A Scalable Solution for Rule-Based Part-of-Speech Tagging on Novel Hardware Accelerators

Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, Hongning Wang
Department of Computer Science, University of Virginia
{elaheh,dg7vp,chunkun,rahimi,skadron,hw5x}@virginia.edu

ABSTRACT

Part-of-speech (POS) tagging is the foundation of many natural language processing applications. Rule-based POS tagging is a well-known solution, which assigns tags to the words using a set of pre-defined rules. Many researchers favor statistical-based approaches over rule-based methods for better empirical accuracy. However, until now, the computational cost of rule-based POS tagging has made it difficult to study whether more complex rules or larger rule-sets could lead to accuracy competitive with statistical approaches. In this paper, we leverage two hardware accelerators, the Automata Processor (AP) and Field Programmable Gate Arrays (FPGA), to accelerate rule-based POS tagging by converting rules to regular expressions and exploiting the highly-parallel regular-expression-matching ability of these accelerators.

We study the relationship between rule set size and accuracy, and observe that adding more rules only poses minimal overhead on the AP and FPGA. This allows a substantial increase in the number and complexity of rules, leading to accuracy improvement. Our experiments on Treebank and Brown corpora achieve up to 2,600× and 1,914× speedups on the AP and on the FPGA respectively over rule-based methods on the CPU in the rule-matching stage, up to 58× speedup over the Perceptron POS tagger on the CPU in total testing time, and up to 253× speedup over the LSTM tagger on the GPU in total testing time, while showing a competitive accuracy compared to neural-network and statistical solutions.

CCS CONCEPTS

• **Computer systems organization → Natural language processing**; *Big data*; • **Computer Architecture** → Accelerators;

KEYWORDS

POS tagging, rule-based, automata, hardware acceleration

ACM Reference format:
Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, Hongning Wang. 2018. A Scalable Solution for Rule-Based Part-of-Speech Tagging on Novel Hardware Accelerators. In *Proceedings of The 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining, London, United Kingdom, August 19–23, 2018 (KDD '18),* 10 pages.
https://doi.org/10.1145/3219819.3219889

KDD '18, August 19–23, 2018, London, United Kingdom
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5552-0/18/08…$15.00
https://doi.org/10.1145/3219819.3219889

1 INTRODUCTION

As we are living in the era of big-data and mobile computing, effective and efficient natural language processing (NLP) applications become increasingly important, and they greatly affect the quality of human-computer interaction (HCI). The most efficient and high-quality NLP applications use extensive, time-consuming statistical or neural-network models, which make them infeasible for real-time applications.

A *part-of-speech tagger* assigns part-of-speech tags (e.g., noun, verb) to words in a sentence. POS tagging is a building block for a wide range of NLP tasks. For example, in parsing, words' parts of speech determine proper word combinations [22]; in named-entity resolution, it identifies the entities and the relationships between them [8]; and in detecting sentiment contrasts, some words could have differing sentiments in different parts of speech [27]. Moreover, in software engineering, POS tagging helps in recognizing essential words from software artifacts such as bug reports [11, 26, 28].

Generally in NLP, and specifically in POS tagging, statistical and neural network (NN)-based approaches have been favored over rule-based approaches, because they have shown higher accuracy and the training is straightforward to automate, while early rule-based tagging required manual rule generation and execution time increased linearly with the number of rules. This limits the number of rules, thus limiting accuracy. However, rules can now be learned automatically and incorporate textual information (i.e., surrounding tags and words) [6].

In this paper, using POS tagging as a case study, we show that hardware accelerators can make rule-based techniques orders of magnitude faster than statistical/ML-based taggers. This allows rule-based approaches to employ more rules and achieve accuracy competitive with statistical techniques. These observations motivate a re-evaluation of rule-based approaches in NLP.

Execution efficiency is addressed by observing that rule-based techniques map well to regular-expressions (regex), which in turn map well to "spatial" hardware that provides a reconfigurable substrate to lay out the rules in hardware. This allows a large number of patterns to be executed in parallel, in contrast to von Neumann architectures such as CPUs that must either handle one rule at a time in each core, or build large lookup tables in memory and the communication between the cores imposes a significant overhead [15]. Specifically, the Automata Processor (AP) and the Field-Programmable Gate Array (FPGA) are two spatial architectures suitable for pattern-matching. They both allow native execution of non-deterministic finite automata (NFAs), an efficient computational model for regular expressions. They achieve this with reconfigurable elements that efficiently implement automata states and matching rules, and reconfigurable routing that efficiently implements next-state activation. A single chip can implement up to tens of thousands of regular expressions, depending on rule complexity, with little or no change in throughput.

We study the relationship between the rule-set size and the accuracy of POS taggers, and observe that more complex rules (from more diverse template rule-sets) and larger rule sets lead to accuracy almost as good as statistical/ML-based techniques, especially with a larger training corpus. With hardware acceleration, this sets up a new tradeoff for designers of NLP applications. The rule-based approach can give much better testing speed, at the expense of a small drop in accuracy and longer training time.

Because the AP or FPGA can efficiently process large and complex sets of regular expressions, we propose that other NLP tasks involving rules or patterns can also be accelerated this way. For instance, in sentiment analysis, negation scope detection solutions [18][9] are typically rule-based.

In summary, our paper makes the following **contributions**.

- We study the effect of different baseline taggers and different number of tagging rules for rule-based POS tagging. We show that using the unigram (i.e, one word at a time) tagger as a baseline tagger as well as larger and more complex tagging rules results in a higher testing accuracy.
- We utilize spatial architectures (the AP and the FPGA) by transforming the rules to regular expressions. This approach is scalable in number and complexity of rules. Increasing the number of rules up to several thousand has no overhead on the AP and a minimal overhead on the FPGA, because all the rules are laid out in space across the chip and executed in parallel.
- We compare our solution on the AP and FPGA with several modern statistical approaches. Results show that we can achieve up to 2,600× and 1,914× speedups on the AP and on the FPGA respectively over CPU-based Brill tagging in the rule-matching stage, up to 58× speedup over the Perceptron POS tagger (CPU solution) in the total testing time, and up to 253× speedup over the LSTM tagger (GPU solution) in total testing time at the expense of approximately 1% accuracy in a large corpus.

