Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 7, Number 1, pages 28–32, January 2018

Parallel Rabin-Karp Algorithm Implementation on GPU (preliminary version)

Lucas S. N. Nunes, Jacir L. Bordim
Department of Computer Science
University of Brasília
70910-900 Brasília - DF - Brazil
{saad,bordim}@unb.br

Yasuaki Ito, Koji Nakano
Department of Information Engineering
Hiroshima University
1-4-1 Kagamiyama, Higashi-Hiroshima, 739-8527, Japan
{yasuaki,nakano}@cs.hiroshima-u.ac.jp

***Abstract*—The task of finding strings that match to a given pattern is of interest in a variety of practical applications, including DNA sequencing and text searching. Owing to its importance, alternatives to accelerate the pattern matching task have been widely investigated in the literature. The main contribution of this work is to present a parallel version of the celebrated Rabin-Karp algorithm. Given a pattern P of size m and a text string T of size n, the Rabin-Karp algorithm finds all occurrences of the pattern P in T with high probability. The proposed scheme can compare k different patterns to the input text concurrently. The proposed, parallelized, version of the Rabin-Karp algorithm has been implemented on the GeForce GTX 960 GPU. The results obtained show that the proposed implementation provides acceleration, compared to a sequential (CPU) implementation, surpassing 140 times for $k = m = 1024$ and $n = 2^{22}$.**

***Index Terms*—Pattern matching, Rabin-Karp algorithm, parallel implementation, GPU, CUDA.**

I. Introduction

String or pattern matching algorithms are used to find the occurrences of a pattern in a given text or a set of input strings [1]. The task of finding strings having a complete or even a partial match to a given pattern has several practical applications, including DNA sequencing, detecting plagiarism, text mining, spam filtering and so on [1], [2], [3]. Suppose that a pattern P and a string T of length m and n ($m \ll n$), respectively, are given. The pattern matching is a task to find all occurrences of the pattern P in T. Brute-force algorithms for the pattern matching perform character comparisons between the scanned text substring and the complete pattern from left to right. In the case of a mismatch or a complete match it shifts exactly one position to the right. Hence, brute-force algorithm runs in $O(mn)$ time [4]. String searching algorithms such as Aho-Corasick [5], Boyer Moore [6], Knuth-Morris-Pratt [7] and Rabin-Karp [8] are well-known and widely used. These algorithms improve the running time to find a matching by avoiding rescanning the input string T. For instance, the Rabin-Karp algorithm can solve the pattern search problem in $O(n)$ time with high probability. Pattern matching algorithms can be categorized as single pattern matching and multiple pattern matching algorithms. The Rabin-Karp algorithm is a string searching algorithm that be applied to both contexts.

To accelerate the pattern matching computation, GPU (Graphics Processing Unit) implementations have been explored in the literature [9], [10]. The GPU is a specialized circuit designed to accelerate computation for building and manipulating images [11]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. The Compute Unified Device Architecture (CUDA) is a parallel computing architecture provided by NVIDIA [12]. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements. NVIDIA GPUs have a number of streaming multiprocessors (SMs) that can execute multiple threads in parallel. CUDA uses different types of memories in the NVIDIA GPUs, of particular importance is the *shared memory* and the *global memory* [12]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. To maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid *bank conflicts*. To maximize the bandwidth between the GPU and the global memory, the consecutive addresses must be accessed at the same time. Thus, CUDA threads should perform *coalesced* access when they access the global memory. When no bank conflict occurs, the shared memory provides a much lower latency than uncached global memory. Clearly, to accelerate applications using GPUs, one needs to pay special attention to shared memory and global memory accesses.

