Brill Tagging using the Micron Automata Processor

Keira Zhou
Department of Systems and Information Engineering
University of Virginia
Charlottesville, VA 22904 USA
qz4aq@virginia.edu

Jeffrey J. Fox
Center for Automata Processing
University of Virginia
jjf5x@virginia.edu

Ke Wang
Department of Computer Science
University of Virginia
kewang@virginia.edu

Donald E. Brown, Fellow IEEE
Data Science Institute
University of Virginia
brown@virginia.edu

***Abstract* – Brill tagging is a classic rule-based algorithm for part-of-speech tagging within Natural Language Processing. However, implementation of the tagger is inherently slow on conventional Von Neumann architectures. In this paper, we accelerate the second stage of Brill tagging on the Micron Automata Processor, a new computing architecture that can perform massive pattern matching in parallel. The designed structure is tested with a subset of the Brown Corpus using 218 contextual rules. The results show a 38X speed-up for the second stage tagger implemented on a single AP chip, compared to a single thread implementation on CPU. This paper introduces the use of this new accelerator for computational linguistic tasks, particularly those that involve rule-based or pattern-matching approaches.**

Keywords-Part-of-speech tagging; Brill tagging; the Automata Processor; Natural Language Processing

I. INTRODUCTION

Part-of-speech (POS) Tagging makes assignments of a tag to input tokens, such as, nouns, verbs, adjectives, adverbs, etc [1]. This has an important role in Natural Language Processing (NLP) as it prepares the information needed for other tasks, for example, 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, a two-stage process is then 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 based on some contextual rules. The Brill algorithm has shown relatively high accuracy in some applications [9]. However, the computational time of the algorithm means that it is slow for both training and tagging [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].

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 Automata (NFA) in parallel to identify patterns in a data stream. The work described here shows that the AP's parallelism can significantly improve the performance of Brill Tagging compared to implementation on a single core CPU and reduce the tagging time by matching the input corpora to all the contextual rules from Brill tagging in parallel.

The next section reviews background and related work in POS tagging and Brill tagging. Session III contains a high-level description of the AP and Session IV details the design of Brill tagging using the AP. Our results comparing the execution time for a CPU versus AP implementation are in Session V. Section VI discusses accuracy and the differences between the implementations and conclusion and some future work are in Session VII.

II. BACKGROUND AND RELATED WORK

A. POS tagging

POS Tagging can be viewed as a preprocessing stage for other NLP. 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].

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 proceeds in three steps. 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.

The first stage assigns the most frequent tag to the tokens in the new corpora. In the second stage, the initial tags are 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 below.

1) NN (noun) VB (verb) PREVTAG 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 species 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 mistag the phrase *as tall as* as.

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

After training a two-stage process tags new untagged corpora. The first stage assigns the most frequent tag to the tokens in the new corpora. In the second stage, the initial tags are updated based on the rules generated from the training corpora. Our work focuses on reducing computational cost in the second stage of the tagging since 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 order RKn steps to a task of order n steps.

III. 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.

A. Major Components of the AP

There are three major components on the AP: State-Transition-Element (STE), Counter Element and Boolean Elements, among which STE is the core component.

One STE can match an 8-bit user-specified symbol in a 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 numbers. It requires 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 such as AND, OR, NOR, etc.

B. Automata Representations

We use circles to represent STEs, labels inside circles represent the user-specified symbol(s) that is(are) being matched, and an arrow-tipped circle represents a starting STE that accepts the input string and a double lined circle represents a reporting STE. Table I provides graphical illustrations of basic STE functions.

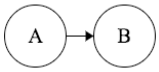
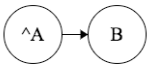
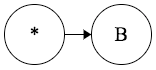
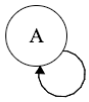
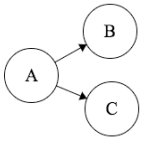
C. Programming and Execution Environment

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 match them against the design at a rate of 128 MBps.

The starting STEs can accept either the entire input data, or only the start of data. The reporting STEs will report if they are activated and matched. The output from the Emulator contains an offset number on which a reporting element is reported, as well as the ID of the reporting element for all the reported STEs.

TABLE I. GRAPHIC ILLUSTRATIONS OF BASIC STE FUNCTIONS

	A starting STE: It can either be *start-of-data,* which will only match the first symbol of the input data and matches against A; Or *all-input-start*, 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

D. 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.

There are 32 chips on an AP board which has 1,572,864 STEs that can work in parallel. This enables highly parallel operations [18]. The designs presented in this work can easily fit inside a single AP chip.

