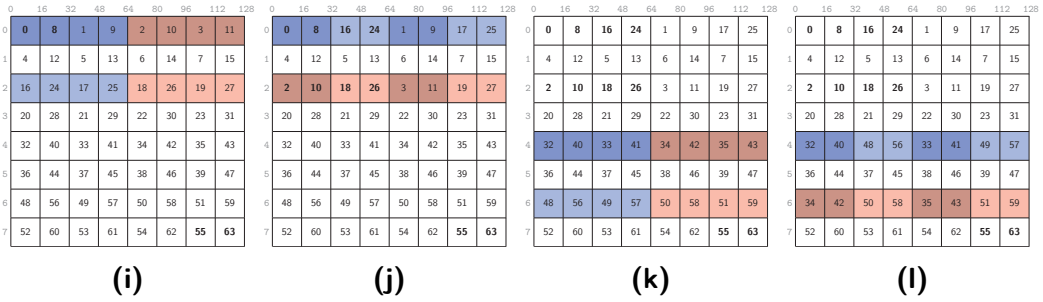
Ergänzende Abbildungen



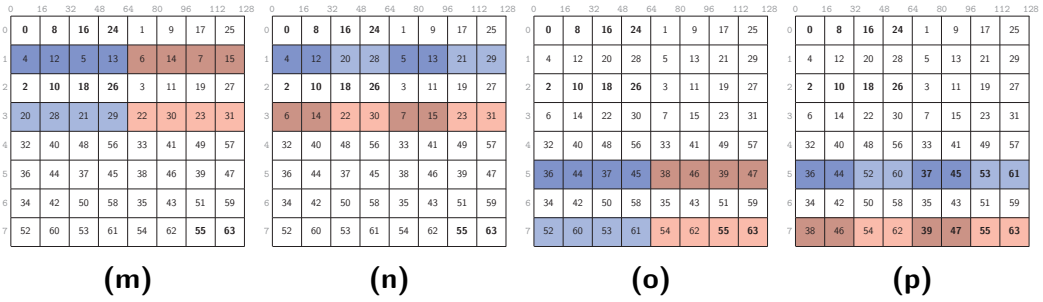
(a) **(b)** **(c)** **(d)**



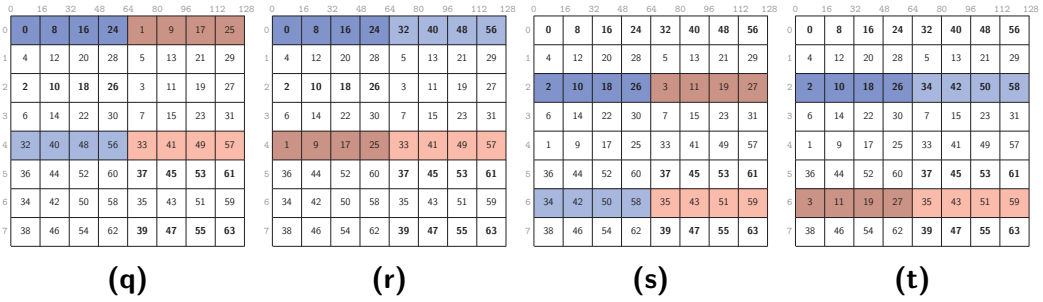
(e) **(f)** **(g)** **(h)**



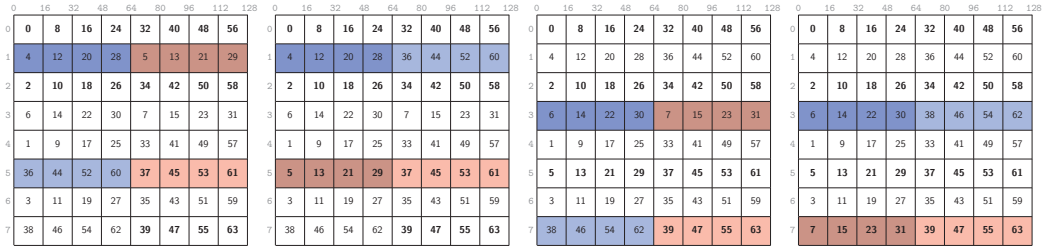
(i) **(j)** **(k)** **(l)**



(m) **(n)** **(o)** **(p)**

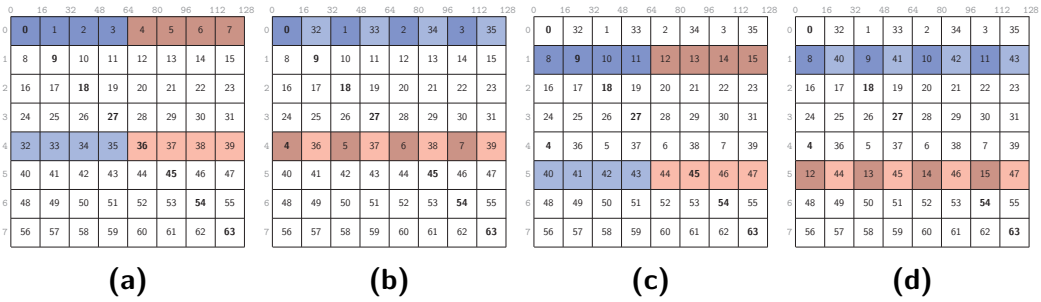


(q) **(r)** **(s)** **(t)**

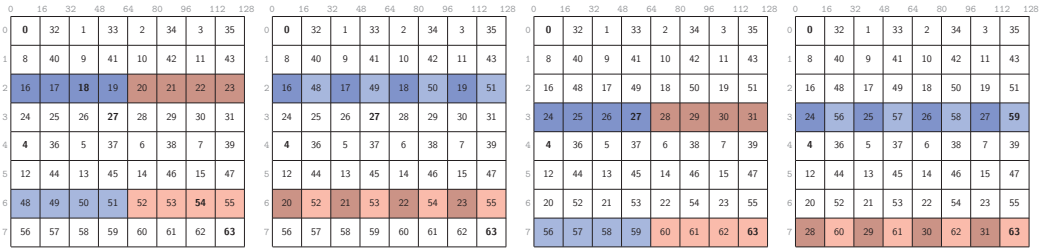


(u) **(v)** **(w)** **(x)**

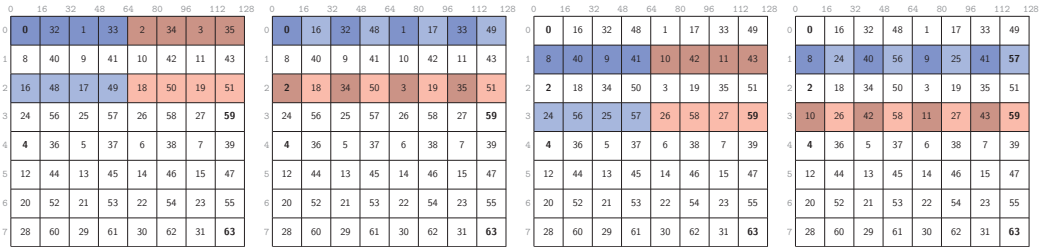
Abbildung A.1: Blockweises Transponieren mittels *Bottom-Up*-Ansatz am Beispiel einer 8×8 Matrix. Die erste und dritte Spalte zeigen die Matrizen vor der Anwendung der jeweiligen `_mm_unpacklo_epixx`- (blau) und `_mm_unpackhi_epixx`-Befehle (rot), die zweite und vierte Spalte zeigen das Ergebnis des vorherigen Aufrufs. Auf die Matrizen (a) bis (h) werden die Befehle mit Suffix `_epi16`, auf (i) bis (p) die Befehle mit Suffix `_epi32` und auf (q) bis (x) die Befehle mit Suffix `_epi64` angewendet. An der letzten Matrix ist außerdem zu erkennen, dass die Zeilen erst beim Zurückschreiben in den Speicher in die richtige Reihenfolge gebracht werden. Fettgedruckte Zahlen geben an, dass das jeweilige Element die Zielposition in der transponierten Matrix erreicht hat.



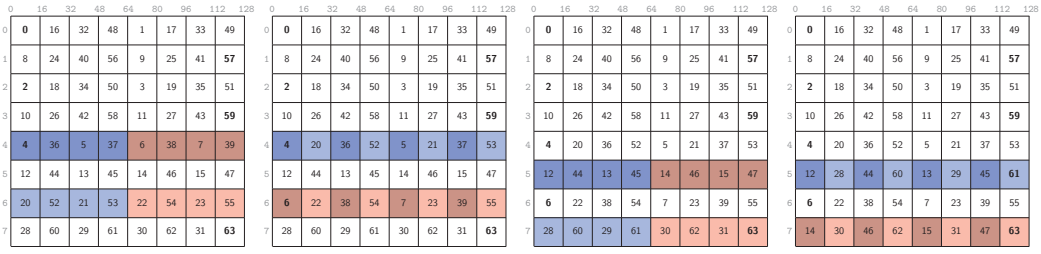
(a) **(b)** **(c)** **(d)**



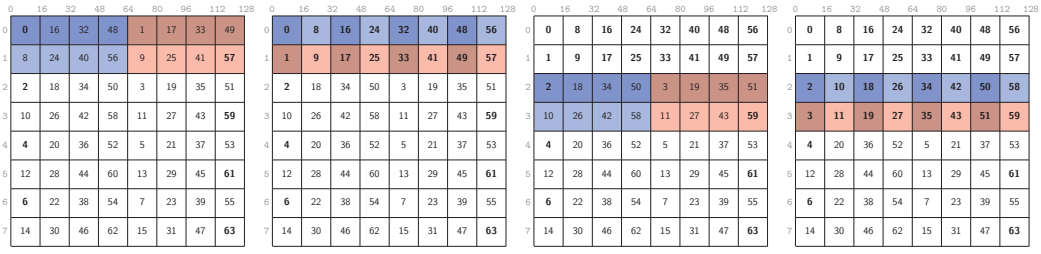
(e) **(f)** **(g)** **(h)**



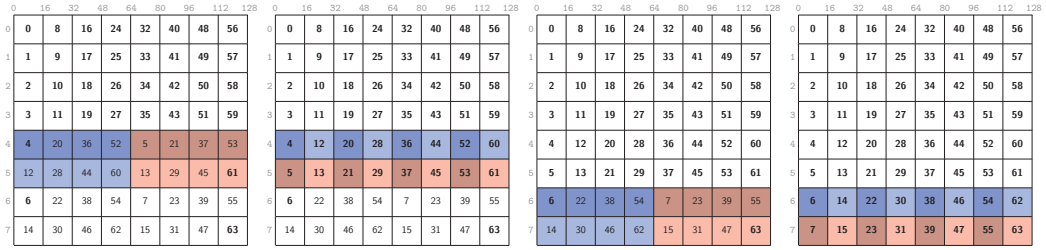
(i) **(j)** **(k)** **(l)**



(m) **(n)** **(o)** **(p)**



(q) **(r)** **(s)** **(t)**



(u) **(v)** **(w)** **(x)**