An important lesson learned from this research is that rule-based POS tagging on hardware accelerators can compete with the accuracy of statistical/ML-based approaches, especially in a larger corpus. As mentioned, this sets up a very interesting tradeoff to evaluate when designing an NLP application: a small decrease in testing accuracy in exchange for vastly faster testing, at the expense of slower training. This suggests that rule-based approaches are valuable for use cases where testing performance is critical, as long as training time can be tolerated.

2 BACKGROUND

2.1 Part of Speech Tagging

Part-of-speech-tagging is the process of assigning parts of speech, such as noun, verb, etc., to each word. It has a wide range of applications in parsing, text-to-speech conversion, named entity resolution, machine translation, etc. POS tagging is generally categorized as a rule-based, statistical-based, or neural network-based model. In rule-based methods, tags are assigned based on rules that embody repeatable patterns indicating a specific tag, and in statistical methods, tags are assigned based on a probability model.

Rule-based POS tagging: The rule-based approach is the earliest POS tagging system, where a set of rules is constructed and applied to the text. The rule-based POS tagging identifies the most appropriate tag for each input token based on contextual rules learned in the training phase. A *transformation-based POS tagger* (TBT) [6] is a rule-based tagger that assigns POS tags to words based on linguistic knowledge learned from a training corpus. Then it uses the training information to tag new untagged corpora in the testing phase in two stages. In the first stage, it uses a simple statistical tagger, called a *baseline tagger* or *back-off tagger*, to assign an initial tag (usually the most frequent POS) for a word. In the second stage, the initial tags are updated based on the contextual rules (the learned rules update the tag if the baseline tag is incorrect in this context). The Brill tagger [5] is a rule-based approach that is the most widely used POS tagger for English texts. The same authors also propose an unsupervised approach [7] that assigns all possible tags to the words in the initial step, and in the next step, uses rules to reduce the number of tags to remove the ambiguity. Mohammed et al. [17] improve the original TBT-based approach and propose *guaranteed pre-tagging*, which fixes the initial tags of the words that are known to be correct. This approach works well if prior information is known.

Statistical-based POS tagging: The *Unigram Tagger* is a statistical tagger that assigns the most likely tag to the word based on the training corpus. To identify the most likely tag for each word, a unigram tagger counts the frequency of tags for each word in the training corpus. The default tag *noun* is used for unseen words. The unigram POS tagger is simple and fast, and it is usually used as a baseline tagger for rule-based approaches.

A *Hidden Markov Model* (HMM) tagger assigns POS tags by searching for the most likely tag for each word in a sentence (similar to a unigram tagger). Unlike with the unigram tagger, an HMM tagger detects a tag sequence for a sentence as a whole, instead of assigning a tag for each word independently. First-order and second-order HMM taggers are usually called Bigram and Trigram taggers, respectively. Given a sentence $w_1 ... w_n$, an HMM-based tagger chooses a tag sequence $t_1 ... t_n$ that maximizes the following joint probability:

$$P(t_1...t_n, w_1...w_n) = P(w_1...w_n | t_1...t_n)P(t_1...t_n)$$

The *TnT [4] Tagger* (also known as a trigram POS tagger) uses second-order Markov models and considers triples of consecutive words to simplify the probability computation. In TnT, the tag of a word is determined by the POS tags of the two previous words.

The *Maximum Entropy (ME) Tagger* incorporates more complex features into probabilistic models [23]. Given a sentence $w_1...w_n$, an ME-based tagger models the conditional probability of a tag sequence $t_1...t_n$ as:

$$P(t_1...t_n | w_1...w_n) \approx \prod_{i=1}^{N} P(t_i | c_i)$$

where $C_1, ...C_n$ are contexts for each word $w_1...w_n$ in the sentence. An ME-based tagger models features as binary-valued functions representing constraints to compute $P(t_i|c_i)$. It will learn the weights of the features that can maximize the entropy of the probability model using the training corpus. The Stanford POS-tagger [16] is an example of ME-based tagger.

Neural network-based POS tagging: POS tagging simply can be seen as a supervised classification problem where the input of classifier is a word (or a feature-based representation of a word) and the output is the score of belonging to each class (tag). In Avarage Perceptron tagger [12], a huge set of hand crafted features is extracted and provided for a single layer perceptron with linear activation function to classify the word based on its tag. This method selects the class with highest score as the potential tag. Deep neural network solves the potential drawback of designing handcrafted features by letting the network to pick the features by itself. Bi-directional recurrent neural networks-based taggers [21] (e.g., LSTM) perform the tag classification for the whole sentence as a single decision problem and provide the opportunity of utilizing information coming from left and right-side at the same time. However, these benefits come with the computational cost of training and testing a deep neural network.

2.2 Hardware Accelerators

The Automata Processor: The Automata Processor (AP) [10], introduced by Micron several years ago but not yet commercially available, is a non-von-Neumann, direct-hardware implementation of non-deterministic finite automata (NFAs) designed for massively parallel pattern searching and can achieve two orders of magnitude speedup over state-of-the art CPU solutions (see the overview here [31]). NFAs are an efficient computational model for regular expressions (regexes).

A regex can be represented by either deterministic finite automata (DFA) or non-deterministic finite automata (NFA). DFAs and NFAs are equivalent in computational power, but NFAs typically only need as many states as there are characters in the regular expression, while DFAs often suffer exponential blowup in state count compared to equivalent NFAs, a side effect of the rule that a DFA can only have one state active at a time, while an NFA can have many.[1] On CPUs, NFAs and DFAs are represented by tables indicating the each state's successor state(s) upon a rule match. DFAs are often the basis for implementing automata on CPUs, because they have predictable bandwidth requirement; while an NFA may require many state lookups to process a single input symbol, a DFA requires just one. On the other hand, DFA tables are often too large to fit in the processor caches. State-of-the-art automata engines combine NFA and DFA representations, using DFAs in regions where too many states are likely to be activated [1].

The AP implements NFAs directly in hardware. The state transition element (STE) implements a state and its associated transition rule; this rule can match any subset of an 8-bit symbol set (i.e., ASCII). A reconfigurable routing network connects an STE to its successors, and when an STE rule matches, these successors are activated. The current beta-hardware AP boards are built on 50nm DRAM technology, running at 133 MHz. Each chip can support approximately 48K STEs, as well as boolean and counter elements that allow certain regular-expression features to be implemented even more succinctly, and reporting buffers to hold the results of state matches (analogous to NFA accept states). Therefore, a single chip can implement a large number of finite automata or regexes, all executing in parallel. (The AP was initially to be marketed in form factors with up to 32 chips.)

