Parallelizing Exact and Approximate String Matching via Inclusive Scan on a GPU

Yasuaki Mitani, Fumihiko Ino, *Member, IEEE,* and Kenichi Hagihara

Abstract—In this study, to substantially improve the runtimes of exact and approximate string matching algorithms, we propose a tribrid parallel method for bit-parallel algorithms such as the Shift-Or and Wu-Manber algorithms. Our underlying idea is to interpret bit-parallel algorithms as inclusive-scan operations, which allow these bit-parallel algorithms to run efficiently on a graphics processing unit (GPU); we achieve this speed-up here because inclusive-scan operations not only eliminate duplicate searches between threads but also realize a GPU-friendly memory access pattern that maximizes memory read/write throughput. To realize our ideas, we first define two binary operators and then present a proof regarding the associativity of these operators, which is necessary for the parallelization of the inclusive-scan operations. Finally, we integrate the inclusive-scan scheme into a previous segmentation-based scheme to maximize search throughput, identifying the best tradeoff point between synchronization cost and duplicate work. Through our experiments, we compared our proposed method with previous segmentation-based methods and indexing-based sequence aligners. For online string matching, our proposed method performed 6.7–16.7 times faster than previous methods, achieving a search throughput of up to 1.88 terabits per second (Tbps) on a GeForce GTX TITAN X GPU. We therefore conclude that our proposed method is quite effective for decreasing the runtimes of online string matching of short patterns.

Index Terms—String matching, bit-parallel algorithm, inclusive scan, Shift-Or algorithm, Wu-Manber algorithm, GPU.

1 Introduction

The problem of string matching is to find all locations at which pattern P of length m matches a substring of text, with T being of length n. Exact string matching finds all occurrences, if any, of pattern P in text T. Conversely, approximate string matching finds all strings whose edit distance from P is smaller than error k, i.e., the tolerable maximum edit distance. String matching is a fundamental problem that occurs in a wide range of practical applications, including computational biology [1], information retrieval [2], and network intrusion detection [3].

String matching algorithms have been intensively studied for over 40 years using several approaches, including dynamic programming, automata, bit parallelism, filtering, and indexing approaches [4]. Among these, automata-based algorithms have the best worst-case time, i.e., $O(n)$ or linear time, which is therefore the lower bound of this irregular problem [4]. These automata-based algorithms can be further classified into two groups according to their underlying automata, namely, deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). For DFAs, the Knuth-Morris-Pratt [5] and Aho-Corasick [6] algorithms have been widely used for exact string matching problems.

With respect to NFAs, a variety of bit-parallel approaches have been used in practice as fast algorithms for exact and approximate string matching problems [7]–[9]. In [7], [8], Baeza-Yates and Gonnet proposed the Shift-Or (SO) algorithm, which finds exact matching positions in $O(nm/w)$ time, where w is the size of a computer word. Although this complexity is not optimal, the SO algorithm is usually fast in practice, particularly for relatively short patterns that fit into a computer word (i.e., $m \leq w$); this inherent speed exists because the algorithm takes advantage of the intrinsic parallelism of bit operations inside a computer word. Exploiting this *bit parallelism* reduces the number of operations by a factor of up to w. The SO algorithm also has advantages in terms of data locality because its simple data structure regularizes memory access patterns. In [9], Wu and Manber extended this SO algorithm to solve the approximate string matching problem in $O(nm/wk)$ time.

In addition to bit parallelism, other data-parallel methods have been presented to achieve further acceleration on multicore CPUs [10], graphics processing units (GPUs) [11], and Xeon Phi coprocessors [12]. These hybrid methods exploit both bit parallelism and data parallelism by partitioning the text into smaller segments, which are then processed in parallel by running a bit-parallel algorithm for each segment. For example, GBPR [11] deployed this hybrid approach to realize a high search throughput of approximately 75 gigabits per second (Gbps) on a GeForce GTX 680 GPU, which was seven to 11 times faster than that of a multithreaded CPU implementation. Here search throughput ρ is given by $\rho = n/t$, where t is the GPU execution time spent for the search.

Nonetheless, using a segmentation-based scheme can result in a degradation of efficiency on a massively parallel machine because *halo regions* of size $m - 1$ are required to correctly find a match ending at the head of a segment, as depicted in Fig. 1a. These halo regions add substantial overhead because the characters within the halo region must be read twice, i.e., once for each of the adjacent segments. Given this duplicate searching, using a segmentation-based

- *Y. Mitani is with the Development Head Office, DWANGO Corporation, Ltd., 4-12-15 Ginza, Chuo-ku, Tokyo 104-0061, Japan.*
- *F. Ino and K. Hagihara are with the Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan. E-mail: ino@ist.osaka-u.ac.jp*

Manuscript received on 23 June, 2016; revised 27 Nov. 2016.

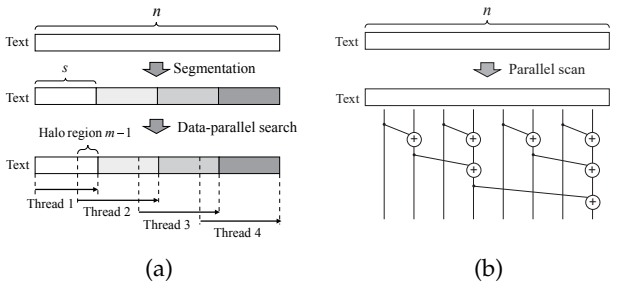


Fig. 1: Parallel string search using (a) a segmentation-based scheme and (b) an inclusive-scan scheme. The former processes a sequential search algorithm for each segment of size s; thus halo regions of size $m-1$ are required to find a potential match ending prematurely at the head of a segment. In contrast, the latter eliminates halo regions but requires global synchronization to obtain the entire scan.

scheme results in lower efficiency with strong scaling, where an increasing number of processing elements (i.e., segments) are applied to a fixed-size problem (i.e., text). Furthermore, parallel threads suffer from strided memory accesses when they simultaneously access different segments. Thus, the challenge of realizing a highly efficient solution remains for increasing the runtimes of bit-parallel algorithms on a massively parallel machine; more specifically, we need a low-cost emerging accelerator capable of exploiting fine-grained parallelism with millions of threads.

A prospective approach for eliminating halo regions is to interpret bit-parallel algorithms with scan primitives [13], which have been used to parallelize many key operations that seem inherently sequential [14], including sorting, stream compaction, and histogram computation. With respect to string comparison, Khajeh-Saeed *et al.* [15] presented an inclusive-scan-based approach for the Smith-Waterman algorithm [16], which can be regarded as a general form of the Wu-Manber (WM) algorithm [9] mentioned above; the Smith-Waterman algorithm computes the similarity scores of arbitrary regions of two strings according to a flexible scoring system (i.e., substitution matrix and gap penalties). Thus, scan operations can be efficiently computed on the GPU [14], [17] because this type of computation has high locality and regularity in terms of memory references. However, as far as we know, scan primitives have not been applied to bit-parallel algorithms, which have different computing strategies and data structures as compared to the Smith-Waterman algorithm.

In this paper, we therefore propose a tribrid parallel method that takes advantages of scan-based parallelization, segmentation-based parallelization, and bit-level parallelization. The scan scheme allows massively parallel threads to run the SO algorithm [7], [8] to find matching patterns without halo regions, as depicted in Fig. 1b; more specifically, a string matching problem can be interpreted as an inclusive-scan problem according to Blelloch's reduction [18]. To enable this interpretation, we find a *companion operator* [18] to rewrite the SO algorithm with a recurrence relation based on associative operators. This associativity allows the SO algorithm to be parallelized in the same manner as that of the parallel scan algorithm [19]. Thus, the automata can be correctly updated by reading characters at arbitrary positions of the text, meaning that there is no need to serially read characters from the head of a text segment. Because the inclusive-scan scheme requires global synchronization of threads, this scheme is cascaded with a segmentation-based scheme, which avoids synchronization, but requires halo regions. We also show that scan-based parallelization can be applied to the WM algorithm for approximate string matching. We implemented our proposed methods on a GPU compatible with the compute unified device architecture (CUDA) [20]. The developed implementation, which we called *cuShiftOr*, is available at http://www-hagi.ist.osaka-u.ac.jp/research/code/.

In addition to this introduction, the remainder of this paper is structured as follows. In Section 2, we introduce related studies regarding approaches to improving the runtimes of various string matching algorithms. In Section 3, we present an overview of bit-parallel algorithms, which forms the basis of our proposed method. In Section 4, we describe our proposed methods for exact and approximate string matching. Next, in Section 5, we present key implementation issues that must be solved to achieve CUDA-based acceleration. In Section 6, we show our experimental results obtained using the Maxwell GPU [21]. Finally, in Section 7, we conclude our paper and discuss avenues of future work.

2 Related Work

Several parallel implementations for accelerating bit-parallel algorithms on a GPU have been presented, some of which are detailed below. Previous methods typically parallelize a string search algorithm via a segmentation-based scheme. By contrast, cuShiftOr focuses on the associativity of search operations, which enables text to be read in an arbitrary order without halo regions. As such, cuShiftOr achieves an efficient memory access pattern for this irregular problem, yielding Tbps speeds for in-core search throughput of short patterns of up to 64 characters.

Lin *et al.* presented GBPR [11], which deploys a segmentation-based scheme to exploit data parallelism of the WM algorithm with optimization to the hierarchical memory architecture of the GPU. Given segment size s, text of length n is divided into $\lceil n/s \rceil$ overlapping segments to allow $\lceil n/s \rceil$ threads to independently run the sequential WM algorithm for their responsible segments. Furthermore, GBPR reduces the amount of off-chip memory access by utilizing on-chip shared memory [20] as a software-managed cache. The achieved search throughput of 75 Gbps was 2.8–104.8 times that of various state-of-the-art approaches [22]–[25]. However, this throughput can be increased if s-strided accesses from a *warp* [20] can be coalesced into a single access to a contiguous memory region. Here, a warp is a collection of 32 GPU threads that execute the same instruction at every clock cycle.

In [26], Tran *et al.* presented XBitPar, which allows a GPU to process the WM algorithm with a long pattern of up to 1024 characters. Their basic idea was to simulate a long computer word by letting each thread within a warp take a 32-bit word as an operand. This simulation is useful for dealing with long patterns, such as biological sequences, which usually cannot fit into a 32-bit computer word. By

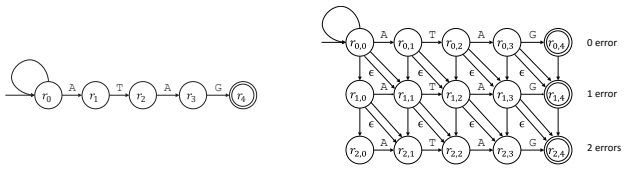
contrast, our proposed method, which deploys a 32/64-bit word, can be faster if a 1024-bit word is wasteful for the input pattern. This waste issue is not critical for multipattern matching problems because the automata states for multiple patterns can easily be packed into a computer word. Further, in [10], Kusudo *et al.* accelerated the SO algorithm for multipattern matching on a multicore CPU. Their implementation utilized advanced vector extensions (AVX) [27], which is an instruction set for single-instruction, multiple-data (SIMD) processing. In [24] and [28], similar approaches were implemented on a GPU. Given these previous studies, we conclude that scan-based parallelization can be applied to the family of bit-parallel algorithms designed for multipattern matching problems.

Similar to the WM algorithm, Baeza-Yates *et al.* [29] realized approximate string matching by packing automaton states in a different manner. The difference here was that they packed the automaton states along the diagonals instead of along the rows or columns, thereby achieving $O(n)$ search for short patterns, i.e., $mk = O(w)$. Their algorithm was implemented on an NVIDIA Tesla S1070 GPU by Onsjö *et al.* [25], who achieved a search throughput of 158 Mbps. Similar to XBitPar [26], they realized a solution that supported a long computer word via the warp-based search technique described above.

