TEXT and rectangles in blue will NOT show on printed copy

Type final title of thesis or dissertation (M.S. and Ph.D.) below . If your title has changed since your submitted an Application for Graduate Degree, notify Graduate Office.

Brill Tagging on the Micron Automata Processor

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Master of Science

by

Name

Qi Zhou

Month degree is awarded August

Year 2015

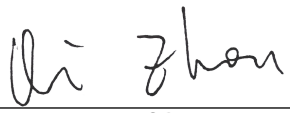
APPROVAL SHEET

The thesis

is submitted in partial fulfillment of the requirements

for the degree of

Master of Science



AUTHOR *signature*

The thesis has been read and approved by the examining committee:

Please insert committee member names below:

Donald Brown

Advisor

Peter Beling

Jeffrey Fox

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

Month degree is awarded August

Year 2015

Print Form

Abstract

There is a growing importance of Natural Language Processing (NLP) as it allows human-machine interaction, drawing insights from text documents and unstructured data, machine translation, etc. Many tasks are involved in the NLP pipeline. Part-of-speech (POS) Tagging is a task within NLP that makes assignments of a tag to input tokens, such as, nouns, verbs, adjectives, adverbs, etc. Various tagging techniques have been developed to accomplish this task. Brill tagging is a classic rule-based algorithm for POS tagging. However, traditional CPU implementation of the tagger is inherently slow. In this work, we take the advantage of different existing computer hardware as well as the Micron Automata Processor, a new computing architecture that can perform massive pattern matching in parallel, and implement the second stage of Brill tagging in a fashion of template matching. The direct implementation is tested with a subset of Brown Corpus using 218 contextual rules. The result shows a significant speed-up for the second stage tagger. To illustrate the general utility of hardware acceleration for other NLP tasks, the 218 contextual rules are then converted into Regular Expressions (Regex), which is more widely in use in various situations for NLP, and compared as single-threaded, multi-threaded versions on CPU, Xeon Phi and the AP. The result shows a promising performance improvement of using the AP as a Regex accelerator. This work serves as a guide of using different accelerators for various computational linguistic tasks, particularly those that involve rule-based or pattern-matching approaches, as well as Regex matching.

Content

I. Introduction

Natural Language Processing (NLP) has seen a growing importance in its usage. It allows human-machine interactions, drawing insights from data contained in the emails, documents and other unstructured materials, translation between languages, etc. There are many tasks in the NLP pipeline and the performance of each stage is equally important as they ensure the quality of the final results for the task is being accomplished. Improving the speed and the accuracy for different NLP tasks has attracted many research interests.

Part-of-speech (POS) Tagging is a process of making assignments of a tag to input tokens as noun, verb, adjective, adverb, etc [1]. It has an important role in Natural Language Processing (NLP) as it prepares the information needed for other tasks such as Question Answering [2], information retrieval [3], etc. 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 in use [32]. It is also called a transformation-based error-driven tagging algorithm [6]. After the tagger is trained, a two-stage tagging is then applied to new untagged corpora. The main idea is to first apply a baseline tagging method – tag each word to its most frequent tag based on training corpora, then update the tags based on some contextual rules. The algorithm provides a relatively high accurate tagging result in some applications [9]. However, it is inherently slow for both training and tagging because of its computational complexity [7]. It may require (RKn) elementary steps to tag an input of n words with R contextual rules requiring at most K tokens of context [8].

As the volume of the data increasing daily and thousands and millions of text documents generated every second, it is important to process information in a more efficient way. Speeding up various NLP tasks like Brill tagging can lead us one step closer towards this goal. Traditional single-threaded CPU programs start to fall short in the midst of this “Big

Data" era. People are now exploring multi-threaded programs such as MPI, OpenCL MapReduce programing framework, etc. In addition to the programming side of the problem, using hardware accelerators has also been an active topic such as implementing various algorithms on GPU, FPGA, etc.

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. Our work shows that AP's parallelism ability can significantly improve the efficiency 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 AP reduces the number of steps required for Brill Tagging from the original order of (RKn) to an order of (n).

Here we propose two methods of implementing Brill tagging using the AP. The first is a direct implementation while the second one is to convert Brill rules into Regex. The motivation for this work is to provide speed-up for tasks within NLP, specifically, Brill tagging in our case. As Regex has a major role in NLP, being able to convert Brill tagging into Regex and thus speed-up the Regex representation could show the promising opportunity of combing the new architecture and traditional CPU to improve computational efficiency for certain tasks that are not limited to Brill tagging.

Our goal is to implement different designs of Brill tagging on the AP and compare the computational time of the new implementations, the traditional CPU version, multicore CPU implementation as well as another high performance computing hardware – Xeon Phi. The results of this work should provide readers with some ideas of using the AP or the combination of different computer architectures to speed-up Regex related NLP tasks.

II. Background and Related Work

A. POS tagging

POS Tagging is a process of making assignments of a tag to input tokens as noun, verb, adjective, adverb, etc. It can be viewed as a preprocessing stage for other NLP tasks such as semantic parsing, question answering, information retrieval, etc. The task was initially done manually. Now these manually tagged corpora are usually used as training data for different taggers [11] [12].

Nowadays, POS tagging algorithms are commonly categorized into two groups: rule-based approaches and stochastic (statistical) approaches. The state-of-art stochastic based approaches include hidden Markov models [14], maximum entropy Markov Models [13][27], conditional random field models [7], etc.

In a hidden Markov model (HMM) POS tagger, the states of the model represent tags and outputs represent the words. The model has two assumptions: 1) current tag only depends on previous k tags; 2) each word in the sequence depends only on its corresponding tag. Transition and output probabilities are estimated from a tagged corpus. TnT tagging system is an example of a hidden Markov model [26].

Maximum entropy Markov models (MEMM) are discriminative models of the tags given the observed input word sequence. Maximum entropy modeling is more widely known as multinomial logistic regression. To use such model for POS tagging, the design of appropriate features and feature combinations is a key to success. When classifying each word, we rely on features from the current word, features from surrounding words, as well as the output of the classifier from previous words. As a comparison, HMM model includes distinct probability estimates for each transition and observation, whereas the MEMM gives one probability estimate per hidden state, which is the probability of the next tag given the previous tag and the observation [37]. Stanford POS tagger is based on MEMM model [38].

However, MEMMs and other non-generative finite-state models based on next-state classifiers, such as discriminative Markov models, share a weakness called the label bias problem: the transitions leaving a given state compete only against each other, rather than against all other transitions in the model [7]. Conditional random field models are thus proposed. The critical difference between CRFs and MEMMs is that a MEMM uses per-state exponential models for the conditional probabilities of next states given the current state, while a CRF has a single exponential model for the joint probability of the entire sequence of labels given the observation sequence. Therefore, the weights of different features at different states can be traded off against each other.

Brill tagging is one of the first and most widely used rule-based approaches and we will talk in detail in the following section. Another rule-based approach is RDRPOSTagger [25].

When performing a POS tagging task, choosing a standard tagset is important. A larger tagset provides more information about the corpora but it will be harder to tag each token accurately. On the other hand, a simple tagset is easy to tag but will leave out information about the corpora [24]. There are some commonly used tagsets available for POS tagging such as Brown Tagset (87 tags) [4], Penn Treebank Tagset (36 tags) [5].

Limited efforts have been made in accelerating POS tagging. Nurwidyantoro and Winarko [34] implemented a MapReduce version of Maximum Entropy model for Bahasa Indonesia, in which the fastest result is achieved by using 1,000,000-word corpus with 30 map processes. Kumar [33] proposed to use Globus Toolkit, a scientific grid computing middleware, to speed up POS tagging.

B. Brill Tagging

Brill tagging is one of the first and most widely used rule-based POS tagging algorithms. It is first trained on some training corpora. After the tagger is trained, the most frequent tag of a token for all tokens are recorded as well as some

contextual rules for updating the tags are generated. Then a two-stage tagging is performed on new untagged corpora.

The first stage of the tagging is to apply the baseline tagging method, in which the most frequent tag of a given token will be assigned to the new input token.

In the second stage, the initial tags are updated based on the rules generated from the training corpora. There are 218 rules trained from Brown Corpus and 284 rules generated from Wall Street Journals. Two of the rules from Brown Corpus are listed in the following. These two rules will be used as examples throughout the rest of the paper.

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*.

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

Our work focuses on the second stage of the tagging since the second stage is time-consuming on CPU because each rule exams the words and tags in a window spanning three positions before and after the focus word [17]. On the other hand, these contextual rules are in a form that can be implemented onto the AP easily. This brings the strong motivation of using AP for Brill Tagging.

C. Regular Expression[1]

Informally, regular expressions are a compact language for representing patterns in strings of characters. We are primarily concerned with PCRE regular expressions although other flavors for pattern matching exist. This is simply because most practical applications and regex engines target this as a gold standard of expressiveness.

Regular expressions were defined alongside regular languages, and in concert with Finite Automata theory (as we will discuss in the next section), and are only capable of matching, or recognizing strings in regular languages. There is a many-to-1 mapping between regular expressions and regular languages, meaning that any regular language has an infinite number of corresponding regular expressions, but every regular expression has only one corresponding regular language. Besides simple matching, PCRE Regexs add a few additional concepts that greatly expand the expressiveness of the language.

