IEICE TRANS. INF. & SYST., VOL.E103–D, NO.12 DECEMBER 2020

PAPER Special Section on Parallel, Distributed, and Reconfigurable Computing, and Networking

A Rabin-Karp Implementation for Handling Multiple Pattern-Matching on the GPU*

Lucas Saad Nogueira NUNES[†], ***Nonmember***, **Jacir Luiz BORDIM**[†a)], **Yasuaki ITO**[††], ***and*** **Koji NAKANO**[††], ***Members***

SUMMARY The volume of digital information is growing at an extremely fast pace which, in turn, exacerbates the need of efficient mechanisms to find the presence of a pattern in an input text or a set of input strings. Combining the processing power of Graphics Processing Unit (GPU) with matching algorithms seems a natural alternative to speedup the string-matching process. This work proposes a Parallel Rabin-Karp implementation (PRK) that encompasses a fast-parallel prefix-sums algorithm to maximize parallelization and accelerate the matching verification. Given an input text T of length n and p patterns of length m, the proposed implementation finds all occurrences of p in T in $O(m + q + \frac{n}{\tau} + \frac{nm}{q})$ time, where q is a sufficiently large prime number and τ is the available number of threads. Sequential and parallel versions of the PRK have been implemented. Experiments have been executed on $p \geq 1$ patterns of length m comprising of $m = 10, 20, 30$ characters which are compared against a text string of length $n = 2^{27}$. The results show that the parallel implementation of the PRK algorithm on NVIDIA V100 GPU provides speedup surpassing 372 times when compared to the sequential implementation and speedup of 12.59 times against an OpenMP implementation running on a multi-core server with 128 threads. Compared to another prominent GPU implementation, the PRK implementation attained speedup surpassing 37 times.

key words: *Rabin-Karp algorithm, prefix-sums, pattern matching, GPGPU, CUDA*

1. Introduction

String or pattern matching algorithms are used to find the occurrences of a pattern in a text or a set of input strings [2]. The task of finding strings that produce a complete or a partial match to a given pattern has many practical applications, such as plagiarism detection, DNA sequencing, text mining, spam filtering, intrusion detection systems, virus scanning, and so on [2]–[4]. Given a pattern P and a string T of length m and n ($m \ll n$), respectively, the pattern matching is a task that asks to find all occurrences of the pattern P in T. A naive strategy is to perform character-by-character comparisons between the text substring and the complete pattern P and then shift T one position to the right. Clearly, this strategy runs in $O(nm)$ time [5]. Popular string searching algorithms such as Boyer-Moore (BM) [6], Aho-Corasick (AC) [7], Rabin-Karp (RK) [8] and Knuth-Morris-Pratt (KMP) [9] reduce the computing time by avoiding to re-scan the input string T to find a match. The Rabin-Karp algorithm, for instance, solves the pattern search problem, with high probability, in $O(n)$ time.

Aiming at accelerating the pattern matching computation, GPU (Graphics Processing Unit) implementations have been considered in the literature [10]. Initially, GPUs have been designed to serve as specialized circuit to accelerate computation for manipulating and rendering 3D images [11]. Latest GPUs are designed for general purpose computing (a.k.a. GPGPU) and can perform computation in applications traditionally handled by the CPU. GPU maximizes processing efficiency by offloading some of the operations from the CPU to the GPU. Zha and Sahni [12] implemented the AC and BM algorithms on the GPU and compared the results to a single and multi-threaded implementation. The implementation showed a speedup for the AC algorithm up to 9 times as compared to a sequential algorithm and speedup for the BM algorithm up to 3.2 times on a multi-thread CPU. Pattern matching algorithms tailored for intrusion detection (IDS) systems implemented on the GPU have been proposed in [13]–[15]. Jacob et al. [13] demonstrate that offloading the IDS computation to the GPU provides higher packet-processing rates. They showed that an open source IDS running on the GPU provides up to 40% improvement as compared to the conventional IDS on the CPU. Lin et al. [14] reported that a direct GPU implementation of a string-matching algorithm may fail to detect pattern matching in certain cases. The proposed alternative improves over the AC algorithm and showed better performance on the GPU as compared to a traditional AC implementation. Sharma et al. [15] present a Rabin-Karp pattern-matching algorithm for Deep Packet Inspection implementation on the GPU. The proposed CUDA-based implementation outperformed a quad-core processor providing speedup of up to 14 times. Kouzinopoulos et al. [16] evaluated several multiple pattern matching algorithms on the GPU. They reported that even a basic implementation of these algorithms in the GPU were between 2.5 and 10.9 faster than a single core CPU implementation. Similarly, Ashkiani et al. [17] analyzed different string-matching approaches, namely cooperative, divide-and-conquer and a hybrid approach. The results showed that the divide-and-conquer performed better on shorter patterns while the cooperative ap-

Manuscript received December 31, 2019.
Manuscript revised May 30, 2020.
Manuscript publicized September 24, 2020.

[†]The authors are with Department of Computer Science, University of Brasilia, 70910-900, Brasilia-DF, Brazil.
[††]The authors are with Department of Information Engineering, Hiroshima University, Higashihiroshima-shi, 739–8527 Japan.

*A preliminary version of this paper has been presented at the Sixth International Symposium on Computing and Networking (CANDAR'18) [1].

a) E-mail: bordim@unb.br

DOI: 10.1587/transinf.2020PAP0002

proach was superior on lengthy patterns. The results showed speedup of 4.81 times compared to traditional CPU methods. Shah and Oza [18] proposed a CUDA implementation of the Rabin-Karp algorithm. The computation of the hash values is performed by left shifting each character and add it to previously computed hash values. The paper compares the implementation of CUDA and serial versions of the proposed Rabin-Karp string matching algorithm. CUDA implementation presented speedup of 23 times over the sequential version running on the CPU. Dayarathne and Ragel [19] proposed a Rabin-Karp implementation on the GPU and evaluated its runtime to a sequential and parallel implementation on the CPU. The experimental results showed speedup gains up to 15.68 times compared to a serial implementation. A peculiarity of this implementation is that the GPU performance degrades rapidly as m increases.

