technische universität
dortmund

Masterarbeit

Effiziente String-Verarbeitung in Datenbankanfragen auf hochgradig paralleler Hardware

Florian Lüdiger
Juni 2019

Gutachter:
Prof. Dr. Jens Teubner
Henning Funke

Technische Universität Dortmund
Fakultät für Informatik
Datenbanken und Informationssysteme (LS-6)
http://dbis.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ein Großteil der Datensätze, die in realen Systemen zum Einsatz kommen enthalten eine große Vielfalt an String-Datensätzen mit verschiedensten Eigenschaften. Die Operationen, die auf diesen Datensätzen ausgeführt werden reichen von Gleichheitsüberprüfungen über das Enthaltensein eines Teilstrings bis zum Erfüllen eines komplexen, regulären Ausdrucks. In dieser Arbeit wird untersucht werden, wie sich die unterschiedlichen Operationen zur String-Verarbeitung verhalten, wenn diese auf einer hochgradig parallel arbeitenden Grafikkarte ausgeführt werden. Außerdem wird analysiert, ob ein Verfahren zur Steigerung der Auslastung der Grafikkarte eine signifikante Leistungssteigerung erzielen kann.

1.1 Motivation und Hintergrund

Die effiziente Berechnung unterschiedlicher Operatoren auf Zeichenketten ist essenziell für das Erreichen eines hohen Durchsatzes und für das Gewährleisten von maximaler Performanz. In diesem Kontext versprechen Grafikkarten durch ihre hochgradig parallele Architektur in der Theorie eine bestmögliche Leistung.

Der Aufbau von moderner, hoch paralleler Hardware wie einer Grafikkarte führt dazu, dass diese besonders effizient mit gleichmäßigen Daten arbeiten kann. Ein String-Datensatz ist dagegen typischerweise sehr heterogen durch die unterschiedlichen Längen der einzelnen Zeichenketten, wodurch das Verarbeiten auf einer Grafikkarte zunächst einige Schwierigkeiten birgt. Aus diesem Grund verwenden bisherige Ansätze zur Verarbeitung von Zeichenketten das Konzept der Dictionaries [9]. Dabei wird eine Tabelle mit allen Strings aufgebaut und zu diesen ein Schlüssel abgespeichert, welcher zusammen mit jedem String in den anderen Tabellen der Datenbank gespeichert wird. Somit können String-Operationen durch andere Operationen auf den Schlüsseln abgebildet werden, wodurch diese eine einheitliche Struktur und damit ein effizientes Ausführungsmuster auf Grafikkarten erhalten. Das Aufbauen und Verwalten des für diese Technik verwendeten Dictionaries erzeugt vor allem bei Daten, die sich häufig ändern, einen hohen Aufwand, wodurch die Leistungsfähigkeit des

Gesamtsystems sinkt. Um diesen Verwaltungsaufwand für eine zusätzliche Datenstruktur zu eliminieren, wäre es wünschenswert eine Lösung zu finden, die den Verwaltungsaufwand eliminiert und direkt auf den ursprünglichen String-Daten arbeitet.

Soll eine komplexere Operation auf den Daten ausgeführt werden, die mit regulären Ausdrücken arbeitet, ist die Verwendung eines Dictionaries nicht mehr möglich und spätestens an dieser Stelle muss auf eine andere Methode zurückgegriffen werden. Die Verwendung von regulären Ausdrücken eröffnet in vielen Anwendungsfällen verschiedenste Möglichkeiten, String-Daten effizient zu verarbeiten und die Leistungsfähigkeiten des Datenbanksystems voll auszunutzen. Reguläre Ausdrücke stellen ein mächtiges Werkzeug dar, welches den meisten Anwendern von Datenbankmanagementsystemen bekannt ist und vor allem relevant ist, da es das hoch effiziente Auswerten komplexer Muster erlaubt.

Die meisten aktuellen Datenbankmanagementsysteme bieten eine Unterstützung für reguläre Ausdrücke, da sie das effiziente Abgleichen komplexer Muster ermöglichen und dabei eine hohe Leistung garantieren. Gäbe es eine solche Funktion nicht, müsste eine entsprechende Selektion nach dem Ausführen des Anfrageplans durch das DBMS manuell im Anwendungsprogramm durchgeführt werden. An dieser Stelle entstehen zahlreiche Probleme, da der Optimierer die Selektion nicht an der optimalen Stelle im Anfrageplan platzieren kann, damit frühzeitig Tupel weg fallen, die nicht in das Ergebnis aufgenommen werden. Es entsteht also schon beim Datenbankserver eine erhöhte Last durch das unnötige Verarbeiten von Tupeln. Diese werden zusätzlich noch über das Netzwerk zur Anwendung übertragen, wodurch erneut eine erhöhte Netzlast entsteht. Die Anwendung wiederum muss anschließend mit einer potenziell geringeren Leistungsfähigkeit als der Datenbankserver die Selektion durchführen, wodurch an dieser Stelle wieder eine unnötig hohe Last entsteht. Sollten also Anwendungsfälle auftreten, bei denen eine Selektion durch reguläre Ausdrücke gewünscht ist, stellt die Unterstützung dieser Operation durch das Datenbankmanagementsystem einen massiven Vorteil dar. Auch für Datenbanksysteme, die auf Grafikkarten arbeiten sollte also sichergestellt sein, dass diese die maximal mögliche Performanz bei der Verarbeitung von regulären Ausdrücken erreichen.

1.2 Zielsetzung

In dieser Arbeit sollen verschiedene Operationen, die auf String-Daten arbeiten im Kontext von kompilierten Anfrageplänen mithilfe des Query Compilers DogQC untersucht werden. Ziel ist es dabei die Performanz der umgesetzten Operatoren zu analysieren und Flaschenhälse, die durch eine schlechte Auslastung der Grafikkarte entstehen, zu identifizieren. Diese Flaschenhälse sollen mithilfe des Lane Refill-Verfahrens eliminiert werden, wodurch eine Steigerung der Leistungsfähigkeit erreicht werden soll. Neben einfachen String-Operatoren sollen auch komplexere Techniken, welche reguläre Ausdrücke zur String-Verarbeitung verwenden, analysiert werden und untersucht werden, ob diese einen Laufzeitgewinn durch das

Lane Refill erreichen. Mithilfe dieser Betrachtungen sollen Rückschlüsse darüber gezogen werden, ob das Lane Refill, welches bereits bei der Durchführung von Join-Operationen erfolgreich eingesetzt wird, auch für die Verarbeitung von Zeichenketten einen Nutzen bringt.

1.3 Aufbau der Arbeit

Um verstehen zu können, warum bei der Verarbeitung von String-Daten auf Grafikkarten verschiedene Probleme auftreten können, werden zunächst der Grundaufbau und die wichtigsten Eigenschaften von Grafikprozessoren erklärt. Anschließend werden die bei DogQC zum Einsatz kommenden Verfahren zur Kompilierung von Anfrageplänen erläutert und die Rahmenbedingungen für die nachfolgenden Untersuchungen festgelegt. In Kapitel 4 wird der einfache String-Vergleich erarbeitet und auf verschiedene Probleme hingewiesen, die im nachfolgenden Kapitel durch den Einsatz des Lane Refill behoben werden. An dieser Stelle wird außerdem erklärt, wie das Lane Refill-Verfahren funktioniert und warum es durch eine höhere Auslastung der Grafikkarte eine bessere Performanz erreichen kann. Anschließend wird in Kapitel 6 erläutert, wie reguläre Ausdrücke mithilfe von endlichen Automaten ausgewertet werden können und ein geeigneter Ansatz für die Umsetzung im Query Compiler ausgewählt. Im folgenden Teil wird der parallele Musterabgleich mit regulären Ausdrücken umgesetzt und ähnlich wie beim einfachen String-Vergleich durch das Lane Refill erweitert. Vorbereitend auf die im letzten Teil der Arbeit durchgeführten Leistungstests der verschiedenen Algorithmen wird aufgezeigt, wie das Optimieren verschiedener Parameter bei der Ausführung von Algorithmen auf einer Grafikkarte einen Einfluss auf die Performanz des Systems nimmt und wie diese Optimierung durchgeführt werden kann. Schließlich folgen zwei Kapitel zur Evaluation des einfachen String-Vergleichs und des Musterabgleichs mit regulären Ausdrücken, in denen untersucht wird, wie sich die unterschiedlichen Algorithmen verhalten und ob durch die Verwendung des Lane Refill-Verfahrens ein Laufzeitgewinn erreicht werden kann. Abschließend wird in einem Fazit Stellung dazu genommen, ob die vorher beschriebene Zielsetzung erreicht werden konnte und ob das Lane Refill bei der Verarbeitung von Zeichenketten eine sinnvolle Anwendung findet.

Kapitel 2

Grundlagen der GPU-Programmierung

Um die in dieser Arbeit vorgestellten Herausforderungen bei der Verarbeitung von Zeichenketten(*Strings*) mit Grafikprozessoren, nachfolgend auch *GPUs* genannt, verstehen zu können, ist zunächst ein Verständnis der grundlegenden Eigenschaften aktueller Hardware nötig. Dabei beschränkt sich diese Untersuchung auf die Grafikkarten-Serie Maxwell von Nvidia, die hier besprochenen Prinzipien lassen sich allerdings auch auf die GPUs anderer Hersteller übertragen und finden dort ebenfalls Anwendung.

2.1 Grundaufbau einer Nvidia-Grafikkarte

Der Hauptprozessor eines Computers, auch *Central Processing Unit (CPU)* genannt, arbeitet eher sequenziell schwerwiegende Threads ab, wodurch individuelle Operationen schnell abgearbeitet werden können, ein hoher Durchsatz allerdings schwierig zu erreichen ist. Für die Verarbeitung großer Datenmengen wurden daher spezielle Co-Prozessoren in Form von Grafikkarten entwickelt, die hochgradig parallel arbeiten und somit einen massiven Durchsatz erreichen können. Die *Graphics Processing Unit (GPU)* bildet das Herzstück der Grafikkarte. Sie besteht aus einer hohen Anzahl an Kernen, die zwar individuell eine vergleichsweise geringe Leistung besitzen, allerdings aufgrund ihrer großen Zahl in datenparallelen Anwendungsfällen in Kombination mit einer hohen Speicherbandbreite eine hervorragende Performanz bieten.

Neben der GPU benötigt eine Grafikkarte noch weitere Peripherie, um effizient funktionieren zu können. Zur Speicherung der zu verarbeitenden Daten gibt es eigenständige Speichermodule, die unabhängig vom Hauptspeicher des Computers verwaltet werden. Für die Nvidia GTX 950, welche im Folgenden als Beispiel genutzt werden soll, beträgt die Größe dieses Speichers 2 GB. Über eine PCI-Express-Anbindung wird die Kommunikation

mit dem Hauptprozessor und die Übertragung der Daten zwischen den Speicherbereichen realisiert.

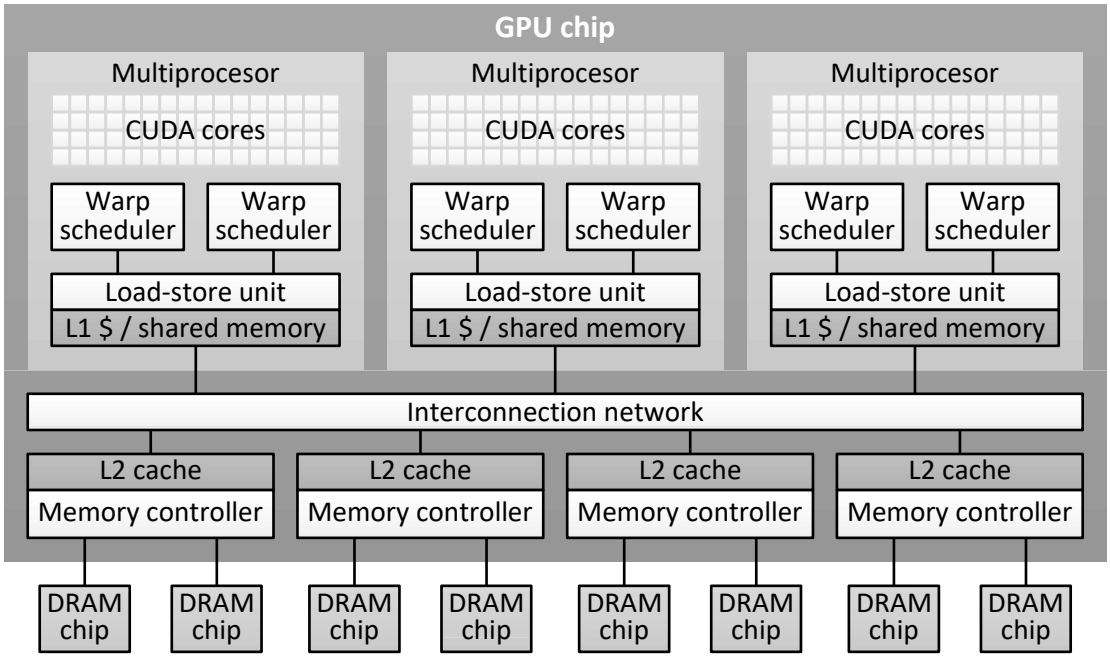

Abbildung 2.1: Architektur einer GPU [16]

Wie in Abbildung 2.1 dargestellt, lässt sich die GPU wiederum in kleinere Module, sogenannte *Streaming Multiprocessors (SM)*, unterteilen, welche jeweils eigenständige Recheneinheiten darstellen. Eine GTX 950 besitzt beispielsweise sechs dieser Streaming Multiprocessors, welche sich ebenfalls in kleinere Einheiten unterteilen lassen. Die SM bestehen aus vier unabhängigen Blöcken von Rechenkernen, welche jeweils 32 skalare Recheneinheiten, auch *CUDA-Kerne* genannt, beinhalten. Jeder dieser Blöcke besitzt einen eigenen Scheduler und etwas Unterstützungselektronik, sodass diese logisch gesehen ebenfalls unabhängig voneinander arbeiten können [12]. Bei sechs SMs mit jeweils vier Blöcken und 32 Recheneinheiten pro Block besitzt die GTX 950 also 768 Kerne, welche über eine Programmierschnittstelle angesprochen werden können.

2.2 Scheduling auf GPUs

Um die hohe Anzahl von Kernen innerhalb einer GPU effizient mit Arbeit versorgen zu können, ist es wegen des großen Overheads nicht praktikabel, ein individuelles Scheduling für die einzelnen Recheneinheiten durchzuführen. Aus diesem Grund werden die Threads eines Programms in sogenannte *Warps* zusammengefasst, welche damit die kleinste Einheit für das Scheduling bilden. Ein Warp enthält dabei genau 32 Threads, die in diesem Kontext auch *Lanes* genannt werden. Mehrere Warps werden außerdem zu *Blöcken* zusammengefasst, welche schließlich als Ganzes an einzelne Streaming Multiprocessors zugewiesen

werden. Innerhalb eines SM können Warps ausgetauscht werden, um beispielsweise durch Speicherzugriffe entstehende Latenzen zu verstecken.

Über die Anzahl der Threads pro Block und die gesamte Anzahl der Blöcke ist die Konfiguration des sogenannten *Grids* definiert. Die Grid-Konfiguration nimmt starken Einfluss auf die Ausführungszeit der Software. Beispielsweise kann eine zu geringe Anzahl von Threads pro Block dazu führen, dass eventuell durch Speicherzugriffe entstehende Latenzen nicht mehr so gut versteckt werden können, da nicht genug Threads innerhalb eines SM vorhanden sind. Eine zu hohe Anzahl von Threads pro Block kann allerdings auch von Nachteil sein, da Hardwareressourcen wie die Speichergröße pro SM gegebenenfalls nicht mehr ausreichen und das Programm nicht mehr korrekt funktioniert. Das Finden der richtigen Parameter gestaltet sich als äußerst schwierig, da die verwendete Hardware ein komplexes Konstrukt mit vielen Faktoren bildet, die unterschiedlich und schwer vorhersagbar auf Änderungen des Grids reagieren [17].

Eine für die Programmierung von GPUs entscheidende Eigenschaft besteht darin, dass die Threads innerhalb eines Warps parallel ausgeführt werden. Ähnlich wie bei dem Prinzip *Single Instruction Multiple Data (SIMD)* [13], führen die Threads in einem Warp die Instruktionen synchron aus, sodass dieses Prinzip auch *Single Instruction Multiple Threads (SIMT)* genannt wird. Die Trennung in mehrere Threads bietet hierbei den Vorteil, dass eigene Register angesprochen werden können, an unterschiedlichen Stellen im Speicher gelesen werden kann und die Kontrollflüsse der Threads leicht divergieren können. Prozesse laufen außerdem zwar logisch parallel ab, allerdings muss dies nicht notwendigerweise physisch auch so sein, sodass in einigen Fällen eine höhere Leistung erzielt werden kann. Für die optimale Performanz einzelner Operationen sollte allerdings gewährleistet sein, dass die Threads größtenteils synchron ausgeführt werden.

Bei der Verwendung von Branching-Instruktionen kann es vorkommen, dass unterschiedliche Threads verschiedene Kontrollflüsse durchlaufen, was auch als *Divergenz* bezeichnet wird. Da allerdings alle Threads identische Instruktionen ausführen müssen, führt dies dazu, dass sämtliche Threads in einem Warp alle notwendigen Kontrollflüsse durchlaufen und dabei gegebenenfalls das Ergebnis verwerfen, wenn diese sich logisch gesehen in einem anderen Zweig befinden. Alle Threads, für die der aktuell bearbeitete Kontrollfluss nicht relevant ist, werden als inaktiv bezeichnet. Inaktive Threads warten somit lediglich darauf, dass die aktiven Threads die Arbeit innerhalb ihres Kontrollflusses abgeschlossen haben, sodass an dieser Stelle massiv Rechenleistung verschwendet wird. In dieser Problematik liegt der Grund dafür, dass die Verarbeitung von Strings auf Grafikkarten durch ihre variable Länge problematisch ist, da die auftretenden Kontrollflüsse divergieren.

2.3 Synchronisation von Threads

Der Compiler und die GPU selbst versuchen, innerhalb eines Warps die Anzahl der synchron ausgeführten Operationen zu maximieren, weil dadurch eine höhere Leistung erzielt wird [11]. Diese Synchronisation kann allerdings auch explizit durch den Entwickler erfolgen, indem er die dafür vorgesehenen Operationen der Entwicklungsschnittstelle verwendet. Das Verwenden solcher Operationen führt dazu, dass alle Threads an dieser Stelle aufeinander warten müssen.

Diese Methoden können außerdem dazu verwendet werden, Informationen über die anderen Threads zu erlangen und die Zusammenarbeit innerhalb der Warps effektiver zu gestalten. Ein Beispiel für eine solche Operation ist das Auswerten eines Prädikats für alle Threads und das anschließende Erstellen einer Bitmaske, die das Ergebnis der Auswertung für alle Threads enthält. Ein weiteres Beispiel ist das Generieren einer Maske für alle Threads, die in dem aktuellen Ausführungszweig aktiv sind. Schließlich können noch alle Threads ohne besondere Berechnung synchronisiert werden. Dies ist zum Beispiel nötig, wenn ein Thread aus dem Speicher lesen will, den andere Threads vorher beschreiben und sicherstellen will, dass die Daten fertig geschrieben wurden [8].