Abbildung A.2: Blockweises Transponieren mittels *Top-Down*-Ansatz am Beispiel einer 8×8 Matrix. Die erste und dritte Spalte zeigen die Matrizen vor der Anwendung der jeweiligen `_mm_unpacklo_epi8`- (blau) und `_mm_unpackhi_epi8`-Befehle (rot), die zweite und vierte Spalte zeigen das Ergebnis des vorherigen Aufrufs. Fettgedruckte Zahlen geben an, dass das jeweilige Element die Zielposition in der transponierten Matrix erreicht hat.

B

Ergänzende Tabellen

ab Version	hinzugefügte Erweiterungen
3.1	• MMX-, SSE-, SSE2-Unterstützung • MMX-, SSE-Intrinsics
3.3	• SSE2-Intrinsics • Rechenoperationen zwischen Vektordatentypen: `+`, `-`, `*`, `/`, `-` (Vorzeichen)
3.4	• Logikoperationen zwischen Vektordatentypen: `^`, `|`, `&`, `~`
4.0	• Automatische Vektorisierung sequentiellen Codes • Attribut `vector_size` statt `mode` um Vektordatentypen zu definieren
4.3	• SSE3, SSE4.1, SSE4.2
4.4	• AVX
4.5	• Allgemeine Headerdatei `x86intrin.h`
4.6	• Modulooperation (%) für Vektordatentypen • Shiftoperationen (<<, >>) auf ganzzahligen Vektordatentypen (nicht C++). Zweiter Operand kann ein Skalar sein, der dann auf die entsprechende Vektorgröße vervielfältigt wird • Indexbasierter Zugriff auf einzelne Vektorkomponenten (nicht C++).
4.7	• AVX2 • Bei allen binären Operationen darf ein Operand ein Skalar sein • Komponentenweise Vergleichsoperationen zwischen Vektordatentypen (nur GNU C): ==, !=, <, <=, >, >= • Permutation von Vektorkomponenten durch interne Funktionen
4.8	• Alle eingeführten Erweiterungen nun ohne Einschränkungen nutzbar
4.9	• AVX-512 • Ternärer Operator (?) für Vektordatentypen (nur C++)

Tabelle B.1: Chronologische Übersicht der GCC Vector Extensions. Genannt wird nur die erste Version jeder neuen Serie. Änderungen wurden möglicherweise erst in einem der kleineren Updates eingeführt.