This paper presents a prefix-sum-based Rabin-Karp implementation (PRK, for short) that encompasses a novel mechanism to speedup the computation of intermediate hash values. PRK uses a fast-parallel prefix-sum algorithm that includes a look-up table to improve parallelization and speedup the matching process. More precisely, the PRK implementation on the GPU adapts the prefix-sum presented in [20] to improve parallelization. Furthermore, PRK explores atomic operations to control hash collisions that may occur when considering multiple patterns p. For an input text T of length n and p patterns of length m, the proposed PRK algorithm finds the matching positions of p in T in $O(m+q+\frac{n}{\tau}+\frac{nm}{q})$ time, where q is a sufficiently large prime number and τ is the available number of threads. To evaluate the performance of the proposed algorithms, sequential and parallel versions have been implemented on a multi-core CPU. The experimental results have been executed on input text string T of length $n = 2^{27}$ with $p = 1, 4, 16, 64, 256$ patterns of length $m = 10, 20, 30$ characters. Experimental results show that the proposed PRK parallel implementation on NVIDIA V100 GPU provides speedup surpassing 372 times and 12.59 as compared to a serial implementation and OpenMP implementation on a multi-core server, respectively. The proposed PRK implementation is compared to the GPU implementation proposed by Dayarathne and Ragel [19]. Experimental results show that the PRK attained speedup of 1.13 for $p = 1$ and surpassing 37 times for $p = 256$.

The rest of this paper is organized as follows. Section 2 defines the pattern search problem, presents a simple matching function and provides an overview of the Rabin-Karp algorithm. Section 3 presents an intuitive parallel Rabin-Karp for multiple patterns and lays the foundation for the proposed parallel algorithm. Section 4 presents the proposed parallel Rabin-Karp algorithm on the GPU. Experimental results are shown in Sect. 5. Finally, Sect. 6 concludes this work.

2. Rabin-Karp Algorithm

In this section we present an overview of the Rabin-Karp algorithm [8], which is a hash-based, string-matching algorithm used for detecting plagiarism, virus scanning, intrusion detection systems, among other applications. We begin by presenting a simple matching function that is used in the proposed-prefix-sums based Rabin-Karp algorithm that will be presented in the next subsections. For this purpose, let $T = t_0 t_1 \dots t_{n-1}$ be a string of n characters (8-bit unsigned integers). Also, let $P_0, P_1, \dots, P_{p-1}$ be p patterns, such that each $p_{i,0} p_{i,1} \dots p_{i,m-1}$ $(0 \le i \le p-1)$ is a string of m characters. The *pattern search problem* asks to find all matching positions in T for all p patterns. More precisely, the pattern searching finds all pairs (i, j) of position j and pattern P_i such that

$$t_j t_{j+1} \dots t_{j+m-1} = p_{i,0} p_{i,1} \dots p_{i,m-1}. \tag{1}$$

A naive pattern matching implementation may use a sliding window of length m and move it one position to the right of the text T after each attempt. Let $Match(i, j)$ be a function such that it returns true if and only if Eq. (1) is satisfied. Algorithm 1 shows a possible implementation of function $Match(i, j)$. Let us assume that $p = 1$, that is, we have a single pattern $P = p_0 p_1 \dots p_{m-1}$ comprising of m characters for the pattern search problem. This straightforward implementation can compute $Match(i, j)$ in $O(m)$ time. Clearly, the pattern search problem, for $p = 1$, can be solved by calling $Match(i, j)$ for all j $(0 \le j \le n - m)$, which takes $O(mn)$ time.

Algorithm 1 Function $Match(i, j)$

```
1: for l = 0 to m − 1 do
2:     if t_{j+l} ≠ p_{i,l} then
3:         return false
4:     end if
5: end for
6: return true
```

The idea of the Rabin-Karp algorithm is to use a hash function to compute $Match(i, j)$ in $O(1)$ time. The computed hashes reduce the number of executed logical operations by resorting to numerical operations. The hash used in Rabin-Karp algorithm is also known as *rolling hash*. Rabin-Karp algorithm works by computing the hash of each pattern p and then storing it. One character of the text T is hashed at a time and compared to the computed hashed patterns. Spurious hits may occur as two distinct strings may have the same hash values. Hence, when a match is found, a brute-force match verification is necessary to verify whether it is a correct hit or a spurious hit. Gonnet et al. [21] showed that hash collisions are infrequent, making the overhead for verification acceptable.

Let h be a hash function for string $s_0 s_1 \dots s_{m-1}$ such that

$$h(s_0 s_1 \dots s_{m-1}) = (d^{m-1} s_0 + d^{m-2} s_1 + \cdots + d^0 s_{m-1}) \bmod q, \tag{2}$$

where d and q are appropriately selected prime numbers. We choose $d = 2$, which is the alphabet size, and $q =$