2.4 Shared Memory

Ein Datenaustausch zwischen Threads innerhalb eines Blocks kann über sogenannten *Shared Memory* geschehen. Dadurch können größere Mengen von Informationen ausgetauscht werden, als dies über die Synchronisations-Operationen effizient möglich wäre. Dieser Speicher ist um einige Größenordnungen schneller als der globale Speicher, da sich dieser direkt auf dem Chip der GPU befindet [5]. Die Speichergröße innerhalb eines Streaming Multiprocessors ist allerdings beschränkt, weshalb die Anzahl der Threads ebenfalls beschränkt ist, sofern sie eine große Menge Shared Memory benötigen.

2.5 Die CUDA-Programmierschnittstelle für C++

Für eine effiziente Entwicklung der hochgradig spezialisierten Grafikhardware stellt Nvidia die *CUDA*-Programmierschnittstelle bereit. Diese ermöglicht es, die GPU aus einer Hochsprache wie C++ heraus anzusprechen und durch verschiedene Hilfestellungen leicht ein funktionierendes Programm zu erstellen. Neben vordefinierten Schlüsselwörtern und Syntaxelementen bietet die Entwicklungsumgebung auch einen eigenen Compiler, welcher das erstellte Programm für den Einsatz auf der Grafikkarte optimiert. Für das Verständnis der Beispiele in dieser Arbeit sollen im Folgenden einige Grundkonzepte des Programmiermodells erläutert werden.

Das Hauptprogramm von CUDA-Programmen besteht aus Code für die CPU, welcher dafür zuständig ist, die Grafikkarte für ihre Aufgabe vorzubereiten und anschließend das Unterprogramm aufzurufen, welches auf der GPU ausgeführt werden soll. Ein solches Unterprogramm wird *Kernel* genannt und besteht im einfachsten Falle aus einer einfachen Funktion, welche durch das Schlüsselwort `__global__` gekennzeichnet wird. In diesem Kontext wird der GPU-Code üblicherweise *Device Code* und der CPU-Code *Host Code* genannt.

Auf die Schnittstellen zur Speicherverwaltung oder zur Festlegung der Grid-Konfiguration aus dem Host Code heraus soll hier nicht weiter eingegangen werden, da für die untersuchten Kriterien lediglich der Device Code interessante Aspekte bietet.

Einem Kernel können verschiedene Parameter wie Zeiger auf Speicherbereiche innerhalb des Grafikspeichers aus dem Hauptprogramm übergeben werden. Zum Durchlaufen eines solchen Speicherbereiches in einem sequenziellen Programm wäre es ausreichend, mit einem Index über das Feld zu iterieren und diesen nach jeder Iteration um eins zu erhöhen. Bei einer parallelen Architektur würden so allerdings sämtliche Threads über den gesamten Datensatz laufen, anstatt wie gewünscht den Datensatz auf die einzelnen Threads aufzuteilen. Zu diesem Zweck muss jeder Thread die Informationen darüber haben, welchen globalen Index er innerhalb des Grids hat, um mit dem entsprechenden Element aus dem Datensatz zu beginnen und wie viele Threads in dem Grid vorhanden sind, damit er den entsprechenden Abstand zu dem nächsten zu untersuchenden Element kennt. Innerhalb eines Kernels kann der Thread auf seinen Threadindex (`threadIdx.x`), die Anzahl der Threads in einem Block (`blockDim.x`), seinen Blockindex (`blockIdx.x`) und die Anzahl der Blöcke im Grid (`girdDim.x`) zugreifen. Der globale Index eines Threads berechnet sich somit aus `blockIdx.x * blockDim.x + threadIdx.x` und die Sprungweite ist definiert durch `blockDim.x * gridDim.x`. Die dafür zur Verfügung gestellten Variablen und eine beispielhafte Iteration über zwei Datensätze sind in Abbildung 2.2 dargestellt.

```
1  __global__
2  void add(int n, float *x, float *y)
3  {
4    int index = blockIdx.x * blockDim.x + threadIdx.x;
5    int stride = blockDim.x * gridDim.x;
6    for (int i = index; i < n; i += stride)
7      y[i] = x[i] + y[i];
8  }
```

Abbildung 2.2: Beispielhafter CUDA-Kernel zum Iterieren über zwei Datensätze [4]

Mit den vorgestellten Informationen zum Grundaufbau von Grafikprozessoren, der Verwaltung von Threads und der Funktionsweise der C++-Programmierschnittstelle werden die in den nachfolgenden Kapiteln vorgestellten Techniken leicht verständlich sein.

Kapitel 3

Kompilierte Anfragepipelines

In dieser Arbeit werden String-Vergleiche im Kontext des Query Compilers *DogQC* für GPUs untersucht. Dieser basiert auf dem Query Compiler *HorseQC* [2] und wurde für das erleichterte Testen von aktuellen Techniken vereinfacht. DogQC erstellt aus einem gegebenen Anfrageplan mehrere Anfragepipelines, die für die Ausführung auf GPUs optimiert sind. Als Grundlage für die späteren Codebeispiele werden hier die grundsätzliche Funktionsweise des Query Compilers erklärt und die Vorteile dieser Technik erläutert.

Klassische in-memory-Datenbanken wie *MonetDB* arbeiten die Operatoren innerhalb eines Anfrageplans nacheinander ab, was auch *Operator-At-A-Time* genannt wird [15]. Dabei wird für den gesamten Datensatz zunächst der erste Operator vollständig ausgeführt, bevor der gesamte Datensatz an den nächsten Operator weiter gegeben wird, bis schließlich der gesamte Anfrageplan abgearbeitet wurde. Der Nachteil dieser Strategie besteht in einer besonders hohen Lese- und Schreiblast für die Zwischenergebnisse der Operatoren, da diese nach jeder Operation im Speicher materialisiert werden müssen. Soll die Berechnung auf einer GPU erfolgen, kann es passieren, dass der begrenzte GPU-Speicher nicht ausreicht, um die Zwischenergebnisse zu speichern. Somit werden während der Berechnung Transfers in den Hauptspeicher des Systems notwendig, um neue Blöcke der Tabelle nachzuladen. Als Konsequenz entstehen massive Flaschenhälse durch die begrenzte Bandbreite.

Das Pipelining-Prinzip, welches bei dem vorgestellten Query Compiler zum Einsatz kommt, legt fest, dass der Anfrageplan in Pipelines aufgeteilt wird, die von den Tupeln immer vollständig durchlaufen werden, bevor das Ergebnis materialisiert wird. Dieses Vorgehen wird *Tuple-At-A-Time* genannt. Die Operatoren innerhalb des Anfrageplans aus Abbildung 3.1 werden zu zwei Pipelines zusammengefasst. In der linken Pipeline wird die dates-Relation gelesen, die Selektion ausgeführt und die Hashtabelle für den Join berechnet. Die rechte Pipeline fasst das Lesen der orders-Relation, die Selektion, die Probe-Operation der Hashtabelle und das Zählen der Ergebnisse zusammen. Technisch werden die Operationen innerhalb der Pipeline zu einem einzigen Operator verschmolzen, indem der Query Compiler für jede Pipeline einen eigenständigen Kernel generiert, welcher mit der CUDA-

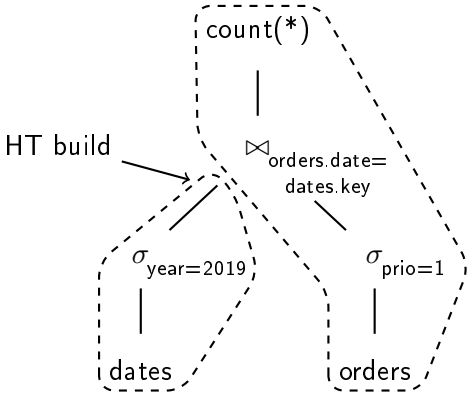

Abbildung 3.1: Beispielplan mit eingezeichneten Pipelines

Schnittstelle ausgeführt werden kann. Der Vorteil des Pipelinings besteht darin, dass die vollständigen Ergebnismengen nicht nach jedem Operator im Speicher materialisiert werden müssen, sondern die einzelnen Tupel stets in den GPU-Registern vorgehalten werden können, bis diese fertig verarbeitet wurden.

Klassische *Tuple-At-A-Time*-Ausführungsmuster wie das *Volcano Iterator Model* definieren Operatoren als eigenständige Bausteine mit festgelegten Schnittstellen, die von den Tupeln nacheinander durchlaufen werden [3]. Im Gegensatz dazu werden die Operatoren bei kompilierten Anfragepipelines zu einem Operator verschmolzen, wodurch Funktionsaufrufe entfallen und die Tupel in einem *tight loop* [10] verarbeitet werden.

In Abbildung 3.2 wird der Code vorgestellt, den der Query Compiler für die rechte Pipeline aus dem in Abbildung 3.1 dargestellten Anfrageplan generiert. Hier ist zu erkennen, dass die drei Operationen innerhalb eines Kernels zusammengefasst wurden, wodurch eine Pipeline entsteht. Um jedem Thread eine Menge von Tupeln zuweisen zu können, wird zunächst der globale Index des aktuellen Threads innerhalb des Grids berechnet, damit dieser als Schleifenindex `loop_var` verwendet werden kann. Anschließend wird über alle Elemente aus dem Datensatz iteriert, für die der aktuelle Thread zuständig ist. Die Variable `active` zeigt im Algorithmus an, ob der aktuelle Thread aktiv läuft oder nur darauf wartet, dass die anderen Threads aus seinem Warp ihre Berechnung abschließen.

In einem Schleifendurchlauf, bei dem das Element mit dem Index `loop_var` untersucht wird, wird zunächst die blau markierte Selektion ausgeführt. Ist diese fehlgeschlagen, da die Priorität der Bestellung nicht bei 1 liegt, wird der Thread deaktiviert und im weiteren Verlauf nicht mehr beachtet, bis er ein neues Tupel erhält. Hat sich das Tupel qualifiziert, folgt darauf der Hash Probe, welcher in grün dargestellt ist. Dabei wird das Tupel in der übergebenen Hashtabelle `hashtable_date_key` gesucht und dementsprechend wieder die `active`-Variable angepasst. Schließlich wird vom Query Compiler noch das Zählen der Ergebnisse umgesetzt, welches hier in rot hervorgehoben ist. Dabei wird mithilfe der Synchronisierungsoperation `__ballot_sync` die Anzahl der aktiven Lanes gezählt, welche

jeweils ein Element des Ergebnisses repräsentieren. Diese Anzahl wird daraufhin vom ersten Thread innerhalb des Warps auf das Ergebnis addiert.

Nach der Auswertung der gesamten Pipeline von Operationen für das untersuchte Tupel wird der Index um die vorher definierte Schrittweite erhöht, sodass im nächsten Schleifendurchlauf das nächste Tupel untersucht wird. Falls der neu gewählte Index hinter dem Ende der Daten liegt, hat der aktuelle Thread seine Arbeit vollständig abgeschlossen und er wird nicht mehr benötigt, sodass die `active`-Variable in Zeile 20 auf `false` gesetzt wird. Wird anschließend mithilfe der `__ballot_sync`-Methode festgestellt, dass sämtliche Lanes inaktiv sind, ist der Datensatz vollständig durchlaufen worden und die Berechnung kann abgeschlossen werden. Das parallele Abarbeiten eines Warps und das Synchronhalten aller Lanes wird *warp synchronous programming* genannt [8].

Durch die Untersuchung des hier vorgestellten Verfahrens der *Tuple-At-A-Time*-Verarbeitung wird klar, dass dieses einen großen Vorteil bei der effizienten Nutzung von schnellem Speicher gegenüber der *Operator-At-A-Time* Verarbeitung bietet. Zwischenergebnisse müssen nicht materialisiert werden, weshalb Tupel in Registern oder Caches vorgehalten werden können und die Operationen für das explizite Materialisieren entfallen.

Im Gegensatz zu der *Operator-At-A-Time*-Technik lässt sich das Pipelining allerdings nicht so einfach auf eine parallele Verarbeitung mit GPUs übertragen. Wird jedem Thread in einem Warp ein Tupel übergeben, kann es beispielsweise passieren, dass dieses bei einer Selektion aus der Ergebnisrelation heraus fällt, somit im weiteren Verlauf nicht weiter beachtet werden muss und die entsprechende Lane inaktiv wird. Dieses Problem wird in Kapitel 4.3 aufgegriffen und in Kapitel 5 wird ein Lösungsvorschlag dafür vorgestellt.

```
1  __global__
2  void joinProbePipeline(
3    int *orders_prio,                // priority attribute of orders table
4    int *orders_date,                // date attribute of orders table
5    unique_ht *hashtable_date_key,   // hashtable from other pipeline
6    int *number_of_matches) {        // return value
7
8    // global index of the current thread,
9    // used as the iterator in this case
10   unsigned loop_var = ((blockIdx.x * blockDim.x) + threadIdx.x);
11
12   // offset for the next element to be computed
13   unsigned step = (blockDim.x * gridDim.x);
14
15   bool active = true;
16   bool flush_pipeline = false;
17   while(!flush_pipeline) {
18
19     // element index must not be higher than number of tuples
20     active = loop_var < TUPLE_COUNT_ORDERS;
21
22     // break computation when every line is finished and therefore
       inactive
23     flush_pipeline = !__ballot_sync(ALL_LANES, active);
24
25     // selection
26     if (active)
27       active = orders_prio[loop_var] == 1;
28
29     // hash join probe
30     if (active)
31       active = hashProbeUnique(hashtable_date_key, HASHTABLE_SIZE,
32         hash(orders_date[loop_var]))
33
34     // count and write
35     numProj = __popc(__ballot_sync(ALL_LANES, active))
36     if (threadIdx.x % 32 == 0)
37       atomicAdd(number_of_matches, numProj);
38
39     loop_var += step;
40   }
41 }
```

Abbildung 3.2: Generierter Kernel für den Beispielplan

Kapitel 4

Einfacher, paralleler String-Vergleich

Um einige Techniken zur String-Verarbeitung in kompilierten Anfragepipelines auf Grafikkarten entwickeln zu können, wird zunächst ein Operator für den einfachen String-Vergleich für den Query Compiler erarbeitet. Ein Vergleich auf Gleichheit stellt dabei die einfachste, sinnvolle Variante einer String-Verarbeitung dar, wodurch der Einfluss vieler Eigenschaften von Strings auf die Ausführung entsprechender Operationen leicht untersucht werden kann.

Zunächst wird eine Umsetzung des einfachen String-Vergleichs mittels der CUDA-Schnittstelle ohne spezielle Optimierungen vorgestellt und für einen alternativen Workload für weitere Tests leicht angepasst werden. Schließlich wird das Potenzial der Lösung beurteilt und auf einen Nachteil der naiven Implementierung eingegangen.

4.1 Umsetzung des einfachen String-Vergleichs

Als Basis für die Untersuchungen wird der String-Vergleich-Operator für den Query Compiler zunächst naiv, also ohne tiefgehende Optimierungen umgesetzt. Mithilfe dieses Operators kann in einer Datenbankabfrage beispielsweise eine Selektion über eine Spalte mit String-Daten durchgeführt werden. Die Anforderung des Operators besteht somit darin, eine Liste von Zeichenketten mit einem vorher spezifizierten String zu vergleichen und zu entscheiden, ob diese identisch sind oder nicht.

Zur Durchführung dieser Operation wird jedem Thread der GPU eine Zeichenkette aus dem Datensatz zugewiesen. Zunächst wird überprüft, ob die Länge des Strings mit der des Suchstrings übereinstimmt, sodass der entsprechende Eintrag direkt verworfen werden kann. Sind die Längen identisch, werden beide Zeichenketten Zeichen für Zeichen durchlaufen und diese an jeder Stelle auf Gleichheit überprüft. Sobald eine Ungleichheit gefunden wurde, wird ein entsprechendes Flag gesetzt und die weiteren Zeichen müssen nicht mehr genauer betrachtet werden.

Sämtliche Threads innerhalb eines Warps werden entsprechend des Verarbeitungsmodells der GPU parallel abgearbeitet. Somit sind die Positionen, an denen die Strings verglichen werden, ebenfalls für alle Threads identisch. Sobald der Vergleichsstring im gesamten Warp vollständig durchlaufen wurde, wird das Zwischenergebnis geschrieben. Sollten alle Threads in dem Warp vorzeitig feststellen, dass keiner der Strings mit dem Suchstring übereinstimmt, wird die aktuelle Untersuchung vorzeitig abgebrochen.

Schließlich wird jedem Thread eine neue Zeichenkette aus dem Datensatz zugewiesen, sodass das Verfahren im weiteren Verlauf wiederholt wird. Sobald der gesamte Datensatz durchlaufen wurde, ist die Berechnung abgeschlossen.

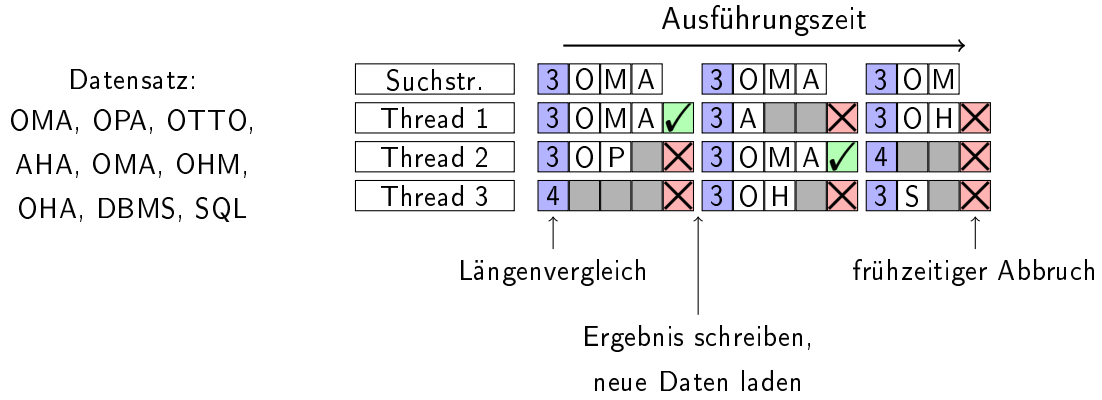

Abbildung 4.1: Funktionsweise des Algorithmus innerhalb eines Warps mit drei Threads

In Abbildung 4.1 ist der Ablauf des Algorithmus innerhalb eines Warps mit drei Threads dargestellt. Es ist erkennbar, an welchen Stellen die Lanes inaktiv werden, wann die Berechnung frühzeitig abgebrochen werden kann und an welchen Stellen das Ergebnis geschrieben wird und neue Daten aus dem Datensatz geholt werden.

```
1  /* execute previous operators in the pipeline */
2
3  data_length = char_offset[loop_var+1] - char_offset[loop_var] - 1;
4
5  // if string lengths are unequal, discard
6  if (active && data_length != search_length)
7    active = false;
8
9  int search_id = 0;
10
11 // iterate over strings completely or until they don't match anymore
12 while(active && search_id < search_length) {
13   int data_id = search_id + char_offset[loop_var];
14
15   // when strings don't match, inactivate the lane
16   if (active && data_content[data_id] != search_string[search_id])
17     active = false;
18
19   search_id++;
20 }
21
22 /* execute following operators in the pipeline */
```

Abbildung 4.2: Naive Implementierung einer Selektion von Strings

In Abbildung 4.2 ist die Implementierung des Operators dargestellt, welcher in einer kompilierten Anfragepipeline die Selektion eines String-Attributs durchführen kann. Dieser würde beispielsweise die rot markierte Selektion in Abbildung 3.2 ersetzen, falls das `prio`-Attribut in dem Beispiel aus Abbildung 3.1 Strings enthalten würde und alle Bestellungen mit der Priorität *"HIGH"* gesucht werden würden. Nach Abschluss der Berechnung ist der Wert der `active`-Variable genau dann `true`, wenn der String mit dem gesuchten String übereinstimmt, sodass im Anschluss mit weiteren Operationen fortgefahren werden kann.

