International Journal of Security and Its Applications
Vol. 11, No. 6 (2017), pp.69-76
http://dx.doi.org/10.14257/ijsia.2017.11.6.06

A Review of String Matching Algorithms and Recent Implementations using GPU

Rolando Ramos-Frías and Miguel Vargas-Lombardo

Technological University of Panama, Faculty of Computer Systems Engineering, GISES-CIDITC, Panama City, Panamá
rolando.ramos1@utp.ac.pa, miguel.vargas@utp.ac.pa

Abstract

String matching algorithms are an important element used in several computer science bb fields. These algorithms process strings of characters to find coincidences. The amount of processing required for modern applications that use string matching algorithms has increased over the years and new algorithms and solutions have been developed. In this paper we present a state of the art for the various types of string matching algorithms; and also review recent implementations done in GPU.

Keywords*: Algorithm; Processing; String Matching; GPU; CUDA*

1. Introduction

String matching algorithms are used in various computer science fields to analyze strings of text and find patterns in them. These algorithms are used in various disciplines like computer security, linguistics, artificial intelligence and medicine.

New technologies like NVIDIA CUDA, which gives users a general purpose parallel computing platform and programming model that allows them to take advantage of graphics processing units (GPUs). The extra processing power found in GPUs applied to string matching algorithms allows to process data much faster. This translates into new applications like network intrusion detection systems that scan traffic packets much faster and more efficiently; or enabling the processing of DNA information and making it accessible to researchers, a previously resource intensive task that consumed a lot of time because of the large amounts of data that need to be analyzed.

This paper is a follow up of the work done by Góngora [1] in 2012. We review new string matching algorithms and implementations done in GPU. This paper is organized as follows, in the Section 2, String Matching Algorithms based on Character Comparisons. The Section 3, String Matching Algorithms based on Deterministic Automata. Then the Section 4, String Matching Algorithms based on bit-parallelism. It is now continuous with the Section 5, improvements to String Matching in GPU using CUDA. Finally, the conclusion in the Section 6.

2. String Matching Algorithms Based on Character Comparisons

The first string matching algorithms such as the Knuth-Morris-Pratt algorithm [2] and the Boyer-Moore algorithm [3], use the classical approach of comparing the characters in two strings to find a pattern.

This approach later saw incremental improvements over the years [4-5] where different elements of the first string matching algorithms are taken, removed or mixed with new original ideas to gain performance on different scenarios. Next we present two of the most recent string matching algorithms based on character comparisons.

ISSN: 1738-9976 IJSIA

2.1. SSEF Algorithm

The SSEF Algorithm [6] is a filter based string matching algorithm. It is composed of two parts, a filtering and a verification process. First the filtering method detects the portions of the text where the pattern is probable and then the verification method checks for the real existence of the pattern on the detected positions.

During the preprocessing stage, the algorithm goes through the given text *t* and compiles the possible filter values of the pattern *p* in a linked list. Then the main algorithm loop investigates each position in the linked list and verifies it against *p* looking for a match.

The algorithm utilizes SIMD (single instruction, multiple data) instructions found in modern processors, allowing the algorithm to benefit from the fast calculation for its filtering method. It is designed to be effective for matching long patterns, bigger than 32 bytes.

The performance of this algorithm is independent of the alphabet size. The best-case time for this algorithm is given by $O(n/m)$ and the worst-case time by $O(n \cdot m)$. Both cases are identical to the Boyer-Moore algorithms [3], but the average case time given by $O(n \cdot m/2^{16})$ is improved.

The SSEF Algorithm is shown in Figure 1.

