Danksagung

Ich danke meinem Doktorvater Professor Norbert Luttenberger für die Möglichkeit, in seiner Arbeitsgruppe zu arbeiten und zu forschen. Insbesondere bedanke ich mich für die mir gewährte Freiheit, mein Forschungsthema nach eigenen Vorstellungen zu entwickeln und zu interpretieren, für die fortwährende Unterstützung meiner Arbeit und die Möglichkeit, Ergebnisse auf internationalen Tagungen zu präsentieren. Ob ich wohl irgendwann wieder einen Chef finde, der gelegentlich für seine Mitarbeiter kocht?

Diese Arbeit ist mit Hilfe und Unterstützung vieler weiterer Personen entstanden. Der Gedankenaustausch mit meinen Kollegen Jochen Koberstein, Jesper Zedlitz und Michael Lodemann sowie unsere gemeinsamen Projekte haben mir geholfen, meine eigene Arbeit zu hinterfragen und aus verschiedenen Blickwinkeln zu betrachten. Dr. Claus Traulsen und Dr. Hauke Fuhrmann haben mir oft wissenschaftlich und persönlich geholfen und mich unterstützt. Maren Lutz, Nicole Maard-Azad und Matthias Westphal waren eine große Hilfe bei organisatorischen und technischen Fragen.

Ganz besonders danke ich Ole Schulz-Hildebrandt, Martin Köper, Jonas Bötel und Christian Hoffmann, die in ihren hervorragenden studentischen Abschlussarbeiten wichtige Beiträge zur vorliegenden Dissertation geleistet haben. Die gemeinsame Arbeit mit ihnen allen und insbesondere mit Ole Schulz-Hildebrandt war außergewöhnlich inspirierend und motivierend.

Ich bedanke mich bei Dr. habil. Heidrun Peters, Ole Schulz-Hildebrandt, Martin Köper, Jonas Bötel und Jesper Zedlitz für die vielen hilfreichen Korrekturen, Anmerkungen und Hinweise zu meiner Dissertationsschrift.

Abschließend danke ich meiner ganzen Familie für ihre Unterstützung und besonders meiner Frau Judith und meinem Sohn Arne für ihren Rückhalt und ihr Verständnis.

XML-Verarbeitung auf Grafikkarten

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Hagen Wilhelm Peters

Kiel
2011

1. Gutachter	Professor Norbert Luttenberger Institut für Informatik Christian-Albrechts-Universität zu Kiel
2. Gutachter	Professor Manfred Schimmler Institut für Informatik Christian-Albrechts-Universität zu Kiel
Datum der mündlichen Prüfung	23. Januar 2012

Die *Extensible Markup Language* (XML) ist ein weit verbreitetes, strukturiertes Format zur Datenspeicherung und zum Datenaustausch. Die Komplexität und die Menge der zu verarbeitenden XML-Daten hat in den vergangenen Jahren erheblich zugenommen, und sowohl in der Wissenschaft wie auch in der Wirtschaft wird an Verfahren zur schnellen Verarbeitung von XML gearbeitet. In einigen Arbeiten werden auch Verfahren zur parallelen Verarbeitung von XML mittels üblicher Mehrkern-CPUs entwickelt.

Seit einigen Jahren gewinnt die parallele Verarbeitung allgemeiner Probleme mittels Grafikprozessoren (GPUs) an Bedeutung. GPUs ähneln in ihrer Architektur und Arbeitsweise dem bekannten *Single Instruction Multiple Data*-Prinzip. Daher sind GPUs besonders gut geeignet für die Verarbeitung massiv-datenparalleler Probleme wie etwa der Matrixmultiplikation oder der Bildverarbeitung.

XML-Dokumente bzw. XML-Datenmodelle haben eine baumartige Struktur und folglich basieren viele der Verfahren zur Verarbeitung von XML auf dieser Baumstruktur. Solche Strukturen und Verfahren können jedoch wegen der Architektur von GPUs nicht trivial von CPUs auf GPUs übertragen werden, selbst wenn es sich um Verfahren zur parallelen Verarbeitung mittels Mehrkern-CPUs handelt. Daher existieren bisher kaum Arbeiten, die die parallele Verarbeitung von XML mittels GPUs untersuchen.

In der vorliegenden Arbeit wird anhand einer konkreten XML-Anwendung, der Transformation von XML-Dokumenten mit XSLT, untersucht, inwieweit die Verarbeitung von XML an die massiv-parallele Architektur von GPUs angepasst werden kann. Dazu wird zunächst ein Konzept zur XSLT-Verarbeitung durch GPUs entworfen und danach werden die vier wichtigsten in diesem Konzept eingesetzten neuen Verfahren vorgestellt: ein schnelles Sortierverfahren für GPUs, ein Verfahren um GPUs als asynchrone Koprozessoren für die CPU einsetzen zu können, ein Verfahren um XPath-Ausdrücke parallel auszuwerten und schließlich ein Verfahren zur XSLT-Verarbeitung mittels GPUs.

In umfangreichen Tests wird gezeigt, dass mit den hier dargelegten Verfahren die Nutzung von GPUs für die XPath- und XSLT-Verarbeitung in vielen Fällen zu einer Leistungssteigerung gegenüber der Verarbeitung durch Mehrkern-CPUs führt. Es ist demnach möglich, GPUs auch zur Verarbeitung von XML gewinnbringend einzusetzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

KAPITEL 1

Einleitung

Die *Extensible Markup Language* (XML) [93] ist ein weit verbreitetes Format, um strukturierte Daten zu speichern beziehungsweise auszutauschen. Durch die Möglichkeit XML-basierte Sprachen formal zu definieren, wird XML nicht nur für viele verschiedene Anwendungen genutzt, sondern bildet auch die Grundlage für die formale Beschreibung der Kommunikation verschiedener Netzwerkdienste [87]. Inzwischen sind auch generische XML-Verarbeitungsprozesse (beispielsweise Navigation in oder Transformation von XML-Dokumenten) weit verbreitet. Diese müssen nicht für die spezifische XML-basierte Sprache angepasst werden, sondern können direkt auf den generischen durch XML definierten Datenmodellen arbeiten. Einerseits durch die zunehmende Verbreitung von XML und andererseits durch die zunehmende Komplexität der XML-Verarbeitung (beziehungsweise der zunehmenden Menge mittels XML ausgetauschter Daten) ist die schnelle Verarbeitung von XML wichtiger geworden.

Nach der erstmals von Moore im Jahre 1965 formulierten Gesetzmäßigkeit [56] steigt die Transistordichte von CPUs exponentiell mit der Zeit. Über mehrere Jahrzehnte hinweg ging mit der steigenden Komplexität der CPUs auch eine steigende Rechenleistung für *sequentielle* Prozesse einher, unter anderem weil die höhere Anzahl Transistoren für mehr Parallelität auf Instruktionslevel verwendet wurde und generell kleinere Bauformen eine höhere Taktfrequenz ermöglichen.

Inzwischen wird jedoch die höhere Anzahl Transistoren dafür genutzt, mehrere *parallel* arbeitende *cores* in einer CPU zu vereinen, im Prinzip also mehrere Recheneinheiten mit ggf. geteiltem Speicher auf einem Chip bereitzustellen. Die

Taktfrequenz dagegen wurde in den vergangenen Jahren kaum noch erhöht. So hat beispielsweise die Firma Intel den Pentium I Prozessor mit einer Taktfrequenz von unter 100 MHz im Jahre 1993 vorgestellt. Im Jahre 2000 wurde der Pentium 4 Prozessor mit mehreren GHz Taktfrequenz eingeführt [25]. Seitdem hat sich die maximale Taktfrequenz handelsüblicher Prozessoren für Desktop-Rechner kaum geändert.

Neben den CPUs werden seit einigen Jahren auch GPUs (wie die CPU eine Standardkomponente eines Computers), vermehrt zur Berechnung allgemeiner Probleme eingesetzt. Diese Hardware wurde ursprünglich nur für Berechnungen im Zusammenhang mir der Darstellung von Grafik am Monitor verwendet (siehe Abschnitt 3.1). GPUs erzielen ihre Rechenleistung noch mehr durch die Verwendung vieler paralleler Recheneinheiten als moderne CPUs. Obwohl die theoretische Rechenleistung (FLOPS) moderner Desktop-GPUs über der Rechenleistung moderner CPUs liegt, ist die Leistung von GPUs bezüglich sequentieller Prozesse aufgrund der parallelen Architektur derzeit deutlich unter der Leistung von CPUs.

Demzufolge steigt die Leistung sequentieller Prozesse viel weniger stark, als die reine Rechenleistung moderner CPUs oder auch GPUs erwarten ließe. Dieses Problem kann nur gelöst werden, wenn Wege gefunden werden, bisher sequentielle Algorithmen in eine parallele Entsprechung zu überführen bzw. bestehende Algorithmen für die speziellen Architekturen leistungsfähiger paralleler Rechner wie GPUs anzupassen.

Es existieren bereits verschiedene Ansätze, um die Verarbeitung von XML-Prozessen zu parallelisieren. Einige Ansätze verwenden dazu dedizierte Hardware, also keine generischen Standardkomponenten wie CPUs und GPUs. Diese Ansätze zielen allerdings in der Regel nicht darauf ab, einzelne XML-Prozesse zu parallelisieren, vielmehr können mit diesen Systemen viele Prozesse, die lesend auf die gleichen XML-Daten zugreifen, parallel verarbeitet werden. In [28] etwa verwenden El-Hassan et al. *Field-Programmable Gate Arrays* (FPGAs), um das Parsen von XML-Dokumenten zu beschleunigen. In [29] entwickeln die Autoren ein durch FPGAs unterstütztes Filtersystem für XML-Dokumente. Mitra et al. entwickeln in [55] ebenfalls ein auf FPGAs basierendes System zur Filterung von XML-Dokumenten. Moussalli et al. stellen in [59, 57] durch die Verwendung von FPGAs beschleunigte Filter- und *Query Matching*-Syteme vor. IBM, Intel und LSI bieten ebenfalls dedizierte Hardware für die parallele XML-Verarbeitung an [53, 43, 44].

Eine Reihe von Verfahren zur Parallelisierung von einzelnen XML-Prozessen

wurde auch für Standard-CPUs entwickelt. Beispielsweise haben Lu und Pan et al. [54, 64, 65], Chen et al. [24] oder auch Shah et al. [78] Ansätze entwickelt, das Parsen und Serialisieren von XML-Dokumenten auf Mehrkern-CPUs zu parallelisieren. Bordawekar et al. haben parallele Varianten der XPath-Verarbeitung entwickelt [19, 18], Sun et al. sowie Feng et al. entwickelten parallele Prozesse zur Verarbeitung von XSLT [79, 30]. Alle diese Ansätze, um einzelne XML-Prozesse parallel zu verarbeiten, verwenden (Mehrkern-) CPUs als Zielhardware.

Die bisher genannten Ansätze zur parallelen XML-Verarbeitung sehen also entweder die Verwendung von dedizierter Hardware vor, wobei einzelne XML-Prozesse nicht parallelisiert werden, oder aber verwenden Standard-CPUs mit einer geringen Anzahl paralleler Recheneinheiten, die nach dem MIMD-Prinzip (*Multiple Instruction Multiple Data*) [32] arbeiten.

GPUs eignen sich aufgrund ihrer Architektur besonders gut für massiv-datenparallele Berechnungen wie etwa eine Matrixmultiplikation. Deshalb erscheint die Verarbeitung eines baumartigen XML-Dokuments mittels Sprachen und Werkzeugen, die ebenso die Baumstruktur des Eingabedokuments voraussetzen und nutzen, zunächst für die GPU ungeeignet.

In der vorliegenden Arbeit wird untersucht, inwieweit komplexe XML-Prozesse dennoch parallel auf GPUs verarbeitet werden können, das heißt unter Verwendung einer massiv-parallelen Hardware, die nach dem SIMD-Prinzip (Single Instruction Multiple Data) [32] arbeitet.

Das Ziel dieser Arbeit ist es, eine prototypische Implementierung eines XSLT-Prozessors für GPUs zu entwickeln. Dazu werden zum einen parallele Ansätze entwickelt, um Teile der XML-basierten Sprachen XPath und XSLT mittels einer GPU zu verarbeiten. Zum anderen wird ein System bereitgestellt, wodurch die GPU als Koprozessor eingesetzt wird, welcher von verschiedenen CPU-Prozessen asynchron gleichzeitig genutzt werden kann.

Die Arbeit ist wie folgt gegliedert: Der erste Teil (I) gibt eine Einführung in die in dieser Arbeit verwendeten Standards, Sprachen und Technologien und stellt das Konzept vor, welches dieser Arbeit zugrunde liegt.

In den beiden folgenden Teilen II und III werden die vier wichtigsten Algorithmen bzw. Verfahren vorgestellt, die für diese Arbeit entwickelt wurden. Jedes dieser Verfahren wird dabei für sich evaluiert. Teil II erläutert die beiden Verfahren, die nicht auf XML basieren: ein optimiertes Sortierverfahren für GPUs und einen Ansatz, um eine GPU als asynchronen Koprozessor verwenden zu können.

Teil III stellt die hier entwickelten Verfahren vor, die auf XML-basieren: Einen parallelen XSLT-Prozessor sowie ein Verfahren zur schnellen parallelen Verarbeitung von XPath-Ausdrücken auf paralleler Hardware.

In Teil IV werden die Ergebnisse der vorgestellten Arbeiten zusammengefasst und diskutiert und offene Fragen und Arbeiten genannt.

Teil I

Voraussetzung & Konzept

KAPITEL 2

XML-Technologien

Die *Extensible Markup Language* (XML) [93] ist eine Auszeichnungssprache, mit der Daten in Textdokumenten strukturiert dargestellt und beschrieben werden können. XML ist eine Untermenge der *Standard Generalized Markup Language* (SGML) [33] und wurde 1998 vom *World Wide Web Consortium* (W3C) als *Empfehlung* (*Recommendation*) [95] verabschiedet. Bei der Entwicklung von XML standen verschiedene Aspekte im Vordergrund, unter anderem sollte XML für Computer einfach zu verarbeiten und für Menschen einfach zu verstehen sein. Sicherlich sind das zwei der wichtigsten Gründe dafür, dass XML heute eine sehr weit verbreitete Auszeichnungssprache ist und XML-basierte Sprachen in den unterschiedlichsten Bereichen verwendet werden, etwa zur Beschreibung von Vektorgrafiken (SVG [94]), Geoinformationen (OSM [12], GML [36]) und Textdokumenten (ODF [35], DOCX [37]). Darüber hinaus wird XML auch zur Definition und Beschreibung komplexer Prozesse eingesetzt, wie beispielsweise Dokumenttransformationen (XSLT [85], XQuery [97]). Eine Übersicht über viele verschiedene XML-basierte Sprachen findet der Leser unter [74].

Wie in der Einleitung angesprochen, ist das Ziel dieser Arbeit die prototypische Entwicklung eines XSLT-Prozessors für GPUs. Dazu wird zunächst in Abschnitt 2.1 eine kurze Einführung in XML gegeben. Für die Verarbeitung von XML werden verschiedene XML-Datenmodelle eingesetzt. Eine Einführung in drei wichtige Modelle gibt Abschnitt 2.2. Anschließend wird in Abschnitt 2.3 XPath und in Abschnitt 2.4 XSLT vorgestellt.

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <html lang="en">
3   <head>
4     <title>Titel</title>
5   </head>
6   <body>
7     <p>
8       X
9     </p>
10    <p>
11      M
12      <br />
13      L
14    </p>
15  </body>
16 </html>
```

Quellcode 2.1: XML-Beispieldokument

2.1 XML

Quellcode 2.1 zeigt beispielsweise ein einfaches XML-Dokument, genauer ein xHTML-Dokument [88]. Die wichtigsten Bestandteile eines XML-Dokuments sind

- *element*-Knoten (Elementknoten oder Elemente)
- *text*-Knoten (Textknoten)
- *attribute*-Schlüssel-Wert-Paare (Attributknoten oder Attribute)

Ein Elementknoten besteht aus einem öffnenden und einem schließenden *tag* (Tag). Dazwischen können sich weitere Elementknoten oder auch Textknoten befinden. Befinden sich keine weiteren Knoten dazwischen, so kann das öffnende und das schließende Tag auch als ein einzelnes Tag dargestellt werden: so kann etwa

 durch

ersetzt werden. Ein Attribut gehört immer zu genau einem Element und wird im XML-Dokument innerhalb des öffnenden Tags des entsprechenden Elements notiert. XML-Dokumente sind so verschachtelt, dass ein Element A ein anderes Element B immer entweder vollständig umschließt (beide Tags von B liegen zwischen den Tags von A) oder überhaupt nicht umschließt (kein Tag von B liegt zwischen den Tags von A). Dadurch wird eine natürliche Kind/Eltern-Beziehung

zwischen Elementen definiert. Da zudem gilt, dass ein XML-Dokument genau ein Element besitzt, welches alle anderen Elemente umschließt (in Bäumen: Wurzel), werden XML-Dokumente häufig als Baum dargestellt (Abbildung 2.1).

Es sei an dieser Stelle darauf hingewiesen, dass genaugenommen in der XML-Spezifikation wie auch in den nachfolgend vorgestellten Modellen eigentlich zwei Bäume definiert werden: Zum einen der Dokumentbaum, der das gesamte Dokument repräsentiert. Das erwähnte alles umschließende Element (`<html>` in Quellcode 2.1) ist ein Kind der Wurzel dieses Dokumentbaums. Zum anderen wird der Elementbaum definiert, in dem `<html>` die Wurzel ist. Leider wird in den Standards für XML und die Datenmodelle für das oberste Element beider Bäume gelegentlich der Begriff *root* verwendet. Wenn nicht anders gekennzeichnet, ist im Folgenden immer der Elementbaum gemeint.

2.2 XML-Datenmodelle

Es existieren verschiedene Datenmodelle, mit denen XML-Dokumente modelliert werden können. Dabei verfolgen die Datenmodelle entsprechend des Kontexts, in dem sie entwickelt wurden, unterschiedliche Ziele. So definiert etwa das DOM zusätzlich zur Struktur des Dokuments auch geeignete Schnittstellen zum Zugriff auf die Daten und Schnittstellen, um Daten in Dokumenten (bzw. im DOM) zu manipulieren und zu entfernen. Andere Datenmodelle, etwa das XML Infoset, sehen solche Schnittstellen zum Zugriff auf das Modell nicht vor. Im Folgenden sollen drei wichtige Modelle für XML-Dokumente kurz vorgestellt und einige Gemeinsamkeiten und Unterschiede aufgezeigt werden.

2.2.1 DOM

Das *Document Object Model* (DOM) [82, 86, 89] ist ein Modell und eine Schnittstelle für HTML [83] und XML-Dokumente. Durch die von DOM definierten Schnittstellen können die Daten von Dokumenten gelesen, manipuliert und auch gelöscht werden. Im DOM werden alle XML-Datenobjekte durch das Interface *node* (Knoten) (Anhang A) dargestellt. Die unterschiedlichen XML-Knoten erweitern dann dieses gemeinsame Interface um spezifische Eigenschaften oder erlauben für bestimmte Interface-Attribute nur spezifische Werte; so erweitert etwa das Interface `text` das Interface *node* um die Methode `splitText`, darf aber als Wert der Eigen-

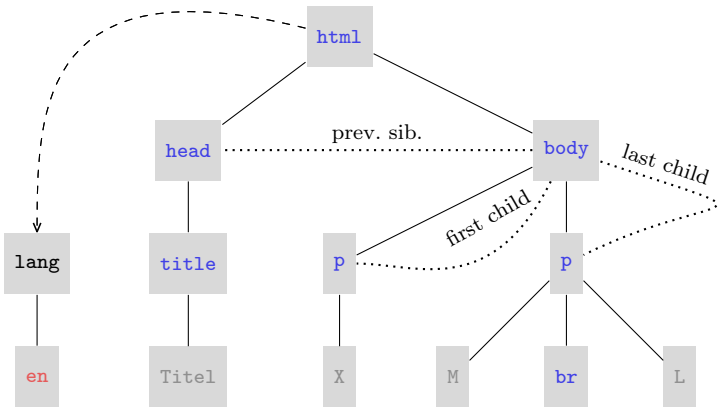


Abbildung 2.1: XML-Dokument als DOM-Baum

schaft `childNodes` nur `NULL` haben. In der vorliegenden Arbeit werden allgemeine XML-Datenobjekte wie Elemente, Attribute etc. auch als „Knoten“ oder „XML-Knoten“ bezeichnet.

Der DOM-Baum ergibt sich aus allen DOM-Knoten, die Kinder von anderen DOM-Knoten sind, es sind also insbesondere nicht alle Datenobjekte im Baum enthalten. Entsprechend repräsentieren die Kanten des DOM-Baums Beziehungen zwischen Objekten, die durch die Attribute `childNodes` bzw. `parentNode` gegeben sind. Die Abbildung 2.1 zeigt den DOM-Baum, der das in Quellcode 2.1 dargestellte Dokument repräsentiert.

Anders als in anderen Modellen kennt im DOM ein Elementknoten nicht nur die Liste seiner Kinder und den Eltern-Knoten, sondern es existieren zusätzlich Zeiger auf das erste und das letzte Kind (bezüglich der Dokumentreihenfolge) sowie Zeiger auf Geschwisterknoten. In der Abbildung 2.1 sind diese zusätzlichen Kanten beispielhaft für das `body`-Element gepunktet dargestellt.

Laut DOM-Standard sind XML-Attributknoten keine Kinder von anderen Knoten, sondern Eigenschaften von Elementen. Deswegen stehen sie außerhalb des DOM-Baums, in der Abbildung durch gestrichelte Linien angedeutet. Nichtsdestoweniger sind die Text-Werte von Attributen wiederum echte Kinder von Attributen.

Eine weitere Besonderheit des DOM ist, dass Attributknoten zwar von Elementknoten referenziert werden können, Attributknoten jedoch keine Referenzen auf die

Elemente besitzen, von denen sie referenziert werden. Dies wird in der Abbildung 2.1 durch gerichtete bzw. ungerichtete Kanten dargestellt.

2.2.2 XML Infoset

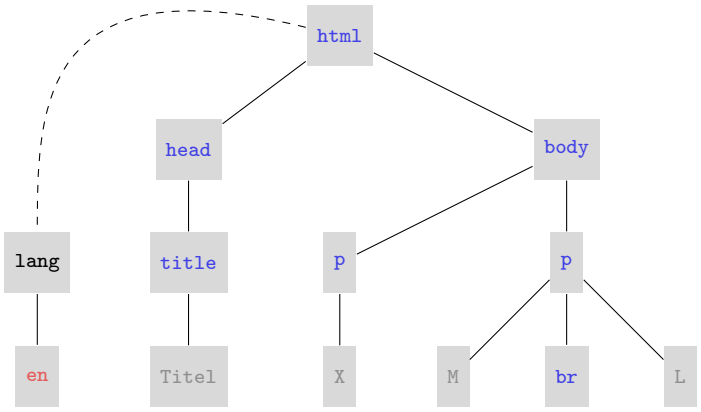


Abbildung 2.2: XML-Dokument als Infoset-Baum

Das XML Infoset [90] stellt im Gegensatz zu DOM keine Schnittstelle zum Manipulieren, Hinzufügen oder Entfernen von Daten aus einem Dokument dar, sondern beschreibt lediglich die Menge der Informationen, die durch das Parsen eines XML-Dokuments gewonnen werden kann. Die Informationen werden ähnlich (jedoch nicht gleich) den Knoten im DOM durch XML Infoitems dargestellt, und das XML Infoset entspricht etwa dem DOM-Baum. Anders als im DOM haben im XML Infoset ausschließlich *element information items* und das *document information item* eine Referenz auf ihre Kinder, weitere Referenzen, etwa auf Geschwisterknoten, gibt es nicht. Attribute stehen auch im XML Infoset außerhalb des eigentlichen Dokumentbaums, anders als im DOM besitzen *attribute information items* jedoch eine Referenz auf das Element, zu dem sie gehören. Wie der DOM-Baum ergibt sich auch der XML Infoset-Baum aus der Wurzel und denjenigen XML Infoitems, die Kinder anderer XML Infoitems sind. Die Abbildung 2.2 stellt das Dokument aus Quellcode 2.1 als XML Infoset-Baum dar.

XPath-*node*	XML Infoitem
root	*document*
element	*element*
text	*character* (mehrere)
attribute	*attribute*
namespace	*namespace declaration*
processing instruction	*processing*
comment	*comment*

Tabelle 2.1: Die sieben XPath-Knoten

Verwandtschaftsbeziehung	Achsen
	self
Nachfahren	child, descendant, descendant-or-self
Vorfahren	parent, ancestor, ancestor-or-self
Seitenlinien (nachstehend)	following, following-sibling
Seitenlinien (vorstehend)	preceding, preceding-sibling
keine Verwandtschaft	attribute, namespace

Tabelle 2.2: Die dreizehn XPath-Achsen

2.2.3 XPath-Datenmodell

Die *XML Path Language* (XPath) [84, 96] ist eine nicht-XML-basierte Sprache, deren Hauptziel es ist, mittels eines XPath-Ausdrucks Teile eines XML-Dokuments zu adressieren. Dafür nutzt XPath ein eigenes Datenmodell für das zu verarbeitende XML-Dokument. Dieses Datenmodell sieht wie das XML Infoset keine Schnittstellen zum Manipulieren des Dokuments vor. Im XPath-Datenmodell wird jedes Datenobjekt durch einen XPath-Knoten repräsentiert. Anders als die beiden anderen Modelle DOM und XML Infoset sind die Beziehungen zwischen den Knoten nicht durch Referenzen bzw. Eigenschaften definiert, sondern durch s.g. XPath-Achsen. Ausgehend von einem ausgewählten Knoten, dem Kontextknoten, spezifiziert eine Achse eine Menge von Knoten, die mit dem Kontextknoten in der durch die Achse definierten Beziehung stehen; man spricht davon, dass bestimmte

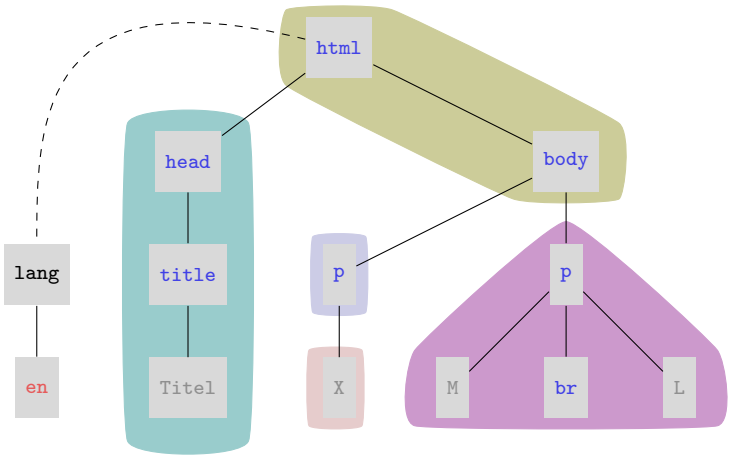


Abbildung 2.3: XML-Dokument als XPath-Datenmodell-Baum

Knoten von einem Kontextknoten aus über eine bestimmte Achse erreichbar sind.

Die Tabelle 2.1 nennt die verschiedenen Knoten im XPath-Datenmodell und zum besseren Verständnis die entsprechenden XML Infoitems im XML Infoset. Im XPath-Datenmodell bezeichnet *root* die Dokumentwurzel, also den Elternknoten des alles umschließenden Elements. Die Tabelle 2.2 listet die Achsen auf, mit denen im XPath-Datenmodell navigiert werden kann, und nennt die entsprechende „Verwandtschaftsbeziehung“ der Knoten im Baum. Im Anhang B befindet sich der dem Standard [84] entnommene Abschnitt zur Definition der Achsen.

Attribute können im XPath-Datenmodell durch die *attribute*-Achse erreicht werden. Anders als im DOM existiert eine Verbindung von den Attributen eines Elements zu diesem Element: Das Element, zu dem Attribute gehören, ist der Elternknoten dieser Attribute und kann daher über die *parent*-Achse erreicht werden (vgl. Abschnitt 8.5).

Wie auch das XML Infoset und das DOM modelliert das XPath-Datenmodell ein XML-Dokument als Baum. Der Kern des Baums besteht dabei wie in den anderen Modellen aus Knoten, die Kinder anderer Knoten sind, das heißt auch im XPath-Baum werden üblicherweise nur die Knoten und die Beziehung zwischen ihnen dargestellt, die sich aus den Achsen *child* und *parent* ergeben. Die Abbildung 2.3 zeigt das XML-Dokument aus Quellcode 2.1, wie es im XPath-Datenmodell-Baum dargestellt werden kann. Zusätzlich sind in der Abbildung beispielhaft für den linken Knoten <p> die fünf Knotenmengen farbig markiert, die durch

- die *self*-Achse,
- die *preceding*-Achse,
- die *following*-Achse,
- die *ancestor*-Achse und
- die *descendant*-Achse

zu erreichen sind.

Die Abbildung illustriert auch eine besondere Eigenschaft des XPath-Datenmodells: Für jeden Kontextknoten K gilt, dass die Knotenmengen, die von K aus über die Achsen *descendant*, *ancestor*, *preceding*, *following* und *self* erreicht werden, disjunkt sind und in der Vereinigung alle Knoten des XPath-Dokuments enthalten, also eine Partition des Dokuments bilden.

2.3 XPath

Die *XML Path Language* (XPath) [84, 96] ist eine Sprache, deren Hauptziel es ist, einzelne Knoten oder auch ganze Knotenmengen in einem XML-Dokument zu adressieren. Zwar wurde XPath entwickelt um in den Sprachen XSLT und XPointer eingesetzt zu werden, inzwischen wird XPath aber auch in anderen Kontexten verwendet, etwa in XML Schema [91]. Über die Adressierung hinaus kennt XPath noch weitere Ausdrücke, beispielsweise String-Funktionen oder boolsche Ausdrücke, die jedoch in der vorliegenden Arbeit nicht genauer diskutiert werden.

Das primäre syntaktische Element von XPath ist *Expression* (Ausdruck). Das Ergebnis eines Ausdrucks (oder genauer: das Ergebnis der Auswertung eines Ausdrucks) ist dabei von einem der Typen Knotenmenge, Wahrheitswert, Zahl oder String.

Der wichtigste Ausdruck ist der Location Path. Durch einen Location Path können Knoten adressiert werden, das heißt durch einen Location Path wird, ausgehend von einem Kontextknoten, eine Knotenmenge definiert.

In XPath existieren so genannte *relative* und *absolute* Location Paths. Der Unterschied ist, dass ein relativer Location Path immer entsprechend eines Kontextes ausgewertet wird, so dass ein relativer Location Path in unterschiedlichen Kontexten auch unterschiedliche Ergebnisse liefert. Ein absoluter Location Path dagegen

ersetzt immer den aktuellen Kontext durch den *root*-Knoten. Ein absoluter Location Path wird demnach als relativer Location Path im Kontext des *root*-Knotens ausgewertet. Im Anhang C befindet sich ein Auszug aus dem XPath Standard mit einigen Beispielen für mögliche Location Paths.

Jeder Location Path besteht aus einem oder mehreren Location Steps, die durch '/' getrennt sind. Steht '/' allein bzw. am Anfang eines Location Steps, so wird dadurch der *root*-Knoten ausgewählt. Absolute Location Paths werden demnach dargestellt als relative Location Paths, denen '/' vorangestellt ist.

Jeder Location Step definiert, ausgehend von einem einzelnen Kontextknoten, eine Knotenmenge. Die Ausgabe-Knotenmenge eines Location Steps ist die Eingabe-Kontextmenge des nachfolgenden Location Steps. Location Steps werden in Textreihenfolge von links nach rechts verarbeitet. Entsprechend des Standards wird dabei der aktuelle Location Step auf jeden Knoten der aktuellen Kontextmenge einzeln angewendet, das heißt, ein Location Step wird immer auf genau einen Kontextknoten zur Zeit angewendet. Alle dabei entstehenden Ergebnis-Kontextmengen werden vereinigt und bilden die Kontextmenge für den nachfolgenden Location Step. Die Vereinigung der Kontextmengen des letzten Location Steps ist das Ergebnis des Location Paths.

Ein Location Step `axis::nodetest[p1][...][pn]` besteht grundsätzlich aus drei Teilen:

- einer XPath-Achse
- einem Knotentest
- einem oder mehreren Prädikaten

Bei der Auswertung eines Location Steps werden zunächst, ausgehend vom aktuellen Kontextknoten, diejenigen Knoten ermittelt, die über die Achse erreicht werden können und dem Knotentest entsprechen. Durch den Knotentest können die erlaubten Namen und Typen (entsprechend des XPath-Datenmodells, vgl. Tabelle 2.1) der Knoten der Ergebnismenge definiert werden. Die so gebildete Knotenmenge wird zuletzt mit den durch die Prädikate definierten Filtern iterativ eingeschränkt, das heißt ein Prädikat wird auf die Ergebnis-Kontextmenge des vorangegangenen Prädikats angewendet. Prädikate können neben bestimmten Eigenschaften der Knoten auch kontextspezifische Eigenschaften prüfen, etwa eine

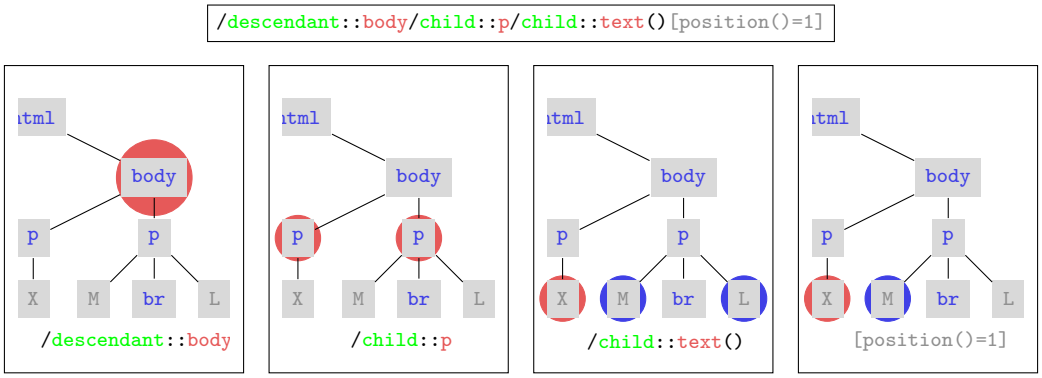


Abbildung 2.4: Beispiel 1: Auswertung eines Location Path

bestimmte Position eines Knotens in der aktuellen Kontextmenge. Da sich die Kontextmenge (und damit der Kontext) durch die Anwendung eines Prädikats ändern kann, kommutieren Prädikate im Allgemeinen nicht.

Im Standard werden auch einige abgekürzte Schreibweisen für bestimmte Location Steps definiert. In dieser Arbeit wird immer die ausführliche Schreibweise verwendet.

Die Abbildung 2.4 zeigt beispielhaft die Auswertung eines Location Path in dem bereits bekannten Beispieldokument aus Quellcode 2.1. Die Auswertung der drei Location Steps ist hier in vier Schritten dargestellt, indem der letzte Location Step, der ein Prädikat enthält, in zwei Schritten abgebildet ist. In jedem Schritt sind diejenigen Knoten des Dokuments mit einem farbigen Kreis hinterlegt, welche sich nach dem jeweiligen Schritt in der Kontextmenge befinden. Unterschiedliche Farben zeigen an, dass Knoten durch die Auswertung des gleichen Location Steps mit unterschiedlichen Kontextknoten ausgewählt wurden. Das Ergebnis des Location Paths ist die Knotenmenge {'X','M'}.

An diesem Beispiel wird deutlich, dass jeder Knoten aus einer Kontextmenge einzeln als Kontextknoten verwendet wird. Deswegen wird im dritten Schritt der Location Step

```
child::text()
```

zunächst auf den linken Knoten `p` angewendet, wodurch alle Kinder des Kontextknotens, die Textknoten sind, ausgewählt werden. Im vierten Schritt wird durch das Prädikat

```
[position()=1]
```

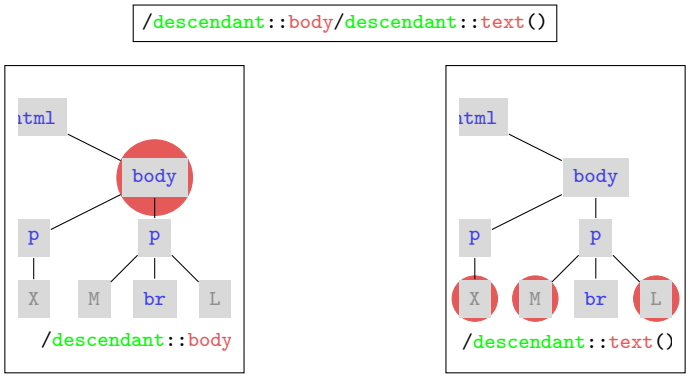


Abbildung 2.5: Beispiel 2: Auswertung eines Location Path

das erste Element ('X') der (ohnehin 1-elementigen) entstehenden neuen Kontextmenge ausgewählt. Danach wird der dritte Location Step auf den rechten Knoten `p` angewendet und wiederum das erste Element ('M') ausgewählt. Das Gesamtergebnis des letzten Location Steps sind somit zwei Elemente, für die in ihrer jeweiligen Kontextmenge das Prädikat `[position()=1]` gilt.

Demnach gilt beispielsweise, dass im hier betrachteten Beispieldokument der Location Path

```
/descendant::body/child::p/child::text()
```

(bei dem das Ergebnis die Vereiniung zweier Knotenmengen ist, vgl. Abbildung 2.4) auch durch den Location Path

```
/descendant::body/descendant::text()
```

ersetzt werden könnte (bei dem das Ergbnis eine einzelne Knotenmenge ist, vgl. Abbildung 2.5), die Ergebnisse sind identisch. Wird jedoch an den letzten Location Step wiederum das Prädikat `[position()=1]` angefügt, so unterscheiden sich die Ergebnisse: Der zweite Location Path liefert dann nur noch einen Knoten (hier: 'X').

2.4 XSLT

Die *Extensible Stylesheet Language—Transformation* (XSLT) [85, 92] ist eine deklarative Sprache, die auf XML basiert. XSLT verwendet das XPath-Datenmodell,

betrachtet ein XML-Dokument also auch als Baum von Knoten. Im Folgenden ist mit „Baum“ oder „Dokumentbaum“ immer ein Dokument in der Repräsentation nach dem XPath-Datenmodell gemeint.

Mittels XSLT kann eine Transformation eines XML-Dokumentbaums in einen anderen Baum beschrieben werden. Genauer gesagt, es wird die Transformation von Knoten des Eingabebaums in Knoten des Ausgabebaums beschrieben, wobei ein Knoten des Eingabebaums immer in einen oder mehrere Knoten des Ausgabebaums transformiert wird. Dabei ist der Begriff der „Transformation“ eigentlich irreführend: Der Eingabebaum wird gerade nicht transformiert sondern bleibt unverändert. Stattdessen wird mittels XSLT auf Basis des Eingabebaums ein neuer Ausgabebaum erstellt. Der Ergebnis-Baum repräsentiert dabei im Allgemeinen kein XML-Dokument; die Ausgabe kann beispielsweise auch ein Baum sein, der nur aus Textknoten besteht, eine reine Textausgabe also.

Transformationen mittels XSLT werden in verschiedenen Szenarien eingesetzt, beispielsweise um Dokumente einer XML-Sprache in eine andere Sprache zu konvertieren, etwa im Kontext verschiedener Formate zur Speicherung von Texten (DOCX, ODP). Ein weiteres bekanntes Beispiel für die Anwendung ist auch *Osmarender* [12], womit Straßenkartendaten (OSM) in Vektorgrafiken (SVG) transformiert werden können. Weiterhin wird XSLT häufig als Abstraktionsschicht zwischen Modell und Sicht eingesetzt, indem mit verschiedenen XSLT-Transformationen verschiedene Sichten auf den gleichen XML-Datenbestand erzeugt werden.

Die Ausdrucksstärke von XSLT ist sehr groß. So existiert mit Schematron [34] ein Framework auf Basis von XSLT, mit dessen Hilfe sich Verifikationsregeln für XML-Dokumente definieren lassen. Die Idee ist, ein XML-Dokument in den Fehlerbericht desselben Dokuments zu transformieren. Dadurch steht für die Verifikation im Prinzip die XSLT-Ausdrucksstärke und -Vielfalt bereit, so dass mittels Schematron/XSLT auch Eigenschaften eines XML-Dokuments geprüft werden können, die mit einem Validator für XML Schema [91] grundsätzlich nicht geprüft werden können.

Der Mechanismus, um einen Eingabebaum mittels XSLT zu transformieren, ist einer Turingmaschine nicht unähnlich: Das Eingabedokument wird an einer bestimmten Stelle gelesen (aktuelle Knotenliste), und Transformationsanweisungen (Templates) bestimmen anhand des gelesenen Wertes (des XML-Knotens), welcher Wert (Templateinhalt) in das Ausgabedokument geschrieben wird und welche Stelle im Dokument als nächstes gelesen werden muss. An dieser Stelle sei angemerkt,

```
1 <xsl:stylesheet version="1.0" xmlns:xsl="...">
2 <xsl:template match="/">
3 Der Inhalt ist:
4 <xsl:apply-templates select="descendant::body/descendant::text()" />
5 Der Titel ist:
6 <xsl:apply-templates select="descendant::head/descendant::text()" />
7 </xsl:template>
8
9 <xsl:template match="child::p/child::text()">
10 Start: <xsl:value-of select="." /> Ende
11 </xsl:template>
12
13 </xsl:stylesheet>
```

Quellcode 2.2: XSLT Stylesheet

dass XSLT und das übliche Turingmodell nicht nur ähnlich sind, sondern XSLT `1.0` und XSLT `2.0` auch Turing-vollständig sind [49, 63].

Ein einzelnes XSLT-Dokument entspricht einer Transformationsanweisung für einen XML-Dokumentbaum und heißt Stylesheet (Beispielquellcode 2.2). Ein Stylesheet besteht aus einer Menge von Template Rules der Form

<xsl:template match="...">TEMPLATE</xsl:template>,

wobei jede Template Rule einer Transformationsregel entspricht. Eine Template Rule besteht aus einem Pattern (`match`) und einem Template (`TEMPLATE`), wobei das Pattern definiert, auf welche Knoten das Template angewendet ('instanziiert') werden muss und das Template den Inhalt der Transformation darstellt, der ggf. in das Ausgabedokument kopiert wird. Grundsätzlich geschieht die Verarbeitung der Template Rules eines Stylesheets immer bezüglich einer s.g. *current node list* (aktuelle Knotenliste), welche die aktuell zu betrachtenden Knoten enthält.

Anders als bei XPath müssen bei XSLT keine Ausdrücke ausgewertet werden, vielmehr definiert XSLT ein Prozessmodell, nach dem die Verarbeitung eines Eingabedokuments erfolgt. Der Quellcode 2.3 zeigt das im Standard definierte Prozessmodell in Pseudocode. Die drei wesentlichen Bestandteile sind:

1. *Selection*, die Auswahl einer neuen aktuellen Knotenliste mittels eines XPath-Ausdrucks.

2. *Instantiation*, die Instanziierung eines Templates für einen bestimmten Knoten.

I. Lege leeren Zielbaum und leere aktuelle Knotenliste an. Füge den root Knoten des Quelldokuments in die aktuelle Knotenliste ein.

II. Für jeden Knoten in der aktuellen Knotenliste:
   1. Finde die Template Rule, deren Pattern am besten zum Knoten passt (*Matching*)
   2. Instanziiere das Template der gewählten Template Rule (*Instantiation*)
   3. Falls bei der Instanziierung eine weitere aktuelle Knotenliste ausgewählt worden sein sollte (*Selection* mit `apply-templates`, `for-each`), wende rekursiv Schritt II auf die neue aktuelle Knotenliste an.

Quellcode 2.3: XSLT-Prozessmodell

3. *Matching*, die Auswahl des passenden Templates für jeden Knoten der aktuellen Knotenliste.

2.4.1 Selection

Eine neue aktuelle Knotenliste kann nur durch bestimmte XSLT-Konstrukte erzeugt werden, etwa durch

```
<xsl:for-each select="...">...</xsl:for-each>
```

und

```
<xsl:apply-templates select="..."/>
```

wie im Quellcode 2.2. Im `select`-Attribut wird dazu ein XPath-Ausdruck angegeben, der zu einer Knotenmenge evaluiert wird. Diese Knotenmenge bildet die neue aktuelle Knotenliste.

2.4.2 Matching

Das Matching oder Pattern Matching bezeichnet den Verarbeitungsschritt, bei dem für einen aktuellen Knoten ein passendes bzw. das "am besten" passende [85, Abschnitt 5.5] Template gefunden werden muss. Dazu wird für jedes Template geprüft, ob das Pattern im `match`-Attribut in

```
<xsl:template match="Pattern">
```

zum aktuellen Knoten passt. Das Pattern besteht aus einem XPath-Ausdruck, der zu einer Knotenmenge evaluiert werden kann und zudem von den XPath-Achsen nur die *child*-Achse sowie die *attribute*-Achse enthalten darf.

Dieser XPath-Ausdruck erzeugt also, angewendet auf einen Kontextknoten, eine Knotenmenge. Ein Template Pattern passt genau dann zu einem aktuellen Knoten, wenn im XML-Dokument ein Kontextknoten existiert, so dass der aktuelle Knoten gerade in der mit dem Kontextknoten und dem XPath-Ausdruck erzeugten Knotenmenge liegt.

Die Überprüfung eines Pattern für einen aktuellen Knoten ist also prinzipiell ein aufwendiger Prozess. Da jedoch im Pattern wie erwähnt nur die *child*-Achse sowie die *attribute*-Achse verwendet werden dürfen, kommen typischerweise nur wenige Knoten im Dokument als Kontextknoten in Frage, nämlich nur der aktuelle Knoten selbst oder ein Vorfahre.

Zudem enthält ein Pattern oft sehr einfache XPath-Ausdrücke, wie in Zeile 9 in Quellcode 2.2 zu sehen ist. Der dort verwendete Match-Ausdruck ist:

```
child::p/child::text()
```

Soll nun beispielsweise für einen Knoten A geprüft werden, ob der Matching-Ausdruck passt, so muss im Prinzip geprüft werden, ob ein Knoten K existiert, so dass A in der Knotenmenge liegt, die sich aus dem XPath-Ausdruck mit dem Kontextknoten K ergibt. Da aber nur einfache XPath-Ausdrücke erlaubt sind, lässt sich diese Überprüfung häufig vereinfachen: Hier genügt es etwa zu prüfen, ob A ein Textknoten ist und ob der Elternknoten von A den Namen `p` hat. Ist das der Fall, so ist der Elternknoten von `p` (also der Elternknoten vom Elternknoten von A) der gesuchte Knoten K.

2.4.3 Instanziierung

Die Instanziierung eines Templates, das heißt die Verarbeitung der Knoten des Templateinhalts `TEMPLATE` in

```
<xsl:template match="...">TEMPLATE</template>
```

entsprechend des aktuellen Knotens, wird in zwei Fälle unterschieden: Ist ein Knoten des Templateinhalts keine XSL-Anweisung (beispielsweise Zeile 3 in Quellcode

2.2), so wird der Knoten einfach an das Ende des Resultatbaums kopiert. Ist der Knoten jedoch eine XSL-Anweisung (Zeile 4 in Quellcode 2.2), so muss die Anweisung evaluiert werden. Statt der XSL-Anweisung werden dann die Ergebnisse der Evaluierung an das Ende des Resultatbaums geschrieben. Unter den XSL-Anweisungen sind viele, die eigentlich nur syntaktischen Zucker darstellen bzw. für einfache Transformationen nicht notwendig sind. Die Menge der für die Transformation wichtigen oder sogar notwendigen Anweisungen dagegen ist recht klein.

`apply-templates`

Ohne die XSL-Anweisung

```
<xsl:apply-templates select="..."/>
```

kann keine praktisch relevante Transformation ausgeführt werden. Wie bereits erwähnt, ist am Anfang des Transformationsprozesses nur der *root*-Knoten in der aktuellen Knotenliste. Wird die `apply-templates`-Anweisung als Teil eines Templateinhalts für einen aktuellen Knoten instanziiert, so erzeugt die Anweisung mit dem aktuellen Knoten als Kontextknoten und dem XPath-Ausdruck im `select`-Attribut eine neue aktuelle Knotenliste, die dann rekursiv (also genau wie die ursprüngliche Liste, vgl. Quellcode 2.3) weiterverarbeitet wird.

`for-each`

Wie auch die `apply-templates`-Anweisung erzeugt die Anweisung

```
<xsl:for-each select="...">TEMPLATE</xsl:for-each>
```

eine neue Knotenliste, die ebenfalls sofort (jedoch nicht rekursiv) ausgewertet wird. Anders als die `apply-templates`-Anweisung gibt die `for-each`-Anweisung nämlich gleich das zu instanziierende Template (`TEMPLATE`) mit an. Das kann natürlich unter Umständen ein Vorteil sein, etwa wenn für eine bestimmte Knotenmenge das gleiche bereits bekannte Template instanziiert werden soll, das heißt auf den Matching-Verarbeitungsschritt verzichtet werden kann. Zwar können auch `for-each`-Anweisungen verschachtelt werden, aber da das zu instanziierende Template immer innerhalb der `for-each`-Anweisung definiert werden muss, werden keine Templates wiederverwendet, es tritt also keine Rekursion auf. Dies ist aber einer der Hauptvorzüge von XSLT, der die Stylesheets nicht nur übersichtlicher macht,

sondern auch mächtiger in ihrer Ausdrucksstärke. Nicht-terminierende Konstruktionen etwa wie das (zugegebenermaßen nicht sinnvolle) nachfolgende Beispiel sind ohne `apply-templates` nicht möglich:

