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 CUDA-Programmierung

Um die in dieser Arbeit vorgestellten Herausforderungen bei der Verarbeitung von String-Daten mit GPUs 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 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 riesigen 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 2GB. Über eine PCI-Express-Anbindung wird die Kommunikation

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

Die GPU wiederum lässt sich in kleinere Module, sogenannte Streaming Multiprocessors (SM), unterteilen, welche jeweils eigenständige Recheneinheiten darstellen. Die 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 beinhalten. Jeder dieser Blöcke besitzt einen eigenen Scheduler und einige Unterstützungselektronik, sodass diese logisch gesehen ebenfalls unabhängig voneinander arbeiten können. 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 GPUs

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 zugewiesen 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 GPUs 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 eige-

ne 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 es aber schon so sein, dass die Threads synchron ausgeführt werden.

Bei der Verwendung von Branching-Instruktionen kann es vorkommen, dass unterschiedliche Threads verschiedene Kontrollflüsse durchlaufen. 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. Das ist auch der Grund dafür, dass die Verarbeitung von Strings auf GPUs aufgrund ihrer variablen Länge problematisch ist.

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. 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.

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

2.5 Die CUDA-Programmierschnittstelle für C++

Kapitel 3

Der Pipelining-Ansatz

Kapitel 4

Einfacher, paralleler String-Vergleich

Für die Evaluation des Lane-Refill-Verfahrens für die Verarbeitung von String-Daten wird zunächst ein einfacher String-Vergleich auf einer GPU untersucht. Ein Vergleich auf Gleichheit ist dabei die einfachste Variante von String-Verarbeitung, die vom Lane-Refill profitieren könnte. Diese Untersuchung wird dabei helfen, zu erfahren, ob die Anwendung des Lane-Refill-Verfahrens bei String-Daten allgemein Potenzial dafür bietet, den Durchsatz entsprechender Anwendungen zu erhöhen.

Zunächst wird dazu ein String-Vergleich mittels der CUDA Schnittstelle ohne spezielle Optimierungen implementiert, um einen Vergleich mit der optimierten Version durchführen zu können. Außerdem wird eine leichte Anpassung an dem Verfahren vorgenommen, sodass ein alternativer Workload für weitere Tests genutzt werden kann.

4.1 Vorgehen

Als Basis für die Untersuchung wird zunächst der Gleichheitstest für Strings naiv, also ohne tiefgehende Optimierungen umgesetzt. Das hier vorgestellte Verfahren soll eine lange Liste von Zeichenketten mit einem anderen String vergleichen. Der Kontext entspricht somit einer Datenbankanfrage, in der eine Selektion über eine Spalte mit String-Daten durchgeführt wird.

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 des Verarbeitungsmodells der GPU parallel abgearbeitet. Somit sind die Positionen, an denen die Strings vergli-

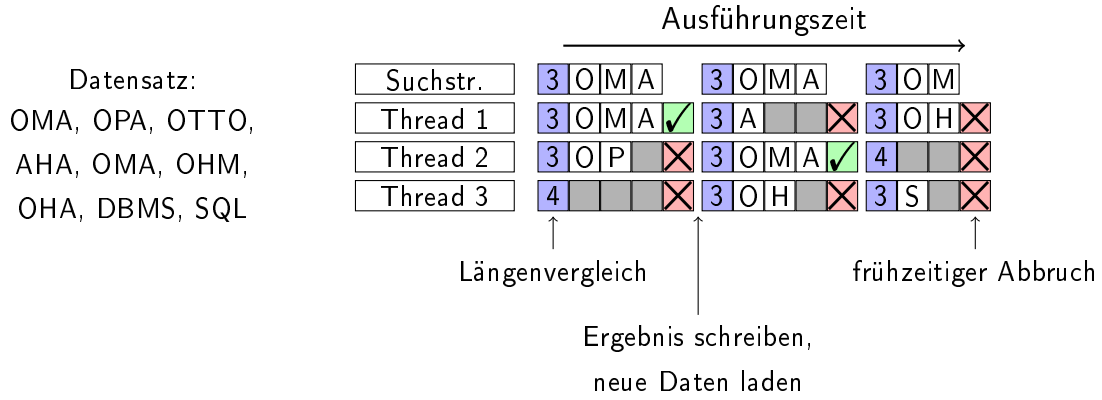


Abbildung 4.1: Funktionsweise des Algorithmus innerhalb eines Warps mit drei Threads