Diese Implementierung erwartet, dass einige Daten vorher vom Hauptspeicher in den Speicherbereich der GPU kopiert wurden und dort zur Verfügung stehen. Der Datensatz, der mit dem Vergleichsstring abgeglichen werden soll, besteht aus einer Aneinanderreihung der entsprechenden Zeichenketten und ist in der Variable `data_content` gespeichert. Damit daraus die ursprünglichen Strings extrahiert werden können, gibt es das Feld `char_offset`, welches Informationen über die Indizes der Einzelstrings innerhalb des Datensatzes enthält. Ebenfalls muss ein Zeiger auf den Suchstring und dessen Länge in den entsprechenden Parametern `search_string` und `search_string_length` vorhanden sein. Um die Berechnung rechtzeitig vor Speicherüberschreitungen abbrechen zu können, wird schließlich noch die Variable `line_count` benötigt, welche die Anzahl der Zeichenketten im Datensatz beschreibt.

Im ersten Schritt wird in Zeile 6 für einen String überprüft, ob dessen Länge mit der des Suchstrings übereinstimmt und das entsprechende Tupel anderenfalls verworfen. Sind die Längen identisch, wird in der in Zeile 12 beginnenden Schleife über beide Zeichenketten iteriert, bis das Ende beider erreicht wurde, oder festgestellt wird, dass ein Zeichen aus dem Vergleichsstring nicht mit dem aus dem Suchstring übereinstimmt. An dieser Stelle fällt die parallele Struktur des Kernels besonders auf, da die Zeichen aller Strings parallel von den Threads durchlaufen werden und diese erst aufhören, wenn der letzte Thread den ihm zugewiesenen Datensatz vollständig durchlaufen hat. Ist die Schleife abgeschlossen oder vorzeitig abgebrochen worden, kann am Zustand der `active`-Variable abgelesen werden, ob die Strings übereinstimmen oder nicht.

4.2 Präfixtest als alternativer Workload

Als zusätzliche String-Operation, für die ein simpler Algorithmus existiert, wurde neben dem exakten String-Vergleich auch ein Präfixtest entwickelt. Dabei soll geprüft werden, ob die Strings aus einer Datenbanktabelle ein vordefiniertes Präfix besitzen.

Der Algorithmus arbeitet ähnlich wie das in Kapitel 4.1 vorgestellte Verfahren zum exakten String-Vergleich, mit einem einzigen Unterschied. Für jede Zeichenkette aus der Tabelle wird zunächst geprüft, ob diese mindestens genau so lang wie das gesuchte Präfix ist, anstatt zu prüfen, ob die Längen identisch sind. Danach verfährt der Algorithmus wie gehabt, sodass wieder beide Strings Zeichen für Zeichen verglichen werden, bis das Ende des Such-Präfixes erreicht ist. An der Implementierung in Abbildung 4.2 ändert sich dementsprechend fast nichts, es muss nur die Bedingung in Zeile 4 durch `active && data_length >= search_length` ersetzt werden.

Beim exakten String-Vergleich können viele Strings schon im ersten Schritt bei der Überprüfung der Längengleichheit ausgeschlossen werden. Je nach Anwendungsfall müssen dagegen beim Präfixtest viele Zeichenketten, deren Länge größer als die des Präfixes ist, länger durchlaufen werden. Durch diesen Umstand und durch die Tatsache, dass ein Präfixtest in realen Systemen ein häufig gefragter Anwendungsfall ist, ergibt diese Operation einen interessanten alternativen Workload.

4.3 Einschätzung der GPU-Auslastung

Das in diesem Kapitel vorgestellte, naive Verfahren zum einfachen String-Vergleich nutzt die Ressourcen der GPU nicht besonders effizient aus. In Abbildung 4.1 sind in grau die inaktiven Threads dargestellt. Diese sind inaktiv geworden, da erkannt wurde, dass sie nicht mit der gesuchten Zeichenkette übereinstimmen, da sie entweder eine unpassende Länge besitzen oder im Laufe des Vergleichs der einzelnen Zeichen ein Unterschied festgestellt wurde. Die untersuchten Strings aus der Tabelle können völlig unterschiedlich geartet sein,

weshalb es häufig vorkommt, dass einige Threads ihre Untersuchung bereits abgeschlossen oder vorzeitig unterbrochen haben, während andere Threads innerhalb des Warps noch lange weiter rechnen müssen. Aufgrund des Programmiermodells von Grafikkarten laufen die inaktiven Threads weiter synchron zu den aktiven Threads des Warps, wobei allerdings das Ergebnis verworfen wird und diese keine nutzbare Arbeit mehr verrichten.

Im schlimmsten Fall werden 31 Threads aus einem Warp Zeichenketten mit unpassender Länge zugewiesen und einem einzigen Thread eine mit dem Vergleichsstring übereinstimmende Zeichenkette zugeteilt. Somit stellen 31 Threads im ersten Schritt fest, dass die Länge des Strings nicht mit der des Vergleichsstrings übereinstimmt und werden daher inaktiv. Der Thread mit dem passenden String muss allerdings noch über jedes Zeichen iterieren, bevor sich der gesamte Warp neue Strings holen kann. Es kann also sein, dass für die Dauer der Iteration über den Suchstring die GPU nur zu $\frac{1}{32}$ ausgelastet ist, weshalb die Performanz dieser Lösung verbesserungswürdig ist.

Inaktiven Threads sollte folglich dynamisch neue Arbeit zugewiesen werden, sobald diese inaktiv geworden sind. Da in dem Algorithmus lediglich zwei Zeichen an zwei Positionen verglichen werden und diese Position für jeden Thread unterschiedlich sein kann, könnte sich ein Thread, sobald er inaktiv geworden ist, eine neue Zeichenkette holen und damit weiter arbeiten. Mit diesem Vorgehen könnte zwar eine hohe Auslastung erreicht werden, da die Threads niemals wirklich inaktiv werden können, allerdings funktioniert das nicht mit dem in Kapitel 3 vorgestellten Pipelining-Modell. Dies liegt daran, dass gegebenenfalls noch beliebig viele andere Operationen in der Pipeline vor dem String-Vergleich durchgeführt werden müssen, bevor eine neue Zeichenkette an den Thread übergeben wird.

Im folgenden Kapitel wird eine weitere Technik vorgestellt, mit der die Auslastung der GPU verbessert werden kann. Das Verfahren wird auch im Umfeld einer Anfragepipeline funktionieren und somit für den praktischen Einsatz geeignet sein.

Kapitel 5

Verbesserung des einfachen String-Vergleichs

Zur Verbesserung der Auslastung der GPU bei der Berechnung des einfachen String-Vergleichs wird ein Verfahren vorgestellt, das auch in kompilierten Anfragepipelines eine optimierte Laufzeit verspricht. Dieses stellt durch Verwendung eines Puffers sicher, dass zur Laufzeit die Auslastung eines Warps immer über einem bestimmten Grenzwert liegt. Das Prinzip wurde im Kontext von kompilierten Anfragepipelines mit SIMD als *consume everything* vorgestellt [7] und wird im Kontext dieser Arbeit auf Grafikkarten angewendet. Im Folgenden wird das Verfahren als *Lane Refill* bezeichnet, da es die inaktiven Lanes in einem Warp dynamisch wieder auffüllt.

5.1 Funktionsweise des String-Vergleichs mit Lane Refill

Zur Verbesserung des einfachen String-Vergleichs kann das naive Verfahren durch das Lane Refill erweitert werden, indem ein Puffer eingeführt wird, in dem teilweise abgearbeitete Tupel zwischengespeichert werden können. Bei der naiven Umsetzung gibt es drei Stellen, an denen Lanes innerhalb eines Warps inaktiv werden können. Zum einen passiert dies, wenn beim Längenvergleich zu Beginn eine unpassende Länge festgestellt wird und zum anderen wenn beim Vergleich von zwei Zeichen ein Unterschied festgestellt wird. Schließlich werden Lanes ebenfalls inaktiv, wenn diese ihren Vergleich abgeschlossen haben und einen passenden String gefunden haben. Jeweils nach diesen Ereignissen soll nun überprüft werden, ob die Auslastung des Warps noch über einem bestimmten Grenzwert liegt. Ist dieser unterschritten, können wiederum zwei Fälle auftreten: Sollten sich im Puffer noch ausreichend Tupel befinden, sodass durch Auffüllen der Grenzwert wieder erreicht ist, so werden die inaktiv gewordenen Lanes mit den Zeichenketten aus den gepufferten Tupeln befüllt und es wird weiter gerechnet. Wurden hingegen zuvor zu wenige Elemente gepuffert, sodass der Grenzwert nicht erreicht werden kann, werden die Tupel aus den aktiven Lanes

im Puffer gespeichert und die Pipeline mit frischen Tupeln neu gestartet, damit später neue Zeichenketten für den String-Vergleich bereitstehen.

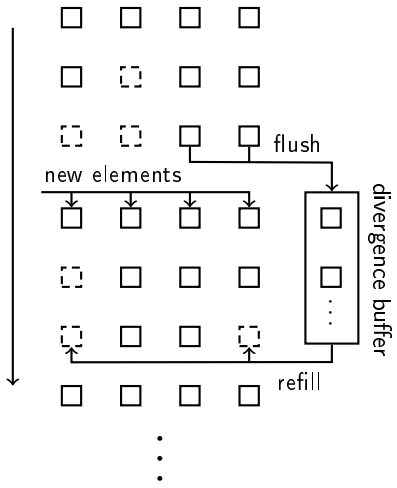

Abbildung 5.1: Funktionsweise des Lane Refill (Quelle: Henning Funke)

In Abbildung 5.1 ist die Funktionsweise der vorgestellten Technik dargestellt. Es ist zu erkennen, dass aktive Tupel in einem schlecht ausgelasteten Warp im Puffer zwischengespeichert werden, was hier als *flush* bezeichnet wird. Im Anschluss fließen neue Tupel über den Weg der Pipeline wieder in den Warp ein, wodurch dieser wieder vollständig ausgelastet ist. Sobald erneut eine Unterauslastung auftritt, werden die zwischengespeicherten Elemente aus dem Puffer in die inaktiven Lanes geladen und der Warp ist wieder effizient ausgelastet.

Die hier vorgestellte Technik ermöglicht es, die Unterauslastung innerhalb der String-Vergleichs-Operation zu verringern, allerdings entsteht dadurch gegebenenfalls eine starke Unterauslastung bei den Folgeoperationen in der Pipeline. Dieser Effekt tritt auf, da die Überprüfung der Zeichenketten zu sehr unterschiedlichen Zeitpunkten abgeschlossen sein kann und somit immer mal wieder einige Tupel bereit sind, die Folgeoperationen der Pipeline auszuführen, während viele andere Tupel sich noch innerhalb des String-Vergleichs befinden. Um dieses Problem zu lösen, kann beispielsweise nach dem String-Vergleich eine weitere Puffer-Operation eingeführt werden, welche sicherstellt, dass genug Zeichenketten fertig überprüft wurden, sodass die nachfolgenden Operationen mit einer ausreichenden Auslastung ausgeführt werden.

5.2 Struktur des optimierten String-Vergleichs im Kernel

Um den einfachen String-Vergleich durch Lane Refill zu verbessern, muss die Struktur des Operators im Kernel wie in Abbildung 5.2 dargestellt angepasst werden. Die äußere Schleife ist bereits aus der Grundstruktur einer Pipeline, wie sie in Kapitel 3 vorgestellt

wurde, bekannt und wurde hier für ein leichteres Verständnis aufgegriffen. Innerhalb der Schleife werden wie bei der naiven Implementierung zunächst die Operatoren ausgeführt, die in der Pipeline vor dem String-Vergleich stehen und es wird in Zeile 5 die Länge der Zeichenkette untersucht und diese gegebenenfalls verworfen.

Um im nächsten Schritt beurteilen zu können, ob eine ausreichende Auslastung besteht, wird zunächst in Zeile 8 mithilfe einer Synchronisierungsoperation überprüft, wie viele Lanes im Warp aktiv sind. Sind im Puffer und im Warp in Summe noch ausreichend aktive Elemente vorhanden, um den vorher definierten Grenzwert zu überschreiten, so kann ab Zeile 16 weiter mit dem String-Vergleich fortgefahren werden. Sollten die aktiven Lanes alleine nicht ausreichen, um den Grenzwert zu erreichen, werden gegebenenfalls zuvor die leeren Lanes mit Tupeln aus dem Puffer wieder aufgefüllt. Der eigentliche Vergleich zweier Zeichen funktioniert hier wieder genau wie bei der naiven Implementierung. Nach dem Vergleich wird in Zeile 20 geprüft, ob der String vollständig durchlaufen wurde und entsprechend die folgenden Operationen in der Pipeline ausgeführt. An dieser Stelle fällt auf, dass im Gegensatz zur naiven Implementierung eine Verschachtelung entsteht, da die Folgeoperationen innerhalb der in Zeile 10 beginnenden `while`-Schleife des String-Vergleichs-Operators ausgeführt werden.

Ist nach erneuter Zählung der aktiven Lanes die Schleifenbedingung in Zeile 10 nicht mehr erfüllt, da zu wenige Tupel existieren, um eine ausreichende Auslastung zu gewährleisten, werden die Tupel aus den restlichen, aktiven Lanes in Zeile 26 in den Puffer geschrieben. Schließlich wird die Pipeline mit frischen Tupeln wieder von vorne gestartet.

```
1  while(!flush_pipeline) {
2    /* execute previous operators in the pipeline */
3
4    // if string lengths are unequal, discard
5    if (active && data_length != search_length)
6      active = false;
7
8    int numactive = __popc(__ballot_sync(ALL_LANES, active));
9    int bufferelements = 0;
10   while(bufferelements + numactive > THRESHOLD) {
11     if (numactive < THRESHOLD) {
12       /* refill empty lanes from buffer in case of underutilization */
13
14       bufferelements = bufferelements - numrefill;
15     }
16     // when strings don't match, inactivate the lane
17     if (active && data_content[data_id] != search_string[search_id])
18       active = false;
19
20     if (++search_id == search_length) {
21       /* execute following operators in the pipeline */
22     }
23     numactive = __popc(__ballot_sync(ALL_LANES, active));
24   }
25   if (numactive > 0) {
26     /* flush active lanes to buffer */
27
28     bufferelements += numactive;
29     active = false;
30   }
31   loop_var += step;
32 }
```

Abbildung 5.2: Struktur der String-Selektion mit Lane Refill

5.3 Technische Umsetzung der Pufferung

Der Puffer wird mithilfe der CUDA-Programmierschnittstelle im Shared Memory umgesetzt, um eine effiziente Kommunikation zwischen den Lanes zu ermöglichen. Für den einfachen String-Vergleich werden dazu zwei Speicherbereiche benötigt, welche mit dem Schlüsselwort `__shared__` initialisiert werden. In einem Feld wird die Position des untersuchten Strings gespeichert (`current_divergence_buffer`) und in dem anderen Feld wird der Index des Zeichens innerhalb des Strings gespeichert, das als nächstes verglichen werden muss (`search_id_divergence_buffer`). Da der Shared Memory immer auf dem Level

eines ganzen Blocks gültig ist, muss dieser so viele Elemente fassen können wie es Threads pro Block gibt. Eine ausführliche Version der Umsetzung befindet sich in Anhang A, hier sollen aber dennoch die Techniken zum Zwischenspeichern und Laden von Elementen kurz beschrieben werden.

```
1  if (numactive < THRESHOLD) {
2    numRefill = min(32 - numactive, bufferelements);
3    numRemaining = bufferelements - numRefill;
4
5    previous_inactive = __popc(~__ballot_sync(ALL_LANES, active) &
         prefixlanes);
6
7    if (!active && previous_inactive < bufferelements) {
8      buf_ix = numRemaining + previous_inactive + bufferbase;
9      search_id = search_id_divergence_buffer[buf_ix];
10     current = current_divergence_buffer[buf_ix];
11     active = true;
12   }
13
14   bufferelements -= numRefill;
15 }
```

Abbildung 5.3: Befüllen inaktiver Lanes mit Elementen aus dem Puffer

Bei dem in Abbildung 5.3 dargestellten Befüllen von leer gelaufenen Lanes mit Elementen aus dem Puffer bleiben die noch aktiven Lanes unberührt. Die inaktiven Lanes müssen zunächst beurteilen, ob sie berechtigt sind, sich ein Tupel aus dem Puffer zuzuweisen. Befinden sich beispielsweise zwei Elemente im Puffer, es gibt aber drei inaktive Lanes, dann erhalten die zwei Lanes mit dem niedrigeren Index ein neues Tupel und die Lane mit höherem Index bleibt weiter inaktiv. Als erstes muss also in Zeile 7 bestimmt werden, wie viele Lanes vor der betrachteten Lane inaktiv sind. Der Wert `prefixlanes` ist dabei eine Bitmaske mit einem Bit für jede Lane im Warp, bei der für alle vor der aktuellen Lane liegenden Lanes ein Bit gesetzt wurde. Sind weniger Lanes vor der betrachteten Lane inaktiv als Elemente im Puffer vorhanden, darf sich diese Lane in den Zeilen 8-11 ein neues Tupel aus dem Puffer holen.

Um das richtige Element zu erhalten, muss zunächst wie in Abbildung 5.4 dargestellt dessen Index im Puffer errechnet werden, welcher sich aus verschiedenen Offsets zusammensetzt. `buffer_base` bestimmt dabei die Position des Speicherbereichs, welcher dem aktuellen Warp im Block zugewiesen wurde. Da der Puffer grundsätzlich von rechts abgearbeitet wird, muss noch der Wert `num_remaining` für die Elemente, die im Puffer verbleiben sollen, addiert werden und schließlich noch die Tupel übersprungen werden, die für Lanes mit niedrigerem Index bestimmt sind.

