SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2433

Engineering order-preserving pattern matching with SIMD parallelism

Tamanna Chhabra[1,*,†], Simone Faro[2], M. Oğuzhan Külekci[3] and Jorma Tarhio[1]

[1] *Department of Computer Science, Aalto University, Espoo, Finland*
[2] *Department of Mathematics and Computer Science, Università di Catania, Catania, Italy*
[3] *Informatics Institute, Istanbul Technical University, Istanbul, Turkey*

SUMMARY

The order-preserving pattern matching problem has gained attention in recent years. It consists in finding all substrings in the text, which have the same length and relative order as the input pattern. Typically, the text and the pattern consist of numbers. Since recent times, there has been a tendency to utilize the ability of the word RAM model to increase the efficiency of string matching algorithms. This model works on computer words, reading and processing blocks of characters at once, so that usual arithmetic and logic operations on words can be performed in one unit of time. In this paper, we present a fast order-preserving pattern matching algorithm, which uses specialized word-size packed string matching instructions, grounded on the single instruction multiple data instruction set architecture. We show with experimental results that the new proposed algorithm is more efficient than the previous solutions. © 2016 The Authors Software: Practice and Experience Published by John Wiley & Sons Ltd.

Received 12 December 2015; Revised 5 July 2016; Accepted 9 July 2016

KEY WORDS: SIMD; SSE; AVX/AVX2 order-preserving pattern matching

1. INTRODUCTION

Let x be a pattern of length m and y be a text of length n, over the alphabet Σ of size σ, and then, the *exact pattern matching problem* consists of finding all substrings in y, of length m, which are same as x. Such a problem is one of the most important subjects in the domain of text processing.

There are many variations of the exact pattern matching problem. One of them is the order-preserving pattern matching (OPPM) problem [1–6]. Some solutions have been devoted to such a problem in recent years. Specifically, given a pattern x and a text y, whose characters are drawn from an alphabet Σ with a total order relation defined on it, OPPM consists in finding all the substrings of y with the same length and the same relative order as the pattern x. Typically, the text and the pattern consist of numbers.

For instance, given the pattern $x = (34, 45, 30, 26, 33, 40)$, value 26 is the smallest number of the string, while the value 33 is the second smallest, and so on. Therefore, the relative order of the pattern is given by $\langle 3, 5, 1, 0, 2, 4\rangle$. Thus, x occurs in the text $y = (12, 08, 14, 30, 40, 16, 13, 21, 33, 26, 23)$ at position 3, because x and the substring $u = (30, 40, 16, 13, 21, 33)$ share the same relative order (Figure 1).

[*] Correspondence to: Tamanna Chhabra, Department of Computer Science, Aalto University, Espoo, Finland.

[†] E-mail: tamanna.chhabra@aalto.fi (Tamanna Chhabra)

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.

© 2016 The Authors Software: Practice and Experience Published by John Wiley & Sons Ltd.

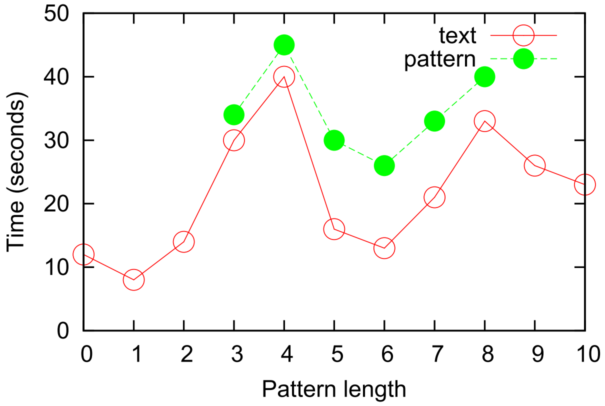

Figure 1. Example of order-preserving pattern matching problem.

The OPPM problem finds applications in all those fields where someone is interested in finding patterns with the same relative order and not in finding patterns with identical values. For example, it can be applied to musical information retrieval and, in particular, matching melodies of two different musical scores [2]. It also finds applications in matching time series, like prices in stock markets [2] and weather data.