Dynamic programming has also been applied to string search problems. Using dynamic programming, a matrix called the dynamic programming matrix must be filled to find matching positions, with new values within the matrix building upon previous values of the matrix. In [30], Myers efficiently parallelized the computation of the dynamic programming matrix by considering the differences along columns instead of the values within the columns. He exploited bit parallelism in the matrix computation, achieving a runtime of $O(nm/w)$ for arbitrary k. Given its time complexity is independent of k here, the Myers algorithm is suitable for approximate string matching,

In [31], Langner parallelized the Myers algorithm using a GPU by simultaneously processing matrix elements on the same antidiagonal. Although this parallelization scheme is efficient for large n and m, efficiency degrades for short patterns because the degree of parallelism is limited to at most $\min(n, m/w)$ elements located on the same antidiagonal. A similar GPU-based dynamic programming implementation was presented by Man *et al.* [32], who achieved a search throughput of 80 Mbps on an NVIDIA GeForce GTX 580.

Lin *et al.* [33] parallelized the Aho-Corasick algorithm [6] on a GPU. Their implementation, called the parallel failureless Aho-Corasick (PFAC) algorithm, modifies the DFA to reduce the number of necessary warp divergences [20], which significantly decreases the efficiency of SIMD execution. A search throughput of up to 143 Gbps was achieved on a GeForce GTX 580 for 1998 exact patterns. This throughput was 14.7 times faster than that of an OpenMP-based CPU implementation; however, the PFAC algorithm suffers from load imbalance issues if too many partial matches are found during searching, as is evident by throughputs ranging from 11.0–218.7 Gbps. By contrast, bit-parallel algorithms avoid such load imbalance issues, opening the door to stable search throughputs regardless of the number of partial matches.



(a) (b)

Fig. 2: NFA examples used for bit-parallel algorithms. Given pattern ATAG, the SO algorithm [7], [8] generates (a) an NFA to be used for exact string matching, while (b) the WM algorithm [9] extends the NFA to perform approximate string matching with at most k errors (in this example, $k = 2$). Note that unlabeled transitions match any arbitrary character.

Finally, an important application category of string matching is biological sequence alignment. Many exact and approximate algorithms [34]–[38] have been proposed with indexing architectures. For short patterns, indexing-based approaches are usually computationally efficient versus indexless (i.e., brute-force) approaches; however, indexing-based approaches assume offline searching and require extra memory space to maintain indexes. We compare cuShiftOr with state-of-the-art aligners [34]–[38] later in Section 6.4.

3 Preliminaries: Bit-Parallel Algorithms

In the following subsections, we use a C-like notation for bitwise operators, i.e., let $\ll$, $\&$, $|$, and $\sim$ be the left shift, AND, OR, and NOT operators, respectively. Further, let $\mathbb{Z}$ and $\mathbb{F}$ be the sets of non-negative integers and a binary finite field, respectively. Also, let $\mathbb{F}^w$ be the set of binary bit-vectors of size w.

3.1 Shift-Or Algorithm for Exact Matching

In this subsection, we describe the SO algorithm for exact string matching. Let Σ be an alphabet of characters. Let $T = t_1 t_2 \cdots t_n$ be the text of length $n \in \mathbb{Z}$ and let $P = p_1 p_2 \cdots p_m$ be the pattern of length $m \in \mathbb{Z}$, where $t_i \in \Sigma$ $(1 \leq i \leq n)$ and $p_j \in \Sigma$ $(1 \leq j \leq m)$ are the i-th and j-th characters of the text and pattern, respectively. The SO algorithm [7], [8] accelerates string matching by running m string comparators in parallel, each reading the same text position concurrently but with a different starting position. Each comparator simulates an NFA to express a matching state; the current state of the NFA is updated by reading a character from the text. Figure 2a shows an example of the NFA generated for pattern ATAG. As shown in the figure, state r_i indicates that the first i characters of the pattern exactly match the last i characters of the text, where $1 \leq i \leq m$. Consequently, an exact match ending at t_j can be detected if the NFA reaches accepting state r_m after reading t_j, where $1 \leq j \leq n$.

Bit parallelism can be exploited here by packing the current states of m NFAs into bit-vector $\mathbf{R} \in \mathbb{F}^m$ of size m such that $\mathbf{R}$ contains information on all matches of prefixes of pattern P. Let $\mathbf{R}_j \in \mathbb{F}^m$ be the value of bit-vector $\mathbf{R}$ after t_j has been read. $\mathbf{R}_j$ is defined as $\mathbf{R}_j[i] = 0$ if

$p_1p_2\cdots p_i = t_{j-i+1}t_{j-i+2}\cdots t_j$; otherwise, $\mathbf{R}_j[i] = 1$. Bit-vector $\mathbf{R}_j$ can then be computed according to a first-order recurrence relation

$$\mathbf{R}_j = \begin{cases} 1^m, & \text{if } j = 0, \\ (\mathbf{R}_{j-1} \ll 1) \mid \mathbf{M}_j, & \text{if } j < 0 \leq n, \end{cases} \tag{1}$$

where 1^m is an all-one vector of size m and $\mathbf{M}_j \in \mathbb{F}^m$ is a bit-vector that depends on t_j: for all $1 \leq i \leq m$, $\mathbf{M}_j[i] = 0$, if $t_j = p_i$; otherwise, $\mathbf{M}_j[i] = 1$. In other words, $\mathbf{M}_j$ is a mask that can be stored in a table indexed by alphabet Σ. The SO algorithm can be rapidly processed if a bit-vector fits within a computer word (i.e., $m \leq w$).

3.2 Wu-Manber Algorithm for Approximate Matching

The basic idea behind the WM algorithm for approximate string matching is to extend the NFA of the SO algorithm to allow transitions between states associated with error distance d ($0 \leq d \leq k$). Let $r_{d,i}$ be the state in which the first i characters of the pattern match the last i characters of the text with d errors. In this way, the extended NFA considers all possible matches with up to k errors. Figure 2b illustrates an NFA example for pattern ATAG with $k = 2$.

Similar to the SO algorithm, the WM algorithm [9] exploits bit parallelism by packing the current states of m NFAs, i.e., bit-vector $\mathbf{R} \in \mathbb{F}^m$ of size m contains information on all matches of prefixes of pattern P with up to k errors. Let $\mathbf{R}_{d,j} \in \mathbb{F}^m$ be the value of the bit-vector after t_j has been read. For approximate matching, we must consider transitions not only with matches but also from insertions, deletions, or substitutions, which we express as

$$\mathbf{R}_{d,j} = \begin{cases} 1^m, & \text{if } d = 0, j = 0, \\ (\mathbf{R}_{d,j-1} \ll 1) \mid \mathbf{M}_j, & \text{if } d = 0, 0 < j \leq n, \\ 1^{m-d}0^d, & \text{if } 0 < d \leq k, j = 0, \\ ((\mathbf{R}_{d,j-1} \ll 1) \mid \mathbf{M}_j) & \\ \quad \& \ \mathbf{R}_{d-1,j-1} & \\ \quad \& \ (\mathbf{R}_{d-1,j-1} \ll 1) & \\ \quad \& \ (\mathbf{R}_{d-1,j} \ll 1), & \text{if } 0 < d \leq k, 0 < j \leq n. \end{cases} \tag{2}$$

3.3 Properties of Fundamental Operators

To facilitate the understanding of our proposed method, we describe properties of fundamental bitwise operations via the lemmas below.

Lemma 1. *The bitwise left shift operator $\ll$: $\mathbb{F}^w \times \mathbb{Z} \to \mathbb{F}^w$ satisfies $(\mathbf{x} \ll u) \ll v = \mathbf{x} \ll (u + v)$, where $\mathbf{x} \in \mathbb{F}^w$ is a bit-vector of size w and $u, v \in \mathbb{Z}$ are non-negative integers.*

Lemma 2. *The bitwise OR operator $|$: $\mathbb{F}^w \times \mathbb{F}^w \to \mathbb{F}^w$ is associative.*

Lemma 3. *$\ll$ is right-distributive over $|$.*

The proofs of the above lemmas are presented in the supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.xxxx.xx.

4 Proposed Parallel Method

The fundamental goal of our proposed method is to introduce associative binary operator $\bullet$ to translate Eqs. (1) and (2) into recurrence relation

$$R_j = \begin{cases} a_0, & \text{if } j = 0, \\ R_{j-1} \bullet a_j, & \text{if } j > 0. \end{cases} \tag{3}$$

To achieve this, we find a companion operator [18] for Eq. (1) and show a proof based on Blelloch's reduction [18]; by using the companion operator, a first-order recurrence with a semiassociative operator [18] and an associative operator can be reduced into Eq. (3). We also extend this idea for Eq. (2), i.e., a recurrence with three different operators, $\ll$, $|$, and $\&$, by defining an associative operator and a semiassociative operator required for Blelloch's reduction.

4.1 Exact Matching

For exact string matching problems, we introduce bitwise operator $\dagger$ as the key operator $\bullet$.

Definition 1. *Bitwise operator $\dagger : (\mathbb{Z} \times \mathbb{F}^w) \times (\mathbb{Z} \times \mathbb{F}^w) \to \mathbb{Z} \times \mathbb{F}^w$ for pairs of a non-negative integer and a bit-vector of size w is defined as*

$$\langle u_1, \mathbf{x_1} \rangle \dagger \langle u_2, \mathbf{x_2} \rangle \stackrel{\text{def}}{=} \langle u_1 + u_2, (\mathbf{x_1} \ll u_2) \mid \mathbf{x_2} \rangle, \tag{4}$$

where $u_1, u_2 \in \mathbb{Z}$ and $\mathbf{x_1}, \mathbf{x_2} \in \mathbb{F}^w$. The operator $+$ here is the companion operator of the semiassociative operator $\ll$.

Using associative operator $\dagger$, an exact string matching problem can be solved as an inclusive-scan problem. Algorithm 1 shows our proposed method for exact matching, where we use $\sum_{l=0}^{j} A_l$ to denote a summation over $\dagger$: $\sum_{l=0}^{j} A_l = A_0 \dagger A_1 \dagger \cdots \dagger A_j$, where $A_0, A_1, \ldots, A_l \in \mathbb{Z} \times \mathbb{F}^m$. The basic idea behind operator $\dagger$ is that it pairs up a mask with an integer to correctly compute accumulated mask $R_j \in \mathbb{Z} \times \mathbb{F}^m$, thus accumulated mask R_j from Algorithm 1 considers all possible states that can be reached after T_j has been read. Consequently, the bit-vector can be updated by an inclusive-scan scheme as presented in lines 4–6 of Algorithm 1. Note that this algorithm assumes $m \leq 64$ $(= w)$ to achieve acceleration on a GPU.

Algorithm 1 SO exact matching with parallel inclusive-scan

Input: Text T of n characters and pattern P of m characters
Output: Bit sequence X of size n that indicates matching positions

1: Initialize the masks $\mathbf{M}_1, \mathbf{M}_2, \cdots, \mathbf{M}_n$ according to T and P
2: $A_0 \leftarrow \langle 0, 1^m \rangle$
3: $A_j \leftarrow \langle 1, \mathbf{M}_j \rangle$, for all $1 \leq j \leq n$
4: **for** $j \leftarrow 0$ to n **do in parallel** ▷ Parallel inclusive-scan
5: $\quad R_j \leftarrow \sum_{l=0}^{j} A_l$ ▷ Summation over $\dagger$
6: **end for**
7: $\langle u_j, \mathbf{x}_j \rangle \leftarrow R_j$, for all $0 \leq j \leq n$
8: $X[j] \leftarrow \mathbf{x}_j[m]$, for all $1 \leq j \leq n$
9: **return** X

Next, we show a proof of the correctness of Algorithm 1. First, we show that $\dagger$ is an associative operator required for parallelizing lines 4–6 of Algorithm 1.

