

Regular Expression Acceleration on the Micron Automata Processor: Brill Tagging as a Case Study

Keira Zhou; Jack Wadden; Jeffrey J. Fox; Ke Wang; Donald E. Brown, Fellow IEEE; Kevin Skadron, Fellow IEEE

University of Virginia

Charlottesville, VA 22904 USA

{qz4aq, jpw8bd, jjf5x, kewang, brown, skadron}@virginia.edu

Abstract—Brill tagging is a classic rule-based algorithm for part-of-speech (POS) tagging that assigns tags, such as nouns, verbs, adjectives, etc., to input tokens. Due to the intense memory requirements of rule matching, CPU implementations of the Brill tagging algorithm have been found to be slow. We show that Micron’s Automata Processor (AP)—a new computing architecture that can perform massively parallel pattern matching—can greatly accelerate the second stage of Brill tagging via rule template matching. The 218 contextual rules are first converted into regular expressions (regex). Regex is used widely in natural language processing (NLP) tasks, thus, this case study involving Brill Tagging also shows how the AP might accelerate other applications that are able to be framed as regexes. We compare single-threaded, and multi-threaded versions of Regex matching on an Intel i7 CPU, an Intel XeonPhi co-processor, and the AP. The results show a 63.90X speed-up using the AP as a regex accelerator over the fastest multi-threaded CPU version. We also investigate how performance of regex matching on both CPU architectures varies depending on the complexity of the regex. Taken together, these results demonstrate the potential for significant performance improvements by using accelerators for various NLP computational tasks, particularly those that involve rule-based or pattern-matching approaches.

Keywords—Part-of-speech tagging; Regular Expressions; the Automata Processor; Natural Language Processing; Multithreading

I. INTRODUCTION

Natural Language Processing (NLP) allows human-machine interactions, drawing insights from data contained in the emails, documents and other unstructured materials, translation between languages, etc., and has become increasingly important in the era of big data. There are many tasks in an NLP pipeline and they all contribute to the final quality of the results for the goal task. Therefore, increasing both the speed and the accuracy of different NLP tasks is an important area of research.

Part-of-speech (POS) Tagging assigns a part of speech tag, such as noun, verb, adjective, adverb, etc. to input tokens, [1]. This has an important role in NLP as it prepares the tokens with information needed for other tasks, such as question answering [2] and information retrieval [3]. POS tagging algorithms are commonly categorized into two groups: rule-based approaches and stochastic approaches.

Brill Tagging is a classic rule-based POS tagging algorithm that is widely used [25]. It is also called a transformation-based error-driven tagging algorithm [6].

After the tagger is trained on a corpus, a two-stage process is applied to new untagged corpora. The first stage tags each word to its most frequent POS based on the training corpora, and a second stage updates the tags, correcting possible errors based on some contextual rules. The Brill algorithm has shown relatively high accuracy in some applications [9]. However, both training and tagging of corpora using Brill tagging is computationally expensive [7]. Specifically, it may require RKn elementary steps to tag an input of n words with R contextual rules with at most K tokens of context [8].

Because traditional single-threaded implementations of these algorithms on CPUs do not perform well, these algorithms cannot scale to the massive amount of information being created in the midst of the “Big Data” era. Therefore researchers are now exploring multi-threaded implementations, exploiting parallel programming frameworks such as MPI, OpenCL, MapReduce, etc. on multiple servers, or individual multi-core processors. However, while parallelism does increase the number of problems that can be solved simultaneously, it does not address the poor performance of individual CPU cores.

The Micron Automata Processor (AP) [10] is a novel non-Von Neumann semiconductor architecture that can be programmed to execute thousands of non-deterministic finite automaton (NFA) in parallel to identify patterns in a data stream. Zhou et al. [28] proposed an implementation of the second stage of Brill tagging on the AP. The AP’s massively parallel rule matching capability was shown to significantly improve the performance of Brill Tagging compared to implementation on a single CPU. However, Brill rule matching is embarrassingly parallel, and might easily be accelerated by matching rules in parallel on multi-core CPUs or many-core accelerators like Intel’s XeonPhi™. Prior work did not address the potential of new, general purpose accelerators.