SSEF (P = $p_0p_1p_2 \cdots p_{m-1}$, T = $t_0t_1t_2 \cdots t_{m-1}$)
Set K = a, $0 \leq a < 8$, according to the alphabet;
i = L =PreProcess(P,K);
while i < N do
    f = sign($t_{i \cdot 16}$ << K) $\cdot 2^{15}$ + sign($t_{i \cdot 16+1}$ << K) $\cdot 2^{14}$ + $\cdots$ + sign($t_{i \cdot 16+15}$ << K)
    for all $j \in$ FList[f] do
        if P = [$t_{(i-L) \cdot 16+j} \ldots t_{(i-L) \cdot 16+j+m-1}$] then
            pattern detected at $t_{(i-L) \cdot 16+j}$;
        end if
    end for
    $i = i$ + L;
end while

Figure 1. SSEF Algorithm

2.2. Le Dang-Le-Le Algorithm

In 2016 Nguyen Le Dang, Dac-Nhuong Le and Vinh Trong Le published a new multiple-pattern matching algorithm [7]. Their matching algorithm is designed for large numbers of patterns using a technique focused on creating a graph transition structure and a dynamic linked list search technique to optimize memory space and improve its speed.

During the preprocessing stage a graph transition structure is created to represent the array of patterns *p*. Afterwards, in the searching stage a list of pointers is used to search through the array for any match.

The worst-case time is given by $O(n \cdot m \cdot L)$, L being the maximum length of the pattern. In their experiments, their algorithm was tested against the Aho-Corasick Algorithm [8], Commentz Walter Algorithm [9] and Wu-Manber Algorithm [10]. The best performing of these algorithms for multiple patterns search being the Commentz Walter Algorithm.

This new algorithm shows slight improvements over the Comment Walter Algorithm in both execution time and consumed space. Figure 2 shows the algorithm.


```
Procedure DNL (T, n, m, p, G)
S = Ø ;
for i=1 → n do
        Init pointer P_i = P_0;
        S = S U {P_i}
        if (T[i] in G) and (P_j in S) then
                P_j[position of T[i] in P[j]] = P_j[position of T[i] in P[j]] -1;
                if (P_j[k] = 0) then
                        Output P[k] detected;
                        Remove P_j;
                end if
                if (P_j in S) and (P_j not changed) then Remove P_j
                end if
        end if
end for
```

Figure 2. Le Dang-Le-Le Algorithm

3. String Matching Algorithms Based on Deterministic Automata

Much of the string matching theory has seen a lot of developments using automatons for string matching algorithms [11]. One example is the backward-automaton-matching (BOM) algorithms that use the Boyer-Moore method to try to match suffixes of the pattern and factors of the pattern by scanning a text window from right to left using a factor automata and factor oracles.

3.1. Extended-Backward-Oracle-Matching Algorithm

One variation of the BOM algorithm, proposed by Simone Faro and Thierry Lecroq, consists in extending the BOM algorithm with a fast-loop over oracle transitions [12], similar to the Tuned-Boyer-Moore algorithm [13].

This fast-loop locates an occurrence of the rightmost character of the pattern by iterating through the bad character heuristic in a checkless cycle. After the fast-loop, the algorithm goes through the transition phase using the factor oracle until a match for the pattern is found. It can be implemented in $O(\sigma^2)$ time and space.

This kind of algorithm is suitable when using short patterns and large alphabets and displays improved performance when compared with the original BOM algorithm.

In Figure 3 we present the Extended-BOM algorithm.

```
Extended-BOM (p, m, t, n)
δ ← precompute-factor-oracle(p)
for a ∈ Σ do
        q ← δ (m, a)
        for b ∈ Σ do
                if q = ⊥ then λ(a, b)←⊥
                else λ(a, b) ← δ(q, b)
t[n ..n + m − 1] ← p
j ← m − 1
while j < n do
        q ← λ(t[j], t[j − 1])
        while q = ⊥ do
                j ← j + m − 1
                q ← λ(t[j], t[j − 1])
        i ← j − 2
        while q ≠ ⊥ do
                q ← δ(q, t[i])
                i ← i − 1
        if i < j − m + 1 then
                output(j)
                i ← i + 1
        j ← j + i + m
```

Figure 3. Extended-BOM Algorithm

3.2. Forward-Backward-Oracle-Matching Algorithm

Another variation of the BOM Algorithm by S. Faro and T. Lecroq employs the idea of looking for the forward character in a window of text [12]. This idea was originally introduced in the Quick-Search algorithm by Daniel M. Sunday [5]. It was later efficiently implemented in the Forward-Fast-Search Algorithm by Domenico Cantone and Simone Faro [14].

