Bitwise Data Parallelism in Regular Expression Matching

Anonymous Authors

Institutions

emails

Abstract

New parallel algorithms for the classical grep (global regular expression print) problem are introduced together with implementations using commodity SIMD and GPU technologies. Building on the bitwise data parallelism underlying previous work in Unicode transcoding and XML parsing, the new algorithms add the important element of nondeterminism for tackling the full generality of regular expression processing. On widely-deployed commodity hardware using 128-bit SSE2 SIMD technology, our algorithm implementations can substantially outperform traditional grep implementations based on NFAs, DFAs or backtracking. The algorithms are also designed to scale with the availability of additional parallel resources such as the wider SIMD facilities (256-bit) of Intel AVX2 or future 512-bit extensions. Our GPU implementations show further acceleration limited only by data transfer speed.

Categories and Subject Descriptors Theory of computation [*Formal languages and automata theory*]: Regular languages; Computer systems organization [*Parallel architectures*]: Single instruction, multiple data

Keywords regular expression matching, grep, parallel bit stream technology

1. Introduction

The use of regular expressions to search texts for occurrences of string patterns has a long history and remains a pervasive technique throughout computing applications today. The origins of regular expression matching date back to automata theory developed by Kleene in the 1950s [**?**]. Thompson [**?**] is credited with the first construction to convert regular expressions to nondeterministic finite automata (NFA). Following Thompson's approach, a regular expression of length m is first converted to an NFA with $O(m)$ nodes. It is then possible to search a text of length n using the NFA in worst case $O(mn)$ time. Often, a more efficient choice is to convert the NFA into a DFA. A DFA has only a single active state at any time in the matching process and hence it is possible to search a text at of length n in $O(n)$ time. However, it is well known that the conversion of an NFA to an equivalent DFA may result in state explosion. That is, the number of resultant DFA states may increase exponentially. In [**?**] a new approach to text searching was proposed based on bit-parallelism [**?**]. This technique takes advantage of the intrinsic parallelism of bitwise operations within a computer word. Given a w-bit word, the Shift-Or algorithm [**?**] algorithm uses the bit-parallel approach to simulate an NFA in $O(nm/w)$ worst-case time.

A disadvantage of the bit-parallel Shift-Or pattern matching approach in comparison to simple string matching algorithms is an inability to skip input characters. For example, the Boyer-Moore family of algorithms [**?**] skip input characters to achieve sublinear times in the average case. Backward Dawg Matching (BDM) string matching algorithms [**?**] based on suffix automata are able to skip characters. The Backward Nondeterministic Dawg Matching (BNDM) pattern matching algorithm [**?**] combines the bit-parallel advantages of Shift-Or and with the character skipping advantages of the BDM algorithm. The nrgrep pattern matching tool is built over the BNDM algorithm, and hence the name nrgrep [**?**].

There has been considerable interest in using parallelization techniques to improve the performance of regular expression matching on parallel hardware such as multi-core processors (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and specialized architectures such as the Cell Broadband Engine (Cell BE). Scarpazza and Braudaway [**?**] demonstrated that text processing algorithms that exhibit irregular memory access patterns can be efficiently executed on multi-core hardware. In related work, Pasetto et al. presented a flexible tool that performs small-ruleset regular expression matching at a rate of 2.88 Gbps per chip on Intel Xeon E5472 hardware [**?**]. Naghmouchi et al. demonstrated that the Aho-Corasick (AC) string matching algorithm [**?**] is well suited for parallel implementation on multi-core CPUs, GPUs and the Cell BE [**? ?**]. On each hardware, both thread-level parallelism (additional cores) and data-level parallelism (wide SIMD units) are leveraged for performance. Salapura et. al., advocated the use of vector-style processing for regular expressions in business analytics applications and leveraged the SIMD hardware available on multi-core processors to acheive a speedup of better than 1.8 over a range of data sizes of interest [**?**]. In [**?**], Scarpazza and Russell presented a SIMD tokenizer that delivered 1.001.78 Gbps on a single Cell BE chip and extended this approach for emulation on the Intel Larrabee instruction set [**?**]. On the Cell BE, Scarpazza [**?**] described a pattern matching implementation that delivered a throughput of 40 Gbps for a small dictionary of approximately 100 patterns, and a throughput of 3.3-3.4 Gbps for a larger dictionary of thousands of patterns. Iorio and van Lunteren [**?**] presented a string matching implementation for automata that achieves 4 Gbps on the Cell BE. In more recent work, Tumeo et al. [**?**] presented a chunk-based implementation of the AC algorithm for accelerating string matching on GPUs. Lin et al., proposed the Parallel Failureless Aho-Corasick (PFAC) algorithm to accelerate pattern matching on GPU hardware and achieved 143 Gbps throughput, 14.74 times faster than the AC algorithm performed on a four core multi-core processor using OpenMP [**?**].

Whereas the existing approaches to parallelization have been based on adapting traditional sequential algorithms to emergent parallel architectures, we introduce both a new algorithmic approach and its implementation on SIMD and GPU architectures. This approach relies on a bitwise data parallel view of text streams as well as a surprising use of addition to match runs of characters in a single step. The closest previous work is that underlying bit-parallel XML parsing using 128-bit SSE2 SIMD technology together with a parallel scanning primitive also based on addition [**?**]. However, in contrast to the deterministic, longest-match scanning associated with the ScanThru primitive of that work, we introduce here a new primitive MatchStar that can be used in full generality for nondeterministic regular expression matching. We also introduce a long-stream addition technique involving a further application of MatchStar that enables us to scale the technique to n-bit addition in $\lceil \log_{64} n \rceil$ steps. We ultimately apply this technique, for example, to perform synchronized 4096-bit addition on GPU warps of 64 threads.

There is also a strong keyword match between the bit-parallel data streams used in our approach and the bit-parallelism used for NFA state transitions in the classical algorithms of Wu and Manber [**?**], Baez-Yates and Gonnet [**?**] and Navarro and Raffinot [**?**]. However those algorithms use bit-parallelism in a fundamentally different way: representing all possible current NFA states as a bit vector and performing parallel transitions to a new set of states using table lookups and bitwise logic. Whereas our approach can match multiple characters per step, bit-parallel NFA algorithms proceed through the input one byte at a time. Nevertheless, the agrep [**?**] and nrgrep [**?**] programs implemented using these techniques remain among the strongest competitors in regular expression matching performance, so we include them in our comparative evaluation.