Although the AP has not yet been commercialized, we can model its performance in simulation, calibrated using measurements of beta hardware to which we have access. The AP illustrates the potential efficiency of native-hardware acceleration for pattern matching. In fact, using the AP as motivation, we have found that the acceleration is so dramatic that other algorithms, never previously considered for an automata implementation, can outperform alternative algorithms by using automata acceleration like the AP; see the complete list of application on the AP in [30].

REAPR, An FPGA Implementation of NFAs: In fact, the potential of NFA acceleration can be realized using FPGAs, which are commodity hardware and are even available in Amazon's EC2 F1 cloud service. Our implementation of NFAs on FPGAs, Reconfigurable Engine for Automata PRocessing [35], is an "end-to-end", high throughput and scalable engine for generic automata processing on FPGAs. FPGAs provide a large pool of logic blocks and reconfigurable routing, and can be configured by users to implement a specific algorithm by mapping directly to these hardware resources. As mentioned before, for NFA processing, we map STEs to logic blocks and use the reconfigurable routing network for next-state activation. We use the FPGAs Block RAM to buffer reports. Therefore, similar to the AP, all automata/regexes to be matched can be laid out on the hardware, and match against the input sequence in parallel. Depending on the specific NFA, a typical FPGA can provide capacity equal to several AP chips. The FPGA also provides a higher frequency. For example, the max frequency supported by the Amazon EC2 F1 FPGA is 250MHz

The compilation results show that one AP chip can accommodate about 2050 POS tagging regexes. It means one AP board (32 chips) can accommodate more than 65,000 regexes. For now, the largest rule sets supported on EC2 F1 FPGA is 2,560, because of the limited hardware resources (specifically, the number of memory banks) on the EC2 F1 boards. One may change the configuration of REAPR or utilize a larger FPGA board to support larger rule set. We are also working on supporting mores rules for limited hardware resources, such as design fast overlay automata processing engine on FPGA or using multiple FPGA boards. These are left for future work.

3 METHODOLOGY

Rule-based part-of-speech tagging is a challenging task on CPUs when the tagging rule set becomes larger and more complex, while the AP and FPGA excel in parallel rule matching even for large numbers of rules. In this section, we show how to implement the tagging task as a parallel regex matching task on these hardware accelerators, including converting tagging rules to NFAs and encoding the input string. Then, we describe how to prepare tagging rules and input text for applying to the AP and the FPGA. Furthermore, we discuss matching results in the post-processing phase and method of tagging rules with character-level features.

[1]For example, for a regular expression such as *prefix.{100}suffix*, which matches if and only if a *prefix* is separated from *suffix* by 100 characters, would require over one million states [1].

3.1 Tagging Rules and Input Text Preparation

We create POS tagging rules based on the fnTBL [20] rule-set template. A rule template defines a dependency pattern in the tagging context without assigning specific tags or words. For example, a rule template w_{-1}, w_0 (previous tag and current word) means that the tagger will check the previous part-of-speech tag and current word to determine if the current tag needs to be corrected. Specific rules can be derived from rule templates by filling in specific part-of-speech tags and words. The fnTBL rule-set with 37 templates is shown in Table 1. For comparison, we also show the original Brill tagging rule-set with 24 templates.

Table 1: The fnTBL Template Set (37 Templates)

w_0, w_1, w_2	* w_0	* (t_1, t_2, t_3)
w_{-1}, w_0, w_1	* w_1	* (t_{-3}, t_{-2}, t_{-1})
w_0, w_1	* w_{-1}	* t_1, w_0, w_1
w_0, w_{-1}	* w_2	t_1, w_0, w_{-1}
w_0, w_2	* w_{-2}	* t_{-1}, w_{-1}, w_0
w_0, w_{-2}	* t_{-1}, t_1	t_{-1}, w_0, w_1
* (w_1, w_2)	* t_1, t_2	t_{-2}, t_{-1}
* (w_{-2}, w_{-1})	* t_{-1}, t_{-2}	t_1, t_2
(w_1, w_2, w_3)	* t_1	t_1, t_2, w_1
(w_{-3}, w_{-2}, w_{-1})	* t_{-1}	—
w_0, t_1	* t_2	** (w_{-1}, w_0)
w_0, t_{-1}	* t_{-2}	** (w_0, w_1)
w_0, t_2	* (t_1, t_2)	** w_{-1}, t_{-1}
w_0, t_{-2}	* (t_{-2}, t_{-1})	** w_1, t_1

* Original Brill templates (24 templates) in the fnTBL sets
** Original Brill templates that are not in the fnTBL sets
() If a specific tag or word is contained in this range

We use the Brill tagger to learn tagging rules, which requires tagged training data and a set of rule templates. We choose the Penn Treebank corpus and the Brown corpus as training data. During training, the Brill tagger generates specific tagging rules based on the rule templates, ranks learned rules by score and picks the top-k rules as learned results. The score of a tagging rule is defined as Equation 1, where N_{fixed} is the number of places that a rule can change an incorrect tag to a correct tag, and N_{broken} is the number of places that a rule changes a correct tag to an incorrect tag. A match will not be counted if a rule changes an incorrect tag to another incorrect tag. A higher score means the the rule can correct more tags in training data while limiting incorrect tag changes.

$$Score(rule) = N_{fixed} - N_{broken} \tag{1}$$

There are two steps for performing a typical rule-based POS tagging on testing data. The first step is to tag the input text initially with a light-weight tagger such as the unigram or bigram tagger. The second step is to correct initial POS tags using learned tagging rules. We use various baseline taggers for the first step, which can be done fast but may have low tagging accuracy. Then we extract the outputs of the first step, which contains the original text and initial part-of-speech tags, and use them as the input data for our rule matching experiments on the hardware accelerators.