13 to explain the examples in this paper. In actual implementations, q must be a larger prime number such as $q = 65521$, because q corresponds to the size of the hash table to compute the hash function. In the Rabin-Karp algorithm, $h(p_0p_1 \dots p_{m-1})$ is computed in advance. For each j $(0 \leq j \leq n-m)$, $h(t_jt_{j+1} \dots t_{j+m-1})$ is computed to determine if it is equal to $h(p_0p_1 \dots p_{m-1})$. Note that if they are not equal, then $Match(i, j)$ never returns true. $Match(i, j)$ may return true only if they are equal. Using this idea, the Rabin-Karp algorithm solves the pattern search problem in $O(n)$ time with high probability. Algorithm 2 shows the Rabin-Karp algorithm for a single input pattern. Note that H_p stores $h(P) = h(p_0p_1 \dots p_{m-1})$. Also, H_t initially stores $h(t_0t_1 \dots t_{m-1})$. They are computed in $O(m)$ time. After the first iteration of the second `for-loop`, H_t stores $(((d^{m-1}t_0 + d^{m-2}t_1 + \cdots + d^0t_{m-1}) \bmod q - d^{m-1}t_0) \cdot d + t_m) \bmod q = (d^{m-1}t_1 + d^{m-2}t2 + \cdots + d^0t_m) \bmod q$, which is equal to $h(t_1t_2 \dots t_m)$. Hence, it should be clear that H_t stores $h(t_jt_{j+1} \dots t_{j+m-1})$ after the j-th iteration. Thus, condition $H_t = H_p$ is equivalent to $h(p_0p_1 \dots p_{m-1}) = h(t_jt_{j+1} \dots t_{j+m-1})$ and this algorithm solves the pattern search problem correctly. If $H_t = H_p$ is false, $Match(i, j)$ is not executed and this iteration of the `for-loop` takes $O(1)$ time. If $H_t = H_p$ is true, $Match(i, j)$ executed and it takes $O(m)$ time. However, the probability that $H_t = H_p$ is very small [21]. Since the values of them are in range $[0, q-1]$, we assume that the $Match(i, j)$ is executed with probability $\frac{1}{q}$. Under this assumption, the Rabin-Karp algorithm runs in $O(m + n + \frac{nm}{q})$ time. Since $m \leq n$ usually holds, the Rabin-Karp algorithm runs in $O(n + \frac{nm}{q})$ time.

Algorithm 2 Rabin-Karp Algorithm [Single Pattern]

```
1: H_p = H_t = 0;
2: for j = 0 to m − 1 do
3:     H_p = (H_p · d + p_j) mod q
4:     H_t = (H_t · d + t_j) mod q
5: end for
6: for j = 0 to n − m − 1 do
7:     if H_t = H_p then
8:         if Match(i, j) then
9:             output(i, j)
10:        end if
11:    end if
12:    H_t = ((H_t − d^{m−1} t_j) · d + t_{j+m}) mod q
13: end for
```

The above Rabin-Karp algorithm for single pattern can be extended to handle multiple patterns. In the Rabin-Karp for a single pattern, $h(P)$ is computed in advance. For multiple patterns $P_0, P_1, \dots, P_{p-1}$, we compute $h(P_k)$ for every k, $(0 \leq k \leq p-1)$. This takes $O(mp)$ time. After that, each iteration of the `for-loop` determines if $H_t = h(P_k)$ holds for every $k (0 \leq k \leq p-1)$. Each iteration takes $O(p)$ time, thus the for loop takes $O(np)$ time. Hence, in total, it takes $O((n+m)p)$ time to perform the matching verification of p patterns. We can accelerate the Rabin-Karp algorithm for multiple patterns using a hash table. Consider p patterns

Algorithm 3 Rabin-Karp Algorithm [Multiple Patterns]

```
1: H_t and HT are computed beforehand
2: for j = 0 to n − m do
3:     if HT(H_t) ≠ −1 then
4:         if Match(i, j) then
5:             output(i, j)
6:         end if
7:     end if
8:     H_t = ((H_t − d^{m−1} t_j) · d + t_{j+m}) mod q
9: end for
```

$P_0, P_1, \dots, P_{p-1}$ and let HT be a hash table of q entries such that

$$HT(r) = \begin{cases} k & \text{if } h(P_k) = r, \\ -1 & \text{otherwise.} \end{cases} \tag{3}$$

The Rabin-Karp algorithm for a single pattern can be modified to run for multiple patterns as follows. If $HT(H_t) = k \neq -1$ then $H_t = h(P_k)$. Thus, this algorithm works correctly. Let us evaluate the computing time. The values of $h(P_k)$ for all k can be computed in $O(mp)$ time. After that the hash table HT is computed in $O(q)$ time. Algorithm 3 shows the Rabin-Karp algorithm for comparing multiple patterns. Note that each iteration of the `for-loop` takes $O(1)$ time if $HT(H_t) = -1$. Otherwise, $Match(i, j)$ is executed in $O(m)$ time. Since the size of the hash table is q and p entries of them have non -1 value, we can assume that the probability that $Match(i, j)$ is executed is $\frac{p}{q}$. Thus, the total computing time for p patterns is $O(mp + q + n + \frac{nmp}{q})$.

3. Intuitive Parallel Rabin-Karp Implementation

This section presents an intuitive parallel Rabin-Karp (IntuitivePRK, for short) algorithm capable of handling multiple patterns. The IntuitivePRK is based on the multiple patterns' version shown in previous section. The details of the IntuitivePRK is presented in Algorithm 4. The IntuitivePRK separates the calculation of the hash pattern and the pattern matching in two distinct functions, called `calculateHashPattern` and `FindMatches`, respectively. To simplify the discussion, in what follows we assume that the number of available threads τ is greater than or equal to the number of patterns p. Function `calculateHashPattern` compute the hash of each pattern P_i $(0 \leq i < p)$ in parallel using a single thread per pattern. The results of each hash is then written in HP array. The `for-loop` in function `calculateHashPattern` runs in $O(\frac{mp}{\tau})$.

Once the HP array is computed, function `FindMatches` divides the text T into S parts, $s_0, s_1, \cdots, s_{\tau-1}$, each containing $\frac{n}{\tau}$ characters. Each thread γ_l $(0 \leq l < \tau)$ is responsible for the processing part of s_i. The initial hash of the fist m characters for each part s_i $(0 \leq i < \tau)$ is computed in the first `for-loop` of function `FindMatches` in parallel. In the second `for-loop`, the hash is recalculated at each iteration using one thread per part s_i. More precisely, at each iteration of the second `for-loop`, the thread

Table 1 Modules for $d = 2$ and $q = 13$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$d^i \bmod q$	1	2	4	8	3	6	12	11	9	5	10	7	1

i	12	13	14	15	16	17	18	19	20	21	22	23	24
$d^i \bmod q$	1	2	4	8	3	6	12	11	9	5	10	7	1