The remainder of this paper is organized as follows. Section 2 briefly describes regular expression notation and the grep problem. Section 3 presents our basic algorithm and MatchStar using a model of arbitrary-length bit-parallel data streams. Section 4 discusses the block-by-block implementation of our techniques including the long stream addition techniques for 256-bit addition with AVX2 and 4096-bit additions with GPGPU SIMT. Section 5 describes our overall SSE2 implementation and carries out a performance study in comparison with existing grep implementations. Given the dramatic variation in grep performance across different implementation techniques, expressions and data sets, Section 6 considers a comparison between the bit-stream and NFA approaches from a theoretical perspective. Section 7 then examines and demonstrates the scalability of our bitwise data-parallel approach in moving from 128-bit to 256-bit SIMD on Intel Haswell architecture. To further investigate scalability, Section 8 addresses the implementation of our matcher using groups of 64 threads working together SIMT-style on a GPGPU system. Section 10 concludes the paper with a discussion of results and areas for future work.

2. Regular Expression Notation and Grep

We follow common Posix notation for regular expressions. A regular expression specifies a set of strings through a pattern notation. Individual characters normally stand for themselves, unless they are one of the special characters `*+?[{\(|^$.` that serve as metacharacters of the notation system. Thus the regular expression `cat` is a pattern for the set consisting of the single 3-character string "`cat`". The special characters must be escaped with a backslash to prevent interpretation as metacharacter, thus `\$` represents the dollar-sign and `\\\\` represent the string consisting of two backslash characters. Character class bracket expressions are pattern elements that allow any character in a given class to be used in a particular context. For example, `[@#%]` is a regular expression that stands for any of the three given symbols. Contiguous ranges of characters may be specified using hyphens; for example `[0-9]` for digits and `[A-Za-z0-9_]` for any alphanumeric character or underscore. If the caret character immediately follows the opening bracket, the class is negated, thus `[^0-9]` stands for any character except a digit. The period metacharacter `.` stands for the class of all characters.

Consecutive pattern elements stand for strings formed by concatenation, thus `[cd][ao][tg]` stands for the set of 8 three-letter strings "`cat`" through "`dog`". The alternation operator `|` allows a pattern to be defined to have to alternative forms, thus `cat|dog` matches either "`cat`" or "`dog`". Concatenation takes precedence over alternation, but parenthesis may be used to change this, thus `(ab|cd)[0-9]` stands for any digit following one of the two prefixes "`ab`" or "`cd`".

Repetition operators may be appended to a pattern to specify a variable number of occurrences of that pattern. The Kleene `*` specifies zero-or-more occurrences matching the previous pattern, while `+` specifies one-or more occurrences. Thus `[a-z][a-z]*` and `[a-z]+` both specify the same set: strings of at least one lower-case letter. The postfix operator `?` specifies an optional component, i.e., zero-or-one occurrence of strings matching the element. Specific bounds may be given within braces: `(ab){3}` specifies the string "`ababab`", `[0-9A-Fa-f]{2,4}` specifies strings of two, three or four hexadecimal digits, and `[A-Z]{4,}` specifies strings of at least 4 consecutive capital letters.

The grep program searches a file for lines containing matches to a regular expression using any of the above notations. In addition, the pattern elements `^` and `$` may be used to match respectively the beginning or the end of a line. In line-based tools such as grep, `.` matches any character except newlines; matches cannot extend over lines. Normally, grep prints all matching lines to its output. However, grep programs typically allow a command line flag such as `-c` to specify that only a count of matching lines be produced; we use this option in our experimental evaluation to focus our comparisons on the performance of the underlying matching algorithms.

3. Matching with Bit-Parallel Data Streams

Whereas the traditional approaches to regular expression matching using NFAs, DFAs or backtracking all rely on a byte-at-a-time processing model, the approach we introduce in this paper is based on quite a different concept: a data-parallel approach to simultaneous processing of data stream elements. Indeed, our most abstract model is that of unbounded data parallelism: processing all elements of the input data stream simultaneously. In essence, data streams are viewed as (very large) integers. The fundamental operations are bitwise logic, stream shifting and long-stream addition.

Depending on the available parallel processing resources, an actual implementation may divide an input stream into blocks and process the blocks sequentially. Within each block all elements of the input stream are processed together, relying the availability of bitwise logic and addition scaled to the block size. On commodity Intel and AMD processors with 128-bit SIMD capabilities (SSE2), we typically process input streams 128 bytes at a time. In this case, we rely on the Parabix tool chain [**?**] to handle the details of compilation to block-by-block processing. For our GPGPU implementation, we have developed a long-stream addition technique that allows us to perform 4096-bit additions using 64 threads working in lock-step SIMT fashion. Using scripts to modify the output of the Parabix tools, we effectively divide the input into blocks of 4K bytes processed in a fully data-parallel manner.

A key concept in this streaming approach is the derivation of bit streams that are parallel to the input data stream, i.e., in one-

input data	a4534q--b29z---az---a4q--bca22z--
B_7	....
B_6	1....1..1..1...11...1.1..111..1..
B_5	11111111111111111111111111111111
B_4	.11111...111....1....11.....111..
B_3	......11..11111.1111...11.....111
B_2	.11.1.11....111..111.1.11......11
B_1	...1....11.1....1........11.111..
B_0	1.11.111..1.1111.1111.111.11...11
[a]	1.............1....1......1.....
[z]	...........1....1............1..
[0-9]	.1111....11.........1......11...

Figure 1. Basis and Character Class Streams

to-one correspondence with the data element positions of the input streams. Typically, the input stream is a byte stream comprising the 8-bit character code units of a particular encoding such as extended ASCII, ISO-8859-1 or UTF-8. However, the method may also easily be used with wider code units such as the 16-bit code units of UTF-16. In the case of a byte stream, the first step is to transpose the byte stream into eight parallel bit streams, such that bit stream i comprises the i^{th} bit of each byte. These streams form a set of basis bit streams from which many other parallel bit streams can be calculated, such as character class bit streams such that each bit j of the stream specifies whether character j of the input stream is in the class or not. Figure 1 shows an example of an input byte stream in ASCII, the eight basis bit streams of the transposed representation, and the character class bit streams `[a]`, `[z]`, and `[0-9]` that may be computed from the basis bit streams using bitwise logic. Zero bits are marked with periods (`.`) so that the one bits stand out. Transposition and character class construction are straightforward using the Parabix tool chain [**?**].

input data	a4534q--b29z---az---a4q--bca22z--
M_1	.1.............1....1......1....
M_2	.11111.........1....11.....111..
M_3	................1.............1.