Theorem 1. $\dagger$ *is associative.*

Proof. The proof here is given by Lemmas 1, 2, and 3. Let $\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3} \in \mathbb{F}^w$ and $u_1, u_2, u_3 \in \mathbb{Z}$. Then,

$$\begin{aligned}
&(\langle u_1, \mathbf{x_1}\rangle \dagger \langle u_2, \mathbf{x_2}\rangle) \dagger \langle u_3, \mathbf{x_3}\rangle \\
&= \langle u_1{+}u_2, (\mathbf{x_1} \ll u_2) \mid \mathbf{x_2}\rangle \dagger \langle u_3, \mathbf{x_3}\rangle \\
&= \langle (u_1{+}u_2) + u_3, (((\mathbf{x_1} \ll u_2) \mid \mathbf{x_2}) \ll u_3) \mid \mathbf{x_3}\rangle \\
&= \langle u_1{+}u_2{+}u_3, (((\mathbf{x_1} \ll u_2) \ll u_3) \mid (\mathbf{x_2} \ll u_3)) \mid \mathbf{x_3}\rangle \\
&= \langle u_1{+}u_2{+}u_3, (\mathbf{x_1} \ll (u_2{+}u_3)) \mid (\mathbf{x_2} \ll u_3) \mid \mathbf{x_3}\rangle. \quad (5)
\end{aligned}$$

Conversely,

$$\begin{aligned}
&\langle u_1, \mathbf{x_1}\rangle \dagger (\langle u_2, \mathbf{x_2}\rangle \dagger \langle u_3, \mathbf{x_3}\rangle) \\
&= \langle u_1, \mathbf{x_1}\rangle \dagger \langle u_2{+}u_3, (\mathbf{x_2} \ll u_3) \mid \mathbf{x_3}\rangle \\
&= \langle u_1 + (u_2{+}u_3), (\mathbf{x_1} \ll (u_2{+}u_3)) \mid ((\mathbf{x_2} \ll u_3) \mid \mathbf{x_3})\rangle \\
&= \langle u_1{+}u_2{+}u_3, (\mathbf{x_1} \ll (u_2{+}u_3)) \mid (\mathbf{x_2} \ll u_3) \mid \mathbf{x_3}\rangle. \quad (6)
\end{aligned}$$

Thus, Eq. (5) is equivalent to Eq. (6), and thereby $\dagger$ is associative. □

Next, we show that Algorithm 1 produces the same results of the SO algorithm.

Theorem 2. *Let* $\langle u_0, \mathbf{x}_0\rangle, \langle u_1, \mathbf{x}_1\rangle, \ldots, \langle u_n, \mathbf{x}_n\rangle \in \mathbb{Z} \times \mathbb{F}^m$ *be the inclusive scans of pairs obtained at line 7 of Algorithm 1, where* $m \in \mathbb{Z}$*. Then, for all* $0 \le j \le n$, $\mathbf{x}_j = \mathbf{R}_j$*, where* $\mathbf{R}_j \in \mathbb{F}^m$ *is the* j*-th bit-vector obtained by Eq. (1).*

Proof. According to Algorithm 1, j-th inclusive scan $\langle u_j, \mathbf{x}_j\rangle$, where $0 \le j \le n$, can be expressed as recurrence relation

$$\langle u_j, \mathbf{x}_j\rangle = \begin{cases} \langle 0, 1^m\rangle, & \text{if } j = 0, \\ \langle u_j, \mathbf{x}_{j-1}\rangle \dagger \langle 1, \mathbf{M}_j\rangle, & \text{if } 0 < j \le n. \end{cases} \quad (7)$$

Conversely, Eq. (4) gives $\langle u_j, \mathbf{x}_{j-1}\rangle \dagger \langle 1, \mathbf{M}_j\rangle = \langle u_j + 1, (\mathbf{x}_{j-1} \ll 1) \mid \mathbf{M}_j\rangle$, meaning that Eq. (1) is part of Eq. (7); the second component of the pairs in Eq. (7) is equal to Eq. (1); hence, $\mathbf{x}_j = \mathbf{R}_j$, for all $0 \le j \le n$. □

4.2 Approximate Matching

We cannot directly apply our exact matching approach to an approximation problem because Eq. (2) consists of $\mid$ and $\&$, which corrupt the associative property. As an example of this, $(1\ \&\ 0) \mid 1 \neq 1\ \&\ (0 \mid 1)$. To address this issue, we introduce three additional bitwise operators $\ll_2$, $\uplus$, and $\ddagger$, where $\ddagger$ is the target associative operator $\bullet$ constructed using $\ll_2$ and $\uplus$. These operators are defined below.

Definition 2. *Bitwise left shift operator* $\ll_2 : \mathbb{G}^w \times \mathbb{Z} \to \mathbb{G}^w$ *for pairs of bit vectors, where* $\mathbb{G}^w \stackrel{\text{def}}{=} \mathbb{F}^w \times \mathbb{F}^w$*, is defined as*

$$\langle \mathbf{x}, \mathbf{y}\rangle \ll_2 u \stackrel{\text{def}}{=} \langle \mathbf{x} \ll u, \mathbf{y} \ll u\rangle, \quad (8)$$

where $\mathbf{x}, \mathbf{y} \in \mathbb{F}^w$ *and* $u \in \mathbb{Z}$*. The operator* $\ll_2$ *here is the semiassociative operator required for Blelloch's reduction.*

Definition 3. *Bitwise operator* $\uplus : \mathbb{G}^w \times \mathbb{G}^w \to \mathbb{G}^w$ *is defined as*

$$\langle \mathbf{x_1}, \mathbf{y_1}\rangle \uplus \langle \mathbf{x_2}, \mathbf{y_2}\rangle \stackrel{\text{def}}{=} \langle \mathbf{x_1} \,|\, \mathbf{x_2},\ (\mathbf{y_1} \,\&\, {\sim}\mathbf{x_2}) \mid \mathbf{y_2}\rangle, \quad (9)$$

where $\mathbf{x_1}, \mathbf{x_2}, \mathbf{y_1}, \mathbf{y_2} \in \mathbb{F}^w$*. The operator* $\uplus$ *here is the associative operator required for Blelloch's reduction.*

Definition 4. *Bitwise operator* $\ddagger : (\mathbb{Z} \times \mathbb{G}^w) \times (\mathbb{Z} \times \mathbb{G}^w) \to \mathbb{Z} \times \mathbb{G}^w$ *is defined as*

$$\begin{aligned}
&\langle u_1, \langle \mathbf{x_1}, \mathbf{y_1}\rangle\rangle \ddagger \langle u_2, \langle \mathbf{x_2}, \mathbf{y_2}\rangle\rangle \stackrel{\text{def}}{=} \\
&\qquad \langle u_1{+}u_2,\ (\langle \mathbf{x_1}, \mathbf{y_1}\rangle \ll_2 u_2) \uplus \langle \mathbf{x_2}, \mathbf{y_2}\rangle\rangle, \quad (10)
\end{aligned}$$

where $u_1, u_2 \in \mathbb{Z}$*, and* $\mathbf{x_1}, \mathbf{x_2}, \mathbf{y_1}, \mathbf{y_2} \in \mathbb{F}^w$*. The operator* $+$ *here is the companion operator of the semiassociative operator* $\ll_2$*.*

Using operator $\ddagger$, the approximate string matching problem can be solved via an inclusive-scan scheme, as shown in Algorithm 2.

Algorithm 2 WM approximate matching with parallel inclusive-scan

Input: Text T of n characters and pattern P of m characters
Output: Bit sequence Y of size n that indicates approximate matching positions with at most k errors

1: Initialize the masks $\mathbf{M}_1, \mathbf{M}_2, \cdots, \mathbf{M}_n$ according to T and P
2: Compute $R_{0,j}$, for all $0 \le j \le n$ using lines 2–6 of Algorithm 1
3: $\langle u_{0,j}, \mathbf{y}_{0,j}\rangle \leftarrow R_{0,j}$, for all $0 \le j \le n$
4: **for** $d \leftarrow 1$ to k **do**
5: $\quad \mathbf{N}_j \leftarrow \mathbf{y}_{d-1,j-1}\ \&\ (\mathbf{y}_{d-1,j-1} \ll 1)\ \&\ (\mathbf{y}_{d-1,j} \ll 1)$, for all $1 \le j \le n$
6: $\quad A_{d,0} \leftarrow \langle 0, \langle 0^m, 1^{m-d}0^d\rangle\rangle$
7: $\quad A_{d,j} \leftarrow \langle 1, \langle {\sim}\mathbf{N}_j, (\mathbf{M}_j \,\&\, \mathbf{N}_j)\rangle\rangle$, for all $1 \le j \le n$
8: $\quad$ **for** $j \leftarrow 0$ to n **do in parallel** ▷ Parallel inclusive-scan
9: $\quad\quad R_{d,j} \leftarrow \sum_{l=0}^{j} A_{d,l}$ ▷ Summation over $\ddagger$
10: $\quad$ **end for**
11: $\quad \langle u_{d,j}, \langle \mathbf{x}_{d,j}, \mathbf{y}_{d,j}\rangle\rangle \leftarrow R_{d,j}$, for all $0 \le j \le n$
12: **end for**
13: $Y[j] \leftarrow \mathbf{y}_{k,j}[m]$, for all $1 \le j \le n$
14: **return** Y

Next, we show a proof of the correctness of Algorithm 2. Equations (1) and (2) indicate that $\mathbf{R}_{0,j}$, where $0 \le j \le n$, can be obtained by processing the SO algorithm, i.e., lines 2 and 3 of Algorithm 2 correctly produce exact matching results for $d = 0$. Regarding remaining lines 4–12, which consider cases of $0 < d \le k$, we show below that Algorithm 2 produces the same results as the WM algorithm.

First, we describe some of the properties of the newly introduced operators. Note here that the associativity of $\ddagger$ can be shown in a similar manner as that of $\dagger$, because Eq. (4) for $\dagger$ can be mapped to Eq. (10) for $\ddagger$ by substituting $\ll$ and $\mid$ for $\ll_2$ and $\uplus$, respectively. Accordingly, we show some fundamental properties of $\ll_2$ and $\uplus$ below.

Lemma 4. $\ll_2$ *has the property*

$$(\langle \mathbf{x}, \mathbf{y}\rangle \ll_2 u) \ll_2 v = \langle \mathbf{x}, \mathbf{y}\rangle \ll_2 (u + v),$$

where $\mathbf{x}, \mathbf{y} \in \mathbb{F}^w$ *and* $u, v \in \mathbb{Z}$*.*

Proof. The proof here can easily be shown in the same manner as that of Lemma 1. □

Lemma 5. $\uplus$ *is associative.*

Proof. When $w = 1$, $(\langle x_1, y_1\rangle \uplus \langle x_2, y_2\rangle) \uplus \langle x_3, y_3\rangle = \langle x_1, y_1\rangle \uplus (\langle x_2, y_2\rangle \uplus \langle x_3, y_3\rangle)$, for all $x_1, x_2, x_3, y_1, y_2, y_3 \in \mathbb{F}$.

This equivalence can easily be confirmed as presented in the supplementary material. This associativity also exists when $w > 1$ because arbitrary bits of the output vectors can be computed independently of one another. □

Lemma 6. *$\ll_2$ is right-distributive over $\uplus$.*

Proof. The proof here can easily be shown in the same manner as that of Lemma 3. □

Second, we show that $\ddagger$ is an associative operator, required for parallelizing lines 8–10 of Algorithm 2.

Theorem 3. *$\ddagger$ is associative.*

Proof. The proof here can be shown in the same manner as that of Theorem 1 by using Lemmas 4, 5, and 6. □

Third, we show that arbitrary inclusive-scans obtained with &, |, and/or $\ll$ can be expressed with $\uplus$ and $\ll_2$ according to a rewrite rule. To facilitate this process, we first consider the first two operators, & and |.

