Advanced Techniques for Bounded and Unbounded Repetition in Parabix Regular Expression Search

by

Dong Xue

B.Sc., University of Science and Technology of China, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Science

© Dong Xue 2017
SIMON FRASER UNIVERSITY
Fall 2017

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name: **Dong Xue**

Degree: **Master of Science**

Title: ***Advanced Techniques for Bounded and Unbounded Repetition in Parabix Regular Expression Search***

Examining Committee: **Chair:** Leonid Chindelevitch
Assistant Professor

Robert D. Cameron
Senior Supervisor
Professor

Fred Popowich
Supervisor
Professor

Arrvindh Shriraman
Examiner
Associate
Professor

Date Defended: 12 September 2017

Abstract

The three-level architecture of the regular expression search tool named icGrep, which is based on a pure parallel Parabix framework, has shown great speedup compared to conventional search tools. This thesis proposed some advanced techniques for bounded and unbounded repetition in Parabix regular expression search. We first accelerated the bounded repetition type of Unicode unit-length regular expressions by utilizing a log2 technique with the UTF8-to-UTF32 pipeline. To reduce the overhead brought about by the UTF-8-to-UTF-32 transformation, the multiplexed character classes concept was proposed. For the unbounded repetition part, we have reviewed finite automata theory for application to Parabix regular expression matching and proposed a totally different compile pipeline for the local language. In the meanwhile, we proposed star-normal-form optimization to make the RE abstract syntax tree less complex and less ambiguous. All of these innovative techniques have demonstrated their performance dealing with repetition against the basic pipeline.

Keywords: icGrep; Regular Expression; Parabix; Repetition

Dedication

I feel honoured to have get the guidance of Professor Robert Cameron. Weekly meetings with Rob really help a lot, and I can't figure out many things by myself. I not only benefit a lot from him in academic studies, but also gain much life experience that can never be got from the textbooks.

I would be grateful to professor Fred Popowich for his great patience in assisting my thesis review. Thanks to all my friends in professor Rob's laboratory. Nigel and Linda have helped me a lot on the icGrep pipeline and Parabix framework.

I would like to acknowledge my parents, who have given as much support as I need.

Table of Contents

Approval ii

Abstract iii

Dedication iv

Table of Contents v

List of Tables vii

List of Figures viii

List of Programs ix

1 Introduction 1

2 Background 4

2.1 Parabix Framework 4

2.2 Kernel Programming in Parabix Framework 6

2.3 Regular Expression Matching 6

2.3.1 Regular Expression 6

2.3.2 Matching Process 7

2.4 Repetition Type 11

2.5 Review of Finite Automata Theory for IcGrep 11

2.5.1 Local Language 11

2.5.2 Automata Construction 11

2.5.3 Star Normal Form 14

3 Design Objective 16

3.1 Accelerate Bounded Repetition 16

3.1.1 Log2 Technique for Fixed-length Bounded Repetition 16

3.1.2 Extend the application of Log2 Technique 17

3.2 Add Star Normal Form Pass 17

3.3 Glushkov Construction's Application to Parabix Regular Expression Matching 18

4 Implementation **20**
4.1 Bounded Repetition 20
4.1.1 UTF-8 to UTF-32 Pipeline 20
4.1.2 Multiplexed Character Classes Pipeline 26
4.2 Unbounded Repetition 30
4.2.1 Star Normal Form Pass 30
4.2.2 New Compile Pipeline for Local Language 32

5 Performance Evaluation **35**
5.1 Bounded Repetition 35
5.2 Unbounded Repetition 38
5.2.1 Star Normal Form 38
5.2.2 New Compile Pipeline for Local Language 39

6 Conclusion and Future Work **41**
6.1 Conclusion 41
6.2 Future Work 41
6.2.1 Log2 Technique for Bounded Repetition of Arbitrary Length Regular Expression 41
6.2.2 Support for Extended Regular Expression Types 42

Bibliography **43**

Appendix A Code **45**

List of Tables

Table 2.1 Matching Time for Complex Expressions (s / GB) [12] 10

Table 4.1 The Structure of UTF-8 Encoding 20
Table 4.2 Kernel Cycles for matching ".{4}" on the 19 MB XML file by U8U32 Pipeline 26
Table 4.3 Bitset which indicates whether the breakpoint belongs to the source CCs 27
Table 4.4 Kernel Cycles for matching ".{4}" on the 19 MB XML file by Multiplexed Pipeline 30

Table 5.1 The Hardware Configuration of the Test Machine 36
Table 5.2 Averaged Matching Time for Different Repetition Times(ms/MB) . . 36

List of Figures

Figure 2.1 Transform bytestream "Parabix" into eight bit streams 4
Figure 2.2 Advance Operation 5
Figure 2.3 ScanThru Operation 5
Figure 2.4 MatchStar Operation 6
Figure 2.5 Architecture for icGrep [12] 8
Figure 2.6 Match "p[a-z]*r" in the text stream 10
Figure 2.7 Match "nihao" in the Chinese text stream 10
Figure 2.8 Rules for Thompson Construction 13

Figure 3.1 Match "abc" by the pipeline enlightened by Glushkov automaton . 18
Figure 3.2 Match "aba" by the pipeline enlightened by Glushkov automaton . 19

Figure 4.1 Transformation from UTF-8 to UTF-32 21
Figure 4.2 Pipeline for icGrep 23
Figure 4.3 Pipeline for deletion on U8U32 bits 24
Figure 4.4 Pipeline for deposit on Matches Stream 25
Figure 4.5 Four streams before swizzling 25
Figure 4.6 Four streams after swizzling 25
Figure 4.7 Pipeline for icGrep with multiplexed character classes 29

Figure 5.1 Performance on Three Different Files 37
Figure 5.2 Averaged Matching Time(Log2 Pipelines vs. pcregrep) 38
Figure 5.3 Performance for star normal form pass with respect to different numbers of starred subexpressions inside the bracket 39
Figure 5.4 Performance for star normal form pass with respect to different nested levels of starred subexpressions 40
Figure 5.5 Performance for local language with respect to different nested levels of starred subexpressions 40

Figure 6.1 Log2 technique for arbitrary length repetition type 42

List of Programs

4.1 Regular Expression AST for "$(a^*b^*)^*$" 31
4.2 Regular Expression AST for "$(a^*b^*)^*$" in Star Normal Form 31
4.3 Pablo Primitives for RE "$(a^*b^*)^*$" in Star Normal Form 31
4.4 Pablo Primitives for RE "$(a^*b^*)^*$" 32
A.1 U8 to U32 Transformation 45
A.2 Multiplexed Character Classes Pipeline 48

Chapter 1

Introduction

Most of the traditional regular expression search tools use the finite automata, which includes the deterministic finite automaton(DFA) and nondeterministic finite automaton(NFA), to recognize the giving regular expressions and then search the given text based on the finite automata. There are many algorithms to convert the regular expression into NFA, such as Thompson [19] and Glushkov constructions [15]. Due to the reality that a number of states become active when traverse the text, it is impossible to match a regular expression of length m in a text of length n less than O(mn) time with this NFA in most cases. Therefore, they try to convert the NFA into DFA, which has one active state exactly at one time and can reduce the time complexity to O(n) [19]. However, this might lead to state explosion when doing the conversion. Besides the utilization of NFA or DFA, there are various regular expression engines using the concept of backtracking. In a backtracking algorithm, if a symbol fails to match, then the regular expression engine will return to a previously saved position or state, where it could have taken a different pass, to continue the search for a match. It will take O(mn) time when the expression doesn't include any alternation constructs. In some worst cases, the search engine may explore all potential paths and result in an explosion of the time we spent.

All of the above-mentioned algorithms are proved to be difficult to parallelize, and must match the regular expressions in the given text byte-at-a-time [9]. Cameron proposed the Parabix framework which utilize the concept of parallel bit streams [4]. IcGrep is a regular expression search tool based on the pure parallel Parabix technology and a three-level compilation architecture [17] .

The first level is the regular expression, which gets the input of regular expressions from the command line, parses them into abstract syntax tree forms, and finally compiles them into three-address Parabix equations. The second level is the Parabix, which gets the Parabix equations as inputs, and finally compiles them into LLVM IR form. The last level is LLVM, which dynamic compiles and links the LLVM IR, and finally generates a function

finding all the matches to the regular repressions. There are corresponding optimizations for each level, in the form of passes.

This thesis mainly focuses on proposing some advanced techniques for bounded and unbounded repetition types in Parabix regular expression search. The traditional searching tools such as GNU grep, rg etc. that are NFA or DFA-based usually expand the repetition into an equivalent form of multiple copies [14]. For example, the expansion of "$R\{3\}$" is "RRR"; the expansion of "$R\{3,\}$" expands to "$RRR+$"; the expansion of "$R\{2,6\}$" is "$RRR?R?R?R?$". The number of states in the finite automaton will grow as the number of repetitions grows. Other tools such as Perl, PCRE, Python, Ruby etc. that are backtracking-based use recursive matching loops to count the repetition type [14]. Either way, the process of matching repetition is slow and large repetition counts will be unwise.

In the Parabix-LLVM framework, the normal pipeline utilizes the log2 technique [3] to optimize repetition of byte-length type, which has shown a great benefit to performance and can support very large count numbers. In this paper, we want to extend the application to more general cases, especially the repetition of Unicode-unit-length type. At first, we proposed the UTF8 to UTF32 pipeline to accelerate this case. Although we have already got significant performance increases with this pipeline, we find that the overhead caused by the UTF8 to UTF32 transformation is becoming more obvious with the times of repetition becoming less. Therefore, the multiplexed character classes concept is proposed to minimize the overhead, and the experimental results are in good agreement with theory.

Besides the cases of bounded repetition type, we have also explored new approaches to optimize the regular expressions with unbounded repetition type. We have reviewed the finite automata theory for application to Parabix regular expression matching, and proposed star-normal-form optimizations in RE level, which makes the RE abstract syntax tree less complex and more unambiguous. We also utilize the algorithm of Glushkov NFA construction to propose totally new compile logic for local language, by which it can be checked more efficiently. Both of these two methods enlightened by automata theory have shown their performance against the conventional tools and the original icGrep pipeline.

The chapters are organized as following: Chapter 2 provides the basic background information about the Parabix framework, kernel programming in this framework, regular expression matching process, repetition type, and some basic knowledge in the automata theory area. Chapter 3 shows the objective of dealing with regular expression with counting. Detailed implementation of UTF8-to-UTF32 pipeline is described first in Chapter 4, and in order to minimize the overhead brought by the transformation from UTF8 to UTF32, we proposed totally new pipeline that based on multiplexed character classes concept. In the chapter 5, we compared their performance with the previous version of icGrep and some unix command-line search tools, such as GNU grep, pcregrep and rg. Finally, the conclusion is outlined in Chapter 6. In the meanwhile, we proposed the potential way to extend

the application of log2 technique to arbitrary repetition type and identified the idea to add support for more regular expression types in the Parabix framework.

Chapter 2

Background

2.1 Parabix Framework

The Parabix toolchain takes a new form of input rather than conventional byte streams [4, 17]. It first transforms the original byte streams into several parallel basis bit streams based on the encoding form. As the example shown in Figure 2.1, we'll get eight basis bit streams for UTF-8, each of which represents a corresponding bit in a byte. In this case, we can take full advantage of the characteristic of SIMD registers, and process block-size data at one time. The block size is decided by the width of SIMD register, which varies from 128 bits to 512 bits based on the different architecture. For instance, we can process 128 bits at one time with SSE2, Neon, and 256 bits at one time with AVX2.

All the Parabix programs are operated on unbounded bit streams, which are considered to be long integers. Conceptually, these unbounded bit streams are computed at the same time. But in practice, they are separated to blocks and computed block-by block, where the block size is the width of SIMD register. If specific operations cause the last bits in the current block to cross the boundary, then we need to use the carry queue to move them

```
Bytestream text         Parabix
  Hex Coding     50 61 72 61 62 69 78
     bit0            0 0 0 0 0 0 0
     bit1            1 1 1 1 1 1 1
     bit2            0 1 1 1 1 1 1
     bit3            1 0 1 0 0 0 1
     bit4            0 0 0 0 0 1 1
     bit5            0 0 0 0 0 0 0
     bit6            0 0 1 0 1 0 0
     bit2            0 1 0 1 0 1 0
```

Figure 2.1: Transform bytestream "Parabix" into eight bit streams

M	`.1.........................1....................`
$Advance(M)$	`..1.........................1...................`

Figure 2.2: Advance Operation

`input text`	`parabix is a framework to process streaming text`
$M1$	`.1.1.......1...1......................1.........`
$M2 = CC[a-m]$	`.1.111..1..1.1.111...1.......11......111111..1..`
$M1 + M2$	`..1...1.1...11....1..1.......11......1.....1.1..`
$(M1 + M2) \wedge \neg M2$	`..1...1.....1.....1.......................1....`

Figure 2.3: ScanThru Operation

to the next block. There are two types of three-address Parabix equations: character class equations and regular expression matching equations. Correspondingly, there are two kinds of streams in the Parabix level, character class bit streams and marker bit streams. The character class bit streams are used to identify the corresponding character classes. They are calculated by a set of specific boolean-logic operations on the eight basis bit streams. The process of calculation is shown in character class equations. Marker streams are used to identify the current match position during the matching process. Similarly, the process is shown by regular expression matching equations. There are also some useful streams which are initialized before matching process, such as Initial, NonFinal, UTF8invalid streams, etc. Some of them can be used to solve the multiple bytes text streams matching problem, and others are used to validate the data for UTF well-formedness

The operations of Parabix program are in three types: bitwise logic, bit-stream shifting and long stream addition. These are some important operations in Parabix framework:

- Advance: The Advance operation takes a marker bitstream as input, and advance all the bits in the stream one bit forward.

- Scanthru: The Scanthru operation accepts the initial marker M and the character class stream C as inputs, and try to set the cursor of the first stream's positions directly after a run of marker positions of the second stream, not one bit advance at a time. This operation is defined as: $ScanThru(M, C) = (M + C) \wedge \neg C$.

- MatchStar: The MatchStar operation accepts the initial marker M and the character class stream C as inputs,. It corresponds to the Kleene star in the regular expression, and will return all the reachable positions with the occurrence of zero or arbitrary times of repetition of the second input character class stream. This operation is defined as: $\text{MatchStar(M, C)} = (((\text{M} \wedge \text{C}) + \text{C}) \oplus \text{C}) \vee \text{M}$.

input text	`parabix is a framework to process streaming text`
$M1$	`.1.1.......1...1......................1.........`
$M2 = CC[a-m]$	`.1.111..1..1.1.111...1.......11......111111..1..`
$T0 = M1 \wedge M2$	`.1.1.......1...1......................1.........`
$T1 = T0 + M2$	`..1...1.1...11....1..1.......11......1.....1.1..`
$T2 = T1 \oplus M2$	`.111111....11..1111...................111111....`
$T3 = T2 \vee M1$	`.111111....11..1111...................111111....`

Figure 2.4: MatchStar Operation

2.2 Kernel Programming in Parabix Framework

In icGrep, a program is composed by kernels and stream sets [4]. The stream set is an ordered set of sequences of fields of bit width 2^k, where $k \in \mathbb{Z}_+$. The kernel will do some specific computation based on the inputs and outputs stream sets, and there might also be some non-stream, named scalar, inputs or outputs when needed. In particular, the source(sink) kernel has no input(output) streams sets. The key idea for Parabix programming is like the following equation:

$$\text{Program} \;=\; \text{Kernels} + \text{StreamSets} \tag{2.1}$$

The kernel writer often needs to declare kernel attributes and update processedItemCount(producedItemCount) for each input(output) stream set. There are three kinds of kernels that perform different numbers of blocks at one time for a different purpose.