IV. DESIGN ON THE AUTOMATA PROCESSOR

This section describes the design of second stage Brill Tagging on the AP. Fig. 1 shows the major steps involved in the AP implementation of stage 2.

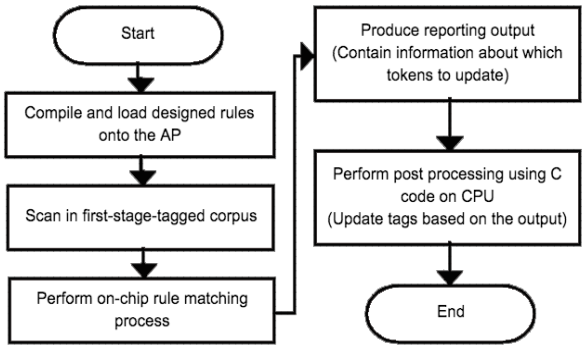


Figure 1. Major Steps of the AP implementation

A. Brill Tagging Initial Steps

Brill tagging is implemented in C and openly available [19]. This 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 input data for the example rules in Section II.

B. Design on the AP

To benefit from the parallel pattern matching ability of the AP, we need to implement our algorithm as pattern/template matching. To do this we view the 218 contextual rules as 218 templates. Among the 218 rules, there are 19 different structures. The structures and their meanings [20][21] are listed in Table III.

Fig. 2 provides example designs of automata structure for the rules mentioned in Section II.B. We use "_" to represent white space.

NN VB PREVTAG TO
IN RB WDAND2AFT as as

Fig. 2(a) has the structure of Rule 2 while Fig. 2(b) has the structure of Rule 5.

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 …
16	CURWD	The current word is …
17	LBIGRAM	The preceding word is … and the current word is …
18	RBIGRAM	The current word is … and the following word is …
19	PREV2TAG	The word two ahead is tagged as …

The reporting element ID contains a rule ID as well as the update tag. For our example here, for the first rule, the reporting element ID will look like: rule2_VB; while for the second rule, the reporting element ID will look like: rule5_RB.

All the rules are designed in the same fashion as the examples. One feature of our design is that the reporting elements will always report at the end of a tag. This will simplify the post processing from the output when updating the tags for the first-stage-tagged corpus. We explain the structure with an example input string in the following section.

… to/TO conflict/NN with/IN… as/IN tall/JJ as/IN….

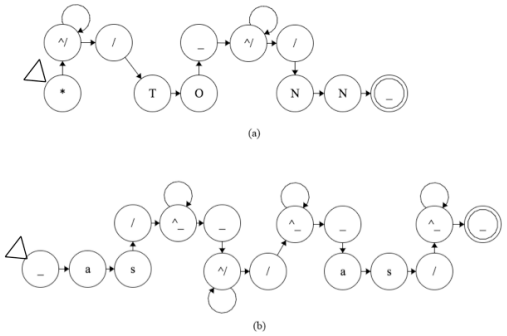


Figure 2. Example Designs of Automata Structure for the Rules

The starting STE for the first rule has a symbol "*". As explained in Section III.B, this means that the STE will be *matched* by any input symbols. At the end of every symbol cycle, it *activates* the second STE that will then accept next input symbol. The user-specified symbol on the second STE is "^/" – *NOT* "/". Thus the second STE will be *matched* on any symbols that are *NOT* a "/". It not only *activates* the third STE, it *activates* itself as well. The self-activating function will keep the second STE *activated* and accepting input symbols until a "/" appears. When that happens, the third STE will be *matched* and *activate* the forth STE. The process will keep going until all the STEs are *activated* and *matched*. Then the reporting STE will report on a white space. The second rule in Figure 2(b) works in the same manner.

This design assumes that every word in the input data is followed by a "/" and then its tag. Each word/tag pair ends with a white space.

Note that during the whole process of matching one rule, at least one of the STEs in the structure needs to be in the *activated* state. Otherwise the process will be deactivated and the pattern will need to be activated from the beginning. For example, if we have an input data:

*… to/TO conflict/NN**P** with/IN…*

The process is *matched* until the second "N", but at the "P", none of the STEs are *matched*. Thus, none of the STEs are *activated* to accept the next input symbol (in this case is a white space). So, the reporting STE will not report and the whole process is deactivated. The pattern will need to be *activated* again from the starting STE.