The first solution for solving the OPPM problem was presented in 2013 by Kubica *et al.* [1]. They presented a solution based on the Knuth–Morris–Pratt algorithm [7] and working in $\boldsymbol{O}(n + m \log m)$ time. In the same year, Kim *et al.* [2] announced another solution to the problem also based on the Knuth–Morris–Pratt approach running in $\boldsymbol{O}(n + m \log m)$ time. In 2013, Cho *et al.* [3] proposed a solution to the OPPM problem based on the Boyer–Moore approach [8] showing a sublinear behavior on average. In the same year, Belazzougui *et al.* [6] proposed an optimal sublinear algorithm with $O\left(\frac{n \log m}{m \log \log m}\right)$ average-case time complexity.

Another sublinear solution to the OPPM problem was presented in 2014 by Chhabra and Tarhio [4], based on a filtering approach. Specifically, their algorithm performs a conversion of the input strings to binary strings; later, the converted pattern is searched in the converted version of the text by using any standard algorithm for exact string matching. A verification procedure is then applied when a candidate occurrence of the pattern is found.

More recently, some solutions to the OPPM problem, based on the word RAM model of computation (see for instance [9, 10]), were presented. Such a model has been used in the last two decades to speed-up string matching algorithms. It consists in operating on computer words, reading and processing blocks of characters at once, so that usual arithmetic and logic operations on words can be performed in one unit of time.

Specifically, two online solutions to the OPPM problem were proposed by Chhabra *et al.* [5]. The online solutions use the single instruction multiple data (SIMD) architecture [11]. Two different SIMD instruction sets streaming SIMD extensions (SSE) and advanced vector extensions (AVX) are used to implement the solutions. Other solutions were also proposed in [12] and [13]. Faro and Külekci [13] presented two filtering approaches in which the original string is translated into a new string over large alphabets. This in turn increases the performance of the solutions as the number of match candidates decrease significantly. Later, Cantone *et al.* [12] proposed another efficient solution based on the Skip Search algorithm [14]. It computes the fingerprint of all substrings of a pattern of a given length. Thereafter, the fingerprints are indexed to obtain the match candidates, which are then located in the text. Cantone *et al.* used the SSE instruction set architecture for the computation of the fingerprint. The solutions were faster than the previous algorithms in many cases.

In this paper, we introduce an efficient and practical algorithm for the OPPM utilizing the SIMD extensions (SSE) technology [11, 15], and the algorithm is shown to be faster than the best algorithms known in the literature. The algorithm, named SIMD-OPPM, uses specialized packed instructions with a low latency.

© 2016 The Authors Software: Practice and Experience
Published by John Wiley & Sons Ltd.

The paper is organized as follows. In Section 2, we give preliminary notions and definitions in relation to the order-preserving matching problem. Then, we present our new solution in Section 3 and evaluate its performance against the previous algorithms in Section 4. Conclusions are drawn in Section 5.

2. NOTIONS AND BASIC DEFINITIONS

Following notations and terminology are used throughout the paper. A string x, of length $m > 0$, is represented as a finite array $x[0..m-1]$ of characters from a finite alphabet Σ of size σ, and $x[i \dots j]$ will denote a *factor* (or *substring*) of x, for $0 \leqslant i \leqslant j < m$. We suppose that a total order relation '$\leqslant$' is defined on the alphabet, so that we could establish if $a \leqslant b$ for each $a, b \in \Sigma$ and we denote by $|x|$ the length of x. We indicate with the symbol w the length of the SIMD registers $(= 128)$.

We say that two strings $x, y \in \Sigma^*$ are order isomorphic if the relative order of their elements is the same. In formal words, we give the following definition.

*Definition 1 (**Order isomorphism**.)*
Let Σ be the alphabet, and let x, y be two strings of the same length over the alphabet, and then, we say that x and y are *order-isomorphic* and write $x \approx y$, if the following conditions hold

1. $|x| = |y|$
2. $x[i] \leqslant x[j]$ if and only if $y[i] \leqslant y[j]$, for $0 \leqslant i, j < |x|$

Similarly relative order can be more formally defined as follows:

*Definition 2 (**Rank function**.)*
Let x be a string of length m over an alphabet Σ. Then, the rank function of x is a mapping $r : \{0, 1, \dots, m-1\} \to \{0, 1, \dots, m-1\}$ such that $x[r(i)] \leqslant x[r(j)]$ holds for each pair $0 \leqslant i < j < m$. If $x[r(i)] = x[r(i+1)]$ holds, then $r(i) < r(i+1)$.