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. For example, *(gr(a|e)y)* recognizes both the American and British spelling of the color gray.

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, although this symbol is often omitted when used along with quantifiers.

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. For example, *(colou?r)* matches both the American and English spelling of the word color. The '*' symbol, also known as the "Kleene star," matches zero or any number of the previous expression. And the '+' symbol matches at least one of the previous expression. For

[1] The content of this section comes from a class report co-authored by a computer science graduate student Jack Wadden.

example, *((B|b)oo+!*)* will recognize the exclamation of any ghost. Ranged or interval quantifiers put restrictions on the number of characters. For example, if we wanted to restrict the number of o's in Boo to be between 2 and 10, we could represent this with the regular expression *((B|b)oo{1,9}!*)*.

Character Classes: Character classes represent sets of characters and are shorthand for groupings of individual characters. For example, *(gr[ae]y)* and *(gr(a|e)y)* are equivalent. Character classes can also be described as ranges of characters. For example [a-z] is all lowercase characters, while [0-9] represents any digit.

Anchors: The '^' symbol anchors an expression to the beginning of a line or input sequence. For example *(^[A-Z])* will recognize all lines that begin with an uppercase character. The '$' symbol behaves in the same way but anchors an expression to the end of a line or input sequence.

D. Finite Automata and Equivalence With Regular Expressions

Theoretical computer science has long used finite automata as a simple introduction to computation theory. Informally, finite automata are a set of states linked together by transition rules. While not turing complete (i.e. strictly less powerful than Turing machines), finite automata represent recognition of regular languages.

Deterministic Finite Automata (DFAs): Deterministic finite automata are formally defined as a 5-tuple containing a set of states, an alphabet of possible input symbols, a set of transition rules among states based on those symbols, a single start state, and a set of multiple accept states. For example, Fig. 1 shows a simple automaton to recognize all strings in the language represented by the regular expression *((AB)*)*.

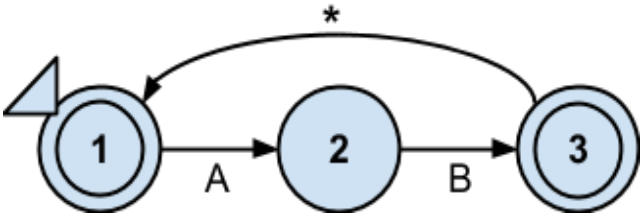


Figure 1: A DFA Accepting All Strings in the Language *((AB)*)*.

Because we were able to write a regular expression, and/or build a finite automaton to accept only strings of this form, we know that the set of strings accepted by *((AB)*)* is a regular language. Regular expressions and finite automata are not able to recognize all strings. For example, the language $(a^n b^n)$ can only be recognized by augmenting automata with some sort of memory.

Operation begins on an input stream of symbols from the defined alphabet; we begin in the "start" state, represented in Fig. 1 with an incoming arrow symbol. DFAs can only ever have one state active at any one point, therefore, it would make no sense to have multiple start states. At each step, a symbol is consumed off the input stream and a transition rule defines the next state (including the current state) to transition to. A string is recognized or accepted when execution terminates in an "accept" state, represented in Figure 1 by a double circle.

Non-Deterministic Finite Automata (NFAs): Non-deterministic finite automata (NFAs) enhance the expressiveness of DFAs by allowing multiple states to be active at any one time. Whereas DFAs were only ever able to have one transition on any one input, and a single start state, NFAs can have any number of transitions to other states (in essence "forking") on an input symbol, and can also have multiple starting states. NFAs also allow "epsilon" transitions, or transitions between states that do not consume input. In this survey, we only consider NFAs without epsilon transitions, as they can algorithmically be removed without changing the functionality of the underlying automata.

Although they may seem more powerful, NFAs are surprisingly NOT more powerful than DFAs; in fact, they are identical in power and are only capable of recognizing the regular languages. NFAs can be converted to DFAs (and vice versa) by using a straightforward "powerset construction" algorithm outlined in Sipser [35].

NFAs do allow more expressive power in that they generally are capable of representing a regular expression using

fewer states or transitions. However, NFAs must also keep track of many more active states and their potential transitions. Becchi calls this the "active set" of the NFA, which can theoretically be every single state in the NFA, but practically is usually much smaller. This difference between NFAs and DFAs (where NFAs use fewer states and transitions but generally require a larger set of transitions) manifests as a tradeoff between the size of the representation of the automata (where NFAs win handily) and the amount of bandwidth necessary to fetch transition rules and update the active set (where DFAs use constant bandwidth and NFAs bandwidth requirements). This means that, while DFAs and NFAs theoretically accept or reject strings after consuming each symbol, the increased bandwidth requirement of NFAs as the active set increases means that NFAs are often practically slower than DFAs. A more detailed summary of the pros and cons of NFAs vs. DFAs are outlined in the table below.

Table 1: The Pros and Cons of NFAs vs. DFAs

	Pros	**Cons**
DFA	- Deterministic, constant bandwidth necessary to make transition on symbol consumption - One transition per input symbol - "Single threaded", requiring only O(1) hardware for processing	- Potentially exponential number of states compared to an NFA, i.e. space requirements increases $O(2^n)$.
NFA	- Requires exponentially fewer states than DFA (can be O(n)).	- Requires O(n) parallel hardware for efficient parallel transitions from all active states - Must keep track of potentially large "active set" requiring large bandwidth to fetch state transition rules

Both NFAs and DFAs may be desirable execution models for different regular expressions, or even different parts of the same regular expression. For example, it may seem like an obvious improvement to use a hybrid finite automata (HFA) [36]. Hybrid automata might implement DFAs when the NFA active set may be very large, and the number of required states is a tractable size, while implementing NFAs when exponential state blowup of a DFA implementation is imminent. In the next section, we discuss a few different algorithms for transforming and improving automata to take advantage of the pros and cons outlined above.

E. Hardware implementations of Finite Automata

Before we explore implementations of finite automata on specialized hardware, we first consider a simple implementation on a standard single-threaded CPU. The basic algorithm for simulating any finite automata is outlined below:

1. *Initialize current active state/s to be the start state/s S (this is the active set)*
2. *For each input symbol C*
    a. *For each active state s*
        i. *Lookup rule for s based on C in database D*
        ii. *Store new state into S'*
    b. *S <- S'*

Note that there is ambiguity over how many states are "start states" and how many states are in the active set S. This reflects the key difference between DFAs and NFAs, i.e. DFAs can only ever have one state in their active state set. Therefore, we only ever need to lookup one transition rule, and the innermost for loop only ever has a single iteration. This is the heart of the often-cited performance difference between NFAs and DFAs. DFAs are (much) larger with respect to the number of states, and therefore the practical size of database D, but the CPU only ever needs to fetch a single rule per transition. This tradeoff is shown in Fig. 2 below.

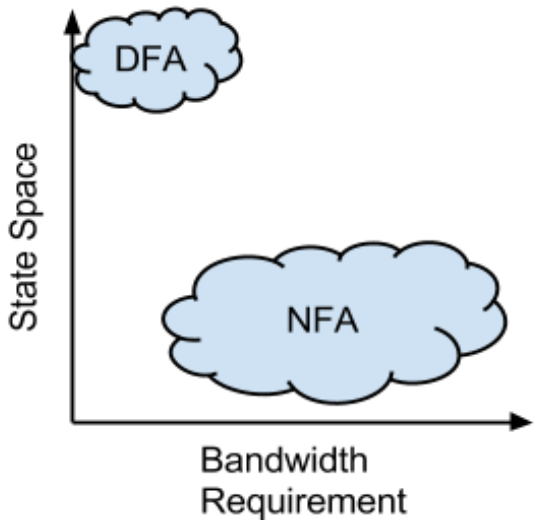


Figure 2: Generalization of tradeoffs between DFAs and NFAs. While DFAs generally require many more states than equivalent NFAs, their bandwidth requirements are small and constant. NFAs require higher bandwidth to fetch transition rules for all active states.

With this basic model in mind, we explore other architectures and implementations of finite automata in hardware. These architectures generally fall into two categories: memory-based architectures, and logic-based architectures.

Memory-based architectures store the finite automata in a traditional memory store, and fetch transition rules based on some input character and the number of states. Our naive CPU implementation can be considered a memory-based architecture.

Logic-based architectures are based on the observation that automata can be laid out in digital logic, just as we usually graphically represent automata. For example, states can be thought of as single-bit registers, and transition rules are simple wire connections ANDed with the broadcast of the correct input symbol. Such architectures generally take advantage of reconfigurable FPGA logic, and offer a complex design space for optimization.

Memory-Based Architectures operate just like the naive CPU implementation above. Transition rules are stored in some database and the correct rule is fetched and executed. Because automata are stored in memory, this scheme can offer massive density gains over logic-based designs for NFAs and DFAs. However, in the case of DFAs, the automata

may be so large that it will not fit in memory. In the case of NFAs, we then must simulate parallel execution, and provide enough memory bandwidth to fetch rules in parallel. Below we survey a few memory-based techniques in the literature.