C

Ergänzende Programmauszüge

```
#define _MM_8SHORT_TO_2X4FLOAT(SHORT8,FLOAT4L,FLOAT4H) \
    _MM_8SHORT_VIA_INT_TO_2X4FLOAT(SHORT8,FLOAT4L,FLOAT4H)

#define _MM_8SHORT_VIA_INT_TO_2X4FLOAT(SHORT8,FLOAT4L,FLOAT4H) \
    FLOAT4L = _mm_cvtepi32_ps(_MM_4SHORTL_TO_4INT(SHORT8)); \
    FLOAT4H = _mm_cvtepi32_ps(_MM_4SHORTH_TO_4INT(SHORT8));

#define _MM_4SHORTL_TO_4INT(SHORT8) \
    _mm_srai_epi32(_mm_unpacklo_epi16(SHORT8,SHORT8), 16)

#define _MM_4SHORTH_TO_4INT(SHORT8) \
    _mm_srai_epi32(_mm_unpackhi_epi16(SHORT8,SHORT8), 16)
```

Programmauszug C.1: Makro zur Konvertierung von Short zu Float

```
#define _MM_4X4FLOAT_TO_16UBYTE(FLOAT4LL,FLOAT4LH,FLOAT4HL,FLOAT4HH) \
    _mm_packus_epi16(_MM_2X4FLOAT_TO_8SHORT(FLOAT4LL,FLOAT4LH), \
                     _MM_2X4FLOAT_TO_8SHORT(FLOAT4HL,FLOAT4HH))

#define _MM_2X4FLOAT_TO_8SHORT(FLOAT4L,FLOAT4H) \
    _mm_packs_epi32(_mm_cvtps_epi32(FLOAT4L), \
                    _mm_cvtps_epi32(FLOAT4H))
```

Programmauszug C.2: Makro zur Konvertierung von Float zu Unsigned Byte

```
#define _MM_NDIST_PS(NUM,DENOM,FAC) \
    _mm_mul_ps(_mm_div_ps(NUM, DENOM), FAC)
```

Programmauszug C.3: Makro zur Berechnung und Skalierung der NSAD

```
#define _MM_4X4FLOAT_TO_16BYTE(FLOAT4AL,FLOAT4AH,FLOAT4BL,FLOAT4BH, \
                               SHORT_VEC_128) \
    _mm_packus_epi16(_mm_adds_epi16( \
                         _MM_2X4FLOAT_TO_8SHORT(FLOAT4AL,FLOAT4AH), \
                         SHORT_VEC_128), \
                     _mm_adds_epi16( \
                         _MM_2X4FLOAT_TO_8SHORT(FLOAT4BL,FLOAT4BH), \
                         SHORT_VEC_128))
```

Programmauszug C.4: Makro zur Umwandlung von Gleitkommazahlen in den vorzeichenbehafteten 8-Bit-Wertebereich inklusive Biasverschiebung. Jeweils vier Register mit 32-Bit-Gleitkommazahlen (`FLOAT4xx`) werden zunächst auf zwei Register mit vorzeichenbehafteten 16-Bit-Ganzzahlen reduziert, zu denen der Bias `SHORT_VEC_128` addiert werden kann. Der Befehl `_mm_packus_epi16` führt dann eine weitere saturierte Konvertierung auf vorzeichenlose 8-Bit-Ganzzahlen durch, sodass alle 16 Werte in ein Register passen.

```
for(p1 = img + y1 * w16Input, p2 = img + y2 * w16Input,                    1
        ps = scaledImage + ys;                                             2
    p1 < img + y1 * w16Input + w16Input;                                   3
    p1 += 16, p2 += 16, ps += 16 * w16Output)                              4
{                                                                          5
    __m128 v1al = _mm_load_ps(p1);                                         6
    __m128 v1ah = _mm_load_ps(p1 + 4);                                     7
    __m128 v1bl = _mm_load_ps(p1 + 8);                                     8
    __m128 v1bh = _mm_load_ps(p1 + 12);                                    9
    __m128 v2al = _mm_load_ps(p2);                                         10
    __m128 v2ah = _mm_load_ps(p2 + 4);                                     11
    __m128 v2bl = _mm_load_ps(p2 + 8);                                     12
    __m128 v2bh = _mm_load_ps(p2 + 12);                                    13
                                                                           14
    __m128 ral = _mm_add_ps(v2al, _mm_mul_ps(w4, _mm_sub_ps(v1al, v2al))); 15
    __m128 rah = _mm_add_ps(v2ah, _mm_mul_ps(w4, _mm_sub_ps(v1ah, v2ah))); 16
    __m128 rbl = _mm_add_ps(v2bl, _mm_mul_ps(w4, _mm_sub_ps(v1bl, v2bl))); 17
    __m128 rbh = _mm_add_ps(v2bh, _mm_mul_ps(w4, _mm_sub_ps(v1bh, v2bh))); 18
                                                                           19
    byte16.r = _MM_4X4FLOAT_TO_16BYTE(_mm_mul_ps(ral, scale4),             20
                                      _mm_mul_ps(rah, scale4),             21
                                      _mm_mul_ps(rbl, scale4),             22
                                      _mm_mul_ps(rbh, scale4),             23
                                      shortVec128);                        24
                                                                           25
    for(int i = 0; i < 16; ++i)                                            26
    {                                                                      27
        ps[i * w16Output] = byte16.b[i];                                   28
    }                                                                      29
}                                                                          30
```

