Bitwise Data Parallelism in Regular Expression Matching

Rob Cameron, Tom Shermer, Arrvindh Shriraman, Ken Herdy, Dan Lin, Ben Hull, Meng Lin
School of Computing Science
Simon Fraser University
Surrey, British Columbia
{cameron,shermer,ashrira,ksherdy,lindanl,bhull,linmengl}@sfu.ca

ABSTRACT

A new parallel algorithm for regular expression matching is developed and applied to the classical grep (global regular expression print) problem. Building on the bitwise data parallelism previously applied to the manual implementation of token scanning in the Parabix XML parser, the new algorithm represents a general solution to the problem of regular expression matching using parallel bit streams. 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. 5X or better performance advantage against the best of available competitors is not atypical. 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 AVX2 implementation showed dramatic reduction in instruction count and significant improvement in speed. Our GPU implementations show further acceleration.

1. INTRODUCTION

The use of regular expressions to search texts for patterns has a long history and remains an important technique. Thompson [22] 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 converted to an NFA with $O(m)$ states. Using the NFA it is possible to search a text of length n in $O(mn)$ time. Frequently, a more efficient choice is to convert an NFA into a deterministic finite automata (DFA). A DFA maintains a single active state throughout the matching process and hence, using a DFA it is possible to search a text of length n in $O(n)$ time[1].

A significant proportion of the research in fast regular expression matching can be regarded as the "quest for efficient automata" [16]. In [2], Baeza-Yates and Gonnet presented a new approach to string search based on bit-level parallelism. This technique takes advantage of the intrinsic parallelism of bitwise operations within a computer word. Thus, given a w-bit word, the number of operations that a string search algorithms performs can be reduced by a factor w. Building on this observation, the Shift-Or algorithm simulates an NFA using bitwise operations and achieves $O(\frac{nm}{w})$ time in the worst-case [14]. A disadvantage of the Shift-Or approach is an inability to skip input characters. Simple string matching algorithms, such as the Boyer-Moore family of algorithms [3, 8], skip input characters to achieve sublinear times in the average case. The Backward Nondeterministic Dawg Matching (BNDM) pattern matching algorithm [25] combines the advantages of the Shift-Or approach with the ability to skip characters. The nrgrep tool is based on the BNDM algorithm. It is generally considered the fastest grep tool for matching complex patterns, and achieves similar performance to the fastest existing string matching tools for simple patterns [14].

Recently, there has been considerable interest in the use of parallel hardware such as multicore processors (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), or specialized architectures such as the Cell Broadband Engine (Cell BE) to accelerate regular expression matching. Generally, speedups are achieved by using parallel hardware features to improve the throughput of multiple instances of a matching problem at a time, i.e., by matching against sets of patterns or multiple input streams. In contrast, our approach uses parallelism to accelerate the throughput of a single problem instance, i.e., a single regular expression matched against a single input stream.

In related work targeting multicore hardware, Scarpazza and Braudaway [21] demonstrated that text processing algorithms that exhibit irregular memory access patterns can be efficiently executed. Pasetto et al [17] 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 [20, 13] demonstrated that the Aho-Corasick (AC) string matching algorithm [1] is well-suited for parallel implementation on multicore CPUs, GPUs and the Cell BE. Salapura et al [18] advocated the use of vector-style processing for regular expressions in business analytics applications and leveraged the SIMD hardware available on multicore processors to achieve a speedup of greater than 1.8 over a range of data sizes. On the Cell Broadband Engine, Scarpazza and Russell [19] described a pattern matching implementation that delivered a throughput of 40 Gbps

[1]It is well known that the conversion of an NFA to an equivalent DFA may result in *state explosion*, i.e., the number of resultant DFA states may increase exponentially.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628079.

for a small dictionary of approximately 100 patterns and a throughput of 3.3-3.4 Gbps for a larger dictionary containing thousands of patterns. Iorio and van Lunteren [9] presented a string matching implementation for automata that achieved 4 Gbps on the Cell BE. On GPUs, Tumeo et al [23] 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 raw data throughput, although system throughput was limited to 15 Gbps [10]. Most recently, Mytkowicz et al [12] have developed a method for combining SIMD parallelism and data parallelism on multicore hardware. Of each of these related works, this approach stands out since it also focuses on the acceleration of matching against a single input stream.

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 [5]. 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 wavefronts 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 [24], Baez-Yates and Gonnet [2] and Navarro and Raffinot [15]. 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 [24] and nrgrep [14] 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 primitive 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 GPU 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 bitstream 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 GPU system. Section 9 concludes the paper with a discussion of results and areas for future work.

2. REGULAR EXPRESSIONS 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 two 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 star operator `*` specifies zero-or-more occurrences matching the previous pattern, while Kleene plus `+` 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 op-

tion in our experimental evaluation to focus our comparisons on the performance of the underlying matching algorithms.

