Filter Based Fast Matching of Long Patterns by Using SIMD Instructions

M. Oğuzhan Külekci

TÜBİTAK-UEKAE
National Research Institute of Electronics & Cryptology
41470 Gebze, Kocaeli, Turkey
kulekci@uekae.tubitak.gov.tr

Abstract. SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data simultaneously via a set of special instructions working on limited number of special registers. Although the usage of SIMD is explored deeply in multimedia processing, implementation of encryption/decryption algorithms, and on some scientific calculations, it has not been much addressed in pattern matching. This study introduces a filter based exact pattern matching algorithm for searching long strings benefiting from SIMD instructions of Intel's SSE (streaming SIMD extensions) technology. The proposed algorithm has worst, best, and average time complexities of $O(n \cdot m)$, $O(n/m)$, and $O(n/m + n \cdot m/2^{16})$ respectively, while searching an m bytes pattern on a text of n bytes. Experiments on small, medium, and large alphabet text files are conducted to compare the performance of the new algorithm with other alternatives, which are known to be very fast on long string search operations. In all cases the proposed algorithm is the clear winner on the average. When compared with the nearest successor, the matching speed is improved in orders of magnitude on small alphabet sequences. The performance is 40 % better on medium alphabets, and 50 % on natural language text.

Keywords: pattern matching, filtering, SIMD, SSE

1 Introduction

Searching for exact or approximate matches of given pattern(s) on a text file is one of the fundamental problems in computer science. Numerous algorithms focusing on some aspects of the general problem have been developed during the last three decades, some of which can be found in [4,5]. Although the main problem is well studied, recent advances in genomics research, new developments in processor architectures, and the accelerated growth of information on the Internet introduces new challenges in the area.

This study focuses on exact matching of long patterns on random sequences via a filtering methodology. Instead of checking the occurrence of the pattern(s) on all over the text, filtering methods first detects the portions of the text, on which the observation of the pattern is probable with a fast heuristic, and then performs a full verification on those positions reported by the filtering phase. Thus, a filter based string matching algorithm is actually composed of two parts, as filtering and the verification. The first part aims to detect possible match positions on the text without a deep investigation, and the verification process is checking the real existence of the pattern on those detected positions.

Some of the previous filter based pattern matching algorithms may be listed as follows. The algorithm of Wu&Manber [18] combines bit-parallelism with a fast 2-gram hashing heuristic filter. Later on, their algorithm is implemented as the *agrep* [17]

M. Oğuzhan Külekci: Filter Based Fast Matching of Long Patterns by Using SIMD Instructions, pp. 118–128.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2 © Czech Technical University in Prague, Czech Republic

approximate match utility program, which is known to be very powerful especially on approximate and multiple pattern matching. The average optimal (AOSO) and fast average optimal (FAOSO) variants of the original shift-or [2] algorithm defined by Fredriksson&Grabowski [7] may also be viewed from a filtering perspective since they include a verification procedure.

The bit-parallel algorithms [14,15,9,7] that suffer from the computer word size limitation[1] in general can also be used in a filtering framework for searching patterns longer than the computers word size. In such cases, the part of the pattern, which is selected to be less than the word length, is searched on the text by the bit-parallel algorithms, and rest of the pattern is verified on match positions. Moreover, character overloading for searching long patterns or multiple patterns with bit-parallel techniques has been proposed previously [6,15,2] also.

More recently, Lecroq [13] has offered one of the most effective representative of filtering algorithms. The simplicity and average speed of the Lecroq's *new* algorithm makes it a strong candidate in all practical cases including search on small alphabets.

The power of a filtering algorithm may be measured by two metrics: i) the distinguishing power of the proposed filtering method, ii) the computation speed of the filtering function. If the filter is not very selective, then the average number of calls to the verification procedure grows, which in turn degrades the performance. On the other side, if the distinguishing power is good, but the computation of the filter is expensive, then the speed again falls as it will consume more time to calculate the filter value, although the recall of verification is small. This study aims to benefit from the intrinsic SIMD instructions of the modern processors for fast calculation of a distinguishing filter.

SIMD instructions let simultaneous execution of some operands on multiple data by the help of a limited number of special registers. Figure 1 sketches the operation on 128 bit SSE registers, x, y, z. In the example, each register is divided into 4 integers of 32 bit each, and the given operation $\ominus$ is performed and stored between the corresponding data. Note that instead of using 4 integer portions, several other type definitions exist on SSE intrinsics, such as viewing the 128 bit as 16 bytes, or 4 floats also.

The original idea of SIMD was to speed up multimedia procedures, such as audio/video/image processing issues. It is also used in cryptographic applications and on some scientific computations. A good review of SIMD may be found in [8]. Despite the fact that it has not been explored deeply in pattern matching, this study shows that it serves as a good basis especially for filtering techniques.

$$\begin{array}{|c|c|c|c|c|}
\hline x_0 & \ominus & y_0 & \Rightarrow & z_0 \\
\hline x_1 & \ominus & y_1 & \Rightarrow & z_1 \\
\hline x_2 & \ominus & y_2 & \Rightarrow & z_2 \\
\hline x_3 & \ominus & y_3 & \Rightarrow & z_3 \\
\hline
\end{array}$$

Figure 1. The sketch of a sample SIMD instruction.