In addition, we define the *equality function* of x, which indicates which elements of the string are equal (if any). More formally, we have the following definition.

*Definition 3 (**Equality function.**)*
Consider x to be a string of length m over an alphabet Σ, and let r be the rank function of x. Then, the equality function of x is a mapping $eq : \{0, 1, \dots, m-2\} \to \{0, 1\}$ such that for each $0 \leqslant i < m$

$$eq(i) = \begin{cases} 1 & \text{if } x[r(i)] = x[r(i+1)] \\ 0 & \text{otherwise} \end{cases}$$

Let r be the rank function of a string x, such that $m = |x|$, and let eq be its equality function. It is easy to prove that x and y are order isomorphic if and only if they share the same rank and equality function, that is, if and only if the following two conditions hold

1. $y[r(i)] \leqslant y[r(i+1)]$, for $0 \leqslant i < m-1$
2. $y[r(i)] = y[r(i+1)]$ if and only if $eq(i) = 1$, for $0 \leqslant i < m-1$

Example 1
Let $x = (4, 6, 5, 1, 3, 6)$ and $y = (3, 7, 5, 1, 2, 7)$ be two strings of length six. The rank r of x is $(3, 4, 0, 2, 1, 5)$ while its equality function is $eq(x) = (0, 0, 0, 0, 1)$. The two strings are order isomorphic according to the conditions given earlier, that is, $x \approx y$.

The problem of OPPM is to find all substrings in the text, which have the same length and relative order as the pattern. Specifically, we have the following formal definition.

*Definition 4 (**Order-preserving matching function**.)*
Let x and y be two strings of length m and n, respectively (and $n > m$), both over an alphabet Σ. The OPPM consists in finding all indexes i, with $0 \leqslant i \leqslant n-m$, such that $y[i \dots i+m-1] \approx x$.

We also make use of bitwise infix operators, like the bitwise `and` '&' and the `left shift` '$\ll$' operator.

3. NEW METHOD FOR ORDER-PRESERVING MATCHING

This section presents a new algorithm for the OPPM. The algorithm utilizes the Intel SSE instruction set [11, 15], hence the name SIMD-OPPM.

In packed string matching [9, 10], sets of adjacent characters are packed into one single word, according to the size of the word in the target machine. Input is standard text, and packing is carried out on line with SIMD instructions. This allows us to compare set of characters in a bulk rather than individually, by comparing the corresponding words. Therefore, when the characters are taken from an alphabet of size σ, $\gamma = \lceil \log \sigma \rceil$ bits are used to encode a single character and $\lfloor w/\gamma \rfloor$ characters fit in a register. In this case, we will use the symbol $\alpha = \lfloor w/\gamma \rfloor$ to indicate the packing factor. In the following section, we will discuss the details of our model.

Several values of α and γ are possible, but we assume that $\alpha = 16$ and $\gamma = 8$, which is the most common case when we deal with a word RAM model with 128-bit registers. In our experimental evaluation (Section 4), we have $\sigma = 256$.

3.1. The model

In the design and implementation of our solution, we make use of specialized word-size packed string matching instructions, based on the SSE instruction set architecture [11, 15]. SIMD instructions allow the processor to execute multiple data on a single instruction using a set of special instructions working on special registers. SSE [11, 15] is a family of SIMD instruction sets supported by Pentium III processors since 1999. It makes use of sixteen 128-bit registers known as XMM0 through XMM15. Because the registers are 128 bits long, 16 integer numbers could be handled at the same time (an integer is considered 8 bits long), thereby providing important speedups in algorithms. In SSE4.2, we have the following data types:

- _m128: four 32-bit floating point values
- _m128d: two 64-bit floating point values
- _m128i: 16/8/4/2 integer values, depending on the size of the integers

The SIMD-OPPM algorithm makes use of the word-size parallel comparison (wspc) and word-size equality checker (wsec) specialized word-size packed instructions. These two instructions are described below.

