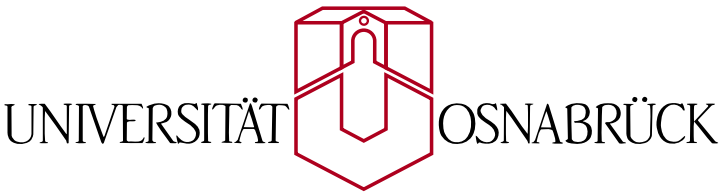


Paralleles konturbasiertes Connected-Component-Labeling für 2D-Bilddaten mit OpenCL und Cuda

Dissertation
zur Erlangung des Doktorgrades (Dr. rer. nat.) des Fachbereichs Mathematik/Informatik der Universität Osnabrück

Vorgelegt von:
Henning Wenke

Betreuer:
Prof. Dr. Oliver Vornberger

Mai 2015

Zusammenfassung

Connected-Component-Labeling (CCL) für 2D-Bilddaten ist ein bekanntes Problem im Bereich der Bildverarbeitung. Ziel ist es, zusammenhängende Pixelgruppen mit gleichen Eigenschaften zu erkennen und mit einem eindeutigen Label zu versehen.

Zur Lösung von CCL-Problemen für 2D-Bilddaten werden sowohl sequentielle als auch parallele Algorithmen untersucht. Unter den bekannten Algorithmen gibt es solche, die asymptotisch optimale Eigenschaften besitzen.

Speziell für den Bereich der Bildverarbeitung interessant sind außerdem auf Konturierung basierende Algorithmen. Die zusätzlich extrahierten Konturen können z.B. für die Buchstabenerkennung genutzt werden.

Seit der jüngeren Vergangenheit werden Grafikprozessoren (GPUs) mit großem Erfolg für allgemeines Computing eingesetzt. So existieren auch mehrere Implementationen von Connected-Component-Labeling-Algorithmen für GPUs, welche im Vergleich mit Varianten für CPUs oft deutlich schneller sind. Diese GPU-basierten Ansätze verarbeiten typischerweise das Pixelgitter direkt.

Im Rahmen der vorliegenden Arbeit werden mehrere neue parallele CCL-Algorithmen vorgeschlagen, welche auf Konturen basieren und sowohl für GPUs als auch für Multicore-CPUs geeignet sind. Diese werden experimentell mit Implementationen aus der Literatur unter Verwendung aktueller GPUs und CPUs verglichen. Dabei erreichen in vielen Fällen die vorgeschlagenen Techniken ein besseres Laufzeitverhalten.

Das ist auf GPUs insbesondere dann besonders deutlich, wenn sich die evaluierten Datensätze durch einen geringen Anteil von Konturen im Vergleich zur Fläche der Connected-Components auszeichnen.

Danksagung

Ich möchte mich hiermit bei allen Personen bedanken, die mich während der Zeit meiner Promotion auf vielfältige Weise unterstützt haben. Besonders bedanken möchte ich mich bei:

- Herrn Prof. Dr. Oliver Vornberger für die gute Betreuung dieser Arbeit
- Herrn Prof. Dr. Udo Frese für seine hilfreichen Anregungen
- Meinen Kollegen, Sascha Kolodzey, Nils Vollmer und Erik Wittkorn aus unserem Gründungsprojekt „ARR-Engine“, für die Geduld während der (etwas länger als geplanten) Zeit bis zur Abgabe dieser Arbeit.
- Meinen Eltern Anna und Günther Wenke sowie meiner Schwester Hanna Ewen für die Unterstützung in verschiedenster Form die ganze Zeit über

Ganz besonderer Dank gilt Sascha Kolodzey für das Korrekturlesen, die Unterstützung bei der Portierung nach C++ / Cuda in Rekordzeit, und die zahlreichen Anregungen in Diskussionen!

Angaben zu Drittmitteln

Für einige Experimente in der vorliegenden Arbeit wurde eine Intel Core i7-5960X CPU verwendet, die aus Mitteln des „EXIST-Gründerstipendiums: ARR-Engine“ (Förderkennzeichen 03EGSNI102) angeschafft wurde. Ich bin als Stipendiat Mitglied des Projekts. Die Mittel stammen aus dem Bundeshaushalt sowie dem Europäischen Sozialfonds (ESF).

Die Ergebnisse dieser Arbeit können auch für das Projekt ARR-Engine genutzt werden und zwar für das im zugehörigen Ideenpapier beschriebene Pilotprojekt „W-App“. Dabei handelt es sich um eine Software zur Echtzeitvisualisierung und Animation über die Zeit von personalisierbaren Wetterkarten für mobile Geräte und das Web. Eine Render-Technik, welche z.B. für die Darstellung des Luftdrucks Verwendung finden kann, sind Isolinien (bzw. hier Isobaren). Isolinien verbinden Punkte gleichen Wertes eines zweidimensionalen Skalarfeldes und werden in der W-App implizit im Bildraum gerendert. Varianten der im Rahmen dieser Arbeit gefundenen Techniken können verwendet werden, um die gerenderten Isolinien mit genau einer Beschriftung (typischerweise deren jeweiliger Wert) je Isolinie in Echtzeit zu versehen.

Inhaltsverzeichnis

Teil I.

Prolog

Kapitel 1.

Einführung: Parallele Algorithmen

In diesem Abschnitt werden die benötigten Grundlagen der Formulierung und Bewertung paralleler Algorithmen für das PRAM-Modell eingeführt. Es folgt ein kurzer Ausblick auf eine Distributed-Memory-Architektur. Inhaltlich basiert Kapitel 1, solange keine anderen Quellen angegeben sind, auf [MB13], [KR90], [Vis83] und [RR10].

1.1. Das PRAM-Modell

Das RAM (Random-Access-Machine)-Modell ist seit seiner Einführung ein Standardwerkzeug zum Formulieren und Analysieren von sequentiellen Algorithmen für einen idealisierten Computer. Es besteht aus einer einzigen Recheneinheit, im folgenden **Prozessor** bezeichnet, welche lediglich eine Operation zur Zeit ausführen kann. Dieser Prozessor ist ein rein theoretisches Konstrukt und meint insbesondere keinen realen Prozessor, wie etwa einen Intel Core i7 2700k.

Zusätzlich gibt es einen Speicher ausreichender Größe. Der Prozessor kann mittels einer Speicherzugriffseinheit auf jede Adresse darin lesend oder schreibend zugreifen und die Kosten dafür sind immer $O(1)$ Zeiteinheiten.

Jede Operation eines Algorithmus, die der Prozessor ausführt, besteht aus den drei Phasen Read, Compute und Write, die immer in dieser Reihenfolge ausgeführt werden und sich zeitlich nicht überlappen.

Das idealisierte PRAM (Parallel-Random-Access-Machine)-Modell [FW78, SS79] kann als parallele Variante des RAM-Modells angesehen werden. Es ist ebenfalls ein sehr verbreitetes Modell zur Analyse paralleler Algorithmen. Hier existiert eine Menge von P identischen Prozessoren, von denen jeder gemäß dem RAM-Modell funktioniert. Alle Prozessoren führen das gleiche Programm aus und werden dabei durch einen globalen Taktgeber kontrolliert. Folglich laufen alle Prozessoren immer synchron. Wie im RAM-Modell können alle Prozessoren auf alle Speicheradressen in O(1) Zeit zugreifen. Aufgrund dieses globalen Speichers spricht man beim PRAM auch von einer Shared-Memory-Architektur.

PRAM-Varianten

In der Read- oder Write- Phase einer Operation kann es zu Konflikten kommen, wenn mehrere Prozessoren in einem Takt auf die gleiche Adresse zugreifen. In Bezug auf Lesezugriffe existieren zwei Modelle:

- **Exclusive-Read (ER):** In diesem Modell haben Prozessoren exklusiven lesenden Zugriff auf Speicheradressen. Folglich ist eine Operation, welche den gleichzeitigen Zugriff zweier Prozessoren auf dieselbe Speicheradresse bewirkt, in einem ER-PRAM verboten. Algorithmen, in denen das nicht garantiert werden kann, sind nicht ausführbar.

- **Concurrent-Read (CR):** Im Gegensatz zum ER-PRAM sind im CR-PRAM gleichzeitige Lesezugriffe von einer Adresse erlaubt.

Analog lassen sich in Bezug auf Schreibkonflikte ebenfalls (im Groben) zwei Modelle unterscheiden. Beim **Exclusive-Write-PRAM** (EW-PRAM) ist der gleichzeitige schreibende Zugriff mehrerer Prozessoren auf eine Adresse verboten. Entsprechend ist diese Operation beim **Concurrent-Write-PRAM** (CW-PRAM) erlaubt. Im letzten Fall muss weiter unterschieden werden, wie im Konfliktfall vorzugehen ist. Einige der existieren Möglichkeiten sind:

- Arbitrary-CW: Hier setzt sich ein willkürlich ausgewählter Prozessor durch und alle anderen schreiben nichts.

- Combining-CW: Alle zu schreibenden Werte werden kombiniert und der so aggregierte Wert geschrieben. Denkbar sind die Summe, das Produkt, Minimum, Maximum, oder logische Operatoren.

- Priority-CW: Prozessoren erhalten Prioritäten und der mit der jeweils höchsten setzt sich durch.

- Common-CW: In diesem Modell dürfen Prozessoren nur dann gleichzeitig auf eine Adresse schreibend zugreifen, wenn der zu schreibende Wert aller Prozessoren identisch ist. Andernfalls ist der Algorithmus ungültig. In dieser Arbeit ist, wenn von einem CW-PRAM gesprochen wird, immer diese Variante gemeint.

Typischerweise wird das Lese- und Schreib-Modell kombiniert angegeben, etwa EREW-PRAM (Exclusive-Read-Exclusive-Write) oder CRCW-PRAM (Concurrent-Read-Concurrent-Write).

1.2. Analyse von parallelen Algorithmen auf einem PRAM

Im Falle eines sequentiellen Algorithmus wird i.d.R. primär die asymptotische Laufzeit analysiert. Dazu wird deren Verhalten als Funktion der Anzahl der auszuführenden elementaren Operationen in Abhängigkeit von der Problemgröße N, für $N \to \infty$ in der O-Notation angegeben. Die Anzahl der auszuführenden Operationen kann datenabhängig sein, weshalb manchmal bei der Laufzeitanalyse zwischen Best-, Average- und Worst-Case unterschieden wird. Die Regel ist jedoch die Untersuchung des Worst-Case und damit eine datenunabhängige Garantie der Eigenschaften des Algorithmus. In dieser Arbeit wird ausschließlich der Worst-Case betrachtet. Die Laufzeit eines Algorithmus ist optimal, wenn bewiesen werden kann, dass es asymptotisch bis auf einen konstanten Faktor keine bessere Laufzeit geben kann. Bei Untersuchungen hinsichtlich Optimalität bezieht sich die Laufzeit immer auf den Worst-Case.

Im Falle eines parallelen Algorithmus für einen PRAM muss zunächst die Variante angegeben werden, auf der dieser analysiert wird. So ist es auf einem EREW-PRAM typischerweise ungleich schwieriger *gute* Eigenschaften zu erreichen als auf einem CREW-PRAM oder CRCW-PRAM. Weiterhin ist bei einem parallelen Algorithmus, anders als bei einem sequentiellen, z.B. die asymptotische Gesamtzahl der Operationen nicht identisch mit der Laufzeit. Dementsprechend muss die Analyse etwas differenzierter ausfallen.

Running-Time

Die Running-Time (Laufzeit) $T = T(N, P)$ eines parallelen Algorithmus wird in Abhängigkeit von der Problemgröße N und für eine Prozessorzahl $P = P(N)$ angegeben. Sie entspricht der Anzahl der Zeitschritte des globalen Taktgebers und damit insbesondere nicht der Gesamtzahl der Operationen. Schließlich können in jedem Takt bis zu $P(N)$ Operationen ausgeführt werden. Manchmal wird in der Literatur die Laufzeit auch nur in der Form $T = T(N)$ angegeben. Dann ist i.d.R. die Laufzeit bei der maximal verwendbaren Prozessorzahl gemeint.

Cost und Work

Sei $T = T(N, P)$ die von einem Algorithmus benötigte Laufzeit, die unter Verwendung von $P = P(N)$ Prozessoren erreicht wird. Dann ist Cost $C = C(N, P)$ gegeben als Produkt von Laufzeit und dafür benötigter Prozessorzahl: $C(N, P) = P(N) \cdot T(N, P)$.

Work $W = W(N)$ gibt dagegen die asymptotische Zahl *tatsächlich* ausgeführten Operationen an [MB13]. Kessler [Kes15] formuliert das so: Parallel-Work entspricht der maximalen Anzahl von Operationen, wenn in jedem (parallelen) Zeitschritt so viele Prozessoren verfügbar sind, wie benötigt werden, um diesen Schritt in $O(1)$ Zeit auszuführen. Folglich gilt immer: Work $\leq$ Cost, bzw. $W(N) = O(C(N, P))$.

Allerdings ist diese Unterscheidung zwischen Work und Cost in der Literatur nicht eindeutig. So verwenden Karp und Ramachandran [KR90] den Begriff Work mit einer Definition, die der obengenannten Definition von Cost entspricht. In dieser Arbeit werden

erstgenannte Definitionen verwendet, wobei zumeist nur Cost, mit $C(N,P) = P(N) \cdot T(N,P)$, benötigt wird.

Cost-Optimal und Work-Optimal

Sei für ein Problem Q ein sequentieller Algorithmus mit optimaler Laufzeit $T_{seq} = T_{seq}(N)$ bekannt. Dann ist ein paralleler Algorithmus mit Cost $C_{par} = C_{par}(N,P)$ cost-optimal, wenn gilt: $C_{par}(N,P) = O(T_{seq}(N))$.

Analog ist ein paralleler Algorithmus mit Work $W_{par}(N)$ work-optimal, wenn $W_{par}(N) = O(T_{seq}(N))$ gilt.

Speedup

Informell beschreibt Speedup das Verhältnis der Laufzeiten eines parallelen Algorithmus und des schnellsten Sequentiellen für das gleiche Problem. Sei für ein Problem Q ein sequentieller Algorithmus mit optimaler Laufzeit $T_{seq} = T_{seq}(N)$ bekannt. Dann ist Speedup $S = S(P,N)$ für einen parallelen Algorithmus mit Laufzeit $T_{par}(N,P)$: $S(P,N) = T_{seq}(N)/T_{par}(N,P)$.

Scalable

Ein paralleler Algorithmus wird als scalable (skalierbar) bezeichnet, wenn die Parallelität mindestens linear mit der Problemgröße steigt.

Optimalität

Karp und Ramachandran definieren die Bedingungen für Optimalität eines parallelen Algorithmus folgendermaßen [KR90]:

1. Der Algorithmus kann, mit $P(N)$ Prozessoren, asymptotisch polylogarithmische Laufzeit erreichen

2. Der Algorithmus ist cost-optimal für dieselbe Anzahl $P(N)$ Prozessoren

Informell kann, dieser Definition zufolge, ein optimaler paralleler Algorithmus hochgradig parallel arbeiten ohne dabei asymptotisch mehr Operationen auszuführen als ein optimaler sequentieller Algorithmus.

Effizient

Die Bedingungen für einen effizienten (efficient) parallelen Algorithmus definieren Karp und Ramachandran folgendermaßen [KR90]:

1. Der Algorithmus kann, mit $P(N)$ Prozessoren, asymptotisch polylogarithmische Laufzeit erreichen

2. Der Algorithmus führt für dieselbe Anzahl Prozessoren asymptotisch um maximal einen polylogarithmischen Faktor mehr Operationen aus als ein optimaler Algorithmus für dasselbe Problem

1.3. Andere parallele Architekturen

Die bisher beschriebene Architektur, der PRAM, ist eine Shared-Memory-Machine, bei der alle Prozessoren ohne Kosten auf alle Speicheradressen zugreifen können. In der Praxis kann ein solches Modell jedoch nicht auf beliebige Größen skaliert werden. So haben aktuelle GPUs einige tausend Recheneinheiten (Cores), von denen jede einzelne in etwa der Definition des theoretischen Prozessors entspricht, die alle auf einen gemeinsamen Speicher zugreifen können. Beispielsweise verfügt Nvidias Topmodell der Kepler Reihe, die Geforce GTX Titan Black, über 2880 Cores von denen jeder einzelne noch einige Hardware-Threads ausführen kann. Wenn mehr Rechenleistung bzw. Parallelität benötigt wird, müssen entsprechend mehrere GPUs eingesetzt werden, die dann aber nicht mehr über einen gemeinsamen Speicher verfügen.

In einem solchen Fall, spricht man von einer Distributed-Memory-Architektur und kann dementsprechend nicht mehr das PRAM-Modell zur Analyse heranziehen. In einem solchen Modell verfügt jeder Prozessor über einen eigenen Speicher und die Daten werden auf diese Speicher aufgeteilt. Zusätzlich gibt es zwischen einzelnen Prozessoren Verbindungen mit denen diese kommunizieren können. Die Gesamtheit dieser Verbindungen wird als Verbindungsnetzwerk bezeichnet. Es sind verschiedene Topologien denkbar, aber es muss garantiert werden, dass es von jedem Prozessor einen Pfad zu jedem anderen gibt. Zwei Beispiele für Verbindungsnetzwerk-Topologien, Ring und Mesh, sind in Abbildung 1.1 zu sehen. Offensichtlich können Pfade zwischen zwei Prozessoren je nach

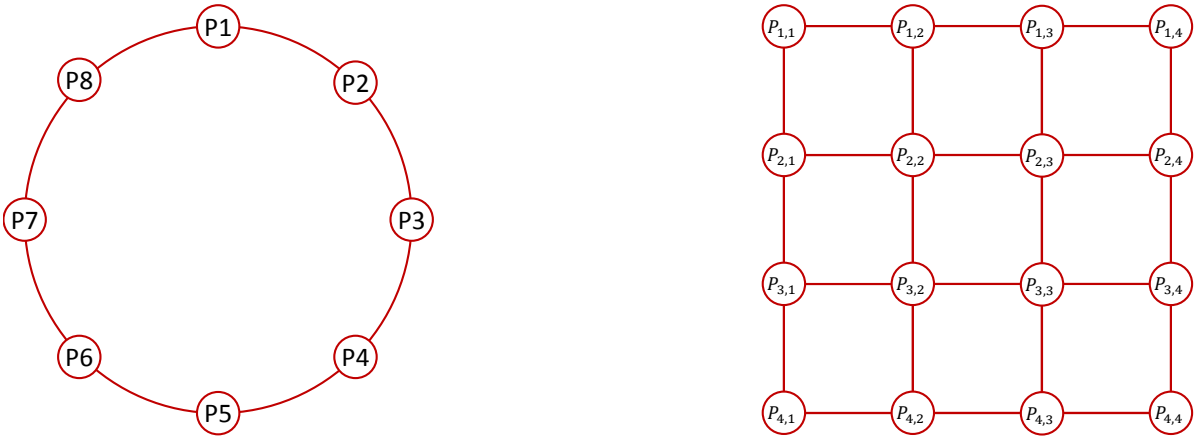


Abbildung 1.1.: Verbindungsnetzwerk-Topologien Ring (links) und Mesh (rechts)

gewählter Netztopologie und verwendetem Algorithmus unterschiedlich lang werden. Die mit dem Datenaustausch verbundenen Kosten können oft bewirken, dass sich Eigenschaften eines PRAM-Algorithmus nicht auf eine Distributed-Memory-Machine übertragen lassen.

In dieser Arbeit werden verschiedene Shared-Memory-Algorithmen formuliert und später mit OpenCL und Cuda für Geräte umgesetzt, deren Recheneinheiten ebenfalls auf einen gemeinsamen Speicher zugreifen können.

Kapitel 2.

Grundbegriffe: Paralleles Computing

In diesem Kapitel werden einige Grundbegriffe des parallelen Computings eingeführt. In diesem Bereich geht es darum, Implementationen von Algorithmen mit bereits bekannten theoretischen Eigenschaften beim Verarbeiten bestimmter Datensätze durch konkrete Hardware zu vergleichen. Dementsprechend ist das methodische Vorgehen, anders als im Bereich paralleler Algorithmen, experimentell und nicht analytisch.

Diese Arbeit legt den Fokus auf den experimentellen Vergleich des Laufzeitverhaltens von Implementationen paralleler Algorithmen bei Ausführung auf Graphics-Processing-Units (GPUs) oder Central-Processing-Units (CPUs). Das wird untersucht in Abhängigkeit des verwendeten Datensatzes, dessen Auflösung und des verwendeten Device (v.a. dessen Parallelität). Wir betrachten in dieser Arbeit ausschließlich Shared-Memory-Devices, also nicht etwa Systeme mit mehreren GPUs.

Die Laufzeit im Bereich des parallelen Computings ist, in dieser Arbeit, definiert als die messbare Ausführungszeit bei Verarbeitung durch ein bestimmtes (aber paralleles) Device. Bei Evaluation auf einem Device ist offensichtlich die Zahl der Recheneinheiten (entspricht dem Begriff Prozessoren des vorherigen Kapitels) fest. Diese Zahl P ist typischerweise deutlich kleiner als die Problemgröße N. Zu erwarten ist demnach auch in Algorithmen mit stark sublinearer Laufzeit (parallele Algorithmen Definition), z.B. $O(N/P)$ mit $P \leq N$, in Abhängigkeit der Problemgröße eine lineare Laufzeit (paralleles Computing Definition). Schließlich gilt hier $P << N$ und $P \neq P(N)$ für alle relevanten Größen von N und verfügbare Größen von P. Erhofft werden kann dagegen ein, hier experimentell zu ermittelnder, **Speedup**. Das ist das Verhältnis der gemessenen Laufzeiten zweier Ansätze. Dieser Begriff wird auch beim Vergleich paralleler Implementationen verwendet und hängt, anders als der gleichlautende Begriff aus dem Bereich paralleler Algorithmen, nicht unbedingt mit der Parallelität zusammen.

Die theoretische Laufzeitformulierung in Abhängigkeit von N und P bleibt aber wichtig, um abzuschätzen, welcher Algorithmus (abhängig von P_{max}) für welches Device (abhängig von der Zahl der Recheneinheiten) sinnvoll sein kann.

Oft wird nicht die gemessene Laufzeit direkt, sondern der **Throughput** verglichen. Dieser ist definiert als die Anzahl verarbeiteter Elemente (in dieser Arbeit i.d.R. Pixel) je Zeiteinheit:

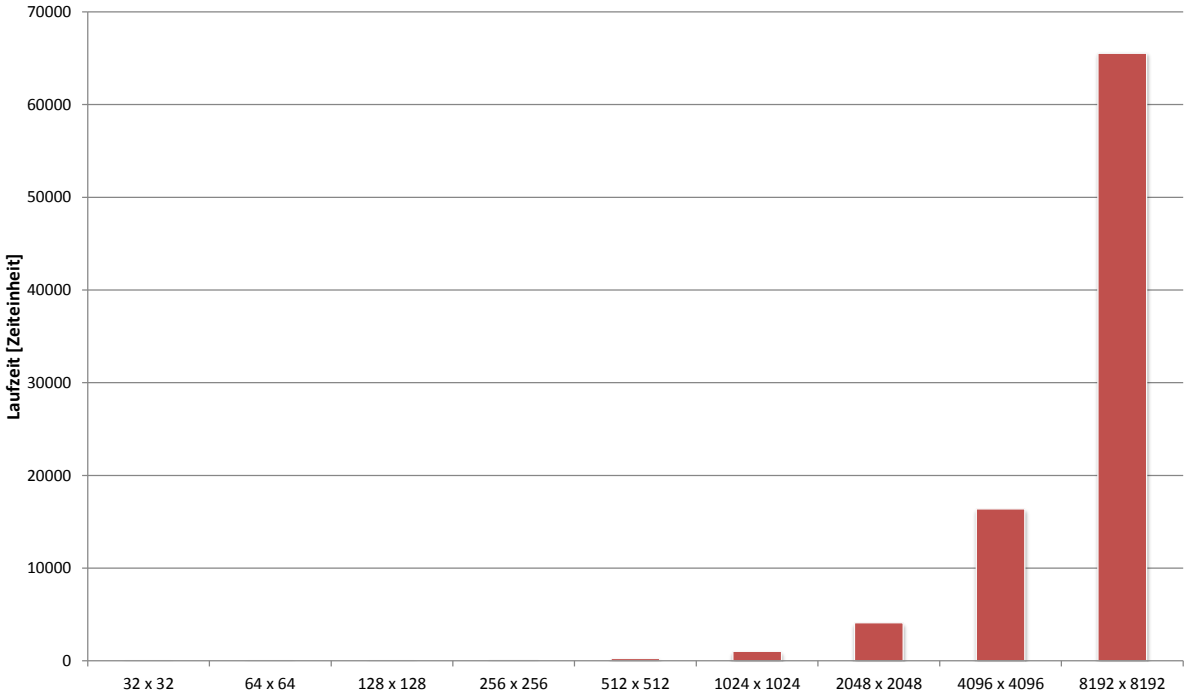
$$\text{Ermittelter Throughput [Pixel / Zeit]} = \frac{\text{Verwendete Datenauflösung [Pixel]}}{\text{Gemessene Laufzeit [Zeit]}} \tag{2.1}$$

Diese Form ist vorteilhaft bei grafischer Darstellung der Ergebnisse, vor allem wenn

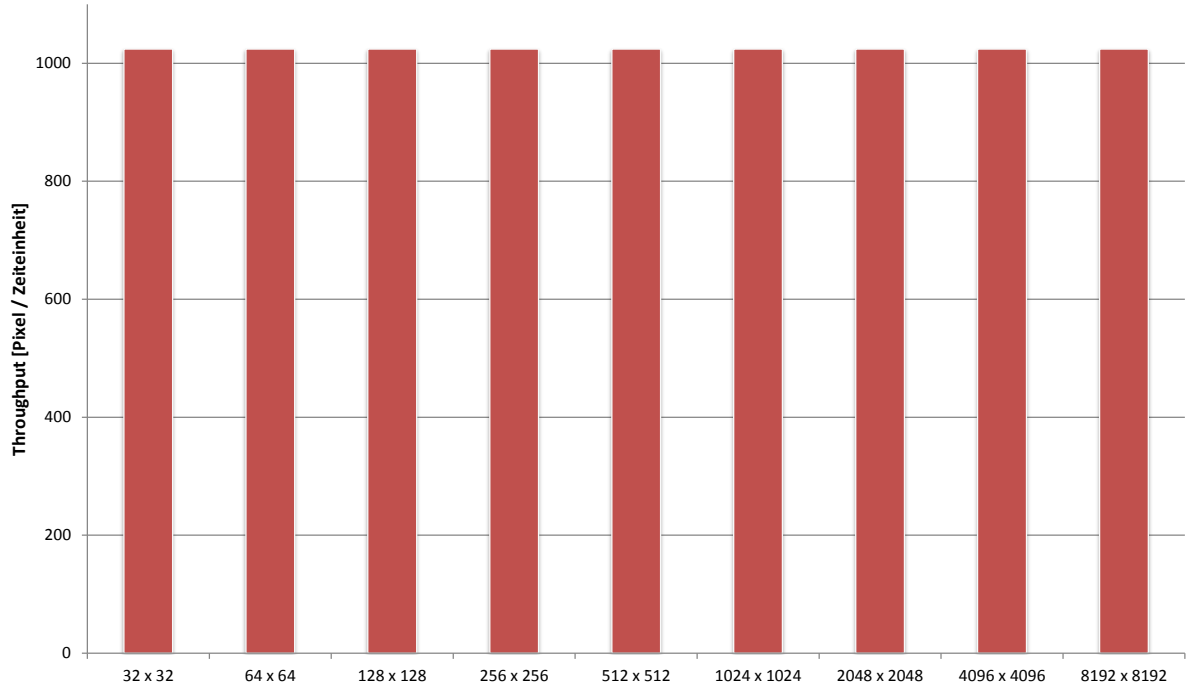
eine annähernd lineare Abhängigkeit der Laufzeit von der Problemgröße zu erwarten ist. Beide Darstellungen sind, für perfekt lineares Laufzeitverhalten, in Abbildung 2.1 zu sehen. Die Throughput-Darstellung ermöglicht einen guten grafischen Vergleich sehr verschiedener Problemgrößen und selbst geringe Abweichungen von einem linearen Laufzeitverhalten sind unmittelbar ersichtlich.

Wir werden in dieser Arbeit die Formulierungen 'höherer Throughput' und 'schneller' (bei gleicher Datenauflösung) synonym verwenden. Das mag auch unabhängig davon sein, ob in einer Grafik Throughput oder Laufzeit dargestellt wird. Es handelt sich schließlich nur um eine andere Darstellungsformen der gleichen Messgröße.

Eine detailliertere Beschreibung der Methodik und Möglichkeiten der Optimierung (Anpassung an Funktionsweise eines bestimmten Device) folgen später in dieser Arbeit.



(a) Gemessene Laufzeit, angegeben in Zeiteinheiten, für verschiedene Bildauflösungen in Pixeln



(b) Ermittelter Throughput, angegeben in verarbeiteten Pixeln je Zeiteinheit, für verschiedene Bildauflösungen in Pixeln. Throughputs basieren auf den Laufzeiten aus Teilabbildung (a)

Abbildung 2.1.: Vergleich der Darstellungen von Laufzeit und Throughput basierend auf denselben (synthetischen) Messdaten in verschiedenen Bildauflösungen. Die Laufzeit steige genau linear in Abhängigkeit von der Problemgröße. Ferner betrage die Laufzeit in der Auflösung 32 x 32 Pixel eine Zeiteinheit, in 64 x 64 Pixeln 4 Zeiteinheiten, usw.

Kapitel 3.

Einleitung

Eine Connected-Component ist eine verbundene Teilmenge der Knoten eines Graphen, welche alle über denselben Wert verfügen. Für einen gegebenen Graphen bestimmt ein Connected-Component-Labeling-Algorithmus für jede Connected-Component ein individuelles Label und weist es jeweils allen zugehörigen Knoten zu.

Das Connected-Component-Labeling-Problem kann für allgemeine Graphen mit N Knoten und K Kanten sequentiell mithilfe der Tiefensuche in O(N + K) Zeit und damit optimal gelöst werden [MB13]. Allerdings sind Möglichkeiten der effizienten parallelen Breiten- und Tiefensuche nicht bekannt [KR90]. Dementsprechend mussten andere Ansätze gefunden werden, um Algorithmen für paralleles Connected-Component-Labeling zu formulieren. Einen Überblick darüber bietet [KR90].

Ein effizienter Ansatz von Hirschberg, Chandra und Sarwate arbeitet in $log(N)$ Stufen [HCS79]. In jedem Schritt sind die Knoten in einem 'Wald' aus gerichteten Bäumen organisiert, wobei jeder Knoten eine Kante zu seinem Vater hat. Alle Knoten je eines Baumes gehören dabei zu einer unabhängigen Connected-Component. In der ersten Stufe ist jeder Knoten Teil eines Baumes und im letzten Schritt sind alle Knoten einer Connected-Component einem Baum der Höhe eins enthalten. Die Operation, welche die Daten von einer Phase zur nächsten überführt wird als 'Hooking-Process' bezeichnet. In Bäumen mit benachbarten Knoten werden einige verlinked und durch Pointer-Jumps gekürzt. Bei einem Pointer-Jump erhält jeder Knoten, der nicht Wurzelknoten oder Kind eines Wurzelknotens ist, den Vater seines Vaters als Vaterknoten. Dabei muss die Baumstruktur erhalten bleiben und es kann garantiert werden, dass das Verfahren nicht mehr als $log(N)$ Stufen benötigt.

Es gibt Varianten dieses grundlegenden Ansatzes für verschiedene PRAM-Modelle: CREW [HCS79], EREW [NM82] und CRCW [AS87, SV82]. Auf einem arbitrary CRCW-PRAM werden $O(N + K)$ Prozessoren benötigt, um das Problem in $O(log(N))$ Zeit zu lösen, wenn N die Knotenzahl und K die Kantenzahl ist. Auf dem EREW-PRAM wird das Problem bei gleicher Prozessorzahl in $O(log(N) \cdot log(N))$ Zeit gelöst.

Ein fortschrittlicherer Ansatz [CV86a], basierend auf optimalem parallelen List-Ranking, kann Connected-Component-Labeling auf einem arbitrary CRCW-PRAM in $log(N)$ Zeit mit $O((N + K) \cdot \alpha(N, K)/log(N))$ Prozessoren lösen. Dabei ist $\alpha(N, K)$ die inverse Ackermann Funktion, welche extrem langsam mit N und K steigt. Ferner existieren randomisierte Ansätze, etwa [Gaz86], welche in dieser Arbeit nicht weiter vertieft werden, da der Fokus auf deterministischen parallelen Techniken liegt. Bislang ist kein opti-

maler deterministischer paralleler Connected-Component-Labeling-Algorithmus bekannt [CNP04].

Speziell für 2D-Bilddaten hat Cypher einen auf Objektkonturen basierenden Ansatz vorgestellt [CSS89]. Dieser geht in drei Schritten vor: Zuerst werden Kontursegmente extrahiert und als jeweils eine zyklische Linked-List je Kontur repräsentiert. Im zweiten Schritt wird mit einer Pointer-Jumping-Technik das Minimum jeder Linked-List bestimmt. Dieser Wert kann als Label verwendet werden. Anschließend bekommen innere Konturen den Wert der zugehörigen äußeren Kontur. Zuletzt werden die Bereiche in den Konturen mit deren Werten gefüllt.

Einen ähnlichen Ansatz hat Agrawahl vorgestellt [ANL87]. Er basiert nicht direkt auf den Pixelkanten, sondern teilt jeden Pixel in vier Subpixel auf, welche dann ähnlich wie Kontursegmente behandelt werden. Dieser Ansatz ist, anders als der von Cypher, nicht auf die Verarbeitung von Binärdaten beschränkt.

Diese konturbasierten Ansätze liefern auf einem EREW-PRAM $O(log(N))$ Laufzeit bei $O(N)$ Prozessoren. Die Funktionsweise dieser Algorithmen ist denen in dieser Arbeit erarbeiteten am ähnlichsten, sodass sie später noch detaillierter damit verglichen werden.

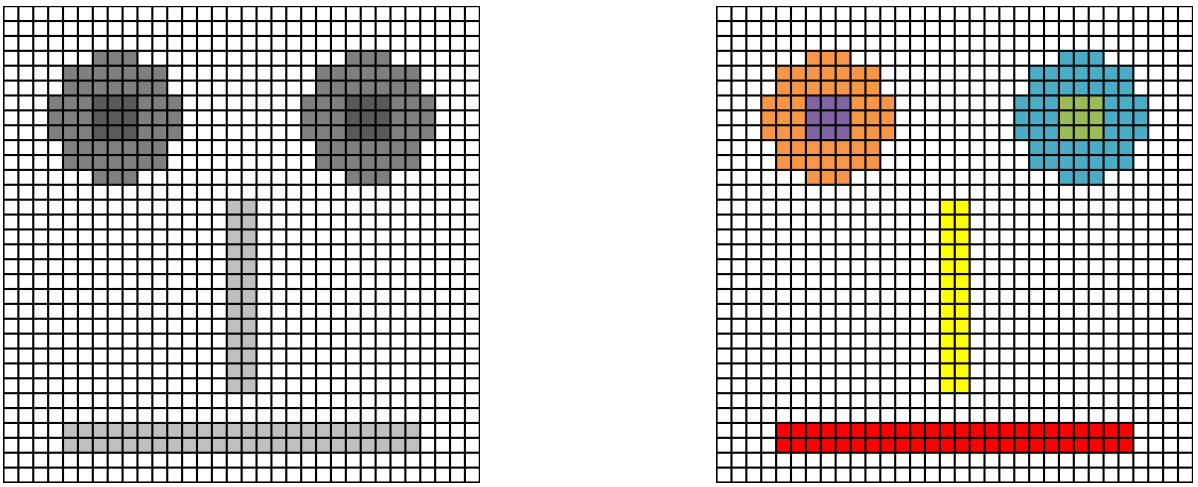
Zuletzt wichtig ist ein von Hagerup 1988 vorgestellter optimaler paralleler Connected-Component-Labeling-Algorithmus für planare Graphen [Hag88], der somit auch auf 2D-Bilddaten anwendbar ist. Der Ansatz verwendet Cole und Vishkins optimalen parallelen Scan-Algorithmus[CV89], basiert aber nicht auf Konturen.

Die Betrachtungen dieser Arbeit beschränken sich auf 2D rectilineare Bilddaten. Diese werden auch als 2D-Rasterdaten bezeichnet. Nachfolgend werden wir von 2D-Bilddaten sprechen und einzelne Knoten bzw. Einträge werden als Pixel bezeichnet.

Im Fall solcher Daten wird die Nachbarschaft eines Pixels typischerweise auf eine von zwei verschiedenen Arten definiert: Vierer-Nachbarschaft oder Achter-Nachbarschaft [Ros70]. Beide Nachbarschaften sind in Abbildung 3.1 gegeben.

Abbildung 3.1.: Nachbarschaften eines Pixels p. Links: Vierer-Nachbarschaft, rechts: Achter-Nachbarschaft

Primäre Anwendungsbereiche von Connected-Component-Labeling-Algorithmen für 2D-Bilddaten sind Pattern-Recognition, Computer-Vision und Image-Processing. Das beinhaltet zum Beispiel Character-Recognition [KKS00, SSR01]. Zusätzlich können die Konturen von Objekten in vielen Fällen für die 2D-Objekterkennung hilfreich sein [CCL04]. Beispiele für Methoden, welche Objektkonturen nutzen sind solche, die auf Chain-Codes basieren, wie [Fre61], oder Fourier-Deskriptoren basierende, wie [PF86]. Weiterhin gibt



(a) Klassifikationen als Graustufen **(b)** Label als Farben

Abbildung 3.2.: Anwendung des CCL-Algorithmus auf die Ergebnisse eines Segmentation Schritts. Links sind verschiedene Klassifikationen als Graustufen dargestellt (sehr dunkel: Pupille, dunkel: Auge, hell: Mund/Nase, weiß: Nichts davon). Die durch den CCL-Algorithmus bestimmten Label sind in der rechten Abbildung zu sehen.

es verschiedene Ansätze, welche bestimmte Eigenschaften zur Klassifikation von Buchstaben nutzen [CLP99], wie [JDM00] und [ODTT95].

Connected-Component-Labeling ist nicht mit Segmentation [HS85, SM00, WB03] zu verwechseln, welches genutzt werden kann, um alle Elemente zu identifizieren, welche zu einem bestimmten Typ von Objekt gehören. Alle Pixel aller Objekte je eines Typs bekommen dann einen eindeutigen Wert, welchen wir als Klassifikation bezeichnen wollen, zugewiesen.

Beispielsweise kann ein Segmentation-Algorithmus alle Pixel im Bild identifizieren, welche Teil eines Auges sind. Diese bekommen alle die gleiche Klassifikation. Im Gegensatz dazu kann ein Connected-Component-Labeling-Algorithmus die Ausgangsdaten des Segmentation-Algorithmus als Eingangsdaten verwenden und ein eindeutiges Label für jedes einzelne (d.h. räumlich getrennte) Auge ermitteln. Dies kann bei der Erkennung einzelner Augen hilfreich sein. Als Beispiel für nicht binäre Daten mag es zusätzlich zum Hintergrund (nicht klassifiziert) und den als Auge klassifizierten Pixeln noch solche geben, die als einer Pupille zugehörig erkannt wurden. In diesem Fall wird der Segmentation-Algorithmus diesen Pixeln eine andere Klassifikation zuweisen und der anschließende Connected-Component-Labeling-Algorithmus wird den Pupillenpixeln folglich ein anderes Label zuweisen als den Augpixeln, auch wenn sich beide berühren. Dieses Beispiel ist in Abbildung 3.2 veranschaulicht.

3.1. Implementation von Connected-Component-Labeling

Ein grundlegender Ansatz für Connected-Component-Labeling-Algorithmen für 2D-Bilddaten in der Praxis wird von Wu et al. [WOS09] beschrieben: Diese einfachste Lösung überprüft die Eingangsdaten (z.B. zeilenweise) so oft, bis keine Änderungen an den Labeln bei Vergleichen mit ihren Nachbarn mehr durchgeführt werden müssen [RK82].

Es ist im Allgemeinen nicht ausreichend, nur die unmittelbare Umgebung jedes Pixels zu betrachten, um global eindeutige Label zu vergeben. Deshalb verwenden viele Algorithmen temporäre Label, welche auch als provisorische Label bezeichnet werden. Falls das Pixelgitter 'vorwärts' verarbeitet wird (zeilenweise von oben nach unten und in jeder Zeile von links nach rechts), muss dafür eine geeignete Forward-Scan-Mask angewendet werden. Diese ist in Abbildung 3.3 für den Test hinsichtlich Vierer-Nachbarschaft gegeben. Weiterhin ist auch die Scan-Mask für Rückwärtsverarbeitung angegeben. Betrachten wir beispielsweise Pixel p in Abbildung 3.3. Die beiden Nachbarn von p in der

Abbildung 3.3.: Scan Masks für Vierer-Nachbarschaft. Links: Forward, rechts: Backward

Scan-Mask heißen n_1 und n_2. Falls n_1 und n_2 anders klassifiziert sind als p oder noch gar kein Label haben, erhält p ein neues provisorisches Label. Falls n_1 und/oder n_2 identisch mit p klassifiziert (und mit einem Label versehen) sind, wird ein repräsentatives Label ausgewählt und p zugewiesen. Dies kann z.B. das Minimum der Label von n_1 und n_2 sein.

Fortgeschrittenere Algorithmen können beispielsweise eine eigene Datenstruktur verwenden, um die Gleichheitsinformation zu verwalten und andere Mechanismen anzuwenden, um die repräsentativen Label auszuwählen. Ein Connected-Component-Labeling-Algorithmus für 2D-Bilddaten wird als optimal bezeichnet, wenn seine asymptotische Laufzeit $O(N)$ ist. Dabei entspricht N der Anzahl der Pixel und somit der Problemgröße.

Gemäß Hernandez-Belmonte [HBARSY11] und weiteren [HCSW09, WOS09] lassen sich Connected-Component-Labeling-Algorithmen in drei Klassen einteilen. Entscheidungskriterium ist dabei die nötige Anzahl der Durchläufe (Passes).

Multipass Die erste Kategorie stellen Multipass-Algorithmen, wie [Har81], dar. Der oben beschriebene grundlegende Algorithmus ist ebenfalls ein Vertreter dieser Kategorie. Weil die Anzahl der nötigen Durchläufe in der Regel datenabhängig ist, sind Multipass-Algorithmen normalerweise nicht optimal. In [WOS09] wird

[SHS03] als besonders schnelles Beispiel für einen Vertreter dieser Klasse eingestuft. Die Autoren von [SHS03] bezeichnen ihren Algorithmus als 'linear-time', evaluieren das jedoch nur experimentell.

Two-Pass Im Gegensatz zu Multipass-Algorithmen bestehen Two-Pass Connected-Component-Labeling-Algorithmen aus zwei verschiedenen Phasen der Datenverarbeitung. Algorithmen dieser Klasse bestehen typischerweise aus drei Phasen.

In der ersten Phase, die als Scanning-Phase bezeichnet wird, wird ein erstes Mal durch die Daten gegangen. Dabei wird die Gleichheit benachbarter Pixel analysiert und es werden provisorische Label vergeben. Zusätzlich wird die zugehörige Gleichheitsinformation aufgezeichnet.

In der zweiten Phase, die als Analysis-Phase bezeichnet wird, wird die Gleichheitsinformation der provisorischen Label analysiert, um finale Label zu bestimmen.

Es folgt als letztes die Labeling-Phase. In dieser wird ein zweites Mal durch alle Daten gegangen, um die finalen Label den korrespondierenden Pixeln zuzuweisen.

Eine effiziente und in der Praxis verbreitete Datenstruktur zur Verwaltung der Gleichheitsinformation ist die Union-Find-Datenstruktur, welche von Fiorio [FG96] vorgestellt wurde. Diese ermöglicht $O(N)$ Laufzeit Komplexität und ist somit optimal. In der Praxis wurden allerdings weitere Verbesserungen erreicht, wie z.B. durch [WOS09]. Ein weiteres Beispiel für eine Two-Pass-Technik, alternativ zur Union-Find-Datenstruktur, stellt die Connection-List [HCS08] dar.

One Pass Die letzte Kategorie stellen One-Pass-Algorithmen dar [HQN05] [UA90] [CC03] [CCL04], welche genau einmal durch das Bild gehen. Dabei wird allerdings oft durch die rekursive Natur dieser Techniken ein irreguläres Speicherzugriffsmuster verwendet.

Eine für diese Arbeit besonders interessante Teilmenge der One-Pass-Algorithmen, welche nur für zweidimensionale Rasterdaten anwendbar ist, stellen solche basierend auf Contour-Tracing [HA89] dar. Dazu zählen die Arbeit von Chang [CC03] und dessen Nachfolger mit linearer Laufzeit [CCL04]. Diese verarbeiten das Pixelraster von oben nach unten und von links nach rechts. Immer wenn dabei eine Kontur erkannt wird, kann ein neues Label generiert werden. Anschließend wird die ganze Kontur verfolgt und das gefundene Label allen Teilen der Kontur zugewiesen. Zuletzt erhalten alle Pixel im Inneren der Kontur deren Label mithilfe eines Scan-Line-Algorithmus. Falls unklassifizierte (oder anders klassifizierte) Bereiche vollständig von einer Connected-Component umschlossen sind, entstehen innere Konturen. Diese erhalten durch den Scan-Line-Algorithmus dasselbe Label wie die zugehörige äußere Kontur, welche aufgrund der sequentiellen Vorgehensweise immer als erstes gefunden wird. Somit kann der Scan-Line-Algorithmus den Bereichen zwischen innerer und äußerer Kontur das Label der Äußeren zuweisen und Bereiche innerhalb der Inneren werden nicht mit diesem Label versehen.

Ein weiteres Beispiel für einen One-Pass-Algorithmus stammt von De Bock und Philips [DBP10]. Hier werden einfach verlinkte Listen von Line-Segments verwen-

det. Die so verlinkten Line-Segments stellen Abschnitte in Zeilen dar und der Ansatz extrahiert auch keine Konturen.

Wie zuvor bereits beschrieben, gab es schon seit den 80er Jahren Bemühungen, Connected-Component-Labeling-Algorithmen, auch speziell in Hinblick auf 2D-Bilddaten, wie in [AP92] zusammenfasst, zu parallelisieren.

Diese, bis dahin oft primär theoretischen, Überlegungen wurden in den letzten Jahren im Zuge der Verwendung von programmierbaren Grafikprozessoren (GPUs) für allgemeine Berechnungen wieder Gegenstand neuer Untersuchungen in der Praxis. GPUs zeichnen sich im Vergleich mit CPUs durch hohe Leistung bei gleichzeitig hoher Energie- und Kosteneffizienz aus.

Beispielsweise haben Barnat et al. [BBBC11] Connected-Component-Labeling für Graphen mit Nvidias API Cuda [Nvi15a] implementiert. Die Autoren stellten dabei fest, dass diese Implementation, obwohl nicht optimal, auf einer GPU schneller ausgeführt werden kann als eine optimale sequentielle Implementation auf einer CPU.

Außerdem haben Hawick et al. [HLP10] neben einigen einfachen Label-Propagation-Ansätzen einen Label-Equivalence-Algorithmus mit Cuda implementiert. Sie stellten dabei fest, dass auf einer GPU z.B. bei direkter Label-Propagation zwischen unmittelbar benachbarten Pixeln die Laufzeit um mehrere Größenordnungen höher ausfallen kann als bei ihrem Label-Equivalence-Ansatz. Letzterer erreicht insgesamt für Rasterdaten im Vergleich mit optimierten CPU Fassungen leicht bessere Laufzeiten.

Oliveira und Lotufo [OL10] haben später den Label-Equivalence-Ansatz mit dem verwandten Union-Find-Algorithmus unter Verwendung von Cuda verglichen. Die Autoren stellten fest, dass der Union-Find-Ansatz gerade in schwierigen Datensätzen (z.B. einer Spirale) deutlich schneller ist und sich sonst oft vergleichbar verhält. Im Vergleich mit einem optimierten sequentiellen Ansatz erreicht dieser deutliche Speedups.

Diese Cuda-basierten Ansätze wurden in späteren Ansätzen weiter optimiert[KRKS11, OS11]. Vor allem Stava et al. [OS11] haben den Union-Find-Algorithmus deutlich für die Ausführung mit Cuda auf GPUs verbessert.

Bedingt durch die parallele Ausführung kann nicht garantiert werden, dass nach einmaliger Ausführung des ersten Pass je Connected-Component genau ein Equivalence-Tree entsteht. Somit erhalten nicht alle Pixel unbedingt ihr finales Label. Der erste Pass muss daher so oft wiederholt werden, bis die richtigen Label gefunden sind. Die Anzahl der notwendigen Durchläufe ist datenabhängig und kann somit nicht vorhergesagt werden. Beispielsweise wird im Falle der Implementation von Hawick [HLP10] dafür nach jedem Durchlauf des ersten Pass ein Test auf Korrektheit der Label durchgeführt. Die Ergebnisse werden dann zum Host übertragen, welcher ggf. eine weitere Iteration des ersten Pass initiiert. Stavas Implementation dagegen kann vollständig auf einer GPU ausgeführt werden und das Synchronisieren mit dem Host entfällt.

Außerdem führt Stava ein rekursives, Tile-basiertes Schema ein, um die Performance weiter zu verbessern. Dazu wird das Pixelgitter in kleine rechteckige Bereiche (Tiles) eingeteilt, für welche jeweils der CCL-Algorithmus unabhängig ausgeführt wird. Von den Tiles müssen anschließend lediglich die Ränder weiter verarbeitet werden. Diese werden danach rekursiv gemäß Nachbarschaft zu Gruppen zusammengefasst und vereinigt. Am

Ende müssen sich alle Pixel nur noch das finale Label holen. Diese Vorgehensweise ermöglicht zum einen den Datenaustausch über das sehr schnelle (aber kleine) Shared-Memory aktueller GPUs, wenn die Tiles und später Randgruppen klein genug gewählt werden. Zum anderen verringert sich so die Problemgröße sehr schnell.

Stava et al. [OS11] vergleichen ihren Ansatz experimentell mit Hawick [HLP10] auf einer neueren Nvidia GeForce GTX 480 GPU [Nvi15e] und können in allen getesteten Datensätzen deutliche Speedups erreichen.

Außer den Varianten für GPUs sind auch für andere moderne parallele Hardware wie den Cell Prozessor (z.B. [KPMS09]) oder FPGAs (z.B. [KBA+13]) Ansätze veröffentlicht worden. In dieser Arbeit liegt der Fokus jedoch auf GPUs und, mit Abstrichen, auf Mehrkern-CPUs, sodass diese nicht weiter besprochen werden.

Kapitel 4.

Zielsetzung und Motivation

Die Ausgangspunkte dieser Arbeit sind der parallele GPU-optimierte Ansatz von Stava et al. [OS11] einerseits und der optimale sequentielle Contour-Tracing basierte Ansatz von Chang et al. [CCL04] andererseits. Die Zielsetzung dieser Arbeit sieht vor, parallele und für GPUs geeignete Ansätze zu finden, welche im Gegensatz zu Stava auf Konturen basieren. Auf diese Weise sollen die Vorteile beider Ansätze vereinigt werden. Es werden mehrere Varianten vorgestellt, aber alle genügen folgendem Ablauf: Zunächst werden je Pixel die Kontursegmente unabhängig extrahiert. Die Konturen liegen dann in Form von gerichteten zyklischen Linked-Lists aus Segmenten vor. Auf dieser, im Vergleich mit dem Pixelgitter vereinfachten, Topologie kann anschließend das Connected-Component-Labeling-Problem gelöst werden. Zuletzt werden den Pixeln im Inneren der Konturen mit einem einfachen Füllverfahren die Label der Konturen übergeben.
Der Fokus auf GPUs ist motiviert durch die im Vergleich mit CPUs weitaus höhere Leistung einerseits, bei gleichzeitig höherer Energieeffizienz andererseits. Das gilt fast immer, wenn sich Probleme gut parallelisieren lassen. Ferner sind Anschaffungskosten aktueller GPUs ungleich geringer als im Falle vergleichbar leistungsfähiger CPUs.

Auch im Vergleich mit nicht programmierbarer Hardware verfügen GPUs über klare Vorteile. So ist Connected-Component-Labeling in der Praxis üblicherweise nur eine Komponente von vielen und die GPU kann, im Gegensatz zu fester Hardware, auch für andere Aufgaben verwendet werden. Beispielsweise wird Connected-Component-Labeling oft auf die Ergebnisse des Segmentation-Schritts angewandt, welcher sich ebenfalls gut auf einer GPU ausführen lässt. In diesem Fall liegen die Ergebnisse bereits im Speicher der GPU vor und es kann unmittelbar der Connected-Component-Labeling-Schritt angewandt werden. Außerdem sind GPUs, anders als feste Hardware, in einer großen Bandbreite verschiedener Zielgeräte aller möglicher Leistungsklassen, von Computing-Clustern bis hin zu Smartphones, enthalten. Das ist vor allem dann von Vorteil, wenn eine Anwendung, z.B. die eigene eingangs vorgestellte Smartphone-Wetter-App, mit der gegebenen Hardware auskommen muss.

Der konturbasierte Ansatz ist zum einen motiviert durch die zusätzlich extrahierten Konturinformationen, welche im Bereich Bildverarbeitung hilfreich sein können. Zum anderen wird untersucht, ob durch die vereinfachte Topologie verbesserte theoretische Eigenschaften des parallelen Algorithmus einerseits und ein besseres Laufzeitverhalten der Implementationen andererseits, erreicht werden kann.

Unabhängig davon identifiziert Stava experimentell großflächige Connected-Components als Worst-Case-Szenario für deren Ansatz. Bei konturbasierten Algorithmen dagegen werden die aufwändigeren Operationen allein auf die Randsegmente angewandt. Weil deren Anzahl gerade im Falle großflächiger Connected-Components im Verhältnis zur Pixelanzahl im Inneren klein ist, kann hier ein deutlich abweichendes Verhalten erwartet werden. Ein weiterer Gegenstand der Untersuchung ist daher die Betrachtung des Verhaltens von Datensätzen mit hohem und geringem Konturanteil im Vergleich. Gerade in letzterem Fall können bessere Laufzeiten in der Praxis erhofft werden.

Meines Wissens nach existieren in der Literatur keine Untersuchungen zu vergleichbaren, auf vorheriger Konturierung basierenden, Connected-Component-Labeling-Ansätzen speziell für GPUs.

Ein Nebenziel dieser Arbeit ist die Untersuchung einer speziellen CPU-Implementation eines parallelen konturbasierten CCL-Algorithmus mit OpenCL. Diese wird experimentell mit Chang verglichen. Die Fragestellung hierbei ist, ob auch im Falle geringer Parallelität Speedups möglich sind.

Kapitel 5.

Wissenschaftlicher Beitrag

Das Ziel, einen auf Konturierung basierenden optimalen parallelen Connected-Component-Labeling-Algorithmus für zweidimensionale Bilddaten zu finden, ist in dieser Arbeit erreicht worden. Allerdings hat sich nach dessen Vollendung herausgestellt, dass Ansätze mit vergleichbaren theoretischen Eigenschaften bereits zuvor publiziert worden sind. Ein eigenes Paper [WKV14], welches einen Zwischenstand der vorliegenden Arbeit enthält, ist noch in diesem Glauben formuliert worden. So gibt es meines Wissens nach, wie im Kapitel 3 zusammengefasst, zwar keine deterministischen optimalen parallelen Algorithmen für allgemeines Connected-Component-Labeling [CNP04]. Aber für den Spezialfall planarer Graphen, welcher zweidimensionale Rasterdaten einschließt, schon. Die im Rahmen dieser Arbeit gefundene Lösung ist unabhängig davon entstanden. Kapitel 14 liefert eine detailliertere Abgrenzung zu diesen vergleichbaren Ansätzen.

Davon unberührt bleiben die Ziele und Beiträge im Bereich Parallel-Computing.

Vor allem ist eine für GPUs gut geeignete Implementierung eines konturbasierten Connected-Component-Labeling-Algorithmus gefunden und experimentell evaluiert worden. Diese stellt im Vergleich mit Stava et al. [OS11] weniger Anforderungen an die Funktionalität einer GPU. So werden keine Atomic-Functions zum dynamischen Lösen von Race-Conflicts benötigt.

Im direkten Vergleich mit Stava auf jeweils identischen GPUs liefert der eigene Ansatz in der Mehrheit der evaluierten Kombinationen aus Datensätzen und GPUs Speedups um ca. Faktor zwei. Das schließt auch Datensätze mit hohem Konturanteil ein. Beim Vergleich auf einer aktuellen Nvidia GTX 980 ist der eigene Ansatz in allen evaluierten Datensätzen schneller. Der größte Speedup tritt, wie erwartet, bei maximal großen Connected-Components ein. Hier ist der eigene Ansatz bei Verwendung der GTX 980 um knapp Faktor vier schneller.

Auch ein experimenteller Vergleich mit dem sequentiellen Ansatz von Chang [CCL04] wird durchgeführt, da dieser ebenfalls Konturinformationen liefert. Dazu werden eine aktuelle GPU (für den Eigenen) und eine aktuelle CPU (für Chang) vergleichbarer Verlustleistung und Preisklasse ausgewählt. Hier ist die eigene Implementation immer, und zwar meistens um eine Größenordnung, schneller. Im Extremfall ist der eigene Ansatz um Faktor 49 schneller.

Zuletzt konnte ein für CPUs gut geeigneter Implementationsansatz eines konturbasierten Connected-Component-Labeling-Algorithmus gefunden werden. Dieser zeichnet sich dadurch aus, den konstanten Mehraufwand im Vergleich zu einem optimalen sequentiellen Ansatz möglichst gering zu halten. Bei Evaluation auf einer CPU mit acht Kernen ist dieser Ansatz in keinem evaluierten Datensatz langsamer als der Contour-Tracing-Algorithmus von Chang und in manchen Fällen um bis zu Faktor fünf schneller.

Kapitel 6.

Aufbau der Arbeit

Die Grobstruktur der vorliegenden Arbeit gliedert sich in die folgenden vier Teile:

Teil I - Prolog (dieser) beschreibt die Ausgangslage dieser Arbeit und nennt Ziele sowie Beiträge der durchgeführten Untersuchung. Außerdem werden einige der benötigten Grundlagen eingeführt.

Teil II - Parallele Algorithmen beschreibt und analysiert die im Rahmen dieser Arbeit gefundenen Algorithmen für konturbasiertes paralleles Connected-Component-Labeling. Diese Algorithmen beinhalten (teilweise abgewandelt) bekannte parallele Algorithmen, wie etwa List-Ranking. Diese werden zuvor eingeführt und analysiert. Abschließend findet ein Vergleich mit bisherigen parallelen konturbasierten Algorithmen statt.

Teil III - Paralleles Computing stellt zunächst verschiedene Ansätze vor, die in Teil II eingeführten Algorithmen mit OpenCL und Cuda zu implementieren. Das Laufzeitverhalten wird schrittweise für verschiedene Datensätze und Devices (GPUs und CPUs) experimentell analysiert und die Implementationen entsprechend angepasst. Dabei stellen sich solche Implementationen als in der Praxis schneller heraus, welche stärker von der Funktionsweise der Devices beeinflusst sind. So entsprechen die finalen (d.h. schnellsten) Fassungen in Teil III nicht mehr genau den in Teil II eingeführten Algorithmen und behalten insbesondere auch nicht deren theoretische Merkmale. Zuletzt werden die so gefundenen Implementationen mit bisherigen Connected-Component-Labeling-Ansätzen experimentell verglichen. Ein genauerer Überblick über die Struktur des Teils III findet sich in Abschnitt 15.

Teil IV - Epilog fasst die Ziele und die entsprechenden Ergebnisse dieser Arbeit zusammen. Die Ergebnisse werden abschließend beurteilt. Basierend auf Beobachtungen aus den Experimenten werden zuletzt Ansätze für weitere, teilweise nicht konturbasierte, Implementationen skizziert.

Teil II.

Parallele Algorithmen

Kapitel 7.

Grundlegende parallele Algorithmen

In diesem Kapitel werden einige bekannte und grundlegende parallele Algorithmen eingeführt und analysiert. Alle werden, wenn auch teilweise etwas abgewandelt, für die im Rahmen dieser Arbeit entstandenen parallelen Connected-Component-Labeling-Algorithmen benötigt.

7.1. Vektoraddition

Ein möglichst einfaches Beispiel für einen parallelen Algorithmus stellt die Vektoraddition dar. Weil diese komponentenweise unabhängig passiert, können alle Komponenten parallel verarbeitet werden. Es gibt dabei offensichtlich keinerlei Konflikte. Seien drei N-komponentige Vektoren a, b und c gegeben. Dann kann mit dem Algorithmus aus Pseudocodeabschnitt 1 die Vektoraddition ausgeführt werden. Diesem werden a und b als Parameter übergeben. Im Körper des Algorithmus werden alle N Komponenten von a und b mit Indices i, mit i $\in \{0, \ldots, N-1\}$ addiert und das Ergebnis jeweils an Position i in c gespeichert. Nach Abschluss der Berechnungen wird c zurückgeliefert. Die einzelnen Anweisungen in den `In Parallel Do` Abschnitten werden für alle Elemente parallel ausgeführt, aber Anweisung $i+1$ wird erst ausgeführt, wenn Anweisung i für alle Elemente abgeschlossen ist. In den Algorithmen nachfolgender Kapitel kann jetzt

Algorithm 1: VecAdd(a, b)

```
In: a, b
Out: c
Precondition: a.length = b.length = c.length = N
// For each element
For Each i ∈ {0, ..., N − 1} In Parallel Do
  | c(i) ← a(i) + b(i)
End
Return: c
```

die parallele Vektoraddition aufgerufen werden mit:

$$c \leftarrow \text{Execute VecAdd(a, b)}$$

In den nachfolgenden Pseudocodeabschnitten wird allerdings zumeist eine etwas einfachere Schreibweise verwendet, wie beispielhaft an der analogen Vektorsubtraktion in Algorithmus 2 zu sehen ist. Außerdem erfolgt die Schreibweise mit Übergabe der zu verarbeitenden Daten als Parameter nur bei solchen Algorithmen, die oft von anderen aufgerufen werden. Andernfalls wird angenommen, dass globale Daten verarbeitet werden.

Algorithm 2: VecSubtract(a, b)

```
For Each i ∈ {0, ..., N − 1} In Parallel Do
|  c(i) ← a(i) - b(i)
End
```

Asymptotische Analyse

Die Gesamtzahl der Operationen bei der parallelen Addition oder Subtraktion N-komponentiger Vektoren ist offensichtlich $O(N)$. Es können bis zu $O(N)$ Prozessoren verwendet werden, von denen jeder eine Komponente des Vektors addiert. Werden um einen Faktor k weniger Prozessoren eingesetzt, kann der Algorithmus so umgestellt werden, dass jeder Prozessor k Komponenten sequentiell verarbeitet. Das kann alternativ auch mit Brents Theorem[Bre74] gefolgert werden. Die Laufzeit ergibt sich damit zu $O(N/P)$ für jedes $P \leq N$. Insbesondere wird die Laufzeit bei $P = O(N)$ Prozessoren zu $O(1)$. Für die gesamten Kosten, welche sich aus dem Produkt der Laufzeit und der dafür nötigen Prozessorzahl berechnet, gilt immer: $O(N/P \cdot P) = O(N)$. Da jeder Prozessor lediglich auf einen getrennten Speicherbereich zugreift, genügt zur Ausführung das am wenigsten restriktive Modell, der EREW-PRAM.

7.2. Parallel-Prefix-Sums (Scan)

Ein weiterer und in vielen anderen Algorithmen verwendeter paralleler Algorithmus ist der Parallel-Prefix-Sums [LF80], oder kurz Scan. Sei x ein Array aus N Elementen und $\otimes$ ein darauf definierter assoziativer Operator. Ferner sei I das neutrale Element bezüglich $\otimes$. Dann berechnet der Scan folgendes N-komponentiges Array:

$$\{x_1, x_1 \otimes x_2, x_1 \otimes x_2 \otimes x_3, \ldots, x_1 \otimes x_2 \otimes \ldots \otimes x_N\}$$

Hier werden für jede Position i alle Elemente $\leq i$ mit $\otimes$ verknüpft. Diese Variante des Parallel-Prefix-Sums-Algorithmus wird auch Inclusive-Scan bezeichnet. Im Gegensatz dazu liefert der Exclusive-Scan:

$$\{I, x_1, x_1 \otimes x_2, \ldots, x_1 \otimes x_2 \otimes \ldots \otimes x_{N-1}\}$$

In dieser Arbeit ist x immer ein Array natürlicher Zahlen, $\otimes$ der Operator $+$ und I ist 0. Sei beispielsweise ein Array x gegeben mit:

$$x = [1, 1, 1, 1, 0, 4, 7, 8]$$

Dann ergibt der Inclusive-Scan über x:

$$[1, 2, 3, 4, 4, 8, 15, 23]$$

Dabei steht an der letzten Position immer die Summe über ganz x. Im Gegensatz dazu liefert der Exclusive-Scan bei Anwendung auf x:

$$[0, 1, 2, 3, 4, 4, 8, 15]$$

Die Summe über alle Elemente ist hier nicht direkt ablesbar, ist aber einfach durch die Summe der jeweils letzten Elemente beider Arrays zu berechnen.

Der Inclusive-Scan kann sequentiell mit dem in Pseudocode-Abschnitt 3 gegebenen Algorithmus berechnet werden. Dabei werden offensichtlich $O(N)$ Operationen ausge-

Algorithm 3: SequentialInclusiveScan()

```
Out: sums
sums(0) ← x(0)
For i ← 1; i < N; i ← i + 1 Do
 |  sums(i) ← sums(i - 1) + x(i)
End
```

führt. Insbesondere ist die Berechnung jedes $sums(i)$ von dem bereits berechneten Wert $sums(i-1)$ abhängig für jedes i außer $i = 0$. Dadurch erschwert sich die Parallelisierung im Vergleich mit der Vektoraddition.

Eine parallele Lösung geht auf Hillis und Steele [HS86] zurück und ist aufgrund ihrer Einfachheit in Pseudocode-Abschnitt 4 vorgestellt. In der hier gegebenen Formulierung werden die Ausgangsdaten überschrieben. Das Prinzip ist zusätzlich in Abbildung 7.1 anhand eines Beispiels veranschaulicht.

Algorithm 4: SimpleScan

```
For d ← 1; d ≤ log2(N); d ← d + 1 Do
    For Each i ∈ {0, ..., N − 1} In Parallel Do
        If i ≥ 2^d Then
            x(i) ← x(i − 2^(d−1)) + x(i)
        End
    End
End
```

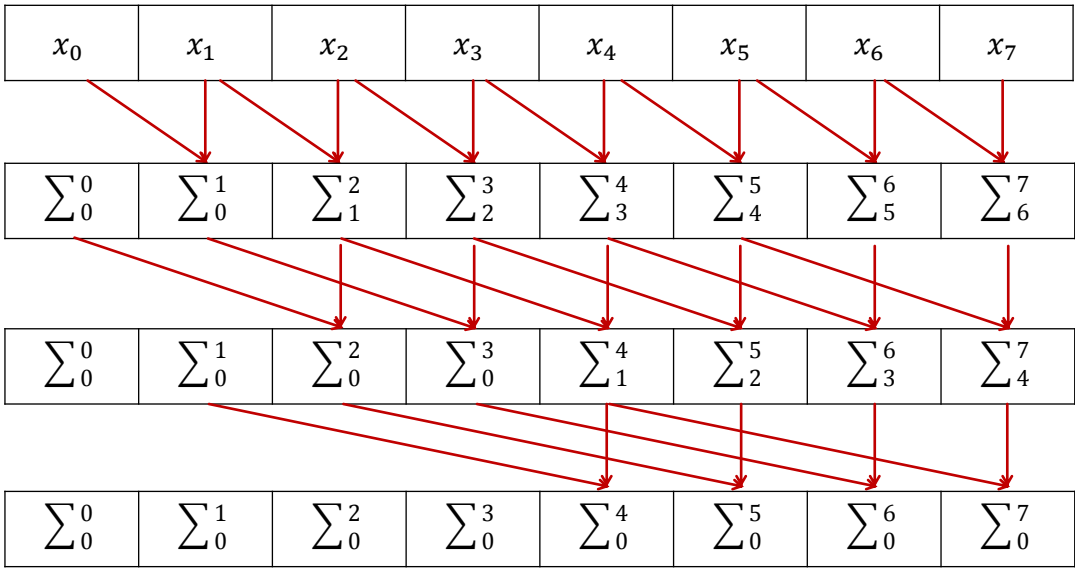


Abbildung 7.1.: Veranschaulichung des naiven parallelen Inclusive-Scans für ein Array mit N=8 Elementen. Gezeigt sind alle $\log_2(8) = 3$ Iterationen.

Asymptotisches Verhalten

Der Algorithmus von Hillis und Steele führt allerdings $O(N \cdot log_2(N))$ Operationen aus und damit mehr als in der sequentiellen Formulierung [HSO07], die mit $O(N)$ auskommt.

Aufgrund der fundamentalen Bedeutung des Scan-Algorithmus für viele parallele Algorithmen ist das Problem gut untersucht und es existieren asymptotisch bessere Lösungen sowie ausgereifte Implementationen.

Bereits der Algorithmus von Ladner und Fischer [LF80] berechnet den Scan auf einem EREW-PRAM in $O(N/P)$ Zeit, für jedes $P \leq N/log_2(N)$. Andere, wie der auf balancierten Bäumen basierende Algorithmus von Blelloch, haben ähnliche Eigenschaften [Ble90]. Einen Überblick, z.B. für andere parallele Modelle, gibt [LD94].

Auf einem CRCW-PRAM kann der Algorithmus für Ganzzahlen endlicher Größe modifiziert werden, um die Laufzeit $O(log_2(N)/log_2(log_2(N)))$ zu erreichen. Siehe dazu die Beschreibung des Algorithmus von Cole und Vishkin [CV89].

Anwendung in dieser Arbeit

In nachfolgende Algorithmen werden Varianten des Scans für ein Array `vals` aus Ganzzahlen folgendermaßen aufgerufen:

$$\begin{aligned} \text{iscannedVals} &\leftarrow \textbf{execute } \text{parScan\_Inclusive(vals)} \\ \text{escannedVals} &\leftarrow \textbf{execute } \text{parScan\_Exclusive(vals)} \end{aligned}$$

Dabei werden die Eingangsdaten nicht überschrieben und das Ergebnis des Scan-Algorithmus als Array zurückgeliefert. Die Wahl des internen Algorithmus hängt von den gewünschten asymptotischen Eigenschaften ab und wird dann jeweils im entsprechenden Abschnitt diskutiert. Diese Entscheidung hat aber keinen Einfluss auf das Ergebnis.

7.3. Radix-Sort

Radix-Sort ist ein stabiles Sortierverfahren für Ganzzahlen, welche sich durch eine feste Anzahl, k, Bits kodieren lassen. Es lässt sich leicht basierend auf dem Scan-Algorithmus des vorherigen Abschnitts formulieren.

Ein Schritt des Algorithmus wendet den Scan auf das i-te Bit aller Werte an. Anschließend werden alle Werte gemäß den Adressen des so erhaltenen Array umkopiert. Dieser Schritt wird nacheinander für alle k-Bits, und zwar vom Least-Significant-Bit zum Most-Significant-Bit, wiederholt.

Für Ganzzahlen endlicher Größe ist k eine Konstante (z.B. 32 oder 64 für die Datentypen `int` und `long` aus Java). Somit sind die asymptotischen Eigenschaften mit denen des verwendeten Scan-Algorithmus identisch.

Für diese Arbeit wichtig ist, dass es sich um ein stabiles Sortierverfahren handelt, d.h. die Reihenfolge von Einträgen gleichen Wertes bleibt erhalten.

7.4. Compact

Der Compact, oder Stream-Compact, Algorithmus überführt Einträge eines Datenarrays `sparse`, die eine bestimmte Bedingung erfüllen, lückenlos in ein Array `dense` (siehe z.B. [BOA09]). Eine typische Anwendung ist die Reduzierung einer dünnbesetzten Datenstruktur auf eine Dichtbesetzte. Weil diese dann weniger Elemente enthält, kann Letztere nachfolgend möglicherweise effizienter weiterverarbeitet werden.

Sei zusätzlich zu dem Array `sparse` ein Array gleicher Größe mit 1 (für Eintrag in `sparse` mit gleichem Index ist interessant) und 0 (für Eintrag uninteressant) gegeben. Dann kann der Algorithmus Compact, wie in Pseudocodeabschnitt 5 gegeben, die neuen Positionen der interessanten Elemente mit einem Exclusive-Scan berechnen und diese anschließend verschieben. Die Ausgangsdaten werden dabei nicht überschrieben. Die

Algorithm 5: parCompact(sparse, newIndex, interesting)

```
newIndex ← Execute parScan_Exclusive(interesting)
For Each i ∈ {0, ..., N − 1} In Parallel Do
    // Copy only interesting elements
    If interesting(i) = 1 Then
        dense(newIndex(i)) ← sparse(i)
    End
End
Return: dense
```

reine Verschiebeoperation ist offensichtlich auf einem EREW-PRAM in $O(N/P)$ Zeit ausführbar für jede Prozessorzahl P mit $P \leq N$. Folglich stellt der verwendete Scan-Algorithmus den limitierenden Schritt im Compact-Algorithmus dar, sodass die asymptotischen Eigenschaften mit denen der eingesetzten Scan-Variante identisch sind.

In nachfolgende Algorithmen wird der Compact-Algorithmus für ein Array aus Elementen `sparse` und einem Array `interesting` mit Nullen und Einsen folgendermaßen aufgerufen:

dense ← **execute** parCompact(sparse, newIndex, interesting)

interesting	1	0	0	1	0	1	1	0
newIndex	0	1	1	1	2	2	3	4
sparse	A	B	C	D	E	F	G	H
dense	A	D	F	G	-	-	-	-

Abbildung 7.2.: Veranschaulichung des Compact-Algorithmus. Eingangsdaten: `interesting`, `sparse`. Berechnung v. `newIndex` mit Exclusive-Scan über `interesting`. Ausgangsdaten: `dense`

7.5. Basic-List-Ranking

Sei eine einfach verkettete, gerichtete Linked-List bestehend aus N Knoten gegeben. Dabei verweist jeder Knoten i, mit Ausnahme des Letzten, auf den nachfolgenden Knoten $suc(i)$. Der letzte Knoten, i_{last} hat keinen Verweis auf einen Nachfolger, sodass gilt: $suc(i_{last}) = null$.

List-Ranking-Algorithmen berechnen für solch eine Datenstruktur den Rang eines Knotens in Relation zu ihrem ersten (oder letzten) Element. Dann erhält relativ zum letzten Knoten der erste Knoten den Rank N-1, der Zweite den Rank N-2, usw., und der Letzte den Rank 0. List-Ranking kann als Prefix-Sum über eine Linked-List angesehen werden, bei der die zu summierenden Werte Einsen sind. Sequentielles List-Ranking kann offensichtlich durch In-Order-Traversierung der Liste in $O(N)$ Schritten berechnet werden.

Ein grundlegender Algorithmus für paralleles List-Ranking wurde durch Wyllie vorgestellt [Wyl79]. Solch ein Algorithmus wird im Folgenden als Basic-List-Ranking bezeichnet und ist im Pseudocodeabschnitt 6 gegeben. Bei den Eingangsdaten handelt es sich um eine Linked-List `L` bestehend aus `N` Knoten mit dem schon beschriebenen Nachfolger `suc` und zusätzlich je einem Wert `val` gemäß:

$$L = (val(i), suc(i)), 1 \leq i \leq N$$
$$\text{mit } val(i) = 1 \; \forall i, \text{ außer } val(i_{last}) = 0$$

Bei jeder Ausführung der Operation in der While-Schleife, passieren zwei Dinge. Zum

Algorithm 6: Basic-List-Ranking

```
For Each i ∈ {1, ..., N} In Parallel Do
    // If not end of list reached
    While suc(i) ≠ null Do
        // Do Pointer Jump Operation
        val(i) ← val(i) + val(suc(i))
        suc(i) ← suc(suc(i))
    End
End
```

einen erhält jeder Knoten i als Wert die Summe aus seinem vorherigen Wert und dem Wert des nachfolgende Knotens. Zum anderen wird der Verweis auf den Nachfolger ($suc(i)$) auf den Nachfolger des Nachfolgers ($suc(suc(i))$) gesetzt. Dieses wird nachfolgend als Pointer-Jump-Operation bezeichnet.

Anfänglich verweist der Wert `suc` jedes Knotens auf den unmittelbaren Nachfolger. Nach der ersten Pointer-Jump-Operation verweist `suc` jedes Knotens auf einen Knoten in Entfernung 2 in der Linked-List, sofern ausreichend weit vom Ende der Liste entfernt. Bei jedem weiteren Pointer-Jump wird die Reichweite erneut verdoppelt, sodass sie nach $\lceil log_2(N) \rceil$ Iterationen `N` ist. Damit ist auch vom ersten Knoten aus Information über die gesamte Liste gesammelt worden.

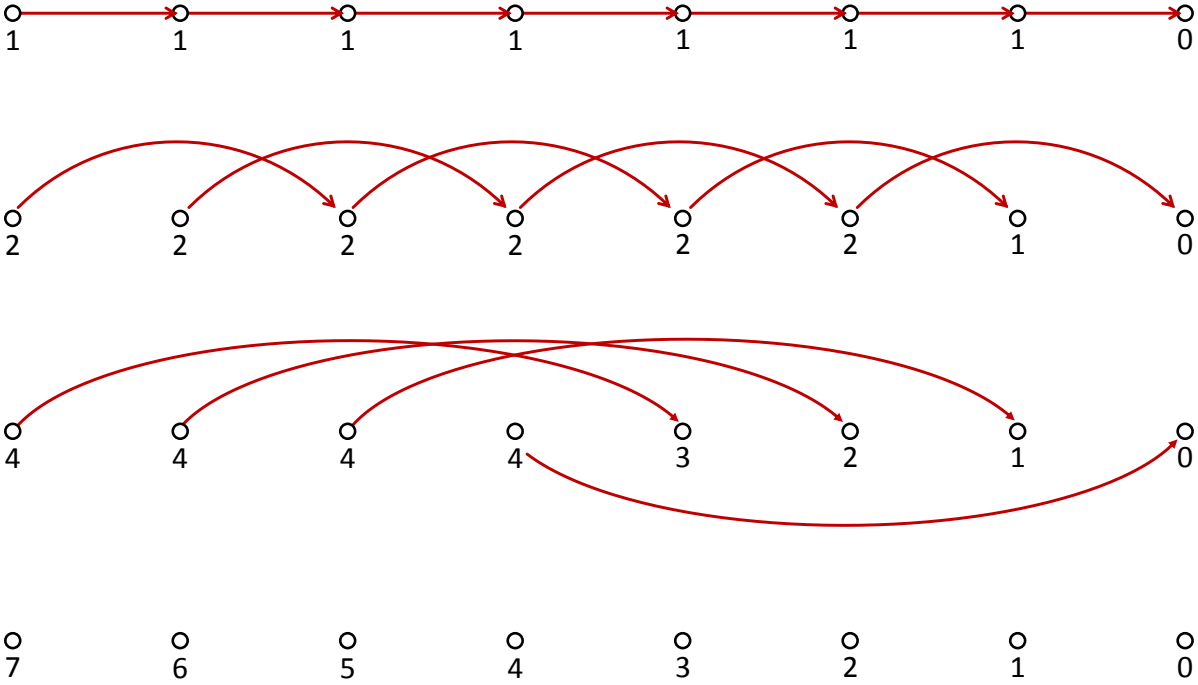


Abbildung 7.3.: Veranschaulichung des parallelen Basic-List-Ranking-Algorithmus mittels Pointer-Jump-Operations für eine Liste mit N=8 Elementen. Gezeigt sind alle $\log_2(8) = 3$ Iterationen. Pfeile geben Nachfolger (`suc`) an und Zahlen den Wert von `val`. Wenn `suc(i) = null` für einen Knoten `i` gilt, wird kein Pfeil gezeichnet, und `i` bleibt fortan inaktiv.

Es existieren unterschiedliche Formulierungen des Algorithmus 6 in der Literatur, vor allem das Abbruchkriterium (am Listenende) betreffend. Alternative Formulierungen rufen z.B. einfach die Pointer-Jump-Operation $\lceil log_2(N) \rceil$ Mal auf und lassen *suc* für alle Knoten auf das letzte Element zeigen.

Dagegen werden bei der im Pseudocode zu Algorithmus 6 gegebenen Variante alle Verweise auf Nachfolger am Ende auf `null` gesetzt und durch die Bedingung in der While-Schleife wird sichergestellt, dass solche Knoten inaktiv bleiben. Dies ist z.B. in [LD94] sowie [MB13] so beschrieben. Dadurch wird zum einen sichergestellt, dass kein Wert von mehreren Prozessoren gleichzeitig gelesen wird, wodurch die Ausführung auf einem EREW-PRAM ermöglicht wird. Zum anderen wird der Wert des letzten Knotens nicht mehrfach aufaddiert, sodass der Algorithmus gültig bleibt, wenn $val(i_{last}) \neq 0$ gilt, was später noch benötigt wird.

Der Basic-List-Ranking-Algorithmus mit dem Pointer-Jump-Prinzip ist beispielhaft in Abbildung 7.3 verdeutlicht.

Abschließend lauten die Eigenschaften des Basic-List-Ranking-Algorithmus unter Verwendung des Pointer-Jump-Prinzips [LD94]:

$$\begin{aligned} \textbf{Machine} &: \text{EREW-PRAM} \\ \textbf{Processors} &: O(N) \\ \textbf{Running-Time} &: O(N \cdot \log_2(N)/P) \\ \textbf{Cost} &: O(N \cdot \log_2(N)) \end{aligned}$$

In diesem Abschnitt wurde ein einfacher, aber offensichtlich nicht optimaler, paralleler List-Ranking-Algorithmus vorgestellt. Gegenstand der verbleibenden Abschnitte des Kapitels 7 wird optimales paralleles List-Ranking sein. Weil dort verwendete Techniken die Grundlage für zentrale Komponenten des Connected-Component-Labeling-Algorithmus bilden, muss dieses vergleichsweise detailliert geschehen.

7.6. Eine Strategie für optimales List-Ranking und r-Ruling-Sets

Gegeben sei ein zusammenhängender, gerichteter Graph G(V, E), bei dem der In-Degree und Out-Degree jedes Knotens genau 1 ist. Ein solcher Graph wird auch als Ring bezeichnet. Sei N die Zahl der Knoten von G. Die Linked-List, für die der List-Ranking-Algorithmus ausgeführt werden soll, sei bis auf weiteres ein eben solcher zyklischer Graph. Eine Strategie für optimales List-Ranking, die in von Cole und Viskin [CV86b] [CV89] verwendet wird, geht in drei Schritten vor:

1. Finde eine möglichst große unabhängige Teilmenge U von Knoten. Dabei bedeutet unabhängig, dass Vorgänger und Nachfolger jedes Knotens aus U nicht in U enthalten sein dürfen. Nur auf diese Knoten wird dann parallel die Pointer-Jump-Operation angewendet und die übersprungenen Knoten werden anschließend aus dem Graphen entfernt. Dies bezeichnen wir nachfolgend als Shortcut-Operation. Schritt (1) wird so oft wiederholt, bis höchstens $N/log_2(N)$ Knoten verbleiben.

2. Anschließend wird der Basic-List-Ranking-Algorithmus auf eben diese verbleibenden Knoten angewandt. Dazu werden $\mathtt{P} = N/log_2(N)$ Prozessoren verwendet, sodass der Schritt nun lediglich O(N) Kosten verursacht und $O(log_2(N))$ Zeitschritte dauert.

3. Zuletzt wird das Entfernen der Knoten aus Schritt (1) in genau umgekehrter Weise rückgängig gemacht und dabei die Ranks aller Knoten berechnet.

Das größte Problem wird das (wiederholte) Finden einer möglichst großen Menge U aus unabhängigen Knoten in Schritt (1) darstellen. Schön wäre es, jeden zweiten Knoten identifizieren zu können. In einem eindimensionalen Array ist so etwas typischerweise in O(1) Zeit möglich, indem jede zweite Position eines linearen Index (z.B. die Speicherposition) gewählt wird. Dies ist bei Linked-Lists, deren Knoten beliebig im Speicher liegen, insbesondere nicht möglich. Lägen die Knoten garantiert linear im Speicher wäre die Linked-List-Datenstruktur auch überflüssig. Stattdessen ist (näherungsweise) jeder zweite Knoten in Bezug auf den Verlauf der Linked-List gesucht.

Ein bekannter Begriff aus der Graphentheorie, der hilfreich beim Finden einer möglichst großen Menge U in Schritt (1) sein könnte, ist die Graphenfärbung. Hier käme die Dreifärbung in Frage, da die Zweifärbung, zumindest bei Ringen ungerader Knotenzahl, undefiniert ist (siehe Abbildung 7.4).

7.6.1. Definition r-Ruling-Set

Anstelle einer n-Färbung wird allerdings von Cole und Vishkin ein r-Ruling-Set verwendet, welcher aber letztendlich ähnliches wie ein n-Coloring leistet. Die nachfolgend vorgestellte Definition des r-Ruling Set stammt aus [CV86b]. Gegeben sei wieder ein zusammenhängender, gerichteter Graph G(V, E), bei dem der In-Degree und Out-Degree

Abbildung 7.4.: Färbung von zyklischen Graphen. Links: Ungerade Knotenzahl, bestenfalls Dreifärbung möglich. Rechts: Bei gerader Knotenzahl ist auch Zweifärbung möglich.

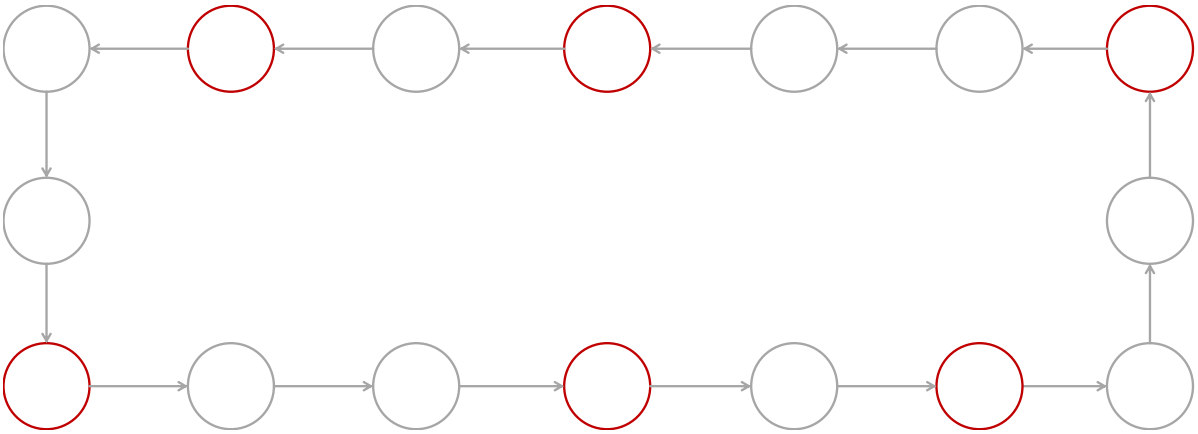


Abbildung 7.5.: Beispiel für einen gerichteten zyklischen Graphen und einen 2-Ruling-Set U als Teilmenge davon. Die Knoten aus U sind rot gefärbt.

jedes Knotens genau 1 ist. Sei N die Zahl der Knoten von G. Eine Teilmenge U von V ist ein r-Ruling-Set von G, falls:

1. Keine zwei Knoten von U sind benachbart
2. Für jeden Knoten v in V existiert ein gerichteter Pfad von v zu einem Knoten in U, dessen Länge höchstens r ist.

Durch die erste Bedingung wird U als unabhängige Teilmenge festgelegt und die zweite Bedingung sagt etwas über die relativen Größenverhältnisse der Knotenmengen aus. Da eine möglichst große Teilmenge U gesucht wird, ist entsprechend der 2-Ruling-Set von besonderem Interesse. In einem 2-Ruling-Set sind mindestens 1/3 und höchstens 1/2 aller Knoten aus V enthalten. Abbildung 7.5 zeigt beispielhaft einen Ring und einen 2-Ruling-Set.

7.7. Berechnung eines r-Ruling-Sets

Die im Folgenden beschriebenen Techniken, von Cole und Vishkin 'Deterministic Coin Tossing' bezeichnet, können die -scheinbare- Symmetrie einer zyklischen Linked-List brechen. Es wird sich zeigen, dass, basierend auf einem eindeutigen Index der Knoten, deterministisch Knoten ausgewählt werden können, die einen r-Ruling-Set bilden. Es werden demnach -scheinbar- gleiche Knoten unterschiedlich behandelt, um die Symmetrie zu brechen. Da diese, hier deterministische, Vorgehensweise sonst im Bereich randomisierter paralleler Algorithmen üblich ist, wurde die Technik 'Deterministic Coin Tossing' genannt.

Nachfolgend werden verschiedene Möglichkeiten beschrieben, angelehnt an [CV86b] und [CV89], einen r-Ruling-Set mittels der Deterministic-Coin-Tossing-Technik zu finden. Besonders interessant ist ein 2-Ruling-Set, da bei diesem die größte Menge unabhängiger Knoten garantiert ist.

7.7.1. Basisschritt: Finden eines $log_2(N)$-ruling-Sets

Zunächst beschreiben Cole und Vishkin in [CV86b], wie ein $log_2(N)$-Ruling Set in $O(1)$ Zeit gefunden werden kann, wenn N die Knotenzahl ist.

Ausgangsdatenformat

Dazu mögen die Knoten in einem Array der Länge N abgelegt sein und jeder Knoten verfüge über einen Index $0, 1, \ldots, N-1$, der Arrayposition entsprechend. Die Indices seien repräsentiert durch eine Folge von Bits der Länge $\lceil log_2(n) \rceil$. Diese Bits verfügen über einen eigenen Index zwischen 0 und $\lceil log_2(n) \rceil - 1$. Das *Least-Significant-Bit* (LSB) befinde sich ganz rechts und habe den Index 0. Analog erhält das Bit ganz links den Index $\lceil log_2(n) \rceil - 1$.

Abgesehen vom eigenen Index verfügt jeder Knoten über einen Verweis (`suc`) auf den nächsten Knoten (in Form von dessen Index) und einen Verweis auf den Vorgänger (`pre`).

Der Wert $serial_k$ und dessen Berechnung

Weiterhin ist für jeden Knoten i ein Wert $serial_k(i)$ definiert, welcher anfänglich (für $k = 0$) mit dem eigenen Index, i, identisch ist. Später wird für jeden Knoten ein neuer Wert, $serial_1(i)$, berechnet. Dieser entspricht dem minimalen Index, in welchem sich $serial_0(i)$ und der entsprechende Wert des nachfolgenden Knotens, $serial_0(i.suc)$, unterscheiden.

Die im Pseudocode Abschnitt calcSerial gegebene Funktion berechnet allgemein $serial_k$ für einen Knoten i aus den vorherigen Werten $serial_{k-1}$ von i und dessen Nachfolger $suc(i)$.

Sei beispielsweise i (= $serial_0(i)$) ...01**0**101 und $suc(i)$ (= $serial_0(suc(i))$) sei ...11**1**101. Dann ist der kleinste Index (von rechts) für den sich die Bits unterscheiden 3. Folglich ist $serial_1(i)$ 3.

Function calcSerial(k, i)

If $k = 0$ **Then**
  $serial_k(i) \leftarrow$ i
Else
  // Determine lowest bit-index,
  // where bits in $serial_{k-1}(i)$ and $serial_{k-1}(suc(i))$ differ
  $serial_k(i) \leftarrow min_{Bitj} \{serial_{k-1}(i).j \neq serial_{k-1}(suc(i)).j\}$
End

Fakt 1: Für alle i ist $serial_1(i)$ eine Zahl zwischen 0 und $log_2(N)-1$ und $\lceil log_2(log_2(n)) \rceil$ Bits genügen zu deren Repräsentation. Aus Gründen der Einfachheit sei diese nachfolgend als ganzzahlig angenommen.

Lokale Extrema

Der Wert $serial_k(i)$ ist definiert als ein lokales Minimum (LMIN), wenn gilt:

$$serial_k(pre(i)) \geq serial_k(i) \leq serial_k(suc(i))$$

Analog liegt für $serial_k(i)$ ein lokales Maximum (LMAX) vor, wenn gilt:

$$serial_k(pre(i)) \leq serial_k(i) \geq serial_k(suc(i))$$

Fakt 2: Die Anzahl der Knoten in dem kürzesten Pfad ausgehend von irgendeinem Knoten in G zu dem nächsten Extremum (LMAX oder LMIN) hinsichtlich $serial_0$ ist $log_2(N)$.

Die Alternierungs-Eigenschaft

Allerdings können aufgrund der Bedingung $\leq$ bzw. $\geq$ mehrere lokale Extrema direkt aufeinander folgen, sodass diese nicht die Bedingung (1) für die Selektion in den r-Ruling-Set U erfüllen. Dieses Problem kann durch die Alternierungs-Eigenschaft behoben werden:

- Falls Bit $serial_1(i)$ von $serial_0(i)$ 0 ist, muss Bit $serial_1(i)$ in $serial_0(suc(i))$ 1 sein.
- Falls Bit $serial_1(i)$ von $serial_0(i)$ 1 ist, muss Bit $serial_1(i)$ in $serial_0(suc(i))$ 0 sein.

Fakt 3: Sei $i_1, i_2, i_3, \ldots$ eine Chain in G, sodass $serial_1(i)$ ein lokales Minimum für alle i in der Chain ist (oder für alle ein lokales Maximum). Dann ist für alle Knoten in der Chain $serial_1(i)$ gleich, also $serial_1(i_1) = serial_1(i_2) = serial_1(i_3) = \ldots$. Dies folgt aus der Definition der lokalen Minima.

Fakt 4: Betrachtet wird weiterhin obige Chain. Dann ist folgende Folge von Bits eine alternierende Sequenz aus Nullen und Einsen:

1. Bit mit Index $serial_1(i_1)$ von $serial_0(i_1)$
2. Bit mit Index $serial_1(i_2)(= serial_1(i_1))$ von $serial_0(i_2)$
3. Bit mit Index $serial_1(i_3)(= serial_1(i_1))$ von $serial_0(i_3)$
4. ...

Das folgt aus der Alternierungseigenschaft.

Auswählen lokaler Minima für U

Nun wird eine Teilmenge der Knoten ausgewählt, die im r-Ruling-Set U sein sollen. Dies wird für einen Knoten i nachfolgend formuliert als: i ist (oder wird) `selected`.

Alle Knoten i werden `selected`, wenn sie ein lokales Minimum darstellen und mindestens eine der folgenden Bedingungen erfüllt ist:

1. Weder $pre(i)$ noch $suc(i)$ sind lokale Minima
2. Das Bit mit Index $serial_1(i)$ ist 1

Damit wird jedes alleinstehende lokale Minimum `selected` und in einer Chain lokaler Minima wird jedes zweite `selected`.

Auswählen lokaler Maxima für U

Anschließend kann ein Knoten i als `available` (verfügbar um ggf. ausgewählt zu werden) eingestuft werden, wenn gilt:

1. i ist nicht `selected`
2. i ist ein lokales Maximum
3. Weder $pre(i)$ noch $suc(i)$ sind `selected`

Dann können zusätzlich alle als `available` eingestuften Knoten i `selected` werden, für die mindestens eine der folgenden Bedingungen gilt:

1. Weder $pre(i)$ noch $suc(i)$ sind `available`
2. Bit mit Index $serial_1(i)$ ist 1

Damit wird jedes alleinstehende lokale Maximum `selected`, falls keiner seiner Nachbarn zuvor (als Minimum) `selected` wurde. Zusätzlich wird in einer Chain lokaler Maxima, deren Nachbarn nicht schon zuvor `selected` wurden, jeder zweite Knoten `selected`.

Ergebnis und Algorithmus

Die insgesamt für U ausgewählten Knoten stellen einen $log_2(N)$-Ruling-Set dar. Bedingung (1) ist erfüllt, da keine zwei benachbarten Knoten `selected` sind. Die Erfüllung der Bedingung (2) ergibt sich, da alle lokalen Extrema entweder `selected` sind oder einer ihrer Nachbarn `selected` ist, aus Fakt 2.

Der Algorithmus zum Berechnen des $log_2(N)$-Ruling-Set U ist in Pseudocodeabschnitt 8 gegeben und die vorherige Initialisierung erfolgt gemäß Pseudocodeabschnitt 7.

Algorithm 7: Prepare Basic Step

```
For Each Vertex i, i ∈ {0, ..., N − 1} In Parallel Do
    selected(i) ← FALSE
    available(i) ← FALSE
    deleted(i) ← FALSE
    Call calcSerial(0, i)
End
```

Offensichtlich kann damit ein $log_2(N)$-Ruling-Set in $O(1)$ Zeit mit N Prozessoren berechnet werden, von denen jeder einen Vertex verarbeitet. Dieser Algorithmus wird nachfolgend als 'Basisschritt' bezeichnet.

7.7.2. Die k-te Anwendung des Basisschritts

In diesem Abschnitt wird beschrieben wie, durch wiederholte Anwendung des im vorherigen Abschnitt beschriebenen Basisschritts, ein 2-Ruling-Set gefunden werden kann.

Vorbereitung des k-ten Schritts

Vor Ausführung des k-ten Schritts werden aus G die Knoten entfernt, die in der Ausführung des (k-1)-ten Schritts als `selected` markiert sind oder einen Nachbarn haben, der `selected` ist. Solche Knoten werden als `deleted` bezeichnet. Als `deleted` markierte Knoten werden in den folgenden Schritten nicht mehr betrachtet. Zusätzlich werden alle Kanten zu diesen Knoten entfernt.

Außerdem wird ab dem 2-ten Schritt ein Vektor `degree` für alle Knoten definiert, der die Anzahl der ein- und ausgehenden Kanten angibt. Die Initialisierung ist im Pseudocode Abschnitt 9 gegeben.

Fakt 5: Seien der Knoten i und dessen Nachfolger $suc(i)$ der Eingangsdaten für den k-ten Schritt gegeben. Dann gilt: $serial_{k-1}(i) \neq serial_{k-1}(suc(i))$. Für $k = 1$ ist das offensichtlich, da die $serial_0$-Werte dem ursprünglichen eindeutigen Index entsprechen. Bleibt der Fall $k > 1$ zu überprüfen. Angenommen, $serial_{k-1}(i)$ und $serial_{k-1}(suc(i))$ seien gleich. Dann wären i und $suc(i)$ in Schritt $k - 1$ lokale Minima oder Maxima gewesen. Aber jedes lokale Extremum ist entweder `selected` worden, oder hat einen Nachbarn, der `selected` worden ist. Dies wiederum bedeutet, dass es entfernt worden wäre. Folglich gibt es den Fall $serial_{k-1}(i) = serial_{k-1}(suc(i))$ nicht.

Algorithm 8: Basic Step

```
// k ≥ 1
For Each Vertex i, i ∈ {0, ..., N − 1} In Parallel Do
    // Init data for current k
    Call calcSerial(k, i)
    odd(i) ← (Bit serial_k(i) of serial_{k−1}(i) = 1)
    lMin(i) ← serial_k(pre(i)) ≥ serial_k(i) ≤ serial_k(suc(i))
    lMax(i) ← serial_k(pre(i)) ≤ serial_k(i) ≥ serial_k(suc(i))
    // Consider local min
    If lMin(i) Then
        If ¬ (lMin(pre(i)) ∨ lMin(suc(i))) ∨ odd(i) Then
            selected(i) ← TRUE
        End
    End
    // Consider local max
    If lMax(i) Then
        // Set available if neighbours not selected
        If ¬ (selected(i) ∨ selected(pre(i)) ∨ selected(suc(i))) Then
            available(i) ← TRUE
        End
        // Select (possibly) if available
        If available(i) Then
            If ¬ (available(pre(i)) ∨ available(suc(i))) ∨ odd(i) Then
                selected(i) ← TRUE
            End
        End
    End
End
```

Algorithm 9: Step K

```
For Each Vertex i, i ∈ {0, ..., N − 1} In Parallel Do
    // Delete selected nodes and their neighbors
    If (selected(i) ∨ selected(pre(i)) ∨ selected(suc(i))) Then
        deleted(i) ← TRUE
    Else
        // Initialize degree
        degree(i) ← 0
        If ¬deleted(suc(i)) Then
            degree(i) ← degree(i) + 1
        End
        If ¬deleted(pre(i)) Then
            degree(i) ← degree(i) + 1
        End
    End
    If ¬deleted(i) Then
        If degree(i) = 0 Then
            selected(i) ← TRUE
        Else If degree(i) = 1 Then
            // Neighbor is suc(i) or neighbor has degree 2
            If deleted(pre(i)) ∨ degree(pre(i)) = 2 Then
                selected(i) ← TRUE
            End
        Else If degree(i) = 2 Then
            Call calcSerial(k, i)
            If degree(pre(i)) = 2 ∧ degree(suc(i)) = 2 Then
                // Apply Basic Step (algorithm 8) on i
                Execute Basic Step for i
            End
        End
    End
End
```

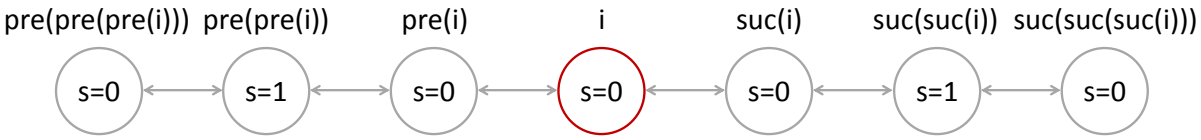


Abbildung 7.6.: Bekannte Umgebung eines Knotens `i`, wenn gilt: `i` ist nicht `deleted` und hat `degree` 0. Legende: (s)elected $\in$ {0: FALSE, 1: TRUE}, `deleted` $\in$ {rot: FALSE, grau: TRUE}. i kann `selected` werden

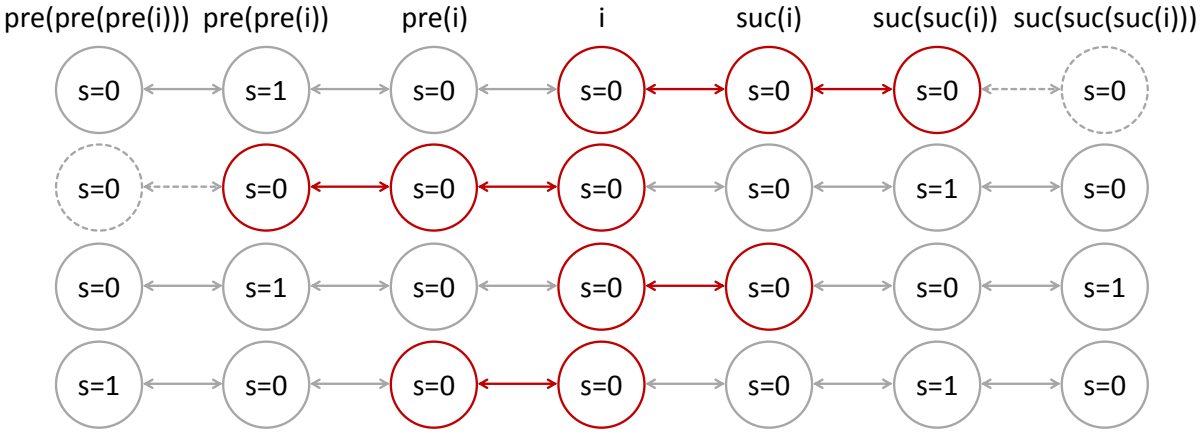


Abbildung 7.7.: Umgebungen eines Knotens `i`, wenn gilt: `i` ist nicht `deleted` und hat `degree` 1. Legende: (s)elected $\in$ {0: FALSE, 1: TRUE}, `deleted` $\in$ {rot: FALSE, grau: TRUE, gestrichelt: Unklar}. Fälle 1/2 von oben: Nachbar hat `degree` 2, folglich kann `i selected` werden. Fall 3 von oben: Nachbar hat `degree` 1, und ist suc(i), also kann `i selected` werden. Letzter Fall: Nachbar hat `degree` 1, aber ist nicht suc(i), also kann `i` nicht `selected` werden.

Ausführung des k-ten Schritts

Mit diesen Vorbedingungen kann nun der k-te Schritt ausgeführt werden. Die verbleibenden (nicht `deleted` markierten) Knoten i werden je nach `degree(i)` anders behandelt.

Knoten mit `degree` 0 können direkt `selected` werden. Schließlich sind dessen Nachbarn im Schritt $k-1$ entfernt und nicht selected, da jeder `selectete` Knoten mit seinen Nachbarn (die nicht `selected` sind) entfernt wird. Generell gilt: An den 'Rändern' kann immer, zumindest etwas, direkt `selected` werden. Diese Situation ist in Abbildung 7.6 veranschaulicht.

Knoten mit `degree` 1 liegen an einem Rand. Hier muss überprüft werden, ob der Nachbar des Knotens i ebenfalls am Rand liegt, da nicht beide `selected` werden dürfen. Falls `degree` des Nachbarn 2 ist, liegt nur i am Rand und kann folglich `selected` werden. Andernfalls liegen beide am Rand und Knoten i wird ausgewählt, wenn er der erste ist ($\Leftrightarrow$ $suc(i)$ ist sein Nachbar). Diese Situationen sind in Abbildung 7.7 veranschaulicht.

Solche mit `degree` 2, deren Nachbarn ebenfalls `degree` 2 haben, können wie im Basisschritt verarbeitet werden. Schließlich liegen die gleichen Bedingungen vor (vergleiche Fakt 5). Hat ein Nachbar den `degree` 1, passiert nichts, da dieser `selected` wird (s. o.).

Ergebnis

Fakt 6: In dem Graphen, der nach entfernen der Knoten (und Kanten) nach 2 Schritten gemäß beschriebener Regeln übrig bleibt, bestehen die längsten möglichen 'einfachen Pfade' aus $log_2(log_2(N))$ Knoten. Folglich kann ein $log_2(log_2(N))$-Ruling-Set in $O(1)$ Zeit mit $O(N)$ Prozessoren gefunden werden. Analog bestehen nach k Schritten die längsten möglichen 'einfachen Pfade' aus folgender Knotenzahl:

$$\underbrace{log_2(log_2(log_2(log_2(\ \ldots\ log_2}_{k}(N)))))$$

Auch hier sei wieder angenommen, dass es sich dabei um eine Ganzzahl handelt.

Weiterhin gelten folgende Beobachtungen [CV86b, CV89]:

- Nach $log_2^*(N)$ Schritten sind alle Knoten `deleted`. Dabei ist $log_2^*(N)$, der iterierte Logarithmus (gesprochen Log Star), hier zur Basis 2, folgendermaßen definiert:

$$log_2^*(n) = \begin{cases} \text{falls } n \leq 1 : 0 \\ \text{falls } n > 1 : 1 + log_2^*(log_2(n)) \end{cases} \tag{7.1}$$

- Die Knoten, welche dann `selected` sind, bilden einen 2-Ruling-Set
- Das Verfahren ist ebenso (und mit gleichen Eigenschaften) auf einen gerichteten Pfad von N Knoten anwendbar.
- Die Kardinalität eines 2-Ruling-Sets in einem Ring ist mindestens (N/3)

Insgesamt ergeben sich damit folgende Ergebnisse [CV86b, CV89]:

- Ein $log_2^k(N)$-Ruling Set kann mit $O(N)$ Prozessoren in $O(k)$ Zeit gefunden werden.
- Folglich kann z.B. ein $log_2(log_2(N))$-Ruling-Set mit $O(N)$ Prozessoren in $O(1)$ Zeit gefunden werden.
- Ein 2-Ruling-Set kann mit $O(N)$ Prozessoren in $log_2^*(N)$ Zeit gefunden werden.

Das Finden eines 2-Ruling-Sets ist nicht cost-optimal, wie leicht verifiziert werden kann, die Berechnung eines $log_2^k(N)$-Ruling-Sets aber schon.

7.7.3. Ein optimaler 2-Ruling-Set-Algorithmus

In diesem Abschnitt wird erläutert, wie mit dem in [CV89] vorgestellten optimalen Algorithmus ein 2-Ruling-Set gefunden werden kann. Dieser ist in zwei Phasen eingeteilt:

Phase I: Finden eines $\frac{log_2(N)}{log_2(log_2(N))}$-Ruling-Sets. Wie im vorherigen Abschnitt beschrieben liefert die zweimalige Anwendung des Basisschritts einen $log_2(log_2(N))$-Ruling-Set. Dieser ist offensichtlich auch ein $\frac{log_2(N)}{log_2(log_2(N))}$-Ruling-Set. Danach werden nur noch Knoten betrachtet, die noch nicht in Phase I `selected` wurden. Wie beschrieben, hat jetzt jeder Knoten i einen Wert $serial_2(i)$, für den gilt: $0 \leq serial_2(i) < log_2(log_2(N))$ und der von den Werten seiner Nachbarn verschieden ist.

Phase II: Mit den beschriebenen Voraussetzungen liefert der in Pseudocode Abschnitt 10 gegebene Algorithmus einen 2-Ruling-Set, indem dem $\frac{log_2(N)}{log_2(log_2(N))}$-Ruling-Set weitere Knoten hinzugefügt werden. Darin geht die äußere Schleife (k) sequentiell über alle

Algorithm 10: PhaseII (naiv)

```
For Each k, k ∈ {0, 1, 2, ..., log2(N)/log2(log2(N))} Do
    For Each Vertex i, for which serial2(i) = k In Parallel Do
        If ¬selected(i) ∧ ¬selected(pre(i)) ∧ ¬selected(suc(i)) Then
            selected(i) ← TRUE
        End
    End
End
```

möglichen Werte von $serial_2$. Parallel werden dann Knoten i, deren Wert von $serial_2(i)$ gerade k entspricht, überprüft, ob sie oder ihre Nachbarn bereits `selected` sind. Falls nicht, wird i `selected`. Da die Nachbarn immer andere Werte für $serial_2$ haben, können sie nicht gleichzeitig `selected` werden. Auf diese Weise werden alle Knoten, die nicht selbst oder einer ihrer Nachbarn, bereits im $\frac{log_2(N)}{log_2(log_2(N))}$-Ruling-Set enthalten waren, `selected`.

Allerdings ist Algorithmus 10 offensichtlich nicht optimal. Deshalb schlagen Cole und Vishkin vor, zunächst einen Bucket-Sort-Algorithmus, welcher im selben Paper beschrieben ist, auf den Vektor $serial_2$ anzuwenden. Dieser hat eine Laufzeit von $O(\frac{log_2(N)}{log_2(log_2(N))})$ bei $n \cdot \frac{log_2(log_2(N))}{log_2(N)}$ Prozessoren. Alternativ kann der Radix-Sort-Algorithmus aus Kapitel 7.3 verwendet werden. Das Ergebnis, der Vektor `rank`, enthält für jeden Knoten einen eindeutigen Wert zwischen 1 und N. Anschließend wird für jeden Knoten i berechnet:

$$rank(i) \leftarrow rank(i) + serial_2(i) \cdot n \cdot \frac{log_2(log_2(N))}{log_2(N)} \tag{7.2}$$

Damit enthält der Vektor `rank` nun eindeutige Werte zwischen 1 und $2 \cdot N$, gemäß derer die Knoten in einen Vektor `sorted` umkopiert werden können. Insbesondere ist `sorted`

dann in Bereiche aus jeweils $n \cdot \frac{log_2(log_2(N))}{log_2(N)}$ Positionen i eingeteilt, die keine verschiedenen Werte für $serial_2$ haben können. Dementsprechend können je $n \cdot \frac{log_2(log_2(N))}{log_2(N)}$ konsekutive Positionen parallel verarbeitet werden, sodass nach $O(\frac{log_2(N)}{log_2(log_2(N))})$ Zeit ein 2-Ruling-Set bestimmt ist. Der nun optimale Algorithmus zum Finden eines 2-Ruling-Sets ist in Pseudocode Abschnitt 11 gegeben. Damit kann ein 2-Ruling-Set auf einem CRCW-

Algorithm 11: Compute 2-ruling Set

```
// Phase I
Execute PrepareBasicStep
Execute BasicStep
Execute StepK
// Phase II
rank ← Execute BucketSort(serial_2)
For Each Vertex i, i ∈ {1, 2, ..., N} In Parallel Do
    rank(i) ← rank(i) + serial_2(i) · N · log_2(log_2(N)) / log_2(N)
End
Rearange Data according to rank
For Each j, j ∈ {1, 2, ..., 2 · log_2(N) / log_2(log_2(N))} Do
    For Each i, with (j − 1) · n · log_2(log_2(N)) / log_2(N) ≤ i ≤ j · n · log_2(log_2(N)) / log_2(N) In Parallel Do
        If ¬selected(i) ∧ ¬selected(pre(i)) ∧ ¬selected(suc(i)) Then
            selected(i) ← TRUE
        End
    End
End
```

PRAM mit $n \cdot \frac{log_2(log_2(N))}{log_2(N)}$ Prozessoren in $O(\frac{log_2(N)}{log_2(log_2(N))})$ Zeit berechnet werden.

7.8. Optimales List-Ranking auf einem CRCW-PRAM

In diesem Abschnitt wird erläutert, wie der optimale parallele List-Ranking-Algorithmus von Cole und Vishkin aus [CV89] funktioniert. Dazu werden die Daten mithilfe der 2-Ruling-Sets wiederholt ausgedünnt, bis höchstens $N/log_2(N)$ repräsentative Knoten verbleiben. Auf diese wird anschließend der Basic-List-Ranking-Algorithmus (siehe Seite 40) angewendet, welcher dann $O(N)$ Operationen ausführt. Zuletzt wird das Ausdünnen rückgängig gemacht und dabei der Rank für die übrigen Elemente bestimmt. Der Algorithmus ist in Pseudocode Abschnitt 12 gegeben und kann am besten mit dem nachfolgenden Text zusammen gelesen werden. Die Globalen Arrays `val`, `suc` entsprechen denen des Basic-List-Ranking-Algorithmus, die übrigen vektoriellen Daten sind temporär. Ferner gibt das Skalar `m` die Anzahl der verbleibenden Knoten an und wird mit `N`, der Problemgröße, initialisiert. Außerdem wird durch das Skalar `t` die fortschreitende Zeit mitgezählt und der letzte Zeitschritt, in dem etwas passiert, wird in `T` vermerkt. Die

Algorithm 12: Optimal Parallel List Ranking

```
m ← N // Remaining Nodes, at first: N
t ← 0
While m ≥ N/log2(N) Do
    // Step I
    For Each Vertex v, v ∈ {0, 1, ..., m − 1} In Parallel Do
        serial0(v) ← v
    End
    // Step II
    ruling ← Execute Compute2RSet(serial0, suc) // See p. 11
    // Step III
    For Each Processor p, p ∈ {1, 2, ..., P} In Parallel Do
        For Each v, with: (p − 1) · m/P ≤ v ≤ p · m/P − 1 Do
            t ← t + 1
            If ruling(v) = 1 Then
                save(p, t) ← (suc(v), v, val(v))
                val(v) ← val(v) + val(suc(v))
                suc(v) ← suc(suc(v))
            End
            t ← t + 1
            If ruling(v) = 1 ∧ ruling(suc(v)) = 0 Then
                save(p, t) ← (suc(v), v, val(v))
                val(v) ← val(v) + val(suc(v))
                suc(v) ← suc(suc(v))
            End
        End
    End
    // Step IV
    newId ← Execute parScan_Exclusive(ruling)
    Do: Compact Vertex-Data in same vectors according to newId and ruling
    m ← ruling(m-1) + newId(m-1)
End
T ← t
// Step V
Execute BasicListRanking(val, suc, _N ← m) // See p. 6
// Step VI
For Each Processor p, p ∈ {1, 2, ..., P} In Parallel Do
    For Each t, with: t ← T, decreasing to 1 Do
        If save(p, t) is defined Then
            val(save(p, t).suc) ← val(save(p, t).v) - save(p, t).val
        End
    End
End
```

eingesetzte Prozessorzahl ist P. Der Algorithmus gliedert sich in sechs Schritte (Steps), von denen Steps eins bis vier durch eine While-Schleife umgeben sind. Die Schleife wird solange aufgerufen, bis die Daten ausreichend ausgedünnt sind. Die Schritte im Einzelnen:

Step I: Initialisierung

Hier werden temporäre Arrays für die nächste Iteration der While-Schleife zurückgesetzt (nicht im Pseudocode gezeigt). Außerdem wird das Array $serial_0$ für die Bestimmung des 2-Ruling-Sets der verbleibenden `m` Knoten initialisiert. Dies ist im Pseudocode für Algorithmus 12 im Abschnitt 'Step 1' zu sehen.

Step II: Bestimme 2-Ruling-Set

In diesem Schritt wird für die verbleibenden `m` Elemente ein (neuer) 2-Ruling-Set berechnet und im Vektor `ruling` abgelegt. Für Knoten v, die zu dem 2-Ruling-Set gehören gilt: $ruling(v) = 1$ (entspricht TRUE). Andernfalls gilt dann: $ruling(v) = 0$ (entspricht FALSE). Siehe dazu auch 'Step 2' im Pseudocode für Algorithmus 12.

Step III: Shortcut-Operations

Auf jedes Element v des aktuellen 2-Ruling-Sets (f.d.g: $ruling(v) = 1$) folgen ein oder zwei Knoten i mit $ruling(i) = 0$. Letztere werden in diesem Schritt, ausgehend von dem ersten Knoten des 2-Ruling-Sets davor, parallel 'übersprungen'. Der verbleibende Graph beinhaltet nun genau die Knoten des 2-Ruling-Sets und das sind höchstens $m/2$. Die übrigen Knoten werden zusammen mit den zugehörigen Kanten entfernt.

Zusätzlich wird vor jeder Shortcut-Operation der Zustand des betreffenden Knotens zu diesem Zeitpunkt gesichert. Dazu sei folgendes Struct eingeführt, welches den Knotenindex v, den Verweis auf den Nachfolger suc(v) sowie des aktuellen Wert val(v) abspeichern kann:

```
Struct saveEntry { Integer suc; Integer v; Integer val; };
```

Diese Einträge werden in einem Array `save` mit einem zweidimensionalen Index `p` und `t` abgelegt. Dabei gibt `t` den Zeitpunkt der Ausführung der betreffenden Shortcut-Operation an und wird vor jeder (potentiellen) Shortcut-Operation erhöht. Der Index p gibt den Prozessor an, welcher eben diese Shortcut-Operation ausführen wird. Auf diese Weise kann später dieser Zustand wiederhergestellt und damit das Entfernen der Knoten exakt invertiert, sowie der Rank für alle Knoten korrekt bestimmt werden. Dies ist im Pseudocode für Algorithmus 12 im Abschnitt 'Step 3' zu sehen und (etwas vereinfacht) anhand eines Beispiels in Abbildung 7.8 dargestellt.

Step IV: Compact Node-Array

Im vierten Schritt wird die Anzahl der Knoten des 2-Ruling-Sets (und damit das neue `m`) bestimmt und in eine dichtbesetzte Datenstruktur überführt. Diese und der neue

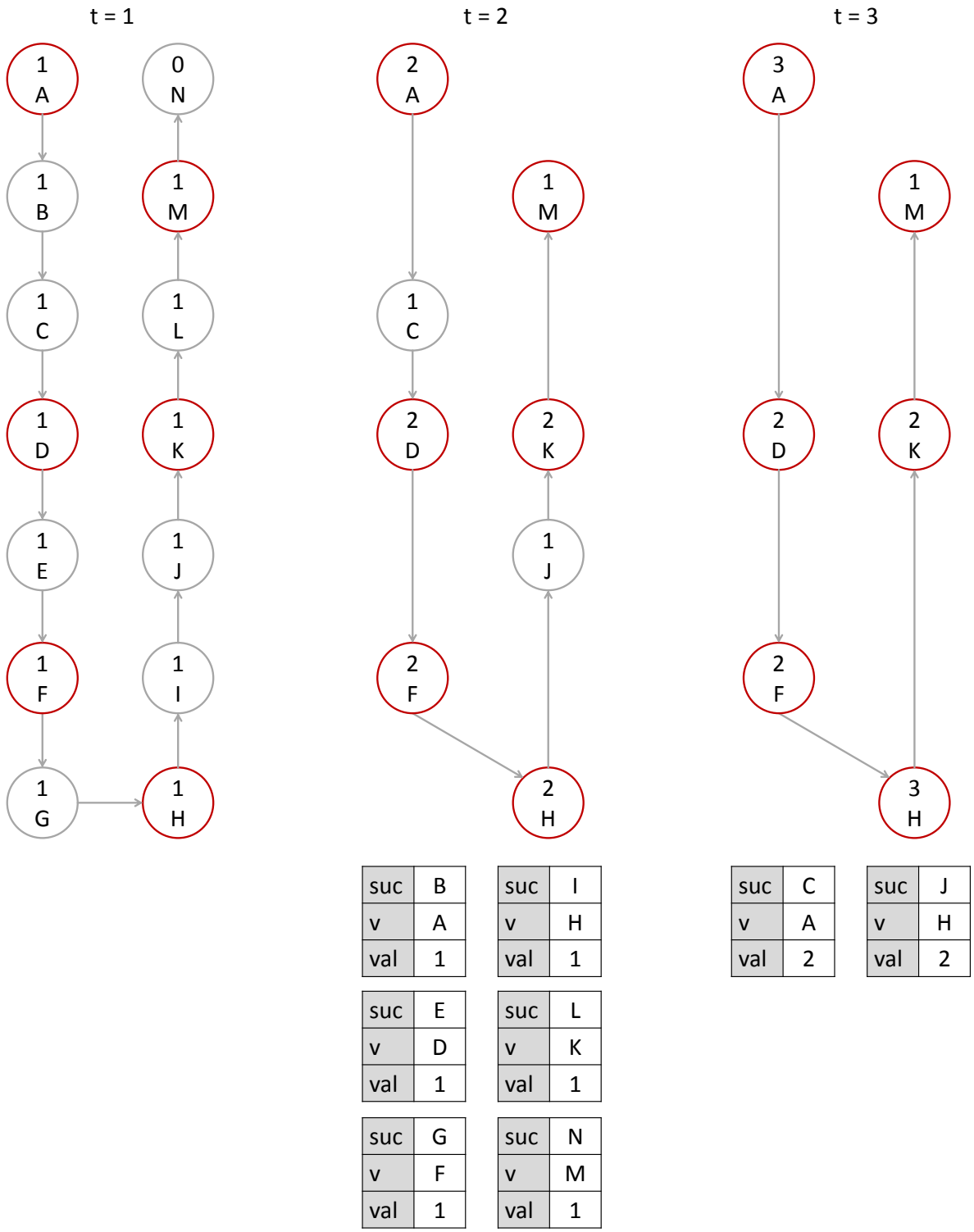


Abbildung 7.8.: Vereinfachtes Beispiel für eine Iteration des Step 3 aus Algorithmus 12. Gegeben ist, zu Zeit $t = 1$, ein gerichteter Pfad und ein 2-Ruling-Set (rote Knoten). Buchstaben in Knoten geben den Knotenindex v an und Zahlen den (aktuellen) Rank $val(v)$. Zeit $t = 2$: Erste Shortcut-Operation ausgeführt. Entstandene Einträge in `save` für diesen Zeitschritt sind als Kästen darunter abgebildet. Zeit $t = 3$: Zweite Shortcut-Operation dieser Iteration ausgeführt und weitere Einträge für `save` zu diesem Zeitpunkt erzeugt.