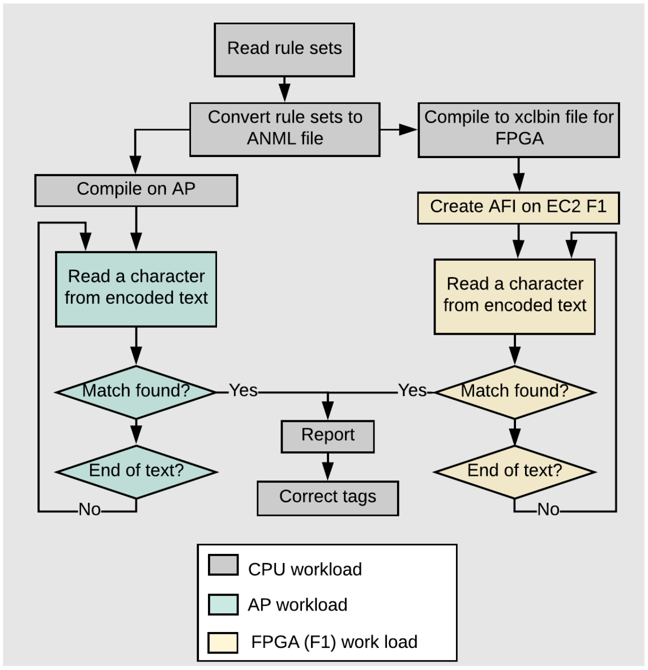

Figure 1: POS Tagging workflow on the AP and FPGA.

3.2 Accelerating Rule-Based POS Tagging on the AP and FPGA

In order to to accelerate the rule-based POS tagging using the AP and the FPGA, we implement the multi-rule tagging task as a parallel regex matching task on the hardware accelerators. Figure 1 represents the workflow of rule matching for updating the baseline tags on the AP and FPGA. We first convert all learned tagging rules to regular expressions and then convert regular expressions to ANML representation [10], which is an XML-based file format expressing finite state machines and connections, used on the AP and FPGA. For the AP, we compile these rules directly onto the hardware using Micron's compiler, and for the FPGA, we use REAPR to generate an FPGA configuration as an xclbin and use Amazon's toolchain to create an Amazon FPGA Image (AFI) for the xclbin file. If users use their own FPGAs on local nodes, they do not need to create the AFI. Then, for both the AP and the FPGA platforms, we stream in the encoded input text (e.g., encoded Treebank) to match against all rules. The matching results can tell us which tags match a learned rule and thus needs to be corrected. The hardware will report these results back and the CPU can correct the corresponding tags. For the AP and FPGA, we apply padding in both NFAs and input text to support different degrees of look ahead in tagging rules, so that we can get matching results from the AP synchronously (see below).

Tagging Rules to Regular Expressions Conversion: Table 2 shows how to convert each tagging rule to a regular expression. For rules with fixed words or tags, we directly fill in the words and tags into regex templates. For rules that check if a word or a tag is in a range, we use the regex string-OR operation to represent them. An example rule with ranges derived from template '(w_{-3}, w_{-2}, w_{-1})' is shown below. It means that, if the word '*hadn't*' is shown in last three words, we need to correct the current tag from VBD to VBN, i.e. from past-tense verb to past-participle verb.

```
VBD ⟶ VBN: if Word:hadn't@[-3,-2,-1]
```

We use string-OR regex syntax to capture the cases when such a word may appear at any places in the range. A regex for the above range rule is shown below. The string-OR regex syntax can be efficiently converted to NFAs with branches, which the AP and FPGA support.

```
/\s+(hadn't\/[ˆ\s]+\s+[ˆ\s]+\s+[ˆ\s]+|
ˆ\s+\s+hadn't\/[ˆ\s]+\s+[ˆ\s]+|
ˆ\s+\s+[ˆ\s]+\s+hadn't\/[ˆ\s]+)
\s+[ˆ\s]+/VBD\s/
```

Padding Technique: In the fnTBL template set, there are rules with 0 to 3 look-ahead steps, i.e. the tag of a word may depend on some words ahead of itself. If we directly convert these rules to NFAs and match them with the input word sequence on the hardware, we may get matching results of the same word asynchronously, i.e., for a specific word, some matching results may appear at the end of this word, while some matching results may appear after streaming in one or two or three more words. This would introduce more overhead when applying the results to update tags. For example, this could occur in a rule derived from template 'w_0, w_1, w_2', which looks two words ahead to determine whether to correct current tag.

To solve such problems, we use a padding technique. We first analyze the maximum look-ahead step in the rule template set, then pad all rules to this maximum look-ahead, so that we can delay some early matching results. The padding technique only consumes a marginal amount of hardware resources for each tagging rule, e.g., two extra NFA state elements for each look-ahead padding step. With this padding technique, the hardware accelerators can conduct regex matching in parallel for consecutive words and tag the words in a pipelined fashion. This technique can generate matching results for each word synchronously, which improves throughput and simplifies the following step. With the padding technique, the output is a vector of 0s and 1s, from which we know which tagging rules are triggered and whether the input POS tag needs to be corrected.

3.3 A Working Example

In the training phase of rule-based POS tagging, the following rule has been learned from the Treebank corpus.

```
NN ⟶ JJ  if  Word:the@[-1]  &  Word:future@[0]
              &  Word:growth@[1]
```

It means if $word[-1] == the$, and $word[0] == future$, and $word[1] == growth$, and $tag[0] == NN$, then, replace $tag[0]$ which is NN with JJ. The rule is then converted to the following regular expression (according to Table 2):

```
/\s+the\/[ˆ\s]+\s+future\/NN\s+growth\/[ˆ\s]+\s/
```

Assume that the maximum look-ahead step is 3, this regular expression is padded to three look-ahead words as follows:

```
/\s+the\/[ˆ\s]+\s+future\/NN\s+growth\/[ˆ\s]+\s/
+[ˆ\s]+\s/+[ˆ\s]+\s/
```

Figure 2 shows the generated automaton on the AP/FPGA for the padded regular expression. The automata generate a report (in the "report" state shown in the figure) when the input stream matches the padded regular expression.

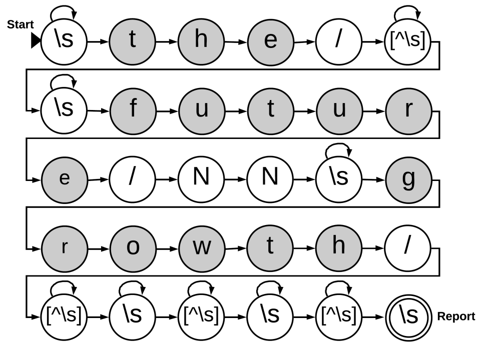

Figure 2: An example automaton for a regex rule with padding.

