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

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-

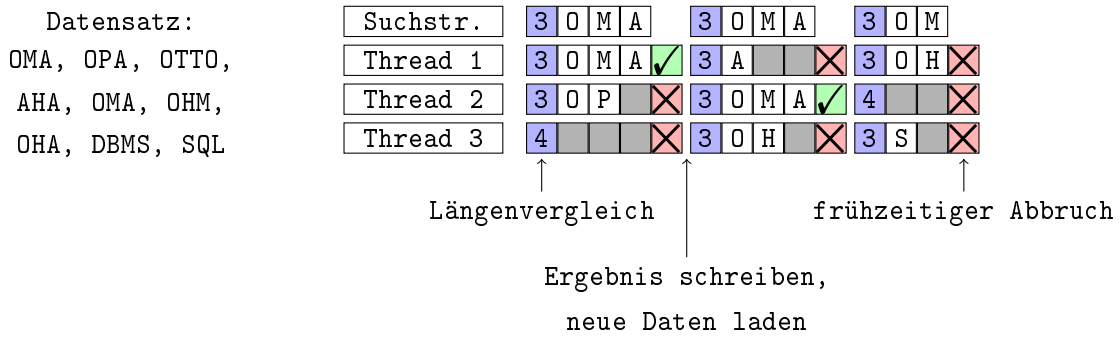


Abbildung 4.1: Funktionsweise des Algorithmus innerhalb eines Warps mit drei Threads

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.