In this work, we convert Brill rules into regular expressions and compare multi-threaded regular expression matching performance of three different parallel architectures, a 6-core (12 thread) Intel i7-5820k CPU, Intel’s Knights Corner XeonPhi™3100 many-core co-processor, and Micron’s AP. We also investigate resource utilization on the AP, showing that the device is underutilized for this task, and could support many more, or more complex contextual rules to improve accuracy. Finally, we study how the complexity of the regular expression impacts the performance of matching on the AP

versus the chosen CPU architectures. The results of this work motivates using the AP to accelerate all regex related NLP tasks.

II. BACKGROUND AND RELATED WORK

A. POS tagging

POS Tagging can be viewed as a preprocessing stage for other common NLP tasks. The task was initially done manually and these manually tagged corpora provide training data for automated taggers [11] [12].

POS tagging algorithms can be categorized into two groups: rule-based approaches and stochastic (statistical) approaches. State-of-art stochastic based approaches include conditional random field models [7], maximum entropy Markov models [13], and hidden Markov models [14]. Brill tagging is one of the first and most widely used rule-based approaches [25]. While Brill tagging has comparable accuracy to state-of-the-art POS tagging algorithms, its potential for parallel acceleration could offer a dramatic speed advantage over the most accurate state-of-the-art techniques. Thus, we focus on Brill tagging as our target application for evaluation.

When performing a POS tagging task, choosing a standard tagset is important. Larger tagsets provide more information about the corpora but are harder to accurately tag. Smaller tagsets are easy to tag but leave out information about the corpora [24]. Two commonly used tagsets for POS tagging are the Brown (87 tags) [4] and Penn Treebank (36 tags) [5] tagsets.

B. Brill Tagging

The Brill algorithm has three steps. 1) It first trains on a corpus. This generates the most frequent tag for all recorded tokens as well as contextual rules for updating the tags. After training a two-stage process tags new untagged corpora. 2) The first stage assigns the most frequent tag to the tokens in the new corpora. 3) The initial tags are then updated based on the rules generated from the training corpora. This process produces 218 rules for the Brown Corpus and 284 rules for the Wall Street Journal corpus. Two rules from Brown Corpus are shown below:

1) *NN (noun) VB (verb) PREV TAG TO (to) [15]*

Explanation: If current word is tagged as NN, the preceding word is tagged as TO, then change the current tag into VB

Example: to/TO conflict/NN with/IN [updated into] to/TO conflict/VB with/IN

2) *IN (preposition) RB (adverb) WDAND2AFT (current word and 2 words after) as as [16]*

Explanation: The Penn Treebank tagging style manual specifies that in the collocation *as...as*, the first *as* is tagged as an adverb and the second is tagged as a preposition. Since *as* is most frequently tagged as a preposition in the training corpus, the initial state tagger will mis-tag the phrase *as tall as* as *as/preposition tall/adjective as/preposition*.

Example: as/IN tall/JJ (adjective) as/IN [updated into] as/RB tall/JJ as/IN

Our work focuses on reducing computational cost in the second stage of the tagging process (step 3 above). Because

each rule in this stage tags the words in a window spanning three positions before and after the focus word [17], these contextual rules can be easily implemented in parallel on the AP, thus reducing a task of RKn steps (input of n words with R contextual rules with at most K tokens of context) to n steps.

C. Regular Expressions

Regular expressions (regex) are a compact language for representing patterns in strings of characters. They were defined alongside regular languages and are only capable of matching, or recognizing strings in regular languages. There is a many-to-one mapping between regular expressions and regular languages. The following are some major components of regex:

Grouping Or: Parenthesis indicates a grouping of multiple different expressions. Each expression is separated by a ‘|’ indicating that any of the expressions within the parenthesis can match in parallel.

Wildcards: Wildcards are additions to represent arbitrary characters in an input string. The ‘.’ for example is meant to represent a single character of the input string.

Quantifiers: Quantifiers act on the previous expression and define repeating characters or sequences. The symbol ‘?’ specifies that there are must be either zero or exactly one of the previous expression.

Character Classes: Character classes represent sets of characters and are shorthand for groupings of individual characters.

III. MICRON’S AUTOMATA PROCESSOR

A physical embodiment of the AP is not yet available; however, we do have access to Micron’s simulator (SDK) of the AP. This allows us to design automata and simulate the on-chip processes and performance without access to physical silicon.