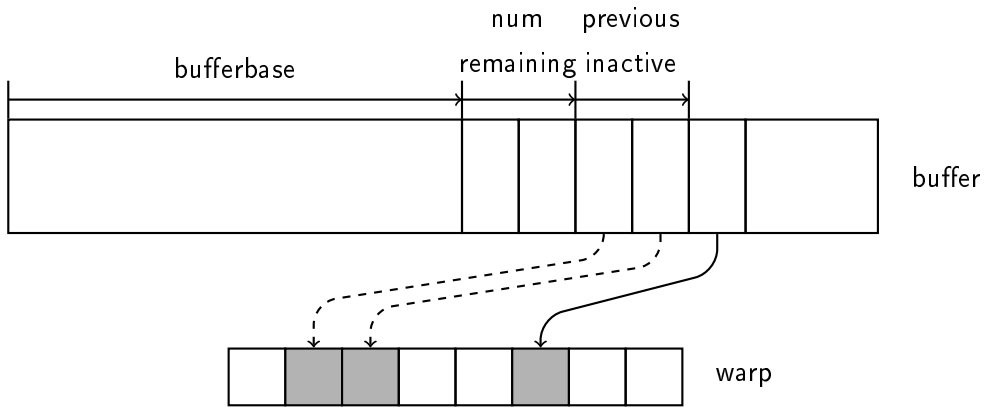

Abbildung 5.4: Berechnung des Indexes für ein Element im Puffer

Ist der Index berechnet, können der String und der Suchindex innerhalb des Strings aus dem Puffer geladen werden und die Lane wieder aktiv geschaltet werden.

```
1  if (numactive > 0) {
2    previous_active = __popc(__ballot_sync(ALL_LANES, active) & prefixlanes
         );
3    buf_ix = bufferbase + bufferelements + previous_active;
4
5    if(active) {
6      search_id_divergence_buffer[buf_ix] = character_index;
7      current_divergence_buffer[buf_ix] = current;
8    }
9
10   bufferelements += numactive;
11   active = false;
12 }
```

Abbildung 5.5: Auslagern übriger, aktiver Lanes in den Puffer

Das in Abbildung 5.5 dargestellte Verfahren zum Einlagern übriger, aktiver Lanes in den Puffer funktioniert ähnlich wie das Wiederbefüllen von Lanes. Zunächst wird wieder der Index berechnet, auf den die aktuelle Lane im Puffer zugreifen kann, was identisch zu dem in Abbildung 5.4 dargestellten Verfahren funktioniert, nur dass in diesem Falle die aktiven Lanes statt der inaktiven Lanes als Offset verwendet werden. Alle verbleibenden, aktiven Lanes speichern daraufhin die Position ihres vorher untersuchten Strings und den als nächstes zu untersuchenden Index innerhalb des Strings im Puffer ab. Schließlich wird noch die Anzahl der Elemente im Puffer erhöht und der Algorithmus kann anschließend wie gehabt verfahren.

5.4 Reduzierung des Overheads

Ein Nachteil dieses Verfahrens liegt darin, dass durch das Verwalten des Puffers ein erhöhter Overhead entsteht, der die bessere Performanz durch eine gute Auslastung der GPU überschatten könnte. Um diesen Overhead zu verringern, kann es sinnvoll sein, in größerem Zeitabstand die Auslastung zu überprüfen und entsprechend Lanes neu zu befüllen. Um dies zu erreichen, muss lediglich der Zeichen-Vergleich in einer kurzen Schleife laufen, sodass dieser ohne Unterbrechung mehrmals nacheinander ausgeführt wird. Dabei muss natürlich nach jeder Überprüfung auch geschaut werden, ob der String bereits fertig durchlaufen wurde, damit gegebenenfalls die Folgeoperationen in der Pipeline ausgeführt werden können. Durch das Schlüsselwort `#pragma unroll` ersetzt der Präprozessor die einfache Schleife durch Duplikate des Codes, wodurch eine weitere, kleine Leistungssteigerung erreicht wird.

Die in diesem Kapitel vorgestellte Technik zur Verbesserung der Leistungsfähigkeit des einfachen String-Vergleichs liefert einen vielversprechenden Ansatz, um die Ausführung von String-Operationen in Pipelining-Umgebungen auf Grafikkarten effizient zu gestalten, ohne dabei auf Dictionaries zurückgreifen zu müssen. Ob dieses Verfahren die gewünschten Leistungsziele erreichen kann, wird in Kapitel 10 anhand einiger praktischer Anwendungsszenarien ermittelt werden.

Kapitel 6

Grundlagen von regulären Ausdrücken

Reguläre Ausdrücke sind ein Mittel, um ein Muster von Zeichenketten zu beschreiben. Alle Wörter, die dem so beschriebenen Muster entsprechen, werden in einer sogenannten *Sprache* zusammengefasst. Erlaubte Operatoren innerhalb von regulären Ausdrücken sind beispielsweise die Konkatenation durch Aneinanderreihen von Elementen, Wiederholung durch den $*$-Operator, Auswahl durch den $|$-Operator oder das Verwenden einer Wildcard.

Die Überprüfung, ob ein Wort Teil der Sprache ist, die durch einen regulären Ausdruck beschrieben wird, wird *Wortproblem* genannt. Um dieses zu lösen, wird ein endlicher Automat verwendet, welcher aus dem regulären Ausdruck generiert wird und durch einen gerichteten Graphen repräsentiert wird. Die Knoten des Graphen werden auch Zustände genannt, wobei genau ein Zustand als Startzustand definiert wird und mindestens ein Zustand als akzeptierender Zustand ausgewählt wird. Jeder Kante, auch Transition genannt, wird eine Menge von Zeichen zugewiesen, sodass beim schrittweisen Durchlaufen des Eingabewortes ausgehend vom Startzustand der Automat über die Transitionen durchlaufen werden kann. Ist am Ende des Wortes ein akzeptierender Zustand erreicht, ist das Wort Teil der Sprache.

6.1 Äquivalenz verschiedener Modelle

Grundsätzlich lässt sich ein regulärer Ausdruck in einen *deterministischen endlichen Automaten (DFA)* oder einen *nichtdeterministischen endlichen Automaten (NFA)* umwandeln. Der Unterschied zwischen diesen beiden Automaten besteht darin, dass nur der NFA ausgehend von einem Zustand mehrere Transitionen mit demselben Zeichen zulässt. An dieser Stelle entsteht ein Nichtdeterminismus, da beim Lesen eines Zeichens des Eingabewortes nicht eindeutig ist, welche Transition verfolgt werden muss. Die Modelle des regulären Ausdrucks, des DFA und des NFA sind äquivalent und lassen sich daher ineinander überführen

[6]. Aus dem regulären Ausdruck `(0|1)*((00)+|001)0` lassen sich somit der in Abbildung 6.1 dargestellte NFA und der in Abbildung 6.2 dargestellte DFA generieren.

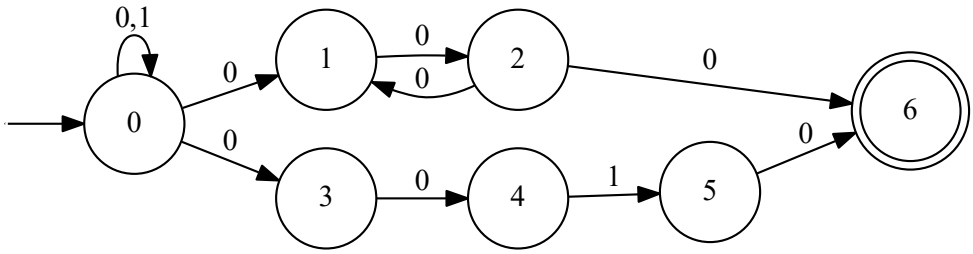

Abbildung 6.1: Visuelle Darstellung des NFA zum regulären Ausdruck `(0|1)*((00)+|001)0`

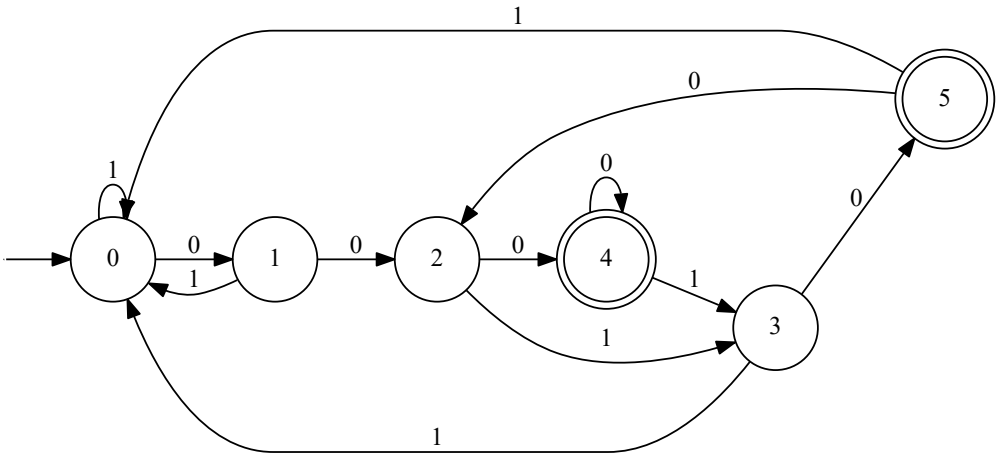

Abbildung 6.2: Visuelle Darstellung des DFA zum regulären Ausdruck `(0|1)*((00)+|001)0`

6.2 Durchführung eines Musterabgleichs

Der vorgestellte DFA lässt sich als zweidimensionale Übergangstabelle darstellen, die für einen gegebenen Ausgangszustand und ein gelesenes Eingabezeichen den Folgezustand liefert. Beginnend mit dem Startzustand muss somit lediglich für jedes Zeichen des Eingabewortes der aktuelle Zustand entsprechend der Tabelle angepasst werden und es kann am Ende überprüft werden, ob ein akzeptierender Zustand erreicht wurde. Dieses Verfahren wird in Abbildung 6.3 schematisch dargestellt.

```
1  dfa = [[1, 0],
2    [2, 0],
3    [4, 3],
4    [5, 0],
5    [4, 3],
6    [2, 0]]
7  cs = 0
8  accepting = [4, 5]
9
10 word = "010010"
11
12 for c in word:
13   cs = dfa[cs][int(c)]
14
15 if cs in accepting:
16   print("Word matches regular expression!")
```

Abbildung 6.3: Durchführung des Musterabgleichs mithilfe eines DFA

Abbildung 6.4 stellt entsprechend das Verfahren zur Durchführung des Musterabgleichs mittels eines NFAs dar. Dieses funktioniert sehr ähnlich zu einem DFA, allerdings müssen gegebenenfalls mehrere Zustände gleichzeitig betrachtet werden.

6.3 Auswahl eines geeigneten Ansatzes

Der Nachteil bei der Verwendung von NFAs liegt darin, dass teilweise mehrere Zustände gleichzeitig aktiv sind, da verschiedene Pfade auf einmal verfolgt werden müssen. Dadurch steigt die Rechenlast beim Lesen von Zeichen und die allgemeine Performanz sinkt. Bei DFAs ist dies nicht der Fall, da durch den Determinismus zu jeder Zeit klar ist, welche Transition verfolgt werden muss und in welchem Zustand der Automat sich befindet.

Der Nachteil eines DFA liegt allerdings darin, dass dieser im Gegensatz zum NFA eine exponentiell große Anzahl von Zuständen haben kann. die Zeit für das Verarbeiten eines Strings wird dadurch zwar nicht direkt beeinflusst, allerdings steigt der Speicherverbrauch enorm, wodurch die Verwendung eines DFA gegebenenfalls unmöglich wird.

Vergangene Arbeiten zeigen, dass ein unkomprimierter DFA die optimale Laufzeit erzielt [18]. Außerdem ist nicht zu erwarten, dass im Kontext einer Datenbankanfrage ein hoch komplexer regulärer Ausdruck benötigt wird, sodass der Automat eine effiziente Größe überschreiten würde. Der exponentiellen Anzahl von Zuständen kann außerdem entgegen gewirkt werden, indem der Automat vor der Verwendung minimiert wird, was in den meisten Anwendungsfällen einen kompakten Automaten ergibt. Aus diesem Grund wird im folgenden Kapitel ein DFA für den Musterabgleich verwendet.

```
1  nfa = [[[0, 1, 3], [0]],
2    [[2], []],
3    [[1, 6], []],
4    [[4], []],
5    [[], [5]],
6    [[6], []],
7    [[], []]]
8  cs = [0]
9  accepting = [6]
10
11 word = "010010"
12
13 for c in word:
14   newCs = []
15   for s in cs:
16     newCs = newCs + nfa[s][int(c)]
17   cs = newCs
18
19 for s in cs:
20   if s in accepting:
21     print("Word matches regular expression!")
22     break
```

Abbildung 6.4: Durchführung des Musterabgleichs mithilfe eines NFA

Kapitel 7

Paralleler Musterabgleich mit regulären Ausdrücken

Aufbauend auf Kapitel 4 wird im Folgenden eine komplexere Operation in Form eines Matchers für reguläre Ausdrücke im Kontext der in Kapitel 3 beschriebenen Compiled Query Pipelines auf GPUs untersucht. Diese Operation stellt ebenfalls eine Selektion über eine Spalte von String-Daten dar, bei der alle Tupel in die Ergebnisrelation übernommen werden, welche dem Muster eines vorgegebenen regulären Ausdrucks entsprechen.

Zunächst wird dazu erläutert, warum das Umsetzen einer solchen Operation im Kontext von Datenbanksystemen relevant ist. Anschließend wird die allgemeine Struktur der Operation dargestellt, gefolgt von der tatsächlichen Umsetzung des Musterabgleichs mithilfe von Automaten. Schließlich werden noch einige alternative Verfahren vorgestellt, welche unterschiedliche Eigenschaften aufweisen und mit der vorgestellten Umsetzung verglichen werden.

7.1 Struktur der Operation

Für das Verarbeiten des regulären Ausdrucks wird ein deterministischer Automat erstellt, der genau dann akzeptiert, wenn der aktuell überprüfte String in das Muster des regulären Ausdrucks passt.

Das allgemeine Vorgehen der Operation ist ähnlich zu dem in Kapitel 4.1 beschriebenen einfachen String-Vergleich. Jedem Thread der GPU wird ein Tupel zugewiesen, welcher mithilfe des Automaten untersucht werden soll. Nachdem der aktuelle Zustand des DFA auf den Startzustand gesetzt wurde, wird der String Zeichen für Zeichen durchlaufen, wobei der Zustand in jedem Schritt entsprechend der Regeln des Automaten aktualisiert wird. Sobald die Zeichenkette vollständig durchlaufen wurde, wird geprüft, ob sich der Automat in einem akzeptierenden Zustand befindet, in welchem Fall der gesuchte String in das Muster des regulären Ausdrucks passt. Ist am Ende kein akzeptierender Zustand erreicht,

oder wurde die Untersuchung vorzeitig abgebrochen, weil ein Fehlerzustand erreicht wurde, wird das Tupel nicht in das Ergebnis übernommen.

Bei der Ausführung wird der gesamte Warp parallel abgearbeitet, wodurch die Positionen, an denen die Strings untersucht werden, für alle Lanes identisch sind. Außerdem wird das Ergebnis erst geschrieben, sobald alle Threads ihren String vollständig durchlaufen haben oder einen Fehlerzustand erreicht haben. Schließlich wird jedem Thread eine neue Zeichenkette zugewiesen, mit der das Verfahren wiederholt wird, bis sämtliche Tupel abgearbeitet sind.

```
1  /* execute previous operators in the pipeline */
2
3  char *p = data_content + char_offset[loop_var];
4  char *pe = data_content + char_offset[loop_var + 1];
5
6  int cs = machine_start;
7
8  while(active) {
9    cs = singleDfaStep(cs, p);
10
11   p++;
12
13   if (p == pe)    // string completely processed
14     active = false;
15
16   if (cs == 0)    // invalid state reached
17     active = false;
18 }
19
20 active = cs >= machine_first_final;
21
22 /* execute following operators in the pipeline */
```

Abbildung 7.1: Naive Implementierung einer Selektion mit einem regulären Ausdruck

In Abbildung 7.1 wird die Implementierung des Operators vorgestellt, der im Kontext der kompilierten Anfragepipelines die Selektion über einen regulären Ausdruck ausführt. Die Datensätze `data_content` und `char_offset` enthalten wie in Kapitel 4.1 die Zeichenketten aus der zu untersuchenden Spalte und die Indizes der einzelnen Tupel innerhalb des Datensatzes.

Zunächst wird die Lane initialisiert, indem ein Zeiger `p` auf den Anfang des zu untersuchenden Strings und der Zeiger `pe` auf das Ende gesetzt wird. Außerdem wird der aktuelle Zustand `cs` auf den Startzustand des Automaten gesetzt.

In der Schleife wird anschließend über den gesamten String iteriert und dabei mithilfe der Methode `singleDfaStep` der Folgezustand des Automaten nach Einlesen des nächsten

Zeichens bestimmt. Daraufhin wird überprüft, ob der iterierende Zeiger `p` auf das Ende des Strings `pe` zeigt, in welchem Falle dieser vollständig durchlaufen wurde und die Lane vorerst deaktiviert werden kann. Die Lane wird ebenfalls deaktiviert, wenn der Fehlerzustand `0` erreicht wurde, von dem aus ein Erreichen eines akzeptierenden Zustandes unmöglich ist.

Nachdem sämtliche Lanes ihre Berechnung abgeschlossen haben, wird überprüft, ob der aktuelle Zustand des DFA zu der Gruppe der akzeptierenden Zustände gehört. Ist dies der Fall, wird die Lane für folgende Operationen aktiviert, ansonsten bleibt diese deaktiviert, sodass die Folgeoperationen nicht ausgeführt werden müssen.

7.2 Erstellen und Durchlaufen des Automaten

Das vorgestellte Verfahren generiert vor der Ausführung des Anfrageplans einen deterministischen Automaten, welcher beim kompilieren der Anfragepipeline in den Kernel eingebaut wird. Der Automat wird mithilfe des *Ragel State Machine Compilers* [14] erzeugt, auf dem auch der Code zur Verarbeitung des Automaten basiert.

Ragel ermöglicht es, für das Auswerten eines gegebenen regulären Ausdrucks C-Code zu erzeugen, der automatisch in ein vorgegebenes Rahmenprogramm eingefügt wird. Somit ist es leicht möglich, den Automaten in den Rahmen einer kompilierten Anfragepipeline einzupflegen, es müssen also keinerlei Schnittstellen zu anderen Sprachen oder Konzepten erstellt werden.

Der generierte Code lässt sich in zwei Hauptbestandteile aufteilen. Zum einen wird etwas Code zur tatsächlichen Durchführung des Musterabgleichs generiert, welcher für unterschiedliche Ausdrücke größtenteils identisch ist. Zum anderen werden einige Tabellen generiert, welche die tatsächlichen Zustände und Zustandsübergänge des Automaten beinhalten und für jeden regulären Ausdruck neu generiert werden.

Abbildung 7.2 zeigt die Methode zur Durchführung eines DFA-Schrittes, welche in Abbildung 7.1 verwendet wird. Die Implementierung basiert auf dem durch Ragel erzeugten Ausführungscode, welcher mit der *flat*-Einstellung generiert wurde. Dies stellt den simpelsten Code dar, der von Ragel generiert wird, welcher sehr ähnlich zu dem in Kapitel 6 vorgestellten Prinzip ist. Hier werden die ebenfalls von Ragel generierten und in Abbildung 7.3 dargestellten Felder verwendet, welche den eigentlichen DFA enthalten. In diesem Beispiel handelt es sich um den Automaten, der aus dem Ausdruck `(0|1)*((00)+|001)0` generiert wurde.

