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

1.1 Motivation und Hintergrund

1.2 Aufbau der Arbeit

Kapitel 2

Grundlagen der GPU-Programmierung

Um die in dieser Arbeit vorgestellten Herausforderungen bei der Verarbeitung von String-Daten mit Grafikprozessoren, nachfolgend auch GPU 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 andere GPU 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 hohen Anzahl 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 GTX950, 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.

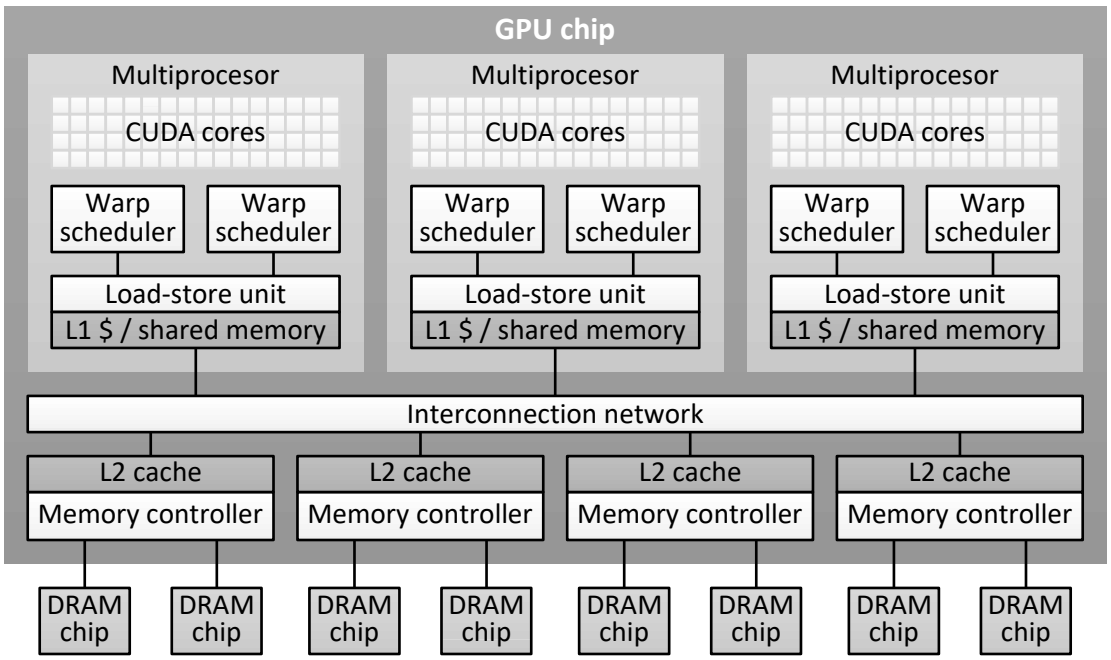


Abbildung 2.1: Architektur einer GPU [7]

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 GTX950 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 einige Unterstützungselektronik, sodass diese logisch gesehen ebenfalls unabhängig voneinander arbeiten können. [6] Bei sechs Streaming Multiprocessors mit jeweils vier Blöcken und 32 Recheneinheiten pro Block besitzt die GTX950 also 768 Kerne, welche über eine Programmierschnittstelle angesprochen werden können.

2.2 Scheduling auf GPU

Um die hohe Anzahl von Kernen innerhalb einer GPU effizient mit Arbeit versorgen zu können, wird schnell klar, dass ein individuelles Scheduling für die einzelnen Recheneinheiten durch den großen Overhead unpraktikabel wäre. Aus diesem Grund werden die Threads eines Programms in sogenannte *Warps* zusammengefasst, was damit die kleinste Einheit für das Scheduling bildet. Ein Warp enthält dabei genau 32 Threads, welche in diesem Kontext auch *Lanes* genannt werden. Mehrere Warps werden außerdem zu *Blöcken* zusammengefasst, welche schließlich als Ganzes an einzelne Streaming Multiprocessors zu-

gewiesen werden. Innerhalb eines SM werden Warps ausgetauscht, wenn der vorher aktive Warp beispielsweise auf einen Speicherzugriff wartet, um die dadurch entstehende Latenz 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 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 auf unterschiedliche Aspekte des Grids Einfluss nehmen.