For our design, the starting STE is all-input-start. Since all the starting STEs can accept symbols in parallel, when the input data is scanned, all 218 rules are being matched simultaneously. It only takes one data pass to apply all 218 rules. 3073 STEs are used for 218 rules. Multiple matches for a single rule can be processed simultaneously. For example, if the input data looks like:

*… **to/TO** to/TO conflict/NN with/IN…*

One of the processes for the rule will start from the first "t" and match until the point of the second "/", then

the process will be deactivated because "T" does not match "N". At the same time, there will be another process for the rule that starts from "t" of the second "to" and match all the way till the white space after "NN" and cause a report.

C. Post processing

The output from the emulator contains offset numbers on which reporting elements reported, as well as the ID of the reporting elements.

We modify the final stage tagging code within the software package for our post processing purpose. In the original code, a word array and a tag array are created when reading in the first-stage-tagged file. In addition to that, our post processing requires another array matches each character in the file with the word to which the character belongs. Table IV show character position array for the example sentence we used previously.

… to/TO conflict/NN with/IN… as/IN tall/JJ as/IN….

After all the arrays are created, the original code then reads in the contextual rule file that contains 218 rules. It then applies one rule at a time to the entire corpus. We modified this part of the code. Instead of reading in the contextual rule file, the code now reads the output file from the AP Emulator. Both rule ID and the update tag information can be obtained from the ID of the reporting STEs. The offset number indicates at which character position an entire rule is matched. Using the offset number, we can then look-up from the character position array the index of the word that needs to be updated.

TABLE IV. THE CHARACTER POSITION ARRAY

Characters	**…**	**t**	**o**	**/**	**T**	**O**	**_**
Index of the Character Position Array	…	11	12	13	14	15	16
Array Content (Index of Word Array)	…	4	4	4	5	5	5
Characters	**c**	**o**	**n**	**f**	**l**	**i**	**c**
Index	17	18	19	20	21	22	23
Array Content	5	5	5	5	5	5	5
Characters	**t**	**/**	**N**	**N**	**_**	**w**	**i**
Index	24	25	26	27	28	29	30
Array Content	5	5	5	5	5	6	6
Characters	**t**	**h**	**/**	**I**	**N**	**_**	**…**
Index	31	32	33	34	35	36	…
Array Content	6	6	6	6	6	6	…
Characters	**a**	**s**	**/**	**I**	**N**	**_**	**t**
Index	56	57	58	59	60	61	62
Array Content	15	15	15	15	15	15	16
Characters	**a**	**l**	**l**	**/**	**J**	**J**	**_**
Index	63	64	65	66	67	68	69
Array Content	16	16	16	16	16	16	16
Characters	**a**	**s**	**/**	**I**	**N**	**_**	**…**
Index	70	71	72	73	74	75	…
Array Content	17	17	17	17	17	17	…

Using our previous examples:

… to/TO conflict/NN with/IN… as/IN tall/JJ as/IN….

The report we get from the AP looks like:

Offset 28 Reporting Element ID: rule2_VB
Offset 75 Reporting Element ID: rule5_RB

From the character position array, we find that the words associated with the characters are *6* and *17* respectively. Notice that this word index does not necessarily indicate the exact tag that needs to be updated. For our first example, we want to update the tag for *word 6*. However, for the second example, we want to update the tag for *word (17 - 2)*. This is the first "*as*" rather than *word 17* itself. This is why we need to keep the rule ID as an indicator to find which word to update for each rule.

Fig. 3 shows the steps involved for the CPU and AP implementation. The ***highlighted*** parts are the execution time we included in the comparison. For the original Brill code on the CPU, we only count the step for reading the rules, matching the rules, and updating tags. We ignore reading the corpus, creating the word/tag arrays, and writing the corpus with the updated tags because those steps are identical in both the CPU and AP implementations. For the AP implementation, we must include two different execution times. The first is an estimate of the matching time on the AP (see Fig. 3, last column). The second is the measured execution time within the modified Brill code on the CPU to: 1) create the character position array; 2) read the file that contains the reported rules and 3) update the tags.

V. TEST DATA AND RESULT

A. Test Data

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

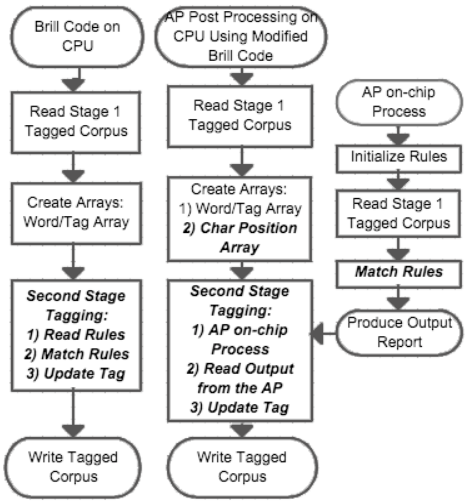


Figure 3. The CPU and AP Processes

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 selected 5 files from 5 different categories including: news, editorial, reviews, religion and hobbies. We then combined the files into 5 different sizes: 20KB, 40KB, 60KB, 79KB, 99KB to test the impact of the size of the input data on the execution time for both the CPU and AP implementations.