1. Block oriented kernels typically perform block-at-a-time processing.
2. Segment oriented kernels implement the doSegment interface to process input segments of a given size K. The kernels that use the segment-oriented ones will call the default do-block functions.
3. Multiblock oriented kernels implement the doMultiBlock call to deal with the minimum number of blocks that the kernel requires each time.

All those three kernels only process full stride of blocks by default. When the program requires specific processing to handle partial blocks at end of file, the kernel writer must need to implement the specific do-final functions. If no specific processing required, then the program will use the normal full stride of blocks processing logic.

2.3 Regular Expression Matching

2.3.1 Regular Expression

The regular expressions follow the standards of Portable Operating System Interface for uniX (POSIX), which regulates two flavors of expressions. The first one is POSIX

Basic Regular Expression (PCRE), and another one is POSIX Extended Regular Expression (ERE) [5]. The default rules of icGrep for regular expressions are as follows:

- a (denoting the singleton set containing the single-symbol string a)
- $[a_1 a_2 ... a_n]$ (denoting a character class that may match any one of the symbols a_i in the bracket)
- $[a_1 - a_n]$ (denoting a character class that may match any symbols in the range of a_1 to a_n in the bracket)
- ST (denoting the set of all possible concatenations of strings from set S and T)
- S | T (where S and T are, in turn, generalized regular expressions; denoting their set's union)
- S∗ (denoting the set of n-fold repetitions of strings from set S, for any n≥0, including the empty string)
- S+ (denoting one or more occurrences of set S)
- S? (denoting the optional occurrence of set S)
- S$\{min, max\}$ (denoting at least min and at most max occurrences of set S)
- S&&T (denoting the intersection of set S and T)
- S−−T (denoting the difference of set S and T)
- ˆS (denoting the complement of set S with respect to the set of all strings of symbols in the language)
- $(? = S)$ (denoting positive lookahead assertion of set S, similarly, "$(?!S)$" denotes negative lookahead assertion of set S; "$(? <= S)$" denotes positive lookbehind assertion; "$(? <!S)$" denotes negative lookbehind assertion)
- (S)...\1... (denoting the backreferences, inside which "\1" match the same text as previously matched by a capturing group of S)

2.3.2 Matching Process

The three-level architecture of icGrep is proposed by Cameron et al and is shown in Figure 2.5 [12]. There are corresponding optimizations for each level, in the form of pass. Users can chain new passes into arbitrary level of Parabix regular expression match tool. There are several analysis and transformation passes in each level, so it is users' responsibility to make sure all the passes interact with each other correctly. For passes in the first

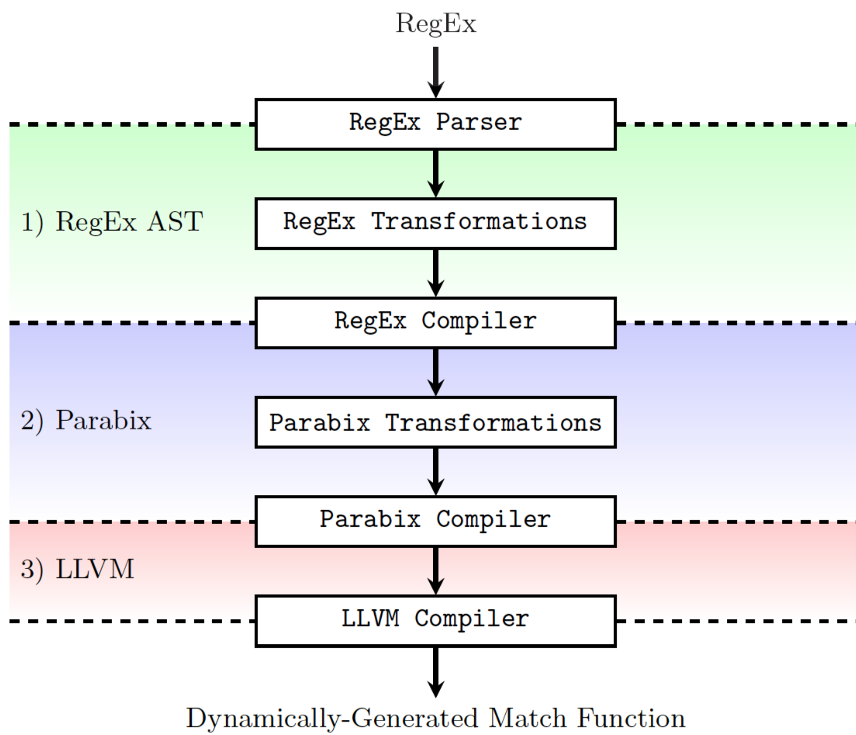


Figure 2.5: Architecture for icGrep [12]

two levels, all of them are not required to be executed in a particular order. The description of the passes for each level is as following:

- Regular Expression Pass

  There are several passes in the level of regular expression. The "remove nullable prefix (suffix)" pass is to remove any prefix (suffix) that is unnecessary in the search process. For example, when searching for lines containing "a^*cdb^*", all we need to do is searching for lines containing "cd". Therefore, the prefix "a^*" and suffix "b^*"will be removed by corresponding passes. The "simplification optimization" pass is to flatten different types and hierarchies in the RE level. For example, the regular expression "$(a\{2,3\})\{4,5\}$" will be transformed into "$a\{8,15\}$"; the alternation hierarchy of "$[0-9]|[a-z]|[A-Z]$" will be flattened and all the character classes inside it will be combined into one: "$[0-9a-zA-Z]$".

- Parabix Pass

  There are some useful passes in the Parabix level. The "dead code elimination (DCE)" pass will remove the blocks that will never be executed. The "common subexpression elimination (CSE)" pass will remove common subexpressions in order to eliminate redundant instructions and simplify the program. What's more, the "if optimization" pass in the Parabix level has shown significantly acceleration to the program. It is in the form of "if E : S". In the block-at-a-time code, if the computation of E for all the bits positions in the block is zero and there are no coming bits from the carry queue, the computation of the block of S will be omitted.

- LLVM Pass

  The LLVM Pass framework is an important part of the LLVM system. Some of passes are just to analyze the program and collect useful information, while others are used to do transformations or optimizations. Users can chain new passes into the compiler framework. In the LLVM platform, the PassManager orders passes to satisfy the dependencies. To find the dependencies exist between the various passes, each pass can declare the set of passes that are required to be executed before the current pass, and passes which are invalidated by the current pass.
  Here are some optimization passes that are useful for a wide variety of code. The "reassociate" pass reassociates expressions into a new form that can be designed to promote better constant propagation. For example, it will transform the expression "$1 + (a + 2)$" to "$a + (1 + 2)$". The "gvn" pass numbers global value, so that it can eliminate redundant instructions and common subexpressions. The "simplifycfg" pass simplifies the control flow graph. To be more specific, it will eliminate the basic block that has no predecessors, PHI nodes that have only one predecessor, the basic block that only contains one single unconditional branch. In the meanwhile, if one basic

```
     input text              parabix is a framework to process streaming text
  M1=Advance(CC[p])"         .1.........................1....................
M2=MatchStar(M1, CC[a-z])    .1111111...................1111111..............
 M3=Advance(M2 ∧ CC[r])      ...1........................1...................
```

Figure 2.6: Match "p[a-z]*r" in the text stream

```
      input text              ni3haoma3 ni3menhao
      CC1=CC(ni3)             ..1.........1......
      CC2=CC(hao)             .....1............1
     m0 = Initial             1..1..1...1..1..1..
       NonFinal               11.11.11..11.11.11.
m1 = ScanThru(m0, NonFinal)   ..1..1..1...1..1..1
  m2 = Advance(m1 ∧ CC1)      ...1.........1.....
m3 = ScanThru(m2, NonFinal)   .....1.........1...
  m4 = Advance(m3 ∧ CC2)      .....1.............
```

Figure 2.7: Match "nihao" in the Chinese text stream

block has only one predecessor and the predecessor has only one successor, the pass will merge the basic block with its predecessor. The "instcombine" will combine several redundant instructions into one. For example, the instructions "%B = add i8 %A, 3" and "%C = add i8 %B, 4" will be combined into "%C = add i8 %A, 7" [1].

There are two examples that match the corresponding regular expressions. Figure 2.5 shows the matching process of "p[a-z]*r" in the text stream which only contains one single byte for each character in UTF-8 encoding. Figure 2.6 shows the matching process of "nihao" in multiple bytes of data sequences. This example also shows the usage of some helper streams, such as "Initial", "NonFinal", etc. IcGrep has shown its performance compared to conventional command-line regular expression search tools. Table 2.1 shows seconds took per GB of input file of icGrep vs two competitors, pcre2grep and ugrep of the ICU (International Component for Unicode). All the test cases are run on an Intel i7-2600 using generic 64-bit binaries for each engine [12].

	icGrep			
Expression	**SEQ**	**MT**	**pcre2grep**	**ugrep541**
Alphanumeric #1	2.4 – 5.0	2.1 – 4.4	8.2 – 11.3	8.8 – 11.3
Alphanumeric #2	2.3 – 4.9	2.0 – 4.1	209.9 – 563.5	182.3 – 457.9
Arabic	1.5 – 3.4	1.2 – 2.6	7.5 – 270.8	8.9 – 327.8
Currency	0.7 – 2.1	0.4 – 1.4	188.4 – 352.3	52.8 – 152.8
Cyrillic	1.6 – 3.9	1.3 – 2.8	30.5 – 49.7	11.2 – 20.1
Email	3.0 – 6.9	2.7 – 6.4	67.2 – 1442.0	108.8 – 1022.3

Table 2.1: Matching Time for Complex Expressions (s / GB) [12]

2.4 Repetition Type

Repetition in regular expression is in the form of "$e\{min, max\}$", in which "e" is a regular expression, "min" represents the minimal repetition times for the regular expression and "max" represents the maximal times the "e" appears. They must be non-negative integers, and the maximal value can't be less than the minimal value. If we omit the "max", then "$e\{min,\}$" means the maximum of matches can be infinite. If we omit the comma as well, then "$e\{min\}$" means the maximum of matches is equal to the minimal value. It is same as conjunction of min "e"s. By the way, "$e\{0,1\}$" is same as "e?", "$e\{0,\}$" is same as "e^*", and "$e\{1,\}$" is same as "$e+$". The bounded repetition is that the times of repetition can't be unlimited, while the unbounded repetition is the opposite case.

2.5 Review of Finite Automata Theory for IcGrep

2.5.1 Local Language

A language L of alphabet A^* is local if there are three subsets "First", "Final", and "Follow" such that

$$\text{L} = (\text{ First A}^* \wedge \text{A}^* \text{ Final }) \text{ - A}^* \text{ Follow A}^*[2], \tag{2.2}$$

where the set "First" included all the first characters of the words in the language L, the set "Final" includes all the final characters, and the set "Follow" includes all the factors of two of all the words. For local language, all words can be decided whether they belong to the language or not only by these three sets.

If a language is not local, then we can replace each character into a new linearization form, where each symbol can occur at most once. Every linear expression represents a local language [2]. The local language can be checked more efficiently compared to the normal regular expression.

2.5.2 Automata Construction

A finite automaton has many states and it will process the incoming strings. As it is consuming the characters in the string consecutively, it will switch from one state to another. Different states are often represented by circles in the diagram. In particular, the accepting states are marked with double circles. The arrows in the figure show the transformation process between states.

It has been proved that each regular expression can be represented by one equivalent NFA(or DFA) and vice versa. There are many straightforward or complex algorithms to represent regular expression with automata, such as Thompson construction [19], Glushkov construction [15], follow construction [16], and the Antimirov construction [8]. The first

two algorithms are described in more detail below, and both of them are proved to run in cubic time of regular expression size [19, 15]. Therefore, lots of optimal methods are proposed to reduce the time complexity and simplify the automata. Brüggemann-Klein et al fine-tune the the recursive functions in the Glushkov construction, and regulate that the regular expression must be in less ambiguous form [10]; The follow construction is to apply an ϵ removal algorithm based on the directly construction, and the NFA constructed in this way is proved to be a quotient of the position automaton with respect to the equivalence given by the follow relation [16]; The Antimirov construction, which is also named partial derivative construction, is based on the concept of Brzozowski's derivatives [8, 11, 18]. The NFAs constructed by the three ways will be ϵ free, which means that they don't contain empty word transitions. However, all of them are proved to be combinations of minimization and epsilon removal algorithms based on Thompson construction [7]. This unified idea of related algorithms serves to simplify the construction of NFA.

Thompson Construction

Thompson construction is the most straightforward implementation of NFA construction. It regulates that there are just one initial state q and only one final state s in the NFA. The basic idea is to build the NFA from its subexpressions recursively. According to Aho et al., the rules are shown in Figure 2.8 [19].

Glushkov Construction

Compare with Thompson's construction, the Glushkov construction are totally "ϵ free", which means the NFA constructed by Glushkov doesn't include empty word transitions. It has already been proved that the Glushkov automata can be constructed by simply applying ϵ-removal to the corresponding Thompson automata [7]. It will definitely reduce the table size when matching the regular expressions.

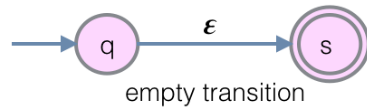
There are three steps to construct a NFA based on the regular expression:

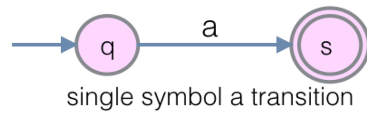
1. Convert the regular expression E into linear expressions E' just by simply replacing each character to a unique one, so that each symbol must occur at most once.

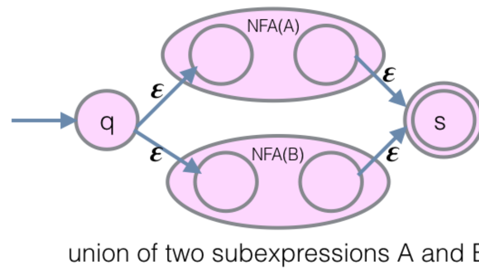
2. Compute the first characters, last characters, and the two factors of all the words w in the language L recursively based on the linear expression E'. According to these three sets, we keep adding transitions and states, and finally get a deterministic finite automaton that recognizes E'.

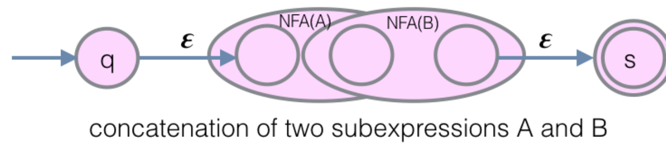
3. Convert the DFA back to NFA by replacing the characters to the corresponding occurrence of letters.