Such automata are stored on the AP/FPGA and the input sequence will be streamed into the hardware. The input string is the baseline-tagged word sequence with dummy word-tag pairs between sentences.[2]

Assume the sentence we intend to tag is "*the future growth of our economy*". After applying the baseline tagger, the words are initially tagged as follows:

```
the/DT future/NN growth/NN of/IN our/PRP economy/NN
```

We encoded the input string with sentence delimiters as follows:

```
the/DT future/NN growth/NN of/IN our/PRP economy/NN
./. ./. ./.
```

The regex rule mentioned above matches the input string at the space character right after '*our/PRP*', so we need to correct the tag for *future* to JJ.

The encoded text is matched against all the tagging rules in parallel. If multiple tagging rules match the input, the tag is updated using the rule with higher score.

One can find the details of our implementations here.[3]

[2]An example input sentence can be separated by space using word-tag pairs, e.g. "$word_0/tag_0$ $word_1/tag_1$ $word_2/tag_2$ ...". If the tagging rules are learned at the sentence-level, we need to use dummy word-tag pairs to separate adjacent sentences, so that the boundary words will not affect other sentences during regex matching, e.g. "$sentence_0$ ./. ./. ./. $sentence_1$ ...". The number of dummy word-tag pairs depends on the maximum look-ahead steps in the rule template set.
[3]https://github.com/elaheh-sadredini/BrillPlusPlus

Table 2: Converting Tagging Rules to Regexes

Rules	**Regex**
Regex rule templates	...word$_{-1}$\/tag$_{-1}$\s+word$_0$\/tag$_0$\s+word$_1$\/tag$_1$\s+word$_2$\/tag$_2$\s+word$_3$\/tag$_3$\s
A known word "word"	word
An unknown word	[^\s]+
A known tag "NN"	NN
An unknown tag	[^\s]+
Word tag delimiter	\/
An unknown word-tag pair	[^\s]+
A word "word" in a range 1 to 3	(word\/[^\s]+\s+[^\s]+\s+[^\s]+|[^\s]+\s+word\/[^\s]+\s+[^\s]+| [^\s]+\s+[^\s]+\s+word\/[^\s]+)

Table 3: Testing accuracy of the Brill tagger with different baseline taggers with 5-fold cross validation on the Treebank corpus and Brown news corpus.

Baseline	Treebank			Brown (News)			Entire Brown		
	Baseline	Brill (24)	fnTBL (37)	Baseline	Brill (24)	fnTBL (37)	Baseline	Brill (24)	fnTBL (37)
RU	91.37	93.76	**93.82**	87.58	91.03	**91.14**	92.60	94.36	94.55
RUB	92.26	92.60	92.65	88.59	89.55	89.71	92.69	94.45	**94.59**
RUBT	92.16	92.32	92.37	88.51	89.28	89.37	92.74	94.18	94.31

3.4 Character-Level Regex Features

Character-level regex can capture features inside a word, which can be more discriminative in POS tagging task. Some example character-level features that can be important include hyphens, uppercase letters, specific prefixes or suffixes, root words, or words with mixed digits and letters. Since the AP and FPGA excel at general regex matching, it will be interesting future work to extend the tagging rule set to include rules with character-level features without significant performance overhead.

4 EXPERIMENTAL RESULTS

4.1 Execution Environment and Data Sets

We perform experiments on a Linux server with a 3.3GHz Intel Core-i7 5820k CPU and 32GB DDR4 RAM. GPU experiments use an NVIDIA K80 in this same system. We use taggers in NLTK 3.2.1 in Python 2.7 as our baseline taggers. In addition, NLTK contains Java interface for running the Stanford log-linear tagger (3.6.0). For all measurements, I/O times are excluded, assuming data are already loaded.

We use the Penn Treebank and Brown corpora, the built-in corpora in NLTK, as our datasets. The Penn Treebank corpus contains 199 tagged documents (wsj_0001 to wsj_0199), 3,914 sentences and 100,676 words. The Brown corpus contains 500 documents, 57,340 sentences and 1,161,192 words. Some experiments are performed using just the news category of the Brown corpus, which has 44 documents, 4,623 sentences and 100,554 words.

On both the AP and FPGA, because all regexes are processed in parallel, a new input symbol can be processed every clock cycle. The kernel execution time of the AP is estimated, because fully-functional AP hardware is not yet available. But it is simple to estimate, because with the input processing rate fixed at one character per cycle at 133 MHz, throughput is 133MB/s. The kernel execution time on FPGA is evaluated on the Amazon EC2 F1 instance equipped with a Xilinx Virtex UltraScale VU9P FPGA and four memory banks. The synthesized clock rate can vary with rule complexity, with a maximum of 250 MHz. The deployment on F1 also allows other users who have access to EC2 to easily use our proposed method or reproduce the results.

4.2 Accuracy of the Brill Tagger

In order to study how baseline taggers affect the overall testing accuracy of a rule-based tagger like Brill, we evaluate the accuracy of Brill tagger using unigram tagger (U), bigram tagger (B), and trigram tagger (T) as the baseline on Treebank, Brown News, and entire Brown corpus with 5-fold cross validation on the datasets. We also test the Brill tagging testing accuracy with both the original 24 rule templates and the fnTBL 37 rule templates.

Results are shown in Table 3. For each corpus, the first column (Baseline) represents the baseline testing accuracy for the corresponding baseline tagger. The second and third columns show testing accuracy when using the corresponding baseline tagger as their back-off tagger for 24 rule-templates and fnTBL 37 rule-templates respectively. The maximum number of rules generated for Brill is 500 for this experiment.

For the unigram tagger, we use the regex tagger as its backoff tagger (denoted as RU). The regex tagger (R) can assign tags to words based on common rules, such as defining ".*able" as adjective and defining ".*ness" as noun. Since we only use 9 common rules, the accuracy of pure regex tagger is very low (23.92% on the Treebank corpus, 30.41% on the Brown news corpus, and 29.61% on the entire brown corpus). For the bigram tagger, we use the unigram tagger as the backoff of the bigram tagger (denoted as RUB). Finally, for the trigram tagger, we use the bigram tagger as the backoff of the trigram tagger (denoted as RUBT).