A. Major Components of the AP

There are three major components on the AP: State-Transition-Elements (STEs), Counter Elements and Boolean Elements, among which, the STE is the core component used for matching characters and character sets of the input stream.

One STE can match an 8-bit user-specified character symbol or set of symbols in a single clock cycle and STEs can connect to each other via edges. Each STE has two states: *activated* and *matched*. Only *activated* STEs will be able to accept the next input symbol to perform a *match* against the user-specified symbol within that STE. Once the symbol on an STE is *matched*, the STEs connected to it will be *activated* to accept the next input symbol and *match* that against their user-specified symbols.

Besides STEs, the Counter Element is used to count activations. It only activates once incoming activations reach a user-specified threshold. Once the threshold is reached, the counter can produce a report or activate STEs that are connected to it. There are also Boolean Elements that function as logic gates on activation signals such as AND, OR, NOR, etc.


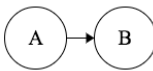
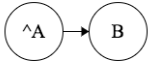
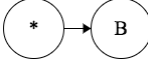

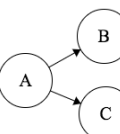

B. Programming and Execution Environment

Users can design their Automata structures using an XML-like language, Automata Network Markup Language (ANML). The ANML code is then compiled and loaded onto the processor.

Once the design code is loaded onto the chip, then a scanning-matching task is performed. The processor will take the input data as a stream and each STE in every automaton on the device will consume at a rate of 133 MB/s.

STEs can be configured as “starting STEs”. Starting STEs are either always activated, matching against the entire input data stream, or only activated on the first input symbol. STEs can be configured to “accept” outputting a single bit report. The reporting STEs will report if they are activated and matched. Each report bit contains an offset number of the cycle each reporting element reported, as well as the ID of the reporting element. Table I provides graphic illustrations of basic STE functions.

TABLE I. GRAPHIC ILLUSTRATIONS OF BASIC STE FUNCTIONS

	A starting STE: It can either be <i>start-of-data</i> , which will only match the first symbol of the input data and matches against A; Or <i>all-input-start</i> , which accepts every symbol from the input and matches against Symbol A
	Matching - Activating: Symbol A is matched on the first STE, and the second STE is activated
	Negation Matching – Activating: Whenever Symbol A is NOT matched on the first STE, the second STE is activated
	Any Symbol Matching – Activating: A “*” means to match any input symbols
	Self-activating: The STE activates itself when the symbol A is matched
	Matching – Activating 2 STEs: Symbol A is matched on the first STE, and both of the STEs on the right are activated
	A reporting STE: Reports when the symbol A is matched

C. Hardware Resources

One single AP chip contains two independent half-cores, each of which has 24,576 STEs. Thus a chip contains a total of 49,152 STEs among which 6144 can report. One chip also has 768 Counter Elements and 2304 Combinatorial Elements.

An AP board contains 32 chips totaling 1,572,864 STEs that can work in parallel. This enables fast, and highly parallel pattern matching operations [18]. The designs presented in this work can easily fit inside a single AP chip.

IV. CONVERTING BRILL RULES INTO REGEX

Zhou, et al. [28] describe one way of implementing the second stage of Brill tagging on the AP with a resulting speed-up of 40X using the AP compared to a single-threaded CPU.

This exciting result motivates us to explore other applications that can benefit from the AP. In particular, since the Brill rules can be expressed as a regex matching problem, we convert these 218 rules into regex and compare the matching performance on the AP with both top-of-the-line multithread CPUs and many-core CPU accelerators. This experiment not only shows the promising matching ability of the AP, it also provides motivation for using the AP for other NLP tasks that have regex or pattern matching components.

A. Brill Tagging Initial Steps

Brill tagging is implemented in C and openly available [19]. The public implementation uses the Penn Treebank tagset and contains 218 contextual rules trained from the Brown Corpus. The original Brill tagging also contains Lexical rules as well as rules for tagging unknown words, but our work focuses solely on the contextual rules.

We use this implementation to run the first stage tagging and write the intermediate result into a separate file. This intermediate result serves as the input for the second stage tagging. Table II shows the sample input data.