Suppose the size of a regular expression is the number of symbols it contains, which is denoted by n. The above-mentioned implementation of Glushkov construction takes









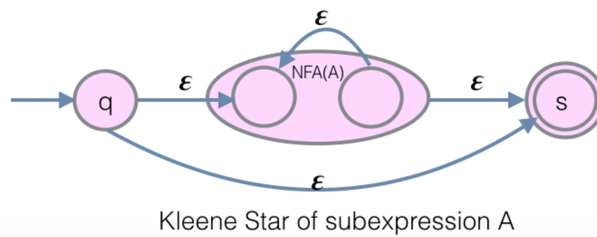


Figure 2.8: Rules for Thompson Construction

cubic time of n, and it is decided by the second step, computing the three sets. Most of the recursive operations can be done from O(1) to O(n^2) time, except calculating the two factors of all words for the regular expressions of repetition type. Therefore, Brüggemann-Klein et al proposed an optimal algorithm that aims at the recursive definition of follow sets in [10]. They require reconstruction of the regular expressions into star normal form. Besides this, many other similar quadratic algorithms are put forward, and each takes a different path toward the very same goal, simplify the follow function.

2.5.3 Star Normal Form

A new form of regular expression, *star normal form*, is defined by Brüggemann-Klein et al [10]. A regular expression is defined to be in star normal form if and only if each starred subexpression H^* in this regular expression satisfies these two conditions [10]:

1. follow(H, last(H)) $\wedge$ first(H) $= \varnothing$
2. $\epsilon \notin$ L(H)

The intuition behind the star normal form is that if a starred subexpression H^* breaks these conditions, then H itself is already in repetition type. Therefore, if we enclose it with Kleene Star, it will become ambiguous. It has been proved that with star normal form of regular expression, we can construct Glushkov NFA in quadratic time. The algorithm is described as follows [10]:

1. Convert an arbitrary regular expression into star normal form in linear time.
2. Construct the NFA for the new star normal form regular expression based on the Glushkov construction. This step takes quadratic time.

The linear time conversion algorithm into star normal form will be introduced in Chapter 4. Here we take a look at an example in advance to show the potential advantages of star normal form, and omit the process of calculation temporarily. The star normal form of regular expression "$(a^*b^*)^*$" is "$(a+b)^*$". In Parabix matching engine, we can search regular expressions more efficiently with this new form. It can reduce the complexity of patterns that have starred subexpressions, so that it can reduce the occurrences of while loops in *Pablo primitives* [4].

We want to add this transformation as an optimization part of the Parabix regular expression match tool. To ensure the correctness of the program, it is very important to preserve unambiguity during the transformation. There are two kinds of unambiguity as follows. A regular expression is defined to be weak unambiguous if there is only one path through the expression that matches the word in the language [13]. For example, let's consider the regular expression "$(a+b)^*aa^*$". In order to mark the position of each character,

we linearize the expression into the new form "$(a_1+b_1)^*a_2a_3^*$". Then if we want to match the word "aaa", there are three possible ways to denote the paths: $a_1a_1a_2$, $a_1a_2a_3$, and $a_2a_3a_3$. Obviously, this regular expression is not weak unambiguous. On the other hand, the strong unambiguity is related not only to symbols, but also to operators [13]. For example, if we want to match the word "aa" against the regular expression "$(a^* + \ b^*)^*$", there are two possible ways. The first way is having one "a" inside the bracket, and then replicating it one more time by the outer star. The other way is directly getting two "a"s inside the bracket by the inner star. Apparently, this expression is weak unambiguous, because each symbol is unique. However, if we count the operator in, then it will not be strong unambiguous. It has been proved that these two types of unambiguity have some relationship with the star normal form. A regular expression E is defined to be strong unambiguous as long as these three conditions are satisfied [10]:

- E is weak unambiguous.
- E is in star normal form.
- Empty word ϵ can be denoted unambiguously in each subexpression of E.

Chapter 3

Design Objective

3.1 Accelerate Bounded Repetition

3.1.1 Log2 Technique for Fixed-length Bounded Repetition

The log2 technique is utilized to accelerate the matching of bounded repetition of a RE that is byte length. Suppose there is a bounded regular expression in the form of E{m, n}, we treat it as the concatenation of two parts: E{m} and E{0, n - m} in order to utilize the log2 technique. Then the match problem will be solved in two steps accordingly:

$$M_1 = \text{Match}(M_0,\ E\{m\})$$

$$M_2 = \text{Match}(M_1,\ E\{0,\ n - m\})$$

We denote the operation that setting the markers directly prior k characters within the marked bits of regular expression E as Prior(C, k). It is computed recursively as follows:

$$\text{Prior(C, 1)} = \text{CC(E)} \ll 1$$

$$\text{Prior(C, 2)} = \text{Prior(C, 1)} \wedge (\text{Prior(C, 1)} \ll 1)$$

$$\text{Prior(C, 4)} = \text{Prior(C, 2)} \wedge (\text{Prior(C, 2)} \ll 2)$$

$$\text{Prior(C, 8)} = \text{Prior(C, 4)} \wedge (\text{Prior(C, 4)} \ll 4)$$

......

$$\text{Prior(C, k)} = \text{Prior(C, [k / 2])} \wedge (\text{Prior(C, }\lceil \text{k / 2} \rceil) \ll (\text{k - }\lceil \text{k / 2} \rceil))$$

For the first part, we can use the Prior operation to directly set the markers m bits ahead within the marked positions of character class E. Then M_1 can be calculated as following:

$$M_1 = \text{Prior(E, m)}$$

To calculate the second part Match(M_1, E{0, n - m}), we first find all the reachable positions by any number of repetitions of E, which is achieved by the operation MatchStar(M_1, CC(E)). Next, we should filter out the positions that are more than n - m bits ahead of marked positions in M_1, which are preceded by at least n - m consecutive zero bits in M_1. Then M_2 can be calculated as following:

$$M_2 = \text{MatchStar}(M_1, \text{CC(E)}) \wedge \neg\text{Prior}(\neg M_1, \text{n - m})$$

The algorithm only uses a fixed number of bit-wise operations to match the bounded repetition of regular expression. The time complexity of this algorithm is $\lceil \log_2 m \rceil + \lceil \log_2 (n - m) \rceil$.

For example, we want to find all the matches of "$[a-z]\{5, 9\}$" in a text stream. Suppose we set initial marker stream M_0 all ones for every position, then

$$M_1 = \text{Match}(M_0, \text{[a-z]}\{5\}) = \text{Prior}(\text{CC}[a-z],\ 5)$$

$$M_2 = \text{MatchStar}(\text{M}_1,\ \text{CC}[a-z]) \wedge \neg\text{Prior}(\neg M_1,\ 4)$$

, where Prior(CC[a-z], 5) is calculated by the following equations:

$$\text{Prior}(\text{CC}[a-z],\ 5) = \text{Prior}(\text{CC}[a-z],\ 2) \wedge (\text{Prior}(\text{CC}[a-z],\ 2) \ll 3)$$

$$\text{Prior}(\text{CC}[a-z],\ 2) = \text{Prior}(\text{CC}[a-z],\ 1) \wedge (\text{Prior}(\text{CC}[a-z],\ 1) \ll 1)$$

$$\text{Prior}(\text{CC}[a-z],\ 1) = \text{CC}[a-z] \ll 1$$

, and Prior($\neg M_1$, 4) can be computed recursively in the same way.

3.1.2 Extend the application of Log2 Technique

The log2 technique is utilized to accelerate the matching of bounded repetition of a RE that is byte length. Our target is to extend the application to more general cases, and it means to make sure the repetition part only takes one bit position in the bitstream for those cases. Therefore, we need to propose a different pipeline, where generates suitable deletion masks and delete the corresponding bits in the bitstreams.

3.2 Add Star Normal Form Pass

We want to transform an arbitrary regular expression into star normal form, so that it can denote the corresponding language more unambiguously. The new form has the exact same Glushkov automata with the original one, which guarantees the correctness of the matching program.

`input text`	`baaaabccdbcdffabcabcabcccc`
$Advance(p)$	`..1111.........1..1..1....`
f	`.....11...1....11.11.11...`
$MatchStar(Advance(p), f)$	`.....111.......111111111..`
$Advance(s)$	`.......11..1.....1..1..111`
$match$	`.......1.........1..1..1..`

Figure 3.1: Match "`abc`" by the pipeline enlightened by Glushkov automaton

In particular, the transformation will simplify the regular expressions whose subexpressions are in Kleene Star form. The Parabix framework will generate Pablo while loop to deal with the Kleene Star. For each step in this while loop, the reachable position will be kept, but the positions matched by previous steps will be removed. Once there are no more remaining bits in the marker stream, we terminate the loop. The match result will be the bitwise-or of the outputs at each step. Apparently, the star normal form with less Kleene Star subexpressions will definitely accelerate the matching process and we realize this transformation in the form of pass.

3.3 Glushkov Construction's Application to Parabix Regular Expression Matching

The Glushkov construction also has some application on the Parabix regular expression match engine. If we have a text stream T defined in the language L, suppose we define a bit stream f such that $f(i) = 1$ if and only if the pair $(T(i-1),\ T(i))$ is factor of length 2 of the regular expression. What's more, suppose we also form bitstreams $p(i) = 1$ if T(i) belongs to fist characters of all words in the language L, and define $s(i) = 1$ if T(i) belongs to last characters.

Then all the matches to language L can be found using the following equation:

$$\text{Match} \ = \ \text{MatchStar (Advance(p), f) \& Advance(s)}. \tag{3.1}$$

Figure 3.1 is an example shows the matching process of "*abc*" with the pipeline enlightened by Glushkov automaton.

However, for some ambiguous cases, this equation might not always be correct. Figure 3.2 shows the matching process of regular expression "*aba*". The result is incorrect because the sets of first and final intersect in this case. Therefore, we restrict our pipeline for unambiguous patterns, which recognize the local language only. For those ambiguous patterns, we may need to add more information to find the exact match. For example, besides creating the marker stream for all factors of length two, we also form the marker stream for length-three.

`input text`	`bababaaaababbbb`
$Advance(p)$	`..1.1.1111.1...`
f	`.11111...111...`
$MatchStar(Advance(p), f)$	`..11111111111..`
$Advance(s)$	`..1.1.1111.1...`
$match$	`..1.1.1111.1...`

Figure 3.2: Match "`aba`" by the pipeline enlightened by Glushkov automaton

Chapter 4

Implementation

4.1 Bounded Repetition

4.1.1 UTF-8 to UTF-32 Pipeline

Bounded Repetition of Regular Expressions whose length is UnicodeUnit

We want to extend the log2 technique so that it cannot only support a bounded repetition type for a single character or character class, but also the regular expression whose length is one defined in the Unicode space. If the inputs are in the form of UTF-8, then Unicode unit-length expressions might be one to four bytes. Therefore, we need to do UTF-8 to UTF-32 conversion, which would internally apply deletion to the 21 streams being produced.

UTF-8 to UTF-32 Transformation

The UTF-8 encoding has variable length, from one to four. Table 4.1 shows the structure of UTF-8 encoding. The character x represents the corresponding bits in the code point.

Characters for each language occupy four bytes in UTF-32 encoding system. The major disadvantage of UTF-32 is that it wastes the space, since the most frequent characters only occupy one or two bytes in other encoding systems. The value of bits in UTF-32 is equal to the Unicode code point value. To realize the transformation, we can do a simple mapping from bits of UTF-8 to the corresponding positions in the UTF-32 as Figure 4.1. After

Number of bytes	Bits for Code Point	Code Point Range	Byte1	Byte2	Byte3	Byte4
1	7	U+0000-U+007F	0xxxxxxx			
2	11	U+0080-U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800-U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000-U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 4.1: The Structure of UTF-8 Encoding

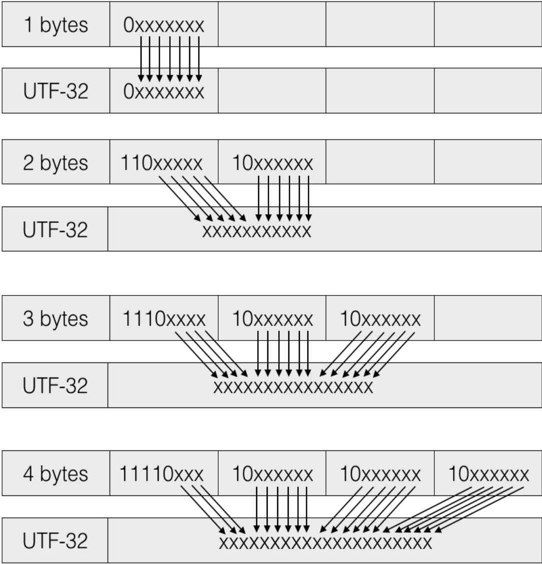


Figure 4.1: Transformation from UTF-8 to UTF-32

transformation, we can use 21 basis bit stream to represent the encoding information of the character(there are only 21 streams instead 32 required; the high 11 bits of every UTF-32 code unit are always zero). The leading bytes for multiple bytes character will be marked as useless bits in the deletion mask, after which would apply deletion logic to the 21 streams being produced. Until now, every character in the Unicode space will only take one bit position in the marker stream.

Pipeline

When a regular expression includes repetition of a RE that is Unicode unit-length, then go to a basic pipeline like the list 4.1 and Figure 4.2.

$$
\begin{aligned}
\text{ByteData} &= \text{MMapSource(FileName)} \\
\text{BasisBits} &= \text{S2P(ByteData)} \\
\text{LineBreaks} &= \text{UnicodeLineBreaks(BasisBits)} \\
\text{(U8u32Bits, DelMask, NegDelMask)} &= \text{U8U32(BasisBits)} \\
\text{LB_Del} &= \text{ParallelBitsDeletion(LineBreaks, DelMask)} \\
\text{U32_Del} &= \text{ParallelBitsDeletion(U8u32Bits, DelMask)} \\
\text{Matches_Del} &= \text{RE_compiler}<\text{regexp}>\text{(U32_Del, LB_Del)} \\
\text{Matches} &= \text{ParallelBitsDeposit(Matches_Del, NegDelMask)} \\
\text{MatchedLines} &= \text{ScanMatch(ByteData, LineBreaks, Matches)} \\
& \text{StdOutSink(MatchedLines)}
\end{aligned}
\tag{4.1}
$$

First, the MMapSource kernel reads the input file, and uses the byte streams to represent the original input file. Next, the S2P kernel, which means serial to parallel, transforms the byte stream into eight parallel bit streams. Eight parallel bit streams are then passed into the UnicodeLineBreaks kernel, in which we use the Pablo primitives to calculate the line breaks of the input file. Then we need to go to the different pipelines based on the regular expression type. If the input regular expression includes any bounded repetition of Unicode unit length, then we first pass the eight basis streams into a pablo kernel, which uses the Pablo primitives to transform the UTF-8 to UTF-32 form. The output of this kernel is 21 U8U32 basis bit streams, where the useless bits marked by deletion mask. To remove those marked bits, we apply deletion operations on the stream sets, and the basic pipeline for the deletion is shown in Figure 4.3. We do the same deletion operations on the Unicode line breaks stream, then pass the new line break stream and the 21 U32 basis bit streams into icGrep kernel. This kernel also belongs to Pablo kernel, and by doing a series of operations on the bitstreams, it outputs a match stream which marked all the matches of the input file. If we only need the count of the matches, then we just apply "popcount" operation on the match stream. Otherwise, we apply ParallelBitsDeposit logic on the match stream to get all the deleted bits back as Figure 4.4 shows, so that the match stream is aligned with the original line break stream. To get the information of matched lines in the input file, we pass the original line break stream and the byte stream and the aligned match stream into the ScanMatch kernel. Those matched lines are finally passed to the StdOut where it is printed as standard output.