A CUDA C implementation of the Rabin-Karp algorithm is presented in [13]. In the paper, the authors compare the average execution times of a serial, GPU and Pthread parallel implementation of Rabin-Karp algorithm on a random DNA sequence data. Their experiments considered random DNA text sequence of 2 Mbytes in length (n) and pattern sizes up 800 base-pair (bp) in length (m). The results show speedup improvements up to 15.68 times for 25 bp and up to 7.58 times for 800 bp. Interestingly, their results show a rapidly decreasing performance for larger m in the GPU. The main contribution of this work is to propose a parallel algorithm for computing the pattern matching problem on the GPU. More precisely, we parallelized the Rabin-Karp algorithm. As a key ingredient, we proposed a mechanism to improve the computation of the intermediate hash values. The proposed scheme can compare k different patterns to the input text concurrently.

The parallelized version of the Rabin-Karp algorithm has been implemented on the GeForce GTX 960 GPU [14]. The results obtained by the "nvprof" profiling tool [15] shows that the proposed implementation provides acceleration surpassing 140 times for as compared to a sequential (CPU) implementation for $k = m = 1024$, $n = 2^{22}$.

The rest of this paper is organized as follows. Section II defines pattern search problem and presents a simple matching function. Section III presents an overview of the Rabin-Karp algorithm and lays the foundation for the proposed parallel algorithm. Section IV presents the proposed parallel Rabin-Karp algorithm on the GPU. Experimental results are shown in Section V. Finally, Section VI concludes this work.

II. Pattern Search Problem

Let $T = t_0t_1 \ldots t_{n-1}$ be a string of n characters (8-bit unsigned integers). Let $P_0, P_1, \ldots, P_{p-1}$ be p patterns, such that each $p_{i,0}p_{i,1} \ldots p_{i,m-1}$ $(0 \leq i \leq p-1)$ is a string of m characters. The pattern search problem is to find all matching position in T for all patterns. More specifically, we find all pairs (i, j) of position j and pattern P_i such that

$$t_jt_{j+1} \ldots t_{j+m-1} = p_{i,0}p_{i,1} \ldots p_{i,m-1}. \qquad (1)$$

First, assume that $p = 1$. Clearly, we have only one pattern $P = p_0p_1 \ldots p_{m-1}$ of m characters for the pattern search problem. Let $EQ(j)$ be a function such that it returns true if and only if Equation (1) is satisfied. Figure 1 shows a possible implementation of function $EQ(j)$. This straightforward implementation can compute $EQ(j)$ in $O(m)$ time. Clearly, the pattern search problem can be solved by calling $EQ(j)$ for all j $(0 \leq j \leq n-m)$, which takes $O(mn)$ time.

III. Rabin Karp Algorithm

The idea of the Rabin-Karp algorithm is to use a hash function to compute $EQ(j)$ in $O(1)$ time. Let h be a hash function for a string $s_0s_1 \ldots s_{m-1}$ such that

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

where d and q are appropriately selected prime numbers. We choose $d = 2$ 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, $h(p_0p_1 \ldots p_{m-1})$ is computed in advance. For

Function $EQ(j)$

```
for $k \leftarrow 0$ to $m-1$ do
    if $t_{j+k} \neq p_k$ return false;
return true;
for $j \leftarrow 0$ to $n-m$ do
    if $EQ(j)$ then output($j$);
```

Figure 1: Straightforward pattern search algorithm

each j $(0 \leq j \leq n-m)$, $h(t_jt_{j+1} \ldots t_{j+m-1})$ is computed to determine if it is equal to $h(p_0p_1 \ldots p_{m-1})$. Note that if they are not equal, then $EQ(j)$ never returns true. $EQ(j)$ 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.