Programmauszug C.5: Angepasster Programmcode der linearen Interpolationsmethode. Die Blockbreite wurde von 8 auf 16 Elemente erhöht, wodurch immer 16 Werte im Voraus interpoliert werden müssen, um nach der Konvertierung die volle Registerbreite auszunutzen. In den Zeilen 6 bis 13 werden die beiden Zeilen, zwischen denen interpoliert werden soll, in Register geladen. Die eigentliche Interpolation findet in den Zeilen 15 bis 18 statt. Danach werden die interpolierten Werte skaliert, konvertiert und spaltenweise in den Speicher geschrieben, indem mittels einer Union auf die einzelnen 8-Bit-Elemente des `xmm`-Registers zugegriffen wird.

```
for(y = 0; y < hInput - 15; y += 16)                                       1
{                                                                          2
    dst = scaledImage + y;                                                 3
    src = img + w16Input * y;                                              4
                                                                           5
    for(const SSEFloat *srcEnd = src + w16Input; src < srcEnd; src += 16)  6
    {                                                                      7
        for(r = row, rowSrc = src; r < row + 16; ++r, rowSrc += w16Input)  8
        {                                                                  9
            *r = _MM_4X4FLOAT_TO_16BYTE(                                   10
                    _mm_mul_ps(*((__m128*)rowSrc), scale4),                11
                    _mm_mul_ps(*((__m128*)(rowSrc + 4)), scale4),          12
                    _mm_mul_ps(*((__m128*)(rowSrc + 8)), scale4),          13
                    _mm_mul_ps(*((__m128*)(rowSrc + 12)), scale4),         14
                    shortVec128);                                          15
```

```
        }                                                                    16
                                                                             17
        _MM_TRANSPOSE16_EPI8(row, dst, w16Output);                           18
                                                                             19
        dst += 16 * w16Output;                                               20
    }                                                                        21
}                                                                            22
                                                                             23
if(hInput != w16Output)                                                      24
{                                                                            25
    dst = scaledImage + y;                                                   26
    src = img + w16Input * y;                                                27
                                                                             28
    for(const SSEFloat *srcEnd = src + w16Input; src < srcEnd; src += 16)    29
    {                                                                        30
        for(r = row, rowSrc = src;                                           31
            r < row + 16 - (w16Output - hInput);                             32
            ++r, rowSrc += w16Input)                                         33
        {                                                                    34
            *r = _MM_4X4FLOAT_TO_16BYTE(                                     35
                    _mm_mul_ps(*((__m128*)rowSrc), scale4),                  36
                    _mm_mul_ps(*((__m128*)(rowSrc + 4)), scale4),            37
                    _mm_mul_ps(*((__m128*)(rowSrc + 8)), scale4),            38
                    _mm_mul_ps(*((__m128*)(rowSrc + 12)), scale4),           39
                    shortVec128);                                            40
        }                                                                    41
                                                                             42
        for(; r < row + 16; ++r)                                             43
        {                                                                    44
            *r = byteVec128;                                                 45
        }                                                                    46
                                                                             47
        _MM_TRANSPOSE16_EPI8(row, dst, w16Output);                           48
                                                                             49
        dst += 16 * w16Output;                                               50
    }                                                                        51
}                                                                            52
```

Programmauszug C.6: Schleifenablauf beim blockweisen Transponieren mit separater Behandlung übriger Zeilen. In den Zeilen 1 bis 22 werden solange möglich immer 16 Zeilen des Quellbildes im Voraus in Register geladen und anschließend transponiert. Ist die Höhe des Bildes kein Vielfaches von 16, werden als erstes die restlichen Zeilen separat durch die Schleife in Zeile 31 in die Register kopiert. Danach werden in den Zeilen 43 bis 46 in die noch nicht gefüllten Register des Arrays konstante Werte (abzüglich des Bias handelt es sich um Nullen) geladen. Dadurch ist auch in diesem Fall ein blockweises Transponieren möglich.

```
for(int y = 0; y < w16Output; y += 16)
{
    dst = scaledImage + y;
    src = img + w16Input * y;

    for(const SSEFloat *srcEnd = src + w16Input; src < srcEnd; src += 16)
    {
        for(r = row, rowSrc = src; r < row + 16; ++r, rowSrc += w16Input)
        {
            *r = _MM_4X4FLOAT_TO_16BYTE(
                    _mm_mul_ps(*((__m128*)rowSrc), scale4),
                    _mm_mul_ps(*((__m128*)(rowSrc + 4)), scale4),
                    _mm_mul_ps(*((__m128*)(rowSrc + 8)), scale4),
                    _mm_mul_ps(*((__m128*)(rowSrc + 12)), scale4),
                    shortVec128);
        }

        _MM_TRANSPOSE16_EPI8(row, dst, w16Output);

        dst += 16 * w16Output;
    }
}
```

Programmauszug C.7: Schleifenablauf (XY Schreibrichtung) beim blockweisen Transponieren mit vorherigem *Padding* des Speichers

```
for(src = img; src < img + w16Input; src += 16, dst += 15 * w16Output)
{
    rowSrc = src;

    for(const SSEByte *dstEnd = dst + w16Output; dst < dstEnd; dst += 16)
    {
        for(r = row; r < row + 16; ++r, rowSrc += w16Input)
        {
            *r = _MM_4X4FLOAT_TO_16BYTE(
                    _mm_mul_ps(*((__m128*)rowSrc), scale4),
                    _mm_mul_ps(*((__m128*)(rowSrc + 4)), scale4),
                    _mm_mul_ps(*((__m128*)(rowSrc + 8)), scale4),
                    _mm_mul_ps(*((__m128*)(rowSrc + 12)), scale4),
                    shortVec128);
        }

        _MM_TRANSPOSE16_EPI8(row, dst, w16Output);
    }
}
```

Programmauszug C.8: Schleifenablauf (YX Schreibrichtung) beim blockweisen Transponieren mit vorherigem *Padding* des Speichers

```
#define _MM_TRANSPOSE4_PS(row0, row1, row2, row3) \
    do { \
        __m128 tmp3, tmp2, tmp1, tmp0; \
        tmp0 = _mm_unpacklo_ps((row0), (row1)); \
        tmp2 = _mm_unpacklo_ps((row2), (row3)); \
        tmp1 = _mm_unpackhi_ps((row0), (row1)); \
        tmp3 = _mm_unpackhi_ps((row2), (row3)); \
        (row0) = _mm_movelh_ps(tmp0, tmp2); \
        (row1) = _mm_movehl_ps(tmp2, tmp0); \
        (row2) = _mm_movelh_ps(tmp1, tmp3); \
        (row3) = _mm_movehl_ps(tmp3, tmp1); \
    } while (0)
```