chen werden ebenfalls für alle Threads identisch. Sobald der Vergleichsstring im gesamten Warp vollständig durchlaufen wurden, wird das Zwischenergebnis geschrieben. Im Falle dieser konkreten Untersuchung wird hier der Einfachheit halber lediglich die Anzahl der passenden Zeichenketten gezählt. Es wäre allerdings auch denkbar, dass die Indizes der entsprechenden Einträge gespeichert wird, oder in einer Pipelining-Umgebung der Eintrag an die nächste Operation in der Pipeline weitergegeben wird. 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.

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.

4.2 Implementierung

Als Basis für die Untersuchung wird zunächst der Gleichheitstest für Strings naiv, also ohne tiefgehende Optimierungen umgesetzt. Dies gibt Gelegenheit dazu, die Programmierung einfacher Algorithmen mithilfe der CUDA Schnittstelle für Grafikkarten darzustellen. Da die Analyse im Rahmen dieser Arbeit innerhalb einer Pipelining-Umgebung durchgeführt werden, lassen sich hier außerdem einige Besonderheiten der Implementierung erläutern.

```
1  __global__
2  void naiveKernel(
3    int *char_offset,          // indices of the first letter of every string
4    char *data_content,        // concatenated list of compare strings
5    char *search_string,       // string that will be searched for
6    int search_length,         // length of the search string
7    int line_count,            // number of lines in the data set
8    int *number_of_matches) { // return value for the number of matches
9    // implementation
10 }
```

Listing 4.1: Methodensignatur des Kernels

Für die Umsetzung des Gleichheitstests ist am interessantesten, wie lange das Ausführen des eigentlichen Kernels zum Abgleich des Datensatzes dauert. Dieser Kernel erwartet, dass die benötigten Daten vorher vom Hauptspeicher in den Speicherbereich der GPU kopiert wurden und dort zur Verfügung stehen. In Listing 4.1 ist die Methodensignatur des Kernels für den einfachen Stringvergleich dargestellt.

Die Position des Datensatzes, welcher mit dem Vergleichsstring abgeglichen werden soll, wird über den Zeiger `data_content` übergeben. Der Datensatz besteht in einer Aneinanderreihung der Entsprechenden Zeichenketten ohne Trennzeichen. Damit daraus die ursprünglichen Strings extrahiert werden können, gibt es einen zweiten Array, welcher Informationen über die Indizes der Einzelstrings innerhalb des Datensatzes enthält. Die Position dieser Informationen wird über die Variable `char_offset` übergeben. Ebenfalls muss natürlich ein Zeiger auf den Suchstring und dessen Länge in den entsprechenden Parametern `search_string` und `search_string_length` mitgeliefert werden. Um die Berechnung rechtzeitig vor Speicherüberschreitungen abbrechen zu können, wird schließlich noch die Variable `line_count` übergeben, welche die Anzahl der Zeichenketten im Datensatz beschreibt. Der letzte Parameter `number_of_matches` dient dazu, dass an die entsprechende Speicherstelle das Ergebnis der Berechnung geschrieben werden kann und dieses aus dem Hauptprogramm heraus wieder ausgelesen werden kann.

```
__global__
void naiveKernel( /* parameters */ ) {
  // global index of the current thread,
  // used as the iterator in this case
  unsigned loop_var = ((blockIdx.x * blockDim.x) + threadIdx.x);

  // offset for the next element to be computed
  unsigned step = (blockDim.x * gridDim.x);

  bool active = true;
  bool flush_pipeline = false;

  while(!flush_pipeline) {
    // element index must not be higher than line count
    active = loop_var < line_count;

    // break computation when every lane is finished and therefore inactive
    flush_pipeline = !__ballot_sync(ALL_LANES, active);

    data_length = char_offset[loop_var+1] - char_offset[loop_var] - 1;

    // if string lengths are unequals, discard
    if (active && data_length != search_length)
      active = false;

    int search_id = 0;

    // iterate over strings completely or until they don't match anymore
    while(__any_sync(0xFFFFFFFF, active) && search_id < search_length) {
      int data_id = search_id + char_offset[loop_var];

      // when strings don't match, inactivate the lane
      if (active && data_content[data_id] != search_string[search_id])
      active = false;

      search_id++;
    }

    // if still active, a match has been found
    if (active)
      atomicAdd(number_of_matches, 1);

    loop_var += step;
  }
}
```