Figure 2. Marker Streams in Matching `a[0-9]*z`

Marker Streams. Now consider how bit-parallel data streams can be used in regular expression matching. Consider the problem of searching the input stream of Figure 1 to finding occurrence of strings matching the regular expression `a[0-9]*z`. The matching process involves the concept of *marker streams*, that is streams that mark the positions of current matches during the overall process. In this case there are three marker streams computed during the match process, namely, M_1 representing match positions after an initial `a` character has been found, M_2 representing positions reachable from positions marked by M_1 by further matching zero or more digits (`[0-9]*`) and finally M_3 the stream marking positions after a final `z` has been found. Without describing the details of how these streams are computed for the time being, Figure 2 shows what each of these streams should be for our example matching problem. Note our convention that a marker stream contains a 1 bit at the next character position to be matched, that is, immediately past the last position that was matched.

MatchStar. MatchStar takes a marker bitstream and a character class bitstream as input. It returns all positions that can be reached by advancing the marker bitstream zero or more times through the character class bitstream.

input data	a4534q--b29z---az---a4q--bca22z--
M_1	.1.............1....1......1....
$D =$ [0-9]	.1111....11.........1......11...
$T_0 = M_1 \wedge D$	.1.................1......1....
$T_1 = T_0 + D$	.....1...11..........1.......1..
$T_2 = T_1 \oplus D$	.11111.............11.....111..
$M_2 = T_2 \mid M_1$	.11111.........1....11.....111..

Figure 3. $M_2 = \text{MatchStar}(M_1, D)$

Figure 3 illustrates the MatchStar method. In this figure, it is important to note that our bitstreams are shown in natural left-to-right order reflecting the conventional presentation of our character data input. However, this reverses the normal order of presentation when considering bitstreams as numeric values. The key point here is that when we perform bitstream addition, we will show bit movement from left-to-right. For example, `111. + 1... = ...1`.

The first row of the figure is the input data, the second and third rows are the input bitstreams: the initial marker position bitstream and the character class bitstream for digits derived from input data.

In the first operation (T_0), marker positions that cannot be advanced are temporarily removed from consideration by masking off marker positions that aren't character class positions using bitwise logic. Next, the temporary marker bitstream is added to the character class bitstream. The addition produces 1s in three types of positions. There will be a 1 immediately following a block of character class positions that spanned one or more marker positions, at any character class positions that weren't affected by the addition (and are not part of the desired output), and at any marker position that wasn't the first in its block of character class positions. Any character class positions that have a 0 in T_1 were affected by the addition and are part of the desired output. These positions are obtained and the undesired 1 bits are removed by XORing with the character class stream. T_2 is now only missing marker positions that were removed in the first step as well as marker positions that were 1s in T_1. The output marker stream is obtained by ORing T_2 with the initial marker stream.

In general, given a marker stream M and a character class stream C, the operation of MatchStar is defined by the following equation.

$$\text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C)|M$$

Given a set of initial marker positions, the result stream marks all possible positions that can be reached by 0 or more occurrences of characters in class C from each position in M.

Compilation. Using the marker stream and MatchStar concept, we now outline our compilation algorithm. This is implemented in a Java program. First the regular expression is parsed and represented as an abstract syntax tree. Second, the various character classes used in the regular expression are extracted. The character class compiler of the Parabix framework is invoked to generate the bit stream equations required for each character class. Then the syntax tree is walked to generate code for each type of regular expression structure as follows.

- An initial marker stream M_0 is set to be all ones, indicating that every position in the input file is a potential match if we have not yet examined any pattern elements.
- If we have a regular expression formed as an alternation of subexpressions, we compile each of these in turn, providing the

current input marker stream as input to each of them. The final marker streams of the compiled forms of each subexpression are then just combined using a bitwise-or to produce the overall final marker stream of the alternation. That is, a match occurs at any position that can be reached by matching any one of the alternatives.

- If we have a marker stream formed as a concatenation of subexpressions, then we compile each of these in turn, providing the output marker stream of each compilation as the input marker stream for the compilation of the next pattern element.
- If a regular expression is a character class expression or a single character, then we form the bitwise-and of the current marker stream and the character class stream to filter out current marker positions that do not have a match to the class. The result is then shifted forward one position to identify the successful matches.
- If a regular expression is an optional expression of the form `R?` for some subexpression `R`, then the output marker stream is simply formed as the bitwise-or of the input marker stream (zero occurrences of `R` matched) and the output stream produced by compiling `R` in the context of the current input marker stream (one occurrence matched).
- If a regular expression is a repetition of a character class of the form `C*`, then the compiled form uses the MatchStar operation to produce the output marker stream from the input stream and the compiled stream for character class `C`.
- If a regular expression is a repetition of a non character class of the form `R*`, then a Pablo while loop is created conditioned on a control marker stream still having bits marking match positions. The body of the while consists of the compiled form of the expression `R`, taking as input the marker at the beginning of the iteration and producing output that becomes the input for the next iteration, if any. The final output is the bitwise-or of matches determined in each loop iteration.
- If a regular expression is a bounded repetition of the form `R{m,n}`, then it is compiled according to the equivalent form consisting of m concatenations of `R` followed by $n - m$ concatenations of `R?`.
- If a regular expression is a bounded repetition of the form `R{m,}`, then it is compiled according to the equivalent form consisting of m concatenations of `R` followed by `R*`.

The output of the regular expression compiler is then fed as input to the Pablo compiler of the Parabix tool chain. The result is then compiled with a C++ compiler linked with the Parabix runtime libraries.

4. Block-at-a-Time Processing

The unbounded stream model of the previous section must of course be translated an implementation that proceeds block-at-a-time for realistic application. In this, we primarily rely on the Pablo compiler of the Parabix toolchain [**?**]. Given input statements expressed as arbitrary-length bitstream equations, Pablo produces block-at-a-time C++ code that initializes and maintains all the necessary carry bits for each of the additions and shifts involved in the bitstream calculations.

In the present work, our principal contribution to the block-at-a-time model is the technique of long-stream addition described below. Otherwise, we were able to use Pablo directly in compiling our SSE2 and AVX2 implementations. Our GPU implementation required some scripting to modify the output of the Pablo compiler for our purpose.

Long-Stream Addition. The maximum word size for addition on commodity processors is typically 64 bits. In order to implement long-stream addition for block sizes of 256 or larger, a method for propagating carries through the individual stages of 64-bit addition is required. However, the normal technique of sequential addition using add-with-carry instructions, for example, is far from ideal.

We have developed a technique using SIMD or SIMT methods for constant-time long-stream addition up to 4096 bits. We assume the availability of the following SIMD/SIMT operations operating on vectors of f 64-bit fields.