3. 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 on 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 [11] to handle the details of compilation to block-by-block processing. On the latest processors supporting the 256-bit AVX2 SIMD operations, we also use the Parabix tool chain, but substitute a parallelized long-stream addition technique to avoid the sequential chaining of 4 64-bit additions. Our GPU implementation uses scripts to modify the output of the Parabix tools, effectively dividing the input into blocks of 4K bytes. We also have adapted our long-stream addition technique to perform 4096-bit additions using 64 threads working in lock-step SIMT fashion.

input data	a453z--b3z--az--a12949z--ca22z7--
B_7	.....
B_6	1...1..1.1..11..1.....1..11..1...
B_5	11111111111111111111111111111111
B_4	.1111...11...1...111111....1111..
B_3	....111..111.111...1.1111....1.11
B_2	.11..11...11..11....1..11.....111
B_1	...11..111...1....1...1..1.1111..
B_0	1.11.11.1.111.1111.1.1.1111...111
[a]	1..........1...1.........1.....
[z9]	....1....1...1.....1.11......1...
[0-9]	.111....1........11111.....11.1..

Figure 1: Basis and Character Class Streams

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-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]`, `[z9]`, 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 [11].

input data	a453z--b3z--az--a12949z--ca22z7--
M_1	.1..........1...1.........1.....
M_2	.1111........1...111111....111...
M_3	.....1........1.....1.11......1..

Figure 2: Marker Streams in Matching `a[0-9]*[z9]`

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]*[z9]`. Note that this is an ambiguous regular expression, which could match texts such as `a12949z` in multiple ways. 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` or `9` 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. 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. Note that all three matches from the third occurrence of `a` are correctly marked in M_3.

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	a453z--b3z--az--a12949z--ca22z7--
M_1	.1..........1...1.........1.....
$C =$ [0-9]	.111....1........11111.....11.1..
$T_0 = M_1 \wedge C$	.1.............1.........1.....
$T_1 = T_0 + C$	....1...1............1......11..
$T_2 = T_1 \oplus C$	.1111...........111111....111...
$M_2 = T_2 \vee M_1$	.1111........1...111111....111...

Figure 3: $M_2 = \textbf{MatchStar}(M_1, C)$

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.

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

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.

MatchStar differs from ScanThru of the Parabix tool chain in that it finds all matches, not just the longest match. This is necessary for general matching involving possibly ambiguous regular expressions.

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 to be considered. The body of the while consists of the compiled form of the expression `R`, taking as input the marker stream at the beginning of the iteration and producing as output all positions that can be reached from the input positions in a single step. These output positions are candidates for further iteration, but positions that have already been considered are removed. This guarantees termination of the loop; iteration continues only if a new marker position is reached that has not been previously considered as an output. 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 run-time libraries.

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.

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 [4].

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.

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

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 [11]. 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 Parabix tool chain is to incorporate 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 use the following general model using SIMD methods for constant-time long-stream addition up to 4096 bits. Related GPU solutions have been independently developed[7], however our model is intended to be a more broadly applicable abstraction. 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.

Here, the `hsimd<64>::mask(X)` and `simd<64>::spread(X)` model the minimum communication requirements between the parallel processing units (SIMD lanes or SIMT processors). In essence, we just need the ability to quickly send and receive 1 bit of information per parallel unit. The `hsimd<64>::mask(X)` operation gathers 1 bit from each of the processors to a central resource. After calculations on the gather bits are performed, we then just need an operation to invert the communication, i.e., sending 1 bit each from the central processor to each of the parallel units. There are a variety of ways in which these facilities may be implemented depending on the underlying architecture; details of our AVX2 and GPU implementations are presented later.

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 called 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 signaled 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 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.

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

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.

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

$$\texttt{q} = \texttt{i >> f}$$

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 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 particular application to regular expression matching as shown herein, it seems reasonable to expect such instructions to become available. Alternatively, it may be worthwhile to simply ensure that the `hmask` and `spread` operations are efficiently supported.

5. SSE2 IMPLEMENTATION

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 4.8.2 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 or count matches as they are encountered.