Kaneta and Yusaku's work [39] implemented a memory-based design using the concept of bit parallelism [40]. There are three known simulation methods to run NFAs [40] - basic simulation method, dynamic programming and bit parallelism. The basic simulation method, similar to the naive CPU method described above, maintains a set S of active states during the simulation process with only state-0 active at the beginning. The input text will trigger the update of the active states based on the transition rules fetched from memory. This process continues until no further transition exists.

The bit parallelism is a bitwise implementation of the basic simulation method. It uses bit vectors to represent whether a state is active or not with 0 meaning active and 1 otherwise. This implementation can take advantage of the fact that the same bitwise operations can be performed at once in parallel over the whole bit vector. Kaneta and Yusaku's design, which takes advantage of this method, includes an input decoder, a collection of pattern matching modules, and an output encoder. This architecture targets three subclasses of Regex:

> ***Extended patterns (EXT)*:** A Regex in linear form;
> ***Network expressions (NET)*:** A Regex without the Kleene-star (A Regex formed by strings, concatenation and union)
> ***Extended network expressions (EXNET)*:** A NET expression over extends patterns (may have Kleene-stars from EXT)
> **Examples:**
> R1 = ABABC => STR (Exact String)
> R2 = ([AB]+)(B.{1,3})([BC]?)(.*)C => EXT
> R3 = A(AB|B)(B|AB)C => NET
> R4 = A(AB|B?)(B?.*|AB)C => EXNET

During the pre-processing mode, it loads the description of input patterns with packets, and during run-time mode, it

receives an input letter, makes a state transition for the target NFA using the pattern matching module, detects matches, and emits match information by receiving and sending packets.

The pattern-matching module (PMM) is a core of the architecture. The authors give the example of constructing the class EXT. it first expands the EXT Regex into an expanded form that does NOT contain bounded repeat (a{x, y}). Note that this involves a linear blowup in states according to the size of the difference between x and y. It then assigns the unique numbers – the bit-positions – to all components of the expanded form. An NFA is then obtained from this expanded Regex and encoded into five bit-masks. During the matching process, the bit-mask indicates the set of active states and transitions are triggered by input letters. However, this architecture only targets the three subclasses of Regex described above.

Lee [41] presented a bit-parallel memory-based hardware architecture implementing the Glushkov-NFA [42]. Glushkov-NFA is a particular family of automata, which are computed from regular expressions following the Glushkov algorithm. It produces an epsilon-free automata. The number of states in the automata grows linearly with the size of the corresponding regular expression [43]. This work also uses multi-striding with a stride of 4. The prototype was done using a Xilinx ML310 FPGA and achieved more than 4 Gbps throughput. However, only a few of the regular expressions from the Snort ruleset were tested, and the architecture is only suitable for regular expressions of small and moderate lengths because larger regular expressions that produce larger NFAs produce increasingly large and unmanageable bit-vectors.

Vasiliadis and Polychronakis [44] introduce a memory-based GPU implementation of DFAs. It first converts a regular expression into an NFA using the Thompson algorithm - a construction algorithm that translates a given regular expression into an equivalent NFA - and then translates the NFA into a DFA using subset construction. Each DFA is represented as a two-dimensional state transition table that is mapped on the GPU memory with rows representing

transition rules for a particular symbol and the current state indexing into the row to identify the next state. Final states are negative numbers while all other states are positive. Each state also contains 256 pointers to other states. Input data is read one Byte at a time and current state is switched according to the state transition table. When a final state is reached, a match has been found and the corresponding offset is marked. The size of the DFA state transition table is number_of_states * 1024 bytes. The implementation improves the performance of pattern matching comparing to traditional CPU implementation by parallelizing input packet analysis. Each parallel GPU thread is given a different packet to process using the underlying DFA, reporting an impressive 5-30 Gbps throughput with larger packet sizes producing increasing speedups over the CPU.

Cascarano and Rolando [45] designed one of the first approaches to pattern matching using GPUs called iNFAnt. This work shows a typical memory-based implementation of NFAs. It adopts an internal format for the finite state automata transition graph and all the transitions are stored in a list of (source, destination) tuples. The list is sorted by the triggering symbol (the input symbol that triggers the transaction) and stored in a global memory array. An ancillary data structure is used to record the first transition for each symbol. The design currently allows up to 65535 states and the maximum number of transitions only depends on global memory size. A bit-vector shows current and future (states that can be reached from the current input symbol) active states and is stored in the GPU scratchpad memory. It also adopts a special representation for self-looping states – once they are reached during a traversal and marked as active, they will never be reset. Each thread is assigned a different portion of the bit-vectors and each thread examines a different transition for the current symbol. When an input is processed, it accesses the transition table, finds the destinations and updates the bit-vector.

This work utilizes multi-striding, which requires less iteration in traversal. Multi-striding yields a larger automata but global memory is adequate. However, as Yu and Becchi [46] pointed out, this implementation suffers unpredictable

performance and poor worst-case behavior because of the amount of computation needed to fully process the input stream.

Yu and Becchi [46] thus proposed an improved implementation using GPUs by optimizing the way the state transitions are organized and laid out in memory. It clusters NFA states into groups that cannot be active at the same time. This design sharply reduces the processing performed on each input character.

Zu and Yang [47] also implemented a similar GPU approach that is improved upon iNFAnt. Their design is based on the observation that while the number of NFA transitions on an input character can be large, the number of NFA states that can be active simultaneously, the "active set", is much smaller. For the NFA of pattern set Snort36, the number of transitions to be processed for an input character can be over five times larger than the maximum number of simultaneously active states, thus most of the threads will find their obtained source state inactive. They then group the states that cannot be active at the same time and let each thread be responsible for only one of the active states. The number of threads needed for matching an input can be greatly reduced thus boosting the matching speed. They report a 10Gbps matching speed.

Logic-Based Architectures have the interesting property that transition rules do not reside in memory. Generally in this scheme, transition rules are explicitly laid out using wires between registers, and Boolean logic to compare input symbols. Therefore, they can implement parallel state transitions without going to memory. However, logic-based implementations can be less efficient in terms of area, as their size scales with the size of the automata corresponding to the target regular expression. Below we survey a few logic-based techniques in the literature.

Roan and Hwang [48] shows a good example of logic-based implementation of pattern matching architecture. It contains M modules, where M is the number of Snort rules [49] for detection. These modules are based on the shift-or algorithm for exact string matching [50]. A shift register is used to perform this shift-or operation. Each module is

responsible for matching one single rule. During the operation time, the input is scanned one symbol at a time and the symbol is then broadcasted into all the modules. Within one single module, there are (m-1) flip-flops and m OR gates where m is the size of the pattern. These flip-flops can be viewed as states in a finite automaton. Their output indicates whether all the input symbols that have been processed so far are matched or not. Their input is an OR gate which has two inputs: 1) The output from previous flip-flop; 2) A bit indicating whether the current input character matches the transition symbol. They also implemented the multi-striding method to increase the throughput.

Mitra, Najjar and Bhuyan 's work [51] shows another example of logic-based design. They implemented 214 PCRE engines on a single FPGA chip based on Snort IDS rules [49] using a two-stage translation process. The first stage generates PCRE opcodes by compiling Snort IDS rulesets using the PCRE compiler. The PCRE opcodes are then translated to VHDL hardware blocks in the second stage. The basic building block for this engine is an NFA. The basic FPGA logic elements in an FPGA are LUTs that store given data. LUTs are connected to each other via routing network. The work implements a lookup table. The address of the LUT consists of current state of the automata and the current input data; the data at that address is the next state. When reading in an input, the engine uses the information of current state and the input data to find the LUT and transit into next state. The design requires only one compilation for each rule. Re-compilation is only necessary for new and updated rules. It reports a 12.9 Gbps throughput.

Becchi and Crowley [52] also describe a logic-based FPGA implementation of NFAs using the techniques mentioned in the earlier sections – edge-compression, alphabet reduction and multi-striding. Each NFA state is represented by a flip-flop and each symbol by a bit that is set when the input character matches the symbol. The output of the flip-flop and the symbols on its outgoing transitions are AND-ed and routed toward the flip-flops encoding the target states. When one input character is processed, several parallel NFA state traversals can be triggered. The input character must first go through an alphabet-translation block because of the alphabet-reduction. They also adopt Single input

optimization and multiple outputs optimization to reduce the number of LUTs. Alphabet-reduction brings a lower LUT utilization while multi-striding leads to higher LUT and overall logic utilization. However, it brings the benefit of higher throughput. The larger the datasets, the smaller the utilization penalty and the larger the throughput improvement will be. The implementation takes the advantage of the reconfiguration capability and parallelism of FPGA. However, it has the limitation in the number of Regex deployable on a single chip.

The Automata Processor is more similar to a logic-based architecture.

III. Automata Processor

The Automata Processor is not yet available. Results from this study were obtained using Micron's SDK, which enables a researcher to design automata and simulate the on-chip process. The SDK enables researchers to obtain preliminary performance estimates of the hardware.