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

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

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

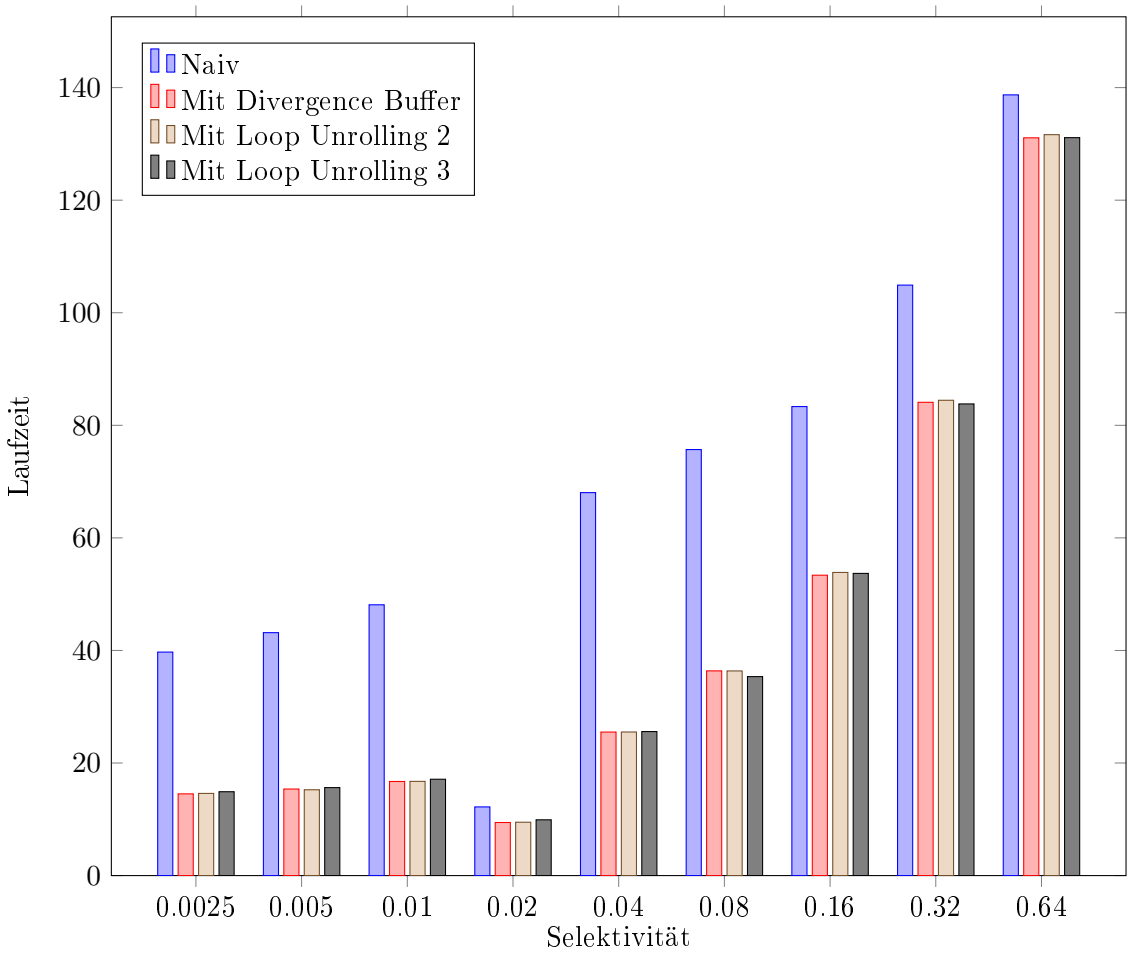
Optimierung der Ausführungsparameter