The algorithm introduced in this study, which will be referred as *SSEF*, uses Intel streaming SIMD extensions (SSE [11]) technology. SSEF finds exact occurrences of patterns longer than 32 bytes on random sequences. Experimental results indicated

[1] Külekci [12] has proposed a bit-parallel algorithm which is not restricted with the computer word size limitation.

that on the average it is approximately 6 times faster than Lecroq's new algorithm, and 15% better than the backward oracle and suffix oracle methods, which are mainly the best choices for long patterns until now.

2 Preliminaries and Basics

Let string S of k characters be shown as $S = s_0 s_1 s_2 \cdots s_{k-1}$. Assuming each character is represented by a single byte, $S[i \dots j]$ shows the byte array $[s_i s_{i+1} s_{i+2} \cdots s_j]$, where $0 \le i \le j < k$. The individual bits of byte s_i are denoted by $s_i = b_0^i b_1^i b_2^i b_3^i b_4^i b_5^i b_6^i b_7^i$, where b_0^i is referred as $sign(s_i)$. In chunks of 16 bytes, same string is represented by $S = C^0 C^1 C^2 \cdots C^{\lfloor (k-1)/16 \rfloor}$, where $C^i = s_{i \cdot 16} s_{i \cdot 16+1} s_{i \cdot 16+2} \cdots s_{i \cdot 16+15}$, for $0 \le i \le \lfloor (k-1)/16 \rfloor$. The last block $C^{\lfloor (k-1)/16 \rfloor}$ is not complete if $k \ne 0$ mod 16. In that case, the remaining bytes of the block are set to zero as $s_j = 0$ for $k - 1 < j$.

Given text T and pattern P of lengths n and m bytes, the number of 16-byte blocks in T and P are denoted by $N = \lceil n/16 \rceil$ and $M = \lceil m/16 \rceil$ respectively. The individual bytes of text T are accessed by t_i, $0 \le i < n$, and similarly the 16-byte blocks are addressed by D^i, $0 \le i < N$. The byte and block symbols for pattern P are p_i, $0 \le i < m$, and Q^i, $0 \le i < M$ respectively. Figure 2 demonstrates the defined structure.

D^0	D^1		D^{N-1}
$t_0 t_1 \dots t_{15}$	$t_{16} t_{17} \dots t_{31}$		$t_{(N-1)\cdot 16} t_{(N-1)\cdot 16+1} \cdots t_{n-1}$

a) The representation of text T.

Q^0	Q^1	....	Q^{M-1}
$p_0 p_1 \dots p_{15}$	$p_{16} p_{17} \dots p_{31}$	....	$p_{(M-1)\cdot 16} p_{(M-1)\cdot 16+1} \cdots p_{m-1}$

a) The representation of pattern P.

Figure 2.

The proposed filtering algorithm is designed to be effective on long patterns, where the lower limit for m is 32 ($32 \le m$). Although it is possible to adapt the algorithm for lesser lengths, the performance gets worse under 32. The number L is defined as $L = \lfloor m/16 \rfloor - 1$, which is the zero-based address of the last 16-byte block of Q whose individual bytes are totally composed of pattern bytes without any padding. For example, if $m = 42$, the 16-byte blocks of the pattern will be $Q = Q^0 Q^1 Q^2$, where the last 6 bytes of Q^2 are padded with zero. The L value for $m = 42$ is $L = 1$, which indicates the last whole block of the pattern is Q^1. Actually, if length of the pattern is a multiple of 16, there is no remainder in the last 16-byte block, and thus, $L = M - 1$. In the other case, L should point to the block preceding the last one as the last one is not a complete block, making $L = M - 2$.

The basic idea of the proposed algorithm is to compute a filter on block $D^{z \cdot L + L}$, where $0 \le z < \lfloor N/L \rfloor$, to explore if it is appropriate to observe pattern P beginning from any byte inside the prior blocks $D^{z \cdot L}$ to $D^{z \cdot L + (L-1)}$. If the filter value indicates some of the alignments are possible, then those fitting ones are compared with the text byte by byte.

Figure 3 demonstrates this basic idea by assuming $i = z \cdot L$. Note that as $m \geq 32$, and $L = \lfloor m/16 \rfloor - 1$, the pattern fills the bytes in D^{i+L} always.