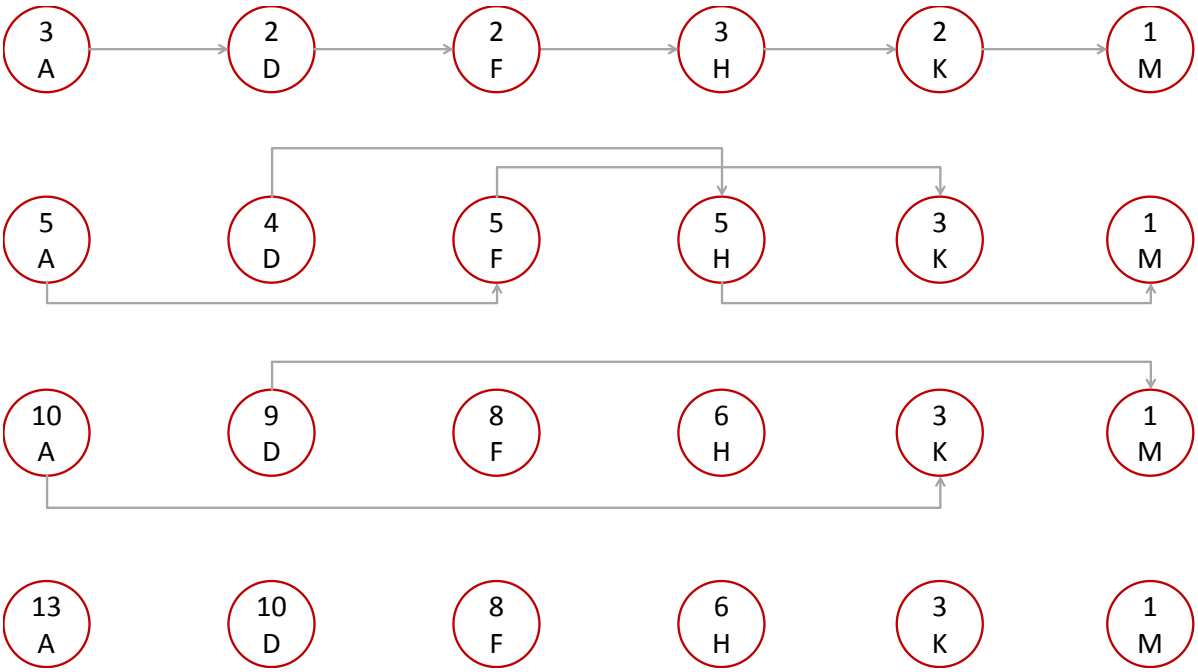


Abbildung 7.9.: Step 5 (Basic-List-Ranking) aus Algorithmus 12, angewandt auf die verbleibenden Knoten des Beispiels aus Abb. 7.8. Es handelt sich bei der Beispielserie nicht um ein vollständiges Beispiel des Algorithmus 12. So ist die verbleibende Knotenmenge noch nicht klein genug. Steps 1, 2, 4 (alle n. gezeigt) und Step 3 aus Abb. 7.8 sind öfter zu wiederholen.

Wert von `m` werden dann in der nächsten Iteration der While-Schleife verarbeitet, sofern noch erforderlich. Dazu muss ein Scan über den Vektor `ruling` ausgeführt werden. Das Umkopieren erfolgt im gleichen Array. Dies ist im Abschnitt Step IV des Pseudocode für Algorithmus 12, in vereinfachter Form, gezeigt. Hinsichtlich der Vereinfachungen sei noch auf Abschnitt 'Anmerkungen zu Vereinfachungen in Steps IV und VI' auf Seite 59 hingewiesen.

Step V: Basic-List-Ranking für Knotenauswahl

Wenn `m`, die Anzahl der verbleibenden Knoten, kleiner gleich $N/log_2(N)$ ist, wird die Steps eins bis vier umgebende While-Schleife verlassen. Anschließend wird in Step 5 der Basic-List-Ranking-Algorithmus auf eben diese Knoten angewandt. Dazu werden `P` $= N/log_2(N)$ Prozessoren verwendet, sodass der Schritt nun lediglich $O(N)$ Kosten verursacht. Damit sind für Knoten dieser Teilmenge des Eingangsgraphen die Ranks bekannt. In Abbildung 7.9, einer Fortsetzung der Beispielabbildung 7.8, ist dieser Schritt zu sehen.

Step VI: Rankberechnung für restliche Knoten

Auf Basis der in Step V bestimmten Ranks können nun zusammen mit den im Vektor `save` hinterlegten Informationen die Ranks aller Knoten bestimmt werden. Dazu wird das

Ausdünnen des Graphen in der Steps 1-4 umfassenden While-Schleife unter Benutzung des Zeitzählers `t` exakt invers rückgängig gemacht.

Sei U die Menge der Knoten des 2-Ruling-Sets einer Iteration der While-Schleife. Wir betrachten nun einen Knoten `v1` aus U. Dessen Rank (`val(v1)`) ist jetzt auf jeden Fall bekannt. Ferner sei von einem Prozessor `p` zu einem Zeitpunkt `t` eine Shortcut-Operation auf `v1` angewandt worden. Durch diese ist der vor Zeitpunkt `t` aktuelle Nachfolger von `v1` 'übersprungen' worden. Dann muss im Array `save` ein Eintrag mit den Indices `p` und `t` existieren, welcher diese Situation vor der Shortcut-Operation gesichert hat. Der zuvor `v1` genannte Knoten ist `save(p, t).v` und dessen aktueller Rank entsprechend `val(save(p, t).v)`. Der zum Zeitpunkt `t` aktuelle Nachfolger von Knoten `save(p, t).v` ist `save(p, t).suc` und sein Rank zum Zeitpunkt `t` ist `save(p, t).val`. Letzterer Wert gibt gerade die Anzahl der Knoten an, die übersprungen werden mussten, damit `save(p, t).suc` zum Zeitpunkt t Nachfolger von `save(p, t).v` sein konnte. Dementsprechend ergibt sich der endgültige Rank des übersprungenen Nachfolgers `save(p, t).suc` aus der Differenz des endgültigen Ranks von `save(p, t).v` und dessen zum Zeitpunkt `t` aktuellen Rank:

```
val(save(p, t).suc) <- val(save(p, t).v) - save(p, t).val
```

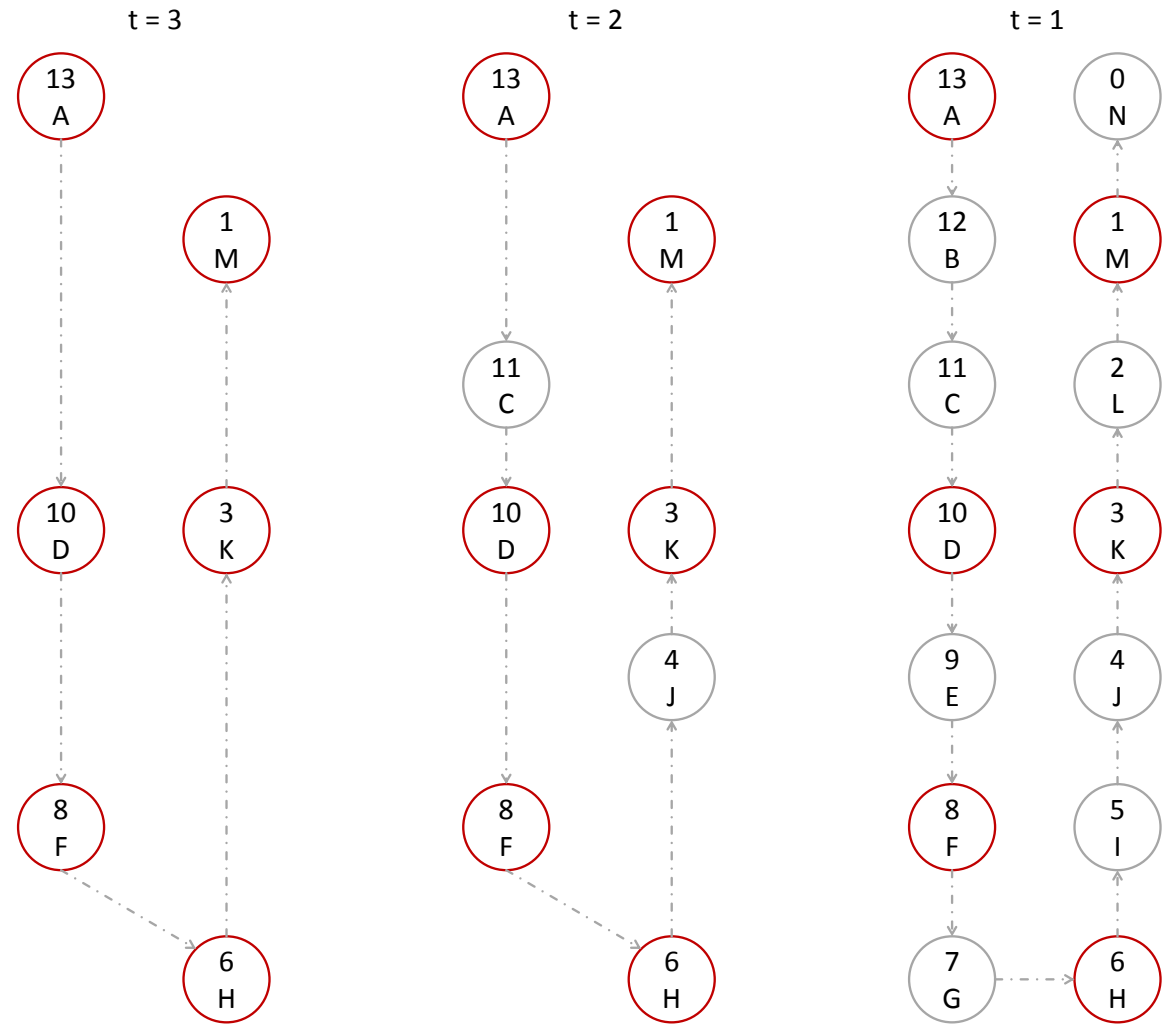
Mit der zuvor erläuterten Anweisung lassen sich für die zu einem Zeitpunkt `t` übersprungenen Nachfolger von Knoten aus U die Ranks bestimmen. Dabei wurde angenommen, dass der finale Rank der Knoten aus U bekannt ist. Dies ist immer der Fall, da das Ausdünnen der Linked-List exakt invertiert wird. Die Knoten des nach der letzten Iteration der While-Schleife verbleibenden 2-Ruling-Sets bekommen durch den Basic-List-Ranking-Algorithmus einen Rank. Dementsprechend können so den Knoten, welche in dem letzten Durchlauf der While-Schleife entfernt wurden, Ranks zugewiesen werden. Folglich haben alle Knoten, welche im zweitletzten Durchlauf der While-Schleife einen 2-Ruling-Set gebildet haben, einen finalen Rank, usw.

Die Rankberechnung für die restlichen Knoten ist im Abschnitt 'Step VI' des Pseudocodes für Algorithmus 12 gegeben und beispielhaft in Abbildung 7.10 dargestellt. Letztere führt die Beispielserie aus Abbildungen 7.8 und 7.9 fort.

Anmerkungen zu Vereinfachungen in Steps IV und VI

Der im Pseudocode Abschnitt 'Algorithmus 12' gegebene Algorithmus ist in einiger Hinsicht vereinfacht dargestellt. So sind in Step IV beim Umkopieren der Nodes im gleichen Array weitere temporäre Datenstrukturen nötig und auch die übersprungenen Nodes müssen irgendwo abgespeichert werden.

Insbesondere werden durch das Umkopieren auch die Adressen der Nodes geändert, sodass Verweise auf Nachfolger, gerade solche aus dem Array `save`, ungültig werden. Entweder nimmt man an, diese passen sich auf *magische Weise* immer wieder an, oder man müsste sich, spätestens bei der Implementation, eine andere Lösung überlegen. Eine Möglichkeit ist, bei Ausführung von Step 6 vor jedem Zeitschritt die Nodes wieder an ihre damaligen Positionen zurück zu kopieren, sodass die Referenzen wieder gelten. Diese Positionen müssten dann entsprechend vor Ausführung einer Shortcut-Operation



suc	C
v	A
val	2

suc	J
v	H
val	2

suc	B
v	A
val	1

suc	I
v	H
val	1

suc	E
v	D
val	1

suc	L
v	K
val	1

suc	G
v	F
val	1

suc	N
v	M
val	1

Abbildung 7.10.: Beispiel für eine Iteration des Step VI aus Algorithmus 12 und Fortführung der Beispielserie aus Abbildungen 7.8 und 7.9. Zum Zeitpunkt $t = 3$ liegen die Knoten mit den (endgültigen) Ranks vor, die in Abb. 7.9 berechnet wurden. Außerdem liegen die in Abb. 7.8 für diesen Zeitschritt angelegten Einträge in `save` vor. Werden diese gemäß der Vorschrift von Step VI integriert, erhält man den Zustand zu Zeitpunkt $t = 2$. Abermalige Integration der `save`-Einträge von Zeitpunkt $t = 2$ liefert schließlich den Zustand zu Zeitpunkt $t = 1$. Damit haben alle Knoten den endgültigen Rank. Hinweis: Verweise auf Nachfolger eigentlich nicht mehr explizit gegeben.

irgendwo vermerkt worden sein. So ist auch in der Implementation für diese Arbeit verfahren worden.

Das ist in 'Algorithmus 12' nicht aufgenommen worden, um dieses Kapitel mit der Beschreibung aus Cole und Vishkins Paper [CV89] konsistent zu halten.

7.8.1. Asymptotische Analyse

Zunächst werden, bei ausreichend verfügbaren Prozessoren, die Gesamtzahl der Operationen und die Laufzeit ermittelt. Für eine Iteration der While-Schleife in Algorithmus 12, aufgeschlüsselt nach den darin enthaltenen Steps eins bis vier, gilt offensichtlich:

- Step I benötigt $O(m)$ Operationen und $O(1)$ Zeit
- Step II benötigt $O(m)$ Operationen und $O(\frac{log_2(m)}{log_2(log_2(m))})$ Zeit
- Step III benötigt $O(m)$ Operationen und $O(1)$ Zeit
- Step IV benötigt $O(m)$ Operationen und $O(\frac{log_2(m)}{log_2(log_2(m))})$ Zeit

Damit benötigt jede Iteration der While-Schleife $O(m)$ Operationen und $O(\frac{log_2(m)}{log_2(log_2(m))})$ Zeit. Dabei wird m, die Anzahl der verbleibenden Knoten, nach jeder Iteration der While-Schleife mindestens halbiert. Dementsprechend verbleiben nach spätestens $O(log_2(log_2(N))$ Iterationen der While-Schleife höchstens die geforderten $O(N/log_2(N))$ Knoten. Folglich ergeben sich als Summe über alle Iterationen der While-Schleife $O(N)$ Operationen und $O(log_2(N))$ Zeit. Für die restlichen Schritte gilt:

- Step V benötigt $O(N)$ Operationen und $O(log_2(N))$ Zeit
- Step VI benötigt die gleiche Anzahl Operationen und Zeit wie alle Iterationen von Step III, da er diesen praktisch invertiert.

Insgesamt ergeben sich damit $O(N)$ Operationen und $O(log_2(N))$ Zeit für den Algorithmus. Unter Anwendung von Brents Theorem zeigen Cole und Vishkin noch, dass in Abhängigkeit von der Prozessorzahl P zusammengefasst gilt [CV89]:

Machine : CRCW-PRAM
Max Processors : $O(N/log_2(N))$
Running-Time : $O(N/P)$
Running-Time(pMax) : $O(log_2(N))$
Cost : $O(N)$

Kapitel 8.

Daten, Layout und verwendete Konstanten

Wir beginnen, vor Formulierung des Connected-Component-Labeling-Algorithmus in den nächsten Kapiteln, mit einigen Überlegungen hinsichtlich der zu verarbeitenden Daten. Die Eingangsdaten bestehen aus einem zweidimensionalen Pixelraster, bei dem X Pixel nebeneinander in jeder Zeile liegen und Y Pixel in jeder Spalte übereinander liegen. Insgesamt gibt es N Pixel mit $N = X \cdot Y$. Die in diesem Kapitel eingeführten Bezeichnungen bleiben für den Rest des vorliegenden Textes gültig. Weiter gebe es einen Pixelindex p für alle Pixel mit:

$$p \in \{0, 1, 2, \ldots N-1\}$$

Die Pixel mögen zeilenweise in den verwendeten Datenstrukturen liegen, folglich sind $0, 1, \ldots X-1$ die Pixelindices der ersten Zeile, $X, X+1, \ldots 2 \cdot X - 1$ die Pixelindices der zweiten Zeile, usw. Bei einer solchen Anordnung kann auf einfache Weise auf die Nachbarpixel right, left, up, down eines Pixels p zugegriffen werden:

$$\begin{aligned} \text{right} &: p+1 \\ \text{left} &: p-1 \\ \text{down} &: p+X \\ \text{up} &: p-X \end{aligned}$$

Analog ergibt sich z.B. der Pixel oben links von p zu: $p - 1 - X$.

Manchmal hilfreich ist zusätzlich ein zweidimensionaler Index, bestehend aus dem Spaltenindex x und dem Zeilenindex y, zum Adressieren derselben Pixel. Für einen gegebenen Pixel mit Index p kann der zweidimensionale Index berechnet werden gemäß:

$$\begin{aligned} \text{x} &= p \; mod \; X \\ \text{y} &= p/X \end{aligned}$$

Dabei ist '*mod*' der für Ganzzahlen definierte Modulo Operator und '/' die Ganzzahl Division ohne Rest. Um das in Pseudocodeabschnitten abzukürzen, schreiben wir nachfolgend nur $p.x$ bzw. $p.y$ gemäß:

$$\begin{aligned} \text{p.x} &:= p \; mod \; X \\ \text{p.y} &:= p/X \end{aligned}$$

Sind umgekehrt von einem Pixel x und y gegeben, errechnet sich der eindimensionale Pixelindex p über:

$$\mathrm{p} = x + X \cdot y$$

Eingangsdaten: Anfänglich für jeden Pixel gegeben ist eine Klassifikation, eingetragen in einem Array `class` der Länge N. Die Klassifikation eines Pixels p kann folglich abgefragt werden gemäß: $class(p)$. Der Wertebereich für gültige Klassifikationen kann z. B. aus nicht negativen ganzen Zahlen bestehen: $0, 1, 2, \ldots$. Zusätzlich möge eine Konstante, $UNCLASSIFIED$, existieren, welche einen Pixel als unklassifiziert ausweist. Diese darf keinen als Klassifikation gültigen Wert haben, demnach bietet sich hier z.B. -1 an.

Ausgangsdaten: Der in dieser Arbeit zu formulierende Algorithmus berechnet für jeden klassifizierten Pixel ein Label. Diese werden, sobald bekannt, in einem N-komponentigen Array `label` hinterlegt. Der gültige Wertebereich für Label möge ebenfalls nicht negative ganze Zahlen umfassen: $0, 1, 2, \ldots$. Nicht klassifizierte Pixel werden auch nicht gelabeld, sondern bekommen eine vom Wertebereich der Label unterscheidbare Konstante, $UNLABELD$, zugewiesen.

8.1. Kontursegmente

Der Ansatz des in dieser Arbeit zu formulierenden Algorithmus extrahiert und analysiert zunächst die Konturen der Connected-Components. Diese Konturen werden, wie später gezeigt wird, lokal in Pixeln in Form von Kontursegmenten extrahiert. Es genügt an dieser Stelle, zu wissen, dass diese in vier Kategorien (Bezeichnung DST-Typ) fallen, mit:

$$\text{DST-Typ} \in \{RIGHT, LEFT, DOWN, UP\}$$

Weiterhin wird sich zeigen, dass von jedem DST-Typ maximal ein Kontursegment je Pixel existieren kann. Folglich kann es maximal vier Kontursegmente je Pixel geben. Damit haben alle Datenstrukturen, welche Kontursegmentdaten enthalten, die Größe $4 \cdot N$. Die einzelnen Kontursegmente werden nach einem Schema in den zugehörigen Datenstrukturen abgelegt, welches eine einfache Zuordnung von einem Pixel zu den zugehörigen Kontursegmenten ermöglicht und umgekehrt. Eine Möglichkeit ist, erst alle Kontursegmente des DST-Typs RIGHT abzulegen, dann alle des Typs LEFT, dann alle des Typs UP und zuletzt alle des Typs DOWN. Innerhalb jeder dieser Gruppen werden die Kontursegmente gemäß des Index des zugehörigen Pixels angeordnet. Somit ergibt sich folgendes Indexlayout für Kontursegmente mit Index c:

$$c \in [\underbrace{0, 1, \ldots, N-1}_{\text{Typ RIGHT}}, \underbrace{N, N+1 \ldots, 2 \cdot N - 1}_{\text{Typ LEFT}}, \underbrace{2 \cdot N, \ldots, 3 \cdot N - 1}_{\text{Typ DOWN}}, \underbrace{3 \cdot N, \ldots, 4 \cdot N - 1}_{\text{Typ UP}}]$$

Bei diesem Indexlayout können für einen gegebenen Pixel p die Indices der zugehörigen vier Kontursegmente für jeden DST-Typ ermittelt werden über:

$$\begin{aligned} RIGHT &: RS(p) = p \\ LEFT &: LS(p) = p + N \\ DOWN &: DS(p) = p + 2 \cdot N \\ UP &: US(p) = p + 3 \cdot N \end{aligned}$$

Umgekehrt lässt sich der zu einem Kontursegment mit Index c gehörige Pixel p finden mit:

$$p = c \ mod \ N$$

Insbesondere können so Pixel-Kontursegment-Beziehungen ausgerechnet werden und müssen nicht in zusätzlichen Datenstrukturen hinterlegt werden.

Der DST-Typ eines Kontursegments mit Index c kann bei Verwendung obigen Schemas ebenfalls ausgerechnet werden und wird daher nicht gesondert abgespeichert:

$$\begin{aligned} \text{Typ RIGHT, falls} &: c < N \\ \text{Typ LEFT, falls} &: c \geq N \wedge c < 2 \cdot N \\ \text{Typ DOWN, falls} &: c \geq 2 \cdot N \wedge c < 3 \cdot N \\ \text{Typ UP, falls} &: c \geq 3 \cdot N \end{aligned}$$

8.2. Globale Daten und Konstanten im Überblick

Dieser Abschnitt gibt einen Überblick hinsichtlich wichtiger Konstanten (Tabelle 8.1) und globaler Daten sowohl für Pixel (Tabelle 8.2) als auch für Kontursegmente (Tabelle 8.3). Deren Bedeutung wird sich in einigen Fällen erst später im Text erschließen. Es existieren weitere globale Daten, die jedoch nur für einzelne Varianten von Sub-Algorithmen benötigt werden. Diese werden dann in den entsprechenden Kapiteln eingeführt.

Tabelle 8.1.: Wichtige Konstanten.

Name	Anmerkung
N	Datenauflösung
X	Pixelspalten
Y	Pixelzeilen
RIGHT, LEFT, DOWN, UP	DST-Typ oder SRC-Typ

Tabelle 8.2.: Globale Daten der Pixel. Datenstrukturgröße: N

Name	Anmerkung	Wertebereich
class	Pixelklassifikation (Eingangsdaten)	$\{UNCLASSIFIED, 0, 1, ...\}$
label	Pixellabel (Ergebnisdaten)	$\{UNLABELD, 0, 1, ...\}$

Tabelle 8.3.: Globale Daten der Kontursegmente. Datenstrukturgröße: $4 \cdot N$

Name	Art	Anmerkung	Wertebereich
existing	Integer	/	$\{TRUE, FALSE\}$
suc	Referenz	Nachfolger	$\{0, 1, ..., 4 \cdot N - 1\}$
pre	Referenz	Vorgänger	s.o.
head, tail	Referenz	Kapitel 10.2.1	s.o.
ioCnt	Integer	S. Kap. 9.5.2	$\{0, 1, 2\}$
cLabel	Integer	Konturlabel	$\{UNLABELD, 0, 1, ...\}$
val	Integer	Vorstufe von cLabel	$\{0, 1, ...\}$
minDepth	Integer	Kapitel 9.5.1	$\{0, 1, ..., 2 \cdot Y - 1\}$
src	Konstante	Kapitel 9.2	$\{RIGHT, LEFT, UP, DOWN\}$

Kapitel 9.

Kontursegmente und deren Extraktion

Der erste Schritt des Algorithmus besteht in der Extraktion der Kontursegmente. Dies passiert unabhängig in jedem Pixel. Um basierend auf den Segmenten in späteren Phasen des Algorithmus parallel die vollständigen Konturen zu erkennen und die zugehörigen Pixel mit einem Label versehen zu können, müssen die Segmente einige Eigenschaften zwingend erfüllen:

1. Die Gesamtheit der erzeugten Segmente muss jede Connected-Component vollständig umschließen. Das gilt explizit auch für den Bildrand.

2. Innere und äußere Konturen müssen im späteren Verlauf des Algorithmus unterscheidbar sein. Die dafür nötige Information, der Drehsinn einer Kontur, muss aus den beteiligten Segmenten einer Kontur aggregierbar sein.

3. In der abschließenden Füll-Phase müssen Eintritt und Austritt in / aus Connected-Components eindeutig sein. Dies muss an den Segmenten eines Pixels entscheidbar sein.

4. Konturen müssen für spätere Phasen des Algorithmus durch zyklische gerichtete Linked-Lists aus Kontursegmenten mit einem In-Out-Degree von 1 (oder 2) repräsentiert sein.

5. Aus 2 und 4 folgt die Notwendigkeit der Definition einer eindeutigen Richtung jedes Segments, um von zwei angrenzenden Nachbarsegmenten eindeutig das Nachfolgesegment und, in manchen Varianten des Algorithmus erforderlich, das Vorgängersegment auszuwählen.

6. Der Algorithmus soll auch Datensätze mit verschiedenen Klassifikationen verarbeiten können. Hierbei kann eine Kante zwischen zwei Pixeln zu zwei verschiedenen Connected-Components gehören. Folglich muss jede Kante, mit Ausnahme derer am Bildrand, zwei Mal unabhängig existieren können.

Zusätzlich zu den für eine parallele Lösung des Connected-Component-Labeling-Problems mittels Konturen nötigen Eigenschaften sind für eine effiziente Lösung des Problems weitere Eigenschaften wichtig:

1. Die Anzahl verschiedener Segmenttypen, welche unterschiedlich zu interpretieren sind, muss konstant sein.

2. Es muss eine konstante Obergrenze für die Anzahl von Segmenten geben, die für einen einzelnen Pixel erzeugt werden können.

3. Bei der Segmentextraktion sollen keine Abhängigkeiten zwischen den zu erzeugenden Segmenten bestehen.

Die obengenannten Anforderungen unterscheiden sich deutlich von denen typischer Contour-Tracing-Algorithmen wie Chang [CCL04]. Diese können viele Entscheidungen durch die durch Contour-Tracing implizierte sequentielle Arbeitsweise treffen. So werden äußere Konturen von Inneren dadurch unterschieden, indem sie bei Verarbeitung ausgehend vom Bildrand als erstes gefunden werden. Diese Entscheidung kann in diesem parallelen Algorithmus jedoch nicht anhand lokaler Informationen getroffen werden, wie später im Kapitel 11 ausgeführt wird. Ferner kann bei sequentiellen Ansätzen etwa das nachfolgende Kontursegment festgelegt werden als eines, welches gerade nicht das Vorherige ist. So wurde etwa in älteren eigenen Ansätzen [WKV07, Wen07], die eine ähnliche Technik zum Beschriften der Isolinien verwenden, verfahren. Insgesamt machen die aus der parallelen Verarbeitungsweise resultierenden Unterschiede es erforderlich, von Contour-Tracing-Algorithmen in Bezug auf die Funktionsweise des Algorithmus Abstand zu nehmen und stattdessen eigene Konturen zu definieren.

9.1. Herleitung für Kontursegmente

Im Folgenden werden Kontursegmente hergeleitet, welche die oben geforderten Eigenschaften bereitstellen. Insbesondere sind diese nur für die Konnektivität gemäß Vierer-Pixelnachbarschaft gültig, welche in dieser Arbeit ausschließlich Verwendung findet.

9.1.1. Repräsentation der Pixelkanten

Wie gefordert, müssen Kontursegmente in der Summe die Connected-Components vollständig umschließen. Deshalb sind die Kontursegmente in einer Weise zu definieren, dass sie alle äußeren Kanten der Randpixel repräsentieren. Das Beispiel in Abbildung 9.1 veranschaulicht dieses Ziel. Hier liegt ein Datensatz vor, welcher zwei verschiedene Connected-Components enthält. Eingezeichnet sind bereits vollständige Konturen, welche je eine Connected-Component umschließen. Nun müssen einzelne Kontursegmente definiert werden, mit denen sich ein solches Ergebnis erreichen lässt. Dazu müssen die Kanten k mit $k \in (rechts, links, oben, unten)$ aller Pixel p, welche die folgenden Bedingungen erfüllen, durch Kontursegmente repräsentiert werden:

- Pixel p ist klassifiziert
- Der von p aus gesehen hinter k liegende Pixel ist nicht klassifiziert oder anders klassifiziert

Eine Ausnahme bilden hierbei Randpixel, welche in der Herleitung an dieser Stelle noch nicht betrachtet werden. Zur Veranschaulichung anhand des Beispiels siehe Abbildung

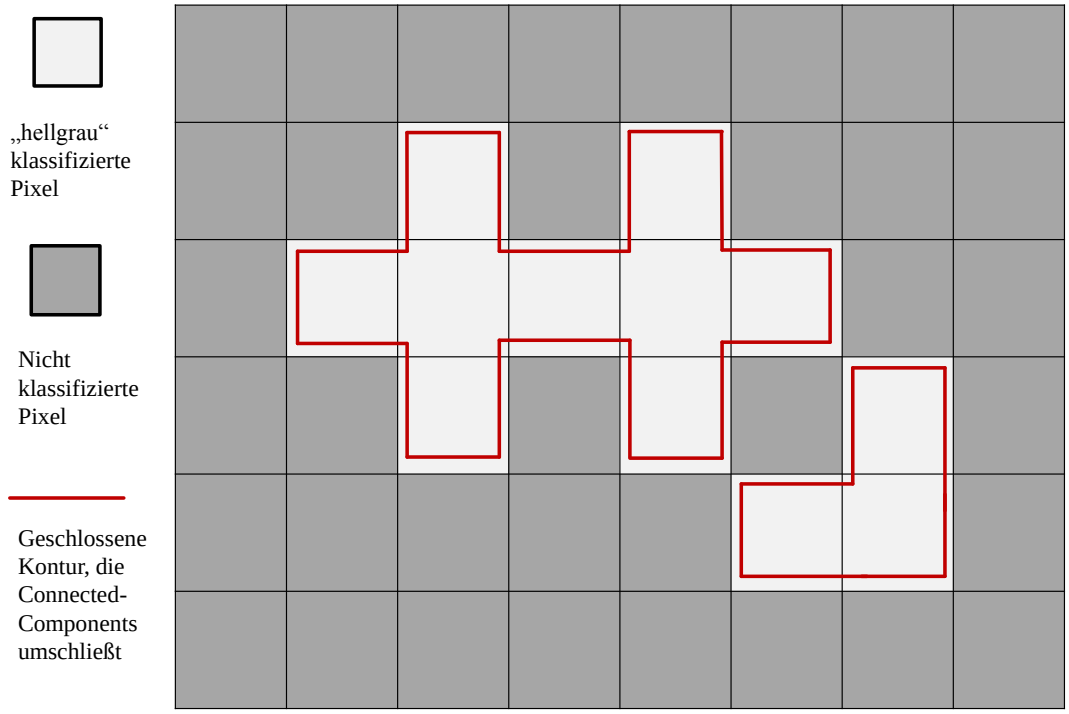


Abbildung 9.1.: Beispiel mit zwei gleich klassifizierten Bereichen, die gemäß Vierer-Nachbarschaft nicht vereinigt werden. Ziel des Verfahrens sind zwei getrennte Konturen die je einen Bereich vollständig umschließen.

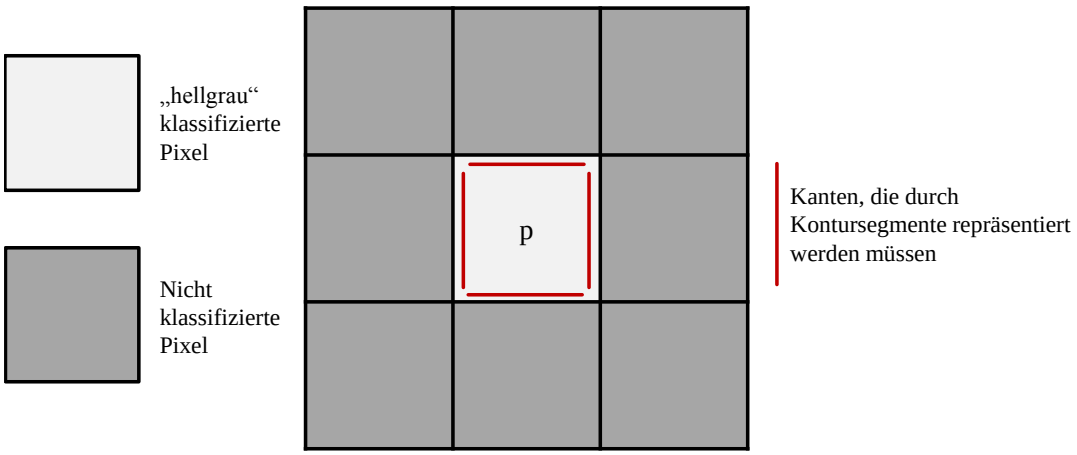


Abbildung 9.2.: Ein klassifizierter Pixel p umgeben von Unklassifizierten. Alle vier Kanten von p müssen durch Kontursegmente repräsentiert werden.

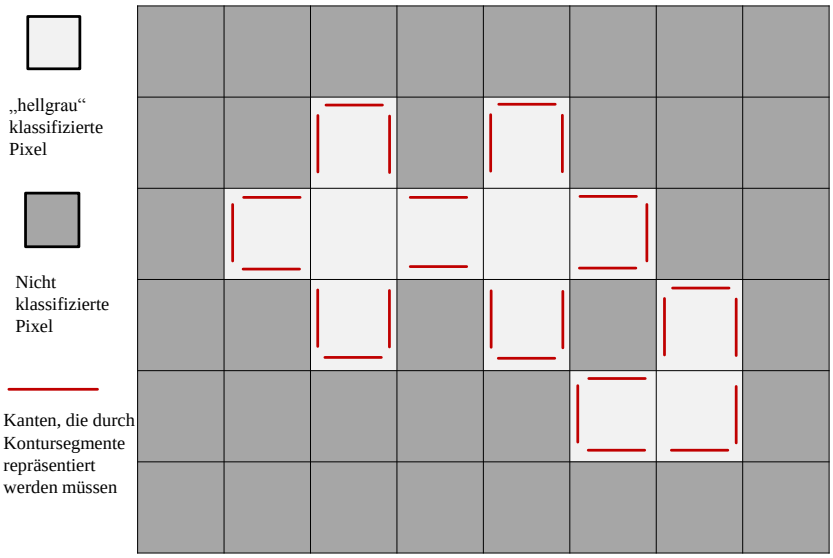


Abbildung 9.3.: Beispiel mit zwei gleich klassifizierten Bereichen, die gemäß Vierer-Nachbarschaft nicht vereinigt werden. Angegeben sind alle Kanten, die durch Kontursegmente repräsentiert werden müssen.

9.2. Dort ist eine aus einem einzigen Pixel bestehende Connected-Component dargestellt. In diesem Fall müssen alle Kanten durch Kontursegmente repräsentiert werden. Die linke Kante, weil der Pixel links unklassifiziert ist, die rechte Kante, da der Pixel rechts unklassifiziert ist, usw. Exakt das gleiche Prinzip ist in der Abbildung 9.3 anhand eines Beispiels einer komplizierteren Connected-Component veranschaulicht.

9.1.2. Verbindungskonturstücke

Aus den bisher beschriebenen Kontursegmenten, die verschiedene Kanten eines Pixels repräsentieren können, müssen im späteren Verlauf des Algorithmus vollständige Konturen generiert werden können. Beispielsweise müssen die in Abbildung 9.1 dargestellten Konturen aus den in Abbildung 9.3 dargestellten Segmenten erzeugt werden können. Dazu kann etwa eine Verbindung der Segmente *über Ecke* zugelassen werden, welche aber nicht recht in das Vier-Richtungs-Prinzip passen will. Alternativ können zusätzliche Verbindungsstücke eingeführt werden, wie in Abbildung 9.4 dargestellt. Im Rahmen dieser Arbeit wird ausschließlich der letztgenannte Ansatz untersucht. Diese Verbindungskonturstücke kann man sich anschaulich vorstellen als Verlängerung kantenrepräsentierenden Kontursegmenten angrenzender Pixel. Sie selbst repräsentieren keine Kante sondern dienen lediglich der Verbindung *echter* Konturstücke. Dementsprechend müssen sie im weiteren Verlauf des Algorithmus oft anders behandelt werden.

9.1.3. Lokale Vereinigung

Innerhalb einzelner Pixel kann die Zusammengehörigkeit verschiedener Konturstücke zu einer Kontur unmittelbar erkannt werden. Aus diesem Grund werden ausschließlich in-

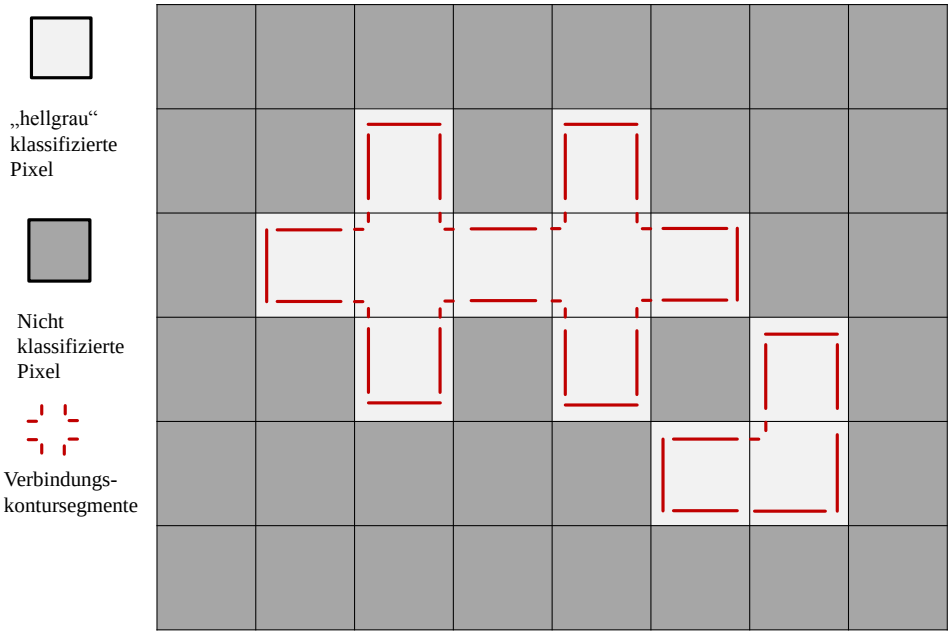


Abbildung 9.4.: Beispiel aus Abbildung 9.3, ergänzt um Verbindungskonturstücke

nerhalb je eines Pixels vereinigte Stücke aus einer Kontur extrahiert. Diese stellen damit elementare Kontursegmente dar. Dies ist beispielhaft in Abbildung 9.5 im Vergleich mit Abbildung 9.4 erkennbar, welche die noch unvereinigten Konturstücke bei ansonsten identischer Konfiguration zeigt.

9.1.4. Sonderfall: Bildrand

Ausnahmslos alle Konturen müssen, wie zuvor gefordert, geschlossen sein. Aus diesem Grund werden am Bildrand zusätzliche Segmente eingefügt. Dazu wird das Pixelgitter als nicht auf den Bildbereich eingeschränkt angenommen. Zusätzlich werden Pixel außerhalb des Bildbereichs als nicht klassifiziert angenommen. Auf diese Weise entstehen automatisch Konturen am Bildrand, wenn dort angrenzende Pixel klassifiziert sind. Zur Veranschaulichung kann das Beispiel in Abbildung 9.6 dienen. Mit dieser Ausnahme dieser Annahme stellt das Verhalten am Rand dann auch keinen Sonderfall mehr dar, da die Bedingungen zur Segmentextraktion davon abgesehen identisch sind. Unter Beachtung des nun gelösten Verhaltens am Bildrand ergibt sich zusammen mit den Erläuterungen zur Repräsentation der Pixelkanten in Abschnitt 9.1.1:

Satz 1 *Alle Connected-Components sind vollständig von anderen Bereichen durch Kontursegmente abgegrenzt.*

Das gilt offensichtlich, da alle Pixelkanten am Rand jeder Connected-Component durch Kontursegmente repräsentiert werden. Daraus folgt unmittelbar folgende Beobachtung:

Hilfssatz 1 *Vorbehaltlich einer geeigneten Vorschrift zum Verbinden aller auf die beschriebene Weise definierten Kontursegmente, sind alle Konturen geschlossen.*

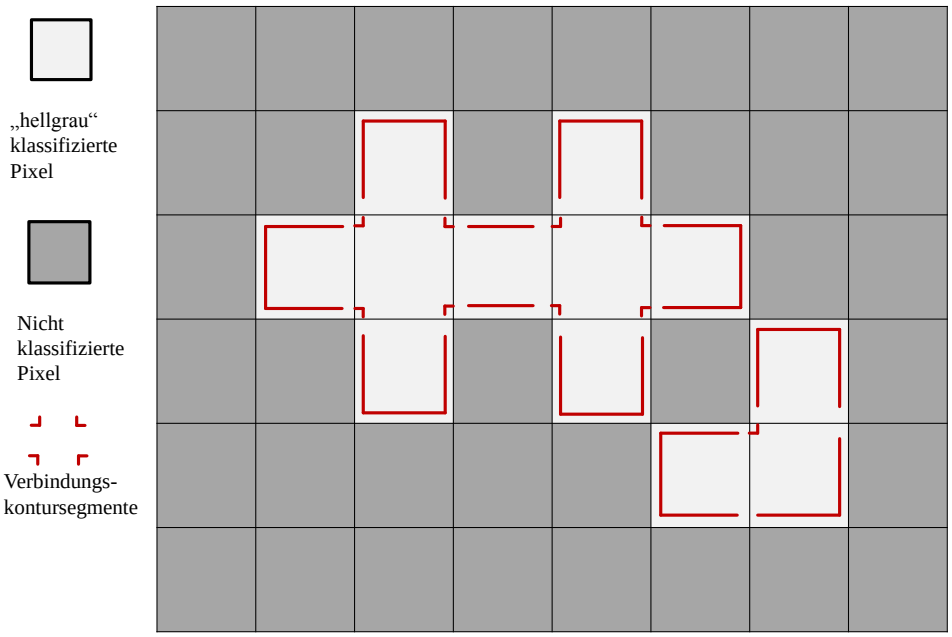


Abbildung 9.5.: Beispiel aus Abbildung 9.4 mit lokal in einzelnen Pixeln zusammengefassten Konturen.

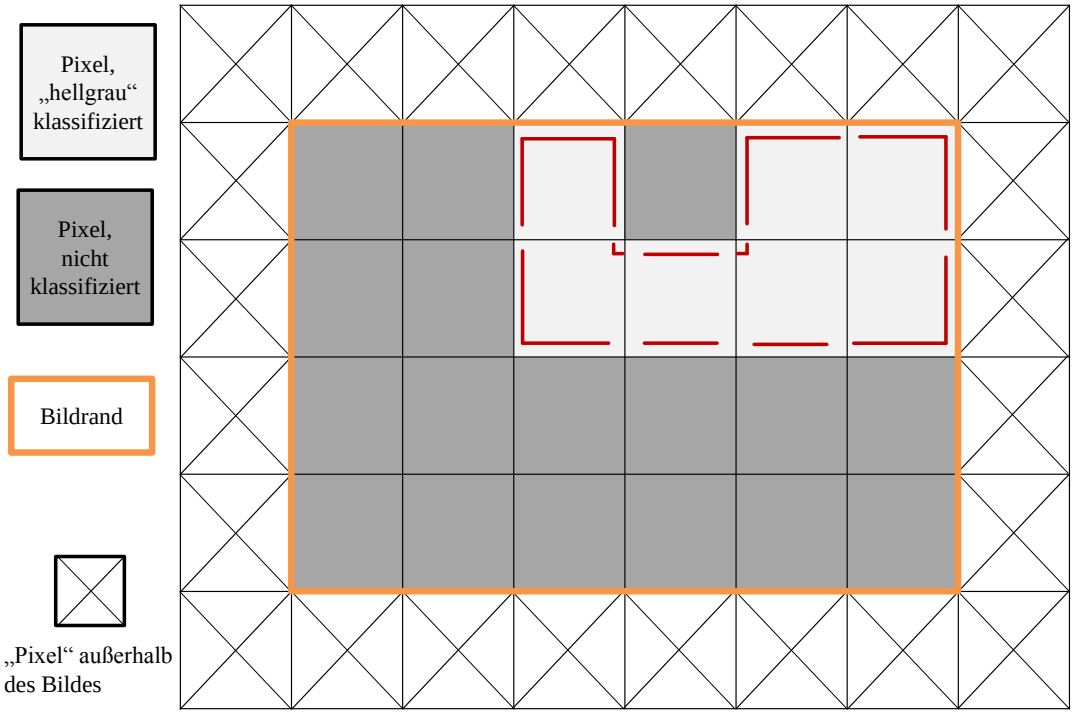


Abbildung 9.6.: Erzwungene Konturextraktion am Bildrand durch Betrachtung des erweiterten Pixelgitters. Pixel außerhalb des Bildbereichs werden als nicht klassifizierte Pixel definiert

Die Kontursegmente am Bildrand entsprechen natürlich (in der Regel) keinen echten Objektkonturen. Dementsprechend müssten solche Segmente, falls die Verwendung der Konturen etwa für Anwendungen in der Bildverarbeitung gewünscht ist, mit einem entsprechenden Flag versehen und nachträglich wieder entfernt werden. Im Rahmen dieser Arbeit wird das allerdings nicht weiter ausgeführt.

An dieser Stelle werden für weitere Abbildungen folgende Konventionen eingeführt, um die Legenden auf das Wesentliche zu beschränken:

- Gitter stellen immer Pixelgitter dar.
- Dunkelgrau gefärbte Pixel sind nicht klassifiziert
- Sehr hell grau (oder weiß) gefärbte Pixel sind (gleich) klassifiziert
- Der Rand eines dargestellten Pixelgitters ist immer auch der Bildrand. Die imaginären und als nicht klassifiziert definierten Pixel dahinter werden nicht dargestellt. Folglich werden am Rand immer Konturen generiert, falls die Pixel dort klassifiziert sind.
- Ausnahmen werden ggf. in der jeweiligen Legende angegeben.

Diese Konventionen gelten für den gesamten Text.

9.1.5. Definition der Richtung

Um Konturen als zyklische gerichtete Linked-Lists aus Kontursegmenten darstellen zu können, wie für die nachfolgende Phase des Algorithmus benötigt, müssen verschiedene Segmenttypen unterschieden und Richtungen für sie festgelegt werden. Die Definitionen lauten für einzelne Konturstücke:

- Konturstücke, die den rechten Pixelrand repräsentieren, sind nach oben gerichtet.
- Konturstücke, die den oberen Pixelrand repräsentieren, sind nach links gerichtet.
- Konturstücke, die den linken Pixelrand repräsentieren, sind nach unten gerichtet.
- Konturstücke, die den unteren Pixelrand repräsentieren, sind nach rechts gerichtet.
- Die Richtung der Segmente der Verbindungskonturstücke ist identisch mit der Richtung der Konturstücke, deren Verlängerung sie darstellen.

Aus dieser Richtungsdefinition folgt ein Drehsinn einer vollständigen Kontur gegen den Uhrzeigersinn, falls es sich um eine äußere Kontur handelt. Zur Veranschaulichung innerhalb eines einzelnen Pixels kann die Abbildung 9.7 dienen. Darin sind links die noch ungerichteten und auch noch nicht zusammengefassten Konturstücke dargestellt. Im mittleren Abbildungsteil wird die Richtungsdefinition übernommen, indem die Konturstücke als Pfeile dargestellt werden. Rechts dargestellt sind die jeweils innerhalb des Pixels zusammengefassten Kontursegmente. Die Richtung wird in den folgenden Abbildungen

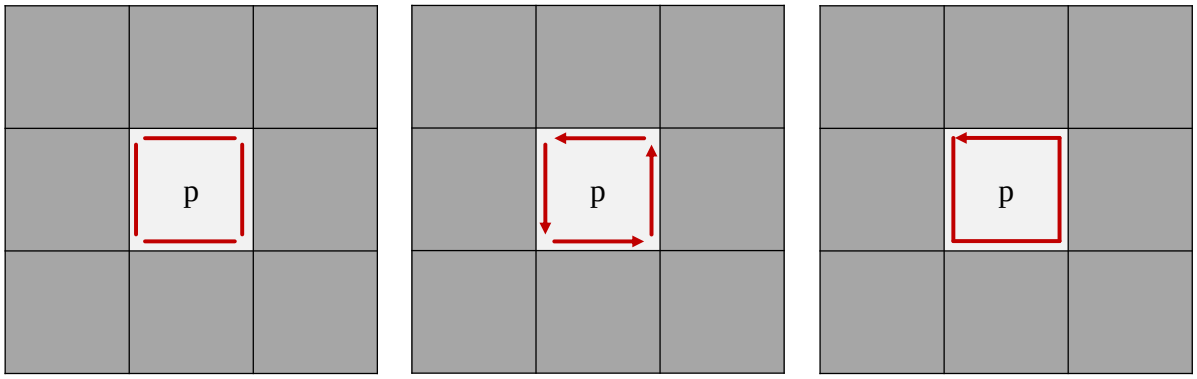


Abbildung 9.7.: Richtung der Kontursegmente in einem Pixel. Links: Noch ungerichtet. Mitte: Richtung der Konturstücke gemäß Definition. Rechts: Richtungsdarstellung bei zusammengefassten Kontursegmenten

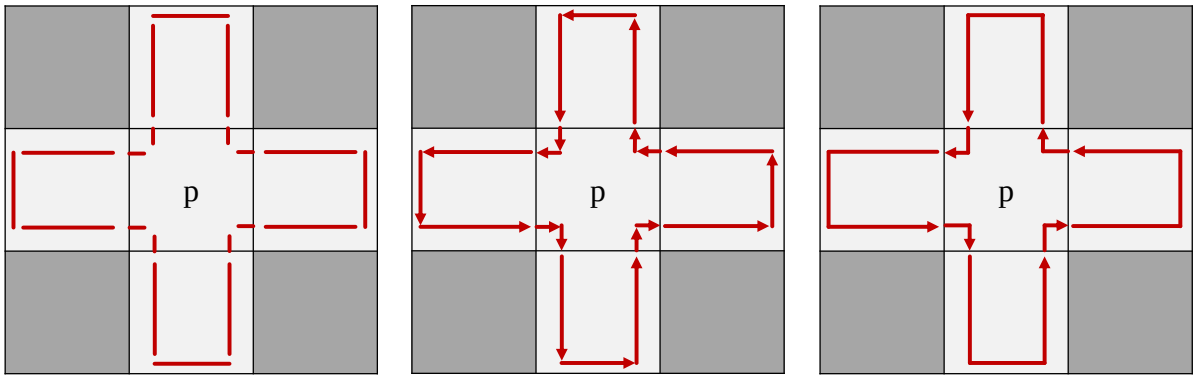


Abbildung 9.8.: Richtung der Kontursegmente in mehreren Pixeln. Links: Noch ungerichtet. Mitte: Richtung der Konturstücke und Verbindungskonturstücke gemäß Definition. Rechts: Richtungsdarstellung bei zusammengefassten Kontursegmenten

nur noch für zusammengefasste Segmente dargestellt, trotzdem bleibt aber die Information über die Richtung ihrer Teile, wie in der mittleren Abbildung zu sehen, erhalten. Ein ergänzendes Beispiel, welches zusätzlich die Richtung der Verbindungskonturstücke demonstriert, ist in Abbildung 9.8 zu sehen. Es ist ebenfalls in drei Teile, ungerichtete Konturstücke (links), gerichtete Konturstücke (Mitte) und gerichtete zusammengefasste Kontursegmente (rechts), unterteilt. Wir fassen zusammen:

Definition 1 *Konturen verlaufen innerhalb eines Pixels, in Bezug auf die Pixelkanten, immer gegen den Uhrzeigersinn. Kontursegmente, auch zusammengesetzte aus elementaren Kontursegmenten konsekutiver Pixelkanten, verlaufen ebenfalls gegen den Uhrzeigersinn.*

9.1.6. Resultierende Segmenttypen

Die Kontursegmente in der bisher beschriebenen Art sind innerhalb je eines Pixels jeweils soweit wie möglich verbunden. Folglich endet ein Kontursegment immer in zwei benachbarten Pixeln oder ist bereits innerhalb des Pixels geschlossen. Letzteres stellt einen Sonderfall dar, welcher genau dann auftritt, wenn eine Connected-Component aus genau einem Pixel besteht. Dieser wird gesondert behandelt:

Definition 2 *Wenn ein einzelner klassifizierter Pixel vorliegt, dessen Nachbarn gemäß Vierer-Nachbarschaft anders oder nicht klassifiziert sind, sprechen wir von einem* ***Einzel-Pixel-Sonderfall****.*

Im Falle des Einzel-Pixel-Sonderfalls wird dieser direkt gelabeld. Es wird daher kein Kontursegment extrahiert. Diese Situation ist in Abbildung 9.9 dargestellt. Wir halten fest:

Hilfssatz 2 *Jedes Kontursegment eines Pixels p verbindet zwei, nicht notwendigerweise verschiedene, Nachbarpixel der Vierer-Nachbarschaft von p.*

Im weiteren Verlauf dieses Textes werden nur noch Kontursegmente behandelt, welche wie zuletzt beschrieben in zwei Nachbarpixeln enden.

Wird noch die Richtung hinzugenommen, kommt jedes Segment aus einem Nachbarpixel (Quellpixel) und wird in einem Nachbarpixel (Zielpixel) fortgesetzt. Aufgrund der verwendeten Vierer-Nachbarschaft gibt es je vier Quellpixel und Zielpixel, nämlich jeweils den Rechten, den Linken, den Oberen und den Unteren. Mithilfe dieser Informationen können Kontursegmente in verschiedene Typen bzw. Kategorien eingeteilt werden.

Definition 3 *Der* ***DST-Typ*** *wird durch den Zielpixel eines Kontursegments festgelegt. Gemäß Hilfssatz 2 endet jedes Kontursegment im Pixel rechts, links, über oder unter dem eigenen Pixel. Nach diesem wird der DST-Typ benannt. Für den DST-Typ eines Kontursegments gilt somit: DST-Typ* $\in \{right, left, up, down\}$

Definition 4 *Der* ***SRC-Typ*** *wird durch den Quellpixel eines Kontursegments festgelegt. Analog zum DST-Typ (Definition 3) gilt: SRC-Typ* $\in \{right, left, up, down\}$

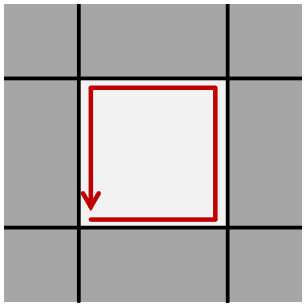


Abbildung 9.9.: Sonderfall einer 1 Pixel großen Connected-Component. Das Segment repräsentiert alle Kanten und ist innerhalb des Pixels geschlossen

Alle Kombinationen daraus sind möglich, sodass es 16 verschiedene Arten von Kontursegmenten gibt, wenn die verbundenen Pixel als Charakterisierung gewählt werden. Diese sind in Abbildung 9.10 dargestellt. Diese Kombinationen der Typen wird nachfolgend als SRC-DST-Typ bezeichnet.

Definition 5 *Der **SRC-DST-Typ** ist gegeben durch die Kombination aus SRC- und DST-Typ. Es gilt: SRC-DST-Typ* $\in \{right, left, up, down\} \times \{right, left, up, down\}$

Wie im Abschnitt 9.1.5 ab Seite 72 definiert, hat jede der vier Pixelkanten eine feste Richtung und keine andere Kante hat die gleiche Richtung. Außerdem endet jedes Kontursegment eines Pixels p in einem von p verschiedenen Pixel (siehe Hilfssatz 2 auf Seite 74). Aus diesen beiden Eigenschaften folgt eine feste Beziehung aus der vordersten repräsentierten Kante eines Kontursegments und des Zielpixels (bzw. DST-Typs):

Hilfssatz 3 *Ein Kontursegment kann nur dann in einem bestimmten Nachbarpixel pn enden, wenn sein vorderstes Teilstück diejenige Pixelkante repräsentiert, welche zu pn zeigt.*

Beispielsweise kann der Pixel rechts neben dem Aktuellen nur erreicht werden, wenn das vorderste Stück eines Kontursegments die untere Kante repräsentiert. Folglich muss der DST-Typ `right` sein ($\Leftrightarrow$ vorderstes Stück repräsentiert untere Kante) aber alle vier SRC-Typen sind möglich. Siehe dazu noch einmal Abbildung 9.10 auf Seite 76. Analog gilt:

Hilfssatz 4 *Ein Kontursegment kann nur dann an einem bestimmten Nachbarpixel pn beginnen, wenn sein hinterstes Teilstück diejenige Pixelkante repräsentiert, welche von pn weg zeigt.*

Beispielsweise ist ein SRC-Typ `UP` eines Kontursegments äquivalent mit der Information, dass dessen hinterstes Teilstück die linke Pixelkante repräsentiert.

Ferner können innerhalb eines Pixels mehrere unverbundene Kontursegmente existieren. Für Konturen ist gefordert, dass sie sich weder schneiden noch berühren, d.h. sie können keine Abschnitte gemeinsam haben. Dies wird später noch gezeigt. Alle Kombinationen von Kontursegmenten, welche diese Bedingungen erfüllen, können gemeinsam in einem Pixel existieren. Einige Beispiele dafür sind in Abbildung 9.11 dargestellt.

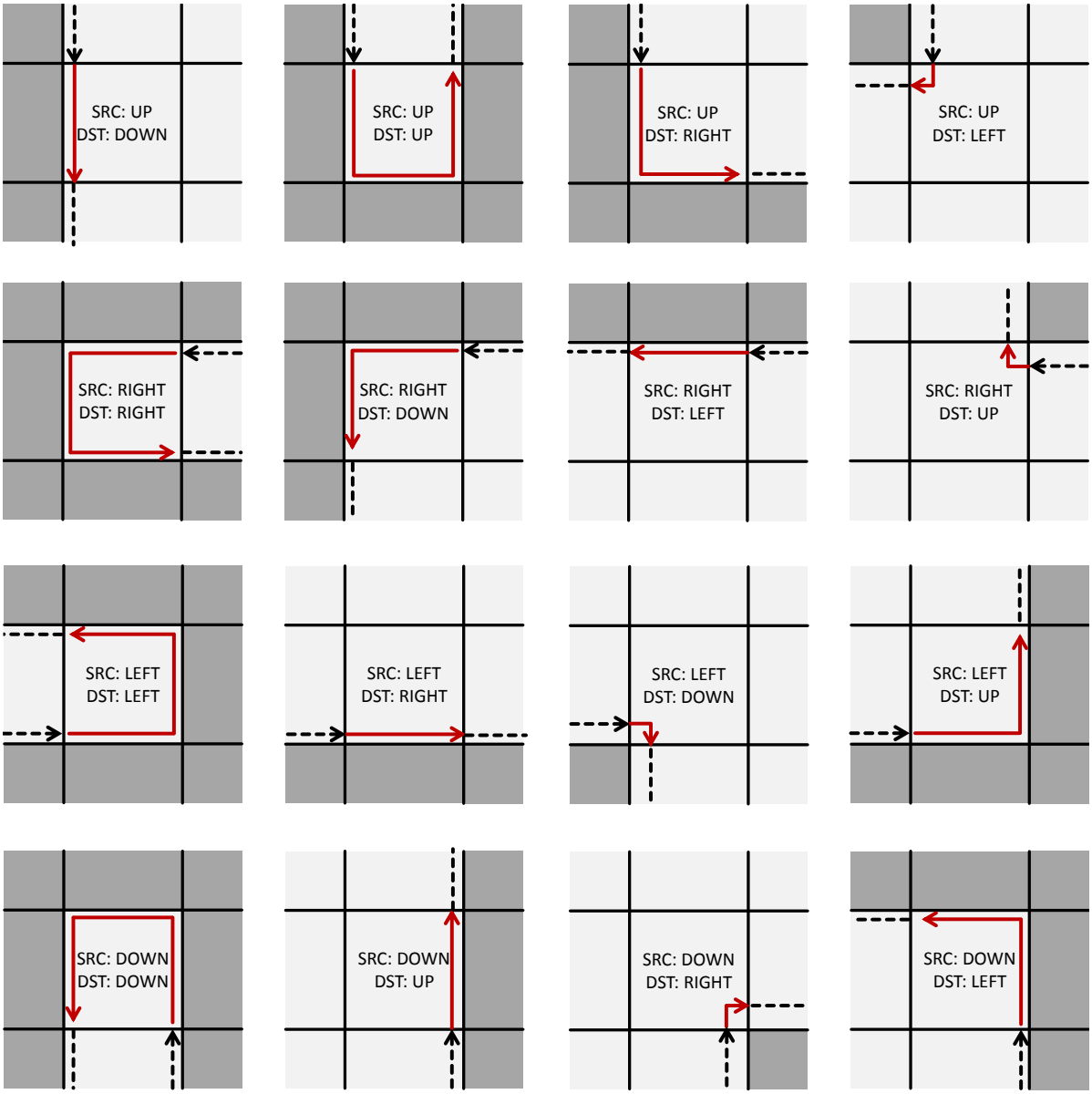


Abbildung 9.10.: Alle 16 möglichen Wege, die ein Kontursegment (rot) durch einen Pixel (Mitte) nehmen kann. Jeder dieser Fälle ist ein eigener SRC-DST-Typ.

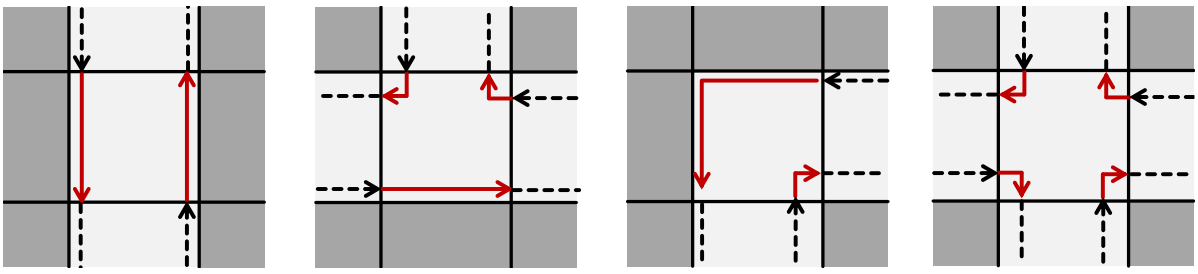


Abbildung 9.11.: Beispiele für Kombinationen von Kontursegmenten innerhalb eines Pixels

9.2. Bedingungen für Segmentextraktion und Festlegung des SRC- und DST-Typs

In den vorherigen Abschnitten des Kapitels 9 wurden Anforderungen für Kontursegmente, welche Abschnitte von Konturen von Connected-Components innerhalb eines Pixels repräsentieren können, festgelegt. Die Ausführungen resultierten in 16 verschiedenen SRC-DST-Typen von Kontursegmenten, welche, teilweise auch kombiniert, in jedem Pixel auftreten können. Alle 16 sind in Abbildung 9.10 dargestellt. Damit sind gewünschte Form und Eigenschaften der Kontursegmente festgelegt.

In diesem Kapitel wird erläutert, wie, abhängig von der Achter-Nachbarschaft eines Pixels, Kontursegmente in eben dieser geforderten Form (für Verbindung gemäß Vierer-Nachbarschaft) extrahiert werden können. Es bleibt dem nachfolgenden Kapitel (9.3, ab Seite 81) überlassen, anschließend die Gültigkeit des Verfahrens nachzuweisen.

Das Verfahren läuft in zwei Schritten ab. Als erstes wird für jeden DST-Typ eines Pixels überprüft, ob ein Kontursegment dieses Typs (unabhängig vom SRC-Typ) erzeugt werden soll. Dabei wird insbesondere für jeden DST-Typ maximal ein Kontursegment erzeugt. Dies wird sich später als ausreichend erweisen, wie in Kapitel 9.3 gezeigt wird. Intuitiv ist sofort klar, dass das gelten muss, wenn Konturen sich nicht schneiden oder berühren sollen. Danach wird in einem zweiten Schritt für jedes erzeugte Kontursegment der SRC-Typ bestimmt.

Segmentextraktion je DST-Typ

Notwendige Bedingung für die Existenz eines Kontursegments s eines bestimmten DST-Typs in einem Pixel p ist das Vorhandensein einer Connected-Component in den durch den DST-Typ von s vorgegebenen zwei Pixeln:

- p ist klassifiziert $\Rightarrow$ p ist Teil einer Connected-Component
- Zielpixel von s ist identisch mit p klassifiziert $\Rightarrow$ Connected-Component wird in Richtung von s fortgesetzt

Wenn beispielsweise s den DST-Typ `right` hat, muss zur Erfüllung der notwendigen Bedingung gelten: p ist klassifiziert und der Pixel rechts neben p ist identisch klassifiziert.

Hinreichende Bedingung für die Existenz eines Kontursegments s eines DST-Typs in einem Pixel p ist zusätzlich das Vorhandensein von nicht wie p klassifizierten Pixeln hinter den mit s assoziierten Pixelkanten, welche eine Kontur erforderlich machen. Dies ist der Fall, wenn eine der folgenden Bedingungen erfüllt ist:

- Der Pixel von p aus hinter der Kante, welche durch den DST-Typ von s gegeben ist (wie in Abschnitt 9.1.6 definiert) hat nicht die Klassifikation von p. Siehe dazu Abbildung 9.12.

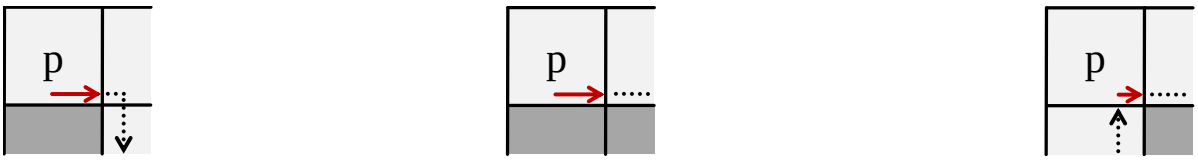


Abbildung 9.12.: Alle Pixelkonfigurationen, für die ein Kontursegment des DST-Typs `right` in einem Pixel `p` erzeugt wird

- Der Pixel hinter der Kante, welche durch die Verlängerung der Kante, die durch den DST-Typ von s gegeben ist, hat nicht die Klassifikation von p. Siehe dazu Abbildung 9.12.

Ein Kontursegment, welches Pixelkanten repräsentiert, erfüllt mindestens die erste Bedingung und ein Verbindungskontursegment ausschließlich die Zweite. Zum Beispiel wird ein Kontursegment des DST-Typs `right` in einem Pixel p erzeugt, wenn p klassifiziert ist, der Pixel rechts neben p identisch klassifiziert ist und der Pixel unter p oder der Pixel rechts unter p nicht gleich klassifiziert ist. Alle möglichen Pixelkonfigurationen, welche in der Erzeugung eines Kontursegments des DST-Typs `right` resultieren sind in Abbildung 9.12 dargestellt. Die Konfigurationen für Kontursegmente der DST-Typen `left, up` und `down` werden durch rotationssymmetrische Überlegungen deutlich. Diese Bedingungen werden für alle möglichen DST-Typen in jedem Pixel überprüft und jeweils ggf. ein Kontursegment des entsprechenden DST-Typs extrahiert. Es wurde gefordert:

Hilfssatz 5 *Wenn die Kontursegmente, wie in Abschnitt 9.2 festgelegt, extrahiert werden, gilt: Jede Pixelkante kann von maximal einem Kontursegment als vorderstes Stück auftreten.*

Bestimmung des SRC-Typs

Wenn die Bedingung für die Existenz eines Kontursegments eines DST-Typs erfüllt ist, kann es erzeugt werden und anschließend muss dessen SRC-Typ bestimmt werden. Aufgrund der per Definition gegen den Uhrzeigersinn laufenden Konturen innerhalb eines Pixels ist die Herkunftsrichtung der erste gleich klassifizierte Nachbarpixel im Uhrzeigersinn (vergleiche Definition 1 auf Seite 73). Die Überprüfung startet im ersten Pixel nach der Zielrichtung im Uhrzeigersinn. Zur Veranschaulichung sind in Abbildung 9.13 anhand des Beispiels eines Kontursegments vom DST-Typ `right` die Bedingungen für alle SRC-Typen dargestellt. Die Nachbarpixelkonfigurationen sind dabei so gewählt, dass es mehrere *Ausgänge* aus dem Pixel `p` gibt. Durch die per Definition gegen den Uhrzeigersinn innerhalb eines Pixels verlaufenden Konturen ist immer klar, welcher *Ein-* und *Ausgang* zusammengehören. Dementsprechend entstehen in `p` teilweise weitere Kontursegmente, die mit dem im Fokus liegenden Kontursegment des DST-Typs `right` nichts zu tun haben und zur deutlichen Unterscheidbarkeit hellgrau gestrichelt statt rot dargestellt sind.

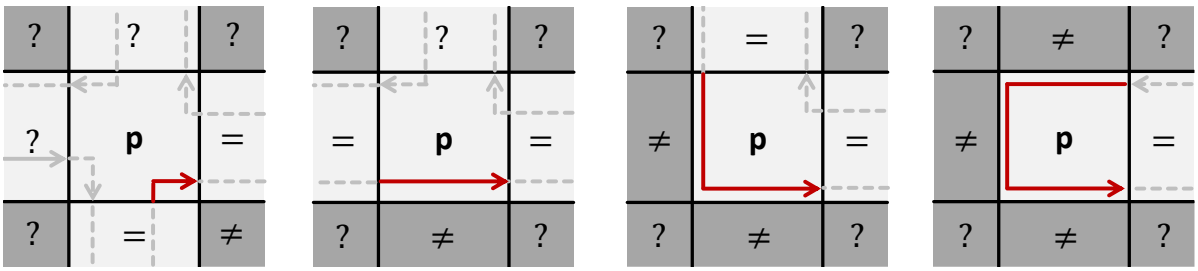


Abbildung 9.13.: Alle SRC-Typen für ein Kontursegment (rot) des DST-Typs `right` eines Pixels `p`. V.l.n.r.: `down, left, up, right`. Einordnung hängt von Übereinstimmung der Klassifikation von `p` mit Nachbarpixeln ab: = muss übereinstimmen, ≠ muss abweichen, ? Übereinstimmung für rotes Segment irrelevant. Graue Linien werden bei eingezeichneter Pixelkonfiguration ebenfalls erzeugt.

Ergebnis

Obige Überlegungen können nun algorithmisch formuliert werden. Die Funktion initSegsTypes, gegeben im entsprechenden Pseudocode Abschnitt, zeigt, wie die Existenz der Kontursegmente eines Pixels für die einzelnen DST-Typen bestimmt wird und wie jeweils der SRC-Typ festgelegt wird. Sie wird für einen Pixel `p` aufgerufen und erzeugt dann die bis zu vier zugehörigen Kontursegmente.

Wie bereits im Kapitel 8 (ab Seite 62), welches das Speicherlayout der Kontursegmente beschreibt, angekündigt, muss der DST-Typ nicht abgespeichert werden, sondern ist durch die Position in den Datenstrukturen für Kontursegmente implizit gegeben. Dagegen wird der SRC-Typ in einem Array `src`, welches die Länge $4 \cdot N$ hat, an der mit dem Kontursegment korrespondierenden Stelle hinterlegt. Wenn beispielsweise in einem Pixel p ein Kontursegment des SRC-Typs `LEFT` und des DST-Typs `DOWN` gefunden wurde, wird entsprechend `src(DS(p))` auf `LEFT` gesetzt. Dazu wurde die in Kapitel 8.1 ab Seite 64 gegebene Adressierungsvorschrift verwendet.

Insgesamt gibt es bei einem klassifizierten Pixel `p` abhängig davon, welche der acht Nachbarpixel eine mit `p` übereinstimmende Klassifikation aufweisen $2^8 = 256$ mögliche Nachbarschaftskonfigurationen. Allerdings führen einige dieser Umgebungspixelkonfigurationen zu identischen Kontursegmenttypen, die für `p` zu erzeugen sind. Ein Beispiel dafür ist in Abbildung 9.14 dargestellt. Darin sind alle Umgebungskonfigurationen eines Pixels `p` angegeben, welche in der Erzeugung der selben beiden Kontursegmente (rot) resultieren. Hellgrau gestrichelte Linien stellen Abschnitte von Kontursegmenten in Nachbarpixeln dar. Sofern anhand der gezeigten Pixelkonfiguration auf deren DST-Typ geschlossen werden kann sind sie zusätzlich mit einer Pfeilspitze versehen.

Function initSegsTypes(Pixel p)

```
// Precondition: p is classified
// Note: See chapter 8 p. 62 et seq. for data layout
// Analyse, which of p's 8 neighbors share p's classification
// E.g. drEq = TRUE: Pixel down-right of p is equally classified
// Default: FALSE (Remains if out of image)
rEq ← dEq ← drEq ← lEq ← uEq ← dlEq ← ulEq ← urEq ← FALSE
If p.x < X − 1 Then
    rEq ← class(p + 1) = class(p)
If p.y < Y − 1 Then
    dEq ← class(p + X) = class(p)
If p.y < Y − 1 ∧ p.x < X − 1 Then
    drEq ← class(p + X + 1) = class(p)
If p.x > 0 Then
    lEq ← class(p - 1) = class(p)
If p.y > 0 Then
    uEq ← class(p - X) = class(p)
// dlEq, ulEq and urEq similar initialized
// Check if Contour-Segment of DST-Type RIGHT exists
If rEq ∧ (¬dEq ∨ ¬drEq) Then
|   existing(RS(p)) ← TRUE
|   If dEq Then // Testing for 'source-exit' begins under p...
|   |   src(RS(p)) ← DOWN
|   Else If lEq Then // ... and proceeds clockwise
|   |   src(RS(p)) ← LEFT
|   Else If uEq Then
|   |   src(RS(p)) ← UP
|   Else
|   |   src(RS(p)) ← RIGHT
|   End
End
// Check if Contour-Segment of DST-Type UP exists
If uEq ∧ (¬urEq ∨ ¬rEq) Then
|   existing(US(p)) ← TRUE
|   src(US(p)) ← ... // As above, but testing begins right of p
End
// Check if Contour-Segment of DST-Type LEFT exists
If lEq ∧ (¬uEq ∨ ¬ulEq) Then
|   existing(LS(p)) ← TRUE
|   src(LS(p)) ← ... // As above, but testing begins above p
End
// Check if Contour-Segment of DST-Type DOWN exists
If dEq ∧ (¬lEq ∨ ¬dlEq) Then
|   existing(DS(p)) ← TRUE
|   src(DS(p)) ← ... // As above, but testing begins left of p
End
```

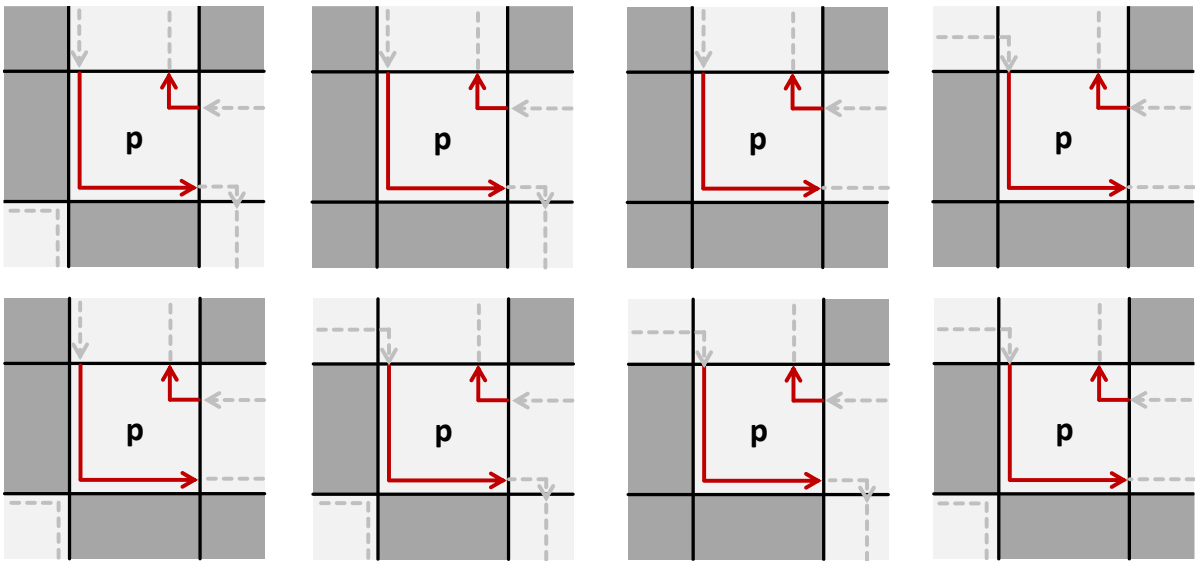


Abbildung 9.14.: Alle Umgebungspixelkonfigurationen eines Pixels p, welche die Existenz genau der beiden rot dargestellten Kontursegmenttypen bewirken. Graue Linien: Abschnitte von Kontursegmenten in Nachbarpixeln. Pfeilspitzen geben DST-Typ an (sofern entscheidbar)

9.3. Untersuchung der Kontursegmenteigenschaften

Im vorherigen Abschnitt wurde angegeben, unter welchen Bedingungen Kontursegmente erzeugt werden und wie SRC- sowie DST-Typ (und damit der Verlauf innerhalb eines Pixels) festgelegt werden. Es bleibt zu zeigen, dass die so erzeugten Kontursegmente die Konturen der Connected-Components vollständig beschreiben (gemäß Satz 1, S. 70) und sich weder schneiden noch berühren können.

Wenn Kontursegmente nach der in Abschnitt 9.2 ab Seite 77 gegebenen Vorschrift erzeugt werden, kann es maximal ein Kontursegment für jeden DST-Typ in einem Pixel geben. Dann gilt unter Hinzunahme des Hilfssatzes 3:

Hilfssatz 6 *Aus jedem Pixel kann jeweils maximal ein Kontursegment den Pixel in Richtung eines jeden der vier Nachbarpixel verlassen.*

Weil Hilfssatz 6 für alle Pixel gilt, insbesondere auch für die vier Nachbarpixel eines Pixels gemäß Vierer-Nachbarschaft, folgt:

Hilfssatz 7 *In jeden Pixel kann jeweils maximal ein Kontursegment den Pixel aus Richtung eines jeden der vier Nachbarpixel betreten.*

9.3.1. Einige Hilfsdefinitionen und deren Eigenschaften

Definition 6 *Ein **Austrittspunkt** existiert für jedes Kontursegment s jedes Pixels p. Er markiert die Stelle einer Kante aus p, an der die Spitze von s aus p herauszeigt.*

Die Bezeichnung ist nur der Anschaulichkeit halber so gewählt. Tatsächlich verlässt ein Kontursegment den zugehörigen Pixel nicht, sondern zeigt eben nur heraus. Analoges gilt für den Eintrittspunkt:

Definition 7 *Ein **Eintrittspunkt** existiert für jedes Kontursegment s jedes Pixels p. Er markiert die Stelle einer Kante aus p, an der der Stiel von s aus p beginnt.*

Es gibt, gemäß der Hilfssätze 6 und 7 offensichtlich auf jeder der vier Pixelkanten je einen Eintrittspunkt und einen Austrittspunkt. Für deren Reihenfolge wird eine feste Vorschrift benötigt.

Um das anschaulich zu vereinfachen, machen wir zunächst noch eine kleine Vorüberlegung hinsichtlich der räumlichen Lage der Kontursegmente in Bezug auf eine durch sie repräsentierte Kante. In Abbildung 9.15 sind zwei benachbarte Pixel mit einer gemeinsamen Kante zu sehen. Außerdem sind in der Abbildung zwei Kontursegmente dargestellt

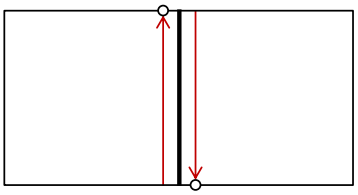


Abbildung 9.15.: Veranschaulichung zur Lage von Kontursegmenten im Vergleich zu der durch sie repräsentierten Kante. Kreise markieren die Austrittspunkte

(eines in jedem Pixel), welche beide eben diese Kante repräsentieren. Wir nehmen die Lage einer Kante nachfolgend als neutral zwischen je zwei Pixeln an. Die Kontursegmente dagegen seien von der Kante aus infinitesimal weit in Richtung 'ihres' Pixels verschoben, sodass sie diesen immer noch komplett umschließen können, den anderen aber nicht mehr berühren.

In gewisser Weise ist die Kante demnach doppelt vorhanden. Das zeigt sich auch darin, dass ihre Richtung in beiden Pixeln eine andere ist. Schließlich ist die in Abbildung 9.15 fettgedruckte Kante für den linken Pixel die rechte Kante und für den rechten Pixel die linke Kante. Das wiederum resultiert in den unterschiedlich ausgerichteten Kontursegmenten.

In jedem Fall bewirkt die 'Verschiebung' der Kontursegmente, dass diese diejenigen Kanten, über welche sie den Pixel verlassen, nicht (genau) in den Pixelecken, sondern an den in Abbildung 9.15 markierten Stellen treffen. Damit können sie auch klar dieser Kante zugeordnet werden.

Mit diesen Vorüberlegungen ergibt sich die Lage aller möglichen Eintrittspunkte und Austrittspunkte wie links (und Mitte) in Abbildung 9.16 für Pixel p dargestellt. Allen Ein- und Austrittspunkten können eindeutig Kontursegmente gemäß SRC- bzw. DST-Typ zugeordnet werden. Beispielsweise ist ein Verlassen nach oben nur möglich, wenn das Kontursegment nach oben zeigt. Das ist genau dann der Fall, wenn die Spitze des Kontursegments die rechte Pixelkante repräsentiert. Das wiederum ist äquivalent mit dem DST-Typ `up`, usw.

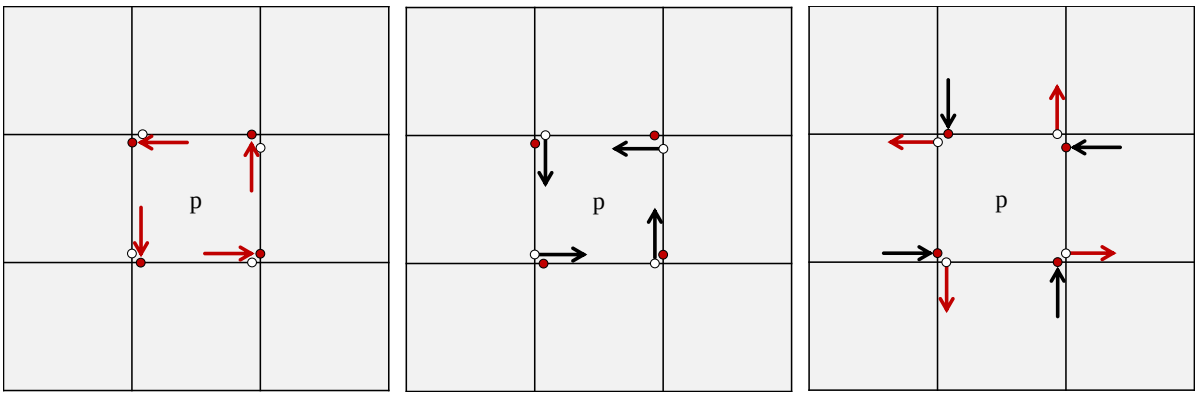


Abbildung 9.16.: Links und Mitte: Alle möglichen Eintrittspunkte (weiß) und Austrittspunkte (rot) für Pixel p. Links: Zu Austrittspunkten gehörende vordere Segmenthälften gemäß DST-Typ (rot). Mitte: Zu Eintrittspunkten gehörende hintere Segmenthälften gemäß SRC-Typ (schwarz). Rechts: Ein- und Austrittspunkte sowie Segmenthälften gemäß SRC- und DST-Typ der Nachbarpixel von p für gemeinsame Kanten zu p. Ein- und Austrittspunkte sind vertauscht.

In der linken Teilabbildung sind die zugehörigen Kontursegmente aus p gegeben, welche gemäß DST-Typ fest mit einem Austrittspunkt verknüpft sind. Dargestellt sind jeweils nur die vorderen Hälften der Kontursegmente, da nur diese über den DST-Typ festgelegt ist. Der mittlere Teil der Abbildung 9.16 zeigt die Kontursegmente aus p, welche gemäß SRC-Typ mit einem der Eintrittspunkte verknüpft sind. Hier sind entsprechend nur die vorderen Hälften bekannt und angezeigt.

Stellen wir uns nun alle Ein- und Austrittspunkte topologisch durch einen Ring dar, verbunden in der Reihenfolge ihres Auftretens beim zyklischen Ablaufen aller Pixelkanten. Dann gilt mit den vorherigen Überlegungen zur Anordnung:

Hilfssatz 8 *Bei Betrachtung der Ein- und Austrittspunkte eines Pixels in zyklischer (oder antizyklischer) Reihenfolge wechseln sich immer Ein- und Austrittspunkte ab. Bei Betrachtung im Uhrzeigersinn und (irgendeine) erste Kante vollständig beinhaltend, ist der erste Punkt ein Eintrittspunkt.*

Das gleiche Prinzip gilt auch für alle p umgebenden Pixel. In dem rechten Teil der Abbildung 9.16 sind alle Ein- und Austrittspunkte der Nachbarn auf einer gemeinsamen Kante mit p samt zugehöriger Kontursegmente gemäß SRC- bzw. DST-Typ angezeigt. Offensichtlich sind die Eintrittspunkte von p in Bezug auf die Nachbarpixel Austrittspunkte und umgekehrt. Das muss auch so sein, da die Richtung jeder fortgesetzten Kante im Nachbarpixel gleich bleibt, diese im betrachteten Pixel aber dem DST-Typ (beim Verlassen) entspricht wohingegen die Richtung im nächsten Pixel mit dem SRC-Typ (beim Betreten) zusammenhängt. Wir halten fest:

Hilfssatz 9 *Alle Eintrittspunkte eines Pixels p sind Austrittspunkte der an die entsprechenden Kanten angrenzenden Nachbarpixel. Analog sind alle Austrittspunkte von p Eintrittspunkte eben dieser Nachbarn.*

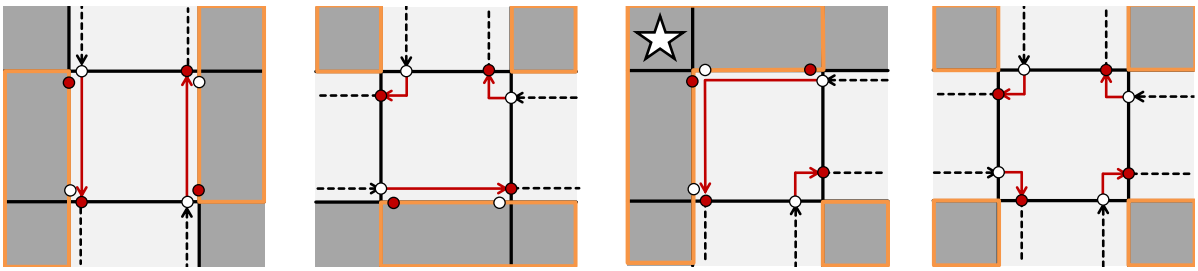


Abbildung 9.17.: Beispiele für Konturerzeugerblöcke (jeweils orange umrandet) für verschiedene Umgebungskonfigurationen eines Pixels p (jeweils in der Mitte). Der mit dem Stern markierte Pixel gehört nicht unbedingt dazu. Zusätzlich angegeben sind Eintrittspunkte (weiß) und Austrittspunkte (rot) in Bezug auf p. Solche auf einer zu einem Konturerzeugerblock gehörenden Kante liegenden sind blockiert.

Zusätzlich zu Ein- und Austrittspunkten führen wir noch zur Unterstützung der Ausführungen den Begriff 'Konturerzeugerblock' ein. Für einen klassifizierten Pixel p, welcher von je mindestens einem gleich klassifizierten (gemäß Vierer-Nachbarschaft) und ungleich klassifizierten Pixel (gemäß Achter-Nachbarschaft) umgeben ist, existiert mindestens ein Konturerzeugerblock. Dieser besteht anschaulich aus Gruppen ungleich klassifizierter Nachbarpixel, mit der Nebenbedingung, dass kein Kontursegment von p zwischen den Pixeln des Konturerzeugerblocks heraus/herein zeigen kann. Ein Konturerzeugerblock ist folgendermaßen definiert:

Definition 8 *Gegeben sei ein Pixel p mit Klassifikation c. Sei p_{nu} ein gleich klassifizierter Nachbarpixel der Vierer-Nachbarschaft von p. Die übrigen Pixel der Achter-Nachbarschaft von p werden nun im Uhrzeigersinn, ausgehend von p_{nu} betrachtet. Wenn der erste Pixel (p_{ng}) der Achter- oder, wenn nicht, dann der Vierer-Nachbarschaft von p nicht mit c klassifiziert ist, existiert Konturerzeugerblock beginnend mit p_{ng}. Zu diesem Konturerzeugerblock gehören alle weiteren nicht c klassifizierten Pixel der Vierer-Nachbarschaft von p im Uhrzeigersinn, bis ein Pixel der Vierer-Nachbarschaft wieder c klassifiziert ist.*

Zur obigen Definition sei noch angemerkt, dass ein Konturerzeugerblock mit einem Pixel der Achter-Nachbarschaft beginnen kann, 'eingeschlossene' Pixel der Achter-Nachbarschaft aber nicht relevant sind. Abbildung 9.17 zeigt beispielhaft einige Konturerzeugerblöcke. Dabei sind verschiedene zu einem Pixel gehörige jeweils farblich hervorgehoben.

Offensichtlich beinhaltet die Definition des Konturerzeugerblocks alles zur Erzeugung eines Kontursegments erforderliche (darum auch der Name). Wir halten fest:

Hilfssatz 10 *Kontursegmente, wie in Kapitel 9.1 gefordert, sind genau dann erforderlich, wenn ein Konturerzeugerblock vorliegt.*

Offensichtlich sind alle Eintritts- und Austrittspunkte der Kanten eines Pixels, hinter denen Pixel eines Konturerzeugerblocks liegen, für diesen Pixel nicht verfügbar. Wenn dagegen kein angrenzender Konturerzeugerblock vorliegt, werden mangels anders klassifizierter Nachbarpixel keine Konturen erzeugt, welche dort verlaufen könnten (auch Hilfssatz 10). Somit folgt:

Hilfssatz 11 *Kontursegmente können Pixel nur über Eintritts- und Austrittspunkte betreten / verlassen, welche unmittelbar an einen Konturerzeugerblock angrenzen.*

Jeder Konturerzeugerblock blockiert von null, einer, zwei oder drei im Uhrzeigersinn aufeinanderfolgenden Kanten jeweils den Ein- und Austrittspunkt. Bei Betrachtung im Uhrzeigersinn befindet sich gemäß Hilfssatz 8 vor dem Konturerzeugerblock der 'zweite' Punkt einer Kante und dann dahinter der 'erste' Punkt einer Kante. Dann gilt, wieder mit Hilfssatz 8:

Hilfssatz 12 *Bei Betrachtung im Uhrzeigersinn befindet sich vor einem Konturerzeugerblock immer als Erstes eine Ausgangsposition und nach einem Konturerzeugerblock immer als erstes eine Eingangsposition.*

Zur Veranschaulichung siehe auch noch einmal Abbildung 9.17.

9.3.2. Anwendung der Hilfsdefinitionen

Wir wiederholen jetzt noch einmal die in Abschnitt 9.2 formulierte Vorgehensweise zum Extrahieren von Kontursegmenten eines bestimmten SRC-DST-Typs unter Verwendung der in diesem Abschnitt eingeführten Begriffe. Die dort gegebene Bedingung zum Extrahieren eines Kontursegments eines bestimmten DST-Typs ist äquivalent mit der Überprüfung, ob, bei Betrachtung im Uhrzeigersinn, ein Anfang eines Konturerzeugerblocks vorliegt. Danach wird, wie im Abschnitt 9.2 beschrieben, im Uhrzeigersinn der erste mit dem betrachteten Pixel gleich klassifizierte Pixel der Vierer-Nachbarschaft gesucht. Durch diesen wird der SRC-Typ definiert. Das Äquivalent mit der Suche nach dem Eintrittspunkt unmittelbar vor dem Konturerzeugerblock. Aus dieser Vorgehensweise ergibt sich zusammen mit Hilfssatz 12:

Hilfssatz 13 *Jedes Kontursegment verbindet den Austrittspunkt direkt vor einem Konturerzeugerblock und den Eingangspunkt direkt hinter demselben Konturerzeugerblock.*

Aus der in Abschnitt 9.2 formulierten Vorgehensweise der Kontursegmentextraktion gemäß DST-Typ folgt in diesem Zusammenhang:

Hilfssatz 14 *Für jeden Austrittspunkt direkt vor einem Konturerzeugerblock existiert genau ein Kontursegment.*

Aus Hilfssätzen 13 und 14 folgt dann:

Hilfssatz 15 *Für jeden Eintrittspunkt direkt hinter einem Konturerzeugerblock existiert genau ein Kontursegment.*

Es ist gerade das für den Austrittspunkt Erzeugte. Aus den Hilfssätzen 13, 14 und 15 folgt weiter:

Hilfssatz 16 *Zu den Kanten, welche ein Pixel mit einem Konturerzeugerblock gemeinsam hat, gehört genau ein Kontursegment.*

Hierbei sind Hilfskonturstücke ausgeklammert, da sie keine Kanten repräsentieren. Weil ohne Konturerzeugerblöcke keine Kontursegmente existieren (Hilfssatz 10) folgt aus Hilfssatz 16:

Satz 2 *Jede Pixelkante kann zu maximal einem Kontursegment gehören, wenn die Kontursegmente wie in Abschnitt 9.2 beschrieben extrahiert werden. Damit können sich Kontursegmente nicht berühren. Konturen, die ausschließlich aus diesen Kontursegmenten bestehen, können sich folglich auch nicht berühren.*

Kontursegmente verlaufen innerhalb eines Pixels gegen den Uhrzeigersinn (siehe Abschnitt 9.1.5) entlang der Kanten zu ihrem zugehörigen Konturerzeugerblock. Damit verlaufen sie insbesondere nicht quer durch einen Pixel. Die Bereiche sind außerdem für verschiedene Kontursegmente voneinander getrennt (Satz 2). Und Kontursegmente existieren nur innerhalb des zugehörigen Pixels. In der Gesamtheit folgt daraus:

Satz 3 *Kontursegmente können sich nicht schneiden, wenn diese wie in Abschnitt 9.2 beschrieben extrahiert werden. Konturen, die ausschließlich aus diesen Kontursegmenten bestehen, können sich folglich auch nicht schneiden.*

Weil gemäß Definition des Konturerzeugerblocks (u.a. alle anders klassifizierten Pixel der Vierer-Nachbarschaft in einem davon enthalten) zusammen mit Hilfssatz 16 die in Abschnitt 9.1.1 geforderten Konturstücke garantiert werden, gilt:

Satz 4 *Kontursegmente, die wie in Abschnitt 9.2 beschrieben, für alle Pixel extrahiert werden, repräsentieren in ihrer Gesamtheit alle erforderlichen Konturen aller Connected-Components.*

9.4. Repräsentation als zyklische gerichtete Linked-Lists

Bislang wurde beschrieben, wie in den einzelnen Pixeln unabhängige Kontursegmente erzeugt werden können, welche aus einem Nachbarpixel stammen (gegeben durch SRC-Typ) und in einem Nachbarpixel enden (gegeben durch DST-Typ), welche in ihrer Gesamtheit die Connected-Components vollständig umschließen.

Gefordert ist die Repräsentation von Konturen durch zyklische gerichtete Linked-Lists aus eben diesen Kontursegmenten (Vergleich Einleitung v. Kapitel 9, ab Seite 66). Gezeigt werden müssen folglich noch drei Dinge:

1. Existenz Nachfolger: Für jedes Kontursegment existiert genau ein Nachfolger.
2. Existenz Vorgänger: Für jedes Kontursegment existiert genau ein Vorgänger.
3. Vorgänger und Nachfolger gehören zur selben Kontur wie das betrachtete Kontursegment selbst und setzen diese an den entsprechenden Kanten der Nachbarpixel fort.

Bedingungen (1) und (2) implizieren die Repräsentation als zyklische gerichtete Linked-Lists. Die Hinzunahme der letzten Bedingung impliziert genau eine unabhängige Linked-List je unabhängiger Kontur, in der auch alle zugehörigen Kontursegmente, 'sinnvoll' verknüpft, enthalten sind.

Wir zeigen als Erstes Forderung (3). Dafür sei bis auf Weiteres die Existenz eines Nachfolgers für jedes Kontursegment angenommen.

Wie im Abschnitt 9.1.6 beschrieben, verbindet jedes Kontursegment zwei Nachbarpixel (Hilfsatz 2) und zwar über die in Abschnitt 9.3.1 definierten Eintritts- und Austrittspunkte. Eine Fortsetzung in diesen Nachbarpixeln ist möglich, weil Austrittspunkte eines Pixels identisch sind mit Eintrittspunkten der Nachbarpixel und Eintrittspunkte eines Pixels identisch sind mit Austrittspunkten der Nachbarpixel (Hilfssatz 9).

Wenn ein auf ein Kontursegment s eines Pixels p nachfolgendes Kontursegment $suc(s)$ eines Pixels pn die geforderten Bedingungen erfüllen soll, muss $suc(s)$ gerade an dem Eintrittspunkt der Kante von pn beginnen, welcher dem Austrittspunkt von s aus p entspricht.

Dann setzt $suc(s)$ immer die entsprechende Kante des Nachbarpixels pn fort, auf der s in p endete. Folglich müssen beide zur gleichen Kontur gehören. Außerdem implizieren die für Kanten definierten Richtungen, dass $suc(s)$ die Richtung von s fortsetzt. Dementsprechend passen DST-Typ von c und SRC-Typ von cn zusammen. Wenn beispielsweise c den DST-Typ RIGHT hat (also in den rechten Pixel zeigt), ergibt sich für cn ein SRC-Typ `LEFT` (Herkunft aus linkem Pixel). Dagegen bleiben SRC-Typ von c und DST-Typ von cn offen. Wir halten, unter Hinzunahme von Hilfssatz 1 (Seite 70), fest:

Hilfssatz 17 *Wenn ein Kontursegment einen Nachfolger über ein zusammengehöriges Ein- Austrittspunktpaar findet, setzt der Nachfolger die Richtung des Kontursegments*

auf der verlängerten Kante fort. Damit gehören auch beide zur gleichen Kontur und es gibt keine 'Lücke' zwischen beiden.

Damit ist Forderung (3) gezeigt. Bleiben Forderungen (1, 2). Weil es für jeden Ein- und Austrittspunkt eines Pixels maximal ein Kontursegment geben kann (Hilfssätze 10, 14 und 15) folgt:

Hilfssatz 18 *Jedes Kontursegment kann je maximal einen Nachfolger und Vorgänger haben.*

Es bleibt zu zeigen, dass für jedes Kontursegment c gilt, dass die an seinem Ein- und Austrittspunkt angrenzenden Kontursegmente in Nachbarpixeln existieren müssen, wenn c existiert. Generell enden Kontursegmente immer an den Rändern eines Konturerzeugerblocks (Hilfssatz 13), d. h. alle anderen Ein- und Austrittspunkte bleiben ungenutzt. Ein Konturerzeugerblock k eines Pixels p, welcher ein Kontursegment c erzeugt, grenzt per Definition mindestens im Sinne der Achter-Nachbarschaft an diejenigen Pixel an, in denen c fortgesetzt wird (pn) oder herkommt (pp). Da Pixel pn und pp identisch wie p klassifiziert sind und durch k die Existenz angrenzender anders klassifizierter Pixel garantiert ist, müssen in pn und pp Kontursegmente existieren. Wir halten fest:

Hilfssatz 19 *Jeder Konturerzeugerblock k eines Pixels p ist, mindestens zum Teil, Teil je eines Konturerzeugerblocks in denjenigen Pixeln, in denen das zu k gehörige Kontursegment fortgesetzt werden muss. Damit ist deren Existenz garantiert.*

Die Lage des so garantierten Konturerzeugerblocks kn in pn ist derart, dass das in pn durch kn garantierte Kontursegment cn an dem Eintrittspunkt in pn beginnen muss, welcher dem Austrittspunkt von c in p entspricht.

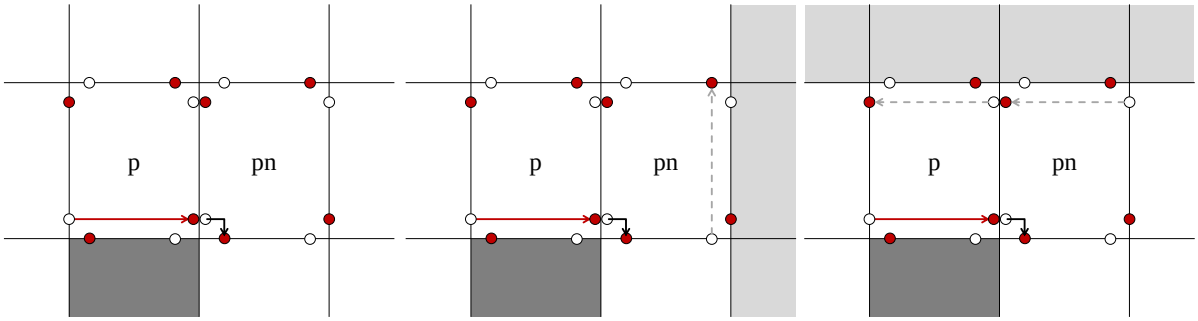
Das folgt daraus, dass die Definition des Beginns eines Konturerzeugerblocks p gleichzeitig die Definition des Endes eines Konturerzeugerblocks für den entsprechenden Nachbarpixel pn erfüllt (und umgekehrt). Das so garantierte vordere Stück von c und das hintere Stück von cn müssen sich somit treffen. Wir halten fest:

Hilfssatz 20 *Wenn alle Kontursegmente auf die in Abschnitt 9.2 formulierte Weise erzeugt werden, hat jedes Kontursegment genau einen eindeutigen Nachfolger in einem angrenzenden Pixel (erkennbar am SRC-Typ).*

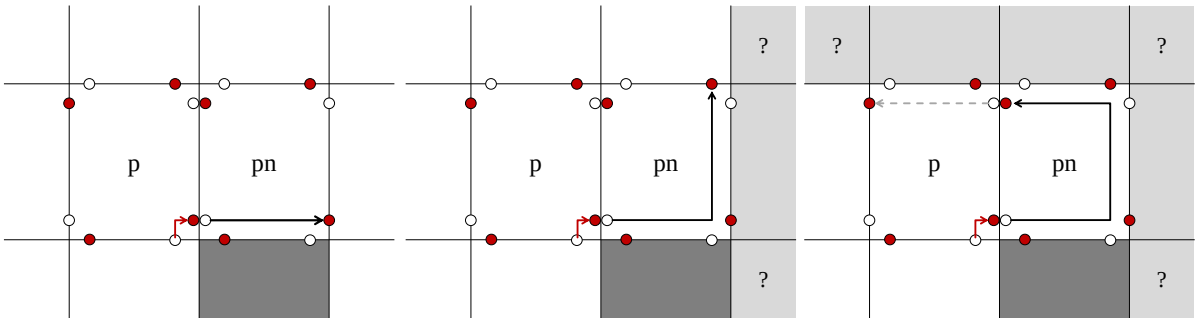
Die garantierte Existenz genau eines Nachfolgers impliziert, zusammen mit Hilfssatz 18, genau einen Vorgänger. Damit sind Forderungen (1) und (2) gezeigt.

Zur Veranschaulichung sind alle Fälle in denen ein Kontursegment c des DST-Typs `RIGHT` erzeugt wird in Abbildung 9.18 angegeben. Wie in Kapitel 9.2 beschrieben, existiert ein solches Kontursegment, wenn p und der rechts neben p liegende Pixel pn gleich klassifiziert sind und entweder der Pixel unter p (Abb. 9.18 (a)), der rechts unter p (Abb. 9.18 (b)), oder beide (Abb. 9.18 (c)) anders klassifiziert sind. In allen Fällen garantiert die Bedingung für ein Kontursegment des DST-Typs `RIGHT` in pn ein Kontursegment cn, welches gerade da anfängt, wo das Kontursegment in p aufhört. Dieses hat immer den SRC-Typ `LEFT`. Weitere anders klassifizierte Pixel um pn, die gegen den Uhrzeigersinn an die mindestens vorhandenen anders klassifizierten Umgebungspixel (gedanklich)

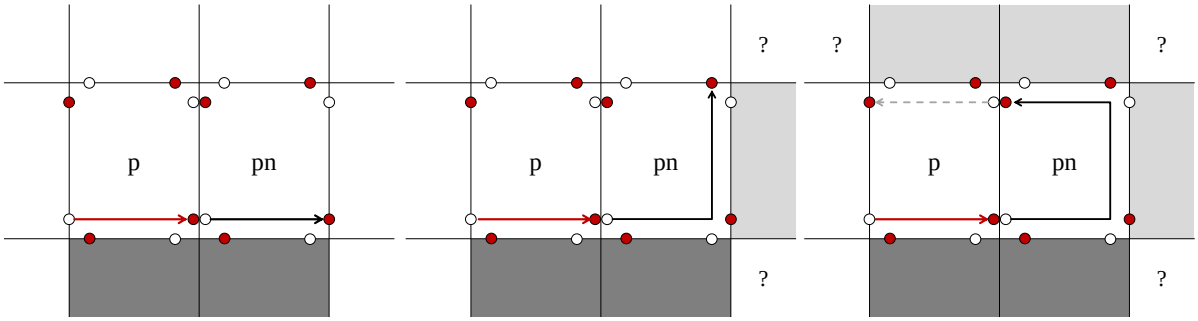
Abbildung 9.18.: Existenz eines Nachfolgers *cn* (schwarz) in Pixel *pn* für ein Kontursegment *c* (rot) des DST-Typs `RIGHT` in einem Pixel *p*. Pixel *p*, *pn* und auch weitere Umgebungspixel haben eine Klassifikation (weiß). Der Konturerzeugerblock, welcher das Kontursegment *c* erzeugt, ist anders als *p* klassifiziert und dunkelgrau markiert. Weitere anders als *p* klassifizierte Pixel sind hellgrau gefärbt. Teilabbildungen (a, b, c) zeigen je eine der drei möglichen Konfiguration, welche die Existenz von *c* implizieren. Weitere Legende: Rote/Weiße Punkte: Aus-/Eintrittspunkte, Gestrichelte Linien: Weitere, irrelevante, Kontursegmente in *p*/*pn*. Kontursegmente außerhalb von *p* und *pn* sind nicht eingezeichnet.



(a) Konturerzeugerblock von *c* besteht nur aus Pixel unter *p*. Dann kann *cn* nur einen Verlauf (DST-Typ DOWN) nehmen, da keine weiteren Pixel an den entsprechenden Konturerzeugerblock in *pn* angehängt werden können. Zusätzlich gezeigt: Weitere an *pn* angrenzende anders klassifizierte Pixel, die unabhängige Kontursegmente bewirken (Mitte / rechts)



(b) Konturerzeugerblock von *c* besteht nur aus Pixel rechts unter *p*. Weitere im UZS angehängte anders klassifizierte Pixel haben nur Einfluss auf den Ausgang von *cn* (Mitte/rechts). Klassifikation der mit '?' markierten, anders klassifizierten Pixel ist für das Gezeigte irrelevant.



(c) Konturerzeugerblock von *c* besteht aus den Pixeln unter und rechts unter *p*. Auf den Verlauf von *cn* hat der zusätzliche anders klassifizierte Pixel im Vergleich zu Teilabbildung (b) keinen Einfluss. Die mit '?' markierten Pixel wurden im Vergleich dazu invertiert.

angehängt werden, verzögern lediglich den Ausgang von cn (und ändern damit den DST-Typ von cn). Der spätestmögliche Ausgang ist der nach p. Er kann nicht 'blockiert' sein, da p und pn gleich klassifiziert sind. Schließlich wäre andernfalls die Voraussetzung für die Existenz eines Kontursegments des Typs `RIGHT` in p nicht erfüllt. Aufgrund der Rotationssymmetrie gilt das ebenso, wenn c einen anderen DST-Typ hätte.

Sei suc ein Array mit $4 \cdot N$ Einträgen, welches für jedes existierende Kontursegment i den Index des nachfolgenden Kontursegments $suc(i)$ enthalten soll. Analog müssen im gleichlangen Array pre Verweise auf Vorgänger gespeichert sein. Nun kann eine Funktion zum Bestimmen des Vorgängers $pre(s)$ und Nachfolgers $suc(s)$ jedes Kontursegments eines Pixels p formuliert werden. Diese ist gegeben im Pseudocode Abschnitt für Funktion linkContourSegs und berechnet pre und suc für alle Kontursegmente, die zu p gehören. Voraussetzung ist, dass Funktion initSegsTypes zuvor für p ausgeführt wurde.

Gemäß Hilfssatz 20 existieren für jedes Kontursegment passende Nachfolger und Vorgänger. Kontursegmente sind gemäß ihres DST-Typs im Array abgelegt, wie in Abschnitt 8.1 auf Seite 64 beschrieben. Dementsprechend ist die Bestimmung des Vorgängers (nur DST-Typ bekannt) einfacher als die des Nachfolgers (Nur SRC-Typ bekannt). Nur Ersterer kann mit den gegebenen Informationen direkt ermittelt werden. Ist der Vorgänger eines Kontursegments s berechnet, kann bei diesem der Verweis auf den Nachfolger gerade auf s gesetzt werden. Damit ist die Verlinkung abgeschlossen.

Function linkContourSegs(Pixel p)

```
// Precondition: initSegsTypes previously executed for p
// Compute pre and suc for all contour-segments of a pixel p
For Each Contour-Seg s, s ∈ {RS(p), LS(p), DS(p), US(p)} Do
    If existing(s) Then
        If src(s) = RIGHT Then // Check if s comes from pixel right of p
            // Predecessor is seg of DST-Type LEFT in Pixel right of p
            pre(s) ← LS(p + 1)
        Else If src(s) = LEFT Then // If s comes from pixel left of p
            pre(s) ← RS(p - 1)
        Else If src(s) = DOWN Then // If s comes from pixel under p
            pre(s) ← US(p + X)
        Else // If s comes from pixel above p
            pre(s) ← DS(p - X)
        End
        suc(pre(s)) ← s
    End
End
```

Insgesamt beschreiben die einzelnen Kontursegmente die Konturen vollständig (Satz 4, Seite 86) und können gemäß Hilfssätzen 17 und 20 geeignet verlinkt werden, sodass

gilt:

Satz 5 *Wenn alle Kontursegmente auf die in Abschnitt 9.2 formulierte Weise erzeugt werden und jedes seinen Nachfolger gemäß Funktion linkContourSegs bestimmt, ist jede Kontur durch eine eigenständige zyklische gerichtete Linked-List bestehend aus allen zugehörigen Kontursegmenten repräsentiert.*

9.5. Weitere Eigenschaften der Kontursegmente

In diesem Abschnitt werden die Eigenschaften `minDepth` und `ioCnt` der Kontursegmente definiert, um sie im entsprechenden Algorithmus für die Segmente initialisieren zu können. Deren genauer Verwendungszweck wird sich erst im Zusammenhang mit späteren Algorithmen erschließen und wird daher hier nur kurz skizziert.

9.5.1. Minimale Segmenttiefe

Eine weitere pro Kontursegment festzulegende Eigenschaft ist die minimale Segmenttiefe *minDepth*, die für Kontursegmente mit waagerechtem Anteil wichtig ist. Ein einzelner Pixel kann maximal zwei übereinander verlaufende Konturen enthalten, welche die korrespondierende Connected-Component zu den Bereichen oberhalb und unterhalb abgrenzen. Dementsprechend gibt es zwei mögliche Höhenkoordinaten je Pixel. Ein Segment kann, in Abhängigkeit des SRC-DST-Typs, eine oder beide beinhalten. Sei in der obersten Kante des Bildes die y-Koordinate 0 und die y-Achse zeige nach unten. Dann definieren wir die lokale minimale Segmenttiefe eines Segments eines SRC-DST-Typs innerhalb eines Pixels wie folgt:

- 0, falls (auch) die obere Kante durch das Segment repräsentiert wird
- 1, Sonst

In Abbildung 9.20 sind die minimalen Segmenttiefen aller SRC-DST-Kontursegmenttypen eingetragen. Für ein konkretes Kontursegment eines Pixels mit y-Koordinate y wird die minimale Segmenttiefe bei Erzeugung berechnet gemäß:

$$minDepth(y, SRC\_DST\_Typ) = 2 \cdot y + Local\_MD(SRC\_DST\_Typ) \tag{9.1}$$

Dabei sind für $Local\_MD$ jeweils die Werte gemäß Abbildung 9.20 in Abhängigkeit des SRC-DST Kontursegmenttyps zu verwenden.

9.5.2. Connected-Component Ein- und Austritts- Anzahl

In der finalen Füll-Phase des Algorithmus werden die Connected-Components mit dem Label der umschließenden Kontur gefüllt. Dazu wird eine Füllrichtung entweder als vertikal oder als horizontal definiert. Für diesen Teil des Algorithmus ist die Zustandsänderung, d.h. innerhalb oder außerhalb einer Connected-Component, für jeden Pixel

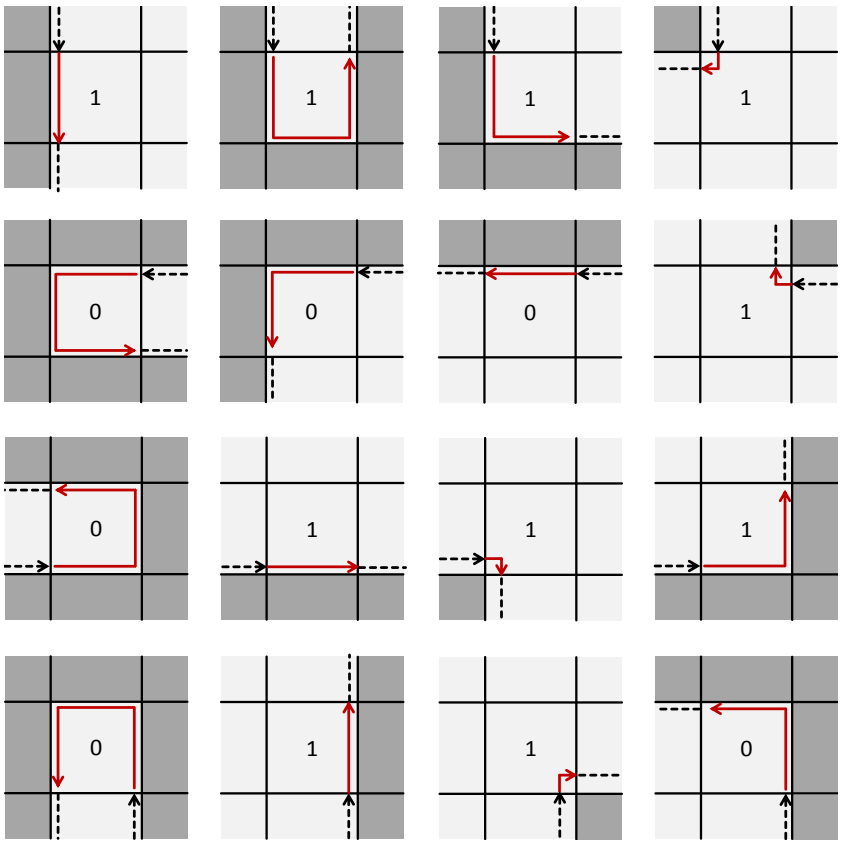


Abbildung 9.20.: Lokale minimale Segmenttiefen für alle SRC-DST-Kontursegmenttypen innerhalb eines Pixels.

zu bestimmen. Um diese später berechnen zu können, muss für jeden SRC-DST-Kontursegmenttyp und für jede mögliche Füllrichtung die Anzahl der Zustandsänderungen, genannt `ioCnt`, festgelegt werden. Diese ist für ein Kontursegment immer 0, 1 oder 2:

- 0, Connected-Component wird weder betreten noch verlassen (keine Zustandsänderung)
- 1, Connected-Component wird betreten oder verlassen (Zustandsänderung)
- 2, Connected-Component wird betreten und verlassen (keine Zustandsänderung)

Der Wert von `ioCnt` entspricht bei vertikaler Füllrichtung der Anzahl der waagerechten pixelkantenrepräsentierenden Konturstücke und bei horizontaler Füllrichtung der Anzahl der senkrechten pixelkantenrepräsentierenden Konturstücke. Schließlich kann nur durch jeweils solche Teilstücke eine Connected-Component betreten bzw. verlassen werden. Da Verbindungskonturstücke keine Pixelkanten repräsentieren ist `ioCnt` hier immer 0. In Abbildung 9.21 sind die Werte für `ioCnt` aller SRC-DST-Kontursegmenttypen sowohl für waagerechte als auch für senkrechte Füllrichtung eingetragen.

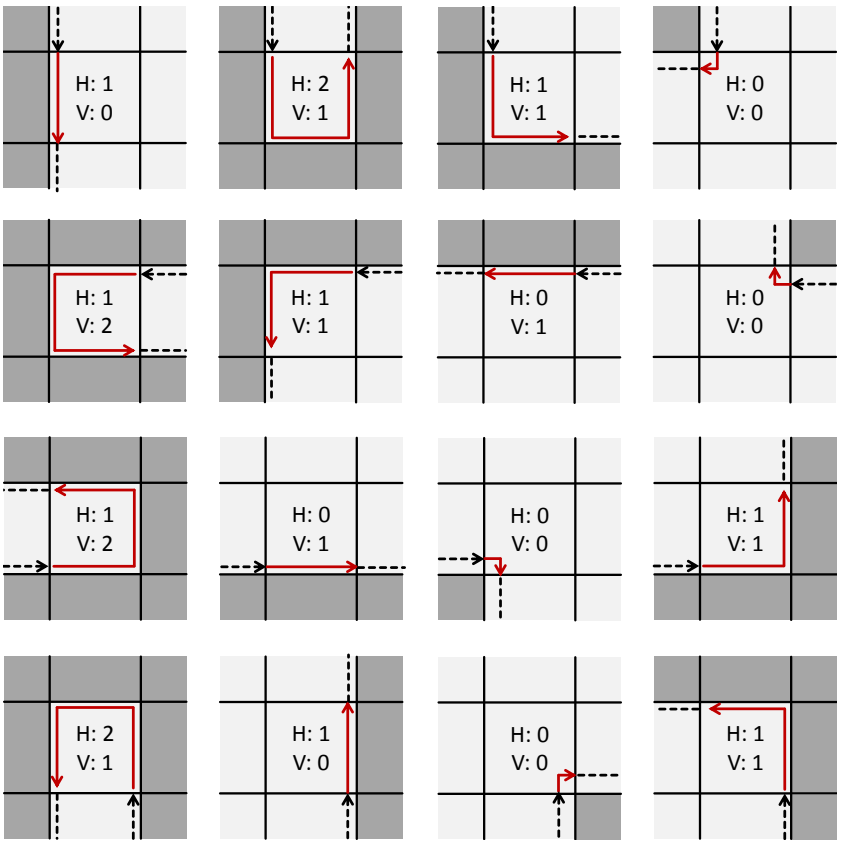


Abbildung 9.21.: Werte für `ioCnt` aller SRC-DST-Kontursegmenttypen für vertikale (V) und horizontale (H) Füllrichtung

9.6. Algorithmus

Unter Berücksichtigung aller Inhalte des Kapitels 9 kann nun der Algorithmus zum Extrahieren der Kontursegmente formuliert werden. Er ist im Pseudocodeabschnitt Algorithm 13 gegeben. Es werden alle Pixel parallel verarbeitet und sämtliche Berechnungen finden lokal auf Basis des jeweiligen Pixels und dessen Achter-Nachbarschaft statt. Das ist möglich, da es keine pixelübergreifenden Abhängigkeiten gibt. Damit liegen ideale Bedingungen für Parallelisierbarkeit vor.

Zusätzlich zu den bereits eingeführten Eigenschaften sind dort die Referenzen `head` und `tail` auf andere Kontursegmente aufgeführt, die später für das Konturlabeling mit datenunabhängigem Parallelitätsschema Verwendung finden. Sie verweisen anfänglich bei jedem Segment auf dieses selbst.

9.6.1. Asymptotisches Verhalten

Der Algorithmus 13 verwendet weder gleichzeitiges Lesen noch gleichzeitiges Schreiben eines Wertes und kann somit auf dem am wenigsten restriktiven EREW-PRAM ausgeführt werden. Für die folgende Untersuchung sei N die Anzahl der Pixel. Für jeden

Algorithm 13: ExtractContourSegments

```
For Each Pixel p, p ∈ {0, 1, ... N − 1} In Parallel Do
    // Init with default values
    For Each Contour-Seg c, c ∈ {RS(p), LS(p), DS(p), US(p)} Do
        existing(c) ← FALSE
        cLabel(c) ← UNLABELD
    End
    // Interesting code starts here
    If class(p) ≠ UNCLASSIFIED Then
        // Consider single pixel connected component special case
        If class(p) ∉ {class(p + 1), class(p − 1), class(p + X), class(p − X)}
        Then
            label(p) ← p
        End
        Call initSegsTypes(p) // See p. 80
        Call linkContourSegs(p) // See p. 90
        For Each Contour-Seg c, c ∈ {RS(p), LS(p), DS(p), US(p)} Do
            // Initialize further segment properties
            If existing(c) Then
                // See p. 91
                minDepth(c) ← Do: compute from y coord and src-dst type of c
                // See p. 91
                ioCnt(c) ← Do: compute from src-dst type of c
                // Initialize head and tail (described later) as self
                head(c) ← c
                tail(c) ← c
            End
        End
    Else
        label(p) ← UNLABELD
    End
End
```

Pixel und jedes der maximal vier einem Pixel zugeordneten Kontursegmente ist eine feste Anzahl an Operationen auszuführen. Damit ergibt sich Work zu $O(N)$. Es gibt keine Abhängigkeiten innerhalb des Algorithmus 13 zwischen den einzelnen Pixeln und den zu extrahierenden Kontursegmenten. Dementsprechend können $O(N)$ Prozessoren je einen Pixel verarbeiten sodass der Algorithmus in $O(1)$ Zeit terminiert. Alternativ können weniger Prozessoren eingesetzt werden, wobei sich die Laufzeit um einen entsprechenden Faktor erhöht. Zusammenfassend ergibt sich:

Machine : EREW-PRAM
Processors : $P \leq N$
Work : $O(N)$
Running-Time : $O(N/P)$
Cost : $O(N)$

9.7. Resultat der Kontursegmentextraktion

Nach Ausführung des Algorithmus 13 liegt jede Kontur jeder Connected-Component in Form einer unabhängigen zyklischen Linked-List aus Kontursegmenten vor. Diese erfüllen alle eingangs in diesem Kapitel geforderten Anforderungen. Dies ist beispielhaft in Abbildung 9.22 zu sehen. Gemäß der Vorschriften zur Verlinkung zu Linked-Lists wird für jedes Kontursegment gemäß DST-Typ (und damit entsprechend der Richtung der vordersten Pixelkante) ein Nachfolgersegment identifiziert, wie im Kapitel 9.4 beschrieben. Dementsprechend kann festgehalten werden:

Hilfssatz 21 *Die Richtung der einzelnen Kontursegmente je einer Kontur gemäß Pixelkanten und deren Verlinkung zu einer zyklischen gerichteten Linked-List korrelieren topologisch.*

Dies kann gut in Abbildung 9.22 gesehen werden. In weiteren Abbildungen dieser Arbeit werden aus Übersichtsgründen die Kontursegmente bzw. Konturen auf zwei verschiedene Arten dargestellt. Entweder wird das Pixelgitter mit Kontursegmenten und Richtungen gemäß Pixelkanten gezeigt, oder die Kontursegmente als Knoten von Linked-Lists unabhängig vom Pixelgitter und dessen Richtungen. Wird nur das Pixelgitter angezeigt, kann der Nachfolger eines Kontursegments trotzdem in Abbildungen 'optisch' bestimmt werden. Das ist aber nicht mit dem Verweis auf den Nachfolger (`suc`) zu verwechseln, insbesondere, da nachfolgende Algorithmen diesen manipulieren. Dagegen bleibt die Richtung der Kontursegmente, gegeben durch DST- und SRC-Typ, unveränderlich.