The instruction wspc(A, B) handles two w-bit registers A and B as a block of α small integers values and computes an α-bit fingerprint from it. It compares in parallel all the α values contained in A against the α values in B. More formally, assuming $B[0...\alpha-1]$ and $A[0...\alpha-1]$ are a w-bit integer parameters, wspc(A, B) returns an α-bit value $r[0...\alpha-1]$, where $r[j] = 1$ if and only if $A[j] < B[j]$, and $r[j] = 0$ otherwise.

The wspc(A, B) instruction uses the following sequence of specialized SIMD instructions and can be completed in constant time:

```
wspc($A, B$)
    $B \leftarrow$ _mm_cmpgt_epi8($B, A$)
    $r \leftarrow$ _mm_movemask_epi8($B$)
    return $r$
```

The instruction _mm_cmpgt_epi8(B, A) computes the 128-bit vector by comparing the 16 signed 8-bit integers in A and the 16 signed 8-bit integers in B for greater than. If a data element in A is greater than the corresponding data element in B, then the corresponding data element in B is set to 1; otherwise, it is set to 0. The 128-bit vector B is then handled by _mm_movemask_epi8(B) instruction as sixteen 8-bit integers, and as a result, a 16-bit mask is formed from the most significant bits of the 16 integers in B.

The instruction wsec(A, B) handles two w-bit registers A and B as a block of α small integers values and computes an α-bit fingerprint from it. Assuming $A[0...\alpha-1]$ and $B[0...\alpha-1]$ are the w-bit integer parameters, wsec(A, B) returns an α-bit value $r[0...\alpha-1]$, where $r[j] = 1$ if and only if $A[j] = B[j]$, and $r[j] = 0$ otherwise.

© 2016 The Authors Software: Practice and Experience
Published by John Wiley & Sons Ltd.

Table I. Latency and throughput of SIMD instructions for Sandy Bridge [17].

Architecture	SIMD instruction	Latency	Throughput
Sandy Bridge	*_mm_cmpgt_epi*8	1	0.5
	*_mm_cmpeq_epi*8	1	0.5
	*_mm_movemask_epi*8	2	1

The wsec(A, B) instruction uses the following sequence of specialized SIMD instructions and can also be completed in constant time:

```
wsec($A, B$)
    $B \leftarrow$ _mm_cmpeq_epi8($A, B$)
    $r \leftarrow$ _mm_movemask_epi8($B$)
    return $r$
```

The _mm_cmpeq_epi8(A, B) instruction compares the unsigned 8-bit or 16 signed integers in A and the unsigned 8-bit or 16 signed integers in B for equality. If a pair of data elements in A and B is equal, the corresponding data element in B is set to 1; otherwise, it is set to 0. The _mm_movemask_epi8 instruction works as described earlier.

We will also make use of the popcount(C) instruction, when we will be interested in counting the number of bit set in an α-bit register C. This can be carried out in $\log(\alpha)$ operations by using a population count function. In our implementation, we make use of a constant time ad hoc procedure [16] designed to work with 16-bit registers.

The performance of SIMD instructions depends on the architecture of the processor. The performance of a single instruction is measured by latency and throughput. Latency is the number of cycles taken by the processor to give the desired outcome form the given input. Throughput [17] refers to the number of cycles between subsequent calls of the same instruction. The processor used in our experiments is i7-3820 QM, and its micro-architecture is Sandy Bridge. The latency and throughput of the SIMD instructions used in our algorithms for this processor is given in Table I.

3.2. *The algorithm*

The SIMD-OPPM algorithm is designed to search order-preserving occurrences of sequences. Before execution, the arrays corresponding to functions r and eq are computed based on the pattern.

Let x be the pattern of length m over the alphabet Σ, and if Y is a block of w bits (α elements) of the text y, we can find all the occurrences of x having their leftmost position in Y. Let $Y = Y_0 Y_1 \dots Y_{k-1}$, where $k = \lfloor n/\alpha \rfloor$+1. The idea behind the algorithm is to check in parallel for groups of occurrences of x in y while scanning each block Y_i of the text. In particular, during each iteration of the algorithm, we check groups of α occurrences of x.