Name	Expression
@	`@`
Date	`([0-9][0-9]?)/([0-9][0-9]?)/([0-9][0-9]([0-9][0-9])?)`
Email	`([^ @]+)@([^ @]+)`
URI	`(([a-zA-Z][a-zA-Z0-9]*)://|mailto:)([^ /]+)(/[^ ]*)?|([^ @]+)@([^ @]+)`
Hex	`[ ](0x)?([a-fA-F0-9][a-fA-F0-9])+[.:,?! ]`
StarHeight	`[A-Z]((([a-zA-Z]*a[a-zA-Z]*[ ])*[a-zA-Z]*e[a-zA-Z]*[ ])*[a-zA-Z]*s[a-zA-Z]*[ ])*[.?!]`

Table 1: Regular Expressions

Comparative Implementations. We evaluate our bitwise data parallel implementation versus several alternatives. We report data for two of these: gre2p and nrgrep version 1.12. The gre2p program is a grep version implemented using the recently developed RE2 regular expression library, using a systematic DFA-based approach (as well as some NFA fallback techniques) [6]. The NFA class is represented by nrgrep, one of the strongest competitors in regular expression matching performance. We also considered GNU grep 2.10, agrep 3.41 as an alternative NFA-based implementation and pcregrep 8.12 as a backtracking implementation, but do not report data for them. GNU grep is a popular open-source implementation that is claimed to be primarily DFA-based with heuristics for important special cases. The agrep implementation does not support some of the 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 i5-4570 (Haswell) processor (3.2 GHz, 4 physical cores, 32+32 kB (per core) L1 cache, 256 kB (per core) L2 cache, 6 MB L3 cache) running the 64-bit version of Ubuntu 12.04 (Linux).

Our performance evaluation focuses on the running time of the regular expression matching process itself, excluding the preprocessing time for regular expression compilation. However, the overhead of the Parabix transform to bit stream form is included in our reported results.

Test Expressions. Each grep implementation was evaluated against the five regular expressions shown in Table 1. @ matches the at-sign character. This expression demonstrates the overhead involved in matching the simplest possible regular expression, a single character. Date, Email, and URI provide examples of commonly used regular expression. This set of expressions were modified from the *Benchmark of Regex Libraries*. Hex matches delimited byte strings in hexadecimal notation, and enforces the constraint that the number of hex digits is even. This expression illustrates the performance of a repetition operator implemented using a while loop in our system. StarHeight is an artificial expression designed to further stress while loop implementation with 4 levels of Kleene closure. All tests were run on a version of a *Linux 3Dfx howto* file of 39,421,555 bytes.

Results. Figure 5 compares each of the grep implementations, with relative performance reported in CPU cycles per byte.

The performance in matching the @ regular expression establishes the base line cost for regular expression processing. All programs report 15,788 matching lines of the 1,086,077

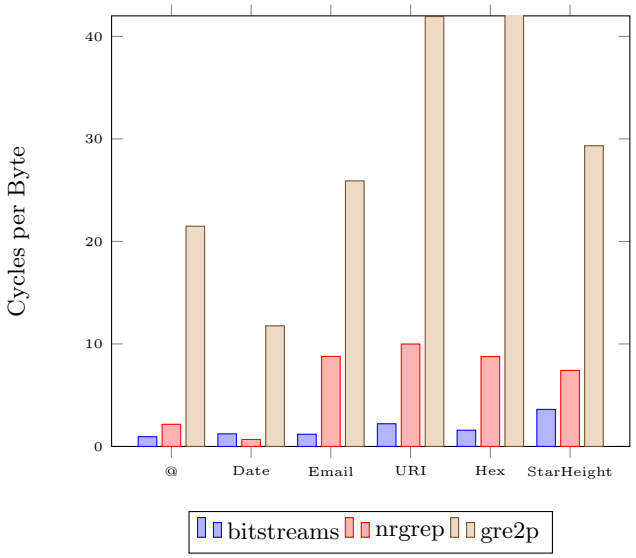

Figure 5: Cycles per Byte

lines in the file. The Parabix SSE2 implementation is clearly the fastest in this case with a cost of 0.95 CPU cycles per byte. The bulk of this represents the overhead of the Parabix transform, the bitwise logic to calculate the single `[@]` character class stream is relatively trivial. It is interesting to note that this example does not represent a baseline cost for either nrgrep or gre2p, each of these benefit from character skipping optimizations in their implementations.

Our results for the matching the Date expression to find the 668 lines containing dates show an increase from 0.95 to 1.22 cycles per byte, corresponding to the additional logic for the regular expression matching steps according to our algorithm. For this relatively simple expression, however, nrgrep outperforms our implementation by taking significant advantage of character skipping. Each time that nrgrep encounters a character that cannot appear in a date it jumps six character positions rather than searching every character in the input text. gre2p also shows a significant benefit from the character skipping optimization.

The results for the Email expression illustrate the relative advantage of the Parabix method when the expression to be matched does not permit character skipping in the NFA- or DFA-based implementations. In this example, our implementation outperforms nrgrep by a factor of 7X, and gre2p by 23X. There are 15,057 lines matching the Email regex.