A. Major Components of the AP

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

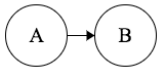
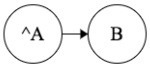
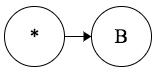
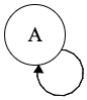
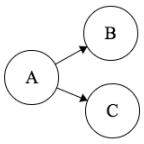
One STE can match an 8-bit user-specified symbol in a clock cycle and STEs can connect to each other via edges. Each STE has two states: *activated* and *matched*. Only *activated* STEs will be able to accept next input symbol to perform a *match* against the user-specified symbol within that STE. Once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

Besides STEs, 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 Combinatorial Elements that function as logic gates such as AND, OR, NOR, etc.

B. Automata Representation

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 whole input string and a double lined circle represents a reporting STE. Table 2 provides graphic illustrations of basic STE functions.

Table 2: 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

C. Programming and Execution Environment

User can design their Automata structures using an XML-like language called ANML, which stands for Automata Network Markup Language. 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.

Since the real chip is not available yet, we test our design using an Emulator within the SDK from Micron.

D. Hardware Resources

One single AP chip contains 49,152 STEs among which 6144 can report. One chip also 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 provides the high parallel computing ability of the Automata Processor [18].

IV. Experiment I - Direct Design Method[2]

[2] Section IV and V are based on a conference paper [54] published in 2015.

In this section, we are going to describe in detail of the software package we use, the input data for the AP implementation, our designs of Brill Tagging on the AP, the on-chip process and the post processing on CPU required for the AP design to achieve the same results as the CPU implementation. Fig. 3 shows the major steps involved in the AP implementation.

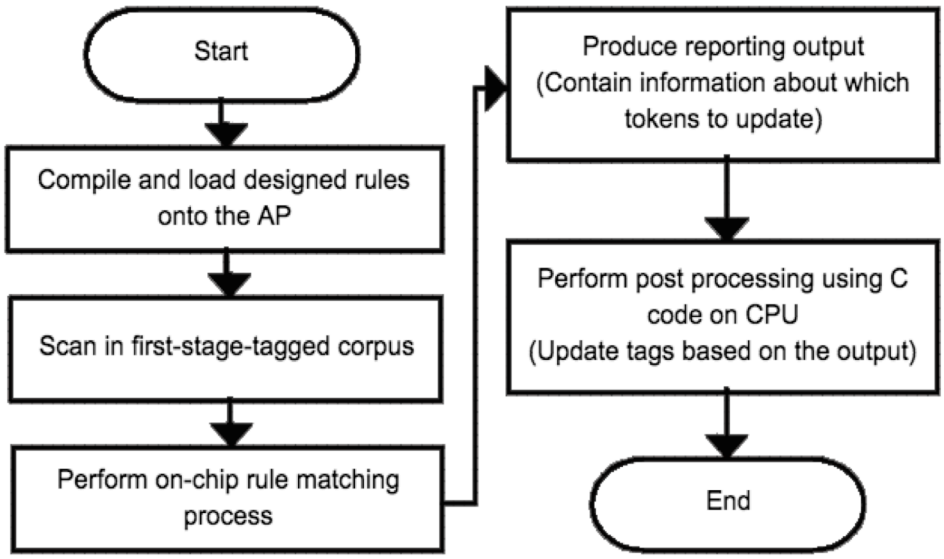


Figure 3: Major Steps of the AP implementation

A. The Software Package and the Design

The major software package we use is programed in C and downloadable from the web [19]. The software package is using Penn Treebank Tagset. Within the package, there are 218 contextual rules trained from Brown Corpus. As stated earlier, our work only focuses on the second stage tagging of Brill tagging. Also, notice that the original Brill Tagging contains Lexical rules as well as rules for tagging unknown words, but our work focuses solely on the contextual rules.

B. The Input Data

We use the C software package to run the first stage tagging and write the intermediate result into a separate file. This intermediate result will later serve as the input data for both the second stage tagging of the CPU implementation as

well as the input data for the AP implementation. Table 3 shows the sample input data.

Table 3: Sample 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 ./.

C. Design on the AP

To be able to benefit from the parallel pattern matching ability of the AP, we need to consider how to implement an algorithm in a fashion of pattern/template matching. The 218 contextual rules can be essentially viewed as 218 templates. Among the 218 rules, there are 19 different structures. The structures and their meanings [20][21] are listed in Table 4.

Table 4: 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 …

Fig. 4 provides two example designs of Automata structure for the rules mentioned in Section II.B.

NN VB PREVTAG TO

IN RB WDAND2AFT as as

We use "_" to represent white space.

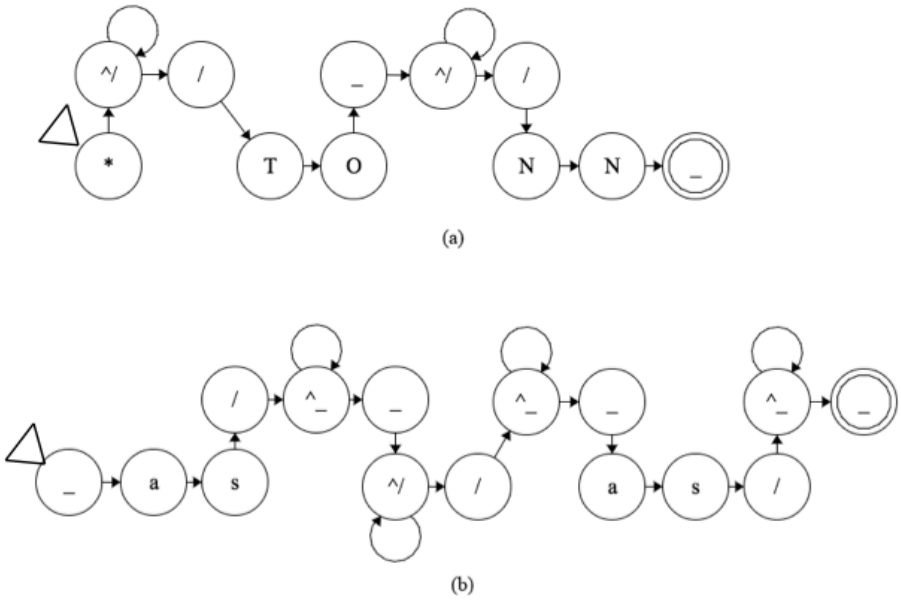


Figure 4: Example Designs of Automata Structure for the Rules

Fig. 4(a) has the structure of Rule 2 while Fig. 4(b) has the structure of Rule 5.

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 will explain the structure with an example input string in the following section.

D. The On-chip Process

Take Figure 1(a) as an example. We will explain the on-chip process for the structure with an example sentence. The sentence itself does not have meaning but it works as an illustration.

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

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 which 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 *matched* and *activated.* Then the reporting STE will report on a white space. The second rule in Figure 1(b) works in the same manner.

Our design is in accordance with the input data. 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 it will need to start from the beginning. For example, if we have an input data:

*… to/TO conflict/NN**P** with/IN…*

Although the process is *matched* until the point of the second "N", but since it is followed by a "P", none of the STE is *matched*, thus none of the STE is *activated* to accept next input symbol (in this case is a white space). Thus the reporting STE will not report and the whole process is deactivated. It will need to start from the starting STE again.

For our design, the starting STE is *all-input-start*. This means that the whole process of matching one rule will be initiated at every input symbol. There could be more than one process going on for one rule. 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.

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.

E. 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 indicating that for each character in the file, which word the character belongs. Table 5, Table 6 and Table 7 show the example sentence we used previously.

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

Table 5: The Word Array

Index of the Word Array	**…**	**4**	**5**	**6**	**…**
Array Content	…	to	conflict	with	…
Index	**…**	**15**	**16**	**17**	**…**
Array Content	…	as	tall	as	…

Table 6: The Tag Array

Index of the Word Array	**…**	**4**	**5**	**6**	**…**
Array Content	…	TO	NN	IN	…
Index	**…**	**15**	**16**	**17**	**…**
Array Content	…	IN	JJ	IN	…

Table 7: The Character Position Array

Characters	**…**	**t**	**o**	**/**	**T**	**O**	**_**
Index of the Character Position Array	…	11	12	13	14	15	16
Array Content	…	4	4	4	5	5	5

(Index of Word Array)							
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	…

After all the arrays are created, the original code then reads in the contextual rule file which 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 is now reading in the output file from the AP Emulator. Both rule ID and the update tag information can be achieved 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.

Continue from our previous examples:

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

The report we get from AP looks like:

Offset 28 Reporting Element ID: rule2_VB

Offset 75 Reporting Element ID: rule5_RB

By looking up the character position array, we will find that the words that are associated with the characters are *word 6* and *word 17* respectively. Notice that this word index does not necessarily indicate the word's 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)* (The first "*as"*) rather than *word 17* itself. This is why we need to keep the rule ID as an indicator to find the actual word to update for each different rule.

Table 8 shows the pseudo code of the CPU implementation for the second stage tagging and the AP post processing on CPU. The highlighted parts are the differences between the code as well as the execution time we compared for the two processes.