Formally, let $Y_i = y[i\alpha \dots i\alpha + \alpha - 1]$ be the current block of the text. The substring $y[j \dots j + m - 1]$ is an order preserving occurrence of x if and only if

1. $y[j + r(h)] \leqslant y[j + r(h + 1)]$, for $0 \leqslant h < m - 1$
2. $y[j + r(h)] = y[j + r(h + 1)]$ if and only if $eq(h) = 1$, for $0 \leqslant h < m - 1$

The pseudocode of the SIMD-OPPM algorithm is shown in Figure 2. During each iteration, the algorithm checks the match candidates whose first position is in the block $Y = y[i \dots i + \alpha - 1]$. At the end of the iteration, the value of i is advanced α positions to the right. Thus, the total number of iterations of the algorithm is $\lceil n/\alpha \rceil$.

During each iteration, the algorithm maintains a bit mask C of α bits, which contains occurrences of the pattern starting in the current block Y. Specifically, at the end of the iteration, the bit $C[j]$ is set if and only if $x \approx Y[j \dots j + m - 1]$, for $j = 0 \dots \alpha - 1$, while $C[i] = 0$ otherwise. At the beginning of each iteration, C is initialized as 1^α (line 3).

In order to understand how such a value is computed, let $A_j = B_{j-1} = y[i + r(j) \dots i + r(j) + \alpha - 1]$ (line 6) and $B_j = y[i + r(j + 1) \dots i + r(j + 1) + \alpha - 1]$ (line 7). For simplicity, let

```
SIMD-OPPM($x, m, y, n, r, eq$)
1.   $k \leftarrow 0$
2.   for $i \leftarrow 0$ to $n - m$ step $\alpha$ do
3.       $C \leftarrow 1^{\alpha}$
4.       $B \leftarrow y[i + r(0) \ldots i + r(0) + \alpha - 1]$
5.       for $j \leftarrow 0$ to $m - 2$ do
6.           $A \leftarrow B$
7.           $B \leftarrow y[i + r(j+1) \ldots i + r(j+1) + \alpha - 1]$
8.           if $eq(j)$ then
9.               $C \leftarrow C$ & wsec($A, B$)
10.          else $C \leftarrow C$ & wspc($A, B$)
11.          if $C = 0$ then goto out
12.      $k \leftarrow k$ + popcount($C$)
13.      out:
14.  return $k$
```

Figure 2. Order-preserving pattern matching algorithm.

us assume that all values of the pattern are distinct. Let $C_j = \mathsf{wspc}(A_j, B_j)$ (line 10). According with the definition of the wspc instruction, we have $C[h] = 1$ if and only if $A[h] < B[h]$ (i.e. $y[i + h + r(j)] < y[i + h + r(j + 1)]$) and $C[h] = 0$ otherwise, for $h = 0 \ldots \alpha - 1$. The value of the bit mask C is computed as $C = C_0 \& C_1 \& \ldots \& C_{m-2}$. It is easy to prove that $C[h]$ is set if and only if $y[i + h + r(j)] < y[i + h + r(j + 1)]$ for $j = 0 \ldots m - 2$, which implies that $x \approx y[i + h \ldots i + h + m - 1]$.

At the end of each iteration, we count the number of bits set in the bit mask C. This is the number of occurrences the algorithm found in the current block. Such a value is accumulated in a counter k (line 12), which will contain the total number of occurrences of x in y.

If $(n - m + 1) \bmod \alpha$ is not zero, the popcount of the last block may contain extra matches. So the popcount of the $\alpha - [(n - m + 1) \bmod \alpha]$ extra bits got from the last block must be subtracted from k after the outside loop.

In our practical experiments, we used a slightly modified version of the SIMD-OPPM algorithm. Because α match candidates are checked at same time, the variable C is not zero on average during the first iterations of the innermost loop. Therefore, the testing of the variable C is rather useless during the first iterations. Thus, peeling of this loop is beneficial. In the beginning of the innermost loop, C holds α set bits. Each iteration roughly halves the number of set bits in C in the case of random data. For $\alpha = 16$, we moved four iterations in front of the loop, and the value of C was tested for the first time after these iterations. In best cases, this optimization doubles the speed of the algorithm on modern processors.

The total time complexity of the algorithm is $O(nm/\alpha)$. In practice, the algorithm shows a linear behavior on average, as we can observe in the subsequent section. If one is interested in retrieving the position of each occurrence, additional $O(s)$ work is needed in order to locate s occurrences. More specifically, if the h-th bit of C is set, then an occurrence at position $i + h$ must be reported.

