Bitwise Data Parallelism with LLVM: The ICgrep Case Study

Anonymous

No Institute Given

Abstract. Bitwise data parallelism has recently been shown to have considerable promise as the basis for a new, fundamentally parallel, style of regular expression processing. This paper examines the application of this approach to the development a full-featured Unicode-capable open-source grep implementation. Constructed using a layered architecture combining Parabix and LLVM compiler technologies, icGrep is the first instance of a potentially large class of text processing applications that achieve high performance text processing through the combination of dynamic compilation and bitwise data parallelism. In performance comparisons with several contemporary alternatives, 10X or better speedups are often observed.

1 Introduction

The venerable Unix `grep` program is an everyday tool widely used to search for lines in text files matching a given regular expression pattern. Historical comments ...

Unicode regular expression matching adds performance challenges...

Efforts to improve the performance of regular expression matching through parallelization have generally concentrated on the use of SIMD, multicore or GPU technologies to accelerate multiple instances of independent matching problems. Scarpazza [13] used SIMD and multicore parallelism to accelerate small ruleset tokenization applications on the Cell Broadband Engine while Valaspura [12] built on these techniques to accelerate business analytics applications using SSE instructions on commodity processors. Zu et al [14] use GPU technology to implement NFA-based regular expression matching with parallelism devoted both to processing a compressed active state array as well as to handling matching of multiple packet instances. These works have not generally tackled Unicode matching problems.

Using parallel methods to accelerate matching of a single pattern on a single input stream is more difficult. Indeed, of the 13 dwarves identified in the Berkeley overview of parallel computing research, finite state machines (FSMs) are considered the hardest to parallelize (embarassingly sequential) [1]. However, some success has been reported recently along two independent lines of research. Mytkowicz et al [10] use SIMD shuffle operations to implement composable DFA transitions using dynamic convergence to reduce the number of states in play

at any one time and range coalescing to compact the transition tables. Unfortunately, the method seems unlikely to apply well to Unicode regular expression matching problems, which routinely require thousands of DFA states for named Unicode properties. Building on the Parabix framework, Cameron at al [4] introduce regular expression matching using the bitwise data parallel approach together with the MatchStar primitive for efficient implementation of Kleene-* character-class repetitions.

In this paper, we report on the use of the implementation of a full Unicode regular expression search tool, building on the bitwise data parallel methods of the Parabix framework combined with the dynamic compilation capabilities of LLVM. The result is ICgrep, a high-performance, full-featured open-source grep implementation with systematic support for Unicode regular expressions addressing the requirements of Unicode Technical Standard #18 [5]. As an alternative to classical grep implementations, ICgrep offers dramatic performance acceleration in ASCII-based and Unicode matching performance alike.

The remainder of this paper is organized as follows. Section 2 presents background material dealing with Unicode regular expressions, LLVM, the Parabix framework and regular expression matching techniques using bitwise data parallelism. Section 3 expands on previous work on bitwise data parallelism by more fully characterizing the paradigm and documenting important techniques. Section 4 addresses the issues and performance challenges associated with meeting Unicode regular expression requirements and presents the extensions to the Parabix techniques that we have developed to address them. Section 5 describes the overall architecture of the ICgrep implementation with a focus on the integration of Parabix and LLVM technologies. Section 6 evaluates the performance of ICgrep on several types of matching problems with contemporary competitors, including the latest versions of GNU grep, pcregrep, ugrep of the ICU (International Component for Unicode) and re2grep. Section 7 concludes the paper with remarks on developing the Parabix+LLVM framework for other applications as well as identifying further research questions in Unicode regular expression matching with bitwise data parallelism.