Algorithm 4 Intuitive Parallel Rabin-Karp

```
1: function CALCULATEHASHPATTERN(P[p][m])
2:     hsh = 0
3:     for i = 0 to m − 1 do
4:         hsh = (d · h(hsh) + P[γ_l][i]) mod q
5:     end for
6:     HP[γ_l] = hsh
7: end function
8: function FINDMATCHES(T,HP[p],P)
9:     h(t) = 0
10:    s = n/τ
11:    start = γ_l · s
12:    for i = 0 to m − 1 do
13:        h(t) = (d · h(t) + T[i]) mod q
14:    end for
15:    for j = start to start + s do
16:        for i = 0 to p − 1 do
17:            if h(t) = HP[i] then
18:                if P[i][1] .. P[i]][m]=T[j] .. T[j+m] then
19:                    Match in position j pattern i
20:                end if
21:            end if
22:        end for
23:    end for
24:    t = (d(t − T[j + 1]d^{m−1}) + T[j + m + 1]) mod q
25: end function
26: HP = CALCULATEHASHPATTERN(P)
27: FINDMATCHES(T,HP,P)
```

γ_i moves one character to the right. The inner loop checks whether $h(t_j \cdots t_{j+m-1})$ equals to $h(P_i)$. Clearly, the first `for-loop` in runs in $O(m)$ time, while the second `for-loop` runs in $O(\frac{np}{\tau} + \frac{np}{\tau} \cdot \frac{mp}{q})$. The whole algorithm runs in $O(\frac{mp}{\tau} + \frac{np}{\tau} + \frac{nmp^2}{q\tau})$. As $p \leq \tau \leq n$ usually holds, then the computing time becomes $O(m + n + \frac{nmp}{q})$.

The IntuitivePRK presents a loss in performance for an increasing number of patterns p. The loop of line 15 is the main bottleneck. In this algorithm, a new comparison of hashes for all p patterns is performed at each iteration. Clearly, to improve the performance, ways to parallelize it must be devised. In the next section we present an alternative to circumvent this problem.

4. Prefix-Sum-Based Parallel Rabin-Karp Implementation

This section presents the main contribution of this paper, that is a Prefix-Sum-Based Parallel Rabin-Karp (PRK, for short) implementation for computing multiple patterns. As a key ingredient, we proposed a mechanism to improve the computation of the intermediate hash values. For later reference, we note the following well-known theorem in number theory:

Theorem 4.1: For any two prime numbers d and q, $d^{q-1} \bmod q = 1$ always holds.

For example, for $d = 2$ and $q = 13$, $d^{q-1} \bmod q = 2^{12} \bmod 13 = 1$. From this theorem, $d^i \bmod q = d^{i+(q-1)} \bmod q$ holds. Thus, we have the following corollary:

Corollary 1: For any two prime numbers d and q, and an integer i, $d^i \bmod q = d^{i \bmod (q-1)} \bmod q$ always holds.

For example, for $d = 2$, $i = 15$, and $q = 13$, $d^{15} \bmod q = 8$ and $d^{15 \bmod (13-1)} = 2^3 \bmod 13 = 8$. For $T = t_0 t_1 \ldots t_{n-1}$ let $a_i = d^{n-i-1} t_i$ for all $i (0 \leq i \leq n-1)$ and $\hat{a}_i = a_0 + a_1 + \cdots + a_i$ be the prefix-sum of a. In other words, $\hat{a}_i = d^{n-1} t_0 + d^{n-2} t_1 + \cdots + d^{n-i-1} t_i$. If we have all prefix-sums $\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}$, we can compute the value of hash function $h(t_j t_{j+1} \ldots t_{j+m-1})$ by the following formula:

$$h(t_j t_{j+1} \ldots t_{j+m-1}) = (\hat{a}_{j+m-1} - \hat{a}_{j-1}) \cdot d^{m-n+j}. \tag{4}$$

Since

$$\begin{aligned} \hat{a}_{j+m-1} - \hat{a}_{j-1} &= a_j + a_{j+1} + \cdots + a_{j+m-1} && (5)\\ &= d^{n-j-1} t_j + d^{n-j-2} t_{j+1} + \\ &\quad \cdots + d^{n-j-m} t_{j+m-1} && (6)\\ &= (d^{m-1} t_j + d^{m-2} t_{j+1} + \\ &\quad \cdots + d^0 t_{j+m-1}) \cdot d^{n-j-m} && (7)\\ &= h(t_j t_{j+1} \ldots t_{j+m-1}) \cdot d^{n-m-j}. && (8) \end{aligned}$$

Note that $m - n + j$ may be non-positive.

Suppose that the value of $d^0 \bmod q, d^1 \bmod q, \ldots, d^{q-2} \bmod q$ are stored in an array of size $q - 1$. Once we have this array, we can compute d^i for any integer i by virtue of Corollary 1. Since $0 \leq i \bmod (q-1) \leq q-2$, we can compute $d^i \bmod q$ by reading $(i \bmod (q-1))$-th element of the array. For example, if $d = 2$, $q = 13$ and $i = 100$, instead of computing $d^i \bmod q = 2^{100} \bmod 13$, we can calculate $i \bmod (q-1) = 100 \bmod 12 = 4$ and access the position 4 of array on Table 1 to get the final result 3. That is, the result of $d^i \bmod q$ can be obtained from the $i \bmod (q-1)$ position of array. Note that the values of $d^i \bmod q$ in Table 1 always repeat for $i > q - 1$.

The description of the Parallel Rabin-Karp (PRK) algorithm is presented in Algorithm 5. In what follows, we detail the PRK steps. Our description focuses on a parallel implementation of the proposed algorithm in OpenMP [22] and GPU. The first step loads the lookup table, which is computed previously as explained before (see Table 1). Thus, step 1 runs in $O(q)$ time. In step 2, we calculate the Hash

Algorithm 5 Parallel Rabin-Karp algorithm

1: Load a preprocessed lookup table for $d^i \bmod q$ $(0 \le i \le q-1)$.
2: Compute the values of $h(P_k)$ for all k $(0 \le k \le p-1)$ in parallel and create the hash table HT using the calculated values.
3: Compute the $a_0, a_1, \ldots, a_{n-1}$ in parallel.
4: Compute the prefix-sums $\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}$.
5: For all j $(0 \le j \le n-m)$, compute $(\hat{a}_{j+m-1} - \hat{a}_{j-1}) \cdot d^{m-n-j}$, which is equal to $h(t_j t_{j+1} \ldots t_{j+m-1})$. If array $control[h(t_j t_{j+1} \ldots t_{j+m-1})] \neq 0$ then compare the characters of text and pattern with $Match(i, j)$.