Eine für die Programmierung von GPU entscheidende Eigenschaft besteht darin, dass die Threads innerhalb eines Warps parallel ausgeführt werden. Ähnlich wie bei dem Prinzip *Single Instruction Multiple Data (SIMD)*, 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 Threads verschiedene Kontrollflüsse verfolgen können. Prozesse laufen außerdem zwar logisch parallel ab, allerdings muss dies nicht notwendigerweise physikalisch 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 auf die aktiven Threads, bis diese die Arbeit innerhalb ihres Kontrollflusses abgeschlossen haben, sodass an dieser Stelle gegebenenfalls massiv Rechenleistung verschwendet wird. In dieser Problematik liegt der Grund dafür, dass die Verarbeitung von Strings auf Grafikkarten aufgrund ihrer variablen 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, da dadurch eine höhere Leistung erzielt wird. [5] 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. Die Instruktionen werden von der Hardware unterstützt, sodass sie typischerweise sehr effizient ausgeführt werden können. Ein Beispiel für eine solche Operation ist das Auswerten eines Prädikates für alle Threads und anschließend das Erstellen einer Bitmaske, welche 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 dieser sicherstellen will, dass die Daten fertig geschrieben wurden. [4]

2.4 Shared Memory

Eine Kommunikation 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. [3] Die Speichergröße innerhalb eines Streaming Multiprocessors ist allerdings beschränkt, weshalb die Anzahl der Threads ebenfalls beschränkt ist, sofern eine große Menge Shared Memory von diesen benötigt wird.

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 Listing 2.1 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 }
```

Listing 2.1: Beispielhafter CUDA-Kernel zum Iterieren über zwei Datensätze [2]

Kapitel 3

Compiled Query Pipelines

In dieser Arbeit werden String-Vergleiche im Kontext des Query Compilers DogQC für GPUs untersucht. Dieser basiert auf dem Query Compiler HorseQC [1] und wurde für das erleichterte testen von aktuellen Techniken vereinfacht. DogQC erstellt aus einem gegebenen Anfrageplan eine Query Pipeline, die für die Ausführung auf GPUs optimiert ist. Als Grundlage für die späteren Codebeispiele, wird 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. 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. Reicht der begrenzte GPU-Speicher nicht aus, um die Zwischenergebnisse zu speichern, 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, besagt, dass der Anfrageplan in Pipelines aufgeteilt wird, welche 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 orders-Relation gelesen, die Selektion ausgeführt und die Hashtabelle für den Join berechnet. Die rechte Pipeline fasst das Lesen der dates-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

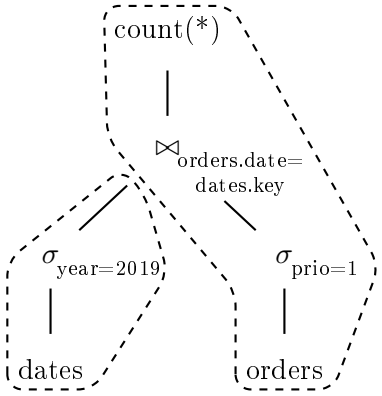


Abbildung 3.1: Beispielplan mit eingezeichneten Pipelines

CUDA-Schnittstelle ausgeführt werden kann. Der Vorteil des Pipelinings besteht drain, dass die Ergebnisse jedes Operators nicht immer wieder im Speicher materialisiert werden müssen, sondern die einzelnen Tupel stets in den GPU-Registern vorgehalten werden können, bis diese fertig verarbeitet wurden.

In Listing 3.1 wird der Code vorgestellt, welcher von dem Query Compiler für den in Abbildung 3.1 dargestellten Anfrageplan generiert wurde. Hier ist zu erkennen, dass die drei Operationen innerhalb eines Kernels zusammengefasst wurden, wodurch eine Pipeline entsteht. Um jedem Thread eine Menge von Tupel 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 rot 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, welche in grün dargestellt ist. Dabei wird das Tupel in der übergebenen Hashtabelle `hashtable_date_key` gesucht und dementsprechend wieder die `active`-Variable angepasst. Schließlich wurde vom Query Compiler noch das Zählen der Ergebnisse umgesetzt, welches hier in gelb hervorgehoben wurde. Dabei wird wieder 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 Durchführung der gesamten Pipeline von Operationen für das untersuchte Tupel, wird der Index 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.