- `simd<64>::add(X, Y)`: vertical SIMD addition of corresponding 64-bit fields in two vectors to produce a result vector of f 64-bit fields.
- `simd<64>::eq(X, -1)` : comparison of the 64-bit fields of `x` each with the constant value -1 (all bits 1), producing an f-bit mask value,
- `hsimd<64>::mask(X)` : gathering the high bit of each 64-bit field into a single compressed f-bit mask value, and
- normal bitwise logic operations on f-bit masks, and
- `simd<64>::spread(x)` : distributing the bits of an f bit mask, one bit each to the f 64-bit fields of a vector.

Given these operations, our method for long stream addition of two $f \times 64$ bit values `X` and `Y` is the following.

1. Form the vector of 64-bit sums of `x` and `y`.

   $$\texttt{R} = \texttt{simd<64>::add(X, Y)}$$

2. Extract the f-bit masks of `X`, `Y` and `R`.

   $$\texttt{x} = \texttt{hsimd<64>::mask(X)}$$

   $$\texttt{y} = \texttt{hsimd<64>::mask(Y)}$$

   $$\texttt{r} = \texttt{hsimd<64>::mask(R)}$$

3. Compute an f-bit mask of carries generated for each of the 64-bit additions of `X` and `Y`.

   $$\texttt{c} = (\texttt{x} \wedge \texttt{y}) \vee ((\texttt{x} \vee \texttt{y}) \wedge \neg \texttt{r})$$

4. Compute an f-bit mask of all fields of `R` that will overflow with an incoming carry bit. This is the *bubble mask*.

   $$\texttt{b} = \texttt{simd<64>::eq(R, -1)}$$

5. Determine an f-bit mask identifying the fields of `R` that need to be incremented to produce the final sum. Here we find a new application of MatchStar!

   $$\texttt{i} = \texttt{MatchStar(c*2, b)}$$

   This is the key step. The mask `c` of outgoing carries must be shifted one position (`c*2`) so that each outgoing carry bit becomes associated with the next digit. At the incoming position, the carry will increment the 64-bit digit. However, if this digit is all ones (as signalled by the corresponding bit of bubble mask `b`, then the addition will generate another carry. In fact, if there is a sequence of digits that are all ones, then the carry must bubble through each of them. This is just MatchStar!

6. Compute the final result `Z`.

   $$\texttt{Z} = \texttt{simd<64>::add(R, simd<64>::spread(i))}$$

Figure 4 illustrates the process. In the figure, we illustrate the process with 8-bit fields rather than 64-bit fields and show all field values in hexadecimal notation. Note that two of the individual 8-bit additions produce carries, while two others produce `FF` values

X	19	31	BA	4C	3D	45	21	F1
Y	22	12	45	B3	E2	16	17	36
R	3B	43	FF	FF	1F	5B	38	27
x	0	0	1	0	0	0	0	1
y	0	0	0	1	1	0	0	0
r	0	0	1	1	0	0	0	0
c	0	0	0	0	1	0	0	1
c*2	0	0	0	1	0	0	1	0
b	0	0	1	1	0	0	0	0
i	0	1	1	1	0	0	1	0
Z	3B	44	0	0	1F	5B	39	27

Figure 4. Long Stream Addition

that generate bubble bits. The net result is that four of the original 8-bit sums must be incremented to produce the long stream result.

A slight extension to the process produces a long-stream full adder that can be used in chained addition. In this case, the adder must take an additional carry-in bit `p` and produce a carry-out bit `q`. This may be accomplished by incorporating `p` in calculating the increment mask in the low bit position, and then extracting the carry-out `q` from the high bit position.

```
i = MatchStar(c*2+p, b)
```

```
q = i >> f
```

As described subsequently, we use a two-level long-stream addition technique in both our AVX2 and GPU implementations. In principle, one can extend the technique to additional levels. Using 64-bit adders throughout, $\lceil \log_{64} n \rceil$ steps are needed for n-bit addition. A three-level scheme could coordinate 64 groups each performing 4096-bit long additions in a two-level structure. However, whether there are reasonable architectures that can support fine-grained SIMT style coordinate at this level is an open question.

Using the methods outlined, it is quite conceivable that instruction set extensions to support long-stream addition could be added for future SIMD and GPU processors. Given the fundamental nature of addition as a primitive and its novel application to regular expression matching as shown herein, it seems reasonable to expect such instructions to become available.

5. SSE2 Implementation and Evaluation

Implementation Notes. Our regular expression compiler directly uses the Parabix tool chain to compile regular expression into SSE2-based implementations. Our compiler essentially scripts three other compilers to perform this work: the Parabix character class compiler to determine basic bit stream equations for each of the character classes encountered in a regular expression, the Pablo bitstream equation compiler which converts equations to block-at-a-time C++ code for 128-bit SIMD, and gcc 4.6.3 to generate the binaries. The Pablo output is combined with a `grep_template.cpp` file that arranges to read input files, break them into segments, and print out or count matches as they are encountered.

Comparative Implementations. We evaluate our bitwise data parallel implementation versus several alternatives. We report data for two of these: GNU grep version 2.10 and nrgrep version 1.12. GNU grep is a popular open-source grep implementation that uses DFAs, as well as heuristics for important special cases. The NFA class is represented by nrgrep, one of the strongest competitors in regular expression matching performance. We also considered agrep 3.41 as and alternative NFA-based implementation and pcregrep 8.12 as a backtracking implementation, but do not report data for them. The agrep implementation does not support some of the

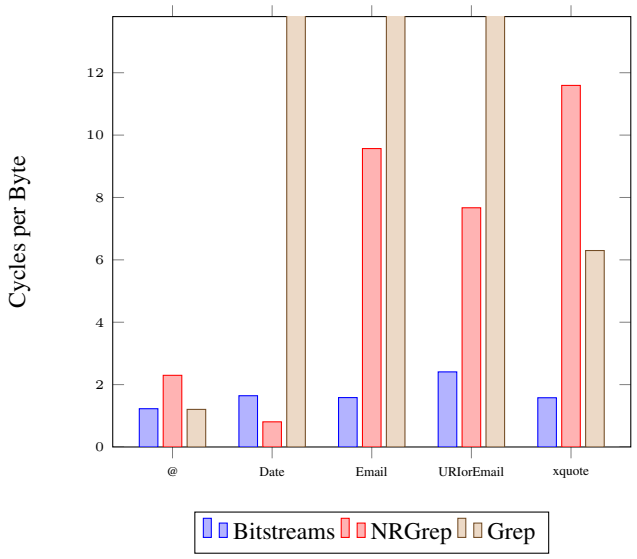

Figure 5. Cycles per Byte

common regular expression syntax feature and is limited to patterns of at most 32 characters. As a backtracking implementation, pcregrep supports more regular expression features, but is not competitive in performance in any example we tested.