Figure 2 shows the Rabin-Karp algorithm for a single input pattern. Note that H_p stores $h(P) = h(p_0p_1 \ldots p_{m-1})$. Also, H_t initially stores $h(t_0t_1 \ldots t_{m-1})$. They are computed in $O(m)$ time. After the first iteration of the second `for-loop`, H_t stores $(((d^{m-1}t_0 + d^{-2}t_1 + \cdots + d^0t_{m-1}) \bmod q - d^{m-1}t_0) \cdot d + t_m) \bmod q = (d^{m-1}t_1 + d^{-2}t2 + \cdots + d^0t_m) \bmod q$, which is equal to $h(t_1t_2 \ldots t_m)$. Hence, it should be clear that H_t stores $h(t_jt_{j+1} \ldots t_{j+m+1})$ after the j-th iteration. Thus, condition $H_t = H_p$ is equivalent to $h(p_0p_1 \ldots p_{m-1}) = h(t_jt_j + 1 \ldots t_{j+m-1})$ and this algorithm solves the pattern search problem correctly. If $H_t = H_p$ is false, $EQ(j)$ is not executed and this iteration of the for-loop takes $O(1)$ time. If $H_t = H_p$ is true, $EQ(j)$ executed and it takes $O(m)$ time. However the probability that $H_t = H_p$ is very small. Since the values of them are in range $[0, q-1]$, we assume that the $EQ(j)$ is executed with probability $\frac{1}{q}$. Under this assumption, the Rabin-Karp algorithm runs $O(m + n + \frac{m}{q}) = O(n + m)$ time.

We can extend the Rabin-Karp algorithm for multiple patterns. In Rabin-Karp for a single pattern, $h(P)$ is computed in advance. For multiple patterns $P_0, P_1, \ldots, P_{p-1}$, 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 $H_t = h(P_k)$ for every $k(0 \leq k \leq p-1)$. Each iteration takes $O(p)$ time, thus the for loop takes $O(np)$ time. Thus, the total computing time is $O((n+m)p)$ time for p patterns. We can accelerate the Rabin-Karp algorithm for multiple patterns using a hash table. For p patterns $P_0, P_1, \ldots, P_{p-1}$, let HT be a hash table of q entries such that

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

For simplicity, we assume no collision, that is, $h(P_k) \neq$

Rabin-Karp Algorithm [Single Pattern]

```
$H_p \leftarrow H_t \leftarrow 0$;
for $j \leftarrow 0$ to $m-1$ do {
    $H_p \leftarrow (H_p \cdot d + p_j) \bmod q$;
    $H_t \leftarrow (H_t \cdot d + t_j) \bmod q$;
}
for $j \leftarrow 0$ to $n-m$ do {
    if $H_t = H_p$ then {
        if $EQ(j)$ then output($j$);
    $H_t \leftarrow ((H_t - d^{m-1}t_j) \cdot d + t_j + m) \bmod q$;

}
```

Figure 2: Rabin-Karp algorithm for a single pattern.

Table I: 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

Rabin-Karp Algorithm [Multiple Patterns]

```
/* $H_t$ and $HT$ are computed beforehand */
for $j \leftarrow 0$ to $n - m$ do {
    if $HT(H_t) \neq -1$ then{
        if $EQ(i, j)$ then output$(i, j)$;
        $H_t \leftarrow ((H_t - d^{m-1}t_j) \cdot d + t_{j+k}) \bmod q$;
    }
}
```

Figure 3: Rabin-Karp algorithm for multiple patterns.

$h(P_{k'})$ for all k and k' such that $k \neq k'$. Let $EQ(i, j)$ denote the execution of $EQ(j)$ for pattern P_i. We can simply modify the Rabin-Karp algorithm for a single pattern 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. Figure 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, $EQ(i, j)$ is executed in $O(m)$ time. Since the size of the hash table is q and m entries of them have non -1 value, we can assume that the probability that $EQ(i, j)$ is executed is $\frac{m}{q}$. Thus, the total computing time for p patterns is $O(mp + q + n + \frac{nm}{q})$.

IV. Parallel Rabin-Karp Algorithm

Let us parallelize Rabin-Karp algorithm. For later reference, we note the following well-known theorem in number theory:

Theorem 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 2. *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 $T = t_0 t_1 \dots t_{n-1}$ 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, $\hat{a}_i = d^{n-1}t_0 + d^{n-2}t_1 + \cdots + d^0 t_{n-1}$. If we have all prefix-sums $\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1}$, we can compute the value of hash function $h(t_j t_{j+1} \dots t_{j+m-1})$ by the following formula:

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

Since

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

we can confirm that the above Equation (4) is correct. Note that $m - n + j$ may be non-positive.