Listing 4.2: Naive Implementierung des String-Vergleichs

In Listing 4.2 ist die Implementierung des Algorithmus dargestellt, welcher alle Strings aus dem Datensatz mit dem Suchstring vergleicht und daraufhin die Anzahl der passenden Zeichenketten zurückliefert. Zunächst wird 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 Anzahl dieser Elemente lässt sich durch $\frac{\text{Datensatzgröße}}{\text{Gridgröße} \times \text{Blockgröße}}$ berechnen.

Die Variable `active` zeigt im Algorithmus an, ob das aktuell untersuchte Datenelement noch aktiv geprüft wird, oder dieses bereits verworfen wurde. Somit zeigt diese Variable auch an, ob der aktuelle Thread aktiv läuft, oder nur darauf wartet, dass die anderen Threads aus seinem Warp ihre Berechnung abschließen. 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 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. Entsprechend des Ergebnisses läuft die Schleife bis zum Ende durch und die Anzahl passender Elemente im Datensatz kann erhöht werden, oder die Untersuchung wird vorzeitig abgebrochen und der Thread als inaktiv markiert.

4.3 Nachteile des Verfahrens

4.4 Präfixtest als alternativer Workload

Kapitel 5

Das Lane-Refill Verfahren

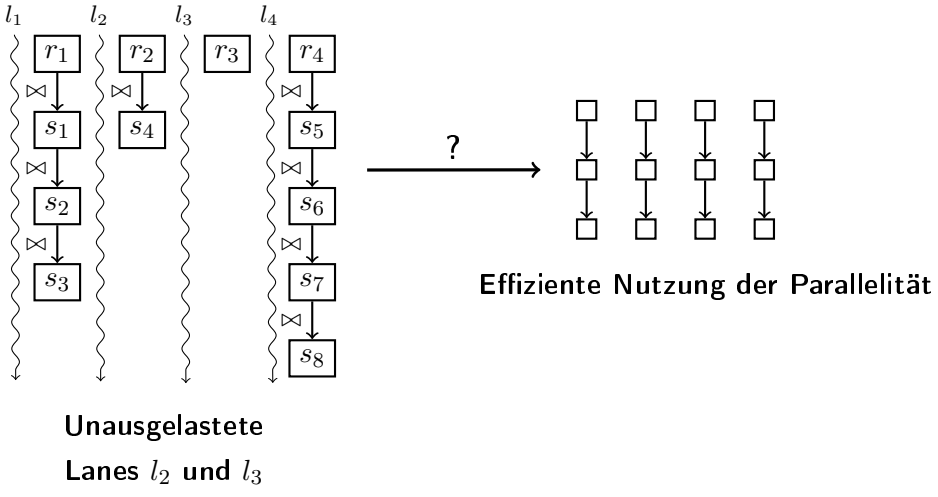


Abbildung 5.1: Berechnung von $R \bowtie S$ mit schlechter Auslastung aufgrund der ungleichmäßigen Verteilung von S. (Quelle: Henning Funke)

