A Bit-Parallel Algorithm for Searching Multiple Patterns with Various Lengths

Ko Kusudo[a,b], Fumihiko INO[a,*], Kenichi HAGIHARA[a]

[a] *Graduate School of Information Science and Technology, Osaka University, 1-5 Yamada-oka, Suita, Osaka 565-0871, Japan*
[b] *Fujitsu Limited, 1-5-2 Higashi-Shimbashi, Minato-ku, Tokyo 105-7123, Japan*

Abstract

In this paper, we present an Advanced Vector Extensions (AVX) accelerated method for a bit-parallel algorithm that realizes fast string search for maximizing stable search throughput. An advantage of our method is that it accelerates string search by regularizing both control flow and data structures. This regularization facilitates the exploitation of the latest vector instruction set to achieve efficient parallel search of multiple patterns of different lengths. We use AVX instructions to increase search throughput per CPU core and employ OpenMP directives to realize data-parallel search of strings. As a result, we found that our data structure doubled search throughput as compared with a previous bit-parallel approach that used a data structure for patterns of the same length. We also found that our method achieved stable search throughput for arbitrary data if the pattern size is large, but small enough to fit into a word. Some experimental results are provided to understand the advantage and disadvantage of our method with a comparison

*Corresponding author. Tel.: +81 6 6879 4353; fax: +81 6 6879 4354.
Email address: ino@ist.osaka-u.ac.jp (Fumihiko INO)

to Aho-Corasick based methods. We believe that our method is useful for large genome texts with many partial matches.
Keywords: string search, bit-parallel algorithm, acceleration, AVX

1. Introduction

String search identifies the location in a large string or *text* at which one or more substring *patterns* appear. Numerous research areas require the acceleration of multipattern search to handle large volumes of real-time data [1, 2]. For example, network intrusion detection systems monitor packets flowing on the network to protect computer systems from malicious access [3, 4]. String search is also useful for locating specific amino acid sequences in biological databases [5, 6, 7]. Given performance requirements of such applications, a sophisticated efficient parallel implementation is required to achieve string search acceleration. However, scaling search throughput with the number of processing elements is not easy, because string search can be regarded as an irregular problem that suffers from burdensome issues such as irregular control flow and unpredictable data access. In particular, solving these issues is essential to efficiently parallelize string search on a single instruction, multiple data (SIMD) parallel machine, because different control paths can result in SIMD serialization, which drops the efficiency of parallel execution. Therefore, accelerating string search is a challenging problem for the parallel processing community.

Many researchers [3, 8, 9] tried to parallelize the Aho-Corasick (AC) algorithm [10], which simultaneously searches multiple patterns to reuse loaded sym-

bols between multiple patterns. Because string search is a memory-intensive problem rather than a computationally intensive problem, this data reuse approach is useful to achieve acceleration. The AC algorithm represents multiple patterns as a trie data structure, which is regarded as a deterministic finite automaton (DFA) that detects a match by loading symbols from the beginning of the text. If there is no valid state transition for the input symbol, then the automaton detects a transition failure and performs backtracking to investigate the possibility of a match from other locations. The time complexity of the AC algorithm is given by $\mathcal{O}(n + \hat{m} + z)$, where n is the length of the text, $\hat{m}$ is the sum of the lengths of the patterns, and z is the total number of occurrences of the patterns [10]. The AC algorithm can be easily implemented on a multiple instruction, multiple data (MIMD) parallel machine by exploiting the data parallelism inherent to string search. This data-parallel AC (DPAC) algorithm divides the text into chunks, allowing threads to process them in parallel. Thus, MIMD machines are more tolerant to the irregularity than SIMD machines, because the former machines can issue different instructions to different processing elements. The DPAC algorithm was deployed on many parallel machines such as the graphics processing unit (GPU) [4, 5, 8, 9, 11, 12, 13], field programmable gate array (FPGA) [14, 15], Cell Broadband Engine [16], and supercomputer [17]. Similar to the AC algorithm, which has the cost $\mathcal{O}(z)$ of printing the output, the DPAC algorithm has different time complexities for best and worst cases. Furthermore, owing to irregular control flow that affects branch prediction accuracy and cache hits, search throughput can vary according to the number of matching strings [9, 17].

The Parallel Failureless Aho-Corasick (PFAC) algorithm [18] extends the AC algorithm [10] to achieve efficient parallelization on a GPU [19]. As compared with CPUs, GPUs not only have higher peak memory bandwidths but also more processing cores. Such rich computational resources are useful for accelerating a memory-intensive problem. The PFAC algorithm creates a thread for every symbol in the text to identify patterns starting at any location. Each thread manages the current status of a deterministic finite state machine. Consequently, search throughput decreases if each thread suffers from multiple state transitions in its state machine. The time complexity for a thread of PFAC ranges from $\mathcal{O}(1)$ to $\mathcal{O}(M)$, where M denotes the longest length of the patterns to be searched simultaneously [18]. Therefore, performance degradation occurs if the text has many partially matching patterns and their matching lengths are relatively long. Such unstable search throughput is not desirable for packet and genome analyses, which must often process high volumes of data. Further, GPUs have smaller memory capacity than CPUs; consequently, maximum search throughput can be restricted by the peak bandwidth of the PCI Express bus, which can diminish the benefits of high-bandwidth video memory.

Another acceleration approach is to exploit bit parallelism in string search. An advantage of the bit-parallel algorithm [20] is that the number of memory references is determined only by the text and pattern lengths. In other words, the bit-parallel algorithm has the same best- and worst-case time complexity, so that it can provide highly robust throughput when faced with various patterns varying in the text and pattern contents. The bit-parallel algorithm was extended by Prasad

et al. [21, 22] to simultaneously search multiple patterns of the same length.

In this paper, to realize fast string search for maximizing stable search throughput, we present a high-throughput method that accelerates a bit-parallel algorithm on a multicore CPU. Our method extends the Prasad's algorithm [21, 22] such that it simultaneously searches multiple patterns of different lengths. In particular, our method is unique in regularizing both control flow and data structures for string search, namely an inherently irregular problem. This regularization facilitates the exploitation of the latest SIMD instructions that are useful to maximize the performance on a CPU. Our method efficiently searches multiple patterns of different lengths by using a data padding scheme that hides the irregularity of pattern lengths. This scheme is integrated into a two-level parallel algorithm that exploits not only data parallelism via OpenMP directives [23] but also bit parallelism via the latest vector instruction set called Advanced Vector Extensions (AVX) 2 [24]. The latter increases search throughput on a CPU core, while the former increases search throughput on a CPU socket. Our CPU-based solution demonstrates competitive search throughput without using special hardware devices such as the GPU and FPGA.

In addition to this introduction, this paper is organized as follows. Section 2 presents related studies in the area of string search. Section 3 summarizes the bit-parallel algorithm. Section 4 presents our proposed method, and shows how it increases the search throughput of the bit-parallel algorithm. Section 5 presents experimental results, and Section 6 provides the conclusions and suggestions for future work.

2. Related Work

Table 1 shows a comparison of previous string matching implementations with their deployed hardware. Prasad *et al.* [21, 22] extended the bit-parallel algorithm [20] to search multiple biological patterns in the text simultaneously. Their algorithm assumes that all simultaneously searched patterns have the same length and a word is large enough to store the patterns. Because the current x64 architecture uses 64-bit words, the total length of simultaneous patterns must therefore be less than 64 symbols. In contrast, our AVX-based algorithm covers multiple patterns of up to 256 symbols in length and accepts simultaneous patterns of different lengths. Our algorithm also exploits data parallelism via OpenMP directives, as detailed in Section 4. Xu *et al.* [25] implemented Prasad's algorithm on a GPU and achieved a search throughput of 0.1 Gbps. A similar bit-parallel algorithm was presented by Yadav *et al.* [26]. Bit-parallel algorithms have a disadvantage in terms of the pattern size. The total length $\hat{m}$ of patterns must be less than the word size.