2 Background

2.1 Unicode Regular Expression Requirements

Unicode is system for organizing the characters from all languages and symbol systems into a single numeric space and encoding those values in convenient formats for computerized processing. The numeric values form a space of integer *codepoints* from 0 through hexadecimal 10FFFF. The computerized formats represent codepoint values as (possibly variable length) sequences of fixed-width *code units*. UTF-8 represents each codepoint using a sequence of one to four octets (8-bit bytes), UTF-16 represents each codepoint using one or two 16-bit code units and UTF-32 represents each codepoint as a single 32-bit unit. The format used most often for storage and transmission of Unicode data is UTF-8; this is the format assumed through this paper.

Traditional grep syntax is oriented towards string search using regular expressions over ASCII or extended-ASCII byte sequences. A grep search for a line beginning with a capitalized word might use the pattern "`^[A-Z][a-z]+`" ("extended" syntax). Here, "`^`" is a zero-width assertion matching only at the start of a line, "`[A-Z]`" is a character class that matches any single character in the contiguous range of characters froms A through Z, while the plus operator in "`[a-z]+`" denotes repetition of one or more lower case ASCII letters.

While explicit listing of characters of interest is practical with ASCII, it is less so with Unicode. In the Unicode 7.0 database, there are 1490 characters categorized as upper case and 1841 categorized as lower case. Rather than explicit listing of all characters of interest, then, it is more practical to use named character classes, such as `Lu` for upper case letters and `Ll` for lower case letters. Using these names, our search might be rewritten to find capitalized words in any language as "`^[[:Lu:]][[:Ll:]]+`" (Posix syntax) or "`^\p{Lu}\p{Ll}+`" (Perl-compatible syntax). The Unicode consortium has defined an extensive list of named properties that can be used in regular expressions.

Beyond named properties, Unicode Technical Standard #18 defines additional requirements for Unicode regular expressions, at three levels of complexity [5]. Level 1 generally relates to properties expressed in terms of individual Unicode codepoints, while level 2 introduces complexities due to codepoint sequences that form grapheme clusters, and level 3 relates to tailored locale-specific support. We consider only Unicode level 1 requirements in this paper, as most grep implementations are incomplete with respect the requirements even at this level. The additional level 1 regular expression requirements primarily relate to larger classes of characters that are used in identifying line breaks, word breaks and case-insensitive matching. Beyond this, there is one important syntactic extension: the ability to refine character class specifications using set intersection and subtraction. For example, `[\p{Greek}&&\p{Lu}]` denotes the class of upper case Greek letters, while `[\p{Ll}--\p{ASCII}]` denotes the class of all non-ASCII lower case letters.

2.2 Parabix

The Parabix toolchain is a set of compilers and run-time libraries designed to take advantage of the SIMD features of commodity processors to support high-performance streaming text processing based on a bit-parallel transform represenation of text. In this representation, a text T is represented as a set of parallel bit streams B_i, such that bit j of stream B_i is the i^{th} bit of character code unit j of T. The Parabix methods have been used to accelerate Unicode transcoding [2], protein search [6], XML parsing [3], and, most recently, regular expression search [4].

2.3 LLVM

The LLVM compiler infrastructure is a set of modular compiler components and tools organized around a powerful generic intermediate representation (LLVM

IR) that is agnostic with respect to source language and code-generation targets. Beginning as an MSc research project at the University of Illinois [7], LLVM is now an open-source codebase supported by a broad community of researchers, developers and commercial organizations.

LLVM features a flexible multi-stage compilation structure that can be organized in passes in many ways, including fully dynamic just-in-time compilation. Of particular importance to the icGrep project, the LLVM IR supports arbitrarily-sized vectors of arbitrary-width integers, well suited to code generation targetting the SIMD integer instructions of commodity processors.

2.4 Parabix Regular Expression Matching

3 Bitwise Data Parallel Paradigm and Methods

The introduction of the method of bitwise data parallelism adds a fundamentally new paradigm for regular expression matching to complement the traditional approaches using DFAs, NFAs or backtracking. Whereas the traditional approaches are all sequential in nature, performing some form of state transition processing on one input code unit at a time, the bitwise data parallel approach takes a conceptually parallel view of the input stream. Rather than parallelizing existing sequential approaches, the method introduces parallelism from the ground up.