The URI expression illustrates the performance of the grep programs with additional regular expression complexity. As expressions get larger, the number of steps required by the

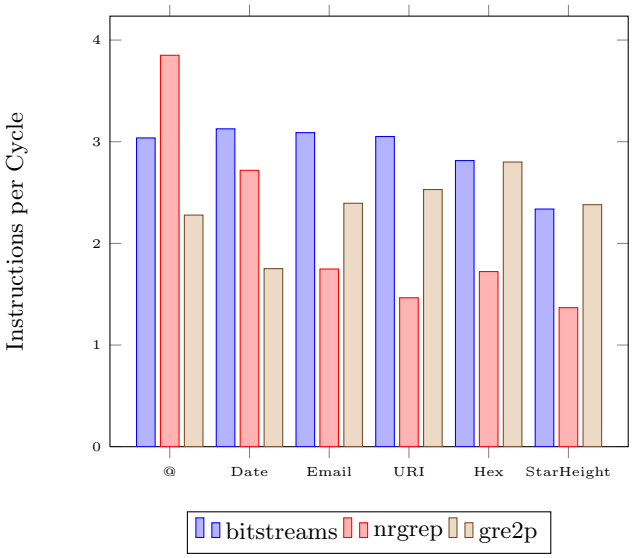

Figure 6: Instructions per Cycle

Parabix implementation increases, so the performance advantage drops to about 4.5X over nrgrep and 19X over gre2p. 32557 lines are matched by the URI regex.

The results for Hex expression illustrate the bitstreams performance in the case of a Kleene-+ operator compiled to a while loop. Performance is nevertheless quite good; our implementation uses just 1.6 cycles per byte to find the 130,243 matching lines. The gre2p program performs quite poorly here, slower than the Parabix implementation by about 70X, while nrgrep is about 5.5X slower.

A more complex triply-nested repetition structure is required by the bitstreams implementation of the StarHeight expression. In this case, there is a noticeable drop off in the performance advantage of the bitstreams implementation over the nrgrep and gre2p. Nevertheless a 2X advantage over nrgrep is maintained.

Figure 6 shows the efficiency of processor utilization achieved by the three programs on each of the test expression in terms of instructions per cycle (IPC). For the first four expressions, in particular, the bitstreams implementation uses the processor resources quite efficiently, avoiding penalties due to cache misses and branch mispredictions. However, with the while loop structures in processing the Hex and StarHeight expressions, branch mispredictions increase considerably and there is a noticeable drop-off in IPC. Both the gre2p and nrgrep suffer from significant penalties due to mispredictions in the character-skipping logic and cache misses in table lookups. The performance of nrgrep, in particular drops off with the growth in regulare expression complexity.

Overall, the bitstreams implementation significantly outperformed both nrgrep and gre2p. In addition, the performance of bitstreams scales well with regular expression complexity.

6. RUNNING-TIME COMPARISON WITH BASE 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 algorithm presented above as well as the compilers of the Parabix toolchain. Compilation is an offline process whose time is not counted in our performance measures, as each of these are research tools that have neither been optimized nor integrated. This leads to a bias in our results, as our timings for nrgrep and gre2p include the time taken for preprocessing. We minimize the bias by performing our tests with reasonably large inputs, so that the text-scanning costs dominate the preprocessing costs. We assume that the length m of regular expressions is typically less than 100 bytes and that data files are typically over 10 MB. Provided that a well-engineered implementation of our regular expression compilation algorithm together with the compilers of the Parabix tool chain requires no more than $10000m$ cycles, the overhead of compilation will not substantially increase the running time. As our regular expression algorithm is $O(m)$, and the other steps of the Parabix tool chain require $O(m \log m)$ time, we expect that such performance is well within reason. It is important to note that our algorithms construct neither NFAs nor DFAs and so are no subject to the exponential-time behaviour of NFA-to-DFA transformation.

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.

Let Σ be our input alphabet and $\sigma = |\Sigma|$. As we are comparing techniques in practice, we assume that Σ is a standard input alphabet, such as ASCII ($\sigma = 128$), UTF-8 ($\sigma = 256$), UTF-16 ($\sigma = 65536$), or UTF-32 ($\sigma = 1114112$). This assumption allows us to equate the number of bits in the encoding of a character (a parameter for the bitstream method) with $\log \sigma$.

The bitstream method compiles a regular expression of size m into bitstream code that is $O(m \log \sigma)$ statements long (with one operation per statement; it is essentially three-address code). This is translated to machine code and placed inside a loop[2] that executes once per w characters, where w is the width of the processor's word. Also inside this loop is the transposition step that converts character-encoded files into their bitstream representation; this transposition takes $O(\log \sigma \log \log \sigma)$ work per loop iteration.

In total, this is $O(m \log \sigma + \log w + \log \sigma \log \log \sigma)$ work per iteration. In current practice, we have $\log w$ around 8 (for 256-bit architectures), and $\log \sigma$ at least 7. Thus, $m \log \sigma$ will dominate $\log w$ with current and foreseeable technology–we do not expect to see $\log w$ skyrocket. So we can absorb the $\log w$ term and state the work as $O(m \log \sigma + \log \sigma \log \log \sigma)$ per iteration. We multiply this by $O(\frac{n}{w})$ iterations to give $O(\frac{n(m+\log \log \sigma) \log \sigma}{w})$ work.