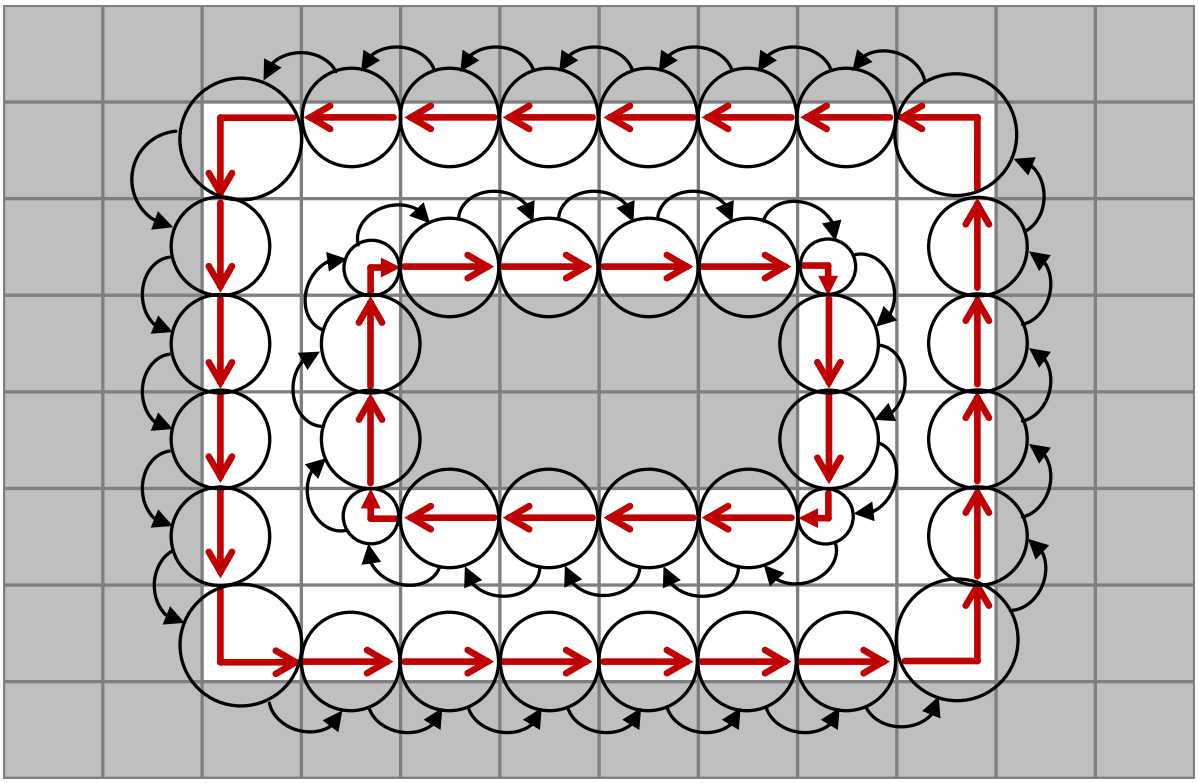


Abbildung 9.22.: Konturen einer Connected-Component repräsentiert durch zyklische Linked-Lists aus Kontursegmenten. Rote Pfeile: Kontursegmente mit Richtung gemäß repräsentierten Pixelkanten und beschrieben durch SRC-DST-Typ. In dieser 'räumlichen' Sicht im Pixelgitter haben Kontursegmente auch eine Koordinate (p.x, p.y). Kreise: Sicht auf dieselben Kontursegmente als Knoten von Linked-Lists. Schwarze Pfeile: Verlinkung der Knoten in den Linked-Lists, gegeben durch `suc`. In dieser Sicht steht die Topologie im Vordergrund. In weiteren Abbildungen in dieser Arbeit wird immer nur eine der beiden Sichtweisen verwendet, die jeweils anderen Informationen liegen aber immer vor.

Kapitel 10.

Labeling der Konturen

Dieses Kapitel befasst sich mit dem zweiten Schritt, dem Labeln der vollständigen Konturen, welcher für den vorgestellten Connected-Component-Labeling-Algorithmus erforderlich ist.

10.1. Ausgangssituation

Ausgangssituation ist das Ergebnis des vorherigen Kontursegmentextraktionsalgorithmus, der in Kapitel 9 beschrieben ist. Dabei ist für jede Kontur eine gerichtete zyklische Linked-List aus Kontursegmenten extrahiert worden. Jedes Kontursegment hat einen Verweis auf seinen unmittelbaren Nachfolger und ggf. Vorgänger erhalten.

Zusätzlich werden die in Kapitel 9 auf Seite 66f geforderten Bedingungen bzw. Eigenschaften erfüllt. Sie werden weiterhin durch die Datenstrukturen repräsentiert, welche in Abschnitt 9.6 zusammengefasst wurden.

Bis jetzt liegen allerdings noch keinerlei Informationen über die Kontur als Ganzes vor. Die vollständigen Konturen zu analysieren und abschließend zu labeln ist Gegenstand dieses Kapitels und der darin vorgestellten Subalgorithmen. Dazu werden zwei Ansätze vorgestellt, welche das Labeling-Problem alternativ oder im Zusammenspiel lösen können. Primäres Unterscheidungsmerkmal dieser alternativen Subalgorithmen ist die Verwendung von datenabhängiger Parallelität.

10.2. Variante I: Datenunabhängiges Parallelitätsschema

Ziel dieses Ansatzes ist die Maximierung der Parallelität, welche unabhängig von den Daten ist und nur auf den bekannten Eigenschaften der Kontursegmente basiert. Welche Kontursegmente parallel zu verarbeiten sind, kann allein anhand deren Position im Pixelgrid festgelegt werden. Somit ist keinerlei zusätzlicher Aufwand hinsichtlich einer vorherigen Datenanalyse zu betreiben. Dieser Ansatz wird formuliert, um optimale Kosten zu erreichen und zusätzlich den konstanten Vorfaktor möglichst gering zu halten. Dies wird sich im Implementationsteil für nicht massiv parallele Hardware als vorteilhaft erweisen.

10.2.1. Vereinigung einzelner Kontursegmente bzw. Kontursegmentzüge

Um Informationen über die vollständigen Konturen zu aggregieren und eine Kontur durch eine einzelne Entity zu repräsentieren, werden bei diesem Ansatz in einem ersten Schritt zusammengehörige Kontursegmente solange vereinigt, bis jeweils genau ein Kontursegment verbleibt. Dieses letzte Kontursegment repräsentiert dann die gesamte Kontur. Nach der Vereinigung von zwei oder mehreren Kontursegmenten entstehen zunächst Linienzüge, welche genau zwei Enden haben. Diese werden durch die Felder `head` und `tail` der Datenstruktur für Kontursegmente repräsentiert.

Bei der Vereinigungsoperation müssen lediglich die Enden zweier Linienzüge angepasst werden. Demnach ist die Vereinigung zweier Linienzüge, unabhängig von deren Komplexität, immer mit konstanten Kosten verbunden, da Linien per Definition genau zwei Enden haben. Daraus folgt aber auch, dass die Verweise auf die Enden im Inneren eines Konturlinienzuges nicht angepasst werden können, wenn diese Eigenschaft erhalten bleiben soll.

Wenn nur die Enden eines Linienzuges für die hier bedeutsame Vereinigungsoperation entscheidend sind, kann ein Linienzug durch ein einzelnes Kontursegment am Ende repräsentiert werden, welches mit den Feldern `head` und `tail` auf die an den Enden gelegenen Kontursegmente verweist. Eines davon ist offensichtlich immer ein Verweis auf das Kontursegment selbst und bei unvereinigten Segmenten handelt es sich bei beiden um Selbstreferenzen, so wie sie im Algorithmus 13 auf Seite 94 initialisiert wurden. Die Vereinigungsoperation ist durch die Funktion unifyOp beschrieben.

```
Function unifyOp(Contour-segment cs)
    If existing(cs) Then
        // If Contour not already closed
        If cs ≠ head(suc(cs)) Then
            // Mark one CS as already processed
            marked(suc(cs)) ← TRUE
            // Apply new other-ends to edge ContourSegments
            // Set head-reference of rear CS's tail to fore CS's head
            head(tail(cs)) ← head(suc(cs))
            // Set tail-reference of fore CS's head to rear CS's tail
            tail(head(suc(cs))) ← tail(cs)
            // Aggregate minDepth
            minDepth(cs) ← Call min(minDepth(cs), minDepth(suc(cs)))
        End
    End
```

Diese vereinigt ein Kontursegment (`cs`) mit dessen Nachfolger `cs.suc`, unabhängig davon, ob diese jeweils ein einzelnes Segment oder bereits einen Konturlinienzug repräsentieren. Hinweis: `cs.suc`, etc. ist eine alternative Schreibweise für suc(cs) usw. außerhalb der Algorithmen. Die Funktion unifyOp wird für alle potentiellen Kontursegmente aufgerufen, führt aber lediglich bei existierenden Operationen aus. Zuerst wird überprüft, ob `cs` sich mit sich selbst vereinigt. Andernfalls wird eines der beiden Kontursegmente (das Hintere) markiert. Weil für jedes Kontursegmente einmal die Funktion unifyOp aufgerufen wird, und nur wenn die Kontur sich schließt, kein Segment markiert wird, bleibt am Ende genau ein Kontursegment je vollständiger Kontur übrig, welches nicht markiert ist.

Außerdem werden die Referenzen von `head` und `tail` an den Rändern des aus der Vereinigung resultierenden Linienzuges angepasst. Ferner erhält ein Kontursegment (das Vordere und unmarkierte) den Wert des Minimums des Feldes `minDepth` von Beiden. Folglich hat das am Ende verbleibende (nicht markierte) Segment hier das Minimum über alle Kontursegmente der Kontur, zu der es gehört, aggregiert.

Abbildung 10.1 veranschaulicht die Anwendung Funktion unifyOp auf ein Kontursegment `cs`, welches bereits einen Kontursegmentzug repräsentiert und dessen Nachfolger ebenfalls einen Kontursegmentzug repräsentiert.

10.2.2. Mögliche Konflikte und deren Verhinderung

Die im vorherigen Abschnitt vorgestellte Funktion unifyOp muss für alle Kontursegmente genau einmal aufgerufen werden. Ziel des parallelen Algorithmus ist dabei die gleichzeitige Verarbeitung möglichst vieler Kontursegmente. Dies kann zu Konflikten führen, da nicht nur das Kontursegment `cs` durch die Funktion unifyOp modifiziert wird, sondern auch andere an den Rändern der zu verknüpfenden Kontursegmentzüge. Diese sind: `cs`, `cs.suc`, `cs.tail` und `cs.suc.head`. Davon sind lediglich `cs` und `cs.suc` von Anfang an

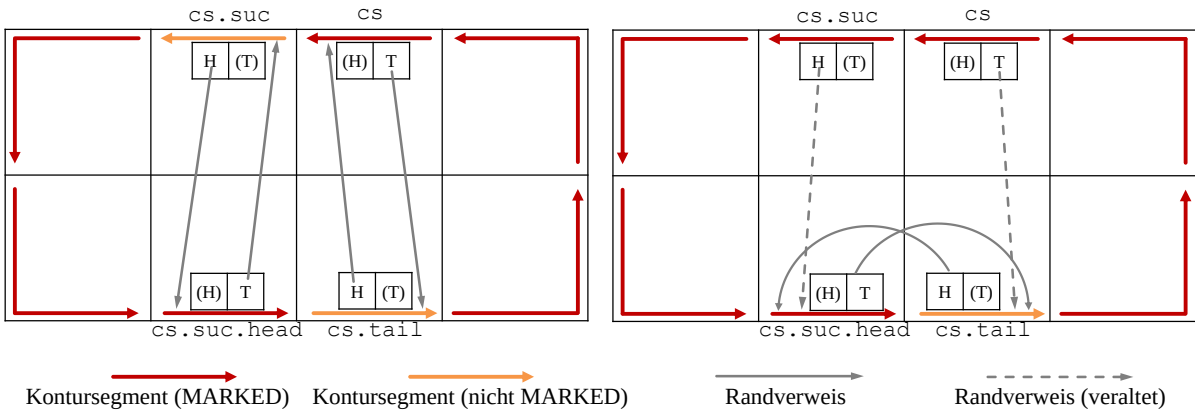


Abbildung 10.1.: Anwendung der Funktion unifyOp auf ein Kontursegment `cs`. H/T: Verweis auf head/tail des Konturlinienzuges. In Klammern: Selbstverweis. Links: Ausgangssituation, rechts: Ergebnis

bekannt und bleiben im Gegensatz zu `cs.tail` und `cs.suc.head` unverändert während der Ausführung dieses Subalgorithmus.

Ein Konflikt entsteht genau dann, wenn `cs` und `cs.suc.head` nicht sequentiell verarbeitet werden. Beispielsweise würde bei gleichzeitiger Ausführung dann cs.tail.head auf den noch unveränderten Wert `cs.suc.head` gesetzt. Das richtige Ergebnis beider Vereinigungen ist aber die Zuweisung des Segments, welches vor der Operation den Index `cs.suc.head.suc.head` hat. Folglich muss der parallele Algorithmus die sequentielle Verarbeitung von `cs` und `cs.suc.head` durch die Funktion unifyOp sicherstellen. Weil `head` und `tail` durch Anwendung der Funktion unifyOp auf andere Kontursegmente im Laufe dieses Subalgorithmus ständig verändert werden können, handelt es sich hierbei um ein datenabhängiges Synchronisationsproblem.

Die Verarbeitung von `cs.suc` und `cs.tail` muss nicht explizit serialisiert werden, da durch diese keine zusätzlichen Konflikte entstehen können. Wenn `cs.suc` identisch ist mit `cs.suc.head` bzw. `cs.tail` mit `cs`, handelt es sich offensichtlich nicht um einen eigenen Konflikt, da dieser bereits überprüft wird. Falls `cs.suc` verschieden ist von `cs.suc.head`, muss die Funktion unifyOp bereits in einem vorherigen Zeitschritt auf `cs.suc` angewandt worden sein. Da unifyOp für jedes Kontursegment genau einmal ausgeführt wird, kann sie zu diesem Zeitpunkt nicht mehr für `cs.suc` angewandt werden und somit ist ein Konflikt ausgeschlossen. Gleiches gilt für den Fall wenn `cs.tail` von `cs` verschieden ist.

10.2.3. Tile, Tile-Pair und Operationen darauf

Zur Auflösung der beschriebenen datenabhängigen Konflikte wird in diesem Ansatz ein Parallelitätsschema verwendet, welches so viele Kontursegmente parallel verarbeitet, wie datenunabhängig erkannt werden können. Dabei wird ausgenutzt, dass die anfängliche Lage des Nachfolgesegments `suc` eines jeden Kontursegments in einem direkt angrenzenden Pixel ist.

Dementsprechend kann das Pixelgrid in verschiedene zusammenhängende Bereiche, die **Tiles**, aufgeteilt werden, welche unabhängig voneinander verarbeitet werden können. Ein Tile und dessen Eigenschaften seien nun wie folgt definiert:

- Ein Tile repräsentiert einen rechteckigen Ausschnitt des Pixelgitters
- Ein Tile besteht aus mindestens einem und maximal N Pixeln
- Jeder Pixel des Gitters ist zu jedem Zeitpunkt genau einem Tile zugeordnet
- Alle Kontursegmente, die innerhalb eines Tiles vereinigt werden können, sind vereinigt
- Alle Kontursegmente, deren Nachfolger nicht innerhalb desselben Tiles liegen, sind nicht vereinigt

Außerdem wird ein Tile-Pair folgendermaßen definiert:

- Ein Tile-Pair besteht aus zwei Tiles mit einer gemeinsamen Kante
- Zu keinem Zeitpunkt ist ein Tile in mehr als einem Tile-Pair enthalten

Ferner wird die Funktion mergeTilePair definiert, welche ein Tile-Pair in ein Tile überführt. Dazu müssen alle noch unvereinigten Kontursegmente im Inneren des Tile-Pairs

Function mergeTilePair(Tile-Pair tp)

```
// CS forthSeg is an array of indices of contour-segments of the
// first tile, which are contained in a Pixel located near the edge
// between both tiles, where DST-Type Points to the second tile.
// Identification simply depends on contour-segments' index
CS forthSegs ← Do: get indices of to be processed segs of first tile of tp
// As above, just vice-versa
CS backSegs ← Do: get indices of to be processed segs of second tile of tp
For Each Contour-segment cs ∈ forthSegs In Parallel Do
  | Call unifyOp(cs); // Calls function unifyOp (see page 99)
End
// Later called: Minor iterations
For Each Contour-segment cs ∈ backSegs In Order Do
  | Call unifyOp(cs);
End
```

vereinigt werden. Diese befinden sich in Pixeln an der gemeinsamen Kante beider enthaltener Tiles und haben einen DST-Typ, der in Richtung des jeweils anderen Tiles zeigt. Dadurch sind die Indices der in Frage kommenden Kontursegmente eindeutig festgelegt. Bei der parallelen Verarbeitung der Kontursegmente der Tiles eines Tile-Pairs durch die Funktion unifyOp kann es jedoch zu Konflikten kommen.

Als erstes können alle Kontursegmente eines der beiden Tiles konfliktfrei parallel verarbeitet werden, solange noch kein einziges Kontursegment des anderen Tiles verarbeitet ist. Ursächlich dafür ist die Lage aller Nachfolger und deren Heads im anderen Tile. Diese Situation ist in Abbildung 10.2 (a) veranschaulicht.

Die Garantie gilt für den Head des Nachfolgers jedoch nur so lange, wie kein Segment des anderen Tiles verarbeitet wurde. Deshalb können Konflikte auftreten, wenn anschließend das zweite Tile verarbeitet wird. Die Art der Konflikte hängt nun vom Verlauf der Konturen ab und ist damit datenabhängig. Abbildung 10.2 (b) veranschaulicht diese Situation. Der hier vorgestellte Ansatz sieht nicht vor, diese Konflikte aufzulösen und wendet deshalb die Funktion unifyOp sequentiell auf alle zu verarbeitenden Kontursegmente des zweiten Tiles an. Für spätere Teile des Algorithmus ist es wichtig, dabei eine feste Reihenfolge zu verwenden. Eine Möglichkeit ist die Verarbeitung der Kontursegmente in einer Reihenfolge gemäß aufsteigender Pixelposition in x bzw. y.

Nachdem die Funktion unifyOp auf alle Kontursegmente des ersten Tiles des Tile-Pairs, deren Nachfolger im zweiten Tile liegen und alle Kontursegmente des zweiten Tiles, deren Nachfolger im ersten Tile liegen, angewandt wurde, endet die Funktion `mergeTilePair`. Damit ist aus dem Tile-Pair ein Tile geworden, wie in Abbildung 10.2 (c) zu sehen.

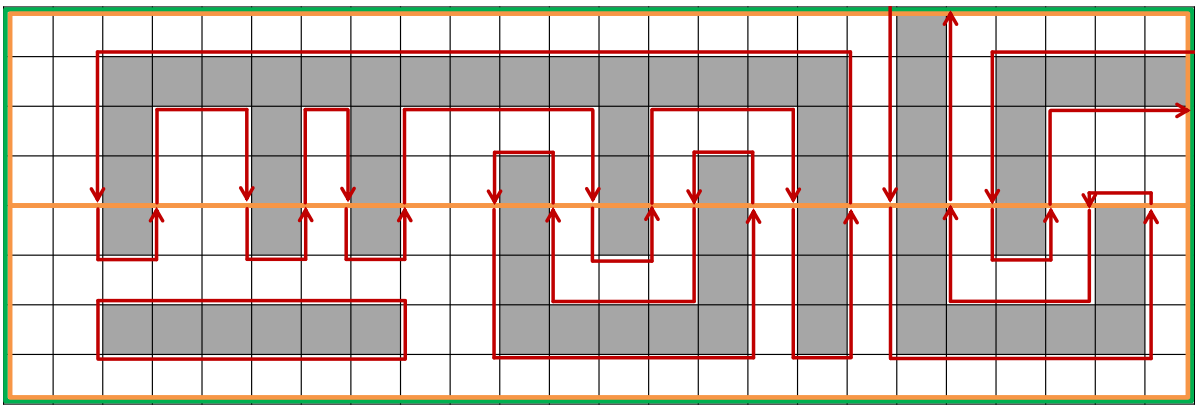
10.2.4. Tile-basierter Algorithmus

Auf Basis der zuvor definierten Tiles, Tile-Pairs sowie den Funktionen mergeTilePair und unifyOp kann nun ein Algorithmus zum Vereinigen aller Kontursegmente je Kontur formuliert werden. Anfänglich, nachdem der Algorithmus 13 aus Abschnitt 9.6 ausgeführt wurde, sind Pixel- und Tile-Gitter identisch. Jeder Pixel ist also ein Tile. Solange es mehr als ein Tile gibt, können alle Tiles zu Tile-Pairs gruppiert werden. Anschließend kann die Funktion mergeTilePair für alle Tile-Pairs parallel ausgeführt werden, ohne Konflikte zwischen Tile-Pairs zu bewirken. Ein Algorithmus zur Vereinigung aller Kontursegmente zu geschlossenen Konturen auf Basis der Tiles besteht nun darin, solange alle Tiles in Tile-Pairs anzuordnen und anschließend die Funktion mergeTilePair auf diese anzuwenden, bis alle Tiles zu einem einzigen vereinigt sind. Aus der Definition eines Tiles folgt, dass dann alle Kontursegmente verarbeitet sind.

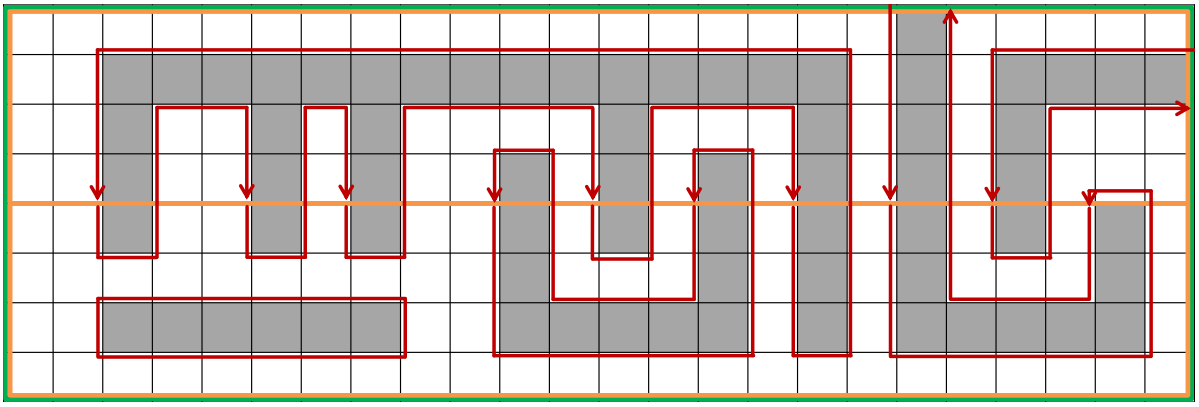
Sei eine Major-Iteration des Algorithmus das Gruppieren der Tiles zu Tile-Pairs und das anschließende parallele Aufrufen der Funktion mergeTilePair für diese. Ferner seien Minor-Iterationen die sequentiellen Schritte entlang der gemeinsamen Pixelkante bei Vereinigung eines Tile-Pairs. Für den Algorithmus kann nun für jede Major-Iteration ein festes Schema formuliert werden, nachdem die Tiles zu Tile-Pairs gruppiert werden. In Abhängigkeit der Kantenlängen X und Y des Pixelgitters kann dieses mit z.B. dem Ziel spezifiziert werden, die Anzahl sequentieller Schritte, nämlich der Summe über die Minor-Iterationen aller Major-Iterationen, zu minimieren.

Der Sub-Algorithmus 14 zeigt ein geeignetes Schema für den Fall, welches diese Bedingung erfüllt. Dabei halbiert sich in jeder Major-Iteration die Anzahl der Tiles, sofern in der entsprechenden Dimension eine Zweierpotenz von Tiles nebeneinander bzw. übereinander liegt. In diesem Fall terminiert der Sub-Algorithmus 14 offensichtlich nach

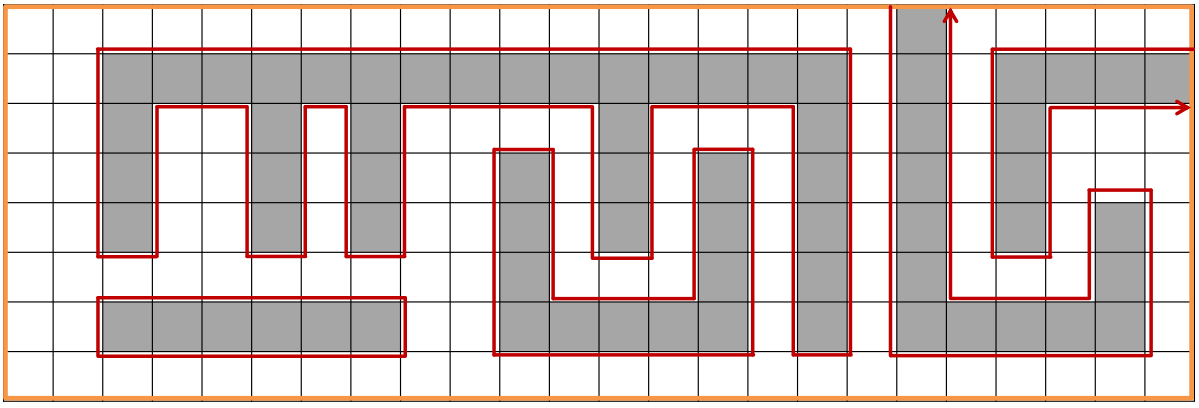
Abbildung 10.2.: Veranschaulichung der Merge-Operation zweier Tiles (orange Kästen) eines Tile-Pairs (grüner Kasten). Bereits vereinigte Segmente sind als durchgezogene Linie dargestellt. Enden mit Pfeilspitzen sind unvereinigt und zeigen auf das Nachfolgesegment, sofern innerhalb des Tile-Pairs. Hinweis: Abbildungsrand ist hier nicht Pixelgitterrand



(a) Ausgangssituation. Alle Segmente eines Tiles, hier das Untere gewählt, deren Nachfolger im anderen Tile liegen, können parallel vereinigt werden



(b) Bei anschließender Verarbeitung der Segmente des oberen Tiles, deren Nachfolger im Unteren liegen, sind Konflikte möglich. In diesem Beispiel ist lediglich das zweite zu vereinigende Segment von rechts konfliktfrei. Deshalb werden immer alle Kontursegmente des zweiten Tiles sequentiell vereinigt.



(c) Ergebnis der Anwendung der Funktion `mergeTilePair` auf ein Tile-Pair ist ein Tile

Algorithm 14: unifyTiles

```
remainingTiles ← Do: view pixel-grid as tiles
Tile-Pairs tpa
// Major iterations
Do ⌈log2(X)⌉ + ⌈log2(Y)⌉ times
    If remainingTiles consists of more tiles in x than in y Then
        tpa ← Do: view remainingTiles as tile-pairs side by side
    Else
        tpa ← Do: view remainingTiles as tile-pairs one upon the other
    End
    For Each Tile-Pair tp ∈ tpa In Parallel Do
        Call mergeTilePairs(tp)
        Do: remove one tile from tp and double the other one's size
    End
End
```

$log_2(X) + log_2(Y) = log_2(N)$ Major-Iterationen. Handelt es sich nicht um Zweierpotenzen, kann die Problemgröße in jeder Dimension um einen Faktor < 2 überschätzt werden, um eine Zweierpotenz zu erhalten. Dazu kann das Pixelgrid - gedanklich - um leere Einträge erweitert werden. Aus dieser Überschätzung ergibt sich weiterhin in jeder Dimension ggf. eine Major-Iteration zusätzlich, sodass es allgemein insgesamt $\lceil log_2(X) \rceil + \lceil log_2(Y) \rceil$ sind.

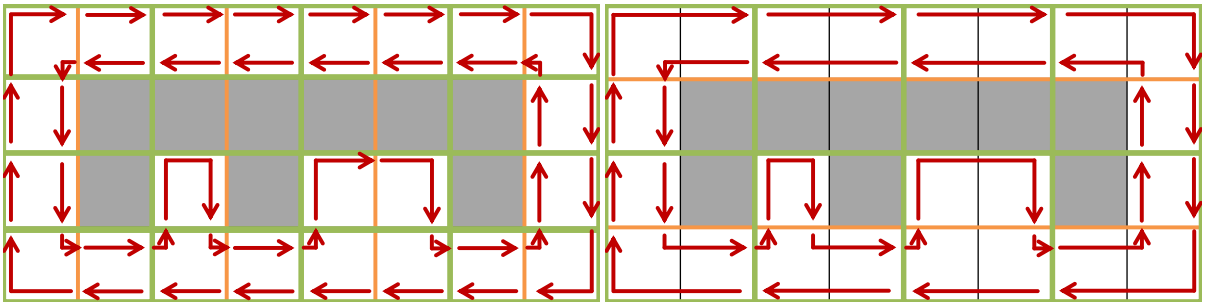
Es sei an dieser Stelle darauf hingewiesen, dass es sich bei den Tiles und Tile-Pairs nicht um Datenstrukturen handelt. Stattdessen bestimmt der Algorithmus allein anhand der Datenauflösung X und Y die Indices der in den einzelnen Major- und Minor-Iterationen zu verarbeitenden Kontursegmente nach einem festen Schema.

Zur Veranschaulichung kann die Abbildung 10.4 dienen. Hier sind alle Major-Iterationen des Sub-Algorithmus 14 für ein 8×4 Pixel großes Gitter aufgeführt. Der gezeigte Datensatz hat, aufgrund der Datenunabhängigkeit, auf das Prinzip keinen Einfluss.

10.2.5. Labeln der nicht markierten Kontursegmente

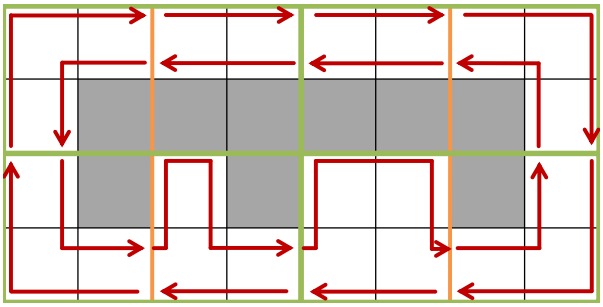
Nachdem der im vorherigen Abschnitt beschriebene Subalgorithmus 14 für alle Kontursegmente die Funktion unifyOp ausgeführt hat, verbleibt je vollständiger Kontur genau ein Kontursegment, welches nicht '`marked`' ist. Dieses enthält außerdem das Minimum über alle Werte von `minDepth` der entsprechenden Kontur in seinem Feld `minDepth`. Basierend auf dieser Information kann entschieden werden, ob ein Segment zu einer äußeren Kontur gehört. Falls dies so ist, erhält es ein Label. Dafür kann z.B. die 1D-Speicheradresse des zugehörigen Pixels des Kontursegments verwendet werden, welche offensichtlich für Pixel eindeutig ist. Der Test auf Zugehörigkeit zu einer äußeren Kontur wird später im Kapitel 11.1 ab Seite 131 erläutert. Der in Pseudocode-Abschnitt 15 gegebene Algorithmus versieht alle existierenden Kontursegmente, die nicht '`marked`' sind, mit einem Label, sofern sie zu einer äußeren Kontur gehören.

Abbildung 10.4.: Alle $\lceil log_2(8) \rceil + \lceil log_2(4) \rceil = 5$ Major-Iterationen des Sub-Algorithmus 14 für ein 8×4 Pixelgitter. Tile-Pairs sind durch grüne Kästen und deren Aufteilung in Tiles durch orange Linien gegeben. In jedem Teilbild sind Tile-Pairs eingetragen, welche im nächsten Teilbild zu einem Tile vereinigt sind. Durchgezogene Linien stellen bereits vereinigte Kontursegmente dar. Enden mit Pfeilspitzen sind unvereinigt und zeigen auf das Nachfolgesegment. In der Ausgangssituation (a) entspricht jeder Pixel einem Tile. Nach Ablauf alle Major-Iterationen repräsentiert ein Tile alle Pixel (f). Das impliziert die Verarbeitung sämtlicher Kontursegmente.

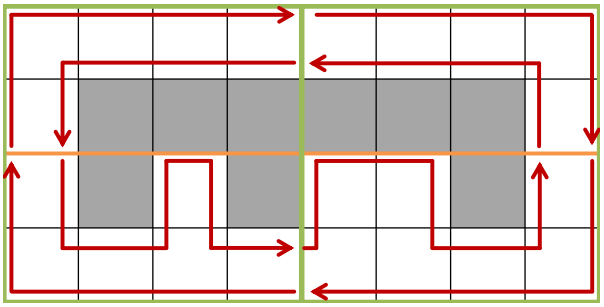


(a) Ausgangssituation

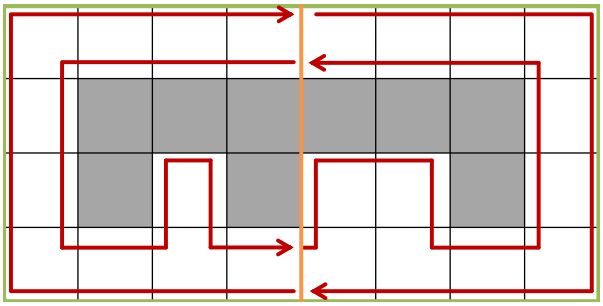
(b) Tiles 1 · x-merged



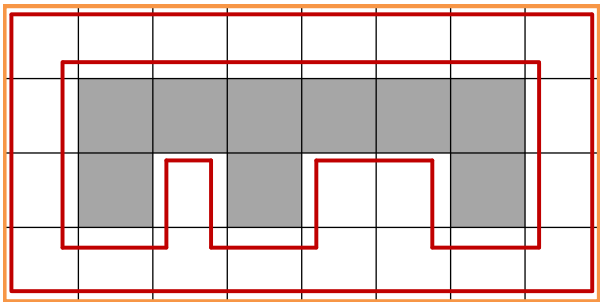
(c) Tiles 1 · x-merged und 1 · y-merged



(d) Tiles 2 · x-merged und 1 · y-merged



(e) Tiles 2 · x-merged und 2 · y-merged



(f) Tile 3 · x-merged und 2 · y-merged

Algorithm 15: setLabel

```
For Each Contour-segment c, c ∈ {0, 1, ..., 4 · N − 1} In Parallel Do
    If existing(c) ∧¬ marked(c) Then
        // If c belongs to an outer contour
        If minDepth(c) mod 2 = 0 Then
            // Set pixel index as label
            cLabel(c) ← c mod N
        End
    End
End
```

10.2.6. Weiterreichen der Label an alle Kontursegmente

Nachdem alle Kontursegmente äußerer Konturen, die nicht 'marked' sind, ein Label erhalten haben, müssen diese Label an alle Kontursegmente der jeweiligen Konturen weitergereicht werden. Dazu wird das gesamte Parallelitätsschema, welches zur Vereinigung der Kontursegmente genutzt wird, exakt invers ausgeführt. Nur auf diese Weise können die ansonsten veralteten Informationen über die Enden, `head` und `tail`, im Inneren der Konturlinienzüge genutzt werden.

Um dies zu erreichen, wird zum einen das Tile-Schema, welches in Sub-Algorithmus 14 gegeben ist, invertiert. Es beginnt demnach mit einem einzigen Tile, welches alle Pixel beinhaltet und überführt es im ersten Schritt in ein Tile-Pair. Resultat ist ein Tile-Gitter, das dem Pixelgitter entspricht. Dieses inverse Tile-Schema nennen wir `passLabels`.

Zum anderen muss die Verarbeitung der einzelnen Kontursegmente beim Split eines Tiles zu einem Tile-Pair exakt invers zu der in Funktion mergeTilePair gezeigten verlaufen. Diese zu mergeTilePair inverse Funktion sei als `splitTile` bezeichnet. Darin läuft die Schleife umgekehrt zu der in Funktion mergeTilePair und anstelle der Funktion unifyOp wird die Funktion passLabelOp aufgerufen. Diese reicht das Label an ein abzusplittendes Kontursegment weiter, welches im Vereinigungsprozess mit dem Aktuellen vereinigt wurde.

Function passLabelOp(Contour-segment cs)

```
If existing(cs) ∧ cLabel(cs) ≠ UNLABELD Then
    cLabel(suc(cs)) ← cLabel(tail(cs))
End
```

10.2.7. Der gesamte Algorithmus

Nachdem die Funktion passLabelOp gemäß des obigen Schemas für alle Kontursegmente aufgerufen wurde, ist für alle Kontursegmente bekannt, ob es sich um eine innere Kontur

handelt und alle Kontursegmente äußerer Konturen haben ein Label. Der gesamte Algorithmus zum Labeln der Konturen mit einem datenunabhängigen Parallelitätsschema ist im Pseudocodeabschnitt Algorithmus 16 gegeben.

Algorithm 16: LabelContoursStatic

```
Execute unifyTiles // See p. 104
Execute setLabel // See p. 104
Execute passLabels // See p. 106
```

10.2.8. Asymptotisches Verhalten

Der Algorithmus 16 verwendet weder gleichzeitiges Lesen noch gleichzeitiges Schreiben eines Wertes und kann somit auf dem am wenigsten restriktiven EREW-PRAM ausgeführt werden. Für die folgende Untersuchung sei N die Anzahl der Pixel, außerdem seien X, die Bildauflösung in der x-Dimension, und Y, die Bildauflösung in der y-Dimension, beide Zweierpotenzen. Dies lässt sich im Vergleich mit einer beliebigen Auflösung durch Überschätzung um einen konstanten Faktor < 4 erreichen. Da die folgenden Betrachtungen vor allem für die abschließende Bewertung des asymptotischen Verhaltens bedeutsam sind, ist dieser irrelevant. Ferner sei die Ausdehnung eventuell in einer Dimension, o.E.d.A. der x-Dimension, um Faktor k, mit $k \geq 1$, größer als in der anderen Dimension. Je Pixel genügt zur Modellierung der Konturen eine konstante Anzahl, das sind vier, Kontursegmente. Die übrigen Eigenschaften, welche in den nachfolgenden Abschnitten genauer erläutert werden, sind:

$$\begin{aligned} \textbf{Machine} &: \text{EREW-PRAM} \\ \textbf{Processors} &: O(\min(X, Y)) \\ \textbf{Work} &: O(N) \\ \textbf{Running-Time} &: O(\max(X, Y)) \\ \textbf{Cost} &: O(N) \end{aligned}$$

Work

Eine Linie hat, unabhängig von ihrer Komplexität, genau zwei Enden. Zur Vereinigung zweier Linien müssen immer nur die Enden angepasst werden, sodass diese Operation konstante Kosten verursacht. Jedes Kontursegment wird genau einmal mit seinem Nachfolger vereinigt, siehe Funktion unifyOp, und gibt genau einmal seine Informationen mit der Funktion passLabelOp an den Nachfolger weiter. Außerdem erhält jedes Segment maximal einmal ein Label mit Sub-Algorithmus 15 (`setLabel`). Alle diese Operationen werden anhand lokaler Informationen ausgewertet und verursachen feste Kosten. Folglich ergibt sich insgesamt für den Algorithmus 16 eine lineare Anzahl von Operationen.

Analyse der Parallelität und der Operationen je Major-Iteration, sowie der Anzahl sequentieller Schritte je Major-Iteration und insgesamt

Bevor eine Analyse von Running-Time und Cost erfolgen kann, ist eine genauere Analyse der erreichbaren Parallelität und damit verbundener Anzahl sequentieller Schritte des Sub-Algorithmus `unifyTiles` nötig, da dieser die Laufzeit begrenzt. Die Laufzeiten von `unifyTiles` und `passLabels` sind aufgrund des analogen Verarbeitungsschemas identisch, sodass es genügt, einen zu untersuchen. Die Laufzeit des Subalgorithmus `setLabel` ist bei N Prozessoren offensichtlich $O(1)$ und limitiert damit nicht.

Eine Operation wird in der Einheit 'verarbeitete Pixel' angegeben, da die Anzahl der Berechnungen je Pixel fest ist. Die Parallelität innerhalb der Tile-Pair-Vereinigung wird, da asymptotisch unbedeutend, ignoriert.

In jeder Major-Iteration (MaI) des in Abschnitt 10.2.4 beschriebenen Verfahrens werden wiederholt alle Tiles zu Tile-Pairs gruppiert und zu je einem Tile vereinigt. Die Gruppierung innerhalb einer Major-Iteration ist entweder nur nebeneinander (x-MaI) oder nur übereinander (y-MaI). Dabei entspricht die Parallelität der Tile-Pair Anzahl der jeweiligen Major-Iteration. Da sich in jeder Major-Iteration die Zahl der Tiles halbiert, gilt Gleiches folglich auch für die Parallelität. Zusammenfassend gilt für die Parallelität:

- Initial $N/2 = X \cdot Y/2 = Y^2 \cdot k/2$
- Halbiert sich nach jeder MaI
- In der letzten Major-Iteration beträgt sie 1

Sequentiell sind alle Minor-Iterationen (MiI) aller Major-Iterationen abzuarbeiten. Es gibt $log_2(k \cdot Y) = log_2(k) + log_2(Y)$ x-MaI und $log_2(Y)$ y-MaI.

Bei den Minor-Iterationen muss unterschieden werden zwischen denen in x-Dimension (x-MiI) und in y-Dimension (y-MiI). Deren Anzahl entspricht jeweils der Länge der gemeinsamen Kante der Tile-Pairs in den einzelnen MaI. Sowohl in x als auch in y ist diese anfänglich 1, da die Tiles am Anfang einen Pixel groß sind und dementsprechend eine Kantenlänge von 1 haben. Bei einer x-MaI verdoppelt sich die y-Kantenlänge der resultierenden Tiles. Dies bewirkt folglich eine Verdopplung der y-MiI Anzahl der folgenden y-MaI. Analog bewirkt eine y-MaI eine Verdopplung der x-MiI Anzahl nachfolgend auszuführenden y-MaI.

Die Gesamtzahl der Operationen einer MaI ergibt sich aus dem Produkt von Parallelität und der zugehörigen MiI-Anzahl. Diese halbiert sich immer im Vergleich zur vorherigen MaI in derselben Dimension. Das ist insbesondere unabhängig von der Reihenfolge, in der die einzelnen x-MaI und y-MaI ausgeführt werden. Ursache dafür ist die halbierte Parallelität nach jeder MaI zusammen mit der Verdopplung der MiI-Anzahl einer Dimension durch Ausführung einer MaI der anderen Dimension. Zusammenfassend gilt für die Operationen je Major-Iteration:

- Initial in jeder Dimension $Y^2 \cdot k/2$
- Halbiert sich in jeder Major-Iteration in einer Dimension für diese Dimension

- Ist in der letzten x-MaI Y
- Ist in der letzten y-MaI $Y \cdot k$

Damit ist die Anzahl der Berechnungen für alle x-MaI und für alle y-MaI fest. Unveränderlich ist auch die Halbierung der Parallelität in jeder MaI. Aber abhängig von der Vorschrift, wann eine x-MaI oder eine y-MaI ausgeführt wird, kann beeinflusst werden, in welcher Dimension paralleler gerechnet wird und in welcher mehr sequentielle Schritte auszuführen sind. Werden z.B. erst alle x-MaI ausgeführt und dann alle y-MaI, ist die x-MiI Anzahl während der x-MaI immer 1, folglich muss innerhalb einer x-MaI nicht sequentiell gerechnet werden. Dafür ist die Anzahl der y-MiI während der Ausführung aller y-MaI immer $Y \cdot k$, da die Tiles in der x-Dimension bereits über die ganze Zeile vereinigt sind. Die Parallelität zu Beginn der y-MaI ist noch Y / 2. Insgesamt werden also über alle MaI $log_2(Y \cdot k) + log_2(Y) \cdot Y \cdot k$ sequentielle Schritte ausgeführt. Zwar kann dieses Schema günstig sein, wenn die Kantenlängen sehr unterschiedlich sind, aber es soll im Folgenden nicht weiter berücksichtigt werden. Es sei noch angemerkt, dass die asymptotische Anzahl sequentieller Schritte logarithmisch wird, wenn eine Kantenlänge eins, bzw. konstant, ist.

Das in Abschnitt 10.2.4 vorgeschlagene Schema sieht vor, je MaI immer in der Dimension die Tiles zu vereinigen, in der mehr vereinigt werden können. Dann werden hier zunächst $log_2(k)$ x-MaI ausgeführt, sodass in x und y gleich viele Tiles vorliegen. Anschließend wechseln sich x-MaI und y-MaI ab, wobei hier willkürlich mit einer x-MaI begonnen werde. Dieses Schema wird in Tabelle 10.1 veranschaulicht.

Die Gesamtzahl der sequentiellen Schritte ergibt sich als Summe über die Minor-Iterationen für alle x-MaI und y-MaI. Sie kann jeweils durch die geometrische Reihe abgeschätzt werden:

$$
\begin{aligned}
Steps_X &= \sum_{\text{x-MaI}=1}^{log_2(Y \cdot k)} (\text{Anzahl x-MiI})_{\text{x-MaI}} \\
&= \underbrace{1 + \ldots + 1}_{log_2(k) \cdot 1} + 1 + 2 + 4 + \ldots + \frac{Y}{8} + \frac{Y}{4} + \frac{Y}{2} \\
&= log_2(k) + \sum_{i=0}^{log_2(Y)-1} \left(\frac{1}{2}\right)^i \cdot Y/2 \\
&\leq log_2(k) + Y
\end{aligned}
$$

Tabelle 10.1.: Alternierendes Tile-Verarbeitungsschema für ein $X \cdot Y$ großes Pixelgitter. Dabei gilt $X = Y \cdot k$ mit $k \geq 1$ und X und Y sind Zweierpotenzen. Die Anzahl der Minor-Iterationen (MiI) gibt die Anzahl sequentieller Schritte innerhalb der jeweiligen MaI an und die Berechnungen sind das Produkt aus MiI und Parallelität einer MaI.

MaI	MaI-Typ	Parallelität	MiI Anzahl	Berechnungen
1	x	$Y^2 \cdot k/2$	1	$Y^2 \cdot k/2$
2	x	$Y^2 \cdot k/4$	1	$Y^2 \cdot k/4$
3	x	$Y^2 \cdot k/8$	1	$Y^2 \cdot k/8$
$\vdots$	x	$\vdots$	1	$\vdots$
$log_2(k)$	x	Y^2	1	Y^2
$log_2(k)+1$	x	$Y^2/2$	1	$Y^2/2$
$log_2(k)+2$	y	$Y^2/4$	$2 \cdot k$	$Y^2 \cdot k/2$
$log_2(k)+3$	x	$Y^2/8$	2	$Y^2/4$
$log_2(k)+4$	y	$Y^2/16$	$4 \cdot k$	$Y^2 \cdot k/4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$log_2(k \cdot Y^2)-3$	x	8	$Y/4$	$Y \cdot 2$
$log_2(k \cdot Y^2)-2$	y	4	$Y \cdot k/2$	$k \cdot Y \cdot 2$
$log_2(k \cdot Y^2)-1$	x	2	$Y/2$	Y
$log_2(k \cdot Y^2)$	y	1	$Y \cdot k$	$k \cdot Y$

$$
\begin{aligned}
Steps_Y &= \sum_{\text{y-MaI}=1}^{log_2(Y)} (\text{Anzahl y-MiI})_{\text{y-MaI}} \\
&= 2 \cdot k + 4 \cdot k + 8 \cdot k + \ldots + \frac{Y \cdot k}{4} + \frac{Y \cdot k}{2} + Y \cdot k \\
&= \sum_{i=0}^{log_2(Y)-1} \left(\frac{1}{2}\right)^i \cdot Y \cdot k \\
&\leq 2 \cdot Y \cdot k
\end{aligned}
$$

Die asymptotische Anzahl der sequentiellen Schritte (Depth), welche Sub-Algorithmus 14 ausführt, ergibt sich aus der Summe der Schritte über alle Vereinigungen in x und y zu:

$$Depth = O(Steps_X + Steps_Y) = O(max(X, Y))$$

Dabei wurde $k \cdot Y$ wieder zu $max(X, Y)$ verallgemeinert, und schließt somit wieder den Fall $Y > X$ ein.

Running-Time und Cost

Die beste erreichbare Laufzeit des Sub-Algorithmus `unifyTiles`, der die Laufzeit des Sub-Algorithmus 16 limitiert, ist mit der im obigen Abschnitt gefundenen Depth identisch. Die dazu in der bisherigen Formulierung, auch in Abschnitt 10.2.4, aus Gründen

der Verständlichkeit verwendete Parallelität ist $\leq N/2 = X \cdot Y/2$. Dementsprechend müssten $O(X \cdot Y)$ Prozessoren zum Erreichen dieser Laufzeitgarantie eingesetzt werden. Total-Cost ergäbe sich dann zu $O(X \cdot Y \cdot max(X, Y))$ und wäre damit offensichtlich nicht optimal.

Im Folgenden soll gezeigt werden, dass $O(min(X, Y))$ Prozessoren genügen, um die Laufzeit $O(max(X, Y))$ zu erreichen. Dazu mögen wieder die Vereinfachungen des vorherigen Abschnitts gelten: X und Y seien Zweierpotenzen und $X \geq Y$ um Faktor k. Damit ergibt sich die maximal verwendbare Prozessorzahl P zu Y. Das Verarbeitungsschema bleibt auf Major-Iterationsebene gleich, es werden also weiterhin jeweils in der Dimension Tiles zu Tile-Pairs gruppiert und anschließend vereinigt, in der mehr nebeneinander bzw. übereinander liegen. Innerhalb einer MaI wird aber nun höchstens Y-fach parallel gerechnet, dementsprechend steigt die Anzahl der notwendigen sequentiellen Operationen ($Steps_{MaI}$) im Vergleich mit der MiI-Anzahl, wie sie bisher formuliert wurde in einigen MaI. Die notwendigen $Steps_{MaI}$ lassen sich je MaI bestimmen gemäß:

```
Falls Parallelität > P
     Steps_MaI = Berechnungen / P
Sonst
     Steps_MaI = MiI
```

Das so veränderte Verarbeitungsschema ist in Tabelle 10.2 zu sehen. Der Wert von $Steps_{MaI}$ ist nun am Anfang aufgrund der Problemgröße in jeder Dimension maximal und halbiert sich so lange, bis die erreichbare Parallelität mit der Prozessorzahl identisch ist. Für die verbleibenden MaI gleicht das Schema dem bereits in Tabelle 10.2 gezeigten.

Die asymptotische Laufzeit kann nun als Summe über die $Steps_{MaI}$ aller MaI bestimmt werden. Für die Abschätzung ist die Aufteilung in vier Partialsummen, x und y jeweils über und unter der MaI, für die Parallelität = P = Y gilt, hilfreich. Jede dieser Partialsummen ist dann streng monoton steigend oder streng monoton fallend und kann mithilfe der geometrischen Reihe abgeschätzt werden. Die Abschätzungen der Summen der Partialsummen im Einzelnen:

$$\begin{aligned} Steps_{y,u} &= Y \cdot k/2 + Y \cdot k/4 + Y \cdot k/8 + \ldots + 4 \cdot k\sqrt{Y} + 2 \cdot k\sqrt{Y} + k\sqrt{Y} \\ &= \sum_{i=0}^{\frac{1}{2}log_2(Y)-2} \left(\frac{1}{2}\right)^i \cdot k \cdot Y/2 \\ &\leq k \cdot Y \end{aligned}$$

$$\begin{aligned} Steps_{x,u} &= Y \cdot k/2 + Y \cdot k/4 + Y \cdot k/8 + \ldots + 8\sqrt{Y/2} + 4\sqrt{Y/2} + 2\sqrt{Y/2} \\ &= \sum_{i=0}^{log_2(k)+\frac{1}{2}log_2(Y)-1} \left(\frac{1}{2}\right)^i \cdot k \cdot Y/2 \\ &\leq k \cdot Y \end{aligned}$$

Tabelle 10.2.: Alternierendes Tile-Verarbeitungsschema für ein $X \cdot Y$ großes Pixelgitter. Dabei gilt $X = Y \cdot k$ mit $k \geq 1$ und X und Y sind Zweierpotenzen. Die Anzahl der Minor-Iterationen (MiI) gibt die minimale Anzahl sequentieller Schritte innerhalb der jeweiligen MaI an und die Berechnungen sind das Produkt aus MiI und maximaler Parallelität einer MaI. Diese Größen sind allein von der Datenauflösung abhängig und identisch mit denen aus Tabelle 10.2. Neu ist $Steps_{MaI}$, die Anzahl erforderlicher sequentieller Schritte je MaI bei Verwendung von P = min(X, Y) = Y Prozessoren.

MaI	**Typ**	**Parallelität**	**MiI**	**Berechnungen**	**Steps$_{\mathbf{MaI}}$**
1	x	$Y^2 \cdot k/2$	1	$Y^2 \cdot k/2$	$Y \cdot k/2$
2	x	$Y^2 \cdot k/4$	1	$Y^2 \cdot k/4$	$Y \cdot k/4$
3	x	$Y^2 \cdot k/8$	1	$Y^2 \cdot k/8$	$Y \cdot k/8$
$\vdots$	x	$\vdots$	1	$\vdots$	$\vdots$
$log_2(k)$	x	Y^2	1	Y^2	Y
$log_2(k)+1$	x	$Y^2/2$	1	$Y^2/2$	$Y/2$
$log_2(k)+2$	y	$Y^2/4$	$2 \cdot k$	$Y^2 \cdot k/2$	$Y \cdot k/2$
$log_2(k)+3$	x	$Y^2/8$	2	$Y^2/4$	$Y/4$
$log_2(k)+4$	y	$Y^2/16$	$4 \cdot k$	$Y^2 \cdot k/4$	$Y \cdot k/4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$log_2(k \cdot Y)-4$	x	$16 \cdot Y$	$\sqrt{Y}/8$	$2 \cdot Y\sqrt{Y}$	$2\sqrt{Y}$
$log_2(k \cdot Y)-3$	y	$8 \cdot Y$	$k\sqrt{Y}/4$	$2 \cdot k \cdot Y\sqrt{Y}$	$2 \cdot k\sqrt{Y}$
$log_2(k \cdot Y)-2$	x	$4 \cdot Y$	$\sqrt{Y}/4$	$Y\sqrt{Y}$	$\sqrt{Y}$
$log_2(k \cdot Y)-1$	y	$2 \cdot Y$	$k\sqrt{Y}/2$	$k \cdot Y\sqrt{Y}$	$k\sqrt{Y}$
$log_2(k \cdot Y)$	x	Y	$\sqrt{Y}/2$	$Y\sqrt{Y}/2$	$\sqrt{Y}/2$
$log_2(k \cdot Y)+1$	y	$Y/2$	$k\sqrt{Y}$	$k \cdot Y\sqrt{Y}/2$	$k\sqrt{Y}$
$log_2(k \cdot Y)+2$	x	$Y/4$	$\sqrt{Y}$	$Y\sqrt{Y}/4$	$\sqrt{Y}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$log_2(k \cdot Y^2)-3$	x	8	$Y/4$	$Y \cdot 2$	$Y/4$
$log_2(k \cdot Y^2)-2$	y	4	$Y \cdot k/2$	$k \cdot Y \cdot 2$	$Y \cdot k/2$
$log_2(k \cdot Y^2)-1$	x	2	$Y/2$	Y	$Y/2$
$log_2(k \cdot Y^2)$	y	1	$Y \cdot k$	$k \cdot Y$	$k \cdot Y$

$$\begin{aligned} Steps_{x,o} &= \sqrt{Y}/2 + \sqrt{Y} + 2\sqrt{Y} + \ldots + Y/8 + Y/4 + Y/2 \\ &= \sum_{i=0}^{\frac{1}{2}log_2(Y)-1} \left(\frac{1}{2}\right)^i \cdot Y/2 \\ &\leq Y \end{aligned}$$

$$\begin{aligned} Steps_{y,o} &= k\sqrt{Y} + 2 \cdot k\sqrt{Y} + 4 \cdot k\sqrt{Y} + \ldots + k \cdot Y/4 + k \cdot Y/2 + k \cdot Y \\ &= \sum_{i=0}^{\frac{1}{2}log_2(Y)-1} \left(\frac{1}{2}\right)^i \cdot k \cdot Y \\ &\leq 2 \cdot k \cdot Y \end{aligned}$$

Die asymptotische Laufzeit, welche Sub-Algorithmus 14 benötigt, ergibt sich für $P = min(X, Y)$ Prozessoren zu:

$$O(Steps_{x,u} + Steps_{x,o} + Steps_{y,u} + Steps_{y,o}) = O(max(X, Y))$$

Dabei wurde $k \cdot Y$ wieder zu $max(X, Y)$ verallgemeinert, und schließt somit wieder den Fall $Y > X$ ein. Da Sub-Algorithmus 14 den für Sub-Algorithmus 16 geschwindigkeitslimitierenden Schritt darstellt, ist auch dessen Laufzeit $O(max(X, Y))$ bei Verwendung von $O(min(X, Y))$ Prozessoren. Total-Cost ergibt sich folglich als:

$$O(max(X, Y) \cdot min(X, Y)) = O(X \cdot Y) = O(N)$$

Damit ist Total-Cost asymptotisch optimal.

10.3. Variante II: Datenabhängiges Parallelitätsschema

Ziel dieses Ansatzes ist die Erhöhung der Parallelität und damit Verkürzung der Laufzeit im Vergleich mit dem im vorherigen Kapitel vorgestellten datenunabhängigen Ansatz. Dazu werden vor den eigentlichen Operationen die Daten analysiert und Elemente bestimmt, welche parallel verarbeitet werden können. Erhalten bleiben sollen aber in jedem Fall die asymptotisch linearen Gesamtkosten.

Der hier vorgeschlagene Ansatz basiert auf den Ideen des Basic-List-Ranking-Algorithmus (Siehe Kapitel 7.5 ab Seite 40 ff.) und insbesondere auch auf Cole und Vishkins optimalem List-Ranking-Algorithmus (Siehe Kapitel 7.6 ab Seite 43). Diese Techniken werden vorausgesetzt und nachfolgend nur die Unterschiede dazu erläutert.

Alle Konturen liegen, wie in den vorherigen Kapiteln beschrieben, als einfach verkettete zyklische gerichtete Linked-Lists aus Kontursegmenten vor. Weitere Informationen, wie SRC-DST-Typ oder das Pixelgrid, werden nicht benötigt. Allerdings muss jedes Kontursegment über einen eindeutigen Index verfügen. Weil die für Kontursegmente spezifischen Informationen nicht benötigt werden und um die Vergleichbarkeit mit den List-Ranking-Algorithmen zu vereinfachen, werden die Kontursegmente in diesem Kapitel als Knoten (einer Linked-List) bezeichnet.

Vor der eigentlichen Formulierung eines Konturlabeling-Algorithmus, angelehnt an die List-Ranking-Algorithmen, seien die Unterschiede festgehalten, auf welche später einzugehen sein wird:

1. Die Ausgangsdaten liegen, sowohl für das Kontur-Labeling, als auch für das List-Ranking, in Form von einfach verlinkten gerichteten Linked-Lists vor. Allerdings handelt es sich beim Kontur-Labeling um Zyklen und beim List-Ranking um Pfade. Es muss folglich beim Labeling bedacht werden, dass es kein letztes oder erstes Element gibt. Das Finden eines r-Ruling-Sets, welcher für $r = 2$ für das optimale List-Ranking benötigt wird, ist von Cole und Vishkin allerdings bereits explizit für zyklische gerichtete Linked-Lists formuliert worden [CV86b, CV89].

2. Es muss etwas anderes, nämlich ein Label (gleicher Wert für alle Knoten je einer Linked-List) und kein Rank (verschiedener Wert für alle Knoten einer Linked-List) bestimmt werden. Folglich ist die Operation beim Vergleich der Werte von je zwei Knoten anzupassen.

3. Bei den genannten List-Ranking-Algorithmen wird von einer einzigen Linked-List mit $O(N)$ Knoten ausgegangen. In einem solchen Datensatz ein (und genau ein) Label zu vergeben ist offensichtlich. Stattdessen muss der Algorithmus einen Datensatz aus $O(N)$ (potentiellen) Knoten bearbeiten, die zu jeder Anzahl bis hin zu $O(N)$ unabhängigen zyklischen Linked-Lists zusammengefügt sein können. Aus diesem Grund wird ein Sonderfall für 'kurze' Linked-Lists beim Ausdünnen hinzuzufügen sein. Die maximale Länge einer einzelnen Linked-List bleibt aber unverändert $O(N)$.

10.3.1. Kontur-Labeling durch Pointer-Jump-Operations auf Linked-Lists

Zunächst wird ein Kontur-Labeling-Algorithmus vorgestellt, welcher eine leichte Abwandlung des in Kapitel 7.5 ab Seite 40 gegebenen parallelen Basic-List-Ranking-Algorithmus darstellt. Er ist im Pseudocodeabschnitt Algorithmus 17 gegeben. Zur Veranschaulichung der Funktionsweise des Algorithmus 17 siehe Abbildung 10.6

Algorithm 17: Basic Contour Labeling

```
For Each Node i ∈ {1, ..., N} In Parallel Do
    // Old: (List-Ranking): 0: (Last node), 1 (Other nodes)
    // New: Apply distinct value to each node
    val(i) ← i
    // Break Condition removed, do all log2(N) Ops for all nodes
    // Old: 'While suc(i) ≠ null do'
    Do ⌈log2(N)⌉ times
        // Consider only existing nodes
        If existing(i) Then
            // Do (modified) Pointer Jump Operation
            val(i) ← Call Min(val(i), val(suc(i)))
            suc(i) ← suc(suc(i))
        End
    End
End
```

Die Eingangsdaten seien - zunächst - eine zyklische gerichtete Linked-List mit N Knoten. Im Gegensatz zum List-Ranking-Algorithmus, der auf gerichteten, einfach verketteten Pfaden operiert, gibt es insbesondere kein Ende. Dementsprechend muss das Abbruchkriterium, ehemals der Vergleich des Verweises auf den Nachfolgeknoten (`suc`) mit null, angepasst werden, da dieses Kriterium nie erfüllt ist.

Wie beim Basic-List-Ranking-Algorithmus wird in jeder Iteration der, hier etwas abgewandelten, Pointer-Jump-Operation für jeden Knoten i der Verweis auf den Nachfolgeknoten $suc(i)$ auf den Nachfolger des Nachfolgers $suc(suc(i))$ gesetzt. Somit ist, wieder wie beim List-Ranking, anfangs die Reichweite der gesammelten Informationen eins, nach einer Pointer-Jump-Operation zwei, nach zwei Pointer-Jump-Operationen vier und nach $log_2(N)$ Iterationen N. Insbesondere hat dann jeder Knoten Informationen über alle Knoten aggregiert. Dementsprechend wird anstelle des alten Abbruchkriteriums die modifizierte Pointer-Jump-Operation $log_2(N)$ Mal parallel für alle Knoten aufgerufen.

Die zweite Änderung betrifft den Wert val, welcher beim List-Ranking mit 0 (letzter Knoten) oder 1 (andere Knoten) initialisiert wurde. Beim parallelen Basic-Contour-Labeling-Algorithmus wird val stattdessen für jeden Knoten ein eindeutiger Wert hinsichtlich des Operators $<$ zugewiesen. Dafür kann z.B. die eindimensionale Speicheradresse verwendet werden. Zusätzlich bestimmt die modifizierte Pointer-Jump-Operation immer

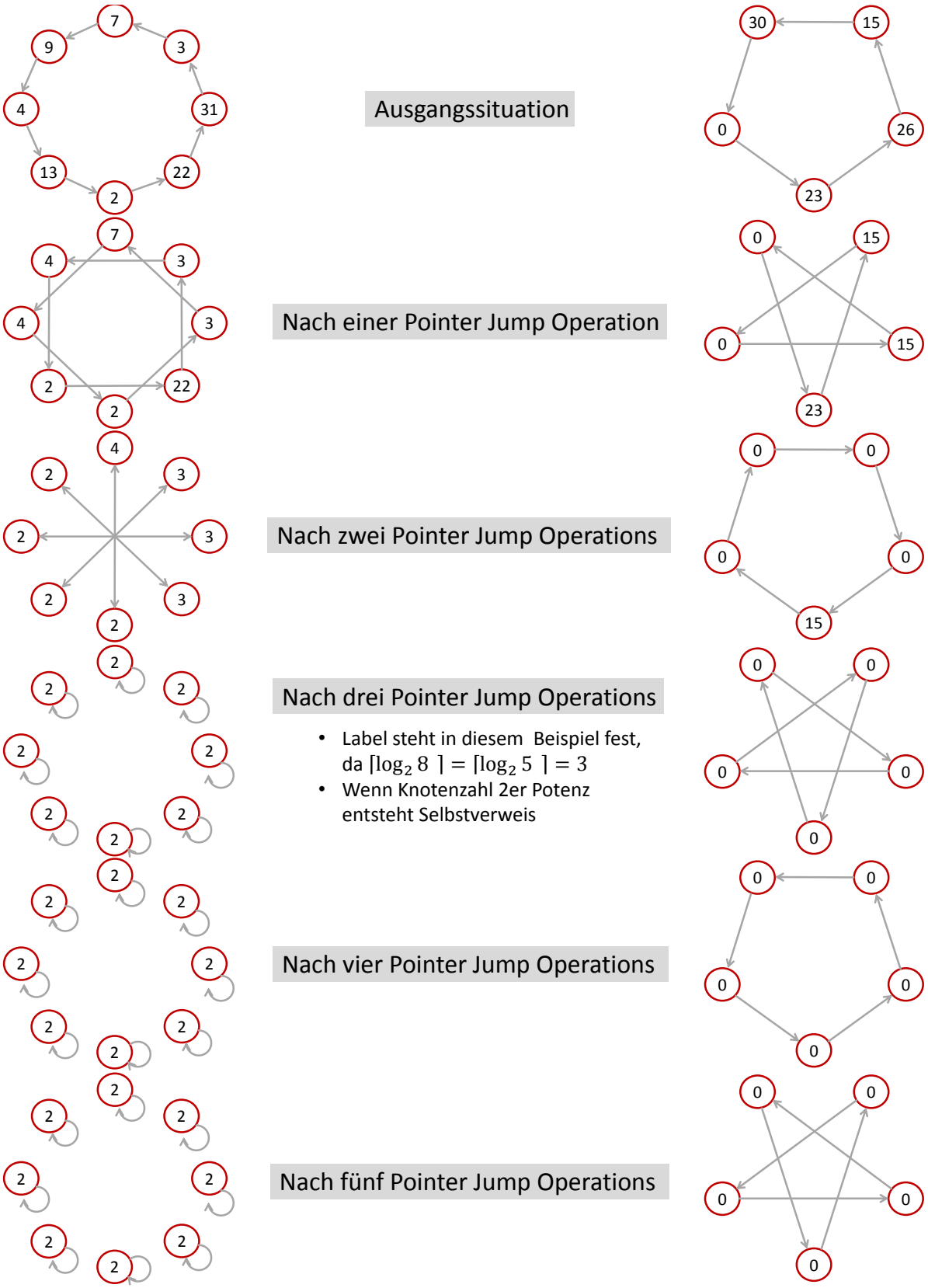


Abbildung 10.6.: Beispiel für den Basic-Contour-Labeling-Algorithmus, angewandt auf einen Datensatz der Größe 32. Folglich sind 5 Iterationen auszuführen. Der Datensatz enthält zwei zyklische Linked-Lists und einige Leer-Einträge

das Minimum aus dem eigenen Wert und dem des Nachfolgers. Folglich liegt nach Ausführung des Algorithmus 17 für alle Knoten i in $val(i)$ der minimale Wert von val aller Knoten der Linked-List vor. Wir halten fest:

Hilfssatz 22 *Nach Ausführung von Algorithmus 17 haben alle Knoten einer Linked-List den gleichen Wert in val.*

Sei nun ein Datensatz gegeben, der aus null, einer, oder mehreren, bis maximal $O(N)$ unabhängigen zyklischen gerichteten Linked-Lists besteht, die zusammen maximal $O(N)$ Knoten haben. Ferner kann es auch leere Einträge geben. In dieser Form liegen die Kontursegmente vor, wenn sie, wie in Kapitel 9 beschrieben, extrahiert wurden. Zur Behandlung der leeren Einträge wird ein zusätzlicher Vektor `existing` herangezogen, um nachzuschlagen, ob überhaupt etwas zu tun ist. Wenn eine Linked-List aus weniger als N Knoten besteht, werden bei Ausführung des Algorithmus 17 weitere (und möglicherweise unvollständige) 'Runden gedreht'. Am finalen Wert von val ändert das nichts, da immer wieder die Minimum Funktion auf die sowieso schon minimalen Werte angewandt wird. Dagegen verweist suc dann am Ende nicht auf einen 'bestimmten' Knoten, wie etwa seinen Vorgänger (bei genau einer Umdrehung). Das ist aber irrelevant, da suc nach Abschluss des Algorithmus 17 nicht mehr benötigt wird. Wichtig ist, dass auch bei unabhängigen Linked-Lists jeder Knoten je einer Linked-List den gleichen Wert für val erhält. Außerdem wurde für jeden Knoten i $val(i)$ mit einem global eindeutigen Wert initialisiert. Somit gilt offensichtlich:

Hilfssatz 23 *Nach Ausführing von Algorithmus 17 haben alle Knoten verschiedener Linked-Lists verschiedene Werte in val.*

Aus Hilfssatz 22 zusammen mit Hilfssatz 23 folgt unmittelbar:

Satz 6 *Algorithmus 17 ist ein Labeling-Algorithmus für zyklische gerichtete Linked-Lists. Und damit ein Kontur-Labeling-Algorithmus, wenn diese so vorliegen.*

Asymptotische Analyse

Das asymptotische Verhalten des Algorithmus 17 entspricht offensichtlich genau dem des parallelen Basic-List-Ranking-Algorithmus. Zwar bleiben alle Verweise auf Nachfolger bis zum Ende aktiv, aber das ändert nichts an den $O(N \cdot log_2(N))$ Operationen insgesamt. Schließlich werden ebenfalls bei $P = N$ Prozessoren $log_2(N)$ Zeitschritte benötigt. Die, nicht optimalen, Eigenschaften insgesamt:

$$\begin{aligned} \textbf{Machine} &: \text{EREW-PRAM} \\ \textbf{Max Processors} &: O(N) \\ \textbf{Running-Time} &: O(N \cdot \log_2(N)/P) \\ \textbf{Cost} &: O(N \cdot \log_2(N)) \end{aligned}$$

Exkurs: Best-Case-Betrachtung

Die obigen Eigenschaften gelten für den Worst-Case einer einzigen Linked-List mit $O(N)$ Knoten. Bei mehreren und dann kürzeren Linked-Lists sind die Werte aller Knoten bereits früher, nämlich in $O(log_2(\text{maximale Länge}))$ Zeit, aggregiert. Allerdings ist das Wissen, ob die gegebenen Knoten zu einer einzigen oder mehreren unabhängigen Linked-Lists gehören, vor Ausführung des Labeling-Algorithmus noch gar nicht vorhanden. Jedoch kann während der Ausführung für einen Knoten v bestimmt werden, ob sein Wert bereits mit dem aller anderen Knoten verglichen wurde. Das ist der Fall, wenn vor Ausführung einer Pointer-Jump-Operation gilt:

$$val(v) = val(suc(v)) \tag{10.1}$$

Dies ist der Fall, da anfänglich die Werte aller Knoten verschieden sind. Somit kann der Nachfolger eines Knotens vor einer Pointer-Jump-Operation nur dann den gleichen Wert haben wie der Betrachtete, wenn bereits eine vollständige 'Runde gedreht' wurde. Somit kann die While-Schleife in Algorithmus 17 vorzeitig abgebrochen werden, wenn obige Bedingung für alle Knoten erfüllt ist. Infolgedessen wird die Laufzeit des so modifizierten Algorithmus 17 bei $P = N$ Prozessoren zu $O(1)$, wenn eine konstante maximale Länge für alle Linked-Lists garantiert werden kann. Weil in dieser Arbeit das asymptotische Verhalten und damit der Worst-Case im Vordergrund steht, sei es bei diesem Exkurs zum Best-Case belassen.

10.3.2. Optimales Kontur-Labeling auf einem CRCW-PRAM

In diesem Kapitel wird beschrieben, wie der optimale parallele List-Ranking-Algorithmus von Cole und Vishkin [CV89] zu einem optimalen parallelen Kontur-Labeling-Algorithmus modifiziert werden kann. Dies geschieht zum großen Teil analog zu dem Basic-Contour-Labeling-Algorithmus im vorherigen Abschnitt.

Insbesondere bleibt die prinzipielle Vorgehensweise identisch mit der in Cole und Vishkins Algorithmus. So ist auch der zu formulierende Labeling-Algorithmus in sechs Steps vergleichbarer Art eingeteilt wie der Algorithmus 12, welcher auf Seite 55 gegeben ist. Dabei sind ebenfalls die Steps eins bis vier durch eine While-Schleife umgeben, welche den Datensatz ausdünnt, bis maximal $N/log_2(N)$ Knoten übrigbleiben. Der Pseudocode für den optimalen Kontur-Labeling-Algorithmus ist im Pseudocode-Abschnitt Algorithmus 18 gegeben und wird in Pseudocode-Abschnitt Algorithmus 19 fortgesetzt. Die Schritte im Einzelnen:

Step I + II: Initialisierung + Bestimmung des 2-Ruling-Set

Die Bestimmung des 2-Ruling-Set ist identisch mit der von Cole und Vishkin in [CV89] formulierten Vorgehensweise, da sie dort für zyklische Linked-Lists explizit gegeben ist (sogar nur dafür). Der Algorithmus funktioniert für unabhängige zyklische Linked-Lists genauso. Es kann somit die in den Kapiteln 7.6 und 7.7 gegebene Formulierung, welche in Algorithmus 11 auf Seite 54 mündet unverändert übernommen werden. Die Laufzeit ist abhängig von der Länge der längstmöglichen Linked-List, die ist ebenfalls $O(N)$, und der Anzahl der erforderlichen Bits (ebenfalls $log_2(N)$). Folglich bleiben alle Eigenschaften im Worst-Case erhalten. Die restliche Initialisierung ist identisch mit der des Basic-Contour-Labeling-Algorithmus.

Step III: Shortcut-Operation

Die Shortcut-Operation wird analog zum Basic-Contour-Labeling-Algorithmus abgeändert. Das Abspeichern übersprungener Knoten funktioniert analog zu Step III in Algorithmus 12. Allerdings ist bei Einträgen in `save` das Abspeichern des aktuellen Knotenwertes nicht nötig, da Label nicht relativ definiert sind.

Neu ist dagegen die Behandlung 'kurzer' Linked-Lists, welche nach einer (oder mehreren) Iterationen der While-Schleife möglicherweise aus nur noch einem Knoten bestehen können. Würde man diese einfach aus dem 2-Ruling-Set U nehmen, verschwänden sie und das Ergebnis wäre möglicherweise falsch. Blieben sie einfach in U enthalten, kann ein ausreichendes Schrumpfen der verbleibenden Knotenzahl nicht mehr garantiert werden. Es ist demnach ein Sonderfall nötig, der ähnlich gelöst werden kann wie das Abspeichern übersprungener Knoten in Algorithmus 12.

Dafür werden alle Knoten v aus U nach jeder Shortcut-Operation daraufhin überprüft, ob sie der letzte Knoten ihrer Linked-List sind. Das ist genau dann der Fall, wenn gilt: $suc(v) = v$. Knoten v, welche diese Bedingung erfüllen, werden 'pausiert'. Dazu werden sie zum einen aus der verbleibenden Knotenmenge genommen, indem $ruling(v)$ auf 0 gesetzt wird. Jeder 'Einzelknoten' v enthält offensichtlich in $val(v)$ den minimalen

Algorithm 18: Optimal Parallel Contour Labeling, Part A

```
m ← N // Remaining Nodes, at first: N
t ← 0
While m ≥ N/log2(N) Do
    // Step I
    For Each Vertex v, v ∈ {0, 1, ..., m − 1} In Parallel Do
        serial0(v) ← v
    End
    // Step II
    ruling ← Execute Compute2RSet(serial0, suc) // p. 11
    // Step III
    For Each Processor p, p ∈ {1, 2, ..., P} In Parallel Do
        For Each v, with: (p − 1) · m/P ≤ v ≤ p · m/P − 1 Do
            t ← t + 1
            If existing(v) ∧ ruling(v) = 1 Then
                save(p, t) ← (suc(v), v)
                val(i) ← Call Min(val(i), val(suc(i)))
                suc(v) ← suc(suc(v))
            End
            t ← t + 1
            If existing(v) ∧ suc(v) = v Then
                paused(p, t) ← v
                ruling(v) ← 0
            End
            t ← t + 1
            If existing(v) ∧ ruling(v) = 1 ∧ ruling(suc(v)) = 0 Then
                save(p, t) ← (suc(v), v)
                val(i) ← Call Min(val(i), val(suc(i)))
                suc(v) ← suc(suc(v))
            End
            t ← t + 1
            If existing(v) ∧ ruling(v) = 1 ∧ suc(v) = v Then
                paused(p, t) ← v
                ruling(v) ← 0
            End
        End
    End
    // Step IV
    newId ← Execute parScan_Exclusive(ruling)
    Do: Compact Vertex-Data in same vectors according to newId and ruling
    m ← ruling(m-1) + newId(m-1)
End
T ← t
// Note: This algorithm is to be continued in algorithm 19 (p. 123)
```

Wert aller Knoten der zugehörigen Linked-List. Dieses kann als Label verwendet werden. Die einzige Möglichkeit für eine vollständige Linked-List, von der Weiterverarbeitung ausgenommen zu werden, ist gerade dieser `paused`-Zustand. Damit gilt:

Hilfssatz 24 *Von allen Linked-Lists, von denen kein Knoten nach Ende der Ausführung der While-Schleife in Algorithmus 18 weiter verarbeitet werden muss, hat der zum letzten Zeitpunkt herausgenommene Knoten ein Label.*

Zum anderen wird im Vektor `paused` für den ausführenden Prozessor und den aktuellen Zeitschritt ein Eintrag für v hinterlegt, sodass 'pausierte' Knoten in Step 6 zum richtigen Zeitpunkt wieder reaktiviert werden können. Dies ist im Pseudocode für Algorithmus 18 im Abschnitt 'Step 3' zu sehen und beispielhaft in Abbildung 10.7 dargestellt.

Weil die 'Einzelknoten' in jedem Durchlauf der While-Schleife noch zusätzlich zu den übersprungenen Knoten entfernt werden, ist das Ausdünnen der Daten mindestens so schnell wie in Cole und Vishkins Algorithmus. Die asymptotischen Eigenschaften bleiben aber unverändert, da es im Worst-Case (eine Linked-List mit $O(N)$ Knoten) identisch ist. Schließlich tritt dieses zusätzliche Ausdünnen dann gar nicht auf. Im anderen Extremfall, wenn es $O(N)$ Linked-Lists mit jeweils der Länge 2 gibt, genügt z. B. garantiert eine Iteration der While-Schleife. Vor allem aber verringert im Worst-Case jede Iteration der While-Schleife die Knotenanzahl auch bei dem Labeling-Algorithmus garantiert um mindestens die Hälfte.

Step IV: Compact Node-Array

Dieser Schritt ist identisch mit Step IV des Algorithmus 12. Hier werden lediglich die verbleibenden Nodes im Array zusammengeschoben und deren Anzahl bestimmt. Dabei ist es irrelevant, ob diese zu verschiedenen Linked-Lists gehören und ob die Linked-Lists zyklisch sind oder nicht. Schließlich hat diese Operation nichts mit der Position in einer Linked-List zu tun.

Step V: Basic Contour-Labeling für Knotenauswahl

Genau wie im Falle des Algorithmus 12 gilt auch hier: Wenn `m`, die Anzahl der verbleibenden Knoten, kleiner gleich $N/log_2(N)$ ist, wird die Steps eins bis vier umgebende While-Schleife verlassen. Anschließend wird in Step 5 der Basic Contour-Labeling-Algorithmus (Algorithmus 17) auf eben diese Knoten angewandt. In Abbildung 10.8, einer Fortsetzung der Beispielabbildung 10.7, ist dieser Schritt zu sehen. Weil die Voraussetzungen identisch zu denen von Algorithmus 17 bzw. Satz 6 sind, gilt hier: In diesem Schritt werden ebenfalls $\texttt{P} = N/log_2(N)$ Prozessoren verwendet, sodass der Schritt auch hier lediglich $O(N)$ Kosten verursacht. Damit sind für Knoten dieser Teilmenge der Eingangsgraphen, zusätzlich zu den 'pausierten' Knoten, die Label bekannt. Wir halten fest:

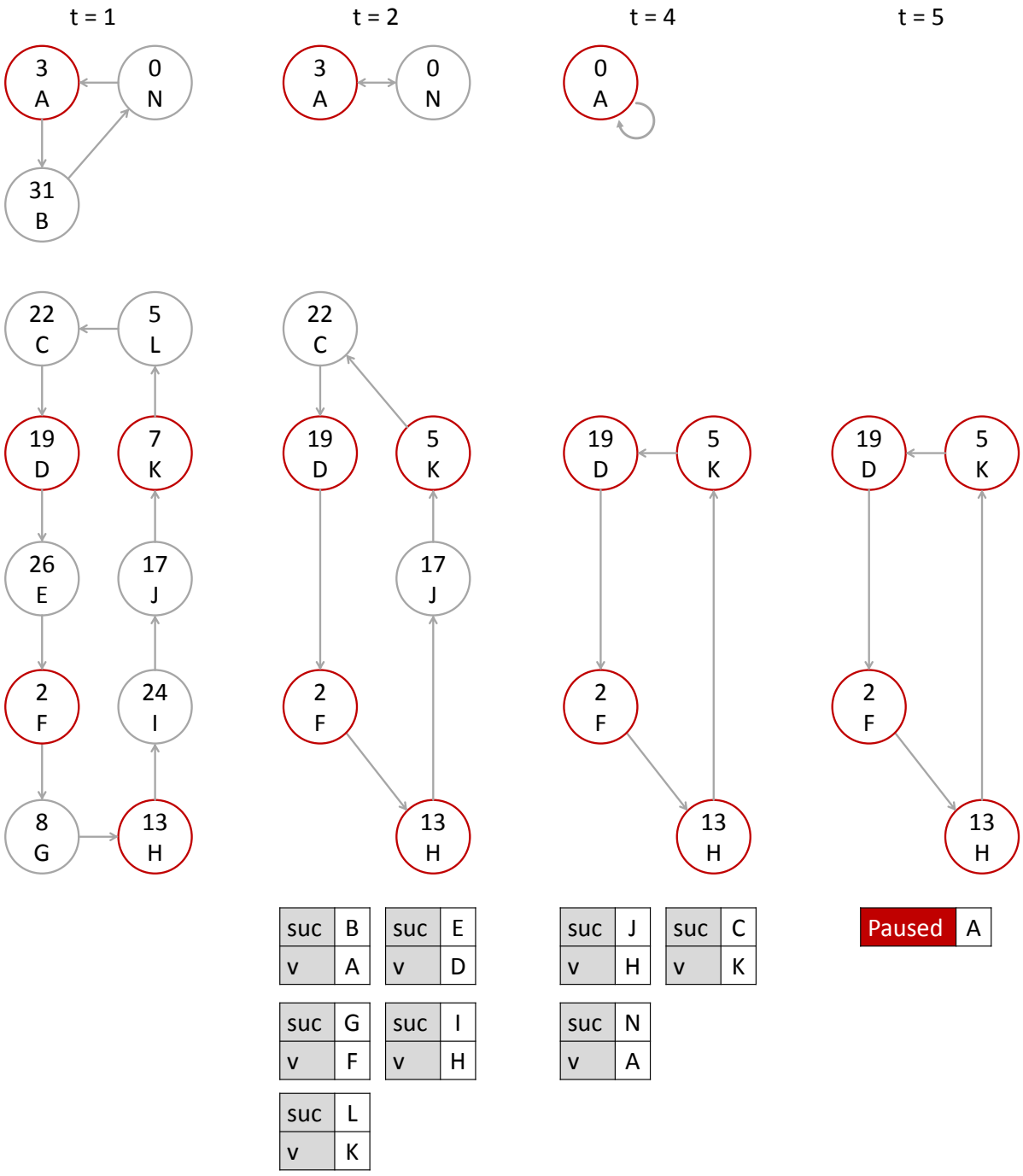


Abbildung 10.7.: Vereinfachtes Beispiel für eine Iteration des Step 3 aus Algorithmus 18. Gegeben ist, zu Zeit $t = 1$, ein Datensatz (Größe N=32), welcher zwei zyklische Linked-Lists enthält und ein 2-Ruling-Set U (rote Knoten). Buchstaben in Knoten geben den Knotenindex v an und Zahlen den (aktuellen) Wert $val(v)$. Zeit $t = 2$: Erste Shortcut-Operation ausgeführt. Entstandene Einträge in `save` für diesen Zeitschritt sind als Kästen darunter abgebildet. Zeit $t = 3$: Nicht gezeigt, da nichts passiert. Zeit $t = 4$: Zweite Shortcut-Operation dieser Iteration ausgeführt und weitere Einträge für `save` zu diesem Zeitpunkt erzeugt. Zeit $t = 5$: Knoten mit Selbstverweis als pausiert markiert und aus U entfernt.

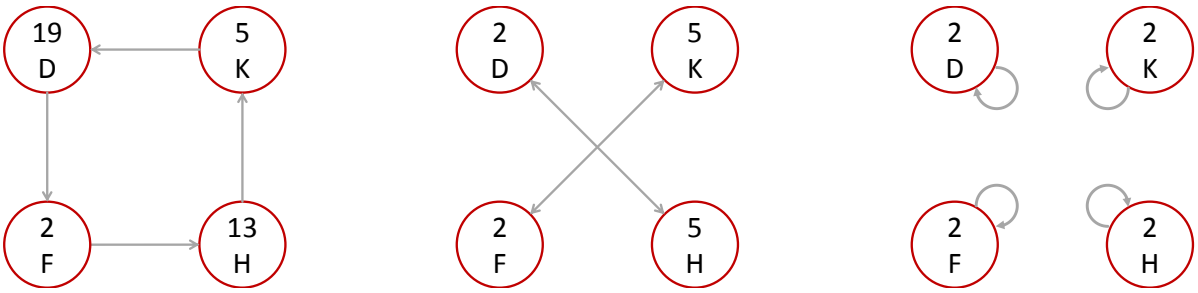


Abbildung 10.8.: Step 5 (Basic-Contour-Labeling) aus Algorithmus 17, angewandt auf die verbleibenden Knoten des Beispiels aus Abb. 10.7. Es handelt sich bei der Beispielserie nicht um ein vollständiges Beispiel des Algorithmus 18/19. So sind z.B. Steps eins, zwei und vier nicht gezeigt.

Hilfssatz 25 *Von allen Linked-Lists, von denen mindestens ein Knoten nach Ende der Ausführung der While-Schleife in Algorithmus 18 weiter verarbeitet werden muss, haben alle Knoten, die weiterverarbeitet werden müssen, nach Step V ein Label.*

Algorithm 19: Optimal Parallel Contour Labeling, Part B

```
// Note: This is the second part of algorithm 18, p. 120
// Step V
Execute BasicContourLabeling(val, suc, _N ← m) // p. 17
// Step VI
t ← T
For Each Processor p, p ∈ {1, 2, ..., P} In Parallel Do
    For Each t, with: t ← T, decreasing to 1 Do
        If paused(p, t) is defined Then
            Do: reactivate Vertex paused(p, t)
        End
        t ← t - 1
        If save(p, t) is defined Then
            val(save(p, t).suc) ← val(save(p, t).v)
        End
        t ← t - 1
    End
End
```

Step VI: Weiterreichen der Label an restliche Knoten

Auf Basis der in Step III und Step V bestimmten Label können nun zusammen mit den in den Vektoren `save` und `paused` hinterlegten Informationen die Label allen Knoten mitgeteilt werden. Dazu wird das Ausdünnen der Graphen in der Steps 1-4 umfassenden While-Schleife unter Benutzung des Zeitzählers `t` exakt invers rückgängig gemacht. Das

Prinzip ist demnach bis auf zwei Punkte mit dem Step VI des Algorithmus 12 identisch, nämlich:

- Vor Invertierung jeder Shortcut-Operation aus Step III muss überprüft werden, ob für diesen Zeitschritt Knoten zu reaktivieren sind. Das kann für den gegebenen Prozessor und Zeitschritt mit dem Vektor `paused` geschehen.
- Ebenso wie die Shortcut-Operation in Step III, ist auch für Step VI die Operation im Vergleich mit dem List-Ranking-Algorithmus etwas anders. So ist hier das Label, ebenfalls gemäß Informationen aus `save`, einfach nur weiterzureichen.

Dies ist im Pseudocode für Algorithmus 19 im Abschnitt 'Step 6' zu sehen und beispielhaft in Abbildung 10.9 dargestellt. Letztere führt die Beispielserie aus Abbildungen 10.7 und 10.8 fort.

Es gilt: Alle am Anfang von Step VI in Algorithmus 19 vorliegenden Knoten haben ein Label (Hilfssatz 25) und alle wiedereingeführten (zuvor pausierten) Knoten haben ein Label (Hilfssatz 24). Daraus folgt:

Hilfssatz 26 *Alle als erstes (bei rückwärts laufender Zeit) in Step VI in Algorithmus 19 auftretenden Knoten jeder Linked-List haben ein Label.*

Ferner wird hier in Step VI, analog zu Cole und Vishkins Algorithmus, Step III invertiert. Das bedeutet im Falle des Algorithmus 19, die Label werden an alle nachfolgenden Knoten, die mithilfe des Vektors `save` wiedereingeführt werden, weitergereicht. Damit haben, unter Berücksichtigung von Hilfssatz 26, am Ende des Step VI alle Knoten ein Label. Wir halten fest:

Satz 7 *Algorithmus 18 / 19 ist ein Labeling-Algorithmus für zyklische gerichtete Linked-Lists. Und damit ein Kontur-Labeling-Algorithmus, wenn diese so vorliegen.*

Anmerkungen zu Vereinfachungen in Steps IV und VI

In dem Pseudocode des Kontur-Labeling-Algorithmus 18 / 19 gelten alle Vereinfachungen, welche auch für den optimalen List-Ranking-Algorithmus 12 gelten. Darauf wurde im Abschnitt 'Anmerkungen zu Vereinfachungen in Steps IV und VI' (Seite 59) eingegangen. Analog zu dem dort beschriebenen Vorgehen hinsichtlich des Wiedereinführens von Knoten aus dem Vektor `save` müsste die Anweisung

```
DO reactivate Vertex paused(p, t)
```

aus Pseudocode-Abschnitt Algorithmus 19 das Kopieren des entsprechenden Knotens an die zum Zeitpunkt t aktuelle Position beinhalten. So wäre die Gültigkeit der Verweise sichergestellt. Um möglichst nah bei Cole und Vishkins Formulierung des List-Ranking-Algorithmus zu bleiben, wurde das hier allerdings ebenfalls nicht explizit so gemacht.

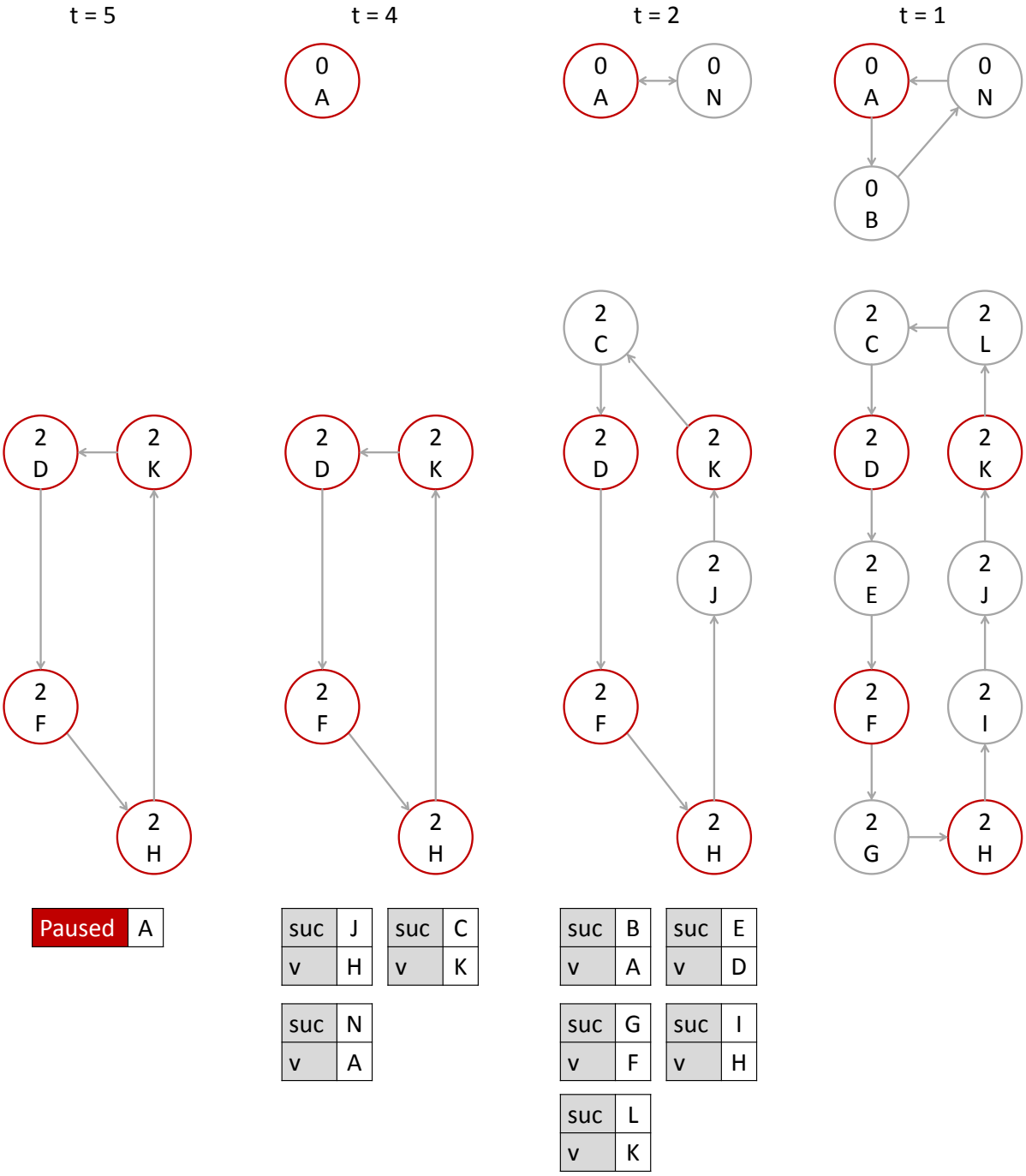


Abbildung 10.9.: Beispiel für einige Zeitschritte des Step VI aus Algorithmus 19, welche die Beispielserie aus Abbildungen 10.7 und 10.8 fortführen. Es wird gerade die Iteration des Step III, welche in Abb. 10.8 gegeben ist, invertiert. Zum Zeitpunkt $t = 5$ liegen die gelabelten Knoten vor, die in Abb. 10.8 berechnet wurden. Außerdem liegen der in Abb. 10.7 für diesen Zeitschritt angelegte Eintrag in `paused` vor. Wird der darin vermerkte Knoten reaktiviert, erhält man den Zustand zu Zeitpunkt $t = 4$. In den übrigen Zeitschritten werden die Label gemäß der Einträge in `save` weitergereicht. Zum Zeitpunkt $t = 1$ haben alle Knoten ihr Label. Hinweise: Verweise auf Nachfolger eigentlich nicht mehr explizit gegeben. Zeit $t = 3$ nicht gezeigt, da nichts passiert.

10.3.3. Asymptotische Analyse

Wie im bisherigen Verlauf des Kapitels beschrieben, wurde für den Kontur-Labeling-Algorithmus 18 / 19 das Prinzip von Cole und Vishkins optimalem List-Ranking-Algorithmus aus [CV89] übernommen. Es wird lediglich beim Vergleich zweier Knoten eine andere Operation (mit ebenfalls konstanten Kosten) ausgeführt. Eine weitere Modifikation, das zusätzliche Entfernen bzw. Pausieren der 'Einzelknoten', lässt den Algorithmus im Best-Case höchstens schneller werden. Ebenso hat die, bedingt durch die zyklische Form der Eingangsdaten, notwendige Modifikation der Abbruchbedingung der While-Schleife im verwendeten Basic-Contour-Labeling-Algorithmus 17 keinen Einfluss auf das asymptotische Verhalten. Ebenfalls wie beschrieben, verursachen alle Teilschritte in Algorithmus 18 / 19 asymptotisch die gleichen Kosten, benötigen die gleiche Laufzeit und Prozessorzahl. Damit gelten auch genau die gleichen Eigenschaften wie von Algorithmus 12, die in Abschnitt 7.8.1 (Seite 61) gegeben sind, für den Kontur-Labeling-Algorithmus 18 / 19:

Machine : CRCW-PRAM
Max Processors : $O(N/log_2(N))$
Running-Time : $O(N/P)$
Running-Time(pMax) : $O(log_2(N))$
Cost : $O(N)$

Kapitel 11.

Finden eines Labels je Connected-Component

Wie in Kapitel 9 beschrieben, sind alle Connected-Components durch Konturen vollständig von allen anderen Bereichen abgegrenzt. Darüber hinaus können Konturen in zwei sich gegenseitig ausschließende Kategorien eingeteilt werden: Innere und Äußere.

Eine äußere Kontur grenzt eine Connected-Component nach außen ab. Weil Connected-Components zusammenhängende Pixelbereiche sind, wird, gemäß der Vorschrift in Kapitel 9, genau eine Kontur die zugehörige Connected-Component nach außen begrenzen.

Eine Connected-Component kann keinen, einen oder auch mehrere andere Bereiche, die nicht oder anders klassifiziert sind, vollständig umschließen. Für jeden unabhängigen vollständig umschlossenen zusammenhängenden Bereich wird dann eine innere Kontur erzeugt. In Bezug auf diese inneren Konturen spielt es keine Rolle, ob ein umschlossener Bereich sich aus verschieden klassifizierten Teilbereichen zusammensetzt. Ein etwas komplexerer Datensatz, welcher mehrere verschachtelte Connected-Components samt daraus resultierender innere Konturen enthält, ist in Abbildung 11.1 dargestellt. Wir halten fest:

Satz 8 *Jede Kontur ist entweder eine innere oder eine äußere Kontur. Eine Connected-Component ist immer von genau einer äußeren Kontur umgeben und kann durch* $0, 1, 2, \ldots, O(N)$ *innere Konturen zu anders klassifizierten Bereichen abgegrenzt sein, welche die Connected-Component vollständig umschließt.*

Wir machen ferner noch einige Beobachtungen zu Kontursegmenten innerer und äußerer Konturen.

Beispielsweise liegt in den mit A und B markierten Pixeln in Abbildung 11.1 die gleiche Pixelumgebungskonfiguration vor. Hier sind jeweils die Pixel darüber sowie darunter gleich und alle anderen der Achter-Nachbarschaft anders klassifiziert. Allerdings gehört bei Pixel A das Segment mit DST-Typ `up` zu einer inneren Kontur und das mit DST-Typ `down` zu einer Äußeren. In Pixel B ist es trotz gleicher Pixelkonfiguration genau anders herum. Folglich kann für die Kontursegmente nicht anhand der lokalen Pixelkonfiguration entschieden werden, ob sie zu einer äußeren oder inneren Kontur gehören.

Innerhalb je eines Pixels sind verschiedene, aber nicht alle, Kombinationen von Kontursegmenten innerer und äußerer Konturen möglich. Die Möglichen sind im Einzelnen:

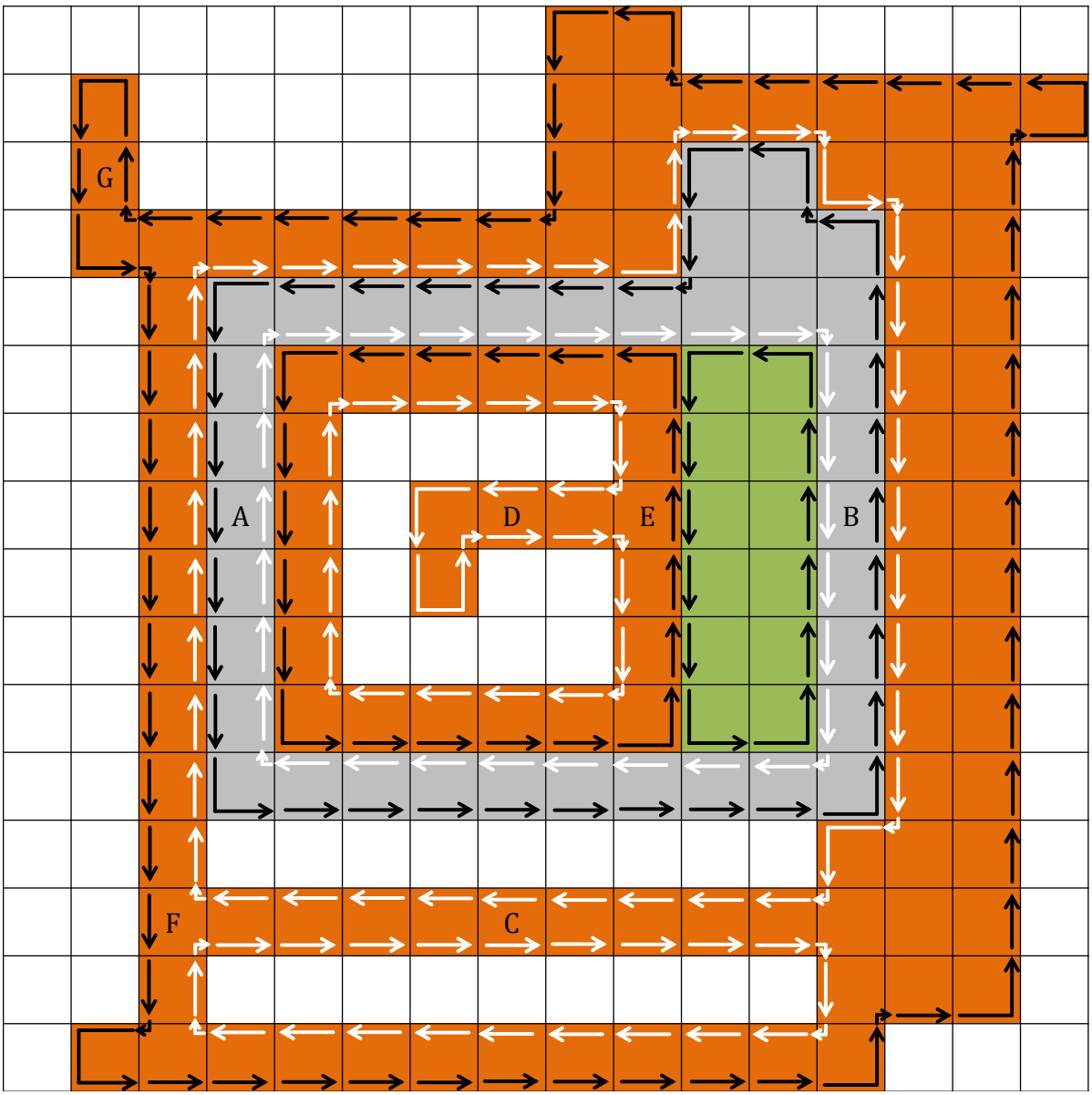


Abbildung 11.1.: Veranschaulichung verschachtelter Connected-Components und innerer Konturen. Farben der Pixel geben (unterschiedliche) Klassifikation an, weiß steht hierbei für nicht klassifiziert. Eingetragen sind alle extrahierten Kontursegmente äußerer (schwarz) und innerer (weiß) Konturen. Es gibt insgesamt vier Connected-Components und damit äußere Konturen, sowie vier innere Konturen. Buchstaben in Pixeln: Siehe Text

- Kontursegmente mehrerer verschiedener innerer Konturen. Beispielsweise dargestellt in Pixel C der Abbildung 11.1
- Kontursegmente der selben inneren Kontur, etwa in Pixel D
- Kontursegmente einer oder mehrerer innerer Konturen und Kontursegmente genau einer äußeren Kontur. Dargestellt z.B. in Pixeln E bzw. F
- Kontursegmente der selben äußeren Kontur, etwa in Pixel G

Dagegen ist insbesondere keine Kombination von Kontursegmenten verschiedener äußerer Konturen innerhalb eines Pixels möglich. Schließlich kann ein Pixel nur zu maximal einer Connected-Component gehören und diese hat genau eine äußere, geschlossene Kontur.

Wie in Kapitel 10 beschrieben, erhält jede Kontur ein eindeutiges Label. Weil zu einer Connected-Component mehrere Konturen gehören können, ist bislang unklar, welches dieser Label der Connected-Component zuzuweisen ist. Allerdings verfügt jede Connected-Component über genau eine äußere Kontur. Wird deren Label verwendet, kann jeder Connected-Component ein eindeutiges Label zugeordnet werden. Wir halten fest:

Satz 9 *Gemäß Satz 8 kann das Label der äußeren Kontur für die umschlossene Connected-Component als eindeutiges Label verwendet werden.*

Es ist folglich eine Unterscheidung zwischen inneren und äußeren Konturen erforderlich. Mit den Ausführungen der bisherigen Kapitel ist das nicht möglich und obige Ausführungen lassen eine Entscheidung basierend auf lokalen Eigenschaften in Pixeln unwahrscheinlich erscheinen.

11.1. Unterscheidung innerer- und äußerer Konturen

Generell ist zur Unterscheidung von inneren und äußeren Konturen Wissen über die gesamte Kontur von Nöten. Dieses kann während der Ausführung des Kontur-Labeling-Algorithmus aggregiert werden.

Ein mögliches Unterscheidungskriterium lässt sich auf zweierlei Arten formulieren. So folgt aus der Definition der Richtungen der Kontursegmente innerhalb eines Pixels, welche Kanten repräsentieren, dass der Drehsinn äußerer Konturen ebenfalls gegen den Uhrzeigersinn ist. Innere Konturen haben dagegen immer einen Drehsinn im Uhrzeigersinn, wie sich anhand der Abbildung 11.1 verifizieren lässt.

Anders dargestellt lässt sich dieser Sachverhalt jedoch einfacher überprüfen. Sei in Bezug auf einen Rand des Pixelgitters, z.B. den oberen, die minimale Tiefe `minDepth` eines Kontursegments definiert. Betrachten wir nun eines derjenigen Kontursegmente einer Kontur, deren Wert von `minDepth` in Bezug auf die gesamte Kontur minimal ist. Es kann dann offensichtlich keine Kontursegmente der selben Kontur geben, welche näher am Rand liegen. Segmente anderer Konturen können durchaus näher an dem Rand

Abbildung 11.2.: Veranschaulichung zur Unterscheidung innerer und äußerer Konturen gemäß Tiefe. Gezeigt ist jeweils eines der dem Rand am nächsten liegenden Kontursegmente einer Kontur und dessen `minDepth`-Wert in Abhängigkeit zur Koordinate des Pixels P. Die Klassifikation des mit ? beschrifteten Pixels ist für das Gezeigte irrelevant. Links: Kontursegment gehört zu äußerer Kontur. Rechts: Kontursegment gehört zu innerer Kontur.

liegen, aber dieser Sachverhalt ist für die zu treffende Entscheidung irrelevant. Dann kann anhand dieses Wertes die Entscheidung getroffen werden, ob das Segment zu einer inneren oder äußeren Kontur gehört. Es gibt zwei mögliche Fälle:

1. Das dem Rand am nächsten liegende Kontursegment einer Kontur repräsentiert (ggf. u.a.) die dem Rand zugewandte Pixelkante. Damit schließt das Segment eine Connected-Component zu diesem Rand hin ab. Eine solche Situation ist in Abbildung 11.2 (links) zu sehen. Da eine Connected-Component per Definition vollständig von einer äußeren Kontur umschlossen ist und zu diesem Rand hin keine mehr kommen kann, muss das Segment folglich zu einer äußeren Kontur gehören.

2. Das dem Rand am nächsten liegende Kontursegment einer Kontur repräsentiert nicht die dem Rand zugewandte Pixelkante. Damit schließt das Segment eine Connected-Component zu einer vom Rand abgewandten Seite hin ab. Abbildung 11.2 (rechts) veranschaulicht einen solchen Fall. Dann ist die Connected-Component zum Rand hin noch offen und kann nicht durch ein Segment dieser Kontur begrenzt werden, da dieses näher am Rand hin liegen muss. Folglich muss es eine weitere Kontur geben, welche die Connected-Component nach außen begrenzt und damit eine äußere Kontur ist. Da es genau eine äußere Kontur je Connected-Component gibt, muss das betrachtete Kontursegment zu einer inneren Kontur gehören.

Die Berechnung der Tiefe eines Kontursegments (`minDepth`) ist bereits mit der Formel 9.1 im Abschnitt 9.5.1 auf Seite 91 festgelegt. Sie besteht aus der Summe der immer gradzahligen Komponente $2 \cdot y$ und einer 0, falls das Kontursegment die dem Rand näher liegende Pixelkante repräsentiert oder einer 1 sonst.

Mithilfe des Wissens über eines der dem Rand am nächsten liegenden Kontursegmente kann die Unterscheidung für alle Kontursegmente vorgenommen werden. Dazu muss für das betrachtete Kontursegment das über alle Kontursegmente der ganzen Kontur aggregierte Minimum der `minDepth`-Werte in dessen Feld `minDepth` vorliegen.

Ist dieser Wert ganzzahlig, repräsentiert eines der dem Rand am nächsten liegenden Segmente die dem Rand zugewandte Pixelkannte. Es handelt sich demnach gerade um den obigen Fall (1) und damit um eine äußere Kontur. Bei ungradzahligen Werten des Minimums von `minDepth` liegt entsprechend Fall (2) vor und es handelt sich um ein Kontursegment einer inneren Kontur. Die zur Unterscheidung von inneren und äußeren Konturen benötigte Berechnung ist in Formel 11.1 gegeben.

$$(Min_{i \in Kontur}(minDepth_i)) \ mod\ 2 = \begin{cases} 0 \Rightarrow \text{äußere Kontur} \\ 1 \Rightarrow \text{innere Kontur} \end{cases} \tag{11.1}$$

Abschließend sei noch auf die Äquivalenz dieses Tests mit der Unterscheidung gemäß Drehsinn hingewiesen. Wie im Kapitel 9 beschrieben, besteht eine Kontur immer aus einer zyklischen Folge von in der Kontur aufeinanderfolgenden Kontursegmenten. Jedes dieser Kontursegmente repräsentiert eine oder mehrere Pixelkanten. Für diese Kanten sind in Kapitel 9.1.5 ab Seite 72 Richtungen definiert, sodass Kontursegmente innerhalb eines Pixels gegen den Uhrzeigersinn verlaufen. Dementsprechend hat eine Kontur immer einen eindeutigen Drehsinn im oder gegen den Uhrzeigersinn.

Sei nun die Tiefe in y-Richtung zur Unterscheidung der inneren Konturen herangezogen, wie in Abbildung 11.2 zu sehen. Dann handelt es sich um eine äußere Kontur, falls das dem Rand am nächsten liegende Kontursegment das Obere im Pixel ist. Weiterhin zeigt das obere Segment innerhalb eines Pixels per Definition immer nach links (Kapitel 9.1.5). Wenn die Richtung oben in der Kontur ‘links‘ ist und die Kontur einen eindeutigen Drehsinn hat, muss dieser gegen den Uhrzeigersinn sein. Durch analoge Überlegungen ergibt sich ein Drehsinn im Uhrzeigersinn für innere Konturen. Wir halten die Ergebnisse des Abschnitts 11.1 fest:

Satz 10 *Äußere Konturen haben immer einen Drehsinn gegen den Uhrzeigersinn und innere Konturen haben immer einen Drehsinn im Uhrzeigersinn. Die Überprüfung des Drehsinns ist äquivalent mit der Auswertung von Formel 11.1.*

Der in diesem Kapitel beschriebene Test wird im Falle des datenunabhängigen Ansatzes im Algorithmus 15 (Seite 106) verwendet. Der datenabhängige Ansatz kann optional entsprechend ergänzt werden. Gerade mit Blick auf eine effizientere Implementierung ist jedoch eine etwas andere Vorgehensweise geschickter. Die `minDepth` Information kann mit in den Wert von `val` der Knoten integriert werden, welcher auch so schon mit der `Min` Funktion für die einzelnen Konturen aggregiert wird. Das wird im Implementationsteil, und zwar in Abschnitt 21.2 ab Seite 222 genauer erläutert.

Kapitel 12.

Füllen der Konturen

Ausgangslage in diesem Kapitel sind Kontursegmente, für die bekannt ist, ob sie zu einer inneren oder äußeren Kontur gehören. In letzterem Fall liegt zusätzlich das Label der Kontur vor. In diesem Kapitel werden nun Möglichkeiten vorgestellt, den zugehörigen Pixeln im Innern der Konturen deren Label zuzuweisen.

Diese Situation ist beispielhaft in Abbildung 12.1 dargestellt. Darin könnte etwa Pixel A unmittelbar ein Label zugewiesen werden, da gelabelte Segmente einer äußeren Kontur in ihm enthalten sind, deren Label für ihn gilt. Pixel B in Abbildung 12.1 dagegen enthält beispielsweise keine Kontursegmente. Alternativ könnte - hier - schrittweise nach rechts, links, oben oder unten vorgegangen werden, bis eine Kontur getroffen wird. Die erste so getroffene Kontur gehört auf jeden Fall zu der Connected-Component, schließlich ist jede Connected-Component vollständig durch Konturen zu anderen Bereichen abgegrenzt. Es kann sich dabei jedoch um innere Konturen handeln, die nicht gelabeld sind und auch die zugehörige äußere Kontur nicht kennen. Im Fall von Pixel B könnte links bzw. unten eine äußere Kontur getroffen werden, oben bzw. rechts jedoch nicht. Insbesondere ist es möglich, dass in allen Richtungen innere Konturen getroffen werden, wie im Fall von Pixel C in Abbildung 12.1 dargestellt.

Die beschriebenen Probleme resultieren aus der Verschachtelung verschiedener Connected-Components ineinander. Dementsprechend wird der Fokus nachfolgender Abschnitte auf deren Auflösung (Abschnitt 12.3) oder zumindest Umgehung (Abschnitt 12.2) liegen.

12.1. Verschachtelung und Verschachtelungstiefe

Eine verschachtelte Connected-Component sei definiert als eine, welche vollständig von einer anderen Connected-Component umschlossen ist. Weil Connected-Components immer von genau einer äußeren Kontur vollständig umschlossen sind und diese sich nicht schneiden können, sind äußere Konturen in identischer Weise verschachtelt wie die entsprechenden Connected-Components selbst. Zur Veranschaulichung sind in Abbildung 12.1 die äußeren Konturen aller unverschachtelten Connected-Components mit einem (U) und die verschachtelten mit einem (V) gekennzeichnet.

Die Verschachtelungstiefe gibt die Anzahl der ineinander verschachtelten Connected-Components an. Sie ist definiert für Pixel bzw. Kontursegmente. Die Verschachtelungstiefe kann für einen Pixel mit einer leichten Abwandlung des Punkt-in-Polygon-Tests

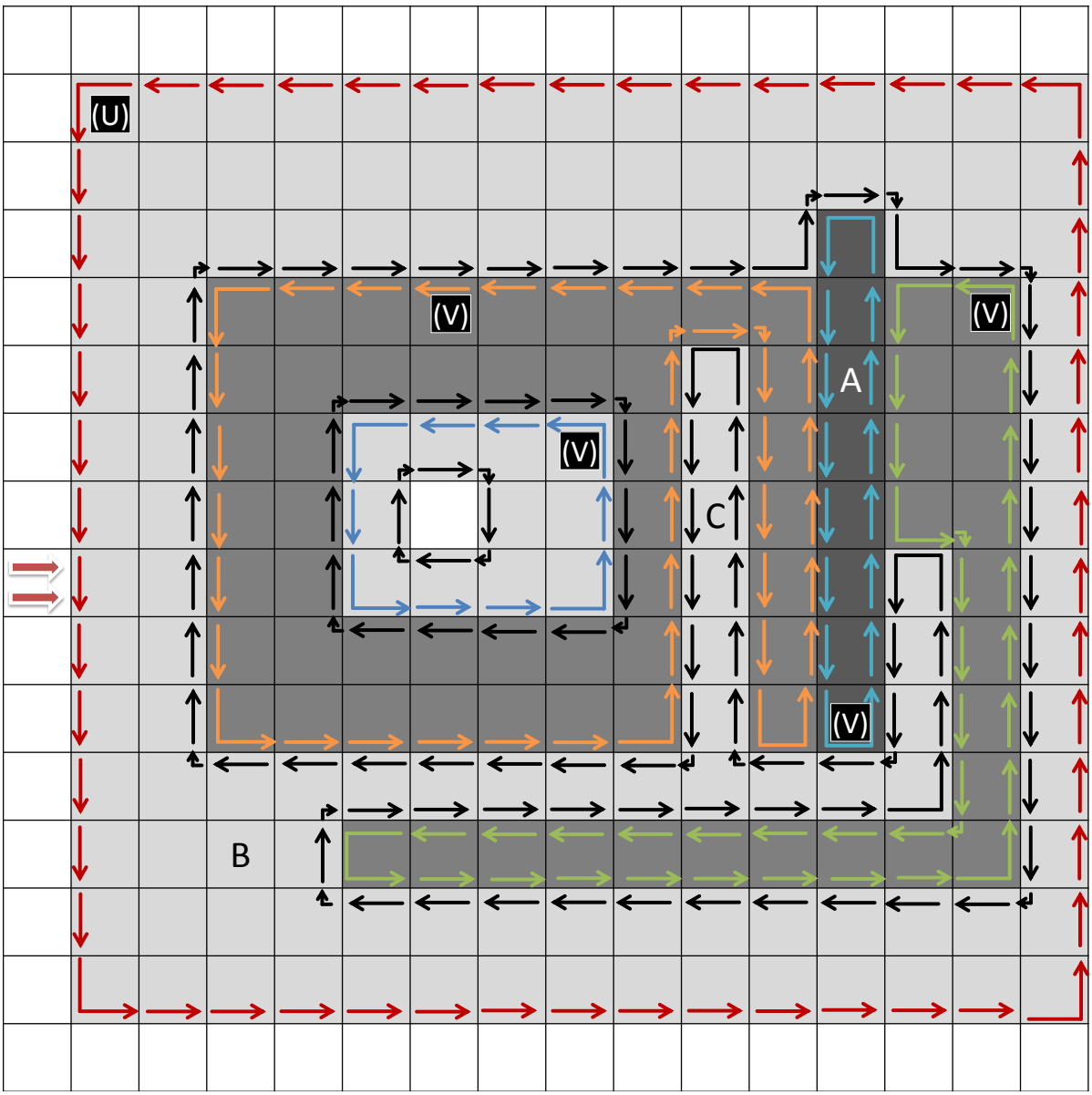


Abbildung 12.1.: Beispielhafte Ausgangssituation vor dem Labeln der Pixel. Deren Graustufe gibt die Pixelklassifikation an. Sonderfall (weiß dargestellt): Nicht klassifiziert. Segmente äußerer Konturen haben ein Label, dargestellt durch Farbe. Innere Konturen (schwarz dargestellt) verfügen nicht über ein Label. Buchstaben: Siehe Text. Links markierte Zeile wird später genauer betrachtet.

aus der 2D-Computergrafik bestimmt werden. Dazu kann man sich einen Strahl vorstellen, der von einem beliebigen Bildrand ausgeht und in dem betrachteten Pixel endet. Der Strahl hat einen Zähler mit dem Anfangswert 0 (außerhalb einer Connected-Component). Wann immer der Strahl in eine Connected-Component eindringt, wird der Zähler um eins erhöht und wenn er eine verlässt um eins verringert. Dabei werden nur die äußeren Konturen berücksichtigt. Da alle Connected-Components vollständig von geschlossenen äußeren Konturen umgeben sind, ist dieser Wert eindeutig, unabhängig davon, von welchem Rand der Strahl ausging. In Abbildung 12.2 wird dieser Sachverhalt für einen Pixel in Bezug auf alle vier Ränder veranschaulicht. Die Verschachtelungstiefe eines Kontursegments sei identisch mit der des assoziierten Pixels. Dies gilt für innere Konturen in gleicher Weise wie für Äußere. Offensichtlich gelten folgende Eigenschaften für die Verschachtelungstiefe:

- Die Verschachtelungstiefe ist für alle Pixel, und damit aller Kontursegmente, einer Connected-Component identisch
- Für alle Connected-Components cs gilt offensichtlich: Alle in cs verschachtelten Connected-Components haben eine höhere Verschachtelungstiefe als cs. Schließlich muss mindestens eine äußere Kontur durchdrungen werden, um in sie einzutreten. Insbesondere haben ineinander verschachtelte Connected-Components folglich niemals die gleiche Verschachtelungstiefe.
- Verschiedene, nicht ineinander verschachtelte, Connected-Components können die gleiche Verschachtelungstiefe haben. Siehe beispielsweise die orange, grün und blau umrandeten Connected-Components in Abbildung 12.2.
- Die maximale Verschachtelungstiefe entspricht $\min(X, Y)/2$. Das ist unmittelbar deutlich, weil jede Verschachtelung einen mindestens 1-Pixel breiten Rand, also links, rechts, über und unter der umschlossenen Connected-Component bewirkt. Eine tiefere Verschachtelung ist unmöglich, da dann das vollständige Umschließen nicht möglich wäre.
- Die minimale Verschachtelungstiefe von Pixeln ist null und von Kontursegmenten eins, da Kontursegmente im Gegensatz zu Pixeln nicht außerhalb von Connected-Components existieren können.

Eine weitere wichtige Beobachtung: Die Verschachtelungstiefe kann unabhängig in einer Zeile oder einer Spalte analysiert werden, was unmittelbar aus der Definition zur Bestimmung der Verschachtelungstiefe folgt. Das Problem kann folglich für alle Zeilen (oder alle Spalten) unabhängig und damit synchronisationsfrei parallel gelöst werden. Eine geeignete Wahl, ob das Problem zeilen- oder spaltenweise gelöst werden soll, kann immer durch das Verhältnis von X und Y bestimmt werden. Aus Gründen der Lesbarkeit wird in den folgenden Abschnitten willkürlich die zeilenweise Lösung diskutiert.

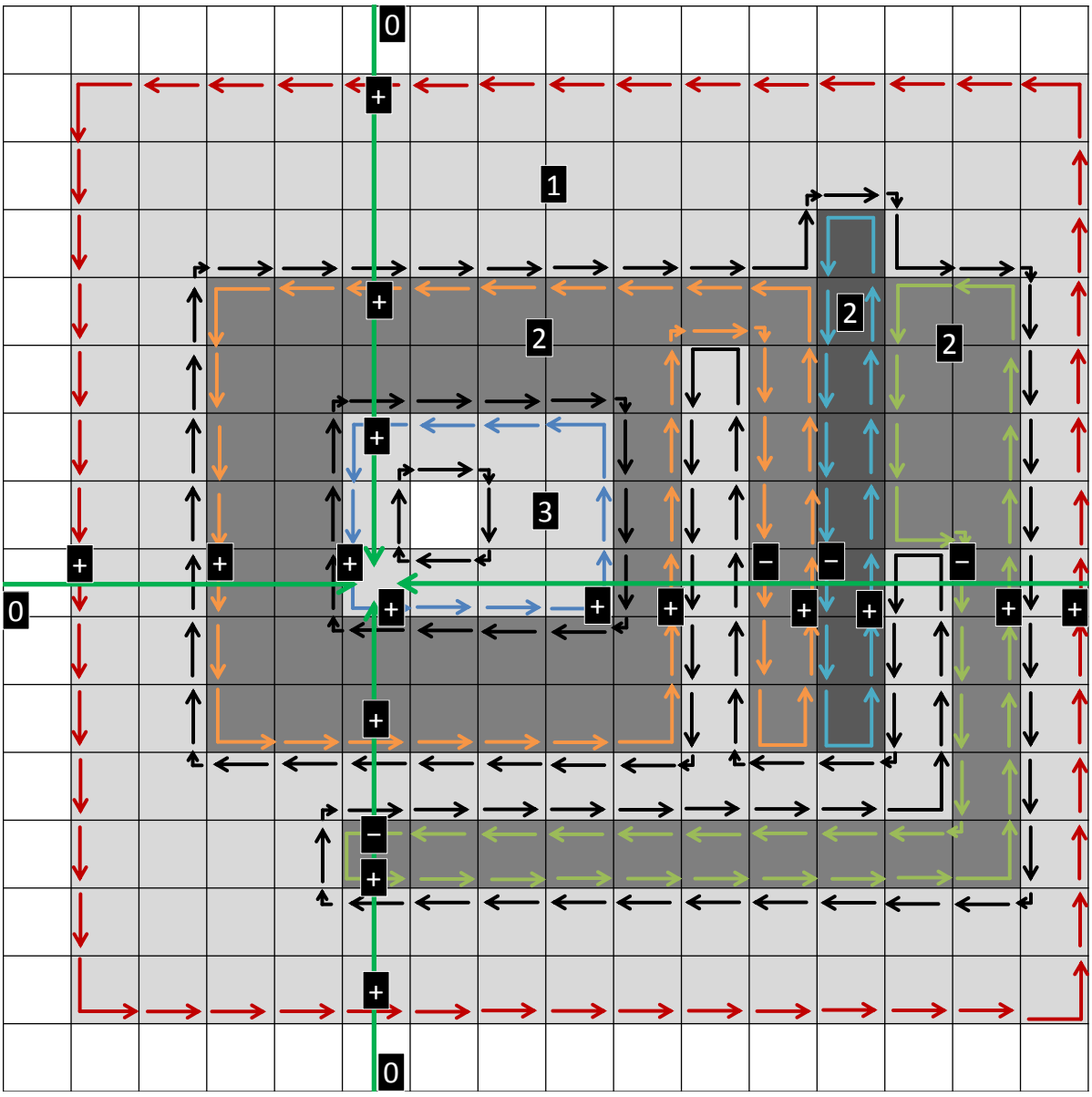


Abbildung 12.2.: Veranschaulichung zur Bestimmung der Verschachtelungstiefe für einen Pixel in Bezug zu allen vier Rändern (grüne Pfeile). Zähler für Verschachtelungstiefe ist am Rand 0 und erhöht sich bei jedem Eintritt in eine Connected-Component um 1, dargestellt durch ein (+). Beim Verlassen verringert sich der Zähler um 1, dargestellt durch ein (−). Dies passiert jeweils nur beim Übertreten äußerer Kontursegmente (farbig dargestellt). Zahlen (außer 0) geben die Verschachtelungstiefe für jede Connected-Component an. Sie ist für jeweils alle Pixel gleich.