Külekci [27] presented a filter-then-search algorithm called Streaming SIMD Extensions filter (SSEF) for searching a single pattern. The SSEF algorithm reduced time complexity by detecting possible matches at low cost, which were then given to the succeeding verification process. This algorithm was implemented using Streaming SIMD Extensions (SSE) instructions [33] to accelerate filtering process for single pattern matching. The SSEF algorithm was ten times faster than the bit-parallel length independent matching (BLIM) algorithm [34], which overcame the word size limitation of the bit-parallel algorithm. However, the time

Table 1: Summary of reported search throughputs and their deployed hardware. q denotes the number of patterns and $\overline{m}$ denotes the average of pattern lengths. The search throughput is given by n/T, where n is the text size and T is the execution time including the data transfer time needed for GPU-based implementations.

Work	Hardware	Throughput (Gbps)	Text size n (MB)	Pattern size q	$\overline{m}$
This paper	4-core Core i7	7.7	4,096	10	20
Prasad [21]	2-core Pentium D	0.06–0.08	24	10	10
Prasad [22]	2-core Pentium D	0.05	122	7	n/a
Xu [25]	GeForce 310M	0.1	4	64	80
Yadav [26]	2-core Pentium D	0.01	0.2	40	4
SSEF [27]	Xeon	82	30	1	2,000
MPSSEF [28]	4-core Core i7	0.4–0.6	4	10,000	1,024
MBLIM [34]	Xeon	0.04	100	32	n/a
MTMP-SG [29]	Two 4-core Xeon	0.1–0.4	100	25,000	7
PFAC [30]	GeForce GTX 295	4	0.2	994	23
PFAC [18]	GeForce GTX 580	3–15	256	1,998	21
Vasiliadis [13]	GeForce GTX 480	30	n/a	50,000	20
Oh [31]	GeForce GTX 480	3–15	3,180	34,915	n/a
Zha [8]	Tesla GT 200	9	904	33	17
Tumeo [12]	Four Tesla C2050	12–48	100	190,000	16
Tumeo [9]	Two Xeon X5560	3–20	100	190,000	16
AC-opt [17]	Cray XMT	28	100	190,000	16
AC-opt [16]	Cell/B.E.	3–4.5	4	20,000	8
Schatz [6]	GeForce 8800 GTX	0.4–0.5	1*	312,500	800
Michailidis [15]	FPGA	0.1	30	1	1,024
Singaraju [32]	FPGA	3	n/a	1,237	13

*: 1 base pair = 2 bits

complexity of the SSEF algorithm ranges from $\mathcal{O}(nm)$ to $\mathcal{O}(n/m)$ according to preprocessing effects, where n is the length of the text and m is that of the pattern. The SSEF algorithm was extended by Faro and Külekci [28] for multiple pattern matching. Their achieved search throughput ranged from 0.4 to 0.6 Gbps on a Core i7 processor.

Oh and Ro [29] presented multi-threaded multiple pattern matching with suffix grouping (MTMP-SG), which extended the Wu-Manber algorithm [35] to overcome performance degradation that appear when the text includes many matching

patterns. Their extended algorithm groups the target patterns in terms of their suffixes, and distributes the patterns over multiple threads. Their CPU-based implementation ran at a throughput of up to 0.4 Gbps, which was faster than a GPU-accelerated AC implementation when searching more than 5,000 patterns simultaneously. They concluded that their implementation could be accelerated using SIMD instructions, which we tackled in the present work.

The PFAC algorithm [18] extended the AC algorithm to achieve efficient parallelization on a manycore GPU. To realize this, the PFAC algorithm eliminated the backtracking procedure of the AC algorithm (i.e., the failure transitions in the automaton), because using backtracking includes branches that can cause thread divergence [36] during highly threaded executions. Instead of backtracking, the PFAC algorithm creates a thread for every symbol in the text. Owing to the highly threaded GPU architecture, threads that encounter a failure can be quickly disappeared [18], and these threads can give their occupying resources to remaining unprocessed threads. The PFAC algorithm was 30% faster than the DPAC algorithm, which assigns a larger chunk of text data to each of threads. A similar result was reported in [37]. For 190,000 patterns, their GPU-based implementation achieved a search throughput of 15 Gbps on a GeForce GTX 580; however, search throughput decreased to 3 Gbps if the text included higher numbers of partially matching data (because threads cannot rapidly complete their execution). Lin and Liu [14] implemented a similar parallel failureless algorithm on an FPGA.

Vasiliadis *et al.* [13] presented a DFA-based algorithm for inspecting network packets in real-time. They designed an efficient packet buffering scheme

for transferring packets from the main memory to the video memory. Using a GeForce GTX 480 card, their GPU-based implementation achieved 30 Gbps for 50,000 patterns. Their algorithm was designed for network inspection systems, which assemble the text from packets flowing over the network. For such incoming packets, the search throughput can be maximized by increasing the pattern size rather than the text size. A DFA for 50,000 patterns occupied approximately 800 MB of device memory, which restricted the text size.

Tumeo *et al.* [9] accelerated the AC algorithm using clusters of CPUs and GPUs. Their DPAC algorithm requires halos (i.e., overlapping regions) to find matching strings that cross the boundaries of decomposed text. Because halos are redundantly loaded by multiple threads, the efficiency of parallel execution decreases as we increase the number of divisions for a fixed text size. Further, for text that has many pattern matches, search throughput decreased to 2 Gbps due to a higher number of cache misses [9] and the increasing cost $\mathcal{O}(z)$ of printing the output. A similar data-parallel algorithm called AC-opt was presented by Villa *et al.* [16, 17]. They confirmed that search throughput was unstable on a Xeon machine, but stable on a 128-processor Cray XMT, which had a high-efficient thread scheduler that avoids pipeline stalls caused by irregular memory access. Oh *et al.* [31] and Zha and Sahni [8] implemented a similar DPAC approach on a GeForce GTX 480 GPU and a Tesla GT 200 GPU, respectively.

Tran *et al.* [11] presented a GPU-accelerated implementation of the AC algorithm that takes advantage of the memory hierarchy. Their implementation achieved a search throughput of 127 Gbps on a GeForce GTX 285 GPU. How-

ever, this throughput did not consider the data transfer overhead incurred during copying input and output data between CPU and GPU. Considering this overhead, search throughput of PFAC [18] reduced from 143 Gbps to aforementioned 3–15 Gbps, which was close to those achieved by other GPU-based approaches [4, 12].

Schatz and Trapnell [6] employed a suffix tree to accelerate string search. Because this approach arranges text data in a tree, it is a different approach from just using raw text data. Although a suffix tree of a text enables fast string operations, it generally requires more space than storing the raw text. The search throughput was around 0.5 Gbps on a GeForce GTX 8800 card.

In summary, our bit-parallel algorithm differs from previous algorithms in regularizing both control flow and data structures to achieve stable search throughput for arbitrary data. This contributes to utilize the latest SIMD instruction set for string search, but its SIMD width limits the pattern size.

3. Bit-Parallel Algorithm

The bit-parallel algorithm [20] accelerates string search for a single pattern by processing multiple comparators that are responsible for identifying matches in contiguous regions of the text. To exploit bit parallelism in the search procedure, this algorithm assumes that the pattern length is at most word length. There are two variations called shift-or and shift-and algorithms, which define true to be 0 and 1, respectively. Our algorithm is based on the shift-or algorithm, which has fewer bitwise instructions than the shift-and algorithm.