We further note 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

[2]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.

most C of these additions in the loop, where C is the number of concatenation and Kleene star operations in the regular expression; $C < m$.

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.

We compare this with base NFA methods; by "base" here we mean NFA methods that do not skip input characters. The performance of input-skipping methods can be approximated by first analyzing the performance of the base method and then multiplying this by the expected fraction of examined (non-skipped) input.

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 will have $\sigma 2^{\Theta(m)}$ entries. 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 $m 2^{k-3} \sigma$ bytes, and so the collection of them takes up $\frac{m^2 2^{k-3} \sigma}{k}$ 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.

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.

In Table 2, we summarize the theoretical analysis of these NFA methods, listing the number of table lookups per input character and the size of the tables for various values of m, the number of states. We assume the ASCII character set ($\sigma = 128$); any of the UTF character sets would yield larger tables.

Of particular importance to the speed of NFA methods is whether the table lookups result in cache hits or not. If the tables are small enough, then they will fit into cache and lookups will all be cache hits, taking minimal time. In this case, the time per input character will be a small constant times the number of lookups.

If the tables are not small enough to fit into cache, some proportion of the lookups will generate cache misses. This will stall the processor and these stalls will come to dominate the computation time. In this case, the time per input character will be some large constant (a cache miss can take about two orders of magnitude longer than a cache hit) times the number of lookups.

	k	1	4	8	$\frac{m}{2}$	m
lookups		m	$\frac{m}{4}$	$\frac{m}{8}$	2	1
memory (KiB)	m					
	5	0.8	1.6	12.5	1.3	2.5
	10	3.1	6.2	50.0	10.0	160.0
	15	7.0	14.1	112.5	120.0	7680.0
	20	12.5	25.0	200.0	640.0	327680.0
	25	19.5	39.1	312.5	6400.0	13107200.0

Table 2: lookups per character and memory consumed by tables in NFA methods (in kibibytes)

Using 256KiB as an estimate of the size of a current standard data cache, we can consider those entries of Table 2 above 256 to be relatively slow. We can summarize these theoretical predictions by saying that the NFA methods with small k scale well with an increase in NFA states, but with large k the method is limited to a small number of states.

We can now directly (but roughly) compare the NFA methods with bitstream methods. Consider small-k (say, $k <= 4$) NFA methods. For the reasonable range of m, the tables fit into cache. The running time is predicted to be a small constant times the $\frac{m}{k} >= \frac{m}{4}$ lookups. The small constant, which we will under approximate with 4 cycles, is for the table addressing computation, combining the lookups with boolean OR, and final state detection and handling. Thus, the running time per input character may be lower-bounded by $4 * \frac{m}{4}$, or simply m, cycles.

Our method, on the other hand, takes time $O(\frac{m \log \sigma + \log\log\sigma \log\sigma}{w})$ per input character, where the constant inside the big-Oh is approximately 2 for the first part of the numerator and 6 for the second part. Furthermore, we expect no cache misses due to the regular stride of our memory accesses. For UTF-8 (or ASCII), this time becomes at most $2\frac{8m}{w} + 6\frac{24}{w} = \frac{16m+144}{w}$ cycles per character.

For processors with a 128-bit word, this is $\frac{16m+144}{128} = \frac{m}{8} + \frac{9}{8}$ cycles per character. Comparing this with the at least m cycles per character of the base NFA methods, we expect these NFA methods to be competitive with our method only when the size of the regular expression is 1. As the size of the regular expression increases, we expect our method to approach a factor-of-8 improvement over the base NFA methods.

In theory, our improvement factor should scale closely with the word size; so that for processors with a 256-bit word, we expect an 16x improvement, and for processors with a 512-bit word, we expect a 32x improvement. In practice, there is some reduction in these improvement factors due to the instruction sets of larger-width processors not yet being as versatile as those of 128-bit processors. (For example, full-width addition is not yet supported.)

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 op-

portunity 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.

```
bitblock_t spread(uint64_t bits) {
  uint64_t s = 0x0000200040008001 * bits;
  uint64_t t = s & 0x0001000100010001;
  return _mm256_cvtepu16_epi64(t);
}
```

Figure 7: AVX2 256-bit Spread

AVX2 256-Bit Addition. Bitstream addition at the 256-bit block size was implemented using the long-stream addition technique. The AVX2 instruction set directly supports the `hsimd<64>::mask(X)` operation using the `_mm256_movemask_pd` intrinsic, extracting the required 4-bit mask directly from the 256-bit vector. The `hsimd<64>::spread(X)` is slightly more problematic, requiring a short sequence of instructions to convert the computed 4-bit increment mask back into a vector of 4 64-bit values. One method is to use the AVX2 broadcast instruction to make 4 copies of the mask to be spread, followed by appropriate bit manipulation. Another uses multiplication to first spread to 16-bit fields as shown in Figure 7.