12.2. Ansatz 1: Füllen verschachtelter Konturen

In diesem Abschnitt wird eine Technik vorgestellt, mit der die Pixel innerhalb von Connected-Components direkt die Label der zugehörigen äußeren Konturen erhalten. Dazu werden die einzelnen Zeilen parallel verarbeitet aber innerhalb einer Zeile wird sequentiell vorgegangen. Ziel des Ansatzes ist es, die Anzahl der Operationen, nicht nur asymptotisch, minimal zu halten. Hier wird jeder Pixel genau einmal und mit einem für die spätere Implementation einfachen Zugriffsmuster betrachtet.

12.2.1. Stack-basierte Vorgehensweise und Sonderfälle beim Füllen

Um die Verschachtelungen zu behandeln, wird ein Label-Stack eingeführt. Auf diesen wird, im Falle eines Eintritts in eine Connected-Component über eine äußere Kontur, das Label der übertretenen Kontur gelegt. Beim Verlassen einer Connected-Component wird das oberste Label des Stacks entfernt. Bei einem sequentiellen Vorgehen durch die Pixelzeile ausgehend von einem Rand befindet sich so immer das Label der äußeren Kontur, innerhalb der man sich gerade befindet, oben auf dem Stack. Wenn der aktuelle Pixel zu der äußeren Kontur gehört, in der man sich gerade befindet, wird das Label ihm zugewiesen. In zwei Situationen ist dies nicht der Fall, die dementsprechend zuvor überprüft werden müssen:

1. Nicht klassifizierte Bereiche sind nicht von einer äußeren Kontur umgeben und werden daher durch die bisherigen, auf Konturen basierten Entscheidungen, nicht anders behandelt als Bereiche innerhalb von Konturen. Dies kann behoben werden, indem ein Pixel höchstens dann gelabeld wird, wenn er klassifiziert ist.

2. Außerdem muss der Sonderfall von ein Pixel großen Connected-Components betrachtet werden, die direkt gelabeld wurden und für die ebenfalls keine äußere Kontur existiert. Um zu verhindern, dass ihnen ein falsches Label zugewiesen wird, muss überprüft werden, ob ein Pixel bereits ein Label hat, was nur in diesem Sonderfall möglich ist. Ein Pixel mit einem Label darf folglich nicht mit einem Label versehen werden.

Der auf diesen Überlegungen basierende Sub-Algorithmus zum Füllen verschachtelter Konturen ist im Pseudocode-Abschnitt 20 gegeben.

In dem Algorithmus wird zunächst je Zeile ein Stack für Label in Form eines Arrays benötigt, dessen Größe der maximal möglichen Verschachtelungstiefe entspricht. Dann werden, ausgehend vom linken Rand, sequentiell alle Pixel der Zeile verarbeitet.

In jedem Pixel wird zunächst die Anzahl der Ein- bzw. Austritte in äußere Konturen von Connected-Components bestimmt. Diese Größe ist, in Bezug auf eine Füllrichtung, für jedes Kontursegment gemäß SRC-DST-Typ definiert. Dabei handelt es sich um den Wert `ioCnt`, welcher in Kapitel 9.5.2 auf Seite 9.5.2 eingeführt wurde. In Bezug auf eine Füllrichtung möglich sind 0, 1 oder 2 Ein- bzw. Austritte. Für jeden Pixel wird nun die Gesamtzahl der Ein- bzw. Austritte `p.ioCnt` bestimmt. Dafür wird die Summe über alle Werte von `ioCnt` der zu `p` gehörigen Kontursegmente bestimmt, die zu einer äußeren

Algorithm 20: fillContour

```
For Each row y, y ∈ {0, 1, ... Y − 1} In Parallel Do
    Stack labelStack
    For Each column x, x from 0 to X - 1 In Order Do
        p ← y · X + x
        If label(p) = UNLABELD Then
            ioCnt(p) ← 0
            For Each Contour-Seg c, c ∈ {RS(p), LS(p), DS(p), US(p)} Do
                If cLabel(c) ≠ UNLABELD Then
                    ioCnt(p) ← ioCnt(p) + ioCnt(c)
                    label(p) ← cLabel(c)
                End
            End
            If ioCnt(p) = 1 Then
                If labelStack.top() = label(p) Then
                    Call labelStack.pop()
                Else
                    Call labelStack.push(label(p))
                End
            End
            If ioCnt(p) = 0 ∧ class(p) ≠ UNCLASSIFIED Then
                label(p) ← labelStack.top()
            End
        End
    End
End
```

Kontur gehören. Ein Kontursegment gehört genau dann zu einer äußeren Kontur, wenn es bereits über ein Label verfügt. Der Wert von `p.ioCnt` ist ebenfalls entweder 0 (keine äußere Kontur im Pixel), 1 (Kontursegment für linke oder rechte Pixelkante existiert und gehört zu äußerer Kontur) oder 2 (äußere Kontursegmente für linke und rechte Kante existieren). Abbildung 12.3 zeigt einige Beispiele, in denen `p.ioCnt` in Bezug auf eine Füllrichtung von links 0, 1 oder 2 ist. Außerdem wird, wenn mindestens eines der Kontursegmente von `p` ein Label hat, dieses dem Pixel übergeben. Es kann mehrere Kontursegmente in `p` mit Label geben, welche dann aber identisch sind. Schließlich ist eine Connected-Components von genau einer äußeren Kontur umgeben und diese hat ein Label, das für alle zugehörigen Segmente gleich ist.

Falls `p.ioCnt` den Wert eins hat, wird in `p` eine äußere Kontur betreten (dann wird `p.label` auf den Stack gelegt) oder verlassen. In letzterem Fall wird das oberste Element vom Stack genommen. Welcher von beiden Fällen vorliegt, kann durch Vergleich des Wertes `p.label` und `labelStack.top()` ermittelt werden. Sind beide identisch, ist bereits in einem vorherigen Pixel in die Connected-Component eingedrungen worden und folglich muss sie hier verlassen werden. Andernfalls wird die Connected-Component

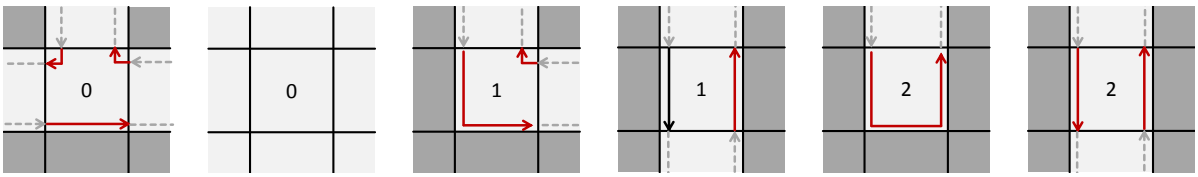


Abbildung 12.3.: Beispiele für Pixelkonfigurationen, deren `ioCnt` bei waagerechter Füllrichtung 0, 1, oder 2 ist. Betrachtet wird jeweils der mittlere Pixel und die eingetragene Zahl ist dessen `ioCnt`. Rote Pfeile gehören zu äußeren Konturen und der schwarze Pfeil zu einer inneren Kontur.

im aktuellen Pixel betreten.

Falls `p.ioCnt` den Wert zwei hat, wird in `p` eine äußere Kontur betreten und auch wieder verlassen. Folglich kann der Stack unverändert bleiben. Es ist auch sonst nichts zu tun, da bereits zuvor das Label eines der Kontursegmente zugewiesen wurde.

Falls `p.ioCnt` den Wert null hat und der Pixel klassifiziert ist, liegt er im inneren einer Connected-Component. Wenn zusätzlich nicht der 1-Pixel-Connected-Component Sonderfall vorliegt, muss er das oberste Label des Stacks erhalten.

Nachdem der Sub-Algorithmus 20 alle Pixel verarbeitet hat, verfügen alle Pixel im Inneren von Connected-Components über ein Label und das Connected-Component-Problem ist damit gelöst. Das Prinzip wird beispielhaft in Abbildung 12.4 anhand einer Zeile veranschaulicht. Dabei handelt es sich um die markierte Zeile aus dem in Abbildung 12.1 gegebenen Beispiel.

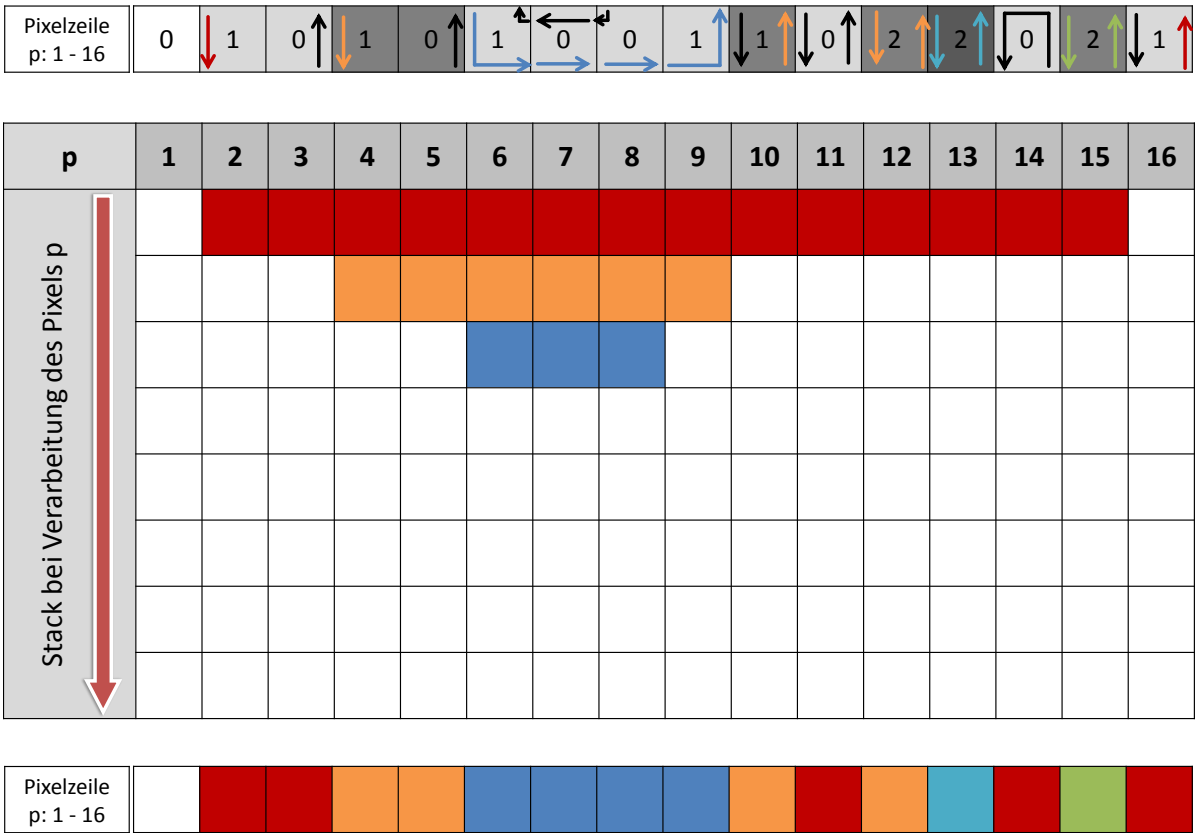


Abbildung 12.4.: Veranschaulichung des Stack-basierten Füllens. **Ausgangsdaten (oben)**: Zeile aus Pixeln mit Indices `p`: 1, ..., 16. Graustufen: Pixelklassifikation. Zahlen in Pixeln: `p.ioCnt`. Farben der Pfeile geben Klassifikation an. Schwarze Pfeile: Gehören zu inneren Konturen. **Tabelle (Mitte)**: Jede der 16 Spalten gibt den Stack nach Verarbeitung des Pixels mit entsprechendem Index an. Top Element ist jeweils das unterste farbige. **Verarbeitete Daten (unten)**: Jeder klassifizierte Pixel hat ein Label. Damit ist der CCL-Algorithmus abgeschlossen. Hinweis: Sonderfälle sind hier nicht zu erkennen

12.2.2. Asymptotisches Verhalten

Der in diesem Abschnitt beschriebene Füll-Algorithmus verarbeitet alle Zeilen oder alle Spalten unabhängig parallel. Daraus folgt die Ausführbarkeit auf einem EREW-PRAM. Bei paralleler Verarbeitung der Zeilen müssen diese jedoch sequentiell abgearbeitet werden. Dies gilt analog für die alternative parallele Verarbeitung der Spalten. Abhängig davon, ob X oder Y größer ist, kann in der jeweils größeren Dimension parallel und in der kleineren sequentiell gearbeitet werden. Dementsprechend können maximal $\max(X, Y)$ Prozessoren verwendet werden, die dann jeweils $\min(X, Y)$ sequentielle Schritte ausführen müssen. Werden um einen Faktor k weniger Prozessoren verwendet, erhöht sich die Anzahl sequentieller Schritte entsprechend um Faktor k. Zusammengefasst ergeben sich die asymptotischen Eigenschaften des Algorithmus, der verschachtelte Connected-Components direkt füllt, zu:

$$\begin{aligned}
\textbf{Machine} &: \text{EREW-PRAM} \\
\textbf{Max Processors} &: O(\max(X, Y) \geq \sqrt{N}) \\
\textbf{Work} &: O(N) \\
\textbf{Running-Time} &: O(N/P) \\
\textbf{Cost} &: O(N)
\end{aligned}$$

12.3. Ansatz 2: Auflösen der Verschachtelung vor Füllung

Für diesen zum im vorherigen Abschnitt vorgestellten Ansatz alternativen Füll-Algorithmus werden vor der eigentlichen Füllung zunächst die Verschachtelungen behoben, sodass für jedes innere Kontursegment auf einfache Weise ein zugehöriges äußeres Kontursegment (und damit Label) gefunden werden kann. So kann die sequentielle Füllung pro Zeile vermieden und die asymptotische Laufzeit verbessert werden.

12.3.1. Definition von Ein- und Austrittssegmenten in Bezug auf Zählrichtung und Datenlayout

Um die Verschachtelungstiefe für eine Position `i` bestimmen zu können wird die Differenz der Summen der Ein- und Austritte in Connected-Components über äußere Konturen, ausgehend von einem Rand, bis `i` berechnet. Abhängig vom Rand, in Bezug auf den die Verschachtelungstiefe bestimmt wird, bewirken dies andere Kontursegmenttypen. Siehe dazu auch das Beispiel in Abbildung 12.2. Da hier von zeilenweiser Füllung gesprochen wird, kommen nur die *Zählrichtungen* vom linken Rand und vom rechten Rand in Frage, welche theoretisch äquivalent sind. Für die Formulierung des Algorithmus ist jedoch die Zählrichtung ausgehend vom linken Rand bis zur betrachteten Position einfacher, weshalb diese verwendet wird.

Bevor der eigentliche Algorithmus ausgeführt werden kann, müssen noch einige Vorarbeiten in Bezug auf die Kontursegmente geschehen, welche auf die zum Füllen in Zählrichtung nötigen Informationen reduziert werden. Diese Überführung ist beispielhaft für eine Zeile in Abbildung 12.5 zu sehen. Bei dieser Zeile handelt es sich um die Markierte des in Abbildung 12.1 gezeigten vollständigen Connected-Component-Problems. Es werden alle Verbindungskonturstücke entfernt, da diese keine Pixelkanten repräsentieren. Außerdem werden alle waagerechten Kontursegmente gelöscht. Schließlich repräsentieren sie keine Pixelkanten in der gewählten Zählrichtung. Würde eine spaltenweise Füllung verwendet, müssten sie dagegen erhalten bleiben.

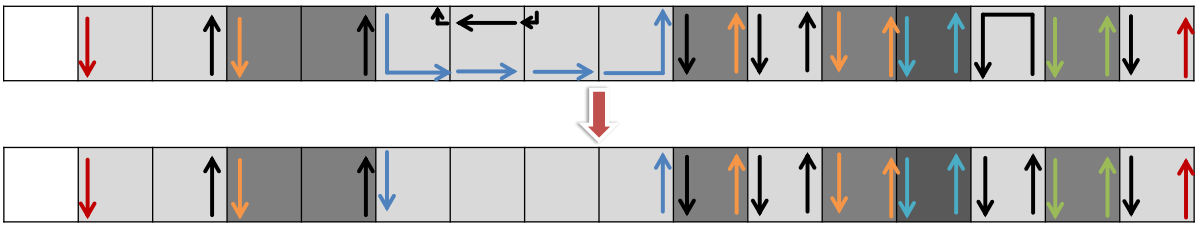


Abbildung 12.5.: Reduzierung auf benötigte Informationen der Kontursegmente für zeilenweise Füllung. Verbindungskonturstücke werden entfernt und die übrigen Kontursegmente werden in ihre Bestandteile zerlegt, welche je eine Pixelkante repräsentieren. Dann werden alle waagerechten Kontursegmente ebenfalls entfernt. Hinweis: Die Zeile stammt aus Abbildung 12.1

Die bisher besprochenen Kontursegmenttypen können mehrere Pixelkanten in Zählrichtung repräsentieren. Deshalb werden sie in ihre Bestandteile zerlegt. Dies ist beispielsweise in der Abbildung 12.5 im dritten Pixel von rechts passiert. Auf diese Weise zusätzlich entstandene waagerechte Komponenten werden entfernt. Hinweis: Die Segmente erfüllen jetzt nicht mehr das bisherige SRC-DST Schema. Diese Informationen werden aber auch nicht mehr benötigt. Von den Kontursegmenten werden demnach nur noch die Information über die repräsentierte Pixelkante und das Label (speziell dessen Vorhandensein) benötigt. Aus Gründen der Einheitlichkeit werden sie aber z.B. weiterhin gerichtet gezeichnet. Die Richtung ist weiterhin äquivalent mit der repräsentierten Pixelkante, wird aber ebenso wie Nachfolger, Vorgänger, etc., nicht mehr benötigt.

Aus Gründen der Einfachheit verwenden wir für dieses Kapitel nun eine zweidimensionale Indizierung der Datenstrukturen. Das Layout der Daten jeder Zeile ist ebenfalls etwas geändert, nämlich folgendermaßen:

- Es gibt X Pixel mit Adressen $0, 1, 2, \ldots, X-1$
- Es gibt zwei mögliche Kontursegmente, Repräsentanten der linken und rechten Kante, je Pixel. Somit ergeben sich $2{\cdot}X$ Kontursegmente mit Adressen $0, 1, 2, \ldots, 2{\cdot}X-1$.
- Ein Kontursegment eines Pixels mit Adresse `p`, welches die linke Kante repräsentiert, befindet sich an Position $2 \cdot p$.
- Ein Kontursegment von `p`, welches die rechte Kante repräsentiert, befindet sich an Position $2 \cdot p + 1$.

Mit obigen Definitionen ist ein Kontursegment ein Eintrittssegment in Bezug auf die Füllrichtung ausgehend vom linken Rand genau dann wenn gilt:

1. Gehört zu äußerer Kontur ($\Leftrightarrow$ hat ein Label)
2. Befindet sich an Position $2{\cdot}i$, mit $i \in \{0, 1, 2, \ldots\}$ ($\Leftrightarrow$ repräsentiert linke Pixelkante)

Analog handelt es sich um ein Austrittssegment genau dann wenn gilt:

1. Gehört zu äußerer Kontur ($\Leftrightarrow$ hat ein Label)
2. Befindet sich an Position $2 \cdot i + 1$, mit $i \in \{0, 1, 2, \ldots\}$ ($\Leftrightarrow$ repräsentiert rechte Pixelkante)

Die Datenstrukturen für Liniensegmente bestehen nun aus Y Zeilen und $2 \cdot X$ Spalten. Alle, wie beschrieben geänderten, Daten der Liniensegmente mögen von nun an in folgender Form vorliegen:

`exists:` Enthält Information, ob ein Segment existiert (1) oder nicht (0). Die Werte können auch als `TRUE` und `FALSE` interpretiert werden. Letzteres gilt für alle Datenstrukturen.

`cLabel`: Enthält Label (oder Konstante `UNLABELD`) der Kontursegmente

`in`: Segment ist Eintrittssegment (1) oder nicht (0)

`out`: Segment ist Austrittssegment (1) oder nicht (0)

`outer`: Segment ist entweder Ein- oder Austrittssegment (1) oder keines von beiden (0). Entspricht somit komponentenweise Summe aus `in` und `out`.

`originalId`: Speichert die ursprüngliche Adresse eines Segments in seiner Zeile. Anfangs gilt also: `originalId`(y)(i) = i für alle i und y.

12.3.2. Labeln der inneren Kontursegmente

Der Sub-Algorithmus zum Labeln der inneren Kontursegmente ist im Pseudocodeabschnitt 21 gegeben und in Abbildung 12.6 beispielhaft für eine Zeile veranschaulicht. Es ist empfehlenswert, beide zusammen mit dem nachfolgenden Text zu lesen. Da alle Operationen für alle Zeilen unabhängig voneinander und parallel ausgeführt werden, sind im Folgenden die je Zeile nötigen Operationen beschrieben. Diese gelten, auch wenn nicht explizit angegeben, immer für alle Zeilen. Wir lassen daher hier in den Erläuterungen den Zeilenindex `y` weg.

I Bestimmung der Verschachtelungstiefe: Zunächst wird die Verschachtelungstiefe bestimmt (siehe je Abschnitt 1 in Sub-Algorithmus 21 und Abbildung 12.6).

Zuerst wird ein Inclusive-Scan auf `in` angewandt und das Ergebnis in `scndIn` gespeichert. Letzteres enthält dann in jeder Position i die Anzahl der Eintrittssegmente vom linken Rand bis einschließlich dieser Position. Es muss einschließlich i sein, weil man sich bereits in der entsprechenden Connected-Component befindet, wenn die aktuelle Position die eines Kontursegments ist. Die Werte in `scndIn` geben entsprechend an, wie groß die Verschachtelungstiefe wäre, wenn man nie eine Connected-Component verlassen hätte. Hinweis: Über eine ganze Zeile wäre das nicht möglich, da am Ende alles wieder verlassen sein muss.

Außerdem wird ein Exclusive-Scan auf `out` angewandt und das Ergebnis in `scndOut` gespeichert. Dann enthält dieses Array an Position i die Anzahl der Austrittsegmente vom linken Rand bis zu Position $i-1$. Diesmal muss es exclusive sein, da man sich an der Position eines Austrittssegments noch innerhalb einer Connected-Component befindet und diese erst an der nächsten Indexposition verlassen hat.

Der Verschachtelungstiefenvektor `nDepth` für alle Kontursegmentpositionen ergibt sich dann offensichtlich durch Vektorsubtraktion gemäß `scndIn`−`scndOut`.

Abbildung 12.6.: Labeln innerer Kontursegmente durch Auflösung der Verschachtelung bei Zählrichtung für Verschachtelungstiefe ausgehend vom linken Rand. Links neben Arrays: Beschreibung oder Name. Die Ausgangslage setzt das in Abbildung 12.5 gegebene Beispiel fort

I Berechne Verschachtelungstiefe (nDepth)

- Für alle Segs: Setze in 'in' 1 für Eintrittsseg und in 'out' 1 für Austrittsseg

E: in	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	0	0
E: out	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1

- Wende inclusive Scan auf 'in' und exclusive Scan auf 'out' an

iScan(in)	0	0	1	1	1	1	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	4	4	5	5	5	5	6	6	6	6
eScan(out)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2	2	2	2	3	3	4	4	4	4	5	5

- Berechne nDepth durch Vektorsubtraktion: iScan(in) - eScan(out)

nDepth	0	0	1	1	1	1	2	2	2	2	3	3	3	3	3	3	3	3	2	2	1	1	2	2	2	2	1	1	2	2	1	1

II Sortieren gemäß nDepth

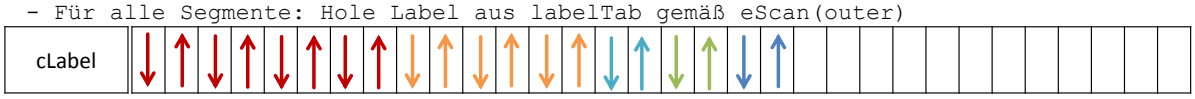
- Sortiere Daten der Segmente aus gemäß nDepth. Behalte Ordnung gleicher Schlüssel
- Speichere Ergebnis in gleichen Datenstrukturen

III Labeln innerer Segmente

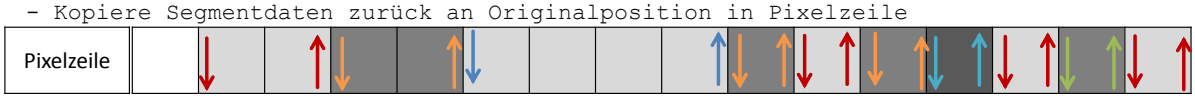
- Wende Scan auf äußere Segmente an. Compacte deren Label entsprechend in labelTab

E: outer	1	0	0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
eScan(outer)	0	1	1	1	1	1	1	1	2	3	3	3	4	5	6	7	8	9	10	11	12	12	12	12	12	12	12	12	12	12	12	12

- Für alle Segmente: Hole Label aus labelTab gemäß eScan(outer)



- Kopiere Segmentdaten zurück an Originalposition in Pixelzeile



Algorithm 21: LabelInnerCS

```
For Each Row y, y ∈ {0, 1, ..., Y − 1} In Parallel Do
    // Part 1: Compute nesting-depth
    scndIn(y) ← Execute parScan_Inclusive(in(y))
    scndOut(y) ← Execute parScan_Exclusive(out(y))
    nDepth(y) ← Execute parVecSubtract(scndIn(y), scndOut(y))
    // Part 2: Sort per row considering nesting-depth
    Do: Sort segment arrays (exists, in, out, outer, originalId, cLabel) in parallel
    per row by their nDepth values. Results are stored in same arrays.
    // Part 3: Label inner CS
    scndOuter(y) ← Execute parScan_Exclusive(outer(y))
    labelTab(y) ← Execute parCompact(outer(y), scndOuter(y), cLabel(y))
    For Each x, x ∈ {0, 1, 2, ..., 2 · X − 1} In Parallel Do
        // Apply labels to all contour-segments
        If existing(y)(x) Then
            cLabel(y)(x) ← labelTab(y)(scndOuter(y)(x))
        End
    End
    Do: Copy elements of segment arrays exists and cLabel in parallel to
    original positions, given by originalId. Results are stored in same arrays.
End
```

II Auflösen der Verschachtelung: Anschließend werden die Daten aller Kontursegmente je einer Zeile durch ein stabiles Sortierverfahren mit der Verschachtelungstiefe als Schlüssel sortiert. Die sortierten Daten werden jeweils im gleichen Array wieder abgelegt. Dieser Schritt entspricht Phase 2 in Sub-Algorithmus 21 und Abbildung 12.6. Unter Beachtung der in Abschnitt 12.1 beschriebenen Verschachtelungseigenschaften gilt dann:

- Alle Kontursegmente gleicher Verschachtelungstiefe liegen als lückenlose Folge vor. ⇒ Alle ursprünglichen Verschachtelungen sind aufgelöst, da diese unterschiedliche Verschachtelungstiefen hatten.
- Die Reihenfolge von Kontursegmenten gleichen Schlüssels (der Verschachtelungstiefe) ist unverändert. Dies impliziert zwei weitere erforderliche Eigenschaften:
  - Es sind keine neuen Verschachtelungen entstanden
  - Alle inneren Kontursegmente liegen eingerahmt zwischen den zugehörigen äußeren Kontursegmenten.

Anstelle des parallelen Radix-Sorts kann auch ein anderes Sortierverfahren verwendet werden, welches aber die Beibehaltung der Reihenfolge von zu sortierenden Einträgen mit gleichem Schlüssel garantieren muss.

III Innere Kontursegmente bekommen Label der zugehörigen Äußeren: Nun liegen alle inneren Kontursegmente zwischen den zugehörigen äußeren Kontursegmenten und es befinden sich insbesondere keine Segmente anderer Connected-Components dazwischen. Jetzt kann auf einfache Weise eine Art Label-Tabelle `labelTab` erstellt werden. Dafür wird zunächst ein Exclusive-Scan auf `outer` angewandt und das Ergebnis in `scndOuter` gespeichert. Anschließend können die Label aus `cLabel` gemäß den Indices aus `scndOuter` in die dichte Datenstruktur `labelTab` überführt werden. Weil sich innere Kontursegmente immer zwischen den zugehörigen Äußeren befinden und der Index in `scndOuter` durch Innere unbeeinflusst bleibt, kann jedes (insbesondere auch innere) Kontursegment `i` das Label eines zugehörigen äußeren Kontursegments bekommen über:

```
cLabel(i) <- labelTab(scndOuter(i))
```

Äußere Kontursegmente erhalten so ihr eigenes Label zurück, weshalb für sie keine Sonderfallbetrachtung nötig ist. Abschließend müssen die noch benötigten Daten der Kontursegmente wieder an ihre originalen Adressen in der Pixelzeile zurückkopiert werden.

Das vorgestellte Labeln der Kontursegmente ist jeweils in Abschnitt III im Codeabschnitt 21 gegeben bzw. in Abbildung 12.6 dargestellt.

12.3.3. Labeln der Pixel und Sonderfallbetrachtung

Nachdem alle Kontursegmente über ein Label verfügen, können die Pixel mit einem Label versehen werden. Siehe dazu den Pseudocode in Sub-Algorithmus 22 und zur Veranschaulichung die Abbildung 12.7, welche die Fortsetzung des Beispiels aus Abbildung 12.6 darstellt.

Offensichtlich sind alle mit einem Label zu versehenden Pixel nun von den, inzwischen gelabelten, Kontursegmenten eingerahmt. Für jeden Pixel lassen sich die zugehörigen Kontursegmente und damit die Label erneut mit einer Scan- / Compact- Kombination bestimmen. Dazu wird zunächst ein Exclusive-Scan auf `exists` angewandt und das Resultat in `scndExists` gespeichert. Damit können die Werte aus `cLabel` in eine dichte Datenstruktur (ebenfalls in `cLabel`) überführt werden. Wenn ein Pixel mit Index `p` und x-Koordinate `x` gelabeld werden muss, so ist das Label:

```
label(p) <- cLabel(scndExists(2 * x))
```

Der Faktor $2 \cdot x$ resultiert dabei daraus, dass zu einem Pixel zwei Kontursegmentadressen gehören. Mit obiger Auswahl wird garantiert das Linke gewählt.

Dieses so gefundene Label darf dem jeweiligen Pixel aber nur übergeben werden, wenn er zu einer Connected-Component gehört ($\Leftrightarrow$ Klassifikation vorhanden) und es sich nicht um den 1-Pixel Connected-Component Sonderfall handelt ($\Leftrightarrow$ Pixel-Label nicht bereits vorhanden). Die Sonderfälle sind damit identisch zu denen im ersten Ansatz, der in Abschnitt 12.3 beschrieben ist.

Nachdem der Sub-Algorithmus 22 terminiert, verfügen alle Pixel im Inneren von Connected-Components über ein Label und das Connected-Component-Problem ist damit gelöst. Das Ergebnis ist identisch zu dem des Ansatzes 1, bei welchem verschachtelte Konturen direkt gefüllt wurden.

Algorithm 22: LabelPixel

```
For Each Row y, y ∈ {0, 1, ..., Y − 1} In Parallel Do
    scndExists(y) ← Execute parScan_Exclusive(exists(y))
    cLabel(y) ← Execute parCompact(exists(y), scndExists(y), cLabel(y))
    For Each Column x, x ∈ {0, 1, ..., X − 1} In Parallel Do
        p ← y · X + x
        If class(p) ≠ UNCLASSIFIED ∧ label(p) = UNLABELD Then
            label(p) ← cLabel(y)(scndExists(y)(2 · x))
        End
    End
End
```

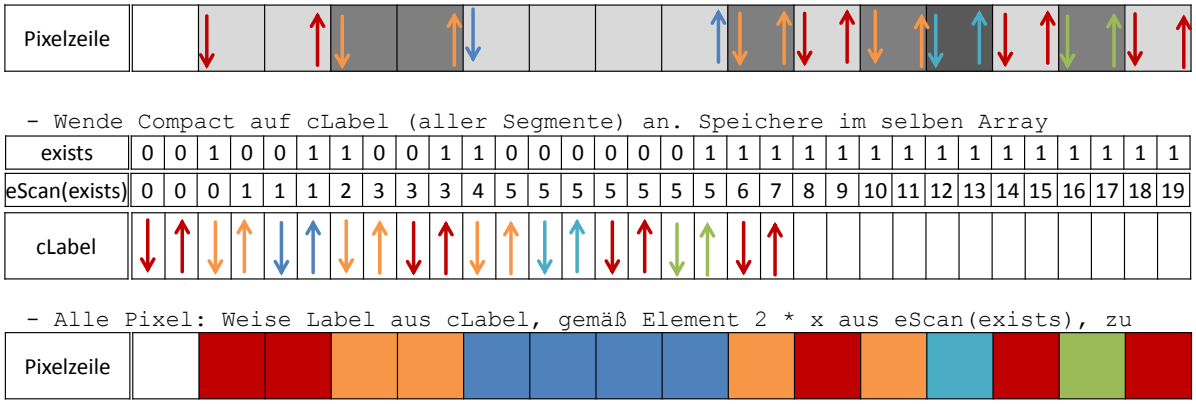


Abbildung 12.7.: Fortführung des Beispiels aus Abbildung 12.6: Labeln der Pixel, die nun eingerahmt zwischen gelabelten Kontursegmenten liegen. Hinweis: Sonderfälle bei Labelzuweisung beachten (siehe Text)

12.3.4. Asymptotisches Verhalten

Das asymptotische Verhalten ist identisch mit dem eines stabilen Sortieralgorithmus für *kleine Ganzzahlen*, da der Algorithmus durch diesen Schritt limitiert ist. Die asymptotischen Eigenschaften sind z. B. in Kapitel 7.3 (Seite 38) für Radix-Sort gegeben.

Kapitel 13.

Der gesamte Algorithmus

Die in den vorherigen Kapiteln vollständig beschriebenen Algorithmen für konturbasiertes Connected-Component-Labeling gliedern sich in drei Abschnitte:

- Der erste Schritt, die Extraktion der Kontursegmente, ist in Kapitel 9 beschrieben. Er wird immer ausgeführt, allerdings werden nicht für alle Varianten des Algorithmus alle der dort gegebenen Daten benötigt.
- Es folgt das Kontur-Labeling. Hier stehen drei Varianten zur Auswahl. Diese verwenden das datenunabhängige Parallelitätsschema (Kapitel 10.2), die Pointer-Jumps (Kapitel 10.3.1) oder die optimale und datenabhängige Technik (Kapitel 10.3.2).
- Im letzten Schritt erhalten die Pixel im Inneren der Konturen die Label der äußeren Konturen. Hier stehen zwei Varianten zur Wahl, nämlich die stack-basierte Technik (Kapitel 12.2) und diejenige, welche die Verschachtelung auflöst (Kapitel 12.3).

Aus den Varianten der einzelnen Teilschritte können verschiedene Gesamtalgorithmen formuliert werden. Sinnvoll ist vor allem ein theoretisch optimaler Algorithmus. Dieser verwendet die in Kapitel 10.3.2 beschriebene optimale Kontur-Labeling-Technik und die auf Sortieren basierende Fill-Technik aus Kapitel 12.3. Seine asymptotischen Eigenschaften sind limitiert durch die Abwandlung des optimalen List-Ranking-Algorithmus. Damit ist der Algorithmus selbst ebenfalls optimal.

Eine zweite Variante, die cost-optimal ist, erscheint in Hinblick auf die Implementierbarkeit für reale Problemgrößen und Devices ebenfalls sinnvoll. Sie verwendet das datenunabhängige Parallelitätsschema für das Kontur-Labeling und den stack-basierten Fill-Algorithmus. Diese Variante zeichnet sich durch einen deutlich geringeren (konstanten) Overhead als der obige Algorithmus aus.

Kapitel 14.

Vergleich mit anderen parallelen und konturbasierten Techniken

Im Prolog ist bereits aufgeführt, dass die Ergebnisse der vorliegenden Arbeit in Unkenntnis anderer paralleler auf Konturen basierender Connected-Component-Labeling-Algorithmen entstanden sind. Das sind die Arbeiten von Cypher et al. [CSS89] und Agrawal et al. [ANL87]. Mit diesen werden die eigenen, unabhängig gefundenen, Algorithmen nachfolgend etwas genauer verglichen.

Wie bei den eigenen Ansätzen gliedert sich der Ablauf bei beiden im Groben in folgende drei Schritte:

1. Extraktion der Kontursegmente und Repräsentation als zyklische gerichtete Linked-List je Connected-Component
2. Finden eines Labels je Kontur
3. Füllen der Bereiche im Inneren der Konturen mit dem Label der jeweiligen äußeren Kontur

Im Detail unterscheiden sich die Ansätze allerdings deutlich. Das wird in den nächsten Abschnitten besprochen.

Schritt 1: Extrahieren der Kontursegmente

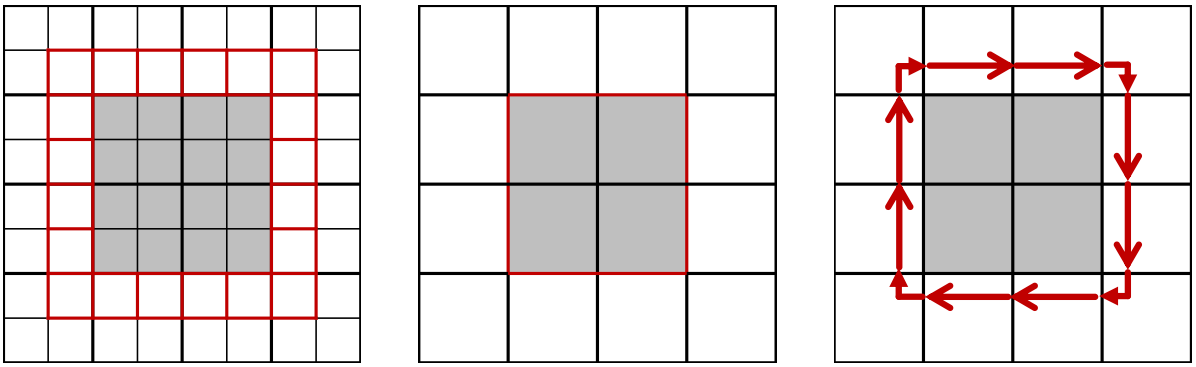
Cypher et al. und Agrawal et al. repräsentieren alle Konturen als zyklische gerichtete Linked-Lists bestehend aus Kontursegmenten. Damit entspricht diese Form derjenigen, welche auch von den Algorithmen der vorliegenden Arbeit verwendet wird. Im Detail gibt es allerdings Unterschiede.

So extrahieren Cypher et al. keine eigenen Kontursegmente, sondern verlinken die Kanten zwischen den Pixeln direkt. Auf diese Weise formuliert, benötigen sie nur $2 \cdot N + 2\sqrt{N}$ Kontursegmente (für $X = Y$). Zum Vergleich: Bei den eigenen Ansätzen sind es knapp doppelt so viele. Deshalb ist deren Ansatz aber auf Binärdaten beschränkt. Schließlich können sich im Falle von nicht Binärdaten zwei Connected-Components berühren. Dafür muss die Kannte zwischen denjenigen Pixeln, in denen sich beide berühren, doppelt existieren können. Im Rest der Beschreibung des Algorithmus wird darauf nicht weiter

eingegangen.

Agrawal et al. formulieren dagegen einen Algorithmus, welcher explizit auch Daten, die verschiedene Klassifikationen beinhalten, verarbeiten kann. Dazu verwenden sie eine 'Pixel fatten' Technik, bei der jeder Pixel in vier Subpixel aufgeteilt wird, welche in gewisser Weise hier die Kontursegmente darstellen. Folglich entspricht bei Agrawal et al. die maximale Segmentanzahl je Pixel derjenigen der eigenen Algorithmen.

Die erzeugten Liniensegmente von Cypher et al., Agrawal et al. und die Eigenen sind zum Vergleich anhand eines Beispiels in Abbildung 14.1 dargestellt.



(a) Agrawal et al. **(b)** Cypher et al. **(c)** Eigen

Abbildung 14.1.: Vergleich der Arten der Kontursegmente bzw. 'fattened Pixels' verschiedener Ansätze anhand einer Beispielszene. Erzeugte Kontursegmente bzw. 'fattened Pixels' sind rot dargestellt. Hinweis: Abbildungsrand ist nicht Bildrand.

Schritt 2: Finden eines Labels je Kontur

Das Kontur-Labeling funktioniert bei beiden mit Techniken, die der eigenen Kontur-Labeling Variante durch Pointer-Jump-Operations auf Linked-Lists (Abschnitt 10.3.1) sehr ähnlich sind.

Cypher et al. bezeichnen Varianten der Pointer-Jump-Technik als 'List Reduction' und Agrawal et al. verwenden die Bezeichnung 'Segmented Scan'.

Zwischenschritt: Unterscheidung innerer und äußerer Konturen

Cypher et al. wählen jeweils das (bereits gefundene) Segment, welches sich ganz oben in einer Kontur befindet. Dann vergleichen sie, ob der Pixel über dem Segment mit null klassifiziert ist. Ist das der Fall, handelt es sich um eine äußere Kontur. Agrawal et al. verfahren analog.

Diese Vorgehensweise unterscheidet sich somit von der eigenen Unterscheidung gemäß Drehsinn.

Zwischenschritt: Vereinigen äußerer und innerer Konturen

Cypher et al. und Agrawal et al. vereinigen jeweils die äußere Kontur mit allen zugehörigen inneren Konturen zu einer einzigen zyklischen Linked-List.

Cypher et al. wählen dafür von jeder Linked-List einen Repräsentanten (das Maximum) und lassen diesen mit der Pointer-Jumping-Technik das nächste Kontursegment zu ihrer rechten Seite suchen. Dieser gehört entweder zu einer inneren Kontur oder der äußeren Kontur, jeweils der selben Connected-Component. Dann erhalten die Repräsentanten als Nachfolger das jeweils gefundene Segment. Damit sind nun alle Konturen je einer Connected-Component durch einen binären Baum repräsentiert. Diese werden dann jeweils mit einer Euler-Tour-Technik in eine einzelne zyklische Linked-List überführt.

Agrawal et al. wählen ebenfalls einen Repräsentanten je Kontur (ganz rechts, davon den höchsten). Jeder davon sucht, wenn er keine äußere Kontur repräsentiert, ebenfalls die nächste Kontur in rechter Richtung. Die betreffenden Konturen werden dann vereinigt, indem der Repräsentant und das gefundene Segment die Verlinkungen austauschen. Die Vervierfachung der Pixel garantiert dabei, dass dieser Austausch konfliktfrei möglich ist. Somit muss die Euler-Tour-Technik, im Gegensatz zu Cypher et al., nicht angewandt werden.

Dieser Schritt hat in den eigenen Algorithmen keine direkte Entsprechung.

Zwischenschritt: Finden eines Labels je Connected-Component

Cypher et al. und Agrawal et al. wenden ein zweites Mal die Pointer-Jumping-Technik auf die nun vereinigten Konturen an, um ein Label zu finden. Dieses ist jetzt eindeutig für jede Connected-Component.

Der Schritt ist bei den eigenen Ansätzen Teil des Füllens. Je nach Variante erhalten bei den eigenen Techniken allerdings innere Konturen nicht unbedingt ein Label.

Schritt 3: Füllen

Cypher et al. und Agrawal et al. füllen zuletzt die Bereiche innerhalb der Konturen, indem sie wieder die Pointer-Jumping-Technik einsetzen.

14.1. Zusammenfassung der Unterschiede und Einschätzung hinsichtlich Implementierbarkeit

Die Ansätze von Cypher et al. und Agrawal et al. führen in ihrer ursprünglichen Formulierung $O(N \cdot \log_2 N)$ Operationen aus und sind damit nicht optimal. Sie können jedoch, analog zu dem eigenen Ansatz, durch Verwendung einer fortschrittlicheren Technik anstelle der Pointer-Jumps zu optimalen Algorithmen umformuliert werden. Eine

Einschränkung des Ansatzes von Cypher et al. gegenüber den eigenen und dem von Agrawal et al. ist die fehlende Unterstützung von nicht binären Daten. Zwar kann man den Algorithmus für jede Klassifikation einzeln durchlaufen lassen. Dann erhält er seine asymptotischen Eigenschaften jedoch nur, wenn man annimmt, dass es höchstens eine konstante Anzahl verschiedener Klassifikationen geben kann. Theoretisch möglich sind offensichtlich $O(N)$ verschiedene Klassifikationen in einem Bild.

Hinsichtlich der Funktionsweise kann zunächst festgehalten werden, dass die erzeugten Kontursegmente aller Ansätze deutlich voneinander abweichen. Allein deshalb würden sich (denkbare) Implementationen erheblich unterscheiden. In der Praxis könnte der Ansatz Cypher et al. aufgrund der geringeren Anzahl von Kontursegmenten schneller sein, sofern lediglich Binärdaten vorliegen. Im Ausblick dieser Arbeit (Kapitel 28, ab S. 337) wird eine reduzierte Variante des eigenen Kontursegmentschemas vorgeschlagen. Diese reduziert die Kontursegmentzahl auf $2 \cdot N$ und ist damit noch geringer als bei Cypher et al. Außerdem bleibt im Gegensatz zu Cypher et al. die Möglichkeit der Verarbeitung von nicht Binärdaten erhalten.

Der folgende Kontur-Labeling-Algorithmus ist bei allen fast identisch, sofern bei dem eigenen Ansatz die einfache Pointer-Jump-Variante gewählt wird. Für die eigene Variante mit datenunabhängigem Parallelitätsschema gibt es keine Entsprechungen. Wir werden im weiteren Verlauf dieser Arbeit sehen, dass diese Technik auf nicht hochgradig paralleler Hardware Vorteile bietet.

Nach erfolgtem Kontur-Labeling haben die eigenen Algorithmen fast keine Gemeinsamkeiten mit den Ansätzen von Cypher et al. und Agrawal et al. Die eigenen Algorithmen füllen nun direkt die einzelnen Zeilen (oder Spalten) des Bildes, und zwar für jede Zeile unabhängig. Durch dieses Aufteilen in unabhängige Teilprobleme wird z.B. die Verwendung des schnellen aber kleinen Shared-Memory einer GPU vereinfacht. Dagegen erzeugen Cypher et al. und Agrawal et al. neue, globale zyklische Linked-Lists und führen ein zweites Mal die Pointer-Jumps darauf aus. Wie wir sehen werden, ist bei allen eigenen Implementationen das Contour-Labeling immer der mit Abstand teuerste Schritt des Algorithmus. Es erscheint daher in der Praxis wenig hilfreich, diesen ein zweites Mal zu wiederholen. Dagegen ist der eigene stack-basierte Fill-Algorithmus, der jeden Pixel genau einmal betrachtet, in allen Experimenten nur für einen sehr geringen Anteil der Laufzeit verantwortlich. Es erscheint somit unwahrscheinlich, dass mit diesen Techniken von Cypher et al. und Agrawal et al. auf GPUs oder CPUs in der Praxis eine Verbesserung gegenüber den eigenen Varianten erreichbar ist.

Außerdem unterscheidet sich der eigene Test auf Zugehörigkeit zu einer äußeren Kontur. Cypher et al. und Agrawal et al. fragen dazu eine Pixelklassifikation ab. Diese ist insbesondere nicht Teil der Kontursegmentdaten. Einige Varianten des eigenen Algorithmus können den Test, basierend auf der (auch so schon berechneten) minimalen Kontursegmentadresse, durchführen. Das sind insbesondere Daten, welche zu den Kontursegmenten gehören. Wie sich zeigen wird, sind gerade optimierte GPU-Fassungen auf schnelle, aber sehr kleine Speicher angewiesen und deren Verbrauch wirkt oft limitierend. Es erscheint daher wenig sinnvoll, zusätzliche Daten dort abzulegen bzw. bei Bedarf aus einem großen, aber langsamen Speicher anzufordern.

Zusammenfassend kann geschätzt werden, dass höchstens das Kontursegmentschema von Cypher et al. gegenüber der Verwendung des eigenen Kontursegmentschemas zu einer Verbesserung des Laufzeitverhaltens auf GPUs oder CPUs führen könnte. Und auch das nur, wenn Binärdaten vorliegen. Außerdem gilt diese Schätzung nicht im Vergleich mit dem eigenen im Epilog vorgeschlagenen reduzierten Kontursegmentschema.

Teil III.

Paralleles Computing

Kapitel 15.

Überblick

Im dritten Teil der vorliegenden Arbeit werden Implementationen auf Basis der in Teil II vorgestellten Algorithmen mit bestehenden CCL-Implementationen aus der Literatur auf CPUs und GPUs experimentell verglichen. Die genauen Zielsetzungen dieser Untersuchung sind im Kapitel 4 ab Seite 28 angegeben. Es folgt ein kurzer Überblick hinsichtlich der Struktur des dritten Teils.

Das Kapitel 16 führt in den Themenbereich GPU-Computing mit OpenCL und Cuda ein. Dabei wird speziell die Programmierung mit OpenCL soweit eingeführt, wie es zum Verständnis der Codebeispiele in dieser Arbeit erforderlich ist.

Im Kapitel 17 wird beschrieben, wie eine Implementation, für aktuelle Nvidia GPUs einerseits und für CPUs andererseits, optimiert werden kann. Dabei wird aufgrund des Fokus dieser Arbeit auf GPUs für diese ein deutlich höherer Optimierungsgrad angestrebt. Außerdem werden hier die verwendeten CPUs und GPUs, sowie deren für diese Arbeit zentralen Eigenschaften aufgeführt.

Die Methodik, mit der die Laufzeitmessungen, sowohl der eigenen Implementationen als auch solcher aus der Literatur, durchgeführt werden, ist in Kapitel 18 beschrieben. Zusätzlich werden verschiedene Details des verwendeten Rechners, der Sprachversionen usw. aufgeführt.

Abgeschlossen wird die Erläuterung der für den dritten Teil der vorliegenden Arbeit benötigten Grundlagen mit der Besprechung der zu evaluierenden Datensätze. Diese erfolgt in Kapitel 19.

Die Kapitel 20, 21, 22 und 23 beschreiben verschiedene Implementationsstrategien für paralleles und konturbasiertes Connected-Component-Labeling. Die Zielsetzung ist dabei einerseits, in der Praxis möglichst schnelle Ansätze zu finden. Andererseits soll das Laufzeitverhalten der jeweiligen Implementation in Abhängigkeit der eingesetzten Hardware und der zu verarbeitenden Daten verstanden werden. Ein Vergleich mit bestehenden Ansätzen der Literatur findet in diesen Kapiteln noch nicht statt.

Ausgangspunkt sind jeweils Implementationen, welche nah an den in Teil II dieser Arbeit formulierten Algorithmen sind. Diese, und auch deren einzelne Sub-Algorithmen,

werden jeweils experimentell analysiert und die Ergebnisse interpretiert. Basierend auf den Beobachtungen werden sie entweder weiter optimiert oder der Ansatz verworfen und der jeweils Nächste dadurch motiviert. Wir werden dabei feststellen, dass gerade für GPUs ein Ansatz in der Praxis schnell ist, der sich deutlich von den ursprünglich formulierten Algorithmen unterscheidet. Dieser ist insbesondere noch nicht einmal work-optimal.

Zuletzt wird, für CPUs einerseits und für GPUs andererseits, jeweils der schnellste Ansatz identifiziert. Die weitere Untersuchung beschränkt sich allein auf diese beiden.

Anschließend werden in den Kapiteln 24, 25 und 26 die eigenen Implementationen mit solchen aus der Literatur experimentell verglichen.

Kapitel 16.

OpenCL und Cuda

In diesem Kapitel wird GPU-Computing mit OpenCL bzw. Cuda soweit eingeführt, wie es zum Verständnis der vorliegenden Arbeit erforderlich ist.

16.1. GPU-Computing Vorgeschichte

GPU-Computing (auch GPGPU, General-Purpose-GPU-Computing) meint die Verwendung von Grafikprozessoren (GPU) für allgemeine Berechnungen. Gemeint sind damit solche Berechnungen, welche nicht unmittelbar mit der Ausführung der Rendering-Pipeline einer Rastergrafik-API zusammenhängen, wofür GPUs ursprünglich entwickelt wurden. Beispielsweise fällt das Rendering mit Ray-Tracing auf einer GPU somit auch in die Kategorie GPU-Computing.

Indirekt gehen die GPU-Computing Anfänge somit auf die verbreiteten Rastergrafikschnittstellen OpenGL [WNDS99] (aktueller: [SSKLK13]) bzw. DirectX [Wik15b] zurück. Erstere wurde 1992 eingeführt und Letztere 1995. Beide APIs beinhalteten in frühen Versionen eine feste, aber konfigurierbare Grafik-Pipeline mit dem Fokus auf Echtzeit-Rendering.

Eine der ersten Grafikkarten, welche im Consumer-Markt auf große Resonanz stieß, war die 3dfx Voodoo Graphics [Wik15a]. Sie kann einige der hinteren Teile der Graphics-Pipeline ab dem Rasterizer in Hardware ausführen. Infolgedessen konnten zeitgenössische Spiele, wie etwa Quake [Wik15d] von id Software, deutlich im Vergleich mit der CPU-Fassung desselben Spiels beschleunigt werden.

Im Gegensatz zu OpenGL und DirectX, deren Fokus auf Echtzeitrendering liegen, existiert im High-Quality-Bereich schon deutlich länger das Konzept sogenannter Shader. Diese Bezeichnung geht auf Cooks Shade-Trees [Coo84] zurück und meint prozedurale Beschreibungen, z.B. von Oberflächeneigenschaften. Verwendet wurde das Konzept damals z.B. in der Renderman Shading-Language. Es bietet eine deutlich intuitivere Programmierung und ist außerdem weitaus mächtiger als die festen APIs. Mit steigender verfügbarer Rechenleistung kam daher auch im Echtzeitgrafikbereich der Wunsch auf, Shader einsetzen zu können.

Ein früher Ansatz für programmierbare Shader wurde von Peercy im Jahre 2000 vorgestellt [POAU00]. Dieser verwendet intern die feste Rendering-Pipeline von OpenGL, welche abstrakt als SIMD-Prozessor gesehen wird. Es kann in jedem Render-Durchlauf lediglich eine Operation auf alle Daten angewandt werden, und zwar in der Per-Frag-

ment-Operations Stufe. Der Ansatz kann auch für einige andere Berechnungen genutzt werden, sodass es sich hierbei um eines der ersten GPU-Computing-Frameworks handelt (auch wenn der Begriff damals noch nicht verwendet wurde).

Ein Jahr später ermöglichte die Nvidia GeForce 3 [Nvi15b] als erste Grafikkarte die Ausführung programmierbarer Shader auf einer GPU, wenn auch mit Einschränkungen [LKM01]. Dabei wird zwischen zwei Arten von Shadern unterschieden: Zum einen werden die Per-Vertex-Operations der festen OpenGL 1.x Pipeline durch ein Programm, den Vertex-Shader, ersetzt. Zum anderen ersetzt der Fragment-Shader die Fragment-Processing-Stufe.

Auf Basis dieser und vergleichbarer Grafikhardware wurden außerdem weitere Untersuchungen im Bereich GPU-Computing unternommen. Beispiele sind Ray-Tracing [PBMH02, CHH02] und lineare Algebra [KW03]

Für die GPU-Programmierung via OpenGL war es erforderlich, ein zu lösendes Problem als grafische Szene darzustellen, diese zu rendern und das Ergebnis aus dem Framebuffer auszulesen. Außerdem waren damalige Shader in vielerlei Hinsicht eingeschränkt. Beispielsweise war die maximale Instruktionszahl sehr gering, sodass umfangreichere Berechnungen auf mehrere Shader, und damit Render-Durchgänge aufgeteilt werden mussten. Daraus resultierte ein Programmierstil, der praktisch nur von Computergrafikexperten angewandt werden konnte. Spätere Arbeiten versuchten daher, die Berechnungen mit programmierbaren Shadern zu vereinfachen. So entwickelte McCool [MQP02, MDTP+04] die auf C++ basierende Shader-Metaprogrammiersprache Sh [MT04]. Die BrookGPU [BFH+04] API führt einen ähnlichen Weg, allerdings mit Fokus auf GPU-Computing, fort. Einige dafür wichtige Operationen, welche in Sh nicht verfügbar sind, wurden, wie beispielsweise die Scatter-Operation, ergänzt.

Auch wenn solche Ansätze die Arbeitsweise für die Programmierung abstrahieren können, sind manche Operationen nur sehr ineffizient durch Vertex- und Fragment-Shader umsetzbar. Verschiedene Artikel in GPU Gems 2 [PF05] geben einen anschaulichen Überblick über die Vorgehensweise bei der Implementation einiger für allgemeines Computing notwendiger Operationen via OpenGL und DirectX für GPUs. Beispielsweise ist die Implementation von Scatter, einer Schreiboperation an eine zu berechnende Position, kaum effizient möglich, weil sie durch Grafik-APIs nicht direkt unterstützt wurde. Ein Ansatz ist das Anbinden der zu schreibenden Information an Vertices, deren Position so zu verändern ist, dass sie am Ende der Pipeline in einen Pixel geschrieben wird, welcher der gewünschten Speicherposition entspricht [Har05]. In vielen Fällen kann aber auch der Algorithmus so umformuliert werden, dass aus Scatter-Operationen Gather-Operationen werden [Buc05].

Einen Meilenstein im Bereich GPU-Computing stellt daher die Einführung von Nvidias API Cuda [Nvi15a] im Jahr 2007 dar, welche eine direkte Programmierung der GPU ohne Grafik-API Overhead ermöglichte. Sie erschien etwa zeitgleich mit der Nvidia GeForce 8 GPU [Nvi15d], welche erstmals keine dedizierten Vertex- und Fragment- (bzw. Pixel-) Prozessoren, sondern universellere Prozessoren, damals als Unified-Shader bezeichnet, einsetzte. Cuda ermöglichte einerseits einen deutlich höheren Optimierungsgrad als bisherige Ansätze (u.a. wurden obengenannte Techniken unnötig) und machte

GPU-Computing gerade auch für nicht Grafikprogrammierer intuitiver.

Viele in Cuda formulierte Algorithmen, etwa wissenschaftliche Berechnungen, lieferten Speedups von oftmals über einer Größenordnung im Vergleich mit CPUs. Beispielsweise konnten Shimokawabe et al. im Falle eines Wettervorhersagemodells einen Speedup um Faktor 80 erreichen [SAM+10]. Damit ist GPU-Computing zu einer echten Alternative für allgemeine Berechnungen geworden.

Trotz des Erfolgs der Cuda API, welche ausschließlich GPUs des Herstellers Nvidia unterstützte, wurden Alternativen weiter vorangetrieben. Nvidias Konkurent Ati (heute AMD) verfolgte mit der (inzwischen eingestellten) API Close to Metal [Wik15e] einen ähnlichen Ansatz. Ein anderer Ansatz, BSGP (Bulk-Synchronous GPU Programming) [HZG08], verfolgte das Ziel, die GPU-Programmierung weiter zu vereinfachen. Intern basiert er auf Cuda.

Ende 2008 wurde die API OpenCL erstmals vorgestellt, welche eine herstellerunabhängige Alternative zu Cuda darstellt. OpenCL wird inzwischen, wie unter anderem OpenGL, von der Khronos Group betreut. Diese API wird im nächsten Abschnitt detaillierter besprochen.

Motiviert durch diese Entwicklungen wurden im Bereich Echtzeitgrafik einige Versuche unternommen, die Flexibilität der GPU-Computing-APIs auch für Rastergrafik zu verwenden. Schließlich beinhalten die Rastergrafik-APIs OpenGL und DirectX unverändert eine feste Graphics-Processing-Pipeline, von der lediglich einige Anschnitte durch programmierbare Shader ersetzt sind.

Einen Vorschlag für einen radikal anderen Ansatz stellt das GRAMPS Programmiermodell dar [SFB+09], mit welchem sich beliebige Grafik-Pipelines (einschließlich Grafen) formulieren und z.B. auf GPU-Programme und Abschnitte für feste GPU-Hardware (z.B. Rasterizer) abbilden lassen. Weitere, weniger radikale, Ansätze haben etwa eine Graphics-Pipeline im traditionellen Stil mit Cuda implementiert, um z.B. reihenfolgeunabhängige Transparenz zu ermöglichen [HLLW11]. Bei Verwendung von OpenGL und DirectX müssen transparente Flächen vor dem Rendering gemäß Tiefe vorsortiert und in unabhängigen Render-Passes für jede Ebene gerendert werden, wenn das gleiche Ergebnis erreicht werden soll.

Allerdings konnten sich solche Ansätze bislang in der Praxis nicht durchsetzen. Unabhängig davon verfügen neuere Versionen sowohl von OpenGL als auch von DirectX mittlerweile über sogenannte Compute-Shader, welche mittels GPU-Computing-Techniken z.B. für 'Post Effects' eingesetzt werden. Dabei werden bereits gerenderte Bilder mit an Image-Processing angelehnten Techniken, z.B. mit Filtern, nachbearbeitet. Einige Beispiele dafür sind in [LHA+09] beschrieben.

16.2. OpenCL

Die Open Computing Language (OpenCL) bietet eine API zur einheitlichen Programmierung verschiedenartiger paralleler Hardware. Dazu gehören GPUs, CPUs, APUs, FPGAs und andere Hardware.

OpenCL wurde durch Apple initiiert und wird inzwischen durch die nicht profitorientierte Khronos Group [khr15] betreut. Die erste öffentliche Vorstellung erfolgte im Dezember 2008 im Rahmen der Siggraph Asia durch Nvidia und AMD.

Die API ist Plattform-, Sprachen-, Betriebssystem- und Herstellerunabhängig. Es existieren Implementationen verschiedener Hersteller, wie Nvidia, AMD, Intel, ARM, Imagination Technologies, Apple, Samsung Electronics, IBM Corporation und QUALCOMM [khr15].

Dieses Kapitel basiert, sofern nicht anders gekennzeichnet, inhaltlich auf [GHK+13, MGM+11, KH13].

16.2.1. Architektur

Ein OpenCL-System besteht aus genau einem Host für die Koordination und einem oder mehreren Devices, welche die Berechnungen ausführen. Letztere führen die Kernel aus, welche in der prozeduralen Sprache OpenCL C formuliert werden, die Teil der OpenCL Spezifikation ist. Diese basiert auf C99 und ist um einige Konzepte für datenparalleles Rechnen und die zu OpenCL gehörige Speicherhierarchie erweitert. Es gibt auch Einschränkungen, so wird z.B. Rekursion nicht unterstützt. Die Kernel werden aus Gründen der Portierbarkeit erst zur Programmlaufzeit durch den Host via OpenCL API calls übersetzt.

Vor dem Start eines in OpenCL C formulierten Kernels auf einem Device muss die Problemgröße für dessen Ausführung angegeben werden. Diese wird als `Global-Work-Size` bezeichnet. Anschaulich kann man sich den Kernel als eine Funktion vorstellen, von der nun `Global-Work-Size` viele Instanzen erzeugt werden, von denen jede, potentiell parallel, einen Teil des Problems bearbeitet. Diese werden als `Work-Items` bezeichnet. Abhängig von der maximalen Anzahl der durch das Device ausführbaren Hardware-Threads müssen in der Praxis nicht selten einige dieser Work-Items sequentiell abgearbeitet werden. Das organisiert aber die OpenCL-Implementation.

Die Work-Items sind gruppiert zu Work-Groups, deren Größe vor Ausführung eines Kernels ebenfalls festgelegt werden muss. Während der Ausführung eines Kernels kann nur jeweils innerhalb der einzelnen Work-Groups synchronisiert werden.

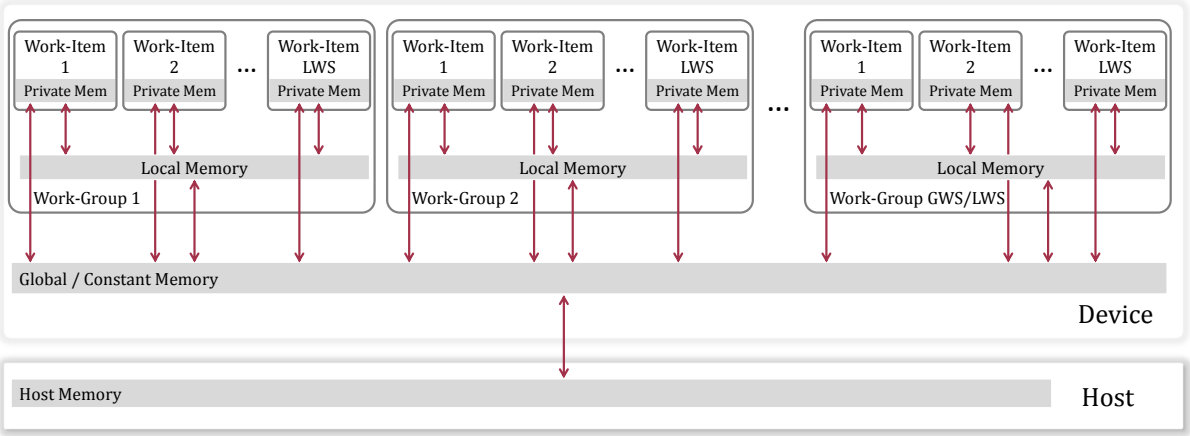


Abbildung 16.1.: Vereinfachte Darstellung des OpenCL-Speichermodells für ein einziges Device. Pfeile geben mögliche Lese- und Schreibzugriffe an. Aus dem Constant-Memory kann innerhalb des Device nur gelesen werden. Abkürzungen: Global-Work-Size (GWS), Local-Work-Size (LWS)

OpenCL definiert ein eigenes Speichermodell, welches für den Spezialfall eines einzelnen Device und eines davon verschiedenen Hosts in Abbildung 16.1 dargestellt ist. Der Host verfügt ausschließlich über das Host-Memory, welches er selbst allokieren kann. Ein Device verfügt dagegen über eine eigene Speicherhierarchie. Dessen Bestandteile sind:

Global-Memory: Ein Speicher, auf den alle Work-Items lesenden und schreibenden Zugriff haben. Die Allokation erfolgt ausschließlich über den Host, welcher auch darauf zugreifen kann.

Constant-Memory: Entspricht dem Global-Memory, die Work-Items haben aber nur lesenden Zugriff.

Local-Memory: Von diesem Speicher existieren unabhängige Instanzen für jede Work-Group, auf welche jeweils nur die Work-Items der jeweiligen Work-Group zugreifen können. Der Speicher kann dynamisch über den Host oder statisch im Kernelcode allokiert werden. Der Host hat keinen Zugriff auf den Speicher.

Private-Memory: Eine gewisse Menge Private-Memory ist jedem Work-Item zugeordnet und nur dieses hat Zugriff darauf. Die Allokation erfolgt ausschließlich statisch im Code des Kernels. Der Host kann es weder allokieren, noch hat er Zugriff darauf.

Es sei noch angemerkt, dass nicht zwingend alle Speicher physisch verschieden sein müssen.

16.2.2. OpenCL C Syntax

Schauen wir uns nun die OpenCL C Syntax an, soweit wir sie zum Verständnis dieser Arbeit benötigen. Ein minimaler Kernel, welcher $a + b = c$ auf Vektoren a, b und c berechnet, ist in Listing 16.1 gegeben. Jedes Work-Item berechnet darin eine Komponente des Vektors c, folglich muss die Global-Work-Size der Komponentenanzahl N der Vektoren entsprechen. Je nach Art des Problems kann OpenCL für ein- zwei- oder dreidimensionale Indizierung der Work-Items konfiguriert werden. In diesem Fall ist eine eindimensionale Indizierung naheliegend. Wir führen an dieser Stelle für den Rest dieser Arbeit eine vereinfachte Schreibweise dafür ein: **GWS(N, 1, 1)**. Ein so konfigurierter Kernel erzeugt $N \cdot 1 \cdot 1 = N$ Work-Items, welche eindimensional (und zwar in der nullten Dimension) indiziert sind. Analog definieren wir die Einstellung einer Local-Work-Size vereinfacht als: **LWS(C, 1, 1)**. In diesem Beispiel kann C irgendeine Konstante sein, welche einen ganzzahligen Teiler von N darstellt. Typischerweise gibt die OpenCL Implementation einen maximalen Wert für C vor. In Listing 16.1 wird die Work-Group-Size nicht explizit verwendet.

```
// Computes c = a + b component-by-component
kernel void vecAdd(
  global int* a,
  global int* b,
  global int* c)
{
    // Use built-in function to get work-item index
    int i = get_global_id(0); // 0, 1, 2, ..., N - 1
    c[i] = a[i] + b[i];
}
```

Listing 16.1: OpenCL C Code eines Kernels für Vektoraddition mit GWS(N, 1, 1)

Betrachten wir nun die Syntax in Listing 16.1 im Detail. Das Keyword `kernel` weist die Funktion mit Namen `vecAdd` als OpenCL C Kernel aus. Eine alternative Schreibweise ist `__kernel`, wir verwenden aber immer die Kurzform. Erlaubter Rückgabetyp ist ausschließlich `void`. Als Parameter werden Pointer auf das Global-Memory, erkennbar am Keyword `global`, übergeben. Dabei handelt es sich um die Vektoren a, b und c, wobei a und b vor Ausführung des Kernels durch den Host mit Werten initialisiert sein müssen. Im Körper des Kernels wird zunächst der globale Index des Work-Items mithilfe der built-in OpenCL C Funktion `get_global_id(uint D)` ausgelesen. Dabei ist D der Index der abzufragenden Dimension, hier aufgrund der eindimensionalen Indizierung demnach 0. Indem Operationen von dem Work-Item Index abhängen, wird sichergestellt, dass verschiedene Work-Items unterschiedliche Arbeit verrichten. In diesem Beispiel liefert `get_global_id(0)` die Werte `0` bis `N-1`, der Index kann somit direkt verwendet werden, um auf die Komponenten von a, b und c zuzugreifen. Die eigentliche Berechnung ist dann offensichtlich.

Fahren wir fort mit einem Beispiel, welches die Verwendung des Local-Memory und einer zweidimensionalen Indizierung demonstriert. Dazu ist in Listing 16.2 ein Kernel gegeben, welcher eine Matrix transponiert. Hierbei wird die Matrix in quadratische Bereiche, genannt Tiles, unterteilt. Diese werden über das Local-Memory transponiert und die Inhalte des Tiles anschließend an transponierte Tile-Positionen der Ergebnismatrix geschrieben. Die Vorgehensweise ist in Abbildung 16.2 veranschaulicht. Alternativ könnten die Werte auch direkt an die transponierte Position der Ergebnismatrix geschrieben werden. Die tile-basierte Transponierung im Local-Memory bewirkt jedoch auf einer GPU deutlich günstigere Zugriffsmuster auf das Global-Memory. Das wird später vertieft, an dieser Stelle liegt der Fokus allein auf der Syntax.

Es werden weitere OpenCL C built-in Funktionen benötigt, um etwa die Zugehörigkeit zu einer Work-Group abzufragen. Außerdem erfolgen Zugriffe auf das Local-Memory (und auch das Global-Memory) asynchron. Somit wird eine Möglichkeit zum Synchronisieren, wenn auch nur innerhalb der einzelnen Work-Groups, benötigt. Diese OpenCL C Funktionen, zusammen mit möglichen Rückgaben bei Konfiguration des Kernels mit GWS(X, Y, 1) und LWS(32, 32, 1), sind:

`get_global_size(uint D):` Liefert die Anzahl der globalen Indizes in Dimension D. Hier: X für D = 0 und Y für D = 1.

`get_global_id(uint D):` Liefert den globalen Index des jeweiligen Work-Items in Dimension D. Mögliche Werte sind hier: $\{0, 1, \ldots, X-1\}$ für D = 0 und $(0, 1, \ldots, Y-1)$ für D = 1.

`get_local_id(uint D):` Liefert den lokalen Index des jeweiligen Work-Items innerhalb seiner Work-Group in Dimension D. Mögliche Werte sind hier: $\{0, 1, \ldots, 31\}$ sowohl für D = 0 und D = 1.

`get_group_id(uint D):` Liefert den Index der zum jeweiligen Work-Item gehörigen Work-Group in Dimension D. Mögliche Werte sind hier: $\{0, 1, \ldots, X/32-1\}$ für D = 0 und $\{0, 1, \ldots, Y/32-1\}$ für D = 1.

`barrier(CLK_LOCAL_MEM_FENCE):` Die Funktion `barrier` mit der übergebenen OpenCL C Konstante `CLK_LOCAL_MEM_FENCE` hindert alle Work-Items einer Work-Group an der Ausführung von Befehlen danach, bis zwei Bedingungen erfüllt sind:

1. Alle Work-Items der jeweiligen Work-Group müssen die `barrier` Funktion ausgeführt haben.
2. Alle vorherigen Zugriffe auf das Local-Memory der Work-Items der jeweiligen Work-Group sind abgeschlossen.

Danach ist das Local-Memory konsistent für alle Work-Items je einer Work-Group.

Außerdem neu ist das Attribut `local`, welches eine Datenstruktur dem Local-Memory zuordnet. Der dafür notwendige Speicher wird in Listing 16.1 für jede Work-Group statisch allokiert.

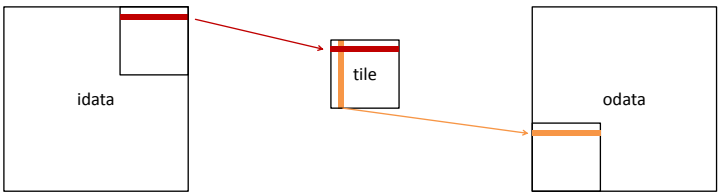


Abbildung 16.2.: Veranschaulichung des Kernels aus Listing 16.2 zur Matrixtransponierung

```
#define TILE_DIM 32 // Width (and height) of a tile

kernel
// Optional: Ensure LWS(TILE_DIM, TILE_DIM, 1).
__attribute__((reqd_work_group_size(TILE_DIM, TILE_DIM, 1)))
void mat_transpose(
// In: Matrix in row-mayor-order. Must be previously initialized
  global int* idata,
// Out: Matrix in column-mayor-order
  global int* odata)
{
  // Static allocation of a tile in local memory
  local int tile[TILE_DIM * TILE_DIM];

  // Get Matrix width (= number of work-items in dimension 0)
  int X = get_global_size(0);

  // Global (matrix) coordinates, using row-major-order scheme
  int x = get_global_id(0);
  int y = get_global_id(1);

  // Local (tile) coordinates, using row-major-order scheme
  int xL = get_local_id(0);
  int yL = get_local_id(1);

  // Read square region of the matrix from global memory and store it
  // in local memory, using row-major-order scheme
  tile[yL * TILE_DIM + xL] = idata[y * X + x];

  // Ensure, all write operations of work-items of a work-group are
  // completed
  barrier(CLK_LOCAL_MEM_FENCE);

  // Compute new global coordinates, where the tile-position is
  // transposed, but not the position within it ...
  x = get_group_id(1) * TILE_DIM + xL;
  y = get_group_id(0) * TILE_DIM + yL;

  // ... and write to this position a value from a transposed
  // memory location within the tile
  odata[y * X + x] = tile[xL * TILE_DIM + yL];
}
```

Listing 16.2: OpenCL C Code des Kernels `mat_transpose` für Matrixtransponierung. Die Matrix liege in row-major-order im Global-Memory und bestehe aus X Spalten und Y Zeilen. Konfiguration des Kernels: GWS(X, Y, 1) und LWS(TILE_DIM, TILE_DIM, 1). Dabei müssen X und Y ganzzahlige Vielfache von TILE_DIM, der Breite bzw. Höhe der Tiles, sein.

16.2.3. OpenCL API Abstraktion

Die OpenCL API selbst ist recht umfangreich und die Kenntnis der genauen Befehle zum Verständnis dieser Arbeit unbedeutend, sodass wir sie durch stark vereinfachten Pseudocode ersetzen, welcher nicht den vollen Funktionsumfang abbildet. Der OpenCL C Code bleibt davon unberührt.

Das Beispiel 23 führt diesen Pseudocode anhand der Ausführung des Kernels zur Matrixmultiplikation (siehe Listing 16.2) ein. In vielen Fällen gliedert sich der Ablauf von Berechnungen mit OpenCL in fünf Schritte, welche in dem Beispiel markiert sind.

Algorithm 23: Pseudocode für einen typischen Ablauf bei Ausführung eines OpenCL Kernels am Beispiel der Matrixtransponierung

```
// Matrix width and height and width of each tile
int X ← 1024;
int Y ← 512;
int tileDim ← 32;
/* I. Initialize OpenCL environment */
...
/* II. Initialize and bind OpenCL Kernel and Buffer objects */

// Compiled Kernel, based on OpenCL C code in listing 16.2,
// must be created
Kernel mat_transpose ← ... ;
// In-matrix: Memory must be allocated and values initialized
Buffer matrix ← ... ;
// Out-matrix: Memory must be allocated
Buffer matrix_T ← ... ;
// Bind buffer-objects to kernel arguments, given their arg-index
mat_transpose.setArg (0, matrix);
mat_transpose.setArg (1, matrix_T);

/* III. Execute computations */

// Enqueue kernel for execution. Set GWS and (optional) LWS as
further arguments
enq (mat_transpose, GWS (X, Y, 1), LWS (tileDim, tileDim, 1));

/* IV. Make sure computation is finished and do something with */
/* result */
/* V. Delete / release OpenCL objects */
```

Als erstes muss immer die OpenCL Umgebung initialisiert werden. Da OpenCL auf sehr heterogenen Systemen ausgeführt werden soll, welche aus beliebigen Kombinationen von verschiedenen Devices unterschiedlicher Hersteller bestehen können, bietet OpenCL

eine recht umfangreiche Objekthierarchie, um diese abbilden zu können. In dieser Arbeit besteht das System jedoch immer aus dem Host und genau einem Device, bei dem es sich entweder um eine GPU oder eine CPU (dann auch gleichzeitig Host) handelt. Somit kann alles damit zusammenhängende abstrahiert werden.

Der zweite Schritt besteht dann aus der Erzeugung von Kernel-Objekten aus den mithilfe von OpenCL API calls übersetzten und in OpenCL C formulierten Kerneln. Außerdem müssen, ebenfalls mit OpenCL API calls, OpenCL Buffer-Objekte erstellt werden, welche Daten im Global-Memory repräsentieren. Dabei ist festzulegen, ob die Daten im Host oder im Device vorgehalten werden sollen. In dieser Arbeit verbleiben die Daten im Falle eines CPU-Device hostseitig und bei einer GPUs wird deren eigener Speicher allokiert. Auch das wird nachfolgend abstrahiert. Ferner müssen die Datenstrukturen vor Ausführung der Kernel an diese angebunden werden.

Den dritten Schritt stellt die Ausführung des Kernels dar. Das geschieht in OpenCL asynchron durch einhängen in eine Queue. Für verschiedene Devices müssen auf jeden Fall verschiedene Queues existieren und auf jedem einzelnen Device kann es auch mehrere geben. Jede einzelne Queue kann für out-of-order und in-order Ausführung konfiguriert werden. Nur letzteres stellt sicher, dass eine eingehängte Aufgabe erst dann begonnen wird, wenn die vorherige in dieser Queue abgeschlossen ist. Es existieren in OpenCL einige Konzepte um bei möglichst asynchroner Arbeitsweise Abhängigkeiten für eine gültige Ausführung zu modellieren. In dieser Arbeit gibt es jedoch immer genau eine Queue und die ist für in-order Ausführung konfiguriert. Somit können diese Konzepte abstrahiert werden. Im Pseudocode wird die Ausführung eines Kernels durch das Keyword `enq` angestoßen.

Denkbarer vierter Schritt ist die Weiterverwendung der Ergebnisse. Beispielsweise können weiter Kernel gestartet werden, welche diese Ergebnisse benötigen. Aufgrund der verwendeten in-order Queue können diese einfach eingehängt werden und nichts weiter ist zu beachten. Um die Abarbeitung aller Aufgaben zu erzwingen, muss jedoch spätestens am Ende durch den Host synchronisiert werden. Das kann beispielsweise durch den OpenCL API Befehl `clFinish` geschehen, der als Argument die Queue erhält.

Zuletzt können dann alle OpenCL Objekte, aufgrund der Abhängigkeiten idealerweise in umgekehrter Erzeugungsreihenfolge, mit API calls gelöscht und die Ressourcen wieder freigegeben werden.

16.2.4. Vergleich mit PRAM-Model

Wir vergleichen in diesem Abschnitt die OpenCL Programmierung mit dem theoretischen PRAM-Modell, um eine Idee zu bekommen, wie die in Teil II dieser Arbeit formulierten Algorithmen in die Praxis umgesetzt werden können. Der Vergleich beschränkt sich auf die bisher eingeführten Einschränkungen, nämlich die Verwendung eines einzigen Device und genau einer in-order Queue. Damit existiert ein Shared-Memory (PRAM-Begriff), nämlich das Global-Memory (OpenCL Begriff) des Device. Auch kann nur ein Kernel zur Zeit ausgeführt werden. Andernfalls wäre das PRAM-Modell wenig zur Modellierung geeignet.

Die OpenCL Architektur basiert auf dem SPMD (Single Program Multiple Data)

Modell [AF98]. Dabei verarbeitet ein Programm (der Kernel) verschiedene Daten parallel, aber abhängig von der maximalen Parallelität des Device üblicherweise nicht alle. Dementsprechend muss auch sequentiell gerechnet werden. Das organisiert OpenCL selbst. Insbesondere gibt es im Gegensatz zu dem PRAM-Modell keinen globalen Taktgeber. Folglich muss explizit synchronisiert werden, wenn das erforderlich ist. Wir haben bereits die `barrier`-Funktion, mit der innerhalb einer Work-Group synchronisiert werden kann, besprochen. Eine globale Synchronisation innerhalb eines Kernels, d.h. über Work-Group Grenzen hinweg, ist in OpenCL nicht vorgesehen. Somit muss, wenn ein globaler Synchronisationspunkt erforderlich ist, ein Kernel beendet werden. Anschließend kann ein neuer Kernel (in der in-order Queue) gestartet werden. Es wird an dieser Stelle klar, dass die (minimale) Aufteilung eines Algorithmus in verschiedene OpenCL Kernel durch erforderliche globale Synchronisationspunkte vorgegeben ist.

Unabhängig davon kann innerhalb eines Kernels die Parallelität nicht geändert werden. Dementsprechend ist die weitere Aufteilung in Kernel oft durch Sub-Algorithmen verschiedener Parallelität motiviert.

Speicherzugriffe sind für jedes Work-Item auf jede Speicheradresse des Global-Memory möglich. Auch gleichzeitiges Lesen einer Adresse ist unproblematisch. Gleiches gilt für gleichzeitiges Schreiben, wenn alle Work-Items den gleichen Wert schreiben. Auch das Aggregieren verschiedener Werte (etwa die Summe darüber) ist möglich. Dazu müssen aber Atomic-Functions verwendet werden. In dieser Arbeit ist letzteres nicht erforderlich, aber der GPU-Ansatz [OS11], mit dem wir die Ergebnisse der vorliegenden Arbeit vergleichen, verwendet diese.

Insgesamt ist eine Implementation der in Teil II dieser Arbeit erarbeiteten Algorithmen mit OpenCL möglich, sofern die zusätzlich erforderliche Synchronisation bedacht wird.

16.3. Cuda

Im Rahmen der vorliegenden Arbeit wird zusätzlich zu den OpenCL Implementationen auch eine Implementation mit Cuda erstellt. Diese Portierung ist motiviert durch eine Reihe unschöner Beobachtungen mit Nvidias OpenCL Implementationen, welche im späteren Verlauf dieser Arbeit deutlich werden. Außerdem verwendet der direkte Vergleichsansatz [OS11] Cuda. Deswegen wird die API an dieser Stelle kurz angesprochen.

Der API Cuda, welche für GPUs des Herstellers Nvidia ausgelegt ist, fehlen alle OpenCL Funktionen zur Modellierung von Ausführungsumgebungen mit heterogenen Devices. Umgekehrt unterstützt Cuda aber mindestens alle Funktionen, welche bei Verwendung von OpenCL mit einer Nvidia GPU bereitgestellt werden. Tatsächlich basiert die OpenCL Implementation aktueller Nvidia Treiber intern auf Cuda.

Jede in der vorliegenden Arbeit verwendete OpenCL Funktion und auch z.B. die OpenCL Speicherhierarchie haben direkte Entsprechungen in Cuda. Eine Portierung der in dieser Arbeit vorgestellten OpenCL C Kernel nach Cuda C ist (fast) durch simple Textersetzung möglich. Somit verzichten wir auf die Einführung der Cuda Syntax und geben alle Beispiele ausschließlich in OpenCL C formuliert an.

Cuda bietet eine Reihe von Funktionen, welche mit OpenCL auf Nvidia Karten nicht genutzt werden können. Einige Beispiele dafür folgen in Kapitel 17 (Optimierung). Im Rahmen dieser Arbeit wird jedoch auch von der Cuda Portierung keinerlei Cuda exklusive Funktionalität verwendet.

Die in der vorliegenden Arbeit verwendeten OpenCL Treiber von Nvidia unterstützen lediglich Version 1.1, welche 2010 [KG15a] eingeführt wurde. Die 2011 [KG15b] bzw. 2013 [KG15c] eingeführten Versionen 1.2 und 2.0 werden nicht unterstützt.

Allerdings kündigt ein im April 2015 veröffentlichter Nvidia Treiber zumindest OpenCL 1.2 Unterstützung an, wenn auch nicht für alle in dieser Arbeit verwendeten GPUs [Nvi15c]. Das ist in dieser Arbeit jedoch nicht mehr berücksichtigt worden.

Kapitel 17.

Optimierung

Optimieren meint in dieser Arbeit das Implementieren in einer Weise, welche bei Ausführung auf einer bestimmten Zielarchitektur einen besonders hohen Throughput ermöglicht. Das geht üblicherweise mit der Anpassung an Eigenschaften (vorhandene Ressourcen, Parallelität, günstige Speicherzugriffsmuster, etc.) der Hardware einher. Im Falle einer für Device A hoch optimierter Implementation ist es nicht ungewöhnlich, dass diese auf Device B einen geringeren Throughput ermöglicht als eine unoptimierte Variante, oder sogar unausführbar ist. Optimieren steht demnach oft ein Stück weit im Wiederspruch mit heterogenen Zielplattformen. In gewisser Weise ist das auch eine Herausforderung gerade für OpenCL. Schließlich ist die datenparallele Programmierung gerade durch hohe Leistung motiviert. So merkt etwa [SFSV13] an, dass OpenCL Code zwar ohne weitere Anpassungen auf jeder Plattform ausführbar ist, welche die Spezifikation erfüllt, aber das Optimieren, gerade für CPUs, nach wie vor individuell geschehen muss.

Optimieren kann alternativ hinsichtlich der Daten geschehen, z.B. wenn viele Leereinträge zu erwarten sind. Das ist aber kein erklärtes Ziel in dieser Arbeit.

Ein primäres Ziel dieser Arbeit ist, die entstandenen Implementationen experimentell mit den Resultaten bisheriger GPU-basierter Fassungen zu vergleichen. Dabei handelt es sich um optimierte Cuda Implementationen. Für einen fairen Vergleich müssen daher auch die eigenen Implementationen optimiert werden und zwar für Nvidia GPUs. Schließlich können die bisherigen Ansätze aufgrund der Verwendung von Cuda, nur auf solchen ausgeführt werden.

Außerdem wird auch eine CPU-basierte Fassung mit OpenCL erstellt und evaluiert. Dabei handelt es sich lediglich um eine Machbarkeitsstudie. Somit ist der erforderliche Optimierungsgrad wesentlich geringer und wir schauen uns nur im groben die Unterschiede zur GPU-Programmierung an.

Ein Hinweis: Trotz des ähnlichen Wortes hat optimieren nichts mit optimal, der theoretischen Eigenschaft eines Algorithmus, zu tun. Bei Implementationen ist immer optimieren (meist als Verb, sonst etwa 'optimierte Fassung') und bei Algorithmen ist immer optimal gemeint. Vertauscht ergeben beide Begriffe keinen Sinn.

17.1. Optimieren für Nvidia GPUs

Dieses Kapitel bespricht die Architektur und Optimierungsstrategien für aktuelle Nvidia GPUs, soweit diese zum Verständnis der Arbeit erforderlich sind. Aktuell meint hier die Architekturen Fermi [Nvi15l] (Markteinführung März 2010) Kepler [Nvi15g, Nvi15k] (Markteinführung Mai 2012) und Maxwell [Nvi15i] (Markteinführung September 2014). Dabei liegt der Fokus auf Kepler, der zur Entwicklungszeit dieser Arbeit aktuellen Architektur.

Mit der Optimierung für Nvidias GPUs einhergehend muss auch deren Vokabular eingeführt werden, welches teilweise mit den Cuda Begriffen identisch ist. Das wiederum birgt Verwechslungsgefahr mit den OpenCL Begriffen. Wir bemühen uns, OpenCL Vokabular für die Programmierung (alle Codebeispiele verwenden OpenCL C) und Cuda Vokabular für die Bezeichnung der Hardware Ressourcen auf einer GPU zu verwenden.

Dieses Kapitel basiert, sofern nicht anders gekennzeichnet, inhaltlich auf [KH13, Nvi15a, Coo13, SK10, Far12].

17.1.1. GPU-Architektur

Eine Nvidia GPU besteht aus unabhängigen **Streaming-Multiprozessoren (SM)**, in manchen Quellen auch als Shader-Cluster bezeichnet. Innerhalb einer GPU-Generation skalieren Grafikkarten von low-end bis high-end primär durch deren Anzahl. Beispielsweise bestehen GPUs der Kepler-Architektur aus einem SM (Geforce GT 720 / GK208) bis hin zu $2 \cdot 15$ (Geforce GTX Titan Z, $2 \cdot$ GK110).

Jeder SM kann die Work-Items einer oder mehrerer Work-Groups, bis hin zu einer hardwareabhängigen Anzahl, gleichzeitig verarbeiten. Eine Work-Group kann niemals auf verschiedene Streaming-Multiprozessoren aufgeteilt werden. Weiterhin verfügt jeder SM über eine bestimmte Menge **Shared-Memory**, das für das Local-Memory aus OpenCL verwendet wird, welches dort je Work-Group definiert ist. Dementsprechend kann der Local-Memory-Verbrauch je Work-Group die maximal ausführbare Anzahl der Work-Groups je SM limitieren, da gelten muss:

$$\text{Work-Groups per SM} \cdot \text{Local-Memory per Work-Group} \leq \text{Shared-Memory per SM}$$

Diese Bedingung gilt nur für gleichzeitige Ausführung, es müssen bei hohem Local-Memory-Verbrauch ggf. mehr Work-Groups sequentiell abgearbeitet werden. Wenn allerdings bereits der Local-Memory-Bedarf einer Work-Group zu hoch ist, kann der entsprechende Kernel nicht ausgeführt werden.

Außerdem gehören zu jedem SM eine Reihe von **Streaming-Prozessoren (SP)** (auch: Cuda-Cores), welche über gemeinsame Kontroll-Logik und Instruction-Cache verfügen. Jeder SP wiederum kann mehrere **Threads** in Hardware starten. Die Threads sind gruppiert zu sogenannten **Warps**. Dabei handelt es sich um eine implementationsabhängige Anzahl von Threads mit konsekutiven ids, welche im SIMD-Stil ausgeführt werden. Die Anzahl der Threads eines Warps ist für alle bisherigen Nvidia GPUs 32. Wenn Threads eines Warps, z.B. durch Fallunterscheidungen im Code eines Kernels, verschiedene Pfade im Kontrollfluss nehmen, spricht man von Warp-Divergenz. Die SIMD-

Hardware muss dann so viele Durchgänge ausführen, wie es divergente Pfade gibt. Dabei werden immer nur diejenigen Threads Operationen erlaubt, die sich im jeweiligen Durchlauf im 'richtigen' Pfad befinden.

Um eine Idee für die Anzahl ausführbarer Threads durch eine GPU zu bekommen, schauen wir uns die Daten einer Nvidia GeForce GTX 670 an, welche primär für Entwicklung und Messungen in dieser Arbeit Verwendung fand:

- Architektur: Kepler (GK104)
- Einführung: Mai 2012
- SM Anzahl: 7
- SP Anzahl: 1344
- Thread Anzahl: 7 * 2048 = 14336

17.1.2. Speicherhierarchie

Eine Nvidia GPU verfügt über eine Speicherhierarchie, deren Struktur der der OpenCL Speicher (Kapitel 16) nicht unähnlich ist. Die Kenntnis darüber, wann welche Speicher einzusetzen sind und welche Zugriffsmuster jeweils vorteilhaft sind, ist wichtig für eine hohe Performance.

Global-Memory

Das Global-Memory, auch Device-Memory, stellt den bei weitem größten Speicher einer GPU dar. Beispielsweise verfügt unsere Nvidia GeForce GTX 670 über zwei Gigabyte Global-Memory. Allokation und erlaubte Zugriffe durch Host und Work-Items entsprechen dem gleichlautenden Speicher aus OpenCL. Aktuelle GPUs verwenden in der Regel GDDR5-DRAM-Speicher, welcher im Vergleich mit dem DDR(3/4)-DRAM-Speicher aktueller CPUs über eine höhere Bandbreite aber auch höhere Latenz verfügt. Letztere wird idealerweise durch ausreichend hohe Parallelität 'versteckt', da Kernel und auch einzelne Threads asynchron ausgeführt werden. Wenn Speicher aus dem Global-Memory angefordert wird, blockiert ein Thread nicht, sondern erst, wenn die entsprechenden Daten benötigt werden. Sind diese dann noch nicht verfügbar, wird, wenn möglich, in einem anderen Thread weiter gearbeitet. An dieser Stelle wird auch klar, weshalb die Thread Anzahl einer GPU die SP-Zahl deutlich übersteigt.

Zum Erreichen der maximalen Bandbreite des Global-Memory, gilt es einige Bedingungen einzuhalten. So wird, wann immer auf eine Adresse eines DRAMs zugegriffen werden soll, intern auf eine Menge konsekutiver Adressen, welche die Angefragte beinhaltet, zugegriffen. Wenn Threads eines Warps auf konsekutive Adressen zugreifen, kann die Hardware dies erkennen und die Zugriffe zu einem Einzigen vereinigen. In einem solchen Fall bezeichnet man einen Zugriff als *coalesced*. Zum Erreichen der maximalen Bandbreite müssen die Zugriffe noch zusätzlich *aligned* sein.

Es gibt in modernen Nvidia GPUs (ab Fermi) einen L2-Cache, den sich alle SMs teilen. Zugriffe auf das Global-Memory können damit gecached werden.

Wir sprechen nachfolgend oft vereinfachend von 'günstigen' Speicherzugriffen, wenn, zumindest teilweise, die obigen Bedingungen erfüllt sind.

Constant-Memory

Das Constant-Memory befindet sich im Global-Memory und erlaubt Zugriffe analog zu denen des Constant-Memory aus OpenCL. Es ist nur begrenzt verfügbar (64 KB bei aktuellen Nvidia GPUs) und kann gut gecached werden. Das Constant-Memory ermöglicht hohe Bandbreiten und geringe Latenzen, wenn alle Threads gleichzeitig auf die gleiche Adresse zugreifen.

Shared-Memory

Das Shared-Memory befindet sich direkt auf den SMs und erlaubt Zugriffe analog zu denen des Local-Memory aus OpenCL. Es ermöglicht hohe Parallelität, hohe Bandbreite und eine geringe Latenz, aber nur eine geringe Menge ist je SM verfügbar. Typisch sind 48 KB (Fermi, Kepler (außer bei Compute-Capability 3.7)) bis 96 KB (Maxwell mit Compute-Capability 5.2). Unabhängig davon kann bei allen aktuellen GPUs durch eine einzelne Work-Group nicht mehr als 48 KB verwendet werden. Das Shared-Memory eignet sich vor allem zum schnellen Datenaustausch innerhalb einer Work-Group. Im Beispiel 16.2 (Seite 165) wird es verwendet, um günstige Zugriffe auf das Global-Memory zu erreichen.

Allerdings müssen zum Erreichen der maximalen Bandbreite beim Zugriff auf das Local-Memory ebenfalls Bedingungen eingehalten werden. Für das Shared-Memory aktueller GPUs existieren 32 Banks, auf welche gleichzeitig zugegriffen werden kann. Bei Verwendung des 32-Bit-Modus (alternativ 64 Bit) werden aufeinanderfolgende 32-Bit-Words auf aufeinanderfolgende Banks abgebildet. Falls mehrere Threads eines Warps auf die selbe Position zugreifen, entsteht ein **Bank-Conflict**. Alle Zugriffe darauf müssen dann serialisiert werden, wodurch sich die Bandbreite verschlechtert. Das gilt allerdings nicht, wenn auf das selbe Word einer Bank zugegriffen wird. Dieses wird einmal gelesen und dann an alle Threads des Warps weitergeleitet. In dem Beispiel 16.2 (Seite 165) treten statische Bank-Conflicts auf. Wie diese bei einem ähnlich gearteten Beispiel (allerdings in Cuda formuliert) gelöst werden können, zeigt [Har15]. In dieser Arbeit spielen bei Verwendung des Shared-Memory datenunabhängige Bank-Conflicts nur eine untergeordnete Rolle, weshalb deren Auflösung hier nicht weiter vertieft wird.

Es gibt in modernen Nvidia GPUs (ab Fermi) je einen L1 Cache je SM. Zugriffe auf das Local-Memory können damit gecached werden.

Register

Register sind einzelnen Threads zugeordnet und nur diese können auf sie zugreifen. Sie stellen den schnellsten Speicher dar und Werte aus dem OpenCL Private-Memory können darin abgelegt werden. Es müssen keine bestimmten Zugriffsmuster beachtet

werden, aber ihre Menge ist begrenzt. Typische Werte je SM sind 128 KB (32K * 32 Bit, Fermi) oder 256 KB (Kepler (außer Compute-Capability 3.7) und Maxwell). Zur Ausführung eines Kernels ist eine bestimmte Registerzahl je Thread erforderlich. Bei hohem Registerverbrauch kann dadurch die Anzahl der startbaren Work-Groups je SM limitiert werden. Benötigt bereits eine einzige Work-Group mehr Register als je SM verfügbar sind, müssen Daten in das Global-Memory ausgelagert werden. In diesem Fall, der typischerweise mit einem Performanceverlust einhergeht, spricht man von Register-Spilling.

17.1.3. Compute-Capability

Nvidia verwendet den Begriff Compute-Capability (CC), um verfügbare Funktionen, vorhandene Ressourcen und die SM-Konfiguration (SP-Anzahl, Cache-Größen, etc.) zusammengefasst durch einen Wert angeben zu können. Weitere Eigenschaften, wie GPU-Taktung, SM-Anzahl, Global-Memory (Menge, Taktung, Anbindung) sind nicht darin enthalten. Somit sagt die Compute-Capability allein nicht viel über die GPU-Leistung aus, ist aber für die Optimierung und auch Ausführbarkeit entscheidend. Ab Cuda 7.0 ist die Unterstützung für Compute-Capability 1.x eingestellt, weshalb diese, ausgenommen dieses Abschnitts, auch nicht mehr besprochen wird.

Funktionen

Ein Beispiel für eine Funktionalität, welche nicht für jede Compute-Capability (nämlich ab CC 3.5) verfügbar ist, stellt Dynamic-Parallelism dar. Damit konnte erstmals eine GPU selbst Kernel starten und zwar in einer durch sie selbst festgelegten Parallelität. Zwar bietet die OpenCL 2.0 Spezifikation mittlerweile eine vergleichbare Funktion an, diese wird aber von Nvidia GPUs derzeit nicht unterstützt.

Weitere Beispiele sind die Unterstützung von 64 Bit Fließkommazahlen (ab CC 1.3) oder Warp-Vote-Functions (ab CC 1.2). Letztere ermöglichen einen besonders effizienten Austausch von Informationen zwischen Threads eines Warps, ohne andere Ressourcen (etwa Shared-Memory) zu verwenden. Obwohl schon recht alt, steht auch diese Funktion für Nvidia Karten unter OpenCL nicht zur Verfügung.

Die Implementationen in dieser Arbeit sind unter Verwendung von Nvidia Grafikkarten für OpenCL entwickelt worden, sodass allein deshalb die Funktionalität den Umfang von OpenCL 1.1 nicht übersteigt. Aber auch diese wird nicht vollständig benötigt. Hinsichtlich der Funktionalität genügen für die Ausführung aller in dieser Arbeit vorgestellten Varianten OpenCL 1.0 und Compute-Capability 1.0.

Ressourcen

Die für die Optimierung in dieser Arbeit wichtigen GPU-Ressourcen sind die Shared-Memory-Größe, die maximale Registeranzahl je SM, die maximale Threadzahl je SM und die gesamte Thread-Anzahl. Für die verwendeten Grafikkarten sind jeweils die Compute-Capability und die einzelnen davon abhängigen Ressourcen in Tabelle 17.1 aufgelistet.

Tabelle 17.1.: Mit der Architektur bzw. Compute-Capability zusammenhängende Eigenschaften der in dieser Arbeit verwendeten GPUs. Quellen: [Nvi15e, Nvi15f, Nvi15j, Nvi15h, Nvi15l, Nvi15g, Nvi15k, Nvi15i, Nvi15a]

GPU	Architektur	Compute-Capability	32 Bit Register je SM	Shared-Memory je SM	Threads je SM
GTX 480	Fermi	2.0	32K	48 KB	1536
GTX 670	Kepler	3.0	64K	48 KB	2048
GTX Titan	Kepler	3.5	64K	48 KB	2048
GTX 980	Maxwell	5.2	64K	96 KB	2048

Tabelle 17.2.: Eigenschaften der in dieser Arbeit verwendeten GPUs, welche auch von der SM-Anzahl abhängen. Quellen: [Nvi15e, Nvi15f, Nvi15j, Nvi15h, Nvi15l, Nvi15g, Nvi15k, Nvi15i, Nvi15a]

GPU	SM-Anzahl	SP-Anzahl	Threads insgesamt	MGCP [W]
GTX 480	15	480	23040	250 [Wik15c]
GTX 670	7	1344	14336	170
GTX Titan	14	2688	28672	250
GTX 980	16	2048	32768	165

Zusätzlich zeigt Tabelle 17.2 einige Eigenschaften der GPUs, welche darüber hinausgehen. In den meisten Fällen ergeben sie sich aus der SM-Anzahl. Der von Nvidia angegebene MGCP-Wert kann als (grober) Hinweis auf den Stromverbrauch einer GPU in Watt gesehen werden. Es gibt eine ganze Reihe weiterer Ressourcen, welche aber in keiner Implementation in dieser Arbeit limitierend wirken, weshalb hier lediglich einige Beispiele aufgeführt werden: Maximale Instruktionszahl je Kernel, maximale Anzahl der Work-Groups je SM und maximale Registeranzahl je Thread.

Zur Ausführung der in dieser Arbeit entstandenen Implementation mit dem höchsten Optimierungsgrad (Impl IV, Kapitel 23) sind einige Ressourcen mindestens erforderlich:

- 32 KB Shared Memory je SM
- 1024 Threads je Work-Group

Diese sind ab Compute-Capability 2.0, und damit für jede noch durch Nvidia unterstützte Version, garantiert. Für die übrigen Varianten in dieser Arbeit ist nichts davon vorausgesetzt.

17.1.4. Optimierung in dieser Arbeit

Das wichtigste Ziel bei der Erstellung einer für GPUs geeigneten Implementation ist das Ermöglichen ausreichend hoher Parallelität, um eine GPU gut auszulasten. Dazu müssen Kernel möglichst mit einer GWS gestartet werden können, welche mindestens so hoch wie die Anzahl insgesamt ausführbarer Threads der GPU ist. Dies hängt, wie wir

Tabelle 17.3.: In dieser Arbeit verwendete Intel CPUs.

CPU	Kerne	Threads	Takt [GHz]	Einführung	TDP [W]
Core i3 2100 [Int15a]	2	4	3,1	Q1'11	65
Core i7 2700k [Int15b]	4	8	3,5	Q4'11	95
Core i7 5960X [Int15c]	8	16	3,0	Q3'14	140

sehen werden, vor allem von Eigenschaften des zu implementierenden Algorithmus ab. Aber auch ein ausreichend parallel formulierter Algorithmus kann eine GPU ungünstig auslasten, wenn der Ressourcenverbrauch (Shared-Memory, Register) je SM der Implementation zu hoch ist. Dementsprechend muss dieser sorgsam ausbalanciert werden.

Sei beispielsweise eine Work-Group für LWS(256, 1, 1) konfiguriert und verwende 32K Shared-Memory. Dann kann z.B. im Falle der Kepler-Architektur, bedingt durch den Shared-Memory-Verbrauch, nur eine Work-Group je SM gestartet werden. Somit werden lediglich 256 der 2048 Threads je SM verwendet. Folglich kann mit dieser Konfiguration, sofern nichts anderes noch weiter limitiert, niemals mehr als 1/8 der maximalen Thread-Auslastung der GPU erreicht werden. Allerdings ist, jedenfalls laut einigen Quellen [Vol10], zumindest maximale Thread-Auslastung für das Erreichen der Peak-Performance nicht zwingend erforderlich.

Auch die anderen in diesem Kapitel genannten Punkte, günstige Zugriffsmuster auf Global- und Local-Memory, Vermeidung von Warp-Divergenz und ein forcieren der asynchronen Arbeitsweise zur Verringerung der Latenz, werden berücksichtigt. Außerdem wird versucht, möglichst wenig zu synchronisieren, das gilt vor allem für Synchronisation mit dem Host.

17.2. Optimierungen für CPUs

Der größte Unterschied zwischen den Anforderungen einer GPU und einer CPU ist die erforderliche Parallelität, um diese auszulasten [GHK+13]. Im Gegensatz zu einer GPU, welche 10000 Threads und mehr starten kann, verfügt keine im Rahmen dieser Arbeit verwendete CPU über mehr als 16 Hardware-Threads (Intel Core i7-5960X, 8 Kerne + SMT). Dementsprechend können auch Algorithmen, welche nur eine geringe Parallelität ermöglichen, auf einer CPU im Gegensatz zu einer GPU zu guter Auslastung führen.

Ein weiterer wichtiger Punkt bei den Optimierungen für eine CPU sind günstige Zugriffsmuster auf das Global-Memory. Wie bei einer GPU ist eine hohe Lokalität für die Maximierung der Bandbreite erforderlich [Int15d]. Allerdings gilt diese Lokalität bei einer GPU für gleichzeitig zu ladende Elemente und bei einer CPU für sequentielle Zugriffe. Diesen Unterschied zu beachten ist ausgesprochen wichtig, wenn in einem Kernel explizit sequentiell vorgegangen werden muss.

Wie in [Int15d] beschrieben, geschieht das Caching im Falle aktueller CPUs durch die Hardware selbst und explizites Cachen via Local-Memory, bewirkt einen (unnötigen) Overhead. Daher wird das Local-Memory bei CPU-optimierten Varianten niemals verwendet.

Andere Evaluationen von OpenCL auf CPUs schlagen die Verwendung einer möglichst großen LWS vor [LPN+13]. Sie argumentieren, dass eine Work-Group typischerweise (sequentiell) durch einen CPU-Kern abgearbeitet wird und kleine Größen somit den Scheduling-Aufwand erhöhen. Im Falle der eigenen CPU-optimierten Implementationen des Algorithmus hat es sich jedoch als besser erwiesen, den minimalen Wert, LWS(1, 1, 1), zu verwenden und stattdessen explizit sequentiell innerhalb der Kernel zu arbeiten. Weil die maximale LWS aktueller CPU-Implementationen von OpenCL auf 1024 begrenzt ist, kann so die Anzahl zu startender Work-Groups weiter verringert werden. Auf diese Weise wird allerdings die automatische Vektorisierung, die in [Int15d] beschrieben ist, verhindert. In dieser Arbeit führt das jedoch nicht zu einem experimentell nachweisbaren Verlust an Performance.

In der vorliegenden Arbeit wird die Optimierung für CPUs aufgrund des Fokus auf GPUs nicht weiter als bis hier beschrieben vertieft. Abschließend sind in Tabelle 17.3 die in dieser Arbeit zur Evaluation verwendeten CPUs gegeben. Das wichtigste Unterscheidungsmerkmal für diese Arbeit ist dabei die Anzahl der Kerne und damit letztendlich der möglichen Parallelität. Zusätzlich verfügen alle CPUs in Tabelle 17.3 über SMT (von Intel als Hyper Threading bezeichnet), sodass jeder Kern zwei Hardware-Threads starten kann.

Kapitel 18.

Messungen und Methodik

Dieser Abschnitt beschreibt, wie die experimentellen Evaluationen der eigenen Implementationen und der Vergleichsansätze von Stava[OS11] und Chang [CCL04] durchgeführt werden. Außerdem enthalten sind Angaben zu verschiedenen Details (Versionsnummern, usw.) der verwendeten APIs, Sprachen, Treibern und des Rechners.

18.1. Laufzeitmessung des Algorithmus

Die Bestimmung der Laufzeit des gesamten Connected-Component-Labeling-Algorithmus wird immer host-seitig durchgeführt. Dazu wird in dem entsprechenden Programm vor Absetzten des ersten damit zusammenhängen Befehls die Systemzeit abgefragt. Davon ausgenommen sind mit der Initialisierung zusammenhängende Operationen und ggf. auch das Hochladen der Klassifikationsdaten zur GPU. Andere GPU-basierte Ansätze, z.B. [HLP10] und [OS11], verfahren ebenso. Schließlich befinden sich die Daten z.B. nach einer auf der GPU ausgeführten Segmentation sowieso bereits dort. Die Messung endet, nachdem der `clFinish` Befehl nach dem letzten mit dem Connected-Component-Labeling-Algorithmus zusammenhängenden Befehl zurückkehrt. Bei Cuda basierten Implementationen endet die Messung analog nach Rückkehr des letzten Aufrufs von `cuStreamSynchronize`. Folglich sind zu diesem Zeitpunkt alle Berechnungen garantiert ausgeführt. Diese Prozedur wird vielfach wiederholt und das Ergebnis gemittelt.

Die frei verfügbaren Quellcodes der Ansätze von Stava (unter [OS15]) und Chang (unter [CCL15]) werden um Routinen zur Messung der Gesamtlaufzeit analog zu der eigenen Implementation ergänzt. Beispielsweise wird im Falle von Stavas Ansatz die Ausführungszeit der Methode 'FindRegions' aus 'ccl.cpp' gemessen. Diese enthält genau diejenigen Teile des Programms wie oben für den eigenen Ansatz beschrieben.

18.2. Laufzeitmessung eines Kernels

Zur Laufzeitbestimmung eines Kernels verwenden wir das OpenCL interne Profiling mit Events, um die interne Ausführungszeit auf dem Device zu ermitteln. Dazu berechnen wir die Differenz zwischen den `CL_PROFILING_COMMAND_START` und `CL_PROFILING_COMMAND_END`, welche wir via `clGetEventProfilingInfo` von dem mit der Ausführung des ent-

sprechenden Kernels verknüpften Event erhalten. Auf diese Weise sind deutlich genauere Messungen möglich, gerade dann, wenn einzelne Kernel eine im Vergleich zur Gesamtlaufzeit des Algorithmus geringe Ausführungszeit haben. Außerdem schwanken diese Werte eigenen Erfahrungen nach deutlich weniger als die host-seitigen Messungen.

Allerdings sind hier die Zeiten für jedwede Interaktion mit dem Host, wie das Rückholen von Zwischenergebnissen zur Neukonfiguration weiterer Kernel, nicht enthalten. Somit liegt die Summe über alle Einzellaufzeiten immer unterhalb der auf dem Host zu messenden Gesamtlaufzeit. Und diese Unterschiede variieren je nach Ansatz.

Infolgedessen verwenden wir die Messung mit Events ausschließlich, um ein besseres Verständnis des Verhaltens der verschiedenen eigenen Implementationen zu erhalten. Zum Vergleich mit anderen Ansätzen ist diese Vorgehensweise nicht geeignet.

18.3. Messumgebung

Alle Messungen werden auf einem Desktop PC mit Microsoft Windows 7. 64 Bit ausgeführt. Dieser verfügt, sofern nichts anderes angegeben ist, über eine Intel Core i7 2700k CPU und 8 GB DDR3-1333 Speicher. Falls eine GPU verwendet wird, nutzt sie einen PCI-Express 2.0 Slot mit 16 Lanes. Im Betriebssystem ist der 'Energiesparplan' 'Höchstleistung' eingestellt. Außerdem wird kein 'Aero'-Design verwendet, um die GPU nicht mit dem Rendern des Desktops und der Fenster zu belasten. Während der Messungen werden ferner keine anderen Programme ausgeführt. Ein optionales 'Overclocking-Profile' des BIOS, in welchem einige Komponenten bereits standardmäßig übertaktet sind, ist deaktiviert. Sofern nichts anderes angegeben ist, sind keine Einstellungen der Hardware (z.B. Taktung) gegenüber den durch den Hersteller voreingestellten Werten verändert.

Jeweils verschieden ist das für die Ausführung des Algorithmus eingesetzte Device. Im Falle einer CPU handelt es sich dabei jeweils um andere, aber ähnlich konfigurierte, Rechner. Die einzelnen verwendeten CPUs und GPUs sind bereits in Kapitel 17 aufgeführt.

18.4. Sprach-, API- und Treiberversionen

Alle eigenen zu OpenCL basierten Implementationen gehörigen Programme sind in Java (Version 7) geschrieben. Die OpenCL API wird mithilfe der Open-Source-Bibliothek LWJGL (Lightweight Java Game Library) [lwj15] angebunden. Wir nutzen von dieser Bibliothek die Version 2.9.3.

Wird als OpenCL Device eine GPU eingesetzt, verwenden wir den Nvidia Grafikkartentreiber 337.88, welcher OpenCL 1.1 Unterstützung bietet und intern auf Cuda 6.0 basiert. Diese Angaben können über OpenCL API Befehle abgefragt werden.

Bei Verwendung einer CPU wird die Intel OpenCL Implementation für x86-CPUs (Version 14.2) genutzt.

Für den Vergleich mit Stava [OS11] werden sowohl für die eigenen Ansätze, als auch für Stava Cuda 7.0 und der Grafikkartentreiber 347.88 verwendet. Die Kernel werden unter Verwendung der 'Driver API' jeweils für die passende Compute-Capability der eingesetzten GPU übersetzt. Weil Stavas Originalcode fest für Compute-Capability 2.0 konfiguriert ist, wird er entsprechend abgeändert, um dieses auch hier zu ermöglichen.

Die C++ Version des eigenen Algorithmus sowie die Vergleichsansätze von Chang [CCL04] und Stava [OS11] werden mit dem internen Compiler von Microsoft Visual Studio 2013 im Release-Mode übersetzt.

Kapitel 19.

Datensätze

Die experimentelle Evaluation der eigenen Ansätze, auch im Vergleich mit Stava [OS11] und Chang [CCL04], geschieht unter anderem hinsichtlich der Datenabhängigkeit. Dafür werden einige Datensätze definiert, welche sich durch folgende Eigenschaften auszeichnen:

- Skalierbarkeit: Die Datensätze müssen in beliebigen Auflösungen generierbar sein. Nur so kann die Abhängigkeit der Laufzeit von der Problemgröße bestimmt werden. Das schließt vor allem hohe Datenauflösungen mit ein, auf denen in dieser Arbeit der Fokus liegt. Insbesondere erfüllen vorberechnete Datenbibliotheken, welche oftmals aus vielen gering aufgelösten Bildern bestehen, diese Anforderung explizit nicht.

- Interpretierbarkeit: Fast jeder Datensatz legt den Fokus auf eine andere Eigenschaft. Dies geschieht mit dem Ziel, Schwierigkeiten der einzelnen Ansätze isolieren zu können. Das wird für die einzelnen Datensätze später detaillierter besprochen.

- Breite Basis: Manche Datensätze erfüllen nicht unbedingt obiges Kriterium, sondern dienen dazu, die Evaluation insgesamt weiter abzusichern.

- Bekanntheit / Verbreitung: Einige der Datensätze sind Standarddatensätze die auch in vielen anderen Veröffentlichungen verwendet werden. Insbesondere auch solchen, mit denen die eigenen Ergebnisse nicht direkt verglichen werden. So wird, auch wenn dort andere Hardware verwendet wird, zumindest ein sehr grober Vergleich möglich.

- Reproduzierbarkeit: Alle beschriebenen Datensätze können mit simplen Vorschriften generiert werden. So ist es für nachfolgende Arbeiten einfach, die Ergebnisse zu reproduzieren und deren Ergebnisse damit zu vergleichen.

- Binärdaten: Einige Connected-Component-Labeling-Algorithmen, etwa auch Chang [CCL04], mit dem die eigenen Ergebnisse verglichen werden, können lediglich Binärdaten verarbeiten. Aus diesem Grund werden zur Evaluation primär Binärdaten verwendet, obwohl die Unterstützung von nicht binären Daten eine gewünschte Eigenschaft des eigenen Ansatzes ist.

- Nachbarschaften: Viele Ansätze in der Literatur, so auch Chang und Stava, verwenden die Definition gemäß Achter-Nachbarschaft für die Pixel. Der eigene Ansatz ist jedoch auf Verwendung der Vierer-Nachbarschaft ausgelegt. Daher werden Datensätze bevorzugt, welche sowohl hinsichtlich Vierer- als auch Achter-Nachbarschaft die gleichen Connected-Components enthalten. Die einzige Ausnahme stellt der Noise-Datensatz dar.

Besprechen wir nun die in der vorliegenden Arbeit verwendeten Datensätze, dargestellt in Abbildungen 19.1 (S. 183) und 19.2 (S. 186), und deren Eigenschaften im Einzelnen:

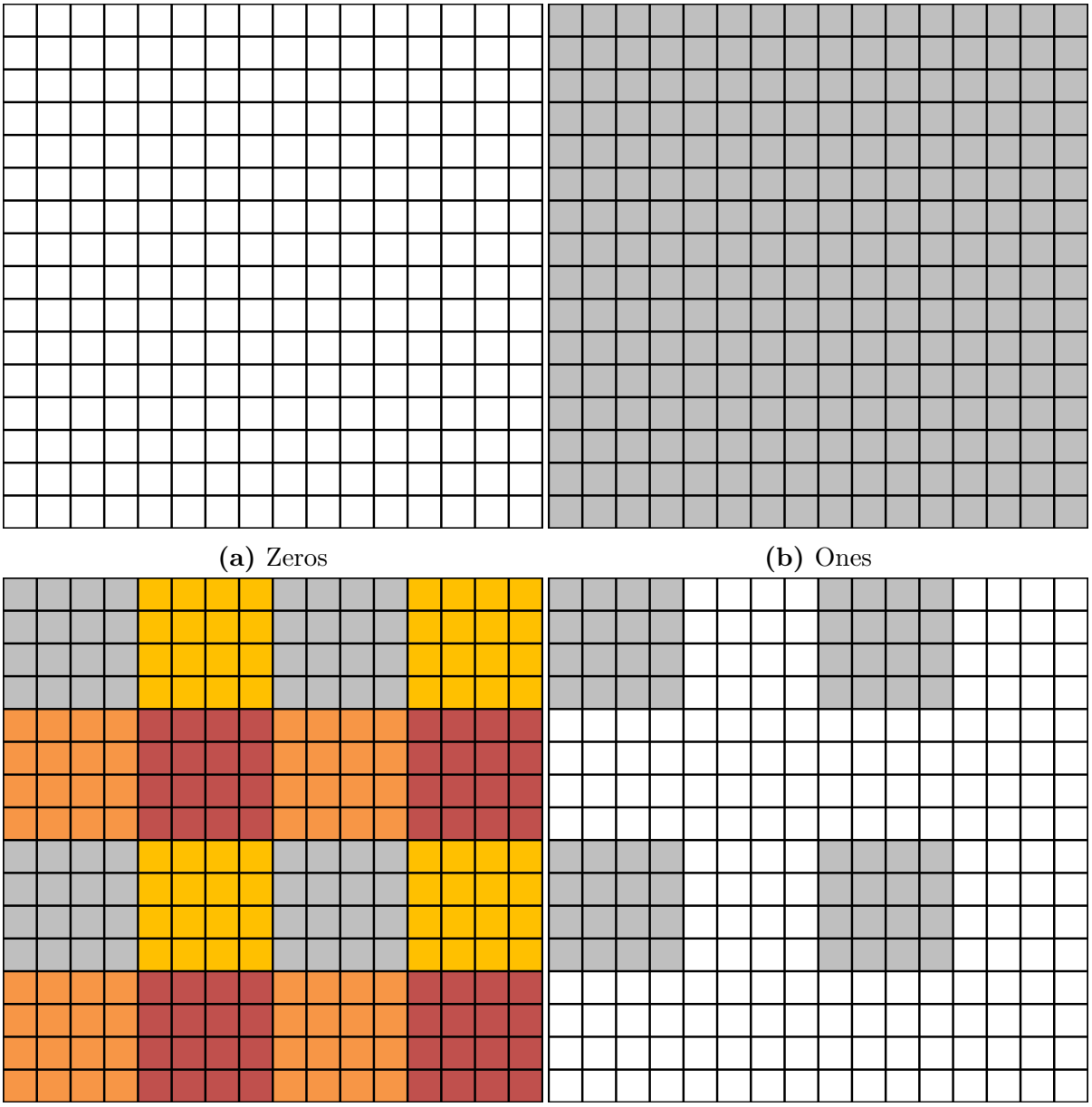
Zeros: Ein Datensatz, bestehend ausschließlich aus Nullen, welcher folglich keine einzige Connected-Component enthält. Er kann somit als Best-Case eingestuft werden. Die Laufzeiten anderer, speziell besonders schwieriger, Datensätze in Relation zu diesem geben Aufschluss über die generelle Datenabhängigkeit eines Connected-Component-Labeling-Ansatzes.

Ones: Großflächige Connected-Components sind von Stava [OS11] und ähnlichen Ansätzen als besonders schwierig eingestuft worden. Wir erwarten dagegen für konturbasierte Ansätze aufgrund des geringen Konturanteils gerade ein besonders günstiges Verhalten. Dagegen kann für konturbasierte Ansätze aufgrund des geringen Konturanteils gerade ein besonders günstiges Verhalten erwartet werden.

Speziell der Ones-Datensatz enthält nur Einsen. Somit existiert genau eine Connected-Component, welche alle N Pixel aber nur ca. 4 $\sqrt{N}$ Kontursegmente enthält. Somit stellt dieser Datensatz den Extremfall für den Vergleich hinsichtlich des obengenannten Kriteriums dar.

Quads: Dieser Datensatz besteht vollständig aus klassifizierten Quadraten der Breite k Pixel. Es werden vier Klassifikationen verwendet, sodass alle acht Nachbarquadrate jedes Quadrats anders klassifiziert sind. Folglich enthält der Datensatz $(N/k)^2$ Connected-Components der Größe k^2 Pixel. Der Parameter k kann eingestellt werden, sodass primär evaluiert werden kann, wie sich das Verhältnis von Kontursegmenten zu Pixeln im Inneren auf das Laufzeitverhalten der eigenen Ansätze im Vergleich mit dem nicht Konturbasierten von Stava [OS11] auswirkt. Mit diesem Datensatz kann außerdem die Auswirkung des Sonderfalls der eigenen Ansätze von ein Pixel großen Connected-Components getestet werden, indem $k = 1$ gesetzt wird. Hinweis: Der Ones-Datensatz ist ein Spezialfall des Quads-Datensatzes für $k = \sqrt{N}$.