If the input regular expression doesn't include any bounded repetition type, then we just go through a basic pipeline, in which the icGrep kernel accepts the eight basis bit streams and the original line break stream. The corresponding output stream is match stream.

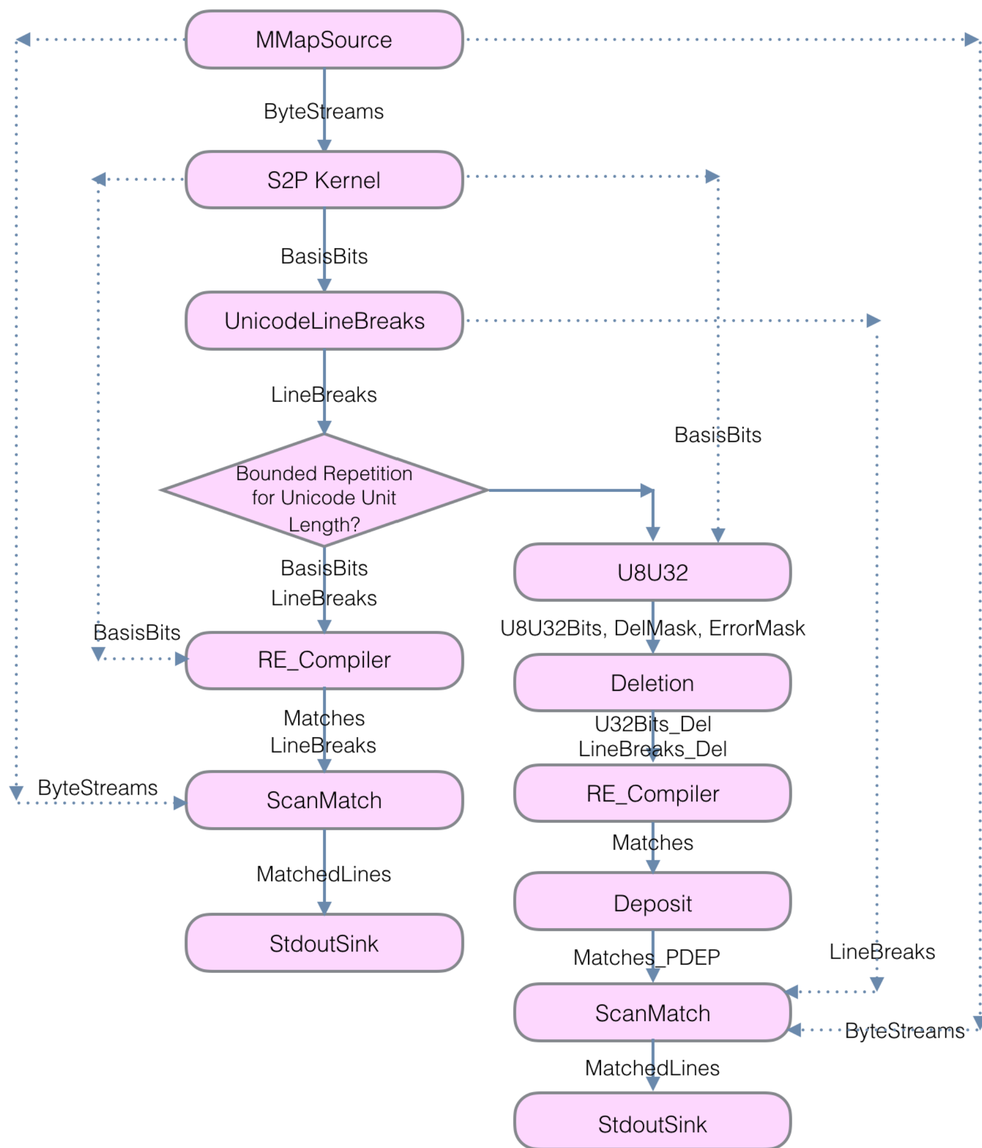


Figure 4.2: Pipeline for icGrep

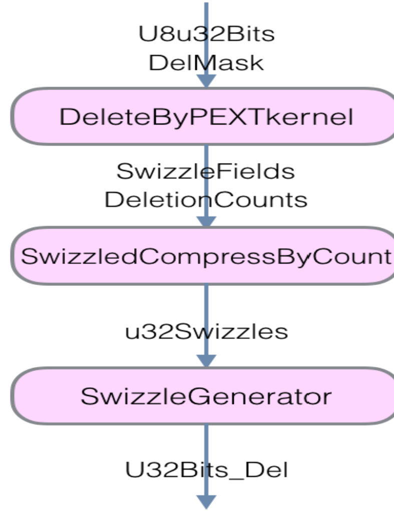


Figure 4.3: Pipeline for deletion on U8U32 bits

With the information of match stream, line break stream and byte stream, we can obtain the matched lines. Finally we use the StdOut kernel to print them.

There are two kernels within the deletion part:

$$\begin{aligned}\text{(u32Swizzles)} &= \text{SwizzledDeleteByPEXTkernel(U8u32Bits, DelMask)} \\ \text{U32Bits} &= \text{SwizzleGenerator(u32Swizzles)}\end{aligned} \tag{4.2}$$

There are three kernels within the deposit part:

$$\begin{aligned}\text{SwizzleMatches} &= \text{SwizzleGenerator(Matches)} \\ \text{(SwizzleMatches_PDEP)} &= \text{PDEP(SwizzleMatches, NegDelMask)} \\ \text{Matches_PDEP} &= \text{SwizzleGenerator(SwizzleMatches_PDEP)}\end{aligned} \tag{4.3}$$

In the deletion(deposit) part, we need to delete(deposit) the corresponding bits in the input stream set based on the mask. To realize this goal, there is one necessary operation: swizzle [4]. The swizzle operation is to transform a set of bit streams into a swizzled form. For example: consider the 4 streams (32 bits each) in figure 4.5, and swizzled outputs using a field width of 8 are shown in Figure 4.6.

In swizzled form, one "swizzle field" each from a set of streams is grouped together to be processed as a unit using SIMD operations. Before calling PEXT deletion(PDEP deposit) kernel, we can only get the streams that have fields of zeroes of variable lengths spaced at irregular intervals, so it is unrealistic to delete(deposit) the masked bits in parallel. By swizzling the result, we can use the SwizzledDeleteByPEXT(PDEP) kernel to perform the same deletion(deposit) instruction on the entire length of a stream. In particular,

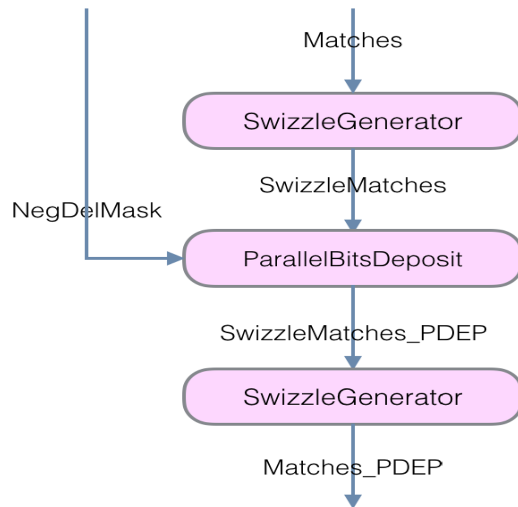


Figure 4.4: Pipeline for deposit on Matches Stream

```
Stream 1  abcdef00 ghi00000 jk000000 lmnop000
Stream 2  qrstuv00 wxy00000 z1000000 23456000
Stream 3  ABCDEF00 GHI00000 JK000000 LMNOP000
Stream 4  QRSTUV00 WZY00000 Z1000000 23456000
```

Figure 4.5: Four streams before swizzling

```
Swizzle 1  abcdef00 qrstuv00 ABCDEF00 QRSTUV00
Swizzle 2  ghi00000 wxy00000 GHI00000 WZY00000
Swizzle 3  jk000000 z1000000 JK000000 Z1000000
Swizzle 4  lmnop000 23456000 LMNOP000 23456000
```

Figure 4.6: Four streams after swizzling

Kernel Name	Items Processed	CPU Cycles	Cycles Per Item
MMapSource	1.97e+07	1.08e+06	0.05
S2P	1.97e+07	1.33e+07	0.68
UnicodeLineBreaks	1.97e+07	4.07e+06	0.21
U8U32	1.97e+07	2.28e+07	1.16
PEXTdel_LB	1.97e+07	4.21e+06	0.21
Swizzle	1.20e+07	2.48e+06	0.21
PEXTdel_UTF32	1.97e+07	2.14e+07	1.09
Swizzle	1.20e+07	8.35e+06	0.70
IcGrep	1.20e+07	2.42e+06	0.20
MatchedLines	1.20e+07	2.04e+06	0.17
Popcount	1.20e+07	1.62e+06	0.14

Table 4.2: Kernel Cycles for matching ".{4}" on the 19 MB XML file by U8U32 Pipeline

the SwizzledDeleteByPEXT kernel will swizzle the input itself before committing the PEX deletion operations. Finally, we must swizzle back to the correct form after being applied SIMD instructions, which are performed by the SwizzleGenerator kernel. The deletion logic to 21 U8U32 basis bit streams is the same as to LineBreaks stream.

4.1.2 Multiplexed Character Classes Pipeline

When the times of repetition are less than some specific value, the running time of our pipeline may take more time than regular pipeline. Table 4.2 shows the kernel cycles information about matching a pattern ".{4}" on a 19 MegaBytes Wikibooks XML file. We find out that it takes time to transform 8 basis bit streams to 21 basis bit streams and the deletion kernel applied on the 21 basis bit streams will slow down the program as well. To remove the overhead, we apply deletion logic to the character class streams, using the multiplexed character classes concept instead. For most of the cases, the number of character class streams will not be greater than 3. The deletion on these limited character class streams and omitting the transformation from UFT-8 to UTF-32 will definitely reduce the overhead.

Multiplexed Character Classes

In order to get fewer character class bit streams for arbitrary regular expressions, we use the idea of multiplexed character classes. The first step is to find the exclusive and collectively exhaustive character classes, each of which are encoded with and represented by consecutive numbers, and finally compute each bit of the set of multiplexed basis bit streams. For example, the regular expression "[a-kp-u][f-mu-y]jw" includes four source character classes as follows:

CC[0]	[a-kp-u]
CC[1]	[f-mu-y]
CC[2]	j
CC[3]	w

To get exclusive and collectively exhaustive character classes, we need to calculate the breakpoints of the source character classes. The breakpoints of character classes set is defined as follows: each codepoint c such that either c is in certain character class and c-1 is not, or c-1 is in certain character class and c is not. The computation of breakpoints is in a quite straightforward way, and that is just iterating through the interval representation of each character class. For each interval (lo, hi), lo and hi+1 are breakpoints. In this case, the breakpoints are a, f, j, k, l, n, p, u, v, w, x, and z.

In the meanwhile, a bitset is computed to identify whether the current breakpoint belongs to the source character classes. Table 4.3 shows the bitsets for our example.

Table 4.3: Bitset which indicates whether the breakpoint belongs to the source CCs

breakpoints	a	f	j	k	l	n	p	u	v	w	x	z
source CC[0]	1	1	1	1	0	0	1	1	0	0	0	0
source CC[1]	0	1	1	1	1	0	0	1	1	1	1	0
source CC[2]	0	0	1	0	0	0	0	0	0	0	0	0
source CC[3]	0	0	0	0	0	0	0	0	0	1	0	0

Next, we find out all the inclusive bitsets, and try to classify the interval codepoint to the corresponding sets. In this case, there are 6 inclusive bitsets: 0000, 0001, 0011, 0111, 0010, 1010, where 0000 indicates that the corresponding interval codepoint doesn't belong to any source character classes. We encode each of the inclusive sets with consecutive numbers. Obviously, if there are N inclusive sets, then there will be $\log_2 N$ bits in the code. The 3-bit codes to represent each of inclusive character classes are as follows:

Encoding bits	Bitsets	Exclusive character classes
000	0000	[^a-mp-y]
001	0001	[a-gp-t]
010	0011	[f-iku]
011	0111	[j]
100	0010	[l-mvx-y]
101	1010	[w]

Finally, by computing each bit of the set of multiplexed encoding, we can get set of multiplexed basis streams. The three bits in our case are shown as follows:

```
bit0  [a-gjp-tw]
bit1    [f-ku]
bit2   [l-mv-y]
```

Restructure the icGrep by Multiplexed Character Classes

The normal pipeline of icGrep passes the 8(16 or 32) basis bit streams into Parabix kernel to finish the matching process. However, information redundancy is the most obvious defect of the current method. Most of the regular expressions have limited character classes, even zero in the worst cases. If we pass multiplexed character classes streams instead, then the Parabix engine will become more modular and get some expected acceleration. The 7-stage restructured icGrep pipeline is shown as follows:

$$\begin{aligned}
\text{ByteData} &= \text{MMapSource(FileName)} \\
\text{BasisBits} &= \text{S2P(ByteData)} \\
\text{LineBreaks} &= \text{UnicodeLineBreaks(BasisBits)} \\
\text{CharacterClasses} &= \text{CharClassesKernel(BasisBits)} \\
\text{Matches} &= \text{RE_compiler<regexp>(CharacterClasses, LineBreaks)} \\
\text{MatchedLines} &= \text{ScanMatch(ByteData, LineBreaks, Matches)} \\
& \text{StdOutSink(MatchedLines)}
\end{aligned} \tag{4.4}$$

Basic Pipeline

The basic pipeline of multiplexed character classes, which support the log2 technique for repetition of a RE that is UnicodeUnitLength, is like following:

$$\begin{aligned}
\text{ByteData} &= \text{MMapSource(FileName)} \\
\text{BasisBits} &= \text{S2P(ByteData)} \\
\text{LineBreaks} &= \text{UnicodeLineBreaks(BasisBits)} \\
\text{(DelMask, NegDelMask)} &= \text{DelMask(BasisBits)} \\
\text{LB_Del} &= \text{ParallelBitsDeletion(LineBreaks, DelMask)} \\
\text{CharacterClasses} &= \text{CharClassesKernel(BasisBits)} \\
\text{CC_Del} &= \text{ParallelBitsDeletion(CharacterClasses, DelMask)} \\
\text{Matches_Del} &= \text{RE_ compiler<regexp>(CC_Del, LB_Del)} \\
\text{Matches} &= \text{ParallelBitsDeposit(Matches_Del, NegDelMask)} \\
\text{MatchedLines} &= \text{ScanMatch(ByteData, LineBreaks, Matches)} \\
& \text{StdOutSink(MatchedLines)}
\end{aligned} \tag{4.5}$$