Programmauszug C.9: Makro zum blockweisen Transponieren einer vierelementigen 32-Bit-Matrix (LLVM, 2014). Die ersten vier Befehle bewirken, dass jeweils Zweiergruppen in der transponierten Reihenfolge nebeneinander stehen. Die letzten vier Befehle sortieren diese dann zu Vierergruppen um, sodass am Ende die komplette Matrix transponiert ist. Das Konstrukt `do {...} while (0)` dient lediglich der Erstellung eines lokalen Sichtbarkeitsbereichs für die temporären Variablen und hat ansonsten keinen Einfluss auf die Berechnung.

```
#define _MM_TRANSPOSE16_EPI8_MERGE_EPI8(IN0, IN1, TMP) \
{ \
    (TMP) = (IN0); \
    (IN0) = _mm_unpacklo_epi8((IN0), (IN1)); \
    (IN1) = _mm_unpackhi_epi8((TMP), (IN1)); \
}

#define _MM_TRANSPOSE16_EPI8_MERGE_EPI16(IN0, IN1, TMP) \
{ \
    (TMP) = (IN0); \
    (IN0) = _mm_unpacklo_epi16((IN0), (IN1)); \
    (IN1) = _mm_unpackhi_epi16((TMP), (IN1)); \
}

#define _MM_TRANSPOSE16_EPI8_MERGE_EPI32(IN0, IN1, TMP) \
{ \
    (TMP) = (IN0); \
    (IN0) = _mm_unpacklo_epi32((IN0), (IN1)); \
    (IN1) = _mm_unpackhi_epi32((TMP), (IN1)); \
}

#define _MM_TRANSPOSE16_EPI8_MERGE_EPI64(IN0, IN1, TMP) \
{ \
    (TMP) = (IN0); \
    (IN0) = _mm_unpacklo_epi64((IN0), (IN1)); \
    (IN1) = _mm_unpackhi_epi64((TMP), (IN1)); \
}

#define _MM_TRANSPOSE16_EPI8_2(ROWS, DST, DST_WIDTH) \
{ \
    __m128i tmp; \
\
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[0], (ROWS)[1], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[2], (ROWS)[3], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[4], (ROWS)[5], tmp); \
```

```
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[6], (ROWS)[7], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[8], (ROWS)[9], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[10], (ROWS)[11], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[12], (ROWS)[13], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[14], (ROWS)[15], tmp); \
\
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[0], (ROWS)[2], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[4], (ROWS)[6], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[8], (ROWS)[10], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[12], (ROWS)[14], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[1], (ROWS)[3], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[5], (ROWS)[7], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[9], (ROWS)[11], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI16((ROWS)[13], (ROWS)[15], tmp); \
\
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[0], (ROWS)[4], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[2], (ROWS)[6], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[8], (ROWS)[12], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[10], (ROWS)[14], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[1], (ROWS)[5], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[3], (ROWS)[7], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[9], (ROWS)[13], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI32((ROWS)[11], (ROWS)[15], tmp); \
\
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[0], (ROWS)[8], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[2], (ROWS)[10], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[4], (ROWS)[12], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[6], (ROWS)[14], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[1], (ROWS)[9], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[3], (ROWS)[11], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[5], (ROWS)[13], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI64((ROWS)[7], (ROWS)[15], tmp); \
\
    _mm_store_si128((__m128i*)(DST), (ROWS)[0]); \
    _mm_store_si128((__m128i*)((DST) + 1 * (DST_WIDTH)), (ROWS)[8]); \
    _mm_store_si128((__m128i*)((DST) + 2 * (DST_WIDTH)), (ROWS)[4]); \
    _mm_store_si128((__m128i*)((DST) + 3 * (DST_WIDTH)), (ROWS)[12]); \
    _mm_store_si128((__m128i*)((DST) + 4 * (DST_WIDTH)), (ROWS)[2]); \
    _mm_store_si128((__m128i*)((DST) + 5 * (DST_WIDTH)), (ROWS)[10]); \
    _mm_store_si128((__m128i*)((DST) + 6 * (DST_WIDTH)), (ROWS)[6]); \
    _mm_store_si128((__m128i*)((DST) + 7 * (DST_WIDTH)), (ROWS)[14]); \
    _mm_store_si128((__m128i*)((DST) + 8 * (DST_WIDTH)), (ROWS)[1]); \
    _mm_store_si128((__m128i*)((DST) + 9 * (DST_WIDTH)), (ROWS)[9]); \
    _mm_store_si128((__m128i*)((DST) + 10 * (DST_WIDTH)), (ROWS)[5]); \
    _mm_store_si128((__m128i*)((DST) + 11 * (DST_WIDTH)), (ROWS)[13]); \
    _mm_store_si128((__m128i*)((DST) + 12 * (DST_WIDTH)), (ROWS)[3]); \
    _mm_store_si128((__m128i*)((DST) + 13 * (DST_WIDTH)), (ROWS)[11]); \
    _mm_store_si128((__m128i*)((DST) + 14 * (DST_WIDTH)), (ROWS)[7]); \
    _mm_store_si128((__m128i*)((DST) + 15 * (DST_WIDTH)), (ROWS)[15]); \
}
```