Da diese Darstellung des Automaten für den Menschen kaum lesbar ist und ein Debugging somit sehr aufwändig wäre, bietet Ragel außerdem ein Werkzeug zur Visualisierung des Graphen. Die vorher gezeigte Abbildung 6.2 zeigt dazu die visuelle Darstellung des erzeugten DFA.

```
1   __device__ int singleDfaStep(int cs, char* p) {
2     int _slen;
3     int _trans;
4     const char *_keys;
5     const char *_inds;
6
7     _keys = _machine_trans_keys + (cs<<1);
8     _inds = _machine_indicies + _machine_index_offsets[cs];
9
10    _slen = _machine_key_spans[cs];
11    _trans = _inds[ _slen > 0 && _keys[0] <=(*p) &&
12      (*p) <= _keys[1] ?
13      (*p) - _keys[0] : _slen ];
14
15    return _machine_trans_targs[_trans];
16  }
```

Abbildung 7.2: Methode zur Durchführung eines DFA-Schrittes

7.3 Alternative Verfahren

Zusätzlich zur *flat*-Einstellung bietet Ragel die *Table*-Option, welche eine Generierung von Code erlaubt, welche für die Verarbeitung regulärer Ausdrücke mit einem großen Alphabet optimiert wurde. Im Gegensatz zur *flat*-Einstellung wird nicht das aktuell untersuchte Zeichen als Index für einen Array von Zustandsübergängen verwendet, sondern in einem Feld eine binäre Suche nach der korrekten Transition durchgeführt. Nach Thurston ist die oben beschriebene *flat*-Option generell schneller, sie lässt sich allerdings nur für ein kleines Alphabet anwenden [14]. Da im Datenbankkontext generell eher simplere Ausdrücke zu erwarten sind, ist es für die meisten Anwendungsfälle zwar ausreichend, die einfache und schnelle Implementierung zu wählen. Es ist aber auch interessant zu untersuchen, wie groß der Leistungsverlust mit der alternativen Implementierung aussieht, falls komplexere Ausdrücke untersucht werden sollen.

Als Alternative zur vollumfänglichen Unterstützung von regulären Ausdrücken bieten viele Datenbankmanagementsysteme den *LIKE*-Operator an, welcher den einfachen String-Vergleich um Platzhalter erweitert. So ist es dem Nutzer möglich, anstelle des genauen Suchstrings ein Muster anzugeben, in dem Platzhalter für beliebige Zeichen enthalten sind. Mithilfe dieser simpel umzusetzenden Operation lassen sich viele einfache Abfragen an eine Datenbank modellieren, sodass in vielen Fällen kein richtiger regulärer Ausdruck benötigt wird. Es ist daher interessant zu sehen, wie sich die Leistungsfähigkeit dieser Operation mit geringerem Funktionsumfang gegenüber der Umsetzung eines vollständigen Matchers für reguläre Ausdrücke verhält.

```
1  static const char _machine_trans_keys[] = {
2    0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0
3  };
4
5  static const char _machine_key_spans[] = {
6    0, 2, 2, 2, 2, 2, 2
7  };
8
9  static const char _machine_index_offsets[] = {
10   0, 0, 3, 6, 9, 12, 15
11 };
12
13 static const char _machine_indicies[] = {
14   0, 2, 1, 3, 2, 1, 4,
15   5, 1, 6, 2, 1, 4, 5, 1,
16   3, 2, 1, 0
17 };
18
19 static const char _machine_trans_targs[] = {
20   2, 0, 1, 3, 5, 4, 6
21 };
22
23 static const int machine_start = 1;
24 static const int machine_first_final = 5;
```

Abbildung 7.3: Generierte Felder, die den DFA enthalten

Kapitel 8

Verbesserung des Verfahrens zum Musterabgleich

Bei dem in Kapitel 7 vorgestellten Verfahren tritt ein ähnliches Problem auf wie bei dem zuvor beschriebenen, einfachen String-Vergleich. Aufgrund der unterschiedlichen Struktur von Strings werden einige Lanes innerhalb eines Warps früher als andere Lanes dadurch inaktiv, dass sie einen Fehlerzustand oder das Ende des Eingabestrings erreicht haben. Dies hat eine Unterauslastung des Warps zur Folge, wodurch Rechenleistung verschwendet wird. Aus diesem Grund ist es wünschenswert, den inaktiv gewordenen Threads dynamisch neue Arbeit zuzuweisen. Dazu wird das in Kapitel 5 vorgestellte Lane Refill-Verfahren zum Einsatz kommen, wodurch im Rahmen der kompilierten Anfragepipelines die Laufzeit optimiert werden kann.

8.1 Struktur des optimierten Musterabgleichs mit Lane Refill

Die grundlegende Funktionsweise des Lane Refill wurde in Kapitel 5.1 beschrieben und ist genau so auch auf den parallelen Musterabgleich anwendbar. Bei dem Musterabgleich wird eine Lane immer dann inaktiv, wenn der untersuchte String vollständig durchlaufen oder ein Fehlerzustand erreicht wurde und es nicht mehr möglich ist, einen akzeptierenden Zustand zu erreichen. Nach genau diesen Ereignissen muss überprüft werden, ob die gewünschte Auslastung des Warps unterschritten wird und gegebenenfalls die aktuellen Elemente in den Puffer geschrieben oder neue Elemente aus dem Puffer geladen werden.

Die allgemeine Struktur des Kernels ist identisch zu der in Abbildung 5.2 vorgestellten Struktur des einfachen String-Vergleichs. Um den parallelen Musterabgleich damit umzusetzen, muss die innere Schleife so angepasst werden, dass statt einem einfachen Vergleich zweier Zeichen der Zustand des Automaten angepasst wird. Der gesamte Kernel ist in Anhang B dargestellt.

Zunächst wird hier von allen aktiven Lanes ein Schritt im DFA durchgeführt und überprüft, ob ein Fehlerzustand erreicht wurde. Anschließend wird für die vollständig durchlaufenen Strings überprüft, ob diese sich in einem akzeptierenden Zustand befinden und gegebenenfalls die Folgeoperationen der Pipeline ausgeführt.

Die technische Umsetzung der Puffer-Operationen funktioniert identisch zu dem in Kapitel 5.3 beschriebenen Vorgehen, mit dem Unterschied, dass hier der Inhalt der Variablen `p`, `pe` und `cs` im Puffer gespeichert werden. Eine Reduzierung des Overheads durch die Puffer-Operation ist ebenfalls analog zu dem in Kapitel 5.4 vorgestellten Verfahren möglich.

Kapitel 9

Optimierung der Ausführungsparameter

Die Ausführungszeit der in den vorherigen Kapiteln vorgestellten Algorithmen wird stark durch unterschiedliche Parameter beeinflusst. Den größten Einfluss nimmt dabei die Art des Datensatzes, also wie viele Matches er enthält, wie lang die enthaltenen Zeichenketten sind, wie viele Strings im Datensatz vorhanden sind und wie die Matches darin verteilt sind. Wie sehr unterschiedliche Datensätze die Ausführungszeit beeinflussen, soll in Kapitel 10 untersucht werden.

Neben den Eigenschaften des Datensatzes gibt es noch Parameter, die bei der Ausführung des Algorithmus der GPU übermittelt werden und dort ebenfalls einen erheblichen Einfluss auf die Laufzeit nehmen. Diese Parameter bestehen, wie in Kapitel 2.2 beschrieben, in der Anzahl der Threads pro Block (*Block Size*) und der Anzahl der Blöcke im Grid (*Grid Size*). Aus den Parametern setzt sich die *Grid-Konfiguration* zusammen, die durch das Ausprobieren unterschiedlicher Werte praktisch optimiert werden kann.

Um dieses Vorgehen zu veranschaulichen, soll anhand eines Beispiels gezeigt werden, wie eine möglichst gute Grid-Konfiguration gefunden werden kann und welchen Einfluss unterschiedliche Konfigurationen auf die Laufzeit haben. Abbildung 9.1 zeigt dazu die Laufzeit für den String-Vergleichsalgorithmus in Millisekunden, welcher für unterschiedliche Grid-Konfigurationen auf dem in Kapitel 10.2 vorgestellten Type-Datensatz mit einer Selektivität von 0.25% ausgeführt wurde. Besonders hohe Laufzeiten sind dabei rot und besonders geringe Laufzeiten grün eingefärbt.

Es fällt auf, dass eine niedrige Block Size in Kombination mit einer niedrigen Grid Size zu einer hohen Laufzeit führt. Außerdem gibt es bei einer Grid Size von unter 100 einige Kombinationen, die besonders geringe Laufzeiten aufweisen, allerdings von weniger guten Konfigurationen umgeben sind. Mit einer Grid Size zwischen 2.000 und 200.000 in Verbindung mit einer Block Size zwischen 64 und 512 werden generell gute Laufzeiten

erzielt. Wird allerdings eine Block Size über 512 gewählt, nimmt die Leistungsfähigkeit des Systems wieder ab.

Generell ist die Analyse eines solchen Ergebnisses schwierig, da die Grid-Konfiguration Einfluss auf unterschiedlichste Bereiche der GPU nimmt und der CUDA-Optimierer einige Effekte erfolgreich versteckt. Die hohe Laufzeit des Algorithmus bei einer geringen Anzahl von Threads ist dadurch zu erklären, dass zunächst nicht alle Kerne der GPU ausgelastet sind, weil nicht genügend Warps für die Anzahl der Streaming Multiprocessors vorhanden sind. Eine steigende Anzahl von Threads führt zwar dazu, dass alle Kerne ausgelastet sind, allerdings können bei Speicherzugriffen nicht genügend Warps vom Scheduler ausgetauscht werden, als dass die Latenz der Speicherzugriffe, wie in Kapitel 2.2 beschrieben, erfolgreich versteckt werden könnte. In dem zuvor beschriebenen Bereich, in dem ordentliche Laufzeiten erzielt werden, steht eine ausreichende Anzahl von Threads zur Verfügung, um eventuelle Latenzen zu verstecken und somit die GPU bestmöglich auszulasten. Wie der Anstieg der Laufzeiten bei einer Block Size von über 512 zu begründen ist, ist an dieser Stelle unklar.

Für die Verwendung des Algorithmus sollte im Allgemeinen eine Grid-Konfiguration aus dem Bereich gewählt werden, der eine durchweg ordentliche Performanz erzielt. Dadurch wird mit hoher Wahrscheinlichkeit eine Konfiguration gewählt, mit der eine Laufzeit erreicht wird, die bis auf eine kleinere Abweichung dem Optimum entspricht. Die Position des Bereichs ändert sich für unterschiedliche Selektivitäten des Datensatzes nur geringfügig, wie in zahlreichen Tests untersucht wurde und beispielhaft in Anhang C für eine Selektivität von 64% analog zu Abbildung 9.1 dargestellt wird. Aus diesem Grund kann die Grid-Konfiguration schon vor der Ausführung bestimmt werden und eine nah am Maximum liegende Leistungsfähigkeit erzielt werden. Bei den Leistungsmessungen in den folgenden Kapiteln sollte sich allerdings nicht darauf verlassen werden, dass durch diese Annäherung ein nahezu optimaler Wert gefunden wird, weshalb für die Tests jeweils das Optimum aus einer Auswahl von Grid-Konfigurationen bestimmt wird. Dazu werden alle Konfigurationen mit einer Grid Size von $G = \{1.000, 2.000, 3.000, 4.000, 6.000, 8.000, 10.000, 20.000, 50.000, 100.000, 150.000, 200.000\}$ und einer Block Size von $B = \{32, 64, 96, 128, 160, 192, 224, 256, 384, 512, 640, 768\}$ überprüft und das Optimum der Messungen für die Auswertung der folgenden Experimente gewählt.

Grid Size \ Block Size	32	64	96	128	160	192	224	256	384	512	640	768	896	1024
10	806	506	352	267	242	187	166	146	102	81	76,2	59,9	54	49,8
12	806	406	283	212	185	150	133	117	82,5	65,5	57,9	48,2	43,2	39,7
14	690	366	250	191	165	137	120	106	76,1	61,1	54,7	80,9	67,9	61,3
16	604	318	219	169	144	121	105	92,5	67,1	53,6	47,7	71,1	63,5	55,3
18	537	283	196	150	128	107	93,3	82,4	59,6	47,8	42	63,3	56,8	51,9
20	498	267	187	145	131	102	89,9	80,6	58,8	48,6	75,6	59,1	52,9	49,5
30	344	186	129	99,5	92,6	71,9	64,5	56,5	41,6	60,3	54,6	58,9	52,9	46,3
40	263	145	101	80,5	74	57,7	51,8	45,6	61,2	47,3	60,4	58,9	52,7	47,1
60	185	99,5	71,9	56,4	53,3	41,4	66,7	59,4	41,7	46,3	53,9	49,9	45,2	42,6
80	145	78,7	57,9	45,6	67,7	57	52	46	46,4	46,9	50,4	52,1	48,2	44,6
100	124	74,2	53,3	67,8	65,2	51,6	46,7	57,6	53,2	50,6	53,6	57,9	52	49
150	92	53,5	61,6	51,3	61,2	51,8	46,8	51,7	46,8	48,8	53,4	56	50	46,9
200	111	64,9	51,2	54,8	53,5	51,8	46,7	49,7	47,8	48,4	53,5	54,9	49,6	46,7
300	86,9	52,1	51,4	51	56,1	47,4	47,4	48,4	46,9	47	50,9	54,6	48,7	46,4
400	90,9	53,9	51,7	49,8	54,5	47,2	48,1	48,3	48	47	50,7	53,9	48	46,4
500	84,5	53,5	53,7	54,8	60,2	54,1	53,4	52,3	50,8	51,2	59,9	57,7	51,4	49,1
1000	80,3	55,9	55,8	54,3	58,3	50,4	51,8	51,5	49,6	49	58,1	57,4	49,5	47,4
2000	78,5	56,5	54,3	53,2	54,2	47,9	49,2	49	48,1	46,9	55	54,1	48,5	45,5
3000	77,6	55,9	53,2	50,9	52,9	47,3	47,3	46,6	47	45,7	52,9	54	48	45,1
4000	76,1	54,5	51,6	49,5	49,4	46,1	46	45,5	46,7	44,5	52,1	53,8	50	46,1
6000	75,3	52,6	48,5	45,5	47,4	44,9	44,9	44,5	44,5	43,9	51,9	55,3	49,7	47,8
8000	74,9	50,7	45,4	44,2	46,2	45,5	44	43,4	44,5	44,8	51,8	56,8	51,7	49,2
10000	74,8	49,7	46	45,3	47	45,1	44,5	45	48,2	50,1	58,6	61,1	55,6	52,6
20000	73,2	45,6	44,3	42,1	44,8	43,5	44,3	44,7	48	49,1	57,9	61,2	56,6	54
50000	69,1	47,1	43,4	43,7	44,5	45,2	45,8	46,3	49,5	52,8	57,8	65,7	67	61,5
100000	64,9	47	43,1	43,5	45,4	45	45,9	46,4	49,6	52,5	56,9	66,2	66,8	62,4
150000	65	46,8	43,1	43,5	44,6	45,2	46	46,6	49,3	52,5	57,8	69,6	69	64,9
200000	64,6	46,7	43,2	43,7	44,7	45,5	46,1	47,1	49,5	53,7	59,2	70,2	71,4	67,4

Abbildung 9.1: Laufzeit des naiven String-Vergleichsalgorithmus in Millisekunden für den Type-Datensatz mit einer Selektivität von 0.25% unter Verwendung unterschiedlicher Grid-Konfigurationen

Kapitel 10

Evaluation des einfachen String-Vergleichs

In Kapitel 5 wurde eine Technik vorgestellt, von der zu erwarten ist, dass sie die Laufzeit des einfachen String-Vergleichs verbessert, indem die Ressourcen der GPU besser genutzt werden und somit eine erhöhte Auslastung erreicht wird. Da diese Technik allerdings einen gewissen Overhead mit sich bringt, bleibt noch zu untersuchen, ob sie tatsächlich eine bessere Laufzeit erzielt, oder ob der Mehraufwand so groß ist, dass die erreichten Vorteile überschattet werden. In diesem Kapitel wird diese Untersuchung anhand realer Arbeitslasten durchgeführt und außerdem überprüft, ob die in Kapitel 5.4 vorgestellte Reduzierung des Overheads eine weitere Leistungssteigerung mit sich bringt.

10.1 Testumgebung

Für die Durchführung der Leistungsmessungen wird der Algorithmus so angepasst, dass er lediglich die Anzahl der passenden Zeichenketten zählt und diese am Ende ausgibt. Die Testumgebung entspricht also einer Selektion auf einer Spalte einer Relation und dem anschließenden Zählen der Ergebnisse. Dieses Vorgehen hat den Vorteil, dass das Zählen der Ergebnisse nicht viel Rechenaufwand verursacht und somit die Leistungsmessungen möglichst wenig verfälscht werden. Trotzdem bleibt es möglich, aufgrund der Ausgabe des Algorithmus beurteilen zu können, ob der Test korrekt durchgeführt wurde.

Sämtliche Tests wurden auf einem Computer durchgeführt, welcher eine Nvidia GTX 950 mit 2GB Grafikspeicher verbaut hat und als Betriebssystem Ubuntu 18.04 verwendet. Außerdem sind ein Intel Core i3-2120 mit zwei Kernen und 16GB Arbeitsspeicher verbaut.

10.2 Verwendete Workloads und deren Merkmale

In analytischen Anwendungsfällen kommen häufig selektive Filter vor [1], weshalb diese ebenfalls für die hier durchgeführten Untersuchungen verwendet werden. Außerdem ist zu erwarten, dass diese besonders stark vom Lane Refill profitieren werden, da bei einer kleinen Menge von Ergebnistupeln oftmals eine starke Unterauslastung auftritt.

Der erste verwendete Workload, welcher im Folgenden *Type* genannt wird, wurde aus dem TPC-H-Benchmark[1] entnommen. Hier wird eine Selektion über die Spalte *Type* durchgeführt, welche Zeichenketten der Länge 16-25 enthält. Diese bestehen aus den Zeichen *A-Z* und dem Leerzeichen. Für die Untersuchung wurde ein Datensatz mit 90.000.000 Tupeln generiert.

Ein weiterer Workload wurde aus dem Datensatz der DBLP Computer Science Bibliography[2] erstellt, welcher die Titel vieler Veröffentlichungen im Informatik-Umfeld enthält. Dazu wurden doppelte Titel entfernt und die übrigen Strings so angepasst, dass diese nur noch Kleinbuchstaben enthalten. Die durchschnittliche Länge der Zeichenketten in diesem Datensatz beträgt 76 Zeichen und es wurde ein Präfix gesucht, das 31 Zeichen beinhaltet. Der generierte Datensatz enthält schließlich 21.513.695 Tupel.

Um die unterschiedlichen Selektivitäten für die folgenden Tests zu erreichen, wurde ein neuer String entsprechend zufällig verteilt in den Datensatz eingebracht. Die gewünschte Datengröße wurde schließlich erreicht, indem der so generierte Datensatz einige male vervielfacht wurde.

10.3 Vorstellung der Messergebnisse

In den Abbildungen 10.1 bis 10.3 sind die Ergebnisse der durchgeführten Messungen aufgetragen. Hierzu wurden die verschiedenen Algorithmen verglichen und auf ihr Verhalten bei unterschiedlichen Anteilen von passenden Strings im Datensatz untersucht. Als Vergleichsgröße wurde die Laufzeit der Algorithmen verwendet, welche ein möglichst allgemein nutzbares und neutrales Maß darstellt.

10.3.1 Gleichheitstest mit dem Type-Datensatz

In Abbildung 10.1 werden die Ergebnisse eines Tests der vorgestellten Algorithmen bei Verwendung des vorher beschriebenen *Type*-Datensatzes ausgewertet. Dabei werden die Laufzeiten der naiven Umsetzung mit denen der verbesserten Variante mit Lane Refill verglichen. Außerdem wurde in der Messung die in Kapitel 5.4 beschriebene Variante zur Reduzierung des Overheads berücksichtigt, welche einmal so durchgeführt wurde, dass immer zwei Zeichen pro Schritt verglichen werden und einmal in Dreierschritten.