Block No	D^i	D^{i+1}		....	D^{i+L-1}		D^{i+L}		
Bytes of T	$t_{i\cdot16}$	$t_{(i+1)\cdot16}$	...	....	$t_{(i+L-1)\cdot16}$	...	$t_{(i+L)\cdot16}$	...	$t_{(i+1)\cdot16+15}$
P aligned to $t_{i\cdot16}$	p_0	p_{16}	...	....	$p_{(L-1)\cdot16}$	...	$p_{L\cdot16}$	...	$p_{L\cdot16+15}$
P aligned to $t_{i\cdot16+1}$	p_0	p_{15}	...	....	$p_{(L-1)\cdot16-1}$	...	$p_{L\cdot16-1}$	...	$p_{L\cdot16+14}$
	....								
P aligned to $t_{i\cdot16+15}$	.... p_0	p_1	...	....	$p_{(L-1)\cdot16-15}$	...	$p_{L\cdot16-15}$	...	$p_{L\cdot16}$
	....								
P aligned to $t_{(i+L)\cdot16-1}$						p_0	p_1	...	p_{16}

Figure 3. Appropriate alignments of pattern P according to the filter value computed from D^{i+L},for any $i = z \cdot L$

3 The SSEF Exact Pattern Matching Algorithm

3.1 Preprocessing

The preprocessing stage of the algorithm consist of compiling the possible filter values of the pattern according to the alignments shown in figure 3. Formally, the filter values for $P[(L \cdot 16) \dots (L \cdot 16 + 15)]$, $P[(L \cdot 16 - 1) \dots (L \cdot 16 + 14)]$, $\dots$, $P[1 \dots 16]$ are computed and stored in a linked list, which will be referred as $FList$ from now on. The pseudo-code of the preprocessing procedure is depicted in Algorithm 1.

Algorithm 1 PreProcess($P = p_0p_1p_2 \cdots p_{m-1}$,$K$)

```
1: for $i = 0$ to 65535 do
2:     $FList[i] = \emptyset$;
3: end for
4: $L = \lfloor m/16 \rfloor - 1$
5: for $i = 0$ to $L \cdot 16 - 1$ do
6:     $r = L \cdot 16 - i$;
7:     $f = sign(p_i << K) \cdot 2^{15} + sign(p_{i+1} << K) \cdot 2^{14} + \cdots + sign(p_{i+15} << K)$
8:     $FList[f] = FList[f] \cup i$;
9: end for
10: return $L$;
```

The corresponding filter of a 16 bytes sequence is the 16 bits formed by concatenating the sign bits of each byte after shifting by K bits as shown in line 7 of Algorithm 1. The reason for shifting is to generate a distinguishing filter. For example, when the search is to be performed on an English text, the sign bits of bytes are generally 0 as in the standard ascii table the printable characters of the language reside in first 128, where the sign bits are always 0. If we do not include a shift operation, then the filter f value will be 0 in all cases, and while passing over the text verification will be called at each byte. On the other hand, if the text we are searching on is composed of uniformly distributed random 256 bytes, then there is obviously no need for shifting.

Hence, the K value is to be decided depending on the alphabet size and character distribution of the text. K should be set to a value that the most informative bit of the byte must become the sign bit after shift operation. Thus, detection of the most informative bit among the 8 bits of a byte is required for best filtering. This is actually the position on which the distribution of the bits among the whole text is close to their expected values. Note that this requires an additional pass over the whole text, which is not good in practice. A more practical approach may be to consider just the alphabet, and assume the distribution of characters is uniform on the given text. In that case, we are left with just the $|\Sigma|$ bytes, and it is more convenient to decide on the bit position. As an example, let's consider pattern matching on an ascii coded plain DNA sequence, where the alphabet is 'a','t','c','g' having ascii codes 01100001, 01110100, 01100011, and 01100111 respectively. The first three bits and the fifth bit are all same. Since the number of 1s and 0s are equal on the sixth and seventh positions from the remaining bits, one of them, say 6^{th}, may be used as the distinguishing bit. Thus, while searching on a DNA sequence, setting $K = 5$ to move this bit to the sign bit position would be a good choice when only the alphabet is considered.

3.2 Main algorithm

The pseudo code given in Algorithm 2 depicts the skeleton of the $SSEF$. After the preprocessing stage, the main loop investigates 16-byte blocks of text T in steps of L. If the filter f computed on D^i, where $i = z \cdot L + L$, and $0 \leq z < \lfloor N/L \rfloor$, is not empty, then the appropriate positions listed in $FList[f]$ are verified accordingly.

Algorithm 2 SSEF($P = p_0p_1p_2 \cdots p_{m-1}$, $T = t_0t_1t_2 \cdots t_{m-1}$)

```
 1: Set $K = a$, $0 \leq a < 8$, according to the alphabet;
 2: $i = L$ =PreProcess(P,K);
 3: while $i < N$ do
 4:     $f = sign(t_{i \cdot 16} << K) \cdot 2^{15} + sign(t_{i \cdot 16+1} << K) \cdot 2^{14} + \cdots + sign(t_{i \cdot 16+15} << K)$
 5:     for all $j \in FList[f]$ do
 6:         if $P = [t_{(i-L) \cdot 16+j} \cdots t_{(i-L) \cdot 16+j+m-1}]$ then
 7:             pattern detected at $t_{(i-L) \cdot 16+j}$;
 8:         end if
 9:     end for
10:     $i = i + L$;
11: end while
```

$Flist[f]$ contains a linked list of integers marking the beginning of the pattern. While investigating the filter on D^i, if $FList[f]$ contains number j, where $0 \leq j < 16 \cdot L$, the pattern potentially begins at $t_{(i-L) \cdot 16+j}$. In that case, a complete verification is to be performed between P and $[t_{(i-L) \cdot 16+j} \cdots t_{(i-L) \cdot 16+j+m-1}]$.