Programmauszug C.10: Makro zum blockweisen Transponieren einer 16×16 Matrix mit 8-Bit-Elementen mittels *Bottom-Up*-Ansatz. Als Eingabe für `ROWS` wird ein Array aus 16 Elementen des Typs `__m128i` erwartet. Die Argumente `DST` und `DST_WIDTH` müssen auf den Zielspeicherbereich für die transponierte Untermatrix verweisen und die Dimension in x-Richtung der transponierten Matrix beinhalten.

```
#define _MM_TRANSPOSE16_EPI8_MERGE_EPI8(IN0, IN1, TMP) \
{ \
    (TMP) = (IN0); \
    (IN0) = _mm_unpacklo_epi8((IN0), (IN1)); \
    (IN1) = _mm_unpackhi_epi8((TMP), (IN1)); \
}

#define _MM_TRANSPOSE16_EPI8(ROWS, DST, DST_WIDTH) \
{ \
    __m128i tmp; \
    \
    /* even */ \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[0], (ROWS)[8], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[4], (ROWS)[12], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[0], (ROWS)[4], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[8], (ROWS)[12], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[2], (ROWS)[10], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[6], (ROWS)[14], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[2], (ROWS)[6], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[10], (ROWS)[14], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[0], (ROWS)[2], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[4], (ROWS)[6], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[8], (ROWS)[10], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[12], (ROWS)[14], tmp); \
    \
    /* odd */ \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[1], (ROWS)[9], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[5], (ROWS)[13], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[1], (ROWS)[5], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[9], (ROWS)[13], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[3], (ROWS)[11], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[7], (ROWS)[15], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[3], (ROWS)[7], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[11], (ROWS)[15], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[1], (ROWS)[3], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[5], (ROWS)[7], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[9], (ROWS)[11], tmp); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[13], (ROWS)[15], tmp); \
    \
    /* write */ \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[0], (ROWS)[1], tmp); \
    _mm_store_si128((__m128i*)(DST), (ROWS)[0]); \
    _mm_store_si128((__m128i*)((DST) + (DST_WIDTH)), (ROWS)[1]); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[2], (ROWS)[3], tmp); \
    _mm_store_si128((__m128i*)((DST) + 2 * (DST_WIDTH)), (ROWS)[2]); \
    _mm_store_si128((__m128i*)((DST) + 3 * (DST_WIDTH)), (ROWS)[3]); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[4], (ROWS)[5], tmp); \
    _mm_store_si128((__m128i*)((DST) + 4 * (DST_WIDTH)), (ROWS)[4]); \
    _mm_store_si128((__m128i*)((DST) + 5 * (DST_WIDTH)), (ROWS)[5]); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[6], (ROWS)[7], tmp); \
    _mm_store_si128((__m128i*)((DST) + 6 * (DST_WIDTH)), (ROWS)[6]); \
    _mm_store_si128((__m128i*)((DST) + 7 * (DST_WIDTH)), (ROWS)[7]); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[8], (ROWS)[9], tmp); \
    _mm_store_si128((__m128i*)((DST) + 8 * (DST_WIDTH)), (ROWS)[8]); \
    _mm_store_si128((__m128i*)((DST) + 9 * (DST_WIDTH)), (ROWS)[9]); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[10], (ROWS)[11], tmp); \
    _mm_store_si128((__m128i*)((DST) + 10 * (DST_WIDTH)), (ROWS)[10]); \
    _mm_store_si128((__m128i*)((DST) + 11 * (DST_WIDTH)), (ROWS)[11]); \
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[12], (ROWS)[13], tmp); \
    _mm_store_si128((__m128i*)((DST) + 12 * (DST_WIDTH)), (ROWS)[12]); \
    _mm_store_si128((__m128i*)((DST) + 13 * (DST_WIDTH)), (ROWS)[13]); \
```

```
    _MM_TRANSPOSE16_EPI8_MERGE_EPI8((ROWS)[14], (ROWS)[15], tmp); \
    _mm_store_si128((__m128i*)((DST) + 14 * (DST_WIDTH)), (ROWS)[14]); \
    _mm_store_si128((__m128i*)((DST) + 15 * (DST_WIDTH)), (ROWS)[15]); \
}
```

Programmauszug C.11: Makro zum blockweisen Transponieren einer 16×16 Matrix mit 8-Bit-Elementen mittels *Top-Down*-Ansatz. Als Eingabe für `ROWS` wird ein Array aus 16 Elementen des Typs `__m128i` erwartet. Die Argumente `DST` und `DST_WIDTH` müssen auf den Zielspeicherbereich für die transponierte Untermatrix verweisen und die Dimension in x-Richtung der transponierten Matrix beinhalten.

```
function transpose_top_down_iterative(ROWS, STEPSIZE)
    SECTIONS = 1

    BIT = 64 / STEPSIZE

    for (STEPSIZE >= 1)
        STEPSIZE = floor(STEPSIZE / 2)

        for i = 0 to (SECTIONS - 1) step 1
            SECTION = (2 * i * STEPSIZE)

            for j = 0 to (STEPSIZE - 1) step 1
                TMP = ROWS[SECTION + j]
                ROWS[SECTION + j] =
                    _mm_unpacklo_epi@{BIT}(TMP,
                                           ROWS[SECTION + j + STEPSIZE])
                ROWS[SECTION + j + STEPSIZE] =
                    _mm_unpackhi_epi@{BIT}(TMP,
                                           ROWS[SECTION + j + STEPSIZE])
            endfor
        endfor

        SECTIONS = 2 * SECTIONS
    endfor

    return ROWS
endfunction
```

Programmauszug C.12: Ablauf des blockweisen Transponierens mittels iterativem *Top-Down*-Ansatz. Die Matrix wird zeilenweise in dem Array `ROWS` übergeben. Die Variable `STEPSIZE` gibt die Schrittweite des jeweiligen Schleifendurchlaufs an. Diese wird zu Beginn jedes Durchlaufs der äußersten Schleife halbiert und dient als Abbruchkriterium, sobald sie unter 1 gelangt. Initial muss für eine $2^n \times 2^n$ Matrix eine Schrittweite von 2^n übergeben werden. Über `BIT` wird festgelegt, für welchen Wertebereich die `_mm_unpackxx_epixx`-Befehle zu verwenden sind. In den beiden Schleifen werden in allen Bereichen die Zeilen der entsprechenden Schrittweite paarweise kombiniert. Es ist ein Register als temporärer Zwischenspeicher notwendig, da ansonsten benötigte Werte überschrieben würden. Am Ende jedes Durchlaufs der äußersten Schleife wird die Schrittweite verdoppelt.

```
__m128i preload[8];                                                              1
int max = (w16 > 128 ? 128 : w16) >> 4;                                          2
const SSEByte *end = img + w16;                                                  3
                                                                                 4
for(int y = 0; y < h16; ++y, ++dst, end += w16)                                  5
{                                                                                6
    as = _mm_setzero_si128();                                                    7
                                                                                 8
    while(src < end)                                                             9
    {                                                                            10
        for(int i = 0; i < max; ++i, src += 16)                                  11
        {                                                                        12
            preload[i] = *((__m128i*)src);                                       13
        }                                                                        14
                                                                                 15
        for(int i = 0; i < max; ++i)                                             16
        {                                                                        17
            as = _mm_adds_epi16(as, _mm_sad_epu8(preload[i], byteVec128));       18
        }                                                                        19
    }                                                                            20
                                                                                 21
    *dst = (SSEShort)_mm_cvtsi128_si32(as) +                                     22
           (SSEShort)_mm_cvtsi128_si32(_mm_srli_si128(as, 8));                   23
}                                                                                24
```

Programmauszug C.13: Direkte Umsetzung der Summe über Absolutwerte mittels des Befehls `_mm_sad_epu8` inklusive *Array Preloading* Optimierung. In diesem Fall wurde die maximale Anzahl an Elementen für das Preloading auf 128 Elemente – 8 Blöcke mit je 16 Elementen – beziehungsweise die Breite des Bildes festgelegt (Zeile 2). Die Schleife in Zeile 9 iteriert zeilenweise über das Bild. Für jede Zeile werden solange abwechselnd das Preloading (Zeile 13) für die maximale Anzahl an Elementen und die Berechnung der Spaltensumme (Zeile 18) durchgeführt, bis alle Elemente verarbeitet wurden.

```
unsigned int regNum = w16 >> 4;
regNum = regNum > 8 ? 8 : regNum;

__m128i srcReg[regNum];
```