[1] `http://www.tpc.org/tpch/`
[2] `https://dblp.org/`

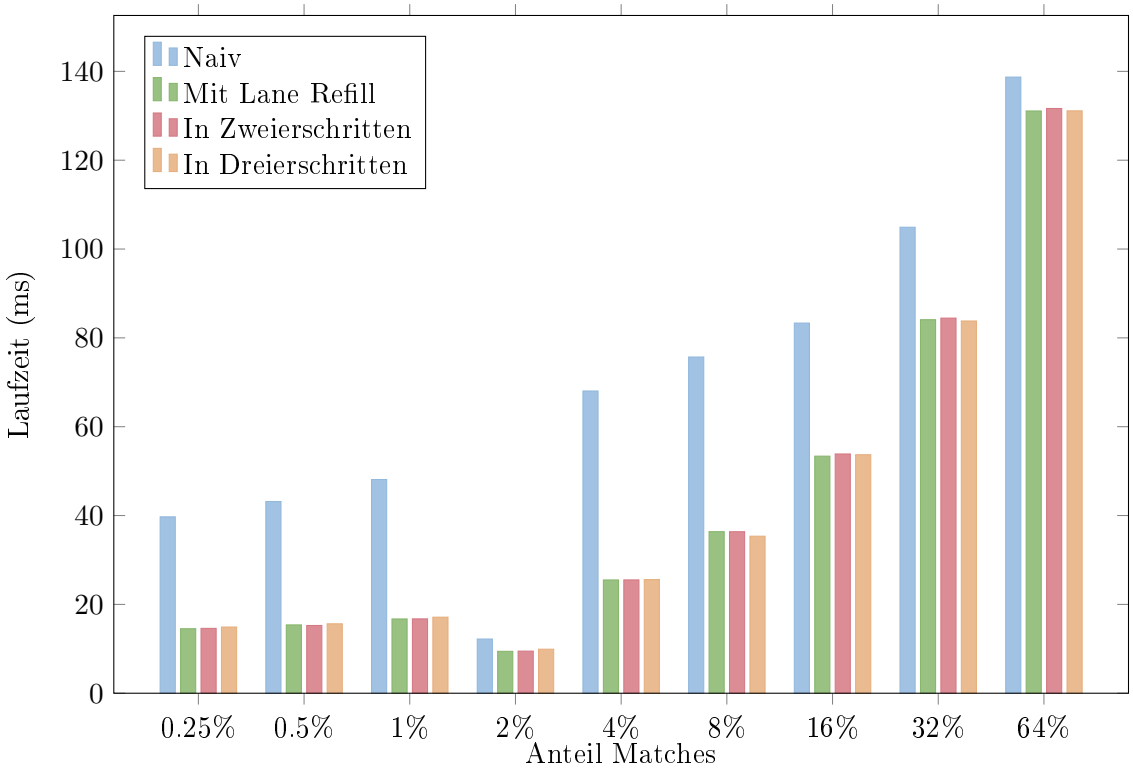

Abbildung 10.1: Laufzeit für Gleichheitstest mit verschiedener Verteilung beim Type-Benchmark

Sofort fällt auf, dass eine höhere Anzahl von Matches im Datensatz eine erhöhte Laufzeit für den Algorithmus bedeutet. Außerdem liegt die Laufzeit der naiven Umsetzung bedeutend höher als die der drei optimierten Varianten, welche das Lane Refill-Verfahren verwenden. Der Algorithmus, welcher das Lane Refill implementiert, ist zwischen 4% und 43% schneller als die naive Implementierung. Dabei entsteht der größte Vorteil bei einem geringen Anteil von passenden Elementen, welcher geringer wird, sobald der Datensatz eine höhere Anzahl von Matches enthält. Bis zu einem Anteil von 8% beträgt die Verbesserung der Laufzeit, die durch das Lane Refill-Verfahren erreicht wurde, mehr als 50%. Es ist außerdem zu erkennen, dass die beiden Varianten, welche die Reduzierung des Overheads erzielen sollten, eine nahezu identische Laufzeit erreichen wie die erste Version, die Lane Refill verwendet. Schließlich entsteht bei der Messung ein Ausreißer bei einem Anteil passender Matches von 2%, der sich durch besonders geringe Laufzeiten für alle Algorithmen auszeichnet.

10.3.2 Präfixtest mit dem Type-Datensatz

Abbildung 10.2 zeigt das Ergebnis eines ähnlichen Tests wie dem vorher beschriebenen Experiment, mit der Ausnahme, dass hier der in Kapitel 4.2 beschriebene Präfixtest untersucht wird. Der verwendete *Type*-Datensatz blieb dabei unverändert, es wurden lediglich

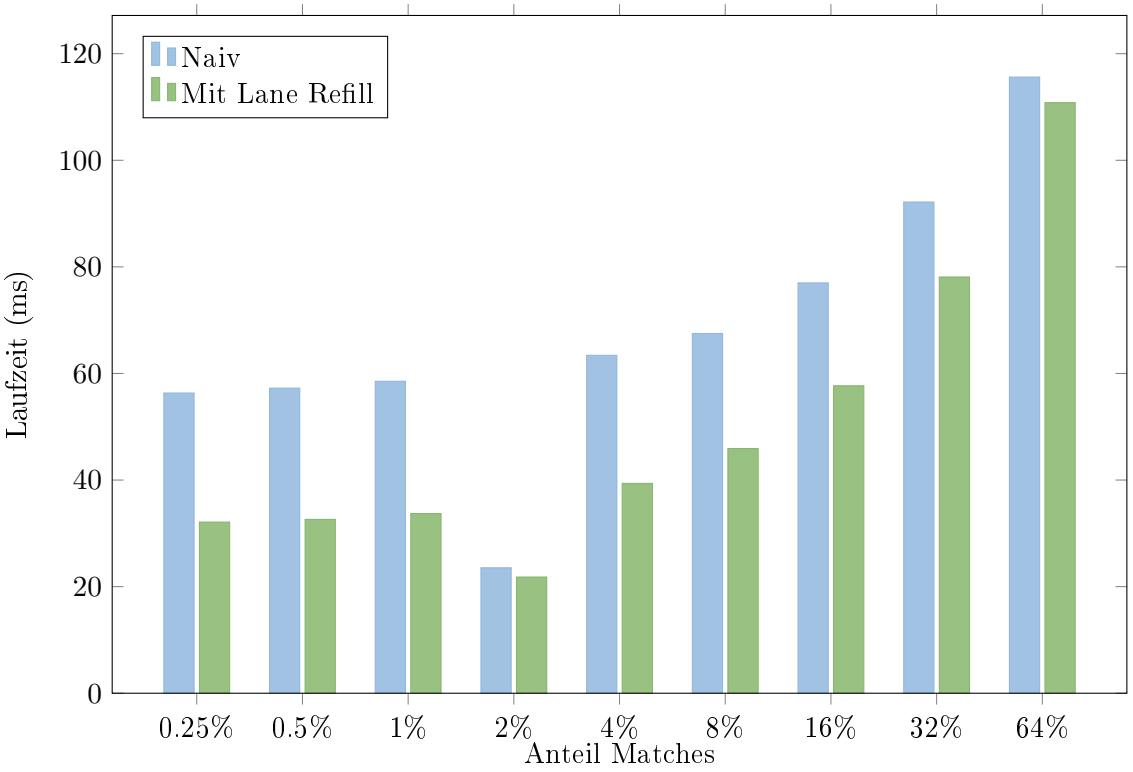

Abbildung 10.2: Laufzeit für Präfixtest mit verschiedener Verteilung beim Type-Benchmark

die Messungen für die Implementierung zur Reduzierung des Overheads des Lane Refill-Verfahrens ausgelassen.

Wie bei dem vorherigen Benchmark ist zu erkennen, dass eine höhere Anzahl von Matches hier auch eine erhöhte Laufzeit verursacht und dass die Laufzeiten der naiven Implementierung bedeutend über denen der durch das Lane Refill optimierten Umsetzung liegt. Die verbesserte Variante des Algorithmus erzielt hier eine Reduzierung der Laufzeit zwischen 4% und 43%, wobei auch hier die Differenz zwischen den beiden Verfahren geringer wird, wenn ein höherer Anteil von Matches im Datensatz vorhanden ist. Bis zu einem Anteil von 8% zutreffender Strings im Datensatz ist der Vorteil allerdings noch höher als 30%. Für einen Anteil von 2% ist wie im vorherigen Test ebenfalls ein Ausreißer zu erkennen.

10.3.3 Präfixtest mit dem DBLP-Datensatz

In Abbildung 10.3 wird schließlich das Ergebnis eines Benchmarks dargestellt, welcher den Präfixtest mit dem *DBLP*-Datensatz untersucht. Wie in den vorherigen Tests ist zu erkennen, dass ein höherer Anteil von Matches im Datensatz eine höhere Laufzeit verursacht und dass das Einführen des Lane Refill-Verfahrens eine deutliche Verringerung der Laufzeiten bewirkt. Diese Verbesserungen liegen zwischen 3% und 54%, wobei die größte Aufspaltung im mittleren Bereich der Selektivität liegt. Zwischen 1% und 16% zutreffender Elemente im Datensatz beträgt die Verbesserung mehr als 30%.

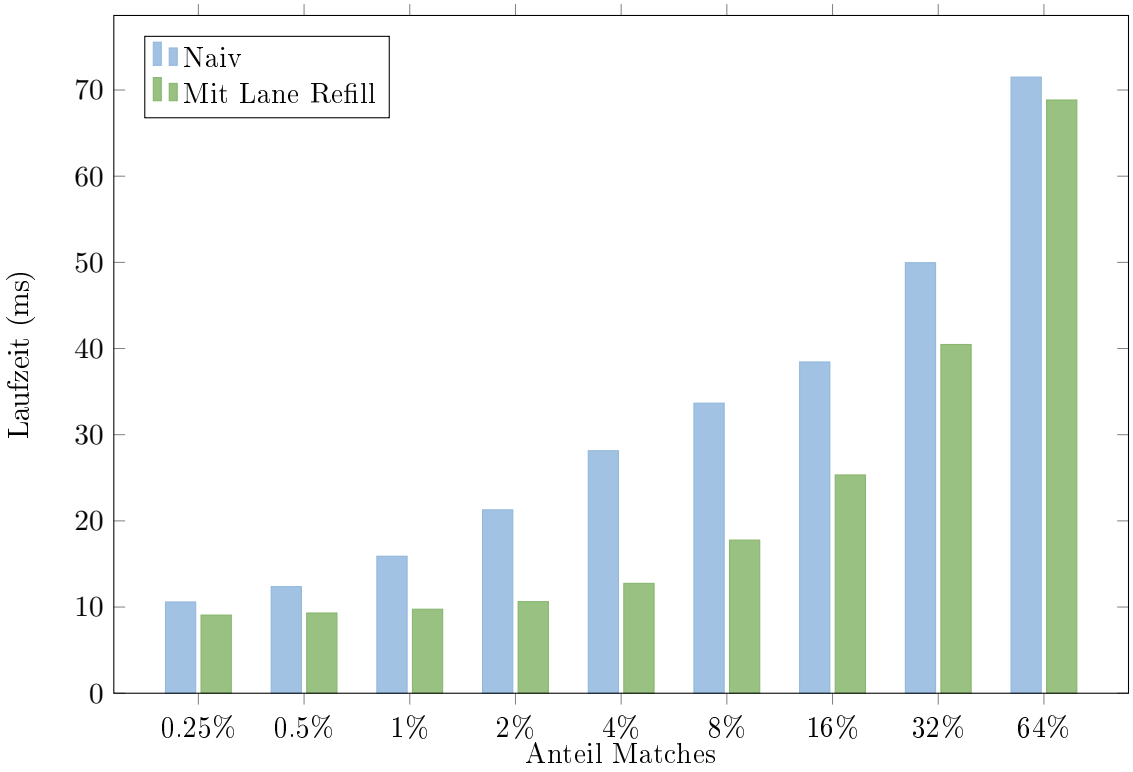

Abbildung 10.3: Laufzeit für Präfixtest mit verschiedener Verteilung beim DBLP-Benchmark

10.4 Diskussion der Ergebnisse

Die Ergebnisse der durchgeführten Tests lassen eindeutige Rückschlüsse auf das Verhalten der zuvor vorgestellten Verfahren für unterschiedliche Anwendungsfälle ziehen. Die geringeren Laufzeiten bei einer niedrigeren Anzahl von Matches lassen sich darauf zurückführen, dass viele untersuchte Strings vorzeitig verworfen werden. Somit ist bei der naiven Umsetzung die Wahrscheinlichkeit geringer, dass ein Warp bis zum Ende des Vergleichsstrings iterieren muss. Auch die verbesserte Variante profitiert davon, dass die Zeichenketten nicht vollständig durchlaufen werden müssen, sodass häufiger neue Strings nachgeladen werden können und weniger tatsächliche Arbeit zu erledigen ist.

Der durchgehend erkennbare Leistungsgewinn, der durch das Lane Refill erreicht wird, lässt erkennen, dass das Erhöhen der Auslastung der Warps eine Steigerung der Performanz mit sich bringt. Dies lässt darauf schließen, dass die Verbesserung der Auslastung einen signifikanteren Einfluss auf die Leistung hat als der durch das Verfahren zusätzlich generierte Overhead. Der geringe Einfluss des Overheads lässt sich außerdem daran erkennen, dass die in Abbildung 10.1 untersuchten Verbesserungen durch Reduzierung des Overheads des Lane Refill-Verfahrens keine signifikante Verringerung der Laufzeit erbringen. Die Speicherzugriffe, die für den eigentlichen String-Vergleich durchgeführt werden müssen, haben

somit einen bedeutend höheren Einfluss auf die Laufzeit als die Instruktionen, die für das Lane Refill ausgeführt werden.

Ebenfalls in allen Diagrammen erkennbar ist, dass die Leistungssteigerung bei einer höheren Anzahl von Matches geringer wird, was darauf zurückzuführen ist, dass bei einem hohen Anteil zutreffender Elemente keine so starke Unterauslastung bei dem naiven Algorithmus auftritt. Dies liegt daran, dass in einem Warp zu jeder Zeit eine höhere Anzahl von zutreffenden Strings geprüft und somit die Zahl der wartenden Lanes geringer wird. Es lässt sich also in diesem Fall gar keine so große Verbesserung mehr durch das Einführen des Lane Refill erreichen, da keine große Unterauslastung vorhanden ist, die es zu beseitigen gilt.

Der Ausreißer bei einem Anteil von 2% zutreffender Elemente im *Type*-Datensatz tritt vermutlich aufgrund eines Fehlers innerhalb des Datensatzes auf. Es wurde versucht, diesen zu beheben, indem der Datensatz analysiert, überprüft und neu generiert wurde, allerdings ließ sich keine Lösung finden, welche den Ausreißer beseitigt hätte. Die restlichen Datenpunkte behalten dennoch ihre Gültigkeit und an ihnen lässt sich das generelle Verhalten der Verfahren einwandfrei ablesen.

Kapitel 11

Evaluation des parallelen Musterabgleichs

Nachdem im letzten Kapitel gezeigt wurde, dass das Lane Refill-Verfahren eine erhöhte Leistungsfähigkeit des einfachen String-Vergleichs erreicht, bleibt nun zu untersuchen, wie sich die vorgestellten Methoden zum parallelen Musterabgleich damit verhalten. Zunächst soll geklärt werden, welcher der vorgestellten Algorithmen ohne tiefgreifende Verbesserungen die beste Laufzeit erzielen kann. Im Anschluss wird untersucht, welcher Algorithmus am meisten vom Lane Refill profitiert und ob sich die Beobachtungen für unterschiedlich geartete reguläre Ausdrücke ändert. Die Testumgebung ist dabei identisch zu dem im Kapitel 10.1 beschriebenen Aufbau.

11.1 Verwendete Workloads und deren Merkmale

Für die ersten Tests wurde der in 10.2 vorgestellte DBLP-Datensatz verwendet. Dieser eignet sich optimal zur Untersuchung der Laufzeitverbesserung durch das Lane Refill, da der analytische Workload mit variabler Selektivität Analysen zu verschiedenen Stufen der Unterauslastung ermöglicht. Außerdem bietet sich dieser Datensatz an, da ein Vergleich zwischen dem einfachen Präfixtest und dem Präfixtest mittels regulärer Ausdrücke durchgeführt werden kann.

Ein weiterer Workload wurde dem TPC-H-Benchmark entnommen, in dem eine Selektion über die Spalte *Name* der Relation *Products* durchgeführt wird. Die enthaltenen Strings haben eine Länge zwischen 26 und 43 Zeichen und bestehen aus den Kleinbuchstaben *a*-*z* und dem Leerzeichen. Der Datensatz wurde einige Male repliziert, sodass er 46.000.000 Tupel enthält.

11.2 Vorstellung der Messergebnisse

Abbildungen 11.1 bis 11.5 zeigen die Laufzeiten der verschiedenen Algorithmen in den entwickelten Benchmarks. Für ein einfacheres Verständnis werden besondere Auffälligkeiten dazu aufgezeigt und im nächsten Abschnitt analysiert.

11.2.1 Vergleich der Basisalgorithmen mit dem DBLP-Datensatz

Bei der ersten Messung soll überprüft werden, wie sich die in Kapitel 7 vorgestellten Verfahren zur Durchführung eines einfachen Musterabgleichs ohne tiefgreifende Verbesserungen wie dem Lane Refill-Verfahren verhalten. Untersucht werden hier die in Kapitel 7.2 beschriebene *Flat*-Variante des von Ragel generierten Algorithmus, die in Kapitel 7.3 vorgestellte *Table*-Variante dieses Verfahrens, die dort ebenfalls angesprochene *LIKE*-Operation, welche aus DogQC entnommen wurde, und die in Kapitel 4.2 vorgestellte Implementierung zum einfachen Präfixtest. Der untersuchte reguläre Ausdruck passt auf eine bestimmte Zeichenfolge am Anfang jedes Strings und lässt keine Zeichen vor diesem Suchstring zu, danach sind dagegen beliebige Strings zulässig.