Table 8: Pseudo Code for the CPU and AP Implementation

CPU	**AP Post Processing on CPU**

Brill_Tagging_CPU {
  Create_word_array;
  Create_tag_array;
 	
  Read_in_rule_file;
  for (each_rule) {
  if (condition_met) {
  Update_tags;
  }
  }
}
	Brill_Tagging_AP {
	  Create_word_array;
	  Create_tag_array;
	  Create_char_position_array;
	
	  Read_in_AP_output_file;
	  for (each_output) {
	  Find_token_position;
	  Update_tags;
	  }
	}

V. Test Data and Result for Experiment I

We will first describe the dataset used to test the AP design. Then we will talk about the execution environment of running the CPU and AP implementation of Brill Tagging. Our results are presented at the end of this Section.

A. Test Data

The dataset we use to test our design is a subset of Brown Corpus downloaded from NLTK website [22].

Brown Corpus [4] is an American English corpus consists of 1,014,312 words. The Corpus is divided into 500 samples with over 2000 words each. Each sample begins at the beginning of a sentence. The Corpus represents a wide range of styles and varieties of prose such as political, sports, financial press, books for skills and hobbies, government documents, etc.

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 (approximately a linear growth) to test the impact of the size of the input data on the execution time for both the CPU and AP implementations.

For the largest file, 99KB, we also tested it with different number of rules. We used a subset of 50, 100 and 150 rules from the 218 rules as well as all the whole 218 rules 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 software package and run it on a dual core single processor (Intel Core i5) Macintosh machine with 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.

Since the actual chip is not available yet, the on-chip time for the AP implementation is estimated by the clock cycle number. 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 to the number of characters contained in a single file.

The post processing for the AP implementation is also 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 of direct AP design vs. original CPU version

We will talk about the results from three aspects in this section: execution time in terms of input data size, execution time in terms of number of rules and resources utilization of the AP chip.

1) Execution time for different input data size

Table 9 shows the execution time of the CPU and AP implementation for different sizes of input data. 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 data does not have a significant impact of the speed-up.

Table 9: Execution Time for Different Sizes of Input Data

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	138660
AP	**On chip**	19874 * 7.5ns = 149	39721 * 7.5ns = 298	60420 * 7.5ns = 453	79182 * 7.5ns = 594	98811 * 7.5ns = 741
	Create Array	196	372	596	875	1031
	Post processing	350	747	1063	1462	1840
	Total	695	1417	2112	2931	3612
Speed-up		38.8X	39.6X	41.0X	38.3X	38.4X

2) Execution time for different number of rules

Table 10 shows the execution time of the CPU and AP implementation for different number of rules. We see the

speed-up grow significantly as the number of rules grows. The AP implementation shows great advantage with larger number of rules. This is because the processor has the ability to matching all the rules in parallel.

Table 10: Execution Time for Different Number of Rules

99 KB (98811 char) Time in microsecond		**50 rules**	**100 rules**	**150 rules**	**218 rules**
CPU		37823	60328	94709	245345
AP	**On chip**	98811 * 7.5ns = 741			
	Create Array	1031			
	Post processing	1093	1546	1775	1840
	Total	2865	3318	3547	3612
Speed-up		13.2X	18.2X	26.7X	38.4X

The growth of the execution times depends on the content of the corpora because even if the corpora have the same number of tokens, they may still trigger different number of updates with for same rule. In addition, different rules may trigger different number of updates for the same corpora. Here we simply assume that the execution times for the CPU and AP implementation have an approximately linear growth with the number of rules in order to show the asymptotic impact of the number of rules on the speed-up. Fig. 5 provides the regression lines for the speed-up of AP vs. CPU.

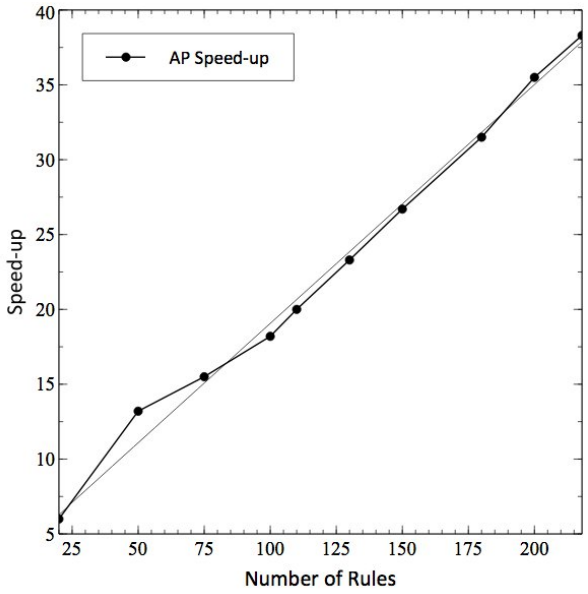


Figure 5: Execution Time in Relation to Number of Rules

1729 rules is the largest number of rules mentioned in the literature [23]. We can estimate that the speed-up could potentially be as large as 283.77X.

3) Resources utilization of the AP chip

Table 11 lists the number of STEs used for different size of rules.

Table 11: 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

Note that there are 49,152 STEs on one AP chip and 1,572,864 STEs on an AP board, the number of STEs consumed for our design is only a very small portion of AP's full capacity.

VI. Accuracy and Discrepancy for Experiment I

Although accuracy of the tagger is not under the emphasis of our work, it is still a crucial part for POS tagging. We thus provide the accuracies of some state-of-art tagging algorithms for reference as well as a discussion of the accuracy difference between the CPU and AP implementation of Brill tagging.

A. State-of-art POS Tagging Accuracy

Association for Computational Linguistics (ACL) [28] is an international scientific and professional association for people who are interested in computational linguistics. It provides the accuracies of some state-of-art POS tagging systems with regard of Wall Street Journal Corpus [29]. Table 12 lists some of these accuracies.

Table 12: Tagging Accuracy for Different Tagging Techniques

	Brill Tagging	Hidden Markov Model	Maximum Entropy Markov Model	Maximum Entropy Cyclic Dependency Network [31]
All Tokens Accuracy	96.5% [30]	96.46%	96.96%	97.32%

B. Brill Tagging Discrepancy

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.

1) Case 1:

CPU Implementation: For a given word, after rule 1 is applied, 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, rule 2 then is triggered

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

This difference has an impact on the accuracy of the AP implementation. We tested on 4 different files from the Brown corpus.

To evaluate the accuracy, we obtained annotated Brown Corpus which using Brown Tagset. However, the 218 rules within the C software package are trained based on 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 of the decreasing

in accuracy for the AP implementation, we count all the unknown differences as the CPU implementation is correct but AP implementation is wrong. Table 13 lists the discrepancies for different samples.

Table 13: Tagging Discrepancies for Different Samples

	ca01 (2242 words)	**cb01 (2200 words)**	**cc01 (2415 words)**	**cd01 (2213 words)**
Difference (between CPU and AP 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
Decreasing on Accuracy (assuming AP is wrong on the unknown ones)	0.223%	0.182%	0.290%	0.362%
Average	0.264%			

One may use more rules on the AP implementation to achieve relatively higher accuracy and decent speed-up comparing to using the CPU implementation with fewer rules. This also leaves us the future work of conducting an overall accuracy comparison for the CPU and AP implementation of the tagger.

In the core of Brill tagging is “pattern matching”, which is similar to Regular Expression (Regex) matching. This brings the thought of converting Brill rules into Regex. Since Regexes are widely in use in many areas within NLP, being able to do the conversion and thus speed-up the Regex representation could show the promising opportunity of combing the new architecture and traditional CPU to improve computational efficiency for certain tasks that are not limited to Brill tagging. With this motivation in mind, we conducted Experiment II.

VII. Experiment II - Convert Brill rules into Regular Expressions

Regular expressions (Regex) are a compact language for representing patterns in strings of characters. They were defined alongside regular languages, and in concert with Finite Automata theory, and are only capable of matching, or recognizing strings in regular languages. There is a many-to-1 mapping between regular expressions and regular languages, meaning that any regular language has an infinite number of corresponding regular expressions, but every regular expression has only one corresponding regular language.

Regex are widely in use in various machine learning or computer science disciplines. The structure of the rules for Brill tagging can be written into Regex. By converting the rules into Regex and then compare the matching speed on single-threaded and multi-threaded program on CPU, Xeon Phi and AP, we will have a better idea how these computer hardware perform for this specific task. It can also provide us with a clear idea of AP's ability of processing Regex.

In the original CPU design of Brill tagging, the first stage tagging will tag each word to its most frequent tag. Then the first-stage-tagged corpus will serve as the input document for the second stage tagging – update tags based on rules. The original code of the second stage tagging will first split the document into a word array and a tag array. The tag updating is done while the code is scanning through the input document, i.e. when the input string matches the rule condition, the corresponding tag will be updated in the tag array.

On the other hand, the direct AP design framework for second-stage tagging first does the matching, reports the position where a tag needs to be updated, and then performs updating as a post processing (i.e. finds the index where the tag needs to be updated, and then updates the tag). In order to focus on the comparison of Regex matching on different computer architectures, we will adopt the framework of the direct AP design for second-stage tagging for all hardware to simplify the experiment, i.e. we will compare the Regex matching using single-threaded and

multi-threaded CPU, Xeon Phi and AP. In this framework, updating the tags will be a post-processing step that will be common for all of the implementations, so we ignore the post-processing in this experiment and only measure the time for the Regex matching part.