Suppose that the value of $d^0 \bmod q, d^1 \bmod q, \dots, d^{q-2} \bmod q$ 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 2. 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 I 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 I always repeat for $i > q - 1$.

The parallel Rabin-Karp algorithm can be described as follows:

- **Step 1** Load a preprocessed lookup table for $d^i \bmod q$ $(0 \leq i \leq q-1)$.
  Step 2 Compute the values of $h(P_k)$ for all k $(0 \leq k \leq p-1)$ in parallel and create the hash table HT using the calculated values.
- **Step 3** Compute the $a_0, a_1, \dots, a_{n-1}$ in parallel.
- **Step 4** Compute the prefix-sums $\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1}$ by parallel scan.
- **Step 5** For all j $(0 \leq j \leq n-m)$, compute $(\hat{a}_{j+m-1} - \hat{a}_{j-1}) \cdot d^{m-n-j}$, which is equal to $h(t_j t_{j+1} \dots t_{j+m-1})$. If $HT(h(t_j t_{j+1} \dots t_{j+m-1})) \neq -1$ then evaluate $EQ(i, j)$ and outputs (i, j) if $EQ(i, j)$ is true.

The prefix computation in **Step** 1 and in **Step** 4 can be done very efficiently using prefix scan algorithm presented in [16], [17]. In what follows we present the implementation details of the parallel Rabin-Karp algorithm on the GPU as well as the experimental results.

V. Experimental Results

The main purpose of this section is to show the experimental results of the parallel Rabin-Karp on the GeForce GTX 960 GPU [14]. The GTX 960 has 8 streaming multiprocessors and 2GB of memory. In the experiments, the compiler nvcc

Table III: Running time results (ms) for the parallel Rabin-Karp on the GTX 960 and CPU, with parameters $d = 2$, $q = 65521$, $k = 512$, and $n = 2^i$, $(22 \leq i \leq 27)$.

n	**Implementation**	**Step** 2	**Step** 3	**Step** 4	**Step** 5	**Total**	Speed-up
2^{22}	CPU	4.53	34.53	26.41	92.37	157.83	118.12
	GPU	0.04	0.19	0.49	0.61	1.34	
2^{23}	CPU	4.52	68.54	52.88	184.30	310.25	126.26
	GPU	0.04	0.46	0.75	1.21	2.46	
2^{24}	CPU	4.59	138.38	107.55	373.98	624.50	130.64
	GPU	0.03	1.16	1.18	2.41	4.78	
2^{25}	CPU	4.54	274.39	213.38	743.02	1235.33	127.47
	GPU	0.04	2.77	2.08	4.80	9.69	
2^{26}	CPU	4.53	558.04	426.62	1496.06	2485.25	126.53
	GPU	0.04	6.36	3.62	9.61	19.64	
2^{27}	CPU	4.52	1159.64	850.68	2974.56	4989.40	130.65
	GPU	0.04	13.90	6.83	17.42	38.19	

Table IV: Running time results (ms) for the parallel Rabin-Karp on the GTX 960, with parameters $d = 2$, $q = 65521$, $k = 1024$, and $n = 2^i$, $(22 \leq i \leq 27)$.

n	**Implementation**	**Step** 2	**Step** 3	**Step** 4	**Step** 5	**Total**	Speed-up
2^{22}	CPU	10.37	40.53	30.29	112.30	193.49	140.70
	GPU	0.05	0.21	0.50	0.61	1.38	
2^{23}	CPU	9.33	73.96	56.75	209.27	349.31	141.58
	GPU	0.05	0.47	0.74	1.21	2.47	
2^{24}	CPU	9.05	142.99	108.69	390.22	650.94	136.02
	GPU	0.05	1.16	1.17	2.41	4.79	
2^{25}	CPU	8.97	276.82	215.64	759.10	1260.53	130.02
	GPU	0.05	2.77	2.07	4.81	9.69	
2^{26}	CPU	9.15	554.71	424.45	1492.03	2480.35	126.70
	GPU	0.05	6.31	3.62	9.59	19.58	
2^{27}	CPU	9.06	1157.91	851.91	3004.75	5023.63	133.00
	GPU	0.04	13.59	6.81	17.32	37.77	