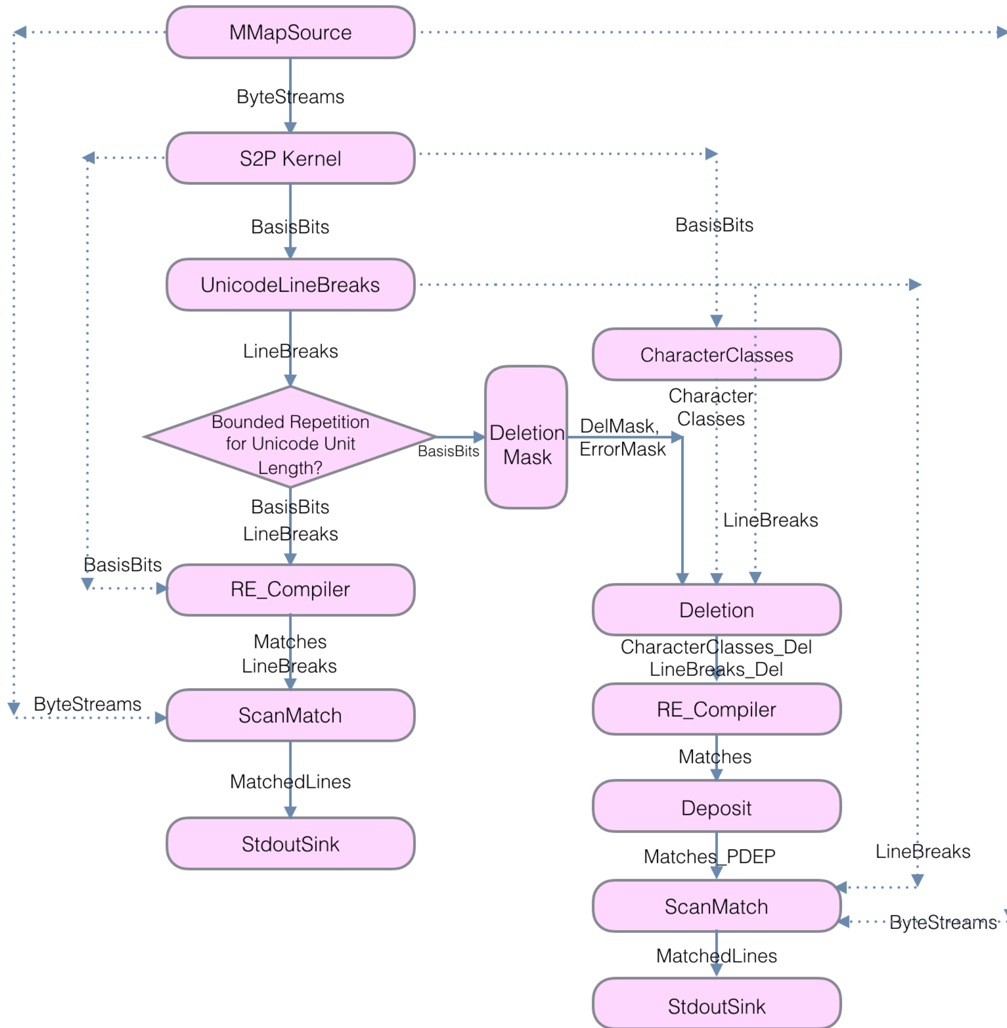


Figure 4.7: Pipeline for icGrep with multiplexed character classes

The basic pipeline that applies the multiplexed character classes concept is shown as Figure 4.7. Compared to the pipeline of UFT8-to-UTF-32, we pass in the multiplexed character class streams to the icGrep kernel instead. We have matched the pattern ".{4}" on the aforementioned 19 MegaBytes Wikibooks XML file again, and Table 4.4 shows the related kernel cycles information. We find that the multiplexed pipeline will eliminate the overhead caused by UTF8-to-UTF32 transformation, and the deletion logic applied on the limited multiplexed character classes streams has also accelerated the matching process.

Kernel Name	Items Processed	CPU Cycles	Cycles Per Item
MMapSource	1.97e+07	1.95e+06	0.10
S2P	1.97e+07	1.37e+07	0.70
UnicodeLineBreaks	1.97e+07	5.47e+06	0.28
Delmask	1.97e+07	5.16e+06	0.26
PEXTdel_LB	1.97e+07	8.69e+06	0.44
Swizzle	1.20e+07	3.84e+06	0.32
CharacterClasses	1.97e+07	3.00e+06	0.15
PEXTdel_CC	1.97e+07	7.82e+06	0.40
Swizzle	1.20e+07	3.68e+06	0.31
IcGrep	1.20e+07	3.16e+06	0.26
MatchedLines	1.20e+07	3.02e+06	0.25
Popcount	1.20e+07	2.93e+06	0.24

Table 4.4: Kernel Cycles for matching ".{4}" on the 19 MB XML file by Multiplexed Pipeline

4.2 Unbounded Repetition

4.2.1 Star Normal Form Pass

If a regular expression E is not in star normal form, then there might be subexpression in the form of H^* which breaks the condition: follow(H, last(H)) $\wedge$ firs(H) $= \varnothing$. Therefore, if all the subexpressions in the form of Kleene Star H^* are replaced with new form $H^{\circ *}$, where the new starred subexpressions satisfy the previous condition, then we can get a new star normal form expression denoted as $E^\bullet$ [10]. The definition of H° is as follows [10]:

$$
\begin{aligned}
[E = \varnothing, \epsilon] \quad & E^\circ = \varnothing \\
[E = a] \quad & E^\circ = E \\
[E = F + G] \quad & E^\circ = F^\circ + G^\circ \\
[E = FG] \quad & E^\circ = \begin{cases} FG & \text{if } \epsilon \notin L(F),\, \epsilon \notin L(G) \\ F^\circ G & \text{if } \epsilon \notin L(F),\, \epsilon \in L(G) \\ FG^\circ & \text{if } \epsilon \in L(F),\, \epsilon \notin L(G) \\ F^\circ + G^\circ & \text{if } \epsilon \in L(F),\, \epsilon \in L(G) \end{cases} \\
[E = F^*] \quad & E^\circ = F^\circ
\end{aligned}
\tag{4.6}
$$

It has been proved that the calculation of the star normal form of regular expression is based on the following two operations:

$$
\begin{aligned}
[E = \varnothing, \epsilon, a] \quad & E^\bullet = E \\
[E = F + G] \quad & E^\bullet = F^\bullet + G^\bullet \\
[E = FG] \quad & E^\bullet = F^\bullet G^\bullet \\
[E = F^*] \quad & E^\bullet = F^{\bullet \circ *}
\end{aligned}
\tag{4.7}
$$

$$
\begin{aligned}
[E = \varnothing, \epsilon] \quad & E^{\bullet\circ} = \varnothing \\
[E = a] \quad & E^{\bullet\circ} = E \\
[E = F + G] \quad & E^{\bullet\circ} = F^{\bullet\circ} + G^{\bullet\circ} \\
[E = FG] \quad & E^{\bullet\circ} = \begin{cases} F^{\bullet}G^{\bullet} & \text{if } \epsilon \notin L(F),\ \epsilon \notin L(G) \\ F^{\bullet\circ}G^{\bullet} & \text{if } \epsilon \notin L(F),\ \epsilon \in L(G) \\ F^{\bullet}G^{\bullet\circ} & \text{if } \epsilon \in L(F),\ \epsilon \notin L(G) \\ F^{\bullet\circ} + G^{\bullet\circ} & \text{if } \epsilon \in L(F),\ \epsilon \in L(G) \end{cases} \\
[E = F^*] \quad & E^{\bullet\circ} = F^{\bullet\circ}
\end{aligned} \tag{4.8}
$$

Here is an example which calculates the star normal form of $(a^*b^*)^*$:

$$
\begin{aligned}
(a^*b^*)^{*\bullet} &= (a^*b^*)^{\bullet\circ *} \\
&= (a^{*\bullet\circ} + b^{*\bullet\circ})^* \\
&= (a^{\bullet\circ} + b^{\bullet\circ})^* \\
&= (a + b)^*
\end{aligned} \tag{4.9}
$$

Just as the example shown in the equation 4.7, the star normal form of regular expression "$(a^*b^*)^*$" is "$(a+b)^*$". Following the formulas shown before, we can easily implement it in the form of pass.

We try to show all the regular expression abstract syntax trees for "$a(a^*b^*)^*b$" in the icGrep, the results are shown as following:

Listing 4.1: Regular Expression AST for "$(a^*b^*)^*$"

```
Simplifier:
(Seq[Name "CC_61" ,Rep(Name "\1" =((Seq[Rep(Name "CC_61" ,0,Unbounded),Rep(
    ↪ Name "CC_62" ,0,Unbounded)])),0,Unbounded),Name "CC_62" ])
```

Listing 4.2: Regular Expression AST for "$(a^*b^*)^*$" in Star Normal Form

```
Star_Normal_Form:
(Seq[Name "CC_61" ,Rep(Name "\1" =((Alt[Name "CC_61" ,Name "CC_62" ])),0,
    ↪ Unbounded),Name "CC_62" ])
```

What's more, we have also shown the Pablo primitives as list 4.3 and 4.4. It shows regular expression "$(a^*b^*)^*$" requires a while loop with a standard algorithm, but it will be implemented using MatchStar when StarNormalForm is applied(no while loop required), which accelerates the program.

Listing 4.3: Pablo Primitives for RE "$(a^*b^*)^*$" in Star Normal Form

```
CC_61+CC_62 = (and_45 & xor_2)
and_47 = (any & CC_61_1)
and_48 = (any & CC_62_1)
and_49 = (any & CC_61+CC_62)
ipp = pablo.Advance(and_47, 1)
and_50 = (initial & ipp)
fpp = pablo.ScanThru(and_50, nonfinal)
unbounded = pablo.MatchStar(fpp, and_49)
CC_62_2 = (and_48 & unbounded)
matchstar = pablo.MatchStar(CC_62_2, any)
```

Listing 4.4: Pablo Primitives for RE "$(a^*b^*)^*$"

```
While test:
  unbounded = pablo.MatchStar(pending, and_45)
  unbounded_1 = pablo.MatchStar(unbounded, and_46)
  not_22 = (~accum)
  and_48 = (unbounded_1 & not_22)
  pending = and_48
  or_25 = (accum | unbounded_1)
  accum = or_25
  test = pending
CC_62_2 = (and_46 & accum)
```

4.2.2 New Compile Pipeline for Local Language

After reviewing of Glushkov Construction in Chapter 2, we try to solve some specific regular expression types of local language in the similar way. In particular, we can avoid many unnecessary while loops in Parabix level when we deal with the unbounded repetition types with this new compiler pipeline. Here we need a helper function $EmptyWord(R)$, which indicates whether the empty word belongs to the regular expression. The EmptyWord(R) is calculated recursively as follows:

- EmptyWord(ϵ) $=$ true
- EmptyWord($\varnothing$) $=$ false
- EmptyWord(R$*$) $=$ true
- EmptyWord(RS) $=$ EmptyWord(R) $\wedge$ EmptyWord(S)
- EmptyWord(R $|$ S) $=$ EmptyWord(R) $\vee$ EmptyWord(S)
- EmptyWord($\neg$R) $=$ true if v(R) $= \varnothing$
- EmptyWord($\neg$R) $=$ false if v(R) $= \epsilon$

With the helper function of calculating EmptyWord, the calculation of first, final and follow sets for different regular expression types of icGrep are shown as following:

- $\text{First}(\epsilon) \ = \ \varnothing$
- $\text{First}(\varnothing) \ = \ \varnothing$
- $\text{First(a)} = \text{a} \quad \text{for all a} \in \text{L(RE)}$
- $$\text{First(RS)} = \begin{cases} \text{First(R)} \vee \text{First(S)}, & \text{EmptyWord(R)}. \\ \text{First(R)}, & \text{otherwise.} \end{cases}$$
- $\text{First}(\text{R} \mid \text{S}) \ = \ \text{First(R)} \vee \text{First(S)}$
- $\text{First}(\text{R} \ast) \ = \ \text{First(R)}$
- $\text{First}(\neg\text{R}) \ = \ \text{true} \ \text{ if v(R)} = \varnothing$
- $\text{First}(\neg\text{R}) \ = \ \text{false} \ \text{ if v(R)} = \epsilon$

- $\text{Final}(\epsilon) \ = \ \varnothing$
- $\text{Final}(\varnothing) \ = \ \varnothing$
- $\text{Final(a)} = \text{a} \quad \text{for all a} \in \text{L(RE)}$
- $$\text{Final(RS)} = \begin{cases} \text{Final(R)} \vee \text{Final(S)}, & \text{EmptyWord(S)}. \\ \text{Final(S)}, & \text{otherwise.} \end{cases}$$
- $\text{Final}(\text{R} \mid \text{S}) \ = \ \text{Final(R)} \vee \text{Final(S)}$
- $\text{Final}(\text{R} \ast) \ = \ \text{Final(R)}$
- $\text{Final}(\neg\text{R}) \ = \ \text{true} \ \text{ if v(R)} = \varnothing$
- $\text{Final}(\neg\text{R}) \ = \ \text{false} \ \text{ if } \ \text{v(R)} = \epsilon$

- $\text{Follow}(\epsilon) \ = \varnothing$
- $\text{Follow}(\varnothing) \ = \varnothing$

- $\text{Follow}(a) = \varnothing \;\; \text{for all } a \in L(RE)$
- $\text{Follow}(RS) = \text{Follow}(R) \;\vee \text{Follow}(S) \;\vee \text{Final}(R)\ \text{First}(S)$
- $\text{Follow}(R \mid S) = \text{Follow}(R) \;\vee \text{Follow}(S)$
- $\text{Follow}(R\,*\,) = \text{Final}(R) \vee \text{Final}(R)\ \text{First}(R)$

With the results of these three sets for local language, we compiled them into character class streams and then the match result will be the following:

$$\begin{aligned} \text{Match} = \;& \text{MatchStar}(\text{Advance}(\text{Marker}(\text{First}), 1),\;\; \text{Marker}(\text{Follow})) \\ & \&\;\; \text{Advance}(\text{Marker}(\text{Final}), 1) \end{aligned} \tag{4.10}$$

Chapter 5

Performance Evaluation

5.1 Bounded Repetition

Suppose the length of a regular expression is n, and the size of the input text is m. The traditional algorithms first try to convert the regular expression into a non-deterministic finite automaton by arbitrary NFA construction algorithms. In Thompson's construction, transforming the regular expression into NFA takes linear time, and removing the ϵ in the NFA will take quadratic time of n. In Glushkov construction, it'll take cubic time of n to construct an ϵ free NFA. There are a few advanced algorithms based on different concepts like star normal form, Brzozowski's derivatives etc, which can reduce the time complexity to quadratic time.

All these operations belong to the preprocessing step. In our algorithm, we compile the regular expression and don't count this preprocessing into the performance measures, because it is just an offline process. However, other Unix tools count it in, so we need to use large test files to minimize the basis.

The test cases are run on an Intel i3-5010 using generic 64-bit binaries for each engine. Table 5.1 shows the detailed hardware configuration of the test machine. Three Wikibooks XML files in three different languages are used to test the time it takes to count all the matches against n times of several regular expressions with Unicode-unit-length bounded repetition, where n varies from 4 to 1000. The bounded repetition parts of the group of regular expressions are in the form of ".{n}", "\p{greek}{n}", "\p{Lu}{n}", "\p{...}{n}", ..., and the sizes of these three files are 19MB, 54MB, and 88MB, respectively. We have compared the performance for two log2 bounded repetition pipelines of icGrep introduced previously, regular pipeline of icGrep, GNU grep, pcregrep and rg. In the meanwhile, we also tested the count with invert flag, which means selecting non-matching lines instead.