We performed our SSE2 performance study using an Intel Core i7 quad-core (Sandy Bridge) processor (3.40GHz, 4 physical cores, 8 threads (2 per core), 32+32 kB (per core) L1 cache, 256 kB (per core) L2 cache, 8 MB L3 cache) running the 64-bit version of Ubuntu 12.04 (Linux).

Test expressions. Each grep implementation is tested with the five regular expressions in Table 1. Xquote matches any of the representations of a single or double quote character occuring in XML content. It is run on roads-2.gml, a 11,861,751 byte gml data file. The other four expressions are taken from Benchmark of Regex Libraries [http://lh3lh3.users.sourceforge.net/reb.shtml] and are all run on a concatenated version of the linux howto which is 39,422,105 bytes in length. @ simply matches the "@" character. It demonstrates the overhead involved in matching the simplest regular expression. Date, Email, and URIOrEmail provide examples of common uses for regular expression matching.

Results. Figure 5 shows CPU cycles per input byte for each expression on each implementation. For the three most complicated expressions, the bitstreams implementation had the lowest cycle count, while grep was an order of magnitude slower.

For the @ expression, GNU grep had very slightly better performance than the bitstreams implementation. The bitstreams implementation has a fixed overhead for transposition that hurts relative performance for very simple expressions.

For the Date expression, nrgrep is able to skip large portions of the input file since every time it encounters a character that can't appear in a date, it can skip past six characters. For the more compicated expressions, it loses this advantage.

The bitstreams implementation has fairly consistent performance. As the regular expression complexity increases, the cycle count increases slowly. The largest difference in cycles per byte for the bitstreams implementation is a ratio of 2 to 1. The same cannot be said of grep or nrgrep. The latter uses more than 10 times the cycles per byte for Xquote than for Date. The number of cycles per

Name	Expression
@	`@`
Date	`([0-9][0-9]?)/([0-9][0-9]?)/([0-9][0-9]([0-9][0-9])?)`
Email	`([^ @]+)@([^ @]+)`
URIOrEmail	`([a-zA-Z][a-zA-Z0-9]*)://([^ /]+)(/[^ ]*)?|([^ @]+)@([^ @]+)`
Xquote	`["']|&quot;|&apos;|&#0*3[49];|&#x0*2[27];`

Table 1. Regular Expressions

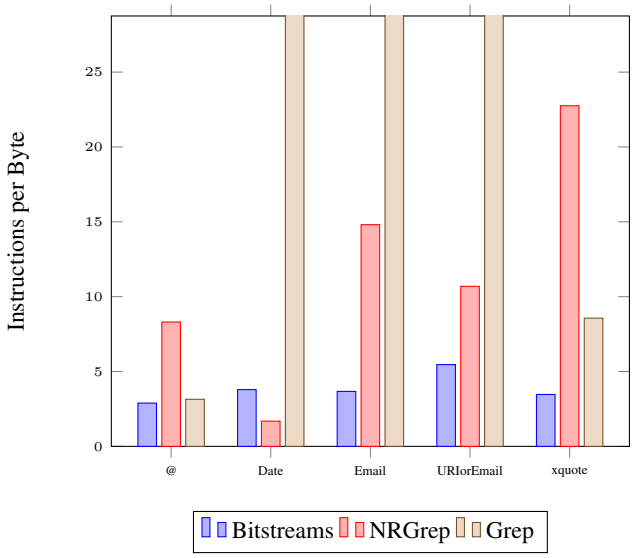

Figure 6. Instructions per Byte

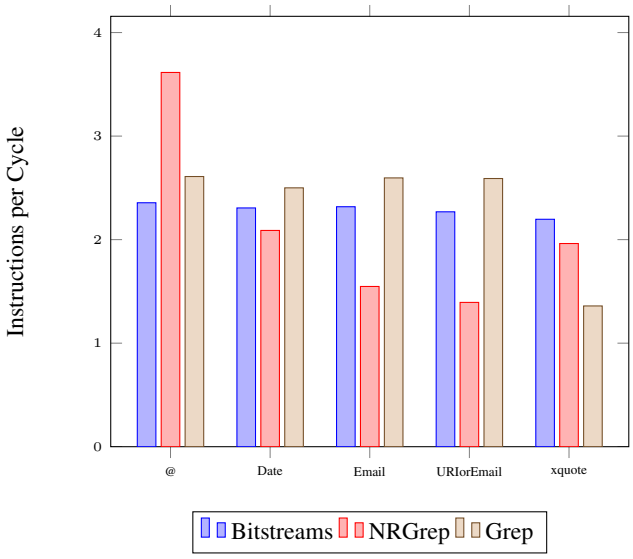

Figure 7. Instructions per Cycle

byte that gGrep uses for URIOrEmail is almost 900 times as many as it uses for @.

Figure 6 shows instructions per byte. The relative values mirror cycles per byte. The bitstreams implementation continues to show consistent performance. This is especially noticeable in Figure 7, which shows instructions per cycle. The bitstreams implementation has almost no variation in the instructions per cycle. Both grep and nrGrep have considerably more variation based on the input regular expression.

Figure 8 shows the branch misses per kilobyte. The bitstreams implementation remains consistent here. Each of nrgrep and grep have branch miss rates that vary significantly with different regular expressions.

Overall, our performance is considerably better than both nrgrep and grep for the more complicated expressions that were tested. Also, our performance scales smoothly with regular expression complexity so it can be expected to remain better for complicated expressions in general.

6. Running-time Comparison with DFA and NFA Implementations

Our experimental results indicate that regular expression matching using bitstreams can outperform current implementations of NFA- and DFA-based matching. It is worth exploring why this is so, and under what conditions one might expect bitstreams to perform better than NFA- or DFA-based matchers, and vice-versa.