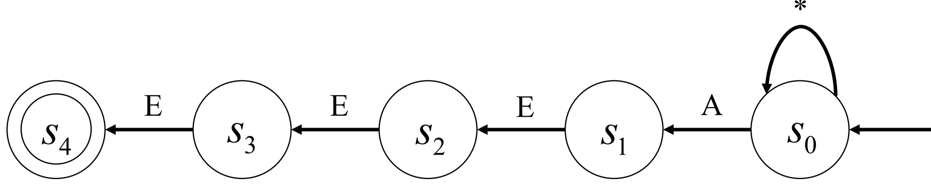


Figure 1: A nondeterministic finite automaton for pattern “AEEE;” note that * on the state transition from state s_0 to itself represents any arbitrary symbol in the given language.

In the following discussion, let n and m be the length of text t and pattern p, respectively. Further, let t_i and p_i be the i-th symbol of text t and pattern p, respectively, where $i \geq 1$.

As a preprocessing phase, the bit-parallel algorithm creates a nondeterministic finite automaton from the given pattern, an example of which is shown in Fig. 1. This automaton has m states $s_1, s_2, \ldots, s_m$ in addition to initial state s_0. State s_i corresponds to i-th symbol p_i of the pattern, where $1 \leq i \leq m$. In other words, s_i is the current state of the automaton when the first i symbols of the pattern match the text. We assume $s_i = 0$ if s_i is the current state of the automaton (otherwise, $s_i = 1$). String search can then be realized by updating bit sequence $S = s_m s_{m-1} \ldots s_1$ for every input symbol from text t. The bitwise operation for this update can be given by

$$S_i = (S_{i-1} \ll 1) \mid B[t_i], \quad (1)$$

where operators $\ll$ and $\mid$ represent logical left shift and logical disjunction, respectively, S_i is the bit sequence after reading i-th symbol t_i, and $S_0 = 11 \ldots 1$. Further, B is a table that consists of the appropriate bitmasks for the input sym-

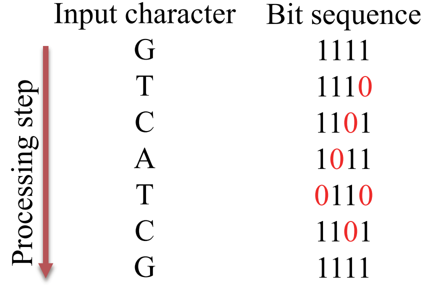


Figure 2: The behavior of the bit-parallel algorithm for text “GTCATCG” and pattern “TCAT;” note that 0s in the bit sequence indicate matches between the text and pattern; the bit sequence after the fifth input symbol becomes 0110, where the most significant bit of 0 indicates a full match and the least significant bit of 0 indicates a prefix match of length 1.

bols. Let $B[x] = b_m b_{m-1} \ldots b_1$ be the bitmasks for input symbol x. Bitmasks $B[x]$ can be given by

$$b_i[x] = \begin{cases} 0, & \text{if } x = p_i, \\ 1, & \text{otherwise.} \end{cases} \tag{2}$$

Thus, as illustrated in Fig. 2, a shift operation realizes a state transition of the automaton, and bitmasks B validate the transition according to input symbol t_i.

3.1. Simultaneously Searching for Multiple Patterns

Prasad *et al.* [21, 22] extended the bit-parallel algorithm to simultaneously search for multiple patterns of the same length m. Let q (≥ 2) be the number of patterns to be searched simultaneously. Given q patterns, they constructed q automata and generated a bit sequence in which a bit corresponds to a state of the automata; as shown in Fig. 3(a), the bit sequence is based on cyclic arrangement $s_m^q s_m^{q-1} \ldots s_m^1 s_{m-1}^q s_{m-1}^{q-1} \ldots s_1^1$, where s_i^j represents the i-th state of the automaton

generated from the j-th pattern with $1 \leq i \leq m$ and $1 \leq j \leq q$. An advantage of this cyclic arrangement is the simplicity of bitwise operations required to update the states of the automata. Because the states of the same automaton are assigned to every q bits of the sequence, these states can be updated using bitwise operations

$$S_i = (S^{i-1} \ll q) \mid B[t_j]. \tag{3}$$

Note that the bit sequence can also be generated using a block arrangement, as shown in Fig. 3(b). Using this arrangement, the bit sequence can be given by $s_m^q s_{m-1}^q \dots s_1^q s_m^{q-1} s_{m-1}^{q-1} \dots s_1^1$; however, this arrangement requires complicated operations to update the states of the automata. For example, shift operations must not be applied to a boundary of the bit sequence (i.e., $s_1^q s_m^{q-1}$) where neighboring bits belong to different automata. Consequently, the bit sequence consists of bits to be shifted and those not to be shifted, making bitwise operations complicated.

4. Proposed Method

Our algorithm extends the bit-parallel algorithm [20] to simultaneously search for patterns of differing lengths. Key differences between our algorithm and the original bit-parallel algorithm are summarized as follows:

1. Simultaneous search for multiple patterns via AVX2 instructions [24].
2. Data structure capable of rapid identification of matching patterns.
3. Data parallel search via OpenMP directives [23].

b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
s_3^4	s_3^3	s_3^2	s_3^1	s_2^4	s_2^3	s_2^2	s_2^1	s_1^4	s_1^3	s_1^2	s_1^1

(a)

b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
s_3^4	s_2^4	s_1^4	s_3^3	s_2^3	s_1^3	s_3^2	s_2^2	s_1^2	s_3^1	s_2^1	s_1^1

(b)

Figure 3: Bit arrangement [21, 22] for simultaneously searching for multiple patterns with (a) a cyclic arrangement and (b) a block arrangement for $q = 4$ patterns of length $m = 3$. The colored block below each array element represents the correspondence between the element and its responsible pattern.

Inputs to our method are text t, q patterns $p_1, p_2, \ldots, p_q$, and table B. Our method outputs numerical sequence R, which stores search results. Sequence R consists of n integers whose values range from 0 to q. The i-th value R_i of R is given by $R_i = j$ if text t matches the pattern p_j $(1 \leq j \leq q)$ and its last matching symbol is the i-th symbol of text t. Value R_i equals zero if there is no matching pattern ending at the i-th symbol of the text.

In the discussion below, let m_j be the length of the j-th pattern, where $1 \leq j \leq q$. Also, note that the data structure representing R does not accept multiple patterns to be matched with the same ending location. Consequently, our method assumes that patterns ending with the same suffix are not processed simultaneously. Accordingly, patterns with a common suffix must be grouped into different sets to process them sequentially but with different patterns.

Similarly, there are two assumptions regarding the length and number of simultaneously searched patterns; these assumptions are

$$q \leq 32, \tag{4}$$

$$(M-1) \cdot q \leq 224, \tag{5}$$

where M is the maximum length of q simultaneous patterns, which is given by $M = \max_{1 \leq j \leq q} m_j$. Details are provided below.

4.1. Simultaneously Searching for Multiple Patterns via AVX2

Our method realizes vectorized search using AVX2 instructions [24] to achieve simultaneous search for more patterns of longer lengths. AVX2 provides a SIMD instruction set and is available on the Intel's Haswell architecture. The AVX2 instruction set includes floating-point and integer operations for 256-bit vector data. Consequently, AVX2 accepts patterns four times longer than that of x64 scalar instructions, which operate with 64-bit words.