Figure 5.1 shows the performance of searching a set of regular expressions with Unicode-unit-length bounded repetition of the six different engines in three different files, where n

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	4
Thread(s) per core	2
Core(s) per socket	2
Vendor ID	GenuineIntel
CPU MHz	2100.000

Table 5.1: The Hardware Configuration of the Test Machine

Repetition times	log2 multiplexed	log2 U8U32	Normal Version	grep	pcregrep	rg
4	2.50	4.30	1.69	1.16	5.85	2.04
10	2.52	4.31	2.01	1.59	6.35	2.16
20	2.52	4.34	2.62	2.40	7.15	2.56
50	2.56	4.34	4.28	4.89	11.32	2.90
100	2.56	4.36	6.83	6.48	14.36	3.58
300	2.60	4.38	12.41	11.11	25.99	8.99
500	2.58	4.40	15.61	1420.28	36.67	17.10
700	2.60	4.39	17.49	11346.46	41.62	852.54
1000	2.58	4.38	19.40	17988.06	46.42	2467.90

Table 5.2: Averaged Matching Time for Different Repetition Times(ms/MB)

varies from 4 to 1000. Table 5.2 summarizes the averaged matching time of different regular expressions in each engine with respect to different repetition times.

Figure 5.1 shows that compared to the regular pipeline of icGrep and the GNU grep, the more times that bounded repetition part repeats, the acceleration achieved by our new bounded repetition pipelines will be more obvious. As shown in Figure 5.1, the performance of pcregrep and our three versions of different pipelines are quite stable. For a clear performance evaluation, we compared these four individually as Figure 5.2. It has shown two versions of log2 pipelines are more stable than pcregrep and regular version of icGrep, and both of them can achieve more than 20x speedup compared to pcregrep when the times of repetition are more than 1000. Furthermore, the multiplexed character classes pipeline of log2 technique is nearly two times faster than UTF8-to-UTF32 pipeline.

Figure 5.2 also shows that when the numbers of repetition are less than some specific value, the times it takes for log2 UTF8-to-UTF32 pipeline are more than most of other pipelines or conventional tools. This is because that it takes time to do the transformation from UTF8 to UTF32. When times of repetition are big enough, the overhead caused by this transformation will become not obvious. The new multiplexed pipeline is therefore proposed to reduce the overhead, and the performance evaluation shows that the theoretical

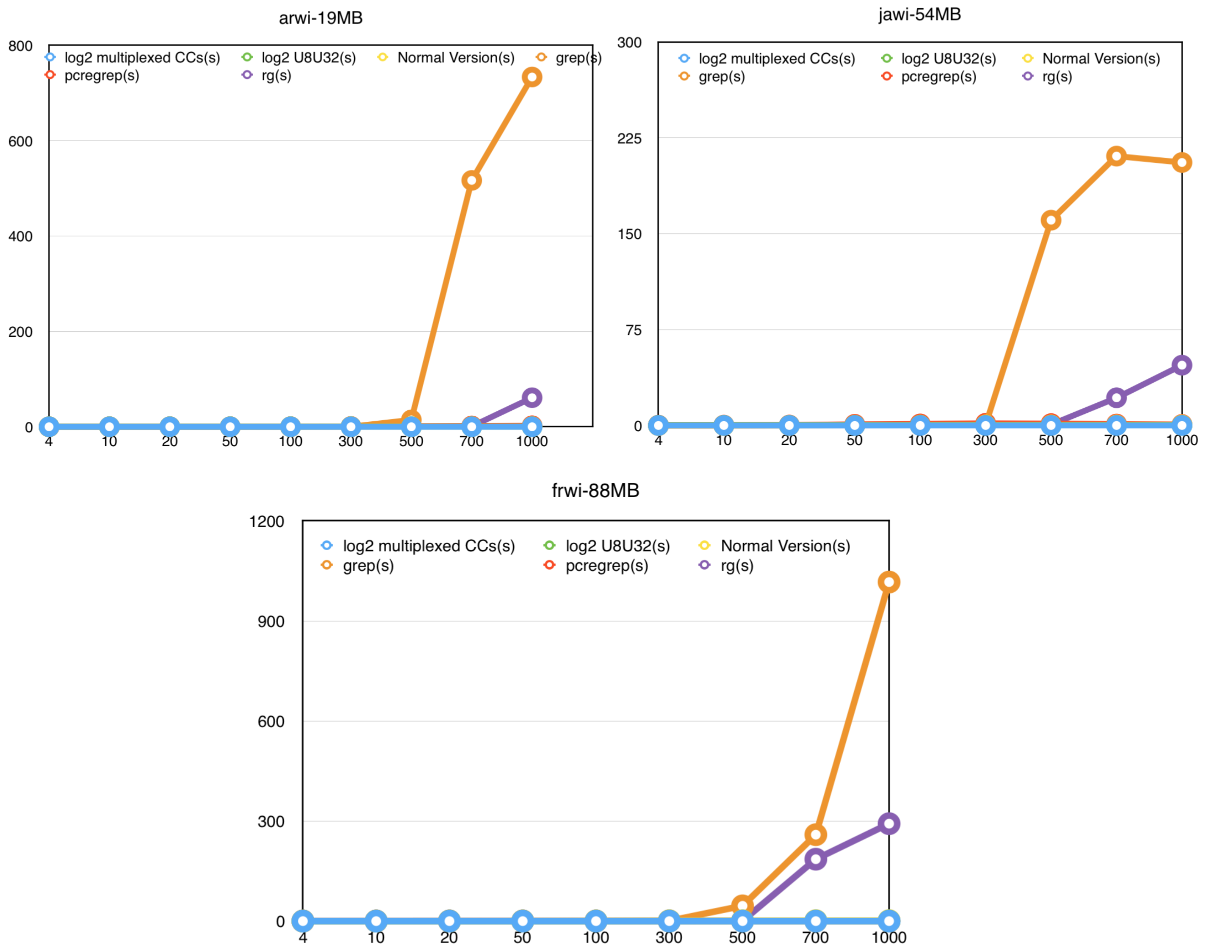


Figure 5.1: Performance on Three Different Files

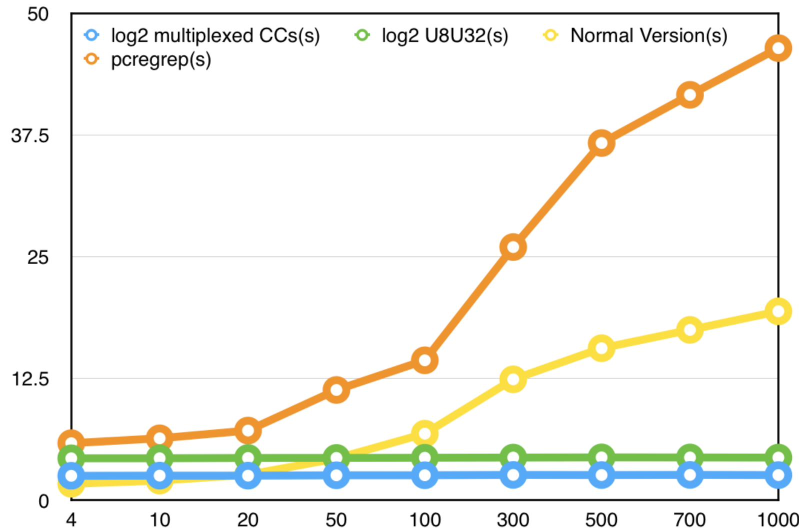


Figure 5.2: Averaged Matching Time(Log2 Pipelines vs. pcregrep)

result is in agreement with the experimental one. However, the times it takes for the new pipeline are still more than our regular version of icGrep. It's because the overhead caused by deletion kernel can't be eliminated. In the meanwhile, there are cases that the time it takes is becoming less or the increase rate is decreasing with the increasing of n, and that is because lots of matches will terminate before they reach the n times of repetition.

5.2 Unbounded Repetition

5.2.1 Star Normal Form

We have tested one bunch of regular expression which has different numbers of starred subexpressions inside the bracket with respect to three MegaBytes Wikibooks XML files. The group of regular expressions are in the following forms: "$R_1(?:R_2^*R_3^*)^*R_4$", "$R_1(?:R_2^*R_3^*R_4^*)^*R_5$", "$R_1(?:R_2^*R_3^*R_4^*R_5^*)^*R_6$", "$R_1(?:R_2^*R_3^*R_4^*R_5^*...)^*R_n$", etc, and the numbers of starred subexpressions inside the bracket are 2, 3, 4, ..., respectively. For each regular expression in the test case, we need to add a subexpression R_1 at the beginning and R_n in the end respectively in order to prevent all of the starred subexpressions being removed from the optimization passes. As the test cases before, we averaged the testing time of repeated measurements to get more reliable results. The test results for the optimized version and normal version of icGrep are shown in Figure 5.3. As the figure shows, the more starred subexpressions one regular expression includes, the acceleration achieved by star normal pass will be more obvious.

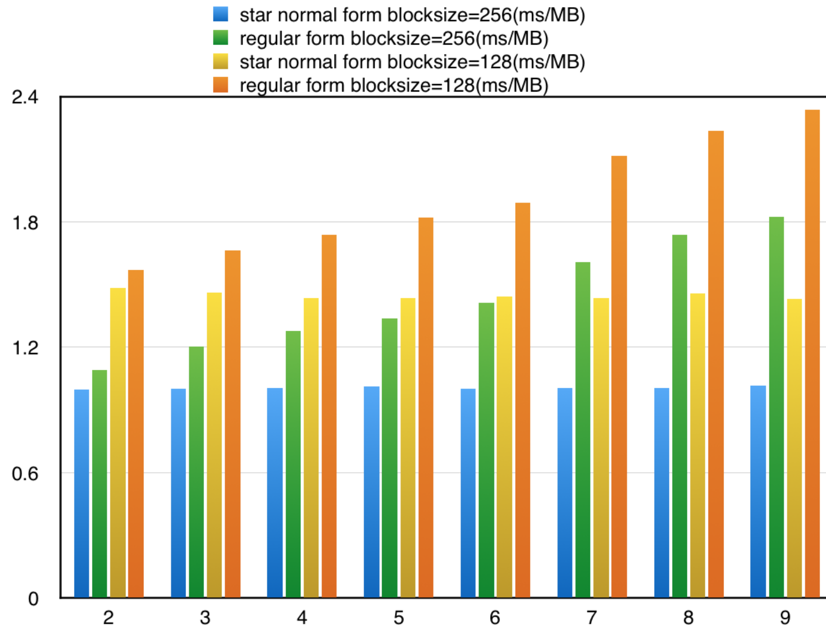


Figure 5.3: Performance for star normal form pass with respect to different numbers of starred subexpressions inside the bracket

We have also tested the regular expressions with different nested level. The group of regular expressions are in the following forms: "$R_1(R_2^*R_3^*)^*R_4$", "$R_1((R_2^*R_3^*)^*)^*R_4$", ..., "$R_1(((R_2^*R_3^*)^*)^*......)^*R_4$". The results are shown in Figure 5.4. As the figure shows, the more nested level one regular expression includeds, the acceleration achieved by star normal pass will be more obvious.

5.2.2 New Compile Pipeline for Local Language

We have tested several regular expressions with different repetition nested level, from 1 to 9, on three different MegaBytes Wikibooks XML files. In the meanwhile, we must make sure that the regular expressions can't be optimized by the star normal form pass. Figure 5.5 shows that we have achieved performance gains by the new pipeline for local language. The running time for local language pipeline is quite stable with the increasing of nested level of starred subexpressions, while the time for normal pipeline increases. This is because that the matchstar operations in the regular expression will be omitted in the new pipeline.

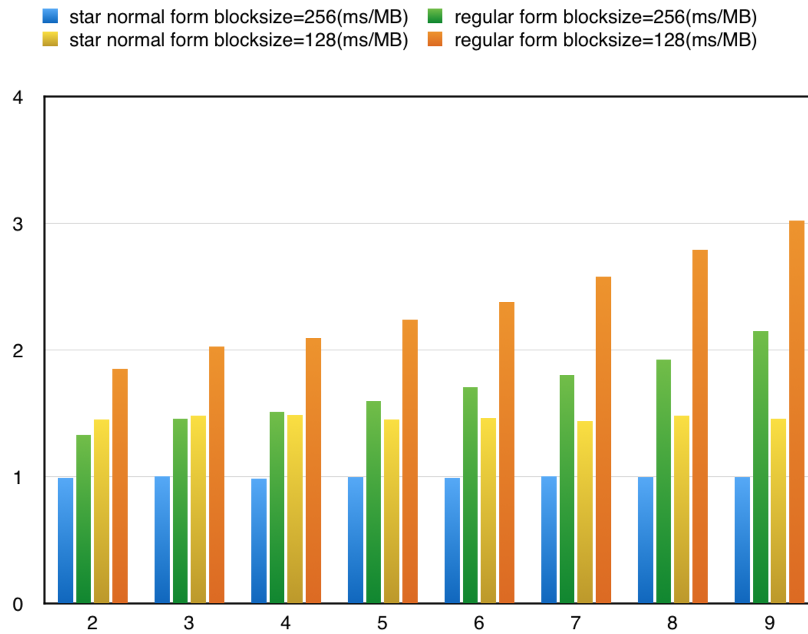


Figure 5.4: Performance for star normal form pass with respect to different nested levels of starred subexpressions

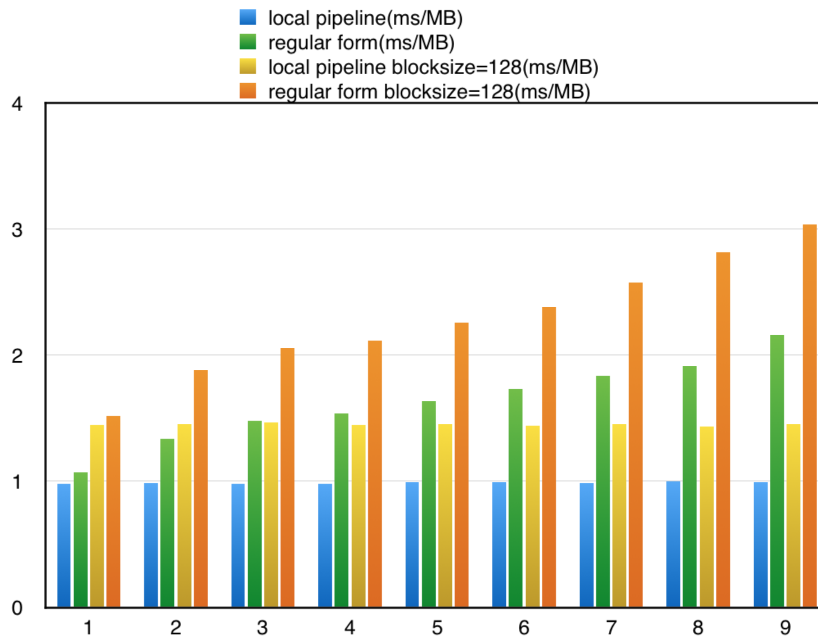


Figure 5.5: Performance for local language with respect to different nested levels of starred subexpressions

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this paper, we optimize the regular expressions with counting in the Parabix-LLVM framework. To utilize the log2 technique, we proposed the UTF8-to-UTF32 pipeline to accelerate this case. Although we have already got significant performance increases with this pipeline, we find that the overhead caused by the UTF-8-to UTF-32 transformation is becoming more obvious with the times of repetition becoming less. Therefore, the multiplexed character classes concept is proposed to minimize the overhead.