Binary Quads: Dieser Datensatz besteht aus einem sich wiederholenden quadratischen Muster der Breite $2 \cdot k$ Pixel. Darin ist jeweils lediglich das obere linke Teilquadrat der Breite k Pixel mit Eins klassifiziert und der Rest sind Nullen. Der Datensatz ähnelt somit dem Quads-Datensatz. Er enthält aber um Faktor vier weniger Connected-Components. Der Untersuchungszweck entspricht ebenfalls dem des Quads Datensatzes. Aufgrund der enthaltenen Binärdaten ist er aber im Gegensatz dazu für den Vergleich mit Changs [CCL04] Ansatz geeignet.



(a) Zeros **(b)** Ones

(c) Quads, hier mit Breite k = 4 Pixel **(d)** Binary Quads, hier mit Breite k = 4 Pixel

Abbildung 19.1.: Verwendete Datensätze in dieser Arbeit, Teil I. N ist 16 * 16 = 256. Weiß: Pixel ist unklassifiziert. Andere Farben: Pixel ist klassifiziert gemäß Farbe. Die Datensätze Quads sowie Binary Quads sind parametrisierbar und hier nur mit festem Parameter angegeben.

Spiral: Hier ist eine einzige Connected-Component enthalten, nämlich eine rechteckige Spirale der Breite ein Pixel. Sie besteht aus etwa $N/2$ Pixeln und jeder Pixel ist auch Teil der Kontur. Die Figur, und damit auch die Kontur, ist ca. $N/2$ Pixel lang. Wie zum Beispiel Hawick [HLP10] beschreibt, stellt dieser Datensatz für einige, gerade einfache, Connected-Component-Labeling-Algorithmen eine große Herausforderung dar. Er wird in einer Vielzahl verschiedener Veröffentlichungen, z.B. [HLP10], [KRKS11], [OL10], [HBARSY11], verwendet.

Weil dieser Datensatz eine Figur enthält, welche mit zunehmender Datenauflösung offensichtlich immer komplexer wird, erscheint er besonders geeignet, das Verhalten aller Ansätze in Abhängigkeit von der verwendeten Datenauflösung zu analysieren.

Aufgrund des hohen Konturanteils erscheint der Datensatz ferner gut geeignet, die eigenen Ansätze mit solchen zu vergleichen, die, wie Stava [OS11], nicht auf Konturen basieren. Der Spiral-Datensatz stellt damit ein denkbares Gegenstück zu dem Ones-Datensatz dar. Es kann hier erwartet werden, dass dessen Verarbeitung für die eigenen Ansätze besonders schwierig ist, gerade auch im Vergleich mit Stava.

Anmerkung: Aus obigen Gründen habe ich, bevor alle experimentellen Ergebnisse der vorliegenden Arbeit ermittelt waren, diesen Datensatz als mögliches Worst-Case-Szenario für den eigenen Ansatz eingeschätzt. Gerade für die Endfassungen gilt das jedoch nicht.

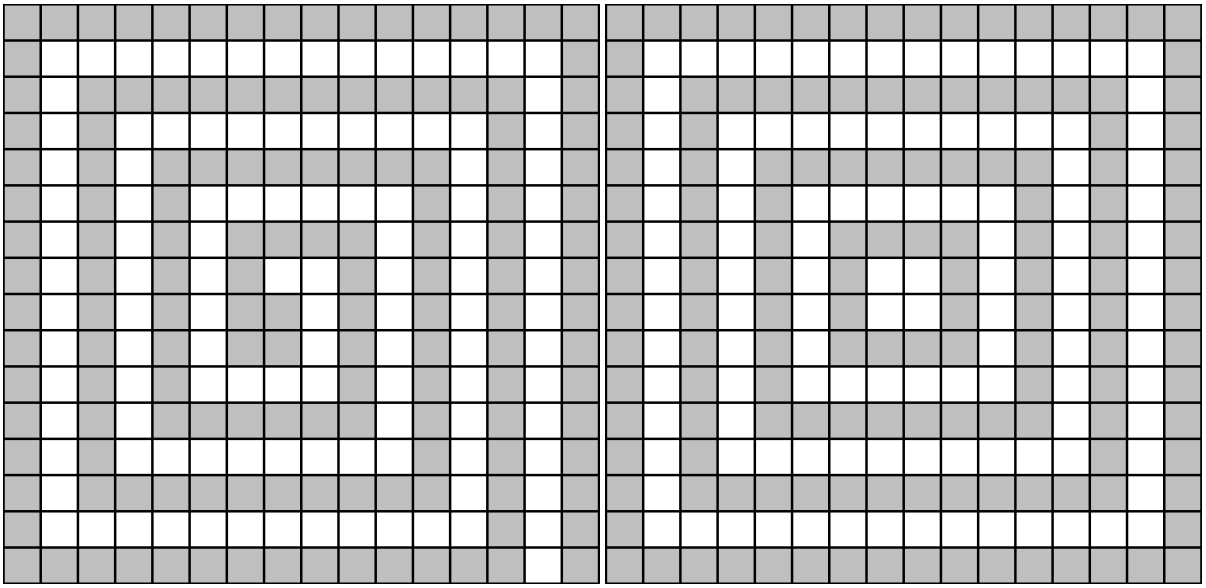
Nested Quads: Dieser Datensatz enthält verschachtelte quadratische Ringe der Breite ein Pixel. Dabei wechseln sich Ringe bestehend aus Einsen mit solchen aus Nullen ab. Die Anzahl der Pixel, welche zu einer Connected-Component gehören und die Anzahl der Kontursegmente entsprechen dabei jeweils etwa den Werten des Spiral-Datensatzes. Die maximale Länge eines Ringes (ca. $4 \cdot \sqrt{N}$) ist aber ungleich geringer. Mit diesem Datensatz sollen die Auswirkung der Verschachtelung von Connected-Components, welche hier offensichtlich maximal (nämlich $\sqrt{N}/4$) ist, evaluiert werden.

Noise: In diesem Datensatz sind zufällig verteilte Nullen und Einsen enthalten. Dabei kann der Anteil von Einsen als Parameter eingestellt werden. Bei 50 Prozent Anteil der Einsen ist der Konturanteil sehr hoch und wenn Nullen oder Einsen überwiegen nimmt er ab. Überwiegen die Einsen recht deutlich (ca. 70 Prozent) entsteht (mit sehr hoher Wahrscheinlichkeit) eine einzige Connected-Component, welche den Großteil aller klassifizierten Pixel enthält. Wir wollen mit dem Noise-Datensatz die Auswirkungen von sehr unregelmäßigen Formen auf die verschiedenen Ansätze testen.

Anmerkung: Dieser Datensatz ist erst sehr spät in den Testparcours aufgenommen worden. Insbesondere wurde er während der Entwicklung der in dieser Arbeit vorgestellten Implementationen nicht berücksichtigt.

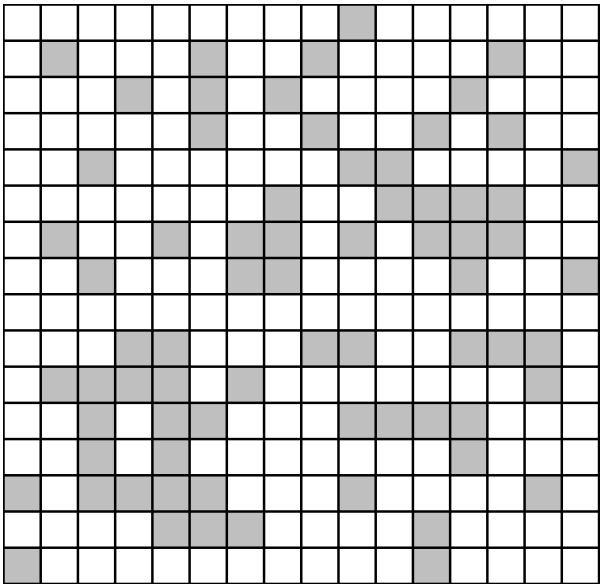
Er stellt für die eigenen Implementationen auf einer GPU eine größere Herausforderung dar als der Spiral-Datensatz.

Sieve: Dieser Datensatz besteht aus einer einzigen Connected-Component, welche 3/4 aller Pixel beinhaltet. Bei einer Einteilung des Gitters in 2 x 2 Pixel große Quadrate ist jeweils der untere rechte Pixel unklassifiziert. Somit werden im Falle der konturbasierten Ansätze insgesamt ca. $N/4$ innere Konturen um diese 'Löcher' des 'Siebs' erzeugt. Ein Datensatz des gleichen Namens wird von [HBARSY11] verwendet, dort jedoch in invertierter Form. Deren Datensatz entspricht somit dem hier vorgestellten Binary Quads Datensatz für $k = 1$.

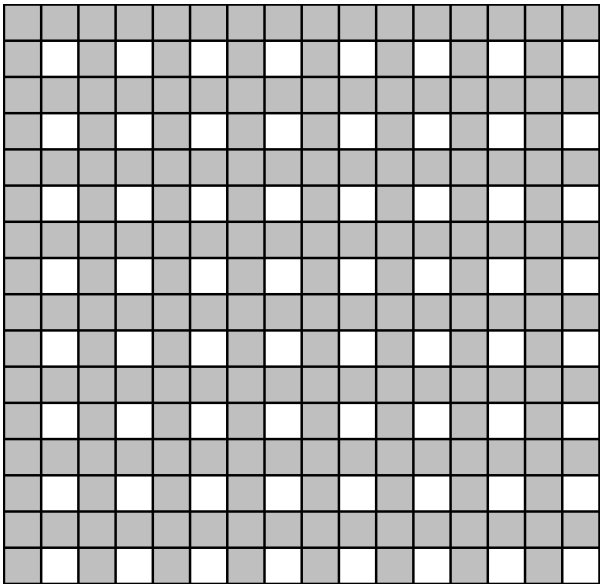


(a) Spiral

(b) Nested Quads



(c) Noise, hier mit ca. 25% Anteil Einsen



(d) Sieve

Abbildung 19.2.: Verwendete Datensätze in dieser Arbeit, Teil II. N ist 256. Weiß: Pixel ist unklassifiziert. Grau: Pixel ist klassifiziert. Der Datensatz Noise ist parametrisierbar und hier nur mit festem Parameter angegeben.

Kapitel 20.

Implementationsstrategie I: Minimierung der Operationen

Gegenstand dieses Kapitels ist die Untersuchung verschiedener Implementationen für GPUs und CPUs, welche in der Praxis die Gesamtheit der Operationen minimieren. Auf das Erreichen logarithmischer Laufzeit wird dagegen verzichtet. Für diese wäre in der Praxis für typische Auflösungen durch einzelne GPUs und vor allem CPUs eine derzeit nicht denkbare Anzahl von Recheneinheiten nötig. Zu erhoffen sind gerade auf nicht massiv paralleler Hardware, in diesem Fall CPUs, im Vergleich mit den übrigen, im Rahmen dieser Arbeit vorgeschlagenen Ansätzen, die besten Ergebnisse.

Die Implementationen für das Konturlabeling basieren auf dem in Kapitel 10.2 ab Seite 98 vorgeschlagenen Algorithmus. Dieser führt insgesamt eine lineare Anzahl von Operationen aus und muss, gerade im Gegensatz zu dem auf optimalem List-Ranking basierenden Ansatz, keinerlei Zusatzaufwand betreiben, um zu ermitteln, welche Elemente parallel verarbeitet werden können. Folglich ist im Falle typischer, hier untersuchter Problemgrößen, die Anzahl der Operationen deutlich geringer.

Für die Implementationen des Füllens der Konturen wird der stack-basierte Fill-Algorithmus aus Kapitel 12.2 ab Seite 136 als Basis verwendet. Er betrachtet jeden Pixel genau einmal. Damit ist die Anzahl der Berechnungen in der Praxis geringer als es bei dem auf Sortieren basierenden Ansatz der Fall ist (vergl. Kapitel 12.3).

Die Funktionsweise dieser und anderer verwendeter Algorithmen, etwa die Kontursegmentextraktion, wird in diesem Kapitel als bekannt vorausgesetzt. Beschrieben werden, davon ausgehend, jeweils nur Details der Implementationen. Letztere werden anhand einiger Datensätze experimentell evaluiert, soweit es für eine erste Einschätzung erforderlich erscheint.

20.1. Globale Daten und Datenlayout

Bevor die eigentliche Implementation besprochen wird, werden die benötigten Daten spezifiziert. Generell wird das im Kapitel 8 eingeführte Datenlayout verwendet und alle Vorschriften, etwa zur Bestimmung der zu einem Pixel zugehörigen Kontursegmente, behalten auch in den Implementationen ihre Gültigkeit. Dieser Abschnitt fügt lediglich

einige für die Implementation mit OpenCL notwendigen Ergänzungen wie den Datentyp und einige Hilfsdatenstrukturen hinzu. Die globalen Datenstrukturen im Einzelnen:

pixel_Classification Die Eingangsdaten, eine gültige Klassifikation (Werte > 0) je Pixel, Konstante UNCLASSIFIED (0) sonst. Datentyp ist `char` und wurde gewählt, da [OS11] diesen ebenfalls verwendet. Speicherverbrauch je Pixel: 1 Byte.

pixel_Label Enthält am Ende der Ausführung für jeden Pixel die Labelinformation. Wird vorher auch für temporäre Informationen genutzt, wie in Abschnitt 20.3 beschrieben. Datentyp ist `int`. Speicherverbrauch je Pixel: 4 Byte.

pixel_SegsProperties Hilfsdatenstruktur, welche (teilweise redundante) Informationen über alle enthaltenen Kontursegmente beinhaltet. Dazu gehört auch die Existenz von io-Segmenten. Datentyp ist `char`. Speicherverbrauch je Pixel: Ein Byte.

pixel_ioCnt Entspricht der Summe über alle Werte von `ioCnt` der zu einem Pixel zugehörigen Kontursegmente. Der Wertebereich ist $\{0, 1, 2\}$ und der Datentyp ist `char`. Verwendet die später nicht mehr benötigte Datenstruktur pixel_SegsProperties weiter. Speicherverbrauch je Pixel: Keiner.

labelStacks Hilfsdatenstruktur für die zum Füllen benötigten Stacks. Datentyp ist `int`. Speicherverbrauch je Pixel: Etwa 4 Byte. Es existieren zusätzlich zur Bildauflösung zwei weitere Zeilen oder Spalten.

suc Kodiert den DST-Typ des unmittelbaren Nachfolgers eines Segments. Zusammen mit Pixeladresse und DST-Typ des betrachteten Segments kann die Adresse des Nachfolgers berechnet werden. Siehe dazu auch Makro SUC_ID in Listing 20.2. Datentyp ist `char`. Speicherverbrauch je Pixel: 4 Byte.

head / tail Enthalten für jedes Segment jeweils die Speicheradresse des Head- bzw. Tail-Segments. Datentyp ist `int`. Speicherverbrauch je Pixel: Jeweils 16 Byte.

minDepth Entspricht der bisherigen, gleichlautenden, Größe, welche in Abschnitt 9.5.1 eingeführt ist und deren Verwendungszweck in Abschnitt 11.1 erläutert ist. Datentyp ist `short`. Speicherverbrauch je Pixel: 8 Byte.

status Kodiert verschiedene Eigenschaften eines Kontursegments in Form von Konstanten, wie dessen Existenz (SEG_EXISTING bzw. SEG_NOT_EXISTING), oder ob es bereits verarbeitet ist (SEG_EXISTING_AND_UNIFIED). Datentyp ist `char`. Speicherverbrauch je Pixel: 4 Byte.

label Speichert ein gültiges Label (Werte ≥ 0) oder andernfalls die Konstante INNER_CONTOUR je Kontursegment. Datentyp ist `int`. Speicherverbrauch je Pixel: 16 Byte.

Es ergibt sich damit insgesamt ein Speicherverbrauch von ca. 74 Byte je Pixel. Die Listings 20.1 und 20.2 enthalten einige unmittelbar das Datenlayout betreffende Makros

und Konstanten. Diese werden von verschiedenen Kerneln dieses Kapitels und darüber hinaus verwendet.

```
// The following are added once image resolution is known
// #define N ... // Number of pixels
// #define X ... // Image width in pixels
// #define Y ... // Image height in pixels

#define RIGHT 0
#define LEFT  1
#define DOWN  2
#define UP    3

#define RIGHT_OFFSET (RIGHT * N)
#define  LEFT_OFFSET (LEFT  * N)
#define  DOWN_OFFSET (DOWN  * N)
#define    UP_OFFSET (UP    * N)
```

Listing 20.1: Allgemeine Konstanten für Implementationsstrategie I

```
#define R_PIXEL(thisPixel) (thisPixel + 1)
#define L_PIXEL(thisPixel) (thisPixel - 1)
#define U_PIXEL(thisPixel) (thisPixel - X)
#define D_PIXEL(thisPixel) (thisPixel + X)

#define RS(pixel) (RIGHT_OFFSET + pixel)
#define LS(pixel) ( LEFT_OFFSET + pixel)
#define US(pixel) ( DOWN_OFFSET + pixel)
#define DS(pixel) (   UP_OFFSET + pixel)

#define SUC_ID(sucDstType, targetPixel) \
        (((sucDstType) * N) + (targetPixel))
```

Listing 20.2: Allgemeine Makros für Implementationsstrategie I

20.2. Variante I(a): Nah am Algorithmus

In diesem Abschnitt wird zunächst eine Implementation für CPUs und GPUs beschrieben, welche nah an der Funktionsweise der Beschreibung des Algorithmus ist. Die Aufteilung in OpenCL Kernel erfolgt unter dem Gesichtspunkt der Maximierung der Parallelität von Berechnungen, innerhalb des durch die verwendeten Sub-Algorithmen möglichen Rahmens. Es ergeben sich die folgenden Kernel, angegeben in Ausführungsreihenfolge:

extractSegments Dieser Kernel implementiert den Sub-Algorithmus 13 (siehe Seite 94) zur Extraktion der Kontursegmente. Er wird als erstes ausgeführt und bekommt als Eingangsdaten ausschließlich die Klassifikationen der Pixel. Dabei ist für jeden Pixel ein Work-Item zuständig, welches die zu diesem Pixel zugehörigen vier Kontursegmente extrahiert, sofern diese existieren. Um die Behandlung der Ränder zu vereinfachen, wird eine zweidimensionale Indizierung der Work-Items verwendet. Der Kernel wird genau einmal gestartet und zwar mit der GWS (X, Y, 1).

mergeTilePairs Varianten dieses Kernels führen im Zusammenspiel die in Abschnitt 10.2.4 ab Seite 102 für Algorithmus 14 beschriebene Vereinigung der Kontursegmente aus. Für jede Major-Iteration wird ein eigens konfigurierter Kernel mit der passenden GWS ausgeführt, welche anfänglich (X/2, Y, 1) und in der letzten Major-Iteration (1, 1, 1) ist. Die Implementation wird in Abschnitt 20.2.1 detaillierter beschrieben.

setSegLabel Dieser Kernel implementiert den Sub-Algorithmus 15. Er führt für das jeweils letzte verbliebene Kontursegment den Test auf Zugehörigkeit zu einer äußeren Kontur durch und vergibt dann ggf. ein Label, wie in Abschnitt 10.2.5 (s. 104) beschrieben. Der Kernel wird genau einmal mit einer GWS von $(4 \cdot N, 1, 1)$ ausgeführt, wobei jedes Work-Item ein Kontursegment betrachtet.

passLabels Varianten dieses Kernels repräsentieren den Pass-Labels-Sub-Algorithmus, wie in Abschnitt 10.2.6 ab Seite 106 beschrieben. Auch hier werden analog zu `mergeTilePairs` vier Kernel benötigt und mit einem dazu inversen Parallelitätsschema ausgeführt.

gatherSegData In diesem Schritt werden für alle Pixel die für das Füllen nötigen Informationen der vier zugehörigen Kontursegmente gesammelt. Im Gegensatz zur Beschreibung des Algorithmus wird (in dieser Implementation) dafür ein eigener Kernel verwendet, weil der Schritt, anders als die angrenzenden Kernel, einmalig mit einer GWS von (N, 1, 1) ausgeführt werden kann. Dabei ist je ein Work-Item für einen Pixel zuständig.

fillContours Dieser Kernel führt den Stack basierten Fill-Algorithmus aus, welcher in Abschnitt 12.2 ab Seite 136 beschrieben ist. Die Work-Items starten in den Pixeln zweier gegenüberliegender Bildränder und arbeiten sich sequentiell bis zur Mitte vor. Die Verarbeitung erfolgt im Falle von GPUs spaltenweise und bei CPUs zeilenweise, um jeweils günstige Speicherzugriffsmuster zu erhalten. Die GWS beträgt demnach $(2 \cdot X, 1, 1)$ oder $(2 \cdot Y, 1, 1)$.

Da einige der obengenannten Kernel im weiteren Verlauf dieser Arbeit in optimierteren Varianten auftauchen und die hier besprochenen Implementationen nah an der Beschreibung des Algorithmus sind, wird an dieser Stelle auf eine detailliertere Besprechung des Quellcodes verzichtet.

20.2.1. Implementationsdetails zu unifyTiles

In diesem Abschnitt wird skizziert, wie der Algorithmus `unifyTiles` (siehe Seite 104) mit OpenCL implementiert werden kann. Die Parallelität ändert sich in dessen Verlauf mehrfach, sodass eine Aufteilung in verschiedene Kernel sinnvoll erscheint. Einer davon, `mergeTilePairs_X_Back`, wird exemplarisch detaillierter beschrieben und ist in Listing 20.3 gegeben. Seine Arbeitsweise ist ferner in Abbildung 20.1 veranschaulicht. Ein einzelner Aufruf dieses Kernels entspricht der Anwendung der zweiten Hälfte (in-order Verarbeitung der 'backSegs') der Funktion mergeTilePair (beschrieben auf Seite 101) auf alle Tile-Pairs einer Major-Iteration. Dabei verarbeitet jedes Work-Item die entsprechenden Kontursegmente des DST-Typs LEFT des rechten Tiles je eines Tile-Pairs. Das passiert entlang der Trennkante des jeweiligen Tile-Pairs in den `steps` Pixeln sequentiell.

Es wird ein zweidimensionaler Kernelindex verwendet und die GWS entspricht in der ersten Dimension der Anzahl der nebeneinanderliegenden Tile-Pairs und in y der Anzahl der übereinanderliegenden Tile-Pairs der jeweiligen Phase. Für jeden Aufruf des Kernels muss die GWS neu gesetzt werden. Ferner werden Offsets für die Indexabstände zwischen den Tile-Pairs und die Anzahl sequentieller Schritte (`steps`, im Abschnitt 10.2 als Minor-Iterations bezeichnet) als Parameter übergeben.

Weil das Adressierungsschema fest codiert ist, kann der Kernel allerdings nur für die Vereinigung der Kontursegmente bei Vereinigung in der x-Dimension eingesetzt werden. Die Gesamtheit der Aufrufe dieses Kernels wenden letztendlich die Funktion unifyOp (Siehe Seite 99) auf alle Kontursegmente des DST-Typs LEFT an.

Wie beschrieben, kann durch den Kernel `mergeTilePairs_X_Back` lediglich die zweite Hälfte einer Major-Iteration in x-Richtung übernommen werden. Somit sind drei weitere OpenCL Kernel nötig, welche kurz skizziert werden:

mergeTilePairs_X_Forth Dieser Kernel ist für die erste Hälfte einer Major-Iteration in x-Dimension zuständig und verarbeitet somit die Kontursegmente des DST-Typs RIGHT. Er wird immer vor dem beschriebenen Kernel `mergeTilePairs_X_Back` in jeder Major Iteration aufgerufen. Weil hier alle Kontursegmente gleichzeitig verarbeitet werden können, existiert keine `for`-Schleife wie in `mergeTilePairs_X_Back`. Stattdessen ist die GWS in der zweiten Dimension um Faktor `steps` größer.

mergeTilePairs_Y_Back Zu `mergeTilePairs_X_Back` analoger Kernel für die Verarbeitung der Kontursegmente des DST-Typs UP. Primärer Unterschied der Implementation sind andere Speicherzugriffsmuster. Sequentiell durch ein Work-Item betrachtete Pixel liegen nicht übereinander, wie im Kernel `mergeTilePairs_X_Back` der Fall, sondern nebeneinander.

mergeTilePairs_Y_Forth Zu `mergeTilePairs_X_Forth` analoger Kernel für die Verarbeitung der Konturesegmente des DST-Typs DOWN. Er muss vor `mergeTilePairs_X_Back` ausgeführt werden.

Mit diesen Kerneln kann nun der Algorithmus `unifyTiles` implementiert werden, wie im Pseudocode Abschnitt 24 gegeben. Zu entnehmen ist vor allem, wann welcher der oben erläuterten OpenCL Kernel wie konfiguriert zu starten ist. Das Schema entspricht dem in Abschnitt 10.2.4 ab Seite 102 erläuterten. Es ist allerdings für X = Y, wobei X immer eine Zweierpotenz darstellt, vereinfacht. Das gilt auch für alle weiteren Codeabschnitte.

```
kernel void mergeTilePairs_X_Back (
 global char*  suc,
 global int*   head,
 global int*   tail,
 global char*  status,
 global short* minDepth,
 const  int    steps,
 const  int    xOffset)
{
  // (temporary) storage for required ids in private memory
  int s0, s1, pixel, tail_S0, head_S1;
  int firstPixel = get_global_id(1) * steps * X
                 + get_global_id(0) * 2 * steps + xOffset + 1;

  // Visit all pixels of common edge of a tile-pair in order
  for (int k = 0; k < steps; k++) {
    pixel = firstPixel + k * X;
    s0 = LS(pixel); // Consider segment, called s0, of DST-Type LEFT

    // Apply unify-operation on s0, if existing
    if(status[s0] != SEG_NOT_EXISTING){
      s1 = SUC_ID(suc[s0], L_PIXEL(pixel));
      if(s0 != (head_S1 = head[s1])){
        tail_S0 = tail[s0];
        if(minDepth[tail_S0] > minDepth[s1]) {
          minDepth[tail_S0] = minDepth[s1];
        }
        head[tail_S0] = head_S1;
        tail[head_S1] = tail_S0;
        status[s1] = SEG_EXISTING_AND_UNIFIED;
      }
    }
  }
}
```

Listing 20.3: OpenCL C Code für Kernel `mergeTilePairs_X_Back`. Makros siehe Listing 20.1

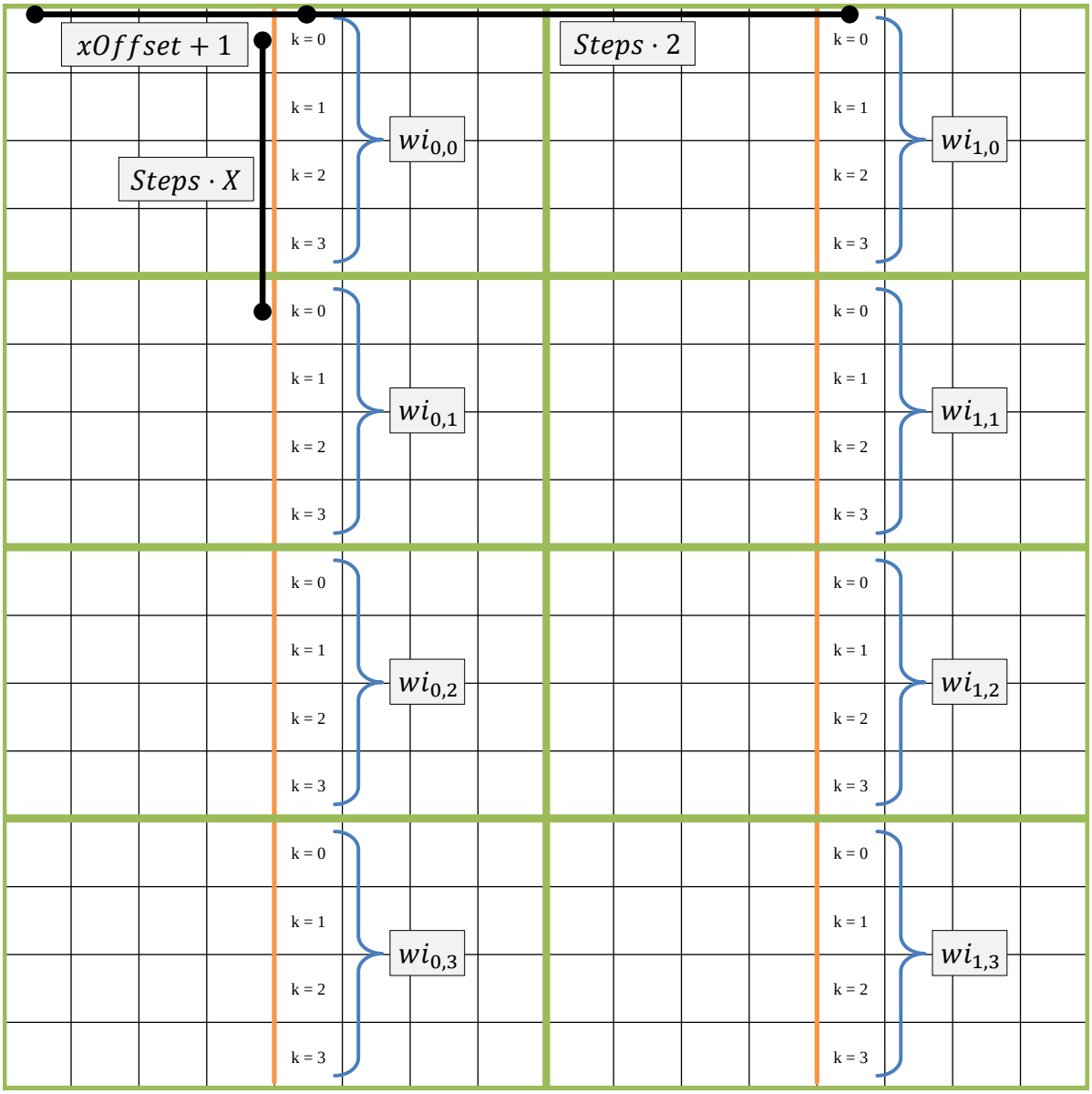


Abbildung 20.1.: Veranschaulichung der ersten Hälfte einer Major-Iteration der Tile-Pair (grüne Kästen) Verarbeitung durch den Kernel `mergeTilePairs_X_Back` aus Listing 20.3. Orange Linien stellen Aufteilung der einzelnen Tile-Pairs in Tiles dar. Blaue Klammern markieren die durch je ein Work-Item (`wi`) betrachteten Pixel. Das zuständige `wi` verarbeitet die darin enthaltenen Konturesegmente des DST-Typs `LEFT`, sofern existent. Dies passiert jeweils sequentiell, gemäß dem Schleifenindex `k`, mit `k` < `Steps` (in dieser Major-Iteration ist der Wert von `Steps` 4). Die `wi` haben zwei Indices, zuerst den Wert von `get_global_id(0)`, dann durch Komma getrennt den Wert von `get_global_id(1)`. In diesem Beispiel ist die GWS: (2, 4, 1) Für die eingezeichneten Abstände sind jeweils die Pixelindexunterschiede angegeben. Dies dient zur Veranschaulichung der Berechnung des Wertes für `firstPixel`.

Algorithm 24: Pseudo-implementation of algorithm unifyTiles

```
// Simplified pseudo-implementation of unifyTiles
// algorithm (see p. 104). Assumptions / preconditions:
// - Be X = Y and X be a power of two
// - Kernels are properly initialized
// - Global data structures are initially bound to all kernels
// - Kernel 'extractSegments' is executed
int x_or_y_Offset, tilePairsY, tilePairsX;
boolean xPhase ← TRUE;
int steps ← 1 ;
int tilePairWidth ← 1;
int tilePairHeight ← 1;
Kernel mergeTilePairs_X_Forth, mergeTilePairs_X_Back;
Kernel mergeTilePairs_Y_Forth, mergeTilePairs_Y_Back;
// Major iterations (2 per loop iteration)
Do log_2(N)/2 times
    // Phase X
    tilePairWidth ← tilePairWidth · 2;
    x_or_y_Offset ← tilePairWidth / 2 - 1;
    tilePairsX ← X / tilePairWidth;
    tilePairsY ← Y / tilePairHeight;
    mergeTilePairs_X_Forth.setArg (6, x_or_y_Offset);
    enq (mergeTilePairs_X_Forth, GWS(tilePairsX, tilePairsY · steps, 1));
    mergeTilePairs_X_Back.setArg (5, steps);
    mergeTilePairs_X_Back.setArg (6, x_or_y_Offset);
    enq (mergeTilePairs_X_Back, GWS(tilePairsX, tilePairsY, 1));
    // Phase Y
    steps ← steps · 2;
    tilePairHeight ← tilePairHeight · 2;
    x_or_y_Offset ← tilePairHeight / 2 - 1;
    tilePairsX ← X / tilePairWidth;
    tilePairsY ← Y / tilePairHeight;
    mergeTilePairs_Y_Forth.setArg (6, x_or_y_Offset);
    enq (mergeTilePairs_Y_Forth, GWS(tilePairsX · steps, tilePairsY, 1));
    mergeTilePairs_Y_Back.setArg (5, steps);
    mergeTilePairs_Y_Back.setArg (6, x_or_y_Offset);
    enq (mergeTilePairs_Y_Back, GWS(tilePairsX, tilePairsY, 1));
End
```

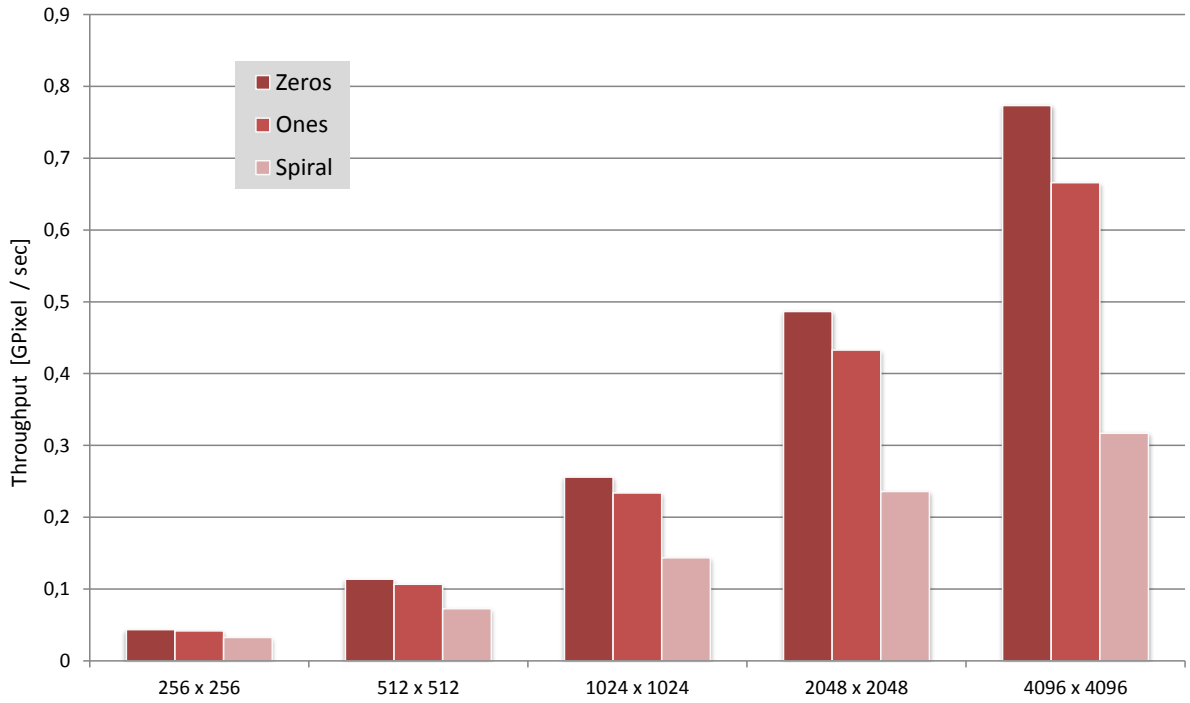


Abbildung 20.2.: Throughput bei Verarbeitung der Datensätze Ones, Zeros und Spiral in verschiedenen Datenauflösungen bei Ausführung auf einer GTX 670 GPU.

20.2.2. Evaluation auf einer GPU

Die zuvor skizzierte Implementation wird in diesem Abschnitt mit der Nvidia GeForce GTX 670 evaluiert. Besprechen wir zunächst den Ressourcenbedarf bei Übersetzung für Compute-Capability 3.0. Alle Kernel verwenden kein Shared-Memory und der maximale Registerbedarf je Thread ist 19 für den `extractSegments` Kernel. Dieser wird mit einer LWS (256, 1, 1) konfiguriert und kann so bei Compute-Capability 2.0 oder größer alle Threads starten. Insgesamt erwarten wir keine limitierte Parallelität und damit Leistung durch den Ressourcenverbrauch. Unabhängig davon stellt diese Implementation keine besonderen Anforderungen an die Funktionalität der Hardware und ist mit Compute-Capability 1.0 ausführbar.

Skalierung mit der Problemgröße

Zuerst kann der Throughput experimentell für die Datensätze Zeros (Best-Case), Ones und Spiral in Abhängigkeit von der Datenauflösung ermittelt werden. die Ergebnisse sind in Abbildung 20.2 dargestellt. Folgende Beobachtungen lassen sich machen:

- Bei allen Datensätzen nimmt der Throughput mit höheren Auflösungen erheblich zu.
- Der Throughput des Ones-Datensatzes liegt um maximal Faktor 1,16 unterhalb

dem des Best-Case-Datensatzes. Im Falle des Spiral-Datensatzes beträgt der Unterschied zum Best-Case-Datensatz maximal Faktor 2,4.

- Die absoluten und relativen Unterschiede zwischen den Datensätzen nehmen mit steigender Datenauflösung zu.

Die starke Zunahme des Throughputs mit höheren Auflösungen deutet auf eine nicht ideale Auslastung der GPU bei kleineren Problemgrößen hin. Ein solches Verhalten ist bei GPUs nicht unüblich, hier aber sehr stark ausgeprägt.

Anteile der Subalgorithmen

Für eine genauere Analyse betrachten wir nun die Anteile der einzelnen Kernel an der gemessenen Laufzeit. Dies ist für die Spirale in Abbildung 20.3 dargestellt. Folgende

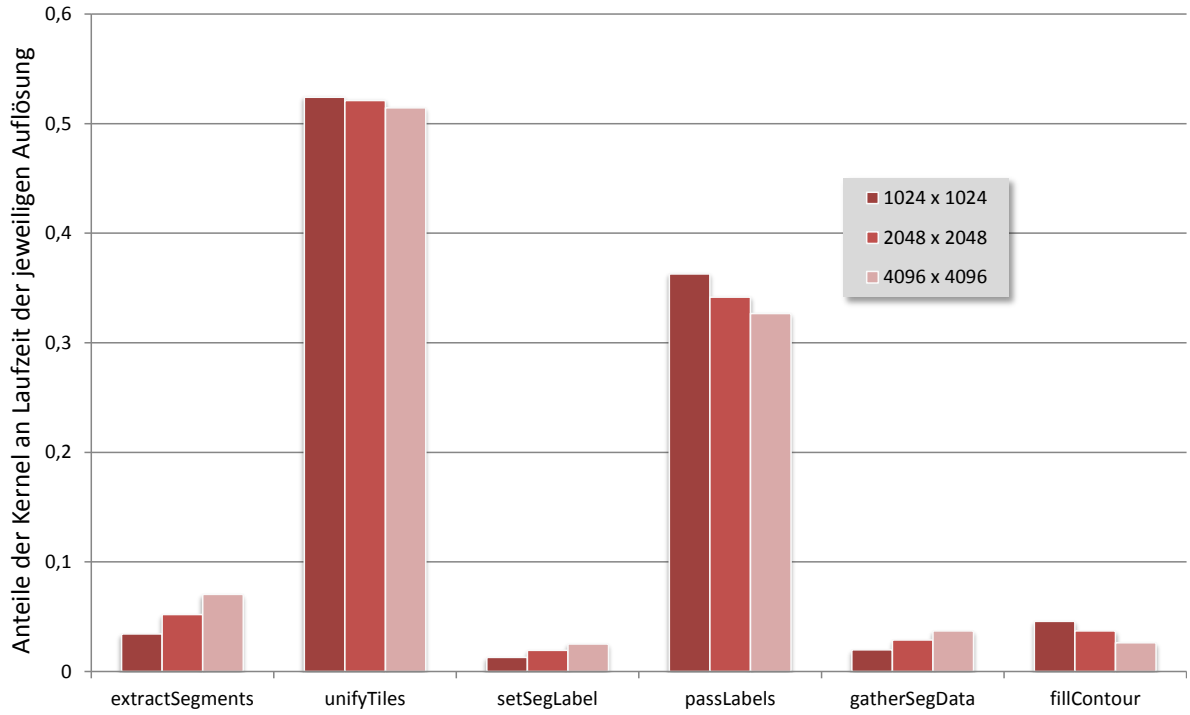


Abbildung 20.3.: Anteile der einzelnen Kernel an der Gesamtlaufzeit in verschiedenen Datenauflösungen. Dabei stellen unifyTiles und passLabels aggregierte Werte der zugehörigen Kernel dar. Datensatz: Spiral

Beobachtungen lassen sich machen:

1. In allen Auflösungen entfällt ein Großteil der Laufzeit auf die Kernel `unifyTiles` und `passLabels`. Das Konturlabeling, welches sich aus ebendiesen Kerneln und `setSegLabel` zusammensetzt, ist für insgesamt mindestens 85 Prozent der Laufzeit verantwortlich.

2. Der Anteil der Kernel `extractSegments`, `setSegLabel` und `gatherSegData` an der Laufzeit nimmt mit steigender Auflösung deutlich zu. Dies sind genau diejenigen Kernel, welche in voller Datenparallelität ausgeführt werden können.

3. Der Anteil des Kernels `fillContours` an der Laufzeit nimmt mit steigender Auflösung deutlich ab.

Beobachtung (2) lässt sich durch eine vergleichsweise gute Auslastung der GPU auch in geringen Datenauflösungen bei Kerneln deuten, die in voller Datenparallelität ausgeführt werden können. Folglich sind für eine Erklärung der starken Throughputzunahme bei höheren Auflösungen primär die übrigen Kernel zu betrachten.

Entsprechend lässt sich Beobachtung (3) durch die nicht ausreichende Parallelität im Falle geringer Datenauflösungen deuten. Diese ist in den verwendeten Auflösungen der Abbildung 20.3: 2048, 4096 und 8192. Die verwendete GTX 670 kann bis zu 14336 Hardware-Threads starten, folglich ist die Threadauslastung in den aufsteigenden Auflösungen gerundet: 14 %, 29 % und 57 %. Spätestens hier wird auch deutlich, dass der stack-basierte Füll-Ansatz im Falle noch geringerer Auflösungen nicht ideal ist.

Eine detaillierte Deutung der Beobachtung (1) ist an dieser Stelle nicht möglich, da die Werte von `unifyTiles` und `passLabels` sich aus mehreren Starts von Kerneln zusammensetzen, welche für Ausführung in unterschiedlicher Parallelität konfiguriert sind. Unabhängig davon kann festgehalten werden, dass der Optimierung des Konturlabeling-Sub-Algorithmus, zumindest auf einer GPU, gesteigerte Aufmerksamkeit geschenkt werden muss.

Betrachtung der mergeTilePairs Kernel

Nachfolgend wird das Verhalten der mit dem Schritt `unifyTiles` assoziierten Kernel genauer betrachtet. Deren Laufzeiten bei Verarbeitung der 4096 x 4096 Spirale sind, angeordnet gemäß Ausführungsreihenfolge, in Abbildung 20.4 dargestellt. Dies dient primär dem Verständnis, deutlich lesbarer ist dagegen die Abbildung 20.5, welche die gleichen Kernellaufzeiten nach Kerneltyp gruppiert zeigt. Für jeden Kernel entspricht die Anordnung hier der Ausführungsreihenfolge innerhalb dieses Kernels. Es ist dabei zu bedenken, dass zwischen zwei Ausführungen eines Kernels alle anderen Kernel ebenfalls je einmal ausgeführt werden. Es sei außerdem in Erinnerung gerufen, dass sich die Anzahl der zu verarbeitenden Kontursegmente für jeden Kernel mit jedem Aufruf halbiert. Die Beobachtungen der Kernel im Einzelnen:

1. Betrachten wir zuerst das Verhalten des Kernels `mergeTilePairs_Y_F`, welcher jeweils für alle zu verarbeitenden Segmente parallel aufgerufen werden kann. Hier nimmt die Laufzeit mit jedem Aufruf erheblich ab. Sie halbiert sich aber jedes Mal, gerade zum Ende hin, nicht ganz. Dies ist deutlich zu sehen in Abbildung 20.6 welche die zugehörigen Throughputs der hier besprochenen Kernellaufzeiten darstellt. Der Throughput, angegeben in verarbeiteten Segmenten je Zeiteinheit, verringert sich mit jedem Aufruf. Dieser Effekt tritt in den letzten Aufrufen des Kernels `mergeTilePairs_Y_F` verstärkt auf.

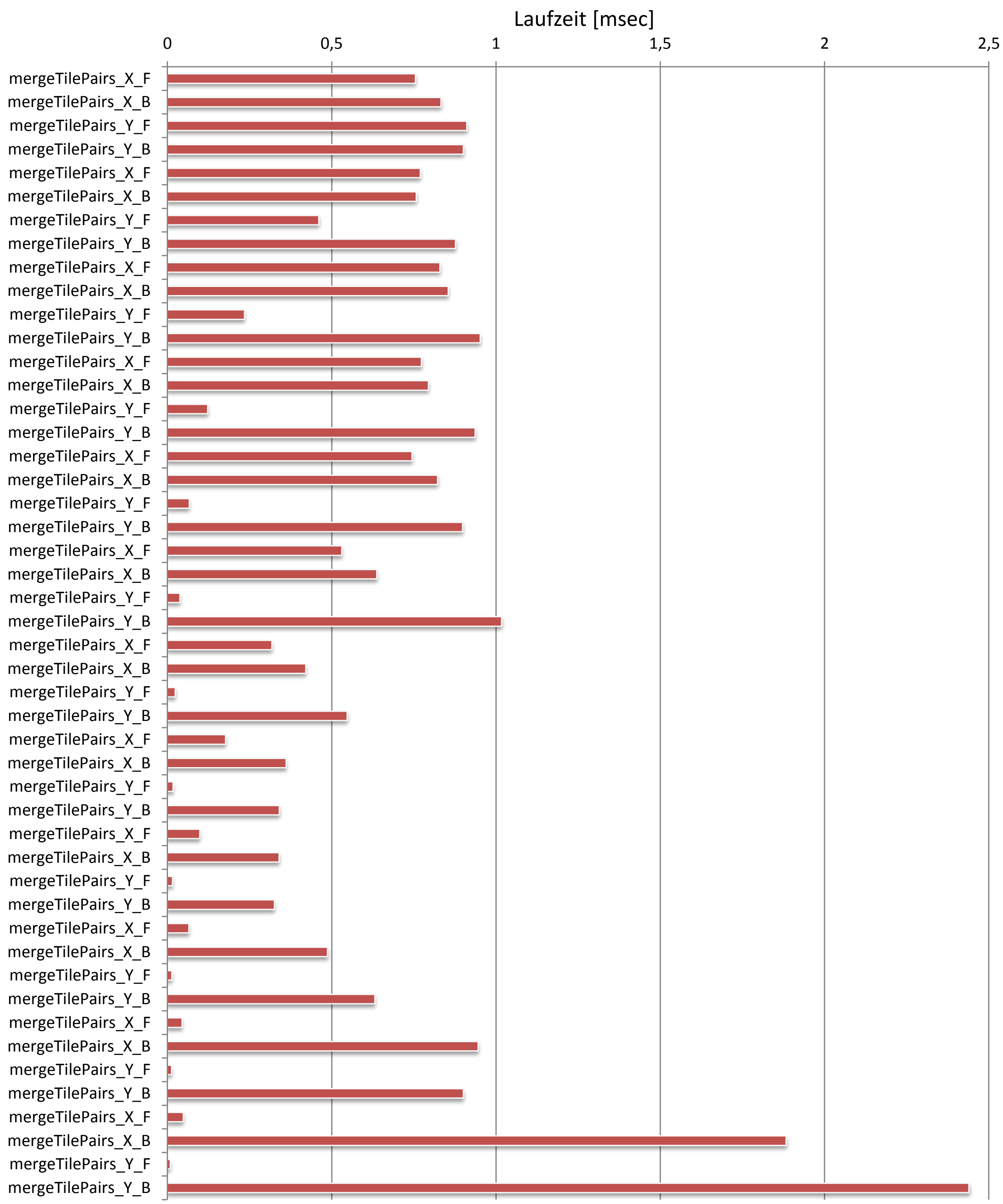


Abbildung 20.4.: Laufzeiten der einzelnen Aufrufe der Kernel `mergeTilePairs_X_F`, `mergeTilePairs_X_B`, `mergeTilePairs_Y_F` und `mergeTilePairs_Y_B`, angeordnet gemäß Ausführungsreihenfolge. Datensatz: Spirale 4096 x 4096, Device: GTX 670

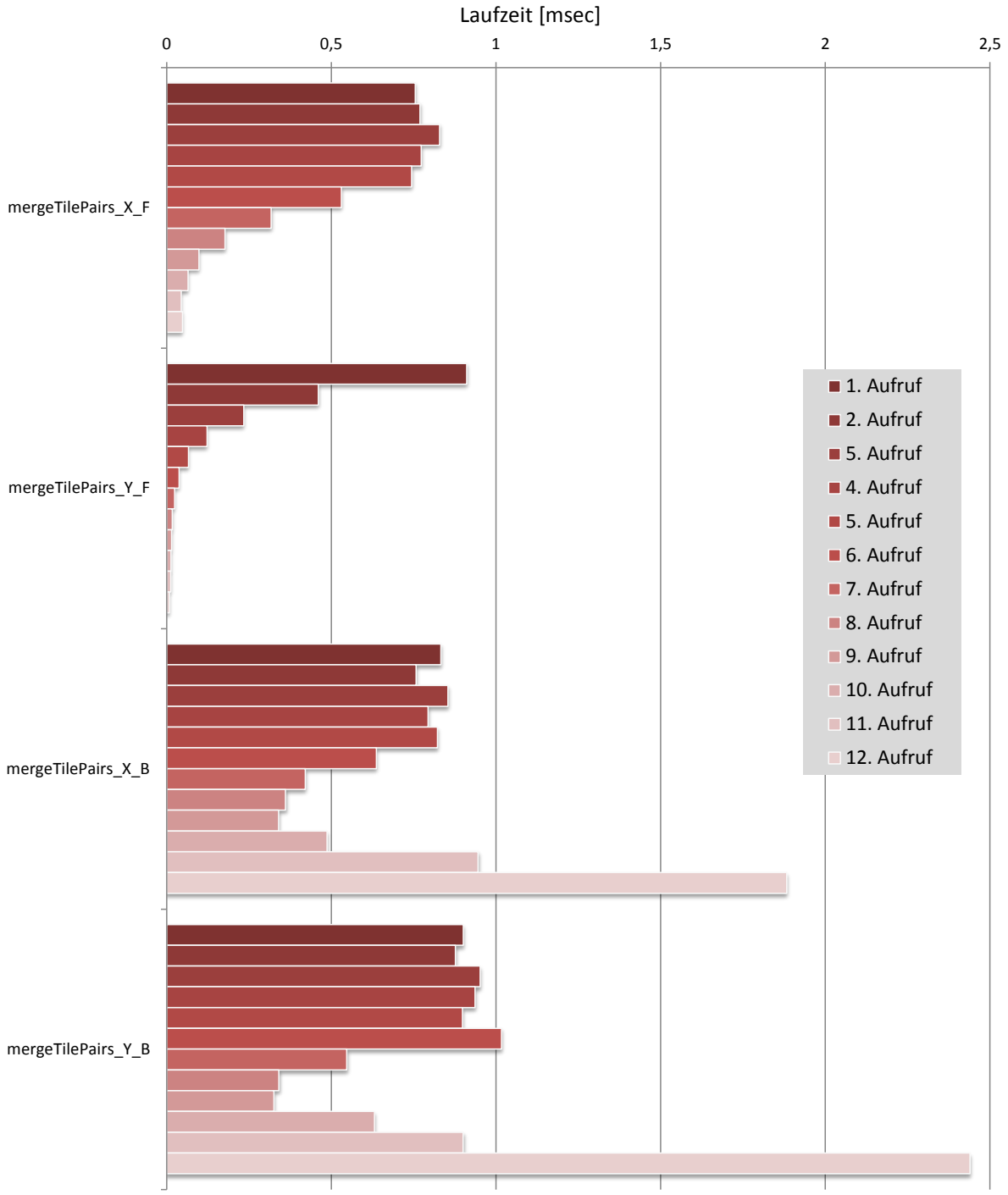


Abbildung 20.5.: Laufzeiten der einzelnen Aufrufe der Kernel `mergeTilePairs_X_F`, `mergeTilePairs_X_B`, `mergeTilePairs_Y_F` und `mergeTilePairs_Y_B`, gruppiert nach Kerneltyp. Innerhalb eines Kerneltyps angeordnet gemäß Ausführungsreihenfolge dieses Kernels. Datensatz: Spirale 4096 x 4096. Hinweis: Inhaltlich identisch mit Abbildung 20.4

2. Der Kernel `mergeTilePairs_X_F` kann ebenfalls für alle zu verarbeitenden Segmente parallel aufgerufen werden. Trotzdem ist, anders als bei Kernel `mergeTilePairs_Y_F`, über die ersten fünf Aufrufe keine Abnahme der Laufzeit erkennbar, wie in Abb. 20.5 zu sehen. Danach nimmt sie zwar ab, bleibt aber höher als bei Kernel `mergeTilePairs_Y_F`. Entsprechend fällt, wie in Abb. 20.6 dargestellt, die Abnahme des Throughputs über alle Aufrufe weitaus deutlicher aus, als beim ersten Kernel.

3. Das Verhalten der Kernel `mergeTilePairs_X_B` und `mergeTilePairs_Y_B`, die beide jeweils nur in Tile-Pair Parallelität ausgeführt werden können, ist recht ähnlich. Über die ersten Aufrufe ist keine Abnahme der Laufzeit erkennbar (siehe Abb. 20.5). Danach nimmt die Laufzeit leicht ab und am Ende stark zu. Insbesondere entspricht jeweils die Laufzeit des letzten Aufrufs in etwa dem doppelten Wert des ersten Aufrufs. Das ist bemerkenswert, schließlich verarbeitet der erste Aufruf um Faktor 2048 mehr Segmente. Im Falle des Kernels `mergeTilePairs_Y_B` übersteigt etwa der Throughput des ersten Aufrufs den Letzten um einen Faktor von über 5500.

Die extreme Abnahme des Throughputs der Kernel `mergeTilePairs_X_B` und `mergeTilePairs_Y_B` in den letzten Aufrufen lässt sich vor allem durch die dann geringe Parallelität erklären. So führt beim letzten Aufruf von `mergeTilePairs_Y_B` ein einziger Thread 4096 sequentielle Schritte aus und bei `mergeTilePairs_Y_B` führen am Ende zwei Threads je 2048 sequentielle Schritte aus, womit sich auch die hohe beobachtete Laufzeit dieser Kernel begründen lässt. Wir erwarten in Abhängigkeit der Problemgröße eine Laufzeit der letzten Aufrufe proportional zu $\sqrt{N}$, der Anzahl sequentieller Schritte entsprechend. Dies kann in Abbildung 20.7 verifiziert werden. Jede Vervierfachung der Datenauflösung bewirkt fast genau eine Verdopplung der Laufzeit. Dementsprechend verringert sich der den Throughput ausbremsende Einfluss der letzten Aufrufe der Kernel `mergeTilePairs_X_B` und `mergeTilePairs_Y_B` mit zunehmender Problemgröße. Dies ist eine Ursache für das in Abbildung 20.2 auf Seite 195 beobachtete Verhalten in Abhängigkeit der Problemgröße. Außerdem kommen bei höheren Auflösungen weitere Major-Iterationen hinzu, welche in ausreichender Parallelität ausgeführt werden können und den Einfluss der letzten Aufrufe weiter verringern.

Die Parallelität ist aber nicht alleinige Ursache für das in Abbildungen 20.5 und 20.6 beobachtete Verhalten, denn die Kernel `mergeTilePairs_X_F` und `mergeTilePairs_Y_F` verhalten sich trotz identischem Parallelitätsmuster unterschiedlich. Dies lässt sich durch die verschiedenen Muster erklären, mit denen die Pixel verarbeitet werden und die damit auch die Speicherzugriffsmuster vorgeben. Die jeweils gleichzeitig verarbeiteten Pixel beider Kernel sind für den jeweils dritten Aufruf in Abbildung 20.8 dargestellt. Der Kernel `mergeTilePairs_Y_F` verarbeitet je zeilenweise angeordnete Elemente parallel und `mergeTilePairs_X_F` spaltenweise angeordnete. In beiden Fällen nimmt der Abstand zwischen den Zeilen bzw. Spalten aufgrund der steigenden Tile-Größe mit jedem Aufruf zu. Das führt aber nur im Falle des Kernels `mergeTilePairs_X_F` unmittelbar zu ungünstigeren Speicherzugriffsmustern, welche hier letztendlich primär für die Abnahme des Throughputs verantwortlich zu machen sind.

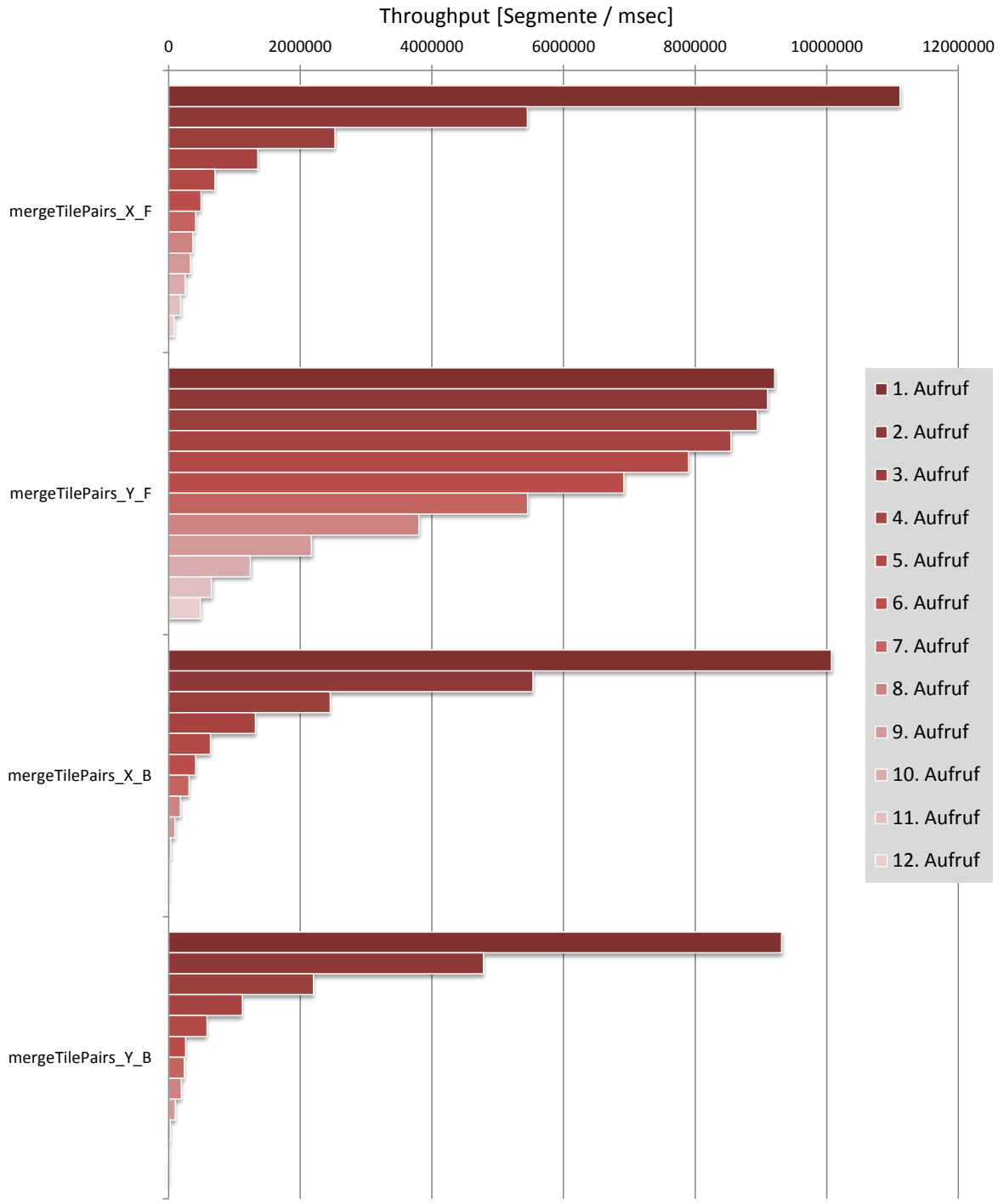


Abbildung 20.6.: Throughputs, angegeben in verarbeiteten Kontursegmenten je Zeiteinheit, der einzelnen Aufrufe der Kernel `mergeTilePairs_X_F`, `mergeTilePairs_X_B`, `mergeTilePairs_Y_F` und `mergeTilePairs_Y_B`, gruppiert nach Kerneltyp. Innerhalb eines Kerneltyps angeordnet gemäß Ausführungsreihenfolge dieses Kernels. Datensatz: Spiral 4096 x 4096

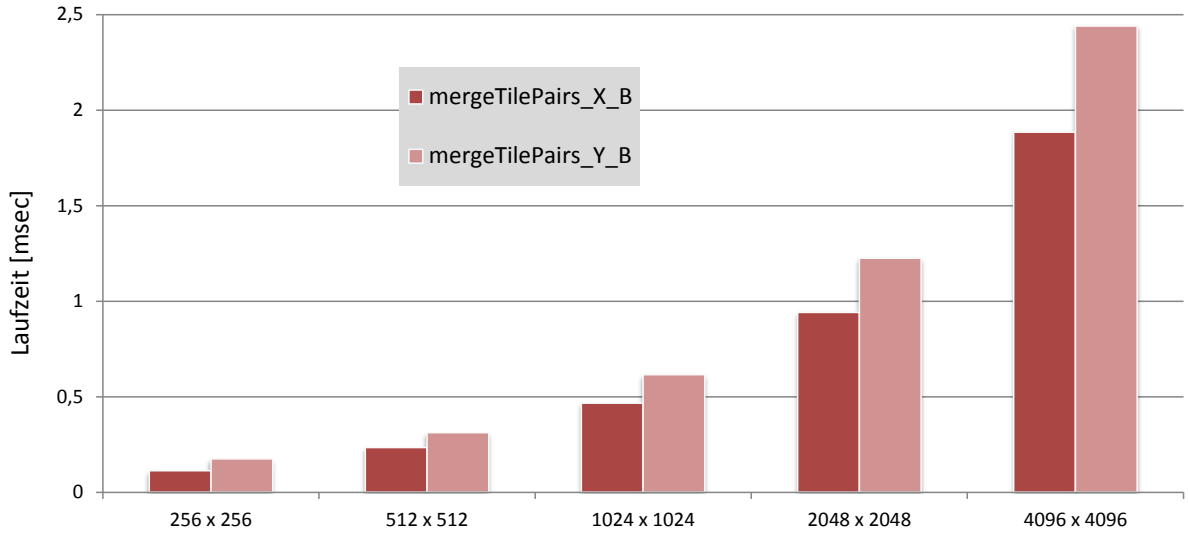


Abbildung 20.7.: Laufzeiten der jeweils letzten Aufrufe der Kernel `mergeTilePairs_X_B` und `mergeTilePairs_Y_B` bei Verarbeitung verschiedener Problemgrößen. Datensatz: Spiral, Device: GTX 670

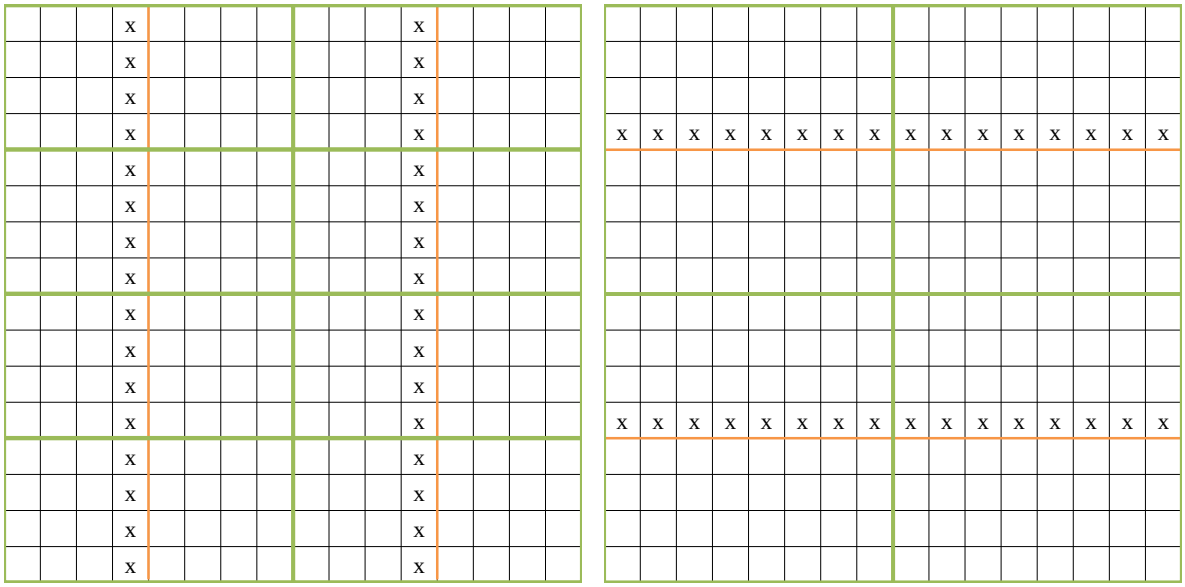


Abbildung 20.8.: Veranschaulichung der gleichzeitig betrachteten Pixel (markiert durch X) und damit Speicherzugriffe der Kernel `mergeTilePairs_X_F` (links) und `mergeTilePairs_Y_F` (rechts) bei ihrem jeweils dritten Aufruf. Tile-Pairs sind durch grüne Kästen angegeben und orange Linien stellen Aufteilung der einzelnen Tile-Pairs in Tiles dar.

Unabhängig davon nimmt der Throughput des Kernels `mergeTilePairs_Y_F` auch etwas ab, was sich nicht hierdurch erklären lässt. Insbesondere in den ersten Aufrufen kann dies auch nicht durch die Parallelität begründet werden. Eine denkbare Erklärung ist die zunehmende Wahrscheinlichkeit ungünstiger Speicherzugriffe auf die Head- und Tail-Segmente, nachdem bereits einige Vereinigungen durchgeführt worden sind und sich folglich komplexere Linienzüge ergeben können. Dieses Verhalten ist im Gegensatz zu den übrigen Beobachtungen bei dieser Implementation, welche unmittelbar aus dem ansonsten festen Arbeitsschema resultieren, stärker datenabhängig.

Beide speicherzugriffsmusterabhängigen Effekte treten in ähnlicher Weise auch in den übrigen `mergeTilePairs` Kerneln auf, lassen sich dort aber schwieriger isolieren.

Abschließend ist noch auf das Verhalten der mit `passLabels` assoziierten Kernel einzugehen. Dieses entspricht, wenn auch in invertierter Reihenfolge, aufgrund der gleichen Speicherzugriffsmuster und des gleichen Parallelitätsmodells dem der entsprechenden `mergeTilePairs` Kernel.

20.2.3. Evaluation auf einer CPU

Die in diesem Kapitel skizzierte Implementationsstrategie wird in diesem Abschnitt mit einer CPU evaluiert. Im Gegensatz zur GPU Implementation des vorherigen Abschnitts arbeitet der `fillContours` Kernel allerdings mit je einem Work-Item pro Zeile (statt Spalte). Außerdem werden die, noch nicht besprochenen, Optimierungen für das asynchrone Laden der Daten nicht verwendet.

Skalierung mit der Problemgröße

Zuerst kann der Throughput experimentell für die Datensätze Zeros (Best-Case), Ones und Spiral in Abhängigkeit von der Datenauflösung ermittelt werden. Die Ergebnisse sind in Abbildung 20.9 dargestellt. Beobachtungen:

1. Der Throughput ist weitaus weniger von der verwendeten Auflösung abhängig, als es bei der GPU der Fall ist (Abb. 20.2, Seite 195). Ab der Auflösung 1024 x 1024 bleibt er etwa konstant. In geringeren Auflösungen fällt er etwas ab.

2. Ausnahme davon ist der Spiral-Datensatz und insbesondere der Peak bei der Auflösung 512 x 512.

3. Der relative Throughputunterschied des Zeros-Datensatzes zu den anderen beiden fällt größer aus als bei der GPU.

4. Die Unterschiede zwischen den Datensätzen werden klein bei den geringen Datenauflösungen.

Die geringe Abhängigkeit des Throughputs von der Datenauflösung war zu erwarten, da die CPU lediglich 8 Hardware-Threads ausführen kann und der Berechnungsanteil des Algorithmus mit einer Parallelität < 8 verschwindend gering ist. Nicht unmittelbar klar ist das Verhalten bei geringen Auflösungen.

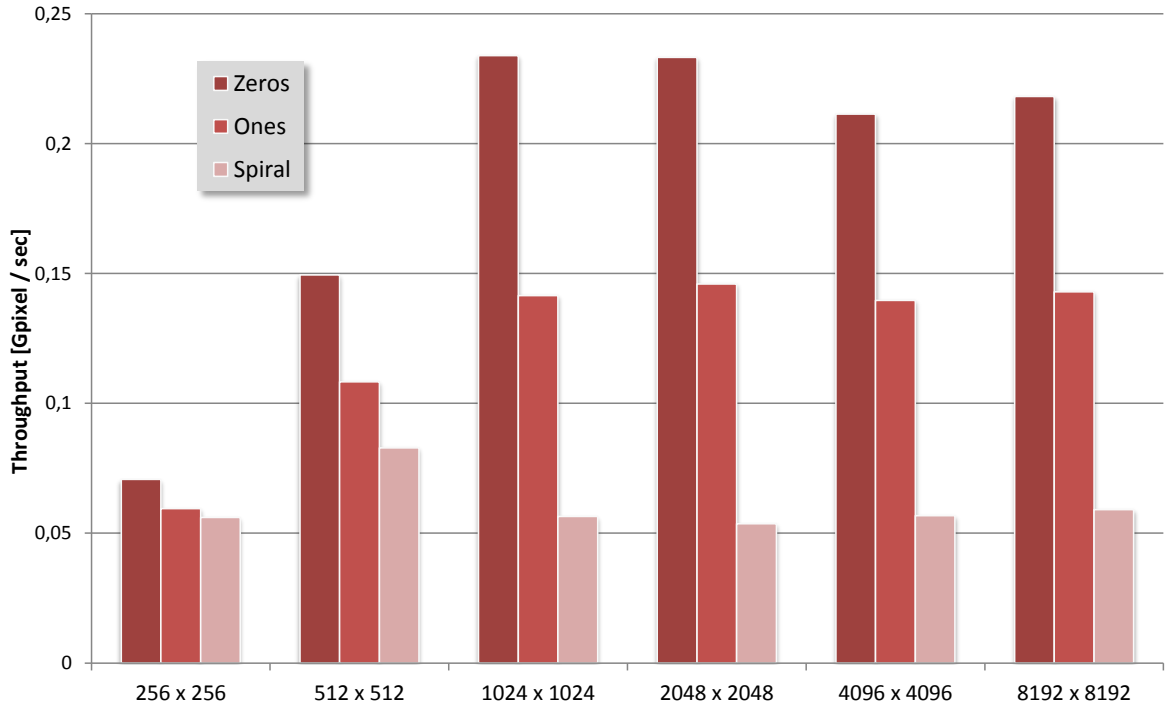


Abbildung 20.9.: Throughput bei Verarbeitung der Datensätze Ones, Zeros und Spiral in verschiedenen Auflösungen bei Ausführung auf einer Core i7 2700k CPU.

Anteile der Subalgorithmen

Für eine genauere Analyse betrachten wir nun die Anteile der einzelnen Kernel an der gemessenen Laufzeit. Dies ist für den Spiral-Datensatz in Abbildung 20.10 dargestellt. Ähnlich wie bei der GPU sind die für das Konturlabeling zuständigen Kernel für einen Großteil der Laufzeit verantwortlich. Allerdings ändern sich die Anteile hier nur unwesentlich. Dies bestätigt die geringe (parallelitätsbedingte) Abhängigkeit von der Datenauflösung weiter, da Kernel unterschiedlicher Parallelität sich gleich verhalten.

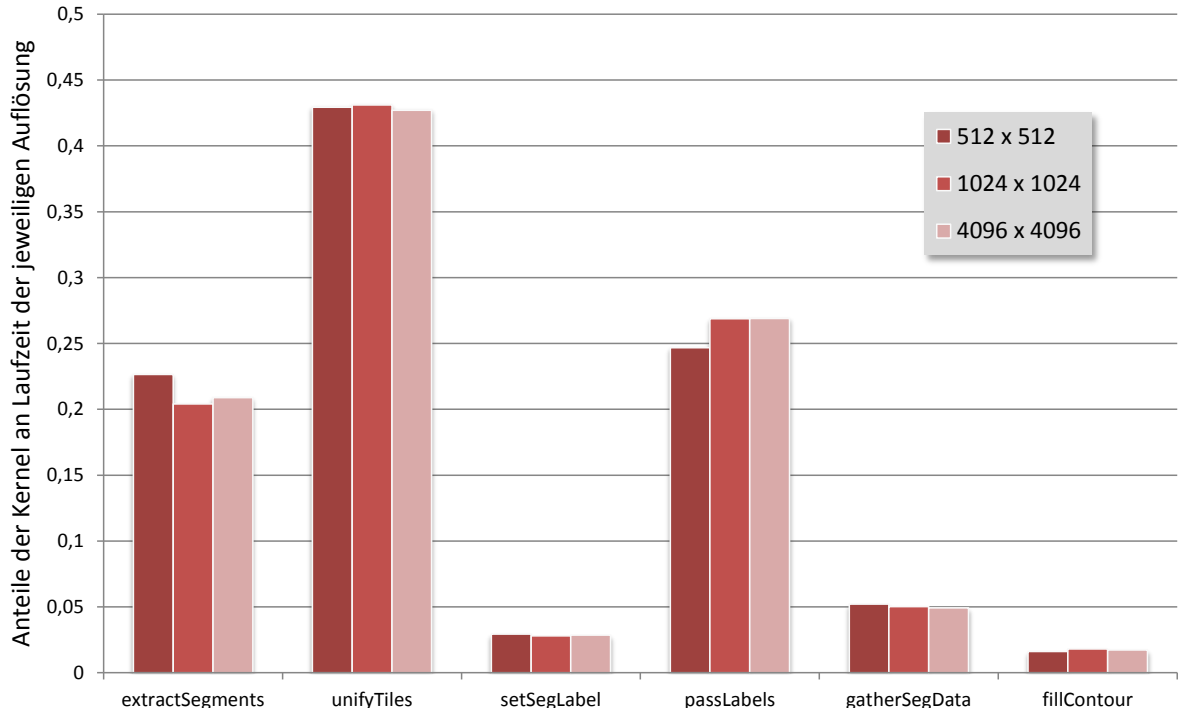


Abbildung 20.10.: Anteile der einzelnen Kernel an der Gesamtlaufzeit in verschiedenen Datenauflösungen. Dabei stellen unifyTiles und passLabels aggregierte Werte der zugehörigen Kernel dar. Datensatz: Spiral, Device: Core i7 2700k

Betrachtung der mergeTilePairs Kernel

Nachfolgend wird das Verhalten der mit dem Schritt `unifyTiles` assoziierten Kernel, `mergeTilePairs_X_F`, `mergeTilePairs_X_B`, `mergeTilePairs_Y_F` und `mergeTilePairs_Y_B`, genauer betrachtet. Deren Throughputs bei Verarbeitung des Spiral-Datensatzes in 4096 x 4096 Pixeln sind in Abbildung 20.11 dargestellt. Beobachtungen:

1. Das Verhalten der Kernel `mergeTilePairs_X_F` und `mergeTilePairs_X_B` einerseits, sowie `mergeTilePairs_Y_F` und `mergeTilePairs_Y_B` andererseits ist jeweils fast identisch. Dies ist ein großer Unterschied zu den Beobachtungen bei Verwendung einer GPU (vergl. Abb. 20.6, S. 201).

2. Der Throughput der `mergeTilePairs_X` Kernel nimmt bei den ersten Aufrufen stark ab und stabilisiert sich dann auf geringem Niveau.

3. Der Throughput der `mergeTilePairs_Y` Kernel bleibt bei den ersten Aufrufen stabil und sinkt erst zum Ende hin ab. Dabei fällt er jedoch nicht deutlich unter 50 Prozent des maximalen Werts und bleibt somit, im Vergleich zu den Kerneln `mergeTilePairs_X` auf der CPU und den Werten der auf der GPU, hoch.

Beobachtung (1) verdeutlicht erneut, dass die über die Aufrufe abnehmende Parallelität bei Ausführung auf der CPU nicht relevant ist. Schließlich verhalten sich gerade nicht diejenigen Paare gleicher Parallelität ähnlich. Gleiches gilt für Beobachtung (3).

Die beobachtete Throughputabnahme (2) muss demnach mit den Kerneln `mergeTilePairs_X` auftretenden Speicherzugriffsmustern zusammenhängen. Diese verarbeiten die Daten Spaltenweise und die Spaltenabstände steigen mit zunehmender Tile-Größe. Dieses Problem ist bereits in der GPU-Fassung aufgetreten.

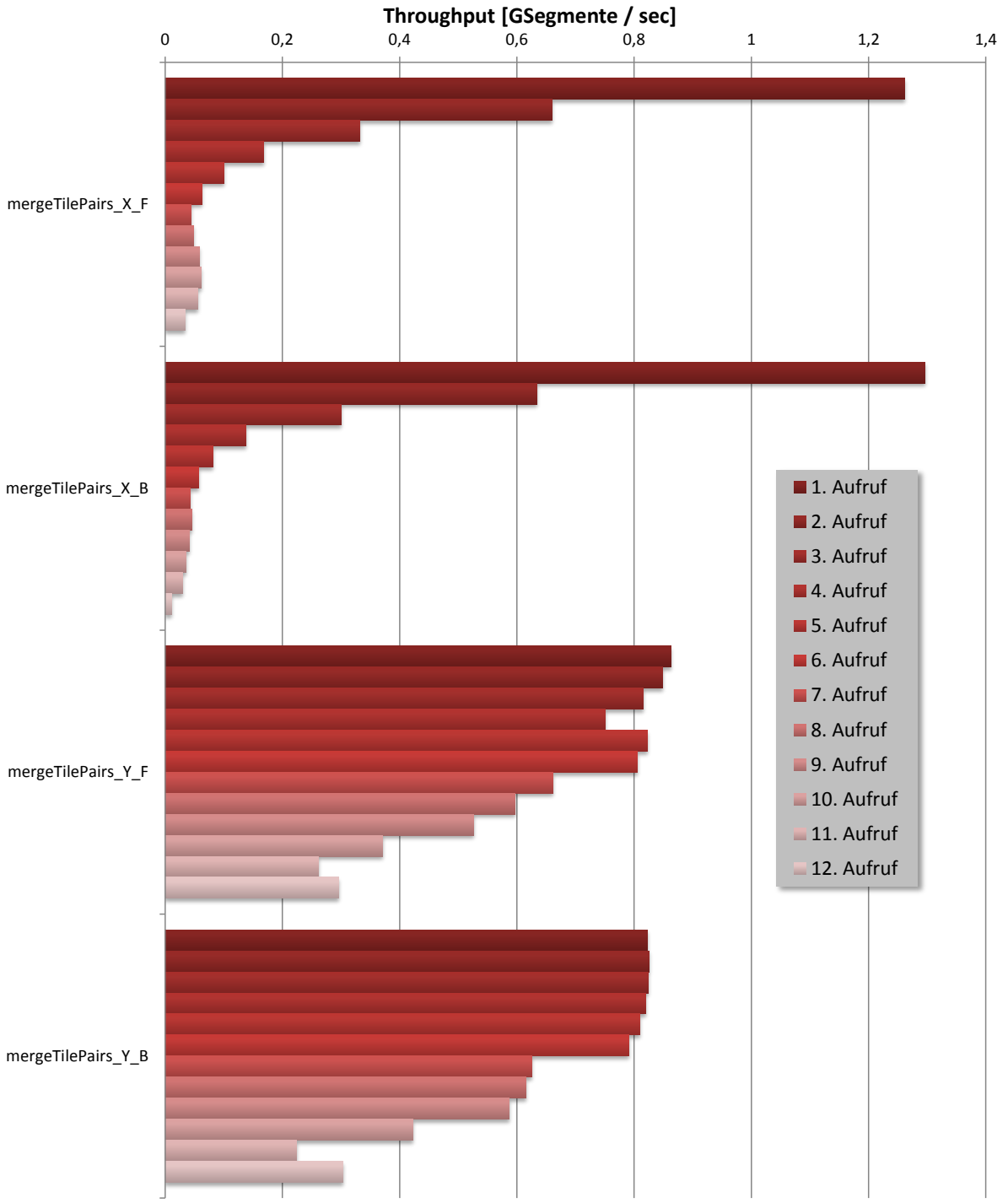


Abbildung 20.11.: Throughputs, angegeben in verarbeiteten Kontursegmenten je Zeiteinheit, der einzelnen Aufrufe der Kernel `mergeTilePairs_X_F`, `mergeTilePairs_X_B`, `mergeTilePairs_Y_F` und `mergeTilePairs_Y_B`, gruppiert nach Kerneltyp. Innerhalb eines Kerneltyps angeordnet gemäß Ausführungsreihenfolge dieses Kernels. Datensatz: Spiral 4096 x 4096, Device: Core i7 2700k

20.3. Variante I(b) - Optimierung des fillContours Kernels für eine GPU

In diesem Abschnitt wird der `fillContours` Kernel für eine GPU optimiert und detaillierter beschrieben. Es handelt sich dabei um die finale Fassung des Kernels zur Ausführung auf einer GPU. CPUs verwenden weiterhin die Ausgangsfassung.

Die Optimierung erfolgt mit dem Ziel, das asynchrone Laden der Daten zu forcieren. Dazu werden für die nächsten zu verarbeitenden Pixel die benötigten Daten angefordert und in Registern hinterlegt. So besteht eine Chance, dass die Daten bereits geladen sind, wenn die entsprechenden Pixel verarbeitet werden, wodurch sich dieser Prozess beschleunigen kann. Ein zu erwartender Nachteil ist der deutlich erhöhte Registerverbrauch je Thread. Da dieser in der Basisversion des Kernels aber gering ist und bedingt durch die Algorithmeneigenschaften die Anzahl startbarer Threads begrenzt ist, erwarten wir dadurch keine nennenswerten Nachteile.

Implementationsdetails

Listing 20.4 zeigt den OpenCL C Code des `fillContours` Kernels, welcher eine unter dieser Zielsetzung entstandene Implementation des in Abschnitt 12.2 ab Seite 136 beschriebenen stack-basierten Fill-Algorithmus darstellt. Die ersten `X` Work-Items beginnen in der oberen Bildzeile und die hinteren `X` beginnen in der unteren Bildzeile. Beide arbeiten sich dann, dem Schema des Algorithmus folgend, sequentiell zur Mitte vor. Dabei werden wiederholt immer erst die Daten der nächsten `PRELOAD_LENGTH` Pixel angefordert und anschließend sequentiell verarbeitet.

Zum Verständnis der Implementation ist zunächst das Datenformat der globalen Datenstrukturen `pixel_Label`, `labelStacks` und `pixel_ioCnt` zu erklären. Die Datenstruktur `pixel_Label` enthält nach Ausführung des `fillContours` Kernels für jeden Pixel dessen Label und damit das Endergebnis. Vor Ausführung des `fillContours` Kernels sind hier Zwischenergebnisse gespeichert. Dies können insbesondere auch die Label der zu den jeweiligen Pixeln gehörenden Kontursegmente sein. Zuvor ausgeführte Kernel müssen sicherstellen, dass alle Einträge von `pixel_Label` folgende Werte bzw. Konstanten enthalten, welche das Verhalten dieses Kernels beeinflussen:

`UNLABELED:` (-1), ist genau dann in den Daten gesetzt, falls der Pixel unklassifiziert ist. Dieser Wert ist demnach endgültig und der Kernel führt für diesen Pixel keine Operation aus. So wird sichergestellt, dass keine unklassifizierten Bereiche im Inneren von Connected-Components versehentlich gefüllt werden.

`UNLABELED_BUT_CLASSIFIED:` (-90) Ein solcher Pixel verfügt noch über kein Label, muss aber eines erhalten. Es ist das oberste des Stacks. Letzterer bleibt unverändert.

`label:` Ein gültiges Label, erkennbar an Werten größer oder gleich null. Es bleibt für den Pixel immer erhalten. Der Wert darf nicht verändert werden, um keine Einzelpixel-Connected-Components zu überschreiben. Unabhängig davon wird im Falle eines gültigen Labels ggf. der Stack manipuliert.

Die Datenstruktur `pixel_ioCnt` enthält die Summe über alle Ein- und Austritte in Bezug auf Connected-Components aller Kontursegmente des betreffenden Pixels. Mögliche Werte sind 0, 1 und 2. Genau dann, wenn der Wert 1 ist, muss der Stack verändert werden. Die Werte 0 und 2 werden nicht explizit unterschieden.

In der Datenstruktur `labelStacks` sind Stacks aller $2 \cdot X$ Work-Items enthalten. Um möglichst günstige Speicherzugriffe erhalten zu können, liegen die nullten Elemente aller Stacks konsekutiv an den Adressen $0 \ldots 2 \cdot X - 1$, die ersten Elemente an den Adressen $2 \cdot X \ldots 4 \cdot X - 1$, usw. Folglich wird das aktuelle Top Element jedes Stacks mit dem globalen Index initialisiert und eine Veränderung des Stacks bewirkt eine Änderung des Top Element Index um $2 \cdot X$. Leere Stacks sind vor Programmausführung so initialisiert, dass sich an der nullten Adresse der Wert `LABEL_STACK_EMPTY` (-3) befindet, welcher insbesondere kein gültiges Label darstellt. Auf diese Weise wird eine ansonsten notwendige (zusätzliche) Fallunterscheidung hinsichtlich leerer Stacks vermieden.

Die Implementation gemäß Implementationsstrategie I mit dem in diesem Abschnitt beschriebenen `fillContours` Kernel für GPUs wird nachfolgend **Impl I GPU** bezeichnet. Sie stellt die finale Fassung der Implementationsstrategie I für eine GPU dar.

```
#define D_STACK_ELEM (2 * X) // Difference betw. subseqent stack elems
#define LABEL_STACK_EMPTY -3 // No valid label <=> stack empty
#define PRELOAD_LENGTH 8

kernel void fillContours(
  global int*    pixel_Label,
  global int*    labelStacks,
  global char*   pixel_ioCnt)
{
  int wi = get_global_id(0);                    // 0 ... 2 * X - 1
  int topIndex = wi;                            // top elem of stack

  char ioCnt[PRELOAD_LENGTH];                   // Preload data ...
  int  label[PRELOAD_LENGTH];                   // ... store

  int topLabel = LABEL_STACK_EMPTY;             // Copy of top stackelem

  int pixel0 = (wi < X) ? wi : wi + N - 2 * X; // First pixel id
  int dPixel = (wi < X) ? X  : -X;              // Diff. betw. subs. ids

  // Process half-column sequentially in steps of PRELOAD_LENGTH
  for(int y = 0; y < Y / 2; y += PRELOAD_LENGTH){
    // 1. Make request for next PRELOAD_LENGTH data entries...
    #pragma unroll
    for(int i = 0; i < PRELOAD_LENGTH; i++){
      ioCnt[i] = pixel_ioCnt[pixel0 + (y + i) * dPixel];
    }
    #pragma unroll
    for(int i = 0; i < PRELOAD_LENGTH; i++){
      label[i] = pixel_Label[pixel0 + (y + i) * dPixel];
    }
    // 2. And execute fill operation on these
    #pragma unroll
    for(int i = 0; i < PRELOAD_LENGTH; i++){
      // If there is exactly one io-seg, modify stack
      if(ioCnt[i] == 1){
        if(topLabel == label[i]){
          // Pop operation
          topLabel = labelStacks[topIndex -= D_STACK_ELEM];
        } else {
          // Push operation
          labelStacks[topIndex += D_STACK_ELEM] = label[i];
          topLabel = label[i];
        }
      }
      // If and only if pixel is unlabeled and classified...
      else if(label[i] == UNLABELED_BUT_CLASSIFIED) {
        // ... it recieves the stack's top label
        pixel_Label[pixel0 + (y + i) * dPixel] = topLabel;
      }
    }
  }
}
```

Listing 20.4: OpenCL C Code für Kernel `fillContours`. Weitere Erläuterungen siehe Text

Tabelle 20.1.: Ressourcenverbrauch und daraus resultierende Limitierung der Parallelität für verschiedene Compute-Capabilities. Die LWS ist in allen Fällen (128, 1, 1).

Compute-Capability	Register	Shared-Memory	Thread-Auslastung
2.0	32	0	67 %
3.0	35	0	75 %
3.5	35	0	75 %

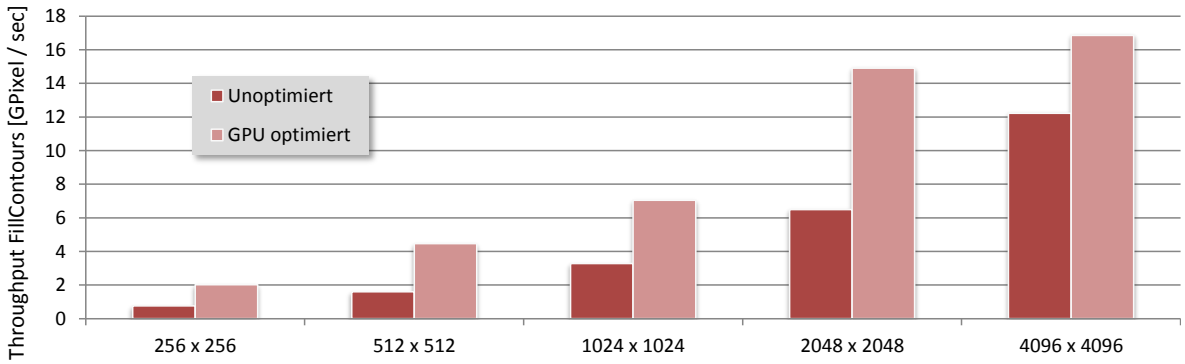


Abbildung 20.12.: Throughputs des `fillContours` Kernels in der unoptimierten und optimierten Variante. Der Throughput bezieht sich nur auf durch diesen Kernel verarbeitete Elemente und nicht den Gesamtalgorithmus. Datensatz: Nested Quads in verschiedenen Auflösungen, Device: GTX 670

Evaluation des fillContours Kernels

Betrachten wir zunächst den Ressourcenverbrauch des `fillContours` Kernel, welcher in Tabelle 20.1 dargestellt ist. Bedingt durch den vergleichsweise hohen Register per Thread Verbrauch kann der Kernel lediglich mit 67 bis 75 Prozent Thread-Auslastung gestartet werden. Dagegen liegt der Registerverbrauch des unoptimierten Kernels bei ca. 10, sodass für jede Compute-Capability alle Threads gestartet werden können. Es kann geschätzt werden, dass dies für die Zielhardware eine annehmbare Einschränkung ist. Beispielsweise könnte die GTX 670 somit, limitiert allein durch den Ressourcenverbrauch, 10752 Threads starten. In einer Auflösung von 4096 x 4096 können jedoch lediglich 8192 Threads beschäftigt werden. Folglich kann dies allenfalls bei noch höheren Auflösungen überhaupt einschränkend wirken. In solch einem Fall kann die Anzahl von vorzuladenden Elementen von acht auf z.B. vier verringert werden, wodurch der Registerverbrauch deutlich sinkt. In dieser Arbeit bleibt die Zahl allerdings in allen Fällen unverändert acht.

Vergleichen wir nun den Throughput des optimierten Kernels mit dem der nicht GPU-optimierten Variante bei Verarbeitung des Nested Quads Datensatzes. Letzterer wird gewählt, da hier erhöhte Anforderungen an den `fillContours` Kernel gestellt werden. Die Ergebnisse sind in Abbildung 20.12 dargestellt. Es lässt sich in allen Auflösungen eine Erhöhung des Throughputs der optimierten Fassung im Vergleich zur Basisversion beobachten. Diese fällt in den geringeren Auflösungen höher aus. Die relative Through-

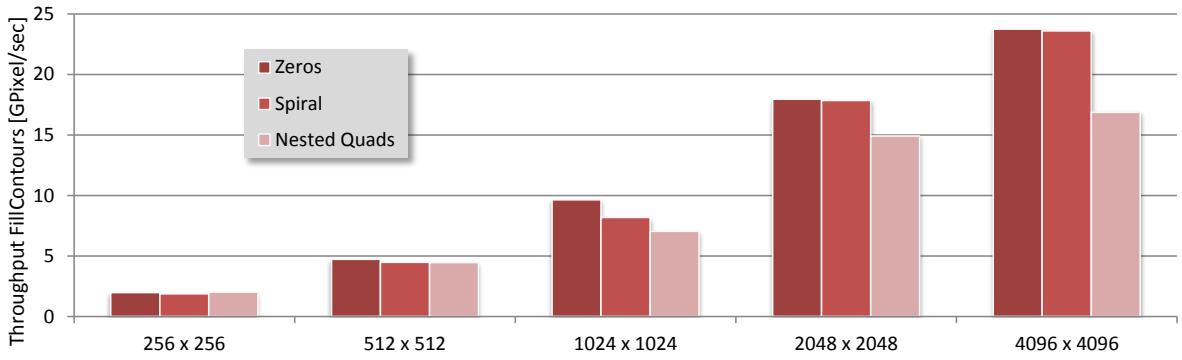


Abbildung 20.13.: Throughputs des GPU-optimierten `fillContours` Kernels bei Verarbeitung der Datensätze Zeros, Spiral und Nested Quads in verschiedenen Auflösungen im Vergleich. Der Throughput bezieht sich nur auf durch diesen Kernel verarbeitete Elemente. Device: GTX 670

putänderung der optimierten Fassung fällt beim Wechsel von der 2048 x 2048 Auflösung zur 4096 x 4096 Auflösung geringer aus, als bei den Auflösungsänderungen darunter. Dies deutet auf eine recht gute Auslastung der GPU hin.

Betrachten wir ergänzend weitere Datensätze bei Verarbeitung durch den optimierten `fillContours` Kernel, dargestellt in Abbildung 20.13. Wie erwartet verursacht die Verarbeitung des Nested Quads Datensatzes einen im Vergleich mit den anderen Datensätzen höheren Aufwand. Auffallend gering ist der Unterschied der Werte bei Verarbeitung der Datensätze Zeros und Spiral. Tatsächlich ist das erwartete Verhalten für diesen Kernel nicht unähnlich:

- Es werden für jeden Pixel je ein Wert von `ioCnt` und `pixel_Label` angefordert.
- Der Stack wird nie (Zeros) oder selten (Spiral, ca. zwei Mal je Spalte) benötigt.
- Es werden, von diesen Stack-Zugriffen abgesehen, keine Daten in den globalen Speicher geschrieben.

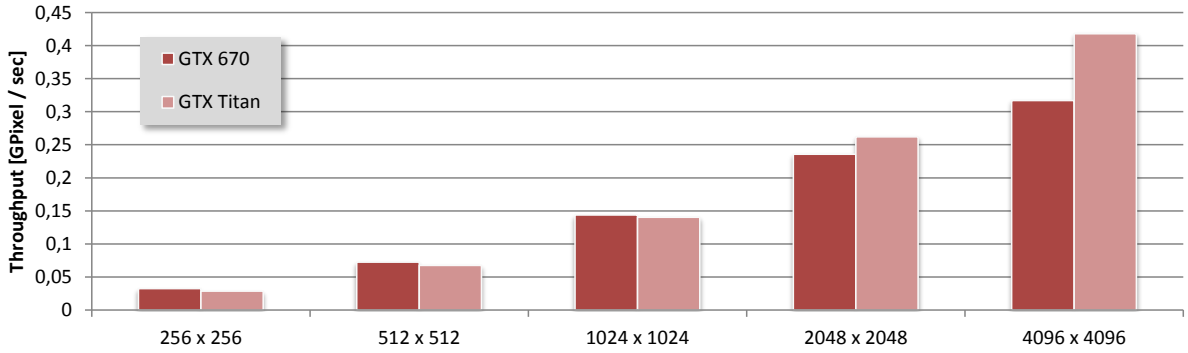


Abbildung 20.14.: Throughput der Implementation Impl I GPU bei Verarbeitung des Spiral-Datensatzes in verschiedenen Auflösungen bei Ausführung auf einer GTX 670 und einer GTX Titan GPU.

Evaluation der Implementation Impl I GPU

Zuletzt wird noch die Skalierung der gesamten Implementation Impl I GPU mit steigender GPU Parallelität bzw. Leistung untersucht. Dies lässt sich nicht beliebig gut isolieren, weil für diese Arbeit keine GPUs identischer Spezifikation aber variabler Parallelität zur Verfügung stehen. Die GTX 670 und die GTX Titan erscheinen aber dafür, natürlich mit Einschränkungen, nicht völlig ungeeignet, da beide auf der Kepler Architektur basieren. Die Titan ist leicht geringer getaktet, verfügt jedoch über die doppelte Anzahl SMs und damit SPs. Bei guter Auslastung ist demnach zu erwarten, dass die GTX Titan sich deutlich von der GTX 670 absetzt. Wir vergleichen nun den mit Impl I GPU ermittelten Throughput bei Verarbeitung des Spiral-Datensatzes in verschiedenen Auflösungen. Die Ergebnisse sind in Abbildung 20.14 dargestellt. Folgende Beobachtungen lassen sich machen:

- Lediglich in der höchsten Auflösung kann sich die GTX Titan von der GTX 670 absetzten
- In den unteren Auflösungen ist der Throughput der GTX 670 sogar minimal höher.

Der Vergleich bestätigt die bisherigen Beobachtungen der Limitierung durch die Parallelität bei Ausführung auf einer GPU. Schließlich benötigt die GTX Titan aufgrund ihrer größeren SP-Zahl eine höhere Parallelität, um ausgelastet zu sein. Ferner hat sich das grundlegende Verhalten im Vergleich zu der Fassung ohne den optimierten `fillContours` Kernel nicht nennenswert geändert. Das war aufgrund dessen geringen Anteils an der Gesamtlaufzeit auch nicht zu erwarten.

20.4. Variante I(c) - Anpassung der Implementation an eine CPU

Wie bereits im Abschnitt 20.2.3 ab Seite 203 beschrieben, ist das Laufzeitverhalten auf einer CPU nicht von der maximalen Parallelität des Algorithmus in seiner ursprünglichen Formulierung abhängig. Dafür fallen die Speicherzugriffe der `mergeTilePairs_X` Kernel ungünstig aus. In diesem Abschnitt wird ein Ansatz geringerer Parallelität beschrieben, welcher auf einer CPU günstigere Speicherzugriffsmuster ermöglicht. Dazu wird der Ablauf der `unifyTiles` (und analog `passLabels`) Kernel folgendermaßen geändert:

Schritt 1 (Kernel unifyX): Verarbeite sequentiell je Zeile alle Segmente der DST-Typen RIGHT und LEFT. Danach sind alle Segmente dieser Typen verarbeitet. Da bis hier keine Segmente der DST-Typen UP und DOWN verarbeitet sind, kann dies für alle Zeilen parallel geschehen.

Schritt 2 (Kernel unifyY): Betrachte anschließend paarweise Zeilen als Tile-Pairs und verarbeite sie entsprechend. Auf eine Unterscheidung zwischen Hin- und Rückrichtung wird dabei verzichtet. Dies wird $log_2(Y)$ Mal wiederholt, bis nur noch ein Tile übrig bleibt.

Auf diese Weise verarbeitet jedes Work-Item immer konsekutiv im Speicher liegende Pixel nacheinander. Zusätzlich liegen im ersten Schritt auch Head und Tail garantiert in der gleichen Zeile. Dies mag mit einer gewissen Wahrscheinlichkeit ebenfalls zu günstigeren Zugriffen führen. Im zweiten Schritt liegen Head und Tail immer garantiert in der unteren oder oberen Zeile eines Tiles. Auch dies begünstigt möglicherweise die Zugriffe. Beides ist im Fall der vormals verwendeten rechteckigen Tiles nicht garantiert.

Die Throughputs der Kernel einer solchen Implementation bei Verarbeitung der Spirale in 4096 x 4096 Pixeln sind in Abbildung 20.15 dargestellt.

Beobachten lässt sich ein nahezu unverändertes Verhalten der Aufrufe des Kernels `unifyY` im Vergleich mit den entsprechenden Aufrufen der Kernel `mergeTilePairs_Y_F` und `mergeTilePairs_Y_B`, welche er ersetzt (vergl. Abb. 20.11, S. 207). Die absoluten Werte sind bei dem neuen Kernel `unifyY` geringfügig höher.

Der Throughput des Kernels `unifyX` ist sehr stark angestiegen im Vergleich mit den entsprechenden Kerneln `mergeTilePairs_X_F` und `mergeTilePairs_X_B`. Er liegt nun sogar deutlich oberhalb der Werte des Kernels `unifyY`. Letzteres lässt sich möglicherweise durch die besonders hohe Datenlokalität erklären. Ein Segment, sein Nachfolger sowie Head und Tail liegen garantiert in einer einzigen Zeile. Außerdem sind alle Segmente unverarbeitet, dementsprechend liegen Head und Tail mit einer höheren Wahrscheinlichkeit noch lokal vor. Zuletzt können sich in einer Zeile kaum komplexe Figuren bilden, da maximal zwei Konturen aneinander vorbeikommen.

Das oben besprochene Schema führt zu einem deutlich verbesserten Laufzeitverhalten der Implementation auf einer CPU. Es lassen sich noch weitere kleinere Optimierungen durchführen. Dabei wird die Parallelität verringert und der Anteil sequentieller Berechnungen weiter erhöht. Sie bleibt aber solange wie möglich größer oder gleich der Anzahl

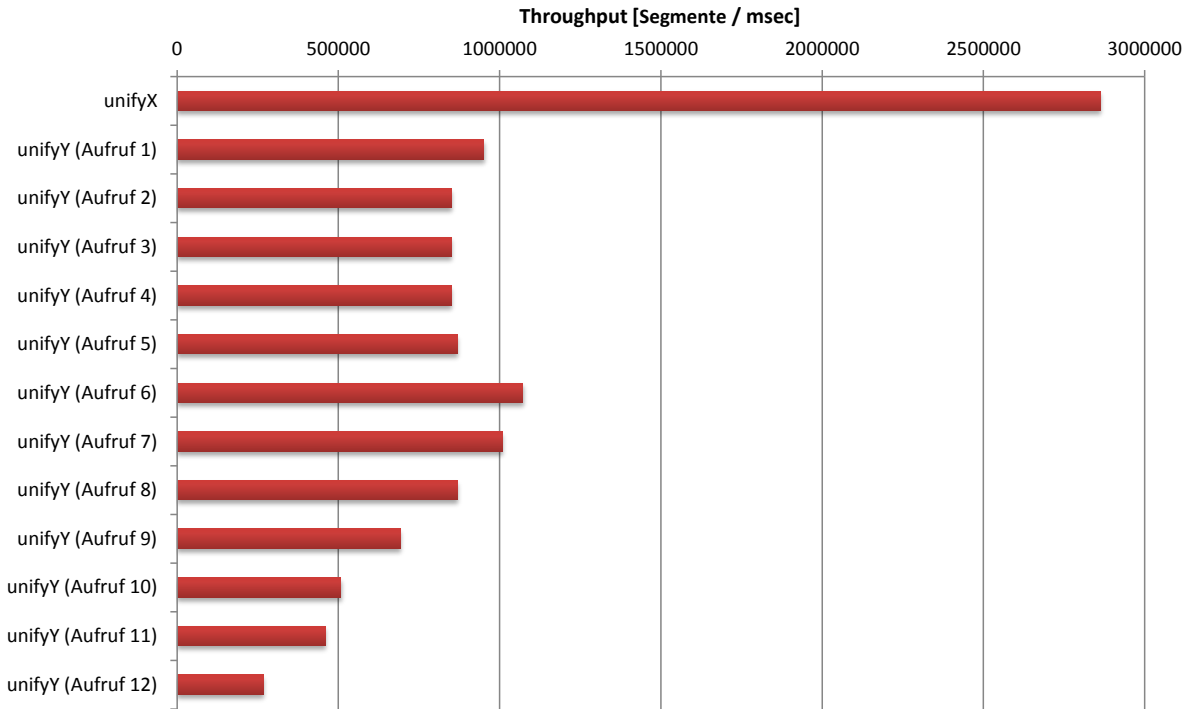


Abbildung 20.15.: Throughputs, angegeben in verarbeiteten Kontursegmenten je Zeiteinheit, des Kernels `unifyX` und der einzelnen Aufrufe des Kernels `unifyY`, angeordnet in Ausführungsreihenfolge. Datensatz: Spiral 4096 x 4096, Device: Core i7 2700k

ausführbarer Hardware-Threads der CPU. Auch wird die Anzahl der Durchgänge durch die Daten verringert, welche ursprünglich durch die Erhöhung der Parallelität motiviert war. Die Anpassungen im Einzelnen:

- Kernel `unifyY` verarbeitet nicht immer zwei Zeilen, sondern gleich mehrere sequentiell. Erst gegen Ende wird auf das Tile-Pair-Schema gewechselt.
- Das Vergeben der Konturlabel wird von den Kerneln `unifyX` und `unifyY` übernommen. Kernel `setSegLabel` entfällt.
- Analog wird die Aufgabe des Kernels `gatherSegInfo` durch `fillContours` übernommen.
- Der Kernel `extractSegments` verarbeitet einzelne Zeilen sequentiell.
- Alle Kernel erhalten eine LWS (1, 1, 1)

Diese Änderungen verbessern das Laufzeitverhalten auf einer CPU weiter, ohne aber, im Gegensatz zur Einführung der Kernel `unifyX` und `unifyY`, das grundlegende Verhalten zu ändern. Zuletzt sei noch darauf hingewiesen, dass jede einzelne der in diesem Abschnitt eingeführten Optimierungen auf einer GPU das Laufzeitverhalten verschlechtert.

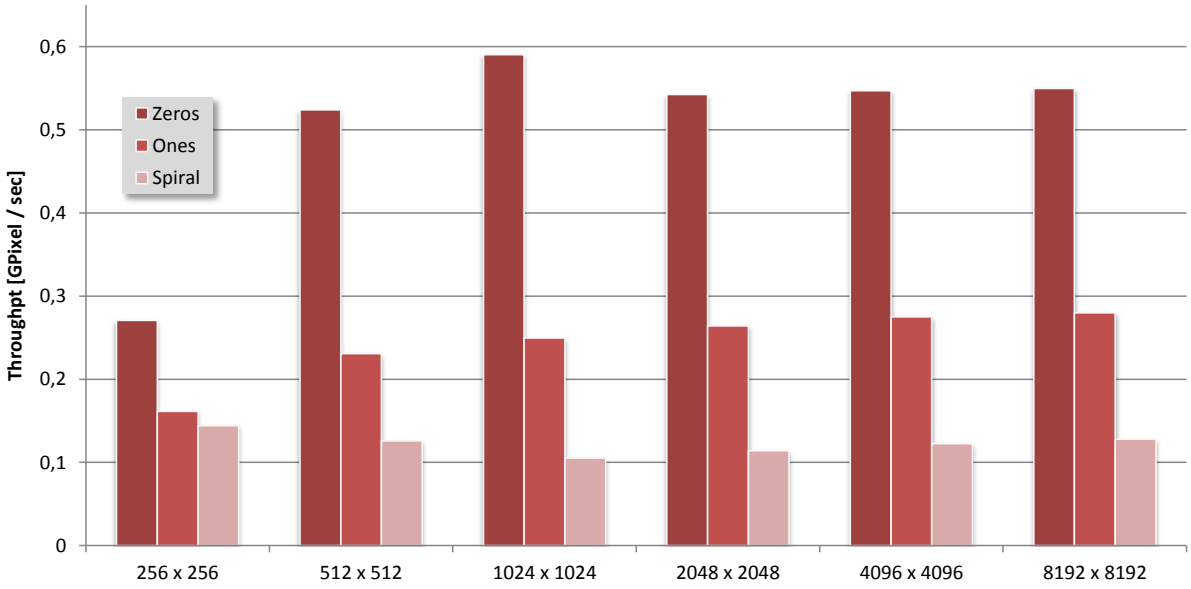


Abbildung 20.16.: Throughput der CPU optimierten Fassung bei Verarbeitung der Datensätze Ones, Zeros und Spiral in verschiedenen Auflösungen bei Ausführung auf einer Core i7 2700k CPU.

Evaluation der CPU-optimierten Fassung

Zur Evaluation der CPU-optimierten Fassung werden zunächst die Throughputs für die Datensätze Zeros, Ones und Spiral in Abhängigkeit von der Datenauflösung ermittelt werden. Die Ergebnisse sind in Abbildung 20.16 dargestellt. Die Beobachtungen des relativen Verhaltens hinsichtlich Datenauflösungen und der Unterschiede zwischen den Datensätzen entsprechen im Wesentlichen denen, die bereits bei der unoptimierten Fassung gemacht wurden (vergl. Abb. 20.9 S. 204) Insbesondere bleibt der Throughput ab der Auflösung 1024 x 1024 etwa konstant.

Dagegen haben sich die absoluten Werte deutlich geändert, wie in der direkten Gegenüberstellung des Throughputs zur unoptimierten Fassung in Abbildung 20.17 ersichtlich ist. In den höheren Auflösungen liegt der Throughput der CPU-optimierten Fassung um mehr als Faktor zwei über dem der Ausgangsfassung. Nachfolgend wird ausschließlich die CPU-optimierte Fassung verwendet.

Betrachten wir nun die Skalierung in Abhängigkeit der Anzahl verwendeter CPU-Kerne. Dazu wird im Bios die Intel Turbo Boost Technik, welche bei Verwendung weniger Kerne einzelne Kerne dynamisch übertaktet, deaktiviert. Dies passiert, um die Messungen nicht zu verfälschen. Zusätzlich wird anschließend SMT, von Intel als Hyper Threading bezeichnet, deaktiviert. Danach werden alle Kerne bis auf einen im Bios deaktiviert, schrittweise wieder reaktiviert und die Messung wiederholt.

Die Ergebnisse bei Verarbeitung der 4096 x 4096 Spirale auf einem Intel Core i7 5960X sind in Abbildung 20.18 dargestellt.

Dabei wird jeweils der ermittelte Throughput durch die Anzahl aktivierter Kerne geteilt. Im Falle von ein bis vier aktivierten Kernen bleibt der Throughput je Kern etwa

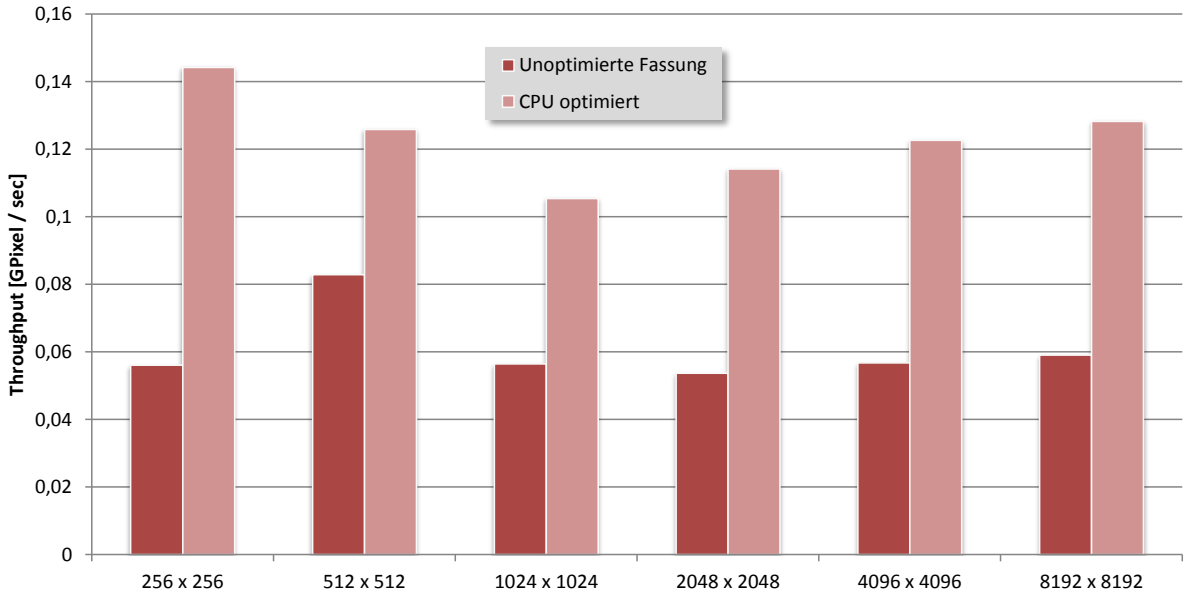


Abbildung 20.17.: Gegenüberstellung des Throughputs zur unoptimierten Fassung bei Verarbeitung des Spiral-Datensatzes in verschiedenen Auflösungen bei Ausführung auf einer Core i7 2700k CPU.

konstant, folglich steigt der Throughput linear an. Bei Aktivierung weiterer Kerne sinkt der der Throughput je Kern leicht, der gesamte Throughput steigt somit nicht mehr ganz linear. Werden zusätzlich bei acht verwendeten Kernen noch SMT und Intel Turbo Boost aktiviert, steigt der Throughput jedes Mal leicht.

20.5. Fazit

Beginnen wir mit der Bewertung der in Abschnitt 20.4 beschriebenen Implementation für CPUs. Ihre Laufzeit skaliert für höhere Auflösungen linear mit der Problemgröße, die Parallelität reicht für aktuelle CPUs aus und das Datenverarbeitungsschema trägt den gewünschten Speicherzugriffen einer CPU Rechnung. Damit wird den Anforderungen einer CPU im allgemeinen Rechnung getragen, allerdings sind weitere Optimierungen für CPUs, etwa für bestimmte Cache-Größen denkbar. Das liegt aber nicht im Fokus dieser Arbeit. Diese Fassung, welche die Endfassung für eine CPU in dieser Arbeit darstellt, ist somit insgesamt für eine CPU gut geeignet und ihr Optimierungsgrad kann als gering eingeschätzt werden.

Dagegen sind die Ergebnisse der Implementationen für GPUs insgesamt unbefriedigend. Aufgrund der in Teilen zu geringen Parallelität ist die Auslastung einer GPU gerade in geringeren Auflösungen schlecht und leistungsstärkere GPUs führen nicht in gewünschtem Umfang zu einem erhöhten Throughput. Außerdem treten auch hier teilweise sehr ungünstige Speicherzugriffsmuster auf. Diese lassen sich auch nicht ohne negative Nebenwirkungen durch ein weniger paralleles Verarbeitungsschema auflösen, wie

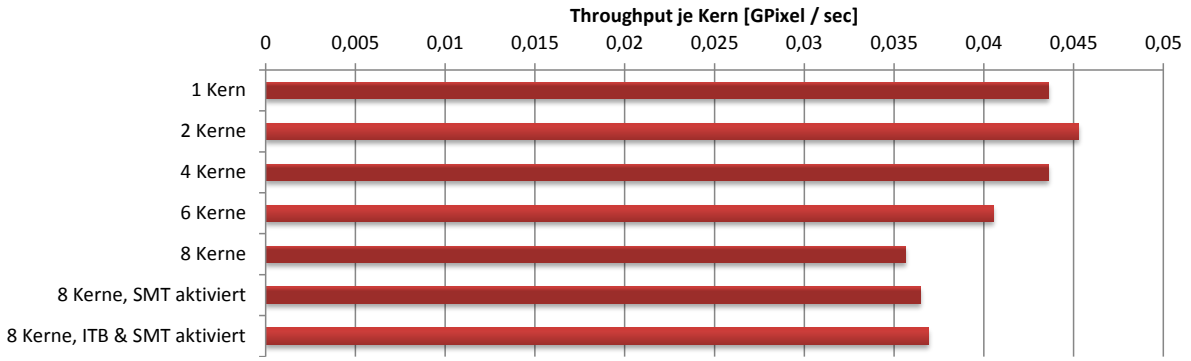


Abbildung 20.18.: Ermittelte Throughputs geteilt durch Anzahl aktivierter Kerne bei Verarbeitung des 4096 x 4096 Spiral-Datensatzes bei Ausführung auf einer Core i7 5960X CPU mit variabler Anzahl im Bios aktivierter Kerne. Dabei sind SMT und Intel Turbo Boost (ITB) deaktiviert, sofern nicht anders angegeben.

bei der CPU Fassung.

Der Optimierungsgrad kann als gering eingeschätzt werden, da zentrale Konzepte, wie das Shared-Memory komplett ungenutzt bleiben. Insgesamt müssen für GPUs andere Lösungen gefunden werden. Die Implementation bietet jedoch einen ersten Referenzwert für konturbasiertes Connected-Component-Labeling auf GPUs.

Ausgenommen davon ist der in Abschnitt 20.3 beschriebene `fillContours` Kernel. Dessen Parallelität ist, zumindest in höheren Auflösungen, für gängige GPUs ausreichend. Da der Kernel die GPU nicht explizit zu ungünstigen Speicherzugriffsmustern zwingt und jeder Pixel genau einmal betrachtet wird, ist sein Anteil an der Gesamtlaufzeit des Algorithmus gering. Es erscheint somit unwahrscheinlich, dass ein auf Sortieren basierter Ansatz in den höheren Auflösungen bessere Ergebnisse liefert. Die Implementation trägt zumindest dem für GPUs günstigen asynchronen Laden der Daten Rechnung. Der Optimierungsgrad kann damit als mittel eingestuft werden. Der Kernel wird deswegen nachfolgend immer für GPUs verwendet.

Kapitel 21.

Implementationsstrategie II: Massiv parallel

Die Implementationsstrategie I des vorherigen Kapitels bereitet auf einer GPU Probleme, da einige Teile nicht in einer für eine GPU ausreichenden Parallelität ausführbar sind. Dieser Effekt tritt bei geringen Datenauflösungen verstärkt auf.

In diesem Kapitel wird deshalb eine alternative, auf der Pointer-Jumping-Technik für das Contour-Labeling basierende Implementation (siehe Abschnitt 10.3.1, ab Seite 115), untersucht. Diese Elementaroperation ist ausgesprochen simpel und kann in voller Datenparallelität ausgeführt werden. Allerdings führt der Ansatz asymptotisch mehr Operationen aus als Impl I GPU, auf die Work-Optimalität wird hier somit verzichtet.

Erfahrungen aus Kapitel 20 deuten darauf hin, dass die Verarbeitung von Datensätzen mit vielen Leereinträgen nur um einen geringen Faktor schneller ist, als im Falle eines schwierigen Datensatzes, wie der Spirale. Deshalb werden in diesem Ansatz die existierenden Kontursegmente zunächst in eine dicht gepackte Datenstruktur überführt. Anschließend müssen die aufwendigen Konturlabeling-Techniken lediglich darauf angewendet werden. Davon kann eine deutliche Verbesserung im Falle von Datensätzen, welche über einen geringen Konturanteil verfügen, erhofft werden. Das von der Implementationsstrategie I verwendete Verarbeitungsschema der Konturliniensegmente hängt eng mit dem Pixelgitter zusammen. Dagegen ist bei der Pointer-Jumping-Technik die Zugehörigkeit der Segmente zu einem bestimmten Pixel (und damit Tile) unbedeutend. Somit ist das Umstrukturieren ungleich einfacher als bei dem ersten Ansatz.

Unter diesen Gesichtspunkten wird eine neue Implementation, `Impl II`, erstellt. Gegenstand dieses Kapitels ist dann zuerst deren experimentelle Untersuchung hinsichtlich des positiven Einflusses der höheren Parallelität, gerade bei geringen Problemgrößen und erhöhter Parallelität der ausführenden GPU.

Außerdem wird untersucht, ob die superlinear mit der Problemgröße steigende Operationsanzahl im Falle realer Problemgrößen einen negativen Einfluss hat.

Zuletzt wird überprüft, ob das Überführen in eine dichtgepackte Datenstruktur einerseits wie erwünscht im Falle dünnbesetzter Daten zu einer Verbesserung führt und andererseits, ob dieses Vorgehen im Falle nicht dünnbesetzter Daten sogar schadet. Auch dies wird experimentell evaluiert und mit Impl I GPU verglichen.

21.1. Scan und Compact

Die Ziele des Kapitels machen es erforderlich, interessante Elemente aus einer dünnbesetzten Datenstruktur in eine Dichtbesetzte zu überführen, welche nur noch diese Elemente enthält, um deren weitere Verarbeitung zu vereinfachen. Dazu werden die Algorithmen Scan (Kapitel 7.2, ab Seite 36) und Compact (Kapitel 7.4, ab Seite 39) benötigt. Zusätzlich bietet die Anwendung von Scan und Compact weitere Vorteile:

- Der Scan-Algorithmus liefert (auch) die Anzahl existierender bzw. noch verbleibender Elemente. Diese kann zur Abschätzung genutzt werden, um ggf. die Anzahl auszuführender Schritte zu reduzieren.
- In den Kerneln zur Segmentverarbeitung für dünnbesetzte Datenstrukturen muss vor jeder Operation abgefragt werden, ob das jeweilige Segment überhaupt existiert. Dafür gibt es eigene Datenstrukturen im globalen Speicher. Im Falle einer dichtgepackten Datenstruktur genügt dagegen der Vergleich des Work-Item-Index mit der Elementanzahl, einer Konstante. Gerade wenn viele kleine Kernel gestartet werden müssen, die jeweils wenige Operationen ausführen, ist dies von Vorteil.

Es folgt ein kurzer Überblick über bisherige Ansätze der Implementation des Scan-Algorithmus für GPUs.

Eine frühe Implementation eines Scan-Algorithmus basierend auf dem nicht work-optimalen Algorithmus von Hillis und Steele [HS86] für eine GPU stammt von Horn [Hor05] und verwendet das Brook Framework [BFH+04].

Die erste GPU-Implementation mit linearem Gesamtaufwand wurde ein Jahr später von Sengupta veröffentlicht [SLO06]. Sie basiert auf dem work-optimalen Algorithmus von Blelloch [Ble90] und ist noch in OpenGL implementiert. Sengupta vermerkt einen Speedup um Faktor vier im Vergleich mit [Hor05].

Im Jahre 2007 veröffentlichten Harris et al. [HSO07] eine work-optimale Scan-Implementation mit der API Cuda. Diese ist für Nvidia GPUs der Compute-Capability 1.x optimiert. Die Autoren vermelden deutliche Speedups im Vergleich sowohl mit [Hor05] als auch mit CPU-basierten Scan-Implementationen.

Ein Jahr später schlagen Dotsenko et al. [DGS+08] einen Matrix basierten Scan mit gleichen theoretischen Eigenschaften vor. Wie Harris [HSO07] verwenden sie ebenfalls Cuda und optimieren für die gleiche GPU. Ihr Ansatz ist im Falle unsegmentierter Daten etwas schneller (ca. 25 Prozent). Zusätzlich stellen sie eine für segmentierte Daten optimierte Variante vor. Diese ist bei solchen Daten um bis zu Faktor 10 schneller als bisherige Ansätze.

Ferner veröffentlichten Billeter et al. [BOA09] einen nicht work-optimalen Scan in Verbindung mit Stream-Compaction für GPUs mit Cuda. Sie argumentieren, dass work-optimale Ansätze diese Eigenschaft erreichen, indem nicht alle Threads in jedem Schritt eine Operation ausführen. Das wiederum hat aufgrund der SIMD-artigen Arbeitsweise einer GPU in der Praxis keinen Nutzen [BOA09]. Eine an Hillis und Steele [HS86] angelehnte Implementation aber habe den Vorteil von $log(N)$ anstelle von $2 \cdot log(N)$ sequentiellen Ausführungsschritten.