To simplify vector operations, our method employs the cyclic arrangement scheme presented by Prasad *et al.* [21, 22]. Our approach extends their original scheme by enabling efficient search for patterns of different lengths (details are presented below). The logical shift and disjunction operations needed for Eq. (3) can be implemented with the `_mm256_slli_epi32()` and `_mm256_or_si256()` mnemonics, respectively.

4.2. Data Structure for Rapidly Identifying Matching Patterns

Key issues that must be resolved for simultaneously searching patterns of different length include the following:

1. Adapting to the cyclic arrangement scheme.
2. Rapidly identifying matching patterns.

In regards to issue 1., short patterns create automata that consist of fewer states. Therefore, the original scheme cannot be directly used for patterns of differing lengths. Consequently, a data padding scheme is required to adapt the cyclic scheme to patterns of different length. A naive padding scheme may add dummy bits at the tail of patterns, but this scheme goes against the simplicity of the updating operation shown in Eq. (3), because the states of the same automaton do not appear at regular intervals in the bit sequence.

In contrast, our method adds dummy bits at the head of patterns, as shown in Fig. 4. This data padding approach maintains the simplicity of the updating operations; however, the bitmasks must be adapted to this scheme to handle the dummy bits (i.e., the dummy states in the automata). In other words, bitmasks B must be modified such that they validate any transition to the dummy states for an arbitrary symbol. For example, bitmask for the i-th bit must be given as 0. In general,

$$b_{(i-1)q+j} = 0, \qquad \text{for all } 1 \leq j \leq q \text{ and for all } 1 \leq i \leq M - m_j. \tag{6}$$

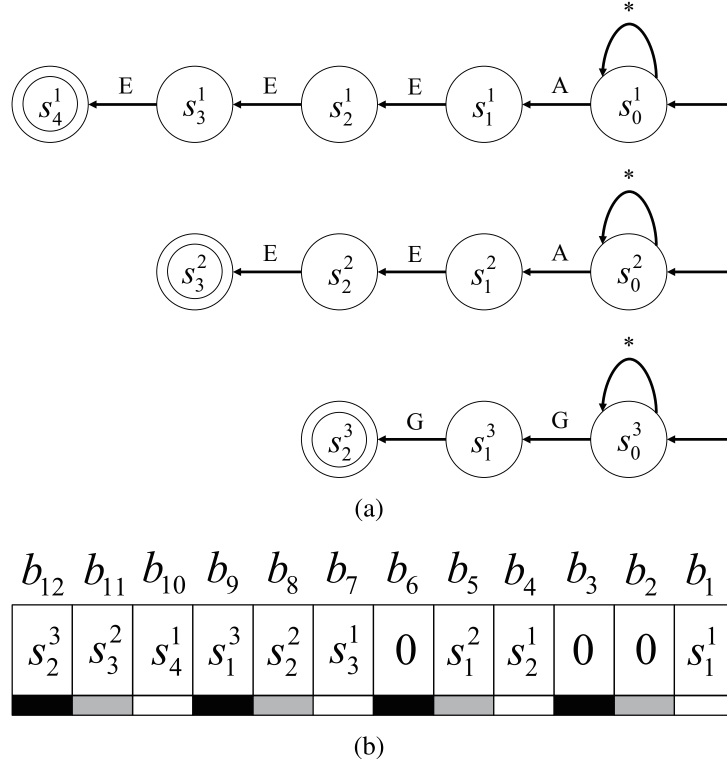


Figure 4: Data padding scheme for different pattern lengths with (a) an automata for three example patterns "AEEE," "AEE," and "GG" and (b) a bit sequence of length 12 after data padding; in this example, b_2, b_3, and b_6 are dummy bits set to 0 to validate a state transition for arbitrary symbols.

Similar to the bitmasks, the corresponding part of bit sequence S_0 must be initialized with 0.

We next discuss issue 2., i.e., how matching patterns can be identified via our method. In the original cyclic scheme [21, 22], we can easily identify the matching pattern p_j $(1 \leq j \leq q)$ by finding $s_i = 0$ from the bit sequence (i.e., checking if the current state is an accepting state), then computing $j = i \mod p$ (i.e., specifying the automaton that has the detected accepting state); however, this strategy cannot be directly used in our method, because dummy states could be

b_{256} $\cdots$ b_{225} $\cdots$ b_{216} b_{215} $\cdots$ b_1

1 1 $\cdots$ 1	s_2^3	s_3^2	s_4^1	s_1^3	s_2^2	s_3^1	0	s_1^2	s_2^1	0	0	s_1^1	0 0 $\cdots$ 0

Figure 5: Our data structure, in which two different sets of dummy bits are added to the most significant bits and the least significant bits; dummy bits added to the least significant bits are set to 0, as mentioned in Fig. 4, while in contrast, dummy bits added to the most significant bits are set to 1 to invalidate accepting bits that do not correspond to any patterns.

detected as accepting states. To avoid this invalid dummy-state detection, we need a filtering procedure that eliminates dummy bits before investigation, but such a preprocessing procedure complicates the search process.

The key approach here is a static assignment of accepting states to specific bits. Our method assumes that accepting bits only appear in the most significant 32 bits: $b_{256}b_{255}\ldots b_{225}$, as depicted in Fig. 5. This assumption introduces additional dummy bits that are added to the most significant bits, i.e., if $q < 32$, we modify bitmasks B such that dummy bits that do not correspond to any automata are always set to 1. In other words, the bitmasks invalidate any transition from an accepting state to a dummy state. The modification to dummy bits $b_{256}b_{255}\ldots b_{225+q}$ is given by

$$b_i = 1, \qquad \text{for all } i > 224 + q. \tag{7}$$

Our static assignment scheme also has the advantage of acceleration. Because the accepting states appear only in the most significant 32 bits, we can apply the tzcnt instruction of AVX2, which accepts a 32-bit unsigned integer value and re-

turns its trailing zero count from the least significant bit. More specifically, it can rapidly identify the location of 1 from the least significant bit. Consequently, we can find the matching pattern by using the tzcnt instruction with a 32-bit value reversed from the most significant 32 bits of current bit sequence S_i. Note that Intel's CPU architectures before Ivy Bridge do not support AVX2, but the tzcnt instruction can be implemented with two instructions of the SSE instructions [33], i.e., `_tzcnt_u32(x)` can be replaced with `_mm_popcnt_u32((~x)&(x-1))`.

The data structure described above has a limitation on the length and number of simultaneous patterns. First, Eq. (4) indicates that at most 32 patterns can be processed simultaneously, because the accepting states are placed in the most significant 32 bits of a 256-bit vector. Second, Eq. (5) limits word length. In our method, all $M - 1$ states except for the initial state of q automata are stored in the vector of 224 bits.

4.3. Data-Parallel Search via OpenMP Directives

To maximize the performance on a multicore CPU, our method exploits data parallelism of the search algorithm by using OpenMP directives. Our method divides text t into c chunks, where c is the number of CPU cores, which are assigned to the CPU core. Similar to Tumeo *et al.* [9], we place extra ghost regions to find matches crossing a boundary of the decomposed region. Because our method searches multiple patterns simultaneously, the ghost region size is given by $M - 1$, where M is the maximum length of the simultaneous patterns.