Besides bounded repetition cases, we have also reviewed the finite automata theory for application to Parabix regular expression matching, and proposed star-normal-form optimizations in RE level, which makes the RE abstract syntax tree less complex and more unambiguous. We also utilize the algorithm of Glushkov NFA construction to propose totally new compile logic for local language. Both of these two methods enlightened by automata theory have shown their performance dealing with unbounded repetition type against the conventional tools and the original icGrep pipeline.

6.2 Future Work

6.2.1 Log2 Technique for Bounded Repetition of Arbitrary Length Regular Expression

Until now, we can only utilize the log2 technique for bounded repetition of unit length type. Actually, we can extend this idea to arbitrary bounded repetition type. The basic idea is to mark all the positions that are not the first of the repetition regular expression as deletion positions. We delete all those positions so that the arbitrary bounded repetition will become unit length. Here we take a look at a specific example of matching "$abc\{3\}$" as following:

```
input text       baaaabccdbcdffabcabcabcccc
Maches to 'abc'  .......1.........1..1..1..
deletion mask    .....11........11.11.11...
new marker              .....1.......111..
```

Figure 6.1: Log2 technique for arbitrary length repetition type

We will apply log2 technique to the new marker and find the consecutive three marked bits as the steps introduced in previous sections. However, this deletion pipeline to the arbitrary Unicode length bounded repetition type is only suitable for local language, which means the alphabet in the regular expression must be used only once. When the repetition part is not local language, then the generation of deletion mask might cause ambiguity.

6.2.2 Support for Extended Regular Expression Types

The Unicode Technical Standard regulates that the regular expression engines can offer three levels Unicode support [6].

- Level 1: Basic Unicode Support. It includes all the basic needs that a regular expression engine must provide, such as the hex notation, properties, simple word boundaries, subtraction and intersection, etc.
- Level 2: Extended Unicode Support. It included the extended Unicode features, such as extended grapheme boundaries, name properties, canonical equivalents, etc. All the features in this level and level 1 are country and language independent.
- Level 3: Tailored Support. It includes the features that must satisfy the end-user's specific expectations, such as tailored punctuation, tailored grapheme clusters, context matching, etc. These Unicode features are often country and language dependent, and are only needed in some specific applications.

Until now, the icGrep can support all the Unicode features in Level 1 and part of features in Level 2. None of the features is supported in Level 3. For example, Diff and Intersect types that have different Unicode length range are not supported yet. They might need different strategies to address them.

Bibliography

[1] Llvm's analysis and transform passes. http://llvm.org/docs/Passes.html#llvm-s-analysis-and-transform-passes. Accessed: 2017-06-19.

[2] Local language (formal language). https://en.wikipedia.org/wiki/Local_language_(formal_language). Accessed: 2017-07-17.

[3] Parabix methods for bounded repetition. http://parabix.costar.sfu.ca/browser/docs/Working/bounded-rep.pdf. Accessed: 2017-09-15.

[4] Parabix technology home page. http://parabix.costar.sfu.ca/. Accessed: 2017-06-19.

[5] Portable operating system interface for unix. http://www.regular-expressions.info/posix.html. Accessed: 2017-06-21.

[6] Uts no.18: Unicode regular expressions - unicode.org. http://unicode.org/reports/tr18/. Accessed: 2017-06-20.

[7] Cyril Allauzen and Mehryar Mohri. A unified construction of the glushkov, follow, and antimirov automata. In *International Symposium on Mathematical Foundations of Computer Science*, pages 110–121. Springer, 2006.

[8] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[10] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[11] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[12] Robert D Cameron, Nigel Medforth, Dan Lin, Dale Denis, and William N Sumner. Bitwise data parallelism with llvm: The icgrep case study. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 373–387. Springer, 2015.

[13] Haiming Chen and Ping Lu. Checking determinism of regular expressions with counting. *Information and Computation*, 241:302–320, 2015.

[14] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). *URL: http://swtch. com/˜ rsc/regexp/regexp1. html*, 2007.

[15] Victor Michailowitsch Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1, 1961.

[16] Lucian Ilie and Sheng Yu. Follow automata. *Information and computation*, 186(1):140–162, 2003.

[17] Dan Lin, Nigel Medforth, Kenneth S Herdy, Arrvindh Shriraman, and Rob Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[18] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(02):173–190, 2009.

[19] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

Appendix A

Code

Listing A.1: U8 to U32 Transformation

```
void U8U32KernelBuilder::generatePabloMethod() {
    //  input: 8 basis bit streams

    const auto u8bitSet = this->getInputStreamVar("u8bit");

    //  output: 32 u8-indexed streams, + delmask stream + error stream

    cc::CC_Compiler ccc(this, u8bitSet);

    PabloBuilder & main = ccc.getBuilder();
    const auto u8_bits = ccc.getBasisBits();

    Zeroes * zeroes = main.createZeroes();

    // Outputs
    // The first 11 bits of u32 are always 0s.

    Var * u32_0[8];
    for (int i = 0; i < 8; i++) {
        u32_0[i] = main.createVar("u32_0" + std::to_string(i), zeroes);
    }

    Var * u32_1[8];
    for (int i = 0; i < 8; i++) {
        u32_1[i] = main.createVar("u32_1" + std::to_string(i), zeroes);
    }

    Var * u32_2[8];
    for (int i = 0; i < 8; i++) {
        u32_2[i] = main.createVar("u32_2" + std::to_string(i), zeroes);
    }

    Var * delmask = main.createVar("delmask", zeroes);
    Var * error_mask = main.createVar("error_mask", zeroes);

    PabloAST * ASCII = ccc.compileCC("ASCII", re::makeCC(0x0, 0x7F), main);
    PabloBuilder ascii = PabloBuilder::Create(main);
    for (int i = 1; i <= 7; i++) {
```

```
        ascii.createAssign(u32_2[i], ascii.createOr(u32_2[i], ascii.
            ↪ createAnd(ASCII, u8_bits[i])));
    }
    main.createIf(ASCII, ascii);

    PabloAST * u8pfx = ccc.compileCC("u8pfx", re::makeCC(0xC0, 0xFF), main);
    PabloAST * nonASCII = ccc.compileCC("u8pfx", re::makeCC(0x80, 0xFF),
        ↪ main);
    PabloBuilder it = PabloBuilder::Create(main);
    main.createIf(nonASCII, it);

    Var * u8invalid = it.createVar("u8invalid", zeroes);
    PabloAST * u8pfx2 = ccc.compileCC(re::makeCC(0xC2, 0xDF), it);
    PabloAST * u8pfx3 = ccc.compileCC(re::makeCC(0xE0, 0xEF), it);
    PabloAST * u8pfx4 = ccc.compileCC(re::makeCC(0xF0, 0xF4), it);
    PabloAST * u8suffix = ccc.compileCC("u8suffix", re::makeCC(0x80, 0xBF),
        ↪ it);

    //

    //
    // Two-byte sequences
    Var * u8scope22 = it.createVar("u8scope22", zeroes);
    PabloBuilder it2 = PabloBuilder::Create(it);
    it.createIf(u8pfx2, it2);
    it2.createAssign(u8scope22, it2.createAdvance(u8pfx2, 1));
    //PabloAST * u8scope22 = it2.createAdvance(u8pfx2, 1, "u8scope22");
    for (int i = 2; i <= 7; i++) {
        it2.createAssign(u32_2[i], it2.createOr(u32_2[i], it2.createAnd(
            ↪ u8scope22, u8_bits[i])));
    }
    it2.createAssign(u32_2[1], it2.createOr(u32_2[1], it2.createAnd(
        ↪ u8scope22, it2.createAdvance(u8_bits[7], 1))));
    it2.createAssign(u32_2[0], it2.createOr(u32_2[0], it2.createAnd(
        ↪ u8scope22, it2.createAdvance(u8_bits[6], 1))));
    for (int i = 3; i <= 5; i++) {
        it2.createAssign(u32_1[i + 2], it2.createOr(u32_1[i + 2], it2.
            ↪ createAnd(u8scope22, it2.createAdvance(u8_bits[i], 1))));
    }

    //
    // Three-byte sequences
    Var * u8scope3X = it.createVar("u8scope3X", zeroes);
    Var * EX_invalid = it.createVar("EX_invalid", zeroes);
    Var * del3 = it.createVar("del3", zeroes);

    PabloBuilder it3 = PabloBuilder::Create(it);
    it.createIf(u8pfx3, it3);

    PabloAST * u8scope32 = it3.createAdvance(u8pfx3, 1, "u8scope32");
    PabloAST * u8scope33 = it3.createAdvance(u8scope32, 1, "u8scope33");
    it3.createAssign(u8scope3X, it3.createOr(u8scope32, u8scope33));
    PabloAST * E0_invalid = it3.createAnd(it3.createAdvance(ccc.compileCC(re
        ↪ ::makeCC(0xE0), it3), 1), ccc.compileCC(re::makeCC(0x80, 0x9F),
        ↪ it3));
    PabloAST * ED_invalid = it3.createAnd(it3.createAdvance(ccc.compileCC(re
        ↪ ::makeCC(0xED), it3), 1), ccc.compileCC(re::makeCC(0xA0, 0xBF),
        ↪ it3));
```

```
it3.createAssign(EX_invalid, it3.createOr(E0_invalid, ED_invalid));

for (int i = 2; i <= 7; i++) {
    it3.createAssign(u32_2[i], it3.createOr(u32_2[i], it3.createAnd(
        ↪ u8scope33, u8_bits[i])));
}
it3.createAssign(u32_2[1], it3.createOr(u32_2[1], it3.createAnd(
    ↪ u8scope33, it3.createAdvance(u8_bits[7], 1))));
it3.createAssign(u32_2[0], it3.createOr(u32_2[0], it3.createAnd(
    ↪ u8scope33, it3.createAdvance(u8_bits[6], 1))));
for (int i = 2; i <= 5; i++) {
    it3.createAssign(u32_1[i + 2], it3.createOr(u32_1[i + 2], it3.
        ↪ createAnd(u8scope33, it3.createAdvance(u8_bits[i], 1))));
}
for (int i = 4; i <= 7; i++) {
    it3.createAssign(u32_1[i - 4], it3.createOr(u32_1[i - 4], it3.
        ↪ createAnd(u8scope33, it3.createAdvance(u8_bits[i], 2))));
}
it3.createAssign(del3, u8scope32);

//
// Four-byte sequences
Var * u8scope4nonfinal = it.createVar("u8scope4nonfinal", zeroes);
Var * u8scope4X = it.createVar("u8scope4X", zeroes);
Var * FX_invalid = it.createVar("FX_invalid", zeroes);
Var * del4 = it.createVar("del4", zeroes);

PabloBuilder it4 = PabloBuilder::Create(it);
it.createIf(u8pfx4, it4);
PabloAST * u8scope42 = it4.createAdvance(u8pfx4, 1, "u8scope42");
PabloAST * u8scope43 = it4.createAdvance(u8scope42, 1, "u8scope43");
PabloAST * u8scope44 = it4.createAdvance(u8scope43, 1, "u8scope44");


it4.createAssign(u8scope4nonfinal, it4.createOr(u8scope42, u8scope43));
it4.createAssign(u8scope4X, it4.createOr(u8scope4nonfinal, u8scope44));
PabloAST * F0_invalid = it4.createAnd(it4.createAdvance(ccc.compileCC(re
    ↪ ::makeCC(0xF0), it4), 1), ccc.compileCC(re::makeCC(0x80, 0x8F),
    ↪ it4));
PabloAST * F4_invalid = it4.createAnd(it4.createAdvance(ccc.compileCC(re
    ↪ ::makeCC(0xF4), it4), 1), ccc.compileCC(re::makeCC(0x90, 0xBF),
    ↪ it4));
it4.createAssign(FX_invalid, it4.createOr(F0_invalid, F4_invalid));


for (int i = 2; i <= 7; i++) {
    it4.createAssign(u32_2[i], it4.createOr(u32_2[i], it4.createAnd(
        ↪ u8scope44, u8_bits[i])));
}
it4.createAssign(u32_2[1], it4.createOr(u32_2[1], it4.createAnd(
    ↪ u8scope44, it4.createAdvance(u8_bits[7], 1))));
it4.createAssign(u32_2[0], it4.createOr(u32_2[0], it4.createAnd(
    ↪ u8scope44, it4.createAdvance(u8_bits[6], 1))));
for (int i = 2; i <= 5; i++) {
    it4.createAssign(u32_1[i + 2], it4.createOr(u32_1[i + 2], it4.
        ↪ createAnd(u8scope44, it4.createAdvance(u8_bits[i], 1))));
}
for (int i = 4; i <= 7; i++) {
```

```
        it4.createAssign(u32_1[i - 4], it4.createOr(u32_1[i - 4], it4.
            ↪ createAnd(u8scope44, it4.createAdvance(u8_bits[i], 2))));
    }
    it4.createAssign(u32_0[7], it4.createOr(u32_0[7], it4.createAnd(
        ↪ u8scope44, it4.createAdvance(u8_bits[3], 2))));
    it4.createAssign(u32_0[6], it4.createOr(u32_0[6], it4.createAnd(
        ↪ u8scope44, it4.createAdvance(u8_bits[2], 2))));
    for (int i = 5; i <= 7; i++) {
        it4.createAssign(u32_0[i - 2], it4.createOr(u32_0[i - 2], it4.
            ↪ createAnd(u8scope44, it4.createAdvance(u8_bits[i], 3))));
    }

    it4.createAssign(del4, it4.createOr(u8scope42, u8scope43));

    // Invalid cases
    PabloAST * anyscope = it.createOr(u8scope22, it.createOr(u8scope3X,
        ↪ u8scope4X), "anyscope");
    PabloAST * legalpfx = it.createOr(it.createOr(u8pfx2, u8pfx3), u8pfx4);
    //  Any scope that does not have a suffix byte, and any suffix byte that
        ↪  is not in
    //  a scope is a mismatch, i.e., invalid UTF-8.
    PabloAST * mismatch = it.createXor(anyscope, u8suffix);
    //
    PabloAST * EF_invalid = it.createOr(EX_invalid, FX_invalid);
    PabloAST * pfx_invalid = it.createXor(u8pfx, legalpfx);
    it.createAssign(u8invalid, it.createOr(pfx_invalid, it.createOr(mismatch
        ↪ , EF_invalid)));
    //PabloAST * u8valid = it.createNot(u8invalid, "u8valid");
    it.createAssign(error_mask, u8invalid);
    it.createAssign(delmask, it.createOr(it.createOr(del3, del4), ccc.
        ↪ compileCC(re::makeCC(0xC0, 0xFF), it)));

    Var * output = this->getOutputStreamVar("u32bit");
    Var * delmask_out = this->getOutputStreamVar("delMask");
    Var * error_mask_out = this->getOutputStreamVar("errMask");


    for (unsigned i = 0; i < 8; i++) {
        main.createAssign(main.createExtract(output, i), u32_0[i]);
    }
    for (unsigned i = 0; i < 8; i++) {
        main.createAssign(main.createExtract(output, i + 8), u32_1[i]);
    }
    for (unsigned i = 0; i < 8; i++) {
        main.createAssign(main.createExtract(output, i + 16), u32_2[i]);
    }
    main.createAssign(main.createExtract(delmask_out, main.getInteger(0)),
        ↪ delmask);
    main.createAssign(main.createExtract(error_mask_out,  main.getInteger(0)
        ↪ ), error_mask);

}
```