B. Brill Rules as Regular Expressions

Among the 218 rules, there are 19 different structures. The example structures and their meanings [20][21] are listed in Table III. We will use the two rules mentioned in Section II.B as our example:

*NN VB PREV TAG TO: / [^/]+/TO [^/]+/NN /
IN RB WDAND2AFT as as: / as/IN [^/]+/[^]+ as/[^]+ /*

TABLE II. EXAMPLE INPUT DATA

This/DT session/NN ./, for/IN instance/NN ./, may/MD have/VBP insured/VBN a/DT financial/JJ crisis/NN two/CD years/NNS from/IN now/RB ./.

TABLE III. 19 STRUCTURES OF 218 RULES

Rule ID	Rule Content	Rule Meaning
1	PREVWD	Preceding word is ...
2	PREVTAG	Preceding Tag is ...
3	PREV1OR2TAG	One of the two preceding words is tagged as
4	PREV1OR2OR3TAG	One of the three preceding words is tagged as
5	WDAND2AFT	The current word is ... and the word two after is ...
6	PREV1OR2WD	One of the two preceding words is
7	NEXT1OR2TAG	One of the two following words is tagged as
8	NEXT1OR2OR3TAG	One of the three following words is tagged as
9	NEXTTAG	Following word is tagged

		as
10	NEXTWD	Following word is
11	WDPREVTAG	The preceding word is tagged as ... and the current word is ...
12	WDNEXTTAG	The current word is ... and the following word is tagged as ...
13	SURROUNDTAG	The preceding word is tagged as ... and the following word is tagged as ...
14	PREVBIGRAM	The two preceding words are tagged as ... and ...
15	NEXTBIGRAM	The two following words are tagged as ... and ...

We converted all 218 rules into equivalent regex. There is an API in the AP workbench that takes regex, converts them to NFAs, and optimizes and compiles them onto the AP chip directly.

V. TEST DATA AND RESULT

A. Test Data

To test our design we used a subset of the Brown Corpus [4] downloaded from NLTK website [22].

Brown Corpus is an American English corpus that is divided into 500 samples with over 2000 words each. Each sample begins at the beginning of a sentence. The Corpus represents varieties of prose such as political, sports, financial press, government documents. We tested our implementation with a file of size 2.2MB which has 2,198,493 characters.

B. Execution Environment

We compared the matching performance of the AP, multithread CPU and Intel’s XeonPhi™ many-core accelerator. The CPU we used is a 6-core (12 thread) Intel i7-5820K running at 3.30GHz. The Knight’s Corner XeonPhi™3100 chip used in the experiment has 61, small, in-order cores, of which one is a master core. The XeonPhi™ co-processor is supported by an i7 CPU host processor.

There are two different ways of running a program on XeonPhi™. One is to cross-compile the program on the host CPU and then offload the executable onto the XeonPhi™ and run the entire program on the device. The other is to specify which part of the program to offload onto the device in advance. After compiling the program on the host CPU, the executable is run on the host but only the offloaded portion will run in parallel on the device. For our experiments, we used the first method of execution, which is to run the program “natively” on XeonPhi™.

For the i7 CPU and XeonPhi™ implementations, we used the POSIX regex library [29] for regex matching and parallelized matching using POSIX pthreads in C++ . The execution time recorded for the C++ implementation is the wall clock time for regex matching only.

Because the AP consistently consumes one symbol, and accomplishes all parallel NFA transitions, per cycle, we can easily estimate AP performance by multiplying the number of input bytes by the cycle time. The frequency of the first generation AP architecture is advertised as 133MHz or 7.5

nanoseconds per clock cycle. Thus the number of clock cycles necessary to accomplish any task equals the number of characters contained in a single file.

C. Results

1) Matching performance of regex implementation for the original 218 rules across all computer architectures

Figure 1 shows the execution time comparison between Intel i7, XeonPhi™, and the AP. We can see that for both the i7 and the Xeon Phi, the performance reaches a plateau at a certain number of threads, indicating an on-chip bottleneck other than parallelism (number of parallel threads) limiting overall performance.