A. Convert Brill Rules into Regex

There are 19 different basic structures for the rules of Brill tagging we are using. All the structures can be written into Regex. We will use the two rules mentioned in the previous sections as our example:

NN VB PREVTAG TO: / [^/]+/TO [^/]+/NN /

IN RB WDAND2AFT as as: / as/IN [^/]+/[^]+ as/[^]+ /

The Regex templates for the 19 structures are listed in Table 14. In the template, "PREVWD" looks for the previous word; "PREVTAG" looks for the previous tag; "CURRENTWD" looks for the current word; "CURRENTTAG" looks for the current tag; "NEXTWD" looks for the next word; "NEXTTAG" looks for the next tag. While converting the rules into Regex, these positions will be replaced by the corresponding words or tags.

Table 14: Regex Template for the 19 Structures of the 218 Rules

Rule ID	Rule Content	Regex Template
1	PREVWD	/ PREVWD/[^]+ [^/]+/CURRENTTAG /
2	PREVTAG	/ [^/]+/PREVTAG [^/]+/CURRENTTAG /
3	PREV1OR2TAG	/([^/]+/PREVTAG [^/]+/CURRENTTAG | [^/]+/PREVTAG [^/]+/[^]+ [^/]+/CURRENTTAG)/
4	PREV1OR2OR3TAG	/([^/]+/PREVTAG ([^/]+/[^]+){1,2}[^/]+/CURRENTTAG | [^/]+/PREVTAG [^/]+/CURRENTTAG)/
5	WDAND2AFT	/ CURRENTWD/CURRENTTAG [^/]+/[^]+ WD2AFT/[^]+ /
6	PREV1OR2WD	/(PREVWD/[^]+ [^/]+/CURRENTTAG | PREVWD/[^]+ [^/]+/[^]+ [^/]+/CURRENTTAG)/
7	NEXT1OR2TAG	/([^/]+/CURRENTTAG [^/]+/NEXTTAG [^/]+/[^]+ | [^/]+/CURRENTTAG [^/]+/[^]+ [^/]+/NEXTTAG)/
8	NEXT1OR2OR3TAG	/([^/]+/CURRENTTAG [^/]+/NEXTTAG [^/]+/[^]+ [^/]+/[^]+ | [^/]+/CURRENTTAG

		[^/]+/[^]+ [^/]+/NEXTTAG [^/]+/[^]+ | [^/]+/CURRENTTAG [^/]+/[^]+ [^/]+/[^]+ [^/]+/NEXTTAG)/
9	NEXTTAG	/ [^/]+/CURRENTTAG [^/]+/NEXTTAG /
10	NEXTWD	/ [^/]+/CURRENTTAG NEXTWD/[^]+ /
11	WDPREVTAG	/ [^/]+/PREVTAG CURRENTWD/CURRENTTAG /
12	WDNEXTTAG	/ CURRENTWD/CURRENTTAG [^/]+/NEXTTAG /
13	SURROUNDTAG	/ [^/]+/PREVTAG [^/]+/CURRENTTAG [^/]+/NEXTTAG /
14	PREVBIGRAM	/ [^/]+/PREVTAGA [^/]+/PREVTAGB [^/]+/CURRENTTAG /
15	NEXTBIGRAM	/ [^/]+/CURRENTTAG [^/]+/NEXTTAGA [^/]+/NEXTTAGB /
16	CURWD	/ CURRENTWD /CURRENTTAG /
17	LBIGRAM	/ PREVWD/[^]+ CURWD/CURRENTTAG /
18	RBIGRAM	/ CURRENTWD /CURRENTTAG NEXTWD/[^]+ /
19	PREV2TAG	/ [^/]+/PREV2TAG [^/]+/[^]+ [^/]+/CURRENTTAG /

Note that there is a many-to-1 mapping between regular expressions and regular languages, meaning that any regular language has an infinite number of corresponding regular expressions, thus each rule may be represented by various different Regex but the matching results should always be the same.

VIII. Test Data and Result for Experiment II

We will first describe the dataset used to test this experiment. Then we will talk about the different execution environments of running the implementation of Brill Tagging. Our results are presented at the end of this Section.

A. Test Data

The dataset we use to test this experiment is still a subset of Brown Corpus. It is a first-stage-tagged file and the size of the file is 2.2MB, which has 2,198,493 characters. We will use this file to run all the testings.

B. Execution Environment

We changed our experiment environment for this set of experiments from Experiment I. Instead of running on a personal local machine, we tested our designs on University of Virginia Computer Science Department' server, which has a 12-core Intel i7 processor as the host and Xeon Phi™ as the device.

Xeon Phi™ is a coprocessor computer architecture developed by Intel [53]. Those coprocessors are PCI Express form factor add-in cards that work synergistically with Intel® Xeon® processors to enable dramatic performance gains for highly parallel code—up to 1.2 double-precision teraFLOPS (floating point operations per second) per coprocessor. There are 61 cores and 244 threads on one Xeon Phi™ chip; one of the cores is the "master" node that allocates tasks across other 60 cores. We also tested our implementation on this hardware to compare the performance for different parallel computing architectures.

The program that is run on the host and the device is written in C++ using POSIX Regex library. The multi-threading function is achieved using PThread (POSIX Threads Programming). There are two different ways of running a PThread program on Xeon Phi™ chip. One is to run the program "natively" – the programmer will first compile the program on the host, then the executable will be copied onto the device and the whole program will be run on the device. Another way to run a PThread program is to add "#pragma offload" in the program for the part that needs to be run on the device. The programmer will also compile the program on the host first, then instead of copying the executable onto the device, the program will be run on the host and only the part that is marked by "#pragma offload" will be offloaded onto the device. In our case here, we choose the "native" mode to run our program. One of the advantages of running the program in the "native" mode is that we do not need to change the PThread code. Instead, we only need to add the "-mmic" compiler flag while compiling the program, thus the code that is run on the host and the device are essentially the same.

Since we've adopted the framework of the direct AP design for second-stage tagging for all hardware to simplify the experiment, i.e. we will first do the Regex matching using single-threaded and multi-threaded program on CPU, Xeon Phi and AP, and then consider updating the tags as a post processing, we can simply compare the performance of the Regex matching across all the implementations.

Table 15 shows the pseudo code for how we measured the Regex matching time of the PThread code that was run on the host Intel i7 as well as the device Xeon Phi™. It first reads in the input text, and then it compiles all the Regex in a Regex file (218 rules) and stores them in a queue. A number of threads are created based on the specification of the programmer. Each thread is responsible of fetching one compiled Regex from the queue if the queue is not empty and then performing the matching against the input text. The part to be timed is bolded. Essentially, it is the part where threads are created and a matching function is called.

Table 15: Pseudo Code for PThread

```
queue<regex_t> compiled_regex;
int NUM_OF_THREAD;
char * input_text;

input_text = read_file();

// Sequentially
for (each_regex_in_the_file) {
   t = compile_regex(current_regex);
   compiled_regex.push(t);
}

// PThread
for (thread_id < NUM_OF_THREAD) {
   pthread_create(matching_function, compiled_regex.front(), input_text);
   compiled_regex.pop();
}
```

For the AP, since we do not have the real hardware yet, we estimated the on-chip matching performance as mentioned in the previous sections. Since we could not measure the real matching time which may also include the time of transferring data, writing output, etc, the reported time may be in favor of the AP. However, the experiment can still provide some insights in terms of the matching ability of different architectures.

C. Results of the experiment

Here, we will report the results of our experiment. We will first compare the resource utilization difference of the AP chip between the direct AP implementation and the Regex implementation. Then we will report the matching performance of the Regex implementation across different hardware.

1) Resources utilization of the AP chip for direct AP implementation vs. Regex implementation

Table 16 reports the resource utilization difference of the AP chip between the direct AP implementation and the Regex implementation. From the result, we can see that the STE consumptions are pretty similar between the two implementations for both before and after optimization. The difference may be caused by the syntax of the Regex or the way those Regex are written. Note that, there are 49,152 STEs on one AP chip and 1,572,864 STEs on an AP board. The STE consumption is only a tiny part of the whole AP chip; it is capable of processing more complex rules.

Table 16: Resource Utilization Difference of the AP Chip

		STE Usage	Last States Size
Direct AP Implementation	Before optimization	3073	215
	After optimization	1514	204
Regex Implementation	Before optimization	3092	241
	After optimization	1184	201

2) Matching performance of Regex implementation for the original 218 rules across all computer architectures

Table 17 shows the matching performance of Regex implementation of Brill tagging on the host Intel i7. We can see a

performance scale as we increase the number of threads. Since the processor only has 12 cores, the performance reaches a plateau at 12 threads.

Table 17: Matching Performance of Regex Implementation on the Host Intel i7

CPU: 2.2 MB file size (2,198,493 **char); Time in microsecond**				
Thread	**1 Thread**	**2 Threads**	**4 Threads**	**6 Threads**
Runtime	4047938	2704563	1085865	743572
Thread	**8 Threads**	**10 Threads**	**12 Threads**	**14 Threads**
Runtime	655385	632916	597654	618925

Fig. 6 plots the execution time in terms of number of threads on Intel i7. We can more easily see the plateau at 12 threads.