As a new paradigm, there is much research to do in building upon the basic framework, characterizing performance depending on input patterns and texts, developing methods for special cases and so on. Here we make some small contributions to the general framework and methods before moving on to discuss Unicode issues.

One important aspect of the bitwise data parallel approach is transposition of input data. In previous work, the Parabix transform has been reported as imposing an amoritized cost of 1 CPU cycle/input byte, when working with SSE2 [8]. This is consistent with `icGrep` results. However, the cost of the cost of this transposition can be hidden through multithreading and pipeline parallelism, having one core perform work ahead performing transposition, while another comes behind to perform matching. We discuss this further in Section 5.

The treatment of character and character class recognition is another area of fundamental difference between the traditional sequential methods and the bitwise approach. It is here that the clearest separation of the sequential and parallel approaches occurs. In the sequential approaches, characters are processed sequentially with table lookups or jump tables used for each transition. In the bitwise data parallel approach, all calculations of character class bit streams are done completely in parallel using bitwise logic.

In the bitwise paradigm, the MatchStar operation elegantly finds all possible matches for Kleene-* repetitions of characters or character classes using a single long-stream addition operation. Interestingly, the MatchStar operation also has application to parallelized long-stream addition[4], as well as use in Myers bit-parallel edit distance algorithm[9]. In the next section, we show how MatchStar can be extended for UTF-8 sequences.

We have incorporated an elegant technique for bounded repetitions in icGrep. This technique allows the matches to Cm, n for some character class C, lower bound m and upper bound n to be determined in $\lceil \log_2 m \rceil + \lceil \log_2 n - m \rceil$ steps. Let C_k be the bit stream identifying positions at which the k prior input bytes are all in C. Then the observation that $C_{2k} = C_k \wedge (C_k \texttt{<<} k)$ enables positions meeting the lower bound to be determined. An upper bound k similarly involves excluding those positions not within k of the pending markers (from the previous match step).

A final general technique worth mentioning is that related to input skipping. For sequential matching, the Boyer-Moore method is well known for the possible skipping through input positions of up to the length of the search string for fixed-string search. NR-grep [11] extends this skipping to regular expression search using the BNDM (backward non-determinstic dawg matching) method. Is there an input-skipping method for the bitwise parallel paradigm? The answer is yes: whenever the bit vector of match positions in play for the current input block reduce to all zero, the remainder of the pattern can be skipped for processing the block. This method has been implemented in icGrep.

4 Unicode Regular Expression Methods

4.1 toUTF8 Transformation

The icGrep parser generates an abstract syntax tree (AST) that represents an input regular expression over code points. This AST is passed as input to a toUTF-8 transformation that generates a new AST that represents the equivalent regular expression over UTF-8 byte sequences. The transformation accomplishes this by first determining the number of UTF-8 bytes that are required to represent each code point contained within each character class. The code points are then split into sequences of bytes, with each byte containing the necessary UTF-8 prefix. The UTF-8 encoded bytes are each assigned to a new character class in the new AST. For an example, consider the following regular expression that consists entirely of multibyte Unicode characters: '`\u{244}[\u{2030}-\u{2137}]`'. The AST from the parser would represent this as a sequence starting with a character class containing the code point `0x244` followed by a second character class containing the range from `0x2030` to `0x2137`. After being passed through the toUTF-8 transformation this AST would become considerably more complex. The first code point in the sequence would be encoded as the two byte sequence '`\u{C9}\u{84}`'. The character class containing the range, which is a range of three byte sequences would be expanded into the series of sequences and alternations that are necessary to specify all of the possible byte encodings that would be contained within the range. The UTF-8 encoded regular expression for the range `[\u{2030}-\u{2137}]` would be encoded as follows:

```
\xE2((\x84[\x80-\xB7])|(([\x81-\x83][\x80-\xBF])|(\x80[\xB0-\xBF])))
```

The benefit of transforming the regular expression immediately after parsing from being a regular expression over code points into a regular expression over bytes is that it simplifies the rest of the compiler, as the compiler then only needs to be concerned with single bytes as opposed to code points, which vary in size.