Listing A.2: Multiplexed Character Classes Pipeline

```
std::pair<StreamSetBuffer *, StreamSetBuffer *> grepPipeline_rep(Driver *
   ↪ grepDriver, std::vector<re::RE *> & REs, const GrepModeType grepMode,
```

```
↪ unsigned encodingBits, StreamSetBuffer * ByteStream, Value * fileIdx)
↪ {

auto & idb = grepDriver->getBuilder();
const unsigned segmentSize = codegen::SegmentSize < 2 ? 2 : codegen::
    ↪ SegmentSize;
const unsigned bufferSegments = (codegen::BufferSegments  < 2 ? 2 :
    ↪ codegen::BufferSegments) * codegen::ThreadNum;
size_t MatchLimit = ((grepMode == QuietMode) | (grepMode ==
    ↪ FilesWithMatch) | (grepMode == FilesWithoutMatch)) ? 1 :
    ↪ MaxCountFlag;

StreamSetBuffer * BasisBits = grepDriver->addBuffer(make_unique<
    ↪ CircularBuffer>(idb, idb->getStreamSetTy(encodingBits, 1),
    ↪ segmentSize * bufferSegments));
kernel::Kernel * s2pk = grepDriver->addKernelInstance(make_unique<kernel
    ↪ ::S2PKernel>(idb));
grepDriver->makeKernelCall(s2pk, {ByteStream}, {BasisBits});

kernel::Kernel * requiredStreamsK = grepDriver->addKernelInstance(
    ↪ make_unique<kernel::RequiredStreams_UTF8>(idb));
StreamSetBuffer * RequiredStreams = grepDriver->addBuffer(make_unique<
    ↪ CircularBuffer>(idb, idb->getStreamSetTy(4, 1), segmentSize *
    ↪ bufferSegments));
grepDriver->makeKernelCall(requiredStreamsK, {BasisBits}, {
    ↪ RequiredStreams});

const auto n = REs.size();

std::vector<std::vector<UCD::UnicodeSet>> charclasses;

unsigned *exclusiveSetIDs_array[2];

for (unsigned i = 0; i < n; i++) {
    REs[i] = resolveNames(REs[i]);
    std::vector<UCD::UnicodeSet> UnicodeSets = re::collect_UnicodeSets(
        ↪ REs[i]);
    UnicodeSets.push_back(UCD::UnicodeSet(0x0A));
    UnicodeSets.push_back(UCD::UnicodeSet(0x0D));
    std::vector<std::vector<unsigned>> exclusiveSetIDs;
    std::vector<UCD::UnicodeSet> multiplexedCCs;

    doMultiplexCCs(UnicodeSets, exclusiveSetIDs, multiplexedCCs);

    for (unsigned i = 0; i < 2; ++i) {
      exclusiveSetIDs_array[i] = new unsigned[exclusiveSetIDs[i].size()
          ↪ ];
      for (unsigned j = 0; j < exclusiveSetIDs[i].size(); j++) {
        exclusiveSetIDs_array[i][j] = unsigned(exclusiveSetIDs[i][j]);
      }
    }

    REs[i] = multiplex(REs[i], UnicodeSets, exclusiveSetIDs);
    charclasses.push_back(multiplexedCCs);
}

kernel::Kernel * linebreakK = grepDriver->addKernelInstance(make_unique<
    ↪ kernel::LineBreakKernelBuilder>(idb, encodingBits));
```

```
StreamSetBuffer * LineBreakStream = grepDriver->addBuffer(make_unique<
    ↪ CircularBuffer>(idb, idb->getStreamSetTy(1, 1), segmentSize *
    ↪ bufferSegments));

grepDriver->makeKernelCall(linebreakK, {BasisBits}, {LineBreakStream});

StreamSetBuffer * DelMask = grepDriver->addBuffer(make_unique<
    ↪ CircularBuffer>(idb, idb->getStreamSetTy(), segmentSize *
    ↪ bufferSegments));
StreamSetBuffer * NegDelMask = grepDriver->addBuffer(make_unique<
    ↪ CircularBuffer>(idb, idb->getStreamSetTy(), segmentSize *
    ↪ bufferSegments));
StreamSetBuffer * ErrorMask = grepDriver->addBuffer(make_unique<
    ↪ CircularBuffer>(idb, idb->getStreamSetTy(), segmentSize *
    ↪ bufferSegments));

kernel::Kernel * delMaskK = grepDriver->addKernelInstance(make_unique<
    ↪ kernel::DelMaskKernelBuilder>(idb));

StreamSetBuffer * LineBreakStream_del_4 = grepDriver->addBuffer(
    ↪ make_unique<CircularBuffer>(idb, idb->getStreamSetTy(4, 1),
    ↪ segmentSize * bufferSegments));
StreamSetBuffer * LineBreakStream_del = grepDriver->addBuffer(
    ↪ make_unique<CircularBuffer>(idb, idb->getStreamSetTy(1, 1),
    ↪ segmentSize * bufferSegments));

StreamSetBuffer * lbSwizzle = grepDriver->addBuffer(make_unique<
    ↪ SwizzledCopybackBuffer>(idb, idb->getStreamSetTy(4), segmentSize *
    ↪  (bufferSegments+2), 1));

kernel::Kernel * delKlb = grepDriver->addKernelInstance(make_unique<
    ↪ kernel::SwizzledDeleteByPEXTkernel>(idb, 64, 1/*4*/));
kernel::Kernel * unSwizzleKlb = grepDriver->addKernelInstance(
    ↪ make_unique<kernel::SwizzleGenerator>(idb, 4, 1, 1));

grepDriver->makeKernelCall(delMaskK, {BasisBits}, {DelMask, NegDelMask,
    ↪ ErrorMask});
grepDriver->makeKernelCall(delKlb, {LineBreakStream, DelMask}, {
    ↪ lbSwizzle});
grepDriver->makeKernelCall(unSwizzleKlb, {lbSwizzle}, {
    ↪ LineBreakStream_del_4});

kernel::Kernel * selectStreamK_lb = grepDriver->addKernelInstance(
    ↪ make_unique<kernel::SelectStream>(idb, 4, 0));
selectStreamK_lb->setName("selectStreamK_lb");
grepDriver->makeKernelCall(selectStreamK_lb, {LineBreakStream_del_4}, {
    ↪ LineBreakStream_del});

std::vector<StreamSetBuffer *> MatchResultsBufs(n);

for(unsigned i = 0; i < n; ++i){
     StreamSetBuffer * CharClasses = nullptr;
     StreamSetBuffer * CharClasses_del_k = nullptr;
     StreamSetBuffer * CharClasses_del = nullptr;

          //k = bitstreamCount, num = number of input sets to the
              ↪ unswizzle kernel.
```

```
        const auto numOfCharacterClasses = charclasses[i].size();
        unsigned k = (numOfCharacterClasses + 3 ) / 4 * 4;
        unsigned num = k / 4;

        CharClasses = grepDriver->addBuffer(make_unique<CircularBuffer>(
            ↪ idb, idb->getStreamSetTy(numOfCharacterClasses),
            ↪ segmentSize * bufferSegments));
        CharClasses_del_k = grepDriver->addBuffer(make_unique<
            ↪ CircularBuffer>(idb, idb->getStreamSetTy(k), segmentSize *
            ↪  bufferSegments));
        CharClasses_del = grepDriver->addBuffer(make_unique<
            ↪ CircularBuffer>(idb, idb->getStreamSetTy(
            ↪ numOfCharacterClasses), segmentSize * bufferSegments));
        std::vector<StreamSetBuffer *> Swizzle_cc;
        for (unsigned j = 0; j < num; ++j) {
            Swizzle_cc.push_back(grepDriver->addBuffer(make_unique<
                ↪ SwizzledCopybackBuffer>(idb, idb->getStreamSetTy(4),
                ↪ segmentSize * (bufferSegments+2), 1)));
        }

        kernel::Kernel * delK_cc = grepDriver->addKernelInstance(
            ↪ make_unique<kernel::SwizzledDeleteByPEXTkernel>(idb, 64,
            ↪ numOfCharacterClasses));
        kernel::Kernel * unSwizzleK_cc = grepDriver->addKernelInstance(
            ↪ make_unique<kernel::SwizzleGenerator>(idb, k, 1, num));
        unSwizzleK_cc->setName("unSwizzleK_cc");
        kernel::Kernel * ccK = grepDriver->addKernelInstance(make_unique
            ↪ <kernel::CharClassesKernel>(idb, std::move(charclasses[i])
            ↪ ));
        grepDriver->makeKernelCall(ccK, {BasisBits}, {CharClasses});
        grepDriver->makeKernelCall(delK_cc, {CharClasses, DelMask},
            ↪ Swizzle_cc);
        grepDriver->makeKernelCall(unSwizzleK_cc, Swizzle_cc, {
            ↪ CharClasses_del_k});

        kernel::Kernel * selectStreamK_cc = grepDriver->
            ↪ addKernelInstance(make_unique<kernel::
            ↪ ExpandOrSelectStreams>(idb, k, numOfCharacterClasses));
        selectStreamK_cc->setName("selectStreamK_cc");
        grepDriver->makeKernelCall(selectStreamK_cc, {CharClasses_del_k
            ↪ }, {CharClasses_del});

    StreamSetBuffer * MatchResults = grepDriver->addBuffer(make_unique<
        ↪ CircularBuffer>(idb, idb->getStreamSetTy(1, 1), segmentSize *
        ↪ bufferSegments));
    kernel::Kernel * icgrepK = grepDriver->addKernelInstance(make_unique
        ↪ <kernel::ICGrepKernel>(idb, REs[i], false,
        ↪ numOfCharacterClasses, true, exclusiveSetIDs_array));
    grepDriver->makeKernelCall(icgrepK, {CharClasses_del,
        ↪ LineBreakStream_del, RequiredStreams}, {MatchResults});
    MatchResultsBufs[i] = MatchResults;
}
StreamSetBuffer * MergedResults = MatchResultsBufs[0];
if (REs.size() > 1) {
    MergedResults = grepDriver->addBuffer(make_unique<CircularBuffer>(
        ↪ idb, idb->getStreamSetTy(1, 1), segmentSize * bufferSegments))
        ↪ ;
```

```
        kernel::Kernel * streamsMergeK = grepDriver->addKernelInstance(
            ↪ make_unique<kernel::StreamsMerge>(idb, 1, REs.size()));
        grepDriver->makeKernelCall(streamsMergeK, MatchResultsBufs, {
            ↪ MergedResults});
    }
    StreamSetBuffer * Matches = MergedResults;

    if (matchesNeedToBeMovedToEOL()) {
        StreamSetBuffer * OriginalMatches = Matches;
        kernel::Kernel * matchedLinesK = grepDriver->addKernelInstance(
            ↪ make_unique<kernel::MatchedLinesKernel>(idb));
        Matches = grepDriver->addBuffer(make_unique<CircularBuffer>(idb, idb
            ↪ ->getStreamSetTy(1, 1), segmentSize * bufferSegments));
        grepDriver->makeKernelCall(matchedLinesK, {OriginalMatches,
            ↪ LineBreakStream_del}, {Matches});
    }

    if (InvertMatchFlag) {
        kernel::Kernel * invertK = grepDriver->addKernelInstance(make_unique
            ↪ <kernel::InvertMatchesKernel>(idb));
        StreamSetBuffer * OriginalMatches = Matches;
        Matches = grepDriver->addBuffer(make_unique<CircularBuffer>(idb, idb
            ↪ ->getStreamSetTy(1, 1), segmentSize * bufferSegments));
        grepDriver->makeKernelCall(invertK, {OriginalMatches,
            ↪ LineBreakStream_del}, {Matches});
    }
    if (MatchLimit > 0) {
        kernel::Kernel * untilK = grepDriver->addKernelInstance(make_unique<
            ↪ kernel::UntilNkernel>(idb));
        untilK->setInitialArguments({idb->getSize(MatchLimit)});
        StreamSetBuffer * AllMatches = Matches;
        Matches = grepDriver->addBuffer(make_unique<CircularBuffer>(idb, idb
            ↪ ->getStreamSetTy(1, 1), segmentSize * bufferSegments));
        grepDriver->makeKernelCall(untilK, {AllMatches}, {Matches});
    }

    StreamSetBuffer * Matches_4 = grepDriver->addBuffer(make_unique<
        ↪ CircularBuffer>(idb, idb->getStreamSetTy(4, 1), 10 * segmentSize *
        ↪  bufferSegments));
    kernel::Kernel * expandStreamK_match = grepDriver->addKernelInstance(
        ↪ make_unique<kernel::ExpandOrSelectStreams>(idb, 1, 4));
    grepDriver->makeKernelCall(expandStreamK_match, {Matches}, {Matches_4});

    StreamSetBuffer * swizzle_match = grepDriver->addBuffer(make_unique<
        ↪ DynamicBuffer>(idb, idb->getStreamSetTy(4, 1), 10 * segmentSize *
        ↪ bufferSegments));

    kernel::Kernel * SwizzleK_match = grepDriver->addKernelInstance(
        ↪ make_unique<kernel::SwizzleGenerator>(idb, 4, 1, 1));
    SwizzleK_match->setName("SwizzleK_match");
    grepDriver->makeKernelCall(SwizzleK_match, {Matches_4}, {swizzle_match})
        ↪ ;

    StreamSetBuffer * swizzle_PDEP_match = grepDriver->addBuffer(make_unique
        ↪ <DynamicBuffer>(idb, idb->getStreamSetTy(4, 1), 10 * segmentSize *
        ↪  bufferSegments));
    kernel::Kernel * PDEPK = grepDriver->addKernelInstance(make_unique<
        ↪ kernel::PDEPkernel>(idb, 4, 4));
```

```
    grepDriver->makeKernelCall(PDEPK, {NegDelMask, swizzle_match}, {
        ↪ swizzle_PDEP_match});

    StreamSetBuffer * PDEP_match = grepDriver->addBuffer(make_unique<
        ↪ CircularBuffer>(idb, idb->getStreamSetTy(4, 1), 10 * segmentSize *
        ↪  bufferSegments));
    kernel::Kernel * unSwizzleK_match = grepDriver->addKernelInstance(
        ↪ make_unique<kernel::SwizzleGenerator>(idb, 4, 1, 1));
    unSwizzleK_match->setName("unSwizzleK_match");
    grepDriver->makeKernelCall(unSwizzleK_match, {swizzle_PDEP_match}, {
        ↪ PDEP_match});

    StreamSetBuffer * PDEP_match_result = grepDriver->addBuffer(make_unique<
        ↪ CircularBuffer>(idb, idb->getStreamSetTy(1, 1), 10 * segmentSize *
        ↪  bufferSegments));
    kernel::Kernel * selectStreamK_match = grepDriver->addKernelInstance(
        ↪ make_unique<kernel::SelectStream>(idb, 4, 0));
    selectStreamK_match->setName("selectStreamK_match");
    grepDriver->makeKernelCall(selectStreamK_match, {PDEP_match}, {
        ↪ PDEP_match_result});

    return std::pair<StreamSetBuffer *, StreamSetBuffer *>(LineBreakStream,
        ↪ PDEP_match_result);

}
```