We also tested the largest file (99KB) with different number of rules. We used 11 subsets ranging from 20 to 200 rules from the 218 rules, as well as, the complete set to test the impact of the number of rules on the execution time for both the CPU and AP implementations.

B. *Execution Environment*

For the CPU implementation, we used the C implementation [19] and ran it on a dual core single processor (Intel Core i5) Macintosh machine with a single thread. The execution time recorded for the CPU implementation is the wall clock time for the step of updating the tags based on each individual rule within the 218 contextual rule file.

For the AP implementation the clock cycle number estimates the on-chip time. It takes one clock cycle for the AP to match one character (one symbol), which is estimated as 7.5 nanoseconds per clock cycle. The number of clock cycles equals the number of characters contained in a single file.

The post processing for the AP implementation is tested on the same Macintosh machine. The execution time included for the post processing is the wall clock time for the step of creating the extra character position array and the step of updating the tags based on the output report from the AP.

C. *Results*

1) *Execution time for different input data size*

Table V shows the execution time of the CPU and AP implementation for different sizes of input data (Time in microsecond). The speed-ups for all the corpora are within the range of 38.3X to 41.0X. This indicates that the size of the input does not have a significant impact on the speed-up, which is expected, since we would predict that both the AP and CPU implementations of rule matching scale linearly with n.

2) *Execution time for different number of rules*

Table VI shows the execution time of the CPU and AP implementation for different number of rules (Time in microsecond). We see the AP speed-up grows significantly as the number of rules grows, because the AP can match all the rules in parallel.

TABLE V. EXECUTION TIME FOR DIFFERENT SIZES OF INPUT

Time in microsecond		20 KB (19874 char)	40 KB (39721 char)	60 KB (60420 char)	79 KB (79182 char)	99 KB (98811 char)
CPU		26944	56130	86545	112289	141810
AP	On chip	19874* 7.5ns = 149	39721 * 7.5ns = 298	60420 * 7.5ns = 453	79182 * 7.5ns = 594	98811 * 7.5ns = 741
AP	Create Array	196	372	596	875	1031
AP	Post process	350	747	1063	1462	1935
AP	Total	695	1417	2112	2931	3707
Speed-up		38.8X	39.6X	41.0X	38.3X	38.3X

From Fig. 4, we can see that the execution time for the CPU implementation has an approximately linear growth with the number of rules (R), which we would expect since the number of steps in the rule matching process scales linearly with the number of rules. The execution time for the AP is more complex because the rule-matching step is independent of R. For large R, the excitation time begins to be dominated by the post-processing step.

Fig. 5 shows the growth of speed-up in relation to number of rules. We can see that there is an approximately linear growth of the speed-up with the number of rules.

TABLE VI. EXECUTION TIME FOR DIFFERENT NUMBER OF RULES

No. of Rules		20	50	75	100	110
CPU		13687	28479	48167	60328	68810
AP	On chip	98811 * 7.5ns = 741				
AP	Create Array	1031				
AP	Post processing	516	956	1332	1546	1664
AP	Total	2288	2865	3104	3318	3436
Speed-up		6.0X	13.2X	15.5X	18.2X	20.0X
No. of Rules		**130**	**150**	**180**	**200**	**218**
CPU		81187	94709	113435	130468	141810
AP	On chip	98811 * 7.5ns = 741				
AP	Create Array	1031				
AP	Post processing	1709	1775	1829	1904	1935
AP	Total	3481	3547	3601	3676	3707
Speed-up		23.3X	26.7X	31.5X	35.5X	38.3X

Figure 4. Execution Time in Relation to Number of Rules

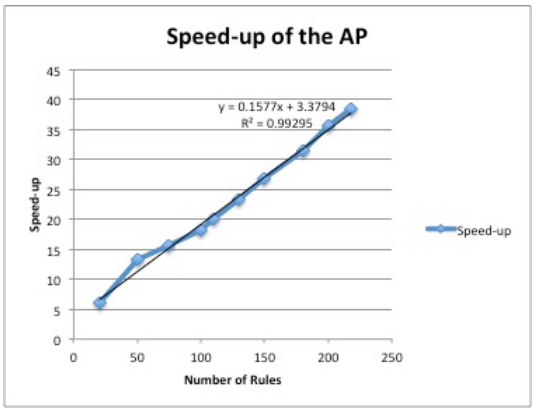


Figure 5. Speed-up in Relation to Number of Rules

1729 rules is the largest number of rules mentioned in the literature [23]. Based on the regression line (y = 0.1577x + 3.3794), we estimate that the potential speed-up for existing rules could be 276.0X.