Theorem 4. *Let $\mathbf{a}_0, \mathbf{a}_1, \cdots, \mathbf{a}_n \in \mathbb{F}^m$ be a sequence of n bit-vectors of size $m \in \mathbb{Z}$. Consider inclusive scans of these bit-vectors summed up with & and/or | as*

$$\mathbf{R}_j = \begin{cases} \mathbf{a_0}, & \text{if } j = 0, \\ \mathbf{R}_{j-1}\ op_j\ \mathbf{a}_j, & \text{if } 0 < j \leq n, \end{cases} \tag{11}$$

where $op_j \in \{\&, |\}$ and $\mathbf{a}_j, \mathbf{R}_j \in \mathbb{F}^m$. Next, let $\langle \mathbf{x}_j, \mathbf{y}_j \rangle$, where $0 \leq j \leq n$, be a pair of bit-vectors given by recurrence relation

$$\langle \mathbf{x}_j, \mathbf{y}_j \rangle = \begin{cases} \langle 0^m, \mathbf{a}_0 \rangle, & \text{if } j = 0, \\ \langle \mathbf{x}_{j-1}, \mathbf{y}_{j-1} \rangle \uplus \langle 0^m, \mathbf{a}_j \rangle, & \text{if } 0 < j \leq n, op_j{=}|, \\ \langle \mathbf{x}_{j-1}, \mathbf{y}_{j-1} \rangle \uplus \langle {\sim}\mathbf{a}_j, 0^m \rangle, & \text{if } 0 < j \leq n, op_j{=}\&, \end{cases} \tag{12}$$

where $\mathbf{x}_j, \mathbf{y}_j \in \mathbb{F}^m$. Then, there is a rewrite rule that maps Eq. (11) to Eq. (12), i.e., $\mathbf{R}_j = \mathbf{y}_j$, for all $0 \leq j \leq n$.

Proof. We show a proof for the case of $m = 1$ by mathematical induction on n. When $n = 1$, the statement obviously holds. Assume that the statement holds for $n = k'\ (> 1)$: for all $0 \leq j \leq k'$, $R_j = y_j$, where $R_j, y_j \in \mathbb{F}$. Let $n = k' + 1$. According to Eq. (11),

$$R_{k'+1} = \begin{cases} R_{k'}, & \text{if } a_{k'} = 0, op_{k'}{=}|, \\ 0, & \text{if } a_{k'} = 0, op_{k'}{=}\&, \\ 1, & \text{if } a_{k'} = 1, op_{k'}{=}|, \\ R_{k'}, & \text{if } a_{k'} = 1, op_{k'}{=}\&\ . \end{cases} \tag{13}$$

Equation (12) indicates that

$$\langle x_{k'+1}, y_{k'+1} \rangle = \begin{cases} \langle x_{k'}, y_{k'} \rangle, & \text{if } a_{k'} = 0, op_{k'}{=}|, \\ \langle 1, 0 \rangle, & \text{if } a_{k'} = 0, op_{k'}{=}\&, \\ \langle x_{k'}, 1 \rangle, & \text{if } a_{k'} = 1, op_{k'}{=}|, \\ \langle x_{k'}, y_{k'} \rangle, & \text{if } a_{k'} = 1, op_{k'}{=}\&\ . \end{cases} \tag{14}$$

Hence, $R_j = y_j$, for all $0 \leq j \leq k' + 1$; the statement holds for $m = 1$. The statement also holds for $m > 1$ because arbitrary bits of the output bit-vectors can be computed independently of one another. □

We next extend Theorem 4 to handle inclusive scans obtained with $\ll$, i.e., in addition to & and |.

Theorem 5. *Let $\mathbf{a}_0, \mathbf{a}_1, \cdots, \mathbf{a}_n \in \mathbb{F} \bigcup \mathbb{F}^m$ be a sequence of $n + 1$ bit-vectors of size 1 and $m \in \mathbb{Z}$. Consider inclusive scans $\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_n$ of these bit-vectors that are summed up using Eq. (11) with an extended set of operators, $op_j \in \{\&, |, \ll\}$. Let $\langle \mathbf{x}_j, \mathbf{y}_j \rangle$ be a pair of bit-vectors given by recurrence relation*

$$\langle \mathbf{x}_j, \mathbf{y}_j \rangle = \begin{cases} \langle 0^m, \mathbf{a}_0 \rangle, & \text{if } j = 0, \\ \langle \mathbf{x}_{j-1}, \mathbf{y}_{j-1} \rangle \uplus \langle 0^m, \mathbf{a}_j \rangle, & \text{if } 0 < j \leq n, op_j{=}|, \\ \langle \mathbf{x}_{j-1}, \mathbf{y}_{j-1} \rangle \uplus \langle {\sim}\mathbf{a}_j, 0^m \rangle, & \text{if } 0 < j \leq n, op_j{=}\&, \\ \langle \mathbf{x}_{j-1}, \mathbf{y}_{j-1} \rangle \ll_2 a_j, & \text{if } 0 < j \leq n, op_j{=}\ll, \end{cases} \tag{15}$$

where $\mathbf{x}_j, \mathbf{y}_j \in \mathbb{F}^m$. Then, $\mathbf{R}_j = \mathbf{y}_j$, for all $0 \leq j \leq n$.

Proof. Let l be the number of $\ll$ operators that appear in $\mathbf{R}_n = \mathbf{a}_0\ op_1\ \mathbf{a}_1\ op_2\ \cdots\ op_n\ \mathbf{a}_n$. The proof is given by mathematical induction on l. When $l = 0$ (i.e., $\mathbf{R}_n$ does not have $\ll$), the statement holds from Theorem 4. Assume that the statement holds for $l = k'\ (> 0)$ and let $l = k' + 1$. Let J be the largest j such that $op_J{=}\ll$. Then, the statement for $l = k' + 1$ can be proved in the following three steps.

- For all $1 \leq j < J$, $\mathbf{R}_j = \mathbf{y}_j$ holds because $\mathbf{R}_{J-1}$ includes $k' \ll$ operators.
- For $j = J$, $\mathbf{R}_j = \mathbf{y}_j$ because (1) $op_J{=}\ll$ indicates $\mathbf{R}_J = \mathbf{R}_{J-1} \ll a_J$ and (2) Eq. (15) gives $\langle \mathbf{x}_J, \mathbf{y}_J \rangle = \langle \mathbf{x}_{J-1}, \mathbf{y}_{J-1} \rangle \ll_2 a_J = \langle \mathbf{x}_{J-1} \ll a_J, \mathbf{y}_{J-1} \ll a_J \rangle$.
- For all $J < j \leq n$, $op_j {\neq} \ll$ holds; that is, $op_j \in \{\&, |\}$ holds. Hence, Theorem 4 can be applied to $\mathbf{R}_n = \mathbf{R}_J\ op_{J+1}\ \mathbf{a}_{J+1}\ op_{J+2}\ \cdots\ op_n\ \mathbf{a}_n$, which indicates $\mathbf{R}_j = \mathbf{y}_j$, for all $J < j \leq n$.

Thus, the statement holds for arbitrary $0 \leq l \leq n$, i.e., $\mathbf{R}_j = \mathbf{y}_j$ holds for all $0 \leq j \leq n$. □

Theorem 6. *For all $0 < d \leq k$, $\mathbf{R}_{d,j}$ of Eq. (2) is equivalent to $\mathbf{y}_{d,j}$ given by*

$$\langle \mathbf{x}_{d,j},\ \mathbf{y}_{d,j} \rangle = \begin{cases} \langle 0^m,\ 1^{m-d}0^d \rangle, & \text{if } j = 0, \\ (\langle \mathbf{x}_{d,j-1}, \mathbf{y}_{d,j-1} \rangle \ll_2 1) \\ \quad \uplus \langle {\sim}\mathbf{N}_{d,j},\ \mathbf{M}_j \,\&\, \mathbf{N}_{d,j} \rangle, & \text{if } 0 < j \leq n, \end{cases} \tag{16}$$

where $\mathbf{N}_{d,j} = \mathbf{y}_{d-1,j-1}\ \&\ (\mathbf{y}_{d-1,j-1} \ll 1)\ \&\ (\mathbf{y}_{d-1,j} \ll 1)$.

Proof. When $j = 0$, it obviously holds. When $0 < j \leq n$, Theorem 5 translates Eq. (2) into a recurrence relation with $\uplus$ and $\ll_2$, i.e., when $0 < d \leq k$ and $0 < j \leq n$, Eq. (2) is a recurrence relation with $op_1 = \ll$, $op_2 = |$, and $op_3 = \&$. Therefore, applying Theorem 5 to Eq. (2) produces recurrence relation

$$\begin{aligned} &\langle \mathbf{x}_{d,j},\ \mathbf{y}_{d,j} \rangle \\ &= (\langle \mathbf{x}_{d,j-1},\ \mathbf{y}_{d,j-1} \rangle \ll_2 1) \uplus \langle 0^m, \mathbf{M}_j \rangle \uplus \langle {\sim}\mathbf{N}_{d,j}, 0^m) \rangle \\ &= (\langle \mathbf{x}_{d,j-1},\ \mathbf{y}_{d,j-1} \rangle \ll_2 1) \uplus \langle {\sim}\mathbf{N}_{d,j},\ \mathbf{M}_j \,\&\, \mathbf{N}_{d,j} \rangle. \end{aligned}$$

□

Thus, Eq. (2) can be mapped to Eq. (16) using $\uplus$ and $\ll_2$ instead of | and $\ll$.

Finally, we next show that Algorithm 2 produces the same results as that of the WM algorithm.

Theorem 7. *Let $\langle u_{d,j}, \langle \mathbf{x}_{d,j}, \mathbf{y}_{d,j} \rangle\rangle$ be an inclusive-scan obtained at line 11 of Algorithm 2. Then, for all $0 \leq j \leq n$ and $0 \leq d \leq k$, $\mathbf{y}_{d,j} = \mathbf{R}_{d,j}$, where $\mathbf{R}_{d,j} \in \mathbb{F}^m$ is the bit-vector obtained by Eq. (2).*

Proof. When $d = 0$ (i.e., exact matching), Theorem 2 provides the proof. We therefore consider the case in which $d > 0$. According to Algorithm 2, $\langle u_{d,j}, \langle \mathbf{x}_{d,j}, \mathbf{y}_{d,j} \rangle\rangle$, where $0 \leq j \leq n$ and $0 < d \leq k$, is given by recurrence relation

$$\langle u_{d,j}, \langle \mathbf{x}_{d,j}, \mathbf{y}_{d,j} \rangle\rangle = \begin{cases} \langle 0, \langle 0^m, 1^{m-d}0^d \rangle\rangle, & \text{if } j = 0, \\ \langle u_{d,j-1}, \langle \mathbf{x}_{d,j-1}, \mathbf{y}_{d,j-1} \rangle\rangle & \\ \quad \ddagger \langle 1, \langle \sim\mathbf{M}_{d,j}, (\mathbf{M}_j \,\&\, \mathbf{N}_{d,j}) \rangle\rangle, & \text{if } 0 < j \leq n. \end{cases}$$

Definition 4 rewrites the second components of this equation into Eq. (16), meaning that $\mathbf{y}_{d,j} = \mathbf{R}_{d,j}$ for all $0 < d \leq k$ and $0 \leq j \leq n$. □

4.3 Reducing the Time Complexity

The time complexity of the parallel inclusive-scan algorithm [19] is $O(n \log n)$, where n is the number of elements to be summed. Therefore, our proposed parallel method incurs a parallelization overhead versus the original SO algorithm, which finds matching positions in $O(nm/w)$. To reduce this overhead, we adopt an optimization strategy that reduces the time complexity to $O(nm/w \log m)$.