```
1 <xsl:template match="html">
2   <xsl:apply-templates select="self::*"/>
3 </xsl:template>
```

`value-of, copy-of, copy`

Für praktisch relevante Transformationen werden natürlich auch Anweisungen benötigt, mit denen Inhalte des Eingabebaums in den Ausgabebaum geschrieben werden können. Wird die Anweisung

```
<xsl:value-of select="..."/>
```

für einen bestimmten Knoten instanziiert, so wird an die entsprechende Stelle im Ausgabebaum die Stringrepräsentation [84, Abschnitt 5] der durch den XPath-Ausdruck im `select`-Attribut definierten Knotenmenge kopiert. Im Unterschied dazu kann mit der Anweisung

```
<xsl:copy-of select="..."/>
```

die durch den XPath-Ausdruck im `select`-Attribut definierte Knotenmenge bzw. können die dadurch definierten Teilbäume (nicht die Stringrepräsentation!) in den Ausgabebaum kopiert werden. Mit der Anweisung

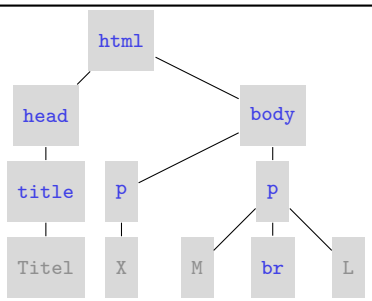
```
<xsl:copy>TEMPLATE</xsl:copy>
```

schließlich kann der aktuelle Knoten selbst kopiert werden, wobei das Template `TEMPLATE` auf dessen Kinder und Attribute angewendet wird.

2.4.4 Beispiel

Die vorangegangenen Erläuterungen sollen nun mit einem einfachen Beispiel illustriert werden. Dazu wird das Stylesheet aus Quellcode 2.2 auf das Beispieldokument in Quellcode 2.1 angewendet. Die Eingabe als Baum und die Ausgabe der Transformation sind in Quellcode 2.4 zu sehen.

Diese Beispieltransformation verdeutlicht gut den XSLT-Prozess: Zunächst wird (vgl. Quellcode 2.3) der *root*-Knoten in die aktuelle Knotenliste eingefügt und




```
1 Der Inhalt ist:
2
3 Start: X Ende
4 Start: M Ende
5 Start: L Ende
6
7 Der Titel ist:
8
9 Start: Titel Ende
```

Quellcode 2.4: Eingabe (links) und Ergebnis (rechts) der Beispiel-Transformation

ein passendes Template gesucht und gefunden ('/' wählt den *root*-Knoten eines Dokuments aus, Abschnitt 2.3):

```
<xsl:template match="/">
```

Anschließend wird das Template instanziiert, das heißt, es wird zunächst der Textinhalt `Der Inhalt ist:` in das Ergebnis kopiert und dann

```
<xsl:apply-templates select="descendant::body/descendant::text()"/>
```

ausgewertet. Hier beginnt die Rekursion: die Anweisung erzeugt eine neue aktuelle Knotenliste, welche alle Nachfolger-Textknoten des `body`-Elements enthält:

'X', 'M', 'L'

Die alte Knotenliste wird gespeichert. Für alle Knoten der neuen aktuellen Knotenliste werden wiederum passende Templates gesucht. Da alle Textknoten in der aktuellen Knotenliste (und überhaupt alle Textknoten im Dokument) Kindknoten von `p`-Elementen sind, wird das Template

```
<xsl:template match="child::p/text()">
```

für jeden der drei Knoten ausgewählt und instanziiert. Es wird also zunächst (unter Verwendung der `value-of`-Anweisung)

```
Start: X Ende
Start: M Ende
Start: L Ende
```

in die Ausgabe eingefügt und dann werden die entsprechenden Knoten aus der aktuellen Knotenliste entfernt. Da die aktuelle Knotenliste nun leer ist, ist die Rekursion beendet, da das Template keine XSL-Elemente enthält, die weitere aktuelle Knotenlisten erzeugen. Es wird also das Template, welches die Rekursion „aufrief“, mit der alten aktuellen Knotenliste weiterverarbeitet, das heißt es wird `Der Titel ist:` in die Ausgabe eingefügt und erneut eine neue aktuelle Knotenliste mit

```
<xsl:apply-templates select="descendant::head/descendant::text()"/>
```

erzeugt. Diese enthält analog zur ersten Rekursion nur den Textknoten ’`Titel`’. Auch für diesen Knoten wird ein passendes Template gesucht und wiederum

```
<xsl:template match="child::p/text()">
```

ausgewählt. Nun wird `Start: Titel Ende` in die Ausgabe kopiert und der Knoten wird aus der aktuellen Knotenliste entfernt. Es wird keine neue aktuelle Knotenliste erzeugt, die Rekursion ist beendet und die alte aktuelle Knotenliste, die nur den Wurzelknoten enthält, wird wiederhergestellt. Auch für diesen Knoten ist die Instanziierung des Templates abgeschlossen und endet mit der Entfernung des Knotens aus der aktuellen Knotenliste. Die Knotenliste ist leer, der Prozess ist beendet.

An der hier verwendeten Beispieltransformation lässt sich auch gut verdeutlichen, dass das Pattern lediglich bestimmt, auf welche Knoten der aktuellen Knotenliste es angewendet wird, also nicht etwa aktiv Knoten aus dem Eingabe-XML-Dokument in die Knotenliste einfügt. Als Beispiel verwenden wir im Pattern in Zeile 4 des Quellcodes 2.2 nun statt des ursprünglichen Ausdrucks den Ausdruck

```
descendant::body/descendant::text()[position()=1].
```

Dann ist das Ergebnis der Transformation:

```
1 Der Inhalt ist:
2
3 Start: X Ende
4
5 Der Titel ist:
6
7 Start: Titel Ende
```

Offensichtlich wurde das Pattern des zweiten Templates nicht verändert, jedoch werden nicht mehr alle Knoten für die aktuelle Knotenliste ausgewählt.

KAPITEL 3

Allgemeine Berechungen auf GPUs

GPUs sind seit vielen Jahren Bestandteil beinahe aller *Personal Computer* (PCs). Der Begriff „GPU“ wurde erst 1999 von NVIDIA geprägt (vgl. [60]), aber spezielle Chips zur Beschleunigung von Grafikberechnungen gibt es seit den 1980er Jahren. Im Folgenden wird undifferenziert der Begriff „GPU“ benutzt. Wie der Name bereits sagt, sind diese Recheneinheiten zunächst dafür entwickelt worden, bestimmte Berechnungen im Zusammenhang mit grafischer Verarbeitung zu übernehmen. Da für diese grafischen Berechnungen ein breiter Markt existiert, wurden in der Vergangenheit GPUs weiterentwickelt und ihre Rechenleistung erheblich gesteigert.

Seit einigen Jahren wird nun auch vermehrt die Berechnung allgemeiner Probleme mittels GPUs durchgeführt und es existieren sogar GPUs für PCs, die dediziert für die Berechnung allgemeiner Probleme konzipiert wurden und über keinen Monitoranschluss verfügen. Inzwischen sind auch spezielle Frameworks zur Programmierung von GPUs verfügbar.

Allerdings eignen sich GPUs aufgrund ihrer ursprünglichen Bestimmung nicht für alle generischen Probleme gleichermaßen gut. Im folgenden Abschnitt 3.1 wird deshalb die Geschichte und Architektur von GPUs detaillierter beschrieben. Im darauf folgenden Abschnitt 3.2 wird CUDA, Architektur und Programmierframework für GPUs der Firma NVIDIA, vorgestellt.

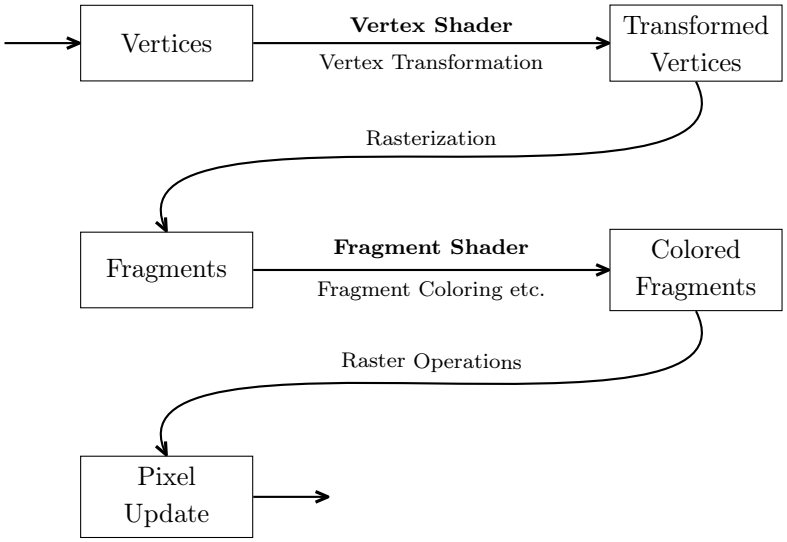


Abbildung 3.1: Die Rendering-Pipeline

3.1 Geschichte und Architektur

Die Motivation zur Entwicklung moderner GPUs war die Beschleunigung von 3D-Anwendungen wie Spielen [14], aber auch Anwendungen im Bereich von *computer-aided Design* (CAD) und Ähnliches. Die notwendigen Schritte, um Objekte aus einem 3-dimensionalen virtuellen Raum auf dem (2-dimensionalen) Bildschirm des Anwenders darzustellen, werden zusammenfassend als Rendering-Pipeline bezeichnet. Abbildung 3.1 zeigt eine vereinfachte Rendering-Pipeline (für detaillierte Informationen sei auf [31] verwiesen).

Die Eingabe der Rendering-Pipeline sind so genannte Vertices. Vertices sind die Koordinaten der Eckpunkte einfacher 3-dimensionaler Objekte (etwa Polygone), aus denen sich wiederum die komplexen Objekte des virtuellen Raumes (zum Beispiel ein Haus in einem Spiel) zusammensetzen. Daneben können aber noch weitere Informationen, etwa bezüglich der Beleuchtung des Objekts oder die Textur betreffend, zu einem Vertex gehören.

In einem ersten Schritt, der Vertex-Transformation, müssen 3-dimensionale Transformationen auf den Vertices ausgeführt werden, beispielsweise um die Koordinaten der Vertices der Position und dem Blickwinkel des virtuellen Beobachters anzupassen. Im zweiten Schritt (Rasterization) werden die komplexen 3-dimensionalen Objekte zunächst in einfache 3-dimensionale Objekte, üblicherweise

Dreiecke, zerlegt. Anschließend werden diese einfachen 3-dimensionalen Objekte in die Ebene projiziert und rasterisiert.

Das Resultat sind so genannte Fragments, das heißt im Prinzip, die durch die Rasterisierung entstandenen und zu dem Objekt gehörende Bildpunkte. Fragments bestehen daher aus einer 2D-Koordinate. Einem Fragment werden noch weitere Informationen hinzugefügt, beispielsweise wird im nächsten Schritt, dem Fragment Coloring, die Farbe der Bildpunkte berechnet und dem Fragment hinzugefügt.

Die Farbe wird meist abhängig von dem ursprünglichen Objekt, zu dem das Fragment gehört, berechnet und kann sich beispielsweise aus der Textur des Objekts und durch Lichteffekte, Schatten und Ähnliches ergeben. Die so mit Informationen angereicherten Fragments werden dann im letzten Schritt (Raster Operations) verwendet, um für jedes Fragment auf dem Bildschirm einen entsprechenden Farbwert zu berechnen.

Während in den Anfängen der 3D-Programmierung alle diese Schritte vollständig auf der CPU berechnet werden mussten, wurden im Laufe der Zeit die GPUs so erweitert, dass immer mehr dieser Schritte von der GPU ausgeführt wurden. Die Recheneinheiten auf der GPU, die die notwendigen Berechnungen der Transformation ausführen, werden Vertex-Shader (vgl. Abbildung 3.1) genannt. Die Einheiten zum Berechnen des Fragment Coloring werden entsprechend Fragment-Shader genannt. Die Vertex-Shader und Fragment-Shader werden zusammengefasst auch Shader genannt.

In der Rendering-Pipeline können viele Operationen parallel ausgeführt werden. Die Transformation eines Vertex ist beispielsweise unabhängig von der Transformation eines anderen Vertex. Gleiches gilt für die Berechnung der Farbe eines Fragments. Auch diese ist vollkommen unabhängig von der Berechnung der Farbe eines anderen Bildpunktes. Um die Leistung der GPUs zu erhöhen, wurde deswegen damit begonnen, nicht nur einen, sondern mehrere gleichartige Vertex-Shader bzw. Fragment-Shader auf einer GPU bereitzustellen. Durch diese Parallelisierung der Hardware erhöhte sich die Rechenleistung der GPUs immer mehr, so dass mittlerweile aktuelle GPUs insbesondere im Bezug auf *Floating Point Operations per second* (FLOPS) eine höhere Leistung als aktuelle CPUs haben (siehe Abbildung 3.2).

Die durch den hohen Grad der Parallelität erzielte große Leistung der GPUs wird inzwischen auch für andere Berechnungen genutzt. Zu dieser Entwicklung trägt weiterhin bei, dass GPUs eine vergleichsweise günstige Hardware für den

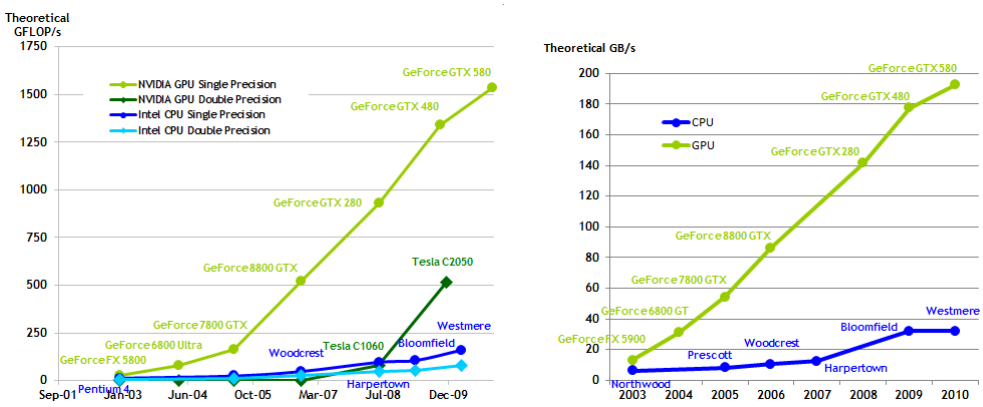


Abbildung 3.2: Vergleich FLOPS GPU vs. CPU, Quelle [62]

Massenmarkt sind und daher das Verhältnis von Rechenleistung zum finanziellen Einsatz sehr günstig ist.

Zunächst konnten die Vertex-Shader und Fragment-Shader allerdings nicht frei programmiert werden. 2001 erschienen die ersten GPUs, die die freie Programmierung der Vertex-Shader erlaubten. Beispiele für diese GPUs sind die GeForce3 von NVIDIA oder die Radeon8500 der Firma ATI. In den folgenden Jahren wurden, beginnend mit GeForceFX von NVIDIA und Radeon9700 von ATI, GPUs verfügbar, bei denen zusätzlich auch die Fragment-Shader frei programmierbar waren. Gleichzeitig entstanden Frameworks, Programmiersprachen etc. zur Programmierung der Vertex-Shader bzw. Fragment-Shader. Für die am meisten verbreiteten Frameworks zur 3D-Programmierung OpenGL [11] und DirectX [6] wurden entsprechende Erweiterungen entwickelt ("ARB vertex program" [2] bzw. "ARB fragment program" [1] oder GLSL [7] für OpenGL, HLSL [8] für DirectX). Daneben entstanden aber auch speziell für die Shader-Programmierung entwickelte Toolkits wie das mit HLSL verwandte Cg von NVIDIA [4]. Entsprechend wurden in diesen Jahren auch die ersten Arbeiten veröffentlicht, die die freie Programmierung der Shader nutzten, um allgemeine Berechnungen bzw. Algorithmen auf GPUs auszuführen. Dieser Einsatz von GPUs für allgemeine Berechnungen wird auch *General Purpose Computations on GPUs* (GPGPU) genannt.

Da die Shader aber für die Nutzung innerhalb der Rendering-Pipeline konzipiert wurden, gab es trotz der freien Programmierbarkeit der Shader einige durch die Hardware bedingte Beschränkungen. Insbesondere war zunächst kein wahlfreier Zugriff auf den Arbeitsspeicher (Scattered-Writing) möglich:

Die GPU speichert zwar die Daten der Vertices und Fragments in ihrem Arbeitsspeicher. Die Vertex-Shader waren jedoch so konzipiert, dass die Eingabe für ein Shader-Programm immer die Koordinate eines Vertex und zusätzliche Informationen (wie etwa Texturen) sind und die Ausgabe entsprechend wieder die Koordinate des Vertex usw. Dafür war ein freier Zugriff auf den Arbeitsspeicher nicht nötig. Fragment-Shader konnten zwar wahlfrei lesend auf Texturen zugreifen, weil die Farbe eines Fragments von der Textur des zugehörigen Objekts abhängen kann. Sie konnten aber ebenfalls weder auf den allgemeinen Arbeitsspeicher noch auf die Texturen schreibend zugreifen.

Mit der nächsten Generation der GPUs ab 2006 wurde das Konzept der Unified Shader eingeführt. Das bedeutet, dass es keine dedizierten Recheneinheiten für Vertex-Shader und Fragment-Shader mehr gibt, sondern nur noch einen Typ Recheneinheiten, welcher nach Bedarf als Vertex-Shader oder Fragment-Shader genutzt werden kann. Außerdem waren mit der Architektur dieser GPUs Scattered-Writings möglich. Die ersten GPUs dieser Generation waren die der GeForce8-Serie von NVIDIA und die der RadeonHD2000-Serie von ATI. Diese Generation von GPUs erleichterte die Programmierung der GPUs erheblich und führte dazu, dass inzwischen viele leistungsfähige Algorithmen bzw. Bibliotheken für GPUs entwickelt wurden. Zudem ist durch die Einführung der Unified Shader und des Scattered-Writing die Programmierung unabhängig von der ursprünglichen Rendering-Pipeline, so dass auch nicht mehr zwischen Vertex-Programmen und Fragment-Programmen unterschieden werden muss.

NVIDIA und ATI entwickelten zunächst eigene Toolkits/Frameworks zur einfachen Programmierung dieser neuen Generation von GPUs. Von ATI stammt ATI Stream [3], und NVIDIA entwickelte CUDA [5], das auch im Rahmen dieser Arbeit verwendet wird. Später stellte die Khronos Gruppe, die auch für die OpenGL-Spezifikation zuständig ist, OpenCL [10] vor. OpenCL ist eine Spezifikation eines allgemeinen Frameworks zur Programmierung paralleler Architekturen (neben GPUs auch übliche Mehrkern-Prozessoren). Es wurde allerdings im Hinblick auf GPUs entworfen. Sowohl von NVIDIA als auch von ATI gibt es mittlerweile Implementierungen von OpenCL. Während ATI die Entwicklung ihres eigenen Frameworks zu Gunsten von OpenCL aufgegeben hat, entwickelt NVIDIA CUDA nach wie vor aktiv weiter und CUDA ist das von NVIDIA empfohlene Framework.

3.2 CUDA

```
1  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
2  {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      int j = blockIdx.y * blockDim.y + threadIdx.y;
5      if ( i < N && j < N )
6      {
7          C[i][j] = A[i][j] + B[i][j];
8      }
9  }
10 int main()
11 {
12     ...
13     // Kernel invocation
14     dim3 threadsPerBlock(16, 16);
15     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
16     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
17     cudaThreadSynchronize();
18     ...
19 }
```

Quellcode 3.1: Beispiel CUDA Programm

Die *Compute Unified Device Architecture* (CUDA) [5] ist eine spezielle parallele Rechnerarchitektur für GPUs und ein Programmiermodell von NVIDIA. Mit CUDA ist es möglich, in der Programmiersprache `C/C++` Prozeduren zu definieren, die von der GPU ausgeführt werden, so genannte *kernel* (Kernel). Der Start eines Kernels, ein so genannter *Kernel Launch*, erfolgt dabei im Allgemeinen asynchron zur Ausführung des Codes auf der CPU, das heißt, der Aufruf eines Kernels ist nicht-blockierend. Mit speziellen Anweisungen wie `cudaThreadSynchronize()` (Zeile 17 im Quellcode 3.1) kann die Ausführung synchronisiert werden, das heißt, `cudaThreadSynchronize()` blockiert, bis der Kernel vollständig verarbeitet ist.

In der Regel wird immer nur ein Kernel zur Zeit auf einer GPU ausgeführt. Einige GPUs der neuesten Generation von NVIDIA unterstützen zwar auch die Ausführung mehrerer Kernel, jedoch nur in sehr beschränktem Umfang (maximal 16 Kernel) und auch nur unter sehr speziellen Bedingungen. Kurz gesagt, es ist derzeit nicht möglich, etwa zu einem bereits arbeitenden Kernel einen weiteren Kernel hinzuzufügen.

Für jeden Aufruf eines Kernels wird eine Anzahl *threads* (Threads) definiert, wobei jeder Thread den Kernel genau einmal ausführt. In Anlehnung an das be-

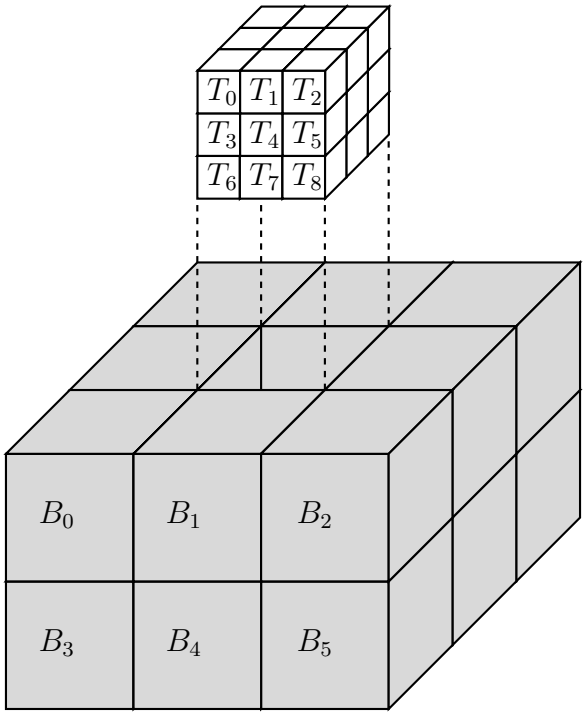


Abbildung 3.3: CUDA Thread-Hierarchie

kannte *Single Instruction Multiple Data*-Prinzip wird diese Vorgehensweise bei CUDA *Single Instruction Multiple Threads* (SIMT) genannt.

Die Threads werden für die Verarbeitung in virtuelle *blocks* (CUDA-Blöcke oder einfach Blöcke) gruppiert, die Blöcke wiederum sind in einem *grid* (Grid) angeordnet. Die Abbildung 3.3 verdeutlicht diese so genannte Thread-Hierarchie: 3-dimensionale Blöcke von Threads werden in einem 3-dimensionalen Grid angeordnet. Im Quellcode 3.1 ist dieser Ansatz in den Zeilen 14, 15 und 16 zu erkennen, in denen zunächst zwei Strukturen vom Typ `dim3`, welche 3-dimensionale Werte darstellen, angelegt werden und dann beim Aufruf des Kernels übergeben werden:

```
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Für jeden Thread kann also durch seine virtuelle Position im Block eine im Block eindeutige Thread-ID berechnet werden, und somit kann auch mittels der Position im Grid eine global eindeutige Thread-ID bestimmt werden. Jeder Thread hat zur Laufzeit Zugriff auf seine virtuellen Positionen im Grid und im Block, so dass auch jeder Thread selbst seine Thread-ID bestimmen kann. Somit ist es beispielsweise

möglich, in Abhängigkeit von der Thread-ID (etwa mit der Thread-ID als Array-Index) unterschiedliche Daten verarbeiten zu lassen.

NVIDIAs GPUs basieren auf der SIMT-Architektur, das heißt, alle Threads eines Kernels führen im Prinzip den gleichen Kernel aus. Threads sind in Blöcken organisiert. Die Threads eines Blocks werden weiterhin unterteilt in s.g. *warps* (Warps). Ein Warp ist dabei eine Gruppe von 32 Threads. Das eigentliche SIMD-Prinzip, also das gleichzeitige Verarbeiten der selben Instruktion durch mehrere parallele Einheiten, wird nur auf Warps angewendet; unterschiedliche Warps werden unabhängig voneinander verarbeitet. Alle Threads eines Blocks können auf einen gemeinsamen sehr schnellen Shared Memory zugreifen. Zudem stehen Threads eines Blocks bestimmte leistungsfähige Mechanismen zur Synchronisation (`__threadfence()`, `__syncthreads()` etc.) bzw. Abstimmung zur Verfügung (`__syncthreads_count()`, `__syncthreads_and()` etc.), welche den Threads verschiedener Blöcke nicht zur Verfügung stehen.

Trotz des verwendeten SIMT-Prinzips können mit CUDA auch bedingte Anweisungen von Threads ausgeführt werden, etwa in Abhängigkeit von der Thread-ID oder auch von Werten von Variablen. Innerhalb eines Warps wird das SIMD-Prinzip angewendet, bedingte Anweisungen werden hier durch Serialisierung umgesetzt. Das heißt, wenn einige Threads eines Warps eine bestimmte Anweisung nicht ausführen sollen, so können sie nicht stattdessen eine andere (die nächste) Anweisung ausführen. Erst wenn die anderen Threads die bedingte Anweisung ausgeführt haben, können alle Threads des Warps zur nächsten Anweisung übergehen. Das führt zu Leistungseinbußen.

Bedingte Anweisungen von Threads verschiedener Warps oder Blöcke dagegen werden völlig unabhängig voneinander verarbeitet, so dass in diesem Fall die Ausführung unterschiedlicher Instruktionen ohne Leistungsverlust durchgeführt werden kann.

Die Grundidee von CUDA ist es, sehr viele Threads und Blöcke innerhalb eines Kernels zu verwenden. Bei der eigentlichen Berechnung wird ein Block immer von genau einem Multiprozessor [62, Kapitel 4] der Grafikkarte vollständig verarbeitet. Auf die konkrete Abbildung der virtuellen Blöcke auf die realen Multiprozessoren, das heißt welcher Multiprozessor in welcher Reihenfolge welche Blöcke verarbeitet, hat der Entwickler keinen Einfluss: Die Verarbeitung aller Blöcke wird als parallel angenommen, die Anzahl der Multiprozessoren und auch die Abbildung von Blöcken auf Multiprozessoren ist transparent.

Die Verteilung der Blöcke auf die Multiprozessoren bzw. das Wechseln zwischen den Blöcken geschieht durch CUDA sehr effizient. Durch eine große Anzahl von Blöcken kann daher nicht nur eine gute Auslastung der GPU erreicht werden. Zudem ist der Kernel dadurch auch sehr unabhängig von der verwendeten Grafikkarte bzw. sogar Grafikkartengeneration, da auch Kernel, die für ältere GPUs geschrieben wurden, die größere Anzahl von Multiprozessoren auf neueren GPUs sofort nutzen können, denn es ändert sich lediglich das Mapping zwischen Blöcken und Multiprozessoren.

Dieses Konzept von CUDA wird ebenfalls im Quellcode 3.1 veranschaulicht. Das Schlüsselwort `__global__` kennzeichnet den Kernel, also die auf der GPU zu berechnende Prozedur. Es sollen zwei $N \times N$-Matrizen addiert werden. Diese Aufgabe wird auf $N/256$ Blöcke der Größe 16×16 verteilt, so dass jeder Thread genau eine Addition zu verarbeiten hat. Auf einen einzelnen Thread entfällt also nur eine kleine Teilaufgabe, es werden dementsprechend viele Threads verwendet.

Jedem Thread stehen aus Sicht des Entwicklers drei Arten von Speicher zur Verfügung:

Thread-lokaler Speicher

Jeder Thread besitzt seinen eigenen privaten Thread-lokalen Speicher. Wann immer möglich, nutzt der Compiler Register zur Speicherung von lokalen Variablen. Unter bestimmten Umständen, etwa wenn sehr viele Variablen verwendet werden oder wenn Arrays im Thread-lokalen Speicher abgelegt werden sollen, wird der Thread-lokale Speicher in den globalen Speicher der Grafikkarte geschrieben. Im Beispiel in Quellcode 3.1 werden die Integer `int i` und `int j` in Registern gespeichert. Der Zugriff auf Register ist sehr schnell, das heißt, er verursacht keine zusätzliche Verzögerung zur ausgeführten Operation. Die maximale Anzahl der Register, die ein Thread verwenden kann, hängt von der konkreten Grafikkarte ab und liegt zwischen 64 und 128 Registern.

Shared Memory

Alle Threads eines Blocks haben Zugriff auf einen gemeinsamen *shared memory* (Shared Memory). Die Zugriffszeit auf diesen Speicher ist sehr gering (u.U. wie Registerzugriffe), so dass Threads eines Blocks über diesen Speicher sehr schnell kommunizieren können. Die Zugriffszeit wird aber auch durch das Zugriffsmuster

der Threads eines Blocks bestimmt. In Abhängigkeit von der verwendeten Grafikkarte und der genutzten Konfiguration stehen einem Block maximal zwischen 16 KB und 48 KB Shared Memory zur Verfügung.

Globaler Speicher

Alle Threads können auf den globalen Speicher, den Grafikspeicher, zugreifen. Zwar ist der Datendurchsatz zwischen Threads und dem globalen Speicher sehr hoch, jedoch ist die Zugriffszeit ebenfalls vergleichsweise hoch und liegt in der Größenordnung mehrerer Hundert Takte.

Bei CUDA wird auf den globalen Speicher immer durch s.g. *memory transactions* (Speichertransaktionen) [62, Abschnitt 5.3.2.1] zugegriffen. Greift also Thread auf ein Element im globalen Speicher zu, so findet eine Speichertransaktion statt, bei der ein Speichersegment übertragen wird. Die übertragenen Speichersegmente haben dabei in der Regel ein Vielfaches der Größe des durch den Thread angeforderten Elements, so werden etwa 4 Byte Integer in 128 Byte Segmenten übertragen. Deswegen wird bei der Evaluierung eines Algorithmus (wie etwa im Abschnitt 5.3) gelegentlich der Speicherdurchsatz aus Hardwaresicht und aus Anwendungssicht unterschieden: im angeführten Beispiel hat die Anwendung nur 4 Byte angefordert, die Hardware hat jedoch 128 Byte übertragen, je nach Perspektive wird also ein unterschiedlicher Speicherdurchsatz erzielt.

Wie gesagt, arbeiten die Threads eine Warps nach dem SIMD-Prinzip, auch der Zugriff auf Elemente im Speicher findet deswegen gleichzeitig statt. Greifen Threads eines Warps gleichzeitig auf Speicherelemente des gleichen Segments zu, so werden diese Elemente in der gleichen Speichertransaktion übertragen: in diesem Fall nennt man die Speicherzugriffe *coalesced*, andernfalls *non-coalesced*. Da *non-coalesced* Speicherzugriffe mehrere Transaktionen erzeugen, wird mit solchen Zugriffen eine geringere Leistung als mit *coalesced* Zugriffen erzielt.

Greifen in obigem Beispiel also alle 32 Threads eines Warps auf 4 Byte-Elemente im gleichen Speichersegment zu, so können alle angeforderten Elemente in nur einer 128 Byte-Transaktion übertragen werden, die Zugriffe sind also *coalesced* und der Speicherdurchsatz aus Anwendungssicht ist identisch zu dem aus Hardwaresicht.

Moderne Desktop-Grafikkarten haben bis zu 2 GB globalen Speicher. Einige dedizierte Grafikkarten für GPGPU verfügen über bis zu 6 GB globalen Speicher. Im Beispiel in Quellcode 3.1 sind die drei Arrays A,B und C im globalen Speicher abgelegt.

KAPITEL 4

Konzept & Übersicht

Wie in der Einleitung bereits erwähnt, wird in der vorliegenden Arbeit untersucht, inwiefern die Verarbeitung bestimmter XML-Prozesse, die sich naturgemäß auf die Baumstruktur von XML-Dokumenten stützen, so parallelisiert werden kann, dass durch die Verwendung von SIMD-Architekturen wie GPUs eine Beschleunigung der Verarbeitung (gegenüber der nicht-parallelen Variante bzw. der Verarbeitung ohne spezielle Hardware) erreicht wird.

In der Vergangenheit wurden bereits Ansätze von Jordan [45, 47, 46] entwickelt, XML auf einer GPU zu verarbeiten, jedoch konnten damit bisher keine Leistungsgewinne gegenüber CPUs erzielt werden, im Gegenteil, bisher waren diese Ansätze nach den Aussagen von Jordan langsamer als die üblichen CPU-Implementierungen. In neuesten Arbeiten haben auch Moussalli et al. [58] ihren Ansatz zur XPath-Filterung von FPGAs auf GPUs übertragen. In diesen Arbeiten werden allerdings zum einen nur ganz bestimmte Filteranfragen auf XML-Daten betrachtet, also keine komplexen XML-Prozesse auf GPUs durchgeführt. Zum anderen betrachten die Autoren in ihrer Arbeit ein Szenario, in welchem Zehntausende oder Hundertausende gleichzeitig vorliegender *filter queries* parallel verarbeitet werden müssen. Deswegen werden einzelne Anfragen (etwa Location Steps) anders als in unserem Ansatz (vgl. nachfolgenden Abschnitt), nicht stark parallelisiert.

In dieser Arbeit werden neue Wege zur Verarbeitung von XML auf einer GPU entwickelt. Diese Arbeit wird exemplarisch durchgeführt, das heißt, es soll als *proof-of-concept* eine konkrete XML-Anwendung, XSLT, auf einer GPU implementiert

und die Leistungsfähigkeit eingeschätzt werden.

Viele der in dieser Arbeit vorgestellten Ansätze und Konzepte sind zwar allgemeingültig oder lassen sich in gewissem Umfang auch auf andere Hardware übertragen, dennoch werden im folgenden GPUs von NVIDIA als Zielhardware des hier entwickelten Systems angenommen.

4.1 Szenario

Es existieren verschiedene XML-Prozesse, die genügend komplex und umfangreich sind, so dass eine Beschleunigung durch spezielle Hardware wünschenswert wäre. Beispielsweise existieren Ansätze, um das Parsen von XML-Dokumenten bzw. das Serialisieren von XML-Daten zu parallelisieren und/oder auf spezieller paralleler Hardware auszuführen

Die vorliegende Arbeit dagegen behandelt die Parallelisierung von XSLT. Wie in Abschnitt 2.4 beschrieben wurde, ist XSLT eine Sprache zur Definition von Transformationen auf XML-Dokumenten. Eine typische Anwendung für XSLT ist die Transformation von XML Daten in verschiedene Sichten auf diese Daten, etwa um nur Teile der Daten darzustellen oder um verschiedene Ausgabeformate der Daten zu erzeugen. Diese Anwendung findet sich häufig im Zusammenhang mit *Client-Server* Diensten, in denen Daten an einem zentralen Punkt, dem Server, gespeichert sind und von verschiedenen Clients in verschiedener Weise abgerufen werden. Das Szenario, in dem viele (bis zu mehrere Hundert) Clients gleichzeitig beliebige Transformationen auf festen XML-Daten ausführen wollen, soll die Ausgangssituation dieser Arbeit sein.

Das Ziel dieser Arbeit ist es, einen prototypischen XSLT-Prozessor für NVIDIAs GPUs zu entwickeln und zu implementieren, der im oben genannten Szenario XSLT-Transformationen verarbeiten kann. Grundlage ist dabei XPath `1.0` [84] und XSLT `1.0` [85] und nicht die neueren Standards XPath `2.0` [96] und XSLT `2.0` [92]. Diese Entscheidung wurde getroffen, da XPath `1.0` bzw. XSLT `1.0` im Wesentlichen Untermengen von XPath `2.0` bzw. XSLT `2.0` sind und in der vorliegenden Arbeit sogar nur Untermengen des Sprachumfangs von XPath `1.0` und XSLT `1.0` betrachtet wurden. Zum anderen jedoch sind XPath `1.0` und XSLT `1.0` noch immer weit verbreitete Standards, wogegen nur wenige XML-Programme konform zu XPath `2.0` und XSLT `2.0` sind. Diese Arbeit bezieht sich also auf XPath `1.0` und XSLT `1.0`.

Das primäre Ziel der Arbeit ist die Parallelisierung eines XML-Prozesses. Es wurde daher darauf verzichtet, den gesamten Sprachumfang von XPath und XSLT zu implementieren. Es wurden für XPath im Wesentlichen nur Location Paths ohne Prädikate implementiert, insbesondere wurde auf die speziellen String-Funktionen, Sortierfunktionen usw. verzichtet. Auch *namespaces* und daher auch die *namespace*-Achse wurden nicht beachtet. Die Implementierung ohne Prädikate wirkt sich unter Umständen auch auf die Komplexität der verwendeten Algorithmen aus, wie in Abschnitt 8 erläutert wird. Auch für XSLT wurden nur die wichtigsten Konstrukte implementiert.

4.2 Konzept & Ziele

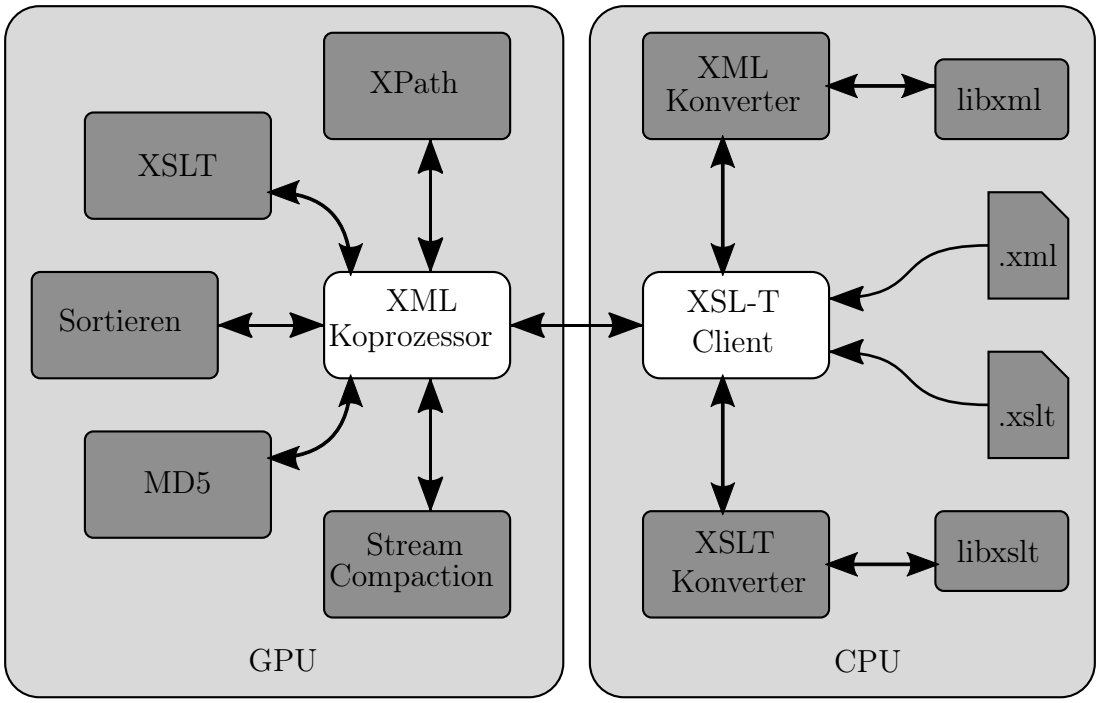


Abbildung 4.1: Schema GPU Koprozessor

Entsprechend dem vorliegenden Szenario ist das Ziel dieser Arbeit, einen (oder mehrere) XSLT-Prozess(e) auf einer GPU parallel zu verarbeiten.

Zur Verarbeitung eines einzelnen XSLT-Prozesses wiederum gehören viele Teilprozesse, die aber im Rahmen dieser Arbeit nicht alle für die Verarbeitung von der

GPU angepasst wurden. So verbleibt das Parsen der Dokumente (XML und XSLT) in eine geeignete Datenstruktur in großen Teilen eine Aufgabe der CPU. Ebenso wird das Serialisieren der (transformierten) XML-Daten von der CPU ausgeführt. Alle Schritte nach dem Parsen des Eingabe-Dokuments und vor dem Serialisieren des Ausgabe-Dokuments werden von der GPU ausgeführt, das heißt der eigentliche XSLT-Prozess, XPath als integraler Bestandteil von XSLT und die für diese Prozesse notwendigen Hilfsfunktionen Sortieren, Hashwertbildung und Stream Compaction.

Die Abbildung 4.1 zeigt schematisch die Aufgabenverteilung. Das Vorbereiten und Nachbereiten der Eingabe- und Ausgabedokumente wird unter Verwendung von Standardbibliotheken von der CPU durchgeführt. Alle anderen Schritte sind Teil der Verarbeitung durch die GPU.

Die Implementierung des Hashing-Algorithmus [72] wurde für das Szenario dieser Arbeit entwickelt und ist in anderen Szenarien nur bedingt sinnvoll einsetzbar; der Stream Compaction Algorithmus zum Einsammeln von Ergebnissen folgt bekannten Ansätzen (etwa [41]). Daher werden diese beiden Funktionen nicht weiter diskutiert. Die Sortierfunktion und die generelle Verwendung der GPU als Koprozessor jedoch verwenden neue Algorithmen bzw. Ansätze, die nicht speziell für den XML-Kontext entwickelt wurden und auch in anderen Szenarien wertvoll sind. Außerdem ist ihre Verwendung ein spezielles Merkmal der hier vorgestellten XML-Verarbeitung, die es in anderen Ansätzen so nicht gibt.

Daher werden in dieser Arbeit neben XSLT und XPath auch die Sortierfunktion und die Verwendung der GPU als Koprozessor im Detail erläutert.

Die Parallelisierung von Anwendungen (oberhalb des Instruktionslevels) kann grob in zwei unterschiedliche Ansätze eingeteilt werden: Zum einen wird versucht, möglichst viele unabhängige Prozesse gleichzeitig auszuführen (Multitasking). Dadurch wird eine möglichst gute Ausnutzung der verfügbaren Ressourcen erreicht, denn die verfügbaren Ressourcen können unter allen Prozessen verteilt werden, so dass, wenn beispielsweise ein Prozess auf ein Ereignis (etwa I/O) wartet, alle anderen Ressourcen genutzt werden können. Außerdem kann mit diesem Ansatz eine nahezu vollständige Ausnutzung der Ressourcen erreicht werden, selbst wenn kein einzelner Prozess eine Ressource vollständig nutzt. Insbesondere bei der Nutzung von GPUs, die ihre Leistung aus der großen Anzahl paralleler Recheneinheiten schöpfen, kann die Leistung der GPU oft nicht durch einen einzigen Prozess vollständig genutzt werden, wohl aber durch mehrere Prozesse. Dieser Ansatz der Par-

allelisierung ist gut geeignet den Durchsatz an verarbeiteten Aufgaben zu erhöhen. Der zweite Ansatz ist die Parallelisierung einer einzelnen Aufgabe, um durch die Parallelverarbeitung einen Leistungsgewinn zu erzielen, das heißt um die benötigte Verarbeitungszeit zu senken.

In dieser Arbeit werden beide Ansätze verfolgt. Konkret bedeutet dies, dass mehrere unabhängige XML-Prozesse von der GPU nebenläufig verarbeitet werden sollen, wobei auch ein einzelner Prozess von mehreren Recheneinheiten parallel verarbeitet wird. Es soll also ein XSLT-Prozess parallelisiert und mehrere XSLT-Prozesse sollen gleichzeitig von der GPU ausgeführt werden.

Dies soll erreicht werden, indem die GPU als paralleler asynchroner Koprozessor für bestimmte Aufgaben eingesetzt wird: Verschiedene CPU-Prozesse können zu beliebigen Zeitpunkten Anfragen an den Koprozessor senden (anders als etwa bei Moussalli et al. [58]), wo jede Anfrage mit einem Teil der verfügbaren Ressourcen verarbeitet wird. In der Abbildung 4.1 bedeutet dies, dass zu jedem Zeitpunkt mehrere verschiedene Funktionen von der GPU verarbeitet werden können.

Diese Arbeit stellt fünf der wichtigsten Teilprobleme vor, die zur Umsetzung des GPU-basierten XSLT-Prozessors gelöst wurden.

- In Teil II werden die beiden wichtigen Basisdienste für den XSLT-Prozessor erläutert, die keinen direkten XML-Bezug haben:
  - Kapitel 5 führt einen Sortieralgorithmus für GPUs ein, der in den vorbereitenden Schritten zur XML-Verarbeitung zum Einsatz kommt und einen wesentlichen Beitrag zur Umsetzung des XSLT-Prozessors liefert.
  - Kapitel 6 stellt das CUDA-OS vor, durch welches die Nutzung der GPU als Koprozessor in der oben genannten Form überhaupt erst möglich wird.
- In Teil III werden die XML-spezifischen Arbeiten vorgestellt:
  - In Kapitel 7 wird die Datenstruktur bzw. die Kodierung des XML-Dokuments auf der GPU vorgestellt, die in dieser Arbeit verwendet wird. In diesem Kapitel wird auch gezeigt, an welcher Stelle der Sortieralgorithmus aus Kapitel 5 verwendet wird.
  - Kapitel 8 erläutert die Umsetzung einer Untermenge des XPath-Sprachumfangs für die parallele Verarbeitung durch GPUs. Dazu gehört auch die Einführung einer formalen Semantik, so dass die Äquivalenz der

Komponente	Merkmal / Version
CPU	Intel Core i7 960 @ 3,2 GHz
GPU	NVIDIA GTX580, GTX480, GTX280
Arbeitsspeicher	6 GB DDR3
Betriebsystem	64 bit Ubuntu 10.04.2 LTS
Xalan	1.10.0
Xerces	2.8.0
Saxon-HE	9.3.0.5J
libXSLT	1.1.26
libXML	2.7.6
CUDA	4.0
NVIDIA Treiber	270.41.19

Tabelle 4.1: Konfiguration des Testsystems

hier verwendeten mengenbasierten Semantik zur im Standard verwendeten knotenbasierten Semantik für den implementierten Sprachumfang gezeigt werden kann.

– Im Kapitel 9 wird der XSLT-Prozessor erläutert und es werden die beiden wesentlichen Punkte, die Umsetzung des rekursiven Prozessmodells aus dem Standard in eine parallelisierbare iterative Variante und die Parallelisierung der einzelnen Bestandteile des XSLT-Prozesses, vorgestellt.

Jede der vier Anwendungen bzw. Funktionen wurde einzeln in umfangreichen Tests evaluiert. Für alle Tests wurde das gleiche System eingesetzt. Tabelle 4.1 zeigt eine Übersicht über die Konfiguration des verwendeten Testsystems. Der parallele XPath-Prozessor sowie der parallele XSLT-Prozessor wurden gegen drei der bekanntesten freien XPath- bzw. XSLT-Prozessoren getestet:

1. den *Xalan-C++*-XSLT-Prozessor [80], der *Xerces-C++* [81] zum Parsen und Validieren der XML-Dokumente verwendet,
2. den *Saxon-HE*-XSLT-Prozessor [48] für Java und
3. die *libXSLT* [70] Bibliothek in C, die auf libXML [9] basiert.

Teil II

Basisdienste

KAPITEL 5

Bitonic Sort

Ein wichtiger Bestandteil der hier vorgestellten schnellen XML-Verarbeitung auf GPUs ist das Sortieren. Im Zuge der Vorverarbeitung der zu behandelnden XML-Dokumente müssen die Knoten des Dokuments nach ihren Namen bzw. ihren Typen sortiert werden (vgl. Abschnitt 7.2). Dazu soll ebenfalls die GPU genutzt werden.

Es gelten grundsätzlich zwei Anforderungen an den hier zu verwendenden Sortieralgorithmus: zum einen soll der Algorithmus vergleichsbasiert arbeiten. Ein vergleichsbasierter Algorithmus kann im Prinzip für beliebige Schlüssel und Ordnungen angepasst werdem, einfach indem die Vergleichsoperation neu definiert wird. Das macht solche Algorithmen besonders flexibel und einfach zu pflegen. Außerdem ist die Laufzeit eines vergleichsbasierten Algorithmus weniger abhängig vom verwendeten Schlüssel (vgl. *sorting by counting*[51]). Zum anderen soll der Algorithmus *in-place* arbeiten, das heißt neben dem für die zu sortierende Sequenz benötigten Speicher wird nur noch eine konstante Menge weiteren Speichers durch den Algorithmus in Anspruch genommen.

