Inf. Process. Lett. 189 (2025) 106557

Contents lists available at ScienceDirect

Information Processing Letters

journal homepage: www.elsevier.com/locate/ipl

String searching with mismatches using AVX2 and AVX-512 instructions ☆

Tamanna Chhabra [a], Sukhpal Singh Ghuman [a], Jorma Tarhio [b,*]

[a] Faculty of Applied Science and Technology, Sheridan College, Ontario, Canada
[b] Department of Computer Science, Aalto University, Finland

ARTICLE INFO

Keywords:
Approximate string matching
Hamming distance
Exact string matching
SIMD computing
Experimental comparison

ABSTRACT

We present new algorithms for the k mismatches version of approximate string matching. Our algorithms utilize the SIMD (Single Instruction Multiple Data) instruction set extensions, particularly AVX2 and AVX-512 instructions. Our approach is an extension of an earlier algorithm for exact string matching with SSE2 and AVX2. In addition, we modify this exact string matching algorithm to work with AVX-512. We demonstrate the competitiveness of our solutions by practical experiments. Our algorithms outperform earlier algorithms for both exact and approximate string matching on various benchmark data sets.

1. Introduction

String matching [1] is a widely studied problem in Computer Science. The problem of string matching consists of two strings, a text and a pattern, and the task is to find all occurrences of the pattern in the text. Given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in a finite alphabet Σ, the problem of exact string matching is defined as follows: to find all the positions i such that $t_i t_{i+1} \cdots t_{i+m-1} = p_0 p_1 \cdots p_{m-1}$. In this paper we consider the k mismatches variation of the problem where P', a substring of T, is an occurrence of P, if $|P'| = |P|$ holds and P' has at most k mismatches with P, $0 \leq k < m$. The mismatch distance of two strings of equal length is also called the Hamming distance.

In our study, we propose algorithms that make use of SIMD (Single Instruction Multiple Data) computing for approximate string matching with k mismatches. By harnessing the AVX2 and AVX-512 features found in modern processors, our algorithms can process multiple characters simultaneously. Especially AVX2 is widely available in new Intel and AMD processors. To build upon existing work, we start with a simple algorithm [2] that already utilizes SIMD for exact string matching. We extend and modify this algorithm to handle mismatches.

Our main focus is on demonstrating experimentally the practical efficiency of the new algorithms. As a result, we not only achieve faster approximate string matching, but also surpass the speed of earlier algorithms designed for exact string matching. The improvement in approximate string matching is significant: In English data, when permitting one mismatch, our algorithm is approximately six times faster than a reference method.

Various algorithms have been developed to solve the string matching with k mismatches problem. Baeza-Yates and Gonnet [3] introduced the Shift-Add (SA) algorithm, the first bit-parallel solution for the k-mismatches problem, which operates in linear time for short patterns. Tarhio and Ukkonen's [4] Approximate Boyer-Moore (ABM) algorithm extends the Boyer-Moore-Horspool method to approximate string matching. Building on this, Liu et al. [5] developed the FAAST algorithm by optimizing ABM for small alphabets. Salmela et al. [6] later implemented EF, an even faster version of FAAST. Fiori et al. [7] developed several solutions applying SIMD. They introduced the ANS (Approximate Naive with SIMD) algorithm and its derivative ANS2B and an enhanced version of EF. The Baeza-Yates–Perlberg (BYP) [8] algorithm, which uses a partitioning strategy, also benefited from Fiori et al.'s [7] application of SIMD techniques. The resulting algorithms are called BYPSB and BYPSC. In addition to these practical approaches, theoretical solutions such as the kangaroo method [9,10], Clifford's solution [11], and algorithms based on the fast Fourier transform [12–14] have also been proposed.

The rest of the paper is organized as follows: Section 2 presents the background. Section 3 introduces our algorithms for approximate string matching, Section 4 describes adaptation of the approach to AVX-512, and Section 5 depicts the results of our practical experiments, and Section 6 concludes the article.

☆ A preliminary version was presented in The Prague Stringology Conference 2023.