This algorithm takes from the Extended-BOM variation of the algorithm, but the fast loop is modified to consider the forward character of the text window. If there is no transition for the first two characters, the text window is shifted 'forward' to the right. The preprocessing phase is performed in $O(m + \sigma^2)$ time and space.

A forward factor oracle is used to recognize all the factors of the pattern p that is being analyzed. This forward factor oracle FOracle(p) is constructed in $O(m + \Sigma)$ time. The Forward-BOM algorithm is also best suited for scenarios where large alphabets and short patterns are used. Figure 4 shows the Forward-BOM Algorithm.


```
Forward-BOM (p, m, t, n)
δ ← precompute-factor-oracle(p)
for a ∈ Σ do
        q ← δ(m, a)
        for b ∈ Σ do
                if q = ⊥ then λ(a, b)←⊥
                else λ(a, b) ← δ(q, b)
q ← δ(m, p[m − 1])
for a ∈ Σ do λ(a, p[m − 1]) ← q
t[n ..n + m − 1] ← p
j ← m − 1
while j < n do
        q ← λ(t[j + 1], t[j])
        while q = ⊥ do
                j ← j + m
                q ← λ(t[j + 1], t[j])
        i ← j − 1
        while q ≠ ⊥ do
                q ← δ(q, t[i])
                i ← i − 1
        if i < j − m + 1 then
                output(j)
                i ← i + 1
        j ← j + i + m
```

Figure 4. Forward-BOM Algorithm

4. String Matching Algorithms Based on Bit-parallelism

Bit-parallelism is a technique designed to take advantage of the parallelism found in the bit operations inside a computer word. It is most efficient for string matching algorithms using non-deterministic automatons, allowing to reduce de number of operations that the algorithm needs to perform by a factor up to *w*, where *w* is the number of bits in the computer word.

One of the first and most known string matching algorithm using bit-parallelism is the Wu-Manber algorithm [15]. Next we present two of the most recent string matching algorithms based on bit-parallelism.

4.1. Forward Simplified Backward Nondeterministic DAWG Matching Algorithm

The Forward SBNDM algorithm [12] is based on the Simplified BNDM algorithm [16], which is a bit-parallel simulation of the BDM algorithm [17] using instead a non-deterministic automaton.

The Simplified BNDM algorithm uses a text window to scan two consecutive text characters from right to left using the non-deterministic version of the factor automaton to find a match of the pattern. It also implements a fast-loop to obtain better results.

The Forward SBDNDM algorithm borrows the same idea from the Forward-BOM algorithm, where the fast-loop is used to look for the forward character position. The resulting FSBNDM algorithm is the most efficient for situations where long patterns and a very large alphabet is used. It has a similar time complexity of $O(n[m/w])$ as the BNDM algorithm and consume $O(\sigma[m/w])$.

In Figure 5 we present the Forward-SBNDM algorithm.

```
Forward-SBNDM (p, m, t, n)
for all c ∈ Σ do B[i] ← 1
for i = 0 to m − 1 do B[p[i]] ← B[p[i]] | (1 << (m − i))
j ← m − 1
while j < n do
        D ← (B[t[j + 1]] << 1) & B[t[j]]
        if D ≠ 0 then
                pos ← j
                while D ← (D + D) & B[t[j − 1]] do j ← j − 1
                j ← j + m − 1
                if j = pos then
                        output(j)
                        j ← j + 1
        else j ← j + m
```

Figure 5. Forward-SBNDM Algorithm

4.2. Simplified Backward Nondeterministic DAWG Matching Algorithm with q-grams

This algorithm proposed by B. Durian [18] takes on the SBNDM algorithm utilizing *q*-grams, *i.e.*, *q* characters for shifting. The SBNDM is a simplification of the BNDM algorithm that ignores prefixes of the pattern and instead shifts the text window in case of a mismatch. This results in faster performance, especially for short patterns.