We consider here the SO algorithm, but the strategy presented below can easily be applied to the WM algorithm by interpreting the associative operator † as ‡ for approximate string matching.

According to Eq. (1), we have

$$\mathbf{R}_j = \mathbf{M}_j \mid (\mathbf{M}_{j-1} \ll 1) \mid \cdots \mid (\mathbf{M}_{j-m+1} \ll (m-1)) \mid (\mathbf{M}_{j-m+2} \ll (m-2)) \mid \cdots \mid (\mathbf{M}_0 \ll j),$$

where $0 < j \leq n$. Because $\mathbf{x} \ll k = 0^m$ holds for all $\mathbf{x} \in \mathbb{F}^m$ and $k \geq m$, Eq. (1) can be rewritten as

$$\mathbf{R}_j = \mathbf{M}_j \mid (\mathbf{M}_{j-1} \ll 1) \mid \ldots \mid (\mathbf{M}_{j-m+1} \ll (m-1)). \quad (17)$$

This equation specifies that exact inclusive-scans are not required to obtain results of the SO algorithm; in other words, as the original SO algorithm implies, only the latest $m-1$ characters in the text are required for detecting an exact string match ending at the current position. Consequently, line 5 of Algorithm 1, i.e., $R_j \leftarrow \sum_{l=0}^{j} A_l$, can be replaced with $R_j \leftarrow \sum_{l=j-m+1}^{j} A_l$ to reduce the time complexity without loss of generality. In this case, n elements are summed via $\log m$ steps, and thereby the time complexity is reduced from $O(n \log n)$ to $O(n \log m)$ if $m = O(w)$.

4.4 Reducing Global Synchronization

A major drawback of the parallel scan algorithm [19] is that it synchronizes all threads, requiring each to walk through $\log m$ steps. Conversely, the CUDA programming model prohibits global synchronization within a kernel [20] (i.e., GPU program); in other words, a kernel completion implicitly synchronizes all GPU threads. Therefore, the kernel must be invoked $\log m$ times to run Algorithm 1 on the GPU, thus search throughput suffers from the increased amount of off-chip memory access. Data in register files and shared memory have lifetimes of the corresponding thread block [20], thus threads must fetch data from off-chip memory at the beginning of every kernel invocation.

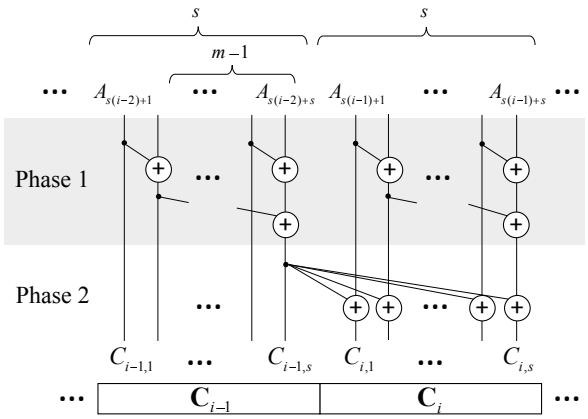


Fig. 3: Overview of the disjoint version of our tribrid method. In the first phase, we perform the parallel inclusive-scan algorithm within each disjoint segment of s ($\geq m$) elements. Next, each segment $\mathbf{C}_i$ is updated by summing up the last element $C_{i-1,s}$ of its left neighbor $\mathbf{C}_{i-1}$, thus taking $m-1$ preceding elements into consideration. By contrast, the overlapping version can be processed within a single phase, in which parallel inclusive-scans for contiguous $s+m-1$ elements are processed independently.

To reduce this overhead, we adopt a tribrid strategy that integrates the inclusive-scan scheme into a segmentation-based scheme. The following two segmented variations adapt the tribrid strategy to CUDA: (1) an overlapping version that requires halo regions and (2) a disjoint version that eliminates halo regions but requires one global synchronization instead of $\log m$ global synchronizations. Both versions reduce synchronization costs by replacing global synchronization with local synchronization. The goal here is to limit synchronization to within a small segment. Similar to the segmentation-based scheme illustrated in Fig. 1a, the overlapping version allows different segments to be processed independently by running the parallel scan algorithm for $s+m-1$ contiguous elements. By contrast, the disjoint version runs the parallel scan algorithm for each disjoint segment of s elements but requires an additional phase to correctly handle the segmentation border, as illustrated in Fig. 3. Note that Theorem 2 realizes this disjoint version; thus, there is a tradeoff between the synchronization cost and halo region size (i.e., duplicate searching).

Algorithm 3 shows the disjoint version of our proposed tribrid method, namely SO exact string matching with disjoint segmented parallel inclusive-scan. First, our proposed method assumes that the text is partitioned into $\lceil n/s \rceil$ segments, where s ($\geq m$) indicates segment size. Second, our method independently executes the parallel inclusive-scan algorithm for each segment. Thus, the i-th segment, where $1 \leq i \leq \lceil n/s \rceil$, produces vector $\mathbf{C}_i \in (\mathbb{Z} \times \mathbb{F}^m)^s$ of local inclusive-scans such that $\mathbf{C}_i \stackrel{\text{def}}{=} (C_{i,1},\ C_{i,2},\ \cdots,\ C_{i,s})$,

Algorithm 3 SO exact matching with disjoint segmented parallel inclusive-scan

Input: Text T of n characters, pattern P of m characters, and segment size s ($\geq m$)
Output: Bit sequence X of size n that indicates matching positions

1: Initialize the masks $\mathbf{M}_1, \mathbf{M}_2, \cdots, \mathbf{M}_n$ according to T and P
2: $A_0 \leftarrow \langle 0, 1^m \rangle$
3: $A_j \leftarrow \langle 1, \mathbf{M}_j \rangle$, for all $1 \leq j \leq n$
4: **for** $i \leftarrow 1$ to $\lceil n/s \rceil$ **do in parallel** ▷ for each segment
5: **for** $j \leftarrow 1$ to s **do in parallel** ▷ for each element
6: $C_{i,j} \leftarrow \sum_{l=1}^{j} A_{s(i-1)+l}$ ▷ Summation over †
7: **end for**
8: **end for**
9: $C_{0,s} \leftarrow A_0$
10: **for** $i \leftarrow 1$ to $\lceil n/s \rceil$ **do in parallel** ▷ for each segment
11: **for** $j \leftarrow 1$ to s **do in parallel** ▷ for each element
12: $R_{s(i-1)+j} \leftarrow C_{i-1,s} \dagger C_{i,j}$
13: **end for**
14: **end for**
15: $\langle u_j, \mathbf{x}_j \rangle \leftarrow R_j$, for all $1 \leq j \leq n$
16: $X[j] \leftarrow \mathbf{x}_j[m]$, for all $1 \leq j \leq n$
17: **return** X

where

$$C_{i,j} = \sum_{l=1}^{j} A_{s(i-1)+l}. \tag{18}$$

The above equation indicates that the first $m-1$ elements of $\mathbf{C}_i$ do not include the $m-1$ left neighbors. Therefore, these elements must refer to left segment $\mathbf{C}_{i-1}$ to obtain the final results, as shown in Fig. 3. The final results can be obtained by simply adding last element $C_{i-1,s}$ of the left segment instead of $m-1$ elements (i.e., line 12 of Algorithm 3) because $C_{i-1,s}$ has already summed its left neighbors at line 6. Note that we avoid a conditional branch by letting all elements participate in this addition because summation of more than $m-1$ preceding elements yields the same result as that produced from exactly $m-1$ preceding elements, i.e., for all $1 \leq j \leq n$, $\sum_{l=0}^{j} A_l = \sum_{l=j-m+1}^{j} A_l$, as presented in Section 4.3.

5 cuShiftOr: GPU-based Implementation

In this section, we present implementation issues and solutions for achieving full optimization of our proposed method on a CUDA-compatible GPU. Solutions are presented for the SO algorithm, but they can easily be applied to the WM algorithm. We consider the Maxwell architecture [21] as the target GPU; accordingly, the word size w is set to 32 or 64 according to pattern size m.

Table 1 shows an overview of parallel schemes deployed in our cuShiftOr implementation. Our tribrid method can be efficiently implemented with CUDA [20], which shares instructions among threads of the same warp. Threads in a warp are naturally synchronized, and thereby, the inclusive-scan scheme runs efficiently at the warp level without additional synchronization costs. Further, a shuffle instruction [20] is useful to exchange data for computing inclusive scans within a warp [39]. The inclusive-scan scheme is integrated into the disjoint segmented scheme to avoid duplicate searches at the lower thread level. As for the thread block level, cuShiftOr deploys the same disjoint segmented scheme, because warps in the same thread block can be synchronized without a kernel completion and data exchange within a thread block is available through on-chip shared memory [20], namely, a software-managed cache. By contrast, the overlapping scheme (i.e., with halo regions) independently runs at the upper level of the thread hierarchy—i.e., the grid level—to avoid data exchange between different thread blocks. The search process then completes with a single kernel invocation.

TABLE 1: Parallelization schemes deployed for each level of the thread hierarchy. Fine-grained parallelism is exploited by the parallel scan scheme whereas coarse-grained parallelism is exploited by the segmentation-based scheme.

Level	Parallel scheme	Halo	Synchronization
Grid	Segmentation-based	yes	no
Thread block	Inclusive-scan	no	yes w/ `__syncthreads()`
Warp	Inclusive-scan	no	yes w/o additional cost
Thread	Sequential bit-parallel	no	n/a

In summary, a thread block is responsible for a segment and its corresponding halo region. This responsible area is further decomposed into disjoint subsegments, with each subsegment assigned to a warp via the disjoint segmented scheme described above. Because a warp contains 32 threads, the segment/subsegment size must be a multiple of 32 to maximize resource utilization. To realize this, the halo size is set to the maximum word size (i.e., 64) for arbitrary pattern lengths.

As for the data structure, cuShiftOr assumes that a character is stored as a 1-byte unsigned integer; thus, alphabet Σ is allowed to have 256 symbols. Given n-byte text data and m-byte pattern data, our proposed implementation produces n-byte data containing bit sequence X; each bit of X is stored as 1-byte data to simplify the memory access pattern. These input/output data are stored in off-chip global memory [20], where memory read/write (R/W) throughput can be maximized when each warp accesses a contiguous memory region of 128 bytes [20]. For such contiguous access, memory transactions from threads of the same warp can be coalesced into a single transaction. Further, shared memory holds a table indexed by alphabet Σ. This table is useful for computing the mask $\mathbf{M}_j$, where $1 \leq j \leq n$, according to input character t_j, as described in Section 3.1. Finally, we store current bit-vector $\mathbf{R}_j$, where $0 \leq j \leq n$, in a register to take advantage of intrinsic parallelism of bit operations.

5.1 Algorithm Cascading for Coalesced Access

Our cuShiftOr implementation realizes memory access coalescing for 1-byte data by assigning α (≥ 4) contiguous characters of text to a thread. Thus, α represents the task granularity for threads. To enable this assignment, the sequential bit-parallel algorithm is cascaded to the parallel inclusive-scan algorithm, as shown in Fig. 4. To more easily describe

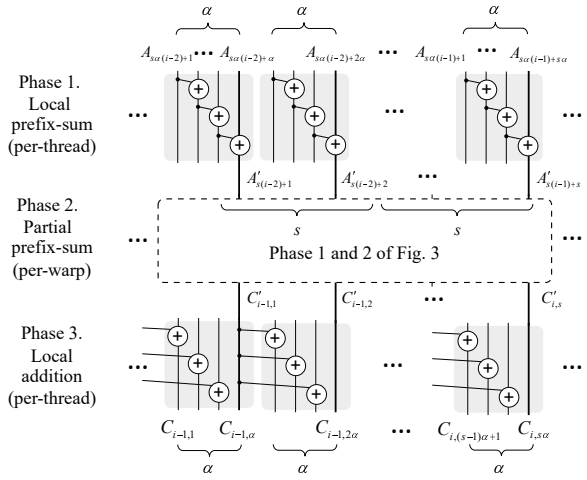


Fig. 4: Overview of algorithm cascading. First, each thread sequentially computes local inclusive-scans of α consecutive elements. Second, inclusive scans for each α value are computed by a warp in parallel according to Algorithm 3. Finally, we obtain global inclusive-scans by adding the result of the left neighbors into the local inclusive-scans. The last addition can be skipped if $m \leq w - \alpha + 1$.

this idea, the grid and thread block levels are omitted in the discussion below. We consider here cascading the disjoint segmented scheme with the sequential bit-parallel scheme, each running at the warp and thread levels, respectively. The cascaded algorithm executes in the following three phases.