By comparing the performance of Intel i7 and XeonPhi™, we can see a clear advantage of running the program on the host Intel i7. Even a single thread on the i7 is about ~28X faster than on the XeonPhi™. There are 12 cores on the i7 and 60 cores on the XeonPhi™. Comparing the best performance of the i7 and the XeonPhi™, we still see a 14X advantage using the Intel i7. Clearly performance is limited by a feature other than the number of cores, such as an architectural difference, or oversubscribed shared on-chip, such as memory bandwidth. We leave a detailed performance analysis for future work.

There are a couple of possible reasons behind this behavior of the device. The i7 is arguably the highest-performing general purpose out-of-order CPU available today, while the Knight’s Corner XeonPhi™ uses slower in-order cores. Another possibility is that the much larger caches in Intel i7 most likely give a huge benefit for rule fetches for the NFAs used for regex processing in the POSIX regex library. Thus, regex processing, even if embarrassingly parallel, may not always benefit from a greater number of cores over other resources such as cache size or memory bandwidth.

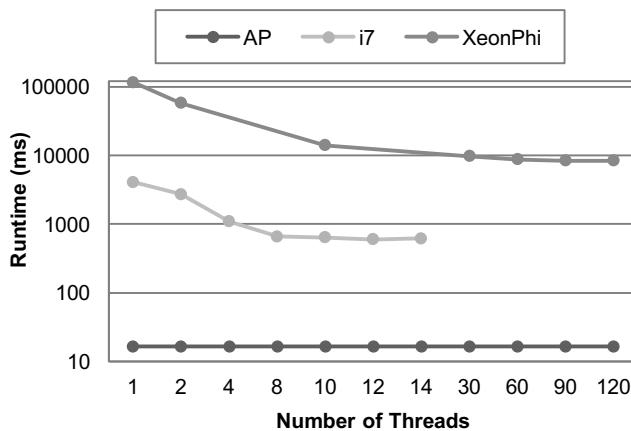


Figure 1. Execution Time Comparison between XeonPhi™, Intel i7 and the AP. The XeonPhi™ implementation always performs worse than even a single i7 core, suggesting resources other than core-count as the application bottleneck.

2) Matching performance of regex with different complexity

In order to test how the complexity of the regex impacts performance on different hardware, we separate regex rules

into 3 different categories – 200 simple (exact match), 200 original (200 regexes from the original 218 rules), and 200 complex (all the regexes have “alternation” e.g. boy|girl). Complexity is measured by counting the number of STEs required to implement the automata on the AP. Table IV reports the STE usage of all three categories.

TABLE IV. RESOURCE UTILIZATION DIFFERENCE OF THE AP CHIP FOR THREE CATEGORIES

	STE Usage
200 Simple	2737
200 Original	2934
200 Complex	5843

Table V shows the performance of Intel i7 with 12 threads and the best performance on XeonPhi™ comparing to the AP in each category. Since the Intel i7 always has better performance when compared to the XeonPhi™, we only report AP speed-up over the Intel i7. Figure 2 plots the speed-up the AP can gain over the Intel i7 in terms of the complexity of the rules. The XeonPhi™ incurs a similar penalty for complexity increase as the Intel i7, indicating that complexity of regex rules is not the main bottleneck to performance, and any regex matching is extremely expensive on a XeonPhi™ core compared to an i7 core.

TABLE V. BEST MATCHING PERFORMANCE COMPARISON BETWEEN XEONPHI™, INTEL I7 AND THE AP ON DIFFERENT RULE COMPLEXITY

2.2 MB file size (2,198,493 characters); Time in microseconds (μs)			
	200 Simple (2737 STEs)	200 Original (2934 STEs)	200 Complex (5843 STEs)
Xeon Phi	8,159,165μs (20 Threads)	8,367,107μs (100 Threads)	8,882,996μs (200 Threads)
Intel i7 (12 Threads)	513,631μs	563,720μs	1,053,676μs
AP	16,489 □μs		
AP Speed-up over Intel i7	31.15X	34.19X	63.90X