```
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;
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 }
```

Listing 3.1: 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 von String-Verarbeitung dar, wodurch der Einfluss vieler Eigenschaften von Strings auf die Ausführung entsprechender Operationen leicht untersucht werden kann.

Zunächst wird dazu die Motivation hinter einer derartigen Untersuchung erläutert und eine bestehende Technik zur String-Verarbeitung erklärt. Außerdem 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 beurteilt, ob die Lösung das Potential hat eine optimale Performanz zu bieten und auf einen Nachteil der einfachen Implementierung eingegangen.

4.1 Motivation

4.2 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 Spring-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 Warp werden entsprechend dem Verarbeitungsmodell 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.

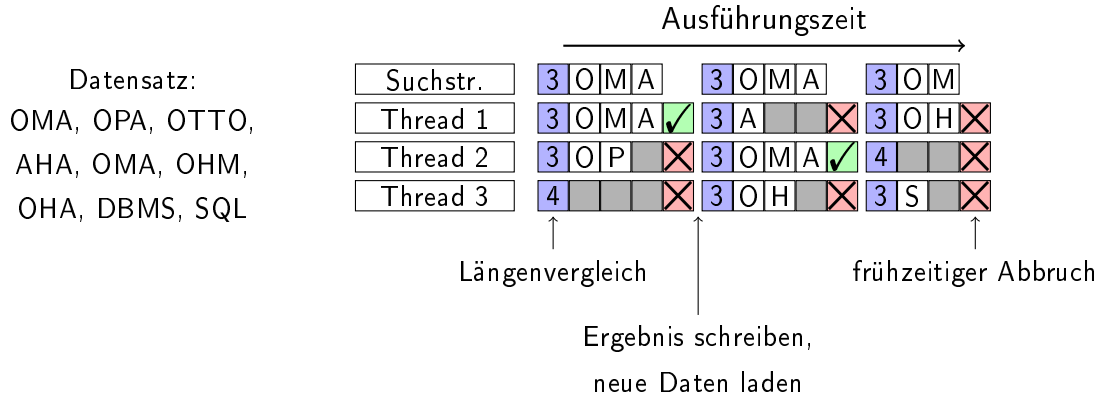


Abbildung 4.1: Funktionsweise des Algorithmus innerhalb eines Warps mit drei Threads

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  data_length = char_offset[loop_var+1] - char_offset[loop_var] - 1;
2
3  // if string lengths are unequals, discard
4  if (active && data_length != search_length)
5    active = false;
6
7  int search_id = 0;
8
9  // iterate over strings completely or until they don't match anymore
10 while(active && search_id < search_length) {
11   int data_id = search_id + char_offset[loop_var];
12
13   // when strings don't match, inactivate the lane
14   if (active && data_content[data_id] != search_string[search_id])
15     active = false;
```

```
16
17    search_id++;
18  }
```

Listing 4.1: Naive Implementierung einer Selektion von Strings

In Listing 4.1 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 Listing 3.1 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 ohne Trennzeichen 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 natürlich 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 für einen String überprüft, ob dessen Länge mit der des Suchstrings übereinstimmt und dieser anderenfalls verworfen. Sind die Längen identisch, wird in der 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.