Results show that by choosing the unigram tagger (RU) as the baseline, the Brill tagger achieves the highest testing accuracy for

Table 4: Testing accuracy (%) and testing time (in seconds) for Brill++ on CPU, AP, and FPGA for Treebank and Brown (news) corpora while increasing the number of tagging rules.

#Rules	Treebank				Brown (News)			
	Test Acc (%)	Test Time (second)			Test Acc (%)	Test Time (second)		
		CPU	AP	FPGA		CPU	AP	FPGA
100	93.57	0.23	0.0015	0.0008	90.36	0.345	0.0015	0.0008
200	93.73	0.37	0.0015	0.0008	90.78	0.475	0.0015	0.0008
300	93.76	0.52	0.0015	0.0009	91.00	0.594	0.0015	0.0009
400	93.82	0.55	0.0015	0.0009	91.09	0.720	0.0015	0.0009

Table 5: Testing accuracy (%) and testing time (in seconds) for Brill++ on CPU, AP, and FPGA for the entire Brown corpus while increasing the number of tagging rules.

#Rules	Entire Brown Corpus				
	Test Acc(%)		Test Time (second)		
	Brill++ RU	Brill++ RUB	Brill++		
			CPU	AP	FPGA
100	93.48	94.02	2.40	0.0172	0.0093
200	93.98	94.28	3.82	0.0172	0.0093
300	94.25	94.41	5.44	0.0172	0.0098
400	94.43	94.52	6.90	0.0172	0.0098
500	94.58	94.6	8.40	0.0172	0.0098
1000	94.90	94.8	15.7	0.0172	0.0098
2000	95.17	94.91	30.05	0.0172	0.0157
3000	95.25	94.94	40.02	0.0172	NA
4000	**95.29**	94.96	44.59	0.0172	NA

Treebank and Brown (news) corpora, which is 1% more than choosing other taggers as baseline taggers. The reason for this is that Treebank corpus and Brown (news) are small, so there are many unseen words in the bigram and trigram taggers, and they may overfit the training data. Interestingly for entire Brown corpus, by using bigram tagger (RUB) as the baseline tagger, the brill tagger achieves the highest testing accuracy. This is because there are fewer unseen words in the bigram tagger model for larger datasets. Furthermore, for all corpora, the larger rule template set (fnTBL 37) helps to improve the accuracy. This shows that more diverse and complex template for rule-set is beneficial to accuracy, but processing them on the CPU is very time-consuming, and this is where having a hardware accelerators can play an important role.

4.3 Brill tagging with different number of rules

In the training phase of Brill tagging, we can set a score threshold and the number of rules to be learned. More rules usually lead to higher training/testing accuracy, although too many rules may cause overfitting. In this section, we show that a larger number of rules significantly increases computation time on the CPU and slows down the training and testing speed. However, the AP and FPGA shows a constant or near-constant processing time for testing when the number of rules increase. In this work, we focus on improving the testing time, because the learning phase is executed rarely, and the results are used many times for new texts.

Table 4 presents the testing accuracy and testing time (for rule-matching stage) of the Brill tagger when increasing the number of generated rules on Treebank and Brown news corpora. We refer to Brill as Brill++ when using our approach for increasing the number of rules. We choose the unigram tagger as the baseline tagger, and learn 100 to 400 rules from the training folds based on the fnTBL 37 rule templates (based on the results in Section 4.2, unigram tagger and fnTBL rule templates perform better). We observe that testing accuracy improves in both corpus when increasing the number of learned rules.

Table 4 also shows rule-matching time, i.e., the testing stage, for the Brill++ tagger on the CPU, AP, and FPGA for the Treebank and Brown news corpora. The testing time of the Brill tagger on the CPU is proportional to the number of rules and it increases when generating more rules. However, the computation time on the AP remains constant (0.0015 seconds) as long as the rule-set can fit on the AP board. Moreover, the computation time for the FPGA is even less than the AP, which is about 0.0009 second for both corpora. This is because all the rules configured on the AP and FPGA can be matched against the input stream (the baseline tagged word sequence) in parallel at the rate of 133MB/s for the AP and 250MB/s for the FPGA.

We perform a set of similar experiments on the entire Brown corpus, which has 500 documents and 1.16 million words. Table 5 shows the testing accuracy of Brill++ when the number of rules increases from 100 to 4,000. We run the experiments for Brill++ using both unigram tagger and bigram tagger as the back-off tagger, with baseline accuracy of 92.60% and 92.69% respectively. Accuracy is improved up to 95.21% when increasing the number of learning rule from the training folds. In Section 4.2, we observed that bigram tagger performs better as the baseline tagger for Brill on the entire Brown corpus. However, when increasing the number of rules, we see that the unigram tagger starts to perform better. This implies that more tagging rules work best with a simpler baseline tagger. Therefore, we use unigram tagger as a reliable baseline tagger for rule-based taggers, independent of the corpus size.

Table 5 also presents the rule-matching time for the Brill++ tagger on the CPU, AP, and FPGA for the full Brown corpus. The length of input string generated from testing folds of the Brown corpus is about 2.3MB and the AP frequency is 133MB, so the AP testing time is only 17ms (calculated as 2.3MB / 133MB/s). The FPGA rule-matching kernel is around 2× faster than the AP, and this is because the rule processing frequency on the FPGA is higher than the AP (about 250MB/s). However, the testing time of the Brill++ on the CPU consumes up to 44.59 seconds.

As a result, if we only compare the matching part, that is, deducting the baseline tagging time from the Brill++ tagging time, the AP and FPGA can achieve up to 2600× and 1914× speedup over the CPU-based implementation respectively.

4.4 Performance Discussion and Future Work

Errors in initial stages of an NLP pipeline have negative effects on the overall accuracy. Therefore, the main focus of many state-of-the-art POS taggers is to improve the accuracy. However, the runtime of POS taggers is very important for time-sensitive tasks (e.g., online machine translation). Therefore, in this section, we discuss the trade-off between accuracy and time for different methods.