Calculating the corresponding filter of D^i via SSE intrinsics The computation of the filter f of D^i in line 4 of pseudo code given in Alg. 2 is performed by 2 SSE2 intrinsic functions as

```
$tmp128 = \_mm\_slli\_epi64(D^i, K)$;
$f = \_mm\_movemask\_epi8(tmp128)$;
```

First instruction shifts the corresponding 16 bytes of the text D^i by K bits and stores the result in a temporary 128 bit register aiming not to destruct D^i itself.

Second, the instruction `mm_movemask_epi8` returns a 16 bit mask composed of the sign bits of the individual 16 bytes forming the 128 bit value. Figure 4 demonstrates this function.

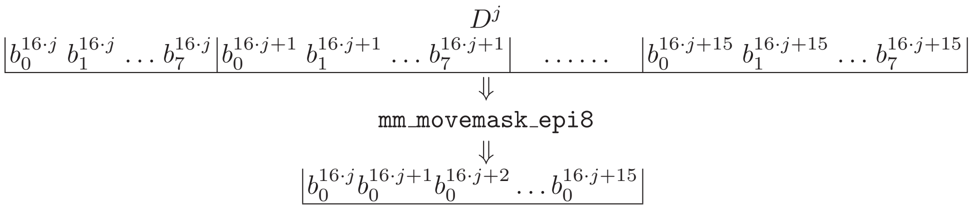
D^j
$b_0^{16\cdot j}\ b_1^{16\cdot j} \ldots b_7^{16\cdot j}$
$b_0^{16\cdot j+1}\ b_1^{16\cdot j+1} \ldots b_7^{16\cdot j+1}$
$\ldots\ldots$
$b_0^{16\cdot j+15}\ b_1^{16\cdot j+15} \ldots b_7^{16\cdot j+15}$
$\Downarrow$
mm_movemask_epi8
$\Downarrow$
$b_0^{16\cdot j} b_0^{16\cdot j+1} b_0^{16\cdot j+2} \ldots b_0^{16\cdot j+15}$

Figure 4. The `mm_movemask_epi8` SSE instruction as the filter.

4 Complexity Analysis

The preprocessing stage of the $SSEF$ algorithm requires an additional space to store the 65536 items of $FList$ linked list. On a 32 bit machine, assuming each node consist of an integer and a next pointer, this makes up a total of 512 KB (= 65536 × 8 byte) memory requirement.

The first loop in Algorithm 1 just initializes the $FList$ list, and the second for loop is run $L \cdot 16$ times during the preprocessing. Thus, time complexity of preprocessing is $O(L \cdot 16)$ that approximates to $O(m)$.

$SSEF$ algorithm investigates the N 16-byte block text T in steps of L blocks. Total number of filtering operations is exactly $\lfloor N/L \rfloor$. At each attempt, maximum number of verification requests is $L \cdot 16$, since the filter gives information about that number of appropriate alignments of the pattern. This situation can also be viewed from figure 3. On the other hand, if the computed filter is empty, then there is obviously no need for verification. The verification cost is assumed to be $O(m)$ with the brute-force checking of the pattern.

From these facts, the best case complexity is $O(\lfloor N/L \rfloor)$, and worst case complexity is $O(\lfloor N/L \rfloor \cdot (L \cdot 16) \cdot m)$. Remembering the definitions of N and L as $N = \lceil n/16 \rceil$, and $L = \lfloor m/16 \rfloor - 1$, the best/worst time complexities approximately converges to $O(n/m)$ and $O(n.m)$ respectively, which are equivalent to standard Boyer-Moore [3] algorithm.

There are at most $L \cdot 16$ distinct filter values for any given pattern among the possible 65536 values. Hence, the probability that the filter computed on D^{i+L} hits to a non-empty set is $L \cdot 16/65536$. This indicates that verification will be requested for $\lfloor N/L \rfloor \times (L \cdot 16/65536)$ times during the whole execution, assuming characters of the text is randomly uniform distributed. The average case complexity, being sum of the filter computation time and verification computation time, is then $O(\lfloor N/L \rfloor + \lfloor N/L \rfloor \times (L \cdot 16/65536) \times m)$, which converges to $O(n/m + n \cdot m/65536)$.

5 Implementation and Experimental Results

The SSEF algorithm is implemented on 64 bit Intel Xeon processor with 3 GB of memory. All of the algorithms included in tests are compiled with *GNU C* compiler `gcc 4.1.2` with full optimization turned on by `-O3` flag.