4.3 Präfixtest als alternativer Workload

4.4 Schlechte Auslastung der GPU

Kapitel 5

Verbesserung des einfachen String-Vergleichs

5.1 Umsetzung mit Lane-Refill

5.2 Implementierung

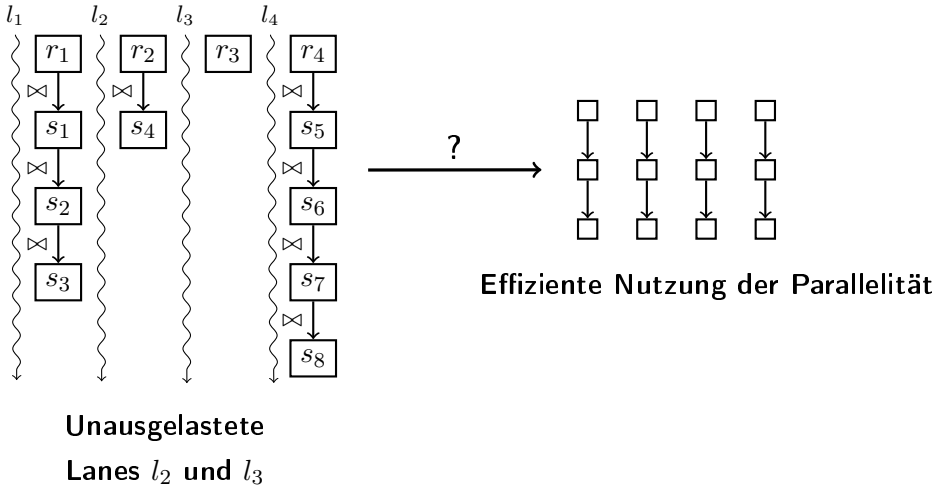


Abbildung 5.1: Berechnung von $R \bowtie S$ mit schlechter Auslastung aufgrund der ungleichmäßigen Verteilung von S. (Quelle: Henning Funke)