4.2 UTF-8 Advance using ScanThru

Each bit position in the character class bitstream of a single byte ASCII character marks either the location of, or the absence of the search character. To match the location of a character the current position of the cursor is checked to see if the bit is set and then the cursor is advanced by one position. To match the position of a multibyte search character the procedure is different. For multibyte UTF-8 characters of length `k`, it is the last `(k-1)`th byte of the multibyte sequence in the bitstream that marks the character's location. Figure 1 illustrates the process of matching a character class of a three byte multibyte character. The locations of the first two bytes of each character in the character class CC have been marked with zeros while the bitstream M_1 marks the current cursor positions. To match multibyte characters, first a *nonfinal* helper bitstream must be formed. The *Nonfinal* bitstream is formed by marking the locations of the first bytes of two byte sequences, the first two bytes of three byte sequences, and the first three bytes of any four byte sequences. The `ScanThru(current, nonfinal)` operation is then applied, in order to advance all of the current cursor positions to the locations of the `(k-1)`th final character positions. To find any matches the result is then compared with the bits that are set in the UTF-8 character class bitstream. After this, the cursor is advanced by one position to be ready for the next matching operation.

$$
\begin{aligned}
CC &\ \texttt{001...001.........} \\
M_1 &\ \texttt{1.....1....1.....} \\
nonfinal &\ \texttt{11....11.........} \\
T_1 = ScanThru(M_1, nonfinal) &\ \texttt{..1.....1.........} \\
T_2 = CC \wedge T_1 &\ \texttt{..1.....1.........} \\
M_2 = Advance(M_1) &\ \texttt{...1.....1.........}
\end{aligned}
$$

Fig. 1: Processing of a Multibyte Sequence

4.3 MatchStar for Unicode character classes

Figure 2 shows how the MatchStar operation can be used to find all matches of a multibyte UTF-8 sequence. The problem is to find all matches to the character class CC that can be reached from the current cursor positions in M_1. First

we form two helper bitstreams *initial* and *nonfinal*. The initial bitstream marks the locations of all single byte characters and the first bytes of all multibyte characters. Any full match to a multibyte sequence must reach the initial position of the next character. The nonfinal bitstream consists of all positions except those that are final positions of UTF-8 sequences. It is used to "fill in the gaps" in the CC bitstream so that the MatchStar addition can move through a contiguous sequence of one bits. In the figure, the gaps in CC are filled in by a bitwise-or with the nonfinal bitstream to produce T_1. This is then used as the basis of the MatchStar operation to yield T_2. We then filter these results using the initial bitstream to produce the final set of complete matches in M_2.

$$
\begin{aligned}
CC &\ \texttt{001001001.........} \\
M_1 &\ \texttt{1..........1..1..} \\
initial &\ \texttt{1..1..1..1..1..1..} \\
nonfinal &\ \texttt{11.11.11.11.11.11.} \\
T_1 = nonfinal \vee CC &\ \texttt{11111111111.11.11.} \\
T_2 = MatchStar(M_1, T_1) &\ \texttt{111111111111......} \\
M_2 = T_2 \wedge initial &\ \texttt{1..1..1..1........}
\end{aligned}
$$

Fig. 2: Processing of MatchStar for a Multibyte Sequence

4.4 Predefined Unicode classes

Every character in the Unicode database has been assigned to a general category classification based upon the character's type. As the categories seldom change the parallel bitstream equations for the categories have been statically compiled into icGrep. Each of the categories contain a large number of code points, therefore an *If Hierarchy* optimization has been included in the statically compiled implementation of each category. The optimization works under the assumption that most input documents will only contain the code points of the characters from a small number of writing systems. Processing the blocks of code points for characters that exist outside of this range is unnecessary and will only add to the total running time of the application. The optimization tests the input text to determine the ranges of the code points that are contained in the input text and it only processes the character class equations and the regular expression matching equations for the code point ranges that the input text contains. The optimization tests the input text with a series of nested *if else* statements, using a process similar to that of a binary search. As the nesting of the statements increases, the range of the code points in the conditions of the *if* statements narrow until the exact ranges of the code points in the text has been found.