Example 2

We illustrate the algorithm using an example. Let $x = (8, 5, 13, 10)$ be a pattern of length 4 and $y = (7, 9, 5, 14, 13, 22, 16, 10, 3, 13, 11, 10, 11, 8, 9, 2)$ be a text of length 16. The rank values and equality values for the pattern x are $(1, 0, 3, 2)$ and $(0, 0, 0)$. The result is computed in $m - 1 = 3$ steps. We know that $x[r(i)] \leqslant x[r(i + 1)]$ holds for $0 \leqslant i \leqslant m - 1$. In order to have an occurrence beginning at position j of $Y = y$, we must have $Y[j + r(i)] \leqslant Y[j + r(i + 1)]$, for $0 \leqslant i \leqslant m - 1$. Then, C_i is a 16-bit register where $C_i[j]$ is set to 1 if $Y[j + r(i)] \leqslant Y[j + r(i + 1)]$ and $C_i[j]$ is set to 0 otherwise.

Now, C is a 16-bit register where $C = C_0 \ \& \ C_1 \ \& \ \cdots \& \ C_{m-2}$ and $C[j]$ is set if we have an occurrence of x at position j of Y and $C[j] = 0$ otherwise.

Step 1 is as follows:

Y <<1	9	5	14	13	22	16	10	3	13	11	10	11	8	9	2	0
Y <<0	7	9	5	14	13	22	16	10	3	13	11	10	11	8	9	2
C_0	0	1	0	1	0	1	1	1	0	1	1	0	1	0	1	1

Step 2:

Y <<0	7	9	5	14	13	22	16	10	3	13	11	10	11	8	9	2
Y <<3	14	13	22	16	10	3	13	11	10	11	8	9	2	0	0	0
C_1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0

Step 3:

Y <<3	14	13	22	16	10	3	13	11	10	11	8	9	2	0	0	0
Y <<2	5	14	13	22	16	10	3	13	11	10	11	8	9	2	0	0
C_2	0	1	0	1	1	1	0	1	1	0	1	0	1	1	0	0

Then we can compute the value of C as follows:

C_0	0	1	0	1	0	1	1	1	0	1	1	0	1	0	1	1	&
C_1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	&
C_2	0	1	0	1	1	1	0	1	1	0	1	0	1	1	0	0	&
C	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	=

Thus, we found three occurrences of x in y. The first is at position 1, the second at position 3, and the last at position 7.

4. EXPERIMENTAL RESULTS

This section presents experimental results in order to compare the behavior of the SIMD-OPPM algorithm against the best known solutions in the literature for the OPPM problem.

The tests were run on an Intel 2.70 GHz i7 processor running Ubuntu 12.10 with 16 GB of memory. All the algorithms were implemented using the C programming language and run in the modified testing framework of Hume and Sunday [18].

We tested our algorithm SIMD-OPPM against the most effective previous solutions, which include S2OPPM and S4OPPM [4], SSEOPPM and AVXOPPM [5], FFK-OPPM [13], and SKIP-OPPM [12]. S2OPPM and S4OPPM [4] solutions are based on SBNDM2 and SBNDM4 [19]. SSEOPPM and AVXOPPM [5] represent the online solution grounded on SSE4.2 and AVX instruction set, respectively. FFK-OPPM [13] presents the filtration approach by Faro and Külekci. SKIP-OPPM [12] represents the solution based on Skip Search algorithm.

We tested our algorithms against a random and a real data set. The real data comprises the time series of relative humidity of UK. The data contains 33,510 integers representing the relative humidity of UK in percentage in the years 1961–1990. The random text of 4 MB contains integers between −128 and 127. We randomly selected 300 and 200 patterns of length 5, 10, 15, 20, 25, 30, and 50 from random and real data, respectively. Tables I and II show the average execution times

Table II. Execution times of algorithms in seconds for random data.