The bitstream method starts with a preprocessing step: the compilation of the regular expression using the parabix toolchain. Compilation is an offline process whose time is not counted in our per-

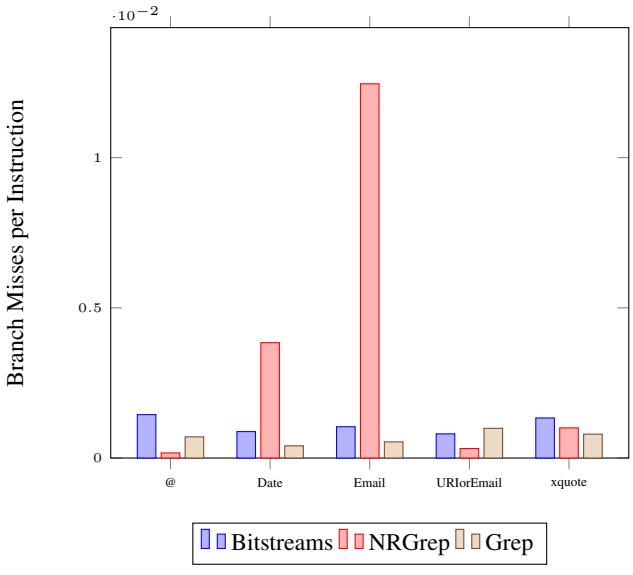

Figure 8. Branch Misses per Instruction

formance measures, as parabix is a experimental research compiler that is not optimized. This leads to a bias in our results, as our timings for nrgrep and grep include the time taken for preprocessing. We have attempted to minimize the bias by performing our tests with large inputs, so that the text-scanning costs dominate the preprocessing costs. We furthermore believe that, if a special-purpose optimized compiler for regular expressions were built, that its inclusion in bitstream grep would not substantially increase the running time, particularly for large input texts–the compilation involved is straightforward.

For simplicity, we will first assume that the input regular expressions are restricted to having Kleene closures only of single characters or alternations of single characters. This is a broad class of regular expressions, covering the majority of common uses of grep.

The bitstream method compiles a regular expression of size m into bitstream code that is $O(m)$ statements long (with one operation per statement; it is essentially three-address code). This is translated to machine code and placed inside a loop[1] that executes once per w characters, where w is the width of the processor's word. This gives $O(\frac{nm}{w})$ work. We can further break this down by noting that all of the work in the loop is done by superscalar instructions, with the exception of the additions, which require carry propagation. There will be at most C of these additions in the loop, where C is the number of concatenation and Kleene star operations in the regular expression.

Almost all intermediate bitstreams in the loop body can be kept in registers, requiring no storage in memory. Good register allocation–and limited live ranges for bitstream variables–keeps register spillage to a minimum. For those bitstreams that do require storage in memory, long buffers are allocated, allowing the successive iterations of the loop to access successive memory locations. That is, for the few streams requiring it, memory is accessed in a sequential fashion. As this is the best case for hardware prefetching, we expect few cache misses with bitstream method.

Compare this with NFA methods. In the base NFA method, a state set of approximately m states is kept as a bit set in $\frac{m}{w}$ machine words (or $\frac{m}{8}$ bytes). For each character c of the input, a precomputed transition table, indexed by the c and the current state set, is accessed. Since there are $2^{\Theta(m)}$ state sets, the transition table can have $\sigma 2^{\Theta(m)}$ entries, where σ is the size of the input alphabet. Each entry is a new state set, which requires $\frac{m}{8}$ bytes. Thus, the transition table is of size $\sigma m 2^{\Theta(m)}$, which is quite large: it can become expensive to precompute, and it consumes a lot of memory. For even fairly small m a table of this size will probably not fit in cache memory. Thus, we would expect many cache misses with this base method.

To improve the table size, several authors have separated the transition table into several tables, each indexed by a subset of the bits in the bit set representing the current state. Suppose one uses k bits of the state set to index each table. Ignoring ceilings, this requires $\frac{m}{k}$ tables, each with $\sigma 2^k$ entries of $\frac{m}{8}$ bytes apiece. Each table therefore takes up $2^{k-3}m\sigma$ bytes, and so the collection of them takes up $\frac{m 2^{k-3}\sigma}{k}$ bytes. At each character, the NFA algorithm does one lookup in each table, combining the results with $\frac{m}{k} - 1$ boolean OR operations.

The original NFA method of Thompson uses $k = 1$, which gives a m tables of $\frac{m\sigma}{4}$ bytes each, along with m lookups and $m-1$ boolean OR operations to combine the lookups, per character.

[1] Technically, it is inside two loops: an inner one that executes once per w characters in a large buffer, and an outer one that successively fetches buffers until the input is exhausted.

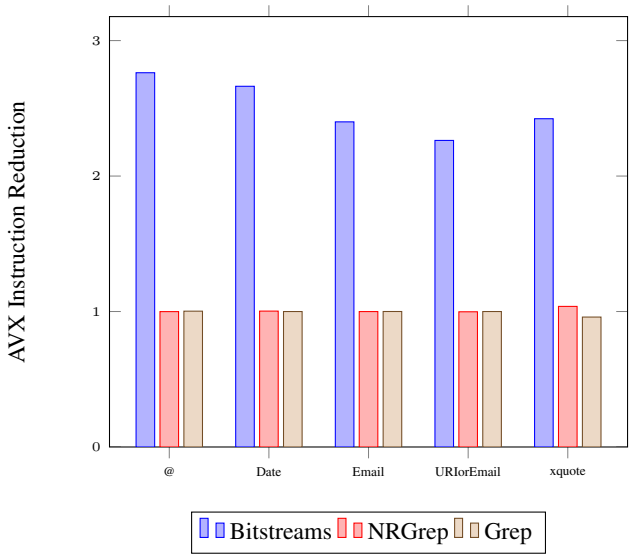

Figure 10. Instruction Reduction

Navarro and Raffinot use $k = \frac{m}{2}$, giving 2 tables of $2^{\frac{m}{2}-3}m\sigma$ bytes each, two lookups per character, and 1 boolean OR operation per character to combine the lookups.

7. SIMD Scalability

Although commodity processors have provided 128-bit SIMD operations for more than a decade, the extension to 256-bit integer SIMD operations has just recently taken place with the availability of AVX2 instructions in Intel Haswell architecture chips as of mid 2013. This provides an excellent opportunity to assess the scalability of the bitwise data-parallel approach to regular expression matching.

For the most part, adapting the Parabix tool chain to the new AVX2 instructions was straightforward. This mostly involved regenerating library functions using the new AVX2 intrinsics. There were minor issues in the core transposition algorithm because the doublebyte-to-byte pack instructions are confined to independent operations within two 128-bit lanes.

7.1 AVX Stream Addition

Bitstream addition at the 256-bit block size was implemented using the long-stream addition technique. Figure 9 shows our implementation. Spreading bits from the calculated increments mask was achieved somewhat awkwardly with a 64-bit multiply to spread into 16-bit fields followed by SIMD zero extend of the 16-bit fields to 64-bits each.