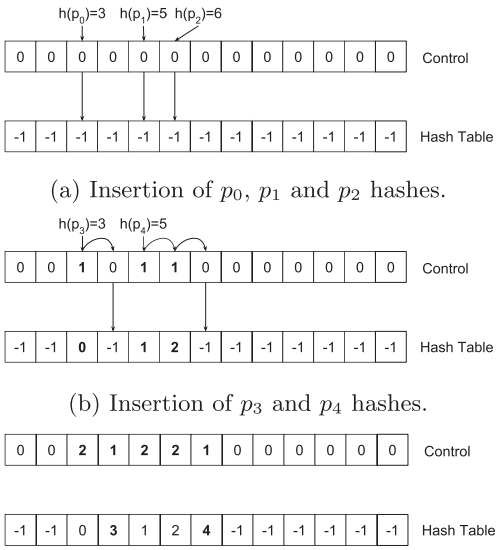


Fig. 1 An example denoting the insertion of 5 elements in the hash table.

Table (HT) considering that two different patterns can have the same hash value. In this step the hashes of each pattern $h(P_k)$ are calculated in parallel. To ensure that the data is correct, two arrays are used, one for control and one being the HT itself. Figure 1 shows an example of insertion of 5 patterns. In the control array an atomic operation is performed in the respective position to the hash of that pattern. In such way, this instruction allows a thread to increment the value of a variable and receives its old value atomically. The control array is initialized with zeros and the Hash Table initializes with -1. Figure 1 (a) shows the initialized arrays and the inclusion of 3 hashes. Atomic addition is used in the respective positions of the calculated hashes. As the return of the atomic operation in the 3 cases is 0, the threads write the pattern ID in the same position on HT, as shown in bold in Fig. 1 (b). In the same figure, two other hashes are inserted, but the return of the atomic operation is different from 0. In this case, the threads repeat the operation for the next position of control array, as shown in Fig. 1 (c). In the case of $h(p_3)$, the next position already returns 0 and the ID 3 is recorded in this position. For $h(p_4)$, we have a return value different from 0 in the calculated hash and also in the next position, so we have to write two positions ahead of the calculated hash.

In step 3, each $a_i = d^{n-i-1} \cdot t_i$ is calculated in parallel.

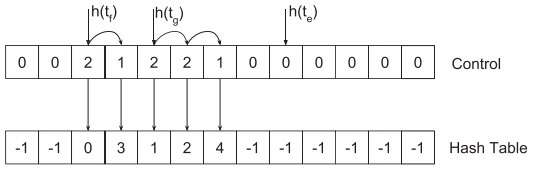


Fig. 2 Match verification at positions t_e, t_f and t_g.

Next, in step 4, the prefix-sum of the values of the previous step are computed. A divide-and-conquer approach has been used to compute the prefix-sums in OpenMP. In the case of GPU, the CUDA UnBounded library (CUB) [23] is used. CUB is a C++ library that provides efficient kernels that can be used for different GPU applications and architectures. In this work, we use the *"decoupled look-back"* algorithm to calculate the sum of global prefixes [20]. The code was slightly modified so that the sum of two terms a and b in the prefix-sum was calculated using $(a + b) \bmod q$.

In step 5, each thread is independent and the hash of each part of the text $h(t_j t_{j+1} \ldots t_{j+m-1})$ is calculated in parallel. This hash is computed through the Eq. (4) using the prefix-sum calculated in step 4. With the hash calculated, it is possible to check whether the value of the control array is different from zero in that position. In case it is 1, we check the HT only in that position. If it is greater than 1, we check the pattern relative to that HT position and the process starts again at the next position. In case the hashes are equal, the $Match()$ function is used to compare the characters to prevent against false positives. Figure 2 illustrates this process. In this example, the hash of position e of the text has value equal to 0, which indicates there is no pattern occurrence with this hash. For the hash position f, it is necessary to check the calculated position and, as the return is greater than 1 we also check the following position. The same logic is applied to position g, where 3 positions must be checked.

Let us evaluate the execution time of Algorithm 5. Recall that step 1 runs in $O(q)$ time. To compute the hashes $h(P_k)$ $(0 \le k \le p-1)$ and fill the HT, step 2 takes $O(\frac{mp}{\tau} + q)$ time. Step 3 runs in $O(\frac{n}{\tau})$ since each term a_i $(0 \le i \le n-1)$ is independent. Step 4 runs in $O(\frac{n}{\tau})$. In step 5, we have a main loop with n iterations over τ threads, and within this loop we call the $Match$ function for $\frac{p}{q}$ times, totalizing $O(\frac{n}{\tau} + \frac{n}{\tau} \cdot \frac{mp}{q})$. Thus, the complexity of all steps is $O(\frac{mp}{\tau} + q + \frac{n}{\tau} + \frac{nmp}{q\tau})$. Since $p \le \tau \le n$ usually holds, then the computing time becomes $O(m + q + \frac{n}{\tau} + \frac{nm}{q})$. The following theorem summarizes the discussion above.

Theorem 4.2: Given a text of length n and p patterns of length m, the proposed prefix-sum based Rabin-Karp algorithm finds all occurrences of p in n in $O(m + q + \frac{n}{\tau} + \frac{nm}{q})$ time, where q is a sufficiently large prime number and τ is the available number of threads.

5. Experimental Results

The main purpose of this section is to show the experimen-

Table 2 Runtime (ms) results for the implementations of the IntuitivePRK and PRK as well as the achieved GPU and OpenMP speedup in comparison to the sequential implementation.