Programmauszug C.14: Bestimmung der minimalen Registeranzahl beim dynamischen *Array Preloading*. Bei dieser Implementierung werden maximal 8 Register verwendet, bei kleineren Bilder hingegen auch weniger. Der restliche Ablauf ist analog zum statischen Preloading (Progr.-ausz. C.13).

```
for(int y1 = 0; y1 < h16; ++y1)
{
    src1 = l1 + w16 * y1;

    for(int y2 = 0; y2 < h16; ++y2)
    {
        src2 = l2 + w16 * y2;
```

```
        as = _mm_setzero_si128();

        for(int x = 0; x < w16; x += 16)
        {
            as = _mm_adds_epi16(as, _mm_sad_epu8(*((__m128i*)(src1 + x)),
                                                 *((__m128i*)(src2 + x))));
        }

        as = _mm_shuffle_epi32(as, 0x0020);
        as = _mm_hadd_epi32(as, as);
        *dst = (SSEShort)_mm_cvtsi128_si32(as);

        ++dst;
    }
}
```

Programmauszug C.15: Umsetzung der *natürlichen* Schleifenreihenfolge zur Berechnung der SAD. Dabei werden die beiden Bildspalten vollständig durchlaufen, das Element der Ergebnismatrix komplett berechnet und in den Speicher geschrieben.

```
SSEFloat *ssOrig = newSIMDFloatImageFromImage(ss, w16, h);
SSEFloat *cvOrig = newSIMDFloatImageFromImage(cv, w16, h);

int hEdge = h - 1;
const int w16EdgeScaled = (hEdge + 15) & ~15;
SSEFloat *cvEdge = newSIMDFloatImage(w16, hEdge);
SSEFloat *ssEdge = newSIMDFloatImage(w16, hEdge);
verticalEdgeFilterSIMDFloatImage(cvOrig, w16, h, w16, cvEdge);
verticalEdgeFilterSIMDFloatImage(ssOrig, w16, h, w16, ssEdge);

SSEByte *cvOrigEdgeScaled = newSIMDColPadByteImage(w16EdgeScaled, w16);
SSEByte *ssOrigEdgeScaled = newSIMDColPadByteImage(w16EdgeScaled, w16);
SSEByte *cvEdgeScaled = newSIMDColPadByteImage(w16EdgeScaled, w16);
SSEByte *ssEdgeScaled = newSIMDColPadByteImage(w16EdgeScaled, w16);
SSEByte *cvEdgeScaledPtr, *ssEdgeScaledPtr;
SSEShort *cvOrigAbsSum = newSIMDShortImage(w16);
SSEShort *ssOrigAbsSum = newSIMDShortImage(w16);
SSEShort *cvAbsSum = newSIMDShortImage(w16);
SSEShort *ssAbsSum = newSIMDShortImage(w16);
SSEShort *cvAbsSumPtr, *ssAbsSumPtr;
SSEShort *dirTerm = newSIMDShortImage(w16, w16);

copyAndScaleSIMDFloat2ColPadByteImageBlockTranspose(cvEdge, pixelScale,
    w16, hEdge, w16EdgeScaled, cvOrigEdgeScaled);
sumOfAbsSIMDColPadByte(cvOrigEdgeScaled, w16EdgeScaled, w16,
    cvOrigAbsSum);
copyAndScaleSIMDFloat2ColPadByteImageBlockTranspose(ssEdge, pixelScale,
    w16, hEdge, w16EdgeScaled, ssOrigEdgeScaled);
sumOfAbsSIMDColPadByte(ssOrigEdgeScaled, w16EdgeScaled, w16,
    ssOrigAbsSum);

for(unsigned int spIndex = 0; spIndex < scaleFactors.size(); spIndex++)
{
    double scale = scaleFactors[spIndex];

    if(scale > 1.0)
    {
        float multiplyScaleInv = multiplyByScale ? pixelScale / scale :
```

```
                                                    pixelScale;
    magnifyAndScaleSIMDFloat2ColPadByteImageBlockTranspose(cvEdge,
        interpolation, 1.0 / scale, multiplyScaleInv,
        w16, hEdge, w16EdgeScaled, horizon - 1,
        cvEdgeScaled);
    cvEdgeScaledPtr = cvEdgeScaled;
    sumOfAbsSIMDColPadByte(cvEdgeScaled, w16EdgeScaled, w16,
        cvAbsSum);
    cvAbsSumPtr = cvAbsSum;
    ssEdgeScaledPtr = ssOrigEdgeScaled;
    ssAbsSumPtr = ssOrigAbsSum;
}
else if(scale < 1.0)
{
    cvEdgeScaledPtr = cvOrigEdgeScaled;
    cvAbsSumPtr = cvOrigAbsSum;
    float multiplyScale = multiplyByScale ? pixelScale * scale :
                                            pixelScale;
    magnifyAndScaleSIMDFloat2ColPadByteImageBlockTranspose(ssEdge,
        interpolation, scale, multiplyScale,
        w16, hEdge, w16EdgeScaled, horizon - 1,
        ssEdgeScaled);
    ssEdgeScaledPtr = ssEdgeScaled;
    sumOfAbsSIMDColPadByte(ssEdgeScaled, w16EdgeScaled, w16,
        ssAbsSum);
    ssAbsSumPtr = ssAbsSum;
}
else
{
    cvEdgeScaledPtr = cvOrigEdgeScaled;
    cvAbsSumPtr = cvOrigAbsSum;
    ssEdgeScaledPtr = ssOrigEdgeScaled;
    ssAbsSumPtr = ssOrigAbsSum;
}

normSumAbsDiffTermColPadByte(cvEdgeScaledPtr, ssEdgeScaledPtr,
    w16EdgeScaled, w16, dirTerm);

SSEByte *destP = scalePlane[spIndex];
SSEShort *dirP = dirTerm;
unsigned int *modulo2woff = modulo2wpw;

for(int i = 0; i < w16; i++)
{
    __m128i cvAbsSumx8 = _mm_set1_epi16(cvAbsSumPtr[i] + 1);
    SSEShort *ssAbsSumP = ssAbsSumPtr;
    unsigned int *shufflepj = shuffle;

    for(int j = 0; j < w16; j += 16)
    {
        __m128i absSumSumL = _mm_adds_epi16(cvAbsSumx8,
                                *(__m128i*)ssAbsSumP),
                absSumSumH = _mm_adds_epi16(cvAbsSumx8,
                                *(__m128i*)(ssAbsSumP + 8));

        __m128 LfLL, LfLH, LfHL, LfHH;
        _MM_8SHORT_TO_2X4FLOAT(absSumSumL, LfLL, LfLH);
        _MM_8SHORT_TO_2X4FLOAT(absSumSumH, LfHL, LfHH);

        __m128i DL = _mm_load_si128((__m128i*) dirP),
                DH = _mm_load_si128((__m128i*) (dirP + 8));
```

```
            __m128 DfLL, DfLH, DfHL, DfHH;                                  101
            _MM_8SHORT_TO_2X4FLOAT(DL, DfLL, DfLH);                         102
            _MM_8SHORT_TO_2X4FLOAT(DH, DfHL, DfHH);                         103
                                                                            104
            __m128i C = _MM_4X4FLOAT_TO_16UBYTE(                            105
                        _MM_NDIST_PS(DfLL, LfLL, postScale4),               106
                        _MM_NDIST_PS(DfLH, LfLH, postScale4),               107
                        _MM_NDIST_PS(DfHL, LfHL, postScale4),               108
                        _MM_NDIST_PS(DfHH, LfHH, postScale4));              109
            _mm_store_si128((__m128i*)Cstore, C);                           110
                                                                            111
            unsigned int *modulo2woffmj = modulo2woff - j;                  112
                                                                            113
            for(int k = 0; k < 16; k++)                                     114
            {                                                               115
                *(destP + shufflepj[k] + *modulo2woffmj) = Cstore[k];       116
                modulo2woffmj--;                                            117
            }                                                               118
                                                                            119
            dirP += 16;                                                     120
            ssAbsSumP += 16;                                                121
            shufflepj += 16;                                                122
        }                                                                   123
                                                                            124
        modulo2woff++;                                                      125
    }                                                                       126
}                                                                           127
```