1) Local inclusive-scan per thread. The l-th thread of the i-th warp, where $1 \leq l \leq 32$ and $1 \leq i \leq \lceil n/s\alpha \rceil$, sequentially computes $A'_{s(i-1)+l}$, which is the sum of α contiguous elements of a segment, where $A'_j \stackrel{\text{def}}{=} \sum_{l=1}^{\alpha} A_{\alpha(j-1)+l}$. Note that each warp is responsible for $s\alpha$ elements (instead of s elements) owing to algorithm cascading.
2) Parallel inclusive-scan per warp. According to Algorithm 3, the i-th warp, where $1 \leq i \leq \lceil n/s\alpha \rceil$ performs the parallel inclusive-scan algorithm of $A'_{s(i-1)+1}, A'_{s(i-1)+2}, \cdots, A'_{s(i-1)+s}$ to generate every α elements within a segment, i.e., $C'_{i,1}, C'_{i,2}, \cdots, C'_{i,s}$. Note that these s elements are part of the entire segment; the remaining $s(\alpha - 1)$ elements still hold local sums.
3) Local addition per thread. To obtain the remaining global sums, the l-th thread of the i-th warp, where $1 \leq l \leq 32$ and $1 \leq i \leq \lceil n/s\alpha \rceil$, updates its local sums by adding the global sum of its left neighboring thread; the global sum to be added is given by $C'_{i,l-1}$, if $l \geq 2$, otherwise it is given by $C'_{i-1,s}$.

The last addition phase can be skipped if $m \leq w - \alpha + 1$. The key idea behind this elimination is to extend the pattern such that it ends with $\alpha - 1$ wildcards, which match to arbitrary characters of the text. Obviously, this extension reduces the maximum length of the pattern from w to $w - \alpha + 1$; however, as shown in Fig. 5, wildcards allow bit-vector $\mathbf{R}_j$ of Eq. (1) to save the current automaton states for the future $\alpha - 1$ steps by extending the bit-vector with

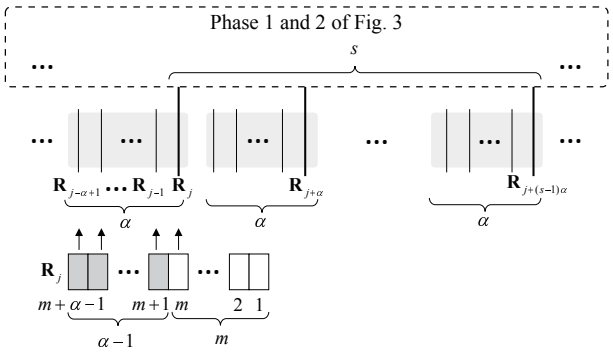


Fig. 5: Wildcard extension for skipping the last addition phase. After the partial inclusive-scan algorithm of Fig. 4 executes, $\mathbf{R}_j$ holds global results, whereas its preceding $\alpha - 1$ elements $\mathbf{R}_{j+\alpha-1}, \mathbf{R}_{j+\alpha-2}, \cdots, \mathbf{R}_{j-1}$ hold local results. The global results of the $\alpha - 1$ elements can be estimated from those of $\mathbf{R}_j$ by extending the vector by $\alpha - 1$ bits.

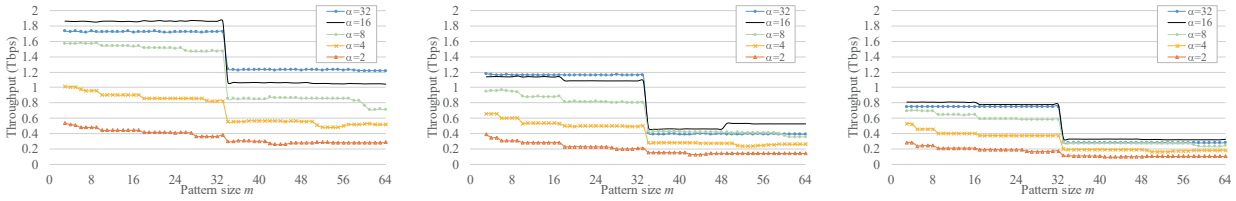
$\alpha - 1$ bits, i.e., for all $0 \leq i < \alpha$ and $\alpha < j \leq n$, $\mathbf{R}_j[m + i] = \mathbf{R}_{j-i}[m]$. This equivalent relation is useful for retrieving the entire segment from every α element that hold the complete inclusive-scans. The j-th thread is allowed to substitute $\mathbf{R}_j[m], \mathbf{R}_j[m+1], \ldots, \mathbf{R}_j[m+\alpha-1]$ for $\mathbf{R}_j[m], \mathbf{R}_{j+1}[m], \ldots, \mathbf{R}_{\alpha-1}[m]$, respectively.

As for the appropriate value of α, we experimentally determined the best value that maximized search throughput. As a guideline, α is at least four for 1-byte data. One may feel that $\alpha > 4$ fails to coalesce memory transactions because a contiguous memory region of larger than 128 bytes is simultaneously accessed by a warp; however, our preliminary results indicate that the number of global memory transactions does not increase if the memory region size is 512 bytes, which may be due to the demand from computer graphics applications, which usually deploy the `float4` data structure to pack RGBA data into 512-byte data.

5.2 Complete Loop Unrolling

Our proposed method has two distinct loops within the kernel function. One is the d-loop at line 4 of Algorithm 2; the other is the j-loop at line 8, required for the inclusive-scan operation. As mentioned in Section 4.3, every thread iterates through the latter loop $\log m$ times instead of $\log n$ times, while the former loop executes k times. Search throughput can be increased by completely unrolling these loops. To do so, we use the C++ template presented in [40]. More specifically, the unrolled kernel function for specific values of m and k is automatically generated by the C++ template at compile time. This template-based scheme generates $m \times (k + 1)$ kernel functions. Because constant values of m and k are embedded at compile time, the generated kernel consumes fewer register files.

One drawback of the template-based scheme here is that it is not robust against increases in m and k. Let M and K be the maximum values of m and k acceptable at compile time, respectively, with $m \leq M$ and $k \leq K$. Consider a kernel function generated for pattern size m and maximum error k. The generated function then contains



(a) (b) (c)

Fig. 6: Search throughput of our proposed cuShiftOr implementation with a variety of values for granularity size α. Results are shown for (a) $k = 0$, (b) $k = 1$, and (c) $k = 2$. Text and alphabet sizes were fixed to $n = 2^{31}$ and $\sigma = 64$, respectively.

$k \log m$ statements because there is a double-nested loop, where the outer and inner loops iterate k and $\log m$ times, respectively. Hence, the total number of statements is given by $\sum_{k=1}^{K} \sum_{m=1}^{\log M} k \log m$. Thus, the code size is bounded by $O(K^2 M \log M)$, which restricts m and k.

6 Experimental Results

We evaluated our proposed cuShiftOr implementation in terms of search throughput ρ. Note that search throughput does not include data transfer time between the CPU and GPU. Our experimental machine had an Intel Xeon CPU E5-2660 v3 CPU with 64 GB RAM and an NVIDIA GeForce GTX TITAN X (Maxwell) GPU with 12 GB VRAM. We used CUDA 7.5 [20], GPU Driver 353.82, and Visual Studio 2013 running on Windows 7.

For a long string that exceeds the capacity of GPU memory, ρ is obviously limited by the bandwidth of the PCIe bus. On the machine used in our experiments, the data transfer rate from the CPU to GPU was 10.6 GB/s and 11.2 GB/s for the opposite direction. By contrast, the effective bandwidth of the GPU memory reached 248.8 GB/s on the experimental GPU according to the bandwidthTest program [39]. Because string matching is usually a memory-intensive operation, we focused on the in-core situation rather than the out-of-core situation. Therefore, in-core search throughput ρ is bounded by 248.8 GB/s or 1.99 Tbps.

As for a comparable method, we implemented GBPR [11] as a segmentation-based method. Our method differs from GBPR in realizing (1) a parallel scan algorithm that removes duplicate searches by eliminating the halo regions and (2) algorithm cascading that facilitates coalesced memory access by mapping appropriate algorithms to the thread hierarchy. We implemented GBPR by ourselves because it was not publicly available. Our GBPR implementation used the same data structure as the original GBPR; we used shared memory to store bit masks and global memory to store the text and search results. Segment size s was experimentally determined to be $s = 8$, which maximized search throughput ρ on our experimental machine. The readers can refer to the supplementary material for detailed description on GBPR including a sensitivity study on s and efficiency analysis. We also implemented an extended version of GBPR that realizes coalesced memory access with a tiling technique. Similar to cuShiftOr, the tiled GBPR allows every thread to fetch α (> 1) contiguous characters at a time to locally update automata states α times. This procedure is repeated until the end of its responsible segment is reached.

6.1 Parameter Tuning

We first conducted a preliminary experiment to determine the best task granularity α that maximized the search throughput ρ for cuShiftOr. Figure 6 shows search throughput ρ measured while varying task granularity α and maximum error k, with maximum text size $n = 2^{31}$ and maximum alphabet size $\sigma = 64$. The text and patterns were generated randomly. As shown in the figure, α greatly impacts search throughput ρ, which ranged from 0.26 to 1.88 Tbps when $k = 0$ (i.e., exact matching). According to these results, we decided to use $\alpha = 16$, which achieved the highest throughput for most pattern sizes.

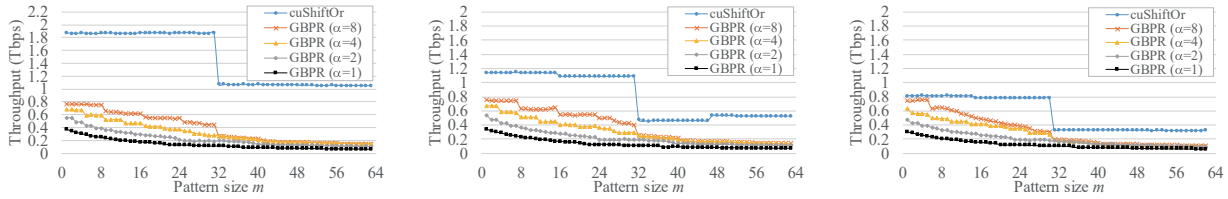
In Fig. 6a, we observe that cuShiftOr significantly dropped search throughput to 1.05 Tbps when $m > 32$. For such a long pattern, cuShiftOr uses a 64-bit word to store the bit-vector but the current CUDA does not provide shuffle instructions for 64-bit data. Consequently, switching word size w degraded the efficiency of the shuffle instructions required for inclusive-scan computation; in other words, a 64-bit shuffle operation was implemented using 32-bit shuffle instructions. This performance degradation was not observed with smaller α, where search throughput ρ was determined by off-chip memory access rather than instruction issue rate (i.e., on-chip memory access). Therefore, we conclude here that choosing the best task granularity α is key to maximizing search throughput ρ.

6.2 Performance Analysis

Using task granularity α determined using the methods above, we measured search throughput ρ while varying text size n, pattern size m, maximum error k, and alphabet size σ. The text and patterns were again generated randomly.