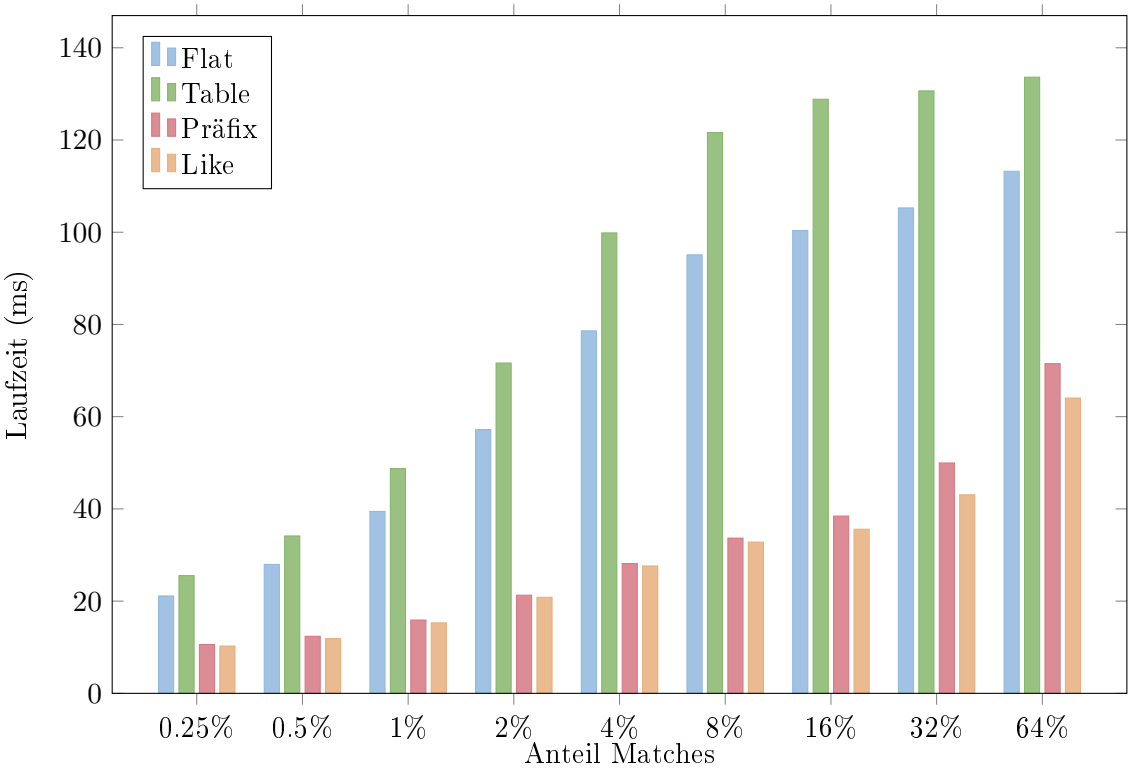

Abbildung 11.1: Laufzeit für Präfixtest mit Basisalgorithmen für den DBLP-Datensatz

Bei den in Abbildung 11.1 dargestellten Ergebnissen fällt zunächst auf, dass die Ausführungszeit aller Algorithmen bei einem größeren Anteil Matches monoton ansteigt. Die beiden Implementierungen, die reguläre Ausdrücke verwenden, werden ab einem Anteil von

8% kaum noch langsamer, wohingegen die Geschwindigkeit der anderen beiden Verfahren dabei immer schneller sinkt. Die *Flat*-Variante ist zwischen 15% und 22% schneller als die *Table*-Variante der Ragel-Umsetzung, wobei der Unterschied für mittelgroße Selektivitäten am größten ist. Der *LIKE*-Operator aus DogQC ist zwischen 1% und 13% schneller als der Präfixtest, wobei der Unterschied für einen höheren Anteil Matches größer wird. Der Präfixtest ist bei einer Selektivität von unter 32% zwischen 50% und 65% schneller als die *Flat*-Variante. Bei einem Anteil Matches von 64% beträgt dieser Unterschied 37%.

11.2.2 Verbesserung der Algorithmen durch das Lane Refill

Die nachfolgende Untersuchung dient dazu, den Einfluss des Lane Refill auf die Varianten *Flat* und *Table* des regulären Musterabgleichs sowie den einfachen Präfixtest zu analysieren. Der hier verwendete reguläre Ausdruck ist derselbe wie im vorherigen Abschnitt.

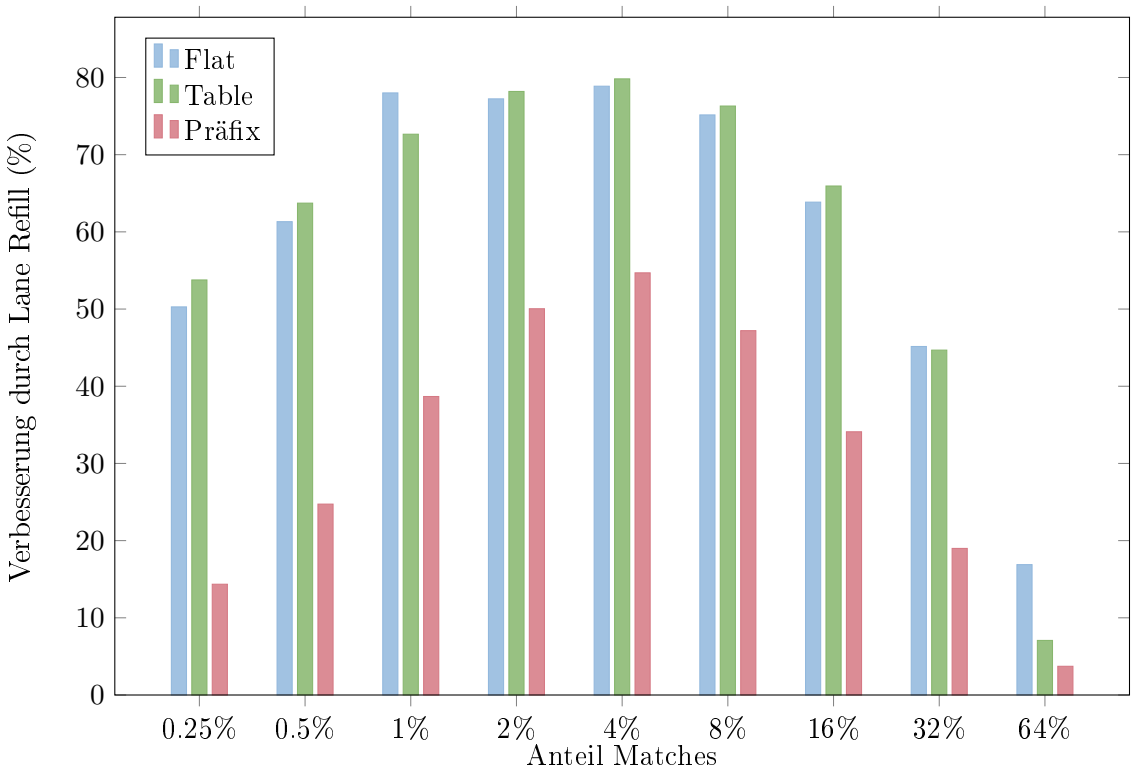

Abbildung 11.2: Verbesserungen der Algorithmen durch das Lane Refill für den DBLP-Datensatz

Abbildung 11.2 zeigt dazu die Verbesserung der Laufzeit, welche durch das Einbinden des Lane Refill-Verfahrens in die Basisalgorithmen erreicht wird. Bei den beiden Verfahren, die auf dem regulären Musterabgleich basieren, ist eine bedeutend größere Verbesserung zu sehen als bei dem Präfixtest. Beide auf Ragel basierenden Algorithmen werden in ähnlichem Maße durch das Lane Refill verbessert, wobei die *Flat*-Variante etwas mehr davon profitiert. Bis zu einer Selektivität von 16% beträgt die Verbesserung der beiden Verfahren für den Musterabgleich mehr als 50% gegenüber der ursprünglichen Laufzeit. Die Verbesserung

aller untersuchter Algorithmen ist für einen mittelgroßen Anteil Matches am größten und fällt besonders für größere Anteile stark ab.

11.2.3 Vergleich der optimierten Algorithmen

Schließlich werden die absoluten Laufzeiten der durch das Lane Refill verbesserten Verfahren untereinander und mit dem *LIKE*-Operator verglichen. Der reguläre Ausdruck bleibt derselbe wie in den letzten beiden Abschnitten beschrieben.

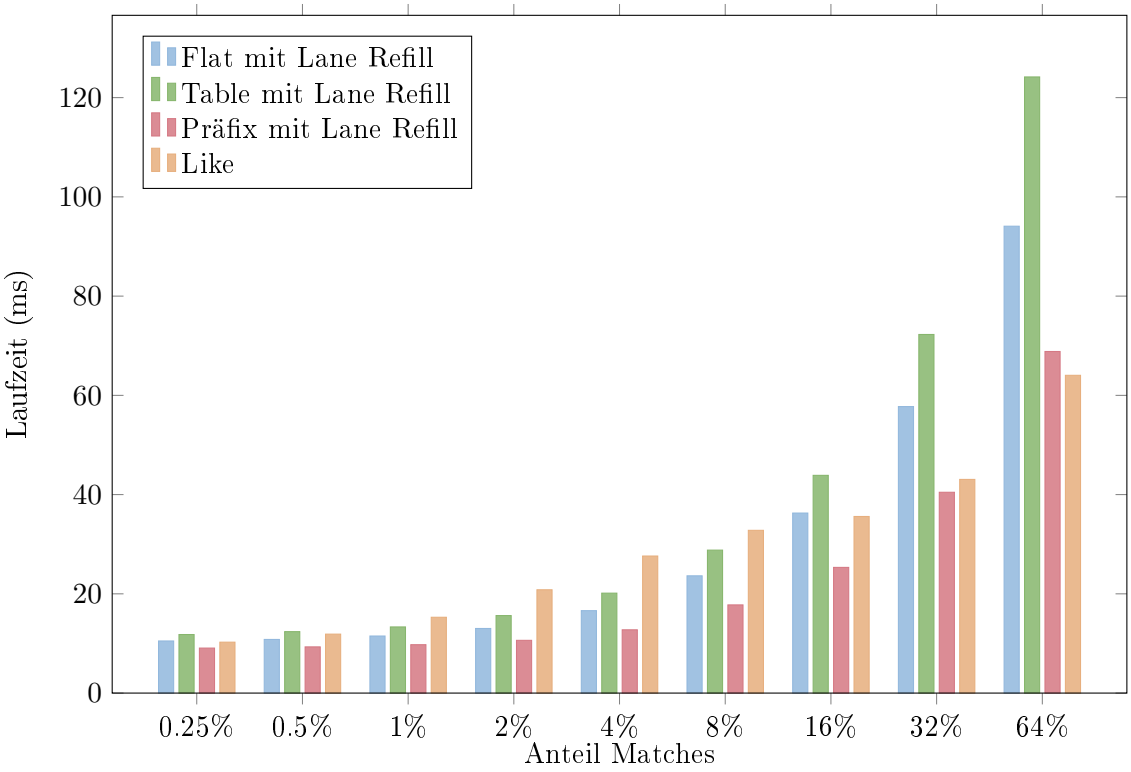

Abbildung 11.3: Laufzeiten der optimierten Algorithmen für den DBLP-Datensatz

Die in Abbildung 11.3 dargestellten Laufzeiten aller Algorithmen steigen für einen höheren Anteil von Matches monoton an. Im Allgemeinen liefert der durch das Lane Refill verbesserte Präfixtest die beste Laufzeit, was nur nicht für eine Selektivität von 64% gilt. Für eine steigende Anzahl passender Matches steigen die gepufferten Operationen besonders stark an, weshalb die ungepufferte *LIKE*-Implementierung für hohe Selektivitäten eine vergleichsweise geringe Laufzeit bietet, obwohl diese für niedrige Selektivitäten zu den langsamsten gehört. Die *Flat*-Variante des regulären Musterabgleichs bietet stets eine geringere Laufzeit als die entsprechende *Table*-Variante.

11.2.4 Einfluss beliebiger Anfangszeichen in dem DBLP-Datensatz

Als nächstes wird untersucht, wie sich die Laufzeiten der Algorithmen entwickeln, wenn beliebige Zeichen vor dem gesuchten String stehen dürfen. Der Datensatz bleibt dazu identisch zu den vorherigen Tests, es wird lediglich der reguläre Ausdruck dementsprechend erweitert, dass dieser beliebige Zeichen vor dem Suchstring zulässt. Diese Modifikation führt dazu, dass der Präfixtest an dieser Stelle nicht mehr untersucht werden kann, da dieser keine beliebigen Zeichen unterstützt.

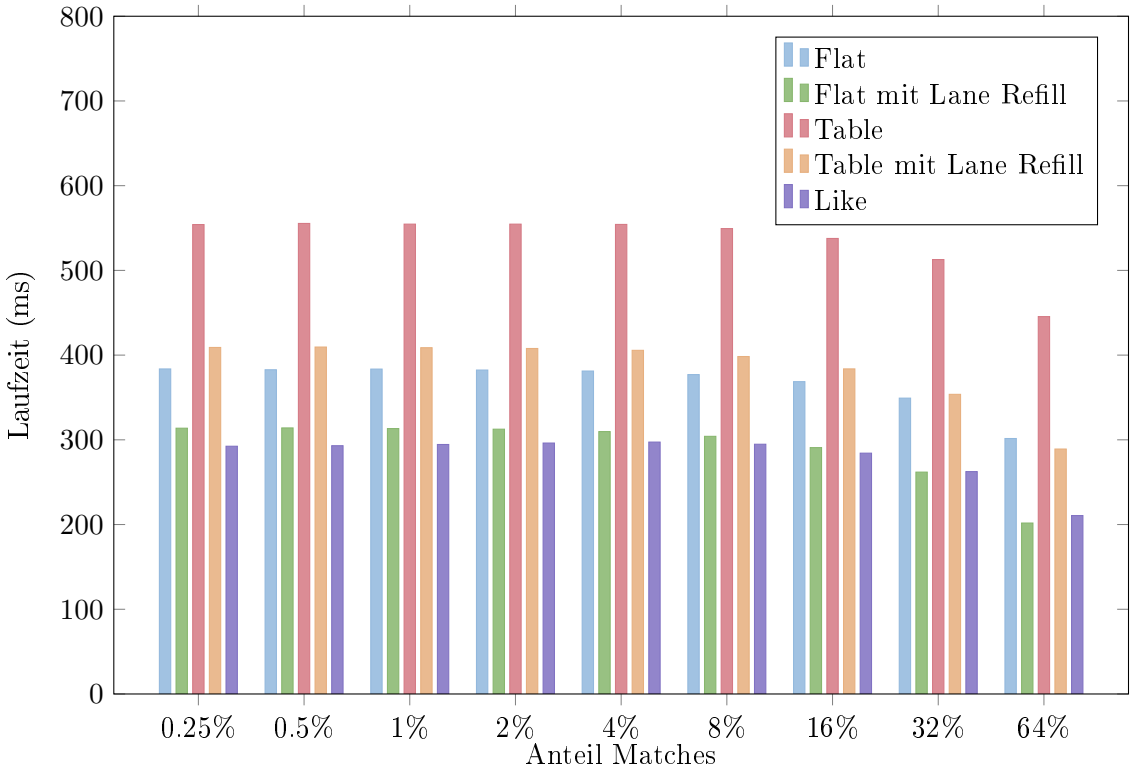

Abbildung 11.4: Laufzeiten für Benchmark, der beliebige Anfangszeichen zulässt

In Abbildung 11.4 ist zu erkennen, dass die Ausführungszeiten aller Algorithmen bis zu einer Selektivität von ungefähr 8% unabhängig von der Anzahl der Matches im Ergebins sind. Je höher die Selektivität wird, desto geringer wird die Ausführungszeit. Beide Algorithmen, welche die *Flat*-Einstellung von Ragel verwenden, sind im Allgemeinen schneller als die entsprechenden Gegenstücke mit der *Table*-Einstellung. Am schnellsten ist hier die *LIKE*-Operation aus DogQC. Die Verbesserung der *Flat*-Variante des Musterabgleichs durch das Lane Refill beträgt zwischen 26% und 35% und wird für eine höhere Anzahl von Matches größer. Für die *Table*-Variante beträgt die Verbesserung zwischen 17% und 33% und wird ebenfalls für höhere Anzahlen von Matches größer.

11.2.5 Vergleich verschiedener Automatengrößen mit dem TPC-H-Datensatz

Zum Schluss wird untersucht, welchen Einfluss die Größe des Automaten auf die beiden Algorithmen zum parallelen Musterabgleich hat. Dazu wurde eine Selektion über die *Name*-Spalte der *Products*-Relation aus dem TPC-H-Datensatz durchgeführt, wobei die Größe des Automaten variiert wurde. Zu diesem Zweck wurde ein regulärer Ausdruck erstellt, welcher ein Suchwort annimmt und beliebige Zeichen vor und nach diesem Wort zulässt Die Selektivität dieses Ausdrucks beträgt etwa 11%. Um die Automatengröße zu erhöhen, wurden weitere Suchwörter, welche in dieser Form nicht im Datensatz enthalten sind, mit dem bestehenden Ausdruck *oder*-verknüpft. Dadurch wird ein größerer Automat bei gleichbleibender Selektivität erreicht.

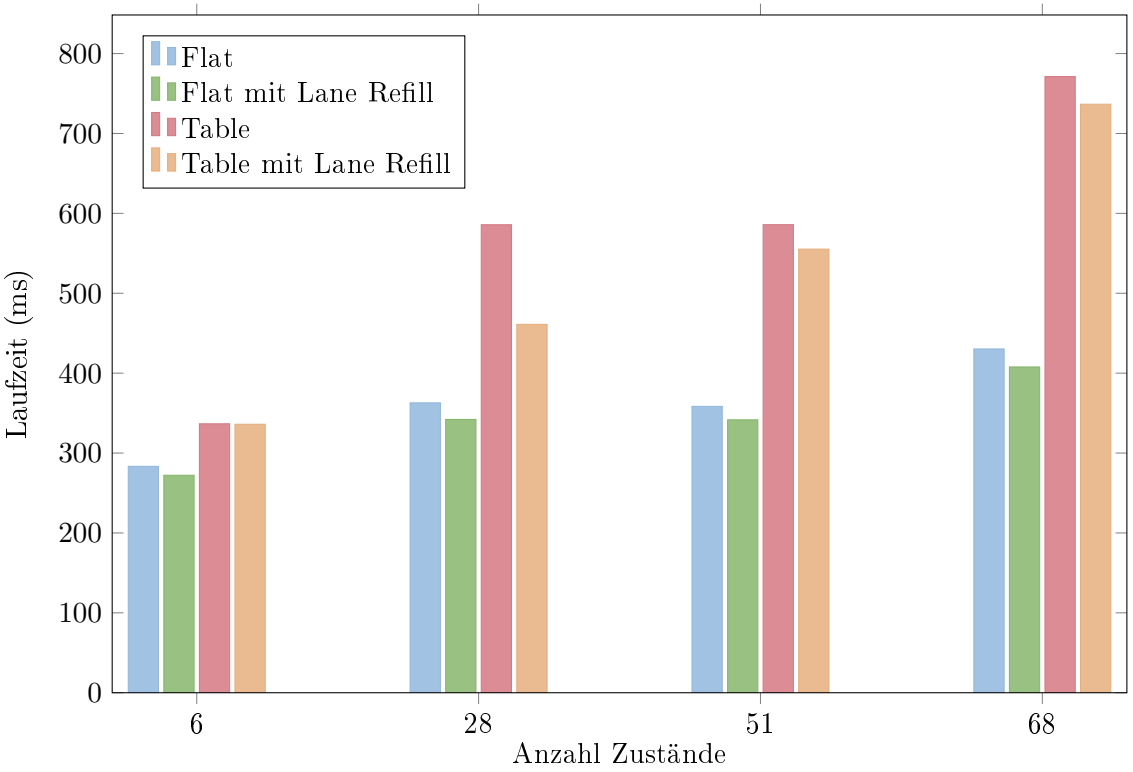

Abbildung 11.5: Laufzeiten für unterschiedliche Automatengrößen mit dem TPC-H Datensatz

Abbildung 11.5 zeigt, dass die Ausführungszeit aller Algorithmen für größere Automaten ansteigt. Der Unterschied zwischen 28 und 51 Zuständen fällt dabei recht klein aus, wohingegen die anderen beiden Schritte einen signifikanteren Unterschied zeigen. Eine Ausnahme bildet der Musterabgleich mit der *Table*-Einstellung, der auch zwischen 28 und 51 Zuständen einen signifikanten Anstieg zeigt. Die *Flat*-Variante des Algorithmus ist in jedem Falle schneller als die *Table*-Variante, wobei der Unterschied für größere Automaten größer wird. Die Verbesserung durch das Einführen des Lane Refill-Verfahrens für den *Flat*-Algorithmus beträgt zwischen 4% und 6%. Der *Table*-Algorithmus profitiert vom

Lane Refill mit einer Verbesserung zwischen 0% und 5% mit einem Ausreißer von 20% bei 28 Zuständen.