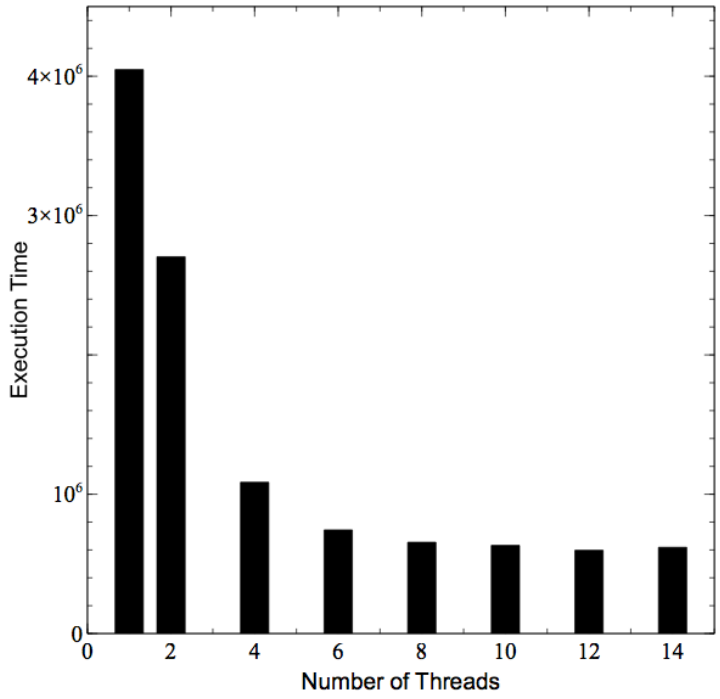


Figure 6: Execution Time in Relation to Number of Threads on Intel i7

Table 18 shows the matching performance of Regex implementation of Brill tagging on the device Xeon Phi™. We can see a performance scale as we increase the number of threads. It reaches a plateau at around 90 threads.

Table 18: Matching Performance of Regex Implementation on Xeon Phi

Xeon Phi: 2.2 MB file size (2,198,493 **char); Time in microsecond**							
Thread	**1 Thread**	**2 Threads**	**4 Threads**	**10 Threads**	**20 Threads**	**30 Threads**	**40 Threads**
Runtime	115505574	57701342	29155922	13971906	10768066	9705664	9245049
Thread	**50 Threads**	**60 Threads**	**70 Threads**	**80 Threads**	**90 Threads**	**100 Threads**	**120 Threads**
Runtime	8936398	8709339	8658992	8571378	8379498	8407550	8377593

Fig. 7 plots the execution time in terms of number of threads on the device Xeon Phi™. Fig. 7 (a) shows the zoomed-in plot from 1 thread to 70 threads; Fig. 7 (b) is the plot for all the test runs from 1 thread to 120 threads.

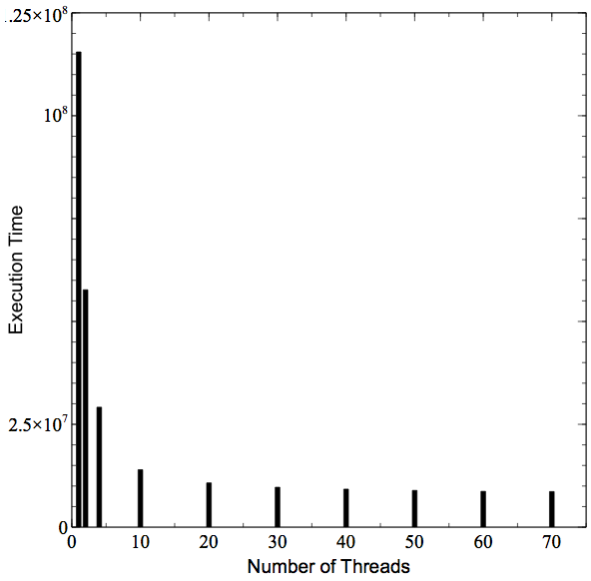


Figure 7 (a) : Threads 1 - 70

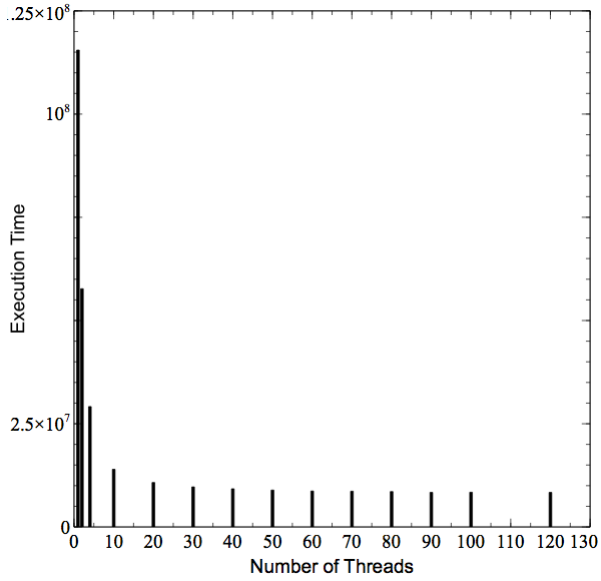


Figure 7 (b) : Threads 1 - 120

Figure 7: Execution Time in Relation to Number of Threads on the device Xeon Phi™

The above matching performance was measured on the real hardware. For the AP, we estimated the on-chip matching performance as mentioned in the previous sections. Table 19 shows the speed-up comparing the performance of the single-threaded, multi-threaded program on the host, the device, as well as the estimated AP performance. Note that since the AP hardware is not yet available, we could not measure the real matching time which may also include the time of transferring data, writing output, etc. However, the result still shows the potential of the AP chip as an

accelerator of performing pattern-matching tasks.

Table 19 (a): Matching Performance Comparison between Xeon Phi™ and Intel i7

Xeon Phi	**1 Thread**	**2 Threads**	**10 Threads**	**30 Threads**	**60 Threads**	**90 Threads**	**120 Threads**
Runtime	115505574	57701342	13971906	9705664	8709339	8379498	8377593
CPU	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**	**10 Threads**	**12 Threads**	**14 Threads**
Runtime	4047938	2704563	1085865	655385	632916	597654	618925
Speed-up	28.534	21.335	12.867	14.809	13.761	14.021	13.536

Table 19 (b): Matching Performance Comparison between Intel i7 and AP

i7	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**	**10 Threads**	**12 Threads**	**14 Threads**
Runtime	4047938	2704563	1085865	655385	632916	597654	618925
AP on-Chip	2,198,493 * 7.5ns = 16489						
Speed-up	245.493	164.022	65.854	39.747	38.384	36.246	37.536

Table 19 (a) reports the performance comparison between the host Intel i7 and the device Xeon Phi™. We can see a clear advantage of running the program on the host Intel i7. The single threaded program has an about 28X speed up in favor of Intel i7. There are 12 cores on the host and 61 cores (244 threads) on the device. Comparing the best performance of the host and the device, we can still get about 14X speed-up using Intel i7. As we described in Section VIII-B, we attempted to use the same code for both the host and the device and we ran the program natively on the device. Although we can see a scale-up of the program run on Xeon Phi™, clearly we aren't fully using the device. There are a couple of possible reasons behind this behavior of the device - the different generations of the chips may cause some of the performance difference. Another possibility is that the larger caches in Intel i7 really benefit the rule fetches for the NFAs. Thus it's better to design a program that is specific for Xeon Phi™ that can take full advantage of

the accelerator.

Table 19 (b) compares the performance of Intel i7 and the AP. For both of the experiments, we can see that the AP can achieve great performance improvement for the Regex matching kernel. For the single-threaded program on Intel i7, the AP can achieve about 245.5X speed-up and 36X speed-up comparing to the best performance on the i7. Although the performance of the AP is estimated and it may not include all the execution time to run the matching function, it still shows the promising performance gain.

3) Matching performance of Regex with different complexity

Another interesting thing to compare is to see how the complexity of the Regex impacts the performance of different hardware. In order to do this comparison, we separate Regex rules into 3 different categories – 200 simple (all the Regexes are linear), 200 original (200 Regexes from the original 218 rules), and 200 complex (all the Regexes have "alternation"). Table 20 reports the STE usage of all three categories. Table 21 shows the matching performance on Intel i7 in terms of the three categories; Table 22 shows the matching performance on Xeon Phi™. Table 23 reports the performance of Intel i7 with 12 threads and the best performance on Xeon Phi™ comparing to the AP in each category. As mentioned in the previous section, this experiment only include matching performance which may not reflect the time of transferring data, writing output, etc., but the experiment can still provide some insights in terms of the matching ability of different architectures.