Implementation		IntuitivePRK			PRK		
CPU/GPU	p	$m = 10$	$m = 20$	$m = 30$	$m = 10$	$m = 20$	$m = 30$
Sequential	1	712.96	725.56	720.58	1685.71	1868.27	1871.96
	4	726.25	734.86	729.99	1704.31	1874.91	1854.88
	16	1925.76	1860.33	1,844.50	1711.93	1893.02	1881.48
	64	4,679.32	4,343.91	4,607.87	1845.50	1883.84	1869.63
	256	17505.80	17130.00	17085.70	2441.45	1968.87	1931.52
OpenMP	1	13.90	12.95	12.79	50.50	53.97	54.00
	4	22.48	16.95	18.38	52.55	54.45	54.39
	16	35.57	29.30	29.84	59.11	53.72	53.58
	64	105.59	102.05	101.58	65.70	54.70	54.03
	256	419.69	361.19	359.35	78.31	65.89	65.26
GPU	1	9.13	10.05	10.83	3.54	5.06	5.04
	4	10.37	11.48	12.32	3.73	5.07	5.08
	16	11.37	12.32	13.44	4.27	5.12	5.08
	64	18.05	15.28	15.97	4.90	5.14	5.12
	256	52.56	44.54	46.42	5.43	5.22	5.18
				Speedup			
OpenMP Speedup to Sequential	1	51.29	56.03	56.34	33.38	34.62	34.67
	4	32.31	43.35	39.72	32.43	34.43	34.10
	16	54.14	63.49	61.81	28.96	35.24	35.12
	64	44.32	42.57	45.36	28.09	34.44	34.60
	256	41.71	47.43	47.55	31.18	29.88	29.60
GPU Speedup to Sequential	1	78.09	72.20	66.54	476.19	369.22	371.42
	4	70.03	64.01	59.25	456.92	369.80	365.13
	16	169.37	151.00	137.24	400.92	369.73	370.37
	64	259.24	284.29	288.53	376.63	366.51	365.16
	256	333.06	384.60	368.07	449.62	377.18	372.88

tal results of the parallel Rabin-Karp presented in the previous sections. The experimental results have been carried out on the NVIDIA Tesla V100, which comprises of 5120 processing cores, running at 1.380GHz with 16GB HBM2 memory. The source code programs of the GPU implementation are compiled using the nvcc version 9.2 with -O2 and -arch=sm_70 options on Ubuntu release 16.04. For comparison purpose, the OpenMP and sequential versions of the algorithms presented in the previous sections have been implemented on a multi-core server with 4 icosa-core (20-core) Intel Xeon E7 − 8870 v4 CPUs running at 2.10 GHz. This multi-core server has $4 \times 20 = 80$ physical cores each of which acts as 2 logical cores via hyper-threading technology. The OpenMP and sequential algorithms have been executed on this machine. For the OpenMP experiments, $\tau = 128$ threads have been used. The proposed PRK GPU implementation is also compared to GPU implementation proposed in [19].

In the experiments and simulations, we considered $p = 1, 4, 16, 64, 256$ patterns with $m = 10, 20, 30$ characters each. The input string T has $n = 2^{27}$ characters ($\approx$ 128 Mbytes). The input string T and the p patterns $P_0, P_1, \ldots P_{p-1}$ are randomly generated over the alphabet size $d = 2$. The results are averaged over 20 runs with different seeds for each run. For both intuitivePRK (Algorithm 4) and PRK (Algorithm 5), the input parameters are stored in the global memory. For the PRK, these parameters also include the preprocessed lookup table of step 1.

Table 2 shows the execution time results for the intuitivePRK and PRK presented in previous sections. More precisely, the table shows the runtime for the sequential, OpenMP and GPU implementations as well as the GPU and OpenMP speedup over the sequential implementations. In what follows, let us analyze the runtime performance of the intuitivePRK. As discussed in Sect. 3, the number of patterns p was expected to have a stronger influence on execution time. Comparing the results for $p = 1$ to $p = 256$ with $m = 30$, sequential and OpenMP implementations increase the execution time on more than 23 times. In the GPU, the runtime increase for this case is below 5 times. In terms of the number τ of threads, Function `calculateHashPattern` uses one CUDA thread per pattern to compute the values of $h(P_k)$, while Function `FindMatches` uses 256 threads in 1024 CUDA blocks. Note that an increase on the pattern size m may not increase the computation time. Indeed, for $p \geq 16$, the runtime results for $m = 20$ express better results than that of $m = 10$. The reason behind it is mainly attributed to the number of spurious hits and real matches. As the pattern size increases from 10 to 20, the number of hits and matches reduces significantly, which impacts the results. For the sequential results of the IntuitivePRK, the gcc optimization flag -O2 has been used instead of the usual -O3 as it provided better results. Table 2 presents the speedup results for OpenMP and

Table 3 PRK and MatchStr [19] runtime results (ms) on the GPU with $m = 30$.

	PRK						
p	Step 2	Step 3	Step 4	Step 5	Total	MatchStr	Speedup
1	0.08	1.17	1.17	2.62	5.04	5.69	1.13
4	0.08	1.18	1.17	2.65	5.08	5.90	1.16
16	0.08	1.18	1.16	2.66	5.08	12.60	2.48
64	0.08	1.17	1.16	2.71	5.12	49.16	9.60
256	0.08	1.17	1.16	2.77	5.18	193.46	37.35

GPU as compared to the Sequential implementation. Comparing the parallel implementation to the sequential one, the results show a significant improvement of OpenMP and GPU over the sequential. For $p = 256$ and $m = 30$, OpenMP and GPU provided speedup surpassing 46 and 368 times, respectively.