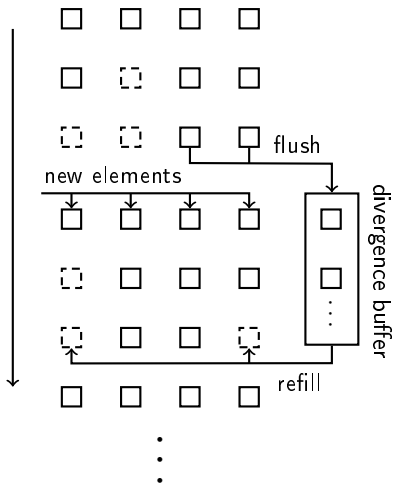


Abbildung 5.2: Divergence Buffer

Kapitel 6

Verbesserung des einfachen String-Vergleichs

6.1 Ansatzpunkte für Lane-Refill

6.2 Umsetzung mit Lane-Refill

Kapitel 7

Grundlagen von regulären Ausdrücken

Kapitel 8

Paralleler Musterabgleich mit regulären Ausdrücken

8.1 Vorgehen

8.2 Implementierung

Kapitel 9

Verbesserung des Verfahrens zum Musterabgleich

9.1 Ansatzpunkte für Lane-Refill

9.2 Umsetzung mit Lane-Refill

Kapitel 10

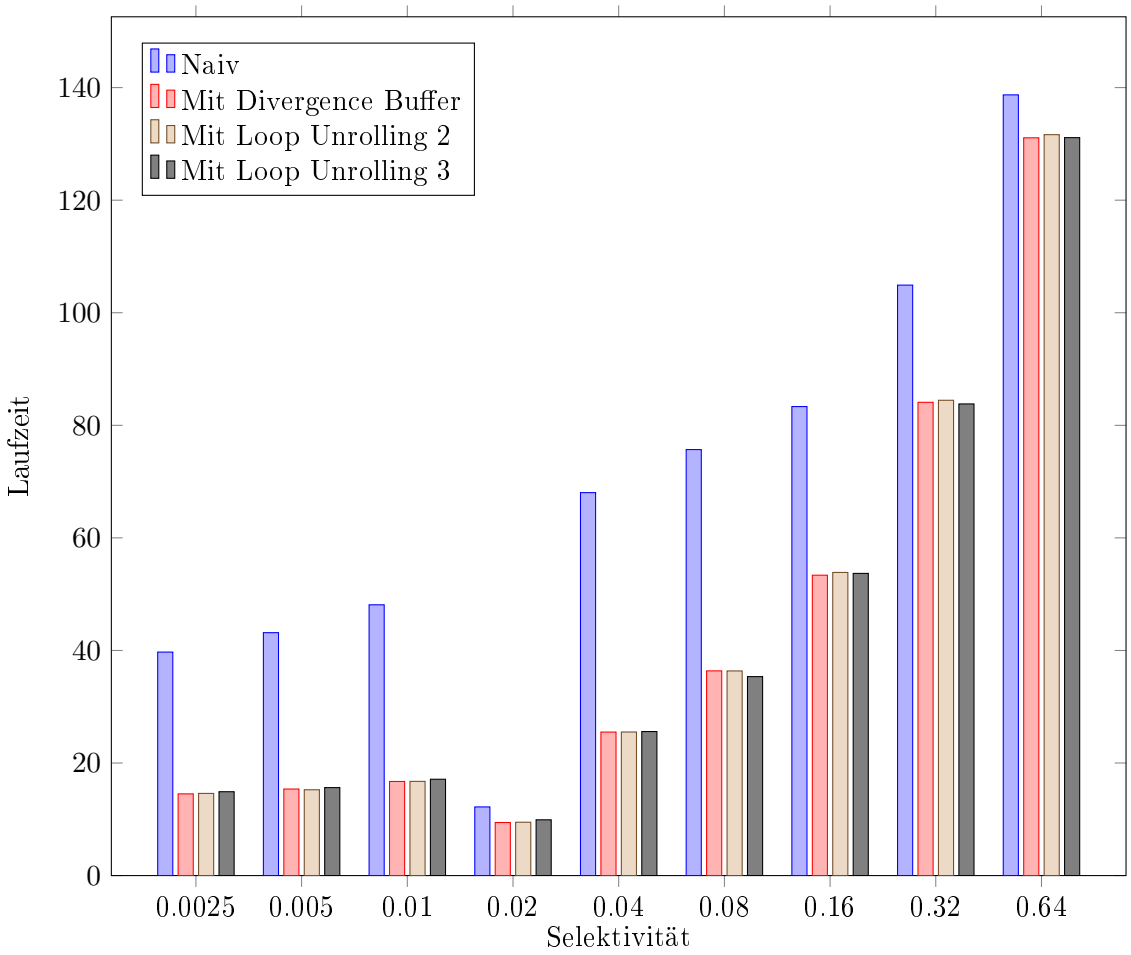
Optimierung der Ausführungsparameter

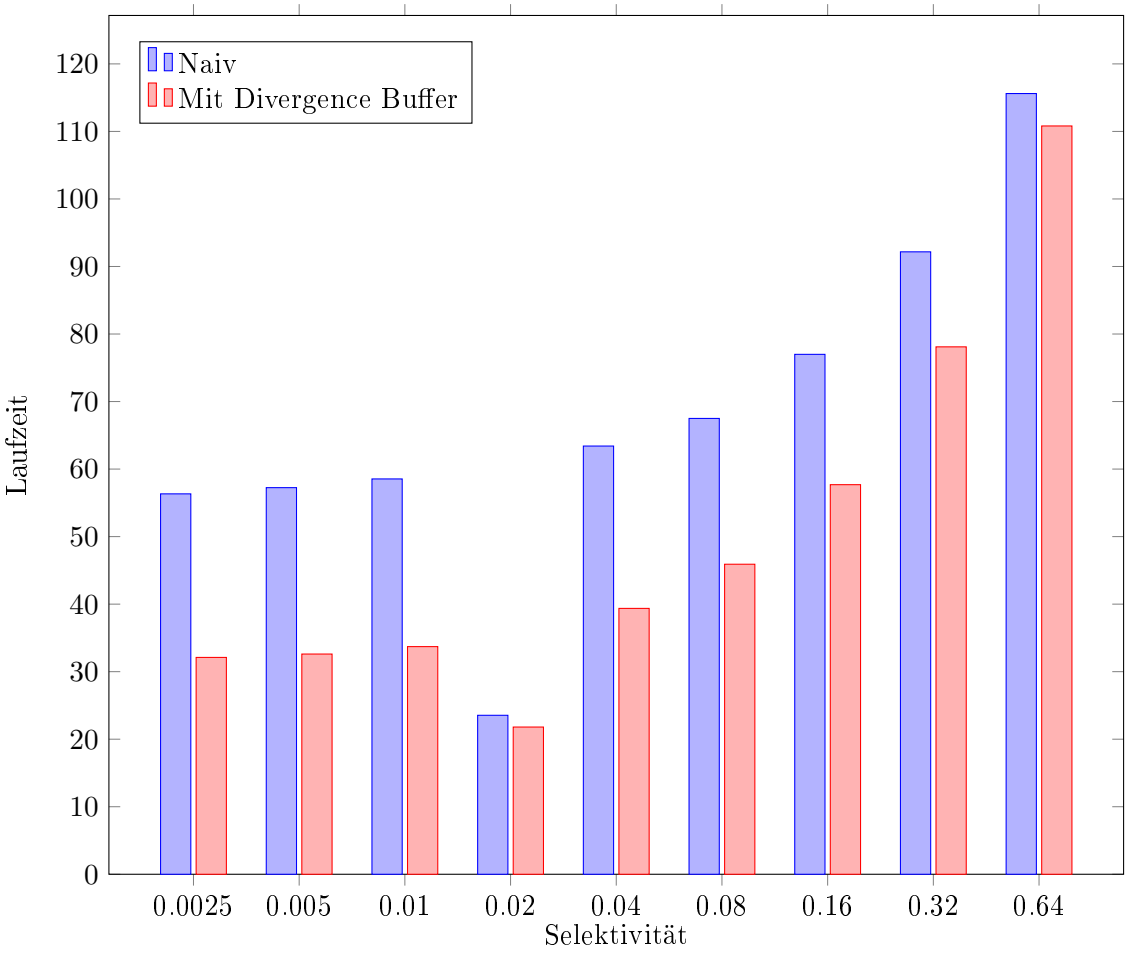
Kapitel 11

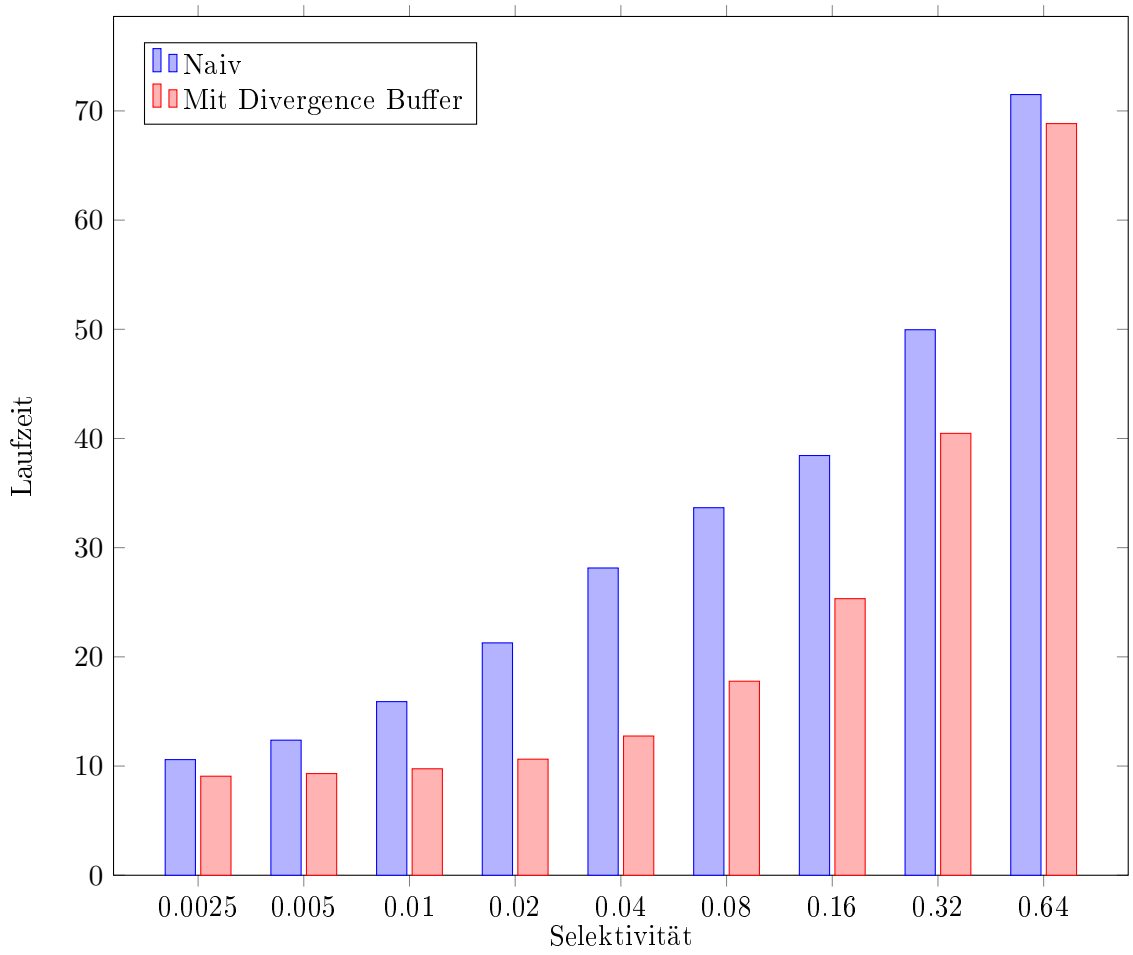
Evaluation des einfachen String-Vergleichs

11.1 Verwendete Workloads und deren Merkmale

11.2 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



11.3 Diskussion der Ergebnisse

Kapitel 12

Evaluation des parallelen Musterabgleichs

12.1 Verwendete Workloads und deren Merkmale

12.2 Vorstellung der Messergebnisse

12.3 Diskussion der Ergebnisse

Kapitel 13

Ergebnis und Fazit

Anhang A

Weitere Informationen

Abbildungsverzeichnis

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 11. Februar 2019

Florian Lüdiger