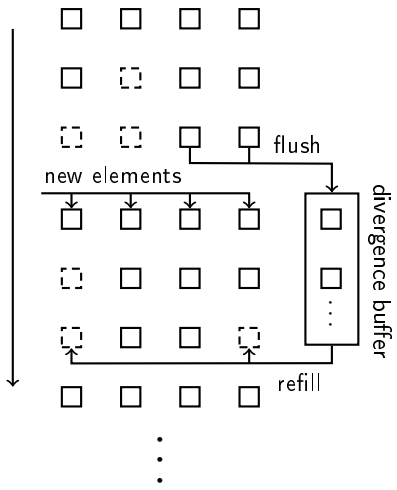


Abbildung 5.2: Divergence Buffer

Kapitel 6

Grundlagen von regulären Ausdrücken

Kapitel 7

Paralleler Musterabgleich mit regulären Ausdrücken

7.1 Vorgehen

7.2 Implementierung

Kapitel 8

Verbesserung des Verfahrens zum Musterabgleich

8.1 Ansatzpunkte für Lane-Refill

8.2 Umsetzung mit Lane-Refill

Kapitel 9

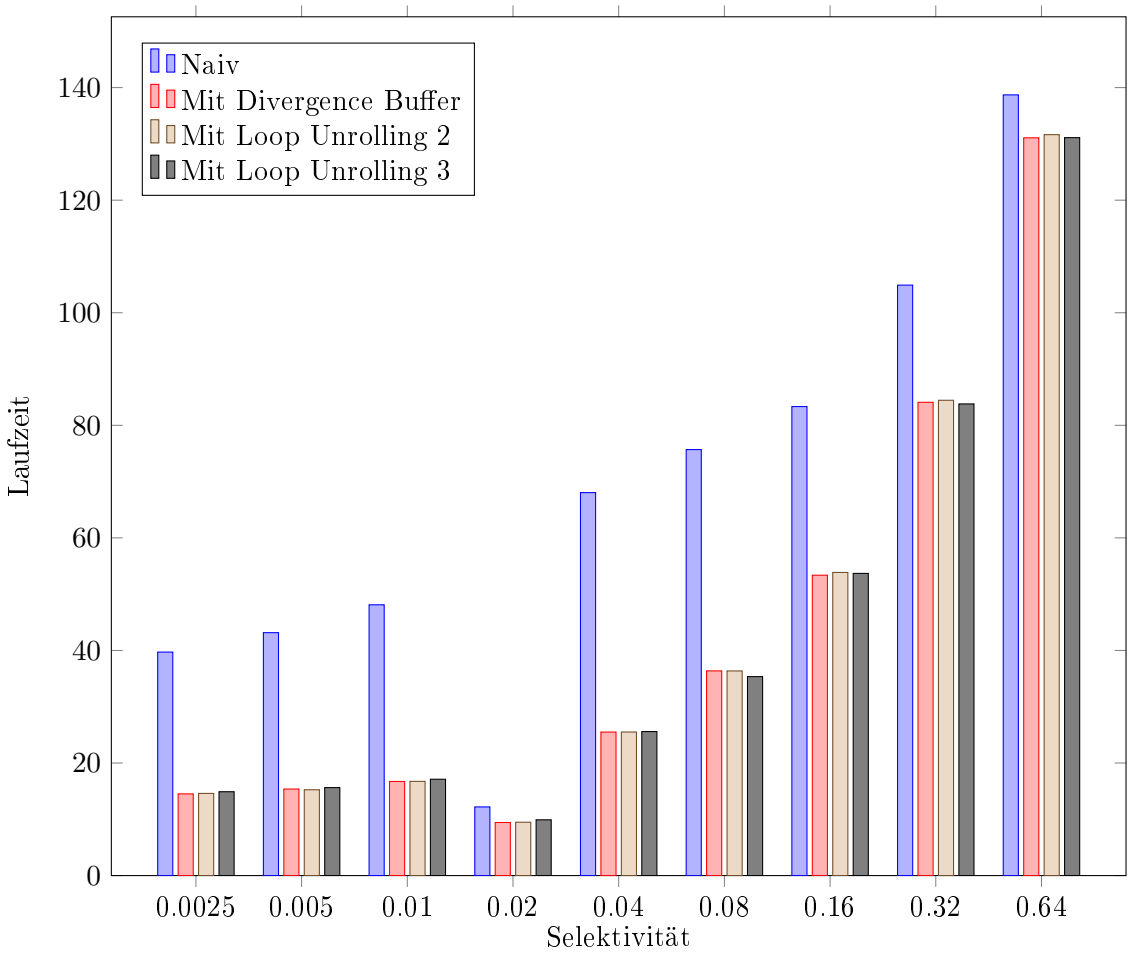
Optimierung der Ausführungsparameter

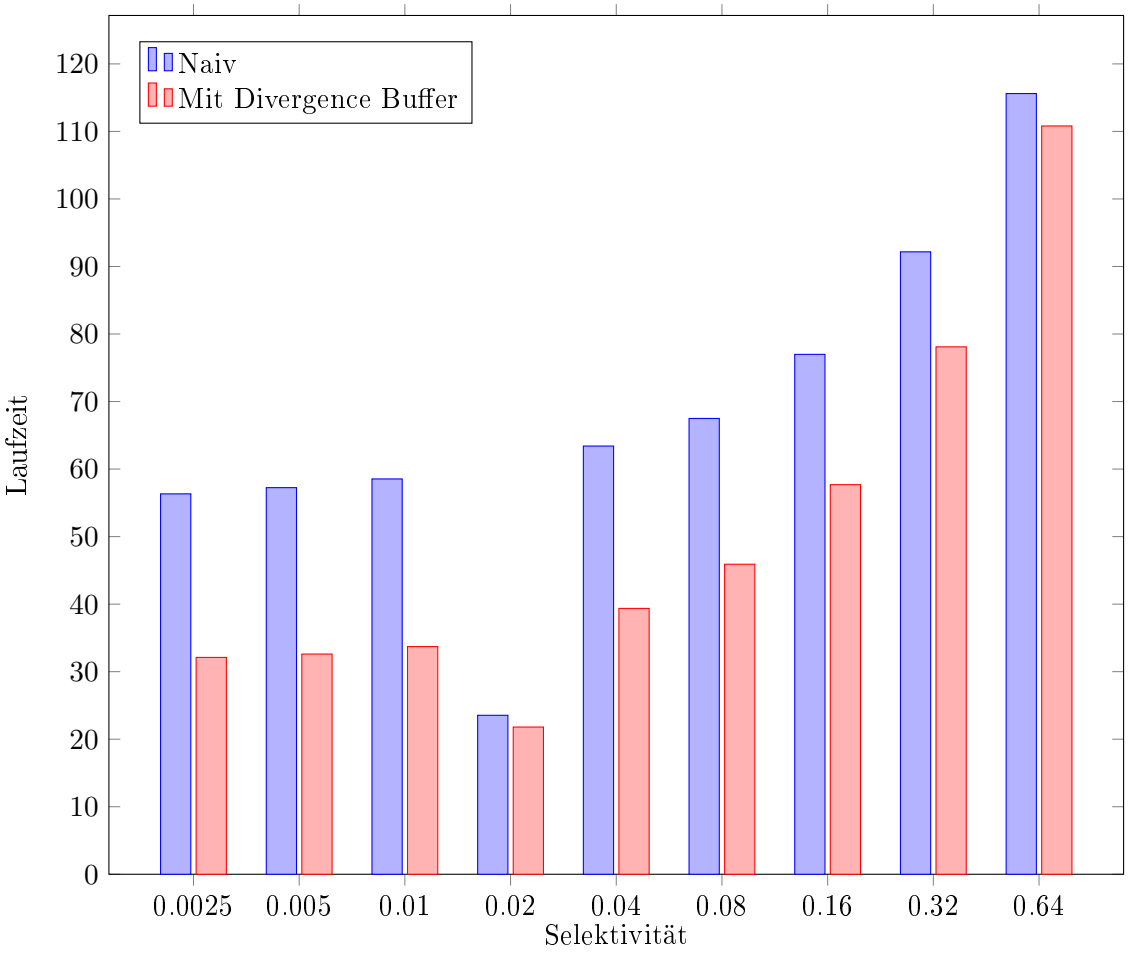
Kapitel 10

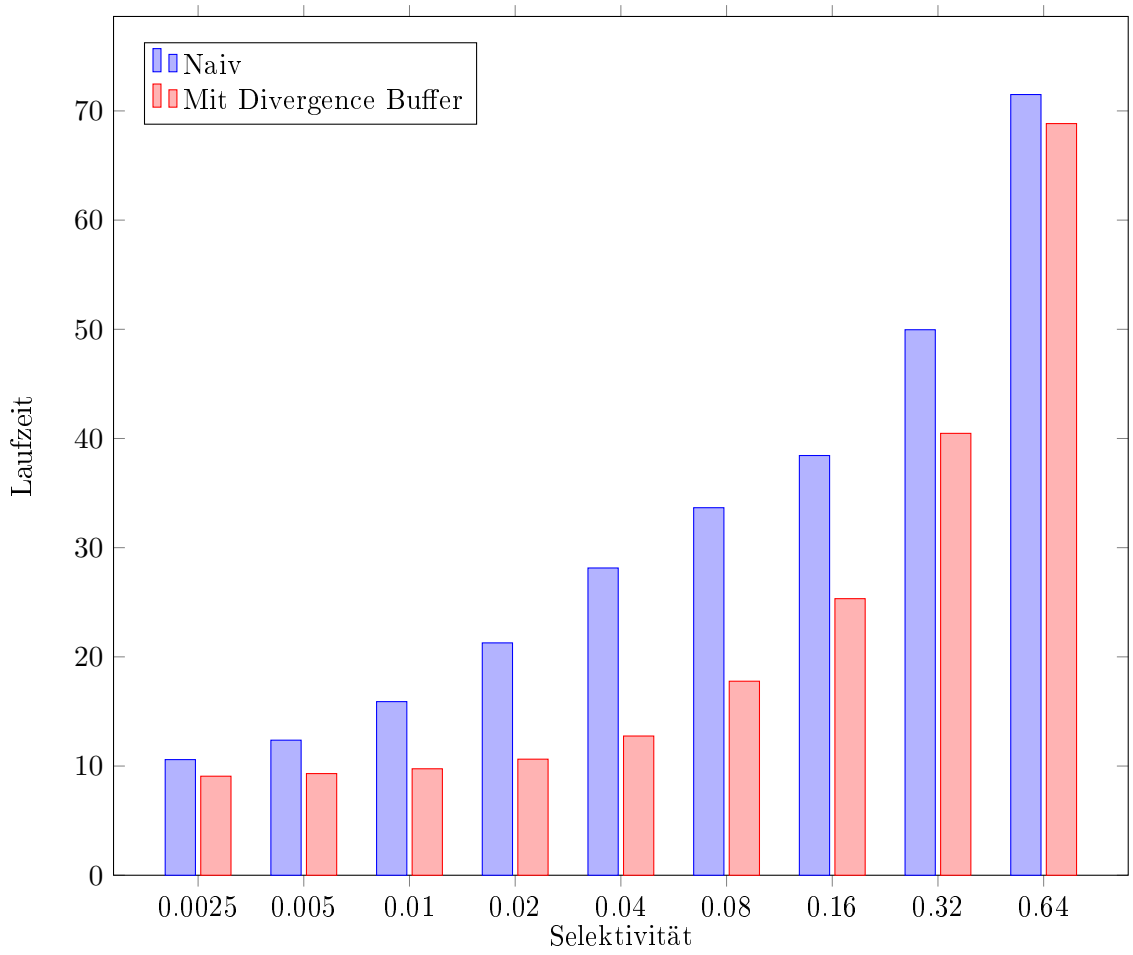
Evaluation des einfachen String-Vergleichs

10.1 Verwendete Workloads und deren Merkmale

10.2 Vorstellung der Messergebnisse




Naiv
Mit Divergence Buffer
Laufzeit