For the PRK, the parameters $q = 65521$, which is the largest prime number less than 2^{16}, and $d = 2$ were used. In terms of the number τ of threads, step 2 of the PRK uses one CUDA thread for each pattern and compute the values of $h(P_k)$. In step 3, we use 256 threads in 128 CUDA blocks to improve occupancy, which is defined as the ratio of active warps on a stream multiprocessor (SM) to the maximum number of active warps supported by the SM. In step 4, we use the prefix-sum of the CUDA UnBounded (CUB) library version 1.7.3 [23]. In step 5, we also used 256 threads with 128 blocks for best occupancy. Table 2 shows the runtime results for the proposed PRK algorithm. As before, sequential and parallel versions of the PRK algorithm have been implemented. Contrarily to the IntuitivePRK, the number of patterns p has a minor impact on the PRK performance. The pattern size m does not degrade significantly the performance of the PRK both for the sequential and parallel implementations. In fact, for $m = 20$ and $m = 30$, the sequential and parallel implementations have shown to be competitive in performance. In terms of speedup, both OpenMP and GPU provided speedups surpassing 29 times and 372 times over the sequential implementation, respectively, for $p = 256$ and $m = 30$. Considering the parallel implementations for the PRK, the GPU provided speedup of 12.59 times over the OpenMP implementation for $p = 256$ and $m = 30$. Overall, the PRK implementation on GPU provided speedup surpassing 10 times over the OpenMP implementation. The PRK algorithm achieved an average occupancy of 0.94, 0.72 and 0.68 for steps 3, 4 and 5, respectively. These results show that the PRK implementation attains a high level of thread parallelism.

Comparing the results in Table 2 and considering $p \leq 16$, the intuitivePRK provided better results for the sequential and OpenMP implementations as compared to the PRK implementation. This is due to the fact that the proposed PRK requires more steps for computing the prefix-sum in the sequential and OpenMP implementation as compared to the IntuitivePRK. On the other hand, the GPU may use optimized implementations for computing the prefix-sums, such as the CUDA UnBounded library (CUB) [23]. Indeed, the PRK, GPU implementation, provided better results than that of the IntuitivePRK for all p and m values. Considering the GPU implementations, the PRK provided gains surpassing 2.14 times for $p = 1$ and $m = 30$ and 8.96 times for $p = 256$ and $m = 30$ over the IntuitivePRK Implementation.

The PRK execution time for step 2 to 5 have been recorded and averaged. The computing time for each step of the PRK algorithm with $m = 30$ is shown in Table 3. As can be seen in the table, steps 2 to 4 have similar execution time independently of the number of patterns p. In fact, the same occurs for other values of m, not shown due to space limitation. Note that steps 3 and 4 have similar computational complexity and their execution time on the GPU was expected to be similar, particularly due to the use of the CUB for computing the prefix-sums in step 4. Step 5 is more sensitive to changes in pattern size and number of patterns. Indeed, this step incurs in computing hash values as well as to compare the characters to verify possible false positive occurrence. Nevertheless, step 5 has an average runtime of 2.77ms for $p = 256$ while the average execution time for $p = 1$ is 2.62ms. That is, the average difference is less than 6%, even though the number of input patterns increased from 1 to 256. The table also shows comparison results of the PRK to the GPU implementation proposed in [19], hereafter referred to as "MatchStr". The latter has been reported to perform well on shorter text patterns, which is the case in our experiments, making it a reasonable choice for comparison purposes. For the reader benefit, in what follows a brief overview of the MatchStr is provided. At the CPU side, the MatchStr algorithm arranges the input string T and pattern strings p into string arrays. More precisely, the input text string T is broken into $n - m$ sub-strings of size m, which are arranged in a two-dimensional array, called "textArr". Each column of the textArr holds a sub-string of T. Next, these arrays are transferred to the GPU. To each column of the textArr, a thread γ_l $(0 \leq l < \tau)$ is assigned. Each thread compares the characters in its column with those in the input pattern P. If all characters match, the results are then registered into the result array and transferred back to CPU at the end of the process. Clearly, MathSTR runs in $O(nm/\tau)$ time. Note that, in this arrangement, MatchSTR performs coalesced memory accesses and avoids memory block conflicts. Contrary to the PRK algorithm, the MatchStr does not produce false positives, as the text to pattern match is perform character-by-character. In this work, MatchStr implementation has been adapted to handle multiple patterns (i.e. $p > 1$) by issuing multiple kernel calls. Thus, the MatchStr for handling multiple patters runs in $O(pmn/\tau) = O(mn)$, for $p \leq \tau$. As can be observed in Table 3, PRK attains speedup gains over the MatchSTR varying from 1.13 times for $p = 1$

up to 37.35 times for $p = 256$. That is, even for the case of $p = 1$, the proposed PRK implementation on GPU achieves significant improvement over MatchStr.

6. Conclusion

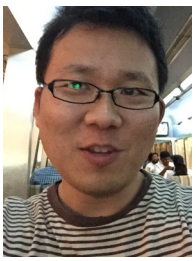
This work addressed the problem of multiple-pattern matching on GPUs. More precisely, we proposed a Prefix-Sum-Based Parallel Rabin-Karp (PRK, for short) algorithm tailored for parallel execution on the GPU. At its core, PRK uses a fast-parallel prefix-sums algorithm to maximize parallelization together with a look-up table to accelerate the task of matching on multiple patterns. The proposed PRK algorithm finds all occurrences of p patterns of length m in an input text T in $O(m + q + \frac{n}{\tau} + \frac{nm}{q})$ time, where q is a sufficiently large prime number and τ is the available number of threads. Both sequential and parallel versions of the PRK algorithm have been implemented. Experimental results show that the parallel implementation of the PRK algorithm on the NVIDIA V100 GPU provides speedup surpassing 372 times and 12.59 times as compared to the sequential and OpenMP implementations, respectively. Compared to another prominent GPU implementation, PRK attained speedup surpassing 37 times. As future work, we plan to apply the insights obtained in this work on correlated problems.