Im Jahre 2011 veröffentlichen Harris und Garland [HG11] eine für die Fermi Architektur optimierte Variante von [HSO07]. Sie verwenden dabei einige neue, mit Compute-Capability 2.0 eingeführte, Funktionen. Allerdings sind diese größtenteils unter OpenCL für aktuelle Nvidia Grafikkarten nicht verfügbar. Die Autoren zeigen, dass sich damit Laufzeit einzelner Binäroperationen, wie der Binary-Reduction, halbieren lassen.

Die Optimierung des Scan-Algorithmus für GPUs bleibt weiter Forschungsgegenstand. So werden in neueren Veröffentlichungen weitere (geringfügige) Speedups publiziert [HH13]. Diese sind teilweise auch bedingt durch Nvidia Kepler spezifische Features, wie read-only Data-Cache und Shuffle-Instructions, etwa in[DAD14].

Die Neueren der obengenannten Ansätze können derzeit im Falle von OpenCL nicht zusammen mit einer Nvidia GPU verwendet werden. Deshalb sind im Rahmen dieser Arbeit die Ansätze von Harris et al. [HSO07] und Billeter et al. [BOA09] basierend auf den Erläuterungen in den jeweiligen Papern mit OpenCL nachimplementiert und verglichen worden. Es zeigten sich dabei keine gravierenden Unterschiede. Es wurde dann der an Harris et al. [HSO07] angelehnte Ansatz weiterverwendet und weiterhin etwas an die Fermi Architektur angepasst, vor allem hinsichtlich der Bank-Conflict-Vermeidung. Anders als das Original führt diese Implementation einen Scan über ein `char` Array durch und speichert die Ergebnisse in einem `int` Array.

21.2. Finden des Konturlabels

In der bisherigen Formulierung findet der Konturlabeling-Algorithmus ein Label, welches der minimalen Adresse aller Segmente einer Linked-List entspricht. Diesen Wert nennen wir ab jetzt `minId`. Für die Unterscheidung innerer- und äußerer Konturen wird, wie in Kapitel 11 ab Seite 127 beschrieben, die minimale Segmenthöhe der jeweiligen Linked-List benötigt. In der Praxis ist es sinnvoll, beide, sowohl hinsichtlich der Berechnungen als auch die Datenstrukturen betreffend, zu vereinigen. Eine Möglichkeit wird nachfolgend erläutert.

Für ein eindeutiges Label genügt es, einem Segment die Pixeladresse und nicht die Segmentadresse zuzuweisen, weil es in einem Pixel maximal eine äußere Kontur geben kann. Schließlich kann ein Pixel nur zu maximal einer Connected-Component gehören. Dadurch können zwei Bits zur Kodierung gespart werden. Das wird sich später noch als hilfreich erweisen. Um zusätzlich die Höheninformation des Segments festzulegen, initialisieren wir `minId` folgendermaßen:

$$\begin{aligned}\text{Repräsentiert (unter anderem) obere Kante} : minId &\leftarrow 2 \cdot pixelAdresse \\ \text{Sonst} : minId &\leftarrow 2 \cdot pixelAdresse + 1\end{aligned}$$

Ist für alle Elemente das Minimum von `minId` über die gesamte Linked-List bestimmt, kann gemäß den Makros aus Listing 21.1 bestimmt werden, ob das Label zu einer inneren oder äußeren Kontur gehört.

```
// Apply after contour labeling is finished
// Note: & 1 is the same as mod 2
#define OUTER(minId) ((minId & 1) == 0)
#define INNER(minId) (minId & 1)
```

Listing 21.1: Makros zum Testen der Zugehörigkeit zu einer inneren bzw. äußeren Kontur

Dies entspricht dem in Kapitel 11 erläuterten Test. Falls `OUTER(minId)` 1 liefert, kann `minId` als Label verwendet werden.

21.3. Implementationsdetails

Die Implementation des Connected-Component-Labelings, basierend auf der Basic-Pointer-Jumping-Technik, besteht aus folgenden Schritten, angegeben in Ausführungsreihenfolge:

extractSegments Dieser Schritt entspricht im Wesentlichen dem aus der Implementationsstrategie I. Lediglich einige der zu initialisierenden Datenstrukturen weichen ab.

Scan / Compact Überführt existierende Segmente in dichtbesetzte Datenstrukturen, wie im Abschnitt 21.1 zuvor skizziert.

Pointer-Jump Contour-Labeling Wendet auf alle Segmente der dichtbesetzten Datenstrukturen die Pointer-Jumping-Technik an und findet Label für diese. Auf die Implementation wird im nachfolgenden Abschnitt eingegangen.

Wiederherstellung Gefundene Label werden an die Ursprungspositionen der zugehörigen Segmente zurück kopiert.

gatherSegInfo Dieser Schritt entspricht im Wesentlichen dem aus der Implementationsstrategie I. Hier wird der neue Test auf Zugehörigkeit zu einer inneren oder äußeren Kontur ausgeführt

fillContours Es wird unverändert der `fillContours` Kernel aus Abschnitt 20.3 verwendet.

21.3.1. Datenstrukturen

Die verwendeten Datenstrukturen stimmen zum Großteil mit denen der Implementation I, wie in Abschnitt 20.1 ab Seite 187 beschrieben, überein. Ausnahmen sind:

minId Ersetzt, gemäß des in Abschnitt 21.2 beschriebenen Schemas, die Datenstrukturen `label` und `minDepth`. Datentyp ist `int`.

suc Verweis auf die Adresse des jeweils aktuellen Nachfolgesegments. Benötigt daher, im Gegensatz zu Implementation I, den Datentyp `int` statt `char`.

originalId Die ursprüngliche Adresse eines Kontursegments. Datentyp: `int`.

head / tail / status Diese Datenstrukturen werden nicht benötigt.

Damit werden für die Kontursegmente selbst zunächst weniger Daten benötigt. Allerdings existieren weitere Hilfsdatenstrukturen für den Scan und aufgrund der verwendeten Doublebuffer-Technik liegen die Datenstrukturen für Kontursegmente doppelt vor. Insgesamt ergibt sich mit 102 Bytes per Pixel ein gestiegener Speicherverbrauch.

21.3.2. Implementation der Pointer-Jump-Technik

Nach Anwendung von Scan/Compact ist die Anzahl der zu verarbeitenden Kontursegmente, `length`, bekannt und in den Datenstrukturen `minId` und `suc` liegen entsprechend viele Einträge. Allein in Abhängigkeit von der Datenauflösung muss der Pointer-Jump $\lceil log_2(N) \rceil$ Mal auf alle Elemente angewendet werden. Diese Abschätzung kann nun in $\lceil log_2(length) \rceil$ geändert werden, mit: $\lceil log_2(length) \rceil \leq \lceil log_2(N) \rceil$.

Der Pseudocode-Abschnitt 25 gibt einen Überblick über die Implementation und sollte zusammen mit den Listings 21.2 und 21.3 für die benötigten Kernel `minThisSuc` und `pointerJump` gelesen werden. Eine Pointer-Jump-Phase ist in je eine Anwendung von beiden Kerneln auf alle existierenden Daten aufgeteilt. Zunächst berechnet immer der Kernel `minThisSuc` für alle Elemente i: $minId(i) \leftarrow min(minId(i), minId(suc(i)))$.

Algorithm 25: Simplified pseudo-implementation of Pointer-Jumping technique

```
// Kernels minThisSuc and pointerJump given in Listings 21.2
// and 21.3
// - length = number of existing elements
// - Kernels are properly initialized
// - Global data structures are initially bound to all kernels
// - Steps till Scan / Compact executed
int length256 = length rounded up to next integer times 256;
minThisSuc.setArg (2, length);
pointerJump.setArg (2, length);
// Apply Pointer-Jumping technique
Do ⌈log2(length)⌉ − 1 Times
    // Phase X
    enq (minThisSuc, GWS(length256, 1, 1), LWS(256, 1, 1);
    enq (pointerJump, GWS(length256, 1, 1), LWS(256, 1, 1);
    Do: swap and rebind suc_Old and suc_New;
End
// Last execution of pointerJump not necessary
enq (minThisSuc, GWS(length256, 1, 1), LWS(256, 1, 1);
```

Er lässt insbesondere den Pointer suc auf das jeweils nachfolgende Element unverändert. Grund dafür ist die, anders als im PRAM-Algorithmus, nicht synchron parallele Ausführung für alle Elemente. So wird sichergestellt, dass kein Element übersprungen wird. Auch kann so nie erst der Pointer-Jump und danach die Minimumsberechnung ausgeführt werden, was ebenfalls falsch wäre. Nicht verhindert wird der Fall, dass das Nachfolgeelement bereits das Minimum von sich und dessen Nachfolger abgespeichert hat. Dieser Fall ist aber unkritisch, denn es hat sich lediglich die Reichweite erhöht, ohne ein Element zu überspringen.

Durch die Beendigung des Kernels an dieser Stelle wird ein globaler Synchronisationspunkt eingefügt. Erst danach wird der Kernel `pointerJump` ausgeführt, welcher die zweite Hälfte eines Pointer-Jumps, nämlich die Berechnung $suc(i) \leftarrow suc(suc(i))$ ausführt. Es wird dabei für suc eine Doublebuffer-Technik, mit suc_Old und suc_new, verwendet. Andernfalls besteht Gefahr, einzelne Elemente zu überspringen.

Dies beides wird nur $\lceil log_2(length) \rceil - 1$ Mal ausgeführt, da es in der letzten Phase genügt, nur den Kernel `minThisSuc` anzuwenden.

Von beiden Kerneln werden jeweils nicht genau `length` Work-Items erzeugt, sondern die Problemgröße wird auf die nächstgrößere *gute* Zahl, z.B. ein ganzzahliges Vielfaches von 256 aufgerundet. Diese Vorgehensweise ist typisch für Kernel, welche datenabhängige Problemgrößen verarbeiten. Andernfalls ist eine Aufteilung in Work-Groups, welche eine sinnvolle Nutzung der Ressourcen einer GPU ermöglicht, nicht garantiert. Das ist natürlich ganz besonders schwerwiegend, wenn `length` eine Primzahl ist.

```
// Computes minId(i) <- min(minId(i), minId(suc(i))) for each i
kernel void minThisSuc(
 global int*  suc,
 global int*  minId,
 const  int   length)
{
  int t = get_global_id(0); // t(his)

  if(t >= length) return; // process only existing elements

  int s       = suc  [t]; // s(uc)
  int minID_t = minId[t];
  int minID_s = minId[s];

  if(minID_s < minID_t) // Write min(minID_s, minID_t) if necessary
    minId[t] = minID_s;
}
```

Listing 21.2: OpenCL C Code für die erste Hälfte eines Pointer-Jumps

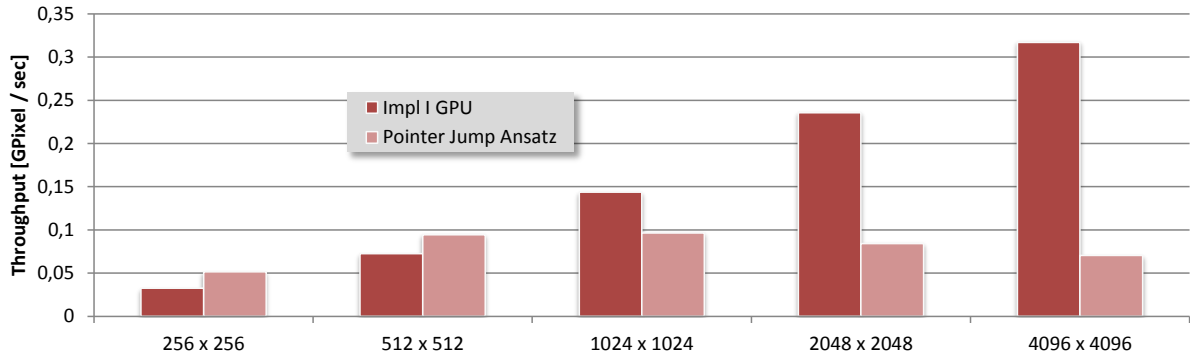
```
// Computes suc(i) <- suc(suc(i)) for each i
kernel void pointerJump(
 global int* suc_Old,
 global int* suc_New,
 const  int  length)
{
  int t = get_global_id(0);
  if(t >= length) return;           // process only existing elements
  suc_New[t] = suc_Old[suc_Old[t]]; // Do pointerjump
}
```

Listing 21.3: OpenCL C Code für die zweite Hälfte eines Pointer-Jumps

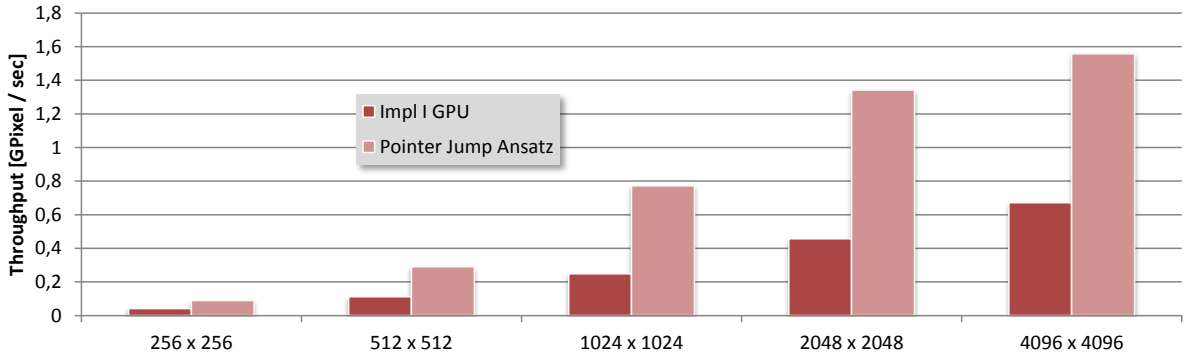
21.4. Evaluation

Zunächst wird experimentell überprüft, inwieweit die eingangs formulierten Annahmen bezüglich des Pointer-Jumping-basierten Ansatzes zutreffen. Dazu werden zunächst die Datensätze Spiral und Ones betrachtet und mit den Ergebnissen von Impl I GPU (siehe Kapitel 20) verglichen (siehe Abbildung 21.1). Wie bei den bisherigen Ansätzen ist auch hier eine bestimmte Mindestauflösung erforderlich, um ein effektives Arbeiten zu ermöglichen. In starkem Gegensatz zu Impl I GPU sinkt der Throughput bei Verarbeitung der Spirale für Auflösungen größer als 1024 x 1024 jedoch. Eine denkbare erste Deutung führt dies auf den superlinearen Gesamtaufwand zurück, welcher aber möglicherweise auch in diesen Auflösungen noch zum Teil durch die steigende GPU-Auslastung aufgefangen wird. Wir werden das weiter unten noch genauer betrachten. In absoluten Zahlen ist der Throughput des Pointer-Jump-basierten Connected-Component-Labeling-Algorithmus in geringen Auflösungen höher und in Auflösungen ab 1024 x 1024 geringer als der von Impl I GPU.

Bei Betrachtung des Ones-Datensatzes ergibt sich ein sehr anderes Bild. Hier ist der Throughput des Pointer-Jump-Ansatzes in allen Auflösungen größer. Der relative Un-



(a) Spiral-Datensatz in verschiedenen Auflösungen



(b) Ones-Datensatz in verschiedenen Auflösungen

Abbildung 21.1.: Vergleich der Throughputs des Pointer-Jump-basierten Connected-Component-Labeling-Algorithmus mit denen des Ansatz Impl I GPU bei verschiedenen Datensätzen. GPU: GTX 670

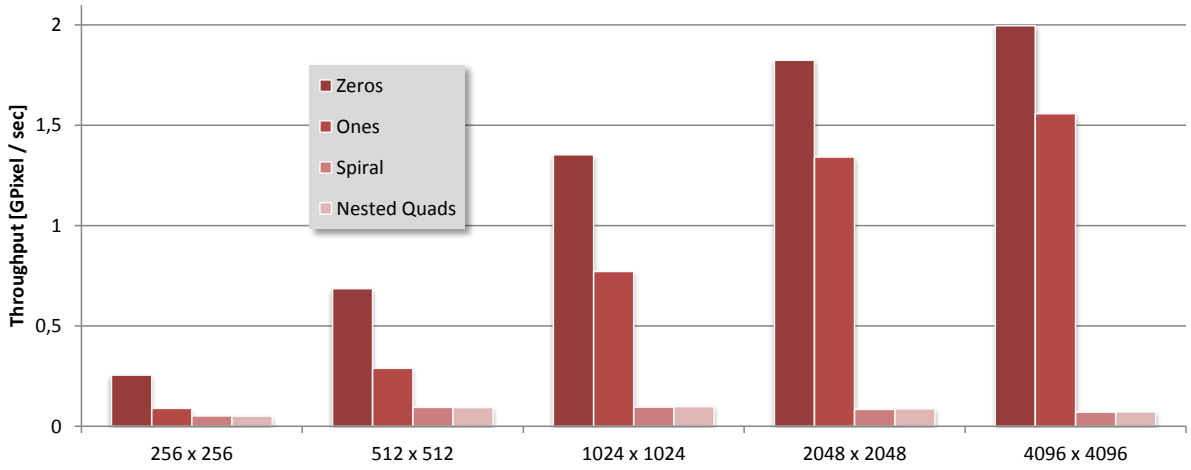


Abbildung 21.2.: Ermittelte Throughputs bei Verarbeitung der Datensätze Ones, Zeros und Spiral in verschiedenen Auflösungen. GPU: GTX 670

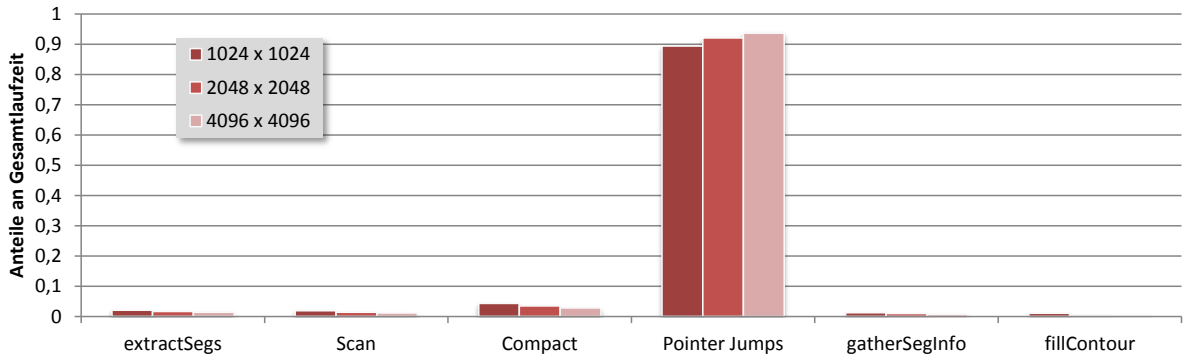
terschied wird mit zunehmender Auflösung etwas kleiner.

Außerdem wird der in Gegensatz zu Impl I GPU erheblich größere Unterschied zwischen Datensätzen, welche viele Kontursegmente generieren und solchen, bei denen das nicht der Fall ist, deutlich. Bei Impl I GPU übersteigt der Throughput des Ones-Datensatzes den des Spiral-Datensatzes in der Auflösung 4096 x 4096 auf einer GTX 670 um einen Faktor von etwa 2. Im Falle des Pointer-Jumping-Ansatz dagegen ist der entsprechende Faktor ca. 22.

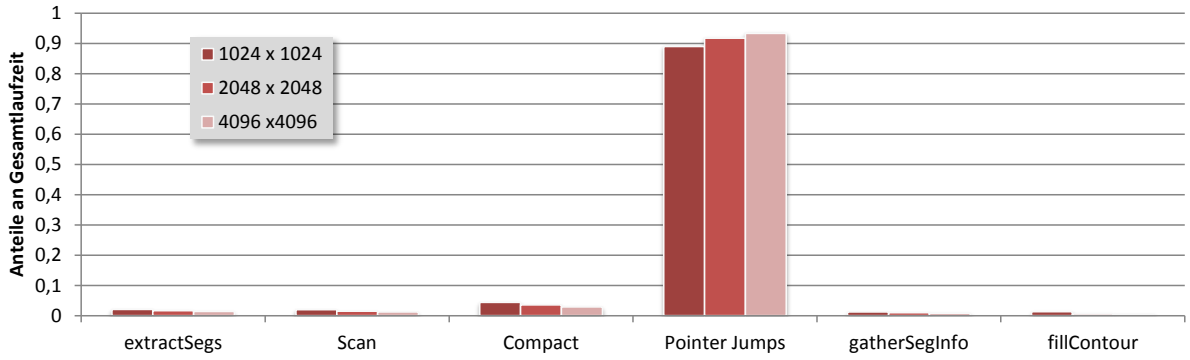
Schauen wir uns nun weitere Datensätze im Vergleich für den Pointer-Jump-Ansatz an, wie in Abbildung 21.2 dargestellt. Die Throughputwerte bei Verarbeitung der Datensätze Nested Quads und Spiral, welche nahezu die gleiche Segmentanzahl extrahieren, sind beinahe identisch. Auch die Throughputwerte der Datensätze Ones und Zeros, welche keine oder sehr wenig Segmente extrahieren, nähern sich in höheren Auflösungen an.

Insgesamt deuten die Beobachtungen auf eine starke Abhängigkeit des Throughputs von der Anzahl der extrahierten Segmente hin. Dies festigt sich bei Betrachtung der Anteile einzelner Subalgorithmen an der Gesamtlaufzeit, welche in Abbildung 21.3 dargestellt sind. Existieren viele Segmente (Nested Quads und Spiral) ist die Pointer-Jumping-Komponente ganz alleine für den Großteil der Laufzeit verantwortlich. Dieser Wert stellt die aufsummierten Laufzeiten aller Aufrufe der Kernel `minThisSuc` und `pointerJump` dar. Zum erwarteten superlinearen Verhalten dieses Schritts passt auch der erhöhte Anteil an der Gesamtlaufzeit bei steigender Auflösung. Die Schritte Scan und Compact fallen im Vergleich damit nicht ins Gewicht. Es lässt sich allerdings noch beobachten, dass `Compact` etwa die doppelte Laufzeit von `Scan` hat.

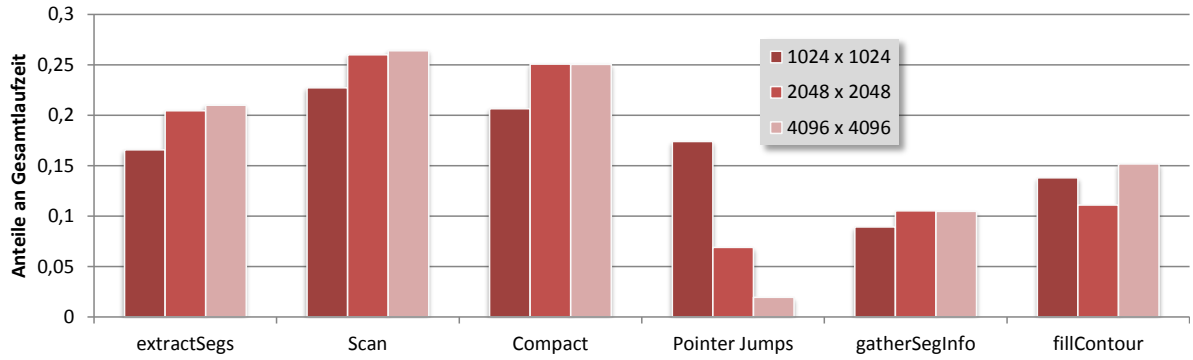
Falls sehr wenige Segmente existieren (Ones und Zeros), spielt die Pointer-Jumping-Komponente, und damit das superlineare Verhalten, eine untergeordnete Rolle. Auch dies war zu erwarten, weil die Datensätze zuvor durch Scan/Compact ausgedünnt sind. Letztere nehmen bei solchen Datensätzen zwar einen vergleichsweise großen Teil der



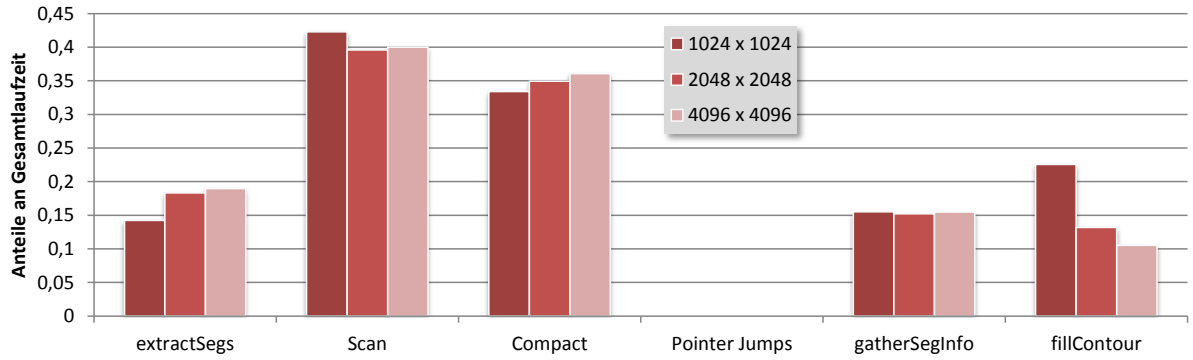
(a) Spiral Datensatz in verschiedenen Auflösungen



(b) Nested Quads Datensatz in verschiedenen Auflösungen



(c) Ones Datensatz in verschiedenen Auflösungen



(d) Zeros Datensatz in verschiedenen Auflösungen

Abbildung 21.3.: Anteile der einzelnen Schritte bzw. Kernel an der Gesamtlaufzeit des Algorithmus bei verschiedenen Datensätzen. GPU: GTX 670

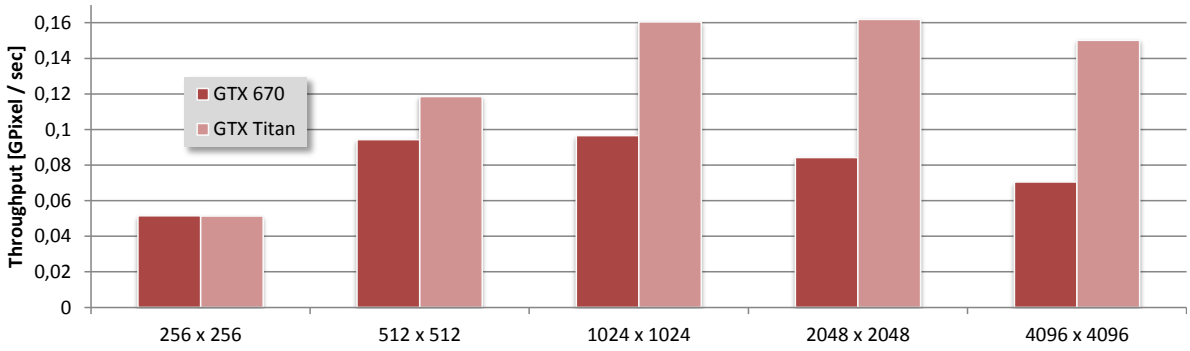


Abbildung 21.4.: Vergleich der Throughputs zwischen GTX 670 und GTX Titan bei Verarbeitung des Spiral-Datensatzes in verschiedenen Auflösungen.

Laufzeit ein, aber gerade sie sind es ja, die diesen Speedup bei Datensätzen mit wenigen Konturen ermöglichen. Im Gegensatz zu dichtbesetzten Datensätzen liegt die Laufzeit von Compact nicht über der von Scan. Das lässt sich folgendermaßen erklären: Der Scan-Algorithmus führt datenunabhängig immer die gleichen Operationen aus, dagegen geht der Compact-Algorithmus zwar ebenfalls über alle Daten, verschiebt aber nur die existierenden Elemente.

Wir untersuchen zuletzt noch die Skalierung des Pointer-Jump-basierten Ansatzes mit steigender GPU-Parallelität bzw. GPU-Leistung. Dafür werden wieder die GTX 670 und die GTX Titan verwendet. Die Ergebnisse sind in Abbildung 21.4 zu sehen. In den höheren Auflösungen erreicht die GTX Titan etwa den doppelten Throughput der GTX 670. Damit skaliert dieser Ansatz weitaus besser mit paralleleren GPUs als im Falle der Impl I GPU. Eine Begründung dafür ist die ausreichend hohe Parallelität (fast) aller Kernel. Das gilt zwar für den Kernel `fillContours` mit Abstrichen, aber sein Anteil an der Gesamtlaufzeit ist äußerst gering.

21.5. Fazit

Der Pointer-Jump-basierte Ansatz, nachfolgend **Impl II** genannt, verhält sich insgesamt entsprechend den in diesem Kapitel eingangs formulierten Erwartungen. Die höhere Parallelität bewirkt im Vergleich mit Impl I GPU einerseits eine Steigerung des Throughputs in geringen Auflösungen und andererseits eine verbesserte Skalierung mit zunehmender GPU-Parallelität. Zusätzlich führt das vorherige Ausdünnen der Daten zu einer Verbesserung des Throughputs bei Datensätzen, welche einen sehr geringen Konturanteil haben.

Allerdings bewirkt die superlineare Gesamtzahl der Berechnungen der Pointer-Jumping-Technik wie befürchtet eine Verschlechterung des Throughputs bei Verarbeitung höherer Datenauflösungen, immer dann, wenn der Konturanteil hoch ist. Dies stellt eine erhebliche Limitierung dar. Bei Verarbeitung der 4096 x 4096 Spirale auf einer GTX 670 ist der Throughput bei Impl I GPU um Faktor 4,5 höher.

Kapitel 22.

Implementationsstrategie III: Theoretisch optimal

Als großer Nachteil der im vorherigen Kapitel evaluierten Implementationsstrategie II hat sich der superlineare Berechnungsaufwand der Pointer-Jumping-Technik herausgestellt. Somit ist es naheliegend, stattdessen den theoretisch optimalen Konturlabeling-Algorithmus zu verwenden, welcher in Kapitel 10.3.2 ab Seite 119 gegeben ist. Mit diesem verbunden ist jedoch ein erheblicher Zusatzaufwand, z.B. um wiederholt mittels 2-Ruling-Sets eine unabhängige Knotenmenge zu bestimmen. Dieser Zusatzaufwand ist konstant, sodass er ab einer gewissen Problemgröße keine Rolle mehr spielt.

In diesem Kapitel wird zunächst eine Implementation des optimalen Connected-Component-Labeling-Algorithmus mit OpenCL skizziert. Gegenstand der Untersuchung derselben ist anschließend primär die Frage, ob dieser Ansatz sich von der ungleich simpleren Pointer-Jumping-Technik bereits bei realen Problemgrößen absetzten kann.

22.1. Implementationsdetails

Die Implementation des Connected-Component-Labelings, basierend auf der optimalen Konturlabeling-Technik, besteht im Wesentlichen aus den selben Schritten wie im Falle der Implementationsstrategie II. Lediglich der Schritt Pointer-Jump-Contour-Labeling wird ersetzt. Im Falle der übrigen Schritte bleiben alle Kernel, mit Ausnahme des Compact Kernels, unverändert. Der grobe Ablauf entspricht, sofern nicht anders beschrieben, der Algorithmusbeschreibung, sodass wir uns an dieser Stelle auf Details der Implementation konzentrieren können.

Es müssen, vor allem bei der 2-Ruling-Set Bestimmung, oft Informationen zwischen einem Knoten und seinen Nachbarn ausgetauscht werden. Da diese noch nicht durch aktuellere Werte überschrieben sein dürfen, wird der Algorithmus in viele Kernel zerlegt, welche oft nur eine Einzige oder wenige Operationen auf jeden Knoten anwenden. Außerdem müssen viele temporäre Daten im globalen Speicher verfügbar gemacht werden, um ebendiesen Austausch zu ermöglichen. Beide genannten Eigenschaften sind sehr charakteristisch für diese Implementation.

22.1.1. Datenstrukturen

Alle von Impl II verwendeten Datenstrukturen werden auch hier und mit der selben Bedeutung verwendet. Ebenfalls wird eine Doublebuffer-Technik eingesetzt, sodass alle mit den Kontursegmenten zusammenhängenden Datenstrukturen doppelt vorliegen. Zusätzlich werden weitere Datenstrukturen, alle je Kontursegment, benötigt:

`pre`: Verweis auf die Adresse des jeweils aktuellen Vorgängersegments. Datentyp: `int`.

`originalId`: Entspricht der gleichlautenden Datenstruktur in Impl II, benötigt hier aber die Doublebuffer-Technik. Datentyp: `int`.

`minId_Original`: Zusätzlich zu dem Doublebuffer `minId` gibt es noch eine dritte Datenstruktur, in welcher die finalen minIds der Originalposition der Segmente entsprechend hinterlegt werden. Datentyp: `int`.

`props2R`: Ein Bitfeld, in welchem verschiedene für die Bestimmung des 2-Ruling-Sets benötigte Eigenschaften codiert und zwischen benachbarten Knoten ausgetauscht werden. Diese sind: selected, deleted, available, lokales Minimum und lokales Maximum. Datentyp: `char`. Vor jeder Bestimmung eines 2-Ruling-Sets werden alle Einträge mit 0 initialisiert, daher ist kein Doublebuffer nötig. Wenn bestimmte Eigenschaften gesetzt oder abgefragt werden, finden die Makros aus Listing 22.1 Verwendung. Eingetragen sind dort nur die für weitere Codebeispiele erforderlichen Makros und Konstanten.

`degree, serialkOld, serialk`: Weitere Datenstrukturen zum Austausch von Informationen bei der 2-Ruling-Set Bestimmung, welche nicht bitweise vorliegen. Dabei stellt serialkOld den Wert $serial_{k-1}$ eines Knotens i dar und serialk den Wert $serial_k$. degree entspricht genau der Algorithmusbeschreibung. In allen Fällen wird keine Doublebuffer-Technik verwendet und Datentyp ist jeweils `char`.

`save_src, save_dst`: Datenstrukturen für die SAVE-Operation aus der Algorithmusbeschreibung. Ein Eintrag zu einem Zeitschritt t besteht aus einem Paar dieser Einträge. Führt ein Knoten v eine Shortcut-Operation aus und überspringt dabei w, wird der aktuelle Index von v in `save_src` und der von w in `save_dst` eingetragen.

Beim späteren Invertieren kann dann zum Zeitschritt t das Label, in der Implementation die `minId`, von `src` nach `dst` weitergereicht werden. Auch diese Datenstrukturen werden compacted, es ist somit ein Doublebuffer nötig. Datentyp ist jeweils `int`. Für den Scan darüber existieren weitere Hilfsdatenstrukturen.

Insgesamt ergibt sich mit ca. 245 Bytes per Pixel ein deutlich erhöhter Speicherverbrauch im Vergleich mit Impl I und II.

```
#define BIT_SELECTED  1 << 1
#define BIT_AVAILABLE 1 << 2
#define BIT_LMIN      1 << 4
#define BIT_LMAX      1 << 5

#define SET_LMIN(props)      (props | BIT_LMIN)
#define SET_LMAX(props)      (props | BIT_LMAX)
#define SET_SELECTED(props)  (props | BIT_SELECTED)
#define SET_AVAILABLE(props) (props | BIT_AVAILABLE)

#define IS_LMIN(props)      ((props & BIT_LMIN     ) == BIT_LMIN)
#define IS_LMAX(props)      ((props & BIT_LMAX     ) == BIT_LMAX)
#define IS_SELECTED(props)  ((props & BIT_SELECTED ) == BIT_SELECTED)
#define IS_AVAILABLE(props) ((props & BIT_AVAILABLE) == BIT_AVAILABLE)

#define RULING(props) IS_SELECTED(props)
```

Listing 22.1: Makros zum Setzen und Abfragen von einigen, mit dem 2-Ruling-Set zusammenhängenden, Eigenschaften der Segmente

22.1.2. Berechnung eines 2-Ruling-Sets

Für jede Iteration der While-Schleife des in Pseudocodeabschnitt Algorithmus 18 auf Seite 120 gegebenen Contourlabeling-Algorithmus ist ein 2-Ruling-Set zu bestimmen. Das entspricht in Algorithmus 18 Step I und Step II. Eingangsdaten sind die jeweils noch verbleibenden Kontursegmente in dichtgepackter Form. Deren Anzahl nennen wir `length`. Die LWS kann für alle in diesem Kapitel besprochenen Kernel frei gewählt werden, sodass sich LWS(256, 1, 1) anbietet. Die (eindimensionale) GWS wird für das kleinste ganzzahlige Vielfache von 256 eingestellt, welches $\geq$ `length` ist.

Die Idee zur Bestimmung eines 2-Ruling-Sets wird im Vergleich mit der Beschreibung in Kapitel 7.6 (ab Seite 43) für die praktische Implementation vereinfacht. Beobachten wir zunächst, dass ein $log_2(log_2(log_2(log_2(N))))$-ruling Set bereits ein 2-Ruling-Set ist für alle irdischen N. Dieser lässt sich bestimmen, indem einmal der Basisschritt (Algorithmus 8, (S. 49)) und danach drei Mal der k-te Schritt (Algorithmus 9, Seite 50) ausgeführt wird.

Behandeln wir zunächst eine Implementation der ersten Ausführung des Basisschritts der Berechnung eines 2-Ruling-Sets. Wie bereits beschrieben, muss die Berechnung eines 2-Ruling-Sets in verschiedenen Kernel aufgeteilt werden, um globale Synchronisationspunkte zu ermöglichen. Ohne synchronisieren zu müssen, können für alle Knoten die Werte für serialk bestimmt werden und ob, davon abhängig, es sich um lokale Minima oder Maxima handelt. Dies führt der Kernel `set_serialk_LocalMin_localMax_0` aus, welcher in Listing 22.2 gegeben ist. Darin, und auch in den folgenden Kerneln dieses Kapitels, werden Makros aus Listing 22.1 verwendet.

```
// Implementation of function calcSerial(k, i). Uses function
// clz (count leading zeros)
#define CALC_SERIALK(k, i) (LAST_BIT_INDEX - clz(k ^ i))
// Better would be:
// #define CALC_SERIALK(k, i) (ctz(k ^ i))
// But ctz (OpenCL 2.0) is currently not supported by Nvidia...

// Computes serialk, for each node and determines if it is a local
// minimum or maximum. Important: Only applicable in iteration k = 1
// Then, serialkOld[i] is i for each i
kernel void set_serialk_LocalMin_localMax_1(
 global char* props2R,
 global int*  suc,
 global int*  pre,
 global char* serialk,
 const  int   length)
{
  int i = get_global_id(0);
  if(i >= length) return;
  char props = props2R[i];
  int sucL = suc[i];

  // Compute serialk for this, pre and suc element
  char serialk_T = CALC_SERIALK(i,      sucL     );
  char serialk_P = CALC_SERIALK(pre[i], i        );
  char serialk_S = CALC_SERIALK(sucL,   suc[sucL]);

  // Write this serialk. Needs to be exchanged in next iteration
  serialk[i] = serialk_T;

  // Determine and write if node is a local minimum or maximum
  if(serialk_T <= serialk_P && serialk_T <= serialk_S){
    props2R[i] = SET_LMIN(props);
  } else if(serialk_T >= serialk_P && serialk_T >= serialk_S){
    props2R[i] = SET_LMAX(props);
  }
}
```

Listing 22.2: OpenCL C Code für den Abschnitt 'Init data for current k' aus Algorithmus 8, (s. 49). Makro CALC_SERIALK implementiert die Funktion calcSerial (s. 46) für $k > 0$

In Algorithmus 8 entspricht dies in etwa dem Abschnitt 'Init data for current k'. Allerdings wird `odd` nicht gespeichert, da es nur lokal verwendet wird.

Die beschriebene Vorgehensweise ist nur im Falle der ersten Ausführung möglich, weil hier die Werte von `serialk` allein vom Index abhängen. Deshalb können die Werte der Knoten `pre` und `suc` berechnet werden und ein Austausch ist nicht nötig. Ab der zweiten Berechnung des Basisschritts muss dieser Kernel demnach durch zwei Kernel ersetzt werden.

Der nächste Teil der Implementation des Basisschritts stellt der in Listing 22.3 gegebene Kernel `select_If_LMin_1` dar, welcher ebenfalls exklusiv für den ersten Aufruf

anwendbar ist.

```
// Checks if bit bitId in bits is 1
#define IS_BIT_SET(bits, bitId) \
    ((bits & (1 << (bitId))) == (1 << (bitId)))

// Important: Only applicable in iteration k = 1
// Then, serialkOld[i] is i for each i
kernel void select_If_LMin_1(
  global char* props2R,
  global int*  suc,
  global int*  pre,
  global char* serialk,
  const  int   length)
{
  int i = get_global_id(0);
  if(i >= length) return;
  char props = props2R[i];

  if(
    IS_LMIN(props)
    &&
    (
      (!IS_LMIN(props2R[suc[i]]) && !IS_LMIN(props2R[pre[i]]))
      || IS_BIT_SET(i, serialk[i])
    )
  ){
    props2R[i] = SET_SELECTED(props);
  }
}
```

Listing 22.3: OpenCL C Code für Abschnitt 'Consider local min' aus Algorithmus 8, (s. 49).

In Algorithmus 8 entspricht dies dem Abschnitt 'Consider local min'. In diesem Kernel wird insbesondere abgefragt, ob die Nachbarknoten lokale Minima sind. Dies ist im vorherigen Kernel berechnet worden und deshalb ist dazwischen ein globaler Synchronisationspunkt gesetzt.

Der nun folgende Kernel, `setAvailableIfMax_1`, implementiert den Teil 'Set available if neighbours not selected' des Abschnitts 'Consider local max' aus Algorithmus 8. Er ist in Listing 22.4 gegeben. Es wird darin die Information benötigt, ob die Nachbarknoten `selected` sind, daher ist eine erneute Synchronisation erforderlich.

```
kernel void setAvailableIfMax_1(
  global char* props2R,
  global int*  suc,
  global int*  pre,
  const  int   length)
{
  int i = get_global_id(0);
  if(i >= length) return;
  char props = props2R[i];

  if(IS_LMAX    (props          ) &&
    !IS_SELECTED(props          ) &&
    !IS_SELECTED(props2R[suc[i]]) &&
    !IS_SELECTED(props2R[pre[i]])
  ) {
    props2R[i] = SET_AVAILABLE(props);
  }
}
```

Listing 22.4: OpenCL C Code für Abschnitt 'Set available if neighbours not selected' aus Algorithmus 8, (s. 49).

Der Rest des Algorithmus 8, nämlich der Teil 'Select (possibly) if available' aus Abschnitt 'Consider local max', wird durch Kernel `selectAvailable_1` implementiert. Dieser ist in Listing 22.5 gegeben. Es muss dabei überprüft werden, ob die Nachbarn `available` sind, weshalb ein weiterer Synchronisationspunkt erforderlich ist.

```
// Important: Only applicable in iteration k = 1
// Then, serialkOld[i] is i for each i
kernel void selectAvailable_1(
  global char* props2R,
  global int*  suc,
  global int*  pre,
  global char* serialk,
  const  int   length)
{
  int i = get_global_id(0);
  if(i >= length) return;
  char props = props2R[i];

  if(
    IS_AVAILABLE(props) &&
    (
      (
        !IS_AVAILABLE(props2R[suc[i]]) &&
        !IS_AVAILABLE(props2R[pre[i]])
      )
      || IS_BIT_SET(i, serialk[i])
    )
  ) {
    props2R[i] = SET_SELECTED(props);
  }
}
```

Listing 22.5: OpenCL C Code für Abschnitt 'Select (possibly) if available' aus Algorithmus 8, (s. 49). Makro IS_BIT_SET ist in Listing 22.3 (Seite 234) gegeben.

Diese beschriebenen vier Kernel stellen die Implementation des Basisschritts für dessen erste Ausführung dar. Analog dazu kann der k-te Schritt (siehe Algorithmus 9, Seite 50), welcher ebenfalls den Basisschritt beinhaltet, implementiert werden. Auch hier erfolgt die Aufteilung in Kernel wieder gemäß Synchronisationspunkten für den Datenaustausch benachbarter Knoten. Wie zuvor angemerkt, weicht die Implementation des Basisschritts für $k > 1$ leicht ab. Insgesamt sind für die Implementation des k-ten Schritts weitere sieben Kernel nötig, welche hier jedoch nicht mehr besprochen werden sollen. Soll nun ein log(log(log(log(N))))-Ruling-Set bestimmt werden, müssen einmal die vier Kernel des Basisschritts und dreimal die sieben Kernel des k-ten Schritts ausgeführt werden. Um für in der Praxis auftretende Größen einmal einen 2-Ruling-Set zu bestimmen, sind folglich $4 + 3 \cdot 7 = 25$ Kernelstarts erforderlich.

22.1.3. Step III

Den nächsten Schritt einer Iteration der While-Schleife nach Bestimmung des 2-Ruling-Sets stellt Step III in Algorithmus 18 dar. Hier werden die Shortcut-Operations auf Elemente des 2-Ruling-Set angewendet. Dieser Schritt wird durch Kernel `cv_Step_III` (siehe Listing 22.6) implementiert. Er wird auf die (noch) dichtgepackten Konturliniensegmente der jeweiligen While-Schleifen-Iteration angewendet. Der Kernel benötigt bei

Übersetzung für Compute-Capability 3.0 15 Register. Dieser Wert ist von allen Kerneln, welche speziell für Algorithmus 18 / 19 benötigt werden, der höchste. Dementsprechend ist bei Ausführung auf einer entsprechenden GPU nicht von einer Limitierung der Parallelität durch den Ressourcenverbrauch auszugehen.

```
kernel void cv_Step_III (
 global char* props2R,
 global int*  suc,            global int*  pre,
 global int*  minId,          global int*  originalId,
 global char* saveExisting,   global int*  saveSrc,
 global int*  saveDst,        global int*  minId_Original,
 const  int   length,         const  int   saveOffset)
{
  int i = get_global_id(0); if(i >= length) return;

  // If i is in 2"=Ruling"=Set, apply shortcut operation
  if(RULING(props2R[i])){
    int suc_T   = suc  [i    ];
    int minId_T = minId[i    ];
    int sucSuc  = suc  [suc_T];
    int newSuc  = sucSuc;

    // Create entry in save for first removed node (SAVE)
    saveExisting[saveOffset + suc_T] = 1;
    saveSrc     [saveOffset + suc_T] = originalId[i  ];
    saveDst     [saveOffset + suc_T] = originalId[suc_T];

    // Determine new minId
    minId_T = min(minId[suc_T], minId_T);

    // If i.suc.suc not in 2"=Ruling"=Set, apply 2nd shortcut operation
    if(!RULING(props2R[sucSuc])) {
      newSuc = suc[sucSuc];

      // Determine new minId
      minId_T = min(minId[sucSuc], minId_T);

      // Create entry in save for second removed node (SAVE)
      saveExisting[saveOffset + sucSuc] = 1;
      saveSrc     [saveOffset + sucSuc] = originalId[i    ];
      saveDst     [saveOffset + sucSuc] = originalId[sucSuc];
    }

    // If i is not last in list, update list
    if(newSuc != i){
      minId[i     ] = minId_T;
      suc  [i     ] = newSuc;
      pre  [newSuc] = i;
    }
    // Else remove last node and store final minId (PAUSE)
    else {
      props2R       [           i ] = 0;
      minId_Original[originalId[i]] = minId_T;
    }
  }
}
```

Listing 22.6: OpenCL C Code für Abschnitt 'Step III' aus Algorithmus 18, (s. 120). Kommentare in Klammern: Namen der Operationen im Algorithmus

22.1.4. Gesamtablauf

Mit den gegebenen Kerneln der vorherigen Abschnitte kann nun der gesamte Ablauf der Implementation des Algorithmus 18 besprochen werden. Skizzieren wir kurz den Ablauf der While-Schleife, welche wiederholt wird, bis eine einstellbare Anzahl von verbleibenden Kontursegmenten unterschritten wird:

Step I und II Berechne 2-Ruling-Set, wie oben beschrieben.

Step III Führe Kernel `cv_Step_III` aus.

Step IV Wende Scan/Compact einerseits auf die Knoten des aktuellen 2-Ruling-Sets und andererseits auf die Einträge in `save` , jeweils dieser Iteration, an. Einträge in `save` werden dabei an Einträge vorheriger Zeitschritte angehängt.

Wenn die Problemgröße klein genug geworden ist, kann die Basis-Pointer-Jumping-Technik, deren Implementation in Kapitel 21 skizziert ist, angewendet werden. Das entspricht in Algorithmus 19 (siehe Seite 123) **Step V**.

Zuletzt müssen die gefundenen `minIds` gemäß den Einträgen in `save` weitergereicht werden, was in Algorithmus 19 **Step VI** entspricht. Dazu wird der Ablauf obiger While-Schleife invertiert. Mithilfe des Scans der While-Schleife wird dafür mitprotokoliert, wie viele `save` Einträge je Iteration erstellt werden. Beim Invertieren lassen sich so Offset und Anzahl auszuwertender `save` Einträge bestimmen.

22.2. Evaluation

Wir beginnen die Evaluation mit der experimentellen Suche nach einem geeigneten Offset, unterhalb dessen die Basis-Pointer-Jumping-Technik (Step V) verwendet wird. Dieser Offset wird angegeben in der Anzahl der noch verbleibenden Kontursegmente. Für die Suche nach einem geeigneten Offset wird der Spiral-Datensatz verwendet, schließlich ist das Hauptziel der Implementationsstrategie III eine Verbesserung des Verhaltens bei Datensätzen, welche viele Konturen beinhalten. Es wird sich zeigen, dass diese Vorgehensweise zu ausreichenden Erkenntnissen führt. Als Auflösung werden lediglich 2048 x 2048 Pixel verwendet, da eine höhere Auflösung bedingt durch den Speicherverbrauch auf der GTX 670 nicht ausführbar ist.

Es werden nun verschiedene Werte für den Offset zwischen 0 (Basis-Pointer-Jumping-Technik wird gar nicht verwendet) und inf (Basis-Pointer-Jumping-Technik wird ausschließlich verwendet) betrachtet. Bei dieser Konfiguration lassen sich, in Abhängigkeit des eingestellten Offsets, einige Eigenschaften des Programmablaufs ermitteln, welche der Tabelle 22.1 zu entnehmen sind. Diese Eigenschaften sind im Einzelnen: Anzahl der Iterationen der Basis-Pointer-Jumping-Technik (BPJ Iterationen), Anzahl der Iterationen des optimalen Konturlabeling-Algorithmus, welcher die Konturen ausdünnt (Iter. Step I-IV) und der Gesamtzahl aller durch den Algorithmus gestarteten Kernel (Anzahl Kernelstarts). Natürlich hat die Veränderung des Offsets potentiell Auswirkungen auf

Tabelle 22.1.: Eigenschaften des Programmablaufs bei Verarbeitung des 2048 x 2048 Spiral-Datensatzes bei verschiedenen Offsets der Basis-Pointer-Jumping-Technik.

	0k	0,25k	1k	4k	16k	64k	256k	1M	4M	Inf
BPJ Iterationen	0	8	10	11	14	16	18	20	21	23
Iter. Step I - IV	19	13	11	10	8	5	4	2	1	0
Anzahl Kernelstarts	611	468	410	381	325	267	195	123	87	54

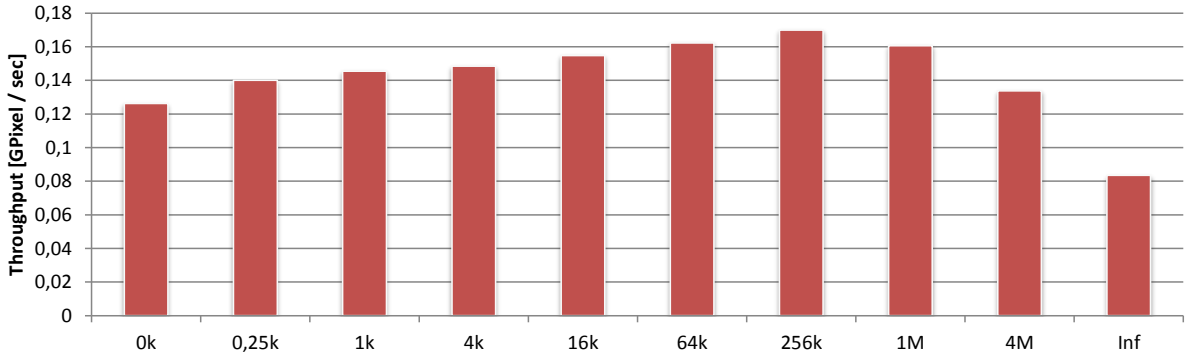


Abbildung 22.1.: Ermittelte Throughputs bei Verarbeitung des Spiral-Datensatzes in der Auflösung 2048 x 2048 für verschiedene Offsets (verbleibende Kontursegment) unterhalb derer die Basis-Pointer-Jumping-Technik verwendet wird. Hinweis: k = 1024, M = k * k. GPU: GTX 670

die Balance zwischen der Anzahl der Iterationen der Basis-Pointer-Jumping-Technik einerseits und der wiederholten Anwendung der Steps I bis IV andererseits. Hier liefert die Tabelle 22.1 ein Beispiel wie diese im Falle der 2048 x 2048 Spirale ausfällt. Je niedriger der Offset, desto öfter werden Steps I bis IV ausgeführt. Das resultiert in einer immer kleineren, oder sogar verschwundenen (Offset = 0) Restproblemgröße. Allerdings müssen in jedem Durchlauf sehr viele Kernel gestartet werden, welche gegen Ende eine sehr geringe Problemgröße verarbeiten. Seien beispielsweise noch 8 Elemente übrig. Dann müssen trotzdem mehrere Dutzend Kernel gestartet werden, um diese Zahl auf z.B. 4 zu verringern. Zusätzlich muss nach jedem Schritt die dann verbleibende Problemgröße mit dem Host synchronisiert werden, um alle Kernel neu zu konfigurieren. Es ist demnach nicht zu erwarten, dass derart geringe Offsets sinnvoll sind. Ab einer gewissen Problemgröße kann aber eine Verbesserung des Skalierungsverhaltens durch dieses Ausdünnen der Daten erhofft werden.

Mit diesen Vorüberlegungen wird nun ein geeigneter Offset experimentell bestimmt. Dazu wird für die gleichen Konfigurationen, wie in Tabelle 22.1 dargestellt, auf einer GTX 670 jeweils der Throughput ermittelt. Die Ergebnisse sind in Abbildung 22.1 gegeben. Wie erwartet, darf der Offset nicht zu gering ausfallen. Das beste Ergebnis erhalten wir für 256k Segmente. Bei noch höheren Offsets fällt der Throughput zunehmend ab. Bei ausschließlicher Verwendung der Basis-Pointer-Jumping-Technik (Offset = Inf) ermitteln wir den geringsten Throughput. Somit stellt der theoretisch optimale Implementationsansatz, nachfolgend als **Impl III** bezeichnet, im Falle der Spirale in 2048 x 2048 eine

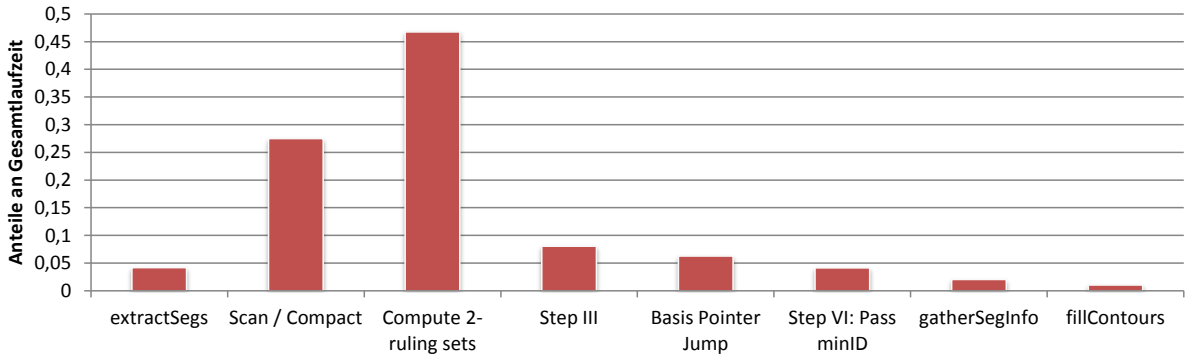


Abbildung 22.2.: Anteile der einzelnen Schritte bzw. Kernel an der Gesamtlaufzeit des Algorithmus bei Verarbeitung der 2048 x 2048 Spirale. GPU: GTX 670.

Verbesserung gegenüber der Impl II dar. Der Offset wird in dieser Arbeit fest auf 256k eingestellt. Es sei angemerkt, dass dies eine Optimierung, auf den getesteten Datensatz einerseits und (zumindest etwas) auf die GTX 670 andererseits, darstellt.

Betrachten wir nun zum besseren Verständnis des Verhaltens die Anteile der einzelnen Schritte bzw. Kernel an der Gesamtlaufzeit des Algorithmus bei Verarbeitung der 2048 x 2048 Spirale, dargestellt in Abbildung 22.2. Wie nicht anders zu erwarten, entfällt weiterhin ein Großteil der Laufzeit auf das Konturlabeling. Im Falle des gewählten Offsets beanspruchen Teile, die mit dem Ausdünnen der Daten zusammenhängen (Scan/Compact, Compute 2-Ruling-Sets, Step III und Step VI: Pass minId) einen deutlich größeren Anteil als die Basis-Pointer-Jumping-Technik. Die interessanteste Beobachtung betrifft jedoch das Verhältnis der Teile des Ausdünnvorgangs. Die eigentliche Operation (Step III und dessen Invertierung Step VI) auf der einen Seite benötigen einen weitaus geringeren Anteil als das Bestimmen der zu verarbeitenden Knoten (Compute 2-Ruling-Sets) und die wiederholten aufrufe von Scan / Compact auf der anderen Seite. Diese sind insbesondere zusammen für über 70 Prozent der gesamten Laufzeit verantwortlich.

Vergleichen wir nachfolgend den neuen Ansatz mit den Implementationen Impl I GPU und Impl II bei Verarbeitung der Spirale in verschiedenen Auflösungen auf der GTX 670. Die ermittelten Throughputs sind in Abbildung 22.3 zu sehen. Im Vergleich mit Impl II lässt sich ein fast identischer Throughput für die Auflösungen 512 x 512 und darunter feststellen. Offensichtliche Ursache ist die lediglich einmalige Ausführung (512 x 512) oder das Überspringen des Ausdünnvorgangs (256 x 256).

In höheren Auflösungen tritt, wie bereits nach Ergebnissen aus Abb. 22.1 zu erwarten, eine Verbesserung gegenüber Impl II ein. Insbesondere ist das Primärziel, die Throughputabnahme bei höheren Segmentzahlen zu verhindern, erreicht worden. Stattdessen steigt der Throughput mit höheren Auflösungen, wenn auch nicht so stark wie der von Impl I GPU. Leider kann der Throughput letzterer Implementation auch in absoluten Zahlen nicht erreicht werden.

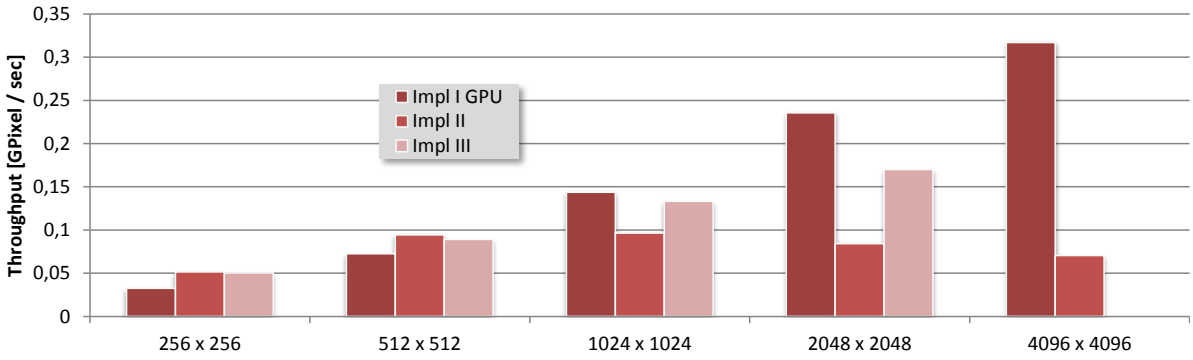


Abbildung 22.3.: Ermittelte Throughputs bei Verarbeitung des Spiral-Datensatzes in verschiedenen Auflösungen und Vergleich mit den Ansätzen Impl I GPU und Impl II. Verwendete GPU: GTX 670

Hinsichtlich der übrigen Datensätze genügen an dieser Stelle einige Anmerkungen. Die Evaluation von Zeros und Ones ist nicht sinnvoll, da die Anzahl der enthaltenen Segmente unterhalb des Offsets liegt. Im Falle des Datensatzes Nested Quads weicht der ermittelte Throughput beispielsweise in der Auflösung 2048 x 2048 um weniger als 1 Prozent von dem der Spirale ab, weshalb hier ebenfalls auf eine grafische Darstellung verzichtet wird.

22.3. Fazit

Die Zielvorgabe, die superlinear mit der Problemgröße steigende Laufzeit der Impl II experimentell nachweisbar zu entfernen, ist durch Impl III erreicht. Zusätzlich kann sich Impl III auch im Falle realer Problemgrößen (ab Auflösung 1024 x 1024 der Spirale) positiv von Impl II absetzen. Aufgrund des hybriden Ansatzes bleiben einige positive Eigenschaften von Impl II (Speedup bei wenigen Konturen, Verhalten bei geringen Problemgrößen und GPU-Skalierung) nahezu uneingeschränkt erhalten.

Insgesamt wird jedoch die Performance von Impl I GPU gerade im Falle des Spiral-Datensatzes in höheren Auflösungen nicht erreicht.

Als Hauptproblem lässt sich der hohe Laufzeitanteil der Berechnungen zum wiederholten Auffinden eines 2-Ruling-Sets einerseits und der Scan / Compact Operationen andererseits identifizieren.

Anzumerken ist ferner der im Vergleich mit Impl I GPU und Impl II deutlich erhöhte Speicherbedarf, auch wenn die Minimierung des Speicherverbrauchs kein explizites Ziel der Untersuchung ist.

Kapitel 23.

Implementationsstrategie IV: GPU optimiert

Motiviert durch die wenig befriedigenden Ergebnisse für GPUs in den zuvor untersuchten Implementationsstrategien I - III wird in diesem Kapitel ein Ansatz vorgestellt, der sich deutlich von den in Teil II dieser Arbeit vorgestellten Algorithmen unterscheidet. Der Fokus liegt darauf, die Techniken in einer Weise zu implementieren, welche der Architektur einer GPU verstärkt Rechnung tragen. Im Gegenzug werden die theoretischen Eigenschaften 'optimal' und sogar 'cost-' bzw. 'work-' optimal aufgegeben.

Die Implementationsstrategie IV, nachfolgend **Impl IV**, sieht, ähnlich wie Stava [OS11], die Verwendung eines mehrstufigen Tile-basierten Schemas vor. Die Segmente einzelner Tiles werden im Local-Memory abgelegt und die simple Pointer-Jump-Technik zum lokalen Konturlabeling verwendet. Anschließend werden nur die Ränder weiterverarbeitet und zwar wieder auf ähnliche Weise. So ist eine effektive Nutzung des schnellen Shared-Memory, im Gegensatz zu vorherigen Ansätzen, möglich. Außerdem kann lokale Synchronisation die Mehrheit der vormals nötigen globalen Synchronisationspunkte ersetzen. Wie wir sehen werden, limitiert die Shared-Memory Größe bei diesem Ansatz mehr als alles andere. Deshalb fällt die Wahl auf die bislang wenig performante Pointer-Jump-Technik (siehe Impl II), deren Segmentrepräsentation am wenigsten Speicher benötigt.

Auf ein vorheriges Ausdünnen der Daten, wie in Impl II / III, wird verzichtet. Wir legen somit bei diesem Ansatz den Fokus auf Datensätze, welche viele Konturen beinhalten. Schließlich haben sich gerade solche in den Ansätzen II / III als besonders problematisch erwiesen.

Der superlinearen Anzahl der Gesamtoperationen wirkt die, zumindest am Anfang, schnelle Problemverkleinerung durch das Tile-Schema entgegen. Hauptziel der experimentellen Untersuchung ist demnach, zu zeigen, dass dies für typische Problemgrößen in der Praxis ausreicht. Im Rahmen dieses Kapitels wird demnach auch beispielhaft untersucht, wie wichtig theoretische Eigenschaften im Gegensatz zu einem hohen Optimierungsgrad für eine GPU sind.

23.1. Beschreibung des Tile-basierten Ansatzes

Liniensegmente werden bei diesem Ansatz mit den jeweils nur innerhalb des Tiles gültigen `suc` und `minId` Eigenschaften codiert. Eine lokale Codierung von `suc` ist erforderlich, um im lokalen Adressraum des Local-Memory gültige Verweise zu erhalten. Dagegen könnten die Werte von `minId` auch global gültig sein. Darauf wird jedoch verzichtet, um weniger Bits zur Kodierung zu benötigen. Auf diese Weise können `suc`, `minId` und weitere, den Zustand eines Kontursegments betreffende Eigenschaften in einem einzigen 32-Bit Integer hinterlegt werden. Diese Vorgehensweise resultiert in zwei positiven Eigenschaften: Erstens wird der Shared-Memory-Verbrauch minimiert. Dies ist ausgesprochen wichtig, denn, wie wir sehen werden, ist dieser Ansatz trotzdem noch über den Shared-Memory Verbrauch limitiert. Zweitens können so alle Eigenschaften eines Segments in einem Aufruf geladen und nach ausgeführter Operation wieder mit einem Zugriff geschrieben werden. Dadurch verringern sich die Zugriffe auf das Shared-Memory und der Synchronisationsaufwand sinkt. Schließlich ist es so gar nicht möglich, dass der Wert von `suc` bereits verändert ist, der von `minId` jedoch noch nicht, weil die Änderungen für andere Work-Items erst nach der vollständigen Operation sichtbar werden.

Die Implementation des Tile-basierten Ansatzes ist, aufgrund ihres im Vergleich mit den zuvor beschriebenen Ansätzen hohen Optimierungsgrades, recht aufwändig. Sie wird daher erst für das Endergebnis anhand eines zentralen Kernels skizziert (Abschnitt 23.5). Sein Verständnis ist im Vergleich mit allen vorherigen Codebeispielen in dieser Arbeit ungleich bedeutsamer, um das Verhalten des konturbasierten Connected-Component-Labeling-Algorithmus in der Praxis auf einer GPU einordnen zu können.

Tile-Pointer-Jumps

Der Kernel `tilePointerJumps` stellt die erste Hälfte der Implementation des Tile-basierten Schritts dar. Er bekommt als Eingangsdaten die durch den Kernel `extractSegs` erzeugten Segmente und zusätzlich Informationen über die Pixel. Diese Daten sind gedanklich in rechteckige Bereiche, die Tiles, aufgeteilt. Die Work-Groups werden derart konfiguriert, dass je eine Work-Group ein Tile repräsentiert und für die Verarbeitung von deren Daten zuständig ist. Es wird ein zweidimensionaler Index verwendet, um die notwendigen Sonderbehandlungen an den Tile-Rändern zu vereinfachen.

Die Tile-Verarbeitung beginnt mit dem Laden der Segmente aus dem globalen Speicher und Ablegen im Local-Memory, gefolgt von einer lokalen Synchronisation. Dann werden darauf die Pointer-Jump-Operationen angewendet. Wie oft das wiederholt wird, hängt nicht von der Datenauflösung, sondern von der maximalen Anzahl der Segmente in den einzelnen Tiles ab. Diese wiederum entspricht der vierfachen Anzahl der Pixel eines Tiles. Somit werden für alle Tiles immer $\lceil log_2(4 \cdot pCnt) \rceil$ Pointer-Jumps auf alle enthaltenen Segmente angewendet, wenn $pCnt$ die Anzahl der Pixel eines Tiles ist. Die Anzahl von Berechnungen, welche 'zu viel' passiert im Vergleich mit einer sequentiellen Verarbeitung, bezeichnen wir nachfolgend als **Overhead-Faktor**. Weil die Tile-Größe eine (fest einstellbare) Konstante ist, darf das aber nicht als Hinweis auf das asymptotische Verhalten in Abhängigkeit von der Problemgröße missverstanden werden.

Nach jedem Pointer-Jump muss lokal synchronisiert werden. Segmente, deren Nachfolger außerhalb des Tiles liegen, sind passiv, d.h. sie selbst führen keine Pointer-Jumps aus, aber andere Segmente können sich mit ihnen vergleichen. Wir bezeichnen solche Segmente als **OOT** (Out of Tile). Andere Segmente, die, ggf. nach Ausführung von Pointer-Jumps, ein OOT-Segment als Nachfolger haben, werden dann ebenfalls in den Passiv-Modus gesetzt.

Nach Ausführung der Pointer-Jumps wird die weitere Verarbeitung der Segmente zunächst in zwei Kategorien eingeteilt: Segmente, welche zu einer innerhalb des Tiles geschlossenen Figur gehören und solche für die das nicht gilt. Letztere sind identifizierbar am gesetzten Passivmodus (und umgekehrt). Der Zustand nach Ausführung der Pointer-Jumps ist, im Vergleich mit der Ausgangssituation, in Abbildung 23.1 veranschaulicht. Je eine der enthaltenen Figuren fällt in jede Kategorie.

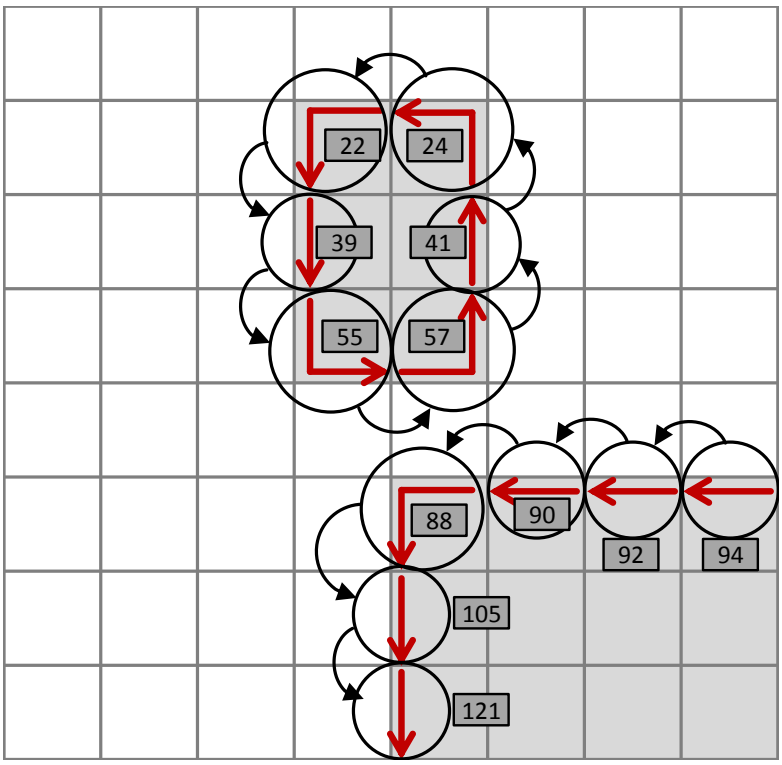
Im Falle einer innerhalb des Tiles geschlossenen Kontur ist deren Verarbeitung nach den Pointer-Jumps abgeschlossen und alle Segmente haben den minimalen Wert aller `minIds` dieser Kontur. Über den Wert von `suc` kann dagegen keine allgemeine Aussage getroffen werden und er wird auch nicht benötigt. Falls es sich um eine äußere Kontur handelt, kann die lokale `minId` der Segmente unter Berücksichtigung des Tile-Index in ein globales Label umgerechnet und für den korrespondierenden Pixel gesetzt werden. Zusätzlich müssen ggf. Eigenschaften des Pixels, wie `ioCnt`, aktualisiert werden. Danach werden die Segmente nicht mehr benötigt und können entfernt werden.

Liegt dagegen eine Kontur nur teilweise innerhalb des Tiles, ist ihre Verarbeitung noch nicht abgeschlossen. Jedes Segment hat als `minId` den minimalen Wert seiner eigenen ehemaligen `minId` und denen aller nachfolgenden Knoten. Somit ist lediglich bei einem Eintrittssegment in das Tile dieser Wert garantiert der Minimale der gesamten Kontur innerhalb des Tiles, da es keine Vorgänger hat. Bei allen Segmenten ist `suc` auf das Segment derselben Kontur gesetzt, welches aus dem Tile herauszeigt. Dieser Wert von `suc` wird für alle Segmente in den globalen Speicher zurückgeschrieben.

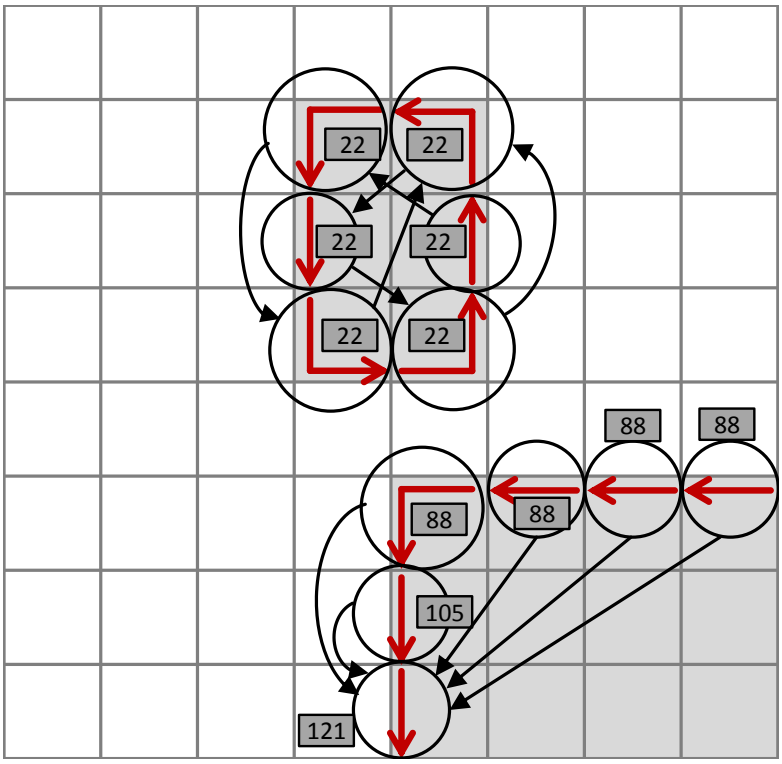
Der Konturlabeling-Algorithmus wird, in weiteren Kerneln, nur noch auf die Eintrittssegmente angewandt. Diese werden unter Zuhilfenahme der Information der OOT-Segmente neu verlinkt. Dabei wird eine eigene Datenstruktur verwendet, sodass die Adresse des OOT-Segments nicht überschrieben wird.

Pass Labels

Auf die Ausführung des Kernels `tilePointerJumps` folgen Kernel, welche das Konturlabeling abschließen, sodass alle ehemaligen Eintrittssegmente der Tiles die globale `minId` ihrer Kontur haben. Der `passLabels` Kernel lädt anschließend, in identischer Tile-Konfiguration, die noch zu verarbeitenden Segmente. Zuerst speichern die Eintrittssegmente ihre Werte von `minId` an der Stelle ihres lokalen Wertes von `suc`, gefolgt von einer lokalen Synchronisation, im Local-Memory. Dieses ist die Adresse des aus dem Tile herauszeigenden Segments, auf welches auch die Werte von `suc` aller anderen Segmente der jeweiligen Kontur des Tiles verweisen. Diese können sich dort dann den finalen `minId` Wert herholen. Anschließend kann mit diesen Segmenten so verfahren werden, wie oben mit denen, welche zu lokal geschlossenen Konturen gehören.



(a) Anfangssituation nach Initialisierung



(b) Nach Ausführung aller Pointer-Jumps

Abbildung 23.1.: Veranschaulichung der Tile-basierten Pointer-Jumps anhand eines Tiles der Größe 8 x 8 Pixel. Pixel klassifiziert (grau), sonst (weiß); Rote Pfeile: Liniensegmente gemäß SRC- / DST- Typ; Schwarze Pfeile: Aktueller Wert von `suc`; Zahlen: Werte von `minId`

Tabelle 23.1.: Eigenschaften verschiedener Tile-Größen sowie daraus resultierender Ressourcenverbrauch und mögliche Auslastung bei Verwendung einer GPU mit Compute-Capability 3.0. Jedes Work-Item verarbeitet acht Segmente. Abkürzungen: S-Mem: Shared-Memory, WG: Work-Group

	S-Mem je WG	Work-Groups je SM	S-Mem genutzt	Threads je WG	Threads je SM	Thread Auslastung
32 x 8	4 kB	12	100 %	128	1536	75 %
32 x 16	8 kB	6	100 %	256	1536	75 %
32 x 32	16 kB	3	100 %	512	1536	75 %
64 x 8	8 kB	6	100 %	256	1536	75 %
64 x 16	16 kB	3	100 %	512	1536	75 %
64 x 32	32 kB	1	2/3	1024	1024	50 %

23.2. Einfluss der Tile-Größe

Zunächst muss eine geeignete Tile-Größe gefunden werden. Mögliche bzw. sinnvolle Tile-Größen ergeben sich unter Beachtung der Funktionsweise und der begrenzten Ressourcen einer GPU. Um günstige Zugriffsmuster auf den globalen Speicher zu bekommen und die Warp-Divergenz möglichst gering zu halten, muss die Tile-Breite ein ganzzahliges Vielfaches von 32 für eine Nvidia GPU sein. Weil auch Daten des Typs `char` geladen werden, kann eine Breite > 32 vorteilhaft sein, weshalb wir Tiles mit Breite $\geq$ Höhe konfigurieren. Zusätzlich ist die maximale Größe eines Tiles durch das vorhandene Shared-Memory limitiert.

Begründet durch die Funktionsweise der GPU lassen sich einige mögliche Tile-Größen festlegen, welche zusammen mit Eigenschaften bei der Ausführung auf einer GPU in Tabelle 23.2 eingetragen sind. Dabei verarbeitet jedes Work-Item zwei Pixel und somit acht Segmente.

Der entsprechende Kernel, `TilePointerJumps`, benötigt unabhängig von der Tile-Größe bei Übersetzung für Compute-Capability 3.0 31 Register. Dementsprechend ist die Parallelität hier nicht durch die Registerzahl sondern allein durch den Shared-Memory Verbrauch limitiert. Immer wenn die Tile-Größe so gewählt wird, dass der Shared-Memory-Bedarf je Work-Group ein ganzzahliger Teiler des verfügbaren Shared-Memory je SM ist, kann das Shared-Memory vollständig genutzt werden und 1536 Threads können je SM gestartet werden. Dies entspricht einer Auslastung von 75 Prozent. Die einzige Ausnahme stellt die größte mögliche Tile-Konfiguration, 64 x 32 Pixel, dar. Eine Work-Group benötigt hier 32 kB der verfügbaren 48 kB, sodass nur eine Work-Group je SM gestartet werden kann. Infolgedessen bleibt ein Drittel des Shared-Memorys ungenutzt und die Thread-Auslastung beträgt lediglich 50 Prozent. Es ist hier ein nachteiliges Laufzeitverhalten zu erwarten. Evaluieren wir dieses, indem wir die Laufzeit des Kernels `TilePointerJumps` bei allen genannten Tile-Größen für die 4096 x 4096 Spirale messen. Die Ergebnisse sind in Abbildung 23.2 zu sehen. Tatsächlich fällt die Laufzeit bei Tile-Größe 64 x 32 deutlich höher aus als bei den anderen Konfigurationen.

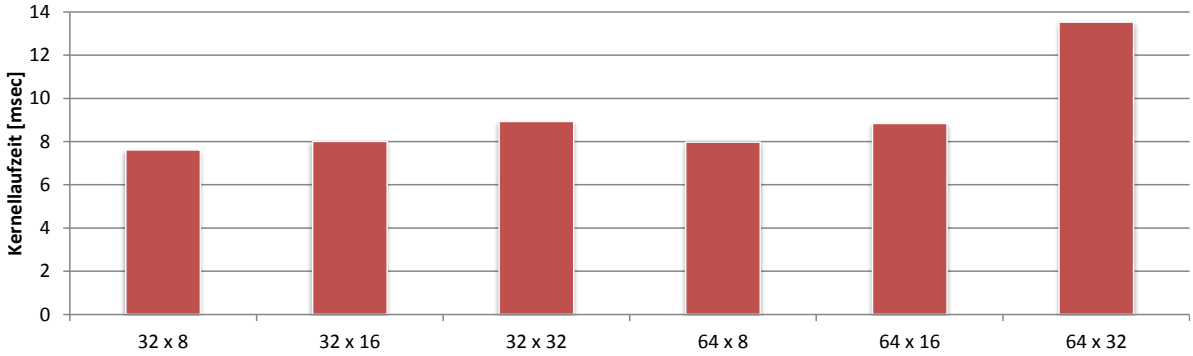


Abbildung 23.2.: Ermittelte Laufzeiten des Kernels `TilePointerJumps` bei Verarbeitung des 4096 x 4096 Spiral Datensatzes für verschiedene Einstellungen der Tile-Größe. Verwendete GPU: GTX 670

Eine Tile-Größe von 64 x 64 Pixeln würde 64 kB Shared-Memory benötigen und ist damit auf einer Nvidia GPU nicht ausführbar. Zwar stehen ab Compute-Capability 5.0 $\geq$ 64 kB je SM zur Verfügung, allerdings können weiterhin maximal 48 kB einer Work-Group zugewiesen werden. Somit ist auch hier die Tile-Größe 64 x 64 unmöglich, allerdings lassen sich in der Konfiguration 64 x 32 zwei (oder mehr) Work-Groups starten, sodass hier im Falle von Compute-Capability 5.x kein Auslastungsnachteil gegenüber anderen Tile-Größen zu erwarten ist.

Bedenken wir nun die Auswirkungen der obigen möglichen Tile-Größen unabhängig von den GPU-Eigenschaften. Je größer die Tiles gewählt werden, desto günstiger ist das Verhältnis von Randsegmenten zu denen im Inneren, sodass die verbleibende Anzahl der Segmente geringer ausfällt. Dem steht allerdings ein steigender Overhead-Faktor gegenüber. Ferner ist auch die Form der Tiles von Bedeutung. Offensichtlich ist im Falle quadratischer Tiles bei gleicher Anzahl enthaltener Pixeln das Verhältnis von Randsegmenten zu denen im Inneren am günstigsten. Ferner wird sich bei der Besprechung der Implementation zeigen, dass gleich große Kantenlängen in Höhe und Breite die Sonderfallbetrachtungen am Rand vereinfachen. Tabelle 23.2 gibt die Anteile verbleibender Segmente an der Gesamtzahl der Segmente sowie den Overhead-Faktor für jede Tile-Größe an. Stellen wir die Laufzeiten aus Abbildung 23.2 noch einmal als Throughputs, verarbeitete Segmente je Zeiteinheit, dar. Dabei sind lediglich die Segmente im Inneren gemeint, sodass sich je nach Tile-Größe zwischen ca. 92 und 97 Prozent der Gesamtsegmente ergeben. Diese Darstellung ist in Abbildung 23.3 zu sehen. Auf den ersten Blick hat sich im Vergleich mit 23.2 wenig geändert. Der Throughput der 64 x 32 Tiles ist deutlich geringer als der aller anderen und auch im Falle der 32 x 32 Tiles erscheint er wenig günstig.

Ein ganz anderes Bild ergibt sich, wenn der Throughput für den vollständigen Algorithmus ermittelt wird. Die Ergebnisse sind in Abbildung 23.4 dargestellt. Den höchsten Throughput liefert die 32 x 32 Konfiguration, die 64 x 32 Variante hat alle Übrigen

Tabelle 23.2.: Device- und implementationsunabhängige Eigenschaften verschiedener Tile-Größen

	Segmente je Tile	Davon Randsegmente	Verarbeitet je Tile	Anteil verbleibender Segmente	Overhead Faktor
32 x 8	1024	80	944	7,8125 %	10
32 x 16	2048	96	1952	4,6875 %	11
32 x 32	4096	128	3968	3,125 %	12
64 x 8	2048	144	1904	7,03125 %	11
64 x 16	4096	160	3936	3,90625 %	12
64 x 32	8192	192	8000	2,34375 %	13

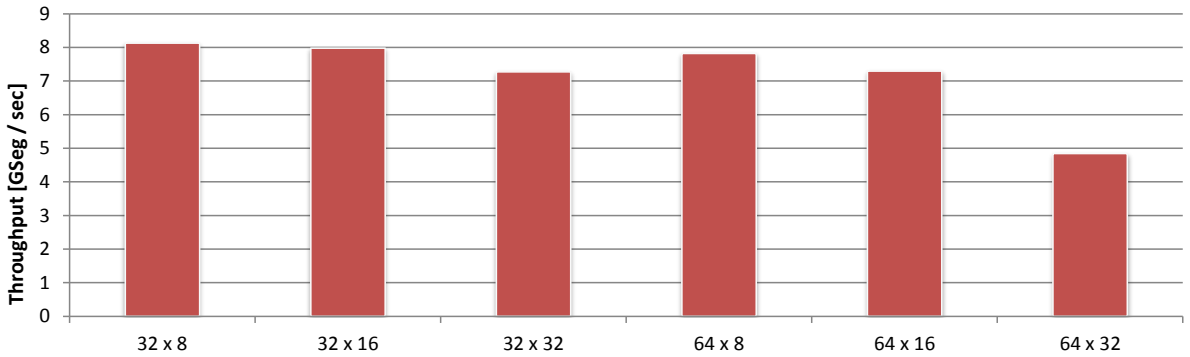


Abbildung 23.3.: Ermittelte Throughputs, angegeben in verarbeiteten (nicht Rand-) Segmenten je Zeiteinheit, des Kernels `TilePointerJumps` bei Verarbeitung des 4096 x 4096 Spiral-Datensatzes für verschiedene Einstellungen der Tile-Größe. Verwendete GPU: GTX 670

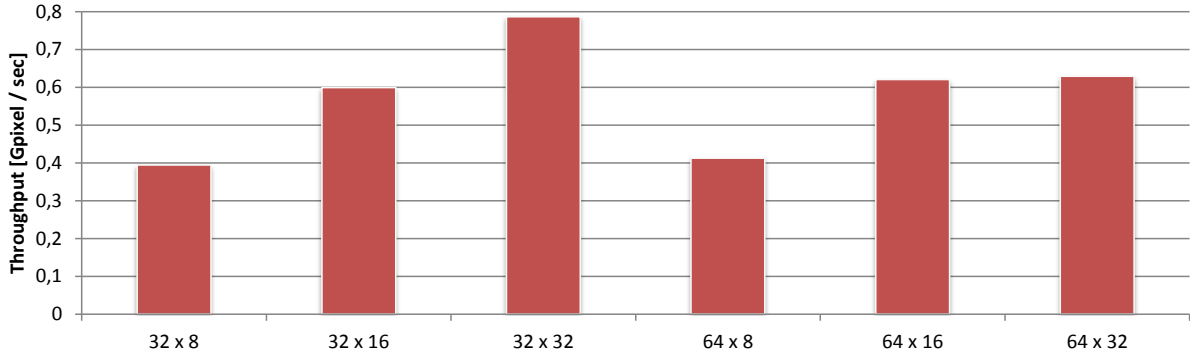


Abbildung 23.4.: Ermittelte Throughputs des vollständigen Algorithmus bei Verarbeitung des 4096 x 4096 Spiral-Datensatzes für verschiedene Einstellungen der Tile-Größe des Kernels `TilePointerJumps`. Verwendete GPU: GTX 670

eingeholt und die 'flachen' Tiles fallen weit zurück. Die Ursache ist die Restproblemgröße, welche weiterhin außerordentlich aufwändig verarbeitet werden muss. Diese wird im Falle der 32 x 32 Tiles (3,125 Prozent) und der 64 x 32 Tiles (2,34375 Prozent) um bis zu Faktor drei kleiner als etwa bei den 32 x 8 Tiles. Der Throughput des Algorithmus bei Verwendung der 32 x 32 Tile Konfiguration ist somit am höchsten, weil das Verhältnis aus Throughput des `TilePointerJumps` Kernels und der Problemverkleinerung durch ebendiesen hier am günstigsten ist. Im Folgenden wird somit ausschließlich diese Tile-Konfiguration verwendet.

23.3. Frühzeitiges Ende der Pointer-Jumps

In den bisher beschriebenen Varianten wird die Anzahl der auf alle enthaltenen Segmente anzuwendenden Pointer-Jumps durch die maximal mögliche Segmentzahl abgeschätzt. Dies ist für die entsprechende `for`-Schleife als Compile-Zeit-Konstante eingetragen und mit einem `#pragma unroll` Statement versehen. Dieser Overhead-Faktor ist bei der gewählten Tile Größe 12. Gegenstand dieses Kapitels ist die Besprechung und Evaluation von Möglichkeiten der Identifikation solcherart Situationen, in denen weniger Pointer-Jumps ausgeführt werden können.

Eine Möglichkeit ist die Limitierung durch die Anzahl tatsächlich vorliegender Segmente. Dazu wird im `extractSegs` Kernel für jedes einzelne Tile mit einem dort integrierten Reduction-Algorithmus deren Anzahl `segCntTile` bestimmt. Infolgedessen erwarten wir ein Ansteigen der Laufzeit dieses Kernels. Der `TilePointerJumps` Kernel stellt dann in jeder Work-Group die Anzahl der Schleifenaufrufe gemäß des jeweiligen Wertes von $\lceil log_2(segCntTile) \rceil$ ein. Die Technik dieser Variante nennen wir **Limit by SegCnt**, die Ausgangsfassung bezeichnen wir als **Fixed**.

Die **Limit by SegCnt** Technik lässt sich um eine Abfrage ergänzen, mit Work-Groups, welche kein Segment beinhalten, unmittelbar nach dem Start beendet werden. Diese Abfrage kann ebenfalls im Kernel `passLabels` integriert werden. Davon kann man sich bei sehr geringem Zusatzaufwand einen erheblichen Vorteil versprechen, wenn viele Tiles gar keine Segmente beinhalten. Die Technik dieser Variante nennen wir **Limit & terminate by SegCnt**.

Beide Varianten liefern eine gute Abschätzung, wenn jeweils alle Segmente aller Tiles innerhalb des jeweiligen Tiles einen einzigen Linienzug bilden. Im Falle mehrerer unabhängiger Linked-Lists je Tile wird die Anzahl der nötigen Pointer-Jumps weiterhin überschätzt.

Alternativ dazu kann zur Ausführungszeit des Kernels `TilePointerJumps` erkannt werden, wenn keine weitere Ausführung der Pointer-Jumps nötig ist. Dies ist der Fall, wenn für alle Segmente `s` eines Tiles bereits vor Ausführung des Pointer-Jumps gilt: `s.minId` = `s.suc.minId`. Weil dies erst nach Ausführung des Pointer-Jumps bekannt ist (wir wollen schließlich keinen extra Zugriff auf `s.suc`), muss somit immer ein Pointer-

Jump zu viel ausgeführt werden. Ausnahme davon ist das immer noch mögliche Erreichen des fest eingestellten Schleifenendes im Worst-Case.

Implementieren lässt sich das folgendermaßen: Es gibt eine reservierte Speicheradresse im Local-Memory, an der notiert wird, ob ein weiterer Pointer-Jump für alle Segmente des Tiles erforderlich ist. Am Anfang jeder Iteration der Hauptschleife wird der Wert an diese Adresse, gefolgt von einer lokalen Synchronisation, auf `FALSE` gesetzt. Dann führt jedes Segment einen Pointer-Jump aus und wertet obige Bedingung aus. Gilt sie nicht, wird ein `TRUE` an die Adresse geschrieben. Ein weiterer Synchronisationspunkt stellt sicher, dass alle Elemente ihren Pointer-Jump ausgeführt haben. Falls dann auch nur ein Segment den Wert `TRUE` geschrieben hat, wird eine weitere Schleifeniteration ausgeführt. Die Technik dieser Variante bezeichnen wir **Dynamic Limit**.

Hinweise zur Implementation: Für die Technik wird kein eigener Speicher verwendet, da dies ein ganzzahliges Aufteilen des Shared-Memory auf Work-Groups verhindern würde. Details dazu folgen später. Außerdem kann die Wahrscheinlichkeit des Eintretens der Abbruchsbedingung in den ersten Iterationen als sehr gering eingeschätzt werden. Deshalb wird dieser Test erst ab der vierten Point-Jump-Iteration durchgeführt. Diese Zahl ist fest im Code angegeben und wird nicht etwa für einzelne Datensätze angepasst.

Evaluieren wir nun die besprochenen Ansätze anhand der Datensätze Zeros, Ones, Spiral und Quads verschiedener Größe, jeweils in der Datenauflösung von 4096 x 4096. Die ermittelten Throughputs sind in Abbildung 23.5 dargestellt. Wir beobachten dort:

- Die Variante Fixed liefert in keinem Fall die höchsten Throughputs. Eine Erklärung dafür ist, dass in keinem Datensatz alle Segmente existieren.
- Der Ansatz Dynamic Limit liefert in den Datensätzen Quads_2, Quads_8 und Spiral die höchsten Throughputs. Das sind genau diejenigen Fälle, in denen mehrere unabhängige Linked-Lists in jedem Tile existieren.
- Existiert genau eine (Quads_32) oder in 75 Prozent aller Quads genau eine, sonst keine (Quads_128) Linked-List, ergeben sich teilweise etwas höhere Throughputs für die Limit by SegCnt Techniken als für Dynamic Limit. Deutlich höher fallen diese erst aus, wenn der Anteil leerer Tiles hoch ist (Ones, Zeros).
- Erst dann kann sich auch die Limit & terminate by SegCnt Variante von Limit by SegCnt absetzten.

Betrachten wir zum besseren Verständnis die Laufzeiten der einzelnen Teile des Algorithmus bei Verarbeitung zweier Datensätze (Spiral, Ones) mit entgegengesetztem Verhalten. Diese sind in Abbildung 23.6 zu sehen. In beiden Datensätzen ist im Falle der Limit by SegCnt Ansätze die Laufzeit des `extractSegs` Kernels, bedingt durch die zusätzliche Bestimmung der Segmentzahl, höher. Beim Spiral-Datensatz entsprechen diese Mehrkosten in etwa der verringerten Laufzeit des `TilePointerJumps` Kernels, wodurch sich in der Summe fast keine Änderung ergibt. Dagegen bewirkt der Dynamic Limit Ansatz eine deutliche Verringerung der Laufzeit dieses Kernels.

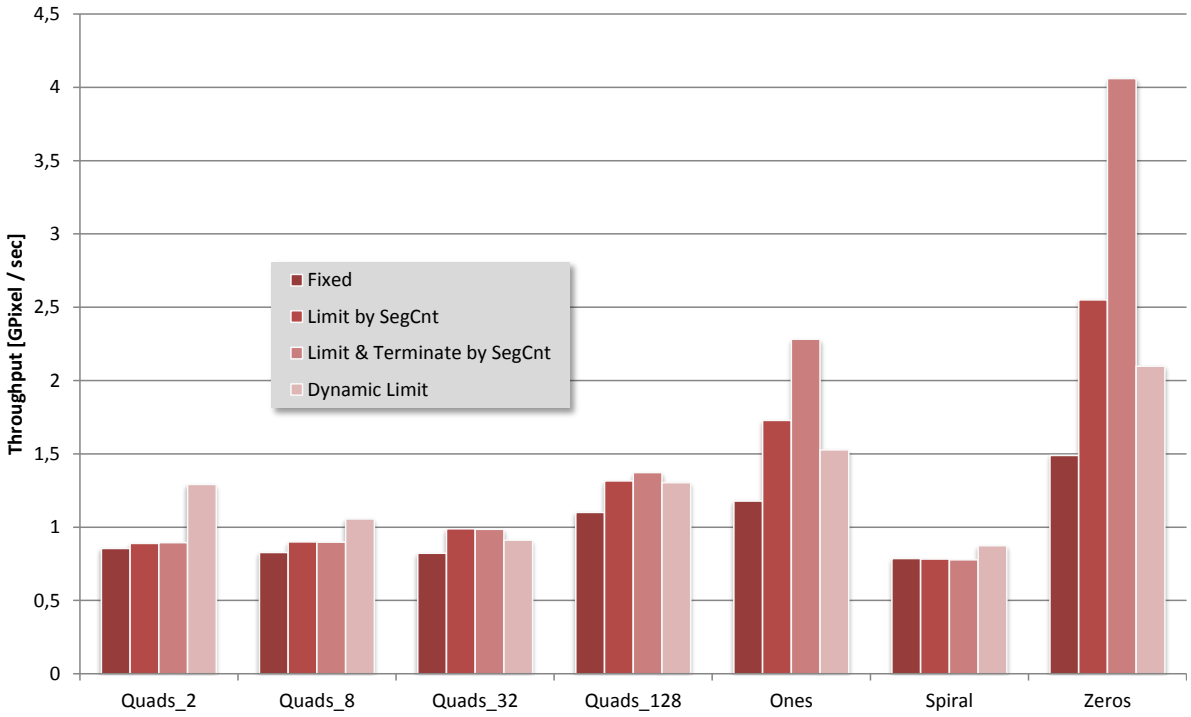


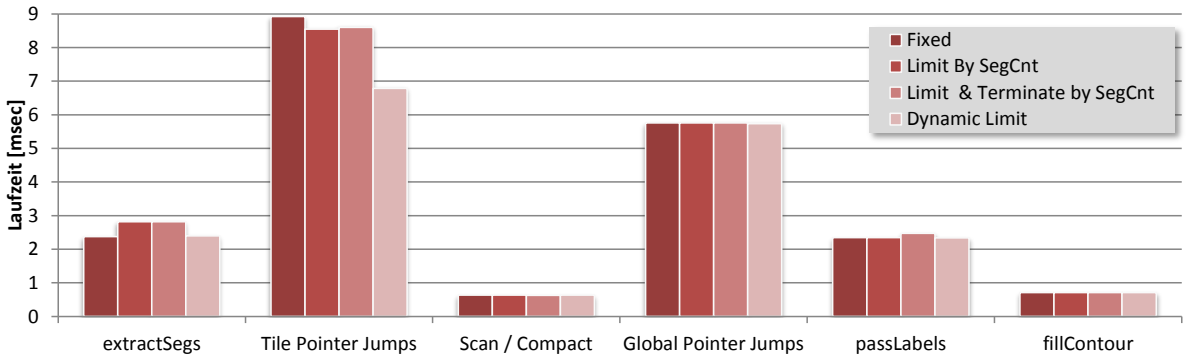
Abbildung 23.5.: Ermittelte Throughputs des Gesamtalgorithmus bei verschiedenen Datensätzen und unter Verwendung des Kernels `TilePointerJumps` mit verschiedenen Abbruchkriterien. Die Datenauflösung beträgt 4096 x 4096 und verwendet GPU ist eine GTX 670.

Unabhängig davon kann an dieser Stelle der weiterhin hohe Anteil an der Laufzeit der nach der Tile-Verarbeitung verbleibenden Segmente (Global Pointer-Jumps) bei Verarbeitung der Spirale bemerkt werden.

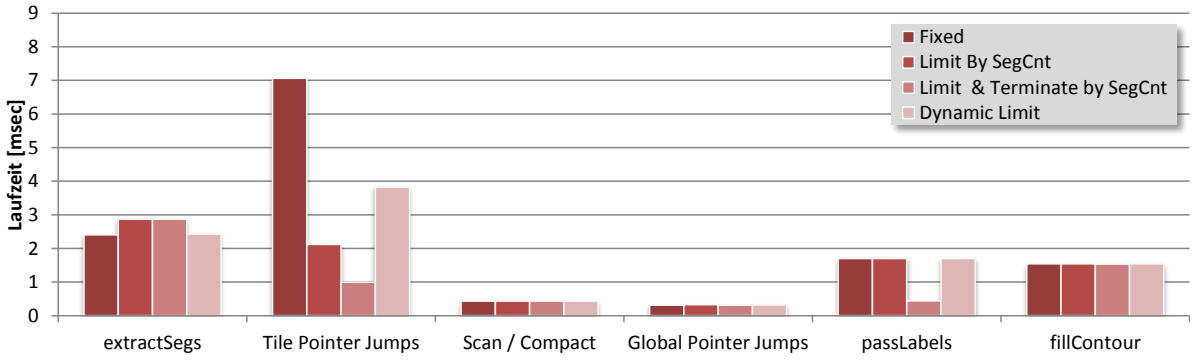
Bei Betrachtung des Ones-Datensatzes ergibt sich ein anderes Bild. Hier fällt die Laufzeit des `TilePointerJumps` Kernels bei dem Limit by SegCnt Ansatz, insbesondere im Falle der Limit & terminate by SegCnt Variante deutlich geringer aus. Letztere bewirkt zusätzlich eine deutliche Verkürzung der Laufzeit der `passLabels` Kernels, da hier ebenfalls die Ausführung einer Work-Group abgebrochen werden kann, wenn für diese keine Segmente vorliegen.

Insgesamt ist somit je nach Datensatz eine andere Variante geeigneter. Weil der Dynamic Limit Ansatz aber außer bei Extremfällen nie deutlich schlechter abschneidet als die Limit by SegCnt Varianten, verwenden wir nachfolgend ausschließlich diesen.

Auch wenn die Wahl der Tile-Größe und des Abbruchkriteriums bereits feststehen, wollen wir das Verhalten des Kernels `TilePointerJumps`, speziell hinsichtlich des Einflusses des Overhead-Faktors, genauer verstehen. Dazu verändern wir in einem Experiment in der Fixed Variante des Kernels manuell die Anzahl der ausgeführten Pointer-Jump-Schritte. Dies führen wir mit dem Spiral-Datensatz durch, bei dem das unvollstän-



(a) Spiral Datensatz



(b) Ones Datensatz

Abbildung 23.6.: Vergleich der Laufzeiten der einzelnen Teile des Algorithmus bei Verwendung des Kernels `TilePointerJumps` mit verschiedenen Abbruchkriterien. Datenauflösung: 4096 x 4096, GPU: GTX 670

dige Verarbeiten nicht zu einer Veränderung des schreibenden Datenmusters führt. Die enthalten Werte sind dann ggf. falsch, sodass nachfolgende Kernel nicht mehr ausgeführt werden können. Die Laufzeiten des Kernels für gar keine bis alle (gemäß Fixed Kriterium) Pointer-Jumps sind Abbildung 23.7 zu entnehmen. Zum Vergleich ist außerdem die Laufzeit des Kernels mit Dynamic Limit Abbruchkriterium eingetragen. Zunächst fällt die vergleichsweise hohe Laufzeit bei Ausführung keines einzigen Pointer-Jumps auf, die bereits mehr als ein Drittel der maximalen, bei 12 Pointer-Jumps auftretenden Laufzeit, in Anspruch nimmt. Somit ist das Laden und Interpretieren der Daten und, gefolgt von einem Synchronisationspunkt, erneute Interpretieren und Schreiben der Daten bereits für einen Großteil der Laufzeit verantwortlich. Werden ein oder mehrere Pointer-Jump-Schritte ausgeführt, steigt die Laufzeit immer weiter. Dabei fällt die höhere Zunahme bei den ersten Schritten auf. Eine mögliche Erklärung ist die Deaktivierung von Segmenten, welche aus dem Tile herauszeigen. Dies sind im Falle der Spirale, sobald ihre Verarbeitung abgeschlossen ist, alle.

Der Kernel mit dem Dynamic Limit Test benötigt eine Laufzeit knapp oberhalb von 6 Schritten bei der festen Variante. Tatsächlich benötigen bei der Spirale die meisten Tiles 5 und manche 6 Schritte (berechnet mit Limit by SegCnt Variante). Die Zusatzkosten

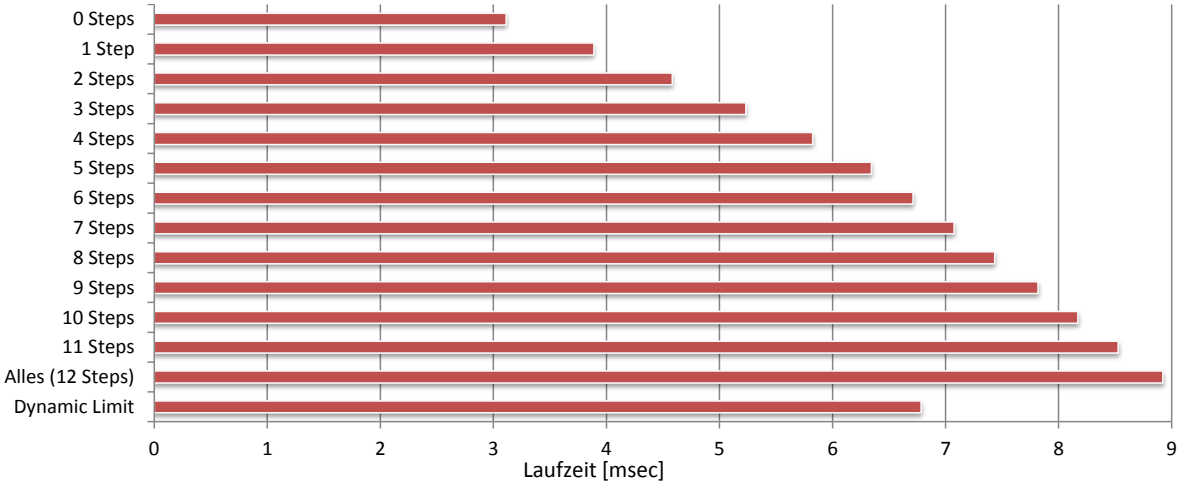


Abbildung 23.7.: Ermittelte Laufzeiten des Kernels `TilePointerJumps` bei Verarbeitung des 4096 x 4096 Spiral-Datensatzes mit manuell eingestellter Anzahl von Pointer-Jump-Schritten. Zum Vergleich angegeben ist der selbe Kernel mit Dynamic Limit Einstellung. Verwendete GPU: GTX 670

für den Dynamic Limit Test sind somit als annehmbar einzustufen.

Schätzen wir abschließend den Overhead-Faktor in der Praxis ein. Dazu wählen wir die Fixed Variante, welche einen Pointer-Jump für alle Segmente ausführt, als Basis. Die Laufzeit des Kernels mit Dynamic Limit Test liegt um lediglich Faktor 1,75 darüber. Zusätzlich liegt der Anteil des Kernels `TilePointerJumps` an der Gesamtlaufzeit, je nach Variante, bei etwa 40 Prozent. Insgesamt kann der Overhead-Faktor in der Praxis somit als deutlich kleiner als 2 eingeschätzt werden. Im Gegensatz zu dem theoretischen Overhead-Faktor 12 lässt dies den Tile-basierten Ansatz in der Praxis als weitaus weniger unelegant erscheinen, als er möglicherweise zunächst wirkt.

23.4. Mehrere Tile-Generationen

In den bisherigen Abschnitten ist eine Verbesserung der Laufzeit durch das Tile-basierte Schema deutlich geworden. Sofern eine gute Nutzung der GPU-Ressourcen möglich ist, führt ein größerer Anteil der verarbeiteten Segmente zu einer Verbesserung des Throughputs. Dies wird durch eine Vergrößerung der Tiles erreicht, wobei die maximale Tile-Größe durch das Shared-Memory begrenzt ist. Um weitere Segmente durch eine vergleichbare Technik zu verarbeiten, können die verbleibenden Randsegmente der Tiles erneut zu Tiles zusammengefasst werden. Wir sprechen nachfolgend von Tiles der zweiten Generation und ergänzen die entsprechenden Kernel zur Unterscheidung um ein `G1` oder `G2`.

Analog zu den Tiles der ersten Generation sind hier auch in der zweiten Generation verschiedene Tile-Größen denkbar, welche einerseits unterschiedlich gute GPU-Auslastungen ermöglichen und andererseits hinsichtlich der Verringerung der Restproblemgröße variieren. Weiterhin sind zusätzliche Generationen denkbar, welche allerdings eine immer

geringere Verkleinerung ermöglichen, da das Verhältnis von Randsegmenten zu denen im Inneren immer ungünstiger wird.

Beides wird im Rahmen dieser Arbeit jedoch nicht evaluiert. Stattdessen wird die Tile-Größe der zweiten Generation zu 8 x 8 Tiles der ersten Generation gewählt. Ein Tile der zweiten Generation beinhaltet demnach $8 \cdot 8 \cdot 128 = 8192$ Segmente und es repräsentiert einen 256 x 256 Pixel großen Bereich. Die Anzahl der Randsegmente beträgt bei dieser Wahl somit $4 \cdot 256 = 1024$, die Problemgröße verringert sich somit um Faktor acht und damit weniger als in der ersten Generation (Faktor 32).

Für die Wahl dieser Tile-Größe spricht die gute Verkleinerung im Vergleich mit anderen möglichen Konfigurationen und ihre quadratische Form. Nachteilig ist die ungünstige Nutzung der Shared-Memory Ressourcen bei den Architekturen Fermi und Kepler (Maxwell nicht), welche identisch ist mit den 64 x 32 Tiles der ersten Generation.

Vergleichen wir nun die Implementation, welche zwei Tile-Generationen verwendet, mit der bisherigen Implementation hinsichtlich der Laufzeiten einzelner Teile des Algorithmus bei Verarbeitung des 4096 x 4096 Spiral-Datensatzes. Die Ergebnisse sind in Abbildung 23.8 dargestellt. Der 2G-Ansatz setzt zwei zusätzliche Kernel ein, welche kei-

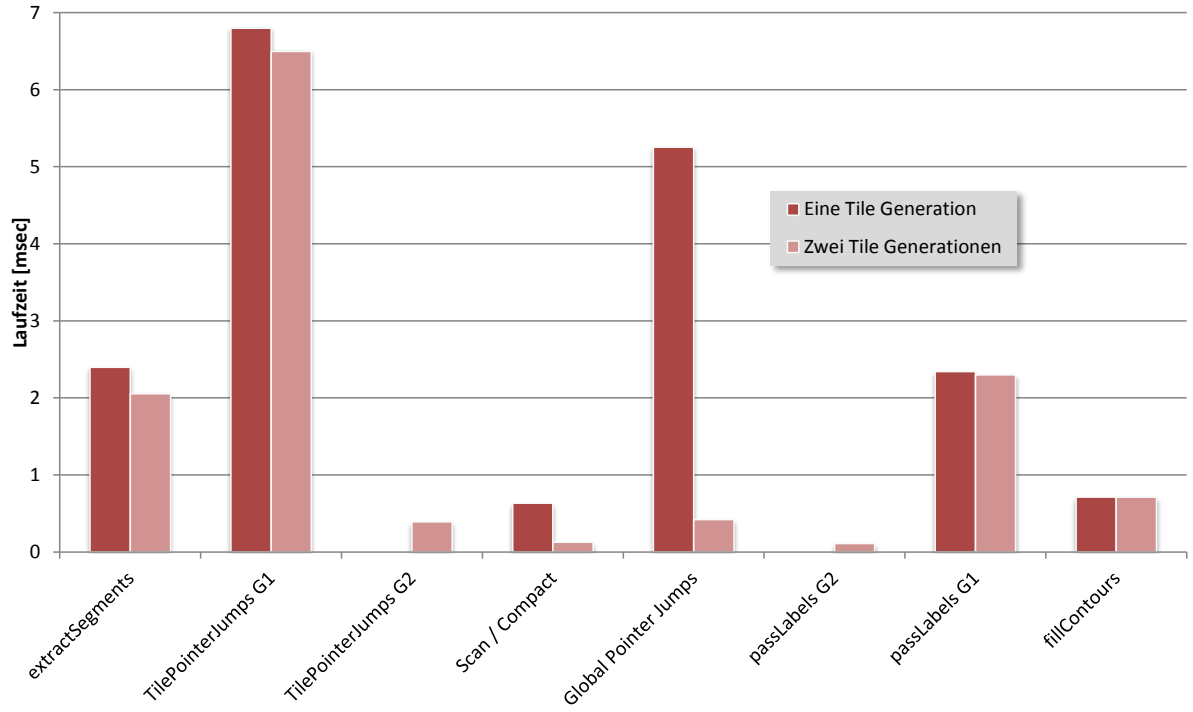


Abbildung 23.8.: Vergleich der Laufzeiten der einzelnen Teile der Ansätze mit einer und zwei Tile-Generationen. Daten: 4096 x 4096 Spiral, GPU: GTX 670

ne Entsprechung im 1G-Ansatz haben. Diese verarbeiten die Randsegmente der ersten Tile-Generation (`TilePointerJumpsG2`) bzw. reichen die finalen Label innerhalb der Tiles der zweiten Generation weiter (`passLabelsG2`). Hinsichtlich der Messwerte fällt die geringere Laufzeit der Kernel `TilePointerJumpsG1` und `extractSegments` beim zwei Tile-Generationen Ansatz auf. Dies hängt mit dem intern veränderten Informationsko-

dierungsschema der Segmente innerhalb der Tiles zusammen und wird an dieser Stelle ignoriert. Interessant sind dagegen die erheblich verringerten Laufzeiten derjenigen Kernel, welche mit Scan/Compact und den globalen Pointer-Jumps zusammenhängen. Diese verringern sich erheblich und wiegen den Aufwand durch die beiden zusätzlichen Kernel mehr als auf. Aufgrund des geringen Anteils der Kernel `TilePointerJumpsG2` und `passLabelsG2` an der Gesamtlaufzeit wirkt sich außerdem die nicht ideale Auslastung der GPU bei deren Ausführung nicht sehr negativ aus. Der 2G-Ansatz setzt zwei zusätzliche Kernel ein, welche keine Entsprechung im 1G-Ansatz haben. Diese verarbeiten die Randsegmente der ersten Tile-Generation (`TilePointerJumpsG2`) bzw. reichen die finalen Label innerhalb der Tiles der zweiten Generation weiter (`passLabelsG2`). Hinsichtlich der Messwerte fällt die geringere Laufzeit der Kernel `TilePointerJumpsG1` und `extractSegments` beim zwei Tile-Generationen Ansatz auf. Dies hängt mit dem intern veränderten Informationskodierungsschema der Segmente innerhalb der Tiles zusammen und wird an dieser Stelle ignoriert. Interessant sind dagegen die erheblich verringerten Laufzeiten derjenigen Kernel, welche mit Scan/Compact und den globalen Pointer-Jumps zusammenhängen. Diese verringern sich erheblich und wiegen den Aufwand durch die beiden zusätzlichen Kernel mehr als auf. Aufgrund des geringen Anteils der Kernel `TilePointerJumpsG2` und `passLabelsG2` an der Gesamtlaufzeit wirkt sich außerdem die nicht ideale Auslastung der GPU bei deren Ausführung nicht sehr negativ aus.

Vergleichen wir jetzt beide Ansätze hinsichtlich des erreichbaren Throughputs bei Verarbeitung des Spiral-Datensatzes in verschiedenen Auflösungen. Dies ist in Abbildung 23.9 dargestellt. Wie erwartet steigt der Throughput zumindest in höheren Auflösungen

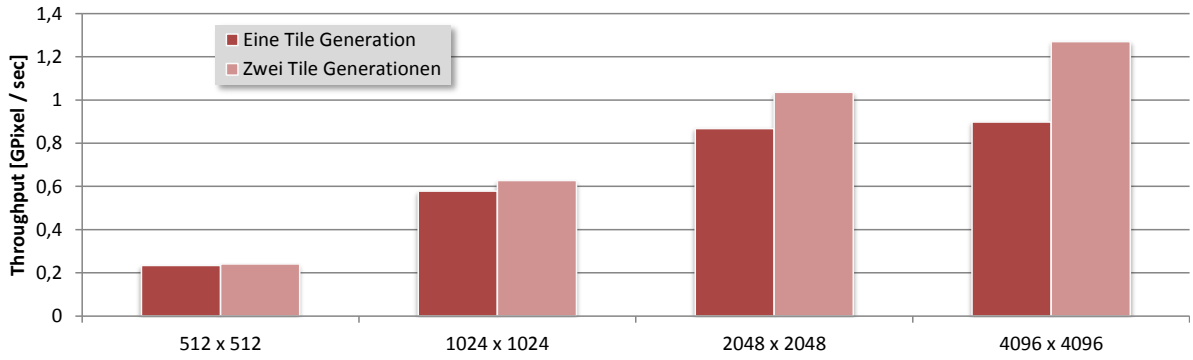


Abbildung 23.9.: Vergleich der Throughputs der Ansätze mit einer und zwei Tile-Generationen bei Verarbeitung des Spiral-Datensatzes in verschiedenen Auflösungen. GPU: GTX 670

deutlich bei Verwendung der zweiten Tile-Generation. Dieser Ansatz mit der oben angegebenen Tile-Größe wird somit übernommen.

Ausgehend von der Wahl der Tile-Größen in der ersten und zweiten Generation, ist in einer denkbaren dritten Generation noch eine Konfiguration bestehend aus 4 x 2 Tiles der zweiten Generation möglich. Je Tile sind wieder 8192 Segmente enthalten, von denen 3072 Randsegmente sind. Die Restproblemgröße lässt sich so bei dieser Wahl zusätzlich

um Faktor 8/3 verkleinern. Dies, sowie noch weitere Generationen mit weiter sinkendem Verkleinerungsfaktor, wird aber im Rahmen dieser Arbeit nicht mehr implementiert und evaluiert.

23.5. Implementations-Details

Nachfolgend werden weitere Details der Implementation und insbesondere der Optimierungen des Ansatzes Impl IV besprochen. Aufgrund dessen Bedeutung und der Höhe des Optimierungsgrades fällt dies ungleich umfangreicher aus, als im Falle der übrigen Implementationsstrategien. Beginnen wir mit einigen Ergänzungen hinsichtlich der Funktionsweise der Implementation, welche in den vorherigen Abschnitten noch nicht genannt sind.

Die Kernel `extractSegments` und `TilePointerJumpsG1` werden vereinigt zu einem einzigen Kernel, den wir `extractSegs_N_TilePJ_G1` nennen. Auf diese Weise kann die globale Synchronisation zwischen beiden Kerneln durch eine Lokale ersetzt werden. Außerdem können dann alle lokalen Daten der Tiles allein über das Local-Memory ausgetauscht werden oder können, noch günstiger, im Private-Memory verbleiben. Bei Verarbeitung des Spiral-Datensatzes in 4096 x 4096 Pixeln durch die GTX 670 ist die Summe der Laufzeiten der einzelnen Kernel um ca. 15 Prozent höher als die des vereinigten Kernels.

Der Optimierungsgrad des Kernels `TilePointerJumpsG2` ist aufgrund seines geringen Anteils an der Gesamtlaufzeit deutlich geringer als der des obigen Kernels. Hier ist z.B. kein Test für frühzeitiges Beenden der Pointer-Jumps implementiert.

Datenstrukturen

Alle Datenstrukturen für Pixel sind identisch mit denen der vorherigen Varianten.

Bedingt durch das auf den Tiles basierende Schema kann für Segmente für viele Datenstrukturen ein kleinerer Datentyp verwendet werden, da der lokale Wertebereich geringer ist. Außerdem werden manche Datenstrukturen lediglich für diejenigen Segmente benötigt, welche nach Ausführung des Kernels `extractSegs_N_TilePJ_G1` noch weiter verarbeitet werden müssen.

Andererseits sind zusätzliche Datenstrukturen erforderlich, um die Zustände der selben Segmente in den verschiedenen Tile-Generationen abzubilden. Diese tragen aber aufgrund der abnehmenden Problemgröße nicht sehr viel zum Speicherverbrauch bei. Die Datenstrukturen für Segmente im Einzelnen:

`segsTile1_Suc:` Nachfolger eines Segments innerhalb eines Tiles der ersten Generation. Verwendet lokalen Adressraum. Größe der Datenstruktur: $4 \cdot N$. Datentyp: `short`

`segsTile2_Data:` Verschiedene Daten (suc in 2G Tiles, minId) derjenigen Segmente, die in 2G-Tiles weiter verarbeitet werden. Größe der Datenstruktur: $N/8$. Datentyp: `int`

`segsTile2_SucInTile1`: Nachfolger in 1G-Tiles, derjenigen Segmente, die in 2G-Tiles weiter verarbeitet werden. Größe der Datenstruktur: $N/8$. Datentyp: `short`

`segsTile2_Props`: Eigenschaften, z.B. Existenz, der 2G-Segmente. Anzahl: $N/8$. Datentyp: `char`

`suc, minId, props`: Diese Datenstrukturen existieren mit analoger Bedeutung wie in vorherigen Varianten auch in voller Genauigkeit (`int`, `int`, `char`) für die Pointer-Jumps im globalen Speicher. Es wird ebenfalls für Scan / Compact eine Doublebuffer-Technik verwendet. Da sie erst nach den G2-Pointer-Jumps angewendet werden, ist der Speicherverbrauch um Faktor 256 geringer.

Insgesamt beträgt der Speicherverbrauch je Pixel knapp 20 Byte und fällt damit deutlich geringer aus als bei den vorherigen Varianten

Der Kernel extractSegs_N_TilePJ_G1

Schauen wir uns nun den in Impl IV verwendeten Kernel `extractSegs_N_TilePJ_G1` genauer an. Er ist für einen Großteil der gemessenen Laufzeit verantwortlich und gibt einen Überblick über die wichtigsten Techniken der Implementation.

Das Listing 23.2 (ab Seite 273) beinhaltet den zugehörigen OpenCL C Code. Zusätzlich sind die durch diesen Kernel verwendeten Makros, Konstanten und eine Funktion, und zwar genau die benötigten, in Listing 23.1 gegeben. Es gibt geringe Redundanzen bei den Konstanten und Makros mit vorherigen Listings, dafür sind Listings 23.1 und 23.2 aber zusammengenommen unverändert compilier- und verwendbar.

Der Kernel stellt den ersten Schritt der Implementation des CCL-Algorithmus dar, sodass ausschließlich die `pixel_Classification` Datenstruktur initialisiert sein muss.

Gerade dieser Codeabschnitt dient primär dazu, einen Hinweis auf die genaue Funktionsweise und vor allem den Optimierungsgrad der Implementation zu geben. Letzterer kann als recht hoch eingeschätzt werden. Dementsprechend ist die Lesbarkeit des Codes immer der Performance untergeordnet. Nimmt man den Umfang des Kernels hinzu, handelt es sich hier im Vergleich mit allen anderen Listings dieser Arbeit um ein weitaus komplexeres aber auch ungleich wichtigeres Codestück zur Beurteilung der Implementation.

Eine Vereinfachung, z.B. Aufteilung des Kernels in mehrere Kernel, stellt keine Option dar, weil eine solche Vorgehensweise massive negative Auswirkungen auf das Laufzeitverhalten hat. Dementsprechend würde der Zweck des Listings, die Beurteilung eben dieses Verhaltens, ad absurdum geführt.

Der markierte Abschnitt 'part I: Extract Segsments' des Kernels `extractSegs_N_TilePJ_G1` in Listing 23.2 wird in abgewandelter Form auch von den Implementationsstrategien I, II und III verwendet, um die Konturesegmente zu extrahieren. Dort entfallen allerdings die Betrachtung der Tile-Ränder und alle Ergebnisse müssen in das Global-Memory zurück geschrieben werden.

Es folgt ein Überblick über die verwendeten Optimierungen, welche im Kernel `extractSegs_N_TilePJ_G1` verwendet werden, von denen einige in den vorherigen Abschnitten

bereits erläutert sind:

Shared-Memory-Verbrauch: Wie zuvor festgestellt, ist eine gute Nutzung des Shared-Memory unabdingbar für ein günstiges Laufzeitverhalten. Später (Abschnitt 23.7, Seite 305) wird noch einmal deutlich, welche erheblichen Performance-Einbrüche möglich sind, wenn die Implementation auch nur ein einziges Byte Shared-Memory mehr verwendet, als hier der Fall. Es wird erheblicher Aufwand betrieben, dieses zu verhindern.

Zusammengefasste Attribute: Oft werden verschiedene Eigenschaften zusammengefasst in einem einzelnen atomaren Datentyp hinterlegt. Auf diese Weise kann z.B. ein Kontursegment vollständig in einem 32-Bit Integer kodiert werden. Dies hat vor allem hinsichtlich deren Hinterlegung im Shared-Memory Vorteile. Zum einen kann so der Shared-Memory-Verbrauch minimiert werden und zum anderen werden ebenfalls die Zugriffe auf das Shared-Memory minimiert. Schließlich kann auf diese Weise ein Segment mit nur einem Zugriff geladen oder geschrieben werden. Weil diese Kodierung drei Auswirkungen hat (Shared-Memory-Verbrauch, Anzahl Zugriffe darauf, weniger Synchronisierung notwendig), sind Auswirkungen schwer experimentell zu isolieren. In der Summe ist ihr Beitrag zur Laufzeitverbesserung größer als bei den meisten anderen Optimierungen.

Unabhängig davon werden auch andere Eigenschaften im Private-Memory fast ausschließlich bitweise codiert. So werden Register gespart und oft können verschiedene Eigenschaften durch einen einzigen Vergleich mit einer Konstante (ggf. nach vorheriger Anwendung einer Bitmaske) überprüft werden. Durch letzteres lassen sich vielfach zusätzliche if-Abfragen vermeiden, welche auf einer GPU oft ungünstig sind.