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

Figure 8 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 very well with reductions in instruction count over a factor of two in every case except for StarHeight. 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 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 9 the reduction in instruction count was reflected in a significant speedup in the bitstreams implementation in all cases except StarHeight. However, the speedup was considerably less than expected. The bitstreams code on AVX2 has suffered from a considerable reduction in instructions per cycle compared to the SSE2 implementation, likely indicating that our grep implementation has become memory-bound. However, the performance of StarHeight deserves particular comment, with an actual slowdown observed. When moving to 256 positions at a time, the controlling while loops may require more iterations than working 128 positions at a time, because the iteration must continue

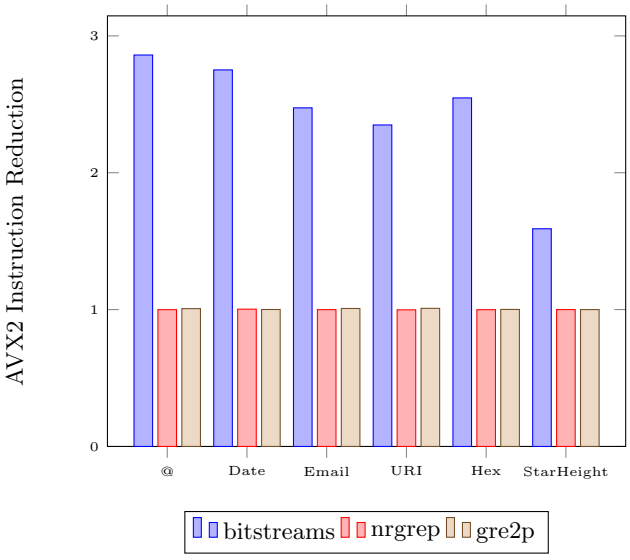

Figure 8: Instruction Reduction

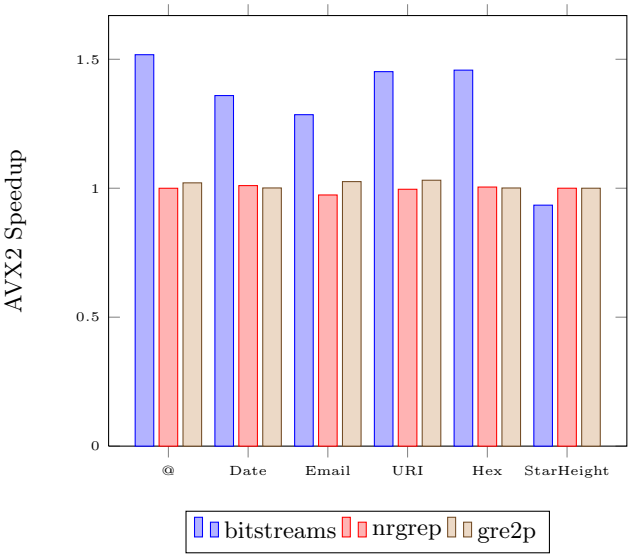

Figure 9: AVX Speedup

Expression	Bitstream grep Speedup		
	vs. nrgrep	vs. gre2p	vs. GNU grep -e
At	3.5X	34X	1.6X
Date	0.76X	13X	48X
Email	9.5X	28X	12X
URIorEmail	6.6X	27X	518X
Hex	8.1X	105X	267X
StarHeight	1.9X	7.6X	97X

Table 3: Speedups Obtained

as long as there are any pending markers in the block. Nevertheless, the overall results on our AVX2 machine were quite encouraging, demonstrating very good scalability of the bitwise data-parallel approach. Significantly, the @ regular expression is matched at 0.63 cycles/byte using our AVX2 implementation indicating a significant reduction in the overhead cost of the Parabix transform.

Table 3 shows the final performance results showing the speedup factors achieved by the Parabix/AVX2 implementation vs nrgrep and gre2p. We have also added comparison with GNU grep (version 2.16), as it is well known and sometimes used as a basis for comparisons.

8. GPU IMPLEMENTATION

To further assess the scalability of our regular expression matching using bit-parallel data streams, we implemented a GPU version in OpenCL. We arranged for 64 work groups each having 64 threads. The size of work group and number of work groups is chosen to provide the best occupancy as calculated by the AMD App Profiler. 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. Although the GPU does not directly support the mask and spread operations required by our long-stream addition model, we are able to simulate them using shared memory. Each thread maintains its own carry and bubble values in shared memory and performs synchronized updates with the other threads using a six-step parallel-prefix style process. Others have implemented long-stream addition on the GPU using similar techniques, as noted previously.