The SSE instructions used in the study require the source data to be 16-byte aligned for best performance. The cost of misalignment is very high [16,10,11], and special attention was paid to make sure that the text is properly aligned. For that purpose the input text is loaded to the memory ensuring that it is 16-byte aligned by using *union* aggregate with *__m128i* data type introduced by SSE intrinsics as shown in figure 5.

```
typedef union{
        __m128i* data16;
        unsigned char* data;
} TEXT;
```

Figure 5. The `TEXT` data type defined for 16-byte alignment of data.

The performance of SSEF algorithm is compared with:

– Lecroq's *q-hash* algorithm, which is one of the best filtering algorithms [13], with ranks $q = 3$ (*3-hash*) and $q = 8$ (*8-hash*).
– The quick search (QS) of Sunday, which is a fast implementation of standard Boyer-Moore [3].
– The BLIM of Külekci [12], as this bit-parallel algorithm is not limited with the computer word size, and thus can be run on long patterns also.
– Fast variants of backward oracle and suffix oracle matching [1]. BOM2 and BSOM2 are especially fast on long patterns.

	$\|\Sigma\| = 256$ 2-bit encoded DNA sequence							$\|\Sigma\| = 128$ English text						
Len.	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
32	11,06	10,69	11,05	10,53	9,46	9,47	12,03	10,38	9,44	9,75	10,05	9,65	10,04	10,88
96	12,16	11,11	10,85	11,35	9,03	9,12	9,69	11,24	9,80	9,57	10,36	8,51	8,56	8,74
160	13,20	15,14	14,64	16,06	6,60	6,88	6,64	12,50	13,31	12,85	12,56	8,08	8,03	5,95
224	14,18	14,33	13,94	15,29	5,09	5,28	4,66	13,72	12,29	12,11	12,20	6,30	6,23	4,21
288	15,14	13,87	13,57	14,43	4,12	4,23	3,58	14,89	11,57	11,66	12,03	5,11	5,06	3,21
352	16,16	13,32	13,09	13,95	3,45	3,59	2,91	16,06	11,12	11,26	11,66	4,46	4,41	2,58
416	17,14	12,81	12,66	13,28	2,96	3,05	2,44	17,22	10,43	10,67	11,32	3,82	3,75	2,17
480	18,07	12,30	12,20	12,84	2,56	2,62	2,16	18,43	10,00	10,33	11,19	3,50	3,41	1,80
544	19,14	11,87	11,85	12,55	2,22	2,24	1,90	19,53	9,38	9,85	11,05	3,03	3,02	1,62
608	20,26	11,58	11,53	12,16	1,93	1,96	1,68	20,34	9,07	9,64	10,62	2,85	2,79	1,45
672	21,01	11,33	11,23	11,86	1,69	1,67	1,54	21,26	8,65	9,36	10,39	2,60	2,55	1,34
736	22,06	11,14	11,07	11,72	1,52	1,50	1,36	22,16	8,52	9,16	10,00	2,48	2,36	1,21
800	23,00	11,00	10,89	11,53	1,37	1,41	1,21	23,22	8,28	9,04	9,91	2,28	2,26	1,13
864	23,99	10,78	10,78	11,30	1,31	1,29	1,15	24,01	7,96	8,78	9,51	2,14	2,08	1,06
928	25,07	10,76	10,74	11,40	1,20	1,24	1,09	25,05	7,75	8,67	9,28	2,00	1,99	1,01
992	26,22	10,66	10,67	11,22	1,19	1,20	0,98	25,92	7,46	8,54	8,99	1,94	1,91	0,90
1056	27,41	10,62	10,63	11,08	1,16	1,18	0,96	27,16	7,26	8,54	8,97	1,78	1,81	0,92
1248	30,82	10,48	10,53	10,89	1,10	1,15	0,92	30,44	7,07	8,39	8,37	1,72	1,70	0,84
1440	34,19	10,40	10,51	10,51	1,11	1,16	0,82	33,75	6,75	8,20	7,93	1,60	1,64	0,80
1632	38,05	10,42	10,40	10,64	1,16	1,17	0,84	37,20	6,53	8,21	7,67	1,57	1,60	0,76
1824	41,85	10,49	10,44	10,56	1,22	1,21	0,82	41,35	6,57	8,21	7,72	1,57	1,60	0,75
2000	47,09	10,40	10,46	10,99	1,28	1,25	0,81	45,84	6,26	8,08	7,62	1,57	1,60	0,76
Avg.	27,79	11,27	11,2	11,67	2,33	2,37	2,14	27,4	8,22	9,16	9,36	2,96	2,96	1,94

Table 1. Experimental comparison of algorithms on large alphabets.

Benchmarks are conducted on various text files having small ($\Sigma = \{2, 4\}$), medium ($\Sigma = \{16, 20\}$), and large ($\Sigma = \{128, 256\}$) alphabets. In practice, small alphabets mimic the nucleic acid sequences, and middle alphabets correspond to biological sequences with larger blocks such as amino acids or proteins. Large alphabets represent the case for natural languages, and series of random bytes such as the compressed files.