Eine Folge dieser Vorgehensweise ist, dass praktisch alle Operationen durch Makros ausgeführt werden müssen, um wenigstens ein Mindestmaß an Lesbarkeit zu gewährleisten.

Asynchrones Arbeiten: Bedingt durch die asynchrone Arbeitsweise einer GPU, ist es oft sinnvoll, Daten explizit früher anzufordern, als sie benötigt werden. Dem steht aber durch das 'unnötig' frühe Deklarieren und Initialisieren von Datenstrukturen ein möglicherweise erhöhter Registerverbrauch gegenüber.

Es kann außerdem sinnvoll sein, Daten explizit anzufordern, von denen nicht sicher ist, ob sie überhaupt benötigt werden. Beispiele sind Fälle, in denen der weitere Kontrollfluss davon abhängt. Hier ist es allerdings schwer, generelle Aussagen zu treffen. Solche Optimierungsentscheidungen sind zumeist experimentell getroffen. Dabei besteht auch die Gefahr der Optimierung auf bestimmte Datensätze.

Außerdem wird, wann immer möglich, ein unabhängiges Arbeiten innerhalb eines Work-Items forciert. Deshalb werden für unabhängige Elemente oft unabhängige Datenstrukturen verwendet, auch wenn alles in Einer zusammengefasst werden könnte. Das führt allerdings teilweise zu Redundanzen.

Redundanz im Private-Memory: Zusätzlich zu der soeben skizzierten Redundanz für unabhängiges Arbeiten werden Kopien der 'eigenen' Daten aus dem Shared-Memory im Private-Memory angelegt, um nur dann auf das Shared-Memory zugreifen zu müssen, wenn es unvermeidbar ist.

Tile-Größe: Die Tile-Größe 32 x 32 Pixel ist ideal für die Fermi und Kepler Architekturen. Diese Wahl ist im Abschnitt 23.2 begründet und evaluiert.

Pointer-Jumps dynamisch beenden: Auch diese Technik ist als Optimierung, ein Stück weit auch auf die Daten, zu sehen. Sie ist in Abschnitt 23.3 beschrieben und evaluiert.

Kernelvereinigung: Das Zusammenlegen der, inhaltlich verschiedenen, Kernel `extractSegs` und `tilePointerJumpsG1` stellt ebenfalls eine Optimierung dar.

Work-Item-Daten-Mapping Jedes Work-Item ist (primär) für zwei übereinander liegende Pixel und die darin enthaltenen Segmente verantwortlich. Es wird ein zweidimensionaler Index verwendet. Die nullte Dimension wird zur x- und die erste Dimension zur y-Indizierung genutzt. Es ergibt sich somit die LWS (32, 16, 1) und 512 Work-Items je Work-Group erzeugt.

Für die Verarbeitung von zwei Pixeln je Work-Item spricht die gute Ressourcen Nutzung im Falle von Nvidia GPUs. Würde alternativ jeder Thread einen Pixel verarbeiten, könnten nur zwei Work-Groups gestartet werden. Grund dafür ist die Limitierung durch die Thread-Anzahl. Eine weitere Folge daraus ist eine unvollständige Nutzung des Shared-Memory.

Generell bewirkt die Verarbeitung mehrerer Pixel in einem Work-Item allerdings auch andere Vorteile. So können in einem gewissen Umfang Private-Datenstrukturen und damit möglicherweise Register eingespart werden. Ein offensichtliches Beispiel ist die x-Koordinate, welche in allen übereinander liegenden Pixeln gleich ist. Ferner vereinfachen sich manche Fallunterscheidungen. So entfallen etwa Sonderbehandlungen hinsichtlich des oberen Tile-Randes in allen durch ein Work-Item verarbeiteten Pixeln mit Ausnahme des Obersten.

Auch denkbar ist die Verarbeitung von vier Pixeln je Work-Item. In diesem Fall können, bedingt durch die Limitierung durch das Shared-Memory, trotzdem nicht mehr Work-Groups gestartet werden. Als Folge halbiert sich die Thread-Auslastung.

Bank-Conflicts: Die beschriebene Zuordnung von Work-Items zu Segmenten stellt beim Zugriff auf 'eigene' Segmente sicher, dass je 32 konsekutive Work-Items auf 32 konsekutive 32-Bit Integer im Shared-Memory zugreifen. Somit werden hier Bank-Conflicts minimiert. Anders sieht es beim Zugriff auf die jeweils aktuellen Nachfolgesegmente aus, die sich an einer beliebigen Adresse befinden können. Somit sind in diesem Fall Bank-Conflicts möglich. Deren genauer Grad ist datenabhängig.

Datenunabhängige Fallunterscheidungen: Wenn möglich werden Fallunterscheidungen, speziell im Zusammenhang mit den Tile-Rändern, in Abhängigkeit von den Indices des jeweiligen Work-Items getroffen. Oft können so teure datenabhängige Fallunterscheidungen vermieden werden.

Vertikale Ränder: Informationen über die vertikalen Ränder liegen teilweise transponiert vor. Andernfalls würde es bei der abschließenden Vorverarbeitung der Ränder für die G2-Tiles zu maximaler Warp-Divergenz kommen. Um günstige Speicherzugriffsmuster zu ermöglichen und um eine Weiterverarbeitung der vertikalen Ränder zu vereinfachen, werden diese am Ende des Kernels in der transponierten Form in das Global-Memory zurück geschrieben.

Global-Memory: Die Zugriffe auf das Global-Memory sind nie datenabhängig und es wird immer auf 32 konsekutive Adressen zugegriffen. Generell werden im Vergleich mit den übrigen Implementationsstrategien wenig Daten aus dem Global-Memory geladen und wieder zurück geschrieben, da der für die Berechnungen notwendige Datenaustausch vollständig über das Local-Memory geschieht.

Sonstiges: Bei der Wertzuweisung wird der unter Performancegesichtspunkten günstigere ?-Operator einer `if-else` Konstruktion vorgezogen. Nvidia Grafikkarten arbeiten hinsichtlich des Private-Memory am effizientesten mit 32-Bit Datentypen. Daher werden diese oft vorgezogen, auch wenn ein kleinerer Datentyp ausreichen würde. Schleifen mit konstanter Iterationszahl werden mit dem `#pragma unroll` Statement versehen. Noch günstiger ist das manuelle Unrollen des Codes. Indirekt gilt das für große Teile des Kernels sowieso, weil immer wieder andere Fallunterscheidungen nötig sind. Eine alternative Formulierung des Kernels mit, zumindest in Teilen, einer Schleife über die acht Segmente eines Work-Items verschlechtert die Performance. Daran ändert auch ein `#pragma unroll` Statement nichts.

Registerverbrauch: Der Registerverbrauch ist so ausbalanciert, dass die Parallelität nicht weiter als durch den Shared-Memory Verbrauch limitiert wird. Das gilt zumindest für die Fermi und Kepler Architektur. Freiheitsgrade, den Registerverbrauch einzustellen, sind: Datentypen für private Datenstrukturen, Verarbeitung von mehr Elementen je Work-Item, Verringerung der asynchronen- oder unabhängigen Arbeitsweise innerhalb der Work-Items.

Alle beschriebenen Optimierungen sind im Rahmen dieser Arbeit experimentell evaluiert (wenn auch nicht alle Kombinationen daraus), allerdings sind nur die wichtigsten in den vorherigen Abschnitten dokumentiert.

Bevor wir nun den Code selbst anschauen, seien noch einige Hinweise gegeben, wie dieser zu lesen ist. Generell wird die Kenntnis der Funktionsweise der verwendeten Subalgorithmen bzw. Techniken, welche in dieser Arbeit beschrieben sind vorausgesetzt. Kommentare erläutern ausschließlich Details der Art von deren Implementation. Sofern Gleichartiges sich wiederholt, wird es nur beim ersten Auftreten kommentiert. Weiterhin werden ggf. interessante Abweichungen, speziell zu betrachtende Sonderfälle, bei

Wiederholungen ähnlicher Codeabschnitte durch Kommentare gekennzeichnet. Bei den Bezeichnungen werden folgende Namenskonventionen verwendet:

- Tile-Generation: G1 oder G2 werden verwendet, wenn etwas entsprechender Bedeutung für beide Generationen existiert. Existiert es nur für eine Generation, (zumeist G1), gibt es keinen Anhang. Lokale Daten haben ebenfalls zumeist keinen Hinweis auf die Generation (außer z.B. bei der Umrechnung von G1 nach G2).
- Pixel: Eine angehängte 0 deutet auf den oberen Pixel des jeweiligen Work-Items hin, eine 1 auf den Unteren.
- Segment: R(ight), L(eft), D(own), U(p) geben einen Hinweis auf das jeweilige Segment gemäß DST-Typ. Beispiel: aL0 gibt Eigenschaft 'a' des Segments des DST-Typs Left des oberen Pixels eines Work-Items an. Außerdem sind auch Kombinationen möglich, wenn etwas für mehr als einen Segmenttyp verwendet wird.
- Lokaler Index oder Daten: Innerhalb der Tiles existiert ein lokaler Adressraum. Indices, Makros und Datenstrukturen dafür bekommen ein L oder oft auch _L angehängt. Beispiel: `idL1` ist die lokale Adresse des unteren Pixels des Work-Items innerhalb des Tiles. Dagegen ist `id1` die globale Adresse desselben Pixels. Semantisch dürfte keine Verwechslungsgefahr mit L für Left bestehen.

Und nun, viel Spaß mit Listing 23.1 und Listing 23.2 (letzteres ab Seite 273). Hinweis: Das nächste Kapitel, Evaluation der Impl IV, beginnt auf Seite 303.

```
// Diverse constants
#define SEG_NOT_EXISTING 0
#define UNCLASSIFIED 0
#define UNLABELED_BUT_CLASSIFIED -90
#define UNLABELED -1
#define UNDEF -1
#define TRUE 1
#define FALSE 0
#define RIGHT 0
#define LEFT  1
#define DOWN  2
#define UP    3
#define EXISTING_BIT  (1 << 0)

// Data resolution (values resolution dependend... )
// #define X 4096
// #define Y 4096
#define N (X * Y)

// Tile size related (first generation)
//  Tile width. Unit: Pixels
#define TILE_X_G1 32
//  Tile height. Unit: Pixels
#define TILE_Y_G1 32
//  Number of elements. Unit: Pixels
#define TILE_PIXEL_CNT_G1 (TILE_X_G1 * TILE_Y_G1)
//  Number of elements. Unit: Segments
#define TILE_SIZE_G1      (TILE_PIXEL_CNT_G1 * 4)
#define TILE_X_G1m2       (2 * TILE_X_G1)
#define TILE_Y_G1_MINUS_1 (TILE_Y_G1 - 1)
#define TILE_X_G1_MINUS_1 (TILE_X_G1 - 1)

// Number of each SRC-type of segment, surviving tiles of G1
//  Note: Will not work, if tiles are not squares.
#define SEGS_RLDU_G1 TILE_X_G1

// Relation between G1 and G2 tiles
#define SUB_TILES_X_G1 8
#define SUB_TILES_Y_G1 8
#define SUB_TILES_G1 (SUB_TILES_X_G1 * SUB_TILES_Y_G1)

// Tile size related (second generation)
//  Tile width. Unit: Pixels
#define TILE_X_G2 (SUB_TILES_X_G1 * TILE_X_G1)
//  Tile height. Unit: Pixels
#define TILE_Y_G2 (SUB_TILES_Y_G1 * TILE_Y_G1)
#define TILE_X_G2m2 (2 * TILE_X_G2)
//  Number of tiles side by side
#define TILE_CNT_X_G2 (X / TILE_Y_G2)
//  Number of elements. Unit: Segments
#define TILE_SIZE_G2 \
  (SEGS_RLDU_G1 * 4 * SUB_TILES_X_G1 * SUB_TILES_Y_G1)
```

```
////////////////////////////////////////////////////////////////////////

// Bitwise coding of all segments' properties of a pixel within a char
// Used only by pixelProps(0/1)

// Bit is set, if corresponding segment exists
#define BIT_RIGHT_SEG_EXISTING      1 << 0
#define BIT_LEFT_SEG_EXISTING       1 << 1
#define BIT_DOWN_SEG_EXISTING       1 << 2
#define BIT_UP_SEG_EXISTING         1 << 3


// Bit is set, if corresponding segment involves one (any) io-Edge
#define BIT_INOUT_CONTOUR_RIGHT_SEG 1 << 4
#define BIT_INOUT_CONTOUR_LEFT_SEG  1 << 5
#define BIT_INOUT_CONTOUR_DOWN_SEG  1 << 6
#define BIT_INOUT_CONTOUR_UP_SEG    1 << 7

// Most of the above can be cleared by bitwise & application of one of
// the foloowing constants:
#define DESELECT_BIT_RIGHT_SEG_EXISTING                                 \
(         1 << 1 | 1 << 2 | 1 << 3 | 1 << 4 | 1 << 5 | 1 << 6 | 1 << 7)
#define DESELECT_BIT_LEFT_SEG_EXISTING                                  \
(1 << 0 |          1 << 2 | 1 << 3 | 1 << 4 | 1 << 5 | 1 << 6 | 1 << 7)
#define DESELECT_BIT_DOWN_SEG_EXISTING                                  \
(1 << 0 | 1 << 1 |          1 << 3 | 1 << 4 | 1 << 5 | 1 << 6 | 1 << 7)
#define DESELECT_BIT_UP_SEG_EXISTING                                    \
(1 << 0 | 1 << 1 | 1 << 2 |          1 << 4 | 1 << 5 | 1 << 6 | 1 << 7)
#define DESELECT_BIT_INOUT_CONTOUR_DOWN_SEG                             \
(1 << 0 | 1 << 1 | 1 << 2 | 1 << 3 | 1 << 4 | 1 << 5 |          1 << 7)
#define DESELECT_BIT_INOUT_CONTOUR_UP_SEG                               \
(1 << 0 | 1 << 1 | 1 << 2 | 1 << 3 | 1 << 4 | 1 << 5 | 1 << 6         )

// Segments of DST-typse RIGHT and LEFT can involve both io-Edges.
// Without using additional bits, it can be coded using a combination
// of above bits, which can not occur otherwise.
//  If a segment of DST type RIGHT exists and contains one (or more)
//  io-edges, it is guaranteed to contain the lower io-edge.
//  If so, even if a segment of DST-type UP exists, it cannot contain
//  any io-edges.
#define BIT_INOUT_CONTOUR_RIGHT_SEG_x2                                  \
(BIT_INOUT_CONTOUR_RIGHT_SEG | BIT_INOUT_CONTOUR_UP_SEG  )
//  If a segment of DST type LEFT exists and contains one (or more)
//  io-edges, it is guaranteed to contain the upper io-edge.
//  If so, even if a segment of DST-type DOWN exists, it cannot contain
//  any io-edges.
#define BIT_INOUT_CONTOUR_LEFT_SEG_x2                                   \
(BIT_INOUT_CONTOUR_LEFT_SEG  | BIT_INOUT_CONTOUR_DOWN_SEG)
```

```
// io-edge information of segments of DST types RIGHT and LEFT can
// be cleared by bitwise & application of on of the constants below.
//  Clear io-info of RIGHT seg. Must not be applied if segment does not
//  contain an io-edge (may destroy io-edge info of Up-seg otherwise)
#define DESELECT_BIT_INOUT_CONTOUR_RIGHT_SEG                              \
(1 << 0 | 1 << 1 | 1 << 2 | 1 << 3 |          1 << 5 | 1 << 6          )
//  Clear io-info of LEFT seg. Must not be applied if segment does not
//  contain an io-edge (may destroy io-edge info of DOWN-seg otherwise)
#define DESELECT_BIT_INOUT_CONTOUR_LEFT_SEG                               \
(1 << 0 | 1 << 1 | 1 << 2 | 1 << 3 | 1 << 4 |                    1 << 7)



////////////// Pixel and segment adress determination /////////////////

// Compute local id of a pixel neighboring p in a tile, given
// p's local id. Only first generation.
#define RIGHT_PIXEL_L(idL) (idL + 1)
#define  LEFT_PIXEL_L(idL) (idL - 1)
#define    UP_PIXEL_L(idL) (idL - TILE_X_G1)
#define  DOWN_PIXEL_L(idL) (idL + TILE_X_G1)

// Global offsets of segs, by DST-type. Unit: Segments
#define RIGHT_OFFSET (RIGHT * N)
#define  LEFT_OFFSET (LEFT  * N)
#define  DOWN_OFFSET (DOWN  * N)
#define    UP_OFFSET (UP    * N)

// Compute global id of a segment of a certain DST-type by global
// pixel id
#define    UP_SEG(id) ( UP_OFFSET + id)
#define  DOWN_SEG(id) (DOWN_OFFSET + id)
#define RIGHT_SEG(id) (RIGHT_OFFSET + id)
#define  LEFT_SEG(id) ( LEFT_OFFSET + id)

// Local offsets of segs, by DST-type, within tile (only 1G).
// Unit: Segments
#define RIGHT_OFFSET_L (RIGHT * TILE_PIXEL_CNT_G1)
#define  LEFT_OFFSET_L ( LEFT * TILE_PIXEL_CNT_G1)
#define    UP_OFFSET_L (   UP * TILE_PIXEL_CNT_G1)
#define  DOWN_OFFSET_L ( DOWN * TILE_PIXEL_CNT_G1)

// Compute local id of a segment of a certain DST-type given local
// pixel id . Only first generation.
#define    UP_SEG_L(idL) ( UP_OFFSET_L + idL)
#define  DOWN_SEG_L(idL) (DOWN_OFFSET_L + idL)
#define RIGHT_SEG_L(idL) (RIGHT_OFFSET_L + idL)
#define  LEFT_SEG_L(idL) ( LEFT_OFFSET_L + idL)
```

```
// Determine if a pixelcoordinate is on a g1-tile's edge
#define RIGHT_EDGE_PIXEL_G1(xL) (xL == TILE_X_G1_MINUS_1)
#define LEFT_EDGE_PIXEL_G1(xL)  (xL == 0)
#define DOWN_EDGE_PIXEL_G1(yL)  (yL == TILE_Y_G1_MINUS_1)
#define UP_EDGE_PIXEL_G1(yL)    (yL == 0)

// Calcs pos in first or last row of a g1 tile
#define POS_IN_LAST_DOWNSEG_ROW_G1(yL)                                  \
  (DOWN_SEG_L(TILE_Y_G1_MINUS_1 * TILE_X_G1 + yL))
#define POS_IN_FIRST_UPSEG_ROW_G1(yL)                                   \
  (UP_SEG_L(yL))

// Compute local id of a segment given local pixel id (idL)
// and DST-type (dst). Only first generation.
#define  GET_SEG_L(dst, idL) ((dst * TILE_PIXEL_CNT_G1) + (idL))

// Local offsets of segs, by SRC-type, within tile of 2nd generation.
// Unit: Segments
#define FROM_R_OFFSET_G2                                                \
  (SEGS_RLDU_G1 * 0 * SUB_TILES_X_G1 * SUB_TILES_Y_G1)
#define FROM_L_OFFSET_G2                                                \
  (SEGS_RLDU_G1 * 1 * SUB_TILES_X_G1 * SUB_TILES_Y_G1)
#define FROM_D_OFFSET_G2                                                \
  (SEGS_RLDU_G1 * 2 * SUB_TILES_X_G1 * SUB_TILES_Y_G1)
#define FROM_U_OFFSET_G2                                                \
  (SEGS_RLDU_G1 * 3 * SUB_TILES_X_G1 * SUB_TILES_Y_G1)


//////////////////////  Segment encoding ////////////////////////////////

// All properties of a segment, which need to be accessed by other
// segments within a tile, can be coded in a single 32 bit integer.
// This scheme is used by G1 and G2 tiles.
// Except for exception below, the encoded information is:
//  - suc: Adress of (currently) successing segment. may point to self.
//    Bits: 0 - 12
//  - OUT_OF_TILE (OOT): If (current) sucessor is not within same tile
//    Bit: 13
//  - sucInfo: Get / set both above together
//    Bits: 0 - 13
//  - minId: (current) minId of this segment and some successors
//    Bits: 14 - 31

// suc starts at bit 0
#define SUC_SHIFT 0
// Bits belonging to suc (bits 0 - 12)
#define SUC_MASK                                                        \
(1|1<<1|1<<2|1<<3|1<<4|1<<5|1<<6|1<<7|1<<8|1<<9|1<<10|1<<11|1<<12)
// Bit 13 codes OOT information
#define OUT_OF_TILE_BIT (1 << 13)
// Bits belonging to sucInfo (suc + OOT)
#define SUC_INFO_MASK (SUC_MASK | OUT_OF_TILE_BIT)
```

```
// minId starts at bit 14
#define MINID_SHIFT 14
// Bits belonging to minId (bits 14 - 31)
#define MINID_MASK                                                    \
  (1<<14|1<<15|1<<16|1<<17|1<<18|1<<19|1<<20|1<<21|1<<22|            \
   1<<23|1<<24|1<<25|1<<26|1<<27|1<<28|1<<29|1<<30|1<<31)

// Create new data-structure containing segment information
#define MAKE_SEG(sucInfo, minId)                                      \
  (((sucInfo) << SUC_SHIFT) | ((minId) << MINID_SHIFT))

// Bitwise & apply the constants to clear desired information
#define CLEAR_SUC_INFO_MASK (MINID_MASK)
#define CLEAR_MINID_MASK (SUC_INFO_MASK)

// Getter for suc
#define GET_SUC(seg)((seg & SUC_MASK) >> SUC_SHIFT)

// Getter and setter for sucInfo
#define GET_SUC_INFO(seg)((seg & SUC_INFO_MASK) >> SUC_SHIFT)
#define SET_SUC_INFO(seg, suc)                                        \
  (seg = (seg & CLEAR_SUC_INFO_MASK) | (suc << SUC_SHIFT))

// Getter and setter for minId
#define GET_MINID(seg)((seg & MINID_MASK) >> MINID_SHIFT)
#define SET_MINID(seg, minid)                                         \
  (seg = (seg & CLEAR_MINID_MASK) | (minid << MINID_SHIFT))

// Check, if OOT bit is set (or not)
#define OUT_OF_TILE(seg) ((seg) & OUT_OF_TILE_BIT)
#define SUC_IN_TILE(seg) (FALSE == (OUT_OF_TILE(seg)))

// Set OOT Bit
#define SET_OUT_OF_TILE(suc) ((suc) | OUT_OF_TILE_BIT)

// Maximum val of minId of firt generation tiles
#define MAX_MINID_G1 (2 * (TILE_X_G1 * TILE_Y_G1) - 1)

// Alternative encoding of sucInfo.
//  Used only for OOT segments of G2
//  tiles. Based on this information, a successor's adress can be
//  computed after processing of G2 tiles.
//  Information to set this (DST-type) is only available in G1 tiles,
//  so it must be already initialized here.

// Encodes the local y coordinate (DST type RIGHT, LEFT) or
// x-coordinate (DST type UP, DOWN) within a G2-tile.
// Uses bits 0 - 10
#define SUC_OOT_COORD_SHIFT 0
#define SUC_MASK_OOT_COORD                                            \
  (1|1<<1|1<<2|1<<3|1<<4|1<<5|1<<6|1<<7|1<<8|1<<9|1<< 10)
#define GET_SUC_OOT_COORD(seg)                                        \
```

```
  ((seg & SUC_MASK_OOT_COORD) >> SUC_OOT_COORD_SHIFT)

// Encodes the direction (<=> DST type) of a segment
// Uses bits 11 and 12
#define SUC_OOT_DIR_SHIFT 11
#define SUC_MASK_OOT_DIR   ( 1 << 11 | 1 << 12)
#define GET_SUC_OOT_DIR(seg)                                          \
  ((seg & SUC_MASK_OOT_DIR) >> SUC_OOT_DIR_SHIFT)

// Create new sucInfo encoding using this scheme. Can afterwards be
// normally used as argument 'sucInfo' for MAKE_SEG(...)
#define MAKE_SUC_INFO_OOTSEG_G2(x_or_y_coord, dstType)                \
  ((x_or_y_coord) | (dstType << SUC_OOT_DIR_SHIFT) | OUT_OF_TILE_BIT)


////////////////// private Segment status encoding ////////////////////

// Encoding of a status (existing, (own) suc in tile) of a segment.
//  Used for stat* data-structures, which reside in private memory.
//  SUC_IN_TILE info is (partially) redundant with OOT info.
//  In later kernels, it can be encoded if the own suc of a segment is
//  in tile. This information is lost in the above scheme, due to the
//  pointer jumps

#define SUC_IN_TILE_BIT  (1 << 1)
#define EXISTING_AND_SUC (EXISTING_BIT | SUC_IN_TILE_BIT)
#define EX__N__SUC_NOT_IN_TILE 1

#define MAKE_STATS_G1(stat)(EXISTING_BIT | (SUC_IN_TILE(stat) ? 2 : 0))
#define MAKE_STATS_G2(stat)(EXISTING_BIT |                            \
                            (SUC_IN_TILE(stat) ? 2 : 0) |             \
                            (SUC_IN_TILE(stat) ? 0 : 4)               \
                           )

// Checks, if current successor of an (existing) segment is in tile
//  Is applied during pointer jumps.
#define EX__N__SUC_IN_TILE(stat) (stat == EXISTING_AND_SUC)

// Checks, after application of all pointer jumps, if segment belongs
// to a contour, which is closed in current tile.
//  Note: Computes the same as above.
#define EX__N__CLOSED(stat) (stat == EXISTING_AND_SUC)

// Checks, after application of all pointer jumps, if segment belongs
// to a contour, which is not closed in current tile (and goes OOT).
#define EX__N__OOT_G1(stat) (stat == EXISTING_BIT)


//////////////////////////// miscellaneous ////////////////////////////////

// Checks if one (or more) bits of a given bitfield are set in var
#define ONE_BIT_SET(var, bitmask)((var) & bitmask)
```

```
// Adress in local data-structure seg_L, where information about
// pointer jump completeness is stored
#define MORE_WORK LEFT_OFFSET_L


///// Initialization, transformation and evaluation of minId //////////

// Computations of local minIds in G1 tiles. io-Bits in pixelProps of
// the particular segment must have been set previously.
// minId is always 2 * local pixel id + 0 (seg contains upper Edge) or
// 1 (otherwise)

// If DST type is RIGHT, the upper edge is contained if and only if
// the segment contains both io-edges.
#define CALC_MINID_R(pixelProps, idL)                                   \
(                                                                       \
  2 * idL + ((pixelProps & BIT_INOUT_CONTOUR_RIGHT_SEG_x2)              \
  != BIT_INOUT_CONTOUR_RIGHT_SEG_x2)                                    \
)

// If DST type is LEFT, the upper edge is contained if the segment
// contains one (or more) io-edges.
#define CALC_MINID_L(pixelProps, idL)                                   \
(                                                                       \
  2 * idL + ((pixelProps & BIT_INOUT_CONTOUR_LEFT_SEG)                  \
  != BIT_INOUT_CONTOUR_LEFT_SEG)                                        \
)

// If DST type is DOWN, the upper edge is contained if the segment
// contains one io-edge
#define CALC_MINID_D(pixelProps, idL)                                   \
(                                                                       \
  2 * idL + ((pixelProps & BIT_INOUT_CONTOUR_DOWN_SEG)                  \
  != BIT_INOUT_CONTOUR_DOWN_SEG)                                        \
)

// If DST type is UP, the upper edge is never contained
#define CALC_MINID_U(idL) (2 * idL + 1)

// Check if a segment belongs to an outer- or inner contour
#define OUTER(minId) ((minId & 1) == FALSE)
#define INNER(minId)  (minId & 1)
```

```
// Transforms a minId or label (oLabel) of a G1 tile with coordinates
// tileX, tileY (Unit: tiles_G1) from G1 local domain to global domain.
// This is done in 3 steps:
//  1. Deconstruct oLabel to local x and y coordinate and upperedge
//     info
//  2. Compute global x and y coordinates
//  3. Compute new label / minId
#define LABEL_TRANSFORM_G1(tileX, tileY, oLabel)                        \
 (                                                                      \
   2 * (                                                                \
         X * (                                                          \
           (oLabel) / TILE_X_G1m2                                       \
           + (tileY) * TILE_Y_G1                                        \
         )                                                              \
         + ((oLabel) % TILE_X_G1m2) / 2                                 \
         + (tileX) * TILE_X_G1                                          \
   )                                                                    \
   + ((oLabel) & 1)                                                     \
 )

// As above, but label or minId is transformed from 1G to 2G tile
// domain. sTileX and sTileY are coordinates of the 1G subtile within
// the 2G Tile (Unit: tiles_1G).
#define LABEL_TRANSFORM_G1G2(sTileX, sTileY, oLabel)                    \
(                                                                       \
  2 * (                                                                 \
        TILE_X_G2 * (                                                   \
          (oLabel) / TILE_X_G1m2                                        \
          + (sTileY) * TILE_Y_G1                                        \
        )                                                               \
        + ((oLabel) % TILE_X_G1m2) / 2                                  \
        + (sTileX) * TILE_X_G1                                          \
  )                                                                     \
  + ((oLabel) & 1)                                                      \
)


//////////////////// Compute suc in G2 tiles ////////////////////////

// Computes and returns sucInfo of a G2 tile segment, given the suc
// info of a G1 OOT segment as well as its tile-indexes
int calcSucIn_Tiles_G2(
  int localSeg,  // Adress of OOT segment (G1)
  int subTileX,  // x component of 2D-SubTile-Id. Unit: tiles_1G
  int subTileY,  // y component of 2D-SubTile-Id. Unit: tiles_1G
  int subTileRL, // 1D-SubTile-Id (transposed scheme). Unit: tiles_1G
  int subTileDU  // 1D-SubTile-Id. Unit: tiles_1G
  ) {

  int nextSubTile, localPixel, x, y;
```

```
// If DST Typ RIGHT
if(localSeg < LEFT_OFFSET_L){
  // next subTile on the right side is placed 'below', due to the
  // transposed layout.
  nextSubTile = subTileRL + SUB_TILES_Y_G1;
  localPixel = localSeg - RIGHT_OFFSET_L;
  y = localPixel / TILE_X_G1;
  return
    // Segment in next generation not OOT?
    (subTileX != SUB_TILES_X_G1 - 1)
    ?
    // True:
    // Compute local adress of suc in G2 tile
    FROM_L_OFFSET_G2  // this seg has DST RIGHT => suc has SRC LEFT
    // Add offset corresponding to G1 subtile in G2 tile
    + nextSubTile * SEGS_RLDU_G1
    // Add id within subtile
    + y
    :
    // False: Encode information to compute adress after G2 tiles
    //        processing
    (
      MAKE_SUC_INFO_OOTSEG_G2(
        subTileY * TILE_Y_G1 + y, // y-coordinate within G2 tile
        RIGHT                     // DST type
      )
    );
}

// If DST Typ LEFT
else if (localSeg < DOWN_OFFSET_L){
  nextSubTile = subTileRL - SUB_TILES_Y_G1;
  localPixel = localSeg - LEFT_OFFSET_L;
  y = localPixel / TILE_X_G1;
  return
    (subTileX != 0)
    ?
    FROM_R_OFFSET_G2 + nextSubTile * SEGS_RLDU_G1 + y
    :
    (
      MAKE_SUC_INFO_OOTSEG_G2(
        subTileY * TILE_Y_G1 + y,
        LEFT
      )
    );
}

// If DST Typ DOWN
else if (localSeg < UP_OFFSET_L){
  nextSubTile = subTileDU + SUB_TILES_X_G1;
  localPixel = localSeg - DOWN_OFFSET_L;
  x = localPixel % TILE_X_G1;
```

```
    return
      (subTileY != SUB_TILES_Y_G1 - 1)
      ?
      FROM_U_OFFSET_G2 + nextSubTile * SEGS_RLDU_G1 + x
      :
      (
        MAKE_SUC_INFO_OOTSEG_G2(
          subTileX * TILE_X_G1 + x,
          DOWN
        )
      );
  }

  // If DST Typ UP
  else {
    nextSubTile = subTileDU - SUB_TILES_X_G1;
    localPixel = localSeg - UP_OFFSET_L;
    x = localPixel % TILE_X_G1;
    return
      (subTileY != 0)
      ?
      FROM_D_OFFSET_G2 + nextSubTile * SEGS_RLDU_G1 + x
      :
      (
        MAKE_SUC_INFO_OOTSEG_G2(
          subTileX * TILE_X_G1 + x,
          UP
        )
      );
  }

}
```

Listing 23.1: Makros für Kernel extractSegs_N_TilePJ_G1

```
// Begin Kernel extractSegs_N_perTilePJ_G1
kernel
// Only square tiles are allowed.
 __attribute__((reqd_work_group_size(TILE_X_G1, TILE_X_G1 / 2, 1)))
 void extractSegs\_N\_TilePJ\_G1(

// Initialized data (only read access)
//  Contains N classifications:
//   - UNCLASSIFIED (0)
//   - Some classification (1, 2, 3, 4, ...)
//  Data is surrounded by a border of zeros (width = 1), otherwise
//  additional exeptions concerning the image's borders would be
//  neccessary.
//  So, all in all, there are (X + 2) * (Y + 2) entries.
  global char*   pixel_Classification,

// Not initialized data. (only write access)
  global int*    pixel_Label,
  global char*   pixel_SegsProperties,
  global short*  segsTile1_Suc,
  global int*    segsTile2_Data,
  global short*  segsTile2_SucInTile1,
  global char*   segsTile2_Props) {

  local int seg_L[TILE_SIZE_G1]; // See below (Datenformat A)

////////////////////////////////////////////////////////////////////////
////////////////// Begin Teil I: Extract Segsments /////////////////////

  // Initialize some indices. Each work-item processes 2 pixels, which
  // are on top of each other. All names of data of the upper pixel
  // ends with a 0. Lower pixel: 1

  // Global x-Coordinate
  int x = get_global_id(0);

  // Global y-Coordinates
  int y0 = 2 * get_global_id(1);
  int y1 = y0 + 1;

  // Global ids (= position in global memory)
  int id0 = X * y0 + x;
  int id1 = id0 + X;

  // Local (according to tile) coordinates and ids (always with a 'L')
  int xL = get_local_id(0);
  int yL0 = 2 * get_local_id(1);
  int yL1 = yL0 + 1;
  int idL0 = yL0  * TILE_X_G1 + xL;
  int idL1 = idL0 + TILE_X_G1;
```

```
// Additional global ids only for pixel_Classification data
int idC0 = (y0 + 1) * (X + 2) + (x + 1);
int idC1 = (y1 + 1) * (X + 2) + (x + 1);

// Load classifications of considered pixels
int valX0 = pixel_Classification[idC0];
int valX1 = pixel_Classification[idC1];

/*
Datenformat A (gilt ab hier)

I. Local Mem Daten (seg_L):
  Fuer alle Elemente, bis auf Ausnahmen (s.u.), ist enthalten, sofern
  das betreffende Element existiert:
  - OOT-Segmente der Typen RIGHT und LEFT (ausser erstes): Eigene
    Adresse, OOT Bit gesetzt und maximale minId
  - OOT-Segmente der Typen UP und DOWN: Kein Eintrag
  - Nachfolger ist OOT eines der Typen UP und DOWN, sowie 1. LEFT:
    Adresse des Nachfolgeelements, OOT-Bit gesetzt und minId
    Nachfolgeelements.
  - Sonst: Adresse des Nachfolgeelements und maximale minId
  - Element existiert nicht: Es wird nicht explizit geschrieben
  Alle Informationen sind in einem 32-Bit Integer je Segment
  zusammengefasst.

  Ausnahmen:

  a) Erste Zeile (32 Werte) der UP-Positionen, gesetzt ueber
     POS_IN_FIRST_UPSEG_ROW_G1(yL0 / yL1):
     Information ueber ein Segment, dessen Vorgaenger OOT ist und das
     den SRC-Typ RIGHT hat. Es kommt also von links ausserhalb des
     Tiles. Gespeichert wird an transponierter Position. Werte:
      - UNDEF: Es exitiert keins fuer diese Position
      - RIGHT, LEFT, UP, DOWN: Dst-Typ des Segments

  b) Letzte Zeile (32 Werte) der DOWN-Positionen, gesetzt ueber
     POS_IN_LAST_DOWNSEG_ROW_G1(yL0 / yL1):
     Information ueber ein Segment, dessen Vorgaenger OOT ist und das
     den SRC-Typ LEFT hat. Es kommt also von rechts ausserhalb des
     Tiles. Gespeichert wird an transponierter Position. Werte:
     - UNDEF: Es exitiert keins fuer diese Position
     - RIGHT, LEFT, UP, DOWN: Dst-Typ des Segments

II. Private Daten:
  a) comesFromInfoDU (upper Row <=> yL0 = 0
     <=> UP_EDGE_PIXEL_G1(yL0) = TRUE):
     Information ueber ein Segment, dessen Vorgaenger OOT ist und
     das den SRC-Typ UP hat. Es kommt also von oben ausserhalb des
     Tiles. Gespeichert wird an eigener Pixel Position. Werte:
      - UNDEF: Es exitiert keins fuer diese Position
      - RIGHT, LEFT, UP, DOWN: DST-Typ des Segments
  b) comesFromInfoDU (lowest Row <=> yL1 = TILE_Y_G1_MINUS_1
```

```
      <=> DOWN_EDGE_PIXEL_G1(yL1) = TRUE):
      Information ueber ein Segment, dessen Vorgaenger OOT ist und das
      den SRC-Typ UP hat. Es kommt also von oben ausserhalb d. Tiles.
      Gespeichert wird an eigener Pixel Position. Werte:
       - UNDEF: Es exitiert keins fuer diese Position
       - RIGHT, LEFT, UP, DOWN: DST-Typ des Segments
  c) comesFromInfoDU (sonst): undefiniert.
  d) pixelProps0, pixelProps1:
     Eigenschaften der jeweils 4 Segmente eines Pixels,
     bitweise codiert:
       - Existenz der Segmente
       - Vorhandene io-Kanten der Segmente
  e) Rest: Temporaere bzw. offensichtliche Daten.
*/

// Some entries need to be initialized as not existing. Completeness
// is ensured afterwards
if(yL0 == 0){
  seg_L[POS_IN_FIRST_UPSEG_ROW_G1(xL) ] = UNDEF;
  seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(xL)] = UNDEF;
}
barrier(CLK_LOCAL_MEM_FENCE);

pixel_Label[id0] = (valX0 == UNCLASSIFIED) ?
                   UNLABELED : UNLABELED_BUT_CLASSIFIED;
pixel_Label[id1] = (valX1 == UNCLASSIFIED) ?
                   UNLABELED : UNLABELED_BUT_CLASSIFIED;

int pixelProps0 = 0;
int comesFromInfoDU = UNDEF;

int val_DL_Eq, val_D__Eq, val_DR_Eq, val__R_Eq,
    val_UR_Eq, val_U__Eq, val_UL_Eq, val__L_Eq;
int this, thisPre;

// Load some neighboring classifications.
// Value layout:
//  val00 val01 val02
//  val10 valX0 val12
//  val20 valX1 val22
//  val30 val31 val32 (required / loaded later)
int val00, val01, val02, val10, val12, val20, val22;
val12   = pixel_Classification[idC0           + 1];
val22   = pixel_Classification[idC0 + (X + 2) + 1];
val10   = pixel_Classification[idC0           - 1];
val01   = pixel_Classification[idC0 - (X + 2)    ];
val02   = pixel_Classification[idC0 - (X + 2) + 1];
val00   = pixel_Classification[idC0 - (X + 2) - 1];
val20   = pixel_Classification[idC0 + (X + 2) - 1];
```

```
// Compute equivalencies of neighboring pixel's classifications to
// upper pixel. Naming scheme:
//  (U, _, D) x (L, _, R) Eq
//  (Up, Ground, Down) x (Left, Middle, Right) Equal
//  For example 'val_UR_Eq = 1' means pixel right above the pixel
//  under consideration (here: upper pixel) has the same
//  classification.
val__R_Eq = (valX0 == val12);
val_D__Eq = (valX0 == valX1);
val_DR_Eq = (valX0 == val22);
val__L_Eq = (valX0 == val10);
val_U__Eq = (valX0 == val01);
val_UR_Eq = (valX0 == val02);
val_UL_Eq = (valX0 == val00);
val_DL_Eq = (valX0 == val20);


// Begin setze Datenformat A

// Consider segments of upper pixel, if classified
if(valX0 != UNCLASSIFIED){

  ////////////////////////// DST: RIGHT ////////////////////////////
  // Check if segment of DST-Type RIGHT exists
  if(val__R_Eq && (!val_D__Eq || !val_DR_Eq)){
    // Default initialization: suc <- this.id, minId <- MAX_MINID_G1
    this = MAKE_SEG(RIGHT_SEG_L(idL0), MAX_MINID_G1);

    // Determine SRC-Type (implies DST Type of pre Segment)
    // Previous Seg DST: UP
    // Pre always in tile, since this is the upper pixel
    if      (val_D__Eq) {
      thisPre = UP_SEG_L(DOWN_PIXEL_L(idL0));
    }
    // Previous Seg DST: RIGHT
    else if (val__L_Eq) {
      thisPre = (LEFT_EDGE_PIXEL_G1(xL)) ?
                  UNDEF : RIGHT_SEG_L(LEFT_PIXEL_L(idL0));
      pixelProps0 |= BIT_INOUT_CONTOUR_RIGHT_SEG;
      if(LEFT_EDGE_PIXEL_G1(xL))
        seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL0) ] = RIGHT;
    }
    // Previous Seg DST: DOWN
    else if (val_U__Eq) {
      thisPre = (UP_EDGE_PIXEL_G1(yL0)) ?
                  UNDEF : DOWN_SEG_L(UP_PIXEL_L(idL0));
      pixelProps0 |= BIT_INOUT_CONTOUR_RIGHT_SEG;
      if(UP_EDGE_PIXEL_G1(yL0))
        comesFromInfoDU = RIGHT;
    }
    // Previous Seg DST: LEFT
    else                    {
```

```
    thisPre = (RIGHT_EDGE_PIXEL_G1(xL)) ?
              UNDEF : LEFT_SEG_L(RIGHT_PIXEL_L(idL0));
    pixelProps0 |= BIT_INOUT_CONTOUR_RIGHT_SEG_x2;
    if(RIGHT_EDGE_PIXEL_G1(xL))
      seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL0)] = RIGHT;
  }

  // If suc OOT: Set this.suc <- this | OOT_BIT
  if(RIGHT_EDGE_PIXEL_G1(xL)) {
    seg_L[GET_SUC(this)] = this | OUT_OF_TILE_BIT;
  }

  // If pre not OOT
  if(thisPre != UNDEF) {
    seg_L[thisPre] = this;
  }
  // Set this segment existing
  pixelProps0 |= BIT_RIGHT_SEG_EXISTING;
}


//////////////////////////// DST: Up ////////////////////////////
// Check if segment of DST-Type UP exists
if(val_U__Eq && (!val__R_Eq || !val_UR_Eq)){
  this = MAKE_SEG(UP_SEG_L(idL0), MAX_MINID_G1);

  // Previous Seg DST: LEFT
  if      (val__R_Eq) {
    thisPre = (RIGHT_EDGE_PIXEL_G1(xL)) ?
              UNDEF : LEFT_SEG_L(RIGHT_PIXEL_L(idL0));
    if(RIGHT_EDGE_PIXEL_G1(xL))
      seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL0)] = UP;
  }
  // Previous Seg DST: Up
  // Pre always in tile, since this is the upper pixel
  else if (val_D__Eq) {
    thisPre = UP_SEG_L(DOWN_PIXEL_L(idL0));
  }
  // Previous Seg DST: RIGHT
  else if (val__L_Eq) {
    thisPre = (LEFT_EDGE_PIXEL_G1(xL)) ?
              UNDEF : RIGHT_SEG_L(LEFT_PIXEL_L(idL0));
    pixelProps0 |= BIT_INOUT_CONTOUR_UP_SEG;
    if(LEFT_EDGE_PIXEL_G1(xL))
      seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL0)] = UP;
  }
  // Previous Seg DST: DOWN
  else                      {
    thisPre = (UP_EDGE_PIXEL_G1(yL0)) ?
              UNDEF : DOWN_SEG_L(UP_PIXEL_L(idL0));
    pixelProps0 |= BIT_INOUT_CONTOUR_UP_SEG;
    if(UP_EDGE_PIXEL_G1(yL0))
```

```
        comesFromInfoDU = UP;
    }

    // If suc OOT
    if(UP_EDGE_PIXEL_G1(yL0)) {
      // OOT-segs of DST Type UP are not stored in
      // local memory.
      // To ensure correctness, this segment is fully
      // initialized and its data passed to the pre segment
      // (if pre in tile)
      this = MAKE_SEG(
         SET_OUT_OF_TILE(UP_SEG_L(idL0)),
         CALC_MINID_U(idL0)
      );
    }

    // If pre not OOT
    if(thisPre != UNDEF) {
      seg_L[thisPre] = this;

    }
    pixelProps0 |= BIT_UP_SEG_EXISTING;
  }


  /////////////////////////// DST: LEFT ///////////////////////////
  if(val__L_Eq && (!val_U__Eq || !val_UL_Eq)){
    this = MAKE_SEG(LEFT_SEG_L(idL0), MAX_MINID_G1);
    // Previous Seg DST: DOWN
    if    (val_U__Eq) {
      thisPre = (UP_EDGE_PIXEL_G1(yL0)) ?
                 UNDEF : DOWN_SEG_L(UP_PIXEL_L(idL0));
      if        (UP_EDGE_PIXEL_G1(yL0)) comesFromInfoDU = LEFT;
    }
    // Previous Seg DST: LEFT
    else if (val__R_Eq) {
      thisPre = (RIGHT_EDGE_PIXEL_G1(xL)) ?
                 UNDEF : LEFT_SEG_L(RIGHT_PIXEL_L(idL0));
      pixelProps0 |= BIT_INOUT_CONTOUR_LEFT_SEG;
      if        (RIGHT_EDGE_PIXEL_G1(xL))
                 seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL0)] = LEFT;
    }
    // Previous Seg DST: UP
    // Pre always in tile, since this is the upper pixel
    else if (val_D__Eq) {
      thisPre = UP_SEG_L(DOWN_PIXEL_L(idL0));
      pixelProps0 |= BIT_INOUT_CONTOUR_LEFT_SEG;
    }
    // Previous Seg DST: RIGHT
    else {
      thisPre = (LEFT_EDGE_PIXEL_G1(xL)) ?
                 UNDEF : RIGHT_SEG_L(LEFT_PIXEL_L(idL0));
```

```
    pixelProps0 |= BIT_INOUT_CONTOUR_LEFT_SEG_x2;
    if        (LEFT_EDGE_PIXEL_G1(xL))
               seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL0)] = LEFT;
  }

  // If suc OOT
  if(LEFT_EDGE_PIXEL_G1(xL)) {
    // The first OOT-seg of DST Type LEFT is only temporary
    // stored in local memory.
    // To ensure correctness, this segment is fully
    // initialized and its data passed to the pre segment
    // (if pre in tile)
    if(UP_EDGE_PIXEL_G1(yL0))
      this = MAKE_SEG(
        (LEFT_SEG_L(idL0) | OUT_OF_TILE_BIT),
         CALC_MINID_L(pixelProps0, idL0)
      );
    seg_L[GET_SUC(this)] = this | OUT_OF_TILE_BIT;
  }

  // If pre not OOT
  if(thisPre != UNDEF) {
    seg_L[thisPre] = this;
  }
  pixelProps0 |= BIT_LEFT_SEG_EXISTING;
}


//////////////////////////// DST: DOWN ////////////////////////////
if(val_D__Eq && (!val__L_Eq || !val_DL_Eq)){
  this = MAKE_SEG(DOWN_SEG_L(idL0), MAX_MINID_G1);
  // Previous Seg DST: RIGHT
  if    (val__L_Eq) {
    thisPre = (LEFT_EDGE_PIXEL_G1(xL)) ?
               UNDEF : RIGHT_SEG_L(LEFT_PIXEL_L(idL0));
    if        (LEFT_EDGE_PIXEL_G1(xL))
               seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL0)] = DOWN;
  }
  // Previous Seg DST: DOWN
  else if (val_U__Eq) {                // Previous Down
    thisPre = (UP_EDGE_PIXEL_G1(yL0)) ?
               UNDEF : DOWN_SEG_L(UP_PIXEL_L(idL0));
    if(UP_EDGE_PIXEL_G1(yL0))
      comesFromInfoDU = DOWN;
  }
  // Previous Seg DST: LEFT
  else if (val__R_Eq) {                // Previous Left
    thisPre = (RIGHT_EDGE_PIXEL_G1(xL)) ?
               UNDEF : LEFT_SEG_L(RIGHT_PIXEL_L(idL0));
    pixelProps0 |= BIT_INOUT_CONTOUR_DOWN_SEG;
    if        (RIGHT_EDGE_PIXEL_G1(xL))
               seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL0)] = DOWN;
```

```
    }
    // Previous Seg DST: UP
    // Pre always in tile, since this is the upper pixel
    else {
      thisPre = UP_SEG_L(DOWN_PIXEL_L(idL0));
      pixelProps0 |= BIT_INOUT_CONTOUR_DOWN_SEG;
    }

    // Suc always in tile, since this is the upper pixel

    // If pre not OOT
    if(thisPre != UNDEF) {
      seg_L[thisPre] = this;
    }
    pixelProps0 |= BIT_DOWN_SEG_EXISTING;
  }

  if(!val_D__Eq && !val__R_Eq && !val_U__Eq && !val__L_Eq){
    pixel_Label[id0] = 2 * id0;
  }

}

// Load new lower data row as required for lower pixel evaluation
int val30, val31, val32;
val30 = pixel_Classification[idC0 + 2 * (X + 2) - 1];
val31 = pixel_Classification[idC0 + 2 * (X + 2)    ];
val32 = pixel_Classification[idC0 + 2 * (X + 2) + 1];

int pixelProps1 = 0;

// Compute new equivalencies
val__R_Eq = val22 == valX1;
val_D__Eq = val31 == valX1;
val_DR_Eq = val32 == valX1;
val__L_Eq = val20 == valX1;
val_U__Eq = valX0 == valX1;
val_UR_Eq = val12 == valX1;
val_UL_Eq = val10 == valX1;
val_DL_Eq = val30 == valX1;

// Consider segments of lower pixel, if classified
if(valX1 != UNCLASSIFIED){

  ////////////////////////// DST: RIGHT //////////////////////////
  if(val__R_Eq && (!val_D__Eq || !val_DR_Eq)){
    this = MAKE_SEG(RIGHT_SEG_L(idL1), MAX_MINID_G1);
    // Previous Seg DST: UP
    if      (val_D__Eq) {
      thisPre = UP_SEG_L(DOWN_PIXEL_L(idL1));
      if(DOWN_EDGE_PIXEL_G1(yL1)){
        thisPre = UNDEF;
```

```
        comesFromInfoDU = RIGHT;
      }
    }
    // Previous Seg DST: RIGHT
    else if (val__L_Eq) {
      thisPre = RIGHT_SEG_L(LEFT_PIXEL_L(idL1));
      if(LEFT_EDGE_PIXEL_G1(xL)) {
        thisPre = UNDEF;
        seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL1)] = RIGHT;
      }
      pixelProps1 |= BIT_INOUT_CONTOUR_RIGHT_SEG;
    }
    // Previous Seg DST: DOWN
    // Pre always in tile, since this is the lower pixel
    else if (val_U__Eq) {
      thisPre = DOWN_SEG_L(UP_PIXEL_L(idL1));
      pixelProps1 |= BIT_INOUT_CONTOUR_RIGHT_SEG;
    }
    // Previous Seg DST: LEFT
    else {
      thisPre = LEFT_SEG_L(RIGHT_PIXEL_L(idL1));
      if(RIGHT_EDGE_PIXEL_G1(xL)){
        thisPre = UNDEF;
        seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL1)] = RIGHT;
      }
      pixelProps1 |= BIT_INOUT_CONTOUR_RIGHT_SEG_x2;
    }

    // If suc OOT
    if(RIGHT_EDGE_PIXEL_G1(xL)) {
      seg_L[GET_SUC(this)] = this | OUT_OF_TILE_BIT;
    }

    // If pre not OOT
    if(thisPre != UNDEF) {
      seg_L[thisPre] = this;
    }
    pixelProps1 |= BIT_RIGHT_SEG_EXISTING;
  }


  /////////////////////////// DST: Up ////////////////////////////
  if(val_U__Eq && (!val__R_Eq || !val_UR_Eq)){
    this = MAKE_SEG(UP_SEG_L(idL1), MAX_MINID_G1);
    // Previous Seg DST: LEFT
    if      (val__R_Eq) {
      thisPre = LEFT_SEG_L(RIGHT_PIXEL_L(idL1));
      if(RIGHT_EDGE_PIXEL_G1(xL)) {
        thisPre = UNDEF;
        seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL1)] = UP;
      }
    }
```

```
    // Previous Seg DST: UP
    else if (val_D__Eq) {
      thisPre = UP_SEG_L(DOWN_PIXEL_L(idL1));
      if(DOWN_EDGE_PIXEL_G1(yL1)) {
        thisPre = UNDEF;
        comesFromInfoDU = UP;
      }
    }
    // Previous Seg DST: RIGHT
    else if (val__L_Eq) {
      thisPre = RIGHT_SEG_L(LEFT_PIXEL_L(idL1));
      if(LEFT_EDGE_PIXEL_G1(xL)) {
        thisPre = UNDEF;
        seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL1) ] = UP;
      }
      pixelProps1 |= BIT_INOUT_CONTOUR_UP_SEG;
    }
    // Previous Seg DST: DOWN
    // Pre always in tile, since this is the lower pixel
    else {
      thisPre = DOWN_SEG_L(UP_PIXEL_L(idL1));
      pixelProps1 |= BIT_INOUT_CONTOUR_UP_SEG;
    }

    // Suc always in tile, since this is the lower pixel

    // If pre not OOT
    if(thisPre != UNDEF) {
      seg_L[thisPre] = this;

    }
    pixelProps1 |= BIT_UP_SEG_EXISTING;
  }


  ////////////////////////// DST: LEFT ////////////////////////
  if(val__L_Eq && (!val_U__Eq || !val_UL_Eq)){
    this = MAKE_SEG(LEFT_SEG_L(idL1), MAX_MINID_G1);
    // Previous Seg DST: DOWN
    // Pre always in tile, since this is the lower pixel
    if    (val_U__Eq) {
      thisPre = DOWN_SEG_L(UP_PIXEL_L(idL1));
    }
    // Previous Seg DST: LEFT
    else if (val__R_Eq) {
      thisPre = LEFT_SEG_L(RIGHT_PIXEL_L(idL1));
      if(RIGHT_EDGE_PIXEL_G1(xL)) {
        thisPre = UNDEF;
        seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL1)] = LEFT;
      }
      pixelProps1 |= BIT_INOUT_CONTOUR_LEFT_SEG;
    }
```

```
    // Previous Seg DST: UP
    else if (val_D__Eq) {
      thisPre = UP_SEG_L(DOWN_PIXEL_L(idL1));
      if(DOWN_EDGE_PIXEL_G1(yL1)) {
        thisPre = UNDEF;
        comesFromInfoDU = LEFT;
      }
      pixelProps1 |= BIT_INOUT_CONTOUR_LEFT_SEG;
    }
    // Previous Seg DST: RIGHT
    else {
      thisPre = RIGHT_SEG_L(LEFT_PIXEL_L(idL1));
      if(LEFT_EDGE_PIXEL_G1(xL)) {
        thisPre = UNDEF;
        seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL1) ] = LEFT;
      }
      pixelProps1 |= BIT_INOUT_CONTOUR_LEFT_SEG_x2;
    }

    // If suc OOT
    if(LEFT_EDGE_PIXEL_G1(xL)) {
      seg_L[GET_SUC(this)] = this | OUT_OF_TILE_BIT;
    }

    // If pre not OOT
    if(thisPre != UNDEF) {
      seg_L[thisPre] = this;

    }
    pixelProps1 |= BIT_LEFT_SEG_EXISTING;
  }


  ////////////////////////// DST: DOWN ////////////////////////////
  if(val_D__Eq && (!val__L_Eq || !val_DL_Eq)){
    this = MAKE_SEG(DOWN_SEG_L(idL1), MAX_MINID_G1);
    // Previous Seg DST: RIGHT
    if        (val__L_Eq) {
      thisPre = RIGHT_SEG_L(LEFT_PIXEL_L(idL1));
      if(LEFT_EDGE_PIXEL_G1(xL)) {
        thisPre = UNDEF;
        seg_L[POS_IN_FIRST_UPSEG_ROW_G1(yL1)] = DOWN;
      }
    }
    // Previous Seg DST: DOWN
    // Pre always in tile, since this is the lower pixel
    else if (val_U__Eq) {
      thisPre = DOWN_SEG_L(UP_PIXEL_L(idL1));
    }
    // Previous Seg DST: LEFT
    else if (val__R_Eq) {
      thisPre = LEFT_SEG_L(RIGHT_PIXEL_L(idL1));
```

```
        if(RIGHT_EDGE_PIXEL_G1(xL)) {
          thisPre = UNDEF;
          seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(yL1)] = DOWN;
        }
        pixelProps1 |= BIT_INOUT_CONTOUR_DOWN_SEG;
      }
      // Previous Seg DST: UP
      else {
        thisPre = UP_SEG_L(DOWN_PIXEL_L(idL1));
        if(DOWN_EDGE_PIXEL_G1(yL1)) {
          thisPre = UNDEF;
          comesFromInfoDU = DOWN;
        }
        pixelProps1 |= BIT_INOUT_CONTOUR_DOWN_SEG;
      }

      // If suc OOT
      if(DOWN_EDGE_PIXEL_G1(yL1)) {
        // OOT-segs of DST Type DOWN are not stored in
        // local memory.
        // To ensure correctness, this segment is fully
        // initialized and its data passed to the pre segment
        // (if pre in tile)
        this = MAKE_SEG(
           (DOWN_SEG_L(idL1) | OUT_OF_TILE_BIT),
           CALC_MINID_D(pixelProps1, idL1)
        );
      }

      // If pre not OOT
      if(thisPre != UNDEF) {
        seg_L[thisPre] = this;
      }
      pixelProps1 |= BIT_DOWN_SEG_EXISTING;
    }

    if(!val_D__Eq && !val__R_Eq && !val_U__Eq && !val__L_Eq){
      pixel_Label[id1] = 2 * id1;
    }

  }
  // End setze Datenformat A

  barrier(CLK_LOCAL_MEM_FENCE); // Make sure part 1 is completed

//// End part I: Extract Segsments ////////////////////////////////////
////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////
//// Begin part II: Pointer Jumps //////////////////////////////////////

  // Coordinates of this tile. Unit: tiles_G1
  int tileX = get_group_id(0);
  int tileY = get_group_id(1);

  /*

  -- Datenformat B -- (gilt ab hier)

  I. Local Mem Daten (seg_L):
    Fuer alle Konturseg mente ist enthalten, sofern das betreffende
    Element existiert:
    - OOT-Segmente Segmente (DST Typ UP, DOWN, erstes LEFT):
      Nicht abgespeichert
    - Sonstige OOT-Segmente
      1) suc
         Unveraenderlich: Eigene Adresse
      2) minID
         Unveraenderlich: Eigene minId
      3) OOT Bit
         Unveraenderlich: Gesetzt
    - Segmente, deren Nachfolger ein OOT-Segment des DST-Typs UP, DOWN
      oder das erste von LEFT ist:
      1) suc
         Unveraenderlich: Adresse des eigenen Nachfolgeelements
      2) minID
         Unveraenderlich: Minimum aus eigener minId und der des
                          Nachfolgers
      3) OOT Bit
         Unveraenderlich: Gesetzt
    - Sonst:
      1) suc
         Initial: Adresse des eigenen Nachfolgeelements
         Ende: Adresse eines Nachfolgesegments
      2) minID
         Initial: eigene minId
         Ende: minimaler Wert aller minId Werte der nachfolgenden
               Segmente innerhalb des Tiles
      3) OOT Bit
         Initial: Nicht gesetzt
         Ende: Ist gesetzt, falls Figur nicht innerhalb des Tiles
               geschlossen
    - Element existiert nicht: Es wird nicht explizit geschrieben

    Ausnahmen:
    Weiterhin: Erste Zeile (32 Werte) der UP-Positionen, Letzte Zeile
               (32 Werte) der DOWN-Positionen: Bleiben erhalten.
    Neu:       Erster Eintrag der LEFT-Positionen reserviert und kann
               nicht mehr fuer Segmente genutzt werden.
```

```
II. Private Daten:
  II (a) comesFromInfoDU, pixelProps0, pixelProps1: Unveraendert
         (Siehe Datenformat A)
  II (b) seg(R,L,D,U)[S](0,1): Teilweise redundante Kopie der
         entsprechenden Werte seg_L eines Segments, sowie dessen
         Nachfolgesegments. Existiert aber im Gegensatz zu seg_L fuer
         alle Segmente. Die meisten Werte werden gemaess den
         Eintraegen in seg_L initialisiert und in gleicher Weise
         geaendert. Ausnahme: OOT-Segment des DST Typ UP oder DOWN.
         Diese werden hier initialisiert und aendern sich nicht.
  II (c) (r,l,d,u)Stat(0,1): Status eines der Segmente. Bitweise
         kodiert. Teilweise redundant mit seg_L, existiert aber fuer
         alle Segmente in gleicher Weise.
      1) Existenz
         Unveraenderlich: Gesetzt gemaess pixelProps(0/1).
      2) Nachfolger im Tile (<=> Not OOT)
         Initial:
          - OOT-Segmente: FALSE
          - Nachfolger ist OOT-Segment des DST Typs UP, DOWN oder
            erster LEFT: FALSE
          - Sonst: TRUE
         Ende:
          - FALSE bleibt FALSE
          - TRUE wird FALSE, falls Figur im Tile nicht geschlossen

  II (d) Rest: Temporaere bzw. offensichtliche Daten.
*/

// Begin setze Datenformat B, Datenformat A wird teilweise
// ueberschrieben

int statR0, statL0, statD0, statU0, statR1, statL1, statD1, statU1;
int segR0,  segL0,  segD0,  segU0,  segR1,  segL1,  segD1,  segU1;

// If segment of DST-type RIGHT exists in upper pixel (pixel 0)
if(ONE_BIT_SET(pixelProps0, BIT_RIGHT_SEG_EXISTING)) {
  // Load and temporary store its Data
  segR0 = seg_L[RIGHT_SEG_L(idL0)];

  // Compute minimum of this.minId and this.suc.minId and
  // store updated segment data in private and local memory
  seg_L[RIGHT_SEG_L(idL0)]
    = SET_MINID(
      segR0,
      min(
        GET_MINID(segR0),
        CALC_MINID_R(pixelProps0, idL0)
      )
    );
  // Determine properties of segment (existing and if suc is
  // in tile. Store in private memory
```

```
  statR0 = MAKE_STATS_G1(segR0);
} else {
  statR0 = SEG_NOT_EXISTING;
}

// Next 7: Consider further 7 segments in a similar way
// Only noteworthy exceptions are noted

if(ONE_BIT_SET(pixelProps0, BIT_LEFT_SEG_EXISTING)) {
  segL0 = seg_L[LEFT_SEG_L(idL0)];
  seg_L[LEFT_SEG_L(idL0)]
   = SET_MINID(
       segL0,
       min(
         GET_MINID(segL0),
         CALC_MINID_L(pixelProps0, idL0)
       )
     );
  statL0 = MAKE_STATS_G1(segL0);
} else {
  statL0 = SEG_NOT_EXISTING;
}

if(ONE_BIT_SET(pixelProps0, BIT_DOWN_SEG_EXISTING)) {
  segD0 = seg_L[DOWN_SEG_L(idL0)];
  seg_L[DOWN_SEG_L(idL0)]
   = SET_MINID(
       segD0,
       min(
         GET_MINID(segD0),
         CALC_MINID_D(pixelProps0, idL0)
       )
    );
  statD0 = MAKE_STATS_G1(segD0);
} else {
  statD0 = SEG_NOT_EXISTING;
}

if(ONE_BIT_SET(pixelProps0, BIT_UP_SEG_EXISTING)) {
  // Check if OOT-segment
  // If not: Load and actualize it, as above
  if(!UP_EDGE_PIXEL_G1(yL0))  {
    segU0 = seg_L[UP_SEG_L(idL0)];
    seg_L[UP_SEG_L(idL0)]
     = SET_MINID(
         segU0,
         min(
           GET_MINID(segU0),
           CALC_MINID_U(idL0)
         )
      );
    statU0 = MAKE_STATS_G1(segU0);
```

```
    }
    // If true: Create it.
    else {
      segU0
      = MAKE_SEG(
        (UP_SEG_L(idL0) | OUT_OF_TILE_BIT),
         CALC_MINID_U(idL0)
      );
      statU0 = EX__N__SUC_NOT_IN_TILE;
    }
  } else {
    statU0 = SEG_NOT_EXISTING;
  }

  if(ONE_BIT_SET(pixelProps1, BIT_RIGHT_SEG_EXISTING)) {
    segR1 = seg_L[RIGHT_SEG_L(idL1)];
    seg_L[RIGHT_SEG_L(idL1)]
     = SET_MINID(
       segR1,
       min(
         GET_MINID(segR1),
         CALC_MINID_R(pixelProps1, idL1)
       )
     );
    statR1 = MAKE_STATS_G1(segR1);
  } else {
    statR1 = SEG_NOT_EXISTING;
  }

  if(ONE_BIT_SET(pixelProps1, BIT_LEFT_SEG_EXISTING)) {
    segL1 = seg_L[LEFT_SEG_L(idL1)];
    seg_L[LEFT_SEG_L(idL1)]
     = SET_MINID(
         segL1,
         min(
           GET_MINID(segL1),
           CALC_MINID_L(pixelProps1, idL1)
        )
      );
    statL1 = MAKE_STATS_G1(segL1);
  } else {
    statL1 = SEG_NOT_EXISTING;
  }

  if(ONE_BIT_SET(pixelProps1, BIT_DOWN_SEG_EXISTING)) {
    // Check if OOT-segment
    // If not: Load and actualize it, as above
    if(!DOWN_EDGE_PIXEL_G1(yL1)) {
      segD1 = seg_L[DOWN_SEG_L(idL1)];
      seg_L[DOWN_SEG_L(idL1)]
       = SET_MINID(
           segD1,
```

```
          min(
            GET_MINID(segD1),
            CALC_MINID_D(pixelProps1, idL1)
          )
        );
    statD1 = MAKE_STATS_G1(segD1);
  } else {
  // If true: Create it.
    segD1
     = MAKE_SEG(
         (DOWN_SEG_L(idL1) | OUT_OF_TILE_BIT),
          CALC_MINID_D(pixelProps1, idL1)
        );
    statD1 = EX__N__SUC_NOT_IN_TILE;
  }
} else {
  statD1 = SEG_NOT_EXISTING;
}

if(ONE_BIT_SET(pixelProps1, BIT_UP_SEG_EXISTING)) {
  segU1 = seg_L[UP_SEG_L(idL1)];
  seg_L[UP_SEG_L(idL1)]
   = SET_MINID(
       segU1,
       min(
         GET_MINID(segU1),
         CALC_MINID_U(idL1)
       )
    );
  statU1 = MAKE_STATS_G1(segU1);
} else {
  statU1 = SEG_NOT_EXISTING;
}
// End setze Datenformat B

int segRS0, segLS0, segDS0, segUS0, segRS1, segLS1, segDS1, segUS1;


/*
Pointer Jumps A - Werden garantiert drei mal angewendet

Allgemeiner Hinweis zu den Pointer Jumps:

Fuer OOT-Segmente gilt immer: EX__N__SUC_IN_TILE(...) = FALSE.
Somit fuehren sie keine Pointer Jumps aus und greifen auf ihre
seg_L Eintraege nicht zu.

Fuer Segmente deren suc ein OOT-Segmente der DST-Typen UP oder DOWN
und der erste von LEFT ist, gilt ebenfalls:
EX__N__SUC_IN_TILE(...) = FALSE. Damit fuehren sie keinen pointer
Jumps aus und greifen insbesondere nicht auf ihre Nachfolger zu.
Deren Daten haben sie bereits bei der Initialisierung verarbeitet.
```

```
Segmente, deren Nachfolger ein OOT-Segment der DST-Typen RIGHT oder
LEFT (ausser erstes Element) ist, greifen einmal auf diese zu.

Die OOT-Info wird weitergereicht, und Segmente, welche sie erhalten
werden fuer die Pointer Jumps inakttiv gesetzt (SUC_IN_TILE <- FALSE)
Somit greifen keine weiteren Segmente auf OOT-Segmente zu
*/

#pragma unroll
for(int i = 0; i < 3; i++){

  // Ensure completeness of data initialization (i = 0) or
  // completeness of previous pointer jump (i > 0)
  barrier(CLK_LOCAL_MEM_FENCE);

  // Consider first segment (R0)
  // Execute pointer jump for segment s, if s exists and (current)
  // s.suc is within tile
  if(EX__N__SUC_IN_TILE(statR0)){
    // Load successor from local memory and store in private memory
    segRS0 = seg_L[GET_SUC(segR0)];

    // If current successor's OOT-Bit is set,
    // set SUC_IN_TILE <- FALSE
    if(OUT_OF_TILE(segRS0))
      statR0 = EX__N__SUC_NOT_IN_TILE;

    // Determine new minId out of this.minId and this.suc.minId
    // Store in segRS0, since there is already the new suc address
    // and OOT-info
    SET_MINID(segRS0, min(GET_MINID(segR0), GET_MINID(segRS0)));

    // Update segR0 and additionally store in local memory
    seg_L[RIGHT_SEG_L(idL0)] = segR0 = segRS0;

  }

  // Consider segment L0 (analogous to segment R0)
  if(EX__N__SUC_IN_TILE(statL0)){
    segLS0 = seg_L[GET_SUC(segL0)];
    if(OUT_OF_TILE(segLS0))
      statL0 = EX__N__SUC_NOT_IN_TILE;
    SET_MINID(segLS0, min(GET_MINID(segL0), GET_MINID(segLS0)));
    seg_L[LEFT_SEG_L(idL0)] = segL0 = segLS0;
  }

  // Consider segment D0 (analogous to segment R0)
  if(EX__N__SUC_IN_TILE(statD0)){
    segDS0 = seg_L[GET_SUC(segD0)];
    if(OUT_OF_TILE(segDS0))
      statD0 = EX__N__SUC_NOT_IN_TILE;
```

```
    SET_MINID(segDS0, min(GET_MINID(segD0), GET_MINID(segDS0)));
    seg_L[DOWN_SEG_L(idL0)] = segD0 = segDS0;
  }

  // Consider segment U0 (analogous to segment R0)
  if(EX__N__SUC_IN_TILE(statU0)){
    segUS0 = seg_L[GET_SUC(segU0)];
    if(OUT_OF_TILE(segUS0))
      statU0 = EX__N__SUC_NOT_IN_TILE;
    SET_MINID(segUS0, min(GET_MINID(segU0), GET_MINID(segUS0)));
    seg_L[UP_SEG_L(idL0)] = segU0 = segUS0;
  }

  // Consider segments R1, L1, D1 and U1 of lower Pixel
  // (analogous to segment R0)

  if(EX__N__SUC_IN_TILE(statR1)){
    segRS1 = seg_L[GET_SUC(segR1)];
    if(OUT_OF_TILE(segRS1))
      statR1 = EX__N__SUC_NOT_IN_TILE;
    SET_MINID(segRS1, min(GET_MINID(segR1), GET_MINID(segRS1)));
    seg_L[RIGHT_SEG_L(idL1)] = segR1 = segRS1;
  }

  if(EX__N__SUC_IN_TILE(statL1)){
    segLS1 = seg_L[GET_SUC(segL1)];
    if(OUT_OF_TILE(segLS1))
      statL1 = EX__N__SUC_NOT_IN_TILE;
    SET_MINID(segLS1, min(GET_MINID(segL1), GET_MINID(segLS1)));
    seg_L[LEFT_SEG_L(idL1)] = segL1 = segLS1;
  }

  if(EX__N__SUC_IN_TILE(statD1)){
    segDS1 = seg_L[GET_SUC(segD1)];
    if(OUT_OF_TILE(segDS1))
      statD1 = EX__N__SUC_NOT_IN_TILE;
    SET_MINID(segDS1, min(GET_MINID(segD1), GET_MINID(segDS1)));
    seg_L[DOWN_SEG_L(idL1)] = segD1 = segDS1;
  }

  if(EX__N__SUC_IN_TILE(statU1)){
    segUS1 = seg_L[GET_SUC(segU1)];
    if(OUT_OF_TILE(segUS1))
      statU1 = EX__N__SUC_NOT_IN_TILE;
    SET_MINID(segUS1, min(GET_MINID(segU1), GET_MINID(segUS1)));
    seg_L[UP_SEG_L(idL1)] = segU1 = segUS1;
  }

}
```

```
/*
Position zum Austausch der Information, ob weitere PointerJumps
auszufuehren sind wird mit FALSE (Keine Pointer Jumps noetig)
initialisiert. Keine Barrier davor notwendig, da ein OOT-Segment
ueberschrieben wird.  Diese fuehren keine Pointerjumps aus und
schreiben daher nicht in seg_L.
*/
seg_L[MORE_WORK] = FALSE;

/*
Pointer Jumps B - Werden maximal acht mal angewendet (insgesamt damit
maximal 11 = 3 + 8 Pointer Jumps) Wenn in einer Iteration kein
Segment des Tiles seg_L[MORE_WORK] auf TRUE setzt, wird die Schleife
und damit die Pointer Jumps beendet. Die Hinweise zu den Pointer
Jumps A gelten weiterhin.
*/
#pragma unroll
for(int i = 0; i < 11 - 3; i++){

  // Ensure completeness of initialization of seg_L[MORE_WORK] above
  // (i = 0) and completeness of previous pointer jumps (any i)
  barrier(CLK_LOCAL_MEM_FENCE);

  // Consider first segment (R0)
  // Execute pointer jump for segment s, if s exists and (current)
  // s.suc is within tile
  if(EX__N__SUC_IN_TILE(statR0)){
    // Load successor from local memory and store in private memory
    segRS0 = seg_L[GET_SUC(segR0)];

    // If current successor's OOT-Bit is set,
    // set SUC_IN_TILE <- FALSE
    if(OUT_OF_TILE(segRS0))
      statR0 = EX__N__SUC_NOT_IN_TILE;

    // Check if this segment has more work to do
    if(GET_MINID(segR0) != GET_MINID(segRS0) && !OUT_OF_TILE(segRS0))
      seg_L[MORE_WORK] = TRUE;

    // Determine new minId out of this.minId and this.suc.minId
    // Store in segRS0, since there is already the new suc address
    // and OOT-info
    SET_MINID(segRS0, min(GET_MINID(segR0), GET_MINID(segRS0)));

    // Update segR0 and additionally store in local memory
    seg_L[RIGHT_SEG_L(idL0)] = segR0 = segRS0;

  }

  // Consider segment L0 (analogous to segment R0)
  if(EX__N__SUC_IN_TILE(statL0)){
    segLS0 = seg_L[GET_SUC(segL0)];
```

```
    if(OUT_OF_TILE(segLS0))
      statL0 = EX__N__SUC_NOT_IN_TILE;
    if(GET_MINID(segL0) != GET_MINID(segLS0) && !OUT_OF_TILE(segLS0))
      seg_L[MORE_WORK] = TRUE;
    SET_MINID(segLS0, min(GET_MINID(segL0), GET_MINID(segLS0)));
    seg_L[LEFT_SEG_L(idL0)] = segL0 = segLS0;
  }

  // Consider segment D0 (analogous to segment R0)
  if(EX__N__SUC_IN_TILE(statD0)){
    segDS0 = seg_L[GET_SUC(segD0)];
    if(OUT_OF_TILE(segDS0))
      statD0 = EX__N__SUC_NOT_IN_TILE;
    if(GET_MINID(segD0) != GET_MINID(segDS0) && !OUT_OF_TILE(segDS0))
      seg_L[MORE_WORK] = TRUE;
    SET_MINID(segDS0, min(GET_MINID(segD0), GET_MINID(segDS0)));
    seg_L[DOWN_SEG_L(idL0)] = segD0 = segDS0;
  }

  // Consider segment U0 (analogous to segment R0)
  if(EX__N__SUC_IN_TILE(statU0)){
    segUS0 = seg_L[GET_SUC(segU0)];
    if(OUT_OF_TILE(segUS0))
      statU0 = EX__N__SUC_NOT_IN_TILE;
    if(GET_MINID(segU0) != GET_MINID(segUS0) && !OUT_OF_TILE(segUS0))
      seg_L[MORE_WORK] = TRUE;
    SET_MINID(segUS0, min(GET_MINID(segU0), GET_MINID(segUS0)));
    seg_L[UP_SEG_L(idL0)] = segU0 = segUS0;
  }

  // Consider segments R1, L1, D1 and U1 of lower Pixel (analogous
  // to segment R0)
  if(EX__N__SUC_IN_TILE(statR1)){
    segRS1 = seg_L[GET_SUC(segR1)];
    if(OUT_OF_TILE(segRS1))
      statR1 = 1;
    if(GET_MINID(segR1) != GET_MINID(segRS1) && !OUT_OF_TILE(segRS1))
      seg_L[MORE_WORK] = TRUE;
    SET_MINID(segRS1, min(GET_MINID(segR1), GET_MINID(segRS1)));
    seg_L[RIGHT_SEG_L(idL1)] = segR1 = segRS1;
  }

  if(EX__N__SUC_IN_TILE(statL1)){
    segLS1 = seg_L[GET_SUC(segL1)];
    if(OUT_OF_TILE(segLS1))
      statL1 = EX__N__SUC_NOT_IN_TILE;
    if(GET_MINID(segL0) != GET_MINID(segLS1) && !OUT_OF_TILE(segLS1))
      seg_L[MORE_WORK] = TRUE;
    SET_MINID(segLS1, min(GET_MINID(segL1), GET_MINID(segLS1)));
    seg_L[LEFT_SEG_L(idL1)] = segL1 = segLS1;
  }
```

```
  if(EX__N__SUC_IN_TILE(statD1)){
    segDS1 = seg_L[GET_SUC(segD1)];
    if(OUT_OF_TILE(segDS1))
      statD1 = EX__N__SUC_NOT_IN_TILE;
    if(GET_MINID(segD1) != GET_MINID(segDS1) && !OUT_OF_TILE(segDS1))
      seg_L[MORE_WORK] = TRUE;
    SET_MINID(segDS1, min(GET_MINID(segD1), GET_MINID(segDS1)));
    seg_L[DOWN_SEG_L(idL1)] = segD1 = segDS1;
  }

  if(EX__N__SUC_IN_TILE(statU1)){
    segUS1 = seg_L[GET_SUC(segU1)];
    if(OUT_OF_TILE(segUS1))
      statU1 = EX__N__SUC_NOT_IN_TILE;
    if(GET_MINID(segU1) != GET_MINID(segUS1) && !OUT_OF_TILE(segUS1))
      seg_L[MORE_WORK] = TRUE;
    SET_MINID(segUS1, min(GET_MINID(segU1), GET_MINID(segUS1)));
    seg_L[UP_SEG_L(idL1)] = segU1 = segUS1;
  }


  // Make sure, each Segment has executed its pointer jump
  barrier(CLK_LOCAL_MEM_FENCE);

  // If no single segment has reported more work to do:
  // Exit pointer jump loop
  if(seg_L[MORE_WORK] == FALSE)
    break;

  // Give each work item a chance to quit the loop
  barrier(CLK_LOCAL_MEM_FENCE);

  // Then re-initialize more work as FALSE again
  if(yL0 == 0 && xL == 0)
    seg_L[MORE_WORK] = FALSE;
}



////////////////////// Analyse and write results ///////////////////////



/*
Betrachtung von Segmenten, die zu innerhalb des Tiles geschlossenen
Segmenten gehoeren.
Falls sie zu aeusseren Konturen gehoeren, erhalten zugehoerige Pixel
ihr Label. Die Segmente werden anschliessend entfernt.
Falls sie zu inneren Konturen gehoeren, werden die Segmente und
zusaetzlich die assoziierte Information ueber io-Kanten entfernt
*/
```

```
// Initialize temporary data-structure for labels as undefined
// Zero, one or two segments may set a label for each of them
int pixelLabel0 = UNDEF;
int pixelLabel1 = UNDEF;

// If statR0 belongts to a contour, which is closed in tile
if(EX__N__CLOSED(statR0)){
  // Remove segment
  pixelProps0 &= DESELECT_BIT_RIGHT_SEG_EXISTING;
  // IF segR0 is part of an outer contour
  if(OUTER(GET_MINID(segR0))) {
    // True: Compute Label
    pixelLabel0 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segR0));
  } else {
    // False: Remove io-Edge information
    pixelProps0 &= DESELECT_BIT_INOUT_CONTOUR_RIGHT_SEG;
  }
}

// Next 7: Consider remaining 7 segments in a similar way

if (EX__N__CLOSED(statL0)){
  pixelProps0 &= DESELECT_BIT_LEFT_SEG_EXISTING;
  if(OUTER(GET_MINID(segL0))) {
    pixelLabel0 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segL0));
  } else {
    pixelProps0 &= DESELECT_BIT_INOUT_CONTOUR_LEFT_SEG;
  }
}

if (EX__N__CLOSED(statD0)){
  pixelProps0 &= DESELECT_BIT_DOWN_SEG_EXISTING;
  if(OUTER(GET_MINID(segD0))) {
    pixelLabel0 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segD0));
  } else {
    pixelProps0 &= DESELECT_BIT_INOUT_CONTOUR_DOWN_SEG;
  }
}

if (EX__N__CLOSED(statU0)){
  pixelProps0 &= DESELECT_BIT_UP_SEG_EXISTING;
  if(OUTER(GET_MINID(segU0))) {
    pixelLabel0 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segU0));
  } else {
    pixelProps0 &= DESELECT_BIT_INOUT_CONTOUR_UP_SEG;
  }
}

if(EX__N__CLOSED(statR1)){
  pixelProps1 &= DESELECT_BIT_RIGHT_SEG_EXISTING;
  if(OUTER(GET_MINID(segR1))) {
    pixelLabel1 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segR1));
```

```
  } else {
    pixelProps1 &= DESELECT_BIT_INOUT_CONTOUR_RIGHT_SEG;
  }
}

if (EX__N__CLOSED(statL1)){
  pixelProps1 &= DESELECT_BIT_LEFT_SEG_EXISTING;
  if(OUTER(GET_MINID(segL1))) {
    pixelLabel1 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segL1));
  } else {
    pixelProps1 &= DESELECT_BIT_INOUT_CONTOUR_LEFT_SEG;
  }
}

if (EX__N__CLOSED(statD1)){
  pixelProps1 &= DESELECT_BIT_DOWN_SEG_EXISTING;
  if(OUTER(GET_MINID(segD1))) {
    pixelLabel1 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segD1));
  } else {
    pixelProps1 &= DESELECT_BIT_INOUT_CONTOUR_DOWN_SEG;
  }
}

if (EX__N__CLOSED(statU1)){
  pixelProps1 &= DESELECT_BIT_UP_SEG_EXISTING;
  if(OUTER(GET_MINID(segU1))) {
    pixelLabel1 = LABEL_TRANSFORM_G1(tileX, tileY, GET_MINID(segU1));
  } else {
    pixelProps1 &= DESELECT_BIT_INOUT_CONTOUR_UP_SEG;
  }
}

// Write properties of remaining segmnets and io-inf
// of the two pixels considered
pixel_SegsProperties[id0] = pixelProps0;
pixel_SegsProperties[id1] = pixelProps1;

// If and only if we found a label, write it
if(pixelLabel0 != UNDEF) {
  pixel_Label[id0] = pixelLabel0;
}
if(pixelLabel1 != UNDEF) {
  pixel_Label[id1] = pixelLabel1;
}


// Initialize some tile related indices (used later).
// Index of the second generation tile, this tile belongs to...
int tile_G2 = TILE_CNT_X_G2 * (tileY / SUB_TILES_Y_G1)
            + tileX / SUB_TILES_X_G1;
// ...and its offset. Unit: segments
int tileOffset_G2 = TILE_SIZE_G2 * tile_G2;
```

```
// x-coordinate of this tile within its 2G-tile. Unit: tiles_1G
int subTileX = tileX % SUB_TILES_X_G1;

// y-coordinate of this tile within its 2G-tile. Unit: tiles_1G
int subTileY = tileY % SUB_TILES_Y_G1;

// Index of this tile within its 2G-tile (= sub-tile index)
//  RIGHT and LEFT segs are transposed
int subTileRL = subTileX * SUB_TILES_Y_G1 + subTileY;
//  DOWN and UP segs are not transposed
int subTileDU = subTileY * SUB_TILES_X_G1 + subTileX;


/*
Betrachtung von Segmenten, die zu nicht innerhalb des Tiles
geschlossenen Segmenten gehoeren.

Es werden die Adressen ihrer Nachfolger innerhalb des Tiles
geschrieben. Dieses sind immer OOT-Segmente. An diesen Positionen
wird sich spaeter (im Kernel passLabels_G1) das Label befinden.
*/
if(EX__N__OOT_G1(statR0)) {
  segsTile1_Suc[RIGHT_SEG(id0)] = GET_SUC(segR0);
}
if(EX__N__OOT_G1(statL0)) {
  segsTile1_Suc[LEFT_SEG(id0) ] = GET_SUC(segL0);
}
if(EX__N__OOT_G1(statD0)) {
  segsTile1_Suc[DOWN_SEG(id0) ] = GET_SUC(segD0);
}
if(EX__N__OOT_G1(statU0)) {
  segsTile1_Suc[UP_SEG(id0)   ] = GET_SUC(segU0);
}
if(EX__N__OOT_G1(statR1)) {
  segsTile1_Suc[RIGHT_SEG(id1)] = GET_SUC(segR1);
}
if(EX__N__OOT_G1(statL1)) {
  segsTile1_Suc[LEFT_SEG(id1) ] = GET_SUC(segL1);
}
if(EX__N__OOT_G1(statD1)) {
  segsTile1_Suc[DOWN_SEG(id1) ] = GET_SUC(segD1);
}
if(EX__N__OOT_G1(statU1)) {
  segsTile1_Suc[UP_SEG(id1)   ] = GET_SUC(segU1);
}


/*
Betrachtung von Randsegmenten, die in nachfolgenden Kerneln weiter
verarbeitet werden muessen, um die vollstaendigen Konturen zu
bestimmen und alle Segmente mit der minimalen minId zu versehen.
```

```
Betrachtet werden immer die eintretenden Segmente, da nur sie die
minimale minId aller nachfolgenden Segmente haben. Dieses muss
jeweils transformiert werden, um auch in den 2G tiles,
wieder lokal, gueltig zu sein.

Die Segmente werden in der G2 gemaess SRC-Typ angeordnet. Bestimmt
werden muss (fast) immer die Adresse des nachfolgenden  Segments
in den Tiles der zweiten Generation. Diese kann berechnet werden,
da Adresse, und damit DST-Typ, des jeweiligen OOT-Segments
bekannt sind.
Ist ein Segment in der naechsten Generation OOT-Segment, wird die
suc-Information genutzt, um Informationen zu hinterlegen, mit denen
nach der zweiten Generation von Tiles der Nachfolger berechnet
werden kann. Dies ist insbesondere keine Adresse und erfordert in
nachfolgenden kernel Sonderbehandlung.

Nachfolgend gibt es einige weitere Sonderfaelle. Schliesslich sind
bestimmte OOT-Segmente (UP, DOWN, das erste LEFT) nicht im local
Memory hinterlegt.
Das ist genau dann ein Problem, wenn ein Eintrittssegment
gleichzeitig auch OOT Segment ist.

Es gilt hier insbesondere zu beachten, dass in einigen Faellen, wenn
wir uns gerade im 'richtigen' work-item befinden, manche private
Daten einfach weiterverwendet werden koennen.
*/

// Some temporary data structures
int suc, stat, offset_G2;

// Consider Left tile-edge and contained segments,
// which enter the tile from left <=> SRC LEFT
if(yL0 == 8){ // Some 32 consecutive work-items

  // Vertical edges use transposed access:
  // xL in (0, 1, ..., 31) is used as row index

  stat = SEG_NOT_EXISTING;
  suc = UNDEF;

  int comesFromInfoLEFT = seg_L[POS_IN_FIRST_UPSEG_ROW_G1(xL)];

  // Compute global offset of this seg in its G2 tile. Unit: segments
  int offset_G2
    = tileOffset_G2
    + FROM_L_OFFSET_G2
    + subTileRL * SEGS_RLDU_G1
    + xL;

  // If a segment with SRC LEFT exists in row xL
  if(comesFromInfoLEFT != UNDEF){
```

```
// Get considered segment and store it in private data structure
segLS0 =
  // Check if considered seg has one of the following DST-types:
  (
    (xL == 0 && // In uppermost pixel and...
      (
        comesFromInfoLEFT == UP || // heading up or...
        comesFromInfoLEFT == LEFT  // heading left
      )
    )
    ||
    (
      xL == TILE_Y_G1_MINUS_1 &&  // Or in lowest pixel and...
      comesFromInfoLEFT == DOWN // heading down
    )

  )
  ?
  // If true, the segment needs to be recomputed
  MAKE_SEG(
    // Calc suc <- this. Can be done using pixel id and DST-type
    GET_SEG_L(
      comesFromInfoLEFT, xL * TILE_X_G1
    ),
    // Calc minId:
                  // 2 * idL =
                     2 * xL * TILE_X_G1
                  // + 0 (SRC LEFT, DST LEFT)
                  // + 1 (SRC LEFT, DST UP)
                  // + 1 (SRC LEFT, DST DOWN)
                  + ((comesFromInfoLEFT != LEFT) ? 1 : 0)

  )
  :
  // If not true, load segment from local memory
  seg_L[GET_SEG_L(comesFromInfoLEFT, xL * TILE_X_G1)];


suc = GET_SUC(segLS0);

// Convert segment to one of the next generation and write its
// data to global memory
segsTile2_Data[offset_G2]
 = segLS0
 = MAKE_SEG(
    calcSucIn_Tiles_G2(
      suc, subTileX, subTileY, subTileRL, subTileDU
    ),
    LABEL_TRANSFORM_G1G2 (
      subTileX, subTileY,
      GET_MINID (
```

```
            segLS0
          )
        )
      );
    // Update status information
    stat = MAKE_STATS_G2(segLS0);
  }

  // Write address of suc in this tile to global memory. If defined,
  // the segs final minId will be written there in kernel
  // passLabels_G1.
  // Note: Must be writen for all elements, since a decicion depends
  // on it in kernel passLabels_I1.
  segsTile2_SucInTile1[offset_G2] = suc;
  // Write segments status to global memory
  segsTile2_Props     [offset_G2] = stat;
}

// Consider right tile-edge and contained segments,
// which enter the tile from right side <=> SRC RIGHT
else if (yL0 == 16) { // Another 32 consecutive work-items
  stat = SEG_NOT_EXISTING;
  suc = UNDEF;

  // Again: Vertical edges use transposed access:
  // xL in (0, 1, ..., 31) is used as row index

  int comesFromInfoRIGHT = seg_L[POS_IN_LAST_DOWNSEG_ROW_G1(xL)];

  int offset_G2
    = tileOffset_G2
    + FROM_R_OFFSET_G2
    + subTileRL * SEGS_RLDU_G1
    + xL;

  if(comesFromInfoRIGHT != UNDEF){

    segRS0 =
    // Check if considered seg has one of the following DST-types:
      (
        // In uppermost pixel and heading up or...
        (xL == 0 && comesFromInfoRIGHT == UP) ||
        // In lowest pixel and heading down
        (xL == TILE_Y_G1_MINUS_1 && comesFromInfoRIGHT == DOWN)
      )
      ?
      // If true, the segment needs to be recomputed
      MAKE_SEG
      (
        // Calc suc <- this. Can be done using pixel id and DST-type
        GET_SEG_L(
          comesFromInfoRIGHT, xL * TILE_X_G1 + TILE_X_G1_MINUS_1
```

```
        ),
        //// Calc minId:
                      // 2 * idL =
                        (2 * (xL * TILE_X_G1 + TILE_X_G1_MINUS_1)
                      // + 0 (SRC RIGHT, DST DOWN)
                      // + 1 (SRC RIGHT, DST UP)
                        + ((xL == 0) ? 1 : 0))
      )
      :
      // If not true, load segment from local memory
      seg_L[
        GET_SEG_L(
          comesFromInfoRIGHT, xL * TILE_X_G1 + TILE_X_G1_MINUS_1
        )
      ];

    suc = GET_SUC(segRS0);

    segsTile2_Data  [offset_G2]
    = segRS0
    = MAKE_SEG(
      calcSucIn_Tiles_G2(
        suc, subTileX, subTileY, subTileRL, subTileDU
      ),
      LABEL_TRANSFORM_G1G2(
        subTileX, subTileY,
        GET_MINID(
          segRS0
        )
      )
    );
    stat = MAKE_STATS_G2(segRS0);
  }

  segsTile2_SucInTile1[offset_G2] = suc;
  segsTile2_Props     [offset_G2] = stat;
}


// Consider upper tile-edge and contained segments,
// which enter the tile from above <=> SRC UP
else if(UP_EDGE_PIXEL_G1(yL0)) { // Must be the first 32
                                 // consecutive work-items
  stat = SEG_NOT_EXISTING;
  suc = UNDEF;

  // Horizontal edges do not use transposed access:
  // xL in (0, 1, ..., 31) is used as column index

  int offset_G2
    = tileOffset_G2
    + FROM_U_OFFSET_G2
```

```
    + subTileDU * SEGS_RLDU_G1
    + xL;

  if(comesFromInfoDU != UNDEF){

    // This time, all work-items are in a fitting row. (The first
    // one, as ensured above).
    // We can simply use the data stored in private memory:
    if(comesFromInfoDU == LEFT)
      segU0 = segL0;
    else if(comesFromInfoDU == DOWN)
      segU0 = segD0;
    else if(comesFromInfoDU == RIGHT)
      segU0 = segR0;
    // else:
    //  keep segU0

    suc = GET_SUC(segU0);

    segsTile2_Data[offset_G2]
      = segU0
      = MAKE_SEG(
          calcSucIn_Tiles_G2(
            suc, subTileX, subTileY, subTileRL, subTileDU
          ),
          LABEL_TRANSFORM_G1G2(
            subTileX, subTileY,
            GET_MINID(
              segU0
            )
          )
        );
    stat = MAKE_STATS_G2(segU0);
  }

  segsTile2_SucInTile1[offset_G2] = suc;
  segsTile2_Props     [offset_G2] = stat;
}


// Consider lower tile-edge and contained segments,
// which enter the tile from below <=> SRC DOWN
else if(DOWN_EDGE_PIXEL_G1(yL1)) { // Must be the last 32
                                   // consecutive work-items
  stat = SEG_NOT_EXISTING;
  suc = UNDEF;

  int offset_G2
    = tileOffset_G2
    + FROM_D_OFFSET_G2
    + subTileDU * SEGS_RLDU_G1
    + xL;
```

```
    if(comesFromInfoDU != UNDEF){

      // If not SRC DOWN, DST DOWN, load segment from local memory
      if(comesFromInfoDU != DOWN)
        segD1
        = seg_L[
            GET_SEG_L(
              comesFromInfoDU, xL + TILE_X_G1 * TILE_Y_G1_MINUS_1
            )
          ];
      // else: Keep ata stored in private memory
      // Note: Alternatively, we could have done it in a similat same
      // way as in case of SRC UP

      suc = GET_SUC(segD1);

      segsTile2_Data[offset_G2]
      = segD1
      = MAKE_SEG(
        calcSucIn_Tiles_G2(
          suc, subTileX, subTileY, subTileRL, subTileDU
        ),
        LABEL_TRANSFORM_G1G2(
          subTileX, subTileY,
          GET_MINID(
            segD1
          )
        )
      );
      stat = MAKE_STATS_G2(segD1);
    }
    segsTile2_SucInTile1[offset_G2] = suc;
    segsTile2_Props     [offset_G2] = stat;

  }
//// End part II: Pointer Jumps ////////////////////////////////////
////////////////////////////////////////////////////////////////////////

}
// End Kernel extractSegs_N_perTilePJ_G1
```

Listing 23.2: OpenCL C Code für Kernel extractSegs_N_TilePJ_G1

23.6. Evaluation und Fazit

Vergleichen wir nun experimentell die beschriebene Impl IV mit den bisherigen Implementationsstrategien Impl I GPU, Impl I CPU, Impl II und Impl III. Die Ergebnisse bei Verarbeitung des Spiral- und Ones-Datensatzes für einige Datenauflösungen sind in Abbildung 23.10 gegeben. Dabei wird die Impl I CPU auf der Vierkern CPU Intel Core

i7 2700k ausgeführt und alle GPU Fassungen verwenden die GTX 670. Bei Verarbeitung

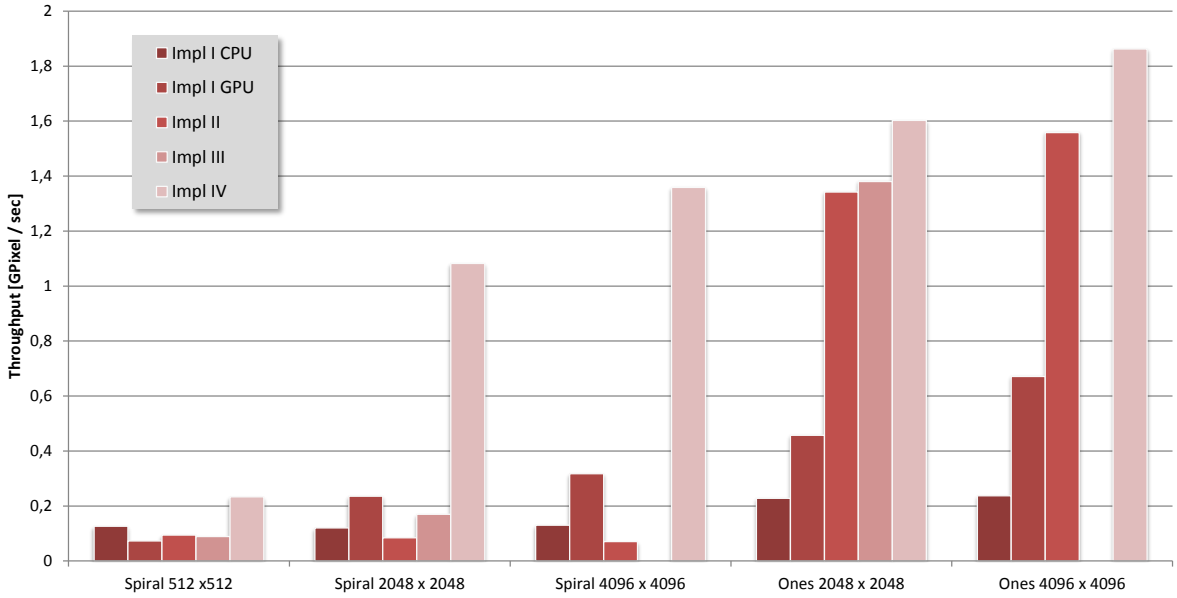


Abbildung 23.10.: Vergleich aller Implementationsstrategien dieser Arbeit bei Verarbeitung einiger Datensätze. Devices: Core i7 2700k (CPU), GTX 670 (GPU, sonst)

des Spiral-Datensatzes erreicht Impl IV weitaus höhere Throughputs als die anderen Ansätze. Die Zielsetzung, eine Fassung zu formulieren, welche gerade bei Datensätzen mit hohem Kontursegmentanteil schnell ist, kann somit als erfüllt angesehen werden. Wie erwartet, profitiert Impl IV weniger von Datensätzen mit geringem Konturanteil als Impl II und Impl III. Diese verarbeiten, z.B. in der Auflösung 2048 x 2048 den Ones Datensatz weitaus schneller als den Spiral-Datensatz. Diese Unterschiede betragen Faktor 16 (Impl II) und Faktor 8 (Impl III). Im Gegensatz dazu ist Impl IV bei Verarbeitung des Ones-Datensatzes lediglich um Faktor 1,5 schneller als im Falle des Spiral-Datensatzes. Insofern ist es bemerkenswert, dass Impl IV trotzdem, in diesem für Impl II und Impl III idealen Fall, schneller ist.

Infolgedessen kann Impl IV im Vergleich mit allen anderen vorgestellten Varianten bei Ausführung auf einer GPU als uneingeschränkt überlegen eingestuft werden.

Gleiches gilt auch für eine weitere Implementationsstrategie, welche im Rahmen eines Zwischenstands der vorliegenden Arbeit [WKV14] veröffentlicht wurde. Hierbei handelt es sich um einen hybriden Ansatz, welcher Elemente von Impl I GPU und Impl III vereinigt. Dabei werden, solange die Parallelität ausreicht, die Segmente im Stil von Impl I GPU verarbeitet. Sobald das nicht mehr gilt, werden die restlichen Segmente analog zu Impl III verarbeitet. Der Ansatz ist im Falle mancher Datensätze etwas schneller als Impl I GPU oder Impl III alleine, aber auch er ist Impl IV immer unterlegen. Deshalb wird nachfolgend für GPUs ausschließlich die Fassung Impl IV weiter behandelt.

Impl I CPU bleibt dagegen, bedingt durch ihren Fokus auf CPUs, weiter relevant. Es ist aber an dieser Stelle bereits offensichtlich, dass sie bei einem direkten Vergleich mit Impl IV eindeutig unterlegen ist.

Wir ergänzen an dieser Stelle die Betrachtung des bislang nicht besprochenen Noise-Datensatzes, welcher sich im Falle aller konturbasierten Ansätze als besonders schwierig herausstellt. In Abbildung 23.11 sind die Laufzeiten der einzelnen Abschnitte der Impl IV bei Verarbeitung des Spiral-Datensatzes und des Noise-Datensatzes (mit 50 Prozent Anteil Einsen) durch die GTX 670 in der Auflösung 4096 x 4096 zu sehen. Deutlich sichtbar ist die weitaus höhere Laufzeit bei Verarbeitung des Noise-Datensatzes und speziell der Kernel extractSegs_N_TilePJ_G1 sowie Tile Pointer Jumps G2. Diese enthalten im Vergleich mit den übrigen Kerneln deutlich mehr datenabhängige Fallunterscheidungen. Wir schätzen daher, dass deshalb die im Falle des Noise-Datensatzes zu erwartende Warp-Divergenz hier größere Auswirkungen hat.

23.7. OpenCL Treiber und Cuda Portierung

Alle bisherigen Angaben zu eigenen Implementationen beziehen sich immer auf OpenCL und alle Messungen sind im Falle von GPUs immer mit Nvidia GPUs unter Verwendung des Treibers 337.88 durchgeführt worden. Ein entscheidender Teil der Optimierung der Impl IV stellt die Einteilung in Tiles dar, welche im Local-Memory abgelegt werden. Dafür verwendet die GPU ihr Shared-Memory. Wie besprochen, limitiert das vorhandene Shared-Memory die Parallelität gerade im Fall der Kepler Architektur. Deshalb sind die Tiles gerade so gewählt worden, dass die Menge des benötigten Shared-Memory ein ganzzahliger Teiler des vorhandenen Shared-Memory eines SM darstellt. Leider verbrauchen alle auf den Treiber 337.88 folgende Versionen bei Verwendung von OpenCL immer etwas mehr Shared-Memory, als angefordert wird. Auch wenn es sich dabei z.B. nur um drei Bytes handelt, genügt das, um ein ganzzahliges Aufteilen zu verhindern. Infolgedessen können bei Verarbeitung der G1-Tiles nur noch zwei anstelle von vorher drei Work-Groups je SM gestartet werden. Es kann somit lediglich noch 2/3 der vorherigen Thread-Auslastung erreicht werden. Die Auswirkungen, wenn wir den Treiber 337.88 mit einem

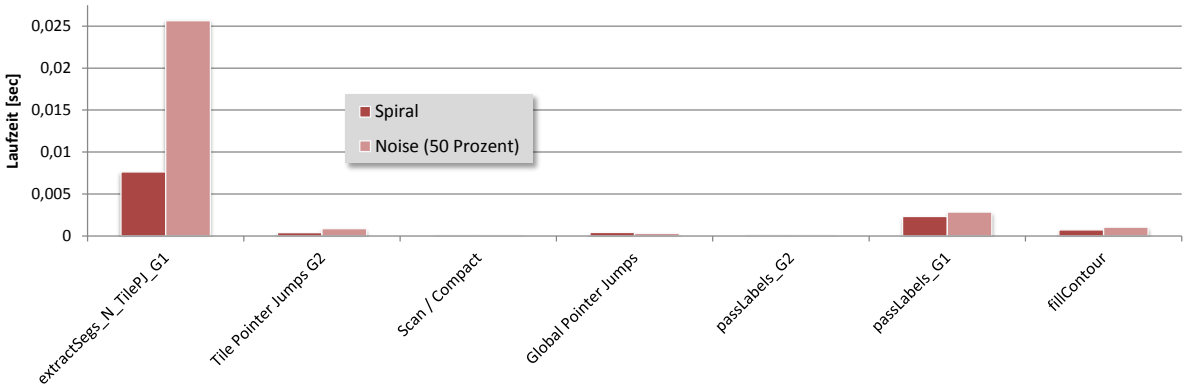


Abbildung 23.11.: Vergleich der Laufzeiten einzelner Abschnitte der Impl IV bei Verarbeitung der Spirale und Noise (50 Prozent Einsen) in der Datenauflösung 4096 x 4096. Device: GTX 670

aktuelleren (347.88) vergleichen, sind in Abbildung 23.12 dargestellt. Offenbar nimmt die Laufzeit beider G1-Kernel jeweils um ca. 40 Prozent zu.

Bei Verwendung von Cuda entspricht auch in neueren Versionen der Shared-Memory Verbrauch der angeforderten Menge und der oben beschriebene Effekt tritt somit nicht auf. Weil in dieser Arbeit auch aktuelle Nvidia Grafikkarten der Maxwell Generation evaluiert werden sollen, welche durch den Treiber 337.88 nicht unterstützt werden, ist eine Cuda Portierung der Impl IV erstellt worden. Bei dieser Gelegenheit überprüfen wir auch, ob die Verwendung von C++ anstelle von Java / LWJGL Auswirkungen auf das Laufzeitverhalten hat. Wir portieren daher in einem ersten Schritt Impl IV nach C++ unter Beibehaltung von OpenCL und evaluieren diese Fassungen mit dem aktuellen Treiber 347.88 im Vergleich. Die Ergebnisse sind in Abbildung 23.13 dargestellt. Offensichtlich ist der Unterschied zwischen Java und C++ vernachlässigbar. Anschließend portieren wir die C++ Fassung zur Verwendung der API Cuda. Dabei handelt es

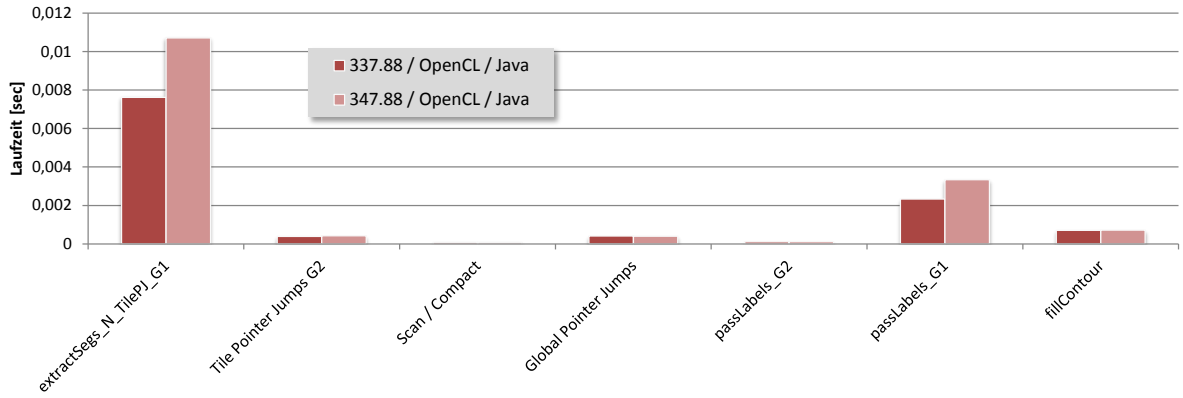


Abbildung 23.12.: Vergleich der Laufzeiten einzelner Abschnitte mit aktueller Treiberversion bei Verarbeitung der 4096 x 4096 Spirale. Device: GTX 670

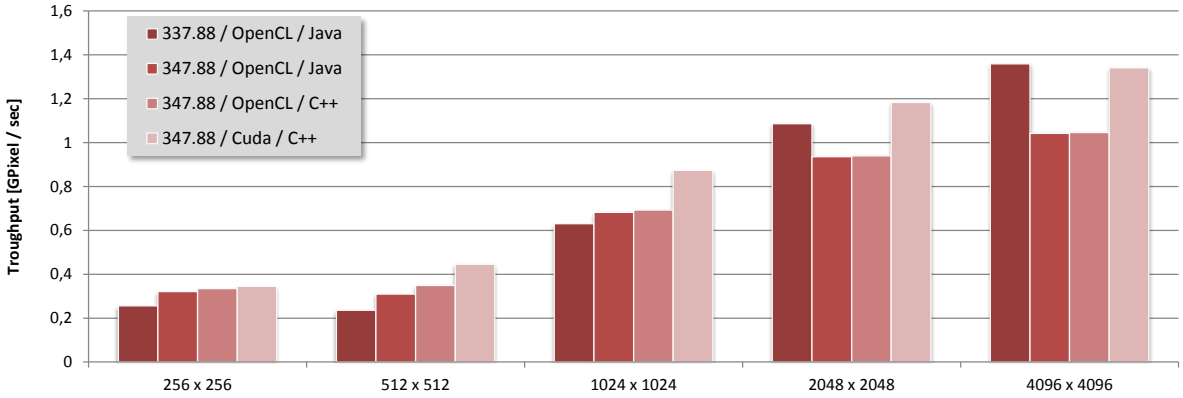


Abbildung 23.13.: Vergleich der Throughputs der gesamten Implementation Impl IV in verschiedenen Host-Sprachen, APIs und Treiberversionen bei Verarbeitung der Spirale in verschiedenen Datenauflösungen. Device: GTX 670

sich um eine eins zu eins Portierung, welche lediglich OpenCL spezifisches durch unmittelbare Entsprechungen in Cuda ersetzt. Das ist im Falle der Kernel (fast immer) durch simple Textersetzung möglich. Insbesondere wird keinerlei Cuda exklusive Funktionalität verwendet. Die Ergebnisse der Evaluation dieser Fassung sind ebenfalls in Abbildung 23.13 zu sehen. Die Cuda Fassung erreicht nun in der Auflösung 4096 x 4096, verglichen mit der OpenCL Fassung unter Verwendung des alten Treibers, einen vergleichbaren Throughput.

Unabhängig davon fällt der höhere Throughput in den geringeren Datenauflösungen sowohl von OpenCL als auch von Cuda bei Verwendung des neuen Treibers auf. Das ist auch ein Stück weit in Bezug auf Messungen geringer Datenauflösung in vorherigen Kapiteln zu bedenken. Allerdings liegt der Fokus in dieser Arbeit auf höheren Auflösungen und die von nun an verwendete Cuda Fassung leistet sowieso beides. Ihr Name bleibt, trotz der Portierung nach Cuda, Impl IV. Diese Variante ersetzt nachfolgend die entsprechende OpenCL Implementation.

Zuletzt stellen wir noch fest, dass die eins zu eins Cuda Portierung, abgesehen vom Ausgleich des Shared-Memory 'Features' neuerer Nvidia OpenCL Treiber, keine weitere Verbesserung bewirkt hat. Das deckt sich auch mit anderen Beobachtungen [FVS11] von Portierungen von OpenCL nach Cuda ohne weitere Anpassung an dessen erweiterte Funktionalität. Andere, z.B. [KDH10] berichten dagegen von generell schnellerer Ausführungszeit der Cuda Kernel trotz nahezu identischen Codes. Die Ergebnisse können aber auch, wie in dieser Arbeit deutlich wird, von der verwendeten Treiberversion abhängen.

23.7.1. Ressourcenverbrauch der Cuda Fassung

Der Ressourcenverbrauch der wichtigsten Cuda Kernel ist für die in dieser Arbeit verwendeten Compute-Capabilities in Tabellen 23.3, 23.4, 23.5 und 23.6 angegeben. Der Kernel `TilePointerJumps G2` wird mit der Konfiguration LWS(1024, 1, 1) gestartet. Seine Ausführung bewirkt im Falle von Compute-Capability 2.0 Register-Spilling. Der Kernel `extractSegs_N_TilePJ_G1` wird im Falle von Compute-Capability 2.0 und 5.2 in etwas abgewandelter Form verwendet. Hier werden nicht alle der oben besprochenen Optimierungen zum Einsparen von Shared-Memory eingesetzt. Das erhöht zwar den Shared-Memory-Verbrauch aber senkt die Anzahl benötigter Register. Das ist vor allem bei Compute-Capability 2.0 wichtig, um nicht über 32 Register je Thread zu verbrauchen. Dieser Schritt war erforderlich, da beim Wechsel zu Cuda und dem neueren Treiber der Registerverbrauch etwas angestiegen ist und so (ohne diese Anpassung) nur noch eine Work-Group (statt zwei) je SM gestartet werden konnte. Im Falle von Compute-Capability 5.2 ist der Unterschied nicht groß.

Tabelle 23.3.: Ressourcenverbrauch einiger Kernel bei Compute-Capability 2.0

	Register (R)	Shared-Memory (S)	Thread-Auslastung	Limitiert durch
extractSegs_N_TilePJ_G1	32	16640	67 %	R/S
TilePointerJumps G2	32	32768	67 %	R/S
passLabels G1	21	16384	67 %	R
fillContours	42	0	50 %	R

Tabelle 23.4.: Ressourcenverbrauch einiger Kernel bei Compute-Capability 3.0

	Register (R)	Shared-Memory (S)	Thread-Auslastung	Limitiert durch
extractSegs_N_TilePJ_G1	35	16384	75 %	R/S
TilePointerJumps G2	54	32768	50 %	R/S
passLabels G1	22	16384	75 %	S
fillContours	43	0	63 %	R

Tabelle 23.5.: Ressourcenverbrauch einiger Kernel bei Compute-Capability 3.5

	Register (R)	Shared-Memory (S)	Thread-Auslastung	Limitiert durch
extractSegs_N_TilePJ_G1	35	16384	75 %	R/S
TilePointerJumps G2	53	32768	50 %	R/S
passLabels G1	22	16384	75 %	S
fillContours	44	0	63 %	R

Tabelle 23.6.: Ressourcenverbrauch einiger Kernel bei Compute-Capability 5.2

	Register (R)	Shared-Memory (S)	Thread-Auslastung	Limitiert durch
extractSegs_N_TilePJ_G1	38	16640	75 %	R
TilePointerJumps G2	48	32768	50 %	R
passLabels G1	30	16384	100 %	
fillContours	42	0	63 %	R

Kapitel 24.

Vergleich mit Union-Find auf GPUs

In diesem Kapitel vergleichen wir den eigenen Ansatz Impl IV mit Stavas [OS11] Cuda Implementation der Union-Find-Technik, welche die schnellste mir bekannte ist. In neueren Cuda Versionen existieren eine Reihe von Funktionen, welche aufgrund der Ausrichtung auf OpenCL bei Optimierung für Nvidia GPUs nicht verwendet werden konnten. Allerdings entspricht der von Stava verwendete Cuda Funktionsumfang, mit Ausnahme der von ihnen benötigten Atomic-Functions, Impl IV. Stavas Implementation ist für die Fermi Architektur optimiert und im Original-Paper mit der GTX 480 evaluiert, einer Grafikkarte, die auch in dieser Arbeit verwendet wird. Insgesamt erscheint der Vergleich somit fair und stellt damit die primäre Referenz in dieser Arbeit dar. Wir evaluieren zunächst das Verhalten jedes Ansatzes für sich und stellen anschließend beide für einen Vergleich der absoluten Werte direkt gegenüber.

Anmerkung: Die Ziele der experimentellen Untersuchung werden in Kapitel 4 beschrieben. Details zur Messmethodik finden sich in Kapitel 18, eine Beschreibung der Datensätze in 19, Details zu Impl IV in Kapitel 23 und Angaben zur Optimierung sowie verwendeten Devices in Kapitel 17.

Evaluation des Verhaltens des Ansatzes von Stava

Wir betrachten zunächst im Rahmen dieser Arbeit erstellte Messungen mit Stavas Implementation für die verschiedenen GPUs und Datensätze, deren Ergebnisse in Abbildung 24.1 gegeben sind.

Im Falle der Spirale (oben in Abbildung 24.1) in verschiedenen Datenauflösungen ist eine deutliche Zunahme des Throughputs mit jeder höheren Auflösung erkennbar. Dieses Verhalten kennen wir bereits von der Mehrheit der eigenen GPU-basierten Fassungen.

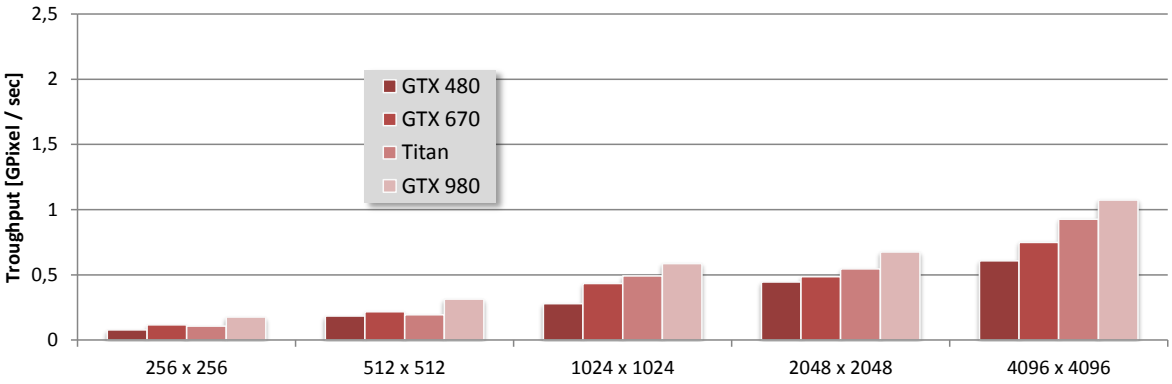
Bei Betrachtung der Messungen des Quads-Datensatzes (zweiter von oben in Abbildung 24.1) stellen wir eine Abnahme des Throughputs bei Zunahme der Quad-Größe fest. Ein solches Verhalten konnte, basierend auf Stavas Beobachtungen, auch erwartet werden.

Nicht besprochen wurde dagegen von Stava der Noise-Datensatz (dritter von oben in Abbildung 24.1). Dieser bewirkt bei allen evaluierten Konfigurationen im Vergleich mit anderen Datensätzen geringe Throughputs. Dabei nimmt der Throughput mit zunehmendem Anteil der Einsen ab. Eine mögliche Erklärung stellen die zunehmend größeren

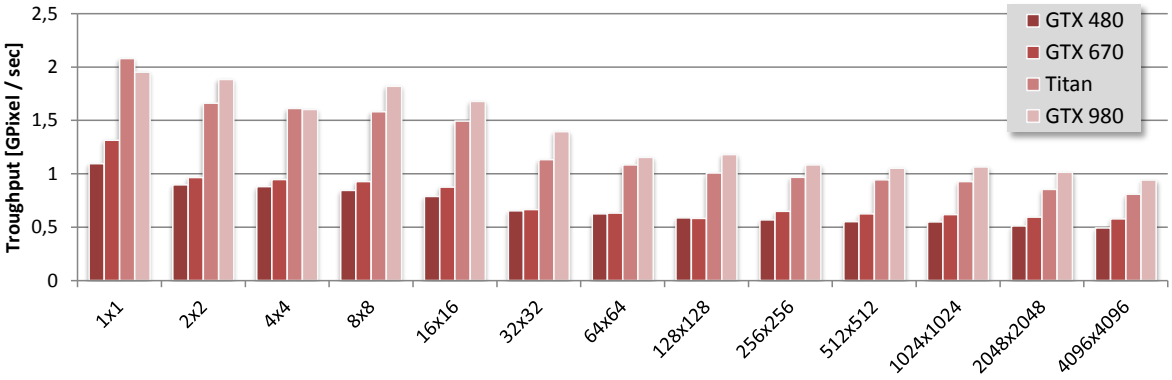
Connected-Components dar, welche generell für diesen Ansatz schwierig sind, und nun außerdem unregelmäßig.

Unten in Abbildung 24.1 ist eine Auswahl verschiedener Datensätze gegenübergestellt. Wie erwartet zählt der Ones-Datensatz, welche die Größe zusammenhängender Connected-Components maximiert, zu den langsamsten. Ebenso langsam ist jedoch auch der Noise-Datensatz (hier 50 Prozent Einsen). Der Unterschied zwischen dem Zeros-Datensatz (Best-Case) und einem besonders schwierigen (Ones) beträgt Faktor 3 (GTX 480) oder knapp 2,5 (GTX 980). Der Sieve-Datensatz ist etwas zügiger bearbeitet als die Spirale und der Nested Quads Datensatz wiederum schneller als der Sieve-Datensatz.

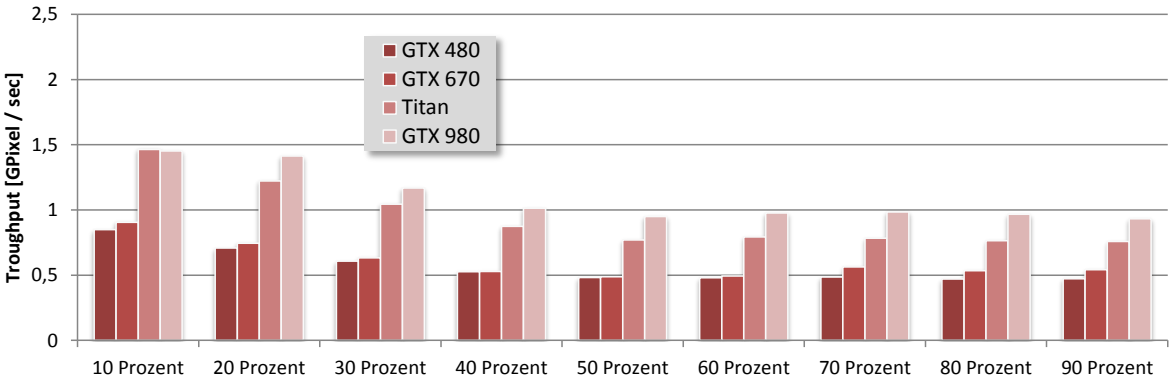
In Abhängigkeit der verwendeten Grafikkarten nimmt in allen Datensätzen der Throughput zu in der Reihenfolge von der GTX 480, über die GTX 670, die Titan bis hin zur GTX 980. Die Unterschiede sind jedoch nicht immer gleichermaßen ausgeprägt. Die GTX 980 ist fast immer doppelt so schnell wie die GTX 480 und die Titan meist nur etwas langsamer als die GTX 980. Stärker variiert dagegen die Performance der GTX 670. Diese ist oft etwa mittig zwischen der GTX 480 und Titan, aber in einigen Fällen nah oder gleichauf mit der GTX 480. Die Titan, ebenfalls Kepler Architektur aber mit doppelter SM-Zahl, erreicht sie nie.



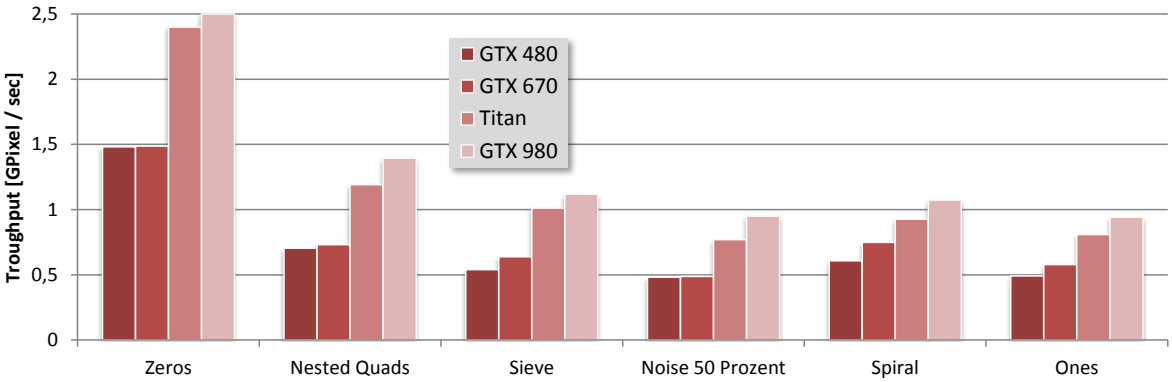
(a) Spiral in verschiedenen Datenauflösungen



(b) Quads verschiedener Größe. Datenauflösung 4096 x 4096



(c) Noise mit verschiedenem Anteil von Einsen. Datenauflösung 4096 x 4096



(d) Verschiedene Datensätze. Datenauflösung 4096 x 4096

Abbildung 24.1.: Messergebnisse mit Stavas Ansatz. Weitere Erläuterungen siehe Text

Evaluation des Verhaltens des eigenen Ansatzes

Betrachten wir nun die Ergebnisse der gleichen Versuchsanordnungen für Impl IV, welche in Abbildung 24.2 dargestellt sind. Im Falle der Spirale in verschiedenen Datenauflösungen ist eine deutliche Zunahme des Throughputs mit fast jeder höheren Auflösung erkennbar. Eine Ausnahme bildet der Wechsel von der 256 x 256 zur 512 x 512 Auflösung. Erstere stellt einen Sonderfall dar, weil die G2-Tiles dieser Auflösung entsprechen. Damit ist das Problem bereits lokal gelöst und alle Schritte bis zum Kernel `passLabels_G2` entfallen. Aufgrund des Fokus dieser Arbeit auf höhere Auflösungen beachten wir das nicht weiter.