m	S2OPPM	S4OPPM	SSEOPPM	AVXOPPM	FFK-OPPM	SKIP-OPPM	SIMD-OPPM
5	1.186	1.549	1.587	—	0.922	0.836	$\boxed{0.125}$
10	0.544	0.587	0.483	0.412	0.691	0.601	$\boxed{0.123}$
15	0.354	0.389	0.312	0.211	0.413	0.357	$\boxed{0.116}$
20	0.257	0.291	0.292	0.188	0.309	0.269	$\boxed{0.111}$
25	0.205	0.242	0.281	0.151	0.254	0.229	$\boxed{0.106}$
30	0.176	0.186	0.256	0.144	0.234	0.204	$\boxed{0.102}$
50	0.172	0.179	0.246	0.131	0.206	0.199	$\boxed{0.089}$

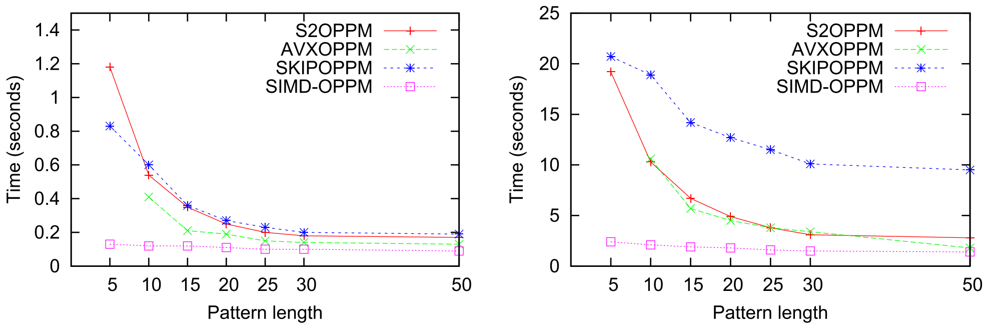

Figure 3. Execution times of algorithms for random data (left) in seconds and for relative humidity data (right) in milliseconds.

Table III. Execution times of algorithms in milliseconds for relative humidity data.

m	S2OPPM	S4OPPM	SSEOPPM	AVXOPPM	FFK-OPPM	SKIP-OPPM	SIMD-OPPM
5	19.24	31.84	28.55	—	21.36	20.72	$\boxed{2.43}$
10	10.35	10.95	11.26	10.68	18.88	18.91	$\boxed{2.15}$
15	6.72	6.86	6.94	5.77	13.54	14.23	$\boxed{1.99}$
20	4.91	4.81	6.38	4.56	9.92	12.76	$\boxed{1.87}$
25	3.83	3.73	5.39	3.85	6.93	11.59	$\boxed{1.63}$
30	3.15	2.95	4.96	3.41	5.36	10.17	$\boxed{1.54}$
50	2.86	2.67	3.87	1.83	3.88	9.56	$\boxed{1.49}$

per pattern set of all the algorithms for random data in seconds and humidity data in milliseconds, respectively. A graph on times for both the data sets is also shown in Figure 3.

The tests were made with 180 repeated runs. In Tables I and II, S2OPPM represents the algorithm based on SBNDM2 filtration, S4OPPM represents the algorithm based on SBNDM4 filtration, SSEOPPM represents the algorithm based on SSE4.2 instruction set architecture, AVXOPPM represents the algorithm based on AVX instruction set architecture, and FFK-OPPM represents best execution times for both the filtration approaches in [13]. SKIP-OPPM represents the solution based on Skip Search algorithm, and SIMD-OPPM represents our new algorithm.

From Tables II and III, it can be seen that SIMD-OPPM is the fastest for all tested values of m. For both the data sets, the difference between the execution times of SIMD-OPPM and other solutions is the maximum when $m = 5$, and thereafter, the difference drops. Moreover, our algorithm is faster in case of real data than for random data. We noticed that our algorithm becomes slightly faster when m increases likely because of reduced number of popcount operations.

5. CONCLUDING REMARKS

We proposed an efficient solution for OPPM. Our solution employs the SSE4.2 instruction set architecture. SIMD instructions were originally developed for multimedia but are recently employed for pattern matching. Our results show that SIMD instructions can also be very efficient in order-preserving matching as well. We place special emphasis on the practical efficiency of the algorithm. Therefore, we show with practical experiments that our solution is faster than the previous solutions.