Table 6 shows training time, testing time, and testing accuracy of rules-based, statistical-based, and neural network (NN)-based approaches on Treebank and entire Brown corpora. Testing times for Brill and Brill++ include both baseline tagger and rule-matching stages. For the Brill++ (AP-FPGA*), numbers on the parenthesis with asterisks represent testing time for the FPGA. The TnT (Trigrams'n'Tags) tagger [4] is a statistical POS tagger based on Markov models. The Stanford POS tagger [13] is also a statistical POS tagger based on a maximum-entropy model. The Stanford tagger is pre-trained on the TreeBank corpus, so we do not report the training time for that. Moreover, because the Stanford tagger is trained on Treebank, its accuracy on Brown corpus is low. The Perceptron tagger (or averaged Perceptron tagger) is a one-layer NN-based solution. TnT, Perceptron, and Stanford Taggers are all from the NLTK package[4]. LSTM[5] is a bidirectional long-short term memory tagger using conditional random field (CRF). LSTM is based on word-level features while LSTM-ChE employs character embedding features in addition to word-level features (both run on the K80 GPU). The testing time is measured using mini-batch size of 20.

The TnT tagger has the lowest accuracy and the longest testing time. However, it has the shortest training time. The Perceptron tagger has the highest accuracy among the methods that does not use character-embedding information, for both corpora. However, its testing time on the CPU is up to 58× slower than Brill++ on the AP and the FPGA. Perceptron tagger would have a better performance on GPUs and will be an interesting point of comparison for the future work. Brill++ has the second highest testing accuracy and by far the lowest testing time (on the AP and FPGA) among the taggers that does not use character-embedding features. The training time of the rule-based approach is higher than Perceptron tagger. Although the training is conducted just once and the rules are used multiple times for the unseen textual data, accelerating the training phase of Brill++ using the AP or the FPGA or other hardware accelerators is an interesting area for future work.

We ran LSTM and LSTM-ChE for 7 epochs on the Treebank corpus and 4 epochs on the Brown corpus. Results show that by adding character-embedding feature to LSTM-ChE, the accuracy can increase by 6.85%. Clearly, LSTM-ChE achieves the highest accuracy among other methods; however, testing time of Brill++ on the AP/FPGA is still 253%× less than LSTM-ChE on the GPU.

Most state-of-the-art POS taggers that report high accuracy (about 96%- 97%) take advantage of character-level features in addition to word-level features [14, 19, 25]. A recent study on machine-learning-based POS taggers [36] compares the accuracy of three state-of-the-art taggers, MarMot[6] (a generic conditional random field framework), bilstm-aux[7] (bidirectional long-short term memory tagger) and its own CNN-based tagger for three variations of input features: word only, character only, and word-character combination ([36], Table 1). The results show that combining word feature and character feature can increase the accuracy by 2%-3%.

Compared to word-level POS taggers, Brill++ achieves competitive accuracy with a superior short runtime. Based on the character-level POS tagger study, we hope that adding character-level rules will increase the accuracy of rule-based POS taggers by a similar margin, and make Brill++ fully competitive in accuracy to these statistical/ML-based approaches with superior efficiency. The AP and FPGA have plentiful capacity to extend the tagging ruleset with character-level features while maintaining good runtime.

5 RELATED WORK

Regex matching is an important computation kernel in many applications. However, the efficiency on CPUs is not satisfying due to memory bandwidth limitations. Therefore, several regular-expression processing hardware acceleration methods have been proposed. Micron's AP is an efficient semiconductor architecture for parallel automata processing [10]. It uses a non-von-Neumann reconfigurable architecture, which directly implements NFAs in hardware, to match complex regular expressions in parallel. REAPR is a reconfigurable engine for automata processing on the FPGA. It provides a flexible framework which synthesizes RTL for applications involving automata processing with high throughput [35]. iNFAnt2, an optimized version of iNFAnt, is a prototype framework for NFA-based automata processing on NVIDIA CUDA-enabled GPU cards [30], and DFAGE[8][9] is a DFA-based automata processing on GPU. However, neither GPU automata processing engine provides clear advantages over CPU, let alone AP/FPGA [30] [2]; therefore, we focus on the AP and FPGA in our paper. Both the AP and FPGA have been proved their strengths in many different applications [3, 24, 29, 32–34].

We are aware of very little work to accelerate POS tagging. A recent study [37] of Brill tagging on the AP shows 30x to 270x speedup over the CPU solution. However, [37] only uses 218 rules in Brill and only evaluates them on a small dataset (e.g. picking 5 sample files from the Brown corpus). Furthermore, they just present the speedup for the second stage of the testing phase, which is simply the rule-matching. However, in our paper, we focus on scalability with number of rules and on accuracy, generating more complicated rule template sets and creating up to 4,000 newly-learned rules using the entire Brown corpus (containing 500 sample files), and achieve a better accuracy than [37]. We also study the performance improvements for the whole testing phase (both the baseline tagger and rule-matching stages). Moreover, [37] uses a character position array to process various look-ahead steps among rules, while we propose a padding technique to synchronize the reports from the AP, which works much faster and significantly simplifies the post-processing of the match-reports.

[4]http://www.nltk.org/api/nltk.tag.html
[5]https://github.com/guillaumegenthial/sequence_tagging
[6]http://cistern.cis.lmu.de/marmot/
[7]https://github.com/bplank/bilstm-aux
[8]https://github.com/vqd8a/DFAGE
[9]https://github.com/vqd8a/iNFAnt2

Table 6: **Timing/accuracy trade-off for different methods on Treebank corpus and entire Brown corpus.**

Method	Character Embedding	Treebank			Entire Brown		
		Train Time (s)	Test Time (s)	Test Acc	Train Time (s)	Test Time (s)	Test Acc
Brill (CPU)	No	27.21	0.55	93.82	4980	45.43	95.29
Brill++ (AP-FPGA*)	No	27.21	**0.091 (0.090*)**	93.82	4980	**0.837 (0.835*)**	95.29
TnT (CPU)	No	0.46	157	89.95	3.74	15736	94.05
Stanford Tagger (CPU)	No	NA	3.39	91.30	NA	117.58	62.86
Perceptron (CPU)	No	17.01	0.82	95.89	941	48.61	96.24
LSTM (GPU)	No	210.64	1.25	89.3	1832	184.29	91.7
LSTM-ChE (GPU)	Yes	223	2.78	**96.15**	2676	212.06	**96.67**