The algorithm first reads a *q*-gram before testing the state. If the *q*-gram is not present in the pattern the algorithm moves the windows to the right. The worst-case time complexity for this algorithm is $O(mn)$ with a space complexity of $O(|\Sigma|)$. The SBNDM*q* algorithm is suitable for long patterns and small alphabets. Figure 6 shows the Simplified BNDM*q* Algorithm.

```
SBNDMq
for a ∈ Σ do B[a] ← 0 end for
for j ← 1..m do
        B[p_j] ← B[p_j] | (1 << (m − j))
end for
Compute (s_0)
i ← m− q + 1
while i ≤ n − q + 1 do
        D ← F(i, q)
        if D ≠ 0 then
                j ← i − (m − q + 1)
                do i ← i − 1
                        D ← (D << 1) & B[t_i]
                while D ≠ 0
                if j = i then
                        report occurrence at j + 1
                        i ← i + s_0
                end if
        end if
        i ← i + m − q + 1
end while
```

Figure 6. Simplified BNDM*q* Algorithm

5. Improvements to String Matching in GPU using CUDA

In 2007, NVIDIA introduced a parallel computing platform and programming model named CUDA. It allows a substantial increase in computing performance by utilizing the graphics processing unit (GPU) to accelerate various kinds of applications. We review the different work that has been done using string matching algorithms in GPU.

Zha and Sahni [19] developed in 2011 an adaption of the Aho-Corasick and Boyer-Moore algorithms and implemented them in GPU using a NVIDIA Tesla GT200 GPU and a Xeon 2.8GHz quad-core CPU. Their work showed a speedup for the AC and BM algorithms in single-threaded implementations by a factor of 3.1-9.5. However, the BM algorithm showed to run 7%-10% slower when compared to a multi-threaded implementation in the quad-core CPU.

In 2013, Xu [20] implemented in GPU the string matching algorithm MASM, and extension of the BPR algorithm for multiple patterns. Using a NVIDIA GeForce 310M GPU and a Core i3 2.27GHz CPU they achieved a speedup by a factor of 28 relative to a single-thread CPU implementation. The same year, Bellekens [21] implemented the Knuth-Morris-Pratt algorithm in a NVIDIA Tesla K20M GPU and compared it versus two Xeon E5-2620. The results showed a 29-fold increase in speed where the GPU was used instead of the CPUs.

Nagaveni [22] in 2014, used string matching algorithms to find DNA Sequences to detect breast cancer. Using a NVIDIA Tesla C2070 GPU and a Core i7 CPU, his experiment showed the GPU implementation 30 times more efficient than the serial implementation in the CPU.

In 2015, Kouzinopoulos [23] did an experiment using a GTX 280 GPU and a Xeon 2.4GHz CPU. They tested the Aho-Corasick, Set Horspool, Set Backward Oracle Matching, Wu-Manber and SOG multiple pattern matching algorithms showing the basic implementation of these algorithm in GPU were between 2.5 and 10.9 faster than the CPU implementation. The Set Horspool and the Set Backward Oracle Matching algorithms showed the most gains when using a GPU. Another work by Sharma [24] using a NVIDIA GeForce GT 635M, implemented the Rabin-Karp pattern matching algorithm [25] used for Deep Packet Inspection in NIDS. The GPU implementation achieved a speedup by a factor of 14 when compared against an Intel quad-core processor.

A recent work by Ashkiani [26] in 2016, using a NVIDIA Tesla K40c GPU, implemented a string matching algorithm based on the Rabin-Karp algorithm showing a 4.81 speedup against a CPU implementation.

6. Conclusion

In this paper, the most recent examples of string matching algorithms based in character comparison, deterministic automata and bit-parallelism are presented. The execution time and best scenario for each algorithm is also discussed. The research conducted on the most recent string matching algorithms has been mainly focused on variations of previously proposed algorithms with limited examples presented after 2010.

Finally, we present a state of the start on several of the most recent implementations of string matching algorithms in GPU using the CUDA framework and the performance that is obtained versus a CPU implementation. The advent of technologies like CUDA has allowed to harness the extra processing power found in GPUs and the performance gained in GPU implementations is found across all the implementations reviewed.

References

[1] M. G. Blandón and M. V. Lombardo, “State of the art for string analysis and pattern search using cpu and gpu based programming”, Journal of Information Security, vol. 3, **(2012)**, pp. 314.