We also compiled new versions of the `grep` and `nrgrep` programs using the `-march=core-avx2` flag in case the compiler is able to vectorize some of the code.

Figure 10 shows the reduction in instruction count achieved for each of the applications. Working at a block size of 256 bytes at a time rather than 128 bytes at a time, the bitstreams implementation scaled dramatically well with reductions in instruction count over a factor of two in each case. Although a factor of two would seem an outside limit, we attribute the change to greater instruction efficiency. AVX2 instructions use a non destructive three-operand form instead of the destructive two-operand form of SSE2. In the two-operand form, binary instructions must always use one of the source registers as a destination register. As a result the SSE2

```
void add_ci_co(bitblock_t x, bitblock_t y, carry_t carry_in, carry_t & carry_out, bitblock_t & sum) {
  bitblock_t all_ones = simd256<1>::constant<1>();
  bitblock_t gen = simd_and(x, y);
  bitblock_t prop = simd_xor(x, y);
  bitblock_t partial_sum = simd256<64>::add(x, y);
  bitblock_t carry = simd_or(gen, simd_andc(prop, partial_sum));
  bitblock_t bubble = simd256<64>::eq(partial_sum, all_ones);
  uint64_t carry_mask = hsimd256<64>::signmask(carry) * 2 + convert(carry_in);
  uint64_t bubble_mask = hsimd256<64>::signmask(bubble);
  uint64_t carry_scan_thru_bubbles = (carry_mask + bubble_mask) &~ bubble_mask;
  uint64_t increments = carry_scan_thru_bubbles | (carry_scan_thru_bubbles - carry_mask);
  carry_out = convert(increments >> 4);
  uint64_t spread = 0x0000200040008001 * increments & 0x0001000100010001;
  sum = simd256<64>::add(partial_sum, _mm256_cvtepu16_epi64(avx_select_lo128(convert(spread))));
}
```

Figure 9. AVX2 256-bit Addition

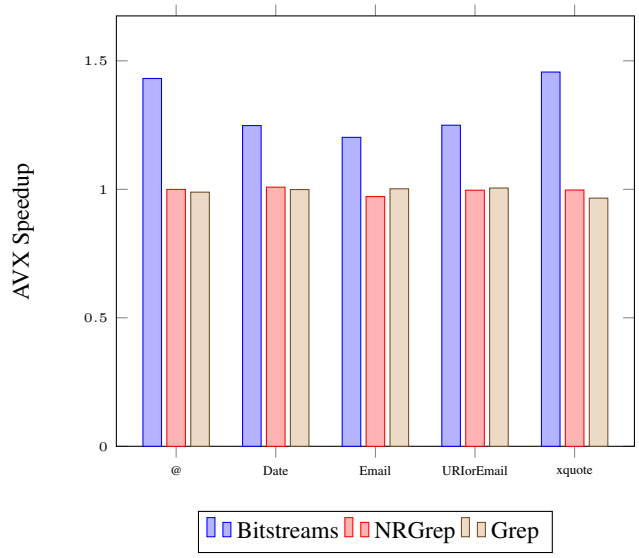

Figure 11. AVX Speedup

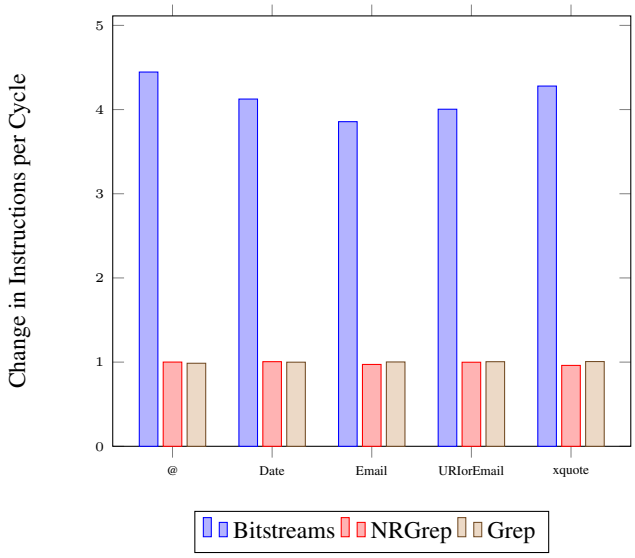

Figure 12. Change in Instructions Per Cycle With AVX

object code generates many data movement operations that are unnecessary with the AVX2 set.

As expected, there was no observable reduction in instruction count with the recompiled grep and nrgrep applications.

As shown in Figure 11 the reduction in instruction count was reflected in a considerable speed-up in the bitstreams implementation. However, the speed-up was considerably less than expected. As shown in the AVX2 version has lost some of the superscalar efficiency of the SSE2 code. This is a performance debugging issue that we have yet to resolve.

Overall, the results on our AVX2 machine were quite good, demonstrating very good scalability of the bitwise data-parallel approach.

8. GPU Implementation

To further assess the scalability of our regular expression matching using bit-parallel data streams, we implemented a GPGPU version in OpenCL. We arranged for 64 work groups each having 64 threads. Input files are divided in data parallel fashion among the 64 work groups. Each work group carries out the regular expression matching operations 4096 bytes at a time using SIMT processing. Figure 13 shows our implementation of long-stream addition on the GPU. Each thread maintains its own carry and bubble values and performs synchronized updates with the other threads using a six-step parallel-prefix style process.

Our GPU test machine was an AMD A10-5800K APU with Radeon(tm) HD Graphics having a processor speed of 4.30 GHz and 32.0GB of memory.

9. Miscellaneous

9.1 Skipping

9.2 Unicode

The introduction of Unicode as a common encoding system including the characters of all the world's written languages and notation systems has introduced some complexity for regular expression matching engines. Nevertheless, most modern tools and libraries do include some form of Unicode support, with varying degrees of performance loss.

```
inline BitBlock adc(int idx, BitBlock a, BitBlock b, __local BitBlock *carry, _
                    _local BitBlock *bubble, BitBlock *group_carry, const int carryno){
BitBlock carry_mask;
BitBlock bubble_mask;

BitBlock partial_sum = a+b;
BitBlock gen = a&b;
BitBlock prop = a^b;
carry[idx] = ((gen | (prop & ~partial_sum))&CARRY_BIT_MASK)>>(WORK_GROUP_SIZE-1-idx);
bubble[idx] = (partial_sum + 1)? 0:(((BitBlock)1)<<idx);

barrier(CLK_LOCAL_MEM_FENCE);
for(int offset=WORK_GROUP_SIZE/2; offset>0; offset=offset>>1){
carry[idx] = carry[idx]|carry[idx^offset];
bubble[idx] = bubble[idx]|bubble[idx^offset];
barrier(CLK_LOCAL_MEM_FENCE);
}

carry_mask = (carry[0]<<1)|group_carry[carryno];
bubble_mask = bubble[0];

BitBlock s = (carry_mask + bubble_mask) & ~bubble_mask;
BitBlock inc = s | (s-carry_mask);
BitBlock rslt = partial_sum + ((inc>>idx)&0x1);
group_carry[carryno] = (carry[0]|(bubble_mask & inc))>>63;
return rslt;
}
```

Figure 14 compares the performance of our SSE2, AVX and GPU implementations.