5. Experimental Results

To evaluate the performance of our method, we measured search throughput $\rho = n/T$ of the following six implementations: (1) our method, (2) our naive method (with a naive padding scheme), (3) PFAC-GPU r1.2 [18], (4) PFAC-CPU r1.2 [18], (5) MultiFast v1.4.2 [38], and (6) DP-MultiFast, namely a data-parallel version of MultiFast. The use of T represents the execution time spent for performing string search. PFAC-GPU and PFAC-CPU implement the PFAC algorithm [18] on a GPU and a multicore CPU, respectively. The PFAC-CPU approach was a parallel implementation that exploited data parallelism by dividing the text into several chunks. MultiFast is an implementation of the AC algorithm; similar to PFAC-CPU, we extended this implementation to use all CPU cores via OpenMP directives, so that we used DP-MultiFast as a DPAC implementation. Note that T excluded the load time of the text and the initialization time needed to create the appropriate bitmasks for the patterns. Also note that the execution time of the PFAC-GPU approach included the data transfer time required to copy data between CPU and GPU.

Table 2 shows the specifications of our experimental environment. We used CUDA 4.2 [36] for PFAC-GPU because this implementation could not be compiled with the latest CUDA 5.5. Conversely, our experimental GPU required the latest driver to run kernels, so we used the latest driver distributed with CUDA 5.5. We implemented the tzcnt instruction via the popcnt instruction because GCC 4.8.1 does not support the tzcnt instruction. For our measurements, we activated Hyper-Threading technology on the deployed CPU such that eight threads were

Table 2: Experimental setup.

Item	Specification
OS	Ubuntu 12.04.3
CPU	Intel Core i7 4770K
Main memory	DDR3-1600 16 GB
GPU	NVIDIA GeForce GTX 780
Video memory	GDDR5 3 GB
I/O bus	PCIe 3.0 x16
Compiler	GCC 4.8.1
Optimization option	`-O2`
CUDA Driver	5.5
CUDA Runtime	4.2

created for the CPU-based implementations except MultiFast.

5.1. Analysis of Search Throughput

To compare our method with the PFAC algorithm, we measured search throughput ρ while varying length l and ratio x of prefix matches. Ratio x here is given by n'/n, where n' is the cumulative number of patterns that have a prefix match in the text. We used dummy text to exhaustively investigate l and x; the dummy text consisted of repeating sequence "abcdefghij" and had a data size of 512 MB. As for the simultaneous patterns, we used $q = 10$ patterns consisting of $m_j = 20$ symbols, where q represents the number of patterns to be searched and $1 \leq j \leq q$. For the pattern, we used a permutated version of the basic sequence that partially replaced lowercase letters with uppercase letters to control the value of l.

For example, we used pattern "abcDEFGHIJABCDEFGHIJ" for $l = 3$ because the first three symbols match the text. Further, we used the cyclic permutated versions of the basic sequence to control the value of x. For $l = 3$

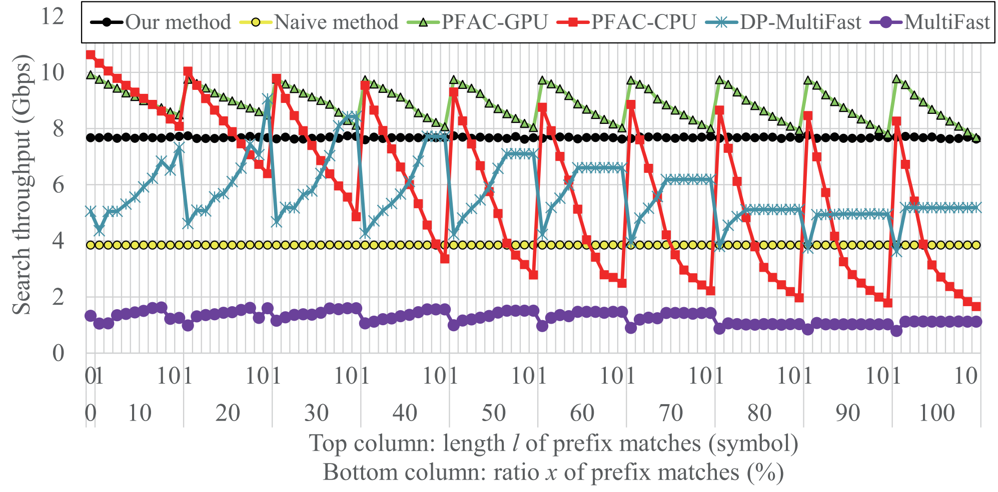


Figure 6: Comparing measured search throughput for our method (with and without our padding scheme) to that of PFAC-GPU, and PFAC-CPU. Ten patterns of length 20 were used for measurement.

and $x = 0.2$, for example, we used “abcDEFGHIJABCDEFGHIJ” and “bcdEFGHIJABCDEFGHIJA” as partially matching patterns of ten simultaneous patterns, the remaining nonmatching patterns set appropriately as “CD . . . B,” “DE . . . C,” “EF . . . D,” “FG . . . E,” “GH . . . F,” “HI . . . G,” “IJ . . . H,” and “JA . . . I.” We have $x = 0.2$ because the first two patterns partially match to every ten symbols in the text. Note that the execution time required to create bitmasks for $q = 10$ patterns was approximately 0.15 ms, and thus was not a performance bottleneck for our method.

Figure 6 presents search throughput measured for our proposed method (with and without our padding scheme), PFAC-GPU, PFAC-CPU, MultiFast, and DP-MultiFast. Our naive method used a cyclic padding scheme that simply added dummy bits from the most significant part of the bit sequence. In this scheme,

accepting bits do not appear in the contiguous region; therefore, we detect accepting automata using bitwise operations. Note that our naive method used the same padding scheme as Prasad's algorithm [21, 22]. We think that our naive method can be roughly regarded as Prasad's algorithm, but they may not be same because it is not clear how Prasad's algorithm identifies the matching pattern from q candidates.

From the figure, we observe that our method achieved robust throughputs ρ of approximately 7.7 Gbps for any length l and ratio x of matches. We also found that our data padding scheme increased ρ from 3.8 to 7.7 Gbps, doubling the search throughput. This increase was due to the simplified procedure for specifying the matching pattern from $q = 10$ patterns. Owing to this simplification, the algorithm reduced the number of instructions as compared with the naive padding scheme. Our naive method spent long time to identify accepting automata, because their accepting bits existed over the two 32-bit contiguous words of the vector. Thus, search throughput increased for any l and x, and the overhead of our data padding scheme was negligible.

In contrast, the search throughput of the PFAC algorithm depends on l regardless of whether its implementation runs on a CPU or GPU. Search throughput ρ decreased as length l of the matching part increased. This performance degradation was due to the amount of computation per thread, which ranged from $\mathcal{O}(1)$ to $\mathcal{O}(M)$ and increased with l, i.e., as the matching part increased in length, threads had to process an increased number of state transitions in their respective automata. Such a series of state transitions must be iteratively processed due

to the elimination of failure transitions. However, threads are allowed to terminate immediately when finding a mismatch from its responsible starting position. Therefore, the PFAC algorithm is efficient for text data that have a few matching patterns.

The throughput of PFAC-CPU also depends on ratio x. As we increased x, more automata had to process more state transitions. PFAC-GPU was robust against such a large number of automata, because the GPU concurrently processes tens of thousands threads; however, PFAC-CPU was processed by eight threads, suffering from an increased amount of computation.

Figure 6 shows that the search throughput of DP-MultiFast was unstable, ranging from 3.6 to 9.1 Gbps. In addition, DP-MultiFast showed a different behavior compared with PFAC; with DP-MultiFast, the search throughput increased with length l of prefix matches. Because DP-MultiFast processes failure transitions, backtracking overhead occurs after loading a nonmatching symbol. As l increased, the occurrence of this overhead became relatively rare, increasing search throughput. However, similar to the behavior reported in [9], search throughput progressively decreased as we increased x. The same behavior was observed for MultiFast, which was approximately 4.5 times slower than DP-MultiFast.