6 CONCLUSIONS

The main objective of this paper is to motivate re-consideration of rule-based approaches when real-time computation is needed for NLP applications. To this end, we utilize two state-of-the-art accelerators, the Automata Processor and FPGA, and propose an efficient, rule-based POS tagging approach. We observe that increasing the number of rules, especially from more diverse template-sets and in a larger corpus, results in a higher accuracy that nearly matches the accuracy of statistical/ML-based approaches. Increasing the number of rules only adds minor computational overhead on the AP and FPGA, while the processing time of CPU solutions increases linearly with the number of rules. This is because both hardware accelerators can process a large number of rules against the input corpus in parallel, due to their abundant hardware resources that lay out all the rules in space for concurrent processing. The results show orders-of-magnitude speedup over CPU-based solutions, thus providing NLP application designers with a tradeoff between losing a small amount of accuracy (approximately 1%) in exchange for much faster processing.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation Grant IIS-1553568 and IIS-1718216, and by the Center for Research on Intelligent Storage and Processing in Memory (CRISP), one of six centers of JUMP, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] Michela Becchi and Patrick Crowley. 2007. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*. ACM.

[2] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 using Automata Processing across Different Platforms. In *24th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE.

[3] Chunkun Bo, Ke Wang, Jefferey Fox, and Kevin Skadron. 2016. Entity Resolution Acceleration using the Automata Processor. In *Proceedings of the IEEE International Conference on Big Data*. IEEE.

[4] Thorsten Brants. 2000. TnT: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*. Association for Computational Linguistics.

[5] Eric Brill. 1992. A simple rule-based part of speech tagger. In *Proceedings of the workshop on Speech and Natural Language*. Association for Computational Linguistics.

[6] Eric Brill. 1995. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational linguistics* 21, 4 (1995).

[7] Eric Brill and Mihai Pop. 1999. Unsupervised learning of disambiguation rules for part-of-speech tagging. In *Natural language processing using very large corpora*. Springer.

[8] Claire Cardie. 1997. Empirical methods in information extraction. *AI magazine* 18, 4 (1997).

[9] Wendy W. Chapman, Will Bridewell, Paul Hanbury, Gregory F. Cooper, and Bruce G Buchanan. 2001. A simple algorithm for identifying negated findings and diseases in discharge summaries. *Journal of biomedical informatics* 34, 5 (2001).

[10] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on* 25, 12 (2014).

[11] Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. 2013. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE.

[12] Jan Hajič, Jan Raab, Miroslav Spousta, et al. 2009. Semi-supervised training for the averaged perceptron POS tagger. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics.

[13] Dan Klein and Christopher D Manning. 2003. Fast exact inference with a factored model for natural language parsing. In *Advances in neural information processing systems*.

[14] Canasai Kruengkrai, Kiyotaka Uchimoto, Jun'ichi Kazama, Yiou Wang, Kentaro Torisawa, and Hitoshi Isahara. 2009. An error-driven word-character hybrid model for joint Chinese word segmentation and POS tagging. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics.

[15] Marzieh Lenjani and Mahmoud Reza Hashemi. 2014. Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities. *IET Computers & Digital Techniques* 8, 1 (2014), 30–48.

[16] Christopher D Manning. 2011. Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In *Computational Linguistics and Intelligent Text Processing*. Springer.

[17] Saif Mohammad and Ted Pedersen. 2003. Guaranteed pre-tagging for the brill tagger. In *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer.

[18] Pradeep G. Mutalik, Aniruddha Deshpande, and Prakash M. Nadkarni. 2001. Use of general-purpose negation detection to augment concept indexing of medical documents. *Journal of the American Medical Informatics Association* 8, 6 (2001).

[19] Tetsuji Nakagawa and Kiyotaka Uchimoto. 2007. A hybrid approach to word segmentation and pos tagging. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics.

[20] Grace Ngai and Radu Florian. 2001. Transformation-based learning in the fast lane. *arXiv preprint cs/0107020* (2001).

[21] Juan Antonio Perez-Ortiz and Mikel L Forcada. 2001. Part-of-speech tagging with recurrent neural networks. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, Vol. 3. IEEE.

[22] Ferran Pla, Antonio Molina, and Natividad Prieto. 2000. Tagging and chunking with bigrams. In *Proceedings of the 18th conference on Computational linguistics-Volume 2*. Association for Computational Linguistics.

[23] Adwait Ratnaparkhi. 1996. A maximum entropy model for part-of-speech tagging. In *Conference on Empirical Methods in Natural Language Processing*.

[24] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. 2017. Frequent subtree mining on the automata processor: challenges and opportunities. In *International Conference on Supercomputing (ICS)*. ACM.

[25] Cicero D. Santos and Bianca Zadrozny. 2014. Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*.

[26] Ramin Shokripour, John Anvik, Zarinah M. Kasirun, and Sima Zamani. 2013. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press.

[27] Vivek Kumar Singh, Mousumi Mukherjee, and Ghanshyam Kumar Mehta. 2011. Sentiment and mood analysis of weblogs using POS tagging based approach. In *International Conference on Contemporary Computing*. Springer.

[28] Yuan Tian and David Lo. 2015. A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[29] Tommy Tracy II, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards machine learning on the Automata Processor. In *International Conference on High Performance Computing*. Springer.

[30] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[31] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy II, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An overview of micron's automata processor. In *Proceedings of the 11th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 14.

[32] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential Pattern Mining with the Micron Automata Processor. In *ACM International Conference on Computing Frontiers*. ACM.

[33] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2017. Hierarchical Pattern Mining with the Micron Automata Processor. In *International Journal of Parallel Programming (IJPP)*.

[34] Ted Xie, Vinh Dang, Chunkun Bo, Jack Wadden, Mircea Stan, and Kevin Skadron. 2018. An End-to-End Reconfigurable Engine for Automata Processing. In *50th Conference on Government Microcircuit Applications and Critical Technology (GOMACTech)*.

[35] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. 2017. REAPR: Reconfigurable Engine for Automata Processing. In *The International Conference on Field-Programmable Logic and Applications (FPL)*.

[36] Xiang Yu, Agnieszka Faleńska, and Ngoc Thang Vu. 2017. A general-purpose tagger with convolutional neural networks. *arXiv preprint arXiv:1706.01723* (2017).

[37] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Brill Tagging on the Micron Automata Processor. In *2015 IEEE International Conference on Semantic Computing (ICSC)*. IEEE.