Es gibt bereits viele Sortieralgorithmen für GPUs (Merge Sort [75], Radix Sort [75, 76], Sample Sort [52], Quick Sort [23] etc.), doch keiner dieser Algorithmen arbeitet in-place und vergleichsbasiert. In der Vergangenheit wurde auch bereits Bitonic Sort für GPUs implementiert bzw. Sortieralgorithmen entwickelt, die auch Bitonic Sort verwenden [98, 22, 71, 50, 39], jedoch sind diese Sortierverfahren zum Teil nicht für aktuelle GPUs und Frameworks verfügbar oder sind im Vergleich zu neueren Verfahren nicht genügend leistungsfähig. Obwohl Bitonic Sort grund-

sätzlich ein in-place-Algorithmus ist, arbeiten aufgrund der speziellen Architektur früherer GPUs einige ältere Implementierungen nicht in-place.

Zudem sind die vorhandenen Implementierungen der meisten Algorithmen für GPUs oft auf wenige Datentypen beschränkt, die so nicht für die vorliegende Arbeit hätten verwendet werden können.

Im Folgenden wird ein Sortieralgorithmus für NVIDIAs GPUs vorgestellt, der sowohl in-place arbeitet als auch vergleichsbasiert ist. Dabei zeigt sich, dass der hier vorgestellte Algorithmus eine sehr hohe Sortierrate hat und für viele Sequenzlängen und Sortierschlüssel-Typen schneller ist als alle anderen vergleichsbasierten Algorithmen und auch als einige nicht-vergleichsbasierte Algorithmen.

Teile des hier beschriebenen Algorithmus sind erstmalig in [69] vorgestellt worden. Verbesserung und Erweiterungen sind in [66] und [68] zu finden.

Der folgenden Abschnitt führt Sortiernetze und einige wichtige Begriffe ein. Der Abschnitt 5.2 stellt Bitonic Sort im Detail vor und in Abschnitt 5.3 wird eine einfache sowie unsere verbesserte Implementierung von Bitonic Sort diskutiert. In Abschnitt 5.4 wird die verbesserte Variante analysiert und in Abschnitt 5.5 werden die Ergebnisse der Laufzeittests präsentiert.

5.1 Einleitung

Die Grundlage für den in dieser Arbeit vorgestellten Sortieralgorithmus ist das von K. Batcher im Jahre 1967 entwickelte Bitonic Sort [16]. Obwohl die Zeit-Komplexität dieses Verfahrens nicht optimal ist ($O(n \log^2 n)$ gegenüber der optimalen Komplexität $O(n \log n)$), ist Bitonic Sort ein weit verbreiteter Algorithmus: Bitonic Sort ist ein so genannter Sortiernetzalgorithmus, welcher sich besonders gut parallel, insbesondere auch in Hardware, implementieren lässt. Die Abbildung 5.1 zeigt das Bitonic Sort-Sortiernetz für Sequenzen der Länge 8.

Die waagerechten Linien repräsentieren dabei so genannte *Data Lanes* (Lanes), auf welchen sich die Elemente der zu sortierenden Sequenz gleichmäßig von links nach rechts bewegen. Die Pfeile zwischen zwei Lanes repräsentieren *Compare-Exchange-Operationen*, kurz COEX-Operationen. Wann immer zwei Datenelemente (das heißt die s.g. Schlüssel) auf eine COEX-Operation treffen, werden die Datenelemente verglichen und das größere an die Pfeilspitze verschoben, das kleinere an das Pfeilende. Die Abbildung 5.2 zeigt eine einzelne COEX-Operation schematisch.

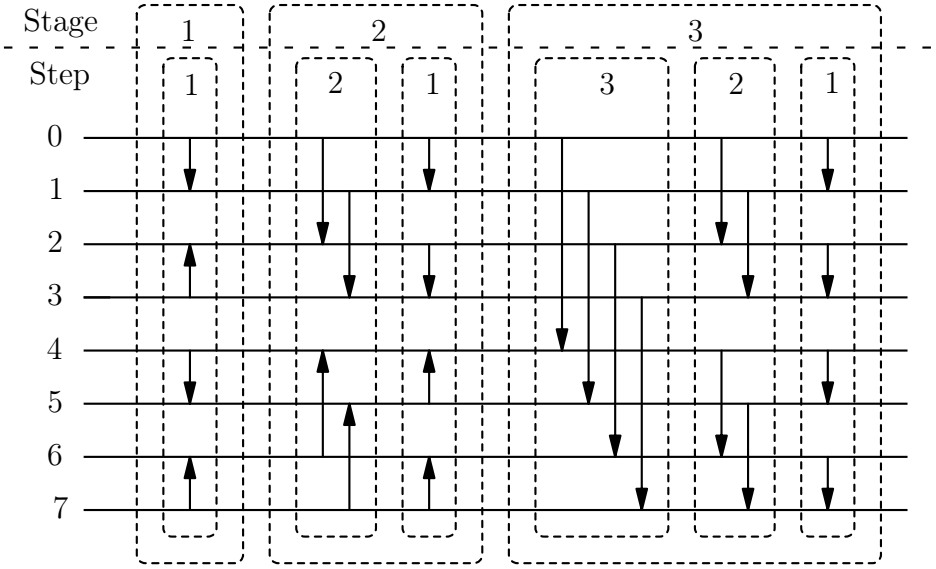


Abbildung 5.1: Bitonic Sort Sortiernetzwerk für Sequenzen der Länge 8

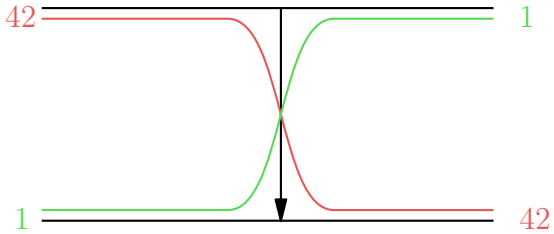


Abbildung 5.2: Schematische Funktionsweise einer COEX-Operation

Die Besonderheit von Sortiernetzen bzw. Sortiernetzalgorithmen gegenüber anderen Verfahren liegt in der statischen Ausführung der COEX-Operationen: In einem Sortiernetzalgorithmus ist die Ausführung der COEX-Operationen unabhängig vom Ergebnis einer COEX-Operation, das heißt wird eine COEX-Operation auf den beiden Schlüsseln k_0 und k_1 ausgeführt, so ist die Menge und Reihenfolge der nachfolgenden COEX-Operationen im Falle $k_0 < k_1$ identisch zu der im Fall $k_0 \geq k_1$. Die Menge und Reihenfolge der notwendigen COEX-Operationen ist also gleich für beliebige Sequenzen einer bestimmten Länge. Dies macht Sortiernetze besonders interessant für die statische Implementierung in Hardware, aber auch allgemein für parallele Implementierungen.

Aus der Unabhängigkeit von Bitonic Sort gegenüber den Werten der Schlüssel folgt eine weitere Eigenschaft: Auch die Laufzeit ist unabhängig von den Werten der Schlüssel einer Sequenz. Das bedeutet beispielsweise, dass Bitonic Sort eine bereits sortierte Sequenz genauso lange verarbeitet wie eine zufällig erzeugte. Das ist natürlich in manchen Fällen ein Nachteil, jedoch ist die gute Vorhersagbarkeit der Laufzeit des Algorithmus in vielen Fällen sehr nützlich.

Die Abbildung 5.3 zeigt die Sortierung einer Beispielsequenz ganzer Zahlen der Länge 8 durch Bitonic Sort. Exemplarisch ist für ein Element der „Weg“ durch das Sortiernetz (bestimmt durch die COEX-Operationen) rot markiert. Während einige COEX-Operationen völlig unabhängig voneinander durchgeführt werden können (das heißt in beliebiger Reihenfolge oder auch parallel), müssen andere COEX-Operationen synchronisiert werden, das heißt eine bestimmte Reihenfolge muss gewährleistet sein. Beispielsweise müssen die COEX-Operationen, die auf dem markierten Weg des Beispielelementes liegen, selbstverständlich in der Reihenfolge von links nach rechts ausgeführt werden.

5.2 Bitonic Sort

Bitonic Sort ist ein so genannter *divide-and-conquer, merge-based* Sortiernetzalgorithmus. Um eine Sequenz der Länge 2^k zu sortieren, beginnt Bitonic Sort (wie andere merge-based Algorithmen auch) damit, benachbarte Subsequenzen der Länge 1 (also sortierte Sequenzen) in einem Merge-Step (*bitonic Merge*) zu sortierten Sequenzen der Länge 2 zu verschmelzen (*merge*). Danach werden dann benachbarte sortierte Sequenzen der Länge 2 zu sortierten Sequenzen der Länge 4 vereinigt usw. Zuletzt vereinigt Bitonic Sort zwei sortierte Sequenzen der Länge 2^{k-1} zu

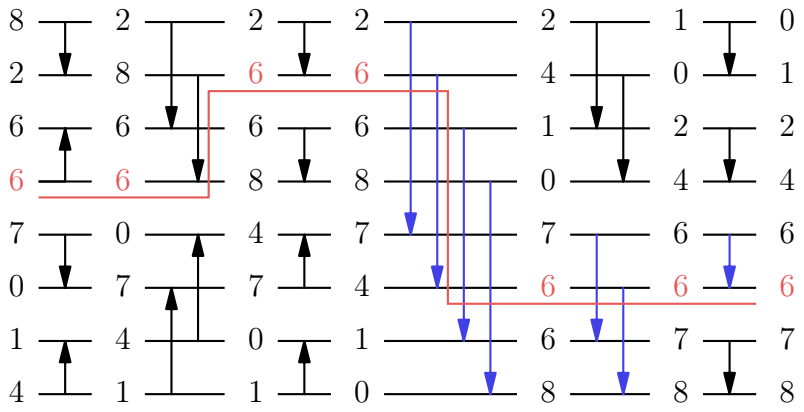


Abbildung 5.3: Beispiel eines Sortiernetzes für Sequenzen der Länge 8

einer sortierten Sequenz der Länge 2^k.

Bitonic Sort ist in *Stages* und *Steps* aufgeteilt (vgl. Abbildung 5.1). Die Sortierung einer Sequenz der Länge 2^k besteht aus k Stages $1, \ldots, k$ und jede Stage s besteht aus s Steps $s, s-1, \ldots, 1$. Die Gesamtzahl der Steps ist demnach:

$$\sum_{s=1}^{k} s = \frac{k \cdot (k+1)}{2} \tag{5.1}$$

Wie bereits gesagt, kann Bitonic Sort gut in Software auf parallelen Architekturen (insbesondere SIMD/SIMT) wie CUDA implementiert werden, denn viele COEX-Operationen können unabhängig voneinander, das heißt in beliebiger Reihenfolge oder auch parallel, ausgeführt werden. So können beispielsweise alle COEX-Operationen eines einzelnen Steps unabhängig voneinander durch verschiedene Recheneinheiten ausgeführt werden. Zwischen bestimmten Operationen zweier aufeinander folgender Steps muss jedoch im Allgemeinen synchronisiert werden. Das Beispiel in Abbildung 5.3 verdeutlicht diesen Zusammenhang: Um im letzten Step die blau markierte COEX-Operation durchführen zu können, muss gewährleistet sein, dass bereits beide blau markierten COEX-Operationen des vorletzten Steps verarbeitet wurden. Dafür wiederum müssen bereits alle COEX-Operationen des drittletzten Steps verarbeitet worden sein.

Ein weiterer bestimmender Faktor in Bitonic Sort ist die Tatsache, dass die Ergebnisse der COEX-Operationen ausgetauscht werden müssen, das heißt zwischen den Recheneinheiten muss eine geeignete Kommunikation stattfinden. In CUDA würden die genannten Recheneinheiten durch Threads implementiert werden.

Es ist in CUDA nicht möglich, zwei beliebige Threads zu synchronisieren, demnach ist keine „echte“ Kommunikation zwischen beliebigen Threads möglich. Um eine bestimmte Reihenfolge der Verarbeitung bestimmter Aufgaben sicherzustellen, müssen die Aufgaben in verschiedenen, aufeinander folgenden Kernels ausgeführt werden, zwischen denen synchronisiert wird (vgl. Abschnitt 3.2). Die „Kommunikation“ zwischen den Threads verschiedener Kernel Launchs kann in diesem Fall geschehen, indem die Threads des ersten Kernels ihre Daten in den persistenten globalen Speicher schreiben, wo die Daten von den Threads des nachfolgenden Kernels wieder gelesen werden können.

Sowohl die Kommunikation durch die Nutzung mehrerer Kernel wie auch die Synchronisation im Allgemeinen senken die Leistung eines Algorithmus in CUDA. Deshalb wurden im hier vorgestellten Algorithmus insbesondere zwei Ziele verfolgt:

1. Reduktion der notwendigen Kommunikation und Synchronisation in Bitonic Sort, wodurch die Anzahl der Kernel Launchs und der zur Kommunikation notwendigen Zugriffe auf den globalen Speicher verringert wird.

2. Umfassende Nutzung des Shared Memory und der effizienten Synchronisation der Threads innerhalb eines Blocks. Auch dadurch wird die Anzahl der Kernel Launchs und der zur Kommunikation notwendigen Zugriffe auf den globalen Speicher verringert.

5.3 Algorithmus

5.3.1 Einfacher Ansatz

In einer einfachen CUDA-Implementierung von Bitonic Sort würde immer ein Thread eine COEX-Operation eines Steps einer Stage ausführen. In diesem Fall müssten alle Threads nach jedem Step synchronisiert werden, um die richtige Reihenfolge der COEX-Operationen zu garantieren, das heißt um zu gewährleisten, dass alle COEX-Operationen in einem Step s verarbeitet wurden, bevor die erste COEX-Operation des nachfolgenden Steps $s-1$ verarbeitet wird. Außerdem müssten nach jedem Step die Ergebnisse der vorangegangenen COEX-Operationen ausgetauscht werden. In dieser einfachen Variante müsste daher die Synchronisation zwischen Steps durch die Verwendung eines Kernel Launchs pro Step sichergestellt

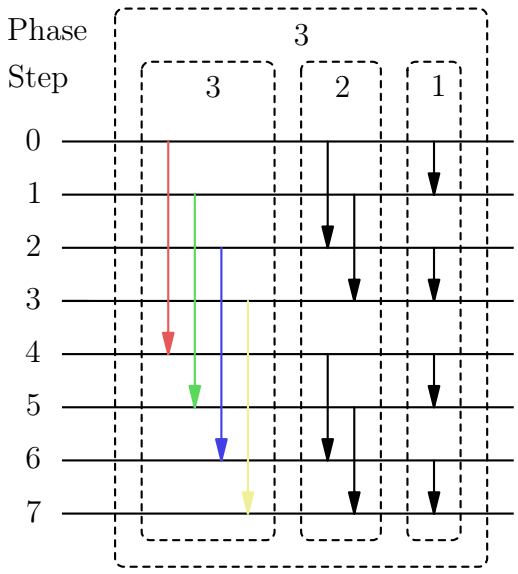


Abbildung 5.4: Beispiel einer einfachen Implementierung von Bitonic Sort

werden. Die „Kommunikation“ zwischen Threads würde durch das persistente Speichern der Ergebnisse im globalen Speicher umgesetzt.

Das Sortieren einer Sequenz der Länge $n = 2^k$ würde durch $k \cdot (k+1)/2$ (vgl. Gleichung 5.1) Kernel Launchs implementiert werden, wobei in jedem Launch alle COEX-Operationen eines Steps einer Stage berechnet werden. In jedem Kernel Launch würden gerade $n/2 = 2^{k-1}$ Threads gestartet werden, so dass jeder Thread die beiden für eine COEX-Operation notwendigen Daten aus dem globalen Speicher liest, die COEX-Operation durchführt und die Daten wieder in den globalen Speicher schreibt.

Die Abbildung 5.4 zeigt diesen Ansatz: Step 3 in Stage 3 wird in der Abbildung durch einen Kernel Launch mit 4 Threads verarbeitet. Jeder Thread liest zwei Datenelemente aus dem Speicher (die Elemente {0,4}, {1,5}, {2,6}, {3,7} in der Abbildung) und verarbeitet die entsprechende der vier COEX-Operationen dieses Steps. Danach werden die Elemente zurückgeschrieben. Die unterschiedlichen Threads sind in der Abbildung durch die unterschiedlichen Farben dargestellt.

Diese einfache Implementierung hat einige gute Eigenschaften: Sehr viele Threads können parallel arbeiten, so dass eine gute Auslastung der GPU erreicht wird. Die Rechenschritte sowie die Zugriffe auf den globalen Speicher sind für jeden Thread gleich, so dass eine gleichmäßige Lastverteilung unter den Threads vorliegt. Der Hauptnachteil dieses Ansatzes ist die große Anzahl der notwendigen

Zugriffe in den globalen Speicher, da jeder einzelne Step in einem eigenen Kernel verarbeitet wird. In jedem Kernel Launch müssen in dieser Variante n Lese- und Schreibzugriffe durchgeführt werden und die Gesamtzahl der Zugriffe auf den globalen Speicher ist entsprechend Gleichung 5.1:

$$2 \cdot n \cdot \frac{k \cdot (k+1)}{2} = n \cdot k \cdot (k+1) \tag{5.2}$$

Tatsächlich ist die einfache Implementierung von Bitonic Sort *memory bandwith bound*, das heißt die Leistungsfähigkeit wird im Wesentlichen durch die zur Verfügung stehende Speichertransferrate limitiert. Beispielsweise müssen für eine Sequenz der Länge $n = 2^k$ mit $k = 20$ bereits

$$\frac{n \cdot k \cdot (k+1)}{n} = k \cdot (k+1) = 20 \cdot 21 = 420 \tag{5.3}$$

Zugriffe pro Element durchgeführt werden. Darüber hinaus ist das Zugriffsmuster auf den Speicher in den kleineren Steps ungünstig für NVIDIAs GPUs, so dass die Speicherzugriffe *non-coalesced* sind. Das führt dazu, dass der Speicherdurchsatz aus Anwendungssicht deutlich kleiner ist als aus Sicht der Hardware (vgl. Abschnitt 3.2). In einem Test wurde beispielsweise für eine GTX280 GPU 77 GB/s Speicherdurchsatz aus Anwendungssicht gegenüber 110 GB/s Speicherdurchsatz aus Hardwaresicht für eine 32 bit-Integer-Sequenz der Länge 2^{20} gemessen.

5.3.2 Reduzierung der Speicherzugriffe

Der hier verwendete Ansatz, um die notwendige Kommunikation und Synchronisation und damit die Anzahl der notwendigen Speicherzugriffe zu reduzieren, besteht darin, einen Thread in einem Kernel Launch mehr als eine COEX-Operation aus mehr als einem Step verarbeiten zu lassen. Dadurch soll erreicht werden, dass Threads für bestimmte COEX-Operationen die Elemente nicht erneut aus dem globalen Speicher laden müssen, sondern diese Elemente wegen vorangegangener COEX-Operationen bereits in ihren lokalen Speichern haben. Die Frage ist nun, wie die korrekte Reihenfolge der COEX-Operationen gewährleistet werden kann, wenn ein Thread mehrere COEX-Operationen aus aufeinanderfolgenden Steps ohne weitere Synchronisation mit anderen Threads verarbeitet.

Dies kann dadurch erreicht werden, dass ein Thread gerade eine Menge von COEX-Operationen verarbeitet, die nur Abhängigkeiten innerhalb dieser Menge aufweisen: Ein Thread verarbeitet in aufeinanderfolgenden Steps gerade die

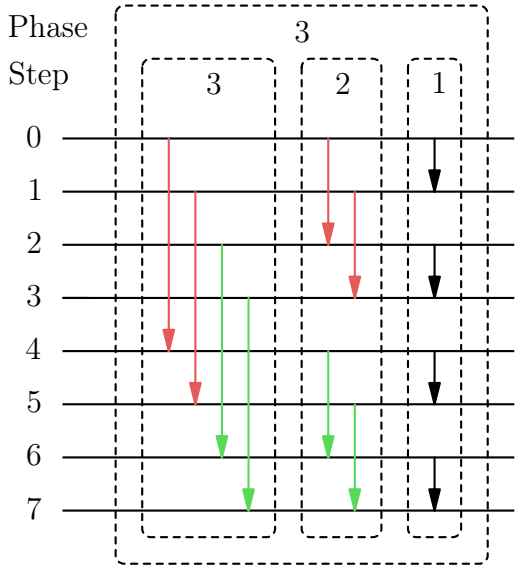


Abbildung 5.5: Verbesserter Ansatz: ungünstig gewählter *Job*

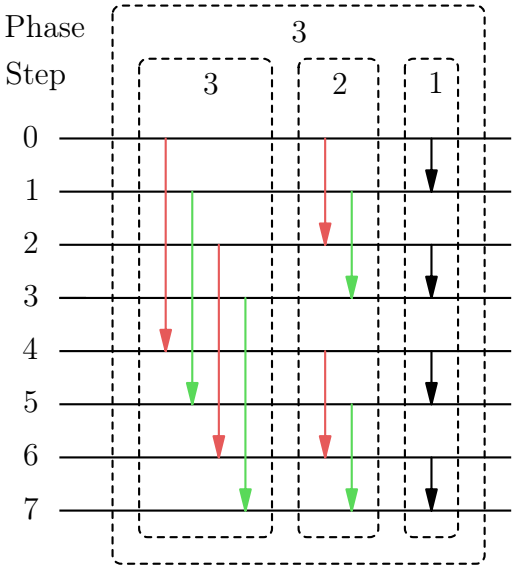


Abbildung 5.6: Verbesserter Ansatz: günstig gewählter *Job*

COEX-Operationen, die sich auf die gleichen Datenelemente beziehen. In diesem Fall nämlich kann der Thread selbst, ohne Synchronisation mit anderen Threads, die korrekte Reihenfolge der COEX-Operationen sicherstellen. Im Folgenden wird die Menge an COEX-Operationen (aus einem oder mehreren Steps), die ein einzelner Thread zu verarbeiten hat, *Job* genannt.

Die Abbildungen 5.5 und 5.6 verdeutlichen diesen Ansatz. In Abbildung 5.5 werden die notwendigen Operationen aus Step 3 und Step 2 in Stage 3 auf zwei Threads (wiederum durch unterschiedliche Farben markiert) aufgeteilt. Der Job des einen Threads ist also gerade definiert durch die COEX-Operationen auf den Elementpaaren $\{(0,4),(1,5),(0,2),(1,3)\}$, der des anderen Threads durch $\{(2,6),(3,7),(4,6),(5,7)\}$. Diese Wahl der Jobs ist ungünstig, denn in diesem Fall benötigen beide Threads in Step 2 Elemente, die im vorhergehenden Step durch den jeweils anderen Thread behandelt werden; es ist also Synchronisation und Kommunikation notwendig.

Anders ist die Situation in Abbildung 5.6: Die Jobs sind hier definiert durch $\{(0,4),(2,6),(0,2),(4,6)\}$ und $\{(1,5),(3,7),(1,3),(5,7)\}$. Beide Threads greifen in beiden Steps auf genau die gleichen Elemente zu und sind unabhängig von den COEX-Operationen, die von anderen Threads ausgeführt wurden. Zudem benötigen in diesem Fall beide Threads nur je 4 Elemente; in der Variante mit ungünstig

gewählten Jobs sind es dagegen 6 Elemente pro Thread. In dieser Variante verarbeitet also ein Thread je zwei COEX-Operationen aus zwei Steps, ohne dass Synchronisation benötigt wird. Zudem wird jedes Element nur einmal aus dem Speicher gelesen und in den Speicher geschrieben (und nicht zweimal wie in Abbildung 5.4).

Im hier vorgestellten Beispiel werden die COEX-Operationen zweier aufeinander folgender Steps so auf zwei Jobs aufgeteilt, dass alle COEX-Operationen beider Steps in einem einzelnen Kernel Launch ohne zusätzliche Synchronisation ausgeführt werden können. Dieser Ansatz kann natürlich erweitert werden, so dass in der Implementierung des hier vorgestellten Ansatzes bis zu 4 Steps in einem einzigen Kernel ausgeführt werden (in Abhängigkeit von der Schlüssellänge). Dadurch wird die Anzahl der Zugriffe auf den globalen Speicher nahezu geviertelt.

Theoretisch ließe sich dieser Ansatz natürlich für jede beliebige Anzahl an Steps, die in einem Kernel Launch verarbeitet werden sollen, erweitern. Allerdings werden mit diesem Ansatz zur Sortierung einer Sequenz der Länge $n = 2^k$ (wobei immer t aufeinanderfolgende Steps in einem Kernel Launch ausgeführt werden sollen) gerade $\frac{n}{2^t}$ Threads benötigt, wobei jeder Thread 2^t Datenelemente in seinem lokalen Speicher (Register) speichern muss. Mit steigendem t wird also die Anzahl der verwendeten Threads kleiner, die Anzahl der Datenelemente, die ein Thread in seinem lokalen Speicher ablegen muss, dagegen wächst. Zum einen ist aber die Anzahl der Register pro Thread beschränkt, zum anderen wird eine große Anzahl Threads benötigt, um die GPU auszulasten. Durch diese beiden Gründe ist die Anzahl der Steps t eine kleine Zahl (<6).

5.3.3 Nutzung des Shared Memory

Der zuvor beschriebene Ansatz kann auch von Threads auf Blöcke übertragen werden. Ein einzelner Job wird von einem Block anstelle eines Threads verarbeitet, wobei die Datenelemente im Shared Memory (dessen Latenz u.U. so gering ist wie die von Registern) anstelle von Registern gespeichert werden. Die COEX-Operationen innerhalb des Jobs werden unter den Threads des Blocks verteilt.

Die Menge an Shared Memory, die ein Block verwenden kann, ist viel größer als die Menge an Registern, die einem Thread zur Verfügung stehen. Demnach wäre es mit diesem Ansatz möglich, eine größere Anzahl t aufeinander folgender Steps in einem einzelnen Kernel Launch zu verarbeiten. Die Anzahl der verwende-

ten Threads wäre trotzdem genügend hoch, da in dieser Variante nicht ein Thread einen Job verarbeitet, sondern alle Threads eines Blocks an der Verarbeitung eines Jobs beteiligt sind. Zwar würde das ggf. wiederum Synchronisation und Kommunikation erfordern, in dieser Variante könnten dafür jedoch die leistungsfähigen Block-Mechanismen für Threads des gleichen Blocks angewendet werden (vgl. Abschnitt 3.2).

Das Problem ist, dass die Daten, die zu den COEX-Operationen eines Jobs gehören, im Allgemeinen nicht aufeinanderfolgend im Speicher liegen (vgl. Abbildung 5.6). Diese Verteilung führt wiederum zu non-coalesced Zugriffen auf den globalen Speicher, wodurch die Leistung der Implementierung so vermindert wird, dass dieser Ansatz nicht gewinnbringend ist.

Trotzdem gibt es einen Weg durch die Nutzung des Shared Memory die Leistung zu erhöhen: Soll eine Sequenz der Länge 2^k sortiert werden, so werden in den Steps $s, s-1, \ldots, 1$ jeder Stage p gerade 2^{k-s} Sequenzen der Länge 2^s völlig unabhängig voneinander verarbeitet. Beispielsweise in Abbildung 5.6 ist die Länge der Eingabesequenz 8, in den Steps 2 und 1 werden jedoch die beiden Subsequenzen der Länge 4 ($(0, 1, 2, 3)$ und $(4, 5, 6, 7)$) unabhängig voneinander verarbeitet.

Wenn nun eine solche Subsequenz der Länge 2^s in Step s vollständig in den Shared Memory passt, so transferiert ein Block zunächst die ganze Subsequenz in den Shared Memory. Anschließend verarbeiten die Threads dieses Blocks die Steps $s, s-1, \ldots, 1$ entsprechend des oben vorgestellten Ansatzes, greifen dabei aber auf den Shared Memory und nicht auf den globalen Speicher zu. Auch in diesem Fall können dann sogar die wenigen im hier verwendeten Ansatz verbliebenen notwendigen Synchronisationen bzw. Kommunikationen zwischen Threads durch die effizienten Block-Mechanismen durchgeführt werden, so dass während der Verarbeitung der letzten s Steps keine weiteren Kernel Launchs oder Zugriffe auf den globalen Speicher notwendig sind.

In der vorliegenden Implementierung werden (in Abhängigkeit vom verwendeten Datentyp des Schlüssels bzw. davon, ob Schlüssel-Wert-Paare sortiert werden sollen) bis zu $s = 9$ Steps vollständig im Shared Memory verarbeitet. Da jede der ersten 9 Stages gerade 9 oder weniger Steps hat, können sogar die ersten 9 Stages vollständig innerhalb eines einzelnen Kernel Launchs verarbeitet werden.

```
1 BS(key* lo, int n, int dir, int stages, int firstStages,
2    int lastSteps)
3 {
4   BS_firstStages<<<grid, block, 0, 0>>>(lo, firstStages, n, dir);
5
6   for (int stage = firstStages + 1; stage <= stages; stage++)
7   {
8     for (int step = stage; step > lastStep; step -= 4)
9     {
10      BS_4_step<<<grid,block, 0, 0>>>(lo, stage, step, dir);
11    }
12
13    BS_lastSteps<<<grid,block, 0, 0>>>(lo, stage, lastSteps, dir);
14  }
15 }
```

Quellcode 5.1: Pseudocode für die optimierte Variante von Bitonic Sort

5.4 Analyse

Die Reduktion der notwendigen Synchronisation und Kommunikation in Bitonic Sort reduziert die Anzahl der notwendigen Kernel Launchs und die Anzahl der notwendigen Speicherzugriffe. Darüber hinaus kann durch die Verwendung von Shared Memory in bestimmten Stages und Steps auch von den effizienten Verfahren zur Synchronisation innerhalb eines Blocks sowie von der schnellen Kommunikation durch den Shared Memory profitiert werden, wodurch ebenfalls Kernel Launchs und Speicherzugriffe vermieden werden. Der Quellcode 5.1 zeigt den Pseudocode der optimierten Implementierung. Die ersten `firstStages` Stages werden durch `BS_firstStages`, wie in Abschnitt 5.3.3 beschrieben, verarbeitet. In späteren Stages wird wiederholt `BS_4_step` aufgerufen, wobei jeder Aufruf 4 aufeinanderfolgende Steps entsprechend der in Abschnitt 5.3.2 vorgestellten Herangehensweise verarbeitet. Die letzten `lastSteps` Steps werdem durch `BS_lastSteps` verarbeitet, wobei wieder analog zu `BS_firstStages` der Shared Memory genutzt wird. Beispielsweise für 32 bit-Integer ist `lastSteps` $= 9 =$ `firstStages`. Daraus folgt, dass für eine zu sortierende Sequenz der Länge $n = 2^k$ die Anzahl der Zugriffe auf den globalen Speicher gegeben ist durch:

$$n \cdot 2 \cdot \left(1 + \sum_{i=10}^{k} 1 + \left\lceil \frac{i-9}{4} \right\rceil\right) \tag{5.4}$$

Das bedeutet beispielsweise für eine Sequenz der Länge 2^{20}, dass die Anzahl der Speicherzugriffe pro Element in der optimierten Variante entsprechend Gleichung 5.4 nur

$$2 \cdot (1 + 4 \cdot 2 + 4 \cdot 3 + 3 \cdot 4) = 60 \tag{5.5}$$

beträgt, während in der einfachen Variante 420 Zugriffe (vgl. Gleichung 5.3) notwendig wären. Zusätzlich wird durch die Verwendung des Shared Memory in den kleineren Steps erreicht, dass keine ungünstigen Zugriffsmuster auf den globalen Speicher benutzt werden (im Falle von 32 bit, 64 bit und 128 bit Schlüsseln). Demnach ist der Speicherdurchsatz aus Sicht der Anwendung gleich dem Durchsatz aus Sicht der Hardware (beispielsweise 51.2 GB/s für eine Sequenz der Länge 2^{20} mit 32 bit-Integern auf einer GTX280-GPU). Insgesamt kann durch diese Optimierungen die Leistung der verwendeten Implementierung wesentlich erhöht werden. Dies wird im folgenden Abschnitt durch experimentell gewonnene Messergebnisse belegt.

5.5 Evaluierung

Um die Leistungsfähigkeit des Sortieralgorithmus zu testen, wurde eine große Anzahl an Tests mit verschiedenen Schlüssel-Typen, Schlüssel-Wert-Paaren und GPUs durchgeführt. Alle Tests wurden mit CUDA 4.0 durchgeführt. Da der Algorithmus zur Verwendung innerhalb des XSLT-Prozesses auf der GPU gedacht ist, wurde die Zeit zum Transferieren der zu sortierenden Sequenzen zur und von der GPU nicht berücksichtigt. Das ist auch das übliche Vorgehen in der Literatur, so dass die Implementierungen anderer Algorithmen, mit denen der hier vorgestellte Algorithmus verglichen wird, dadurch nicht benachteiligt sind. Da Bitonic Sort ein Sortiernetzalgorithmus und somit unabhängig von der Schlüsselverteilung in der zu sortierenden Sequenz ist, wurden hier nur gleichverteilte Zufallssequenzen betrachtet.

In allen Abbildungen wird an der y-Achse die Sortierrate dargestellt, das heißt die Länge der Sequenz geteilt durch die Laufzeit des Sortieralgorithmus. Die x-Achse dagegen zeigt die Länge der sortierten Sequenz.

Um die Leistungsfähigkeit der hier verwendeten Implementierung einschätzen zu können, wurden neben den Sortierraten des hier vorgestellten Algorithmus auch die Sortierraten der schnellsten sonstigen vergleichsbasierten Algorithmen dargestellt: Eine Quick Sort Implementierung von Cederman et al. [23], ein Sample Sort

Algorithmus von Leischner et al. [52], der Merge Sort Algorithmus von Satish et al. [75] und der Warp Sort Algorithmus von Ye et al. [99].

Die Abbildung 5.7 zeigt die Sortierraten für 32 bit-Integer-Sequenzen. Bitonic Sort ist der schnellste der hier getesteten Algorithmen für alle getesteten Sequenzlängen und alle GPUs. Deutlich zu sehen ist jedoch auch die stärkere Abhängigkeit von Bitonic Sort von der Sequenzlänge: Die anderen getesteten Algorithmen haben die optimale Komplexität $O(n \log n)$, während Bitonic Sort nur die Komplexität $O(n \log^2 n)$ hat. Da die x-Achsen in allen Abbildungen logarithmisch sind, ist der Effekt sehr gut zu erkennen.

Die Abbildung 5.8 liefert ein weniger einheitliches Bild. Mit der GTX280-GPU ist zwar wiederum Bitonic Sort schneller als alle anderen Algorithmen für alle Sequenzlängen. Auf der GTX480 und der GTX580-GPU ist jedoch Sample Sort von Leischner et a. schneller für Sequenzlängen größer als 2^{23}. Wie anfangs angedeutet, ist Bitonic Sort ein Algorithmus, der eher *bandwidth-bounded* ist, denn es stehen sehr viele Speichertransaktionen sehr einfachen Berechnungen gegenüber. Es steht zu vermuten, dass Sample Sort ausgewogener arbeitet. Da die hier verwendeten GPUs GTX480 und GTX580 gegenüber der GTX280 eine größere Steigerung in der Rechenleistung als in der Speicherbandbreite bieten, scheint Sample Sort mehr von diesen GPUs profitieren zu können.

Keine der anderen untersuchten Implementierungen unterstützt 128 bit-Integer. Daher können hier nur die Messwerte der Bitonic Sort Implementierung für 128 bit Integer und (128 bit-Integer, 32 bit-Integer)-Schlüssel-Wert-Paare angegeben werden.

Satish et al. haben in einer neueren Veröffentlichung auf ein neues Merge Sort [76] hingewiesen. Leider waren bisher keine näheren Informationen über das dabei verwendete Verfahren erhältlich, jedoch scheint die Leistungsfähigkeit der von Bitonic Sort ähnlich zu sein.

In einem weiteren Test wurde die optimierte gegen die nicht-optimierte Variante von Bitonic Sort für 32 bit-Integer verglichen. Die Ergebnisse sind in Abbildung 5.11 zu sehen. Die Leistungssteigerung der optimierten gegenüber der nicht-optimierten Implementierung liegt bei vier bis fünf.

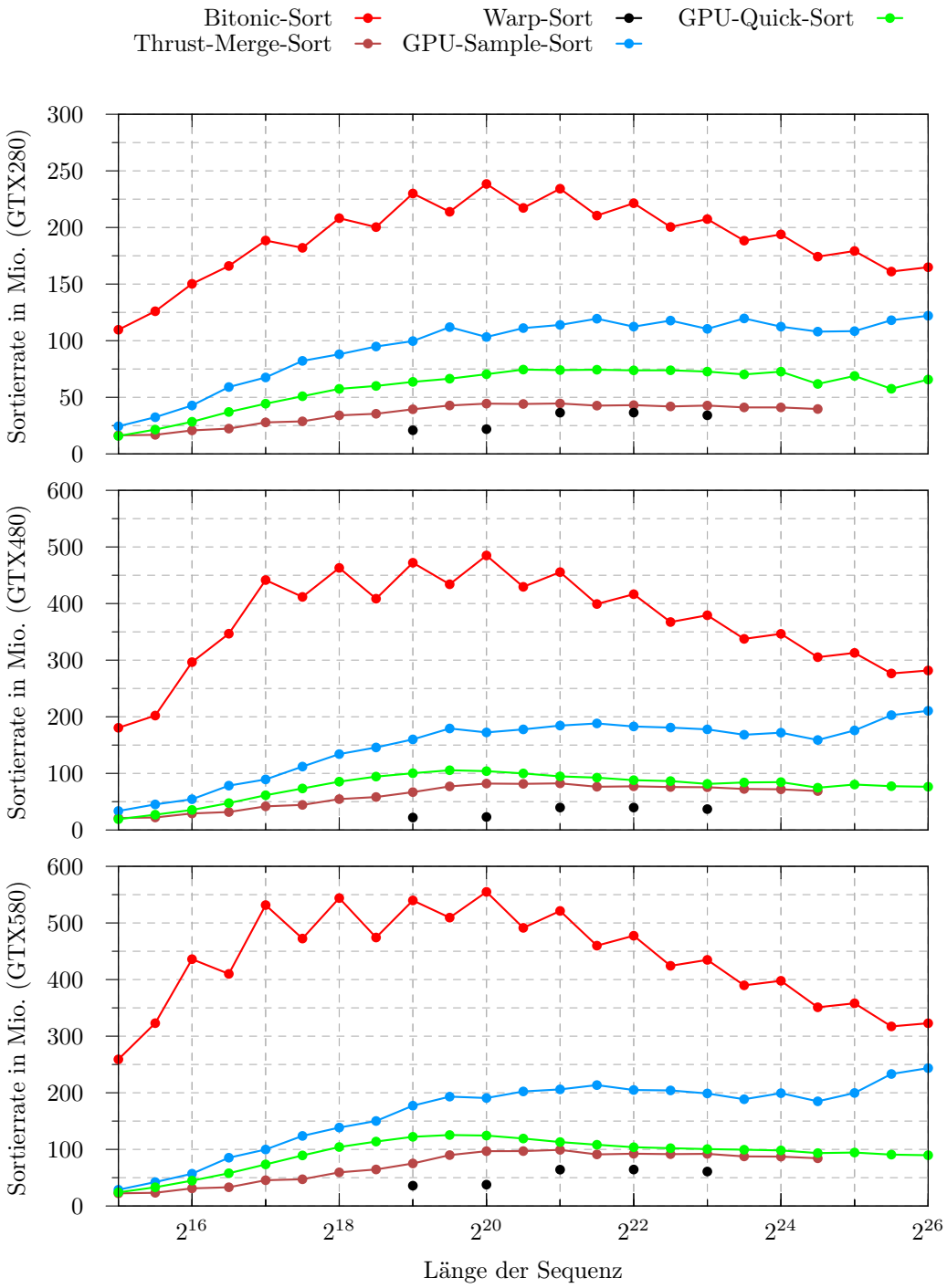


Abbildung 5.7: Evaluierung Bitonic Sort: 32 bit-Integer-Sequenzen

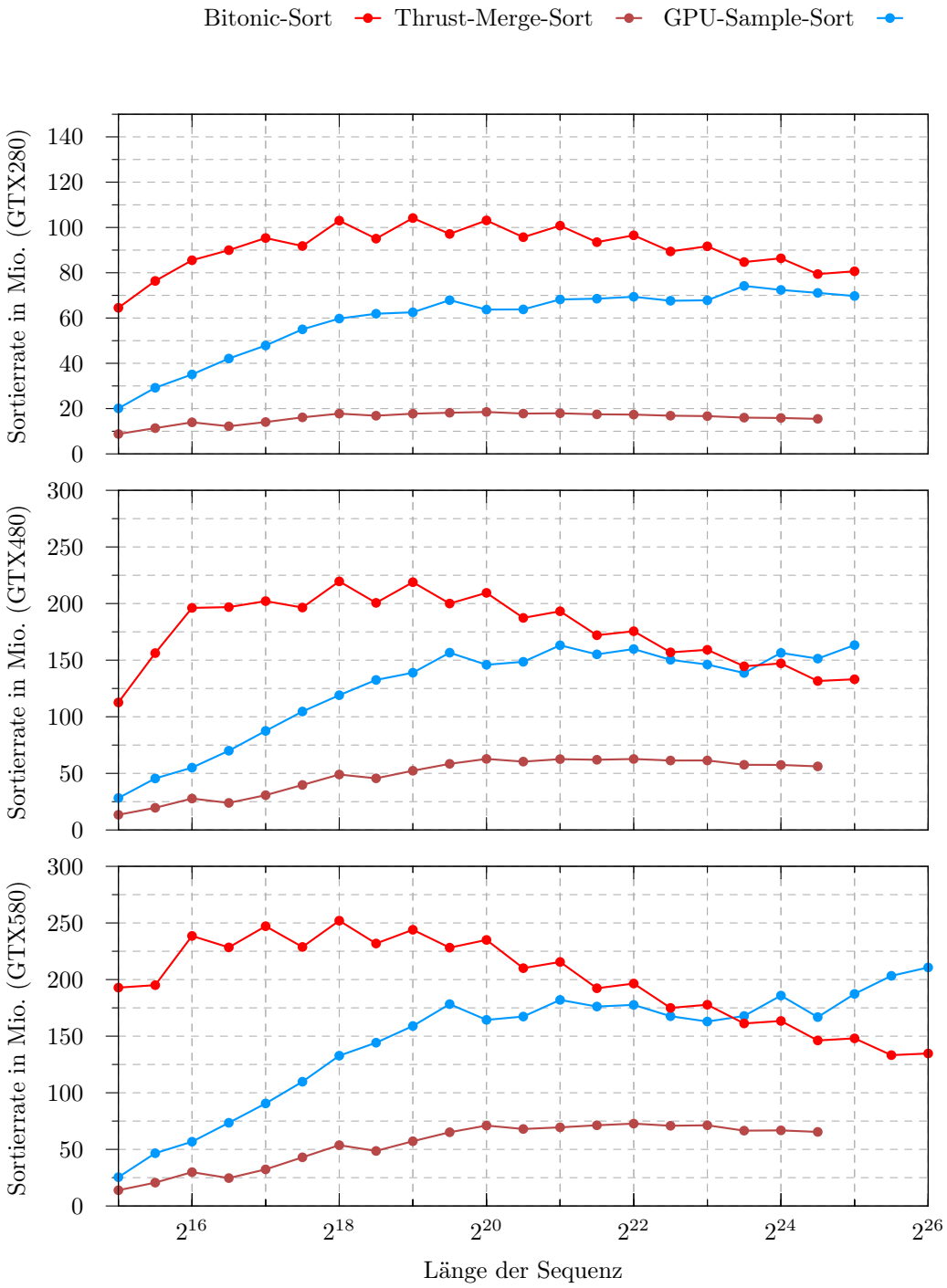


Abbildung 5.8: Evaluierung Bitonic Sort: 64 bit-Integer-Sequenzen

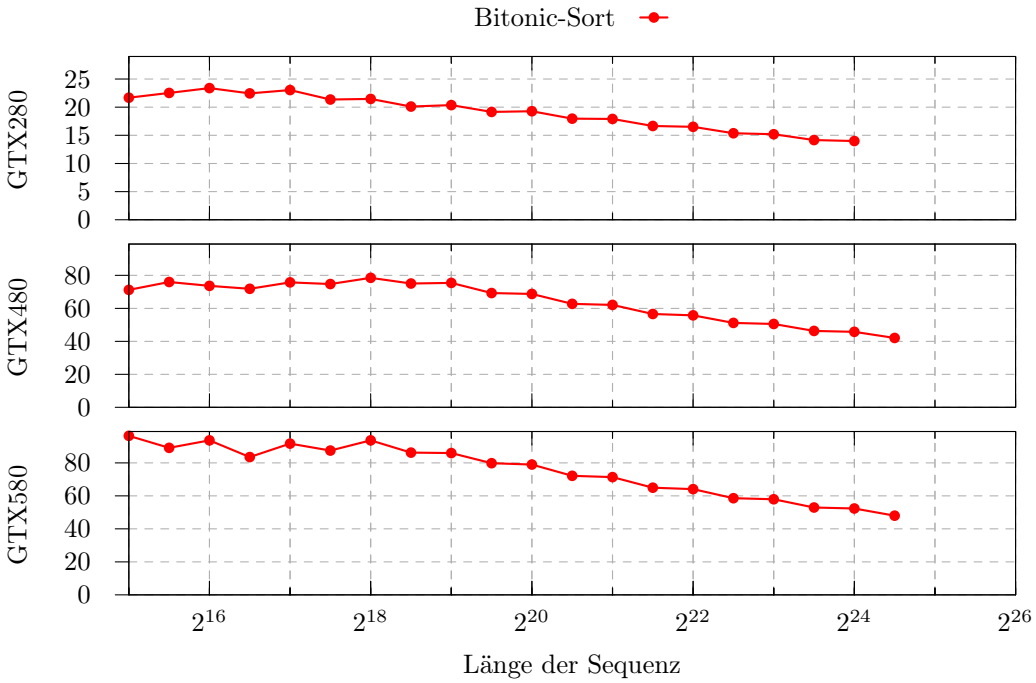


Abbildung 5.9: Evaluierung Bitonic Sort: 128 bit-Integer-Sequenzen

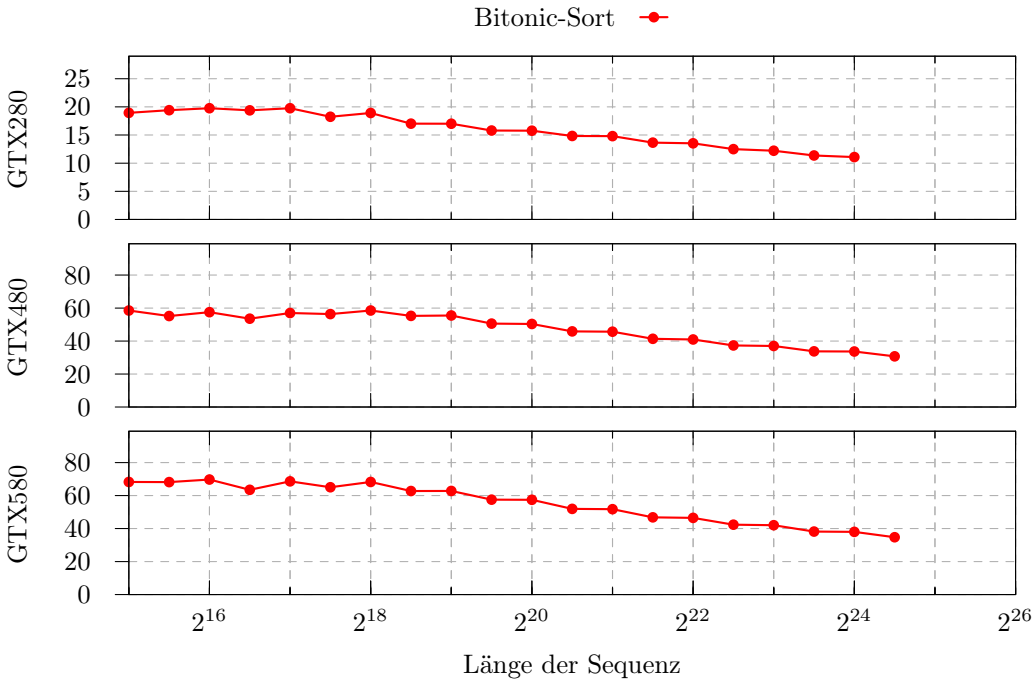


Abbildung 5.10: Evaluierung Bitonic Sort: Sequenzen von (128 bit-Integer, 32 bit-Integer)-Schlüssel-Wert-Paaren

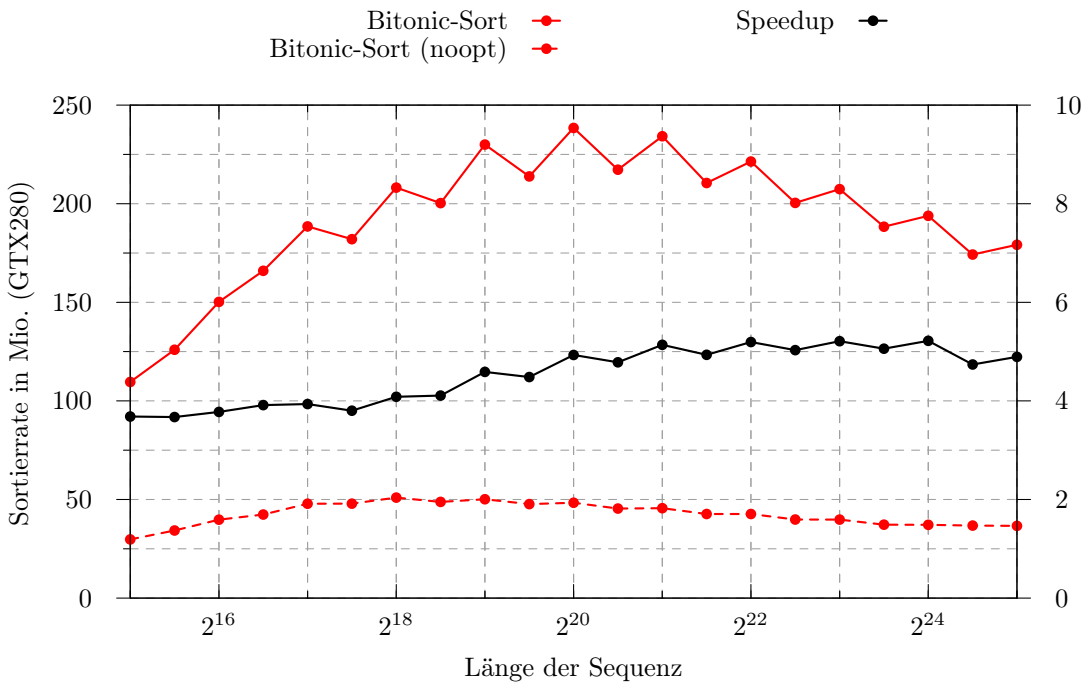


Abbildung 5.11: Evaluierung Bitonic Sort: optimierte/nicht-optimierte Variante

5.6 Zusammenfassung Bitonic Sort

Der hier vorgestellte Algorithmus hat sich in den Tests als sehr leistungsfähig erwiesen. Anders als die anderen aktuellen Algorithmen arbeitet er vergleichsbasiert und in-place, was für die Verwendung im Kontext eines XSLT-Prozessors ein deutlicher Vorteil ist. Ein genereller Nachteil dieses Algorithmus ist jedoch seine Zeitkomplexität, die ihn für längere Sequenzen weniger geeignet macht.