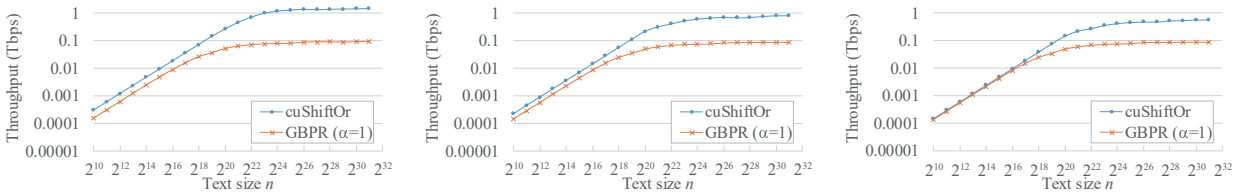
Figure 7 shows search throughput ρ measured while varying pattern size m and maximum error k with maximum text size $n = 2^{31}$ and maximum alphabet size $\sigma = 64$. Here, cuShiftOr was 16.9 times faster than GBPR ($\alpha = 1$) when $k = 0$. In particular, search throughput ρ for a short pattern (i.e., $k = 0$ and $m \leq 32$) reached 1.88 Tbps, demonstrating a high efficiency of 94% in terms of off-chip memory R/W throughput. This achieved efficiency indicates that cuShiftOr successfully exploits the strength of the latency hiding architecture; in other words, scan computation is fully overlapped with off-chip memory access.

Both cuShiftOr and GBPR saw degradation in search throughput ρ as we increased pattern size m from 33 to 64; however, this decrease was relatively small for cuShiftOr. As an example, when $k = 0$, cuShiftOr decreased search



(a) (b) (c)

Fig. 7: Search throughput with different pattern sizes m. Results are shown for (a) $k = 0$, (b) $k = 1$, and (c) $k = 2$. Text and alphabet sizes were fixed to $n = 2^{31}$ and $\sigma = 64$, respectively. The original GBPR corresponds to $\alpha = 1$.



(a) (b) (c)

Fig. 8: Search throughput with varying text sizes n. Results are shown for (a) $k = 0$, (b) $k = 1$, and (c) $k = 2$. Pattern size m and alphabet size σ were fixed to $m = 64$ and $\sigma = 64$, respectively. The vertical axis is presented as a logarithmic scale.

throughput ρ by only 2%, whereas GBPR ($\alpha = 1$) decreased ρ by 45%. We attribute this substantial difference to the deployed parallel scheme. As mentioned above, the instruction issue rate limited the performance of cuShiftOr. The number of instructions per thread is given by $O(\log m)$ for cuShiftOr because threads compute inclusive-scans in $\log m$ parallel steps. By contrast, threads in GBPR (i.e., the segmentation-based scheme) are responsible for $s + m - 1$ characters in the text, meaning that each thread must read a segment of $O(s + m)$ characters from off-chip memory, where $s = 8$. Therefore, cuShiftOr is more robust than GBPR against increases of pattern size m. Obviously, this conclusion assumes that pattern size is short enough to fit within machine word size w. Note that cuShiftOr also deploys the segmentation-based scheme, but segments are assigned to thread blocks, namely, a higher level of thread hierarchy. Therefore, s is much larger than m, which can be ignored at the thread level.

As for approximate string matching, cuShiftOr dropped search throughput ρ as we increased maximum error k. For example, the maximum ρ for $k = 1$ (or $k = 2$) was approximately 39% (or 29%) lower than that of $k = 0$ (or $k = 1$, respectively). By contrast, GBPR ($\alpha = 1$) maintained a high search throughput ρ for arbitrary k. We attribute this difference to the performance bottleneck of the GPU code. The performance bottleneck of cuShiftOr was on-chip memory, whereas that of GBPR was off-chip memory. The amount of on-chip memory access increases with k because the time complexity of the WM algorithm is given by $O(nm/wk)$, which linearly increases with k. By contrast, the amount of off-chip memory access is independent of k because Algorithm 2 reads and writes off-chip memory data only once at lines 1 and 13, respectively. Consequently, cuShiftOr degraded search throughput ρ as k increased, whereas GBPR held ρ almost constant for arbitrary k.

Our tiled version of GBPR ($\alpha > 1$) explains why cuShiftOr was faster than the original GBPR. In Fig. 7c, the tiled GBPR ($\alpha = 8$) achieved up to 93% of search throughput obtained via cuShiftOr; as for approximate string matching, the tiling technique (i.e., memory coalescing) is key for eliminating the performance gap between cuShiftOr and GBPR. However, memory coalescing was insufficient in yielding Tbps speeds for search throughput where on-chip memory access determines performance. In this case, the amount of on-chip memory access must be reduced by eliminating duplicate searches.

Figure 8 shows search throughput measured while varying text size n for maximum pattern size $m = 64$ and $\sigma = 64$. The figure indicates that the text should contain at least 2^{24} characters (i.e., 16 MB text data) to achieve at least 75% of the maximum search throughput. In contrast, GBPR achieved 75% of the maximum search throughput with 2^{22} characters, which was 25% smaller than that needed for cuShiftOr. This difference occurred due to a side effect of algorithm cascading in which a large α causes serialization on each thread. In fact, the ratio for 2^{22} characters was increased from 49% to 74% by reducing task granularity α from 16 to 2. Therefore, cuShiftOr requires a larger text than GBPR to increase the ratio of parallel processing over serial processing, as Amdahl's law [41] points out.

We also found that, when $n \leq 2^{22}$, GBPR varied ρ according to n, which differed from the behavior that we theoretically expected. GBPR processes $\lceil n/s \rceil$ segments of size $s + m - 1$ on p processors. Consequently, assuming that $m = O(w)$ (i.e., short patterns), the search throughput of GBPR is expected to be $sp/(s + m - 1)$, which is independent of n. This gap between theoretical behavior and actual behavior is due to the lack of parallelism that can cause idle time on massively parallel GPU threads. The GPU architecture requires a large number of threads to sufficiently hide off-chip memory latency by overlapping memory access with data-independent computation. Notice

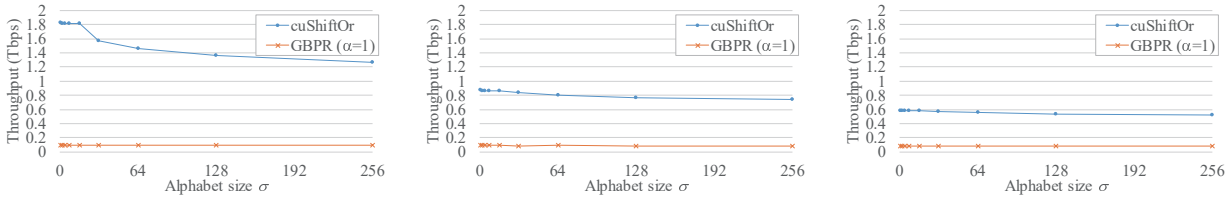


Fig. 9: Search throughput with varying alphabet sizes σ. Results are shown for (a) $k = 0$, (b) $k = 1$, and (c) $k = 2$. Text size and pattern size were fixed to $n = 2^{31}$ and $m = 64$, respectively.

that cuShiftOr also faced on the same issue when $n \leq 2^{22}$.

Finally, as shown in Fig. 9, we measured search throughput ρ while varying alphabet size σ for maximum text size $n = 2^{31}$ and maximum pattern size $m = 64$. When $k = 0$, ρ clearly decreased as we increased σ. By contrast, we did not observe such behavior when $k > 0$. Search throughput ρ for $k = 0$ was determined by on-chip memory access rather than off-chip memory access. Consequently, bank conflicts in shared memory limited ρ when $\sigma > 32$ because the shared memory of the current CUDA is structured with 32 banks; further, the cuShiftOr implementation uses shared memory to store a table indexed by alphabet Σ, as mentioned in Section 5. Realizing conflict-free access to this shared table is not easy owing to the irregularity of characters in the text.

6.3 Performance Comparison with Corpus

In this subsection, we show performance comparison results obtained using practical data sets. For text data, we used the Pizza&Chilli Corpus [42], which includes corpus data sets across a variety of fields. The first 64 characters of the text were selected as the pattern to be searched for. In addition to GBPR [11], we also compared cuShiftOr with XBitPar [26] obtained from http://xbitpar.sourceforge.net/.

Figure 10 shows measured search throughputs. When $k = 0$ and $m = 64$, search throughput ρ of cuShiftOr ranged from 1782 to 1872 Gbps. These throughputs were 30% higher than those expected from results obtained with randomly-generated data sets (i.e., Fig. 9a); we expected search throughput to be approximately 1400 Gbps when $\sigma > 32$. This unexpected behavior was due to the skewed distribution of characters in the corpus, which did not exist in the randomly-generated data sets. As an example, the ENGLISH data set contained 239 characters, but its inverse probability of matching was 28.7, indicating that specific characters in the alphabet frequently appeared in the text. In such a case, cuShiftOr can slightly increase search throughput by avoiding bank conflicts in shared memory. For example, bit masks for A, G, T, and C are stored in the 1st, 7th, 20th, and 3rd banks, respectively, because the table of bit masks is indexed using the corresponding ASCII codes. Consequently, bank conflicts will not occur for biological data that contains these four characters. In contrast, randomly generated datasets contain 256 characters, thus each bank is responsible for eight characters. In this case, bank conflicts occur when 32 threads in a warp access these characters at the same time.

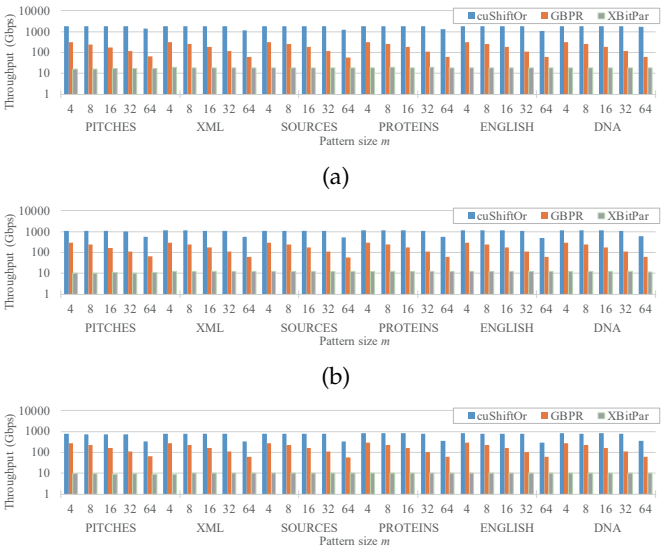


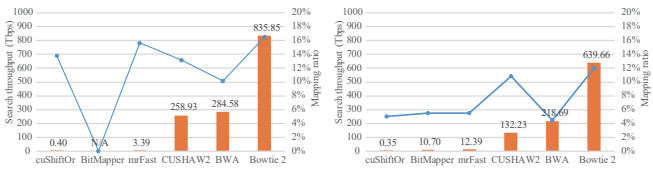
Fig. 10: Search throughput using the Pizza&Chilli Corpus [42]. Results are shown for (a) $k = 0$, (b) $k = 1$, and (c) $k = 2$, where $4 \leq m \leq 64$. PITCHES, XML, SOURCES, PROTEINS, ENGLISH, and DNA data sets have alphabet sizes σ of 133, 97, 230, 27, 239, and 16, respectively, and their inverse probabilities of matching are 39.8, 28.7, 24.8, 17.0, 15.3, and, 3.9, respectively.

The increases in terms of runtimes of cuShiftOr over GBPR were similar to those obtained with the random data presented in Section 6.2 above. Here, cuShiftOr was 15.7–16.7 times faster than GBPR when $m = 32$ and $k = 0$, but the gap between performance measures decreased as we increased k. More specifically, we observed 9.3–10.2 times and 6.7–7.4 times when $k = 1$ and $k = 2$, respectively. Thus, the family of bit-parallel algorithms demonstrated stable search throughput for arbitrary data.