We performed our test on an AMD Radeon HD A10-6800K APU machine. On the AMD Fusion systems, the input buffer is allocated in pinned memory to take advantage of the zero-copy memory regions where data can be read directly into this region by the CPU and also accessed by the GPU for further processing. Therefore, the expensive data transferring time that is needed by traditional discrete GPUs is hidden and we compare only the kernel execution time with our SSE2 and AVX implementations as shown in Figure 10. The GPU version gives up to 55% performance improvement over SSE version and up to 40% performance improvement over AVX version. However, because of implementation complexities of the triply-nested while loop for the StarHeight expression, it has been omitted.

Although we intended to process 64 work groups with 4096 bytes each at a time rather than 128 bytes at a time on SSE or 256 bytes at a time on AVX, the performance improvement is less than 60%. The first reason is hardware limitations. Our kernel occupancy is limited by register usage

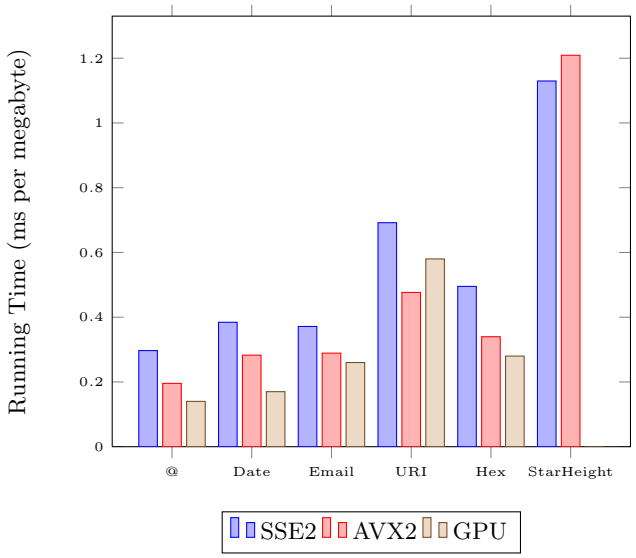

Figure 10: Running Time

and not all the work groups can be scheduled at the same time. The second reason is that the long-stream addition implemented on GPU is more expensive than the implementations on SSE or AVX. Another important reason is the control flow. When a possible match is found in one thread, the rest of the threads in the same work group have to execute the same instructions for further processing rather than jump to the next block with a simple IF test. Therefore, the performance of different regular expressions is dependent on the number of long-stream addition operations and the total number of matches of a given input. Perhaps surprisingly, the overhead of the Parabix transformation was not a dominant factor, coming in at 0.08 ms/MB.

9. DISCUSSION

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 grep offers consistently good performance in contrast to available alternatives. For moderately complex expressions, 10X or better performance advantages over DFA-based gre2p and 5X performance advantage over nrgrep were frequently seen. 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.

A model for parallelized long-stream addition has also been presented in the paper, allowing our techniques to scale beyond the blocks of 128 bytes we use with the SSE2 implementation. This model allowed straightforward extension to the 256-byte block size used in our AVX2 implementation and should continue to scale well up for SIMD vectors up to

4096 bytes in length based on 64-bit additions. The model also supports GPU implementation with some additional effort.

Related Work. Much of the previous work in parallelizing of regular processing has dealt with the problem of using parallel resources to handle multiple instances of a matching problem in parallel. It is thus complementary to our approach which focuses on parallelization to accelerate the matching of a single instance. From this perspective, the recent work of Mytkowicz et al [12] stands out as an important comparator in that it also focusses on acceleration of matching for a single input stream. Mytkowicz use the SIMD byte-shuffle capabilities found, for example, in the SSE3 instruction sets to perform small-table parallel lookups for multiple potentially active states of a FSM. Data parallelism is achieved by initially considering all possible states at the beginning of each data segment, but then relying on convergence and range-coalescing optimizations to quickly reduce the number of active states in play. Examining a large collection of regular expressions used in practice, these techniques were found to be effective, allowing matching to proceed with just one or two shuffles per input symbol.

However, the Mytkowicz approach is still fundamentally considering input elements one byte at a time and would be hard pressed to compete with our reported results of 1-3 CPU cycles per input byte. It is also dependent on the availability of the SIMD byte-shuffle operation, which is unavailable in SIMD instructions sets such as SSE2 and ARM Neon, for example. Our SIMD implementation relies only on the availability of SIMD pack operations to efficiently implement the Parabix transform; SIMD pack is widely available in current SIMD instruction sets. It is also a special case of the more general shuffle operations and hence available on any processor that supports byte shuffle. The Parabix approach also has the further advantage that performance scales with increasing SIMD instruction width, as illustrated by our AVX2 performance results in comparison to SSE2.

It is perhaps surprising that the classic nrgrep application is still competitive in performance for expressions that allow the BNDM algorithm to perform significant character skipping. Although the length of possible skipping reduces with the complexity of the input expression considered, many applications of grep searching tend to use simple expressions in practice. Nevertheless, the Parabix approach offers consistently high performance often faster than nrgrep by a factor of 5X or more.