[2] D. E. Knuth, J. Morris, James H, and V. R. Pratt, “Fast pattern matching in strings”, SIAM journal on computing, vol. 6, **(1977)**, pp. 323-350.

[3] R. S. Boyer and J. S. Moore, “A fast string searching algorithm”, Communications of the ACM, vol. 20, **(1977)**, pp. 762-772.
[4] R. N. Horspool, “Practical fast searching in strings”, Software: Practice and Experience, vol. 10, **(1980)**, pp. 501-506.
[5] D. M. Sunday, “A very fast substring search algorithm”, Communications of the ACM, vol. 33, **(1990)**, pp. 132-142.
[6] M. O. Külekci, “Filter Based Fast Matching of Long Patterns by Using SIMD Instructions”, in Stringology, **(2009)**, pp. 118-128.
[7] N. L. Dang, D. N. Le, and V. T. Le, “A new multiple-pattern matching algorithm for the network intrusion detection system”, International Journal of Engineering and Technology, vol. 8, **(2016)**, pp. 94.
[8] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search”, Communications of the ACM, vol. 18, **(1975)**, pp. 333-340.
[9] B. C. Walter, “A string matching algorithm fast on the average”, in International Colloquium on Automata, Languages, and Programming, **(1979)**, pp. 118-132.
[10] S. Wu and U. Manber, “A fast algorithm for multi-pattern searching”, **(1994)**.
[11] S. Faro and T. Lecroq, “The exact online string matching problem: A review of the most recent results”, ACM Computing Surveys (CSUR), vol. 45, **(2013)**, pp. 13.
[12] S. Faro and T. Lecroq, “Efficient variants of the backward-oracle-matching algorithm”, International Journal of Foundations of Computer Science, vol. 20, **(2009)**, pp. 967-984.
[13] A. Hume and D. Sunday, “Fast string searching”, Software: Practice and Experience, vol. 21, **(1991)**, pp. 1221-1248.
[14] D. Cantone and S. Faro, “Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm”, Journal of Automata, Languages and Combinatorics, vol. 10, pp. 589-608, 2005.
[15] [15]S. Wu and U. Manber, "Fast text searching: allowing errors," Communications of the ACM, vol. 35, **(1992)**, pp. 83-91.
[16] H. Peltola and J. Tarhio, “Alternative algorithms for bit-parallel string matching”, in International Symposium on String Processing and Information Retrieval, **(2003)**, pp. 80-93.
[17] R. B. Yates and G. H. Gonnet, “A new approach to text searching”, Communications of the ACM, vol. 35, **(1992)**, pp. 74-82.
[18] B. Ďurian, J. Holub, H. Peltola, and J. Tarhio, “Tuning BNDM with> q-grams”, in Proceedings of the Meeting on Algorithm Engineering & Expermiments, **(2009)**, pp. 29-37.
[19] X. Zha and S. Sahni, “GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU”, IEEE Transactions on Computers, vol. 62, **(2013)**, pp. 1156-1169.
[20] K. Xu, W. Cui, Y. Hu, and L. Guo, “Bit-parallel multiple approximate string matching based on GPU”, Procedia Computer Science, vol. 17, **(2013)**, pp. 523-529.
[21] X. Bellekens, I. Andonovic, R. Atkinson, C. Renfrew, and T. Kirkham, “Investigation of GPU-based pattern matching”, in The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013), **(2013)**.
[22] V. Nagaveni and G. Raju, “Various String Matching Algorithms for DNA Sequences to Detect Breast Cancer using CUDA Processors”, International Journal of Engineering and Technology, **(2014)**.
[23] 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, **(2015)**, pp. 121-138.
[24] 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, pp. 70, **(2015)**.
[25] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms”, IBM Journal of Research and Development, vol. 31, **(1987)**, pp. 249-260.
[26] S. Ashkiani, N. Amenta, and J. D. Owens, “Parallel Approaches to the String Matching Problem on the GPU”, in 28th ACM Symposium on Parallelism in Algorithms and Architectures, **(2016)**.