References

[1] L.S.N. Nunes, J.L. Bordim, Y. Ito, and K. Nakano, "A Prefix-Sum-Based Rabin-Karp implementation for multiple pattern matching on GPGPU," in 2018 Sixth International Symposium on Computing and Networking (CANDAR), pp.139–145, IEEE, Nov. 2018.
[2] E. Ukkonen, "Algorithms for approximate string matching," Information and Control, vol.64, no.1-3, pp.100–118, 1985.
[3] L.-L. Cheng, D.W. Cheung, and S.-M. Yiu, "Approximate string matching in DNA sequences," in Eighth International Conference on Database Systems for Advanced Applications (DASFAA), pp.303–310, IEEE, March 2003.
[4] H. Gharaee, S. Seifi, and N. Monsefan, "A survey of pattern matching algorithm in intrusion detection system," in 7th International Symposium on Telecommunications (IST), pp.946–953, IEEE, Sept. 2014.
[5] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson, Introduction to Algorithms, 2nd ed., McGraw-Hill Higher Education, 2001.
[6] R.S. Boyer and J.S. Moore, "A fast string searching algorithm," Communications of ACM ACM, vol.20, no.10, pp.762–772, Oct. 1977.
[7] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," Communications of ACM, vol.18, no.6, pp.333–340, June 1975.
[8] R.M. Karp and M.O. Rabin, "Efficient randomized pattern-matching algorithms," IBM Journal of Research and Development, vol.31, no.2, pp.249–260, March 1987.
[9] D.E. Knuth, J.H. Morris, and V.R. Pratt, "Fast pattern matching in strings," SIAM Journal on Computing, vol.6, pp.323–350, March 1977.
[10] E.A. Sitaridi and K.A. Ross, "GPU-accelerated string matching for database applications," The VLDB Journal, vol.25, no.5, pp.719–740, Oct. 2016.
[11] W.W. Hwu, GPU Computing Gems Emerald Edition, 1st ed., Morgan Kaufmann Publishers, San Francisco, CA, USA, 2011.
[12] X. Zha and S. Sahni, "GPU-to-GPU and Host-to-Host multipattern string matching on a GPU," IEEE Trans. Comput., vol.62, no.6, pp.1156–1169, June 2013.
[13] N. Jacob and C. Brodley, "Offloading IDS computation to the GPU," in 22nd Annual Computer Security Applications Conference (ACSAC'06), pp.371–380, IEEE, Dec. 2006.
[14] 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 2010 IEEE Global Telecommunications Conference GLOBECOM 2010, pp.1–5, IEEE, Dec. 2010.
[15] J. Sharma and M. Singh, "CUDA based Rabin-Karp pattern matching for deep packet inspection on a multicore GPU," International Journal of Computer Network and Information Security, vol.7, no.10, pp.70–77, Sept. 2015.
[16] C.S. Kouzinopoulos, P.D. Michailidis, and K.G. Margaritis, "Multiple string matching on a GPU using CUDAs," Scalable Computing: Practice and Experience, vol.16, no.2, pp.121–127, 2015.
[17] S. Ashkiani, N. Amenta, and J.D. Owens, "Parallel approaches to the string matching problem on the GPU," in Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, pp.275–285, ACM, 2016.
[18] P. Shah and R. Oza, "Improved parallel Rabin-Karp algorithm using compute unified device architecture," in Information and Communication Technology for Intelligent Systems (ICTIS 2017) - Volume 2, pp.236–244, Springer International Publishing, 2018.
[19] N. Dayarathne and R. Ragel, "Accelerating Rabin Karp on a Graphics Processing Unit (GPU) using Compute Unified Device Architecture (CUDA)," in 7th International Conference on Information and Automation for Sustainability, pp.1–6, IEEE, Dec. 2014.
[20] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled look-back," Technical Report, NVIDIA Corporation, March 2016.
[21] G.H. Gonnet and R.A. Baeza-Yates, "An analysis of the Karp-Rabin string matching algorithm," Inf. Process. Lett., vol.34, no.5, pp.271–274, May 1990.
[22] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," IEEE Computational Science and Engineering, vol.5, no.1, pp.46–55, Jan. 1998.
[23] D. Merrill, "CUB: CUDA unbound, a library of warp-wide, block-wide, and device-wide GPU parallel primitives," NVIDIA Research, 2018.

Lucas Saad Nogueira Nunes received B.E. degree in Computer Engineering in 2015 from University of Brasilia. Mr. Nunes is currently enrolled in the Master Course in Informatics at the University of Brasilia where he is pursuing his Masters Degree. His interest includes reconfigurable architectures, parallel computing, and algorithms and architectures.

Jacir Luiz Bordim received B.Sc. and M.Sc. degrees in Computer Science in 1994 and 2000, respectively. Received the Ph.D. degree in Information Science from Japan Advanced Institute Of Science And Technology in 2003, with honors. He worked as a researcher at ATR-Japan from 2003 to 2005. Since 2005 he is an Associate Professor with the Department of Computer Science at University of Brasilia. Dr. Bordim has published and served in many international conferences and journal. His interest includes mobile computing, collaborative computing, trust computing, distributed systems, opportunistic spectrum allocation, MAC, routing protocols and reconfigurable computing.



Yasuaki Ito received B.E. degree from Nagoya Institute of Technology (Japan), M.S. degree from Japan Advanced Institute of Science and Technology in 2003, and D.E. degree from Hiroshima University (Japan), in 2010. From 2004 to 2007 he was a Research Associate at Hiroshima University. Since 2007, Dr. Ito has been with the School of Engineering, at Hiroshima University, where he is working as an Associate Professor. His research interests include reconfigurable architectures, parallel computing, computational complexity and image processing.

Koji Nakano received the BE, ME and Ph.D degrees from Department of Computer Science, Osaka University, Japan in 1987, 1989, and 1992 respectively. In 1992–1995, he was a Research Scientist at Advanced Research Laboratory. Hitachi Ltd. In 1995, he joined Department of Electrical and Computer Engineering, Nagoya Institute of Technology. In 2001, he moved to School of Information Science, Japan Advanced Institute of Science and Technology, where he was an associate professor. He has been a full professor at School of Engineering, Hiroshima University from 2003. He has published extensively in journals, conference proceedings, and book chapters. He served on the editorial board of journals including IEEE Transactions on Parallel and Distributed Systems, IEICE Transactions on Information and Systems, and International Journal of Foundations on Computer Science. His research interests include image processing, hardware algorithms, GPU-based computing, FPGA-based reconfigurable computing, parallel computing, algorithms and architectures.