Figure 13. OpenCL 4096-bit Addition

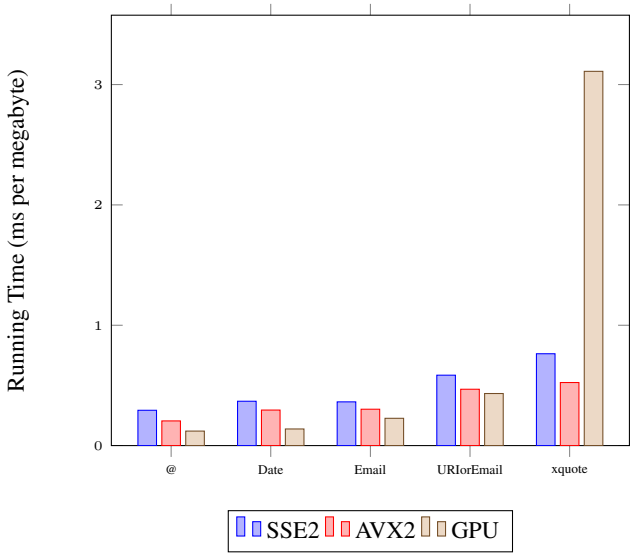

Figure 14. Running Time

UTF-8, UTF-16 and UTF-32 are the three common transformation formats of Unicode depending on whether character code points are encoded using 8-bit, 16-bit or 32-bit code units. In the case of the 32-bit code units of UTF-32, each Unicode character is encoded as a single 32-bit unit, with the high 11 bit positions all zero. A stream of UTF-32 encoded text can then be processed using a straighforward application of the techniques described above by first transforming to a set of 21 parallel bit streams for each of the significant bit positions within UTF-32.

For most practical purposes, UTF-16 can also be processed similarly, considering that each 16-bit code unit represents a single character. For the rarely used characters of the Unicode supplementary plane, two 16-bit code units are required. Such a two code unit sequence is known as a surrogate pair. Following the common practice of treating each member of a surrogate pair as pseudo-character, UTF-16 can also be processed by the straightforward transposition to 16 parallel bit streams and application of the techniques above.

UTF-8 is probably the most widely used of the Unicode formats, primarily because of its compatibility with widely deployed networking software based on 8-bit extended-ASCII character representations. UTF-8 is a variable length coding system in which each Unicode character is represented using one to four 8-bit code units.

In order to safely process UTF-8, it is necessary to validate that it is well-formed. Each byte is classified as either an ordinary ASCII byte (high bit clear: range 0x00-0x7F), a UTF-8 prefix byte of a 2-, 3- or 4- byte sequence (in ranges 0xC2-0xDF, 0xE0-0xEF, and 0xF0-F4, respectively) or as a UTF-8 suffix byte in the range 0x80-0xBF. Parallel bit stream technology achieves this validation easily and efficiently [**?**].

The UTF-8 byte classification streams produced as a byproduct of validation then enable regular expression on the UTF-8 input streams as follows. For each multibyte character used in the pattern, a character classification bitstream may be formed to identify the occurrence of such characters at each position in the source stream at which the final byte of the character is found. Figure **??** illustrates. (Describe).

Using this approach to multibyte classification, matching of a single k-byte character is straightforward. All current match posi-

tions are shifted forward $k-1$ positions and then combined with the computed character class using bitwise-and. The result is then shifted forward one position to produce the result of the multibyte character match. This method easily extends to character classes comprising multibyte characters all of the same length.

Matching a character class comprising characters of different lengths requires a slightly different strategy. In this case, we take advantage of a bitstream `nonfinal` formed from the UTF-8 byte classification streams to consist of all those positions at which a UTF-8 prefix byte is found, as well as those positions at which the second byte of a 3-byte sequence or the second or third byte of a 4-byte sequence are found. We then apply `ScanThru(current,nonfinal)` to advance all current matches to the final position of the next character in the input. Bitwise-and combination with the character class bitstream produces the result identifying matches with this class, the result is then shifted forward 1 position.

The MatchStar operation for matching arbitrary sequences of a character class can similarly take advantage of the `nonfinal` stream. For this case, we form the bitwise-or of the `nonfinal` stream with the character class stream prior to applying MatchStar. The result will propagate bits to the first character position of matches and to final positions of nonmatches. We then clear the nonmatches by combining the result stream with the `suffix` byte stream using bitwise-and.

10. Conclusion

Contributions A new class of regular expression matching algorithm has been introduced based on the concept of bit-parallel data streams together with the MatchStar operation. The algorithm is fully general for nondeterministic regular expression matching; however it does not address the nonregular extensions found in Perl-compatible backtracking implementations. Taking advantage of the SIMD features available on commodity processors, its implementation in a grep too offers consistently good performance in contrast to available alternatives. While lacking some special optimizations found in other engines to deal with repeated substrings or to perform skipping actions based on fixed substrings, it nevertheless performs competitively in all cases. The algorithm tends to scale very well with regular expression complexity, often with order-of-magnitude performance advantage over even the best of its competitors.

A parallelized algorithm for long-stream addition has also been introduced in the paper making a key contribution to the scalability of the bit-parallel matching technique overall and that of MatchStar in particular. This algorithm has enabled straightforward extension of the matching algorithm to the implementation using 256-bit AVX2 technology as well as 4096-bit SIMT implementation on an AMD GPU.

Future Work An important area of future work is to develop and assess multicore versions of the algorithm to handle regular expression matching problems involving larger rulesets than are typically encountered in the grep problem. Such implementations could have useful application in tokenization and network intrusion detection for example.

Another area of interest is to extend the capabilities of the underlying method with addition features for substring capture, zero-width assertions and possibly backreference matching. Extending Unicode support beyond basic Unicode character handling to include full Unicode character class support and normalization forms is also worth investigating.

Acknowledgments

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada and MITACS, Inc.