Es gibt zwar Ansätze [77], mit Verfahren ähnlich zu Adaptive Bitonic Sort [17] Bitonic Sort-basierte Verfahren für die GPU mit optimaler Komplexität zu entwickeln, jedoch ist der dabei entstehende Algorithmus kein reiner Sortiernetzalgorithmus mehr, und zudem arbeitet er nicht mehr in-place. Deswegen wurde er nicht im Kontext des XSLT-Prozessors eingesetzt.

KAPITEL 6

CUDA-OS

Wie im Abschnitt 4 erläutert wurde, soll in dieser Arbeit ein Ansatz ermittelt werden, mit dem mehrere XSLT-Prozesse gleichzeitig auf der GPU verarbeitet werden können. Dabei wird der GPU-XSLT-Prozessor in einem Client-Server-Umfeld eingesetzt, die verschiedenen XSLT-Prozesse liegen also nicht alle gleichzeitig vor, sondern sollen zeitversetzt auf der GPU gestartet werden können.

Dafür könnte sicherlich eine spezielle Lösung für XSLT entwickelt werden, die beispielsweise in der Lage ist, mehrere XSLT-Prozesse durch *interleaving* zu parallelisieren. Für diese Arbeit wurde jedoch auf Grund der höheren Flexibilität ein System entwickelt, das CUDA-OS, welches generisch arbeitet. Die GPU soll als allgemeiner Koprozessor mit einer gewissen Anzahl virtueller Recheneinheiten arbeiten, wobei alle Recheneinheiten eine bestimmte Menge an Funktionen (die zur Kompilier-Zeit bekannt sind) unabhängig voneinander und nebenläufig verarbeiten können. Dieser Ansatz für NVIDIAs GPUs wurde erstmals in [67] veröffentlicht und soll im Folgenden vorgestellt werden.

Im folgenden Abschnitt werden die Problemstellung und die Idee der Lösung vorgestellt. Im Abschnitt 6.2 werden der Ansatz im Detail erläutert und die notwendigen Bedingungen für die Umsetzung des Ansatzes genannt. Im Abschnitt 6.3 wird die Umsetzung des CUDA-OS dargelegt. Die Eigenschaften der Funktion, die vom CUDA-OS angeboten werden können, werden in Abschnitt 6.4 beschrieben. Im Abschnitt 6.5 werden die resultierenden Eigenschaften des CUDA-OS anhand geeigneter Tests evaluiert und in Abschnitt 6.6 werden die Ergebnisse zusammengefasst.

6.1 Einleitung

Die Entwicklung und Implementierung eines Kernels wird erheblich durch die SIMT-Architektur von NVIDIAs GPUs bestimmt: Der aufrufende Prozess gibt innerhalb des Funktionsaufrufs an, wie viele Threads den aufgerufenen Kernel gleichzeitig ausführen sollen. Die derzeit verfügbaren GPUs von NVIDIA können entweder genau einen oder nur wenige Kernel nebenläufig auf der GPU ausführen, und das auch nur unter speziellen Bedingungen. Zudem ist es nicht trivial, eine GPU von mehreren CPU-Prozessen oder Threads nutzen zu lassen, also Berechnungen verschiedener Prozesse effizient nacheinander auf der Grafikkarte auszuführen. Deswegen gilt im Allgemeinen, dass CUDA-fähige GPUs während der Laufzeit eines Kernels exklusiv von genau einem CPU-Prozess genutzt werden. In der Regel gilt sogar, dass die GPU exklusiv über die gesamte Laufzeit eines CPU-Prozesses diesem einen Prozess zur Verfügung steht.

Wegen der genannten Gründe werden GPUs nicht als *shared resource* eingesetzt, das heißt, dass GPGPU nur dann gewinnbringend eingesetzt werden kann, wenn die GPU durch einen einzelnen Kernel voll ausgelastet wird. In den folgenden Fällen ist GPGPU demnach häufig nicht nutzbringend:

- Berechnung von Funktionen, die nicht oder nur mit großem Mehraufwand auf massiv-datenparallele Weise formuliert werden können
- Berechnung von Funktionen, deren Aufwand zu klein ist, um die GPU hinreichend auszulasten
- Berechnung von mehreren Funktionen unbekannter Priorität und Laufzeit und unbekannter Aufrufzeit

Durch diese Einschränkungen wird der gewinnbringende Einsatz der GPU für ganze Anwendungsgebiete erschwert bzw. verhindert. Für den in dieser Arbeit angestrebten XSLT-Prozessor gelten beispielsweise gleich alle drei genannten Punkte, denn nicht alle Teile eines XSLT Prozesses können massiv-parallel ausgeführt werden, unter Umständen ist eine einfache Transformation nicht aufwändig genug um die GPU auszulasten, und die Laufzeit und Aufrufzeit des XSLT-Prozesses sind in dem hier gewählten Szenario (vgl. Abschnitt 4) unbekannt.

Ein sehr anschauliches Beispiel für die drei oben genannten Punkte ist die Verschlüsselung durch DES [27] im *Cipher Block Chaining Mode*, kurz CBC. Aufgrund

der dabei explizit gewünschten Verzahnung (*chaining*) von Berechnungen mit den Ergebnissen vorheriger Berechnungen kann dieses Verschlüsselungsverfahren nicht massiv-datenparallel ausgedrückt werden. Demnach ist dieses Verfahren nicht geeignet, um auf einer GPU, die ggf. über mehrere Hundert Recheneinheiten verfügt, ausgeführt zu werden.

Eine weniger sichere Alternative zu DES/CBC ist DES im *Electronic Code Book Mode*, kurz ECB, bei der Klartext in viele kleine Blöcke zerlegt wird, die anschließend unabhängig voneinander verarbeitet werden. Auch bei diesem Verfahren kann die GPU nur dann gewinnbringend verwendet werden, wenn genügend viele Daten verarbeitet werden müssen, das heißt, wenn der zu verschlüsselnde Klartext lang genug ist. Beispielsweise auf einer GTX280 müssen immerhin mindestens $60 \cdot 128 = 7680$ Threads verwendet werden, damit die GPU voll ausgelastet werden kann.

Doch selbst wenn eine Funktion massiv-parallel formuliert werden kann und auch genügend Verarbeitungsschritte auszuführen sind, also etwa wenn DES/ECB mit sehr großem Klartext eingesetzt wird, kann die Verarbeitung auf der GPU ungünstig sein, weil in dieser Zeit keine anderen CPU-Prozesse Zugriff auf die GPU haben und der laufende Kernel auch nicht unterbrochen werden kann. In einem Szenario, in dem beispielsweise auch Berechnungen hoher Priorität oder Echtzeitberechnung zu einem vorab nicht bekannten Zeitpunkt verarbeitet werden sollen, müsste die GPU für diese Funktionen exklusiv zur Verfügung stehen, und andere Berechnungen (möglicherweise mit unbekannter Laufzeit) könnten nicht auf der GPU durchgeführt werden.

Zusammengefasst gilt in obigem Beispiel, dass nur dann GPGPU sinnvoll eingesetzt werden könnte, wenn die GPU exklusiv einem einzelnen Prozess zur Verfügung steht, welcher Ver- oder Entschlüsselungsberechnungen mittels DES/ECB auf genügend großen Daten durchführen soll.

Obwohl diese Beschränkungen von GPGPU bekannt sind, gibt es nur wenige Arbeiten, die für dieses Problem eine Lösung suchen. Es gibt Ansätze [15, 21], bei denen viele Aufgaben gleichen Typs zu verarbeiten sind, so dass GPU-Threads Aufgaben aus einer gemeinsamen Aufgabenmenge entnehmen können, bis diese Menge leer ist. Diese Ansätze funktionieren jedoch nur dann, wenn alle diese Aufgaben vom gleichen Typ sind und zudem bereits vor dem Start bekannt sind.

Im Folgenden soll ein generischer Ansatz vorgestellt werden, der diese Einschränkungen teilweise überwindet, es im Kontext dieser Arbeit also ermöglicht, mehrere

XSLT-Prozesse unabhängig voneinander nebenläufig auf der GPU zu verarbeiten, wobei der einzelne XSLT-Prozess ebenfalls parallel ausgeführt wird.

Eine Art GPGPU-Service, im Folgenden CUDA-OS genannt, startet dazu einen nicht-terminierenden Kernel, der eine bestimmte Menge an Funktionen bereit stellt. Im Gegensatz zu bisherigen Ansätzen kontrolliert das CUDA-OS den Kernel (beziehungsweise die ausführenden Threads) durch *asynchronous memory transfers*, das heißt Speicherübertragungen von der CPU zur GPU oder umgekehrt, während der Kernel ausgeführt wird.

Das CUDA-OS kontrolliert dabei den Kernel auf Blocklevel, es wird für jeden Block von CUDA-Threads individuell bestimmt, welche Funktion durch diesen Block ausgeführt werden soll. Auf der CPU-Seite stellt das CUDA-OS Schnittstellen zur Verfügung, durch die CPU Prozesse oder Threads (im Folgenden Clients genannt) auf Funktionen zugreifen können, die auf der GPU ausgeführt werden.

Durch die gleichzeitige Berechnung der verschiedenen Aufgaben mehrerer verschiedener CPU Clients kann die GPU voll ausgelastet werden, auch wenn jede einzelne Aufgabe dazu nicht ausreichen würde (also etwa wie DES mit kleinen Datenmengen oder im CBC-Modus). Dabei können die verschiedenen Clients auch die Berechnung verschiedener Funktionen aufrufen und die Funktionsaufrufe müssen auch nicht gleichzeitig vorliegen, sondern können zeitversetzt eintreffen. Dadurch können beispielsweise Anfragen bestimmter Clients priorisiert werden oder auch einige der virtuellen Recheneinheiten der GPU (das heißt Blöcke) für bestimmte Aufgaben reserviert werden.

Durch das CUDA-OS kann GPGPU auch für Anwendungsgebiete wie in der vorliegenden Arbeit eingesetzt werden, in denen die Rechenleistung der GPU von verschiedenen Funktionen beziehungsweise verschiedenen Clients gleichzeitig genutzt werden soll.

6.2 Ansatz

Wie bereits erwähnt wurde, wird die Anzahl an Threads, die einen bestimmten Kernel verarbeiten sollen, beim Aufruf des Kernels festgelegt. Die Threads haben Zugriff auf ihre eindeutigen Positionen im Block und im Grid und können daraus eindeutige IDs ableiten (Thread-ID, Block-ID) [62, Abschnitt 2.2]. Mittels dieser IDs können die Threads nicht nur unterschiedliche Daten adressieren, sondern auch bedingte Entscheidungen treffen, also auch unterschiedliche Instruktionen ausfüh-

ren. Wie in Abschnitt 3.2 erläutert wurde, können Threads unterschiedlicher Blöcke in der SIMT-Architektur von NVIDIAs GPUs durchaus ohne Leistungsverlust unterschiedlichen Instruktionspfaden folgen.

Die hier verwendete Idee ist, einen persistenten, also nicht-terminierenden Kernel, der verschiedene Funktionen als Subroutinen bereitstellt, zu verwenden. Jeder einzelne Block, der den Kernel ausführt, soll dabei als ein unabhängiger virtueller Prozessor arbeiten. Dazu liest jeder Block in einer unendlichen Schleife seinen so genannten *Function Identifier* (Funktions-ID) von einer bestimmten Stelle im Speicher aus. Dieser Wert wird „von außen“, also vom CPU-Prozess gesetzt. In Abhängigkeit von dem gelesenen Wert der Funktions-ID kann ein Block nun entscheiden, ob eine (beziehungsweise welche) Funktion ausgeführt werden muss, und kann die entsprechenden Argumente und Nutzerdaten lesen. Nachdem eine Funktion von einem Block beendet wurde, beginnt der Prozess von Neuem.

Um diesen Ansatz umzusetzen, sind einige Bedingungen zu erfüllen:

1. Jeder Block, der den Kernel ausführt, muss GPU-Zeit erhalten
2. Die Nutzerdaten (inklusive der Ergebnisse) auf der GPU müssen entsprechend ihrer Verwendung durch die Funktionen im Kernel mit den Nutzerdaten auf der CPU synchronisiert werden.
3. Die Funktions-IDs und die Funktionsargumente müssen für jeden einzelnen Block individuell übertragen werden
4. Der Zustand eines Blocks muss bekannt sein, das heißt der Zustand eines Blocks (*idle*,*running*,...) muss zu geeigneter Zeit auf geeignete Weise an den Kontrollprozess auf der CPU übertragen werden.

Die erste Bedingung kann erfüllt werden, indem einige GPU-spezifische Charakteristika beachtet werden, unter anderem die Anzahl an Registern, die ein Thread benutzen darf, und die Menge an Shared Memory, die ein Block nutzt. Die anderen Bedingungen können alle erfüllt werden durch die geschickte Verwendung von asynchronem Speichertransfer beziehungsweise von *Zero Copy* [61, Kapitel 3.1.3].

6.3 Implementierung

Die Implementierung des CUDA-OS folgt in ihrem Aufbau dem im vorherigen Abschnitt vorgestellten Ansatz. Die Abbildung 6.1 stellt eine Übersicht über die

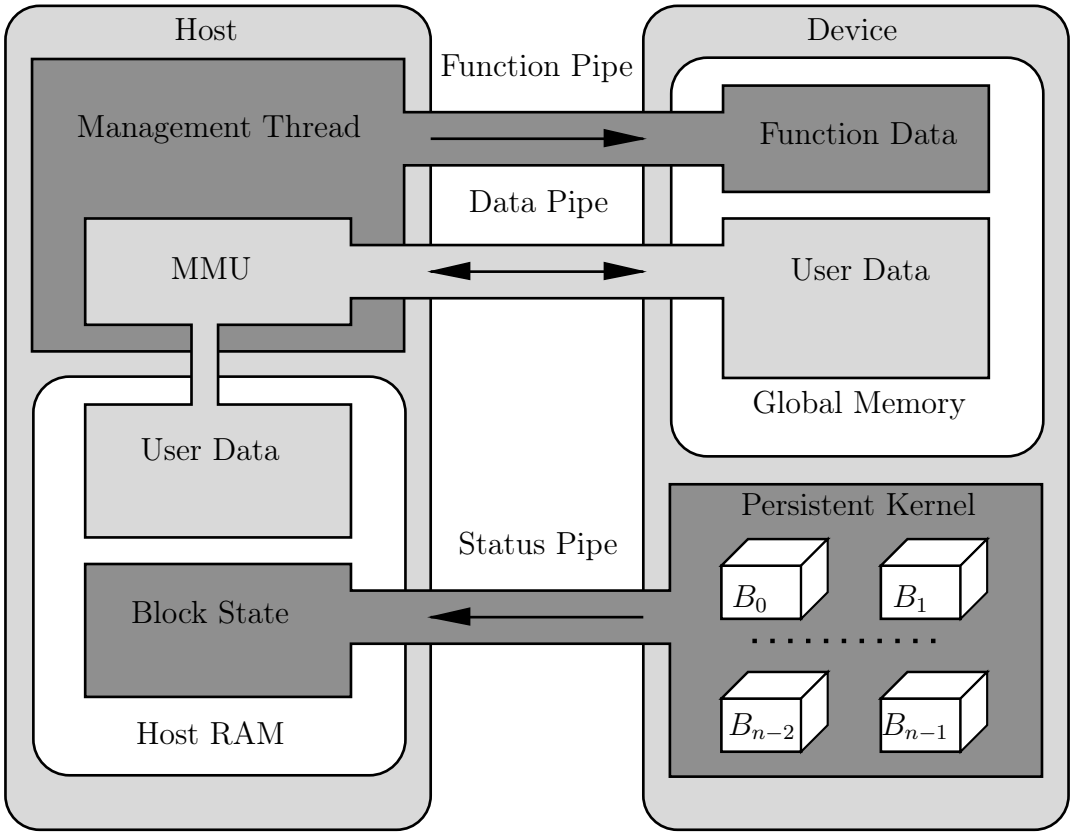


Abbildung 6.1: Schema CUDA-OS (aus [67])

Struktur des CUDA-OS dar. Die Einzelheiten werden in den folgenden Abschnitten erläutert.

Aktive Blöcke

Wenn die GPU einen Kernel mit einer bestimmten Anzahl an Blöcken und Threads verarbeitet, kann immer nur eine bestimmte Anzahl von Blöcken *active* (aktiv) sein, das heißt, nur diese Blöcke, oder genauer die Threads dieser Blöcke, erhalten Rechenzeit auf der GPU.

Grundsätzlich kann die GPU nur eine bestimmte Anzahl von Threads in einem aktiven Zustand halten. Darüber hinaus müssen auch alle Ressourcen, die von aktiven Threads oder Blöcken genutzt werden (Register, Shared Memory etc), gleichzeitig zur Verfügung stehen. Aus diesen beiden Beschränkungen (die Anzahl der Threads sowie die Menge an Ressourcen) folgt eine Beschränkung in der Anzahl

der Blöcke, die gleichzeitig aktiv sein können [62, Abschnitt 5.2].

Am Anfang der Verarbeitung eines Kernels wird entsprechend der genannten Beschränkungen eine bestimmte Menge von Blöcken auf den Zustand aktiv gesetzt, die übrigen Blöcke sind inaktiv. Nur wenn ein Block (der folglich gerade aktiv ist) terminiert, das heißt alle zugehörigen Threads terminieren, kann der Zustand eines anderen Blocks von inaktiv auf aktiv gesetzt werden. Wenn aber ein persistenter Kernel gestartet wird, dessen Threads nie terminieren, so kann auch niemals der Zustand eines Blocks von inaktiv auf aktiv wechseln.

Insgesamt bedeutet dies, dass die Anzahl von Blöcken, die einen persistenten Kernel ausführen, beschränkt ist durch die Anzahl der Blöcke, die (für diesen speziellen Kernel) gleichzeitig aktiv sein können, denn nur so bekommen überhaupt alle Blöcke Rechenzeit der GPU. In der Abbildung 6.1 sind die Blöcke dargestellt durch die Boxen $B_0, \ldots, B_{n-1}$.

In den in Abschnitt 6.5 vorgestellten Tests wurde eine GTX480 bzw. eine GTX580 GPU verwendet, wobei 4 Blöcke pro Multiprozessor gestartet wurden. Demnach kann das CUDA-OS im Falle der GTX480 mit 60 Blöcken, im Falle der GTX580 mit 64 Blöcken (vgl. [62, Tabelle A-1]) betrieben werden.

Speicherverwaltung

Wie anfangs dargelegt wurde, ist es nicht oder nur unzureichend möglich, Kernel auf der GPU nebenläufig auszuführen. Neuere GPUs von NVIDIA unterstützen allerdings den so genannten *asynchronous memory transfer* [62, Abschnitt 3.2.5], durch den Daten zwischen dem Speicher der CPU und dem Speicher der GPU transferiert werden können, während ein Kernel auf der GPU verarbeitet wird. Es ist also grundsätzlich möglich, Funktionsargumente und Funktions-IDs, Nutzerdaten und auch den Zustand der Blöcke zwischen CPU und GPU zu übertragen, während der persistente Kernel verarbeitet wird.

Es ist möglich, Speicher auf der GPU zu allozieren (`cudaMalloc()`), während ein Kernel verarbeitet wird. Es ist jedoch nicht möglich, diesen Speicher zur Laufzeit des Kernels durch `cudaFree()` wieder freizugeben. Wenn also zur Laufzeit eines Kernels wiederholt Speicher auf der GPU alloziert wird, steigt der Speicherverbrauch so lange, bis kein weiterer Speicher mehr zur Verfügung steht und das System nicht weiterarbeiten kann. Für den hier vorgestellten persistenten Kernel wird deshalb eine zusätzliche Speicherverwaltung (*Memory Management Unit* (MMU)) benötigt, die unabhängig von den gegebenen Funktionen `cudaMalloc()`

und `cudaFree()` arbeitet.

In der hier vorgestellten Implementierung der MMU wird einmalig beim Start des CUDA-OS eine bestimmte Menge Speicher sowohl auf der CPU wie auch auf der GPU alloziert. Während der Laufzeit des Kernels hat die MMU die Aufgabe, den Speicher der CPU und den Speicher der GPU synchron zu halten. In der Abbildung 6.1 wird dies durch die Boxen für *User data* auf der *host*- und der *device*-Seite dargestellt, die durch die *Data Pipe* über die MMU miteinander verbunden sind. Wenn nun ein Client eine Anfrage an das CUDA-OS stellt, bei der Nutzerdaten benötigt werden, so reserviert die MMU den benötigten Speicher sowohl auf der CPU- wie auch auf der GPU-Seite und transferiert die Daten des Clients in beide Speicherbereiche. Wenn die Anfrage des Clients abgearbeitet ist, werden zunächst die Daten von der GPU zur CPU (jeweils im von der MMU kontrollierten Speicher) transferiert und anschließend zurück in den Speicher des Client kopiert. Danach wird der benötigte Speicher wieder freigegeben.

Bei diesem Ansatz mit CUDA benötigt die MMU so genannten *page-locked* [62, Abschnitt 3.2.5] Speicher auf der CPU-Seite, um den asynchronen Speichertransfer durchzuführen. Der Client hingegen ist unabhängig von den Bedingungen für asynchronen Speichertransfer.

Funktionsargumente und Status der Blöcke

Um die Blöcke, die den persistenten Kernel ausführen, zur Laufzeit zu kontrollieren, müssen Funktionsargumente und Funktions-IDs übertragen werden. Außerdem müssen die Blöcke ihren Status an die CPU übertragen können.

Wie oben ausgeführt wurde, werden die Nutzerdaten durch eine dedizierte MMU behandelt. Es ist selbstverständlich möglich, auch die Argumente, IDs usw. durch diese MMU behandeln zu lassen. Allerdings ist durch die Struktur des CUDA-OS immer bekannt, welche Funktionen überhaupt angeboten werden, und durch die in Abschnitt 6.3 dargelegten Gründe ist auch die Anzahl (bzw. die obere Schranke derselben) der ausführenden Blöcke bekannt. Dadurch ist es möglich, den zusätzlichen Aufwand durch die Verwendung der MMU zu vermeiden, einfach indem für jeden Block der benötigte Speicher für den Zustand, die Funktions-IDs und die Argumente jeder Funktion zu Beginn einmal alloziert wird.

Die Abbildung 6.1 illustriert die *Function Pipe*, durch die die Funktionsargumente und -IDs übertragen werden. In der *Status Pipe* dagegen übertragen die Blöcke ihren Status an die CPU. Dies geschieht durch *zero copy*, eine Variante des

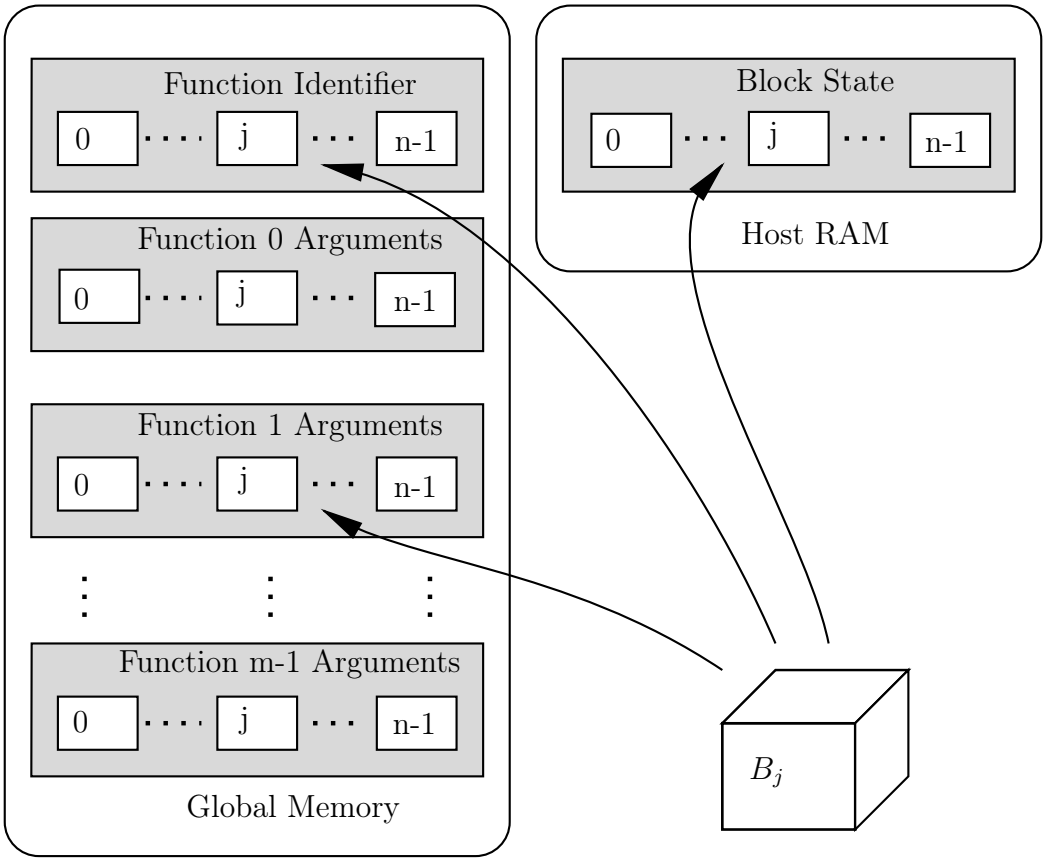


Abbildung 6.2: Schema der Funktionsargument-Verwaltung im CUDA-OS

asynchronen Speichertransfers, der durch die GPU aufgerufen werden kann.

Die Abbildung 6.2 zeigt das Prinzip, nach dem ein Block die Funktions-IDs, die Argumente und seinen Status behandelt. Jeder Block $B_j, j \in \{0, \ldots, n-1\}$ liest (wiederholt in einer Schleife) an der ihm zugeordneten Speicherstelle die Funktions-ID aus. Wird kein valider Wert gelesen, so wird der Vorgang wiederholt. Bezeichnet die Funktions-ID dagegen eine der Funktionen $0, \ldots, m-1$, so liest der Block die entsprechenden Funktionsargumente (die ggf. auch einen Zeiger zu den Nutzerdaten enthalten) an der für ihn für diese Funktion reservierten Stelle und führt die Funktion aus. Wenn der Block die Funktion beendet hat, wird der Status des Blocks an die entsprechende Stelle im Speicher der CPU geschrieben. Danach wird der ganze Vorgang wiederholt.

6.4 Kernel & Device Functions

```
1 __device__ void MatMult(Mat A, Mat B, Mat C){...}
2 __device__ void Sorting(List a) {...}
3
4 __global__ void kernel(int* func, arg** a) {
5    while ( true ) {
6       arg var;
7       switch ( func[blockIdx.x] ) {
8          case FUNC_0:
9             var = arg[0][blockIdx.x];
10            MatMult(var.A, var.B, var.C);
11            break;
12         case FUNC_1:
13            var = arg[1][blockIdx.x];
14            Sorting(var);
15            break;
16         case FUNC_QUIT:
17            return 0;
18      }
19   }
20 }
```

Quellcode 6.1: Pseudocode für persistenten Kernel

Der persistente Kernel besteht im Wesentlichen aus einer unendlich laufenden Schleife und einer Menge von Subroutinen, die hier als *device functions* implementiert werden. Device Funktions werden durch `__device__` gekennzeichnet und werden auch auf der GPU ausgeführt. Im Gegensatz zu Kerneln, die durch `__global__` gekennzeichnet sind, können diese Funktionen jedoch nicht aus dem CPU-Programm gestartet werden, sondern können nur von Kerneln aufgerufen werden

Der Quellcode 6.1 zeigt den Pseudocode für einen solchen Kernel, der Funktionen für Matrixmultiplikation und Sortierung anbietet.

Bei der Implementierung der Funktionen, die durch das CUDA-OS angeboten werden sollen, müssen zwei hauptsächliche Konzepte beachtet werden.

Problemgröße & Anzahl der Threads

Zum einen ist es eine gängige Herangehensweise beim Implementieren von datenparallelen CUDA-Funktionen, den Kernel so zu gestalten, dass der Programmcode im Ablauf des Programms die Größe des Eingabeproblems kaum berücksichtigt.

Stattdessen ist es üblich, die Anzahl der gestarteten Blöcke an die Problemgröße anzupassen. Der Quellcode 3.1 in Abschnitt 3.2 folgt beispielsweise diesem Ansatz: Bei der Addition zweier Matrizen wird die Anzahl der gestarteten Threads an die Anzahl der zu addierenden Elemente angepasst, in jedem Fall addiert ein Thread gerade zwei Elemente. Im CUDA-OS ist es prinzipiell nicht möglich, die Anzahl der gestarteten Threads an ein konkretes Problem anzupassen, da der Kernel nicht für ein konkretes Problem gestartet wird. Zur Verwendung im CUDA-OS müssen die Funktionen also so gestaltet werden, dass beliebige Problemgrößen mit konstanter Anzahl Threads behandelt werden können. Der Quellcode 6.2 zeigt beispielhaft den Code für einen Block des CUDA-OS, bei dem durch die Verwendung eines *Wrapper* eine Matrixaddition mit fester Anzahl Threads für beliebige Problemgrößen angepasst wurde.

```
1 __device__ void MatAdd(Mat A, Mat B, Mat C)
2 {
3    width  = B.width  / blockDim.x;
4    height = A.height / blockDim.y;
5
6    for(int vX = 0; vX < width; vX++)
7       for(int vY = 0; vY < height; vY++)
8          MatAddKernel(A, B, C, vX, vY);
9 }
```

Quellcode 6.2: Pseudocode für einen *Wrapper*

Synchronisation

Zum anderen werden Kernel-Aufrufe üblicherweise zum „Synchronisieren“ von Threads benutzt: Soll zwischen zwei Teilaufgaben eine für alle beteiligten Threads gültige Barriere installiert werden, so werden die Teilaufgaben oft in je einem separaten Kernel behandelt und auf der CPU-Seite entsprechend der nachfolgende Kernel erst gestartet, wenn der vorhergehende Kernel vollständig verarbeitet wurde (vgl. Bitonic Sort in Abschnitt 5). Dieser Ansatz lässt sich so natürlich nicht im CUDA-OS umsetzen, da nur ein permanent laufender Kernel zur Verfügung steht. Da im CUDA-OS jedoch Funktionen blockweise verarbeitet werden, können stattdessen die für die Threads eines Blocks definierten effizienten Synchronisationsfunktionen `__syncthreads()` bzw. `__threadfence()` genutzt werden (vgl. 3.2).

6.5 Evaluierung

Für die Tests des CUDA-OS wurden zwei einfache Funktionen implementiert, die bestimmte Eigenschaften des CUDA-OS gut verdeutlichen. Zum einen wurde ein einfaches Bitonic Sort implementiert, welches dem einfachen Ansatz (angepasst an das CUDA-OS) in Abschnitt 5.3 entspricht. Zum anderen wurde das in [62] gegebene Verfahren für Matrixmultiplikation mit Verwendung von Shared Memory an die Bedingungen des CUDA-OS angepasst und implementiert.

Diese beiden Funktionen repräsentieren gut zwei unterschiedliche Funktionsklassen: Die (hier verwendete) Matrixmultiplikation ist ein massiv datenparalleles Verfahren, bei dem sehr viele Threads eingesetzt werden können. Eine Synchronisation zwischen Threads verschiedener Blocks ist auch in der ursprünglichen Implementierung nicht notwendig. Zwar ist das hier verwendete Sortierverfahren ebenfalls massiv datenparallel, jedoch ist ein hoher Synchronisationsaufwand (vgl. Abschnitt 5.3) notwendig.

Das CUDA-OS ist, wie eingangs beschrieben, entworfen worden, um mehrere Prozesse gleichzeitig zu berechnen. Daher kann es grundsätzlich einen einzelnen Prozess nicht so schnell verarbeiten wie ein dedizierter Kernel. Um die Leistungsfähigkeit des CUDA-OS einzuschätzen, muss daher der Durchsatz gemessen werden, das heißt die Anzahl an Prozessen, die in einer bestimmten Zeit verarbeitet werden kann. In unseren Tests werden also immer große Mengen an Prozessen verarbeitet, wobei im Fall der Verwendung des CUDA-OS immer mehrere der Prozesse gleichzeitig laufen, während im anderen Fall die Prozesse nacheinander gestartet werden.

Für die Matrixmultiplikation ist zu erwarten, dass durch die Verwendung des CUDA-OS ein Leistungsverlust entsteht, denn da eine einzelne Matrixmultiplikation die GPU sehr gut auslastet, wird auch eine Sequenz einzelner Matrixmultiplikationen die GPU sehr gut auslasten. Der zusätzliche Aufwand, der dagegen im CUDA-OS entsteht, wird zu einem Leistungsverlust führen. Für das Sortierverfahren gilt natürlich auch, dass das CUDA-OS einen zusätzlichen Aufwand bedeutet. Hier allerdings kann davon profitiert werden, dass Synchronisation durch Kernel-Aufrufe durch effiziente Synchronisation innerhalb eines Blocks ersetzt werden kann, da im CUDA-OS immer nur ein Block eine Funktion ausführt.

Um das CUDA-OS zu testen, wurde ein System mit einer Intel Core I7 960@3.2GHz CPU mit 4 Kernen genutzt. Es wurden die Grafikkarten GTX480

sowie GTX580 getestet. Das Betriebssystem war ein Ubuntu 10.04 mit g++ 4.3.3 Compiler und CUDA 4.0.

Am Anfang des Tests des CUDA-OS wurde zunächst das CUDA-OS (inklusive persistentem Kernel) gestartet. Danach wurden unabhängige CPU-Threads erzeugt, die entweder eine Matrixmultiplikation oder die Sortierung einer Sequenz durchzuführen hatten und einen dementsprechenden Aufruf am CUDA-OS starteten. Wenn die Multiplikation oder die Sortierung eines Threads abgeschlossen ist, so beendet sich dieser. Um den Durchsatz zu messen, wurde in den Tests die Anzahl der CPU-Threads gezählt, die eine Aufgabe vollständig verarbeitet hatten.

In unseren Tests wurden zufällige Integersequenzen der Länge 2^{20} sortiert beziehungsweise zwei Matrizen mit je $2^{10} \cdot 2^{10} = 2^{20}$ Elementen multipliziert. Während der Tests wurden zwei Parameter variiert:

1. Die *Utilisierung*, das heißt die Anzahl der gleichzeitig im CUDA-OS verarbeiteten Prozesse (bei konstanter Anzahl gestarteter Blöcke)

2. Das *Verhältnis* (#`Sortierung`:#`Multiplikation`) der Anzahl der verschiedenen Prozesstypen zueinander

Die Utilisierung ist also die Anzahl der Prozesse, die vom CUDA-OS momentan bearbeitet werden und noch nicht beendet sind. In unseren Tests wurden für Sortierung und Multiplikation die Utilisierungen von 5 bis 60 (GTX480) bzw. 64 (GTX580) und die Verhältnisse $20{:}0, 19{:}1, \ldots, 1{:}19, 0{:}20$ getestet.

Um eine bestimmte Utilisierung x zu erreichen, werden einfach x CPU-Threads gestartet, die ihrerseits Prozesse am CUDA-OS aufrufen. Ist ein CPU-Thread (bzw. der zugehörige CUDA-OS-Prozess) fertig, so wird ein neuer CPU-Thread gestartet.

Um ein bestimmtes Verhältnis $a{:}b$ zu erreichen, werden einfach die gestarteten Sortierungen (`#S`) und die gestarteten Multiplikationen (`#M`) gezählt und, wenn ein CUDA-OS-Prozess beendet wurde, wird der nächste Prozess so gewählt, dass #`S`:#`M` möglichst dicht an $a{:}b$ liegt.

Die beiden Parameter Utilisierung und Verhältnis wurden in den Tests verändert, ohne das CUDA-OS zu unterbrechen, so dass das CUDA-OS für die Tests mehrere Stunden ununterbrochen in Betrieb war.

Utilisierung

Die Abbildung 6.3 zeigt die Messwerte bezüglich der Utilisierung (x-Achse) für eine GTX480 GPU. An der y-Achse ist die Anzahl der Clients abgebildet, die in

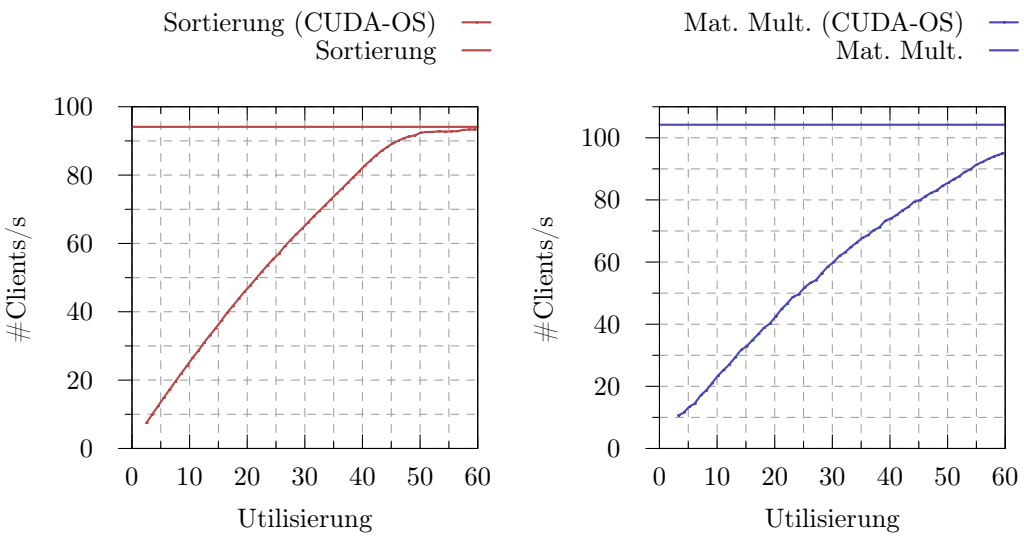


Abbildung 6.3: Evaluierung CUDA-OS: Utilisierung der GPU (GTX480)

einer Sekunde vom CUDA-OS verarbeitet werden können, das heißt die Anzahl der Sortierfunktionen beziehungsweise die Anzahl der Matrixmultiplikationen pro Sekunde. Um die Leistungsfähigkeit des CUDA-OS einschätzen zu können, wurde auch die Laufzeit der ursprünglichen, nicht für das CUDA-OS angepassten Funktionen Sortierung und Matrixmultiplikation gemessen und daraus ein maximaler sequentieller Durchsatz berechnet, der ebenfalls (als Konstante) in der Abbildung dargestellt ist.

Es ist gut zu sehen, dass für beide Funktionen mit dem CUDA-OS eine gute Skalierung bezüglich der Utilisierung (bis zu einem bestimmten Wert) erreicht wird: Je mehr Prozesse gleichzeitig verarbeitet werden, desto höher ist die Gesamtleistung (der Durchsatz) des Systems. Wie erwartet bleibt die Leistung der Matrixmultiplikation im CUDA-OS hinter derjenigen einer reinen Multiplikationsroutine zurück. Dagegen scheint im Fall der Sortierung der zusätzliche Aufwand durch das CUDA-OS nahezu durch die effizientere Synchronisierung ausgeglichen werden zu können. Etwa ab einer Utilisierung von 50 der vorhandenen 60 Blöcke ist der mit dem CUDA-OS erreichte Durchsatz sehr dicht am ohne CUDA-OS erreichten Durchsatz.

In Abbildung 6.4 werden die Messwerte für die GTX580 GPU dargestellt. Der qualitative Verlauf ist für beide GPUs gleich. Eine interessante Eigenschaft der Sortierfunktion ist, dass sie ihre Leistung auf beiden GPUs schon bei einer nicht-

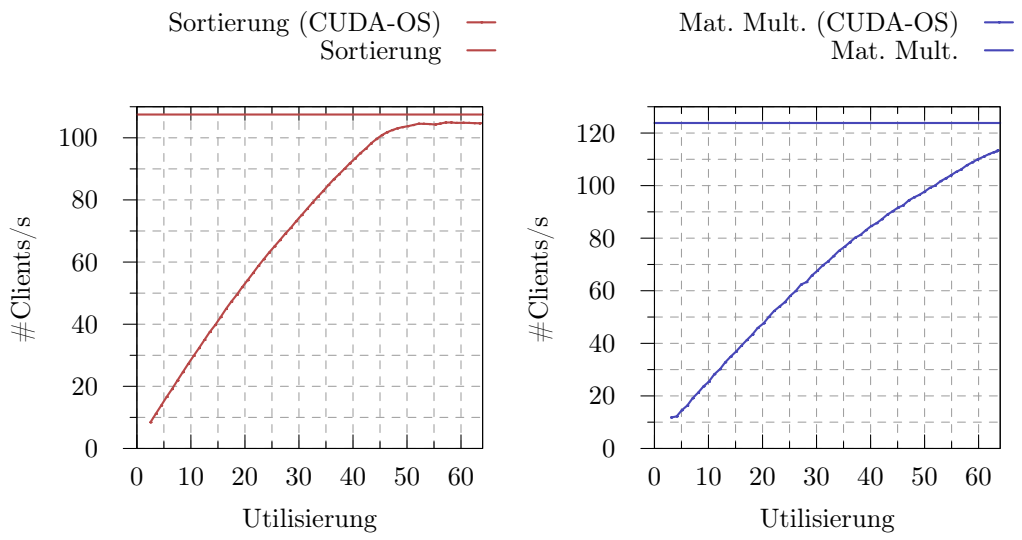


Abbildung 6.4: Evaluierung CUDA-OS: Utilisierung der GPU (GTX580)

maximalen Utilisierung erreicht, während die Leistung der Multiplikationsfunktion weiter steigt. Ein mögliche Erklärung ist, dass die Sortierfunktion erheblich mehr Speichertransfers vom globalen Speicher der GPU zu den Prozessoren benötigt als die Matrixmultiplikation. In diesem Fall nämlich wäre die Gesamtleistung durch den Speicherbus begrenzt, was das starke Abknicken der Leistungskurve ab einer bestimmten Utilisierung erklären könnte.

Verhältnis

Im Gegensatz zu den Tests im vorherigen Abschnitt wurde in den Tests, die in diesem Abschnitt beschrieben werden, immer die höchstmögliche Utilisierung verwendet. Statt der Utilisierung wurde hier das oben beschriebene Verhältnis zwischen den Anzahlen der beiden verschiedenen Funktionen verändert. Mit diesen Tests soll geprüft werden, wie sich die Gesamtleistung des CUDA-OS verändert, wenn verschiedene Aufgaben in verschiedenen Mischungsverhältnissen bearbeitet werden müssen.

Die Abbildung 6.5 zeigt die Messwerte, die mit einer GTX480 GPU gemessen wurden, in Abbildung 6.6 sind die Ergebnisse mit einer GTX580 GPU zu sehen. Die beiden x-Achsen geben den prozentualen Anteil der Sortierfunktionen (unten) beziehungsweise der Matrixmultiplikation (oben) an der gesamten Menge verarbeiteter Funktionen an. Die y-Achse dagegen zeigt für beide Funktionen die

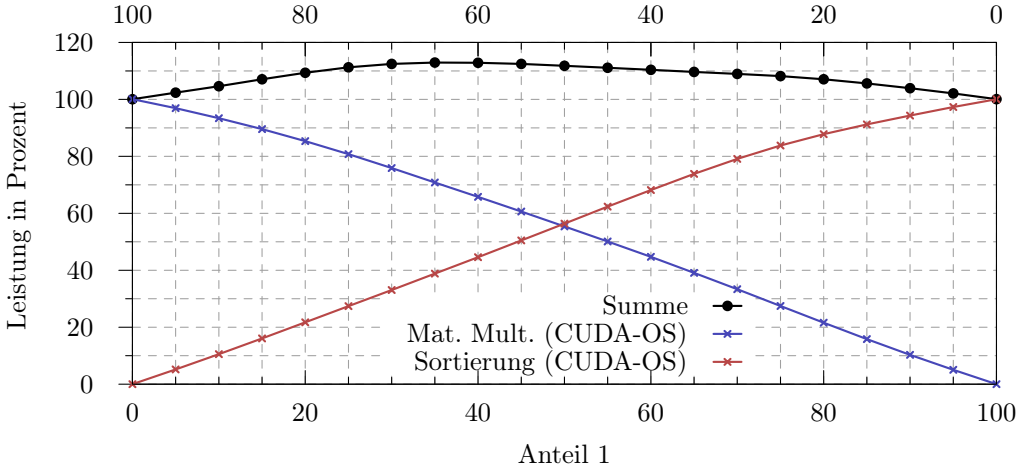


Abbildung 6.5: Evaluierung CUDA-OS: Verhältnisse der Funktionsaufrufe (GTX480)

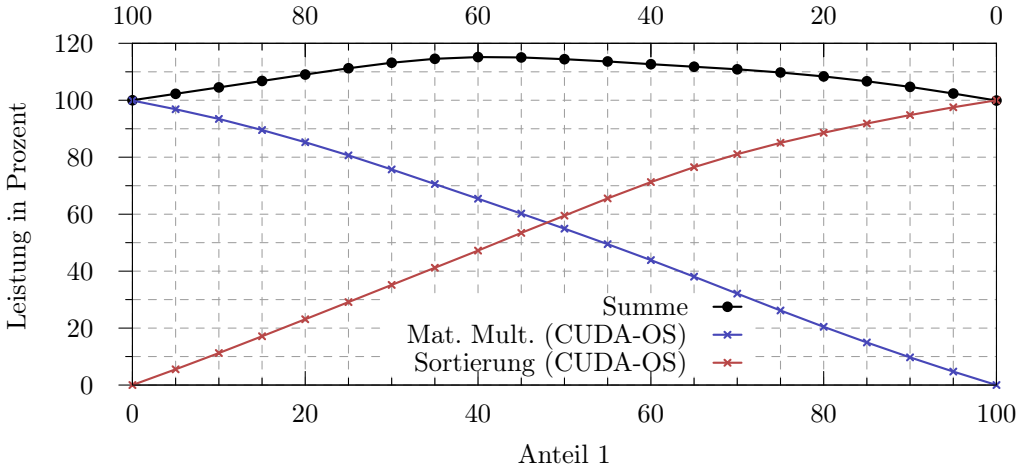


Abbildung 6.6: Evaluierung CUDA-OS: Verhältnisse der Funktionsaufrufe (GTX580)

prozentuale Leistung bezüglich der maximal erreichbaren Leistung.

Die maximal erreichbare Leistung einer Funktion im CUDA-OS ist diejenige, die erreicht wird, wenn ausschließlich diese Funktion mit voller Utilisierung verarbeitet wird (vgl. Abbildungen 6.3, 6.4). Entspricht beispielsweise die volle Leistung einer Funktion f gerade einem Durchsatz von 200 Funktionsaufrufen pro Sekunde, so wird dieser Wert mit 100% angesetzt. Werden nun Funktionsaufrufe von f mit Funktionsaufrufen irgendeiner anderen Funktion gemischt und es werden beispielsweise nur noch 100 abgearbeitete Funktionsaufrufe der Funktion f pro Sekunde erreicht, so ist die Leistung bezüglich f gerade 50%.