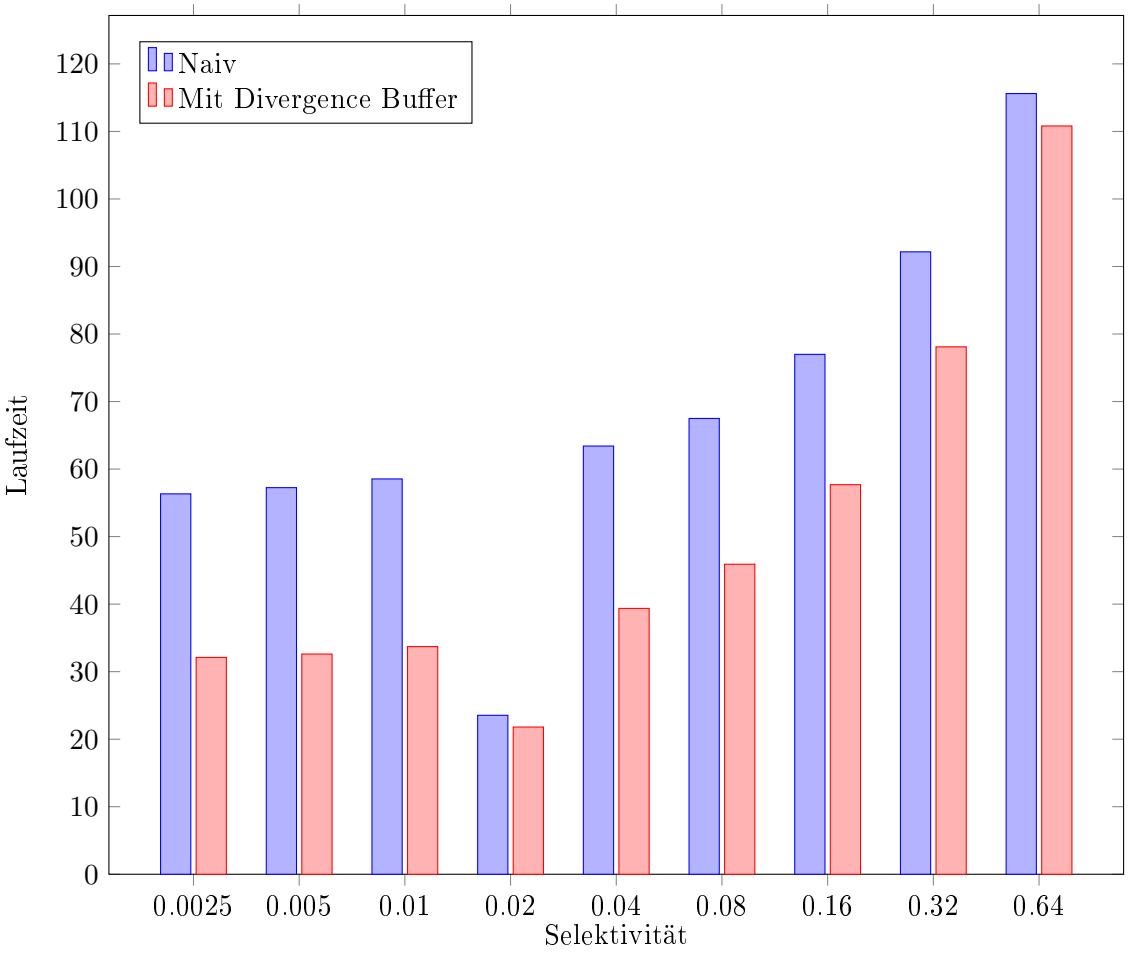
Kapitel 11

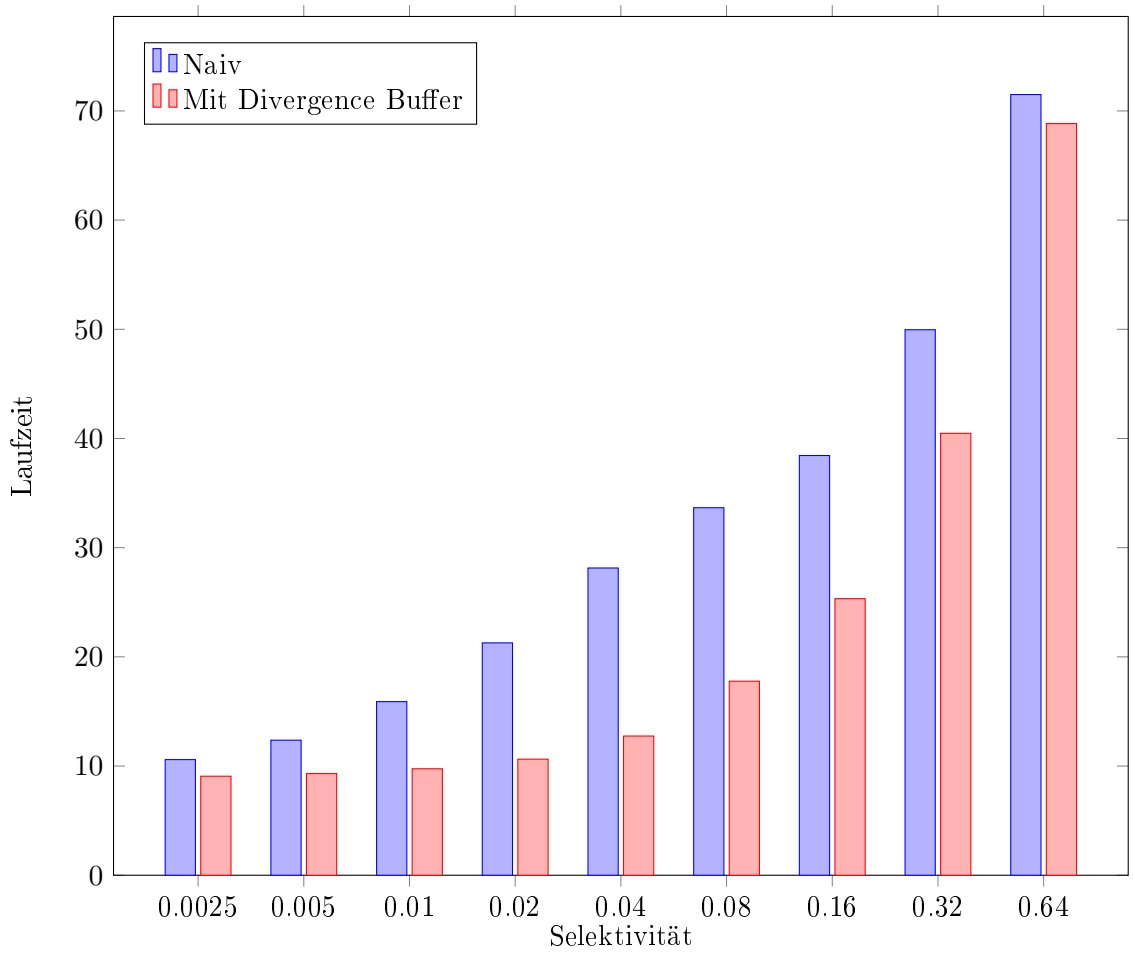
Evaluation des einfachen String-Vergleichs

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 4. Februar 2019

Florian Lüdiger