We next investigated how our achieved throughput was robust against the imbalance of pattern lengths. As shown in Fig. 7, we fixed average length l and ratio x of matches, but varied length m_j $(1 \leq j \leq q)$ of simultaneous patterns. We used $q = 3$ patterns whose first $(m_j - 1)$ symbols matched the given text. Because the total length $\sum_j m_j$ of patterns was fixed as 103 symbols, 100 symbols of the three

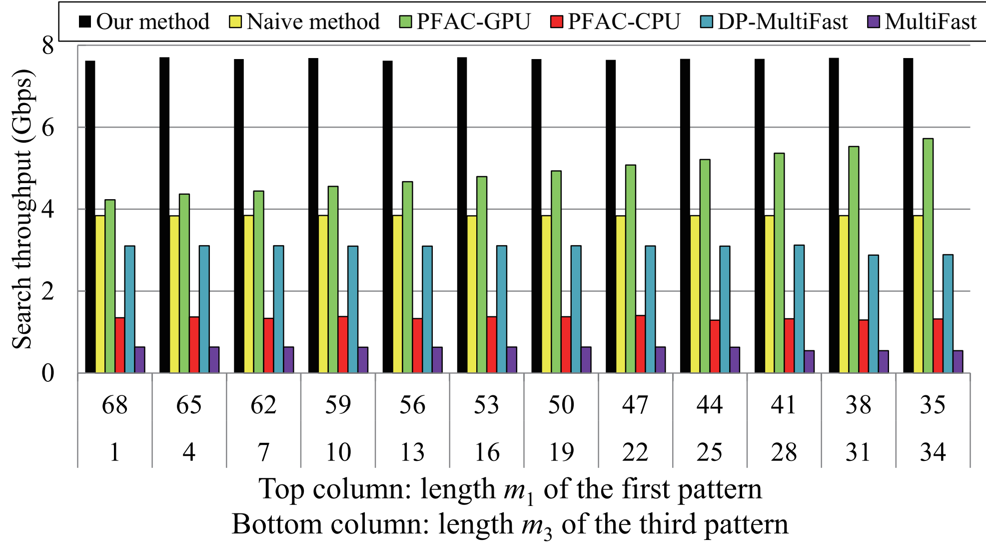


Figure 7: Search throughput measured while varying lengths m_1 and m_3 of simultaneous patterns; length m_2 of the second pattern is fixed at $m_2 = 34$. The text data had a data size of 512 MB.

patterns matched the text. For simplicity, we fixed length $m_2 = 34$ of the second pattern while remaining lengths m_1 and m_3 were varied accordingly.

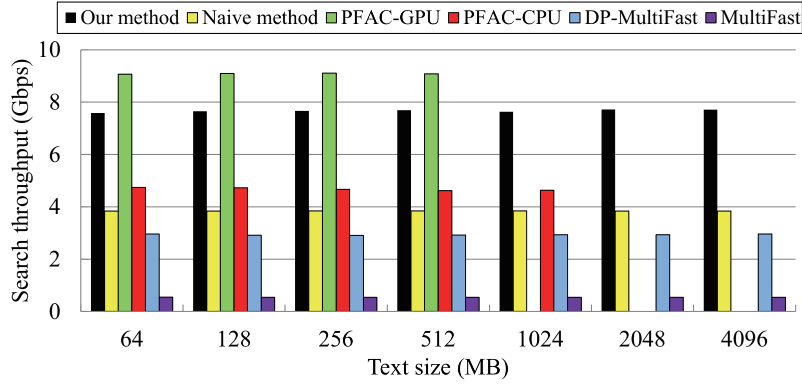
In the figure, the search throughput of our method proved robust against the imbalance of pattern lengths. Conversely, the PFAC-GPU approach increased search throughput from 4.2 to 5.7 Gbps, as the lengths of the simultaneous patterns became closer. This increase was the net effect of load balancing that occurred between threads. When $m_1 = 68$ and $m_3 = 1$, the number of state transitions per thread were 0, 33, and 67 for the first, second, and third automata (i.e., patterns), respectively. In contrast, when $m_1 = 35$ and $m_3 = 34$, the number of state transitions were 34, 33, and 33, respectively. Because the GPU issues instructions for 32 consecutive threads (these groups are called warps [36]), the latter case allowed threads in the same warp to have a similar workload and thus achieve

higher throughput than the former case. As opposed to Fig. 6, Fig. 7 show that our method was faster than PFAC-GPU. This was due to the occurrence of long prefix matches. Length l of prefix matches in Fig. 7 was larger than that in Fig. 6: $34 \leq l \leq 67$ in Fig. 7, whereas $0 \leq l \leq 10$ in Fig. 6.

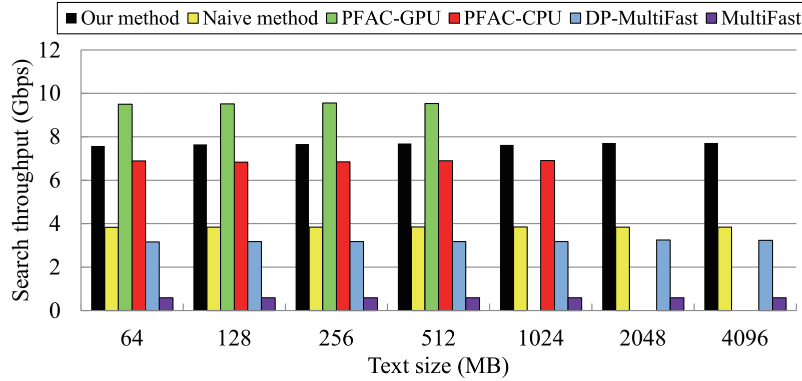
Similar to our results, the search throughputs of PFAC-CPU, MultiFast, and DP-MultiFast were also robust against the combination of pattern lengths, but they performed at best 1.4, 0.6, and 3.1 Gbps, respectively. The number of symbols loaded by PFAC-CPU is determined by the length of matching strings, which we fixed at 100 symbols. Consequently, search throughput did not significantly change even though the workload was imbalanced. Further, the load-balancing issue described above did not appear in the CPU-based implementations, which independently run different threads (i.e., issues different instructions to running threads) by exploiting data parallelism of the search algorithm.

5.2. Case Study with Real Data

As a case study, we measured search throughput using a genome dataset [39] and a corpus dataset [40] in the area of natural language processing (NLP). The genome dataset consisted of full-length complementary DNA of human; because genome datasets consist of only four symbols (i.e., A, T, G and C), the text includes many partial matches. In contrast, the NLP corpus dataset consisted of 239 symbols, including uppercase and lowercase letters, numbers, and control characters. Consequently, corpus datasets usually have far fewer partial matches than genome datasets. For our measurements, we eliminated the header of the



(a)



(b)

Figure 8: Search throughput measured with different text sizes. Results for (a) a genome dataset and (b) an NLP corpus dataset. PFAC-GPU and PFAC-CPU failed to obtain search results for large text with data sizes larger than 512 MB and 1024 MB, respectively.

genome dataset; similarly, non-ASCII characters (such as Greek characters) were eliminated from the corpus dataset

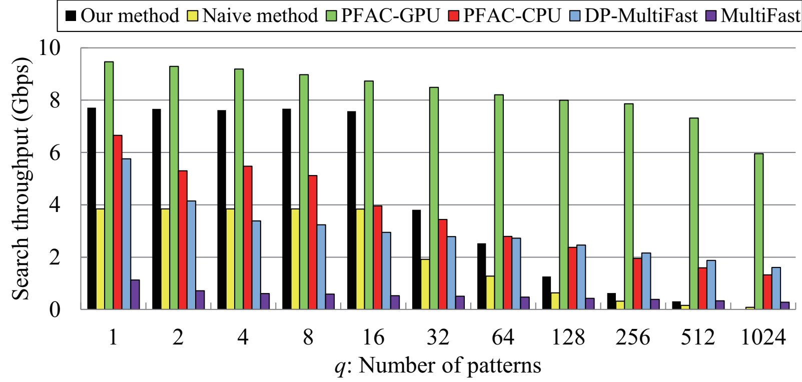
Figure 8 shows search throughputs for the aforementioned six implementations, measured with varying text sizes ranging from 64 to 4096 MB. We used $q = 8$ patterns for both datasets. For the genome dataset, we used patterns consisting of 26–28 symbols extracted randomly from H. sapiens|ENST00000408996.