* Corresponding author.
E-mail address: jorma.tarhio@aalto.fi (J. Tarhio).

https://doi.org/10.1016/j.ipl.2025.106557
Received 4 November 2023; Received in revised form 7 January 2025; Accepted 8 January 2025

Available online 13 January 2025
0020-0190/© 2025 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

2. Background

For exact string matching Tarhio et al. [2] presented a naive algorithm which uses the SIMD instruction architecture. The algorithm compares α characters in parallel, where α is 16 or 32. The 32 variation utilizes the AVX2 instruction set and it is called N32.

The key idea of N32 is to test α consecutive potential occurrences of the pattern in parallel. For that purpose, a comparison vector containing α copies of the same character is constructed for each character of the alphabet. N32 first compares the vector of p_0 with $t_0 \cdots t_{31}$, and then it compares the vector of p_1 with $t_1 \cdots t_{32}$ and so on. A bitvector of 32 bits keeps track of active match candidates. Matches are counted with a popcount function.

Tarhio et al. [2] introduced three versions of N32 which use three different orders for comparing the characters of the pattern: plain order (N32), fixed order (N32F), and reverse English frequency order (N32E). The plain order advances from left to right in the pattern. The fixed order applies the following heuristic order:

$$p_0,\ p_{m-1},\ p_3, p_6, \dots,\ p_2, p_5, \dots,\ p_1, p_4, \dots$$

excluding space characters which are compared last.

In addition to the SIMD instructions, loop peeling has a key role in the efficiency of N32. In loop peeling, a number of iterations is moved in front of the loop. As a result, the code becomes faster because of fewer loop tests. In loop unrolling, the whole loop is peeled. In the following, the term 'peeling factor' denoted as r refers to the number of the moved iterations. In their study, Tarhio et al. [2] applied values of $r = 2$ or $r = 3$ when working with the English data, and $r = 5$ when dealing with DNA data.

3. Algorithms for approximate matching

Our aim is to develop efficient algorithms for approximate string matching. Algorithm 1 (as shown below) counts all the occurrences of a given pattern string P in a text string T, with at most k mismatches. The algorithm uses SIMD and is a variation of the N32 algorithm extended with mismatch counting. It works by handling α consecutive starting positions of an occurrence candidate of P in parallel at a time. In the case of AVX2, α is 32. Bitvectors $found[0], \dots, found[k]$ of α bits are used to keep track of mismatches. The jth bit of these vectors represents the jth candidate. Initially, every bit of each $found[i]$ is set. Before starting the search process, a comparison vector for each position of the pattern is computed in line 1. Each comparison vector consists of α copies of the corresponding pattern character. Search proceeds from the leftmost to the rightmost position of the pattern. At each position, a bitvector cmp is calculated, representing the result of comparing the comparison vector of that position with a segment of the text, which contains α positions and which is shifted one by one. Subsequently, the $found[i]$ vectors for $i = k, k-1, \dots, 0$ are updated sequentially to track potential matches.

During computation, if the jth bit of $found[i]$ becomes zero, then more than i mismatches has been found while checking the jth candidate. If the jth bit of $found[k]$ stays one until the last position of the pattern has been processed, the jth candidate is an approximate occurrence of the pattern with k mismatches. If all the bits of $found[k]$ become zero, then none of the α candidates can be an occurrence of the pattern. The operators &, |, and >> denote bit-parallel and, or, and right shift, respectively.

The intrinsic function `_mm_popcnt_u32` [15] is used for counting matches in line 11. The SIMDcompare function for AVX2 is implemented using three intrinsic functions [15] as follows [2]:

```
SIMDcompare(x, y, 32)
    x_ptr = _mm256_loadu_si256(x)
    y_ptr = _mm256_loadu_si256(y)
    return _mm256_movemask_epi8(
              _mm256_cmpeq_epi8(x_ptr, y_ptr))
```

Algorithm 1 SIMD-approximate-search

```
1  for j ← 0 to m − 1 do construct vector(j) for p_j
2  count ← 0; i ← 0
3  while i ≤ n − m do
4     for j ← 0 to k do found[j] ← 2^α − 1
5     for j ← 0 to m − 1 do
6        cmp ← SIMDcompare(t_{i+j}, vector(j), α)
7        for s ← k downto 1 do
8           found[s] ← found[s] & (found[s − 1] | cmp)
9        found[0] ← found[0] & cmp
10       if found[k] = 0 then goto out
11    count ← count + popcount(found[k])
12    out: i ← i + α
13 count ← count − popcount(found[k] >> (n − m − i + α + 1))
```

Table 1
An example of computation of $found[1]$ in `aabaacaaa` for P = `abca`, $k = 1$, $\alpha = 6$. Vectors are shown in the order of the text. The starting positions of matches are underlined.