REFERENCES

1. Kubica M, Kulczynski T, Radoszewski J, Rytter W, Walen T. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters* 2013; **113**(12):430–433. DOI: 10.1016/j.ipl.2013.03.015.
2. Kim J, Eades P, Fleischer R, Hong S, Iliopoulos CS, Park K, Puglisi SJ, Tokuyama T. Order-preserving matching. *Theoretical Computer Science* 2014; **525**:68–79. DOI: 10.1016/j.tcs.2013.10.006.

3. Cho S, Na JC, Park K, Sim JS. Fast order-preserving pattern matching. *Proceedings of Combinatorial Optimization and Applications - 7th International Conference, COCOA 2013, Chengdu, China, December 12-14, 2013*, Springer International Publishing, Switzerland, 2013; 84–95. DOI: 10.1007/978-3-319-02432-5 13.
4. Chhabra T, Tarhio J. Order-preserving matching with filtration. *Proceedings of Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014*, Springer International Publishing, Switzerland, 2014; 307–314. DOI: 10.1007/978-3-319-07959-2 26.
5. Chhabra T, Külekci MO, Tarhio J. Alternative algorithms for order-preserving matching. *Proceedings Of The Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015*, Czech Technical University, Prague, 2015; 36–46. Available from: http://www.stringology.org/event/2015/p05.html [last accessed 27 July 2016].
6. Belazzougui D, Pierrot A, Raffinot M, Vialette S. Single and multiple consecutive permutation motif search. *Proceedings of Algorithms and Computation - 24Th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013*, Springer-Verlag Berlin Heidelberg, 2013; 66–77. DOI: 10.1007/978-3-642-45030-3 7.
7. Knuth DE, Jr. James HM, Pratt VR. Fast pattern matching in strings. *SIAM Journal on Computing* 1977; **6**(2): 323–350. DOI: 10.1137/0206024.
8. Boyer RS. A fast string searching algorithm. *Communications of the ACM* 1977; **20**(10):762–772.
9. Faro S, Külekci MO. Fast packed string matching for short patterns. *Proceedings of the 15th Meeting On Algorithm Engineering And Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013*, SIAM, Philadelphia, USA, 2013; 113–121. DOI: 10.1137/1.9781611972931.10.
10. Faro S, Külekci MO. Fast and flexible packed string matching. *Journal of Discrete Algorithms* 2014; **28**:61–72. DOI: 10.1016/j.jda.2014.07.003.
11. Jeong H, Kim S, Lee W, Myung S. Performance of SSE and AVX instruction sets. *CoRR* 2012; **abs/1211.0820**.
12. Cantone D, Faro S, Külekci MO. An efficient skip-search approach to the order-preserving pattern matching problem. *Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015*, Czech Technical University, Prague, 2015; 22–35. Available from: http://www.stringology.org/event/2015/p04.html [last accessed 27 July 2016].
13. Faro S, Külekci MO. Efficient algorithms for the order preserving pattern matching problem. *Proceedings of the 11th International Conference on Algorithmic Aspects in Information and Management, AAIM 2016, July 18-20, Bergamo, Italy*, Springer, Switzerland, 2016; 185–196.
14. Charras C, Lecroq T, Pehoushek JD. A very fast string matching algorithm for small alphabeths and long patterns (extended abstract). *Proceedings of Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998*, Springer-Verlag Berlin Heidelberg, 1998; 55–64. DOI: 10.1007/BFb0030780.
15. Intel. *Intel Architecture Instruction Set Extensions Programming Reference*, 2004. Available from: https://software.intel.com/sites/default/files/m/9/2/3/41604/ [last accessed 27 July 2016].
16. Arndt J. *Matters computational*, 2009. Available from: http://www.jjj.de/fxt/ [last accessed 27 July 2016].
17. Intel. *Intrinsic Guide*. Available from: https://software.intel.com/sites/landingpage/IntrinsicsGuide [last accessed 27 July 2016].
18. Hume A, Sunday D. Fast string searching. *Software—Practice and Experience* 1991; **21**(11):1221–1248.
19. Durian B, Holub J, Peltola H, Tarhio J. Improving practical exact string matching. *Information Processing Letters* 2010; **110**(4):148–152. DOI: 10.1016/j.ipl.2009.11.010.

© 2016 The Authors Software: Practice and Experience
Published by John Wiley & Sons Ltd.