The summary of the data sets[2] used in the experiments are given in Table 2. The distribution of characters are randomly uniform on all data sets except the 5^{th} one, which is a natural language text. Remembering the discussion in section 3, it is enough to consider the character codings of the alphabet while deciding on the value of bit shift amount K on test files except the English text. On natural language text file, the experiment is repeated for all possible K values as $K = 0, \dots, 7$. It is observed that the performances are compatible for $K \in \{3, 4, 5, 7\}$, and significantly worse on $K \in \{1, 2\}$. Obviously, selecting $K = 0$ is the worst since it does not include any distinguishing power on the set of printable ascii characters.

	$\|\Sigma\|$	Data set	Size	K-bit shift
1	2	Uniformly distributed random sequence of two characters ('a' and 'b').	30 MB	6
2	4	Plain ASCII coded DNA sequence from Manzini's DNA corpus	21.6 MB	5
3	16	Uniformly distributed random sequence of 16 characters ('a' ... 'p').	30 MB	7
4	20	Uniformly distributed random sequence of 20 characters ('a' ... 't').	30 MB	7
5	128	English text from `enwik8` corpus.	20 MB	7
6	256	2-bit encoded DNA sequence from Manzini's DNA corpus.	22.7 MB	0

Table 2. Test files used in the experiments.

Patterns of length 32 to 2000 are randomly selected from the input text, and searched via the included algorithms. 100 samples are taken for each length, and each sample is matched 10 times on the text. The mean user times are recorded by `getrusage` function.

Tables 1 and 3 compare the timings of BLIM, 3-hash, 8-hash, QS, BOM2, BSOM2 and SSEF for various pattern lengths in *milliseconds*. Experimental results indicate that the SSEF algorithm is the clear winner on all tested alphabet sizes and followed by the BOM2 and BSOM2 algorithms, which are actually known to be the fastest ones on long pattern matching. The performance of BOM2 and BSOM2 are quite good, but with the increasing length of the patterns, the SSEF becomes more dominant. The performances of BOM2/BSOM2 and SSEF improves with the increased length, where Lec3 and Lec8 are not very much effected with the length.

Table 4 summarizes the average measured speeds of the algorithms in mega byte per seconds on tested alphabet sizes. Based on the overall speeds depicted in this table, the performance gain is maximum on small alphabets. SSEF is 3.62 and 2.47 times faster than its nearest successor on binary alphabet and plain text DNA sequences respectively. When medium size alphabets are concerned, it is 40 % faster than the following best. On natural language text, the performance of the BOM2/BSOM2 degrades a little bit since the underlying data is not uniform now, and thus, SSEF is 50 % more speedy in this case. When timings on 256-byte alphabets are investigated, 10 % improvement is observed according to the next best BOM2 algorithm.

SSEF is approximately more than 5 times faster than the *q-hash* family, which is one of the best representative of *filter-then-search* algorithms. Note that the speed

[2] Manzini's DNA compression benchmark corpus can be downloaded from `http://web.unipmn.it/manzini/dnacorpus`.
The enwik8.txt file is the subject of the Hutter Prize compression competition, and can be downloaded from `http://prize.hutter1.net`