Für jedes Mischungsverhältnis wurde die Summe der prozentualen Leistungen beider Funktionen als die Gesamtleistung des Systems in die Abbildung eingefügt. Das wichtigste Ergebnis dieses Tests besteht darin, dass das CUDA-OS keine Leistung verliert, wenn verschiedene Funktionen nebenläufig ausgeführt werden. Im Gegenteil steigt die Gesamtleistung sogar, wenn verschiedene Funktionen gemischt werden. Das könnte dadurch erklärt werden, dass verschiedene Funktionen durch unterschiedliche Parameter primär in der Leistung beschränkt werden. Ist etwa die Sortierfunktion durch die Speichertransferrate beschränkt (das heißt Blöcke warten auf Speichertransfers), so können die Blöcke, die Matrixmultiplikationen ausführen, dennoch rechnen. In einfachen Worten ausgedrückt zeigt dieser Test, dass, falls sehr viele Funktionen unterschiedlichen Typs berechnet werden müssen, eine höhere Gesamtleistung erzielt wird, wenn die verschiedenen Funktionen gemischt werden, statt alle Funktionen eines Typs und danach alle Funktionen eines anderen Typs zu berechnen.

6.6 Zusammenfassung CUDA-OS

In diesem Abschnitt wurde ein Ansatz aufgezeigt, wie eine GPU von NVIDIA als geteilte Ressource von verschiedenen CPU-Prozessen gleichzeitig genutzt werden kann. Unter der Voraussetzung, dass genügend viele Prozesse zur Verarbeitung vorliegen, hat dieses System beim Verarbeiten massiv-paralleler Funktionen nur geringe Leistungsverluste gegenüber einem dedizierten Kernel.

Unter Umständen können mit dem CUDA-OS auch Leistungsgewinne erzielt werden, etwa wenn die Funktionen nicht massiv-parallelisierbar sind (DES/CBC). Ebenfalls kann dieses System gewinnbringend eingesetzt werden, wenn Funktionen gezielt für die Voraussetzungen des CUDA-OS implementiert werden, wenn also bei der Implementierung ausgenutzt wird, dass die Funktionen immer nur einen Block nutzen werden, Synchronisation und Kommunikation also durch die effizienten Funktionen innerhalb eines Block umgesetzt werden können.

Ein weiterer Vorteil des Systems ist, dass die Grafikkarte von verschiedenen Funktionen verschiedener CPU-Prozesse ausgelastet werden kann, was mit einem herkömmlichen Ansatz nicht möglich ist.

Teil III

XPath & XSLT

KAPITEL 7

Datenstruktur

Die natürliche Repräsentation von XML-Dokumenten im Speicher des Computers ist eine Repräsentation als Baum (oder Graph) basierend auf den Eltern/Kind Beziehungen im Dokument. Dabei werden Kanten als Zeiger dargestellt. Diese Repräsentation wird beispielsweise durch DOM nahegelegt. Diese Kodierung ist intuitiv nutzbar, da sie recht gut der gewohnten Repräsentation durch XML-Textdokumente entspricht. Es existieren Bibliotheken, die diese Repräsentation verwenden. Es gibt allerdings Gründe, die im Umfeld der schnellen parallelen Verarbeitung von XML gegen diese Repräsentation sprechen.

Zum einen ist für die parallele Verarbeitung (insbesondere mit SIMD-Architekturen) eine durch Zeiger vernetzte Datenstruktur meist nicht günstig; Arraystrukturen, in denen mit gleichem Aufwand gleichzeitig auf jedes Objekt zugegriffen werden kann, sind in diesen Fällen besser geeignet. Zudem sind mehrere parallele Speicherzugriffe auf Arrays oft günstiger als parallele Zugriffe an beliebige Stellen im Speicher.

Zum anderen können bestimmte Eigenschaften von XML-Knoten in einer zeigerbasierten Repräsentation nicht algebraisch ausgedrückt werden, sondern nur als Verfahrensanweisung. Sollen beispielsweise alle Knoten ausgewählt werden, die Nachfahren eines bestimmten Knotens sind, so gibt es keine algebraische Formel, um diese Knoten in einer zeigerbasierten Repräsentation zu bestimmen.

Im Umfeld von XML-Verarbeitung mit hoher Leistung wird deshalb diese intuitive Repräsentation häufig nicht eingesetzt. Insbesondere wenn XML-Daten in Datenbanken gespeichert werden sollen, werden häufig Kodierungen eingesetzt, die

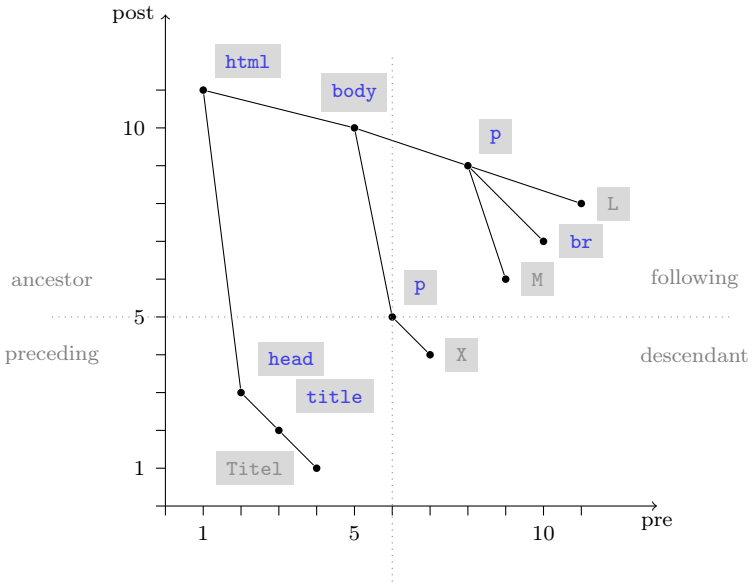


Abbildung 7.1: Dokument in Quellcode 2.1 in pre/post-Kodierung

jedem XML-Knoten ein Zahlen-Tupel zuweisen, welches die Relationen im XML-Dokument mehr oder weniger direkt beschreibt. Eine häufig verwendete Kodierung ist die *pre/post*-Kodierung. Dabei wird jedem XML-Knoten ein Zahlenpaar zugeordnet. Einfach gesagt, ist die erste Zahl (pre) die Position des öffnenden Tags eines XML-Elements in Bezug auf die öffnenden Tags der anderen Elemente in Dokumentreihenfolge. Analog dazu ergibt sich die zweite Zahl (post) durch das schließende Tag. Dadurch ergibt sich für jeden Knoten ein Zahlenpaar, welches sowohl in der ersten wie auch in der zweiten Komponente im Dokument eindeutig ist, da es nicht zwei XML-Knoten mit der gleiche Position in Dokumentreihenfolge geben kann.

Die Abbildung 7.1 zeigt das bekannte Beispieldokument aus Quellcode 2.1 (ohne Attribute) in der pre/post-Kodierung. Zur Verdeutlichung sind die Eltern/Kind Beziehungen zusätzlich eingetragen. Das durch die pre/post-Kodierung definierte Koordinatensystem wird im Beispiel durch eine horizontale und eine vertikale Gerade, die sich in einem der XML-Knoten schneiden (im Beispiel in Element `p`), in vier Quadranten zerlegt. Die entstehenden Quadranten sind äquivalent zu den vier Knotenmengen, die sich aus den vier Achsen *preceding*, *following*, *ancestor* und *descendant* bzgl. `p` ergeben (vgl. Abbildung 2.3 auf Seite 12). Die *self*-Achse entspricht natürlich gerade dem (per Definition einzigen) Knoten im Schnittpunkt.

In dieser Kodierung des Dokuments kann die oben gesuchte Formel leicht angegeben werden: Nachfahren eines Knotens mit Koordinate (a, b) sind gerade diejenigen Knoten (x, y), für die $x > a$ und $y < b$ gilt (siehe dazu auch [40]).

Es existieren weitere Kodierungen wie etwa die *pre/size/level*-Kodierung, die in MonetDB/XQuery [26] verwendet wird, oder auch *Inverted Index* [100]. Teilweise lassen sich diese Kodierungen für jeden Knoten mit konstantem Aufwand ineinander überführen.

Im folgenden Abschnitt werden die Datenstrukturen vorgestellt, die für die Berechnung von XML-Prozessen durch die GPU verwendet werden. Sie ähneln der pre/post-Kodierung. Danach wird der Zusammenhang zwischen der verwendeten Datenstruktur und dem in Abschnitt 5 vorgestellten Sortierverfahren diskutiert.

7.1 Repräsentation der XML-Dokumente

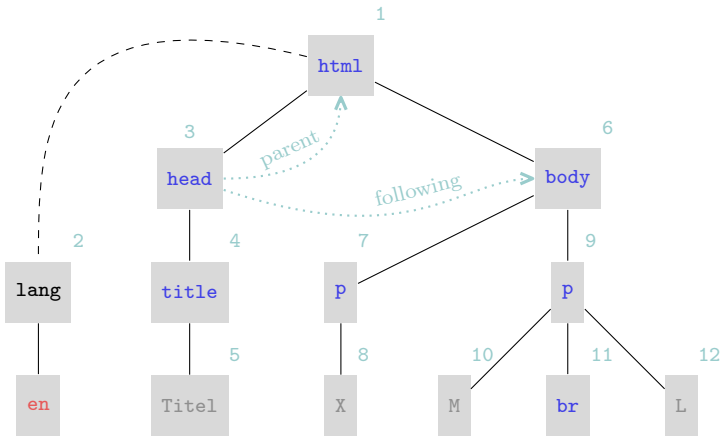


Abbildung 7.2: Dokument in Quellcode 2.1 kodiert für den GPU-Prozess

Wie bereits angedeutet, müssen die XML-Daten in geeigneter Weise im Speicher abgelegt werden um eine schnelle parallele Verarbeitung von XPath bzw. XSLT durchführen zu können, das heißt, so dass möglichst viele Zugriffe auf (verschiedene) Daten möglichst effizient durchgeführt werden können. Dabei wird die verwendete Datenstruktur natürlich auch beeinflusst durch das XML-Datenmodell, welches mit dieser Struktur abgebildet werden soll. In einem XML-Datenmodell beispielsweise, in welchem die Dokumentreihenfolge überhaupt keine Rolle spielt

id	parent	following	type	hash	flags	
1	-	-	element	`ff08...`	0	0
2	1	3	attribute	`156f...`	0	0
3	1	6	element	`2c2d...`	0	0
4	3	6	element	`bbf1...`	0	0
5	4	6	text	`bdcc...`	0	0
6	1	-	element	`e4db...`	0	0
7	6	9	element	`2a5e...`	0	0
8	7	9	text	`bdcc...`	0	0
9	6	-	element	`2a5e...`	0	0
10	9	11	text	`bdcc...`	0	0
11	9	12	element	`2b7e...`	0	0
12	9	-	text	`bdcc...`	0	0

Tabelle 7.1: Schematische Darstellung der XML-Datenstruktur für die GPU

(das heißt es existieren keine *following*-Achsen oder Ähnliches), ist die oben vorgestellte pre/post-Kodierung möglicherweise nicht die günstigste Kodierung des Dokuments.

Mit der hier vorgestellten Datenstruktur soll speziell das XPath-Datenmodell effizient umgesetzt werden können. Es wird keine zeigerbasierte Baumstruktur angelegt, sonder stattdessen werden in unserem Ansatz (wie bei der pre/post-Kodierung) XML-Knoten in Datenarrays abgelegt. Dabei wird für jeden Knoten seine Position im Dokument bezüglich der Dokumentreihenfolge als eindeutiger Identifizierer (Knoten-ID) genutzt. Zusätzlich zu der Knoten-ID eines Knotens werden auch die Knoten-ID des parent-Knotens und die Knoten-ID des following-Knotens gespeichert. Für jeden Knoten ist hierbei sein „following"-Knoten definiert als der Knoten mit der kleinsten Knoten-ID, der über die *following*-Achse erreicht wird. Abbildung 7.2 zeigt für das Beispieldokument aus Quellcode 2.1, welche Knoten-ID jedem Knoten zugewiesen wird. Zusätzlich ist beispielhaft für den `head`-Knoten durch Pfeile markiert, welche Knoten als parent-Knoten und following-Knoten gespeichert sind. Die hier verwendete Kodierung ID/*parent*-ID/*following*-ID entspricht also gerade einer um die Knoten-ID des Elternknoten erweiterten pre/post-Kodierung, da in der hier verwendeten Struktur die *following*-ID gerade *post*+1

entspricht.

Zudem wird für jeden Knoten sein Knotentyp sowie der Hashwert seines Namens gespeichert. Mittels zweier *flag*-Datenfelder kann für einen Knoten markiert werden, ob er in der aktuellen Kontextmenge einer XPath-Auswertung ist. Die Tabelle 7.1 zeigt die vollständige Repräsentation des XML-Baums aus Abbildung 7.2. In unserer Implementierung ist die Tabelle spaltenweise in Arrays abgelegt. In den in dieser Arbeit betrachteten SIMD-Architekturen wird häufig für viele Knoten parallel auf die gleiche Eigenschaft zugegriffen, etwa für alle Knoten auf ihren Typ. In einer spaltenweisen Repräsentation der Tabelle im Speicher können diese parallelen Zugriff *coalesced* durchgeführt werden, da zusammenhängende Speicherbereiche übertragen werden.

type	id
attribute	2
element	1
element	3
element	4
element	6
element	7
element	9
element	11
text	5
text	8
text	10
text	12

hash	id
`156f...`	2
`2c2d...`	3
`2a5e...`	7
`2a5e...`	9
`2b7e...`	11
`bbf1...`	4
`bdcc...`	5
`bdcc...`	8
`bdcc...`	10
`bdcc...`	12
`e4db...`	6
`ff08...`	1

Tabelle 7.2: Schematische Darstellung der Indizes über Typen und Namen (MD5) von XML-Knoten

In der hier entwickelten Implementierung befindet sich nur der Hashwert des Knotennamens, nicht jedoch der Knotenname selbst in der Datenstruktur. Das hat den Vorteil, dass die Repräsentation des Namens eines jeden Knotens die gleiche Größe im Speicher hat, was die Speicherzugriffe vereinfacht und die Verarbeitung beschleunigt. Ein Nachteil ist, dass bestimmte Funktionen von XPath, etwa String-Funktionen, auf dieser Struktur nicht verarbeitet werden können. Ebenso sind die

Inhalte von Textknoten sowie Attribut-Werte nicht in der Datenstruktur vorgesehen; unsere Datenstruktur bildet nur die Struktur des Dokuments ab, nicht jedoch den Inhalt. Selbstverständlich könnte man diese Daten zusätzlich zur eigentlichen Datenstruktur auf der Grafikkarte vorhalten und durch die eindeutige Knoten-ID referenzieren.

Die flag-Datenfelder werden benutzt, um anzuzeigen, ob ein Knoten in der aktuellen Kontextmenge eines Location Steps ist. Würde nur ein flag-Datenfeld genutzt werden, so müssten die Zugriffe der verschiedenen Threads synchronisiert werden, das heißt es muss sichergestellt werden, dass alle Threads alle notwendigen Werte gelesen haben, bevor der erste Thread einen Wert modifiziert. Das ist in der Variante mit zwei Datenfeldern nicht notwendig, wenn für jeden Location Step eines der Felder als Eingabe und das andere als Ausgabe verwendet wird. Zusätzlich vereinfacht diese Variante die Implementierung.

Neben der Datenstruktur, welche das XML-Dokument repräsentiert (Tabell 7.1), werden zwei weitere Strukturen auf der GPU gespeichert: Indizes über die Hashwerte der Namen der Knoten (genauer: die Hashwerte werden über Typen und Namen gebildet) sowie über die Typen der Knoten (Tabelle 7.2). Ein Index ist dabei eine sortierte Sequenz von Schlüssel-Wert-Paaren, wobei der Schlüssel im ersten Fall der Hashwert über den Typen und den Namen und im zweiten Fall der Knotentyp ist. In beiden Fällen ist der Wert die Knoten-ID des Knotens. Diese Strukturen ermöglichen einen schnellen Zugriff auf Knoten eines bestimmten Namens oder eines bestimmten Typs durch binäre Suche in den Indizes. Der Nutzen dieser Eigenschaft für die vorliegende Arbeit wird im Abschnitt 8 erläutert.

7.2 Datenstruktur & Bitonic Sort

Für die Bildung der oben angesprochenden Indizes über den Typ und den Namen (bzw. den entsprechenden MD5-Hashwert) müssen Schlüssel-Wert-Paare sortiert werden. Insbesondere im Szenario dieser Arbeit, in dem das sortierte Ergebnis im Speicher der GPU benötigt wird, kann besonders gut ein GPU-basierter Algorithmus genutzt werden. In formalen Repräsentationen von XML-Dokumenten im XPath-Datenmodell werden neben den Elementen auch alle Attribute und alle Textinhalte als Knoten dargestellt. Insbesondere die Textinhalte können je nach Herangehensweise sehr viele Knoten erzeugen, da auch Zeilenumbrüche, Leerzeichen usw. als Textknoten abgebildet werden. Wird etwa auf das dem Baum in

Abbildung 7.3: Straßenkarte Kiel

Abbildung 7.2 zugrunde liegende Beispieldokument 2.1 der XPath-Ausdruck

```
count(/descendant::text())
```

angewendet, das heißt werden alle Textknoten gezählt, so ist das Ergebnis 12, denn es werden (anders als in der hier verwendeten vereinfachten Baumdarstellung) auch Textknoten berücksichtigt, die nur aus nicht-druckbaren Zeichen bestehen. Insgesamt hat das Dokument 20 Knoten und besteht aus nur 198 *ASCII*-Zeichen (also etwa 10 Byte pro Knoten). Hochgerechnet ergibt sich für ein Dokument der Größe 10 MByte eine Knotenanzahl von einer Million. Im Kontext von XML-Dokumenten zur Datenhaltung (OSM, SVG) ist 10 MByte keine große Größe. Die Abbildung 7.3 stellt beispielsweise einen Ausschnitt der Straßenkartendaten von *Openstreetmap* [12] dar. Dargestellt ist ein etwa $3 \times 6\,\mathrm{km}^2$ umfassender Ausschnitt aus Kiel. Dieser Ausschnitt wird im *Openstreetmap*-Format durch ein fast 10 Mbyte großes XML-Dokument mit über 800,000 Knoten dargestellt.

Selbst in XML-Dokumenten, die keine Datenbanken, sondern menschenlesbaren Text bereitstellen bzw. annotieren, kommen durchaus nennenswerte Größen vor: Nicht nur SVG-Dateien können groß sein, auch der W3C-Standard zu SVG [94] hat in der xHTML-Darstellung beinahe 200.000 Knoten.

Wegen der großen Zahl der zu sortierenden Knoten ist eine sorgfältige Wahl des Sortieralgorithmus wichtig (vgl. Abschnitt 5.5). Im Fall der Sortierung nach dem Typ der Knoten wäre ggf. ein nicht-vergleichsbasierter Algorithmus günstiger, da die Menge der verschiedenen Schlüssel sehr klein ist. Für lange Schlüssel wie die hier verwendeten 128 Bit MD5-Hashwerte allerdings kann Bitonic Sort sehr gut eingesetzt werden.

KAPITEL 8

XPath

XPath ist eine der Kernkomponenten der XSLT-Verarbeitung. Da innerhalb einer einzelnen XSLT-Verarbeitung ggf. eine große Anzahl von XPath Evaluierungen durchgeführt wird, ist die Gesamtleistungsfähigkeit eines XSLT-Prozessors stark von der Leistungsfähigkeit der verwendeten XPath-Verarbeitung abhängig. In diesem Kapitel wird unser Ansatz zur XPath-Verarbeitung mittels SIMD-ähnlicher Hardware vorgestellt.

Der hier vorgestellte Ansatz wurde zusammen mit Jonas Bötel im Rahmen seiner Diplomarbeit [20] entwickelt. Die in den Tests genutzte Einbettung von XPath in Xalan wird ebenfalls in [20] detailliert beschrieben.

8.1 Einleitung

Wie bereits im Abschnitt 4 beschrieben wurde, konzentriert sich diese Arbeit auf die Verarbeitung von XPath unter Auslassung von Prädikaten. Bisher wurde auch die *namespace*-Achse nicht implementiert. In diesem Abschnitt wird also ein Ansatz zur Verarbeitung von Location Paths vorgestellt, welche nur aus XPath-Achsen (ohne *namespace*-Achse) und XPath-Knotentests bestehen. In vielen Fällen können Prädikate jedoch auch „nachträglich“ mit dem hier vorgestellten Ansatz verbunden werden, etwa wenn die Prädikate einfache Knotenfilter sind.

Eine sequentielle Verarbeitung eines XPath-Location Steps `axis::test`, der auf eine Kontextmenge V angewendet wird, sieht beispielsweise so aus wie in Quellcode 8.1: Für jeden Knoten in einer Kontextmenge wird die Knotenmenge ermittelt, die

```
1 for (v in V)
2 {
3    for (w in axis(v))
4    {
5       if ( test(w) )
6       {
7          select w
8       }
9    }
10 }
```

Quellcode 8.1: Beispiel 1: Pseudocode für Verarbeitung von Location Steps

über die Achse zu erreichen ist. Für jeden Knoten dieser Menge wiederum wird der Knotentest durchgeführt.

Dieser Algorithmus ist allerdings oft nicht geeignet, um für eine SIMD-Architektur angepasst zu werden:

1. Zwar kann die äußere Schleife parallelisiert werden, das heißt es könnte beispielsweise für jedes $v \in V$ ein Thread gestartet werden, weil die Kontextmenge vor der Auswertung des Location Steps bekannt ist. Die Anzahl der Wiederholungen in der inneren Schleife wird jedoch erst zur Laufzeit bekannt, so dass die Ausführungspfade für verschiedene Knoten $v, v' \in V$ sehr unterschiedlich werden können.

2. Zudem gilt, dass von einem Kontextknoten aus über eine Achse häufig mehr als ein Knoten im Dokument erreicht wird. Dies gilt insbesondere in Verbindung mit XSLT, wo an verschiedenen Stellen *forward*-Achsen wie *child* oder *descendant* der Regelfall oder sogar erforderlich sind. In diesen Fällen wäre also die Parallelisierung nicht optimal, denn jeder Thread muss in der inneren Schleife allein über eine bestimmte Knotenmenge iterieren, das heißt die Verarbeitung dieser Menge wird nicht parallelisiert.

3. In Abhängigkeit von der verwendeten Achse können für verschiedene $v, v' \in V$ die beiden Mengen $axis(v)$ und $axis(v')$ einen großen Schnitt haben. In diesen Fällen werden viele Knoten mehrfach behandelt.

Es sei an dieser Stelle erneut darauf hingewiesen, dass das in Punkt 3 kritisierte Vorgehen unter Umständen nicht vermieden werden kann, wenn nämlich kontextspezifische Prädikate wie im Beispiel in Abschnitt 2.3 verwendet werden. Für alle

```
1 for (w in test(N))
2 {
3    if ( w in axis(V) )
4    {
5          select w
6    }
7 }
```

Quellcode 8.2: Beispiel 2: Pseudocode für Verarbeitung von Location Steps

anderen Prädikate, insbesondere also auch im hier betrachteten Fall ohne Prädikate, ist dieses Vorgehen nicht notwendig.

8.2 Idee

Die drei oben genannten Schwierigkeiten können überwunden werden, wenn es gelingt, die Verschachtelung der Schleifen im Quellcode 8.1 unter Beibehaltung der Korrektheit zu vermeiden. Dies wird durch den Quellcode 8.2 angedeutet.

In dieser Variante sei nun N die Menge aller Knoten im Dokument. Es wird zuerst die Menge $test\,(N)$ aller Knoten im Dokument bestimmt, die den Knotentest erfüllen. Im Gegensatz zum zuvor formulierten Ansatz wird nun für alle Knoten, die den Knotentest erfüllen ($w \in test\,(N)$), geprüft, ob sie auch über die entsprechende Achse von einem Knoten der Kontextmenge erreicht werden können, also ob $w \in axis\,(V)$ gilt.

In diesem Ansatz gibt es nur eine Schleife, die nun parallelisiert werden kann indem etwa ein Thread für jedes $w \in test\,(N)$ genutzt wird. Die Ausführungspfade der beteiligten Threads sind in diesem Fall gleich. Zudem hat jeder Thread nur einen Knoten zu überprüfen, der Parallelitätsgrad ist also sehr hoch. Im ersten Beispielcode war eine Schwierigkeit, dass viele Knoten mehrfach getestet wurden. Dies kann in diesem Ansatz nicht geschehen.

Allerdings könnte es hier sein, dass etwa $axis\,(V)$ eine sehr kleine Menge ist, während $test\,(N)$ eine sehr große Menge ist. Dann werden im Beispielcode 8.2 viel mehr Knoten getestet als im Beispielcode 8.1. Dennoch wird in der vorliegenden Arbeit die hier präsentierte Idee weiter verfolgt, weil nämlich grundsätzlich das „zu viel“ Verarbeiten bei optimaler Parallelität und Auslastung der Ressourcen günstiger erscheint als das „zu viel“ Verarbeiten im ersten Beispiel (Problem 3).

Der hier vorgestellte Ansatz folgt für einige Achsen dem im Quellcode 8.2 prä-

sentierten Beispiel. In den folgenden Abschnitten werden Lösungen aufgezeigt, wie $test\,(N)$, die Menge der Knoten, die einen bestimmten Knotentest erfüllen, und $axis\,(V)$, die Menge der Knoten, die über eine bestimmte Achse von einer anderen Menge von Knoten aus erreichbar sind, geeignet für SIMD umgesetzt werden können.

Um die Korrektheit der hier entwickelten Auswertung von XPath-Ausdrücken für massiv-parallele Umgebungen zu zeigen, soll im Folgenden eine passende formale Semantik entwickelt werden, die auf Mengen von Knoten basiert und nicht wie die sonst üblichen XPath Semantiken knotenweise formuliert ist. Außerdem muss die Äquivalenz dieser mengenweisen Semantik zu etablierten Semantiken gezeigt werden. Für XPath `1.0` existiert im W3C-Standard jedoch keine formale Semantik. Basierend auf den Ansätzen in [38, 73] wird hier zunächst eine formale knotenweise Semantik (ohne Prädikate) angegeben, die der informell im Standard beschriebenen Semantik (vgl. C) entspricht.

8.3 Knotenweise Semantik $\mathcal{L}$

Zunächst wird die Bedeutung einer Achse (bzw. des formalen Achsenoperators) definiert durch die Menge der Knoten, die mit einem bestimmten anderen Knoten in der durch die Achse definierten Beziehung stehen:

$$axis : N \to N, axis\,(v) ::= \{w \mid v \xrightarrow{axis} w\} \tag{8.1}$$

Ein Knotentest wird definiert als boolscher Operator, der einen Knoten bezüglich einer bestimmten Achse auf einen boolschen Wert abbildet:

$$test : axis \times N \to \mathbb{B}, test_{axis}\,(v) ::= \begin{cases} \top & \text{falls } v \text{ gültig ist gemäß } test \\ & \text{unter Berücksichtigung des} \\ & \text{Hauptknotentyps von } axis \\ \bot & \text{sonst} \end{cases} \tag{8.2}$$

Nun wird die Auswertung von Location Paths gegeben. Die Notation orientiert sich an [38, 73]. Dabei ist die Semantik durch die Funktion $\mathcal{L}[\![path]\!](v)$ gegeben, welche einen Ausdruck $path$ und einen Knoten v auf eine Knotenmenge abbildet. Zunächst wird die Auswertung absoluter Location Paths (mit v_0 als $root$-Knoten) definiert:

$$\mathcal{L}[\![/rela]\!](v) ::= \mathcal{L}[\![rela]\!](v_0) \tag{8.3}$$

Die Auswertung relativer Location Paths erfolgt (entsprechend dem Standard [84, Abschnitt 2]) rekursiv, indem definiert wird, wie der letzte Location Step eines Location Path ausgewertet wird:

$$\mathcal{L}[\![rela/step]\!](v) ::= \bigcup_{w \in \mathcal{L}[\![rela]\!](v)} \mathcal{L}[\![step]\!](w) \tag{8.4}$$

Demnach gilt auch:

$$\mathcal{L}[\![step_1/\ldots/step_n]\!](v)$$

$$\overset{8.4}{=} \bigcup_{k \in K} \mathcal{L}[\![step_n]\!](k) \quad \text{mit } K := \mathcal{L}[\![step_1/\ldots/step_{n-1}]\!](v)$$

Auch auf K kann natürlich Gleichung 8.4 angewendet werden und es gilt demnach:

$$K = \bigcup_{z \in Z} \mathcal{L}[\![step_{n-1}]\!](z) \quad \text{mit } Z := \mathcal{L}[\![step_1/\ldots/step_{n-2}]\!](v)$$

K ist also ebenfalls nur eine Vereinigung. Da K als Index einer anderen Vereinigung benutzt wird, kann diese Vereinigung auch als Vereinigung von Vereinigungen (also mittels Z) ausgedrückt werden:

$$\bigcup_{k \in K} \mathcal{L}[\![step_n]\!](k) \;=\; \bigcup_{z \in Z} \Big(\bigcup_{w \in \mathcal{L}[\![step_{n-1}]\!](z)} \mathcal{L}[\![step_n]\!](w) \Big)$$

Die innere Vereinigung kann einfach wieder mit Gleichung 8.4 umgeformt werden zu:

$$\bigcup_{w \in \mathcal{L}[\![step_{n-1}]\!](z)} \mathcal{L}[\![step_n]\!](w) \overset{8.4}{=} \mathcal{L}[\![step_{n-1}/step_n]\!](z)$$

Demnach gilt:

$$\begin{aligned} \mathcal{L}[\![step_1/\ldots/step_n]\!](v) \;&=\; \bigcup_{z \in Z} \mathcal{L}[\![step_{n-1}/step_n]\!](z) \\ &= \bigcup_{w \in \mathcal{L}[\![step_1/\ldots/step_{n-2}]\!](v)} \mathcal{L}[\![step_{n-1}/step_n]\!](w) \end{aligned}$$

Diese Umformungen können fortgesetzt werden zu:

$$\mathcal{L}[\![step_1/\ldots/step_n]\!](v) \;=\; \bigcup_{w \in \mathcal{L}[\![step_1/\ldots/step_i]\!](v)} \mathcal{L}[\![step_{i+1}/\ldots/step_n]\!](w)$$

Danach gilt allgemein:

$$\mathcal{L}[\![rela_1/rela_2]\!](v) = \bigcup_{w \in \mathcal{L}[\![rela_1]\!](v)} \mathcal{L}[\![rela_2]\!](w) \tag{8.5}$$

Einfach gesagt, kann nach dieser Gleichung ein Location Path an beliebiger Stelle „aufgeteilt“ werden. Abschließend wird angegeben, wie Knotentests innerhalb eines Location Steps ausgewertet werden:

$$\mathcal{L}[\![axis::test]\!](v) ::= \{w \mid w \in axis(v) \wedge test_{axis}(w)\} \tag{8.6}$$

Die formale Auswertung von XPath-Prädikaten bzw. von beliebigen XPath-Expressions wird hier wie gesagt nicht betrachtet, da sich unsere Implementierung von XPath für GPUs vor allem mit der schnellen Auswertung von Achsen und Knotentests beschäftigt. Die angegebene Semantik enthält nur die Teile, die sich nicht auf Prädikate beziehen.

Diese Semantik wie auch die verschiedenen in der Literatur angegebenen Semantiken für XPath `1.0` beziehen sich in der Regel auf die Auswertung eines einzelnen Kontextknoten.

8.4 Mengenweise Semantik $\mathcal{M}$

Übliche formale Definitionen einer Semantik für XPath sehen vor (vgl. Gleichung 8.4), dass ein Location Step für eine Knotenmenge ausgewertet wird, indem alle Knoten einzeln ausgewertet und die Ergebnisse vereinigt werden. Das bedeutet, dass die Auswertung eines Location Steps (vgl. Gleichung 8.6) unabhängig von der Knotenmenge geschieht, also unabhängig davon, welche Knoten sonst noch in der Knotenmenge sind. Ein Vorgehen wie in Quellcode 8.2 ist mit dieser Semantik nicht möglich, weil beispielsweise die Auswertung einer Achse für eine Knotenmenge nicht vorgesehen ist. Als anschauliches Beispiel sei hier die Auswertung der *descendant*-Achse ohne Knotentest erwähnt: Angenommen, ein Knoten und seine Kindknoten sind in der Kontextmenge. Es ist intuitiv klar, dass die Auswertung der Achse dann nur für den Elternknoten erfolgen muss. Mit der formalen knotenweisen Semantik kann dies jedoch grundsätzlich nicht umgesetzt werden.

Um eine Semantik zu erhalten, die sich leicht auf einen parallelen (SIMD-) Ansatz zur Verarbeitung übertragen lässt, muss die knotenweise definierte Semantik deshalb auf eine mengenweise definierte Semantik überführt werden. Gewünscht

ist eine Semantik, die sich auf Mengen von Knoten bezieht, das heißt, bei der ein Location Step bezüglich einer Knotenmenge ausgewertet wird. Außerdem muss die mengenweise Semantik natürlich konsistent zur knotenweisen Semantik sein. Es soll also etwa gelten:

$$\mathcal{L}[\![rela_1/rela_2]\!](v) = \mathcal{M}[\![rela_2]\!](\mathcal{L}[\![rela_1]\!](v))$$

In dieser Darstellung ist $\mathcal{L}[\![rela_1]\!](v)$ für die mengenweise Semantik eine einfache Knotenmenge; die mengenweise Semantik soll nicht berücksichtigen, wie diese Menge erzeugt wurde.

Der Standard definiert bekanntlich nur die Auswertung von Location Steps bezüglich einzelner Kontextknoten. Enthält die Ausgabe eines Location Steps mehrere Knoten, so soll der nachfolgende Location Step (wie es auch im Beispiel in Abschnitt 2.3 gezeigt wurde) auf alle Knoten einzeln angewendet und die Ergebnisse vereinigt werden. Intuitiv ist die Auswertung von Knotenmengen also klar, und in Anlehnung an die entsprechende Gleichung 8.4 definieren wir die mengenweise Semantik durch:

$$\mathcal{M}[\![path]\!](V) ::= \bigcup_{v \in V} \mathcal{L}[\![path]\!](v) \tag{8.7}$$

Es folgt sofort, dass wie gewünscht gilt

$$\begin{aligned} \mathcal{L}[\![rela_1/rela_2]\!](v) &\overset{8.5}{=} \bigcup_{w \in \mathcal{L}[\![rela_1]\!](v)} \mathcal{L}[\![rela_2]\!](w) \\ &\overset{8.7}{=} \mathcal{M}[\![rela_2]\!](\mathcal{L}[\![rela_1]\!](v)) \end{aligned} \tag{8.8}$$

Weiter ist leicht zu sehen, dass die knotenweise und die mengenweise Semantik identische Ergebnisse liefern für einelementige Mengen:

$$\begin{aligned} \mathcal{M}[\![rela]\!](\{v\}) &\overset{8.7}{=} \bigcup_{w \in \{v\}} \mathcal{L}[\![rela]\!](w) \\ &= \mathcal{L}[\![rela]\!](v) \end{aligned} \tag{8.9}$$

Ebenso sind die Semantiken austauschbar für absolute Location Steps:

$$\begin{aligned} \mathcal{M}[\![/rela]\!](V) &\overset{8.7}{=} \bigcup_{w \in V} \mathcal{L}[\![/rela]\!](w) \\ &\overset{8.3}{=} \bigcup_{w \in V} \mathcal{L}[\![rela]\!](v_0) \\ &= \mathcal{L}[\![rela]\!](v_0) \end{aligned} \tag{8.10}$$

Es gilt außerdem:

$$\begin{aligned}
\mathcal{M}[\![rela_1/rela_2]\!](V) &\overset{8.7}{=} \bigcup_{w \in V} \mathcal{L}[\![rela_1/rela_2]\!](w) \\
&\overset{8.5}{=} \bigcup_{w \in V} \bigcup_{z \in \mathcal{L}[\![rela_1]\!](w)} \mathcal{L}[\![rela_2]\!](z) \\
&= \bigcup_{z \in \bigcup_{w \in V} \mathcal{L}[\![rela_1]\!](w)} \mathcal{L}[\![rela_2]\!](z) \\
&\overset{8.7}{=} \bigcup_{z \in \mathcal{M}[\![rela_1]\!](V)} \mathcal{L}[\![rela_2]\!](z) \\
&\overset{8.7}{=} \mathcal{M}[\![rela_2]\!](\mathcal{M}[\![rela_1]\!](V)) \qquad (8.11)
\end{aligned}$$

Diese letzte Gleichung 8.11 erlaubt es, die Auswertung von Location Paths mittels der mengenweisen Semantik beliebig zu zerlegen, ähnlich wie es in Gleichung 8.5 für die knotenweise Semantik gezeigt wurde. Dabei können die Semantiken für die Auswertung beliebiger Teil-Location Paths ausgetauscht werden, solange sich die Auswertung des gesamten Location Path auf einen einzelnen Kontextknoten bezieht (was in allen XPath-Anwendungen der Fall ist). Für diese Fälle sind die Semantiken demnach äquivalent.

Bis hierhin wurde eine formale mengenweise Semantik entwickelt, die den intuitiven Ansatz bestätigt. Im Unterschied zur knotenweisen Semantik kann nun jedoch ein einzelner Location Step bezüglich einer ganzen Knotenmenge ausgewertet werden:

$$\begin{aligned}
\mathcal{M}[\![axis::test]\!](V) &\overset{8.7}{=} \bigcup_{v \in V} \mathcal{L}[\![axis::test]\!](v) \\
&\overset{8.6}{=} \bigcup_{v \in V} \left\{ w \;\middle|\; w \in axis(v) \wedge test_{axis}(w) \right\} \\
&= \left\{ w \;\middle|\; w \in \bigcup_{v \in V} axis(v) \wedge test_{axis}(w) \right\}
\end{aligned}$$

In dieser Darstellung werden alle Knoten gesucht, die von einem Knoten in V ausgehend über *axis* erreicht werden können und zudem den Knotentest $test_{axis}(\cdot)$ erfüllen. Es könnten umgekehrt auch diejenigen Knoten gesucht werden, die in der Menge der Knoten des Dokuments sind, welche den Knotentest erfüllen und die darüber hinaus noch über die Achse *axis* von einem Knoten in V erreicht werden

können:

$$\begin{aligned}&\mathcal{M}[\![axis::test]\!](V)\\&= \left\{ w \;\middle|\; w \in \bigcup_{v \in V} axis\,(v) \wedge w \in \{t \in N \mid test_{axis}\,(t)\} \right\}\end{aligned} \tag{8.12}$$

Entsprechend dieser Umformungen definieren wir nun für die mengenweise Semantik $axis\,(V)$ und $test_{axis}\,(V)$ wie folgt:

$$\begin{aligned}&axis : \mathcal{P}(N) \to N, && axis\,(V) ::= \bigcup_{v \in V} axis\,(v)\\&test : axis \times \mathcal{P}(N) \to N, && test_{axis}\,(V) ::= \{v \in V \mid test_{axis}\,(v)\}\end{aligned}$$

Damit kann die Gleichung 8.12 kompakt dargestellt werden als:

$$\begin{aligned}\mathcal{M}[\![axis::test]\!](V) &= \left\{ w \in test_{axis}\,(N) \;\middle|\; w \in axis\,(V) \right\}\\&= test_{axis}\,(N) \cap axis\,(V)\end{aligned} \tag{8.13}$$

Mit dieser Gleichung liegt uns eine mengenweise Semantik vor, die auf den mengenweisen Operatoren aus dem Beispiel-Pseudocode 8.2 basiert. Darüber hinaus ist die Äquivalenz der Semantiken entsprechend der vorgestellten Gleichungen gegeben. Werden also die XPath-Algorithmen korrekt nach der hier eingeführten mengenweisen Semantik implementiert, so sind diese Algorithmen ebenso korrekt nach der üblichen knotenweisen Semantik.

In manchen Fällen ist die in 8.13 dargestellte Gleichung allerdings nicht günstig; insbesondere die Berechnung von $axis\,(V)$ ist unter Umständen aufwendig. Im folgenden Abschnitt wird aufgezeigt, wie dieses Problem gelöst werden kann.

8.5 Inverse Achsen

In einem Location Step werden im Allgemeinen Knoten gesucht, die von einem Kontextknoten ausgehend über eine bestimmte Achse erreicht werden können. Es wurde bereits festgestellt, dass dafür ein Ansatz wie in Gleichung 8.13 günstig sein kann. Allerdings ist die Berechung von $axis\,(V)$ aufwendig und unter Umständen auch überflüssig, denn eigentlich ist die Menge $axis\,(V)$ in Gleichung 8.13 unwichtig; wichtig ist nur, ob ein bestimmter Knoten in dieser Menge liegt.

Eine Alternative dazu ist es, die zu der im Location Step verwendeten Achse *inverse* Achse zu nutzen. Mittels einer solchen inversen Achse könnten zu einem

Knoten a im Dokument gerade diejenigen Knoten bestimmt werden, von denen ausgehend a über die entsprechende Achse überhaupt erreicht werden kann. Für diese Kandidatenknoten muss dann geprüft werden, ob sie in V sind.

Diese Alternative ist gerade dann sehr günstig, wenn die Menge der Knoten, die über die inverse Achse erreicht wird, klein ist. So kann etwa die *child*-Achse beliebig viele Knoten enthalten. Die inverse Achse, die *parent*-Achse, dagegen kann nur höchstens einen Knoten enthalten.

Daher soll nun eine formale Methode entwickelt werden, einen Location Step $axis::test$ nicht unter Verwendung der im Location Step verwendeten Achse zu implementieren, sondern unter Verwendung der inversen Achse. Außerdem werden zwei wichtige inverse Achsen definiert.

Für die inverse Achse gilt:

$$v, w \in N, w \xrightarrow{axis^{-1}} v \iff v \xrightarrow{axis} w \tag{8.14}$$

Formal definiert gehört nun ein Knoten $w \in test_{axis}(N)$ zur Ergebnismenge des Location Steps, wenn der Schnitt der Knotenmenge, die über die entsprechende inverse Achse von w aus erreicht wird, und der Kontextmenge V nicht leer ist, da für alle Knoten $w \in N$ gilt:

$$w \in N: \quad \exists v \in V \cap axis^{-1}(w) \iff w \xrightarrow{axis^{-1}} v \iff v \xrightarrow{axis} w$$

Mit dieser Erkenntnis kann eine weitere äquivalente Darstellung für die mengenweisen Semantik angeben werden:

$$\begin{aligned} \mathcal{M}[\![axis::test]\!](V) &\overset{8.13}{=} \left\{ w \in test_{axis}(N) \;\middle|\; w \in axis(V) \right\} \\ &= \left\{ w \in test_{axis}(N) \;\middle|\; axis^{-1}(w) \cap V \neq \emptyset \right\} \end{aligned} \tag{8.15}$$

Einfach gesagt kann in dieser Darstellung darauf verzichtet werden, $axis(V)$ zu berechnen. Stattdessen wird für alle Knoten im Dokument, welche den Knotentest erfüllen, geprüft, ob die inverse Achse zu Knoten in der bereits bekannten Kontextmenge V führt. Dies kann leicht überprüft werden, da in der hier verwendeten Datenstruktur für jeden Knonten markiert ist, ob er zur Kontextmenge gehört.

`child`$^{-1}$ & `attribute`$^{-1}$

Im W3C-Standard zu XPath [84] befindet sich folgende Aussage: „Each element node has an associated set of attribute nodes; the element is the parent of each

of these attribute nodes; however, an attribute node is not a child of its parent element.“.

Offensichtlich folgt daraus, dass nicht zu jeder Achse aus dem Standard bereits die inverse Achse im Standard existiert. Jedoch können diese inversen Achsen vergleichsweise einfach angegeben werden, wenn zusätzlich zu den bisherigen Funktionen eine boolsche Funktion angegeben wird, die den Typ eines Knotens prüfen kann. Sei dazu die Menge der Knotentypen definiert (vgl. Tabelle 2.1) als:

$$NTypes := \{\text{root, element, text, attribute, namespace,} \\ \text{processing instruction, comment}\}$$

Die Prüffunktion τ wird nun definiert als:

$$\tau : NTypes \times N \to \mathbb{B}, \tau_{type}(v) ::= \begin{cases} \top, \text{falls } v \text{ vom Knotentyp } type \text{ ist} \\ \bot, \text{sonst} \end{cases}$$

Für die im folgenden Abschnitt vorgestellten Implementierungen der XPath-Achsen benötigen wir die inverse *attribute*-Achse sowie die inverse *child*-Achse. Damit die in 8.14 formulierte Eigenschaft für inverse Achsen gilt, muss dazu mittels der Funktion τ der Definitionsbereich eingeschränkt werden. Die Definition der inversen *attribute*-Achse ist:

$$\texttt{attribute}^{-1}(v) = \begin{cases} \texttt{parent}(v) & \text{falls } \tau_{\texttt{attribute}}(v) \\ \emptyset & \text{sonst} \end{cases} \tag{8.16}$$

Ganz analog ist die inverse *child*-Achse definiert:

$$\texttt{child}^{-1}(v) = \begin{cases} \texttt{parent}(v) & \text{falls } \neg\tau_{\texttt{attribute}}(v) \\ \emptyset & \text{sonst} \end{cases} \tag{8.17}$$

8.6 Implementierung

Wir verfügen nun über eine mengenweise Semantik zur Auswertung von Location Paths ohne Prädikate. Diese Semantik ist sowohl für die Nutzung von Achsen als auch für die Nutzung bestimmter inverser Achsen definiert. In beiden Fällen (Gleichungen 8.13 und 8.15) wird mit $test_{axis}(N)$ die Menge der Knoten des Dokuments bestimmt, für die der Knotentest gilt.

Wie schon in der Einführung zu XPath in Abschnitt 2.3 angesprochen wurde, kann mit dem Knotentest festgelegt werden, welche Knotennamen bzw. welche Knotentypen in der Ergebnismenge erlaubt sind.

Die Knotenmengen, für die ein Knotentest gilt, lassen sich mit Hilfe der in Abschnitt 7 beschriebenen Indizes sehr schnell ermitteln (mittels binärer Suche in logarithmischer Zeit). Durch diese Indizes kann auf die gewählten Knoten auch sehr leicht zugegriffen werden. Sollen etwa alle Knoten mit dem Namen *name* gewählt werden, so muss im Index der Knotennamen nur bestimmt werden, an welcher Position s der erste Eintrag *name* vermerkt ist. Außerdem muss bestimmt werden, wie viele Einträge *name* existieren, das heißt an welcher Position $e > s$ der erste Eintrag $\neq$ *name* vermerkt ist. Die gesuchten Knoten sind dann gerade an den Positionen $s, \ldots, e-1$. Im Folgenden wird das Array der gewünschten Knoten einfach dargestellt als Array `id`.

Die Implementierung eines Location Paths wird als Hintereinanderausführung seiner Location Steps implementiert, wobei jeder Location Step eine Kontextmenge als Eingabe bekommt und eine Kontextmenge als Ausgabe liefert (vgl. Abschnitt 2.3). Ein Location Path wird „getaktet“ verarbeitet, das heißt, ein Thread verarbeitet erst dann den n-ten Location Step, nachdem alle Threads den $(n-1)$-ten Location Step verarbeitet haben.

Durch das getaktete Vorgehen können die Location Steps einfach durchgezählt werden; die Eingabekontextmenge eines Location Steps i ist einfach definiert als diejenigen Knoten, für die in einem `flag`-Datenfeld (vgl. Tabelle 7.1) die Zahl i eingetragen ist. Diejenigen Knoten, die im Location Step i für die Ausgabekontextmenge ausgewählt werden, werden einfach im zweiten `flag`-Datenfeld mit dem Wert $i+1$ gekennzeichnet. Die Eingabekontextmenge des Location Steps $i+1$ sind also dementsprechend alle Knoten, für die im zweiten `flag`-Datenfeld die Zahl $i+1$ vermerkt ist.

Wie in Abschnitt 7 beschrieben wurde, sind die Datenfelder der XML-Datenstruktur spaltenweise im Speicher abgelegt, das heißt beide `flag`-Datenfelder sind in je einem Array gespeichert. Nach jedem Location Step können deshalb leicht die Zeiger auf diese Arrays getauscht werden und als Konsequenz wird (aus Sicht eines Threads) die Eingabekontextmenge immer durch das erste `flag`-Datenfeld (`flag1`), die Ausgabekontextmenge immer durch das zweite `flag`-Datenfeld (`flag2`) repräsentiert.

In diesem Abschnitt wird nun die Implementierung der Location Steps angege-

ben. Da die Implementierung im Wesentlichen durch die im Location Step verwendete Achse bestimmt wird, werden im Folgenden die Implementierungen der Location Steps entsprechend der verschiedenen verwendeten Achsen erläutert.

8.6.1 *self*-Achse

```
1 self_id  = id[thread_id];
2 self_flag = flag1[self_id];
3 if (self_flag == current_flag )
4 {
5    flag2[self_id] = current_flag + 1;
6 }
```

Quellcode 8.3: *self*-Achse

Über die *self*-Achse wird, ausgehend von einem Kontextknoten, nur dieser Kontextknoten erreicht. Durch die *self*-Achse selbst wird also die Kontextmenge nicht verändert, ggf. jedoch durch die Knotentests.

Die *self*-Achse wird implementiert, indem für jeden Knoten, der den Knotentest erfüllt, ein Thread den obigen Pseudocode ausführt. Jeder Thread liest also „seine“ Knoten-ID `self_id` aus dem Array der Knoten-IDs `id` und den Eintrag in `flag1[self_id]`. Durch `self_flag ==current_flag` wird geprüft, ob der entsprechende Knoten zur aktuellen Kontextmenge gehört. Ist dem so, wird durch

```
flag2[self_id]=current_flag + 1
```

der Knoten der nächsten Kontextmenge hinzugefügt.