For the corpus dataset, we used patterns of random words composed of 5–10 symbols, including director, Americans, adventure, Bolshevism, class, ridden, think, and workingmen.

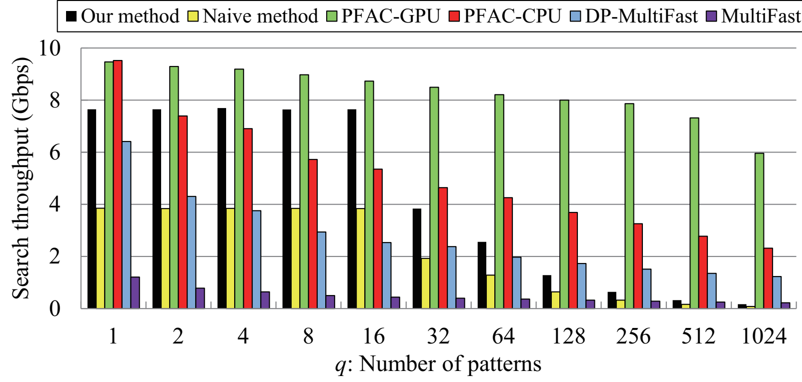
From the figure, we observe that our data padding scheme doubled search throughput for real datasets; however, the achieved throughput was 7.7 Gbps, whereas that of PFAC-GPU was 9.5 and 9.1 Gbps for the genome dataset and corpus dataset, respectively. Owing to the exhaustion of the video memory, PFAC-GPU failed to obtain search results for large text with data sizes larger than 512 MB. In contrast, our method ran on the CPU and successfully produced search results for such large text (as long as the text could be stored in main memory).

For the NLP corpus dataset, the PFAC-CPU approach achieved a throughput of 6.9 Gbps, slightly lower than that of our achieved throughput; however, the throughput for the genome dataset decreased to 4.7 Gbps. As compared with the corpus dataset, the genome dataset usually produced many partial matches because of its lack of symbols; such a distribution caused performance degradation in the PFAC algorithm (as noted in the performance analysis presented in Section 5.1).

Eqs. (4) and (5) are the disadvantages of our method, which limit the length and number of patterns. To evaluate this point, we investigated search throughput with different numbers of patterns. We randomly selected patterns from the text. The length of patterns was fixed to 10. Figure 9 shows search throughputs measured with $1 \leq q \leq 1024$. Given many patterns that cannot fit into a word, our method must be iterated with varying the patterns. Consequently, the



(a)



(b)

Figure 9: Search throughput measured with different numbers of patterns. Results for (a) a genome dataset and (b) an NLP corpus dataset. The data size of the text was 512 MB.

search throughput reduces to $1/k$, where k represents the number of iterations needed to process all of the patterns. As shown in Fig. 9, such iterations decreased search throughput when $q \geq 32$, and our method turned out to be slower than DP-MultiFast if the patterns did not satisfy Eqs. (4) and (5). Consequently, our method may not be suitable for network inspection systems, which can deal with many patterns for flowing packets.

For long patterns, we can select the part of the patterns such that they can be fit

into a word. The selected part is then searched by using our method as a filter, and the rest of the patterns is verified on matching positions in the text. We believe that the future architecture with longer vector registers will relax the limitation on the length and number of patterns. For example, Xeon Skylake and Xeon Phi Knights Landing processors support AVX-512, a 512-bit version of AVX.

As shown in Fig. 9, AC-based algorithms decreased search throughput as number q of patterns increased. This performance degradation shows how irregular control flow affects search throughput. Because AC-based algorithms construct a state machine from q patterns, the complexity of the state machine increases with q. For example, each state have many outgoing transitions, which result in complicated flow. Consequently, the irregularity of control flow can decrease the search throughput of AC-based algorithms. This irregularity also explains why DP-MultiFast decreased search throughput for the real data compared to the synthetic data. For example, DP-MultiFast achieved more than 4 Gbps in Fig. 6, but resulted in 3.2 Gbps in Fig. 8. For the synthetic data, we constructed the patterns by permutation of the basic sequence. Consequently, the state machine for the synthetic data has more regular state transitions than that for the real data. With respect to the real data, the simplest state machine was generated when $q = 1$. As shown in Fig. 9, such a simple state machine maximized the search throughputs of AC-based algorithms.

In summary, our method achieves stable search throughput if the text and patterns fit into the main memory and a word size, respectively. This robust stability comes from the regularization of control flow and data structures. For large pat-

terns that cannot fit into a word, our method must be iterated until processing all of the patterns. In contrast, PFAC, AC, and DPAC approaches can vary search throughput according to the text and pattern contents. In particular, the length of matches determines search throughput. Furthermore, PFAC can degrade search throughput if the text has many partially matching patterns. We think that this frequently occurs for datasets with a small alphabet size. Finally, AC and DPAC approaches can decrease search throughput due to irregular control flow.

6. Conclusion

In this paper, we presented a high-throughput method for a bit-parallel algorithm that realizes fast string search for robust throughput maximization. Our method used AVX2 instructions to increase search throughput on a CPU core and exploited the data parallelism of the search algorithm via OpenMP directives. Further, our method was based on a data structure applicable for efficiently searching multiple patterns of different lengths. Increasing efficiency was achieved by data padding two parts of the bit sequence. Owing to our data padding scheme, only a single instruction was needed to detect an accepting automaton.

In our experiments, we found that our method doubled search throughput by padding dummy bits accordingly. We also found that our bit-parallel method achieved robust throughput measures even when faced with variations in the text and pattern contents. This robust stability comes from the regularization of control flow and data structures. As compared with the CPU implementation of the PFAC algorithm, our method achieved 1.4 times higher throughput for a genome

dataset that had many partial matches between the text and patterns; however, our method has a limitation on the total length of patterns, which must be large, but smaller than the word size. Similar comparative results were obtained with the GPU implementation of the PFAC algorithm, but our method was able to handle large datasets that could not be stored in video memory. We think that our regularization approach is useful to accelerate string search on a SIMD machine.

Our future work includes an investigation of approximated string search [41], which is an improvement of the bit-parallel algorithm; this type of search has proven effective in handling with biological sequences that include many errors due to high-speed sequencers.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 23300007 and 23700057 and the Japan Science and Technology Agency CREST program, "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Computing Systems." We also thank Dr. Takehiko Kashiwagi of NEC Corporation for his insights regarding our research. We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] G. Navarro, A guided tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.

[2] S. Faro, T. Lecroq, The exact online string matching problem: A review of the most recent results, ACM Computing Surveys 45 (2), article 13, 42 pages.