Table II: Running time (ms) for the CPU implementation for the Rabin-Karp Single Pattern (RK) and GPU implementation of the parallel Rabin-Karp algorithm for $k = 1$, $d = 2$, $q = 65521$ and $n = 2^i$ $(22 \leq i \leq 27)$.

n	**Implementation**	**Total**	**Speed-up**
2^{22}	RK	159.25	120.05
	GPU	1.33	
2^{23}	RK	285.37	114.89
	GPU	2.48	
2^{24}	RK	567.79	119.01
	GPU	4.77	
2^{25}	RK	1094.83	113.12
	GPU	9.68	
2^{26}	RK	2135.12	109.49
	GPU	19.50	
2^{27}	RK	4306.30	113.56
	GPU	37.92	

version 7.5.17, CUDA version 7.5 and the Arch Linux OS version 4.6.3.1 have been used. For comparison purpose, the sequential Rabin-Karp algorithm for single and multiple patterns, presented in the previous section, has been implemented in C++ language on an Intel I5 760 2.80GHz using the g++ compiler version 7.2.0. The experimental results are drawn from 20 runs. For each run, the execution time of each **Step** 2 to 5 have been recorded and averaged.

In the experiments, we considered $k = 1, 512, 1024$ patterns with $m = 1024$ characters each, the input string n varies from 2^{22} to 2^{27} characters (≈ 4 to 128 Mbytes). The input parameters are stored in the global memory along with the preprocessed lookup table of **Step** 1. The parameters $d = 2$ and $q = 65521$, which is the largest prime number less than 2^{16} were used. In **Step** 2, we use one CUDA block with 64 threads for each pattern and compute the values of $h(P_k)$. In **Step** 3 we use 64 threads in 64 CUDA blocks to improve occupancy. In **Step** 4, we use the prefix-sum of the CUDA UnBounded (CUB) library version 1.7.3 [17]. CUB is a C++ template library which utilizes policy-based design to provide highly-configurable kernel components that can be tuned for different GPU architectures and applications. In this work, we used the "decoupled look-back" algorithm for performing global prefix-scan [16]. However, the code has been slightly modified so that the sum of two terms a and b in prefix-sum is calculated using $(a + b) \bmod q$. In **Step** 5 we also used 64 threads with 64 blocks for best occupancy.

Table II shows the running time results (ms) comparing the proposed parallel algorithm with the Rabin-Karp Algorithm for Single Pattern. Note that **Step** 1 is not considered in the table because we use preprocessed values. With $k = 1$, the parallel implementation does not have the advantage of using the hash table for multiple patterns. Even in this case, when compared to the CPU implementation, the GPU attains an speed-up above 100 times for $d = 2$, $q = 65521$ and $n = 2^i$ $(22 \leq i \leq 27)$.

Tables III and IV show the results for $k = 512$ and $k = 1024$, respectively. The GPU implementation achieved a speed-up surpassing 130 and 140 times for $k = 512$ and $k = 1024$, respectively, as compared to the sequential CPU implementation. As before, **Step** 1 is not considered in the results. In **Step** 2, the processing time depends of the size of m and k. It can be can seen that with $k = 1024$, the CPU takes about twice the time compared to the case where $k = 512$. In the other steps, the processing time increases gradually with the input size for the GPU. Table V shows the GPU results for $k = 512$ or $k = 1024$. One can observe that doubling the number of patterns k has little impact on the GPU computing time, however, the processing time increases proportionally to n.