4.5 Character Class Intersection and Difference

4.6 Unicode Case-Insensitive Matching

5 Architecture

5.1 Regular Expression Preprocessing

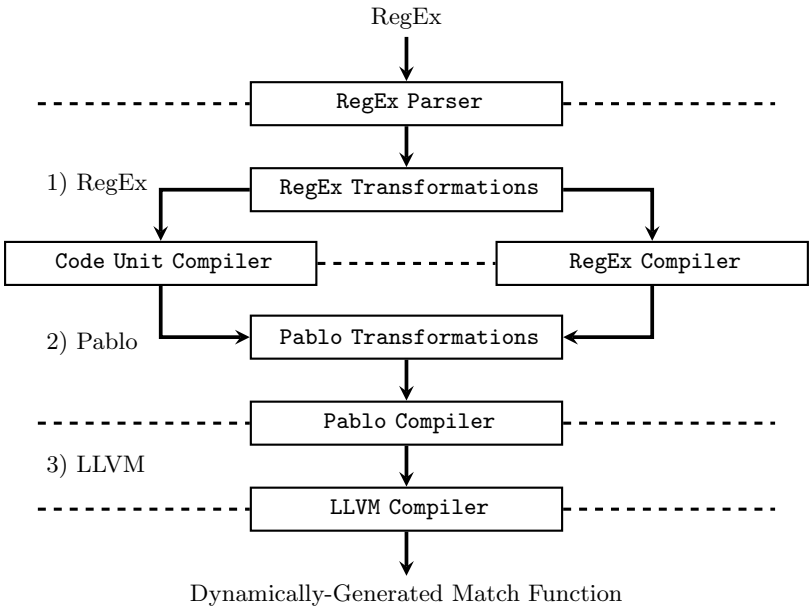

Fig. 3: icGrep Architectural Diagram

As show in Figure 3, icGrep is composed of three logical layers: RegEx, Pablo and the LLVM layer, each with their own intermediate representation (IR), transformation and compilation modules. As we traverse the layers, the IR becomes significantly more complex as it begins to mirror the final machine code. The RegEx Parser validates and transforms the input RegEx into an abstract syntax tree (AST). The AST is a minimalistic representation that, unlike traditional RegEx, is not converted into a NFA or DFA for further processing. Instead, icGrep passes the AST into the transformation module, which includes a set of RegEx specific optimization passes. The initial *Nullable* pass, determines whether the RegEx contains any prefixes or suffixes that may be removed or modified whilst still providing the same number of matches as the original expression. For example, "`a*bc+`" is equivalent to "`bc`" because the Kleene Star (Plus) operator matches zero (one) or more instances of a specific character.

The *toUTF8* pass converts the characters in the input RegEx into the equivalent expression(s) that represent the sequences of 8-bit code units necessary to identify the presence of a particular character. Since some characters have multiple logically equivalent representations, such as ????, this may produce nested sequences or alternations. This is described in more detail in §4.1. To alleviate this, the final *Simplification* pass flattens nested sequences and alternations into their simplest legal form. For example, "`a(b((c|d)|e))`" would become "`ab(c|d|e)`" and "`([0-9]{3,5}){3,5}`", "`[0-9]{9,25}`".