[3] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, S. Ioannidis, Gnort: High performance network intrusion detection using graphics processors, in: Proc. 11th Int'l Symp. Recent Advances in Intrusion Detection (RAID'08), 2008, pp. 116–134.

[4] A. Tumeo, O. Villa, D. Sciuto, Efficient pattern matching on GPUs for intrusion detection systems, in: Proc. 7th Int'l Conf. Computing Frontiers (CF'10), 2010, pp. 87–88.

[5] A. Tumeo, O. Villa, Accelerating DNA analysis applications on GPU clusters, in: Proc. 8th Symp. Application Specific Processors (SASP'10), 2010, pp. 71–76.

[6] M. C. Schatz, C. Trapnell, Fast exact string matching on the GPU, `http://www.cbcb.umd.edu/software/cmatch/Cmatch.pdf`.

[7] S. Gog, K. Karhu, J. Kärkkäinen, Multi-pattern matching with bidirectional indexes, in: Proc. 18th Int'l Computing and Combinatorics Conf. (COCOON'12), 2012, pp. 384–395.

[8] X. Zha, S. Sahni, GPU-to-GPU and host-to-host multipattern string matching on a GPU, IEEE Trans. Computers 62 (6) (2013) 1156–1169.

[9] A. Tumeo, O. Villa, D. G. Chavarría-Miranda, Aho-Corasick string matching on shared and distributed-memory parallel architectures, IEEE Trans. Parallel and Distributed Systems 23 (3) (2012) 436–443.

[10] A. V. Aho, M. J. Corasick, Efficient string matching: An aid to bibliographic search, Communications of the ACM 18 (6) (1975) 333–340.

[11] N.-P. Tran, M. Lee, S. Hong, Y. Choi, High throughput parallel implementation of Aho-Corasick algorithm on a GPU, in: Proc. IEEE 27th Int'l Symp. Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW'13), 2013, pp. 1807–1816.

[12] A. Tumeo, S. Secchi, O. Villa, Experiences with string matching on the Fermi architecture, in: Proc. 24th Int'l Conf. Architecture of Computing Systems (ARCS'11), 2011, pp. 26–37.

[13] G. Vasiliadis, M. Polychronakis, S. Ioannidis, Parallelization and characterization of pattern matching using GPUs, in: Proc. 6th IEEE Int'l Symp. Workload Characterization (IISWC'11), 2011, pp. 216–225.

[14] W. Lin, B. Liu, Pipelined parallel AC-based approach for multi-string matching, in: Proc. 14th Int'l Conf. Parallel and Distributed Systems (ICPADS'08), 2008, pp. 665–672.

[15] P. D. Michailidis, K. G. Margaritis, A programmable array processor architecture for flexible approximate string matching algorithms, J. Parallel and Distributed Computing 67 (2) (2007) 131–141.

[16] O. Villa, D. P. Scarpazza, F. Petrini, Accelerating real-time string searching with multicore processors, IEEE Computer 41 (4) (2008) 42–50.

[17] O. Villa, D. Chavarría-Miranda, K. Maschhoff, Input-independent, scalable and fast string matching on the Cray XMT, in: Proc. 23rd IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS'09), 2009, pp. 384–395.

[18] C.-H. Lin, C.-H. Liu, L.-S. Chien, S.-C. Chang, Accelerating pattern matching using a novel parallel algorithm on GPUs, IEEE Trans. Computers 62 (10) (2013) 1906–1916, `http://code.google.com/p/pfac/`.

[19] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: A unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55.

[20] R. Baeza-Yates, G. H. Gonnet, A new approach to text searching, Communications of the ACM 35 (10) (1992) 74–82.

[21] R. Prasad, S. Agarwal, I. Yadav, B. Singh, A fast bit-parallel multi-patterns string matching algorithm for biological sequences, in: Proc. Int'l Symp. Biocomputing (ISB'10), no. 46, 2010, 4 pages.

[22] R. Prasad, A. K. Sharma, A. Singh, S. Agarwal, S. Misra, Efficient bit-parallel multi-patterns approximate string matching algorithms, Scientific Research and Essays 6 (4) (2011) 876–881.

[23] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel Programming in OpenMP, Morgan Kaufmann, San Mateo, CA, 2000.

[24] Intel Corporation, Intel architecture instruction set extensions programming reference, http://download-software.intel.com/sites/default/files/managed/71/2e/319433-017.pdf (Dec. 2013).

[25] K. Xu, W. Cui, Y. Hu, L. Guo, Bit-parallel multiple approximate string matching based on GPU, in: Proc. 1st Int'l Conf. Information Technology and Quantitative Management (ITQM'13), 2013, pp. 523–529.

[26] I. Yadav, B. Singh, S. Agarwal, R. Prasa, An efficient bit-parallel multi-patterns word searching algorithm through splitting the text, in: Proc. Int'l Conf. Advances in Recent Technologies in Communication and Computing (ARTCom'09), 2009, pp. 406–410.

[27] M. O. Külekci, Filter based fast matching of long patterns by using SIMD instructions, in: Proc. Prague Stringology Conf. (PSC'09), 2009, pp. 118–128.

[28] S. Faro, M. O. Külekci, Fast multiple string matching using streaming SIMD extensions technology, in: Proc. 19st Symp. String Processing and Information Retrieval (SPIRE'12), 2012, pp. 217–228.

[29] D. Oh, W. W. Ro, Multi-threading and suffix grouping on massive multiple pattern matching algorithm, The Computer J. 55 (11) (2012) 1331–1346.

[30] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, J.-M. Shyu, Accelerating

string matching using multi-threaded algorithm on GPU, in: Proc. Global Communications Conf. (GLOBECOM'10), 2010, 5 pages.

[31] Y. Oh, D. Oh, W. W. Ro, GPU-friendly parallel genome matching with tiled access and reduced state transition table, Int'l J. Parallel Programming 41 (4) (2013) 526–551.

[32] J. Singaraju, J. A. Chandy, FPGA based string matching for network processing applications, Mircoprocessors and Microsystems 32 (4) (2008) 210–222.

[33] Intel Corporation, Intel SSE4 Programming Reference, http://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf (Jul. 2007).

[34] M. O. Külekci, BLIM: A new bit-parallel pattern matching algorithm overcoming computer word size limitation, Mathematics in Computer Science 3 (4) (2010) 407–420.

[35] S. Wu, U. Manber, Fast text searching allowing errors, Communications of the ACM 35 (10) (1992) 83–91.

[36] NVIDIA Corporation, CUDA C Programming Guide Version 6.0, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (Feb. 2014).

[37] S. Arudchutha, T. Nishanthy, R. G. Ragel, String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm, in: Proc. 8th

IEEE Int'l Conf. Industrial and Information Systems (ICIIS'13), 2013, pp. 231–236.

[38] K. Kanani, MultiFast: Multiple string search via Aho-Corasick C library, `http://sourceforge.net/projects/multifast/` (2013).

[39] R. Yamashita, H. Wakaguri, S. Sugano, Y. Suzuki, K. Nakai, DBTSS provides a tissue specific dynamic view of transcription start sites, Nucleic Acid Research 38 (2010) D98–104, `http://dbtss.hgc.jp/`.

[40] P. Ferragina, G. Navarro, Pizza&chili corpus compressed indexes and their testbeds, `http://pizzachili.dcc.uchile.cl/` (Sep. 2005).

[41] T. T. Tran, M. Giraud, J.-S. Varré, Bit-parallel multiple pattern matching, in: Proc. 9th Int'l Conf. Parallel Processing and Applied Mathematics (PPAM'11), Part II, 2011, pp. 292–301.