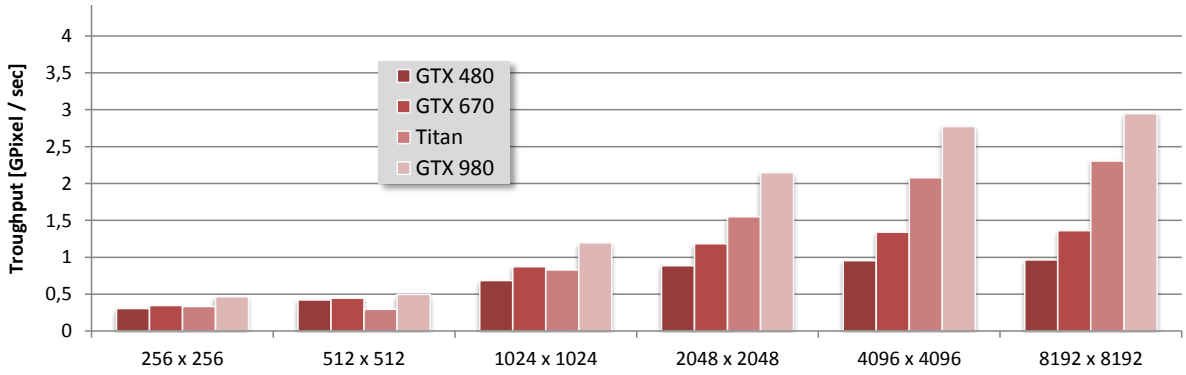
Im Falle der Quads verschiedener Größe muss differenziert werden. Quads der Größe 1 x 1 stellen bei den eigenen Ansätzen einen Sonderfall dar, welcher direkt erkannt wird. Dementsprechend sind die Throughputs immer hoch. Diesen Sonderfall beachten wir nicht weiter. Davon abgesehen bleibt der Throughput bis zur Quad-Größe 32 x 32 oft gleich oder sinkt sogar. Das ist der Fall, obwohl die Anzahl der Segmente sinkt. Ursache für das Verhalten sind längere Linienzüge innerhalb je eines Tiles, wodurch mehr Pointer-Jumps erforderlich werden. Ab 64 x 64 sind die Quads nicht mehr innerhalb eines Tiles geschlossen, weshalb einzelne Linienzüge (innerhalb der Tiles) vergleichsweise kurz bleiben. Infolgedessen steigt der Throughput mit jeder Vergrößerung der Quads. Somit ist das deutlich andere Verhalten bis 32 x 32 einerseits und ab 64 x 64 andererseits klar.

Bei Verarbeitung des Noise-Datensatzes durch Impl IV ermitteln wir im Vergleich mit anderen Datensätzen generell deutlich geringere Throughputs. Die Anzahl der Konturensegmente übertrifft (zumindest nicht in allen Einstellungen) diejenigen der Spirale nicht. Dies kann somit nicht die Ursache sein. Stattdessen ist bereits im Abschnitt 23.6 der ausgesprochen hohe Anteil der datenabhängigen Fallunterscheidungen in den ersten beiden Kerneln als mögliche Ursache für dieses Verhalten identifiziert worden. Besonders aufwändig ist die Verarbeitung von Noise mit einem Anteil von Einsen zwischen etwa 50 und 70 Prozent. Überwiegen Einsen oder Nullen, nehmen die Throughputs jeweils (zumindest leicht) weiter zu. Dies kann mit der jeweiligen Abnahme der Konturen erklärt werden. Wesentlich deutlicher ist die Throughput-Zunahme bei einem starken Übergewicht der Nullen. Das kann durch die dann hohe Wahrscheinlichkeit von isolierten Einsen erklärt werden. Diese stellen wieder den Sonderfall dar und es werden für sie keine Segmente erzeugt. Die Folge ist eine weitere Abnahme der Segmentzahlen. Im umgekehrten Fall (Übergewicht der Einsen) gilt das nicht.

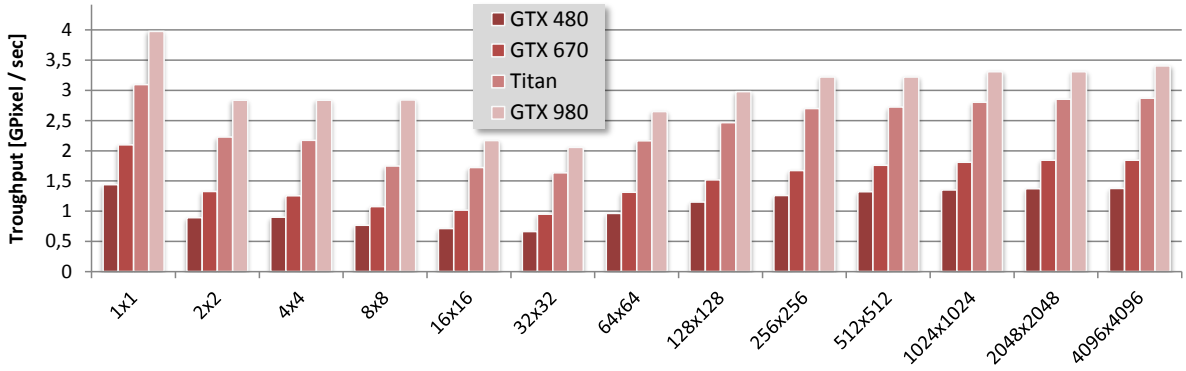
Unten in Abbildung 24.1 wird wieder eine Auswahl verschiedener Datensätze gegenübergestellt. Wie erwartet, ist der Throughput bei Verarbeitung des Zeros-Datensatz (Best-Case) deutlich weniger höher im Vergleich zum Ones-Datensatz (Faktor 1,3) als im Falle von Stavas Ansatz (Faktor 2,50 bis 3). Die Verarbeitung des Nested Quads Datensatzes ist lediglich geringfügig langsamer als die der Spirale. Daraus kann gefolgert werden, dass maximale Verschachtelung kein besonderes Problem für den eigenen Ansatz darstellt. Das haben wir bereits zuvor bei der Evaluation des `fillContours` Kernels festgestellt. Der Sieve-Datensatz ist noch etwas langsamer als der Nested Quads Datensatz, aber immer noch deutlich schneller als der Noise-Datensatz. Der größte Throughput-Unterschied besteht (hier) zwischen Noise (50 Prozent Einsen) und Zeros. Er beträgt

bei Ausführung auf der GTX 980 Faktor 3,7. Er ist damit größer als der maximale Unterschied (Faktor 2,5) bei Stava im Falle der GTX 980.

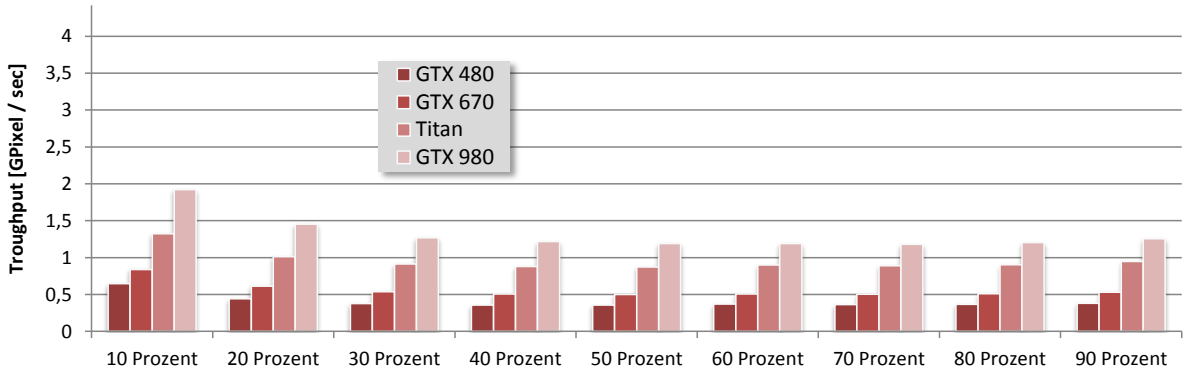
Hinsichtlich der verwendeten GPUs steigt der Throughput in fast allen Messungen von der GTX 480 zur GTX 670, über die Titan bis hin zur GTX 980. Die Reihenfolge entspricht somit der bei der Evaluation von Stavas Ansatz. Die GTX 980 ist zwischen Faktor 2,5 (Zeros, Ones) und Faktor 3,5 (Noise mit 50 Prozent Einsen) schneller als die GTX 480. Impl IV profitiert somit mehr von einer leistungsfähigeren GPU als der Ansatz von Stava (immer ca. Faktor 2).



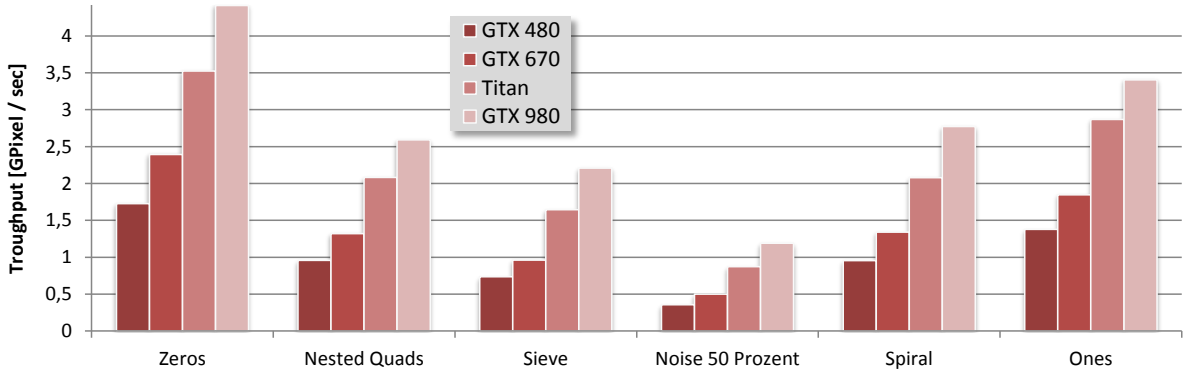
(a) Spiral in verschiedenen Datenauflösungen



(b) Quads verschiedener Größe. Datenauflösung 4096 x 4096



(c) Noise mit verschiedenem Anteil von Einsen. Datenauflösung 4096 x 4096



(d) Verschiedene Datensätze. Datenauflösung 4096 x 4096

Abbildung 24.2.: Messergebnisse mit dem eigenen Ansatz. Weitere Erläuterungen siehe Text

Direkter Vergleich

Stellen wir nun dieselben Ergebnisse der Evaluationen von Stavas Ansatz denen von Impl IV direkt gegenüber, um die Werte besser vergleichen zu können.

Betrachten wir zuerst den **Spiral**-Datensatz, dargestellt in Abbildung 24.3, für verschiedene Datenauflösungen und gruppiert nach der jeweils verwendeten GPU. Zu den einzelnen GPUs sind hier und in den folgenden Abbildungen immer (sehr grobe) Angaben zu deren Einordnung angegeben (Architektur, Markteinführung, Richtwert für Stromverbrauch).

Offenbar liefert Impl IV in allen Fällen deutlich bessere Ergebnisse als der Ansatz von Stava. Dieser Unterschied nimmt mit steigender GPU Leistung zu. So ist Impl IV (jeweils in der Auflösung 4096 x 4096) auf der GTX 480 um Faktor 1,6 schneller, auf der GTX 670 um Faktor 1,8, auf der Titan um Faktor 2,3 und auf der GTX 980 um Faktor 2,6. Der Anstieg des Throughputs beider Ansätze bei steigender Datenauflösung ist ähnlich. Beispielsweise steigt der Throughput von Stavas Ansatz mit einer GTX 980 beim Wechsel der Auflösungen von 1024 x 1024 zu 4096 x 4096 um Faktor 1,8. Im Falle von Impl IV beträgt der entsprechende Unterschied Faktor 2,3.

Schauen wir uns nun die Ergebnisse für den **Quads** Datensatz, dargestellt in Abbildung 24.4, zum Vergleich beider Ansätze an. Im Falle von Quads ab 64 x 64 Pixeln Größe ist Impl IV immer deutlich schneller und der Unterschied wächst mit zunehmender Quad-Größe. Bei Quads der Größe 32 x 32 und kleiner sind die Unterschiede deutlich geringer. Bei Ausführung auf der GTX 480 entsprechen die Ergebnisse von Impl IV etwa denen von Stava. Mit leistungsfähiger werdenden GPUs setzt sich der Throughput von Impl IV zunehmend nach oben hin ab.

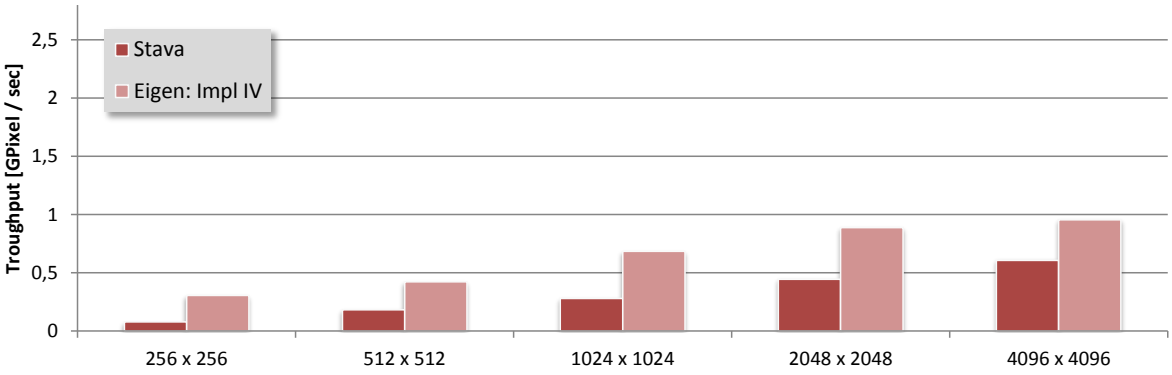
Vergleichen wir jetzt die Ergebnisse beider Ansätze bei Verarbeitung des Noise-Datensatzes mit verschiedenen Anteilen von Einsen, die in Abbildung 24.5 gegeben sind. Gerade bei diesem Datensatz ist die verwendete GPU entscheidend. Bei einem geringen Anteil von Einsen ($\leq$ 30 Prozent) ist Stavas Ansatz schneller bei allen GPUs außer der GTX 980. Im Falle eines höheren Anteils der Einsen ist Impl IV meist gleich schnell (GTX 670) oder schneller (Titan). Bei Verwendung der GTX 480 liefert Stavas Ansatz immer den höheren Throughput und bei der GTX 980 ist immer Impl IV schneller.

Betrachten wir zuletzt den Überblick mit verschiedenen Datensätzen, der in Abbildung 24.6 gegeben ist. Zu nennen sind hier zunächst die noch fehlenden Datensätze, Sieve und Nested Quads, in denen Impl IV immer schneller ist.

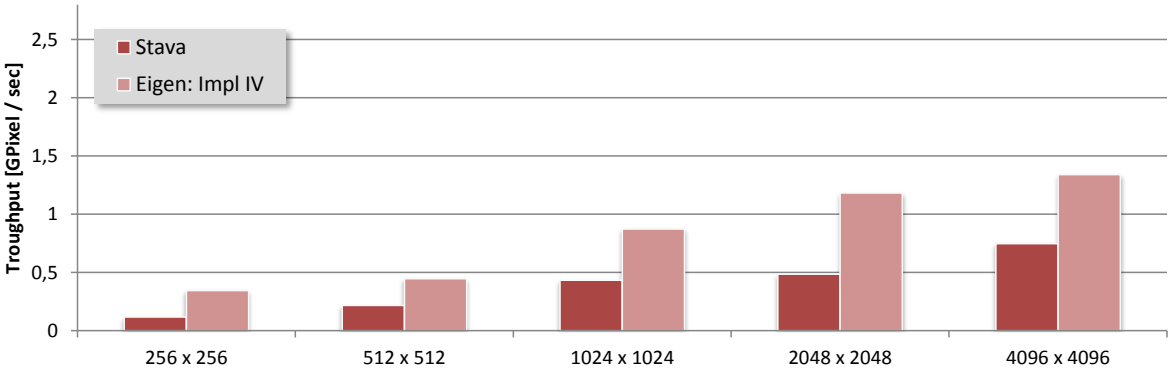
Unabhängig davon wird hier erneut deutlich, dass der Noise-Datensatz bei dem eigenen Ansatz zu einem stärkeren Einbruch des Throughputs im Vergleich mit anderen Datensätzen führt, als dies bei Stava der Fall ist. Eine denkbare Erklärung ist der weitaus höhere Anteil datenabhängiger Fallunterscheidungen von Impl IV, welcher bereits als mögliche Ursache für den Einbruch identifiziert wurde. Von allen durchgeführten Experimenten stellt der Noise-Datensatz mit 50 Prozent Einsen für Impl IV den schwierigsten

Fall dar. Hier ist Stava bei Verwendung der GTX 480 um 35 Prozent schneller. Bei Verwendung stärkerer GPUs wendet sich das Blatt und so ist Impl IV bei Einsatz der GTX 980 25 Prozent schneller.

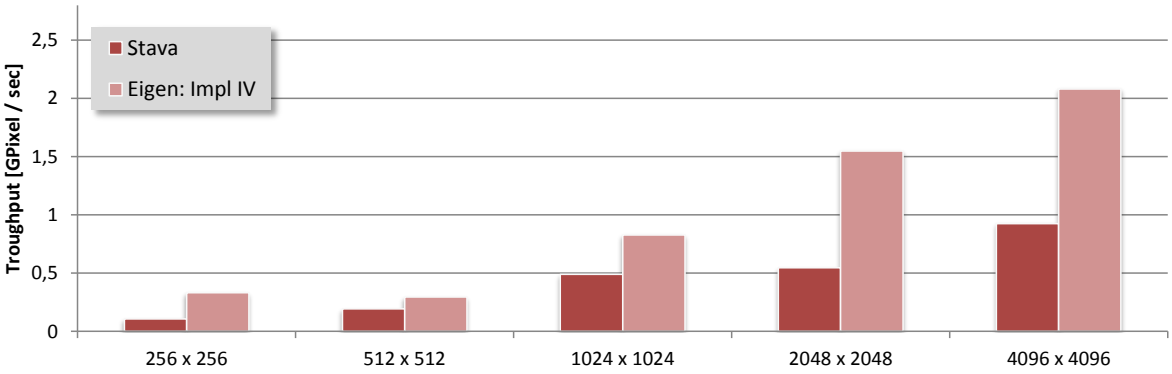
Der größte Unterschied zwischen den Throughputs der beiden Ansätze tritt beim Ones-Datensatz auf. Hier ist Impl IV zwischen Faktor 2,8 (GTX 480) und 3,6 (GTX 980) schneller als der Ansatz von Stava. Das war zu erwarten, weil Stava selbst maximal große Connected-Components als problematisch identifiziert hat und bei dem eigenen Ansatz das Verhältnis von Konturen zur Fläche im Inneren ideal ist. Allerdings ist der eigene Ansatz auch z.B. beim Spiral-Datensatz immer deutlich schneller, nämlich zwischen Faktor 1,6 (GTX 480) und Faktor 2,6 (GTX 980) im Falle der 4096 x 4096 Datenauflösung.



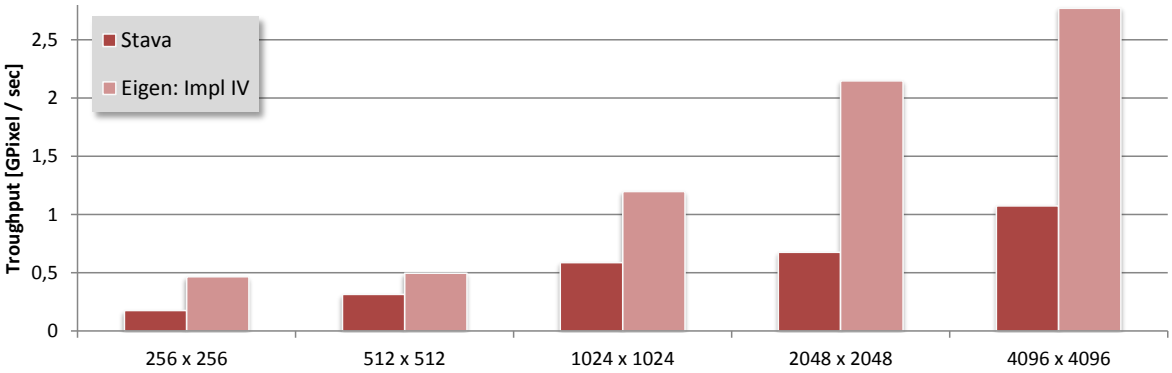
(a) Device: Nvidia Geforce GTX 480 (Fermi, März 2010, 250 W)



(b) Device: Nvidia Geforce GTX 670 (Kepler, Mai 2012, 170W)

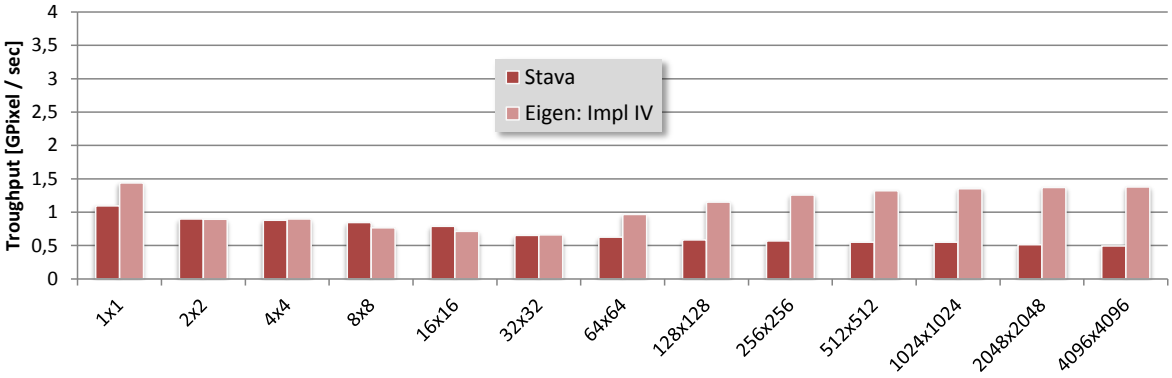


(c) Device: Nvidia Geforce GTX Titan (Kepler, Februar 2013, 250W)

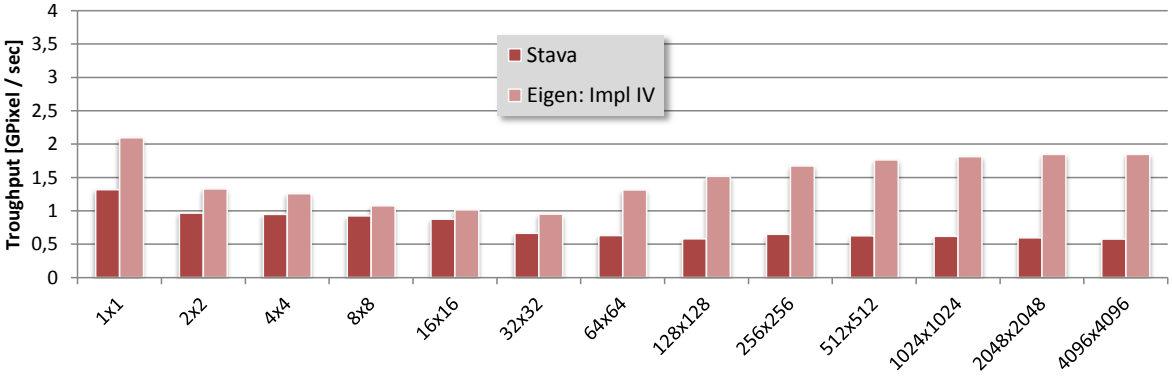


(d) Device: Nvidia Geforce GTX 980 (Maxwell, September 2014, 165W)

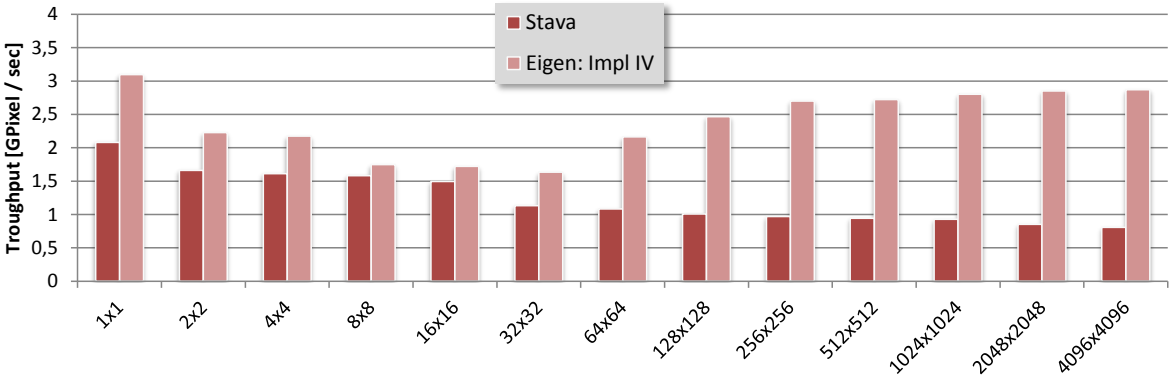
Abbildung 24.3.: Vergleich mit Stavas Ansatz. Daten: Spiral. Erläuterungen: Siehe Text



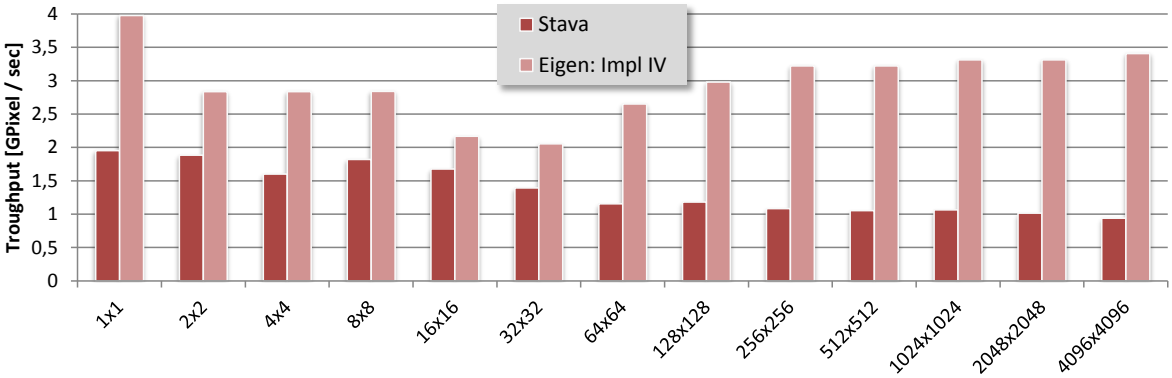
(a) Device: Nvidia Geforce GTX 480 (Fermi, März 2010, 250 W)



(b) Device: Nvidia Geforce GTX 670 (Kepler, Mai 2012, 170W)

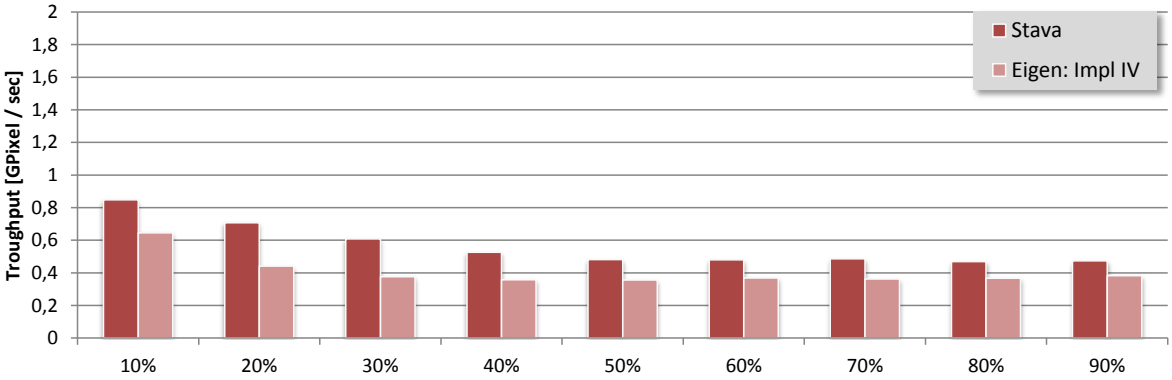


(c) Device: Nvidia Geforce GTX Titan (Kepler, Februar 2013, 250W)

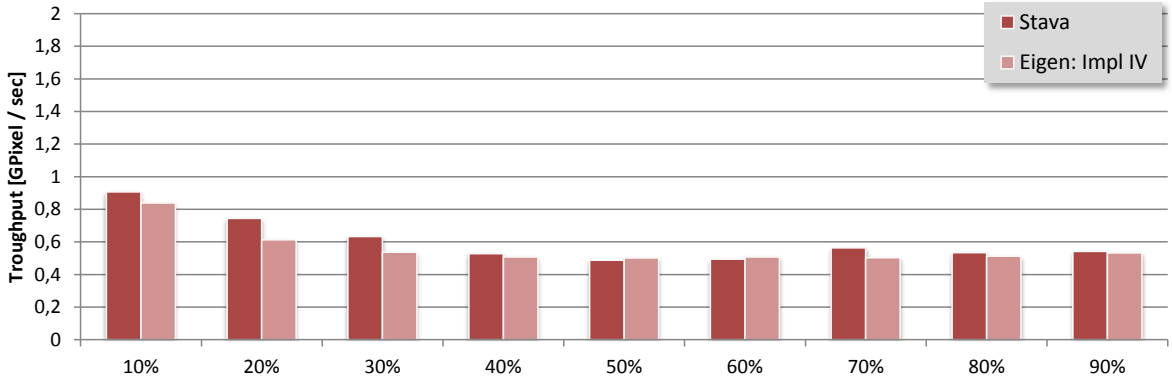


(d) Device: Nvidia Geforce GTX 980 (Maxwell, September 2014, 165W)

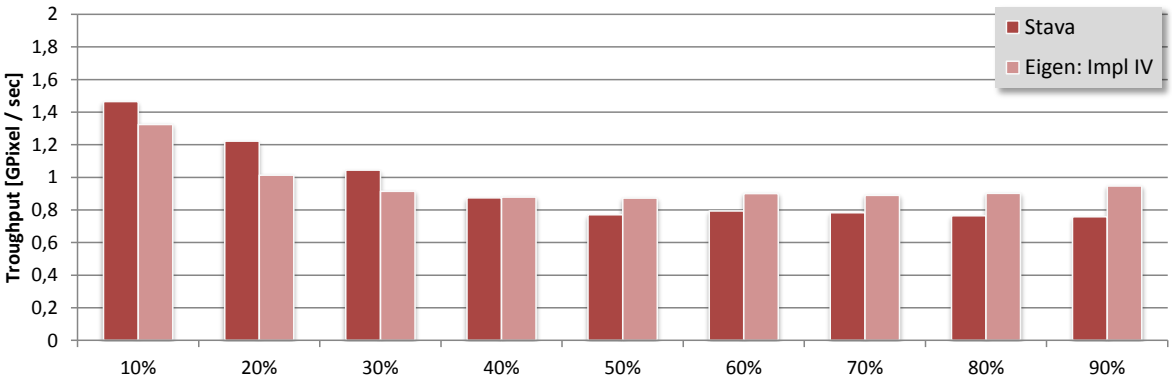
Abbildung 24.4.: Vergleich mit Stavas Ansatz. Daten: Quads. Erläuterungen: Siehe Text



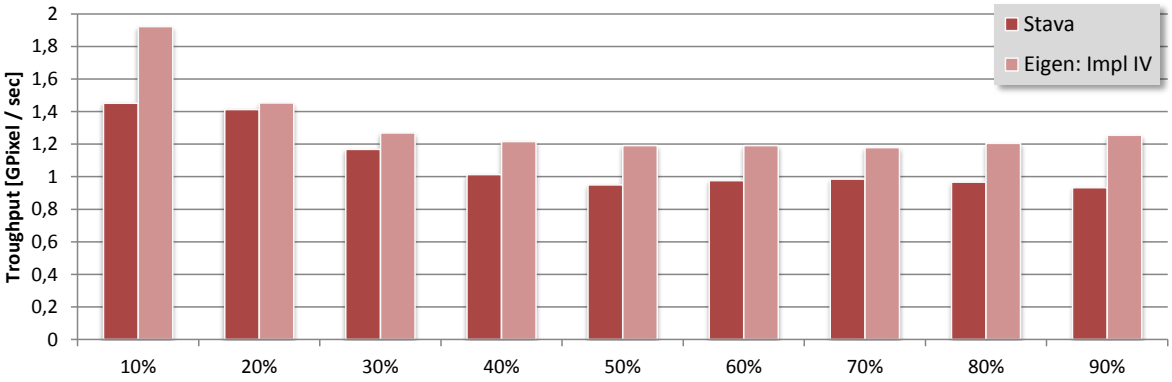
(a) Device: Nvidia Geforce GTX 480 (Fermi, März 2010, 250 W)



(b) Device: Nvidia Geforce GTX 670 (Kepler, Mai 2012, 170W)

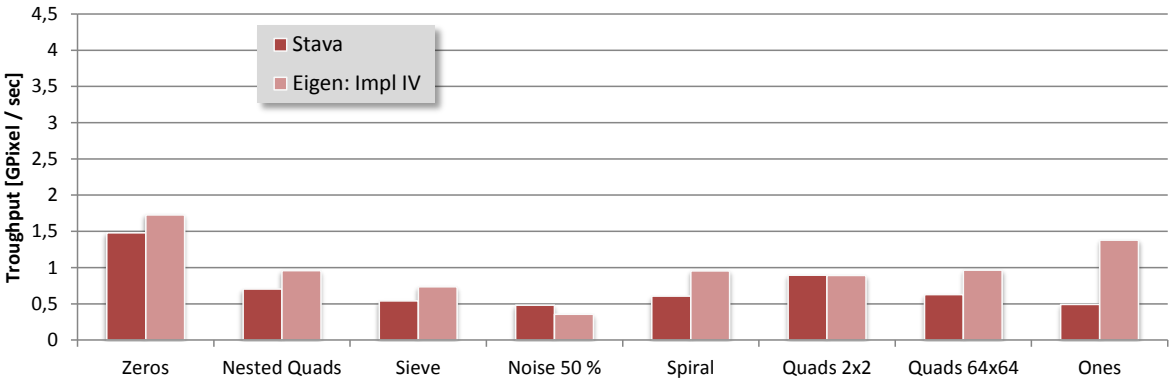


(c) Device: Nvidia Geforce GTX Titan (Kepler, Februar 2013, 250W)

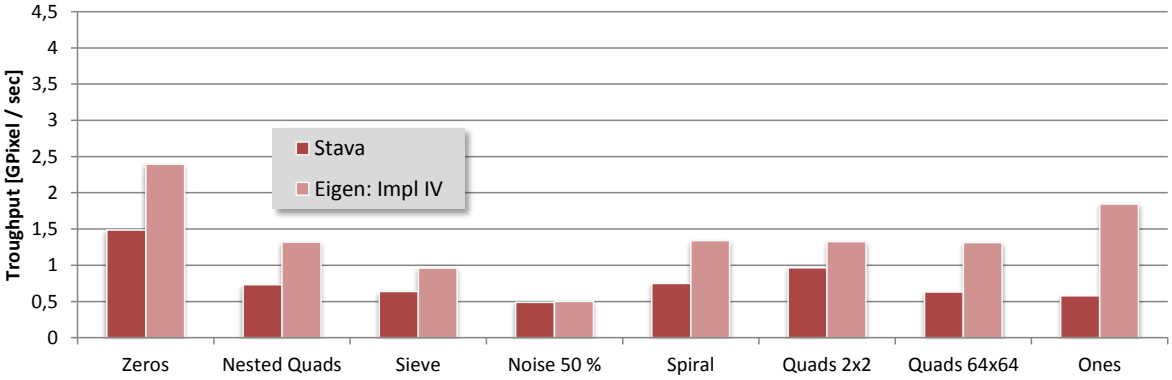


(d) Device: Nvidia Geforce GTX 980 (Maxwell, September 2014, 165W)

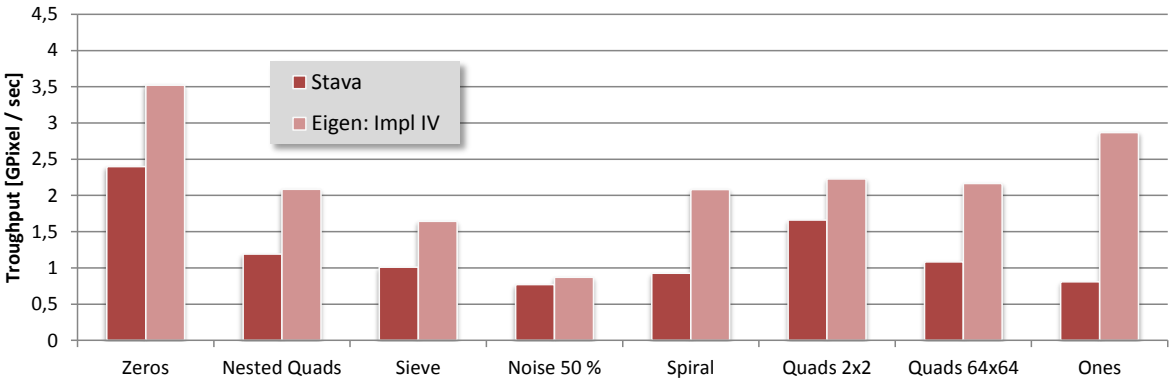
Abbildung 24.5.: Vergleich mit Stavas Ansatz. Daten: Noise. Erläuterungen: Siehe Text



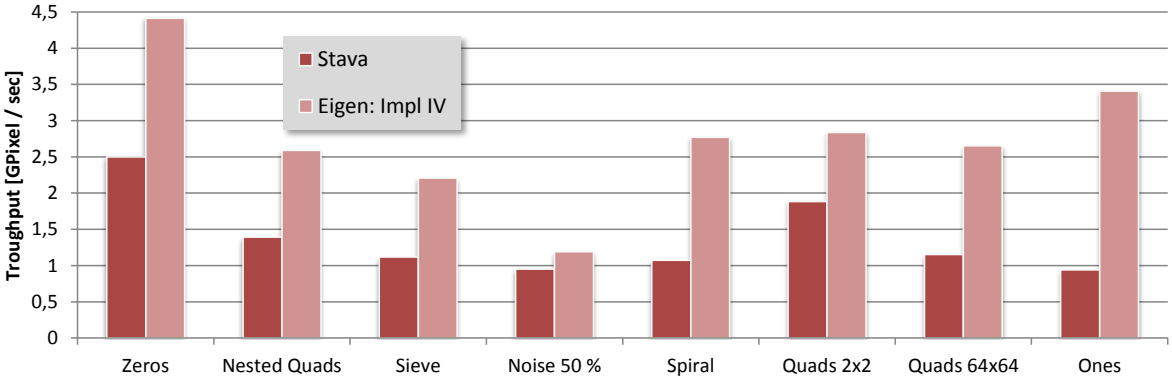
(a) Device: Nvidia Geforce GTX 480 (Fermi, März 2010, 250 W)



(b) Device: Nvidia Geforce GTX 670 (Kepler, Mai 2012, 170W)



(c) Device: Nvidia Geforce GTX Titan (Kepler, Februar 2013, 250W)



(d) Device: Nvidia Geforce GTX 980 (Maxwell, September 2014, 165W)

Abbildung 24.6.: Vergleich mit Stavas Ansatz. Daten: Verschieden. Erläuterungen: Text

Kapitel 25.

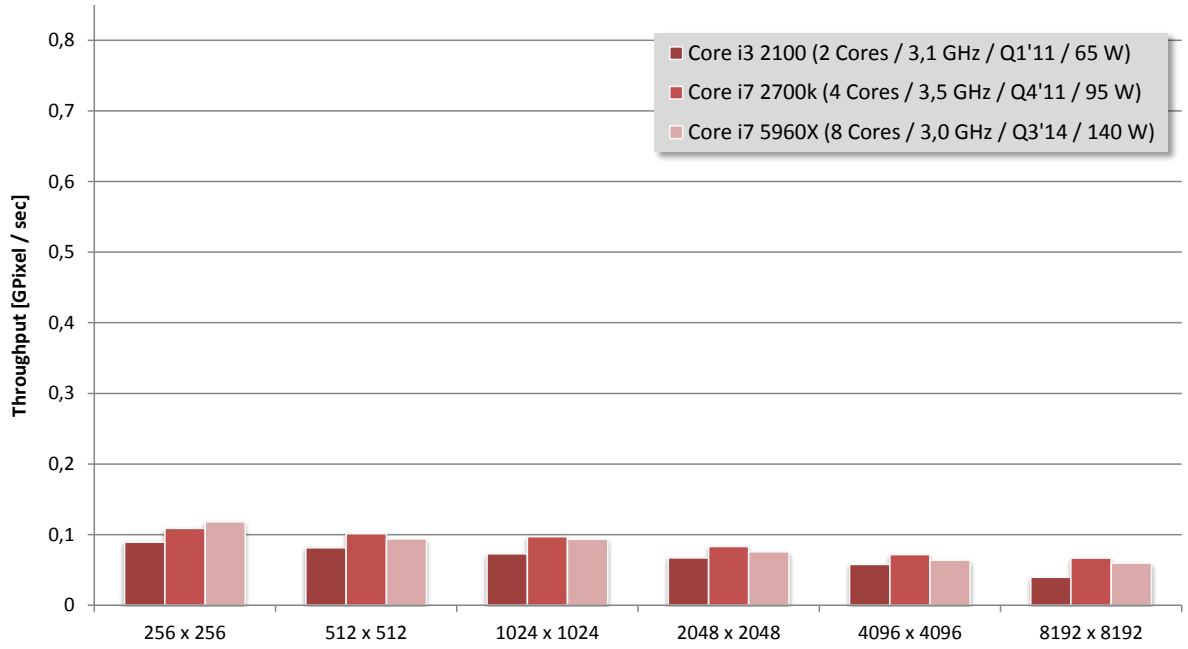
Vergleich mit Contour-Tracing auf CPUs

In diesem Kapitel wird der eigene Ansatz Impl I CPU mit der Implementation des sequentiellen linear-time Contour-Tracing-Algorithmus von Chang [CCL04] experimentell verglichen. Dafür wird die frei verfügbare Implementation ([CCL15]) verwendet, welche um Routinen zum Messen der Berechnungsdauer und um einige in dieser Arbeit verwendete Datensätze ergänzt wurde. Der Optimierungsgrad des gegebenen Codes ist als gering einzustufen. Da dieses auch auf den von Impl I CPU zutrifft, erscheint er damit für einen fairen Vergleich geeignet.

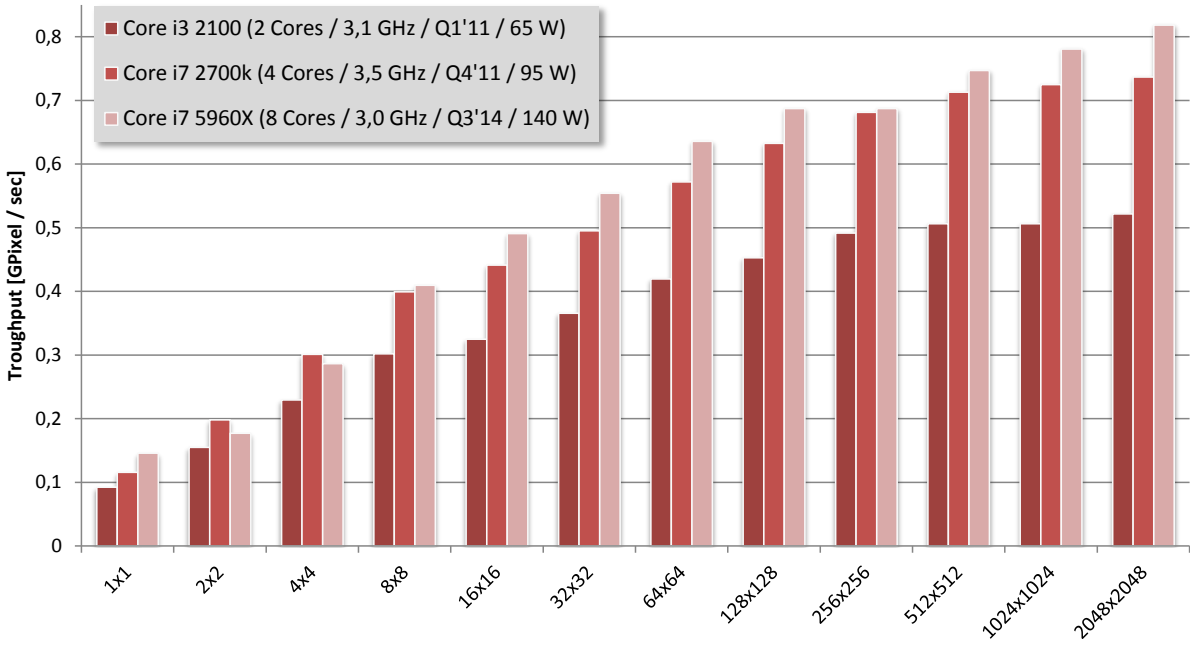
Der Fokus dieser Arbeit liegt auf der Evaluation der GPU-Fassung, sodass wir uns hier auf die Betrachtung zweier Datensätze, nämlich Spiral und Quads, beschränken, um eine grobe Einordnung des Ansatzes zu ermöglichen. Dabei interessiert vor allem, in welchem Maße die Anzahl der eingesetzten CPU-Kerne das Verhalten von Impl I CPU im Vergleich mit dem Contour-Tracing-Algorithmus beeinflusst.

Evaluieren wir zunächst den Contour-Tracing-Algorithmus für beide Datensätze auf unseren drei CPUs (Vergleich Abschnitt 17.2) mit 2, 4 und 8 Kernen, welche alle über SMT / Hyper-Threading verfügen. Die Ergebnisse sind in Abbildung 25.1 dargestellt. In allen Fällen ist keine Abhängigkeit von der Anzahl der CPU-Kerne erkennbar. Das ist bei einem nicht parallelen Algorithmus auch zu erwarten. Der Vierkerner schneidet sogar trotz seines Alters in einigen Messungen besser ab als der Achtkerner. Eine mögliche Erklärung (zum Teil) könnte dessen um 500 MHz höherer Takt sein. Der Zweikerner liefert deutlich weniger Throughput als der Vierkerner, welcher auf der gleichen Architektur basiert. Mögliche Gründe können der geringere Takt und das Fehlen von ITB, der dynamischen Übertaktung bei Verwendung nur eines Kerns, sein.

Bei Verarbeitung des Spiral-Datensatzes nimmt der Throughput mit zunehmender Problemgröße etwas ab. Generell ist der Throughput bei Verarbeitung dieses Datensatzes geringer als im Falle des Quads-Datensatzes. Bei letzterem steigt der Throughput mit zunehmender Größe der Quads deutlich an. Der Throughput im Falle der größeren Quads übertrifft denjenigen bei sehr kleinen um bis zu Faktor fünf.



(a) Spiral-Datensatz in verschiedenen Auflösungen



(b) 4096x4096 Quads-Datensatz, mit Quads in verschiedenen Größen (Angabe in Pixeln)

Abbildung 25.1.: Vergleich der Throughputs des sequentiellen Contour-Tracing-CCL-Algorithmus bei verschiedenen Daten unter Verwendung dreier Intel CPUs mit verschiedener Kernzahl. In Klammern hinter dem CPU Namen: Anzahl der Kerne, Basistakt, Einführungsdatum, TDP in Watt. SMT ist immer aktiviert und ITB bei den Core i7s. Der Core i3 unterstützt kein ITB.

Schauen wir uns nun noch einmal Impl I CPU bei Verarbeitung der selben Datensätze an. Die Ergebnisse sind in Abbildung 25.2 dargestellt. Hinweis: Das Verhalten bei 1x1 Pixeln großen Quads ist auf den entsprechenden Sonderfall zurückzuführen und die höchste Datenauflösung fehlt bei dem Core i3 aufgrund dessen zu geringer Hauptspeichermenge.

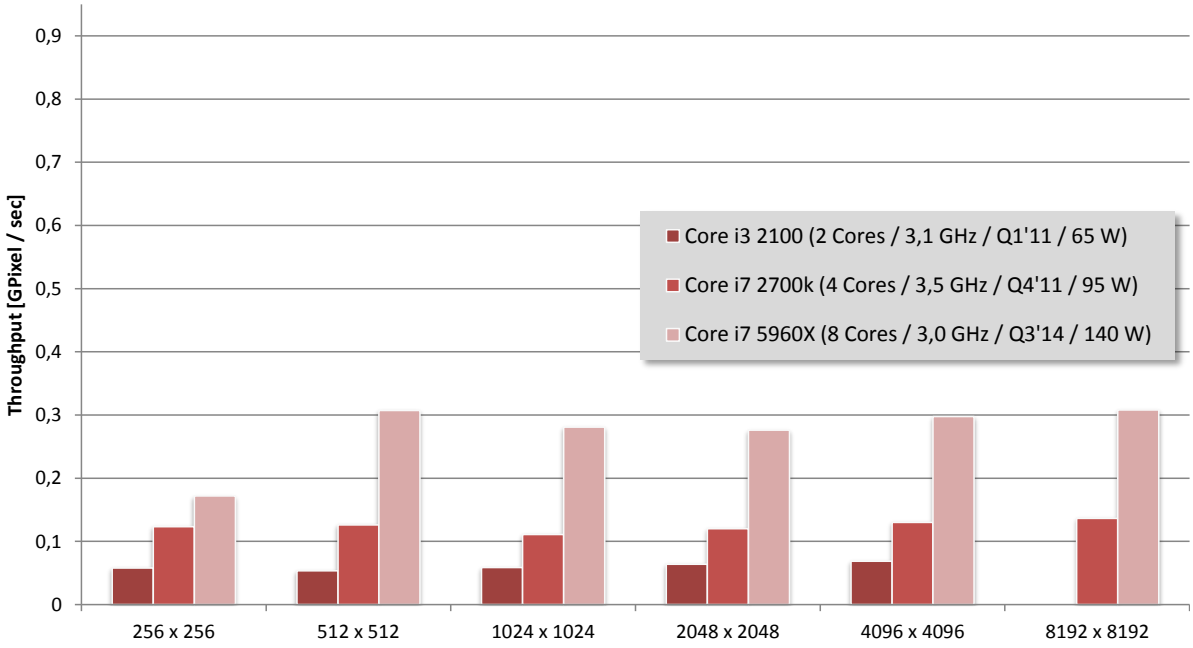
Sehr deutlich ist in Abbildung 25.2, im Gegensatz zum sequentiellen Algorithmus, die starke Abhängigkeit von der Anzahl der Kerne erkennbar. Der Achtkerner ist fast immer ca. doppelt so schnell wie der Vierkerner und der wiederum doppelt so schnell wie der Zweikerner. Generell sinkt der Throughput mit steigender Auflösung nicht. Der Unterschied zwischen Quads verschiedener Größe fällt mit ca. Faktor drei (ohne Sonderfall) etwas geringer aus als bei Chang.

Um die Throughput-Werte beider Ansätze besser direkt vergleichen zu können, stellen wir nun die Werte aus Abbildungen 25.1 und 25.2 gegenüber. Der Quads-Datensatz ist nun aufgeteilt für die drei CPUs in Abbildung 25.3 dargestellt.

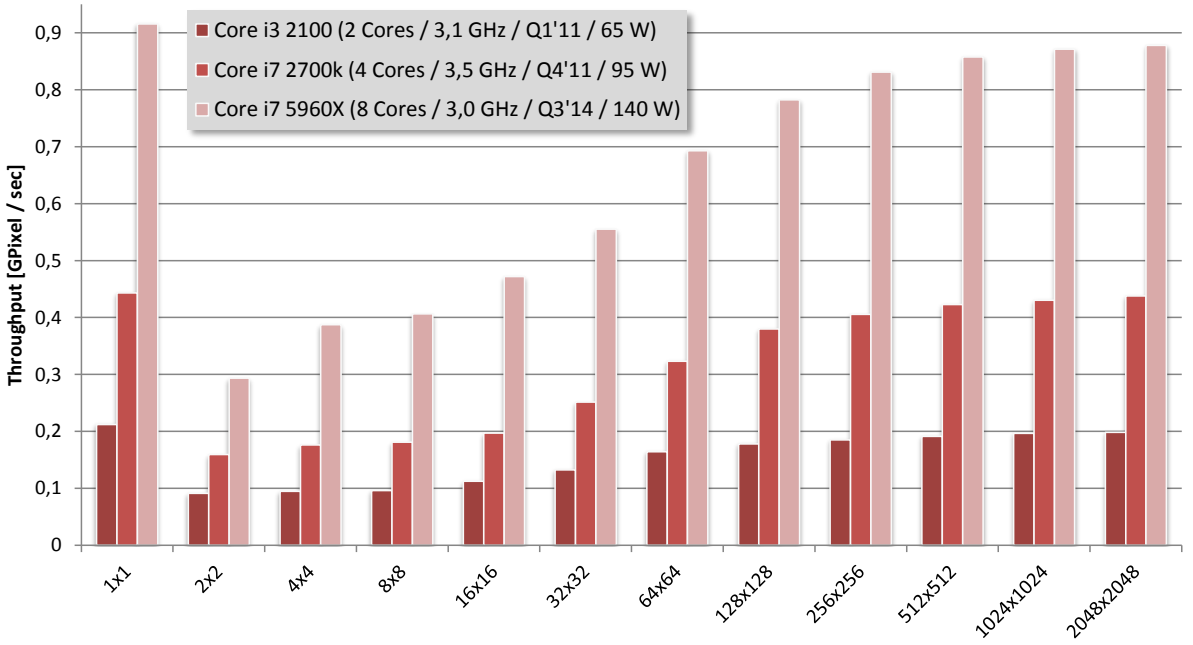
Hier wird deutlich, dass der sequentielle Contour-Tracing-Algorithmus in diesem vergleichsweise einfachen Datensatz im Falle von zwei und vier Kernen bessere Ergebnisse liefert als Impl I CPU. Der Abstand verringert sich bei Verwendung des Vierkerners im Vergleich zum Zweikerner. Werden die Algorithmen allerdings bei Ausführung durch den Achtkerner verglichen, liegen die Ergebnisse nahe beieinander, wobei Impl I CPU in fast allen Fällen leicht besser abschneidet.

Analog dazu zeigt Abbildung 25.4 die Throughputs für den Spiral-Datensatz. Dieser bereitet dem Contour-Tracing-Algorithmus deutlich größere Probleme als Impl I CPU. Hier liefert Impl I CPU auf dem Zweikerner ein etwas schlechteres Ergebnis, auf dem Vierkerner ein besseres und auf dem Achtkerner ein weitaus besseres. Im Extremfall, der 4096x4096 Spirale bei Ausführung auf dem Achtkerner, liefert Impl I CPU einen um Faktor fünf höheren Throughput.

Beim Vergleich die Ergebnisse für beide Datensätze kann festgestellt werden, dass der sequentielle Ansatz stärker von der Einfachheit des Quads-Datensatzes profitieren kann als Impl I CPU.

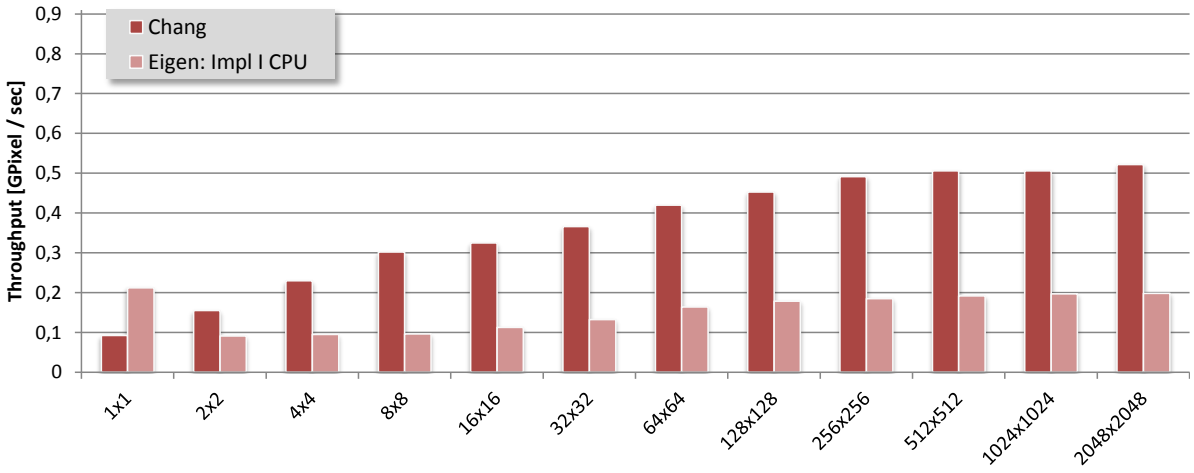


(a) Spiral-Datensatz in verschiedenen Auflösungen

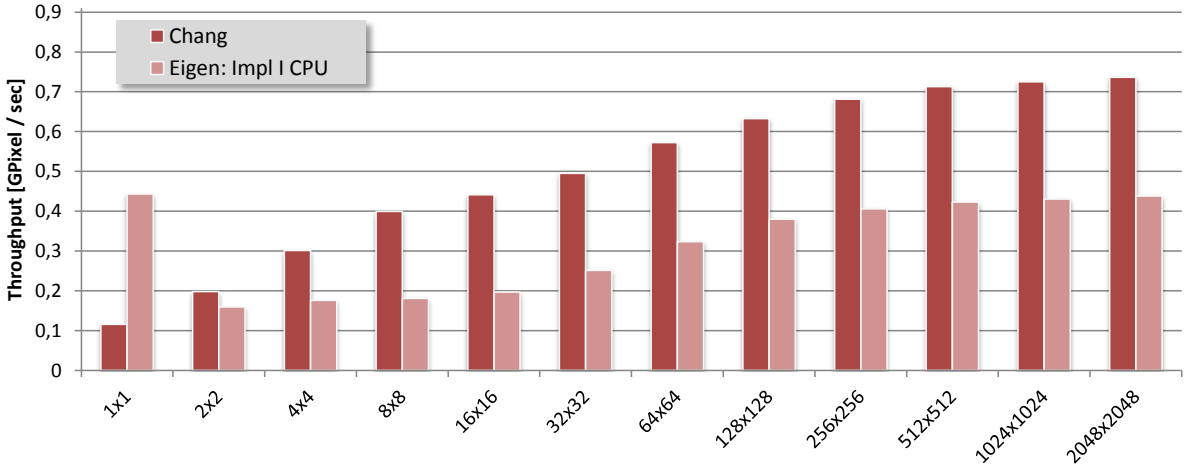


(b) 4096x4096 Quads-Datensatz, mit Quads in verschiedenen Größen (Angabe in Pixeln)

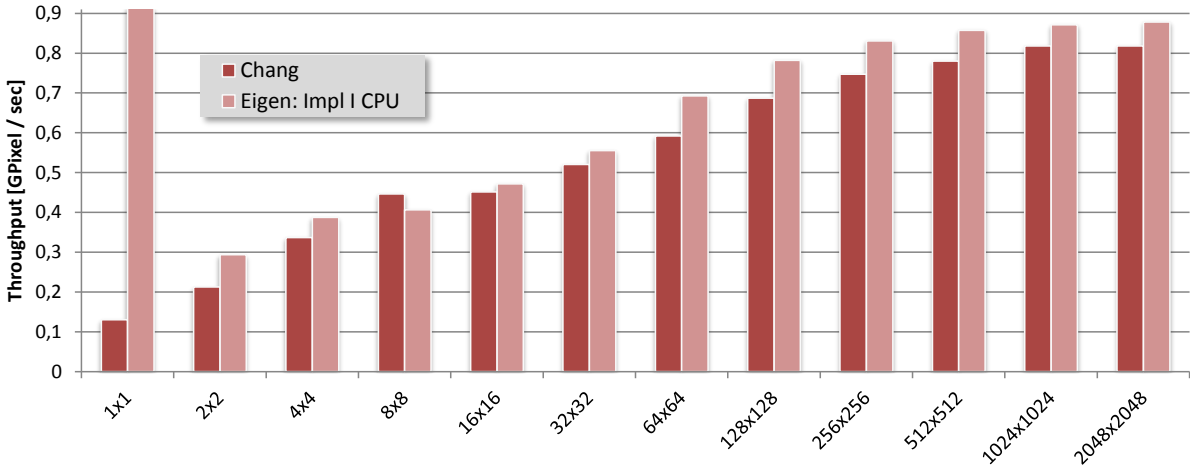
Abbildung 25.2.: Vergleich der Throughputs von Impl I CPU bei verschiedenen Daten unter Verwendung dreier Intel CPUs mit verschiedener Kernzahl. In Klammern hinter dem CPU Namen: Anzahl der Kerne, Basistakt, Einführungsdatum, TDP in Watt. SMT ist immer aktiviert und ITB bei den Core i7s. Der Core i3 unterstützt kein ITB.



(a) Intel Core i3 2100 (2 Cores / 3,1 GHz / Q1'11 / 65 W), SMT, Kein ITB

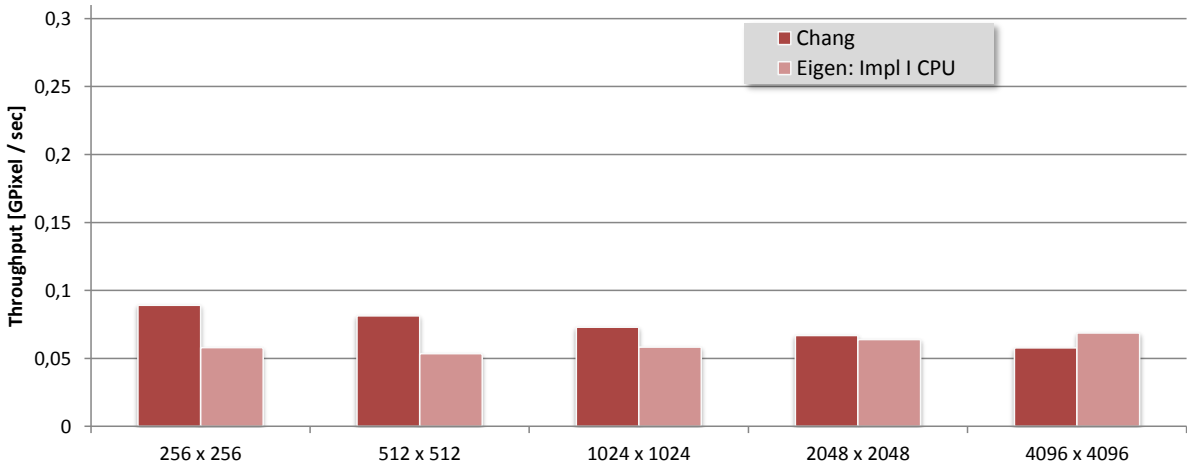


(b) Intel Core i7 2700k (4 Cores / 3,5 GHz / Q4'11 / 95 W), SMT, ITB

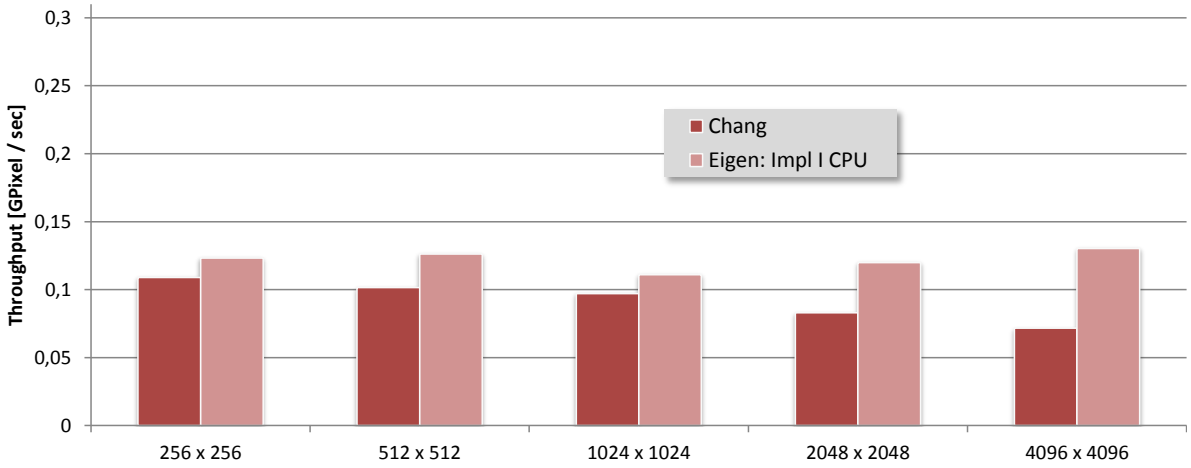


(c) Intel Core i7 5960X (8 Cores / 3,0 GHz / Q3'14 / 140 W), SMT, ITB

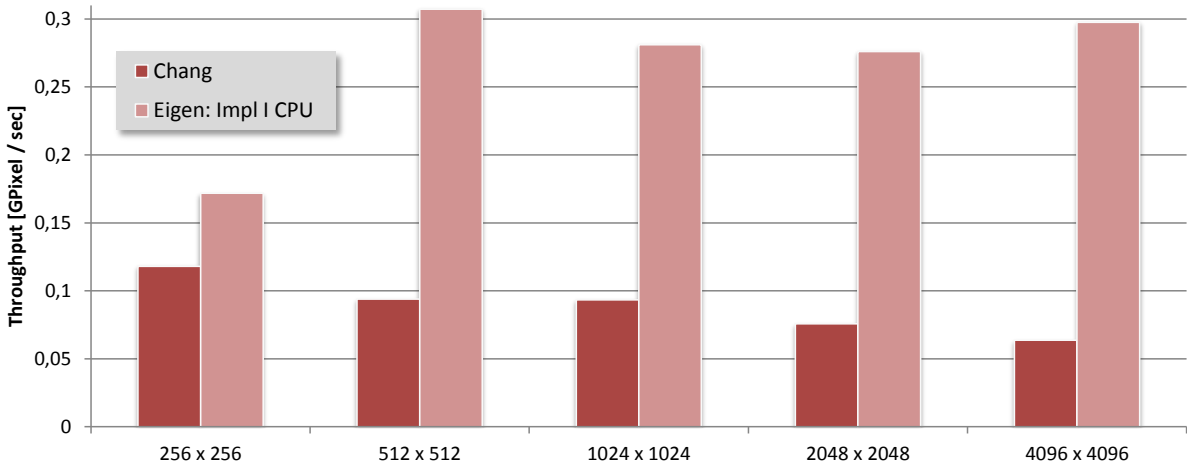
Abbildung 25.3.: Vergleich der Throughputs des Contour-Tracing-CCL-Algorithmus von Chang mit Impl I GPU für den Quads-Datensatz mit verschiedenen Quad-Größen. Es werden drei Intel CPUs mit verschiedener Kernzahl verwendet. In Klammern hinter dem CPU Namen: Anzahl der Kerne, Basistakt, Einführungsdatum, TDP in Watt



(a) Intel Core i3 2100 (2 Cores / 3,1 GHz / Q1'11 / 65 W), SMT, Kein ITB



(b) Intel Core i7 2700k (4 Cores / 3,5 GHz / Q4'11 / 95 W), SMT, ITB



(c) Intel Core i7 5960X (8 Cores / 3,0 GHz / Q3'14 / 140 W), SMT, ITB

Abbildung 25.4.: Vergleich der Throughputs des Contour-Tracing-CCL-Algorithmus von Chang mit Impl I GPU für den Spiral-Datensatz in verschiedenen Auflösungen. Es werden drei Intel CPUs mit verschiedener Kernzahl verwendet. In Klammern hinter dem CPU Namen: Anzahl der Kerne, Basistakt, Einführungsdatum, TDP in Watt

Kapitel 26.

Vergleich des eigenen GPU-Ansatzes mit Contour-Tracing auf CPU

In diesem Kapitel wird der eigene GPU-optimierte Ansatz Impl IV mit Changs [CCL04] Implementation ([CCL15]) verglichen.

Letzterer ist nicht in gleichem Maße optimiert wie der eigene Ansatz, der Vergleich ist somit nicht ganz fair. Er erscheint dennoch angebracht, weil dieser Ansatz dem eigenen Ansatz ähnlicher ist als solche, die auf Union-Find oder der Connection-List basieren. Schließlich liefert der eigene Ansatz, genau wie der von Chang, zusätzliche Informationen über die Konturen. Außerdem ist bereits im vorherigen Kapitel festgestellt worden, dass Changs Ansatz, ebenso wie der eigene Ansatz, von großen Connected-Components profitiert.

Wir wählen für den Vergleich die derzeit leistungsstärkste Desktop CPU, den Intel Core i7 5960X, für Changs Contour-Tracing-Ansatz und die leistungsstärkste GPU unserer Auswahl, die Nvidia Geforce GTX 980, für Impl IV. Unter Berücksichtigung der Ergebnisse der Kapitel 24 und 25 ist zu erwarten, dass diese Wahl zu besonders deutlichen Unterschieden führt.

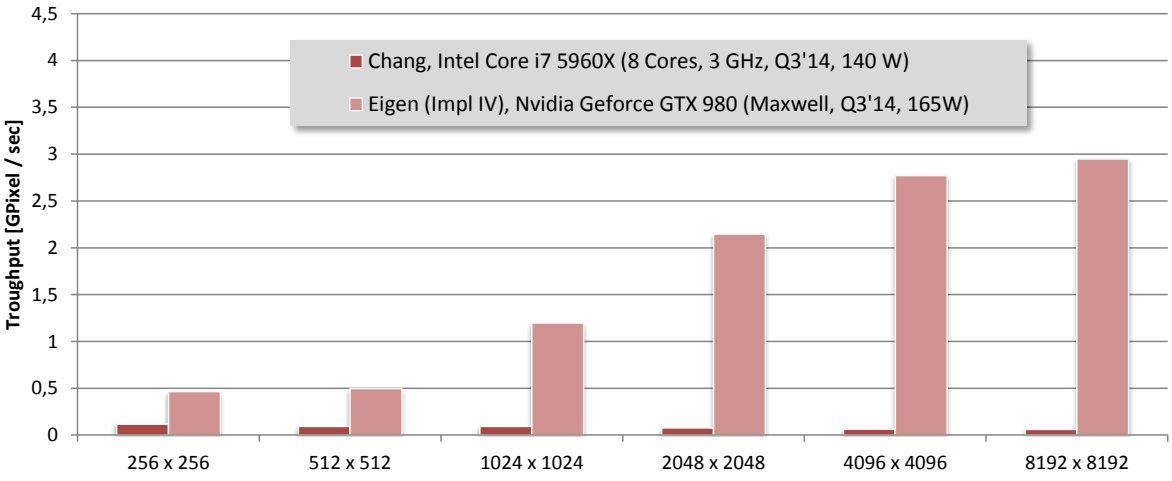
Wir vergleichen nun die gemessenen Werte miteinander in Abbildung 26.1. Zunächst fällt der erheblich höhere Throughput des eigenen Ansatzes in allen Messungen auf. Das war zu erwarten, weil bereits vorherige GPU basierte Ansätze (etwa [OS11]) deutliche Speedups im Vergleich mit CPU basierten Implementationen erreicht haben.

Für den Spiral-Datensatz ermitteln wir immer größere Throughput Unterschiede zwischen beiden Ansätzen, je höher die zu verarbeitende Datenauflösung ist. So ist Impl IV in der 256 x 256 Auflösung lediglich um Faktor vier und in der 8192 x 8192 Auflösung um Faktor 49 schneller. Dieses Verhalten ist mit den Beobachtungen aus den Kapiteln 24 und 25 nicht überraschend.

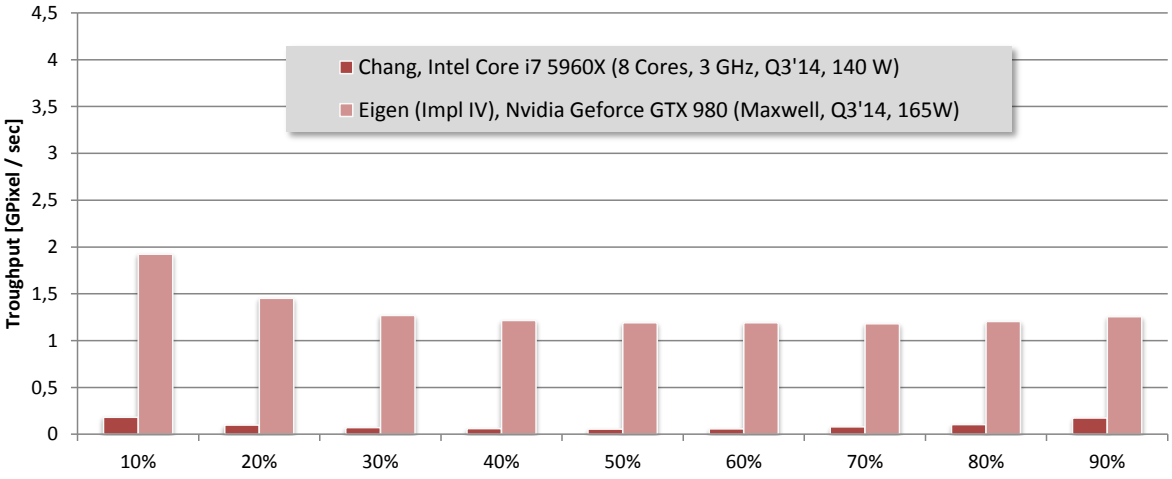
Ebenso wie bei Impl IV stellt auch im Falle von Changs Implementation der Noise-Datensatz mit 50 Prozent Einsen die größte Anforderung von allen getesteten Datensätzen dar. Auch profitiert Changs Ansatz von einem Übergewicht der Einsen oder Nullen, allerdings im Vergleich deutlich stärker als Impl IV. Das ändert aber nichts daran, dass Impl IV bei Verarbeitung des Noise-Datensatzes um minimal Faktor sieben (90 Prozent Einsen) und maximal Faktor 22 (50 Prozent Einsen) schneller ist.

Den kleinsten Speedup von allen getesteten Datensätzen in der 4096 x 4096 Auflösung

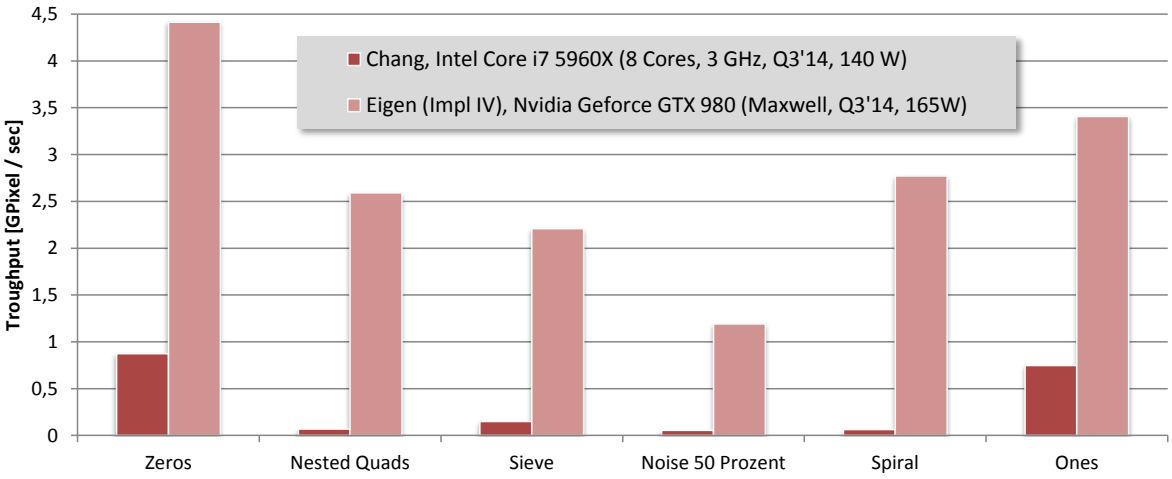
ermitteln wir als Faktor 4,5 für den Ones-Datensatz. Wie bereits im Kapitel 25 festgestellt, profitiert Changs Implementation von Connected-Components, deren Verhältnis von Rand zu Innenfläche günstig ist. Und dieser Effekt ist weitaus deutlicher als bei dem eigenen Ansatz. Nehmen wir, um das zu belegen, als Gegenstück zum Ones-Datensatz einen, der aus maximal vielen Randsegmenten besteht (z.B. die Spirale). Im Vergleich mit diesem Datensatz erreicht Chang in der 4096 x 4096 Auflösung mit dem Ones-Datensatz einen um Faktor 12 höheren Throughput. Im Falle des eigenen Ansatzes beträgt der selbe Unterschied lediglich Faktor 1,2.



(a) Spiral-Datensatz in verschiedenen Datenauflösungen



(b) Noise mit verschiedenem Anteil von Einsen. Datenauflösung 4096 x 4096



(c) Verschiedene Datensätze der Auflösung 4096 x 4096

Abbildung 26.1.: Vergleich von Impl IV (GPU) mit Changs Contour-Tracing-Ansatz (CPU)

Teil IV.

Epilog

Kapitel 27.

Diskussion der Ergebnisse

Im Rahmen der vorliegenden Arbeit sind verschiedene parallele und auf vorheriger Konturierung basierende Connected-Component-Labeling-Algorithmen vorgeschlagen und hinsichtlich ihrer Praxistauglichkeit evaluiert worden. Ein Ausgangspunkt der Arbeit und primärer Vergleichspartner auf GPUs ist der Cuda basierte Ansatz von Stava [OS11], welcher in der Praxis schnell ist. Anders als dieser und vergleichbare Ansätze liefern die Ansätze der vorliegenden Arbeit zusätzliche Informationen über die Objektkonturen, welche z.B. im Bereich Bildverarbeitung genutzt werden können.

Diese Konturinformationen liefert ebenfalls der in der Praxis schnelle und optimale sequentielle Contour-Tracing-basierte Ansatz von Chang[CCL04]. Im Vergleich mit diesem ist vor allem evaluiert worden, welche Speedups durch die Parallelisierung erreichbar sind.

Es ist insgesamt gelungen, Ansätze zu finden, welche in der Praxis gut skalieren, und zwar sowohl mit der Datenauflösung als auch mit der Device-Parallelität, und zusätzlich die hilfreichen Konturinformationen liefern. Dadurch konnten, sowohl auf GPUs als auch auf (Mehrkern)-CPUs, die positiven Eigenschaften der Ansätze von Stava und Chang vereinigt werden.

Diese Ergebnisse werden detaillierter in den folgenden Abschnitten besprochen.

Nutzen der theoretischen Eigenschaften in der Praxis

Im Rahmen der vorliegenden Arbeit ist untersucht worden, ob speziell durch die theoretischen Eigenschaften, welche mit der vereinfachten Topologie möglich werden, in der Praxis ein besseres Laufzeitverhalten erreichbar ist. Die Algorithmen können, je nach Art der verwendeten Contour-Labeling-Technik, cost-optimal oder optimal sein. Beides wird möglich durch Algorithmen speziell für zyklische gerichtete Linked-Lists.

Für die Implementation auf einer CPU ist der cost-optimale Ansatz gewählt worden, welcher Impl I CPU bezeichnet wurde. Dieser führt in Abhängigkeit von der Datenauflösung eine lineare Anzahl von Operationen aus und minimiert zusätzlich den konstanten Vorfaktor. Die Parallelität ist für alle aktuellen CPUs ausreichend. Außerdem bietet der Ansatz auf einer CPU günstige Speicherzugriffsmuster. Der optimale Ansatz wird hier nicht verwendet, da dieser lediglich eine erhöhte Parallelität bietet, welche auf einer CPU

nicht nutzbar ist, da auch so schon alle Kerne ausgelastet werden können.

Experimentell ist eine lineare Abhängigkeit der Laufzeit von der Datenauflösung ermittelt worden. Im Vergleich mit dem etablierten Contour-Tracing-Ansatz von Chang skaliert Impl I CPU bei Verarbeitung des zunehmend komplexer werdenden Spiral-Datensatzes besser mit der Problemgröße.

Die Verwendung von CPUs verschiedener Kernzahl bei ähnlicher Taktung erhöht den experimentell ermittelten Throughput etwa linear mit der Anzahl der eingesetzten Kerne. Im Gegensatz dazu ist bei Chang natürlich keine Abhängigkeit von der eingesetzten Kernzahl erkennbar.

Als Resultat ist der eigene Ansatz bei Verwendung einer CPU mit acht Kernen im Falle aller getesteten Datensätze zumindest etwas schneller.

Insgesamt sind damit die Skalierung, sowohl in Abhängigkeit der Datenauflösung, als auch diejenige in Abhängigkeit der Kernzahl, zufriedenstellend. Damit kann als Ergebnis festgehalten werden, dass die Eigenschaft cost-optimal ideal ist für CPUs, da diese nicht massiv parallel arbeiten.

Dagegen verhält sich der cost-optimale Ansatz Impl I GPU, bei experimenteller Evaluation auf einer GPU, ungünstig. So muss ein zu hoher Anteil der Berechnungen in einer (für eine GPU) nicht ausreichenden Parallelität ausgeführt werden. Außerdem können hier teilweise ungünstige Speicherzugriffsmuster nicht vermieden werden.

Zwar steigt der Throughput stark mit der Datenauflösung an. Aber das ist lediglich der ausgesprochen ungünstigen GPU-Auslastung in geringen Auflösungen geschuldet. Außerdem fällt die Skalierung mit zunehmender GPU-Parallelität (SP-Anzahl) extrem schlecht aus.

Als Ergebnis kann festgehalten werden, dass im Falle massiv paralleler Devices, wie GPUs, die Minimierung der Berechnungen nicht wünschenswert erscheint, wenn dies die Parallelität zu sehr einschränkt.

Motiviert durch die Beobachtungen des cost-optimalen Ansatzes Impl I GPU ist der optimale Ansatz Impl III untersucht worden, welcher eine höhere Parallelität ermöglicht. Dieser kann bereits in geringen Datenauflösungen eine GPU gut auslasten und ist deshalb in solchen Fällen schneller als Impl I GPU. Auch skaliert der Throughput wesentlich besser mit zunehmender GPU Parallelität.

In hohen Auflösungen dagegen fällt der Throughput von Impl III deutlich hinter denjenigen von Impl I GPU zurück.

Als Ursache dafür wurde der hohe Aufwand, sowohl zur Bestimmung unabhängiger Elemente als auch zum Ausdünnen der Daten, identifiziert.

Begründet durch die wenig befriedigenden Ergebnisse des cost-optimalen und des optimalen Ansatzes wurde ein weiterer Ansatz untersucht, welcher zugunsten erhöhter GPU Optimierung auf die positiven theoretischen Eigenschaften verzichtet. Dieser wurde Impl IV genannt. Er verwendet ein rekursives Tile-Schema, mit welchem sich die Problemgröße am Anfang schnell um Faktor 256 verkleinern lässt.

Es hat sich gezeigt, dass die gemessene Laufzeit von Impl IV trotz superlinear mit der

Datenauflösung ansteigender Anzahl von Operationen lediglich sublinear ansteigt. Das gilt für alle Datenauflösungen, die noch in den Speicher der verwendeten GPUs passen. In vergleichbarer Weise lässt sich das auch für die Cuda Implementation von Stava, der ein ähnliches rekursives Tile-Schema verwendet, beobachten. Die experimentellen Ergebnisse dieser Arbeit deuten somit auf ein ähnliches Skalierungsverhalten beider Ansätze in Abhängigkeit von der Problemgröße hin.

Hinsichtlich der Skalierung in Abhängigkeit von der Device-Leistung, damit auch Parallelität (wenn auch kaum isolierbar), schneidet der eigene Ansatz besser ab. Mit zunehmender GPU-Leistung setzt sich Impl IV in allen Testszenarien zunehmend von Stava ab.

Außerdem ist Impl IV im Vergleich mit allen anderen, in dieser Arbeit vorgeschlagenen, Ansätzen in jeder Auflösung und in jedem Datensatz überlegen.

Es kann folglich als Ergebnis festgehalten werden, dass nicht einmal Cost-Optimalität benötigt wird, um ein gutes Skalierungsverhalten mit der Problemgröße einerseits und günstiges allgemeines Laufzeitverhalten andererseits zu erreichen. Alle theoretischen Eigenschaften, welche durch den konturbasierten Ansatz ermöglicht werden, haben damit im Falle einer GPU zu keinem erkennbaren Nutzen geführt. Stattdessen ist der Optimierung für die Funktionsweise einer GPU eine höhere Bedeutung beizumessen.

Nutzen der Verschiebung des Fokus auf Konturen in der Praxis

Stava identifiziert experimentell großflächige Connected-Components als besonders schwierig für deren Ansatz. Dagegen werden bei konturbasierten Algorithmen die aufwändigeren Operationen allein auf die Randsegmente angewandt. Somit können in solchen Fällen Vorteile erhofft werden. Ferner ist angenommen worden, dass bei der Mehrheit aller Objekte die Anzahl der Randpixel klein ist gegenüber der Anzahl aller enthaltenen Pixel. Im Rahmen der vorliegenden Arbeit sind die vorgeschlagenen Implementationen speziell unter diesem Gesichtspunkt untersucht und mit dem Ansatz von Stava verglichen worden.

Tatsächlich sind in allen Experimenten für alle eigenen Ansätze mit zunehmendem Fläche / Rand Verhältnis der Objekte in den Daten zunehmende Throughputs ermittelt worden. Im Extremfall des Ones-Datensatzes (alle Pixel mit Eins klassifiziert) wird jeweils fast der Throughput des Best-Case (leerer Datensatz) erreicht. Damit ist das Verhalten gerade invers zu dem von Stava dokumentierten. Im Falle des Ones-Datensatzes wird somit auch der Vorteil von Impl IV im Vergleich mit Stava maximal (Faktor 3,6 schneller, Nvidia GeForce GTX 980).

Allerdings ist Impl IV auch im Falle des Spiral-Datensatzes deutlich schneller, deren enthaltene Figur ausschließlich aus Konturen besteht (Faktor 2,6 schneller, Nvidia GeForce GTX 980). Somit sind die Auswirkungen eines, für den eigenen Ansatz, einfachen Datensatzes im Falle von Impl IV recht gering.

Andere eigene untersuchte Ansätze (Impl II und III), welche die Daten vor der Konturverarbeitung ausdünnen, profitieren weitaus mehr als Impl IV von Datensätzen mit

geringem Konturanteil. Allerdings sind sie trotzdem im Extremfall (Ones-Datensatz) noch langsamer als Impl IV.

Anders als Stava profitiert der Contour-Tracing-Ansatz von Chang von Datensätzen mit geringem Konturanteil. Beim experimentellen Vergleich von Chang mit der eigenen Impl I CPU auf verschiedenen CPUs ist festgestellt worden, dass Chang mehr von Datensätzen mit geringem Konturanteil profitiert.

Resultierende Beurteilung des Verhaltens von Impl IV

Die im Vergleich aller eigenen Ansätze schnellste Variante, Impl IV, profitiert wenig von denjenigen Gründen, welche den konturbasierten Ansatz ursprünglich motiviert haben. Ein Nutzen der einfachen Topologie zyklischer Linked-Lists ist nicht erkennbar. So ist Impl IV noch nicht einmal work-optimal.

Der Fokus auf die Verarbeitung von Konturen zeigt zwar auf den ersten Blick Vorteile bei Datensätzen mit großem Fläche / Rand Verhältnis. Fraglich ist jedoch, ob dieser ausreicht, den konturbasierten Ansatz zu rechtfertigen.

Im Rahmen dieser Arbeit ist bereits beobachtet worden, dass die mit der Verarbeitung der Kontursegmente zusammenhängenden Operationen für einen Großteil der Laufzeit von Impl IV verantwortlich sind.

Definieren wir nun ein Maß für den Nutzen des Fokus auf Konturverarbeitung. Dieses entspreche dem Throughput-Verhältnis aus einem konturarmen (Ones) und einem konturreichen (Spirale) Datensatz in der Auflösung 4096 x 4096 Pixel bei Verarbeitung durch die Nvidia GeForce GTX 980. Der Wert für Impl IV beträgt 1,23. Das ist ein Nutzen, den die eigenen Ansätze haben und Stava nicht. Hier beträgt der entsprechende Wert 0,88.

Aber um das zu erreichen, wird die Problemgröße beim Übergang von Pixeln zu Kontursegmenten vervierfacht. Der Nutzen des konturbasierten Ansatzes, zumindest in dieser Form, ist somit möglicherweise nicht ideal. Diese Beobachtung wird zur Motivation weiterer Connected-Component-Labeling-Algorithmen im nächsten Kapitel (offene Problemstellungen) wieder aufgegriffen.

Resultierende Beurteilung des Verhaltens von Impl I CPU

Der experimentelle Vergleich von Impl I CPU mit dem sequentiellen Contour-Tracing von Chang zeigt vor allem, dass Parallelisierung erforderlich ist, um auf modernen CPUs deutlich bessere Ergebnisse zu erhalten als auf Vorgängermodellen. So ist der sequentielle Algorithmus auf Intels aktuellem Desktop-Topmodell, einem Achtkerner, kaum schneller oder sogar langsamer als auf Unterklasse- und Mittelklasse-CPUs desselben Herstellers aus dem Jahre 2011 mit zwei bzw. vier Kernen.

Dagegen ist die Ausführung von Impl I CPU auf dem Achtkerner zweimal bzw. viermal so schnell wie auf den Modellen mit vier bzw. zwei Kernen. Das genügt, um im Falle des Spiral-Datensatzes bei Ausführung auf dem Achtkerner ein deutlich besseres Ergebnis zu erreichen.

Einem deutlicheren Vorteil des parallelen Ansatzes auf einer CPU steht somit die geringe Kernzahl aktueller CPUs im Wege, auch wenn sich zumindest der Achtkerner teilweise klar absetzen kann.

Speedups im Vergleich mit Stava und Chang

Fassen wir zunächst die Ergebnisse der experimentell für aktuelle Nvidia GPUs ermittelten Throughputs von Stavas Ansatz im Vergleich mit der eigenen Impl IV zusammen. Im direkten Vergleich auf jeweils identischen GPUs liefert der eigene Ansatz in der Mehrheit der evaluierten Kombinationen aus Datensätzen und GPUs Speedups um ca. Faktor zwei. Das schließt auch Datensätze mit hohem Konturanteil ein. Beim Vergleich auf einer aktuellen Nvidia GTX 980 ist Impl IV in allen evaluierten Datensätzen schneller. Der größte Speedup tritt, wie erwartet, bei maximal großen Connected-Components ein. Hier ist Impl IV bei Verwendung der GTX 980 um knapp Faktor vier schneller. Eine Ausnahme bildet der Noise-Datensatz, welcher während des Entwicklungsprozesses der Implementationen dieser Arbeit noch nicht berücksichtigt wurde. Hier ist Stavas Ansatz manchmal schneller. Vor allem wenn die schwächste GPU (GTX 480) eingesetzt wird und der Anteil von Einsen gering ist. Beim Vergleich unter Verwendung der leistungsfähigsten GPU (GTX 980) ist Impl IV auch hier mit jeder Noise-Konfiguration schneller.

Außerdem wurde die eigene Impl I CPU mit dem Ansatz von Chang auf CPUs mit verschiedener Kernzahl verglichen. Im Falle eines Zweikerners ist Changs Ansatz fast immer schneller. Bei Evaluation auf einer CPU mit acht Kernen ist Impl I CPU in keinem evaluierten Datensatz langsamer als der Contour-Tracing-Algorithmus von Chang und in manchen Fällen um bis zu Faktor fünf schneller.

Zuletzt wurde ein plattformübergreifender Vergleich mit dem sequentiellen Ansatz von Chang durchgeführt, da dieser ebenfalls Konturinformationen liefert. Dazu wurden eine aktuelle GPU (für Impl IV) und eine aktuelle CPU (für Chang) vergleichbarer Verlustleistung und Preisklasse ausgewählt. Hier ist die eigene Implementation immer, und zwar meistens um eine Größenordnung, schneller. Im Extremfall ist der eigene Ansatz um Faktor 49 schneller.

Kapitel 28.

Offene Problemstellungen

Abschließend werden einige Anregungen für die Weiterentwicklung von parallelen Connected-Component-Labeling-Algorithmen für CPUs und GPUs gegeben, welche durch Beobachtungen in der vorliegenden Arbeit motiviert sind.

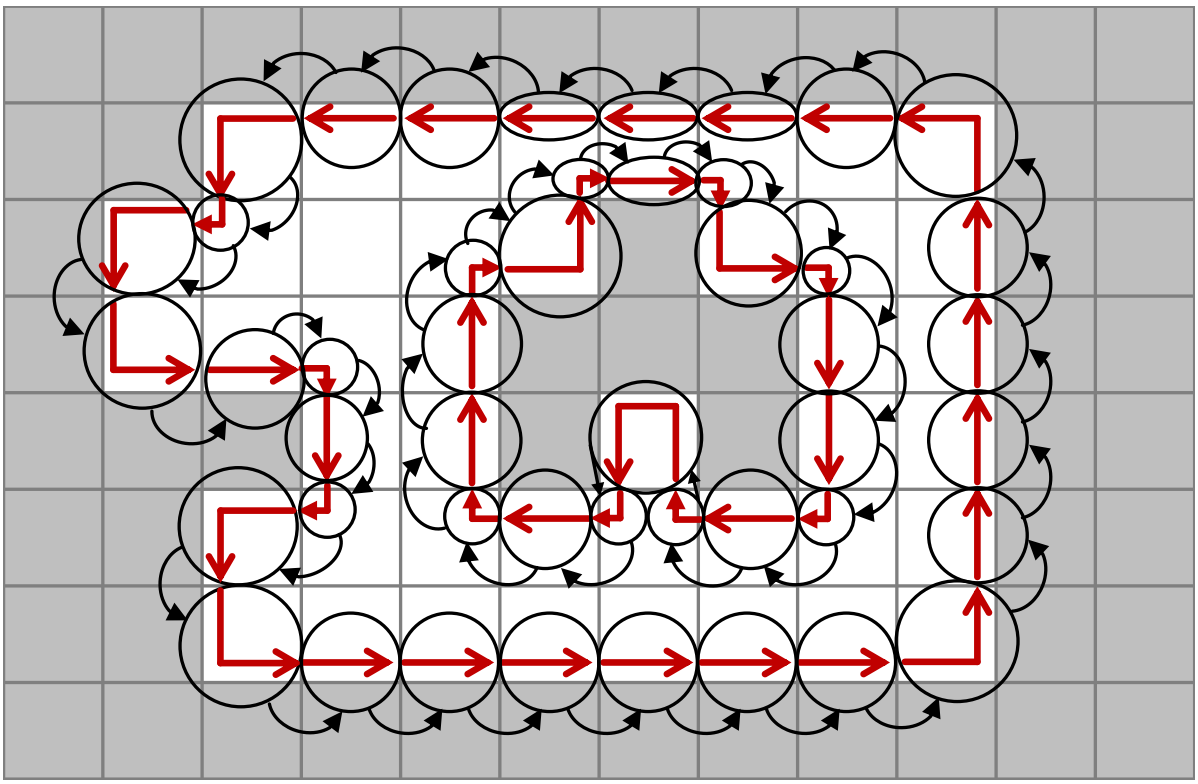
Zu nennen sind zunächst weitere Aspekte der bereits vorgestellten Techniken. Von der, auch so bereits recht zufriedenstellenden, CPU-Fassung Impl I CPU könnte eine CPU-optimierte Variante erstellt werden. Ferner bleibt offen, warum der Ansatz von Chang [CCL04] im Vergleich dazu stärker von Datensätzen mit geringem Konturanteil profitiert.

Abgesehen davon kann auch die GPU-Fassung Impl IV weiter optimiert werden. Zu nennen sind ein mehr als zweistufiges Tile-Schema, frühzeitiges Pointer-Jump-Beenden auch in den anderen Tile-Generationen und die Behebung des Register-Spillings im Falle der Fermi Architektur. Zusätzlich ermöglicht die Verwendung von OpenCL auch eine Evaluation auf AMD GPUs. So ließe sich erstmals eine Referenz für zukünftige Arbeiten erstellen, da alle uns bekannten Veröffentlichungen die Nvidia exklusive API Cuda verwenden.

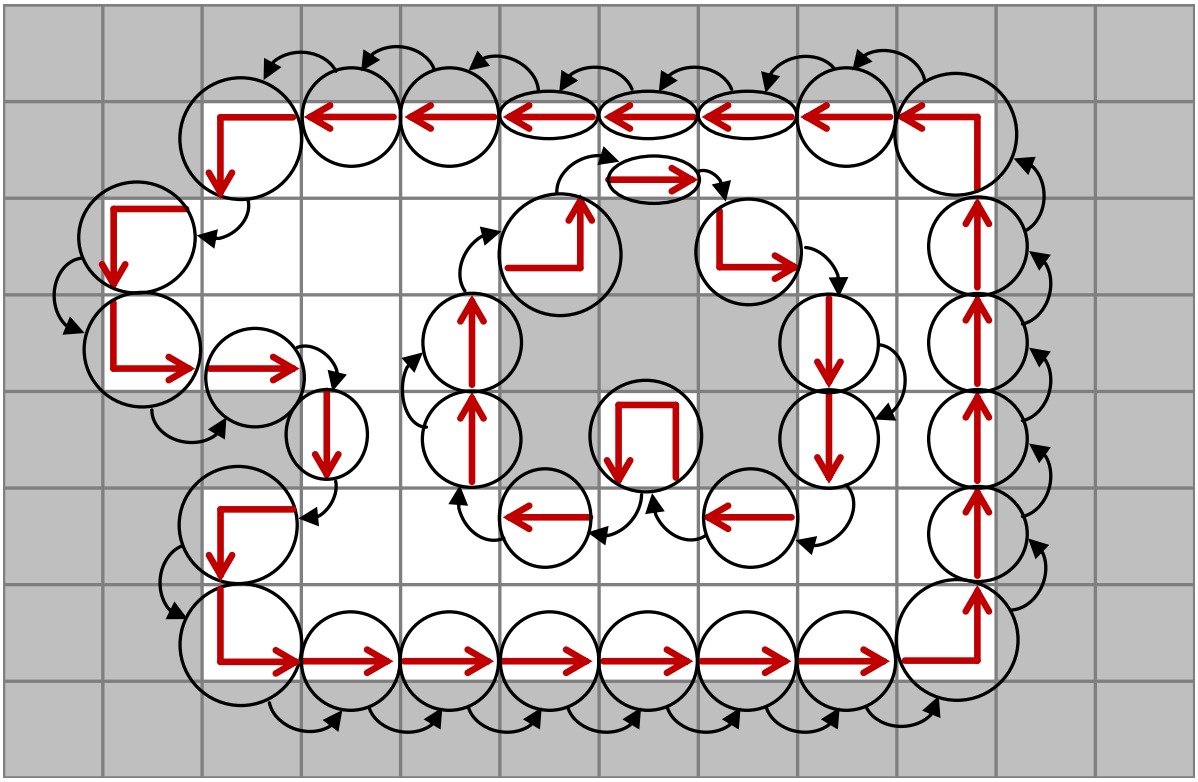
Reduziertes Kontursegmentschema

Vor allem sollen an dieser Stelle neue Ansätze vorgeschlagen werden. Als primäres Problem der eigenen Ansätze hat sich die Vergrößerung der Datenauflösung um Faktor vier bei Überführung der Pixel in die Kontursegmentdaten herausgestellt. Als mögliche Verbesserung hinsichtlich dieses Gesichtspunktes ist ein alternatives Kontursegmentschema denkbar. Dieses verzichtet auf alle Segmenttypen, welche wir als Verbindungskonturstücke bezeichnet haben, und erlaubt stattdessen die Verbindung gemäß Achter-Nachbarschaft. Die übrigen Segmenttypen werden in gleicher Weise beibehalten wie bisher aber können nun direkt mit dem nächsten nicht Verbindungskonturstück verbunden werden. Das oben beschriebene Schema für Kontursegmente ist in Abbildung 28.1 im Vergleich mit dem in dieser Arbeit verwendeten Schema zu sehen. Auf diese Weise sind weiterhin alle Ränder aller Connected-Components vollständig durch Kontursegmente in Form von zyklischen Linked-Lists repräsentiert. Aber im Gegensatz zu dem bisherigen Schema können niemals mehr als zwei Kontursegmente in einem Pixel existieren.

Infolgedessen halbiert sich die zu verarbeitende Problemgröße im Vergleich mit den bisherigen eigenen Ansätzen von $4 \cdot N$ auf $2 \cdot N$. Das reduzierte Kontursegmentschema ermöglicht außerdem die Verarbeitung größerer Tiles (z.B. 64 x 64 Pixel für G1) in Impl



(a) In dieser Arbeit verwendetes Kontursegmentschema



(b) Reduziertes Kontursegmentschema

Abbildung 28.1.: Vergleich des reduzierten Kontursegmentschemas mit dem in dieser Arbeit verwendeten. Rote Pfeile stellen Kontursegmente gemäß bisherigem SRC- und DST-Typ dar. Schwarze Pfeile geben die anfängliche Verlinkung der Knoten (schwarze Kringel) an.

IV. In dieser Arbeit wurden bereits die zu erwartenden positiven Auswirkungen größerer Tiles dargelegt.

Auch im Vergleich mit dem von Cypher [CSS89] vorgeschlagenen Kontursegmentschema existieren zwei Vorteile. Erstens bleibt die Möglichkeit der Verarbeitung von nicht Binärdaten erhalten. Und zweitens erzeugt Cypher etwas mehr als $2 \cdot N$ Segmente, wenn N die Pixelzahl ist. Da das auch lokal, d.h. in Tiles, gilt, ist ein ganzzahliges Aufteilen des Shared-Memory bei *schönen* Tile-Größen nicht mehr möglich. Die negativen Auswirkungen eines solchen Szenarios sind in dieser Arbeit dokumentiert.

GPU optimierter Ansatz II - Fokus auf konturarme Daten

Generell sei hiermit vorgeschlagen, die weitere Untersuchung GPU basierter Ansätze in zwei alternative Zweige aufzuspalten. Beide sind motiviert durch die Beobachtung, dass die Problemgröße um Faktor vier (oder zwei, mit obigem Ansatz) vergrößert wird, aber Impl IV nicht in vergleichbarem Maße (also Faktor vier) von einem fast konturlosen Datensatz im Vergleich zu einem sehr konturreichen an Throughput gewinnt.

Der erste Vorschlag für einen neuen Ansatz (A1) stellt eine Kombination von Ideen aus Impl II / III und Impl IV dar. Es konnte in dieser Arbeit beobachtet werden, dass Impl II und III durch das Überführen der Kontursegmente in eine dichtbesetzte Datenstruktur vor Ausführung des Konturlabeling-Algorithmus deutlich mehr als Impl IV von konturarmen Datensätzen profitieren.

Der neue Ansatz kann weiterhin ein rekursives Tile-Schema nutzen und darin lokal die Pointer-Jumps auf Konturliniensegmente anwenden. Aber zuvor können die Daten lokal für die Tiles verdichtet werden. Dafür genügt dann auch ein Segmented-Scan, welcher in solchen Fällen um eine Größenordnung schneller ist als ein *globaler* Scan [DGS+08]. Idealerweise wird für A1 ein dynamisches und irreguläres Tile-Grid verwendet, dessen Konfiguration von den Ergebnissen des Segmented-Scans abhängt. So kann verschiedenen Segmentdichten innerhalb eines Datensatzes Rechnung getragen werden. Außerdem ist auf diese Weise die maximale Tile-Größe nicht mehr für alle Tiles durch den Worst-Case der Segmentzahl limitiert.

Insgesamt könnte A1 die GPU-Optimierung von Impl IV erhalten und trotzdem in ähnlicher Weise wie Impl II/III von kontursegmentarmen Datensätzen profitieren. Außerdem kann A1 das weiter oben vorgeschlagene reduzierte Kontursegmentschema verwenden.

GPU-optimierter Ansatz III - Nicht konturbasiert

Es konnte in dieser Arbeit beobachtet werden, dass die simplen Pointer-Jumps in Kombination mit dem Tile-basierten Schema zu guten Ergebnissen führen. Insbesondere fällt der superlineare Aufwand, sowohl lokal per Tile als auch global, kaum ins Gewicht.

Dadurch ist der Vorschlag für einen weiteren Ansatz (A2) motiviert, welcher nicht auf Konturen basiert. Stattdessen werden zunächst, wie z.B. bei Stava der Fall, Equivalence-Trees aufgebaut. Anschließend sollen die Label mit der Pointer-Jump-Technik und

nicht mit dem Union-Find-Algorithmus gefunden werden. Das bewährte rekursive Tile-basierte Schema wird für A2 beibehalten.

Im Vergleich mit Stava sind zwei Vorteile denkbar, nämlich erstens das vorhersehbarere Verhalten. So ist das von Stava beobachtete Verhalten des *Hängenbleibens* in lokalen Minima, welche durch die parallele Ausführung entstehen können, nicht möglich. Zweitens muss sich deren Ansatz auf die Verwendung von Atomic-Functions verlassen, um die dynamisch auftretenden Konflikte zu lösen. Dies ist im Falle der Pointer-Jump-Technik nicht erforderlich.

Verglichen mit der eigenen Impl IV können vor allem Vorteile aus der Verringerung der Anzahl der zu verarbeitenden Elemente erhofft werden. Schließlich sind höchstens N statt zuvor $4 \cdot N$ Elemente zu verarbeiten. Allerdings ist der Unterschied mit dem neu vorgeschlagenen Kontursegmentschema (max. $2 \cdot N$ Elemente) bereits geringer. Zusätzlich kann der Aufbau des Equivalence-Trees als ungleich einfacher eingeschätzt werden als das bisherige Extrahieren der Liniensegmente, wofür eine sehr hohe Anzahl datenabhängiger Fallunterscheidungen notwendig ist. Zuletzt entfällt der Fill-Contours-Schritt ersatzlos.

Von Datensätzen mit großem Fläche / Rand Verhältnis kann A2, im Gegensatz zu allen anderen eigenen vorgeschlagenen Ansätzen, nicht mehr profitieren.

Zusammengefasst könnte A1 den Vorteil bei konturarmen Daten gegenüber Impl IV weiter ausbauen. Dagegen schlägt A2 den umgekehrten Weg ein und verzichtet darauf vollständig zugunsten einer generellen Vereinfachung. Beide Ansätze erscheinen damit konsequenter als Impl IV, welche sich trotzdem als mindestens konkurrenzfähig erwiesen hat im Vergleich mit den besprochenen Ansätzen der Literatur.

Literaturverzeichnis

[AF98] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998.

[ANL87] Ajit Agrawal, Lena Nekludova, and Willie Lim. A parallel o(log N) algorithm for finding connected components in planar images. In *International Conference on Parallel Processing, ICPP'87, University Park, PA, USA, August 1987.*, pages 783–786, 1987.

[AP92] Hussein M. Alnuweiri and Viktor K. Prasanna. Parallel architectures and algorithms for image component labeling. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(10):1014–1034, October 1992.

[AS87] Baruch Awerbuch and Yossi Shiloach. New connectivity and msf algorithms for shuffle-exchange network and pram. *Computers, IEEE Transactions on*, 100(10):1258–1263, 1987.

[BBBC11] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing strongly connected components in parallel on cuda. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 544–555, 2011.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[Buc05] Ian Buck. Taking the plunge into gpu computing. *GPU Gems 2*, pages 509–519, March 2005.

[CC03] Fu Chang and Chun-Jen Chen. A component-labeling algorithm using contour tracing technique. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 2*, ICDAR '03, pages 741–, Washington, DC, USA, 2003. IEEE Computer Society.

[CCL04] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu. A linear-time component-labeling algorithm using contour tracing technique. *Comput. Vis. Image Underst.*, 93(2):206–220, February 2004.

[CCL15] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu. *Implementation zu Changs (2004) linear-time Contour-Tracing CCL Algorithmus*, 2015. http://ocrwks11.iis.sinica.edu.tw/~dar/Download/WebPages/Component.htm.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[CLP99] Fu Chang, Ya-Ching Lu, and Theo Pavlidis. Feature analysis using line sweep thinning algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(2):145–158, February 1999.

[CNP04] Ka Wong Chong, Stavros D Nikolopoulos, and Leonidas Palios. An optimal parallel co-connectivity algorithm. *Theory of Computing Systems*, 37(4):527–546, 2004.

[Coo84] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18(3):223–231, January 1984.

[Coo13] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[CSS89] Robert Cypher, JLCC Sanz, and L Snyder. An erew pram algorithm for image component labeling. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(3):258–262, 1989.

[CV86a] Richard Cole and Uzi Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 478–491. IEEE, 1986.

[CV86b] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control*, 70(1):32–53, July 1986.

[CV89] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, June 1989.

[DAD14] Adrian P Dieguez, Margarita Amor, and Ramon Doallo. Efficient scan operator methods on a gpu. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 190–197. IEEE, 2014.

[DBP10] Johan De Bock and Wilfried Philips. Fast and memory efficient 2-d connected components using linked lists of line segments. *Trans. Img. Proc.*, 19(12):3222–3231, December 2010.

[DGS+08] Yuri Dotsenko, Naga K Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 205–213. ACM, 2008.

[Far12] Rob Farber. *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[FG96] Christophe Fiorio and Jens Gustedt. Two linear time union-find strategies for image processing. *Theor. Comput. Sci.*, 154(2):165–181, February 1996.

[Fre61] H Freeman. Techniques for the digital computer analysis of chain-encoded arbitrary plane curves. *17th National Electronics Conference*, pages 412–432, 1961.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[Gaz86] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 492–501, Oct 1986.

[GHK+13] Benedict R. Gaster, Lee W. Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL - Revised OpenCL 1.2 Edition.* Morgan Kaufmann, 2013.

[HA89] T.D. Haig and Y. Attikiouzel. An improved algorithm for border following of binary images. In *Circuit Theory and Design, 1989., European Conference on*, pages 118 –122, sep 1989.

[Hag88] Torben Hagerup. Optimal parallel algorithms on planar graphs. In *VLSI Algorithms and Architectures*, pages 24–32. Springer, 1988.

[Har81] R.M. Haralick. Some neighborhood operations. In *Real Time Parallel Computing: Image Analysis*, pages 11–35, 1981.

[Har05] Mark Harris. Mapping computational concepts to gpus. *GPU Gems 2*, pages 493–508, March 2005.

[Har15] Mark Harris. *An Efficient Matrix Transpose in CUDA C/C++ (Tutotial)*. Nvidia, 2015. http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/.

[HBARSY11] Uriel H. Hernandez-Belmonte, Victor Ayala-Ramirez, and Raul E. Sanchez-Yanez. A comparative review of two-pass connected component labeling algorithms. In *Proceedings of the 10th international conference on Artificial Intelligence: advances in Soft Computing - Volume Part II*, MICAI'11, pages 452–462, Berlin, Heidelberg, 2011. Springer-Verlag.

[HCS79] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[HCS08] Lifeng He, Yuyan Chao, and K. Suzuki. A run-based two-scan labeling algorithm. *Image Processing, IEEE Transactions on*, 17(5):749 –756, may 2008.

[HCSW09] Lifeng He, Yuyan Chao, Kenji Suzuki, and Kesheng Wu. Fast connected-component labeling. *Pattern Recogn.*, 42(9):1977–1987, September 2009.

[HG11] Mark Harris and Michael Garland. Optimizing parallel prefix operations for the fermi architecture. *GPU Computing Gems Jade Edition*, pages 29–38, 2011.

[HH13] Sang-Won Ha and Tack-Don Han. A scalable work-efficient and depth-optimal parallel scan for the gpgpu environment. *IEEE Trans. Parallel Distrib. Syst.*, 24(12):2324–2333, 2013.

[HLLW11] Mengcheng Huang, Fang Liu, Xuehui Liu, and Enhua Wu. A programmable graphics pipeline in cuda for order-independent transparency. In Wen-Mei W. Hwu, editor, *GPU Computing Gems, Emerald Edition*, pages 427–435. Morgan Kaufmamnn, 2011.

[HLP10] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Comput.*, 36(12):655–678, December 2010.

[Hor05] Daniel Horn. Stream reduction operations for gpgpu applications. *GPU Gems 2*, pages 573–589, March 2005.

[HQN05] Qingmao Hu, Guoyu Qian, and Wieslaw L. Nowinski. Fast connected-component labelling in three-dimensional binary images based on iterative recursion. *Comput. Vis. Image Underst.*, 99(3):414–434, September 2005.

[HS85] Robert M. Haralick and Linda G. Shapiro. Image segmentation techniques. *Computer Vision, Graphics, and Image Processing*, 29(1):100 – 132, 1985.

[HS86] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[HZG08] Qiming Hou, Kun Zhou, and Baining Guo. Bsgp: Bulk-synchronous gpu programming. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 19:1–19:12, New York, NY, USA, 2008. ACM.

[Int15a] Intel. *Intel Core i3-2100 (Datenblatt)*, 2015. http://ark.intel.com/de/products/53422/Intel-Core-i3-2100-Processor-3M-Cache-3_10-GHz.

[Int15b] Intel. *Intel Core i7-2700K Processor (Datenblatt)*, 2015. http://ark.intel.com/de/products/61275/Intel-Core-i7-2700K-Processor-8M-Cache-up-to-3_90-GHz.

[Int15c] Intel. *Intel Core i7-5960X Processor Extreme Edition (Datenblatt)*, 2015. http://ark.intel.com/de/products/82930.

[Int15d] Intel. *Intel SDK for OpenCL Applications 2012 Optimization Guide*, 2015. https://software.intel.com/sites/landingpage/opencl/optimization-guide/.

[JDM00] Anil K. Jain, Robert P. W. Duin, and Jianchang Mao. Statistical pattern recognition: A review. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(1):4–37, January 2000.

[KBA+13] M.J. Klaiber, D.G. Bailey, S. Ahmed, Y. Baroud, and S. Simon. A high-throughput fpga architecture for parallel connected components analysis based on label reuse. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 302–305, Dec 2013.

[KDH10] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[Kes15] Christoph Kessler. Fda125 app lecture 2: Foundations of parallel algorithms (2003 lecture notes), 2015. https://www.ida.liu.se/~chrke55/courses/APP/ps/f2pram-2x2.pdf.

[KG15a] Khronos-Group. *Release of OpenCL 1.1 Specification (Press Release)*, 2015. https://www.khronos.org/news/press/2010/06.

[KG15b] Khronos-Group. *Release of OpenCL 1.2 Specification (Press Release)*, 2015. https://www.khronos.org/news/press/2011/11.

[KG15c] Khronos-Group. *Release of OpenCL 2.0 Specification (Press Release)*, 2015. https://www.khronos.org/news/press/2013/11.

[KH13] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[khr15] Khronos group, 2015. https://www.khronos.org/.

[KKS00] Jin Ho Kim, Kye Kyung Kim, and Ching Y. Suen. An hmm-mlp hybrid model for cursive script recognition. *Pattern Analysis and Applications*, 3:314–324, 2000.

[KPMS09] Praveen Kumar, Kannappan Palaniappan, Ankush Mittal, and Guna Seetharaman. Parallel blob extraction using the multi-core cell processor. In Jacques Blanc-Talon, Wilfried Philips, Dan Popescu, and Paul Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems*, volume 5807 of *Lecture Notes in Computer Science*, pages 320–332. Springer Berlin Heidelberg, 2009.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J.van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume A: Algorithms and Complexity*, pages 869–941. Elsevier, Amsterdam, 1990.

[KRKS11] Oleksandr Kalentev, Abha Rai, Stefan Kemnitz, and Ralf Schneider. Connected component labeling on a 2d grid using cuda. *Journal of Parallel and Distributed Computing*, 71(4):615 – 620, 2011.

[KW03] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM, 2003.

[LD94] S. Lakshmivarahan and Sudarshan K. Dhall. *Parallel computing using the prefix problem.* Oxford University Press, 1994.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

[LHA+09] Aaron Lefohn, Mike Houston, Johan Andersson, Ulf Assarsson, Cass Everitt, Kayvon Fatahalian, Tim Foley, Justin Hensley, Paul Lalonde, and David Luebke. Beyond programmable shading (parts i and ii). In *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, pages 7:1–7:312, New York, NY, USA, 2009. ACM.

[LKM01] Erik Lindholm, Mark J. Kligard, and Henry P. Moreton. A user-programmable vertex engine. In *SIGGRAPH*, pages 149–158, 2001.

[LPN+13] Joo Hwan Lee, Kaushik Patel, Nimit Nigania, Hyojong Kim, and Hyesoon Kim. Opencl performance evaluation on modern multi core cpus. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1177–1185, Washington, DC, USA, 2013. IEEE Computer Society.

[lwj15] *Lightweight Java Game Library (LWJGL)*, 2015. http://www.lwjgl.org/.

[MB13] Russ Miller and Laurence Boxer. *Algorithms Sequential and Parallel: A Unified Approach*. Cengage Learning, 2013.

[MDTP+04] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, August 2004.

[MGM+11] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[MT04] Michael D. McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. A K Peters, 2004.

[NM82] Dhruva Nath and SN Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Information Processing Letters*, 14(1):7–11, 1982.

[Nvi15a] Nvidia. *Cuda C Programming Guide, Version 7.0*, 2015. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3V3POAyDb.

[Nvi15b] Nvidia. *GeForce 3*, 2015. http://www.nvidia.com/page/geforce3.html.

[Nvi15c] Nvidia. *Graphics Drivers for Windows, Version 350.12*, 2015. http://us.download.nvidia.com/Windows/350.12/350.12-win8-win7-winvista-desktop-release-notes.pdf.

[Nvi15d] Nvidia. *Nvidia GeForce 8*, 2015. http://www.nvidia.com/page/geforce8.html.

[Nvi15e] Nvidia. *Nvidia GeForce GTX 480 (Datenblatt)*, 2015. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications.

[Nvi15f] Nvidia. *Nvidia GeForce GTX 670 (Datenblatt)*, 2015. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-670/specifications.

[Nvi15g] Nvidia. *NVIDIA GeForce GTX 680 Whitepaper*, 2015. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[Nvi15h] Nvidia. *Nvidia GeForce GTX 980 (Datenblatt)*, 2015. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications.

[Nvi15i] Nvidia. *NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made. (Whitepaper)*, 2015. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[Nvi15j] Nvidia. *Nvidia GeForce GTX Titan (Datenblatt)*, 2015. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications.

[Nvi15k] Nvidia. *NVIDIA Kepler GK 110 Whitepaper*, 2015. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[Nvi15l] Nvidia. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi (Whitepaper)*, 2015. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[ODTT95] A. K. Jain O. D. Trier and T. Taxt. Feature extraction methods for character recognition - a survey, 1995.

[OL10] Victor M. A. Oliveira and Roberto A. Lotufo. A study on connected components labeling algorithms using gpus (undergraduate work). 2010. http://adessowiki.fee.unicamp.br/media/Attachments/courseIA366F2S2010/aula10/gpu_victor.pdf.

[OS11] Bedrich Benes Ondrej Stava. Connected component labeling in cuda. In Wen-Mei W. Hwu, editor, *GPU Computing Gems, Emerald Edition*, pages 569–581. Morgan Kaufmamnn, 2011.

[OS15] Bedrich Benes Ondrej Stava. *Implementation zu Stavas (2011) Cuda CCL Algorithmus*, 2015. http://www.gpucomputing.net/content/connected-component-labeling-cuda-democode.

[PBMH02] Timothy J Purcell, Ian Buck, William R Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 703–712. ACM, 2002.

[PF86] E Persoon and K S Fu. Shape discrimination using fourier descriptors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(3):388–397, March 1986.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[POAU00] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *SIGGRAPH*, pages 425–432, 2000.

[RK82] Azriel Rosenfeld and Avinash C. Kak. *Digital Picture Processing*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 1982.

[Ros70] Azriel Rosenfeld. Connectivity in digital pictures. *J. ACM*, 17(1):146–160, January 1970.

[RR10] Thomas Rauber and Gudula Ruenger. *Parallel Programming - for Multicore and Cluster Systems.* Springer, 2010.

[SAM+10] Takashi Shimokawabe, Takayuki Aoki, Chiashi Muroi, Junichi Ishida, Kohei Kawano, Toshio Endo, Akira Nukada, Naoya Maruyama, and Satoshi Matsuoka. An 80-fold speedup, 15.0 tflops full gpu acceleration of non-hydrostatic weather model asuca production code. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[SFB+09] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1):4:1–4:11, February 2009.

[SFSV13] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance traps in opencl for cpus. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '13, pages 38–45, Washington, DC, USA, 2013. IEEE Computer Society.

[SHS03] Kenji Suzuki, Isao Horiba, and Noboru Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1 – 23, 2003.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[SLO06] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A work-efficient step-efficient prefix sum algorithm, in: Workshop on edge computing using new commodity architectures, 2006.

[SM00] Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888 –905, aug 2000.

[SS79] Walter J. Savitch and Michael J. Stimson. Time bounded random access machines with parallel processing. *J. ACM*, 26(1):103–118, January 1979.

[SSKLK13] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[SSR01] Jasjit S. Suri, Sameer Singh, and Laura Reden. Computer vision and pattern recognition techniques for 2-d and 3-d mr cerebral cortical segmentation: A state-of-the-art review. *JOURNAL OF PATTERN ANALYSIS AND APPLICATIONS*, page 2002, 2001.

[SV82] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[UA90] Jayaram K. Udupa and Venkatramana G. Ajjanagadde. Boundary and object labelling in three-dimensional images. *Comput. Vision Graph. Image Process.*, 51(3):355–369, July 1990.

[Vis83] Uzi Vishkin. Synchronous parallel computation - a survey. Technical report, 1983.

[Vol10] Vasily Volkov. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference, GTC*, 10, 2010.

[WB03] Yang Wang and Prabir Bhattacharya. Using connected components to guide image understanding and segmentation. *MG&V*, 12(2):163–186, February 2003.

[Wen07] Henning Wenke. Earth weather 3d - visualisierung und animation dynamisch bestimmter teilmengen von wetterdaten auf webseiten mit opengl. Master's thesis, University of Osnabrueck, Osnabrueck, Germany, 2007.

[Wik15a] Wikipedia. *3dfx Voodoo Graphics*, 2015. http://en.wikipedia.org/wiki/3dfx_Interactive#Voodoo_Graphics_PCI.

[Wik15b] Wikipedia. *DirectX 1.0 - 7.1*, 2015. http://en.wikipedia.org/wiki/DirectX.

[Wik15c] Wikipedia. *GeForce 400 series (Products, GTX 480)*, 2015. http://en.wikipedia.org/wiki/GeForce_400_series.

[Wik15d] Wikipedia. *Quake*, 2015. http://en.wikipedia.org/wiki/Quake_(video_game).

[Wik15e] Wikipedia. *Wikipedia - Close to Metal*, 2015. http://en.wikipedia.org/wiki/Close_to_Metal.

[WKV07] Henning Wenke, Ralf Kunze, and Oliver Vornberger. World weather 3d. *UDayV, Informieren mit Computeranimation*, pages 158–165, 2007.

[WKV14] Henning Wenke, Sascha Kolodzey, and Oliver Vornberger. A work-optimal parallel connected-component labeling algorithm for 2d-image-data using pre-contouring. *International Workshop on Image Processing*, pages 154–161, august 2014.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[WOS09] Kesheng Wu, Ekow Otoo, and Kenji Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Anal. Appl.*, 12(2):117–135, February 2009.

[Wyl79] James C. Wyllie. The complexity of parallel computations. Technical report, Ithaca, NY, USA, 1979.