8.6.2 *attribute*-Achse

Für die Attributachse wird die Semantik auf Basis der inversen Achsen (Gleichung 8.15) verwendet. Dies erscheint günstig, da die zur Attributachse inverse Achse höchstens den Elternknoten eines Knotens enthalten kann (vgl. Gleichung 8.16); ein Knoten ist demnach genau dann in der Ausgabekontextmenge, wenn er den Knotentest erfüllt und die inverse Attributachse einen Knoten enthält, der in der aktuellen Kontextmenge liegt.

Wiederum mittels der Indizes werden die Knoten gewählt, die den Knotentest erfüllen. Für jeden dieser Knoten führt ein Thread den obigen Pseudocode aus.

```
1 self_id     = id[thread_id];
2 parent_id   = parent[self_id];
3 parent_flag = flag1[parent_id];
4 if (parent_flag == current_flag)
5 {
6   flag2[self_id] = current_flag + 1;
7 }
```

Quellcode 8.4: *attribute*-Achse

Wie in Gleichung 8.16 zu sehen ist, müsste die Verarbeitung des Programms abgebrochen werden, wenn der zu verarbeitende Knoten kein Attribut ist. Hier ist jedoch eine explizite Prüfung nicht nötig, da mittels der Indizes direkt nur die Knoten ausgewählt werden, die den Knotentest erfüllen und auch ein Attribut sind (da die Indizes über die Hashwerte über den Namen und den Typ gebildet wird).

Für die Attributknoten ist die inverse Attributachse gerade die *parent*-Achse. Deshalb wird im Pseudocode nur geprüft, ob der Elternknoten in der Eingabekontextmenge ist. Ist dies der Fall, fügt der Thread „seinen“ Attributknoten der Ausgabekontextmenge hinzu.

8.6.3 *child*-Achse

```
1 self_id = id[thread_id];
2 self_type = type[self_id];
3 if (self_type == attribute)
4 {
5   return;
6 }
7 parent_id = parent[self_id];
8 parent_flag = flag1[parent_id];
9 if (parent_flag == current_flag)
10 {
11   flag2[self_id] = current_flag + 1;
12 }
```

Quellcode 8.5: *child*-Achse

Wie für die *attribute*-Achse wird für die *child*-Achse die Semantik auf Basis der inversen Achsen genutzt, da die inverse Achse zur *child*-Achse höchstens einen (Eltern-)Knoten enthalten kann; ein Knoten ist demnach genau dann in der Ausgabekontextmenge, wenn er den Knotentest erfüllt und die inverse *child*-Achse

einen Knoten enthält, der in der Kontextmenge ist.

Wieder werden mittels der Indizes die Knoten gewählt, die den Knotentest erfüllen. Hier kann jedoch nicht direkt mittels der Indizes garantiert werden, dass nur Knoten ausgewählt werden, für die die inverse *child*-Achse ein Ergebnis liefert.

Deshalb prüft jeder Thread zunächst für seinen Knoten, ob der Knotentyp nicht vom Typ Attribut ist. Danach ist das Verfahren wie für die *attribute*-Achse.

8.6.4 *descendant[-or-self]*

```
1 self_id = id[thread_id];
2 self_type = type[self_id];
3 if (self_type == attribute)
4 {
5   return;
6 }
7
8 parent_id = parent[self_id];
9 while (parent_id != NULL)
10 {
11   parent_flag = flag1[parent_id];
12   if (parent_flag == current_flag)
13   {
14     flag2[self_id] = current_flag + 1;
15     return;
16   }
17   parent_id = parent_[parent_id];
18 }
```

Quellcode 8.6: *descendant*-Achse

Diese Achse ist die Fortsetzung der *child*-Achse, das heißt es gilt

$$\mathcal{M}[\![descendant :: test]\!](V) = \mathcal{M}[\![child :: test]\!](V) \cup \mathcal{M}[\![descendant :: test]\!](\mathcal{M}[\![child :: node()]\!](V))$$

Die Implementierung folgt dabei eng der Implementierung der *child*-Achse. Statt nur den Elternknoten zu prüfen, werden hier allerdings in einer Schleife alle Vorfahren geprüft, bis entweder ein Elternknoten gefunden wurde, der in der Eingabekontextmenge ist, oder die Wurzel gefunden wurde.

Man könnte an dieser Stelle auch noch prüfen, ob der Elternknoten bereits in der Ausgabekontextmenge ist, weil er bereits von einem anderen Thread eingefügt

wurde. Dazu müsste man allerdings aus `flag2` lesen und somit die klare Trennung zwischen lesendem Zugriff auf `flag1` und schreibendem Zugriff auf `flag2` aufheben.

Analog zur *descendant*-Achse ist die *descendant-or-self*-Achse definiert.

8.6.5 *parent*-Achse

```
1 self_id = thread_id;
2 self_flag = flag1[self_id];
3 if (self_flag ≠ current_flag)
4 {
5    return;
6 }
7 parent_id = parent[self_id];
8 parent_hash = hash[parent_id];
9 if (parent_hash == filter_hash)
10 {
11   flag2[parent_id] = current_flag + 1;
12 }
```

Quellcode 8.7: *parent*-Achse

Da die *parent*-Achse nur höchstens einen Knoten liefert, erscheint hier die Nutzung der üblichen knotenweisen Semantik günstig. Es wird für jeden existierenden Knoten ein Thread genutzt, der prüft, ob der Knoten ein Kontextknoten ist, und, wenn dem so ist, den Elternknoten des Knotens bestimmt. Für den Elternknoten wird dann noch ggf. der Knotentest durchgeführt (hier beispielhaft `parent_hash ==filter_hash`) und im Erfolgsfall der Elternknoten in die Ausgabekontextmenge eingefügt.

8.6.6 *ancestor[-or-self]*-Achse

Ganz analog zur *descendant*-Achse als Fortsetzung der *child*-Achse wird die *ancestor*-Achse als Fortsetzung der *parent*-Achse implementiert. Statt nur den Elternknoten zu prüfen, werden alle Vorfahren geprüft.

8.6.7 *following*-Achse

In den bisher dargelegten Implementierungen war die Bestimmung der Knotenmenge, die über eine Achse erreicht werden kann, entweder einfach, weil die Achse

```
1 self_id   = thread_id;
2 self_flag = flag1[self_id];
3 if (self_flag ≠ current_flag)
4 {
5    return;
6 }
7 following_id = following[self_id];
8 atomicMin(min, following_id);
```

Quellcode 8.8: *following*-Achse, globales Minimum

```
1 self_id   = id[thread_id];
2 self_type = type[self_id];
3 if (self_type == attribute)
4 {
5    return;
6 }
7 if (self_id ≥ min)
8 {
9    flag2[self_id] = current_flag + 1;
10 }
```

Quellcode 8.9: *following*-Achse

oder aber die inverse Achse nur einelementige Mengen oder einfach zu konstruierende Mengen liefert. Zudem war in den bisher präsentierten Implementierungen der Ansatz, bei dem jeder Knoten der Kontextmenge oder sogar jeder Knoten im Dokument unabhängig voneinander behandelt wurde, plausibel, denn die bisher untersuchten Achsen liefern, angewendet auf eine Kontextknotenmenge V, für je zwei verschiedene Knoten aus V ggf. disjunkte Ergebniskontextmengen.

Anders ist die Situation bei der Auswertung der *following*-Achse, denn es gilt im Gegenteil für jede Kontextknotenmenge V

$$\exists v \in V \mid \mathit{following}(V) = \mathit{following}(\{v\})$$

Das bedeutet, dass es in der Kontextknotenmenge einen Knoten gibt, für den die *following*-Achse eine Ausgabekontextmenge liefert, die der Ausgabekontextmenge der *following*-Achse angewendet auf die ganze Kontextmenge entspricht. Dieser Kontextknoten ist derjenige in der Eingabekontextmenge mit dem kleinsten Wert im *following*-Datenfeld (vgl. Tabelle 7.1).

Die *following*-Achse eines einzelnen Kontextknotens wiederum ist sehr einfach zu bestimmen: Alle Knoten im Dokument, deren Knoten-ID größer oder gleich dem

Eintrag im *following*-Datenfeld des Kontextknotens ist, sind über die *following*-Achse zu erreichen. Eine formale Beschreibung der hier verwendeten Überlegungen zur *following*-Achse sind in [20] zu finden.

Unser Ansatz zur Implementierung der *following*-Achse ist also zweistufig. In der ersten Stufe wird derjenige Knoten v ermittelt, für den $\mathit{following}(V) = \mathit{following}(\{v\})$ gilt. In der zweiten Stufe wird anhand dieses Knotens die Ausgabekontextmenge bestimmt. Der Pseudocode 8.8 zeigt die erste Stufe. Für jeden Knoten im Dokument wird je ein Thread genutzt, der zunächst prüft, ob der Knoten in der aktuellen Kontextmenge liegt. Über die Einträge im *following*-Datenfeld aller Knoten der aktuellen Kontextmenge wird ein globales Minimum gebildet. Der Pseudocode 8.9 zeigt die zweite Stufe. Für jeden Knoten im Dokument, der die Knotentests erfüllt, wird ein Thread genutzt. Es wird durch `self_id` $\geq$ `min` geprüft, ob der Knoten zur Ausgabekontextmenge gehört und der Wert im `flag2`-Array entsprechend gesetzt.

8.6.8 *preceding*-Achse

Die *preceding*-Achse ist symmetrisch zur *following*-Achse. Die Implementierung erfolgt entsprechend unter Verwendung des globalen Maximums anstelle des globalen Minimums.

8.6.9 *following-sibling*-Achse

```
1 self_id   = thread_id;
2 self_flag = flag1[self_id];
3 if (self_flag ≠ current_flag)
4 {
5    return;
6 }
7 following_id = following[self_id];
8 parent_id    = parent[self_id];
9 atomicMin(temp[parent_id], following_id);
```

Quellcode 8.10: *following-sibling*-Achse, lokale Minima

Die *following-sibling*-Achse enthält alle Knoten der *following*-Achse, die auch Geschwisterknoten des Kontextknotens sind; die *following-sibling*-Achse enthält also alle in Dokumentreihenfolge nachfolgenden Geschwisterknoten. Demnach gilt

```
1 self_id   = id[thread_id];
2 self_type = type[self_id];
3 if self_type == attribute)
4 {
5    return;
6 }
7 parent_id  = parent[self_id];
8 parent_min = temp[parent_id];
9 if (self_id ≥ parent_min)
10 {
11    flag2[self_id] = current_flag + 1;
12 }
```

Quellcode 8.11: *following-sibling*-Achse

ähnlich wie im Falle der *following*-Achse, dass es genügt, unter den Geschwister-Kontextknoten denjenigen zu finden, der die kleinste Knoten-ID hat. Demzufolge übertragen wir den Ansatz für die *following*-Achse auf die *following-sibling*-Achse, indem in einem ersten Schritt nicht mehr globale Minima, sondern lokale Minima (das heißt Minima unter Geschwisterknoten) gesucht werden. Im zweiten Schritt muss dann wieder für jeden Knoten entsprechend des gefundenen lokalen Minimums entschieden werden, ob er zur Ausgabekontextmenge gehört.

Der Pseudocode 8.10 zeigt die erste Stufe: da die lokalen Minima immer unter Geschwisterknoten gesucht werden, das heißt also unter Knoten mit dem selben Elternknoten, bietet es sich an, die lokalen Minima an einer durch die Knoten-ID des Elternknotens definierten Stelle zu speichern. Dadurch wird genau für Geschwisterknoten eine eindeutige Speicherstelle erzeugt, über die das Minimum gebildet werden muss, und es kann nicht zu Konflikten mit anderen Knoten im Dokument kommen. In der zweiten Stufe wird wiederum für alle Knoten, die dem Knotentest genügen, geprüft, ob sie entsprechend des am Elternknoten verzeichneten lokalen Minimums zur Ausgabekontextmenge gehören.

8.6.10 *preceding-sibling*-Achse

Die *preceding-sibling*-Achse ist wiederum symmetrisch zur *following-sibling*-Achse. Die Implementierung erfolgt entsprechend unter Verwendung der lokalen Maxima anstelle der lokalen Minima.

8.7 Evaluierung

8.7.1 Semantik & Prädikate

In der Entwicklung der verwendeten mengenweisen Semantik und dementsprechend auch in der Implementierung der Location Steps wurden, wie bereits erwähnt, Prädikate ignoriert. Zwar sind die meisten Ausdrücke, die in Prädikaten vorkommen, einfache Knotenfilter, die Knoten auf bestimmte knotenspezifische Eigenschaften, etwa den Namen oder das Vorhandensein bestimmter Attribute, testen und ggf. aus der Ausgabekontextmenge ausschließen. Es gibt jedoch auch Prädikate, die kontextspezifische Eigenschaften prüfen, etwa die Position in Dokumentreihenfolge in der Ergebniskontextmenge bzgl. eines Knotens der Eingabekontextmenge, wie es schon im Beispiel in der Einführung in Abschnitt 2.3 dargestellt wurde. Diese Information, welche Position ein Knoten in der von einem bestimmten Kontextknoten erzeugten Ausgabekontextmenge hat, geht in der mengenweisen Semantik bzw. der entsprechenden Implementierung verloren, da Location Steps eben nicht knotenweise ausgewertet werden.

Da für alle anderen Prädikate der hier vorgestellte Ansatz funktioniert, hat die mengenweise Auswertung natürlich trotzdem ihre Berechtigung. Mehr noch, es wurde gezeigt, dass die Semantiken für jeden Location Step ausgetauscht werden können. Daher kann die mengenweise Semantik auch einfach für passende Location Steps genutzt werden, selbst wenn es im verarbeiteten Location Path Prädikate gibt, die nicht mengenweise ausgewertet werden können.

8.7.2 Komplexität

Bevor die Ergebnisse der Tests unserer Implementierung dargelegt werden, soll zunächst die Zeit-Komplexität des hier verwendeten Ansatzes diskutiert werden. Der Aufwand für das Sortieren der Daten, die Bildung der Hashwerte usw., die einmalig anfangs anfallen, werden hier nicht weiter betrachtet.

Die Auswertung eines Location Paths folgt in der hier dargelegten Implementierung der Gleichung 8.11. Die Eingabe eines jeden Location Steps ist eine Teilmenge der Gesamtknotenmenge, die Ausgabe ist ebenfalls eine Teilmenge der Gesamtknotenmenge. Mit diesen Eingaben und Ausgaben können die Location Steps einfach hintereinander ausgeführt werden, so dass die Laufzeit linear in der Anzahl der Lo-

cation Steps skaliert. Wie auch in den folgenden Tests gezeigt wird, folgen andere XSLT-Prozessoren eher der Gleichung 8.4. Dies führt jedoch zu einer Komplexität von $O(c^k)$ für einen Location Path mit k Location Steps und mit Kosten $O(c)$ zur Auswertung eines Location Steps.

Die Zeitkomplexität der einzelnen Location Steps lässt sich leicht aus den angegebenen Implementierungen ablesen. Grundsätzlich gilt natürlich für alle Achsen, dass für beliebige (das heißt ohne weitere Einschränkungen) Eingabedokumente mit n Knoten die Komplexität in $\Omega(n)$ liegt, da ggf. für alle Knoten ein Thread gestartet werden muss. Da die Kosten für die Nutzung der Indizes nur einmalig für die Auswertung eines Location Steps anfallen, ändert diese Operation nichts an der Gesamtkomplexität. Bei den meisten Achsen wird von jedem Thread nur eine konstante Anzahl Operationen durchgeführt, so dass für diese Achsen die Gesamtkomplexität $O(n)$ ist. Nur bei den Achsen mit Schleifen, also bei der *descendant*- sowie der *ancestor*-Achse, ist die Anzahl der durchgeführten Operationen nicht konstant, sondern muss mit der Höhe des XML-Baums abgeschätzt werden. Demnach ist für diese Achsen die Komplexität gerade $O(n \cdot d)$ bei einer Höhe d des XML-Baums.

Insgesamt ist in der hier verwendeten Implementierung der Zeitaufwand zur Auswertung eines Location Paths mit k Location Steps, einer Baumhöhe d und einer Knotenzahl n also $O(k \cdot d \cdot n)$.

8.7.3 Testumgebung

Es wurde hier keine vollständige Implementierung des XPath-Standards erarbeitet, sondern nur die Verarbeitung von Achsen und Knotentests zur Auswertung von Location Paths betrachtet. Im Rahmen der oben genannten Arbeit von Bötel [20] wurden die hier vorgestellten Algorithmen zur Ausführung mittels einer GPU in CUDA implementiert. Der CUDA-Code wurde in den Xalan-XSLT-Prozessor [80] eingebettet. Die Idee ist dabei, die GPU transparent für den Nutzer einzusetzen; der Nutzer sollte also nicht bemerken (müssen), ob das übliche Xalan Programm oder aber die um die CUDA-Unterstützung erweiterte Variante genutzt wird.

Dazu werden beim Start des XSLT-Prozessors zunächst neben den Xalan-Datenstrukturen auch die entsprechendenen Strukturen für die GPU erstellt. Muss im Verlauf einer XSLT-Transformation ein geeigneter XPath-Ausdruck verarbeitet werden, so bricht der Prozess aus Xalan aus und verwendet die GPU-basierten Al-

gorithmen und liefert die Ergebnisse wiederum an Xalan zurück. Grundsätzlich ist also ein gewisser Mehraufwand bei der Nutzung der neuen Algorithmen vorhanden, sowohl einmalig bei der Initialisierung als auch bei jeder erneuten Anwendung, da jeweils zunächst bestimmte Daten auf die GPU übertragen werden müssen, ein Kernel gestartet wird und Ergebnisse eingesammelt werden.

Durch die Einbettung in Xalan entsteht eine sehr komfortable Umgebung zum Evaluieren der Algorithmen. Die zu testenden XPath-Ausdrücke können einfach in XSLT-Stylesheets angegeben werden, und durch die Ausgabe der Transformation ist auch eine einfach Korrektheitsüberprüfung möglich. Da zudem die neuen Algorithmen nur für die gewünschten Teile der Auswertung eingesetzt werden müssen und die anderen Bestandteile weiterhin von Xalan umgesetzt werden, stellt die Einbettung der Algorithmen in Xalan einen vollwertigen hybriden XSLT-Prozessor dar, der den vollen XSLT `1.0` (und demnach XPath `1.0`) Standard unterstützt.

Die Zeitmessung der Verarbeitung der XPath-Ausdrücke kann durch die Einbettung in Xalan sehr genau erfolgen und es können auch exakt die entsprechenden Operationen des Xalan-Codes verglichen werden. Zusätzlich zur reinen Xalan Variante und der Variante mit unseren Algorithmen wird auch Saxon-Prozessor [48] mit der *S9API* für Java genutzt. Weiterhin wurde der libXSLT-Prozessor [70] getestet. In diesen Prozessor wurden jedoch für die Tests die Mechanismen zur Zeitmessung nicht integriert. Stattdessen werden immer zwei Zeitmessungen durchgeführt. Zunächst wird eine Normierungszeitmessung mit einem geeigneten Stylesheet durchgeführt. Im zweiten Durchgang wird nur der Ausdruck

```
<xsl:value-of select="count(XPATH-AUSDRUCK)">
```

an geeigneter Stelle in das Normierungsstylesheet eingefügt. Dieser Ausdruck zählt nur die Knoten der zurückgelieferten Knotenmenge. Da in der seriellen Verarbeitung von libXML das Zählen der Knoten keine große Rolle spielt und zudem durch diesen Ausdruck keine weiteren XSLT-Aktionen angestoßen werden, wird die Verarbeitungszeit des XPath-Ausdrucks mit der Differenz der beiden Messungen abgeschätzt. Es wurde in den Tests das in Abschnitt 4 vorgestellte Testsystem verwendet. Die GPU-Algorithmen wurden auf der GTX580 GPU verarbeitet.

8.7.4 Test 1: Parallele / sequentielle Ausführung

Zunächst wurden Tests durchgeführt, bei denen die Komplexität der Algorithmen eine untergeordnete Rolle spielt und stattdessen die Leistung der Ausführung auf

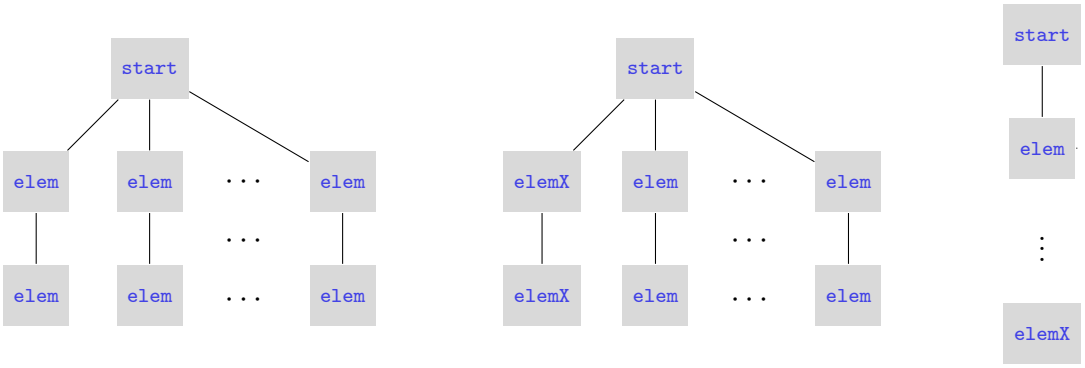


Abbildung 8.1: Evaluierung XPath: Test-XML-Dokumente

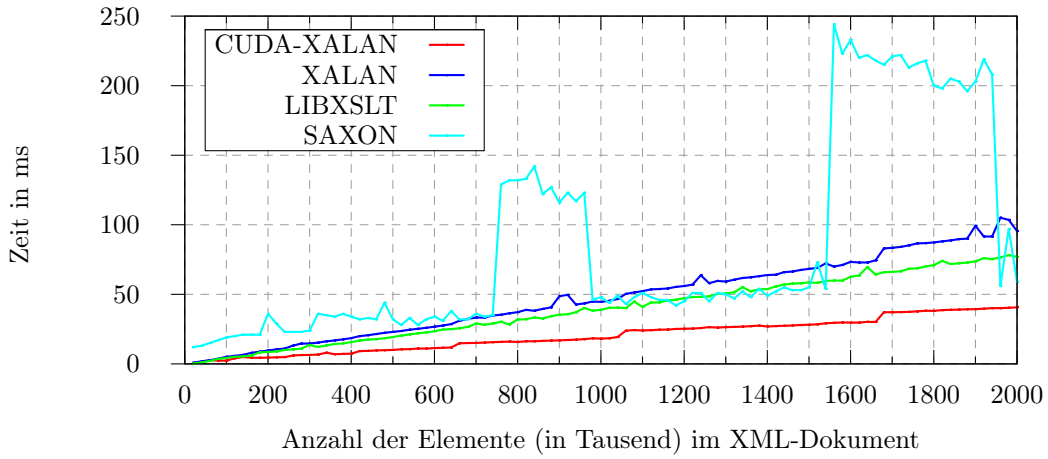


Abbildung 8.2: Evaluierung XPath: `/descendant::elem`

der CPU mit der Leistung der Ausführung auf der GPU verglichen wird. Für den ersten Test wurde ein sehr einfaches XML-Dokument verwendet, welches schematisch im XML-Baum links in Abbildung 8.1 dargestellt wurde: Die Höhe des Baums ist nur drei, der Wurzelknoten hat eine gewisse Anzahl von Kindknoten, die jeweils genau einen Nachkommen haben. Alle Knoten außer dem Wurzelknoten heißen `<elem>`. Die Größe des Dokuments wird durch die Anzahl der Kinder des Wurzelknotens konfiguriert.

Es wurde der Ausdruck `/descendant::elem` ausgewertet. Da dabei bis auf den Wurzelknoten alle Knoten des Dokuments ausgewählt werden, entsteht durch die Parallelität ein bestmöglicher Vorteil. Gleichzeitig entstehen aber keine weiteren Vorteile (Komplexität o.Ä.) durch die Nutzung unserer Algorithmen, da jedes Ele-

ment mindestens einmal bearbeitet werden muss. Im Gegenteil, durch den explizit durchgeführten Knotentest entsteht sogar ein gewisser Mehraufwand, der jedoch wegen der nur logarithmischen Komplexität der Suche in den Indizes das Ergebnis kaum beeinflussen wird.

Abbildung 8.2 zeigt die Ergebnisse dieses Tests. Wie zu erwarten steigt die Laufzeit aller Prozessoren linear in der Anzahl der Knoten des XML-Dokuments. Die Laufzeit der GPU-basierten Variante ist kleiner als die der anderen Varianten. Der vergleichsweise geringe Gewinn (an der Anzahl der parallelen Recheneinheiten gemessen) ist durch den bereits erwähnten Mehraufwand begründet, der bei jeder Nutzung der GPU entsteht, bei einer gleichzeitig recht geringen absoluten Laufzeit. Bei einer größeren absoluten Laufzeit, etwa bei der Durchführung aufwendigerer XPath-Ausdrücke, würde der Vorteil der großen Parallelität sicherlich deutlicher werden.

Die Laufzeitschwankungen beim Saxon-Prozessor sind keine Schwankungen in der Messung, sondern lassen sich reproduzieren. Es wurde nicht genau ergründet, warum die Laufzeit so stark schwangt. Da solche Schwankungen aber auch bei anderen Tests beobachtet wurden und die Schwankungen immer in Abhängigkeit von der Dokumentgröße auftreten, liegt die Vermutung nahe, dass im Prozessor nach einer bestimmten Heuristik andere Subroutinen zur Verarbeitung gewählt werden.

Passend zum ersten Test, der für die parallele Ausführung optimal ist, soll nun auch der ungünstigste Fall betrachtet werden. In einem XML-Dokument wie in Abbildung 8.1 rechts, welches einen Pfad variabler Länge darstellt, soll dazu der XPath-Ausdruck `/descendant::elemX/ancestor::start` ausgewertet werden. Das bedeutet, dass im ersten Location Step genau der letzte Knoten des Pfades ausgewählt wird, von dem ausgehend wiederum im zweiten Location Step alle Vorfahren `<start>` ausgewählt werden sollen. Entsprechend des von uns verwendeten Algorithmus für *ancestor*-Achsen muss dazu ein einzelner Thread über alle Knoten im Dokument iterieren. Es handelt sich also um eine rein sequentielle Ausführung.

Die Abbildung 8.3 zeigt die Ergebnisse dieses Tests. Wieder steigt die Laufzeit aller getesteten Prozessoren linear in der Anzahl der Elemente im Dokument. In diesem Test benötigt jedoch die CUDA-Variante deutlich mehr Zeit. Dies liegt, neben dem bereits erwähnten generellen Mehraufwand, unter anderem am niedrigeren Takt der GPU gegenüber der CPU.

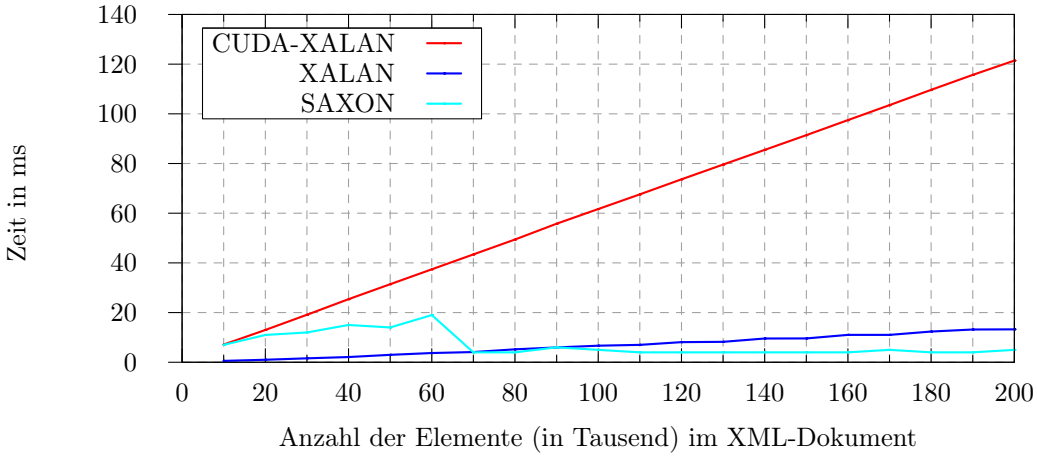


Abbildung 8.3: Evaluierung XPath: `/descendant::elemX/ancestor::start` (1)

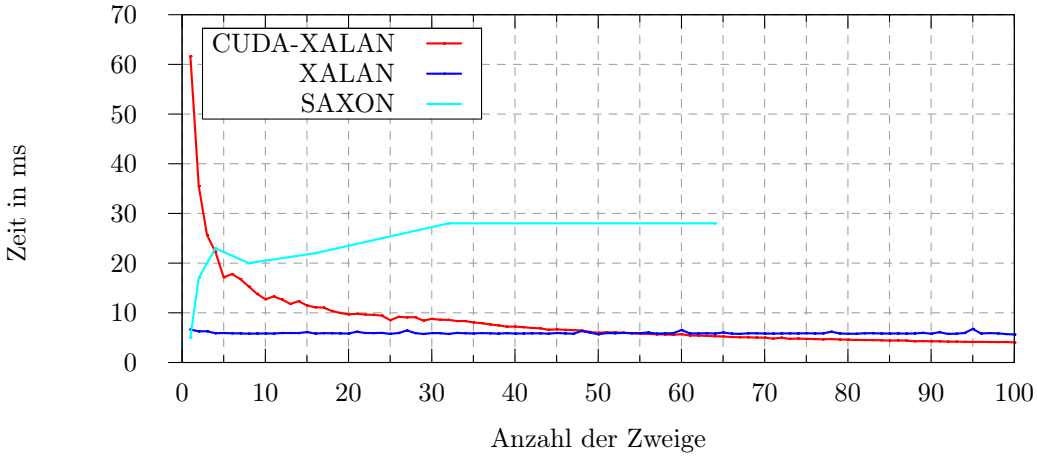


Abbildung 8.4: Evaluierung XPath: `/descendant::elemX/ancestor::start` (2)

Es soll an einem weiteren Beispiel verdeutlicht werden, wie von der Parallelverarbeitung profitiert wird. Dazu wird im Prinzip der gerade durchgeführte Test wiederholt. Diesmal jedoch wird eine feste Anzahl an Elementen im Dokument verwendet, die gleichmäßig auf eine variable Anzahl Pfade (Äste) des Baums verteilt wird. Außerdem befindet sich nun am Ende eines jeden Astes ein `<elemX>` Element.

Die Abbildung 8.4 zeigt die Ergebnisse. Da die gleiche Anzahl an Knoten zu verarbeiten ist, bleibt die Laufzeit der CPU-Prozessoren auch nahezu konstant (bis auf die erwähnten Schwankungen des Saxon-Prozessors). Die Laufzeit unserer

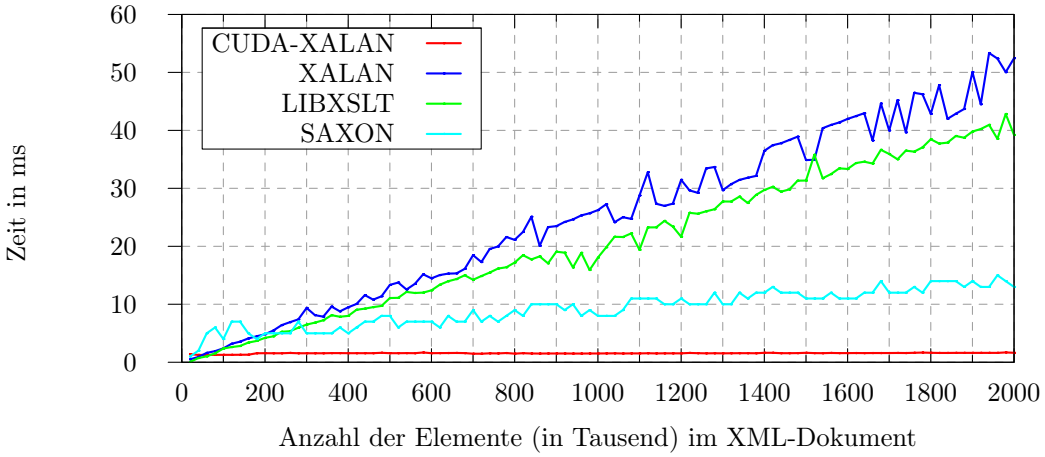


Abbildung 8.5: Evaluierung XPath: `/descendant::elemX`

Algorithmen jedoch fällt stark ab, da alle Pfade parallel verarbeitet werden können.

Es sei noch erwähnt, dass die hier getesteten XML-Dokumente mit einer großen Höhe sehr ungewöhnlich sind. So kann etwa der libXML Prozessor Dokumente mit einer Höhe größer als 256 nur unter Angabe spezieller Optionen parsen. Doch selbst mit den entsprechenden Optionen treten beim Parsen durch libXML Fehler auf bei den hier verwendeten Baumhöhen, so dass diese Dokumente mit libXML nicht bearbeitet bzw. getestet werden können. Xalan kann diese Dokumente verarbeiten, jedoch scheinen solche Baumhöhen auch hier nicht vorgesehen zu sein: Zwar treten beim Parsen keine Fehler auf, jedoch werden zum Parsen des Dokuments mit der größten Höhe bereits etwa 20 Minuten benötigt, während Dokumente mit gleicher Anzahl an Elementen aber beispielsweise mit logarithmischer Höhe in Sekunden vom Parser verarbeitet werden.

8.7.5 Test 2: Knotentest

Um die Auswertung der Knotentests zu evaluieren, wurde wieder ein angepasstes Dokument verwendet: Abbildung 8.1 (Mitte) zeigt das Dokument schematisch. Es ähnelt dem Dokument aus dem ersten Test, jedoch befindet sich im Dokument genau ein spezielles Element `<elemX>`. Ausgewertet wurde der XPath-Ausdruck `/descendant::elemX`, so dass nur dieses eine Element ausgewählt wird.

Die Abbildung 8.5 zeigt die gemessenen Laufzeiten. Die Laufzeit der CPU-Prozessoren ist ähnlich zu der im ersten Test. Dagegen ist die Laufzeit unserer Algo-

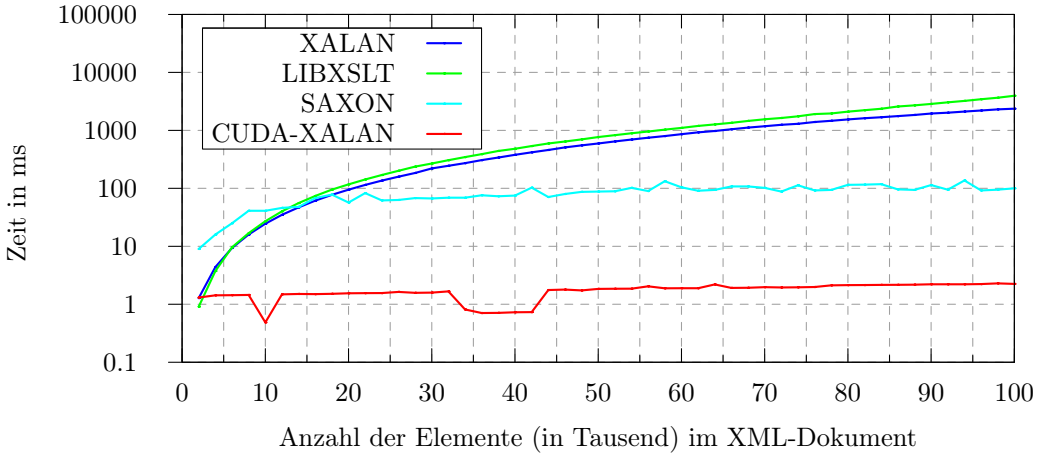


Abbildung 8.6: Evaluierung XPath: `/descendant::elem/descendant::elem`

rithmen sehr viel geringer und erscheint in der Abbildung konstant. Natürlich ist die Laufzeit entsprechend der binären Suche in den Indizes logarithmisch, jedoch ist der Einfluss der binären Suche in den hier getesteten Dokumenten so gering, dass er nicht in der Abbildung sichtbar wird.

8.7.6 Test 3: Komplexität

In diesem Abschnitt sollen die Messergebnisse für diejenigen Achsen bzw. Ausdrücke dargestellt werden, für die die getesteten Algorithmen unterschiedliche Komplexitäten haben. Zunächst verwenden wir wieder Dokumente entsprechend des Schemas aus Abbildung 8.1 (Links). Diesmal soll jedoch der Ausdruck

`/descendant::elem/descendant::elem`

ausgewertet werden.

Die Abbildung 8.6 zeigt die Ergebnisse. Die Laufzeit unserer Algorithmen liegt wieder in der gleichen Größenordnung wie im ersten Test. Die Laufzeit der CPU-Prozessoren allerdings ist nicht nur deutlich höher, sondern hat offenbar auch einen über-linearen, vermutlich quadratischen Verlauf bezüglich der Dokumentgröße. Man beachte, dass in diesem Fall nur Dokumente bis zu einer Größe von 0,1 Millionen Elementen betrachtet wurden, während in Abbildung 8.2 noch Dokumente mit bis zu 2 Millionen Elementen getestet wurden.

Diese Ergebnisse passen zu den Überlegungen zur Komplexität der Xalan-Auswertung, die die Anzahl der Location Steps im Exponenten vorsieht. In einem

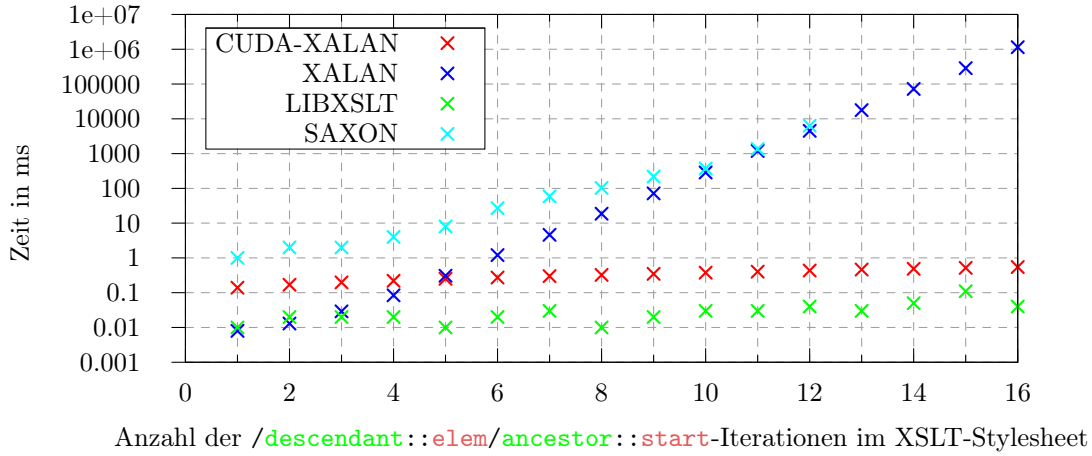


Abbildung 8.7: Evaluierung XPath: /descendant::elem/ancestor::start

weiteren Test soll dies überprüft werden: Dazu wird ein Dokument mit nur vier Elementen erzeugt (wie in Abbildung 8.1 (Links) mit nur zwei Ästen). Auf diesem Dokument werden nur Ausdrücke der Form

$$(/\texttt{descendant::elem/ancestor::start})^k$$

ausgewertet, also Location Paths mit k Iterationen des obigen Location Paths. Das Ergebnis ist in Abbildung 8.7 dargestellt. Wie vermutet steigt bei dem Xalan- sowie dem Saxon-Prozessor die Laufzeit exponentiell in der Anzahl der Location Steps, während der Anstieg unter Nutzung unserer Algorithmen nur linear ist. Der libXSLT-Prozessor scheint anders als die anderen Prozessoren diesen Fall gesondert zu behandeln. Seine Laufzeit liegt noch deutlich unter der des GPU-Prozessors, da natürlich generell ein CPU-Prozessor für ein Dokument mit nur vier Elementen besser geeignet ist als ein GPU-Prozessor.

Es sei noch erwähnt, dass der Test des Saxon-Prozessors bei 12 Iterationen abgebrochen werden musste, da dieser Prozessor im getesteten Fall auch exponentielle Speicherkomplexität aufweist. Bei 13 Iterationen reichten die dem Prozessor zur Verfügung gestellten 2 GByte Speicher nicht mehr zur Verarbeitung des wenige Byte großen Dokuments aus.

8.7.7 Test 4: following-[sibling]-Achse

Zum Testen der *following-sibling*-Achse verwenden wir Dokumente wie in Abbildung 8.8 dargestellt. Unter dem Wurzelelement befinden sich dabei nebeneinan-

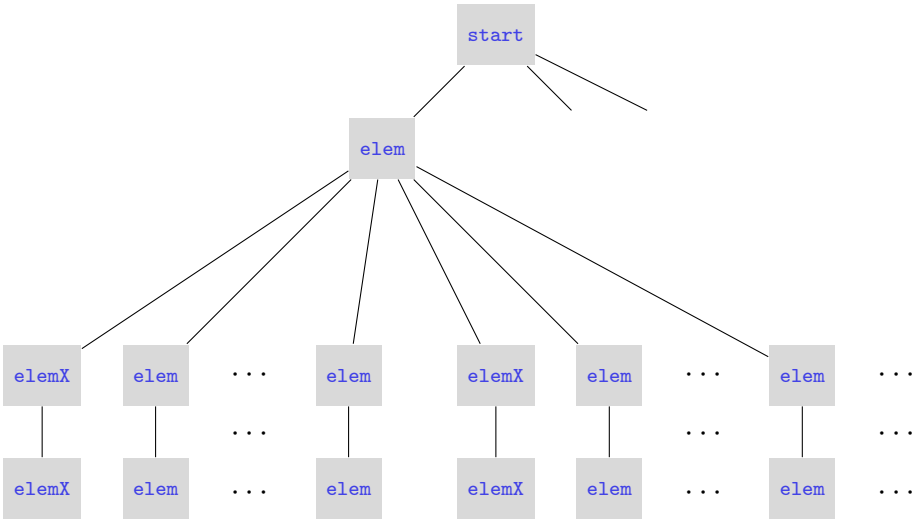


Abbildung 8.8: Evaluierung XPath: Test-XML-Dokumente (2)

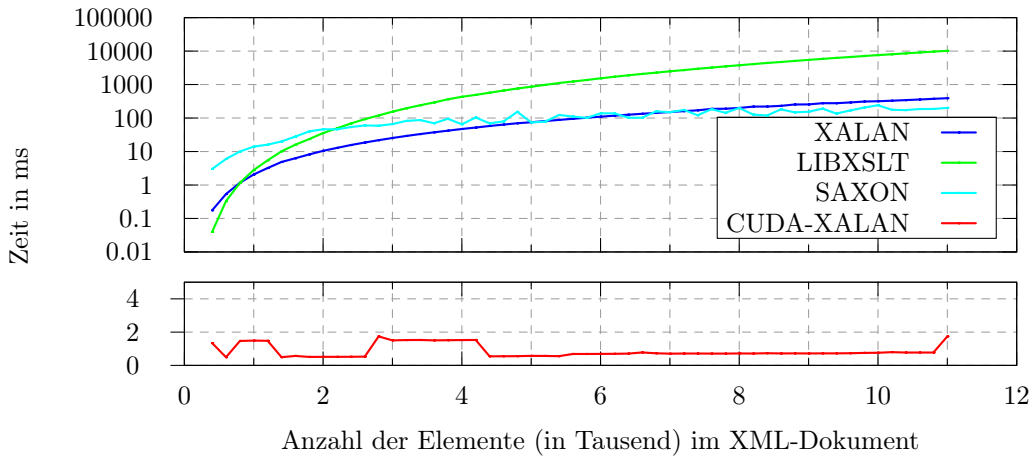


Abbildung 8.9: Evaluierung XPath: `/descendant::elemX/following::*`

der zehn identische Unterbäume (nur einer dargestellt), in denen `<elemX>`- und `<elem>`-Elemente im Verhältnis 1:9 gemischt sind. Ähnlich sieht auch der Baum zum Testen der *following*-Achse aus: Es wird genau einer der Unterbäume als Dokument verwendet, das Mischungsverhältnis ist 1:99.

Die Auswertung der *following* bzw. *following-sibling*-Achse geschieht in unserer Variante mit Hilfe atomarer Additionsoperationen. Diese sind in CUDA vergleichsweise schnell, so dass die Leistungsfähigkeit unserer Algorithmen auch für *following*- und *following-sibling*-Achsen recht hoch ist. Im Gegensatz dazu sind die

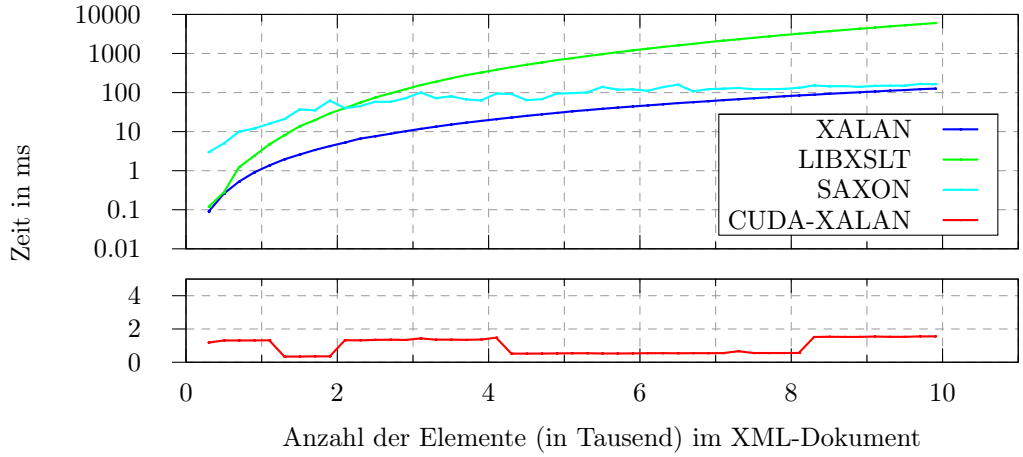


Abbildung 8.10: Evaluierung XPath: `/descendant::elemX/following-sibling::*`

CPU-Implementierungen dieser Achsen (insbesondere im Xalan-Prozessor) erheblich langsamer als die Implementierungen anderer Achsen. Die Abbildungen 8.9 und 8.10 stellen die Ergebnisse der Laufzeituntersuchung dar.

8.7.8 Test 5: XMark

In diesem Test werden nun einige ausgewählte XPath-Ausdrücke auf Dokumente angewendet, die nicht speziell für die konkreten Ausdrücke ausgewählt worden sind. Dazu verwenden wir das XMark [13] Programm, mit dem XML-Dokumente mit ähnlicher Struktur aber unterschiedlicher Größe erzeugt werden können. Die Dokumente orientieren sich dabei in ihrer Struktur, der Länge von Knotennamen, der Anzahl und Bezeichnung der Attribute usw. an Dokumenten aus realen Anwendungen. Die getesteten Ausdrücke sind in der Tabelle 8.1 angegeben.

Die Abbildung 8.11 zeigt die Ergebnisse der Auswertungen der XPath-Ausdrücke. Dargestellt ist nur der direkte Vergleich zwischen Xalan und der um unsere Algorithmen erweiterte Xalan-Variante. Die Skala der y-Achse ist logarithmisch abgetragen. Das verwendete Dokument hat ungefähr die Größe 12 MByte.