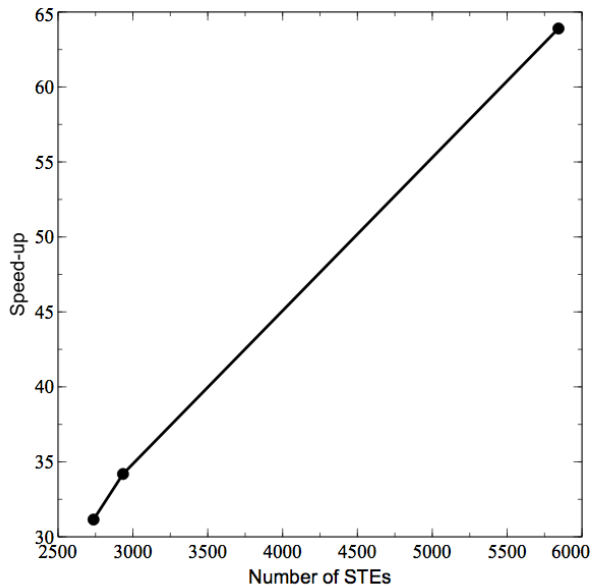


Figure 2. Speed-up of regex processing on the AP over the Intel i7

From the result, we can see that the execution time on both Intel i7 and the XeonPhi™ increases as the complexity of regex increases. This is expected; more complex rules require more CPU instructions to process. On the other hand, the matching time on the AP is only related to the length of the input string, regardless of the complexity of the regex. Although more complex regexes will require more STEs, the 218 implemented Brill rules occupy only about 6% of the resources of a single AP chip, allowing for many more complex rules without any performance penalty. Even if all rules do not fit onto a single AP chip, other chips can be added to scale to the required number of resources. As long as all the regexes can fit onto the available AP chips, the execution time will stay the same if the length of the input string does not change.

VI. CONCLUSIONS AND FUTURE WORK

The Micron Automata Processor (AP) is a novel non-Von Neumann semiconductor architecture which can be programmed to implement massively parallel pattern matching.

In this work, we investigated how to implement Brill tagging, an important step in part-of-speech tagging, on the AP. By converting the Brill rules into regular expressions (regex), we could compare regex matching performance across three different architectures, an Intel i7 CPU, Intel’s XeonPhi™ many-core co-processor, and the AP.

This work also analyzed the Regex matching performance differences across these same computer architectures. In theory, the AP can achieve a significant speed-up in terms of regex matching function, a function small cores in common scientific accelerators are bad at. The AP is capable of processing more complex rules and larger rule sets; furthermore, the more complex the rules are and the larger the rule set is, the bigger the improvement the AP can gain over any von-Neumann processor. The results also show a great potential of using the AP for other NLP tasks with the same pattern matching nature.

Future work should also explore other regex matching libraries other than POSIX for the C++ implementation, especially given the vastly different performance of the POSIX regex library on different CPU architectures. Different libraries may have different matching speeds, and should be investigated.

Another extension of this work is to compare the regex performance on the AP, against GPU and FPGA implementations [30,31], although we suspect NFA-based pattern matching on the GPU’s SIMD cores will suffer similar limitations as the XeonPhi™ cores. Furthermore, the advantages of the AP for regex processing—especially complex rules and large rule sets—over FPGA solutions has been described by Dlugosch et al [32].

ACKNOWLEDGMENTS

The authors thank the Center for Future Architectures Research (C-FAR), Micron Technology for providing access to the AP SDK, and Matt Tanner for his insights into using the AP.