Table V: GPU running time results (ms) for $k = 512$ and $k = 1024$

n	k	**Step** 2	**Step** 3	**Step** 4	**Step** 5	**Total**
2^{22}	512	0.04	0.19	0.49	0.61	1.34
	1024	0.05	0.21	0.50	0.61	1.38
2^{23}	512	0.04	0.46	0.75	1.21	2.46
	1024	0.05	0.47	0.74	1.21	2.47
2^{24}	512	0.03	1.16	1.18	2.41	4.78
	1024	0.05	1.16	1.17	2.41	4.79
2^{25}	512	0.04	2.77	2.08	4.80	9.69
	1024	0.05	2.77	2.07	4.81	9.69
2^{26}	512	0.04	6.36	3.62	9.61	19.64
	1024	0.05	6.31	3.62	9.59	19.58
2^{27}	512	0.04	13.90	6.83	17.42	38.19
	1024	0.04	13.59	6.81	17.32	37.77

Table VI: GPU occupancy of each step of the parallel implementation for $m = 2^{27}$

	Step 2	**Step** 3	**Step** 4	**Step** 5
Theorical	100%	100%	62.5%	100%
Achived	91.5%	99.9%	61.1%	99.9%

Table VI shows the "theoretical" and "achieved" occupancy for the proposed algorithm. The occupancy results have been taken from the NVIDIA profiler tool [15]. In **Step** 2, the archived occupancy was under 92% because after each thread process the value of the pattern P_k, we need to use parallel reduction on the shared memory. In **Steps** 4 and 5, each thread has an independent task. The **Step** 4 is processed with CUB and has a lower occupancy because the CUB's prefix-sum implementation must aggregate values of many CUDA blocks. On the other hand, **Steps** 3 and 5 are close to the theoretical values.

VI. Conclusion

This work presented two variations of the Rabin-Karp algorithm, one sequential and one parallel. Experimental results shown that the parallel variation provides speed-up surpassing 100 times when compared to the sequential Rabin-Karp algorithm. The proposed parallel algorithm is capable of comparing k different patterns to the input text concurrently. The proposed, parallelized, version of the Rabin-Karp algorithm has been implemented on the GeForce GTX 960 GPU. The results obtained show that the proposed implementation provides acceleration, compared to a sequential (CPU) implementation, surpassing 140 times for $k = m = 1024$ and $n = 2^{22}$.

Acknowledgment

This work was partially supported by DPP/UnB, CNPq and CAPES.

References

[1] E. Ukkonen, *Algorithms for approximate string matching*. Information and Control, vol. 64, no. 1-3, pp. 100-118, 1985.
[2] L.-L. Cheng, D. Cheung, and S.-M. Yiu, "Approximate string matching in DNA sequences," in *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, March 2003, pp. 303–310.
[3] H. Gharaee, S. Seifi, and N. Monsefan, "A survey of pattern matching algorithm in intrusion detection system," in *Telecommunications (IST), 2014 7th International Symposium on*, Sept 2014, pp. 946–953.
[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.
[5] 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. [Online]. Available: http://doi.acm.org/10.1145/360825.360855
[6] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977. [Online]. Available: http://doi.acm.org/10.1145/359842.359859
[7] P. V. Knuth D.E., MORRIS (Jr) J.H., "Fast pattern matching in strings," *SIAM Journal on Computing*, 1977.
[8] R. M. Karp and M. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, March 1987.
[9] D. Man, K. Nakano, and Y. Ito, "The approximate string matching on the hierarchical memory machine, with performance evaluation," in *7th International Symposium on Embedded Multicore Socs (MCSoC 2013)*, Sept 2013, pp. 79–84.
[10] Y. Utan, M. Inagi, S. Wakabayashi, and S. Nagayama, "A GPGPU implementation of approximate string matching with regular expression operators and comparison with its FPGA implementation,," in *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications*, 2012.
[11] W.-m. W. Hwu, *GPU Computing Gems Emerald Edition*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
[12] NVIDIA Corporation, "NVIDIA CUDA C Programming guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/, July 2016.
[13] 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*, Dec 2014, pp. 1–6.
[14] NVIDIA Corporation, "GeForce GTX-980," http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-960, April 2015.
[15] ——, "NVIDIA Visual Profiler," https://developer.nvidia.com/nvidia-visual-profiler, May 2016.
[16] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled look-back," http://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf, Oct 2017.
[17] N. Research, "CUDA UnBound (CUB)," https://nvlabs.github.io/cub/index.html, Oct 2017.