Programmauszug C.16: Ablauf der Berechnung der *Normalised Sum of Absolute Differences* bei der 16-Bit-Version. In den ersten beiden Zeilen werden die beiden Eingangsbilder in die einheitliche Darstellung mittels Gleitkommazahlen konvertiert. Anschließend wird in den Zeilen 4 bis 9 der Kantenfilter angewendet. Für die Originalgrößen der Bilder werden vorab in den Zeilen 23 bis 30 die Nennerterme der NSAD berechnet, da diese mehrmals verwendet werden. In der Schleife über alle *Scale Planes* wird je nach Skalierungsfaktor unterschieden, ob das aktuelle Bild (Zeile 36) oder das Zielbild (Zeile 51) vergrößert werden muss. Im dritten Fall (Zeile 66), in dem die Bilder unverändert verglichen werden, ist es ausreichend, die Zeiger auf die Originalbilder zu setzen. Die Vorbereitungen werden durch die Berechnung des Zählers in Zeile 74 abgeschlossen. Vor der 4-fach parallelen Division müssen die Nennerterme addiert (Zeilen 89 bis 92) und die 16-Bit-Werte in 32-Bit-Gleitkommazahlen umgewandelt werden. Letzteres geschieht in den Zeilen 94 bis 103 und ist nur über eine zwischenzeitliche Konvertierung in 32-Bit-Ganzzahlen möglich (`_MM_8SHORT_TO_2X4FLOAT`, Progr.-ausz. C.1). Das Makro `_MM_NDIST_PS` (Progr.-ausz. C.3) führt die Division durch und skaliert die Ergebnisse, bevor sie mittels `_MM_4X4FLOAT_TO_16UBYTE` (Progr.-ausz. C.2) in den Wertebereich des *Scale Plane Stacks* umgewandelt werden. Ab Zeile 114 werden die Werte *geshuffled* in den Speicher geschrieben. Das erfordert zwar an dieser Stelle ein elementweises Kopieren, ist aber für die zweite Phase des Min-Warping notwendig.

```
printf("copyAndScaleSIMDFloat2ShortImage␣");
fflush(stdout);
t0 = getTimeUsec();
for(int i = 0; i < ITERATIONS_CASSF; ++i)
    copyAndScaleSIMDFloat2ShortImage(floatImageWH, multiplyScale, w, h,
                                     shortImageWH);
t1 = getTimeUsec();
printf("%lld\n", t1 - t0);
```

Programmauszug C.17: Zeitmessung am Beispiel der Funktion `copyAndScale`

```
#define _mm_extract_epi32(x, imm) \
    _mm_cvtsi128_si32(_mm_srli_si128((x), 4 * (imm)))
```

Programmauszug C.18: Makro zum Extrahieren von ganzzahligen 32-Bit-Elementen unter SSSE3

Literaturverzeichnis

Flynn, M. *Very high-speed computing systems*. Proceedings of the IEEE, 54 (12):1901–1909, December 1966. ISSN 0018-9219.

Fog, A. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2014a. URL `http://www.agner.org/optimize/instruction_tables.pdf`.

Fog, A. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2014b. URL `http://www.agner.org/optimize/microarchitecture.pdf`.

Hennessy, J. L. ; Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5. Edition, 2011. ISBN 012383872X, 9780123838728.

Intel® Corporation. *Intel® C++ Intrinsic Reference*, 2007. Document Number: 312482-003US.

Intel® Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, March 2014a. URL `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

Intel® Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, February 2014b. URL `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C.

Intel® Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, February 2014c. URL `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developers-manual.pdf`. Documentation Changes.

Intel® Corporation. *Intel® Intrinsics Guide*, 2014d. URL `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. Letzter Zugriff: 8. August 2014.

The LLVM Compiler Infrastructure. *xmmintrin.h*, 2014. URL `http://clang.llvm.org/doxygen/xmmintrin_8h_source.html`. Letzter Zugriff: 8. August 2014.

Möller, R. ; Horst, M. ; Fleer, D. *Illumination Tolerance for Visual Navigation with the Holistic Min-Warping Method*. Robotics, 3(1):22–67, 2014.

Möller, R. ; Krzykawski, M. ; Gerstmayr, L. *Three 2D-Warping Schemes for Visual Robot Navigation*. Autonomous Robots, 29(3):253–291, 2010.

nick_1234. *Transpose16x16A.asm*, 2012. URL `https://software.intel.com/en-us/forums/topic/279431#comment-1469379`. Letzter Zugriff: 8. August 2014.

Paul R. *Transpose for 8 registers of 16-bit elements on SSE2/SSSE3*, 2010. URL `http://stackoverflow.com/a/2518670`. Letzter Zugriff: 8. August 2014.

Wikipedia. *Sum of absolute differences*, 2014. URL `http://en.wikipedia.org/w/index.php?title=Sum_of_absolute_differences&oldid=618212985`. Letzter Zugriff: 8. August 2014.