REFERENCES

- [1] Voutilainen, Aro. "Part-of-speech tagging." *The Oxford handbook of computational linguistics* (2003): 219-232.
- [2] Yu, Hong, and Vasileios Hatzivassiloglou. "Towards answering opinion questions: Separating facts from opinions and identifying the polarity of opinion sentences." *Proceedings of the 2003 conference on Empirical methods in natural language processing. Association for Computational Linguistics*, 2003.
- [3] Krovetz, Robert. "Homonymy and polysemy in information retrieval." *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics. Association for Computational Linguistics*, 1997.
- [4] Francis, W.N. & Kucera, H. Brown Corpus Manual. Brown University, 1964.
- [5] Santorini, Beatrice. "Part-of-speech tagging guidelines for the Penn Treebank Project (3rd revision)." 1990.
- [6] Brill, Eric. "A simple rule-based part of speech tagger." *Proceedings of the workshop on Speech and Natural Language. Association for Computational Linguistics*, 1992.
- [7] Lafferty, John, Andrew McCallum, and Fernando CN Pereira. "Conditional random fields: Probabilistic models for segmenting and labeling sequence data.", 2001.
- [8] Roche, Emmanuel, and Yves Schabes. "Deterministic part-of-speech tagging with finite-state transducers." *Computational linguistics* 21.2 (1995): 227-253.
- [9] Hasan, Fahim Muhammad, Naushad UzZaman, and Mumit Khan. "Comparison of different POS Tagging Techniques (N-Gram, HMM and Brill's tagger) for Bangla." *Advances and Innovations in Systems, Computing Sciences and Software Engineering. Springer Netherlands*, 2007. 121-126.
- [10] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, 2014.
- [11] B. Greene and G. Rubin, "Automatic Grammatical Tagging of English", Technical Report, Department of Linguistics, Brown University, Providence, Rhode Island, 1971.
- [12] S. Klein and R. Simmons, "A computational approach to grammatical coding of English words", *JACM* 10, 1963.
- [13] McCallum, Andrew, Dayne Freitag, and Fernando CN Pereira. "Maximum Entropy Markov Models for Information Extraction and Segmentation." *ICML*. 2000.
- [14] Kupiec, Julian. "Robust part-of-speech tagging using a hidden Markov model." *Computer Speech & Language* 6.3 (1992): 225-242.
- [15] Brill, Eric. "Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging." *Computational linguistics* 21.4 (1995): 543-565.
- [16] Brill, Eric. "Some advances in transformation-based part of speech tagging." *arXiv preprint cmp-lg/9406010* (1994).
- [17] Van Halteren, Hans, Jakub Zavrel, and Walter Daelemans. "Improving data driven wordclass tagging by system combination." *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1. Association for Computational Linguistics*, 1998.
- [18] Roy, Indranil, and Srinivas Aluru. "Finding Motifs in Biological Sequences Using the Micron Automata Processor." *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE*, 2014.
- [19] Brill, Eric. the Department of Computer and Information Science, University of Pennsylvania, and the Spoken Language Systems Group, Laboratory for Computer Science, MIT, 1994.
http://www.tech.plym.ac.uk/soc/staff/guidbugm/software/RULE_BASED_TAGGER_V.1.14.tar.Z
- [20] Yonghong Mao Natural Language Processing Module. Cornell University, October 1997.
http://www.csic.cornell.edu/201/natural_language/.
- [21] Megyesi, Beáta. "Brill's rule-based part of speech tagger for Hungarian." Master's thesis, University of Stockholm, 1998.
- [22] NLTK Corpora. Natural Language Toolkit. Web. 2013.
- [23] Brill, Eric. "Unsupervised learning of disambiguation rules for part of speech tagging." *Proceedings of the third workshop on very large corpora. Vol. 30. Somerset, New Jersey: Association for Computational Linguistics*, 1995.
- [24] Xia, Fei. "The part-of-speech tagging guidelines for the Penn Chinese Treebank (3.0).", 2000.
- [25] Mohammad, Saif, and Ted Pedersen. "Guaranteed pre-tagging for the brill tagger." *Computational Linguistics and Intelligent Text Processing. Springer Berlin Heidelberg*, 2003. 148-157.
- [26] Ratnaparkhi, Adwait. "A maximum entropy model for part-of-speech tagging." *Proceedings of the conference on empirical methods in natural language processing. Vol. 1. 1996*.
- [27] "POS Tagging (State of the art)". *Wiki of the Association for Computational Linguistics*. Web. 2013.
- [28] Zhou, Keira, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. "Brill Tagging on the Micron Automata Processor." *Semantic Computing (ICSC), 2015 IEEE International Conference on. IEEE*, 2015.
- [29] "The GNU C Library: Regular Expressions." *The GNU C Library: Regular Expressions*. 2015.
- [30] Becchi, Michela, and Gerard Allwein. "Hardware Synthesis from Functional Embedded Domain-Specific Languages: A Case Study in Regular Expression Compilation." *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings. Vol. 9040. Springer*, 2015.
- [31] Yu, Xiaodong, and Michela Becchi. "Exploring different automata representations for efficient regular expression matching on GPUs." *ACM SIGPLAN Notices. Vol. 48. No. 8. ACM*, 2013.
- [32] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "Supplementary material for an efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, 2014.