Ongoing and Future Work. Based on the techniques presented here a fully integrated grep version with a dynamic code generator implemented with LLVM is being developed by another team working with the Parabix technology (Dale Denis, Nick Sumner and Rob Cameron). An initial version is available at `http://parabix.costar.sfu.ca/icGREP`. With icgrep-0.8, total compile-time overhead to translate our test expressions into executable x86 code ranges from 0.002 seconds to 0.008 seconds for our test cases. Although this represents a tolerable overhead of 0.64 cycles/byte for our 40 MB test file, we expect that a substantial reduction of this overhead is feasible.

Further work on the compilation algorithms includes the extending the algorithms to use MatchStar in Kleene-* repetitions beyond those of single characters (bytes). Each such extension would replace while-loop iteration with addition and bitwise logic. The UTF-8 example shows this is possible for repetitions of variable-length byte sequences having particular synchronizing properties, for example.

Future work also includes the development of multicore versions of the underlying algorithms to further accelerate performance and to handle regular expression matching problems involving larger rule sets than are typically encountered in the grep problem. Such implementations could have useful application in tokenization and network intrusion detection for example. Additional GPU implementation work could take advantage of specialized instructions available on particular platforms but not generally avaiable through OpenCL. For both multicore and GPU implementations, data-parallel division of input streams could benefit from techniques such as the principled speculation of Zhao et al [26], for example.

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

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

10. REFERENCES

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, June 1975.

[2] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Communications of the ACM*, vol. 35, no. 10, pp. 74–82, 1992.

[3] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[4] R. D. Cameron, "A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding," in *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2008, pp. 91–98.

[5] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich, "Parallel scanning with bitstream addition: An XML case study," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 2–13.

[6] R. Cox, "Regular expression matching in the wild," 2010. [Online]. Available: http://swtch.com/r̃sc/regexp/regexp3.html

[7] R. Crovella, "Long stream addition with CUDA," Stack Overflow question 12957116, 2012. [Online]. Available: http://stackoverflow.com/questions/12957116

[8] R. N. Horspool, "Practical fast searching in strings," *Software: Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[9] F. Iorio and J. V. Lunteren, "Fast pattern matching on the cell broadband engine," in *2008 Workshop on Cell Systems and Applications (WCSA), affiliated with the*, 2008.

[10] C. Lin, C. Liu, L. Chien, and S. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," *IEEE Transactions on Computers*, vol. 62, no. 10, 2013.

[11] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *18th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2012, pp. 1–12.

[12] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 529–542.

[13] J. Naghmouchi, D. P. Scarpazza, and M. Berekovic, "Small-ruleset regular expression matching on GPGPUs: quantitative performance analysis and optimization," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 337–348. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810130

[14] G. Navarro, "Nr-grep: A fast and flexible pattern matching tool," *Software Practice and Experience (SPE*, vol. 31, p. 2001, 2000.

[15] G. Navarro and M. Raffinot, "A bit-parallel approach to suffix automata: Fast extended string matching," in *Combinatorial Pattern Matching*. Springer, 1998, pp. 14–33.

[16] ——, "Fast and flexible string matching by combining bit-parallelism and suffix automata," *ACM Journal of Experimental Algorithmics (JEA)*, vol. 5, p. 2000, 1998.

[17] D. Pasetto, F. Petrini, and V. Agarwal, "Tools for very fast regular expression matching," *Computer*, vol. 43, no. 3, pp. 50–58, 2010.

[18] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira, "Accelerating business analytics applications," in *18th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2012, pp. 1–10.

[19] D. P. Scarpazza and G. F. Russell, "High-performance regular expression scanning on the Cell/BE processor," in *Proceedings of the 23rd International Conference on Supercomputing*. ACM, 2009, pp. 14–25.

[20] D. P. Scarpazza, "Top-performance tokenization and small-ruleset regular expression matching," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 3–32, 2011.

[21] D. P. Scarpazza, O. Villa, and F. Petrinni, "Fast string searches & multicore processors mapping fundamental algorithms on parallel hardware," p. 20, 2008.

[22] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[23] A. Tumeo, O. Villa, and D. Sciuto, "Efficient pattern matching on GPUs for intrusion detection systems," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*. ACM, 2010, pp. 87–88.

[24] S. Wu and U. Manber, "Agrep - a fast approximate pattern-matching tool," *Usenix Winter 1992*, pp. 153–162, 1992.

[25] ——, "Fast text searching: allowing errors," *Communications of the ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[26] Z. Zhao, B. Wu, and X. Shen, "Challenging the "embarrassingly sequential": parallelizing finite state machine-based computations through principled speculation," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 543–558.