3) Resources utilization of the AP chip

Table VII lists the number of STEs used for 4 different sizes of rules. Note that there are 49,152 STEs on one AP chip and 1,572,864 STEs on an AP board, thus the number of STEs consumed for our design is only a very small portion of AP's full capacity, leaving ample room for implementing other NLP tasks.

VI. Accuracy and Discrepancy

We discuss accuracy differences between the CPU and AP implementation of Brill tagging.

A. State-of-art POS Tagging Accuracy

The Association for Computational Linguistics (ACL) provides the accuracies of some state-of-art POS tagging systems for the Wall Street Journal corpus [27]. Table VIII lists some of these accuracies.

B. Brill Tagging Differences

The CPU implementation of Brill Tagging applies one rule at a time to the entire corpus, while the AP implementation can match all the rules in parallel. However, this speed advantage of the AP will cause differences in updating tags. The two potential cases are mentioned in the following.

TABLE VII. STE Utilization for Different Number of Rules

	50 Rules	100 Rules	150 Rules	218 Rules
Number of STEs	784	1480	2161	3073
Number of reporting STEs	50	100	150	218
Average STEs per Rule	15.68	14.80	14.41	14.10

TABLE VIII. Tagging Accuracy for Different Tagging Techniques

	Brill Tagging	Hidden Markov Model	Maximum Entropy Markov Model	Maximum Entropy Cyclic Dependency Network
All Tokens Accuracy	96.5% [26]	96.46%	96.96%	97.32%

1) Case 1:

CPU Implementation: For a given word, after rule 1 is applied and the tag is updated, rule 2 may not be triggered.

AP Implementation: Both rule 1 and rule 2 will be triggered as they are matched in parallel.

2) Case 2:

CPU Implementation: For a given word, after rule 1 is applied and the tag is updated, rule 2 then is triggered.

AP Implementation: Only rule 1 will be triggered since tags are not updated after each rule.

We tested 4 different files from the Brown corpus to estimate the discrepancies in tagging between the AP and CPU implementation.

To evaluate the accuracy, we obtained the annotated Brown Corpus which used the Brown tagset. However, the 218 rules within the C software package for the CPU implementation are trained based on the Penn Treebank tagset. We thus compared the tags to the best of our knowledge but still left some tag differences categorized as unknown. In order to set an upper bound for the decrease in accuracy for the AP implementation, we count all the unknown differences for the CPU implementation as correct and AP implementation as wrong. Table IX lists the results for different samples.

Because of the dramatic speedup of the AP implementation, we can improve the accuracy of the approach by using more rules.

TABLE IX. Tagging Discrepancies for Different Samples

	ca01 (2242 words)	cb01 (2200 words)	cc01 (2415 words)	cd01 (2213 words)
Difference (between the 2 implementation)	9	7	10	16
CPU Correct	5	2	6	6
AP Correct	2	1	1	3
Both Wrong	0	1	1	1
Unknown	2	3	2	5
Difference in Accuracy	0.223%	0.182%	0.290%	0.362%
Average	0.264%			

VII. Conclusion and Future Work

The Micron AP is a novel non-Von Neumann semiconductor architecture that can be programmed to identify thousands of patterns present in a data stream in parallel. This paper introduces an AP implementation of Brill tagging for application to NLP. For this application, the AP achieves a significant speed-up with a very small loss of accuracy.

Further work with Brill tagging is needed. For example, we plan to implement the lexical rules, rules for unknown words, and the training stage on the AP. We also will more rigorously evaluate the accuracy of the CPU and AP implementations. We plan to compare the AP implementation to other parallel implementations, including multi-core CPU and GPU. Finally, the speed and accuracy of Brill tagging can be compared to state-of-the-art POS tagging techniques.

This study suggests that the AP may be a promising platform for other NLP tasks. We plan to explore other applications of the AP within NLP domain such as parsing, semantic labeling, question answering, and machine translation.

Acknowledgment

The authors thank Mateja Putic for assisting setting up the software development tool and providing insights on various aspects of programming. In addition, the authors thank Micron Technology for providing access to the AP SDK and Matt Tanner for his insights into programming the AP. This work was partially supported by the Army Research Laboratory under grant number W911NF-10-2-0051.

References

[1] Voutilainen, Atro. "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 linguistics21.4 (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.