Both cuShiftOr and GBPR failed to search a pattern longer than 64 characters, whereas XBitPar successfully processed a long pattern of up to 1024 characters. Nonetheless, cuShiftOr was 93–108 times faster than XBitPar when $m = 32$ and $k = 0$. XBitPar adopts a 1024-bit virtual word for arbitrary m, thus the execution efficiency drops for short queries. More specifically, $w - m$ bits in a word were wasted during search, thus only 3% of threads were utilized when $m = 32$. Therefore, cuShiftOr and XBitPar should be appropriately switched according to pattern length m. Note

that cuShiftOr can be extended to search a long pattern with a virtual word. However, this extension degrades search throughput due to the overhead required to move data between physical words.

Thus, the limitation on pattern length is critical for long patterns, but there are many real-world cases that require support only for short patterns. For example, a text search query typically consists of two to four words and the pattern length distribution follows a power law [43]. With respect to biological applications, next-generation sequencing systems produce short reads up to several hundred characters, which are then aligned to a long sequence of millions of characters.



(a) (b)

Fig. 11: Alignment results of cuShiftOr and sequence aligners [34]–[38]. Results for (a) National Center for Biotechnology Information (NCBI) SRR003084 (8.8 M reads of 36 base pair (bp)) and (b) NCBI SRR003092 (16.1 M reads of 51 bp). BitMapper failed to align SRR003084.

6.4 Performance Comparison with Sequence Aligners

Finally, we compared cuShiftOr with indexing-based sequence aligners running on a CPU: BitMapper [34] and mrFAST [35] as all-mappers, which find all mappings of a given sequencing read in the reference genome by setting a maximum edit distance, and CUSHAW2 [36], BWA-MEM [37], and Bowtie2 [38] as best-mappers, which aim to report the best few mappings under similar constraints. Millions of short reads from the 1000 Genomes Project were aligned to Homo sapiens chromosome 1, GRCh38.p7. As shown in Fig. 11, cuShiftOr achieved search throughputs of 0.35–0.4 Tbps. In contrast, sophisticated indexes allowed the aligners to focus on possible matching locations in the text. Consequently, all-mappers and best-mappers aligned reads at approximately 3–12 Tbps and 132–836 Tbps, respectively.

Figure 11 also shows mapping ratios, i.e., the number of mapped reads divided by the total number of reads. The mapping ratios of cuShiftOr were not exactly the same as those of mrFAST [35], although the same maximum edit distance of two was commonly used for search. This difference was due to reverse complementary sequences and ambiguous nucleotides; cuShiftOr accepts a single pattern per search, thus the search process must be invoked twice for the original pattern and its reverse complementary pattern. Moreover, these results must be merged into a single result. Finally, cuShiftOr neither skips reads that include many ambiguity characters nor finds a match between the ambiguity character (N) and any character (A, G, T, or C).

Consequently, we think that cuShiftOr is useful for accelerating online searching where indexes cannot be constructed before searching. For example, cuShiftOr can be used for text stream mining in social networks, where text streams are analyzed on the fly. Because CPU-GPU data transfer usually limits the performance of GPU systems, the impact of cuShiftOr increases if text streams are iteratively analyzed on the GPU. Similar promising examples are network intrusion detection and code debugging, where log data are iteratively examined to find regions of interest.

7 Conclusion

In this paper, we presented a tribrid parallel method, named cuShiftOr, that is capable of exploiting massive data parallelism inherent in string matching. Our proposed method interprets the SO and WM algorithms as inclusive-scan operations. This interpretation allows massive numbers of threads to run without halo regions, but requires global synchronization. To reduce these synchronization costs, we integrated the scan scheme into a segmentation-based scheme, which does require halo regions (i.e., duplicate work) but avoids synchronization. We showed that these contrasting methods can be appropriately mapped onto the hierarchy of GPU threads. As such, the scan scheme exploits fine-grained parallelism to minimize duplicate work, whereas the segmentation-based scheme exploits coarse-grained parallelism to avoid global synchronization.

We analyzed the performance of cuShiftOr and compared its performance to previous segmentation-based methods. In our results, cuShiftOr achieved 1.88 Tbps of in-core search throughput for short queries, which was equivalent to 94% execution efficiency in terms of off-chip memory R/W throughput. However, the throughputs were not satisfactory compared to those of sequence aligners that construct indexes before searching. Thus, we conclude that cuShiftOr is useful for accelerating online string matching for short patterns containing up to 64 characters.

Acknowledgments

This study was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Numbers 15K12008, 15H01687, and 16H02801. We are also grateful to the anonymous reviewers for their valuable comments.

References

[1] A. Tumeo and O. Villa, "Accelerating DNA analysis applications on GPU clusters," in *Proc. 8th Symp. Appl. Specific Processors*, Jun. 2010, pp. 71–76.
[2] S. Faro and T. Lecroq, "The exact online string matching problem: A review of the most recent results," *ACM Comput. Surv.*, vol. 45, no. 2, Feb. 2013, article 13, 42 pages.
[3] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," *IEEE Trans. Comput.*, vol. 62, no. 10, pp. 1906–1916, Oct. 2013,
[4] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surveys*, vol. 33, no. 1, pp. 31–88, Mar. 2001.
[5] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, Jun. 1977.
[6] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.
[7] R. Baeza-Yates, "Efficient text searching," Ph.D. dissertation, University of Waterloo, Waterloo, ON, Canada, May 1989.
[8] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Commun. ACM*, vol. 35, no. 10, pp. 74–82, Oct. 1992.
[9] S. Wu and U. Manber, "Fast text searching allowing errors," *Commun. ACM*, vol. 35, no. 10, pp. 83–91, Oct. 1992.

[10] K. Kusudo, F. Ino, and K. Hagihara, "A bit-parallel algorithm for searching multiple patterns with various lengths," *J. Parallel Distrib. Comput.*, vol. 76, pp. 49–57, Feb. 2015.
[11] C. H. Lin, G. H. Wang, and C. C. Huang, "Hierarchical parallelism of bit-parallel algorithm for approximate string matching on GPUs," in *Proc. Symp. Comput. Appl. and Commun.*, Jul. 2014, pp. 76–81.
[12] T. T. Tran, S. Schindel, Y. Liu, and B. Schmidt, "Bit-parallel approximate pattern matching on the Xeon Phi coprocessor," in *Proc. 26th Int. Symp. Comput. Archit. and High Perform. Comput.*, Oct. 2014, pp. 81–88.
[13] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.
[14] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, Aug. 2007, ch. 39, pp. 851–876.
[15] A. Khajeh-Saeed, S. Poole, and B. Perot, "Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors," *J. Comput. Physics*, vol. 229, no. 11, pp. 4247–4258, Jun. 2010.
[16] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Molecular Biol.*, vol. 147, no. 1, pp. 195–197, Mar. 1981.
[17] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS Symp. Graph. Hardware*, Aug. 2007, pp. 97–106.
[18] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, Nov. 1990.
[19] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986.
[20] NVIDIA Corporation, "CUDA C Programming Guide Version 7.5," Sep. 2015. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
[21] ——, "NVIDIA GeForce GTX 980," Nov. 2014. [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF
[22] Y. Liu, L. Guo, J. Li, M. Ren, and K. Li, "Parallel algorithms for approximate string matching with k mismatches on CUDA," in *Proc. IEEE 26th Int. Symp. Parallel and Distrib. Process. Workshops and PhD Forum*, May 2012, pp. 2414–2422.
[23] D. Man, K. Nakano, and Y. Ito, "The approximate string matching on the hierarchical memory machine, with performance evaluation," in *Proc. IEEE 7th Int. Symp. Embedded Multicore/Manycore System-on-Chip*, Sep. 2013, pp. 79–84.
[24] K. Xu, W. Cui, Y. Hu, and L. Guo, "Bit-parallel multiple approximate string matching based on GPU," in *Proc. 1st Int. Conf. Inf. Technol. and Quantitative Manage.*, May 2013, pp. 523–529.
[25] M. Onsjö and O. Watanabe, "Implementation of a bit-parallel approximate string matching algorithm," in *IPSJ SIG Technical Report*, vol. 2009-AL-124, no. 3, May 2009.
[26] T. T. Tran, Y. Liu, and B. Schmidt, "Bit-parallel approximate pattern matching: Kepler GPU versus Xeon Phi," *Parallel Comput.*, vol. 54, pp. 128–138, May 2016.
[27] Intel Corporation, "Intel architecture instruction set extensions programming reference," Dec. 2013. [Online]. Available: http://download-software.intel.com/sites/default/files/managed/71/2e/319433-017.pdf
[28] R. Prasad, S. Agarwal, I. Yadav, and B. Singh, "A fast bit-parallel multi-patterns string matching algorithm for biological sequences," in *Proc. Int. Symp. Biocomput.*, no. 46, Feb. 2010, 4 pages.
[29] R. Baeza-Yates and G. Navarro, "Faster approximate string matching," *Algorithmica*, vol. 23, no. 2, pp. 127–158, Feb. 1999.
[30] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999.
[31] L. Langner, "Parallelization of Myers fast bit-vector algorithm using GPGPU," Ph.D. dissertation, Freie Universität Berlin, Berlin, Germany, Apr. 2011.
[32] D. Man, K. Nakano, and Y. Ito, "An optimal implementation of the approximate string matching on the hierarchical memory machine, with performance evaluation on the GPU," *IEICE Trans. Inf. Syst.*, vol. E97-D, no. 12, pp. 3063–3071, Dec. 2014.
[33] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu, "Accelerating string matching using multi-threaded algorithm on GPU," in *Proc. Global Commun. Conf.*, Dec. 2010, 5 pages.
[34] H. Cheng, H. Jiang, J. Yang, Y. Xu, and Y. Shang, "Bitmapper: an efficient all-mapper based on bit-vector computing," *BMC Bioinformatics*, vol. 16, no. 192, Jun. 2015.
[35] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler, "Personalized copy number and segmental duplication maps using next-generation sequencing," *Nature Genetics*, vol. 41, no. 10, pp. 1061–1067, Aug. 2009.
[36] Y. Liu and B. Schmidt, "Long read alignment based on maximum exact match seeds," *Bioinformatics*, vol. 28, no. 18, pp. i318–i324, Sep. 2012.
[37] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," arXiv:1303.3997 [q-bio.GN], 2013.
[38] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, Mar. 2012.
[39] NVIDIA Corporation, "CUDA Code Samples," 2014. [Online]. Available: http://developer.nvidia.com/cuda-code-samples/.
[40] M. Harris, "Optimizing parallel reduction in CUDA," Nov. 2007. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.
[41] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. the AFIPS Conf.*, 1967, pp. 483–485.
[42] P. Ferragina and G. Navarro, "Pizza&chili corpus compressed indexes and their testbeds," Sep. 2005. [Online]. Available: http://pizzachili.dcc.uchile.cl/.
[43] A. Arampatzis and J. Kamps, "A study of query length," in *Proc. 31st Ann. Int. ACM SIGIR Conf. Res. and Development in Inf. Retrieval*, Jul. 2008, pp. 811–812.

PLACE PHOTO HERE

Yasuaki Mitani received the B.E. and M.E. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 2014 and 2016, respectively. He is currently with DWANGO Corporation, Ltd. His current research interests include high performance computing and string processing.

PLACE PHOTO HERE

Fumihiko Ino (S'01–A'03–M'04) received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently an Associate Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation.

PLACE PHOTO HERE

Kenichi Hagihara received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1974, 1976, and 1979, respectively. From 1994 to 2002, he was a Professor in the Graduate School of Engineering Science, Osaka University. Since 2002, he has been a Professor in the Graduate School of Information Science and Technology, Osaka University. His research interests include the fundamentals and practical application of parallel processing.