Table 20: Resource Utilization Difference of the AP Chip for Three Categories

	STE Usage
200 Simple	2737
200 Original	2934
200 Complex	5843

Table 21: Matching Performance of Complex vs. Simple Regex on the Host Intel i7

Intel i7: 2.2 MB file size (2,198,493 **char); Time in microsecond**				
No. of Thread	**1 Thread**	**2 Threads**	**4 Threads**	**6 Threads**
200 Simple Rules	3420775	1770075	934856	634993
200 Original Rules	3813265	1935178	1035936	706789
200 Complex Rules	6631973	3392653	1852137	1290916
No. of Thread	**8 Threads**	**10 Threads**	**12 Threads**	**14 Threads**
200 Simple Rules	568813	534045	513631	522118
200 Original Rules	646080	605576	563720	572210
200 Complex Rules	1186821	1102516	1053676	1098875

Table 22: Matching Performance of Complex vs. Simple Regex on the Xeon Phi

Xeon Phi: 2.2 MB file size (2,198,493 **char); Time in microsecond**							
No. of Thread	**1 Thread**	**2 Threads**	**20 Threads**	**40 Threads**	**60 Threads**	**80 Threads**	**100 Threads**
200 Simple	95233305	47645001	8159165	8165023	8168897	8191992	8279226
200 Original	111519602	55634124	10748087	9238252	8682138	8553882	8367107
200 Complex	230954739	119205721	17833242	12180464	10599455	9976508	9672716
No. of Thread	**120 Threads**	**140 Threads**	**160 Threads**	**180 Threads**	**200 Threads**	**220 Threads**	-
200 Simple	8270046	8290763	8283347	8295673	-	-	-
200 Original	8429485	8435266	8459828	8469981	8534926	-	-
200 Complex	9349682	9182961	9147677	8948600	8882996	8958258	-

Table 23: Best Matching Performance Comparison between Xeon Phi™, Intel i7 and the AP on Different Rule Complexity

2.2 MB file size (2,198,493 **char); Time in microsecond**			
	200 Simple Rules (2737 STEs)	**200 Original Rules** (2934 STEs)	**200 Complex Rules** (5843 STEs)
Xeon Phi	8159165 **(20 Threads)**	8367107 **(100 Threads)**	8882996 **(200 Threads)**
Intel i7 (12 Threads)	513631	563720	1053676
AP	16489		
AP Speed-up over Intel i7	31.15X	34.19X	63.90X

Although the program is not fully utilizing the true capacity of Xeon Phi™, the result still shows that the execution time on both Intel i7 and the phi increase as the complexity of Regex increases. 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 the more complex the Regexes are, the more STEs will be consumed, as long as all the Regexes can fit onto one AP chip, the execution time will stay the same if the length of the input string does not change. Again, there are 49,152 STEs on one AP chip and 1,572,864 STEs on an AP board. The STE consumption of 218 Brill rules is only about 6% of one AP chip. Combining with the result from Experiment I, we can see that the AP is capable of processing more complex rules and larger rule set. The more complex the rules are and the larger the rule set is, the bigger the improvement the AP can gain over CPU.

IX. Conclusion and Future Work

The Micron Automata Processor is a novel non-Von Neumann semiconductor architecture which can be programmed to identify thousands of patterns present in a data stream in parallel. Xeon Phi™ is a coprocessor computer architecture developed by Intel which enables dramatic performance gains for highly parallel code.

In this work, we investigated how to implement an algorithm, Brill tagging, for an application in Natural Language Processing, part-of-speech tagging, directly on the AP, as well as how to convert the Brill rules into Regex, and then compare the Regex matching performance across CPU, Xeon Phi™ and the AP.

This work also analyzed the Regex matching performance differences across multiple computer hardware. In theory, the AP can achieve a significant speed-up for both the direct implementation as well as Regex matching function. It also shows a great potential of using the AP for other NLP tasks with the same nature.

To continue the work with Brill tagging, we can implement the lexical rules, rules for unknown words or the training stage on the AP. We will also need to evaluate the accuracy of the CPU and AP implementations of the tagger. It will also worth the time to try another Regex matching library other than POSIX for the C++ implementation. Different libraries may bring very different matching speed.

Another extension of this work is to compare the Regex design on the AP, multi-threaded implementation on CPU against GPU and FPGA as well as design a program that is specific for Xeon Phi™. As described in Section II, Regex can be implemented in various ways on GPU and FPGA (ie. Memory-based or logic-based). It would be an interesting comparison to see how implementing Regex as Memory-based or logic-based DFAs or NFAs perform on GPU or FPGA.

Finally, it might be interesting to explore hardware accelerators for other applications within Natural Language Processing domain that are not just rule-based, such as various parsing, semantic labeling algorithms, question answering, machine translation, etc. For example, FPGAs and GPUs may be useful to accelerate statistic based NLP tasks.

Reference

[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] Nguyen, Dat Quoc, et al. "Ripple down rules for part-of-speech tagging."Computational Linguistics and Intelligent Text Processing. Springer Berlin Heidelberg, 2011. 190-201.

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

[27] Denis, Pascal, and Benoît Sagot. "Coupling an Annotated Corpus and a Morphosyntactic Lexicon for State-of-the-Art POS Tagging with Less Human Effort." PACLIC. 2009.

[28] n.p. Association for Computational Linguistics. Web. n.d.

[29] "POS Tagging (State of the art)". *Wiki of the Association for Computational Linguistics*. Web. 2013.

[30] 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.

[31] Toutanova, Kristina, et al. "Feature-rich part-of-speech tagging with a cyclic dependency network." Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1. Association for Computational Linguistics, 2003.

[32] Mohammad, Saif, and Ted Pedersen. "Guaranteed pre-tagging for the brill tagger." Computational Linguistics and Intelligent Text Processing. Springer Berlin Heidelberg, 2003. 148-157.

[33] Kumar, Naveen, Saumesh Kumar, and Padam Kumar. "Parallel Implementation of Part of Speech Tagging for Text Mining Using Grid Computing." Advances in Computing and Communications. Springer Berlin Heidelberg, 2011. 461-470.

[34] Nurwidyantoro, Arif, and Edi Winarko. "Parallelization of Maximum Entropy POS Tagging for Bahasa Indonesia with MapReduce." arXiv preprint arXiv:1208.3047 (2012).

[35] Michael Sipser. 1996. Introduction to the Theory of Computation (1st ed.). International Thomson Publishing.

[36] Michela Becchi, “Data structures, algorithms and architectures for efficient regular expression evaluation,” Ph.D. dissertation, Washington University, 2009. Department of Computer Science and Engineering., 2009.

[37] Martin, James H., and D. Jurafsky. "Speech and language processing."International Edition (2000).

[38] Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In Proceedings of HLT-NAACL 2003, pp. 252-259.

[39] Kaneta, Yusaku, et al. "‘High-Speed String and Regular Expression Matching on FPGA." Asia-Pacific Signal Information Processing Association Annu. Summit Conf., Xi’an, China. 2011.

http://www.apsipa.org/proceedings_2011/pdf/APSIPA268.pdf

[40] Holub, Jan. "Bit parallelism-NFA simulation." Implementation and Application of Automata. Springer Berlin Heidelberg, 2002. 149-160.

[41] Lee, Tsern-Huei. "Hardware architecture for high-performance regular expression matching." Computers, IEEE Transactions on 58.7 (2009): 984-993.

[42] Glushkov, Victor Michailowitsch. "The abstract theory of automata." Russian Mathematical Surveys 16.5 (1961): 1-53.

[43] Caron, Pascal, and Djelloul Ziadi. "Characterization of Glushkov automata."Theoretical Computer Science 233.1 (2000): 75-90.

[44] Vasiliadis, Giorgos, Michalis Polychronakis, and Sotiris Ioannidis. "Parallelization and characterization of pattern matching using GPUs."Workload Characterization (IISWC), 2011 IEEE International Symposium on. IEEE, 2011.

[45] Cascarano, Niccolo, et al. "iNFAnt: NFA pattern matching on GPGPU devices."ACM SIGCOMM Computer Communication Review 40.5 (2010): 20-26.

[46] 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.

[47] Zu, Yuan, et al. "GPU-based NFA implementation for memory efficient high speed regular expression matching." ACM SIGPLAN Notices. Vol. 47. No. 8. ACM, 2012.

[48] Roan, Huang-Chun, Wen-Jyi Hwang, and Cnia-Tien Dan Lo. "Shift-or circuit for efficient network intrusion detection pattern matching." Field Programmable Logic and Applications, 2006. FPL'06. International Conference on. IEEE, 2006.

[49] Snort Open Source Intrusion Detection System. https://www.snort.org/.

[50] Baeza-Yates, Ricardo, and Gaston H. Gonnet. "A new approach to text searching." Communications of the ACM 35.10 (1992): 74-82.

[51] Mitra, Abhishek, Walid Najjar, and Laxmi Bhuyan. "Compiling pcre to fpga for accelerating snort ids." Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems. ACM, 2007.

[52] Becchi, Michela, and Patrick Crowley. "Efficient regular expression evaluation: theory to practice." Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. ACM, 2008.

[53] Intel. “Intel® Xeon Phi™ Coprocessors – Advanced performance for highly parallel workloads”.

http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-overview.html

[54] Zhou, Keira, et al. "Brill Tagging on the Micron Automata Processor." Semantic Computing (ICSC), 2015 IEEE International Conference on. IEEE, 2015.