Es ist deutlich zu sehen, dass unsere Implementierung für dieses Anwendungsbeispiel für alle Ausdrücke schneller arbeitet als die Xalan-Variante. Oft beträgt der Unterschied Faktor 10 oder sogar Faktor 100. Wie schon die vorangegangenen Tests vermuten ließen, sind die Leistungsunterschiede für einfache Ausdrücke wie `/descendant::item` vergleichsweise klein. Für Ausdrücke mit mehr Location Steps

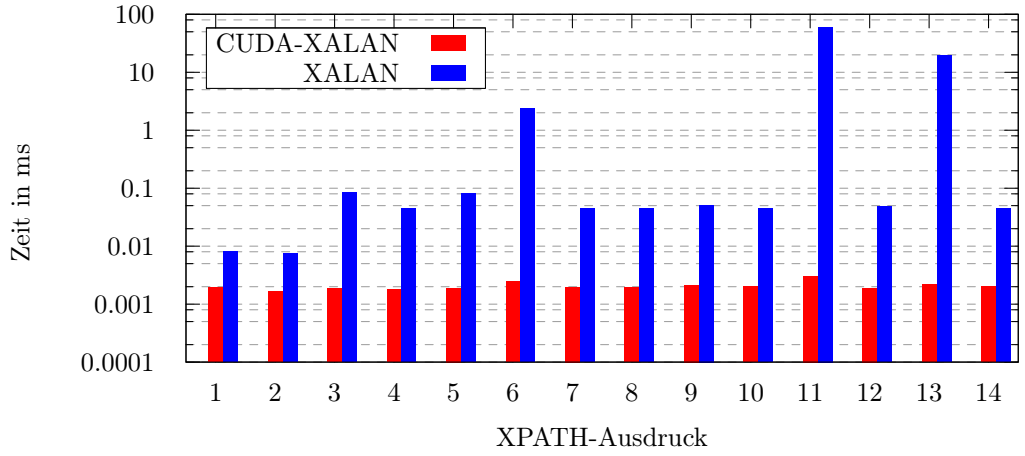


Abbildung 8.11: Evaluierung XPath: XMark (1)

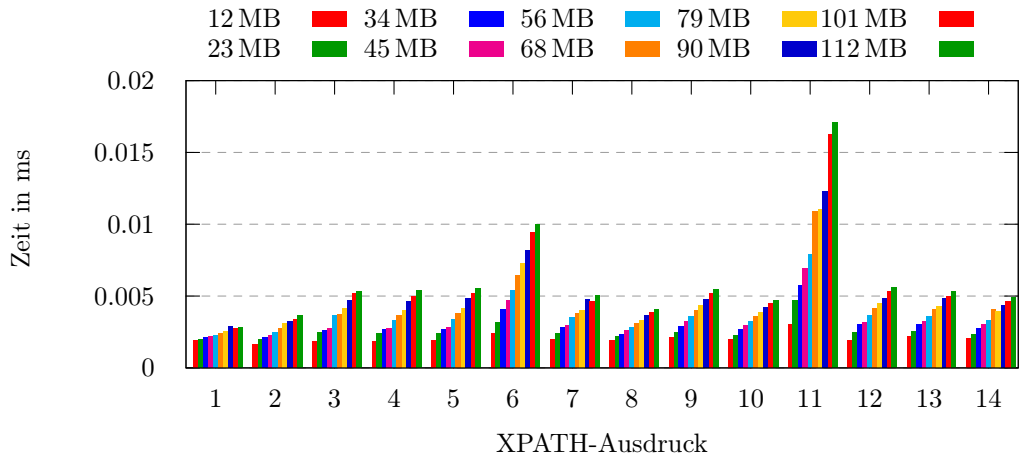


Abbildung 8.12: Evaluierung XPath: XMark (2)

1. /descendant::item
2. /descendant-or-self::bidder
3. //self::bidder
4. //bidder
5. //bidder/personref/@person
6. //bidder/descendant::text()
7. //annotation/descendant-or-self::keyword
8. //item/parent::europe
9. //emph/ancestor::description
10. //item/ancestor-or-self::*
11. //bidder/following::*
12. //bidder/following-sibling::bidder
13. //quantity/preceding::item
14. //quantity/preceding-sibling::location

Tabelle 8.1: Auswahl der getesteten XPath-Algorithmen

und insbesondere für Ausdrücke, die *following* oder *preceding*-Achsen verwenden, sind die Unterschiede dagegen sehr groß.

Die Skalierung der hier entwickelten parallelen Algorithmen mit der Dokumentgröße sind in Abbildung 8.12 dargestellt. Für alle XPath-Ausdrücke ist die Skalierung linear.

8.8 Zusammenfassung XPath

In diesem Abschnitt wurden unsere Ansätze der Verarbeitung von XPath-Location Paths basierend auf der im vorherigen Abschnitt definierten Datenstruktur vorgestellt. Es wurde für unseren Anwendungsfall eine mengenweise Semantik vorgestellt und die Äquivalenz zur knotenbasierten Semantik und damit die Korrektheit der hier entwickelten Ansätze gezeigt.

Es zeigt sich, dass schon durch die Wahl der zugrunde liegenden Semantik eine bestimmte Komplexität der Implementierung vorgegeben ist.

In umfangreichen Tests der Implementierung unserer Algorithmen als Erweiterung des bekannten Xalan-XSLT-Prozessors wurden viele Aspekte der Algorithmen getestet. In vielen Fällen ist ein Leistungsgewinn durch die große parallele

Rechenleistung der GPU möglich. Für manche XML-Dokumente, die jedoch zu den „ungewöhnlichen“ gezählt werden dürfen, ist die GPU-basierte Implementierung langsamer als die CPU-basierte.

In den Tests konnte auch für verschiedene XPath-Ausdrücke festgestellt werden, dass die Komplexität unserer Algorithmen besser als die der anderen XSLT-Prozessoren ist, etwa für Location Paths mit vielen Location Steps oder auch für die Auswertung von Knotentests. Dies ist jedoch kein Gewinn, der durch die große Parallelität der GPU erreicht wurde. Da in dieser Arbeit festgestellt werden soll, ob durch die Verwendung der GPU Vorteile entstehen können, werden diese Fälle nicht weiter diskutiert.

In den Tests mit den durch das XMark-Werkzeug generierten Dokumenten, war die hier vorgestellte Implementierung im direkten Vergleich immer schneller als der Xalan-Prozessor.

Als Ergebnis kann festgehalten werden, dass durch die Verwendung von GPUs durchaus ein Leistungsgewinn erzielt werden kann, wenn die verarbeiteten XML-Dokumente genügend groß sind und eine „gewöhnliche“ Struktur haben. Dies zeigen insbesondere die Abbildungen 8.2 und 8.4 sowie die Tests mit den XMark-Dokumenten in Abbildung 8.11.

KAPITEL 9

XSLT

In dem vorangegangenen Kapitel wurde die parallele Variante von XPath vorgestellt. In diesem Kapitel nun soll eine parallele Variante eines XSLT-Prozessors beschrieben werden. Im Abschnitt über XPath wurden Location Paths untersucht, die sich in Location Steps zerlegen lassen. Ein einzelner Location Step beschreibt wiederum einen auszuwertenden Ausdruck. Im hier verwendeten Ansatz wird diese Auswertung auf Basis einer mengenbasierten Semantik in massiv-parallelen Algorithmen gelöst.

Anders ist die Situation in XSLT: XSLT definiert nicht nur eine Syntax, in der Transformationen formuliert werden können. Es definiert auch ein Prozessmodell, nach dem diese Anweisungen ausgewertet werden müssen. Dieses Prozessmodell wiederum ist im Standard rekursiv beschrieben.

Unsere Herangehensweise an einen parallelen XSLT-Prozess sieht daher zunächst vor zu untersuchen, inwiefern sich das rekursive Prozessmodell von XSLT für die Architektur von GPUs umsetzen lässt. In einem zweiten Schritt wird einzeln für die drei grundsätzlichen Unterfunktionen des XSLT-Prozesses geprüft, inwiefern sich diese parallelisieren lassen.

Im Folgenden wird zunächst ein Überblick über die Umgebung gegeben, in der die genannten Eigenschaften des XSLT-Prozessors diskutiert werden sollen. In Abschnitt 9.2 wird das Prozessmodell von XSLT näher betrachtet und eine Herangehensweise erläutert, dieses Modell für GPUs zu übertragen. Abschnitt 9.3 diskutiert die Frage nach der Äquivalenz der unterschiedlichen Prozessmodelle. Danach werden in Abschnitt 9.4 die Unterfunktionen des Prozessmodells analysiert und

verschiedene Parallelisierungsansätze diskutiert. In Abschnitt 9.6 werden Ergebnisse der vorgenommenen umfangreichen Tests vorgestellt.

9.1 Überblick

Der hier vorgestellte XSLT-Prozessor wurde zusammen mit Christian Hoffmann im Rahmen seiner Diplomarbeit [42] entwickelt. In der vorliegenden Arbeit soll nur der grundsätzliche Ansatz zur Parallelisierung des XSLT-Prozessors beschrieben werden, nicht aber CUDA-spezifische Details der Implementierung und Ähnliches.

Deswegen wird hier zunächst ein grober Überblick über die Implementierung des XSLT-Prozessors gegeben, um das Szenario für die in den folgenden Abschnitten diskutierte Parallelisierung zu definieren. Für Details bezüglich der konkreten Implementierung des Prototyps und einiger Optimierungen sei auf [42] verwiesen.

Der XSLT-Prozessor wird als Modul des CUDA-OS eingesetzt. Eine Transformation wird deswegen grundsätzlich mit der festen Anzahl Threads eines CUDA-Blocks durchgeführt. Daher wird auch eine an das CUDA-OS angepasste Version der Algorithmen aus Abschnitt 8 verwendet.

Wie im Abschnitt 4 erläutert, wird für das Szenario angenommen, dass viele parallele Transformationen auf bekannten XML-Daten durchgeführt werden. Für diesen Fall gehen wir davon aus, dass das zu transformierende XML-Dokument in der in Abschnitt 7 erläuterten Struktur vorliegt. Wie dort bereits erwähnt wurde, befinden sich Textinhalte von Knoten, die Knotennamen und die Werte von Attributen nicht auf der GPU; der XSLT-Prozess arbeitet ausschließlich mit den Repräsentationen der Knoten.

Die Ausgabe des XSLT-Prozesses ist ebenfalls eine Struktur ähnlich zu der in Abschnitt 7. Bei der Verarbeitung durch das nachfolgend vorgestellte Prozessmodells gehen allerdings einige Eigenschaften (insbesondere bzgl. der Reihenfolge) verloren. Das ist für den XSLT-Prozess kein Problem, da das Ausgabedokument im CPU-Serialisierungsprozess wieder korrekt rekonstruiert wird (*Tree Glueing*). Allerdings kann deshalb die Ausgabe einer Transformation beispielsweise nicht sofort (das heißt ohne Serialisierungsprozess) wieder als Eingabe verwendet werden. Es gilt jedoch grundsätzlich (vgl. Abschnitt 2.4), dass die Ausgabedokumente eines XSLT-Prozesses im Allgemeinen keine XML-Dokumente sind und in diesen Fällen prinzipiell nicht als Eingabe einer Transformation verwendet werden können.

Im Folgenden soll weder der Prozess des Parsens noch der Prozess der Serialisie-

I. Lege leeren Zielbaum und leere aktuelle Knotenliste an. Füge den root Knoten des Quelldokuments in die aktuelle Knotenliste ein.

II. Für jeden Knoten in der aktuellen Knotenliste:
    1. Finde die Template Rule, deren Pattern am besten zum Knoten passt (*Matching*)
    2. Instanziiere das Template der gewählten Template Rule (*Instantiation*)
    3. Falls bei der Instanziierung eine weitere aktuelle Knotenliste ausgewählt worden sein sollte (*Selection* mit `apply-templates`, `for-each`), wende rekursiv Schritt II auf die neue aktuelle Knotenliste an.

Quellcode 9.1: Rekursives XSLT-Prozessmodel

rung besprochen werden; für diese Aspekte sei nochmals auf [42] verwiesen. Stattdessen werden das abstrakte Prozessmodell des verwendeten XSLT-Prozessors sowie die verschiedenen Parallelisierungsansätze für die Unterfunktionen diskutiert.

9.2 XSLT Prozessmodell

Eine Definition des XSLT Prozessmodells findet sich (in Prosa) im Standard [85, Abschnitt 5.1]. Ein Ausschnitt ist in Anhang D gegeben. Die Beschreibung ist äquivalent zum Quellcode 9.1.

Dieser Ansatz ist rekursiv und (zumindest in der vorliegenden Beschreibung) auf die Knoten zentriert: Ein Dokument wird verarbeitet, indem der *root*-Knoten in die aktuelle Knotenliste eingefügt und die Verarbeitung dieser Knotenliste gestartet wird. Wird die Verarbeitung eines Knotens aus der aktuellen Knotenliste begonnen, so wird dieser Knoten vollständig bis zum Ende verarbeitet (inklusive der Rekursion), bevor die Verarbeitung des nächsten Knotens der gleichen aktuellen Knotenliste begonnen wird.

Dieser knotenzentrierte Ansatz ist für unsere Anwendung, die Verarbeitung von XSLT auf GPUs, nur bedingt geeignet. Der rekursiven Definition in der Implementierung folgend wäre es schwierig, eine parallele Darstellung des gesamten Prozesses zu finden. Würde beispielsweise für eine gegebene Knotenliste gerade eine Recheneinheit einen Knoten vollständig verarbeiten (also auch ggf. in der Rekursion an-

I. Instanziiere Template für *root*-Knoten. XSL-Elemente werden dabei nicht rekursiv ausgewertet, sondern in den Resultatsbaum kopiert (*Lazy Instantiation*).

II. Durchsuche den Resultatsbaum nach allen noch nicht evaluierten XSL-Elementen (Suchrunde).

III. Falls kein XSL-Element gefunden wurde, ist der Prozess beendet. Falls XSL-Elemente gefunden wurden, evaluiere die XSL-Elemente. Wird dabei eine neue Knotenliste erzeugt (*Selection*), dann für jeden Knoten in der neuen Knotenliste:

    1. Finde die Template Rule, deren Pattern am besten zum Knoten passt (*Matching*)
    2. Instanziiere das Template der gewählten Template Rule. XSL-Elemente werden dabei nicht rekursiv ausgewertet, sondern in den Resultatsbaum kopiert (*Lazy Instantiation*).

IV. Springe zu II.

Quellcode 9.2: Iteratives XSLT-Prozessmodel

fallende Berechnungen), so ist die Lastverteilung zwischen den Recheneinheiten völlig unklar.

Zudem ist Rekursion kein triviales Konzept und ist auch erst mit NVIDIAs GPUs der neuesten Generation möglich. Grundsätzlich ist es aber auf parallelen Architekturen mit sehr leichtgewichtigen Recheneinheiten (wie Threads auf GPUs) gut, auf Rekursion zu verzichten, da Rekursion zusätzlichen Verwaltungsaufwand und Speicherplatz für jede Recheneinheit benötigt.

Der Standard erlaubt jedoch auch andere Prozessmodelle, solange das Resultat äquivalent zum im Standard vorgeschlagenen Modell ist. In dieser Arbeit wurde deshalb eine andere Darstellung des XSLT-Prozesses entwickelt, welche iterativ arbeitet und weniger knotenzentriert ist (Quellcode 9.2).

Am Anfang der Verarbeitung wird wie im Standard zunächst der *root*-Knoten instanziiert. Wird ein Knoten der aktuellen Knotenliste verarbeitet, das heißt, es wird das für ihn passende Template instanziiert, so würde im rekursiven Fall,

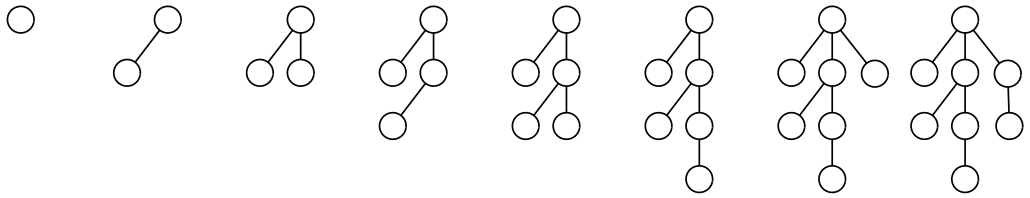


Abbildung 9.1: Rekursives Prozessmodell ∼ *DFS*

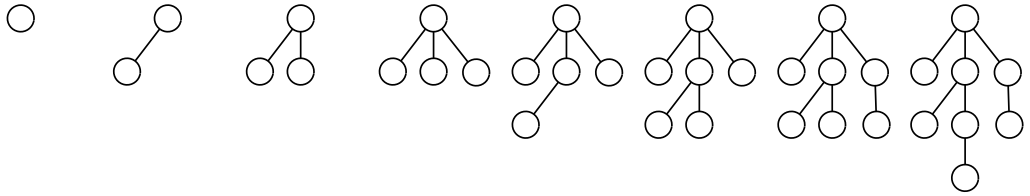


Abbildung 9.2: Iteratives Prozessmodell ∼ *BFS*

wenn eine entsprechende XSL-Anweisung bei der Instanziierung gefunden wird, die XSL-Anweisung sofort durch Rekursion aufgelöst. Im hier verwendeten iterativen Modell wird stattdessen die XSL-Anweisung wie ein gewöhnliches zu instanziierendes Element zunächst in den Resultatsbaum kopiert. Außerdem wird zusammen mit dem XSL-Element zur späteren korrekten Verarbeitung auch gespeichert, für welchen Knoten das Template gerade instanziiert wird.

Erst wenn für alle Knoten der aktuellen Knotenliste ein passendes Template instanziiert wurde, beginnt die nächste Runde, in der dann der Resultatsbaum nach noch nicht ausgewerteten XSL-Anweisungen durchsucht wird.

Diese Art der Instanziierung, bei der die Evaluierung der XSL-Elemente erst verzögert in der nächsten Suchrunde stattfindet, wird im Folgenden *Lazy Instantiation* genannt.

Unser Prozessmodell ist damit weniger knotenzentriert als listenzentriert: Es wird für alle Knoten einer aktuellen Knotenliste genau ein Template instanziiert. Erst wenn eine Knotenliste auf diese Weise abgearbeitet ist, beginnt die Bearbeitung der nächsten (Such-)Runde.

Der Unterschied im Prozessmodell kann besonders gut verdeutlicht werden, wenn die baumförmige Struktur des Ausgabedokuments einer XSLT-Transformation betrachtet wird. Dann nämlich entspricht die im Standard vorgeschlagene rekursive Verarbeitung einer *depth-first*-Verarbeitung (in Anlehnung an die Tiefensuche *DFS*), so dass die Elemente in der späteren Dokumentreihenfolge in den Resul-

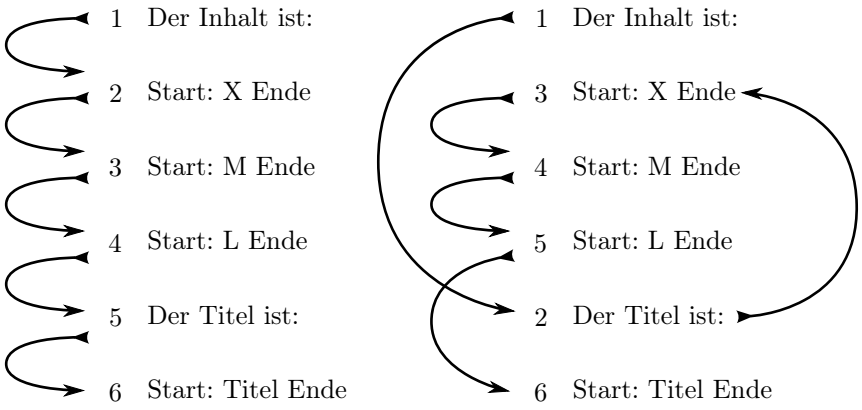


Abbildung 9.3: Ausführungspfade im rekursiven und iterativen Modell

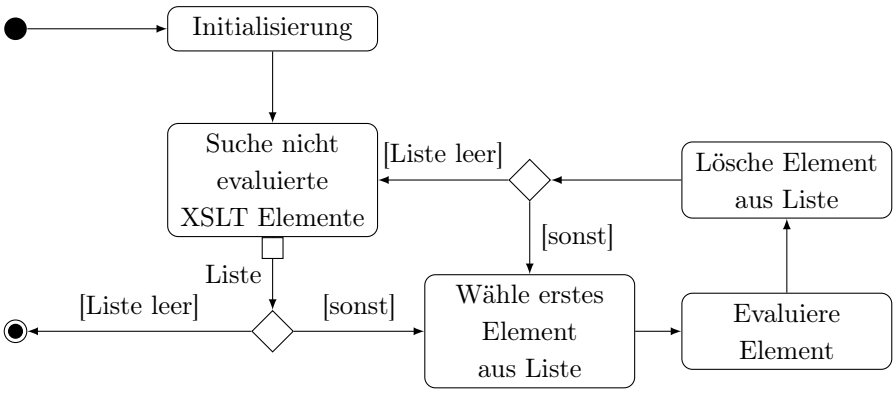


Abbildung 9.4: Aktivitätsdiagramm des iterativen Prozessmodells (aus [42])

tatsbaum eingefügt werden. Die hier verwendete iterative Variante entspricht einer *breadth-first*-Verarbeitung (Breitensuche *BFS*), bei der der Resultatsbaum nicht in der späteren Dokumentreihenfolge erstellt wird. Die Abbildungen 9.1 und 9.2 verdeutlichen diesen Unterschied.

Als Anwendungsbeispiel betrachten wir erneut die Transformation des XML-Dokuments aus Quellcode 2.1 mit dem Stylesheet aus Quellcode 2.2. Das Template, welches zum *root*-Knoten passt, enthält im Beispiel zwei

```
<xsl:apply-templates ...>
```

Anweisungen. Die unterschiedlichen Ausführungspfade der beiden Prozessmodelle sind in Abbildung 9.3 dargestellt und verdeutlichen auch den Unterschied der beiden Modelle bezüglich der Dokumentreihenfolge: links in der Abbildung wird

das Resultat in Dokumentreihenfolge erzeugt, rechts nicht.

Abbildung 9.4 zeigt das vereinfachte Aktivitätsdiagramm des XSLT-Prozessors. Die Initialisierung enthält hier bereits die Instanziierung des Templates, welches zum *root*-Knoten passt. Die im Aktivitätsdiagramm nur durch einen einzelnen Zustand dargestellte Evaluierung eines XSL-Elements ist der aufwendigste Verarbeitungsschritt im Prozess und beinhaltet die drei wesentlichen Unterfunktionen *Selection*, *Matching* und *Lazy Instantiation*.

9.3 Äquivalenz der Prozessmodelle

Das rekursive Modell und das iterative Modell unterscheiden sich nur in dem Punkt, wie XSL-Elemente ausgewertet werden, die eine neue Knotenliste auswählen. In der ursprünglichen Variante wird die Auswertung sofort vorgenommen und es werden ggf. Inhalte an fortlaufender Position (in Dokumentreihenfolge) in das Resultat eingefügt. In der iterativen Variante wird stattdessen zunächst nur ein Platzhalter zusammen mit der Knoten-ID des gerade verarbeiteten Knotens an die entsprechende Stelle in das Resultat geschrieben und die Auswertung des XSL-Elements später vorgenommen.

Es gilt:

- Die Auswertung der XSL-Elemente, die eine neue aktuelle Knotenliste erzeugen, ist unabhängig vom Zustand des Prozesses oder ähnlichem und wird nur durch den Knoten, für den das XSL-Element ausgewertet werden soll, bestimmt. Da aber gerade dieser Knoten zusammen mit dem XSL-Element im Resultatbaum gespeichert wird, kann die neue Knotenliste identisch zu der im rekursiven Modell erzeugt werden.

- In der rekursiven Variante werden die Inhalte des Resultatsbaums in der späteren Dokumentreihenfolge in das Dokument eingefügt. Diese Reihenfolge wird in der hier verwendeten Variante nicht eingehalten. Allerdings werden die Platzhalter für die später zu evaluierenden XSL-Elemente fortlaufend eingefügt. Das bedeutet, der Platzhalter eines noch nicht ausgewerteten XSL-Elements befindet sich genau an der Stelle im Resultat, an der auch die rekursive Variante begonnen hätte, fortlaufend Inhalte in das Resultat zu schreiben. Weil der Platzhalter aber an der „richtigen“ Position steht, kann

die korrekte Reihenfolge der Inhalte des Resultats auch in der iterativen Variante beim späteren Serialisieren leicht wiederhergestellt werden.

Da also in der iterativen Variante des Prozessmodells die gleichen aktuellen Knotenlisten wie in der rekursiven Variante erzeugt werden, und zudem auch schlussendlich die richtige Reihenfolge der Inhalte im Resultatsbaum wiederhergestellt werden kann, sind die Ergebnisse beider Prozessmodelle gleich. Demnach sind die Modelle äquivalent bezüglich der Ergebnisse, so wie vom Standard gefordert.

9.4 Parallelisierung der Bestandteile eines XSLT-Prozesses

In diesem Abschnitt sollen nun die Bestandteile eines XSLT-Prozesses auf ihre Parallelisierbarkeit bezüglich der XSLT-Verarbeitung auf GPUs im Kontext der Implementierung als Modul des CUDA-OS untersucht werden. Wie in Abschnitt 3 erläutert, ist es durch die SIMT-Architektur der NVIDIA-Grafikkarten nicht effizient möglich, Threads eines Warps unterschiedliche Instruktionen verarbeiten zu lassen. Threads unterschiedlicher Warps und Blöcke hingegen können durchaus ohne Leistungsverlust unterschiedliche Instruktionen verarbeiten. Sollen mehrere Aufgaben mit unterschiedlichen Ausführungsfaden parallel verarbeitet werden, so kann demnach nur blockweise oder warpweise parallelisiert werden, das heißt, alle Threads eines Blocks bzw. eines Warps arbeiten an der gleichen Aufgabe. Aufgaben mit gleichen Ausführungspfaden hingegen können auch threadweise verarbeitet werden. Die Verarbeitung einer Aufgabe durch alle Threads mehrerer Blöcke wird hier nicht betrachtet, denn im Kontext des CUDA-OS ist dies nicht vorgesehen.

Die Evaluierung der in einer Suchrunde des iterativen Prozessmodells (Quellcode 9.2) gefundenen XSL-Elemente wurde bereits als der aufwendigste Teil des gesamten Verarbeitungsprozesses erkannt. Zum einen kann die Evaluierung eines XSL-Elements parallelisiert werden, zum anderen könnten auch mehrere XSL-Elemente gleichzeitig evaluiert werden.

Gegen die gleichzeitige Verarbeitung mehrerer XSL-Elemente spricht, dass ggf. nur eine kleine Anzahl nicht-evaluierter XSL-Elemente im Resultat existiert, etwa in der ersten suchrunde der Transformation. Zudem führen unterschiedliche zu evaluierende XSL Elemente zu unterschiedlichen Ausführungspfaden, eine threadweise Parallelisierung wäre demnach ungünstig. Da aber auch die Verarbeitungs-

dauer unterschiedlicher XSL-Elemente sehr unterschiedlich ist, würde auch bei warpweiser paralleler Evaluierung der XSL-Elemente zusätzlicher Synchronisationsaufwand entstehen.

Deswegen wurde in der vorliegenden Arbeit die parallele Evaluierung je eines XSL-Elements untersucht. Die wichtigsten drei Bestandteile der Evaluierung im iterativen Prozessmodell sind:

1. *Selection*, die Auswahl einer neuen aktuellen Knotenliste mittels eines XPath-Ausdrucks.
2. *Lazy Instantiation*, die Verarbeitung eines Templates (ohne Rekursion) für jeden Knoten der aktuellen Knotenliste.
3. *Matching*, die Auswahl des passenden Templates für jeden Knoten der aktuellen Knotenliste.

In den folgenden Abschnitten werden diese drei Bestandteile auf ihre Parallelisierbarkeit untersucht.

9.4.1 Selection

Selection bezeichnet im Standard die Auswahl neuer aktueller Knotenlisten durch Auswertung der XPath-Ausdrücke in den `select`-Attributen in XSL-Elementen, etwa wie in Beispiel 2.2 durch:

```
<xsl:apply-template select="...">
```

Erlaubt sind dabei alle XPath-Ausdrücke, die zu Knotenmengen ausgewertet werden [85, Abschnitt 5.4]). Dementsprechend kommt für diesen Schritt die in Abschnitt 8 vorgestellte parallele Auswertung von XPath-Ausdrücken zum Einsatz. Da in einem XSL-Element nur ein XPath-Ausdruck auszuwerten ist, werden dafür alle Threads eines Blocks verwendet.

9.4.2 Matching

Das Matching sucht für jeden einzelnen Knoten der aktuellen Knotenliste das passende Template aus der Menge der Templates des Stylesheets heraus. Ob ein Template passt, wird dabei durch den XPath-Ausdruck im Pattern

```
<xsl:template match="XPATH-Ausdruck"/>
```

definiert. Laut Standard sind diese Ausdrücke Untermengen allgemeiner XPath-Ausdrücke, die zu Knotenmengen ausgewertet werden. Weiterhin ist nur die Verwendung der *child*-Achse und der *attribute*-Achse erlaubt.

Wie in Abschnitt 2.4 dargelegt wurde, passt ein Template gerade dann zu einem aktuellen Knoten, wenn der aktuelle Knoten in einer Knotenmenge liegt, die von dem XPath-Ausdruck bezüglich eines beliebigen Knotens erzeugt wird. Insbesondere bedeutet dies, dass die durch den XPath-Ausdruck erzeugten Knotenmengen für den Prozess nicht von Bedeutung sind; interessant ist lediglich die Frage, ob der aktuelle Knoten in einer der Knotenmengen liegt. In der Einführung in Abschnitt 2.4 wurde bereits erwähnt, dass durch die eingeschränkte Ausdrucksstärke nur wenige Knoten als Kontextknoten in Frage kommen; genauer sind es sogar nur Vorfahren des aktuellen Knotens (vgl. Abschnitt 8.5), die geprüft werden müssen. Da zudem diese Überprüfung für jeden aktuellen Knoten einzeln stattfindet, ist es hier nicht sinnvoll, die in Abschnitt 8 vorgestellten parallelen Algorithmen zu nutzen, um die XPath-Ausdrücke zu evaluieren.

Sattdessen ist es hier sinnvoll, zur Überprüfung eines Patterns und eines Knotens einen sequentiellen Ansatz zu wählen, etwa indem gerade ein Thread ein Pattern für einen Knoten prüft.

Grundsätzlich gilt es also, für eine aktuelle Knotenliste K und eine Patternmenge P alle Kombinationen $(k, p) \in K \times P$ zu überprüfen. Dabei kann je eine Überprüfung von je einem Thread durchgeführt werden. Es könnten also alle notwendigen Überprüfungen einfach gleichmäßig auf alle Threads verteilt werden. In diesem Fall würden im Prinzip schon bei einer im Vergleich zur Anzahl der Threads kleinen aktuellen Knotenliste und kleinen Menge von Pattern für jeden Thread Aufgaben bereitstehen. Auch die Lastverteilung (im Sinne von „Überprüfungen je Thread“) ist optimal.

Doch auch dieser Ansatz hat Nachteile: Je zwei Überprüfungen verschiedener Pattern erfordern nicht gleich viel Aufwand, sondern können entsprechend des überprüften Pattern im Aufwand variieren. Das würde zu abweichenden Ausführungspfaden und erhöhtem Synchronisationsaufwand führen. Unter Umständen muss zudem jeder Thread für jede Überprüfung (k, p) die Daten des Pattern p und des Knotens k laden — oder zumindest überprüfen, ob die Daten neu geladen werden müssen oder ob auch in der vorhergehenden Überprüfung schon die Daten von k oder p verwendet wurden.

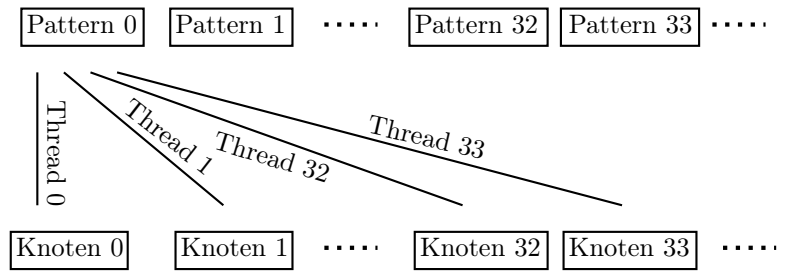


Abbildung 9.5: Pattern Matching: Alle Knoten und ein Pattern

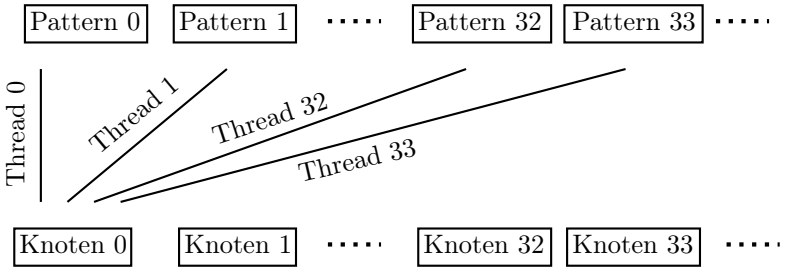


Abbildung 9.6: Pattern Matching: Ein Knoten und alle Pattern

Doch neben dem allgemeinen Ansatz, alle Aufgaben auf alle Threads zu verteilen, gibt es noch weniger allgemeine, dafür aber auch einfachere Lösungen:

1. Alle Threads überprüfen alle Knoten für ein Pattern (Abb. 9.5).
2. Alle Threads überprüfen alle Pattern für einen Knoten (Abb. 9.6).
3. Alle Threads eines Warps überprüfen alle Knoten für ein Pattern. Verschiedene Warps überprüfen nebenläufig verschiedene Pattern (Abb. 9.7).
4. Alle Threads eines Warps überprüfen alle Pattern für einen Knoten. Verschiedene Warps überprüfen nebenläufig verschiedene Knoten (Abb. 9.8).

Alle diese Ansätze haben Vorteile und Nachteile, die auch von der konkreten Anwendung (beispielsweise von dem konkreten XSLT Stylesheet) abhängen. Existieren etwa sehr viele Pattern im Stylesheet, aber nur sehr wenige Knoten in der aktuellen Knotenliste, so sind die Ansätze 2 und 4 günstig. Im Allgemeinen gilt jedoch, dass die Anzahl der Pattern im Stylesheet weit geringer ist als die Anzahl der Knoten eines XML-Dokuments, also auch als die mögliche Größe einer aktuellen Knotenliste. Für unsere prototypische Implementierung wurde deshalb nur

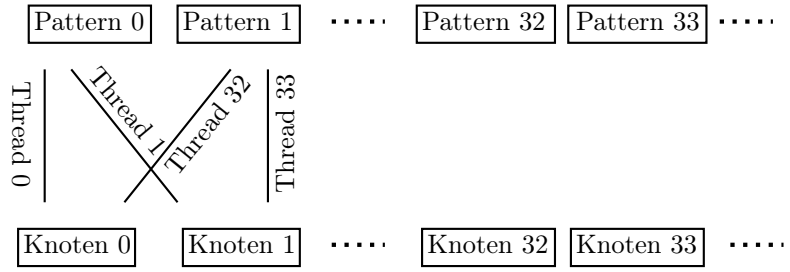


Abbildung 9.7: Pattern Matching: Warpweise alle Knoten und ein Pattern

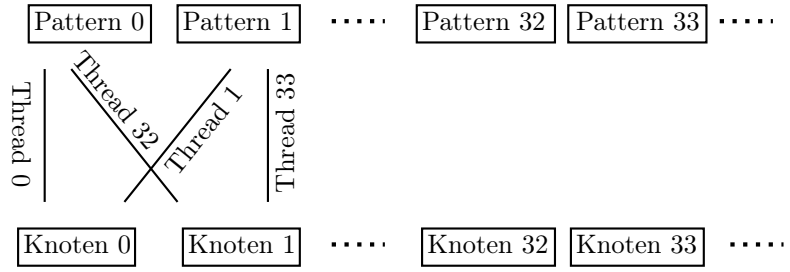


Abbildung 9.8: Pattern Matching: Warpweise ein Pattern und alle Knoten

Ansatz 1 implementiert. In diesem Ansatz ist auch möglichst gut gewährleistet, dass Threads nicht unterschiedliche Ausführungspfade haben, da die Überprüfung des gleichen Templates (mit dem oben genannten geringen Sprachumfang) in der Regel nicht zu stark abweichenden Ausführungspfaden führt: Da nur die *child*-Achse und die *attribute*-Achse vorgesehen sind, ist die maximaler Abweichung der Ausführungspfade zweier Threads begrenzt durch die (in der Praxis geringe) Höhe des Baums.

9.4.3 Lazy Instantiation

Die Instanziierung eines Templates ist in der ursprünglichen Variante des Prozessmodells rekursiv definiert: Der Templateinhalt muss in den Zielbaum kopiert werden und, falls im Templateinhalt weitere XSL-Anweisungen stehen, müssen diese rekursiv ausgewertet werden. Im hier verwendeten iterativen Prozessmodell dagegen kann der ganze XSLT-Prozess besser in einzelne Iterationen zerlegt werde. Im iterativen Modell muss zur Instanziierung eines Templates nur der gesamte Inhalt inklusive der XSL-Anweisungen in das Resultat kopiert werden.

Für diesen Kopiervorgang gilt wieder, dass das Kopieren eines Knotens des Tem-

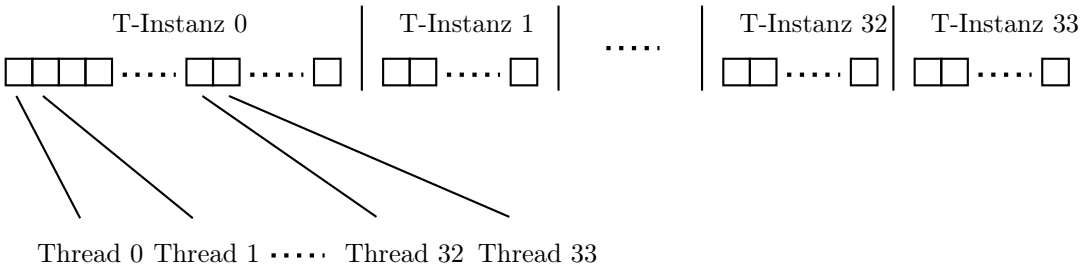


Abbildung 9.9: Template Instanz: Alle Knoten einer Templateinstanz

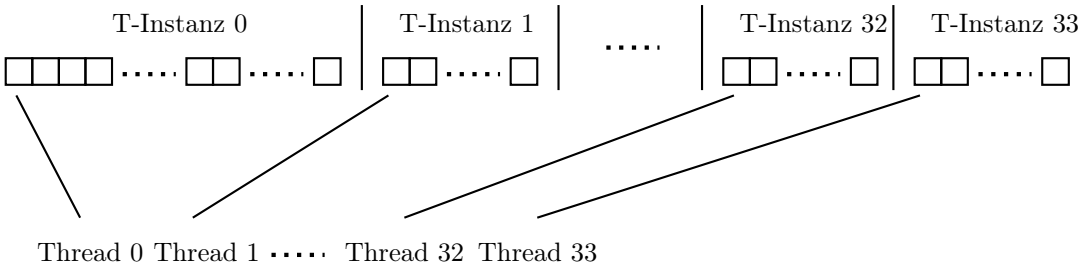


Abbildung 9.10: Template Instanz: Ein Knoten aller Templateinstanzen

plateinhalts (da ein XSLT-Stylesheet auch ein XML-Dokument ist, heißen die Inhalte des Templates ebenfalls Knoten) nicht weiter parallelisiert werden kann. Im Prinzip könnten also wieder alle notwendigen Kopieroperationen, das heißt die Instanziierung der Templates für alle Knoten der aktuellen Knotenliste, frei auf alle Threads verteilt werden. Jedoch hat dies auch in diesem Fall neben dem hohen Verwaltungsaufwand weitere Nachteile. Unter anderem gilt, dass bei einer solchen Vorgehensweise u.U. ungünstige (*non-coalesced*) Zugriffe in den Speicher entstehen, was die Leistung stark einschränken kann.

Zu beachten ist hier, dass für jeden Knoten der aktuellen Knotenliste ein Template zu instanziieren ist, das heißt, ein Template kann auch mehrfach für verschiedene Knoten instanziiert werden. Um Missverständnisse auszuschließen, sprechen wir von einem Template, wenn die Transformationsanweisung im Stylesheet gemeint ist, und von einer Templateinstanz, wenn eine Instanz des Templates für einen Knoten gemeint ist. Kopiert werden also im Folgenden Knoten einer Templateinstanz und nicht Knoten eines Templates.

Ähnlich wie beim Pattern Matching stehen auch hier wieder vier Möglichkeiten zur Parallelisierung zur Verfügung.

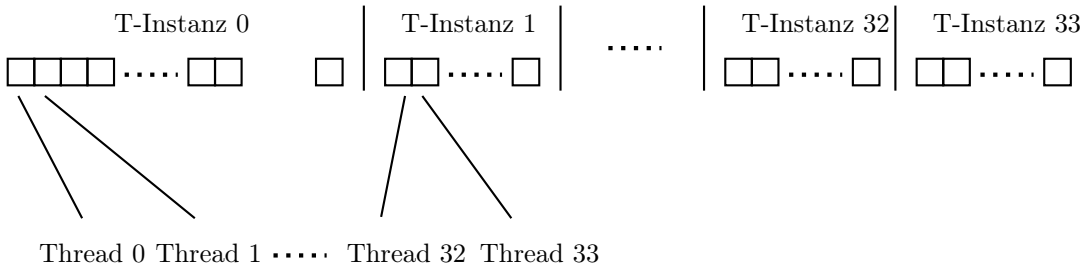


Abbildung 9.11: Template Instanz: Warpweise alle Knoten einer Templateinstanz

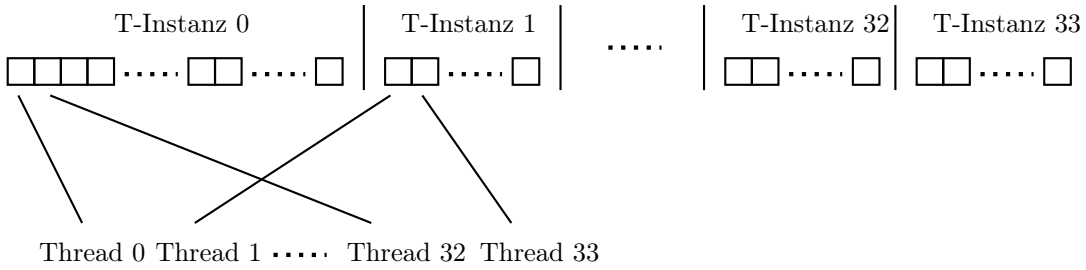


Abbildung 9.12: Template Instanz: Warpweise ein Knoten aller Templateinstanzen

1. Alle Threads kopieren alle Knoten einer Templateinstanz (Abb. 9.9).
2. Alle Threads kopieren einen Knoten des Inhalts aller Templateinstanzen (also etwa den i-ten Knoten jeder Templateinstanz, Abb. 9.10).
3. Alle Threads eines Warps kopieren alle Knoten des Inhalts einer Templateinstanz. Verschiedene Warps verarbeiten verschiedene Templateinstanzen (Abb. 9.11).
4. Alle Threads eines Warps kopieren einen Knoten des Inhalts aller Templateinstanzen. Verschiedene Warps verarbeiten verschiedene Knoten (Abb. 9.12).

Die optimale Strategie ist wiederum von der konkreten Anwendung abhängig. Da die Anzahl der Templateinstanzen wieder von der Anzahl der Knoten der Kontextmenge abhängt, entspricht der intuitive Ansatz hier den obigen Möglichkeiten 2 oder 4. Allerdings muss nicht für jeden Knoten das gleiche Template instanziiert werden, demnach variiert die Menge der Knoten der Templateinhalte der verschiedenen Instanziierungen stark, was zu abweichenden Ausführungspfaden führt. Auf

der anderen Seite ist auch Möglichkeit 1 unter Umständen ungeeignet, da ein Templateinhalt häufig nicht aus genügend vielen Knoten besteht, um alle Threads einzusetzen.

Als Kompromiss wurde daher die Möglichkeit 3 implementiert.

9.5 Funktionsumfang

Bisher wurde die generelle Vorgehensweise erläutert, um einen parallelen XSLT-Prozess auf der GPU auszuführen. Bevor die Leistungsfähigkeit des parallelen XSLT-Prozessors anhand der Ergebnisse verschiedener Tests diskutiert wird, wird an dieser Stelle kurz der Funktionsumfang der prototypischen Implementierung bezüglich der in Abschnitt 2.4 erwähnten wichtigen XSL-Anweisungen und die Erweiterung um Attribut-Wert-Vergleiche genannt.

Der prototypische XSLT-Prozessor unterstützt die Möglichkeit eines einfachen Attribut-Wert-Vergleichs der Form `[@attr='value']` im Prädikat eines XPath-Ausdrucks; die in Abschnitt 8 vorgestellte Implementierung wurde also um eine Funktion erweitert, da dieser Ausdruck häufig in XSLT-Stylesheets genutzt wird. Dazu werden die in Abschnitt 8 besprochenen Algorithmen minimal erweitert, indem einfach ein Thread, der in der bisherigen Variante einen Knoten für die Ausgabekontextmenge markiert hätte, zuvor noch den Attribut-Wert-Vergleich durchführt. Dazu müssen natürlich die Attributwerte auf der GPU vorhanden sein. Diese werden jedoch nur für diese Anwendung genutzt; jede andere Verarbeitung der Attributewerte, beispielsweise durch `value-of`-Anweisungen, wird wie in Abschnitt 9.1 besprochen, nicht durch die GPU durchgeführt.

Für den prototypischen XSLT-Prozessor wurden keine *Modes* [85, Abschnitt 5.7] implementiert. Sowohl in der Definition des Templates als auch in der `apply-templates`-Anweisung werden demnach `mode`-Attribute nicht ausgewertet.

Die `apply-templates`-Anweisung ist die wichtigste Anweisung in XSLT. Ihre Bedeutung bezüglich der Instanziierung wurde in dem hier verwendeten Prozessmodell gegenüber dem im Standard verwendeten Prozessmodell verändert, das Ergebnis ist jedoch äquivalent. Die Auswertung des `select`-Attributs entspricht im Rahmen des eingeschränktem XPath-Funktionsumfangs dem XSLT-Standard. Der XSLT-Standard sieht für die `apply-templates`-Anweisung auch eine optionale Sortierung der ausgewählten Knotenmenge sowie die optionale Wahl eines *mode* vor. Diese Funktionen wurden nicht implementiert.

Wie auch für die `apply-templates`-Anweisung wurde für die `for-each`-Anweisung die Auswertung des `select`-Attributs entsprechend des XPath-Sprachumfangs implementiert. Ebenso wurde die optionale Sortierung nicht implementiert.

Die Anweisung `value-of` ist ein XSL-Anweisung, die String-Repräsentationen von Ergebnissen von XPath-Ausdrücken liefert. In der Praxis jedoch wird häufig nur eine sehr kleine Teilmenge des XPath-Sprachumfangs genutzt. Deshalb wurde in der prototypischen Implementierung analog zum Template Matching nur die *self*-, die *child*- und die *attribute*-Achse implementiert. Anders als die anderen XSLT-Anweisungen wird diese Anweisung erst ganz am Ende, wenn keine anderen XSL-Anweisungen mehr gefunden werden, ausgewertet. Da der XPath-Ausdruck sehr einfach ist, wird dabei wie beim Template Matching jeder Ausdruck von je einem Thread ausgewertet [42, Abschnitt 12.2.5].

Die Anweisungen `copy` und `copy-of` sind entsprechend des Standards implementiert.

9.6 Evaluierung

In diesem Abschnitt sollen nun die Leistungsfähigkeit und einige Charakteristika des hier vorgestellten prototypischen XSLT-Prozessors demonstriert werden.

Wie bereits festgestellt, werden die im Abschnitt 8 vorgestellten Algorithmen hier im Kontext des GPU-basierten XSLT-Prozessors genutzt. Dazu wurden die XPath-Algorithmen, anders als im vorangegangenen Test, nicht mehr in Xalan eingebettet, sondern als Modul in das CUDA-OS implementiert. Die Komplexität der XPath-Algorithmen ändert sich dadurch natürlich nicht, so dass hier diejenigen XPath-Ausdrücke von vornherein nicht betrachtet wurden, bei denen schon die Ergebnisse aus der Evaluierung der XPath-Ausdrücke das Ergebnis der XSLT-Evaluierung vorgeben. Beispielsweise wurden in Tests, in denen die *following*-Achse verwendet wurde, quasi beliebige Leistungsgewinne im XSLT-Prozess erzielt (vgl. Abbildungen 8.9, 8.7). Da diese Gewinne aber nur etwas über die Leistung des XPath-Algorithmus, nicht jedoch über die des XSLT-Prozessors sagen, werden diese Tests hier nicht weiter diskutiert.

Das verwendete Testsystem ist das in Abschnitt 4 eingeführte. Es wurde in den hier diskutierten Tests nur die GTX580-GPU verwendet.

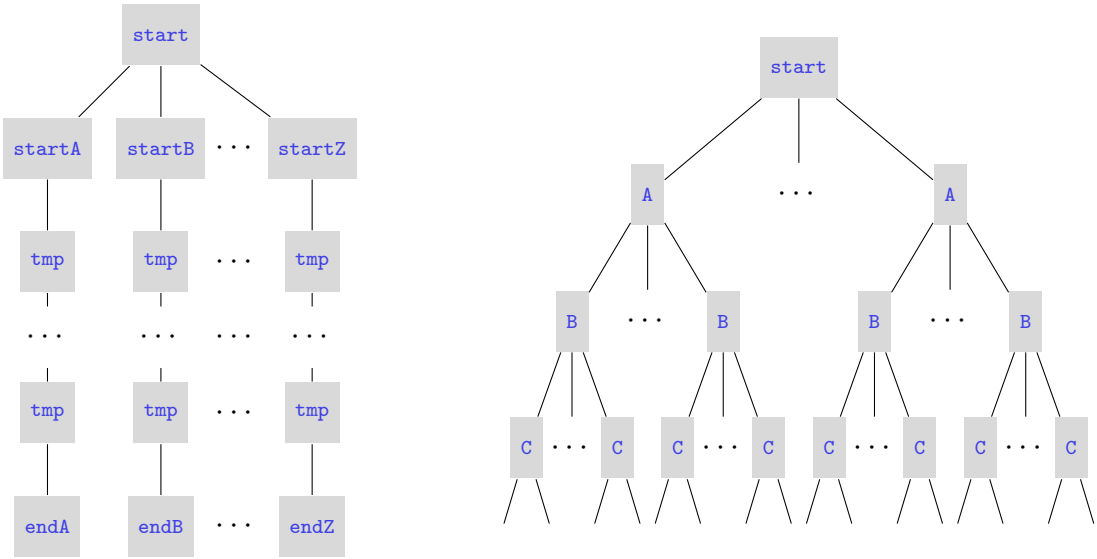


Abbildung 9.13: Evaluierung XSLT: Test-XML-Dokumente

9.6.1 Ungünstigster Fall

Im ersten Test wurde der schlechteste Fall für den hier entwickelten XSLT-Prozessor untersucht. Dazu wird ein Dokument verwendet, wie es in Abbildung 9.13 (Links) schematisch dargestellt ist: Die Wurzel hat die Kinder <startA> bis <startZ>. Unter jedem dieser Kindknoten befindet sich ein Pfad variabler Länge bis zu den entsprechenden Elementen <endA> bis <endZ>.