The RegEx layer has two compilers: the Code Unit and RegEx Compiler, both of which produce Pablo IR. Recall that the Pablo layer assumes a transposed view of the input data. The *Code Unit Compiler* transforms the input code unit classes, either extracted from the RegEx or produced by the *toUTF8* transformation, into a series of bit stream equations. The *RegEx Compiler* assumes that these have been calculated and transforms the RegEx AST into a sequence of instructions. For instance, it would convert any alternations into a sequence of calculations that are merged with `OR`s. The results of these passes are combined and transformed through a series of typical optimization passes, including dead code elimination (DCE), common subexpression elimination (CSE), and constant folding. These are necessary at this stage because the RegEx AST may include common subsequences that are costly to recognize in that form. Similarly, to keep the Code Unit Compiler a linear time function, it may introduce redundant IR instructions as it applies traditional Boolean algebra transformations, such as de Morgan's law, to the computed streams. An intended side-effect of these passes is that they eliminate the need to analyze the data-dependencies inherent in the carry-bit logic, which is necessary for some Pablo instructions but problematic for optimizers to reason about non-conservatively. The Pablo Compiler then converts the Pablo IR into LLVM IR. This is a relatively straightforward conversion: the only complexities it introduces is the generation of Phi nodes, linking of statically-compiled functions, and assignment of carry variables. It produces the dynamically-generated match function used by the icGrep.

5.2 Dynamic Grep Engine

As shown in Figure 4, icGrep takes the input data and transposed it into 8 parallel bit streams through S2P module. The required streams, e.g. line break stream, can then be generated using the 8 basis bits streams. The JIT function retrieves the 8 basis bits and the required streams from their memory addresses and starts the matching process. Named Property Library that includes all the predefined Unicode categories is installed into JIT function and can be called during the matching process. JIT function returns one bitstream that marks all the matching positions. A match scanner will scan through this bitstream and calculate the total counts or write the context of each match position.

We can also apply a pipeline parallelism strategy to further speed up the process of icGrep. S2P and Required Streams Generator can be process in a separate thread and start even before the dynamic compilation starts. The output of S2P and Required Streams Generator, that is the 8 basis bits streams and

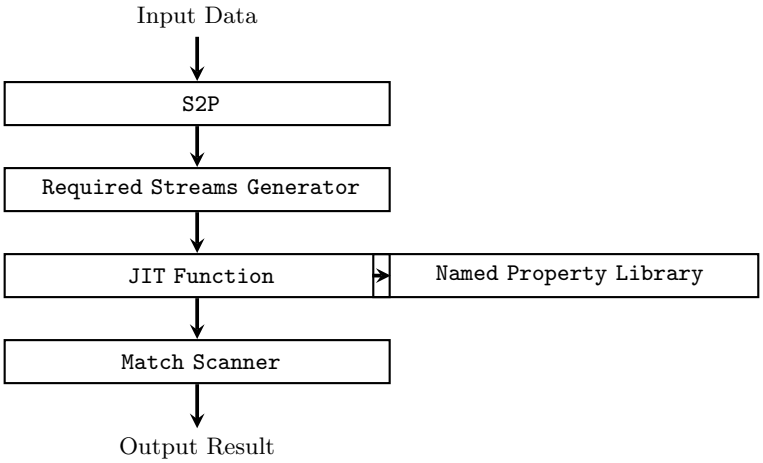

Fig. 4: icGrep Execution Diagram

the required streams, needs to be stored in a shared memory space so that the JIT function can read from it. To be more efficient of memory space usage, we only allocate limit amount of space for the shared data. When each chunk of the shared space is filled up with the bitstream data, the thread will start writing to the first chunk if it is released by JIT function. Otherwise, it will wait for JIT function until it finishes processing that chunk. Therefore, the performance is depended on the slowest thread. In the case that the cost of transposition and required stream generation is more than the matching process, we can further divide up the work and assign two threads for S2P and Required Streams Generator.