	$|\Sigma| = 4$ Plain ascii DNA sequence							$|\Sigma| = 2$ Randomly uniform sequence of 2 characters						
Len.	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
32	15,19	10,95	10,54	49,48	16,20	20,56	11,87	37,09	52,06	14,62	212,37	39,52	52,17	16,17
96	12,30	11,04	10,40	50,27	10,72	11,71	9,45	19,12	51,98	14,54	218,18	21,27	25,40	13,06
160	13,44	12,54	14,06	51,37	15,77	16,30	6,53	20,22	51,67	19,46	224,04	31,59	35,95	8,98
224	14,59	12,60	13,42	53,10	12,22	12,53	4,56	21,38	51,88	18,45	223,00	23,60	26,61	6,32
288	15,68	12,69	12,99	50,60	10,06	10,21	3,46	22,51	50,74	18,02	213,45	19,04	21,51	4,87
352	16,84	12,64	12,65	52,89	8,69	8,81	2,85	23,65	52,40	17,47	225,81	16,20	18,06	4,00
416	17,94	12,67	12,03	50,04	7,53	7,66	2,36	24,68	51,86	16,77	218,05	14,13	15,63	3,36
480	19,04	12,74	11,62	49,88	6,75	6,83	2,03	25,92	51,53	16,27	222,09	12,61	13,87	2,91
544	20,24	12,72	11,25	49,96	6,11	6,12	1,82	26,96	51,48	15,90	218,66	11,43	12,47	2,60
608	21,31	12,57	10,92	52,87	5,56	5,65	1,61	28,09	51,17	15,45	221,21	10,50	11,38	2,36
672	22,45	12,56	10,70	51,64	5,12	5,14	1,46	29,18	49,81	15,20	216,25	9,77	10,50	2,21
736	23,50	12,42	10,47	51,07	4,76	4,78	1,35	30,32	51,23	14,90	215,31	9,16	9,75	1,95
800	24,70	12,48	10,16	51,76	4,44	4,45	1,22	31,45	52,17	14,62	220,85	8,59	9,06	1,81
864	25,87	12,21	9,98	51,40	4,13	4,20	1,15	32,46	49,82	14,58	216,94	8,19	8,55	1,62
928	26,87	12,44	9,98	51,37	3,88	3,91	1,11	33,67	50,32	14,42	219,99	7,76	8,08	1,55
992	28,04	12,25	9,80	49,86	3,68	3,70	1,02	34,88	49,84	14,44	204,53	7,38	7,62	1,46
1056	29,42	12,33	9,74	49,15	3,46	3,50	1,00	36,31	50,32	14,32	217,54	7,07	7,31	1,30
1248	33,10	12,22	9,56	49,44	2,95	3,04	0,92	39,67	49,15	14,07	210,31	6,14	6,23	1,19
1440	36,82	12,26	9,35	48,10	2,59	2,68	0,88	43,54	51,81	14,07	215,02	5,41	5,51	1,10
1632	40,67	12,18	9,16	53,39	2,40	2,46	0,83	47,15	50,50	14,15	228,67	4,82	4,89	1,00
1824	44,68	12,18	9,18	51,01	2,27	2,27	0,84	51,21	51,56	14,27	223,11	4,42	4,44	0,96
2000	50,42	12,17	8,96	50,21	2,18	2,37	0,78	57,04	51,08	14,04	214,72	4,16	4,04	0,93
Avg.	29,62	12,27	10,31	50,81	5,23	5,48	2,12	36,80	51,01	15,04	218,52	10,47	11,64	2,89

a) Benchmarks on small alphabet sequences.

	$|\Sigma| = 20$ Randomly uniform sequence of 20 characters							$|\Sigma| = 16$ Randomly uniform sequence of 16 characters						
Len.	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
32	14,89	14,40	14,60	15,50	13,28	13,50	15,76	15,00	14,46	14,62	16,75	13,32	13,68	16,40
96	15,87	15,30	14,48	15,12	12,32	12,47	12,79	16,00	15,34	14,48	16,15	12,48	12,71	13,06
160	17,04	19,67	19,37	15,21	10,67	11,10	8,78	17,03	18,89	19,44	15,96	11,79	12,13	9,01
224	18,16	19,23	18,50	15,22	8,33	8,66	6,16	18,16	18,61	18,55	16,12	9,26	9,44	6,32
288	19,29	18,90	18,02	15,22	6,96	7,21	4,76	19,34	18,48	17,96	16,18	7,57	7,74	4,91
352	20,42	18,70	17,38	15,20	6,04	6,21	3,86	20,45	18,30	17,40	16,06	6,39	6,54	3,99
416	21,55	18,55	16,83	15,30	5,30	5,48	3,31	21,54	18,30	16,81	16,13	5,53	5,70	3,40
480	22,64	18,34	16,20	15,26	4,69	4,92	2,89	22,67	18,17	16,28	15,94	4,84	5,05	2,92
544	23,78	18,10	15,78	15,20	4,21	4,40	2,58	23,86	18,02	15,89	16,07	4,34	4,53	2,58
608	24,89	17,96	15,39	15,38	3,83	4,06	2,33	24,95	17,90	15,43	16,11	3,98	4,15	2,33
672	26,04	17,70	15,14	15,20	3,54	3,71	2,13	26,10	17,76	15,11	16,04	3,64	3,80	2,15
736	27,19	17,68	14,86	15,22	3,23	3,43	1,94	27,19	17,54	14,87	15,97	3,38	3,53	1,92
800	28,26	17,50	14,56	15,22	3,01	3,14	1,77	28,22	17,54	14,64	16,14	3,14	3,27	1,74
864	29,45	17,32	14,52	15,18	2,74	2,96	1,64	29,50	17,42	14,48	16,06	2,97	3,06	1,64
928	30,49	17,18	14,49	15,18	2,57	2,98	1,49	30,61	17,29	14,38	16,16	2,84	2,89	1,53
992	31,73	17,14	14,31	15,14	2,41	2,65	1,38	31,80	17,26	14,30	16,16	2,69	2,81	1,47
1056	33,08	17,05	14,23	15,26	2,21	2,37	1,32	33,04	17,23	14,31	16,02	2,54	2,68	1,34
1248	36,59	16,84	14,17	15,18	1,97	2,08	1,11	36,78	16,96	14,13	16,10	2,30	2,35	1,17
1440	40,34	16,70	14,06	15,26	1,82	1,86	1,01	40,50	16,85	13,99	16,23	2,11	2,15	1,08
1632	44,08	16,56	14,04	15,24	1,71	1,81	0,96	44,04	16,86	14,04	16,08	2,00	2,01	1,00
1824	48,05	16,45	14,08	15,26	1,73	1,76	0,92	48,04	16,54	14,05	16,13	1,92	1,94	1,00
2000	53,94	16,39	14,09	15,20	1,68	1,80	0,94	53,16	16,51	14,14	16,20	1,93	2,02	0,92
Avg.	33,22	17,23	15,04	15,27	3,84	4,00	2,81	33,06	17,20	15,00	16,11	4,13	4,24	2,90

b) Benchmarks on medium alphabet sequences.