Dieses Dokument wurde mit einem Stylesheet wie in Quellcode 9.3 transformiert. Zur Vereinfachung wird die Funktion dieses Templates im Folgenden in der Sprechweise des rekursiven Prozessmodells beschrieben.

Das Stylesheet verfügt über ein Template, welches zu allen Knoten <tmp> passt. Die Instanziierung dieses Templates bewirkt nur, dass die Rekursion genau auf das eine Kind des <tmp> Knotens angewendet wird. Das Kind ist dann entweder wiederum ein <tmp> Element oder aber das Ende des Pfades ist erreicht. Für das Ende eines jeden Pfades ist ein Template vorgesehen, mit welchem rekursiv der erste <tmp> Knoten des folgenden Pfades verarbeitet wird.

Auf diese Weise wird, in einfachen Worten ausgedrückt, nacheinander für jeden Knoten des Eingabedokuments einmal ein Template instanziiert. Die aktuelle Knotenliste hat immer die Größe 1. Der Ergebnisbaum ist also eine Baum der Höhe *Pfadlänge* × *Anzahl Pfade* des Eingabedokuments. Natürlich verursacht die

```
1 <xsl:stylesheet version="1.0" xmlns:xsl="...">
2   <xsl:template match="tmp" >
3     <xsl:apply-templates select="child::*" />
4   </xsl:template>
5
6   <xsl:template match="/">
7     <xsl:apply-templates select="/descendant::startA/child::tmp" />
8   </xsl:template>
9
10  <xsl:template match="endA" >
11    <xsl:apply-templates select="/descendant::startB/child::tmp" />
12  </xsl:template>
13  ...
14  <xsl:template match="endY" >
15    <xsl:apply-templates select="/descendant::startZ/child::tmp" />
16  </xsl:template>
17 </xsl:stylesheet>
```

Quellcode 9.3: Test 1 XSLT-Stylesheet

Rekursion in Prozessoren, die nach dem rekursiven Modell arbeiten, zusätzlichen Aufwand (anders als in unserem Modell). Für die hier entwickelte parallele Implementierung ist die Situation jedoch noch ungünstiger, da

1. bei der parallelen Auswertung des einfachen XPath-Ausdrucks `child::tmp` kein Leistungsgewinn erzielt wird,

2. im Pattern Matching nur ein einziger Knoten behandelt wird, da das Ergebnis von `child::tmp` bzw. `/descendant::startA/child::tmp` usw. immer höchstens einelementig ist und

3. auch bei der Instanziierung immer nur genau ein Knoten der aktuellen Knotenliste behandelt wird und zudem die Instanziierung (das heißt das Kopieren des Inhalts des Templates in den Ergebnisbaum) ebenfalls nur aus einer Operation besteht, da die Templates nur aus einem Templateknoten bestehen.

Zusätzlich wurde in diesem Test eine variable Menge an Dummy-Templates in das Stylesheet eingefügt, die zwar auf keinen der Knoten im Dokument passen, jedoch immer beim Template Matching geprüft werden müssen.

Das Ergebnis des Tests ist in Abbildung 9.14 zu sehen. Die x-Achse zeigt die Anzahl der Elemente pro Pfad von 1 bis 250, die y-Achse stellt die Anzahl der zu-

sätzlich verwendeten Dummy-Templates dar. Die z-Achse zeigt den Berechnungsdurchsatz, das heißt die Anzahl der verarbeiteten Transformationen pro Sekunde. Die Werte für den Xalan-Prozessor wurden mit nur einem Kern der CPU im Testsystem gemessen, hier aber entsprechend der vier Kerne vervierfacht dargestellt. Das wichtigste Ergebnis dieses Tests ist, dass unsere Implementierung für alle Parameter erheblich langsamer ist als die Xalan-Implementierung. Weiterhin kann festgestellt werden, dass die Anzahl der Dummy-Templates wie erwartet einen ungünstigen Einfluss auf die GPU-Implementierung hat, da im hier implementierten Ansatz wie dargelegt immer ein Template für alle Knoten (im Test nur ein einziger) der aktuellen Knotenliste geprüft wird. Dies führt zu einer vollständigen Serialisierung des Prozesses.

Die Tests wurden auch für den libXSLT-Prozessor und den Saxon-Prozessor

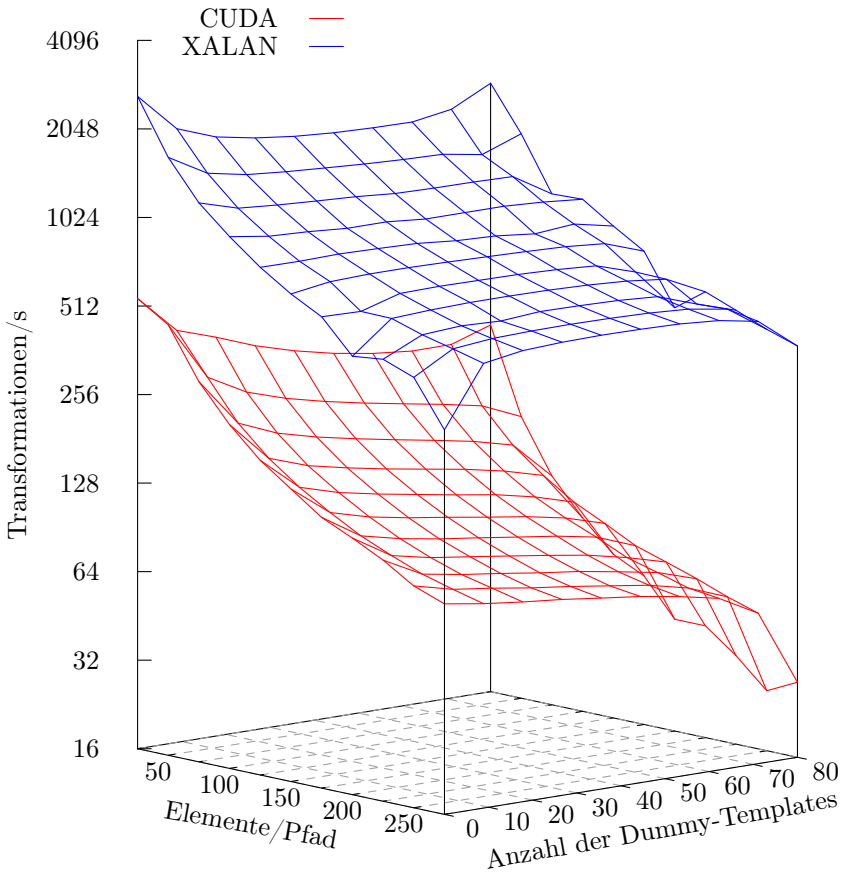


Abbildung 9.14: Evaluierung XSLT: Ungünstiger Fall

```
1 <xsl:stylesheet version="1.0" xmlns:xsl="...">
2   <xsl:template match="/">
3     <xsl:apply-templates select="/child::A" />
4   </xsl:template>
5
6   <xsl:template match="A" >
7     <xsl:apply-templates select="/descendant::*/child::B" />
8   </xsl:template>
9
10  <xsl:template match="B" >
11    <xsl:apply-templates select="/descendant::*/child::C" />
12  </xsl:template>
13  ...
14 </xsl:stylesheet>
```

Quellcode 9.4: Test 2 XSLT-Stylesheet

durchgeführt, die Ergebnisse sind denen des Xalan-Prozessors sehr ähnlich. Wegen der Übersichtlichkeit wurde auf die Darstellung der Ergebnisse des libXSLT-Prozessors und des Saxon-Prozessors verzichtet.

Es sei hier angemerkt, dass dieser für unseren Prozessor ungünstigste Fall wieder recht ungewöhnlich und auch für die anderen Prozessoren ungünstig ist: Die hier verwendete Variante mit 26 Pfaden einer Länge bis zu 250 ist eigentlich komplizierter als nötig, denn es hätte auch ein einzelner Pfad der entsprechenden Länge verwendet werden können. Jedoch ist es in diesem Fall wieder mit dem libXSLT-Prozessor unmöglich und mit dem Xalan-Prozessor sehr zeitaufwendig das Dokument zu parsen. Darüber hinaus mussten im Quellcode des libXSLT-Prozessors erst Parameter konfiguriert werden, damit eine genügend große Rekursionstiefe des XSLT-Prozesses ermöglicht wird.

9.6.2 Günstige Fälle

Nun werden günstigere Fälle betrachtet. Dazu werden Eingabedokumente wie in Abbildung 9.13 (Rechts) verwendet. Diese Dokumente sind vollständig balancierte Bäume, das heißt, jeder Knoten hat die gleiche Anzahl Kinder wie seine Geschwisterknoten und alle Knoten bis auf die Blätter haben die gleiche Anzahl Kindknoten. Alle Knoten auf einer Ebene des Baums haben den gleichen Namen.

Auf dieses Dokument wenden wir ein Stylesheet wie in Quellcode 9.4 an. Für einen Knoten in der aktuellen Knotenliste wird in diesem Stylesheet immer ein Template instanziiert, bei dem gerade alle Knoten der nächsten Ebene des Einga-

bebaums (nicht nur die Nachfahren) in die aktuelle Knotenliste eingefügt werden.

Diese Stylesheets können nach *Leveln* variiert werden: Ein Stylesheet vom Level 1 besitzt dann nur das Template, welches zum *root*-Knoten passt und Elemente <A> auswählt. Das Template für die Elemente <A> ist dann leer, es werden keine Knoten <B> verarbeitet. Ein Stylesheet vom Level 2 dagegen erzeugt im Template, welches zu <A> passt, entsprechend Quellcode 9.4 eine neue Knotenliste mit Knoten mit Namen <B>. In diesem Fall werden keine Elemente <C> mehr aufgerufen usw.

Zunächst wurde im Test ein Eingabedokument der Höhe 10 verwendet, wobei jedes Element genau zwei Kinder hat. Das bedeutet, dass in jeder Ebene des Baums gerade doppelt so viele Elemente wie in der darüberliegenden Ebene existieren. In

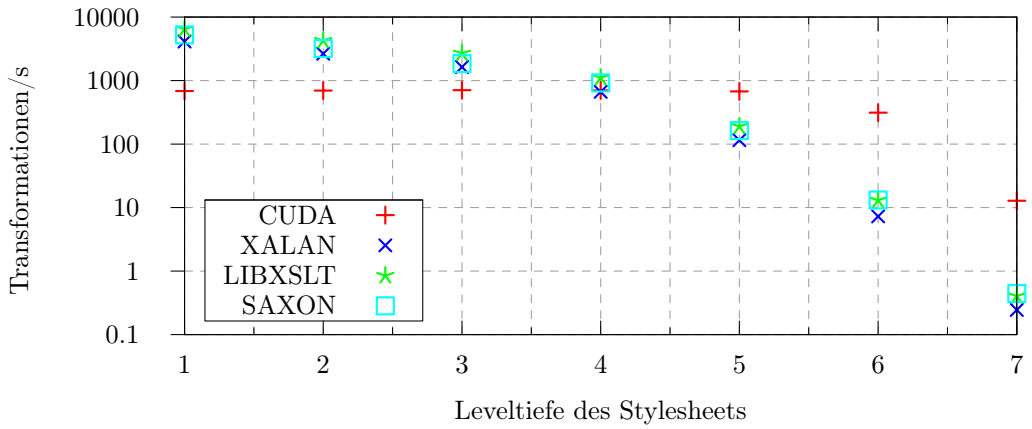


Abbildung 9.15: Evaluierung XSLT: Günstiger Fall (1)

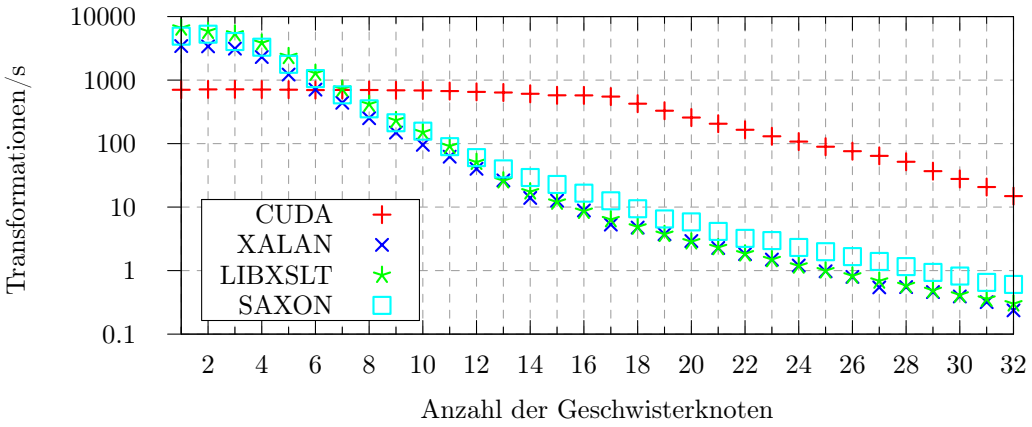


Abbildung 9.16: Evaluierung XSLT: Günstiger Fall (2)

dem Test variiert wurde das Level des Stylesheets.

Die Abbildung 9.15 zeigt die Ergebnisse. Die x-Achse stellt das Level des verwendeten Stylesheets dar, die y-Achse die Anzahl an verarbeiteten Transformationen pro Sekunde. Die CPU-Prozessoren wurden nur auf einem Kern der verwendeten CPU des Testsystems ausgeführt.

Deutlich ist zu sehen, wie die Leistung der CPU-XSLT-Prozessoren einbricht und ab Level 5 die GPU-Implementierung schneller arbeitet. Das liegt daran, dass ab Level 5 genügend große neue aktuelle Knotenlisten in den Templates erzeugt werden, so dass das Pattern Matching sehr gut parallelisiert werden kann. Natürlich ist in diesem Test auch der XPath-Ausdruck, mit dem die neuen Knotenlisten erzeugt werden, günstiger als im vorangegangenen Test, da er ebenfalls besser parallelisierbar ist.

Im nächsten Test wird nicht das Stylesheet variiert (es wird immer ein Level 3-Stylesheet verwendet), sondern das Eingabedokument. Das Eingabedokument entspricht wieder dem Baum aus Abbildung 9.13 (Rechts). Allerdings wurde diesmal ein Dokument mit fester Höhe fünf und variabler Anzahl an Geschwisterknoten generiert.

In der Abbildung 9.16 sind die Ergebnisse zu sehen. Die y-Achse gibt wieder die Anzahl der Transformationen pro Sekunde an, die mit dem jeweiligen Ansatz erreicht wurde. Die x-Achse stellt die Anzahl der Geschwisterknoten im Eingabedokument dar. Die CPU-Prozessoren wurden wieder nur auf einem Kern der verwendeten CPU des Testsystems ausgeführt.

Die Abbildung zeigt deutlich, dass die parallele Implementierung viel weniger Leistung verliert mit steigender Anzahl Geschwisterknoten als die sequentiellen Varianten. Das liegt daran, dass anfangs der größere Rechenaufwand durch mehr Geschwisterknoten einfach durch bessere Ausnutzung der Parallelität sowohl in der XPath-Auswertung wie auch beim Template Matching und teilweise auch bei der Template Instanziierung kompensiert wird. Schon ab einer Anzahl von 10 Geschwisterknoten ist der parallele XSLT-Prozessor schneller als die anderen Prozessoren.

9.6.3 XMark-Tests

Zuletzt wurden wie auch schon bei der Evaluierung der XPath-Algorithmen Tests mit Dokumenten durchgeführt, die mit dem XMark-Werkzeug generiert wurden.

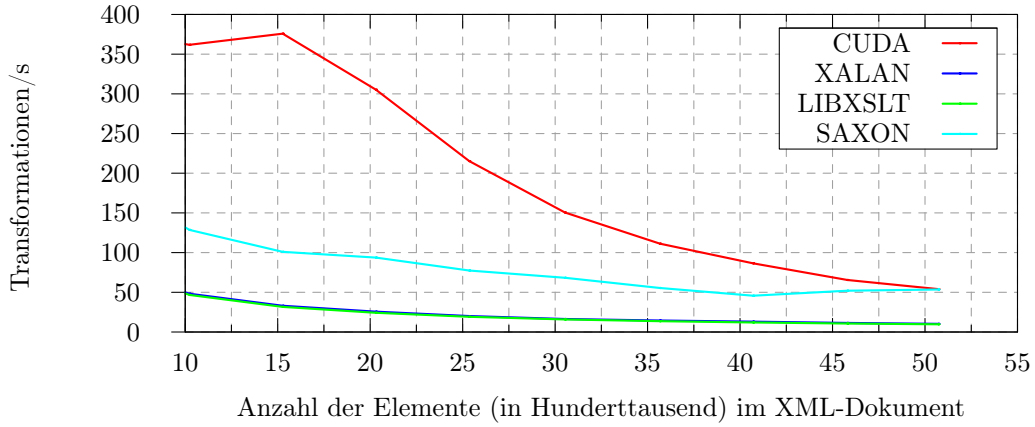


Abbildung 9.17: Evaluierung XSLT: XMark „filter.xsl“

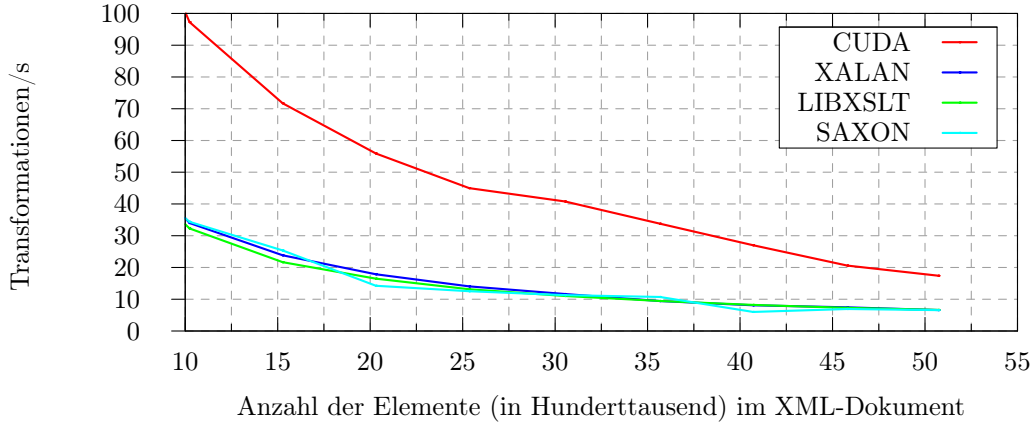


Abbildung 9.18: Evaluierung XSLT: XMark „transform.xsl“

Diese Dokument entsprechen eher den zuvor untersuchten günstigen Fällen, denn die generierten XML-Bäume haben im Allgemeinen eine geringe Höhe und eine große Anzahl Geschwisterknoten. Variiert wurde in den Tests die Größe des Eingabedokuments etwa zwischen 12 MB und 112 MB. Es wurden zwei vergleichsweise einfache Stylesheets getestet:

1. Das Stylesheet „filter.xsl“ filtert und kopiert bestimmte Knoten aus dem Eingabedokument (Anhang E).

2. Das Stylesheet „transform.xsl“ transformiert ein Eingabedokument in eine HTML-Darstellung (Anhang F).

Die Ergebnisse der Tests sind in den Abbildungen 9.17 und 9.18 zu sehen. Die x-Achse stellt die Anzahl der Knoten im getesteten Eingabedokument dar, repräsentiert also etwa die Größe des Dokuments. Die y-Achse stellt wiederum die Anzahl an verarbeiteten Transformationen pro Sekunde dar. Die Ergebnisse der Transformationen sind in diesen Tests recht groß. Da die Ergebnisse jeder Transformation gespeichert wurden, ist die Leistung des Speichers oder der Festplatte für das Ergebnis dieses Tests nicht unerheblich. Deswegen wurden auch die Tests für libXSLT und Xalan diesmal mittels paralleler CPU-Threads unter Auslastung aller CPU-Kerne durchgeführt, da nicht mehr angenommen werden darf, dass die Leistung der CPU-Varianten mit der Anzahl der verwendeten Kerne linear steigt.

Für alle getesteten Dokumentgrößen liefert der parallele XSLT-Prozessor mehr Leistung als die CPU-Prozessoren.

9.7 Zusammenfassung XSLT

In diesem Abschnitt wurde der Ansatz für einen parallelen XSLT-Prozessor für GPUs vorgestellt. Als Alternative zum im Standard definierten rekursiven Prozessmodell wurde ein iteratives Prozessmodell beschrieben, welches besser für die Architektur von GPUs geeignet ist.

Es wurde diskutiert, welche Teile des XSLT-Prozesses parallelisiert werden können. Für die drei wichtigsten Verarbeitungsschritte *Selection*, *Pattern Matching* und *Lazy Instantiation* wurden Strategien zur Parallelisierung aufgezeigt.

Die prototypische Implementierung des XSLT-Prozessors wurde mit einigen konstruierten XSLT-Stylesheets getestet, bei denen die Laufzeit im Wesentlichen vom XSLT-Prozess und nicht von der XPath-Auswertung abhängt. Für speziell konstruierte, in der Praxis nicht verwendete XML-Dokumente, bei denen die Verarbeitung nicht oder nur schwer zu parallelisieren ist, ist der verwendete parallele XSLT-Prozessor erwartungsgemäß langsamer als sequentielle Prozessoren für CPUs. Für Dokumente jedoch, die in ihrer Struktur eher den in der Praxis verwendeten Dokumenten entsprechen, kann mit dem parallelen XSLT-Prozessor ein Leistungsgewinn erzielt werden.

Dieses positive Ergebnisse wurde in Tests bestätigt, in denen nicht speziell konstruierte, sondern mit dem XMark-Werkzeug generierte Dokumente getestet wurden.

Teil IV

Zusammenfassung & Ausblick

KAPITEL 10

Zusammenfassung

In der vorliegenden Arbeit wurde die Möglichkeit untersucht, XML-Dokumente gewinnbringend auf parallelen SIMD-ähnlichen Architekturen wie GPUs zu verarbeiten. Das Ziel war es, als *proof-of-concept* eine konkrete XML-Anwendung, XSLT, für die Verarbeitung durch GPUs der Firma NVIDIA anzupassen. Das Konzept für den GPU-basierten parallelen XSLT-Prozessor wurde in Abschnitt 4 vorgestellt. Die vorliegende Arbeit erläutert die vier wichtigsten dabei verwendeten Algorithmen bzw. Ansätze, die speziell für diese Arbeit entwickelt wurden:

- Es wurde ein schneller paralleler Sortieralgorithmus für GPUs entwickelt (Abschnitt 5), der vergleichsbasiert und *in-place* arbeitet. In den Tests wurde gezeigt, dass mit diesem Algorithmus erstmals ein Algorithmus für GPUs existiert, der die genannten günstigen Eigenschaften mit einer sehr hohen Leistung vereint.

  Der Algorithmus wird in der vorliegenden Arbeit verwendet, um die zur XML-Verarbeitung genutzten Datenstrukturen zu erzeugen.

- Es wurde ein System entwickelt (Abschnitt 6), durch welches es möglich wird, eine GPU gleichzeitig und unabhängig voneinander unterschiedliche Aufgaben berechnen zu lassen, wobei zur Berechnung jeder Aufgabe nur ein Teil der Ressourcen der GPU genutzt wird. Darüber hinaus kann die GPU durch unterschiedliche CPU-Prozesse und Threads gleichzeitig genutzt werden. Die Tests zeigen, dass mit diesem System ohne Leistungsverlust unterschiedliche Aufgaben gleichzeitig bearbeitet werden können. Die unterschiedlichen

Aufgaben müssen dafür nicht synchron gestartet werden. Die Tests zeigen weiter, dass wie erwartet in ungünstigen Fällen ein etwas kleinerer Durchsatz an verarbeiteten Aufgaben gegenüber einer sequentiellen Verarbeitung mittels der GPU erreicht wird. In günstigen Fällen dagegen gibt es keinen Leistungsverlust oder sogar Leistungsgewinne.

Das System wird in dieser Arbeit eingesetzt, um die GPU als asynchronen XSLT-Prozessor nutzen zu können.

- Es wurden Algorithmen zur parallelen Auswertung von XPath-Location Paths ohne Prädikate entwickelt (Abschnitt 8). Die Korrektheit dieser Algorithmen wurde durch die Angabe einer neuen Semantik zur Auswertung von Location Paths, welche äquivalent zu anderen bekannten Ansätzen ist, gezeigt. Es wurden die Ergebnisse von Tests vorgestellt, in denen die hier entwickelten parallelen Algorithmen gegen die Verarbeitungsgeschwindigkeit anderer bekannter CPU-XPath- bzw. XSLT-Prozessoren verglichen wurden. Es lassen sich ungünstige Fälle konstruieren, in denen die sequentiellen Ansätze schneller als die hier verwendete parallele Variante sind. Diese Fälle sind jedoch in der Praxis nicht üblich, und in den anderen Fällen kann mit den parallelen Algorithmen durchaus ein Leistungsgewinn erzielt werden.

  Die parallelen XPath-Algorithmen sind integraler Bestandteil der parallelen XSLT-Verarbeitung.

- Es wurden Ansätze zur Parallelisierung eines XSLT-Prozessors vorgestellt (Abschnitt 9). Dazu wurde zum einen ein gut zur Parallelverarbeitung geeignetes iteratives Prozessmodell entwickelt und zum anderen diskutiert, wie die drei wichtigsten Bestandteile des XSLT-Prozesses unter den gegebenen Umständen parallelisiert werden können. Es wurden die Ergebnisse von Tests präsentiert, in denen die Leistungsfähigkeit des vorgestellten Prozessors möglichst unabhängig von der Leistungsfähigkeit der parallelen XPath-Verarbeitung ermittelt wurde. Es zeigt sich wiederum, dass für ungewöhnliche XML-Dokumente und XSLT-Stylesheets, die auch für andere CPU-basierte XSLT-Prozessoren nicht unproblematisch sind, die parallele Variante langsamer arbeitet als die sequentiellen CPU-Varianten. Für üblichere Fälle jedoch kann ein Leistungsgewinn gegenüber CPU-Varianten erzielt werden.

KAPITEL 11

Offene Fragen & weitere Arbeiten

Während der Entwicklung der in dieser Arbeit vorgestellten Ansätze und Verfahren wurden viele offene Fragen und ungelöste Probleme erkennbar, die im Rahmen dieser Arbeit nicht behandelt werden konnten. Neben den offensichtlichen Arbeiten wie etwa der Erweiterung des Sprachumfangs oder dem Vergleich gegen weitere XSLT-Prozessoren sind die folgenden offenen Arbeiten besonders aufgefallen:

- Die hier vorgestellten Ansätze unterscheiden sich in der Komplexität von anderer Ansätzen. Offen ist daher die Frage, ob die hier entwickelten Ansätze nicht auch in einer sequentiellen CPU-Variante Leistungsgewinne bringen könnten.
- In weiteren Arbeiten könnte eine schnelle Heuristik erarbeitet werden, welche einen XPath- oder XSLT- Algorithmus hinsichtlich seiner Leistungfähigkeit bezüglich einer konkreten Anwendung (etwa einer konkreten Transformation eines bestimmten Dokuments mit einem bekannten Stylesheet), bereits vor der Verarbeitung klassifiziert. So könnten schnelle hybride Ansätze implementiert werden.
- Weiterhin wäre ein wichtiger Untersuchungsgegenstand die formale Untersuchung der in der Realität eingesetzten XML-Dokumente hinsichtlich der für die genannte zu entwickelnde Heuristik wichtigen Eigenschaften.

KAPITEL 12

Fazit

Die Verarbeitung von XML-Dokumenten im Allgemeinen und die parallele Verarbeitung von XML-Dokumenten im Speziellen ist ein komplexes Problem. In der hier untersuchten Anwendung etwa, der Transformation von XML-Dokumenten mit XSLT, wird die Leistungsfähigkeit der Verarbeitung durch die Struktur des XML-Dokuments selbst, die Struktur des Stylesheets, die verwendeten XPath-Ausdrücke und natürlich durch die verwendeten Algorithmen bestimmt. Für jeden der in dieser Arbeit getesteten XPath- oder XSLT-Prozessoren können ungünstige Kombinationen von Eingabedokument, Stylesheet oder XPath-Ausdruck gefunden werden, selbstverständlich auch für den neuen parallelen Prozessor.

Für Eingaben jedoch, die der in der Praxis „üblichen“ Anwendung von XPath und XSLT entsprechen, können die hier entwickelten GPU-basierten Ansätze zur XPath- und XSLT-Verarbeitung einen Leistungsgewinn gegenüber CPU-Algorithmen bedeuten, selbst wenn mehrere CPU-Kerne genutzt werden. Zum Teil ergibt sich dieser Leistungsgewinn natürlich daraus, dass diese Ansätze die große parallele Rechenleistung von GPUs nutzen können. In einigen Fällen ergibt sich der Leistungsgewinn aber auch aus der besseren Komplexität der verwendeten XPath-Algorithmen, etwa durch die vorgestellte Implementierung der Knotentests mittels der mit dem neuen Sortieralgorithmus angelegten Indizes.

Für die Tests wurde der parallele XPath-Prozessor in dieser Arbeit für den Nutzer transparent in einen bestehenden CPU-XSLT-Prozessor eingebettet; der parallele XSLT-Prozessor wurde als asynchroner Koprozessor für die CPU implementiert. In beiden Fällen schließt die Verwendung der parallelen GPU-basierten

Variante die Verwendung herkömmlicher Prozessoren also nicht aus, so dass mit hybriden Verfahren beinahe immer ein Leistungsgewinn erzielt werden kann.

Daraus folgt als Ergebnis dieser Arbeit, dass GPUs für die Verarbeitung komplexer XML-Anwendungen gewinnbringend eingesetzt werden können.

Literaturverzeichnis

[1] ARB fragment program. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.

[2] ARB vertex program. http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt.

[3] Ati Stream. http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx.

[4] CG. http://developer.nvidia.com/cg-toolkit.

[5] CUDA. http://www.nvidia.de/object/cuda_what_is_de.html.

[6] DirectX. http://msdn.microsoft.com/en-us/directx.

[7] GLSL. http://www.opengl.org/documentation/glsl.

[8] HLSL. http://msdn.microsoft.com/en-us/library/bb509635(v=VS.85).aspx.

[9] libxml2. http://www.xmlsoft.org/.

[10] OpenCL. http://www.khronos.org/opencl.

[11] OpenGL. http://www.khronos.org/opengl.

[12] OpenStreetMap (OSM). http://www.openstreetmap.org/.

[13] XMark – An XML Benchmark Project. http://www.xml-benchmark.org/.

[14] Inc. 3dfx Interactive. SST-1(a.k.a. Voodoo Graphics) High Performance Graphics Engine 3D for game acceleration, rev. 1.61, 1999.

[15] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, New York, NY, USA, 2009. ACM.

[16] Kenneth E. Batcher. Sorting Networks and their Applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, volume 32 of *AFIPS '68 (Spring)*, pages 307–314, New York, NY, USA, 1967. ACM.

[17] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18:216–228, April 1989.

[18] Rajesh Bordawekar, Lipyeow Lim, Anastasios Kementsietsidis, and Bryant Wei-Lun Kok. Statistics-based parallelization of XPath queries in shared memory systems. In *EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology*, pages 159–170, New York, NY, USA, 2010. ACM.

[19] Rajesh Bordawekar, Lipyeow Lim, and Oded Shmueli. Parallelization of XPath queries using multi-core processors: challenges and experiences. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 180–191, New York, NY, USA, 2009. ACM.

[20] Jonas Bötel. A CUDA-based XPath Implementation. Diplomarbeit, Christian-Albrechts-Universität, Kiel, 2011.

[21] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Ian Buck and Tim Purcell. *GPU Gems*, chapter 37. A Toolkit for Computation on GPUs. Addison-Wesley, Reading, Massachusetts, USA, 2004.

[23] Daniel Cederman and Philippas Tsigas. A Practical Quicksort Algorithm for Graphics Processors. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Rongxin Chen and Husheng Liao. Paraparse: A parallel method for xml parsing. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 81 –85, may 2011.

[25] Intel Corporation. Microprocessor Quick References Guide. `http://www.intel.com/pressroom/kits/quickreffam.htm`.

[26] CWI. MonetDB. Technical report, CWI, February 2005. `http://monetdb.cwi.nl`.

[27] Des. Data encryption standard. In *In FIPS PUB 46, Federal Information Processing Standards Publication*, pages 46–2, 1977.

[28] Fadi El-Hassan and Dan Ionescu. SCBXP: An efficient hardware-based XML parsing technique. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pages 45–50, April 2009.

[29] Fadi El-Hassan and Dan Ionescu. A hardware architecture of an XML/XPath broker for content-based publish/subscribe systems. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 138–143, December 2010.

[30] Jianhua Feng, Le Liu, Guoliang Li, Jianhui Li, and Yuanhao Sun. An efficient parallel pathstack algorithm for processing xml twig queries on multi-core systems. In *DASFAA (1)*, pages 277–291, 2010.

[31] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[32] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, dec. 1966.

[33] International Organization for Standardization. ISO 8879:1986 Standard Generalized Markup Language, 1986. `http://www.iso.org`.

[34] International Organization for Standardization. ISO/IEC 19757-3:2006 Information technology – Document Schema Definition Language (DSDL) – Part 3: Rule-based validation – Schematron, 2006. `http://www.iso.org`.

[35] International Organization for Standardization. ISO/IEC 26300:2006 Information technology – Open Document Format for Office Applications (OpenDocument), 2006. `http://www.iso.org`.

[36] International Organization for Standardization. ISO 19136:2007 Geographic information – Geography Markup Language (GML), 2007. `http://www.iso.org`.

[37] International Organization for Standardization. ISO/IEC 29500-1,-2,-3,-4:2008 Information technology – Information technology – Document description and processing languages – Office Open XML File Formats, 2008. `http://www.iso.org`.

[38] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30:444–491, June 2005.

[39] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, 2005.

[40] Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 109–120, New York, NY, USA, 2002. ACM.

[41] Mark Harris, Shubhabrata Sengupta, and John D. Owens. *GPU Gems 3*, chapter 39. Parallel Prefix (Scan) with CUDA. Addison-Wesley, Reading, Massachusetts, USA, 2008.

[42] Christian Hoffmann. A CUDA-based XSLT Implementation. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, 2011.

[43] IBM. WebSphere DataPower XML Accelerator XA35. `http://www-01.ibm.com/software/integration/datapower/xa35/`.

[44] Intel. Intel SOA Expressway. `http://www.intel.com/cd/software/products/asmo-na/eng/373233.htm`.

[45] Vincent Jordan. XML query processing using GPU. `http://www.kde.cs.tsukuba.ac.jp/~vjordan/docs/`.

[46] Vincent Jordan. XML query processing using GPGPU. Master's thesis, University of Tsukuba & Université de technologie de Belfort-Montbéliard, 2010.

[47] Vincent Jordan. XML query processing using GPGPU. PhD Research Proposal, University of Tsukuba, February 2011.

[48] Michael Kay. SAXON. `http://saxon.sourceforge.net`.

[49] Stephan Kepser. A proof of the turing-completeness of xslt and xquery. In *In Technical report SFB 441, Eberhard Karls Universitat Tubingen*, 2004.

[50] Peter Kipfer, Mark Segal, and Rüdiger Westermann. UberFlow: a GPU-based particle engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, pages 115–122, New York, NY, USA, 2004. ACM.

[51] D. Knuth. *The Art of computer programming*, volume 3 (Sorting and Searching). Addison-Wesley, 1973.

[52] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Gpu sample sort. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, april 2010.

[53] LSI Corp. XML: Tarari content processors, 2007. `http://www.lsi.com/networking_home/networking_products/tarari_content_processors/xml/`.

[54] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to XML parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 223–230, Washington, DC, USA, 2006. IEEE Computer Society.

[55] Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Vassilis J. Tsotras, and Walid A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research*, January 2009.

[56] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[57] R. Moussalli, M. Salloum, W. Najjar, and V.J. Tsotras. Massively parallel xml twig filtering using dynamic programming on fpgas. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 948 –959, april 2011.

[58] Roger Moussalli, Robert Halstead, Mariam Salloum, Walid A. Najjar, and Vassilis J. Tsotras. Efficient xml path filtering using gpus. In *Proceedings of the The Second International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures (ADMS'11).* VLDB, 2011.

[59] Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis Tsotras. Accelerating XML query matching through custom stack generation on FPGAs. In Yale Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin / Heidelberg, 2010.

[60] NVIDIA. Graphics Processing Unit (GPU). `http://www.nvidia.com/object/gpu.html`.

[61] NVIDIA. NVIDIA CUDA Best Practices Guide. `http://developer.nvidia.com/object/gpucomputing.html`.

[62] NVIDIA. NVIDIA CUDA Programming Guide. `http://developer.nvidia.com/cuda-downloads`.

[63] Ruhsan Onder and Zeki Bayram. Xslt version 2.0 is turing-complete: A purely transformation based proof. In Oscar Ibarra and Hsu-Chun Yen, editors, *Implementation and Application of Automata*, volume 4094 of

Lecture Notes in Computer Science, pages 275–276. Springer Berlin / Heidelberg, 2006. 10.1007/11812128_26.

[64] Yinfei Pan, Wei Lu, Ying Zhang, and Kenneth Chiu. A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 351–362, Washington, DC, USA, 2007. IEEE Computer Society.

[65] Yinfei Pan, Ying Zhang, Kenneth Chiu, and Wei Lu. Parallel XML Parsing Using Meta-DFAs. In *Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 237–244, Washington, DC, USA, 2007. IEEE Computer Society.

[66] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort. *Concurrency and Computation: Practice and Experience*, pages 681–693, 2011.

[67] Hagen Peters, Martin Koper, and Norbert Luttenberger. Efficiently Using a CUDA-enabled GPU as Shared Resource. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1122–1127, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[68] Hagen Peters and Ole Schulz-Hildebrandt. *GPU Gems 4*, volume 2, chapter Comparison-based in-place sorting with CUDA. Morgan Kaufmann, Maryland Heights, MO, USA, 2011.

[69] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with CUDA based on bitonic sort. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, pages 403–410, Berlin, Heidelberg, September 2009. Springer-Verlag.

[70] Gnome Project. libxslt. `http://xmlsoft.org/`.

[71] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics

hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 41–50, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[72] R. Rivest. The MD5 Message-Digest Algorithm, RFC 1321, 1992.

[73] Flavio Rizzolo. DescribeX: A framework for exploring and querying XML web collections. *CoRR*, abs/0807.2972, July 2008.

[74] Robin Cover. XML Applications and Initiatives. `http://xml.coverpages.org/xmlApplications.html`.

[75] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, May 2009. IEEE Computer Society.

[76] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[77] Ole Schulz-Hildebrandt. Vergleichsbasierte GPU-Sortieralgorithmen: Analyse und Verbesserungen. Diplomarbeit, Christian-Albrechts-Universität, Kiel, 2011.

[78] Bhavik Shah, Praveen R. Rao, Bongki Moon, and Mohan Rajagopalan. A data parallel algorithm for xml dom parsing. In *Proceedings of the 6th International XML Database Symposium on Database and XML Technologies*, XSym '09, pages 75–90, Berlin, Heidelberg, 2009. Springer-Verlag.

[79] Yuanhao Sun, Tianyou Li, Qi Zhang, Jia Yang, and Shih-Wei Liao. Parallel xml transformations on multi-core processors. In *ICEBE*, pages 701–708, 2007.

[80] The Apache Software Foundation. Xalan-C++. Available online at `http://xml.apache.org/xalan-c/`.

[81] The Apache Software Foundation. Xerxes-C++. Available online at `http://xml.apache.org/xerces-c/`.

[82] W3C. Document Object Model (DOM) Level 1 Specification, 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[83] W3C. HTML 4.01 Specification, 1999. `http://www.w3.org/TR/html4/`.

[84] W3C. XML Path Language (XPath) Version 1.0, 1999. `http://www.w3.org/TR/xpath/`.

[85] W3C. XSL Transformations (XSLT) Version 1.0, 1999. `http://www.w3.org/TR/xslt`.

[86] W3C. Document Object Model (DOM) Level 2 Core Specification, 2000. `http://www.w3.org/TR/DOM-Level-2-Core/`.

[87] W3C. Web Services Description Language (WSDL) 1.1, 2001. `http://www.w3.org/TR/wsdl`.

[88] W3C. XHTML 1.0 The Extensible HyperText Markup Language, 2002. `http://www.w3.org/TR/xhtml1`.

[89] W3C. Document Object Model (DOM) Level 3 Core Specification, 2004. `http://www.w3.org/TR/DOM-Level-3-Core/`.

[90] W3C. XML Information Set, 2004. `http://www.w3.org/TR/xml-infoset/`.

[91] W3C. XML Schema Part 0: Primer Second Edition, 2004. `http://www.w3.org/TR/xmlschema-0/`.

[92] W3C. XSL Transformations (XSLT) Version 2.0, 2007. `http://www.w3.org/TR/xslt`.

[93] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. `http://www.w3.org/TR/REC-xml/`.

[94] W3C. Scalable Vector Graphics (SVG) 1.1 Specification, 2009. `http://www.w3.org/TR/SVG/single-page.html`.

[95] W3C. W3C Glossary Dictionary, 2010.
`http://www.w3.org/2003/glossary/`.

[96] W3C. XML Path Language (XPath) Version 2.0, 2010.
`http://www.w3.org/TR/xpath/`.

[97] W3C. XQuery 1.0: An XML query language, 2010.
`http://www.w3.org/TR/xquery/`.

[98] Xiaochun Ye, Dongrui Fan, Wei Lin, and Nan Ienne Paolo Yuan. High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[99] Yin Ye, Zhihui Du, and David A. Bader. GPUMemSort: A High Performance Graphic Co-processors Sorting Algorithm for Large Scale In-Memory Data. In *Annual International Conference on Advances in Distributed and Parallel Computing (ADPC 2010)*, November 2010.

[100] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 425–436, New York, NY, USA, 2001. ACM.

Teil V

Anhang

ANHANG A

DOM-node-Interface

```
interface Node {
  // NodeType
  const unsigned short      ELEMENT_NODE       = 1;
  const unsigned short      ATTRIBUTE_NODE     = 2;
  const unsigned short      TEXT_NODE          = 3;
  const unsigned short      CDATA_SECTION_NODE = 4;
  const unsigned short      ENTITY_REFERENCE_NODE = 5;
  const unsigned short      ENTITY_NODE        = 6;
  const unsigned short      PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short      COMMENT_NODE       = 8;
  const unsigned short      DOCUMENT_NODE      = 9;
  const unsigned short      DOCUMENT_TYPE_NODE = 10;
  const unsigned short      DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short      NOTATION_NODE      = 12;

  readonly attribute  DOMString           nodeName;
           attribute  DOMString           nodeValue; // raises(DOMException) on setting
                                                     // raises(DOMException) on retrieval
  readonly attribute  unsigned short      nodeType;
  readonly attribute  Node                parentNode;
  readonly attribute  NodeList            childNodes;
  readonly attribute  Node                firstChild;
  readonly attribute  Node                lastChild;
  readonly attribute  Node                previousSibling;
  readonly attribute  Node                nextSibling;
  readonly attribute  NamedNodeMap        attributes;
  readonly attribute  Document            ownerDocument;
  Node                insertBefore(in Node newChild, in Node refChild) raises(DOMException);
  Node                replaceChild(in Node newChild, in Node oldChild) raises(DOMException);
  Node                removeChild(in Node oldChild) raises(DOMException);
  Node                appendChild(in Node newChild) raises(DOMException);
  boolean             hasChildNodes();
  Node                cloneNode(in boolean deep);
};
```

ANHANG B

XPath-Achsen

2.2 Axes

The following axes are available:

- the `child` axis contains the children of the context node
- the `descendant` axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes
- the `parent` axis contains the parent of the context node, if there is one
- the `ancestor` axis contains the ancestors of the context node; the ancestors of the context node consist of the parent of context node and the parent's parent and so on; thus, the ancestor axis will always include the root node, unless the context node is the root node
- the `following-sibling` axis contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the `following-sibling` axis is empty
- the `preceding-sibling` axis contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the `preceding-sibling` axis is empty
- the `following` axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes
- the `preceding` axis contains all nodes in the same document as the context

node that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes

- the `attribute` axis contains the attributes of the context node; the axis will be empty unless the context node is an element
- the `namespace` axis contains the namespace nodes of the context node; the axis will be empty unless the context node is an element
- the `self` axis contains just the context node itself
- the `descendant-or-self` axis contains the context node and the descendants of the context node
- the `ancestor-or-self` axis contains the context node and the ancestors of the context node; thus, the ancestor axis will always include the root node

NOTE: The `ancestor`, `descendant`, `following`, `preceding` and `self` axes partition a document (ignoring attribute and namespace nodes): they do not overlap and together they contain all the nodes in the document.

ANHANG C

XPath-Location Path

Here are some examples of location paths using the unabbreviated syntax:

- `child::para` selects the `para` element children of the context node
- `child::*` selects all element children of the context node
- `child::text()` selects all text node children of the context node
- `child::node()` selects all the children of the context node, whatever their node type
- `attribute::name` selects the `name` attribute of the context node
- `attribute::*` selects all the attributes of the context node
- `descendant::para` selects the `para` element descendants of the context node
- `ancestor::div` selects all `div` ancestors of the context node
- `ancestor-or-self::div` selects the `div` ancestors of the context node and, if the context node is a `div` element, the context node as well
- `descendant-or-self::para` selects the `para` element descendants of the context node and, if the context node is a `para` element, the context node as well
- `self::para` selects the context node if it is a `para` element, and otherwise selects nothing
- `child::chapter/descendant::para` selects the `para` element descendants of the `chapter` element children of the context node
- `child::*/child::para` selects all `para` grandchildren of the context node
- `/` selects the document root (which is always the parent of the document element)

- `/descendant::para` selects all the `para` elements in the same document as the context node
- `/descendant::olist/child::item` selects all the `item` elements that have an `olist` parent and that are in the same document as the context node
- `child::para[position()=1]` selects the first `para` child of the context node
- `child::para[position()=last()]` selects the last `para` child of the context node
- `child::para[position()=last()-1]` selects the last but one `para` child of the context node
- `child::para[position()>1]` selects all the `para` children of the context node other than the first `para` child of the context node
- `following-sibling::chapter[position()=1]` selects the next `chapter` sibling of the context node
- `preceding-sibling::chapter[position()=1]` selects the previous `chapter` sibling of the context node
- `/descendant::figure[position()=42]` selects the forty-second `figure` element in the document
- `/child::doc/child::chapter[position()=5]/child::section[position()=2]` selects the second `section` of the fifth `chapter` of the `doc` document element
- `child::para[attribute::type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `child::para[attribute::type="warning"][position()=5]` selects the fifth `para` child of the context node that has a `type` attribute with value `warning`
- `child::para[position()=5][attribute::type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`
- `child::chapter[child::title="Introduction"]` selects the `chapter` children of the context node that have one or more `title` children with string-value equal to `Introduction`
- `child::chapter[child::title]` selects the `chapter` children of the context node that have one or more `title` children
- `child::*[self::chapter or self::appendix]` selects the `chapter` and `appendix` children of the context node
- `child::*[self::chapter or self::appendix][position()=last()]` selects the last `chapter` or `appendix` child of the context node

ANHANG D

XSLT-Prozessmodell

A list of source nodes is processed to create a result tree fragment. The result tree is constructed by processing a list containing just the root node. A list of source nodes is processed by appending the result tree structure created by processing each of the members of the list in order. A node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them; the chosen rule's template is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select an additional list of source nodes for processing. The process of matching, instantiation and selection is continued recursively until no new source nodes are selected for processing.

Implementations are free to process the source document in any way that produces the same result as if it were processed using this processing model.

ANHANG E

„filter.xsl“

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
      Transform">
3
4    <xsl:template match="/">
5       <items>
6          <xsl:copy-of select="//item/incategory[@category='category1']/
               parent::*"/>
7       </items>
8    </xsl:template>
9 </xsl:stylesheet>
```

ANHANG F

„transform.xsl“

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
      Transform">
3    <xsl:template match="/">
4       <html>
5          <body>
6             <table>
7                <tr>
8                   <th>ID</th>
9                   <th>Name</th>
10                  <th>Bezahlung</th>
11                  <th>Ort</th>
12                  <th>Anzahl</th>
13                  <th>Kategorien</th>
14               </tr>
15               <xsl:apply-templates select="//item"/>
16            </table>
17         </body>
18      </html>
19   </xsl:template>
20   <xsl:template match="item">
21      <tr>
22         <td><xsl:value-of select="@id"/></td>
23         <td><xsl:value-of select="name"/></td>
24         <td><xsl:value-of select="payment"/></td>
25         <td><xsl:value-of select="location"/></td>
26         <td><xsl:value-of select="quantity"/></td>
27         <td><xsl:for-each select="incategory">
28            <xsl:value-of select="@category"/>,
29         </xsl:for-each></td>
30      </tr>
31   </xsl:template>
32 </xsl:stylesheet>
```