0
10
20
30
40
50
60
70
80
90
100
110
120
0.0025
0.005
0.01
0.02
0.04
0.08
0.16
0.32
0.64
Selektivität



10.3 Diskussion der Ergebnisse

Kapitel 11

Evaluation des parallelen Musterabgleichs

11.1 Verwendete Workloads und deren Merkmale

11.2 Vorstellung der Messergebnisse

11.3 Diskussion der Ergebnisse

Kapitel 12

Ergebnis und Fazit

Anhang A

Weitere Informationen

Abbildungsverzeichnis

Literatur

[1] Henning Funke u. a. "Pipelined Query Processing in Coprocessor Environments". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018. DOI: 10.1145/3183713.3183734. URL: http://doi.acm.org/10.1145/3183713.3183734.

[2] Mark Harris. *An Even Easier Introduction to CUDA*. 2017. URL: https://devblogs.nvidia.com/even-easier-introduction-cuda/.

[3] Mark Harris. *Using Shared Memory in CUDA C/C++*. 2013. URL: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/.

[4] Yuan Lin und Vinod Grover. *Using CUDA Warp-Level Primitives*. 2018. URL: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/.

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

[6] *NVIDIA GeForce GTX 980. Featuring Maxwell, The Most Advanced GPU Ever Made.* Techn. Ber. NVIDIA Corporation, 2014.

[7] Vasily Volkov. *Understanding Latency Hiding on GPUs*. Techn. Ber. University of California at Berkley, 2016.

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 27. Februar 2019

Florian Lüdiger