6 Evaluation

In this section, we report on the evaluation of ICgrep performance, looking at three aspects. First we consider a performance studies in a series of Unicode regular expression search problems in comparison to the contemporary competitors, including pcre2grep released in January 2015 and ugrep of the ICU 54.1 software distribution. Then we move on to investigate some performance aspects of ICgrep internal methods, looking at the impact of optimizations and multithreading.

6.1 ICgrep vs. Contemporary Competitors

6.2 Optimizations of Bitwise Methods

In order to support evaluation of bitwise methods, as well as to support the teaching of those methods and ongoing research, icGrep has an array of command-line

options. This makes it relatively straightforward to report on certain performance aspects of ICgrep, while others require special builds.

For example, the command-line switch `-disable-matchstar` can be used to eliminate the use of the MatchStar operation for handling Kleene-* repetition of character classes. In this case, icGrep substitutes a while loop that iteratively extends match results. Surprisingly, this does not change performance much in many practical cases. In each block, the maximum iteration count is the maximum length run encountered; the overall performance is based on the average of these maximums throughout the file. But when search for XML tags using the regular expression `<[^!?][^>]*>`, a slowdown of more than 2X may be found in files with many long tags.

The `-disable-log2-bounded-repetition` flag allows these effectiveness of the special techniques for bounded repetition of byte classes to be assessed. A slowdown of 30% was observed with the searches using the regular expression `(^|[ ])[a-zA-Z]{11,33}([.!? ]|$)`, for example.

To assess the effectiveness of inserting if-statements, the number of non-nullable pattern elements between the if-tests can be set with the `-if-insertion-gap=` option. The default value in icGrep is 3, setting the gap to 100 effectively turns of if-insertion. Eliminating if-insertion sometimes improves performance by avoiding the extra if tests and branch mispredications. For patterns with long strings, however, there can be a substantial slowdown; searching for a pattern of length 40 slows down by more than 50% without the if-statement short-circuiting.

ICgrep also provides options that allow various internal representations to be printed out. These can aid in understanding and/or debugging performance issues. For example, the option `-print-REs` show the parsed regular expression as it goes through various transformations. The internal Pablo code generated may be displayed with `-print-pablo`. This can be quite useful in helping understand the match process. It also possible to print out the generated LLVM IR code (`-dump-generated-IR`), but this may be less useful as it includes many details of low-level carry-handling that obscures the core logic.

6.3 Single vs. Multithreaded Performance

7 Conclusion

References

1. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
2. Robert D Cameron. A case study in simd text processing with parallel bit streams: UTF-8 to UTF-16 transcoding. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 91–98. ACM, 2008.

3. Robert D Cameron, Ehsan Amiri, Kenneth S Herdy, Dan Lin, Thomas C Shermer, and Fred P Popowich. Parallel scanning with bitstream addition: An XML case study. In *Euro-Par 2011 Parallel Processing*, pages 2–13. Springer, 2011.
4. Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, PACT '14, pages 139–150, New York, NY, USA, 2014. ACM.
5. Mark Davis and A Heninger. Unicode technical standard 18, Unicode regular expressions. *The Unicode Consortium*, 2012.
6. James R Green, Hanan Mahmoud, and Michel Dumontier. Modeling tryptic digestion on the Cell BE processor. In *Canadian Conference onElectrical and Computer Engineering (CCECE '09), 2009.*, pages 701–705. IEEE, 2009.
7. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.
8. Dan Lin, Nigel Medforth, Kenneth S Herdy, Arrvindh Shriraman, and Rob Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.
9. Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.
10. Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 529–542. ACM, 2014.
11. Gonzalo Navarro. NR-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.
12. Valentina Salapura, Tejas Karkhanis, Priya Nagpurkar, and Jose Moreira. Accelerating business analytics applications. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–10. IEEE, 2012.
13. Daniele Paolo Scarpazza. Top-performance tokenization and small-ruleset regular expression matching. *International Journal of Parallel Programming*, 39(1):3–32, 2011.
14. Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. In *ACM SIGPLAN Notices*, volume 47, pages 129–140. ACM, 2012.