Table 3. Experimental comparison of algorithms on small and medium alphabets.

$\|\Sigma\|$	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
2	815,24	588,07	1994,18	137,29	2865,06	2577,82	10382,87
4	729,20	1760,55	2094,16	425,14	4126,17	3939,90	10201,31
16	907,38	1744,17	1999,72	1862,76	7269,64	7074,85	10347,06
20	903,09	1741,58	1994,93	1965,23	7809,45	7507,04	10659,09
128	729,85	2433,05	2182,84	2135,84	6750,49	6748,78	10307,95
256	816,72	2015,00	2026,76	1945,14	9749,29	9591,72	10585,84

Table 4. Average speed of the tested algorithms in MB/sec for each $|\Sigma|$ alphabet size.

of the proposed algorithm is not much effected with the size or distribution of the alphabet unlike its nearest competitors BOM2 and BSOM2.

6 Conclusion

This study introduced a filter-then-search type pattern matching algorithm for long patterns benefiting from computers intrinsic SIMD instructions. Using SIMD intrinsics has not been much addressed in pattern matching, and this study is an initial exploration of designing algorithms according to that technology, which is developing very fast.

The proposed SSEF algorithm is implemented on Intel's SSE (version 2) technology. Experimental benchmarks showed that on every alphabet sizes it is faster than all competitors included in this study. Considering the orders of magnitude performance gain on small and medium alphabet sizes, SSEF becomes a strong alternative for exact matching of long patterns on biological sequences.

The best and worst case time complexities being $O(n/m)$ and $O(n{\cdot}m)$ respectively are identical with the classical Boyer-Moore type algorithms. The main improvement comes with the average case complexity of $O(n \cdot m/2^{16})$. Note that the performance of the algorithm is independent of the alphabet size (assuming $|\Sigma| > 1$), and conducted experiments proves this empirically also.

References

1. C. Allauzen, M. Crochemore, and M. Raffinot: *Factor oracle: A new structure for pattern matching*, in Proceedings of SOFSEM'99, vol. 1725 of LNCS, Springer Verlag, 1999, pp. 291–306.
2. R. Baeza-Yates and G. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
3. R. Boyer and J. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
4. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's Collage Publications, 2004.
5. M. Crochemore and W. Rytter: *Jewels of stringology*, World Scientific Publishing, 2003.
6. K. Fredriksson: *Faster string matching with super–alphabets*, in Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE'2002), LNCS 2476, Springer–Verlag, 2002, pp. 44–57.
7. K. Fredriksson and S. Grabowski: *Practical and optimal string matching*, in Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE'2005), LNCS 3772, Springer–Verlag, 2005, pp. 374–385.

8. M. Hassaballah, S. Omran, and Y. Mahdy: *A review of SIMD multimedia extensions and their usage in scientific and engineering applications.* The Computer Journal, 51(6) November 2008, pp. 630–649.
9. J. Holub and B.Durian: *Fast variants of bit parallel approach to suffix automata.* Unpublished Lecture, University of Haifa, April 2005.
10. I. Hurbain and G. Silber: *An empirical study of some x86 simd integer extensions*, in Proceedings of CPC'2006, 12th International Workshop on Compilers for Parallel Computers, Spain, January 9–11 2006.
11. Intel Corporation, *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 2001.
12. M. O. Külekci: *A method to overcome computer word size limitation in bit-parallel pattern matching*, in Proceedings of 19th International Symposium on Algorithms and Computation, ISAAC'2008, S.-H. Hong, H. Nagamochi, and T. Fukunaga, eds., vol. 5369 of Lecture Notes in Computer Science, Gold Coast, Australia, December 2008, Springer Verlag, pp. 496–506.
13. T. Lecroq: *Fast exact string matching algorithms.* Information Processing Letters, 102(6) 2007, pp. 229–235.
14. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM Journal of Experimental Algorithms, 5(4) 2000, pp. 1–36.
15. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in LNCS 2857, Proceedings of SPIRE'2003, 2003, pp. 80–94.
16. A. Shahbahrami, B. Juurlink, and S. Vassiliadis: *Performance impact of misaligned accesses in simd extensions*, in Proceeedings of ASAP'05, IEEE Conference on Application Spesific Systems Architecture Procesors, Washington,DC, USA, 2005, pp. 393–398.
17. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*, in Proceedings of USENIX Winter 1992 Technical Conference, 1992, pp. 153–162.
18. S. Wu and U. Manber: *Fast text searching allowing errors.* Communications of the ACM, 35(10) 1992, pp. 83–91.