11.3 Diskussion der Ergebnisse

Die durchgeführten Untersuchungen ergeben ein einheitliches Bild zu dem Verhalten der vorgestellten Algorithmen und deren Verbesserungen durch das Lane Refill-Verfahren. In den ersten drei Abschnitten wird ein Präfixtest untersucht, welcher sich für den regulären Musterabgleich und den *LIKE*-Operator ähnlich wie für die zuvor beschriebene einfache Umsetzung des Präfixtests verhält. Die geringe Laufzeit für eine niedrige Anzahl von Matches ist darin begründet, dass viele Zeichenketten vorzeitig verworfen werden können. Für das Durchlaufen eines Automaten bedeutet dies, dass dieser in einen Fehlerzustand gerät und die Lane inaktiv wird. Der *LIKE*-Operator verhält sich in diesem Fall wie der einfache Präfixtest. Dadurch, dass viele Zeichenketten vorzeitig verworfen werden, ist bei den Basisalgorithmen die Wahrscheinlichkeit geringer, dass in einem Warp ein Match vorhanden ist und er oftmals nicht vollständig abgearbeitet werden muss. Innerhalb der durch das Lane Refill intern gepufferten Operationen ist für geringe Selektivitäten die Anzahl der Matches pro Warp geringer, weshalb öfter dynamisch Tupel nachgeladen werden können als für hohe Selektivitäten. Für eine besonders geringe Anzahl von Matches erreicht das Lane Refill allerdings nicht die optimale Verbesserung, wie in Abbildung 11.2 zu erkennen ist. Dies liegt daran, dass in diesem Bereich nur vereinzelte Warps überhaupt Tupel verarbeiten müssen, die einen Match darstellen, weshalb wiederum keine große Unterauslastung auftritt, die durch das Lane Refill beseitigt werden könnte.

Für die Basisalgorithmen kann beobachtet werden, dass die mächtigen Verfahren zum regulären Musterabgleich eine weitaus geringere Performanz bieten als die funktionsärmeren Präfix- oder *LIKE*-Operationen. Dies kann damit begründet werden, dass das Durchlaufen eines Automaten mehr Instruktionen und Speicherzugriffe erfordert als das einfache Vergleichen zweier Zeichen, wie es bei den simpleren Operationen durchgeführt wird. Die Untersuchung aus Abschnitt 11.2.2 zeigt, dass das Erhöhen der Auslastung der Warps durch das Lane Refill einen signifikanten Laufzeitgewinn erbringt, wodurch vor allem die Algorithmen, die mit regulären Ausdrücken arbeiten, profitieren und näher an die geringere Laufzeit der simpleren Operationen heran rücken. Für einen großen Anteil von Matches wird die Verbesserung durch das Lane Refill geringer, was daran liegt, dass in den naiven Verfahren gar keine starke Unterauslastung mehr auftritt. Die Warps sind grundsätzlich höher ausgelastet, da ein höherer Anteil Matches besteht und somit seltener der Fall eintritt, dass nur wenige Lanes arbeiten, während die restlichen Lanes auf deren Fertigstellung warten. Es lässt sich also durch das Lane Refill kein so großer Vorteil mehr erreichen.

Werden, wie in Abschnitt 11.2.4 untersucht, beliebige Zeichen vor dem Suchstring im regulären Ausdruck erlaubt, kann eine weitgehend konstante Laufzeit für alle Algorithmen

beobachtet werden. Dies liegt daran, dass immer die gesamte Zeichenkette durchlaufen werden muss, da der Suchstring an einer beliebigen Stelle vorkommen kann. Die Tatsache, dass die Laufzeit in diesem Experiment für eine höhere Anzahl von Matches geringer wird, ist darin begründet, dass die in den Datensatz eingesetzten Strings, welche ein Match ergeben, eine unterdurchschnittliche Länge besitzen und somit insgesamt weniger Zeichen verarbeitet werden müssen. Durch die Einführung des Lane Refill wird hier ein gewisser Laufzeitgewinn erreicht, da die String-Längen im Datensatz unterschiedlich sind. Daher werden einige Lanes, die ihren String bereits abgearbeitet haben, früher inaktiv als andere und können dynamisch neue Tupel nachladen. Bei der Verwendung des TPC-H-Datensatzes in Abschnitt 11.2.5 zeigt sich nur eine geringfügige Laufzeitverbesserung durch das Lane Refill, was daran liegt, dass die Strings alle ähnliche Längen haben und aufgrund der beliebigen Anfangszeichen vollständig durchlaufen werden müssen.

Die Laufzeit der *Flat*-Einstellung von Ragel ist sowohl mit als auch ohne Lane Refill schneller als die *Table*-Einstellung. Dies ist ein Indikator dafür, dass die Größe des verwendeten Alphabets nicht zu groß für die *Flat*-Variante ist und die Verwendung einer binären Suche wie sie in der *Table*-Einstellung zu finden ist, nicht nötig ist, sondern zu Laufzeiteinbußen durch den erhöhten Overhead führt. Auch die Größe des Automaten ändert an dieser Beobachtung nichts, da auch im Abschnitt 11.2.5 zu erkennen ist, dass die einfache Flat-Variante in jedem Falle schneller ist.

Kapitel 12

Fazit

Um die Arbeit abzuschließen, soll in diesem Kapitel ein Überblick über die Ergebnisse der Arbeit für praktische Anwendungsfälle gegeben werden und außerdem ein Ausblick auf Anwendungen des Verfahrens und weiterführende Forschungen zusammengetragen werden.

12.1 Ergebnis der Arbeit

Als Basis für die späteren Untersuchungen wurde eine Umsetzung des einfachen String-Vergleichs, eines Präfixtests und eines parallelen Musterabgleichs mit regulären Ausdrücken im Kontext von kompilierten Anfragepipelines auf GPUs vorgestellt. Diese Operationen sind so strukturiert, dass sie leicht in den Query Compiler DogQC übernommen werden können und damit dessen Funktionalität erweitern. Die Algorithmen wurden analysiert und dadurch Engpässe durch eine Unterauslastung der Warps erkannt, welche durch den Einsatz des Lane Refill-Verfahrens beseitigt wurden.

Durch die Verbesserung der Auslastung wurde eine Leistungssteigerung erzielt, welche in zahlreichen Tests beobachtet werden kann. Das Lane Refill zeigt sich als hervorragendes Verfahren, um die Leistungsfähigkeit von Algorithmen, die mit String-Daten auf Grafikkarten arbeiten, zu erhöhen. In jedem der durchgeführten Tests erzielte das Lane Refill eine Verbesserung, welche in einigen Fällen einen Leistungsvorteil von bis zu 80% erreichte.

12.2 Ausblick

Da durch das hier untersuchte Verfahren die Leistung verschiedenster String-Operationen verbessert werden konnte, entfällt in einigen Anwendungsfällen die Notwendigkeit, ein Dictionary zu verwenden, um eine akzeptable Leistung erzielen zu können. Dadurch, dass direkt auf den String-Daten gearbeitet wird, entfällt der Aufwand, eine weitere Datenstruktur pflegen zu müssen, und es wird eine allgemein effizientere Verarbeitung verschiedener Arbeitslasten erreicht.

Datenbankmanagementsysteme, die Grafikkarten als Coprozessoren zur Leistungssteigerung verwenden, können vom Lane Refill-Verfahren profitieren, da diverse String-Operationen und besonders die Verarbeitung von regulären Ausdrücken effektiver durchgeführt werden können. Auch ohne die Verwendung von kompilierten Anfragepipelines kann durch dieses Verfahren ein Geschwindigkeitsvorteil erreicht werden, was in folgenden Arbeiten weiter analysiert werden könnte.

Andere Datenbanksysteme, die noch keine Unterstützung für Grafikprozessoren bieten, könnten um diese erweitert werden, um bestimmte Operationen auf der hochgradig parallelen Hardware effizienter ausführen zu können. Dabei kann eine zusätzliche Leistungssteigerung durch das Lane Refill dabei helfen, diese Lösung profitabel gegenüber der Berechnung der Operationen auf der CPU zu machen. Ist keine GPU-Unterstützung möglich, kann die Verwendung der SIMD-Fähigkeiten moderner Prozessoren zur parallelen Datenverarbeitung ebenfalls eine Leistungssteigerung bieten. Vergangene Arbeiten zeigen bereits, dass auch in diesem Falle eine Leistungssteigerung durch das Lane Refill erreicht werden kann [7].

Neben der Verwendung des Lane Refill für String-Daten könnte durch das Verfahren auch eine Steigerung des Durchsatzes anderer Operationen, die auf heterogenen Datensätzen arbeiten, erzielt werden. Die bisherige Implementierung von DogQC verwendet das Verfahren beispielsweise bereits für die parallele Verarbeitung der JOIN-Operation.

Anhang A

Umsetzung der String-Selektion mit Lane Refill

```
1  // shared memory for the divergence buffers
2  __shared__ int search_id_divergence_buffer[THREAD_COUNT];
3  __shared__ int current_divergence_buffer[THREAD_COUNT];
4
5  unsigned warpid = (threadIdx.x / 32); // index of warp in block
6  unsigned bufferbase = (warpid * 32);    // buffer offset for warp in block
7  unsigned warplane = (threadIdx.x % 32); // index of lane in warp
8  unsigned prefixlanes = (0xffffffff >> (32 - warplane)); // previous lanes
9  int bufferelements = 0;         // number of elements in buffer
10
11 while(!flush_pipeline) {
12   current = loop_var;
13
14   /* execute previous operators in the pipeline */
15
16   data_length = char_offset[current+1] - char_offset[current] - 1;
17
18   // if string lengths are unequal, discard
19   if (active && data_length != search_length)
20     active = false;
21
22   int numactive = __popc(__ballot_sync(ALL_LANES, active));
23   while(bufferelements + numactive > THRESHOLD) {
24
25     // refill empty lanes from buffer in case of underutilization
26     if (numactive < THRESHOLD) {
27       numRefill = min(32 - numactive, bufferelements);
28       numRemaining = bufferelements - numRefill;
29
30       previous_inactive = __popc(~__ballot_sync(ALL_LANES, active) &
             prefixlanes);
```

```
31
32        if (!active && previous_inactive < bufferelements) {
33          buf_ix = numRemaining + previous_inactive + bufferbase;
34          search_id = search_id_divergence_buffer[buf_ix];
35          current = current_divergence_buffer[buf_ix];
36          active = true;
37        }
38
39        bufferelements -= numRefill;
40      }
41
42      int data_id = search_id + char_offset[current];
43
44      // when strings don't match, inactivate the lane
45      if (active && data_content[data_id] != search_string[search_id])
46        active = false;
47
48      search_id++;
49
50      if (search_id == search_length) {
51
52        /* execute following operators in the pipeline */
53
54        active = false;
55      }
56
57      numactive = __popc(__ballot_sync(ALL_LANES, active));
58    }
59
60    // flush active lanes to buffer
61    if (numactive > 0) {
62      previous_active = __popc(__ballot_sync(ALL_LANES, active) & prefixlanes
          );
63      buf_ix = bufferbase + bufferelements + previous_active;
64
65      if(active) {
66        search_id_divergence_buffer[buf_ix] = character_index;
67        current_divergence_buffer[buf_ix] = current;
68      }
69
70      bufferelements += numactive;
71      active = false;
72    }
73
74    loop_var += step;
75  }
```

Listing A.1: Umsetzung der String-Selektion mit Lane Refill

Anhang B

Umsetzung des Musterabgleichs mit Lane Refill

```
1  // shared memory for the divergence buffers
2  __shared__ char* p_divergence_buffer[BLOCK_SIZE];
3  __shared__ char* pe_divergence_buffer[BLOCK_SIZE];
4  __shared__ int cs_divergence_buffer[BLOCK_SIZE];
5
6  unsigned warpid = (threadIdx.x / 32);
7  unsigned bufferbase = (warpid * 32);
8  unsigned warplane = (threadIdx.x % 32);
9  unsigned prefixlanes = (0xffffffff >> (32 - warplane));
10 unsigned bufferelements = 0;
11
12 while (!flush_pipeline) {
13
14   /* execute previous operators in the pipeline */
15
16   char *p = book_content + character_offset[loop_var];
17   char *pe = book_content + character_offset[loop_var + 1];
18
19   int cs = machine_start;
20
21   // if end of string is reached, inactivate the lane
22   if (p == pe)
23     active = false;
24
25   int numactive = __popc(__ballot_sync(ALL_LANES, active));
26   while(bufferelements + numactive > THRESHOLD) {
27
28     // refill empty lanes from buffer in case of underutilization
29     if (numactive < THRESHOLD) {
30       numRefill = min(32 - numactive, bufferelements);
31       numRemaining = bufferelements - numRefill;
```

```
32
33      previous_inactive = __popc(~__ballot_sync(ALL_LANES, active) &
            prefixlanes);
34
35      if (!active && previous_inactive < bufferelements) {
36        buf_ix = numRemaining + previous_inactive + bufferbase;
37        p = p_divergence_buffer[buf_ix];
38        pe = pe_divergence_buffer[buf_ix];
39        cs = cs_divergence_buffer[buf_ix];
40        active = true;
41      }
42
43      bufferelements -= numRefill;
44    }
45
46    if (active) {
47      cs = singleDfaStep(cs, p);
48
49      if (cs == 0 && !STARTS_WITH_ANY)    // invalid state reached
50        active = false;
51    }
52
53    p++;
54
55    if (active && p == pe) {    // string completely processed
56      if (cs >= machine_first_final) {    // finishes with accepting state
57
58        /* execute following operators in the pipeline */
59
60      } else {    // finishes with non accepting state
61        active = false;
62      }
63    }
64
65    numactive = __popc(__ballot_sync(ALL_LANES, active));
66  }
67
68  // flush active lanes to buffer
69  if (numactive > 0) {
70    previous_active = __popc(activemask & prefixlanes);
71    buf_ix = bufferbase + bufferelements + previous_active;
72
73    if(active) {
74      p_divergence_buffer[buf_ix] = p;
75      pe_divergence_buffer[buf_ix] = pe;
76      cs_divergence_buffer[buf_ix] = cs;
77    }
78
```

```
79      bufferelements += numactive;
80      active = false;
81    }
82
83    loop_var += step;
84  }
```

Listing B.1: Umsetzung des parallelen Musterabgleichs mit Lane Refill

Anhang C

Laufzeiten für alternative Selektivität des Type-Datensatzes

Grid Size \ Block Size	32	64	96	128	160	192	224	256	384	512	640	768	896	1024
10	2101	1053	725	543	459	386	344	301	217	176	154	146	142	140
12	1750	877	605	454	384	321	288	253	185	153	145	142	139	139
14	1499	779	541	410	353	305	268	233	173	149	143	201	200	184
16	1312	683	474	359	309	268	236	206	155	144	139	186	172	166
18	1166	607	422	320	276	239	211	185	147	141	139	173	155	150
20	1051	547	387	303	258	220	199	177	146	140	195	155	151	147
30	714	384	278	213	186	163	150	145	139	180	142	157	151	148
40	547	303	225	174	154	146	143	139	169	158	165	159	152	152
60	384	213	164	145	144	139	194	179	139	145	152	141	139	139
80	305	175	147	140	187	168	153	155	149	151	146	144	141	143
100	257	153	141	192	162	144	142	164	150	152	142	148	145	144
150	186	141	190	146	167	148	144	158	139	147	144	145	141	140
200	253	190	149	166	142	152	143	154	143	144	144	142	139	140
300	189	148	152	156	154	139	143	148	139	140	140	141	141	139
400	205	166	158	153	142	142	143	143	142	140	141	142	140	140
500	173	143	139	148	139	144	144	142	143	139	139	141	139	141
1000	174	151	140	142	139	140	140	139	140	139	139	141	139	139
2000	160	142	139	140	139	140	140	139	139	142	139	141	139	139
3000	156	140	139	140	142	139	139	139	139	139	139	141	139	139
4000	156	139	139	139	139	142	139	139	139	139	140	141	139	139
6000	157	140	139	139	139	139	139	139	139	139	139	141	139	139
8000	155	139	139	139	139	139	142	139	139	139	139	143	139	139
10000	154	140	139	139	139	139	139	139	139	139	139	141	139	139
20000	153	139	139	139	139	139	139	141	139	139	139	142	143	139
50000	154	139	139	139	139	139	139	139	139	139	139	143	141	140
100000	154	139	139	139	139	139	139	139	141	139	140	144	143	144
150000	154	140	140	139	139	139	139	139	139	140	141	146	145	144
200000	154	139	139	140	139	139	139	139	140	143	143	148	147	147

Abbildung C.1: Laufzeit des String-Vergleichsalgorithmus in ms für den Type-Datensatz mit einer Selektivität von 64% unter Verwendung unterschiedlicher Grid-Konfigurationen

Abbildungsverzeichnis

Literatur

[1] Peter A. Boncz, Thomas Neumann und Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark". In: *TPCTC*. 2013.

[2] Henning Funke u. a. "Pipelined Query Processing in Coprocessor Environments". In: *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018.

[3] Goetz Graefe. "Volcano - An Extensible and Parallel Query Evaluation System". In: *IEEE Transactions On Knowledge And Data Engineering*. 1994.

[4] Mark Harris. "An Even Easier Introduction to CUDA". In: *NVIDIA Developer Blog*. 2017.

[5] Mark Harris. "Using Shared Memory in CUDA C/C++". In: *NVIDIA Developer Blog*. 2013.

[6] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. "Einführung in die Automatentheorie, Formale Sprachen und Komplexität". 2002.

[7] Harald Lang u. a. "Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines". In: *Proceedings of the 14th International Workshop on Data Management on New Hardware*. 2018.

[8] Yuan Lin und Vinod Grover. "Using CUDA Warp-Level Primitives". In: *NVIDIA Developer Blog*. 2018.

[9] Ingo Müller u. a. "Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems". In: *International Conference on Extending Database Technology*. 2014.

[10] Thomas Neumann. "Efficiently Compiling Query Plans for Modern Hardware". In: *Proceedings of the VLDB Endowment*. 2011.

[11] John Nickolls und David Kirk. "Graphics and Computing GPUs". In: *Computer Organization and Design: The Hardware/Software Interface*. 2009.

[12] NVIDIA. "NVIDIA GeForce GTX 980 - Featuring Maxwell, The Most Advanced GPU Ever Made". *Whitepaper*. 2014.

[13] Angela Pohl u. a. "An Evaluation of Current SIMD Programming Models for C++". In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. 2016.

[14] Adrian Thurston. "Ragel State Machine Compiler". *Benutzerhandbuch*. 2009.

[15] Mihai Varga. "Just-in-time compilation in MonetDB with Weld". *Masterarbeit an der Univerität Amsterdam*. 2018.

[16] Vasily Volkov. "Understanding Latency Hiding on GPUs". *Dissertation an der University of California in Berkley*. 2016.

[17] Henry Wong u. a. "Demystifying GPU Microarchitecture through Microbenchmarking". In: *IEEE International Symposium on Performance Analysis of Systems & Software*. 2010.

[18] Xiaodong Yu und Michela Becchi. "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space". In: *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2013.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 13. Juni 2019

Florian Lüdiger