$found[1]$	1	1	1	1	1	1	aabaacaaa
$found[0]$	1	1	1	1	1	1	
cmp for a	1	1	0	1	1	0	aabaac
$found[1]$	1	1	1	1	1	1	
$found[0]$	1	1	0	1	1	0	
cmp for b	0	1	0	0	0	0	abaaca
$found[1]$	1	1	0	1	1	0	
$found[0]$	0	1	0	0	0	0	
cmp for c	0	0	0	1	0	0	baacaa
$found[1]$	0	1	0	1	0	0	
$found[0]$	0	0	0	0	0	0	
cmp for a	1	1	0	1	1	1	aacaaa
$found[1]$	0	1	0	1	0	0	
$found[0]$	0	0	0	0	0	0	

Because $n - m + 1$ is not divisible by α in a general case, the last execution of line 11 may add extra matches to $count$ in some rare cases. For example, this may happen when searching for aaaaa with $k \geq 1$ mismatches in a text ending with aaaa. Line 13 eliminates such extra matches from $count$. Because $found[k]$ is reversed at the implementation level, the vector is shifted to the right in order to hide real matches. Here we assume that it is allowed to access some text positions beyond t_{n-1}. If that is not the case, texts shorter than $\alpha + m - 1$ and the end of a text should be processed with another algorithm.

Table 1 shows an example how vectors $found[k]$ develop while processing a text with a short pattern. For demonstration purposes, α is set to 6.

Algorithm 1 solves the counting version of approximate string matching with k mismatches. It can be transformed into the reporting version by printing positions in line 11.

Tuning up

The pseudocode of Algorithm 1 presents the principles that can be applied to any scenario for $k < m$. Recognizing that the approach is primarily advantageous for small values of k, we developed algorithms for fixed $k = 1, 2, \dots, 5$. By combining these algorithms, we were able to craft a more efficient implementation. First we split the for loop in line 5 of Algorithm 1 into two parts and reduce some unnecessary assignments. The outcome is shown in Algorithm 2.

When k is fixed, we can unroll the three loops in lines 1, 3, and 8 of Algorithm 2 and the initialization loop in line 4 of Algorithm 1. After implementing these changes, the code still includes computations that are not strictly necessary, but the compiler can effectively optimize and eliminate them.

Algorithm 2 Loop (line 5) of Alg. 1 split.

```
1  for j ← 0 to k do
2     cmp ← SIMDcompare(t_{i+j}, vector(p_j), α)
3     for s ← j downto 1 do
4        found[s] ← found[s] & (found[s − 1] | cmp)
5     found[0] ← found[0] & cmp
6  for j ← k + 1 to m − 1 do
7     cmp ← SIMDcompare(t_{i+j}, vector(p_j), α)
8     for s ← k downto 1 do
9        found[s] ← found[s] & (found[s − 1] | cmp)
10    found[0] ← found[0] & cmp
11    if found[k] = 0 then goto out
```

We call the tuned version N32A. In the same way as in Section 2, we made variations N32FA and N32EA for handling pattern positions in the fixed heuristic order and the reverse English frequency order. In N32FA and N32EA, the call of the SIMDcompare function in line 6 of Algorithm 1 is in the form

$$\text{SIMDcompare}(t_{i+\pi(j)}, \text{vector}(\pi(j)), \alpha)$$

where π is a permutation of pattern positions.

Algorithms N32 and N32A use different approaches for constructing comparison vectors. In N32, vectors are constructed for each character of the alphabet, while in N32A, vectors are constructed for each position of the pattern. Thus N32A needs $\alpha \cdot m$ bytes for the vectors. This approach saves space when m is less than $|\Sigma|$.

4. Adaptation to AVX-512

We decided to adapt N32A to leverage AVX-512 extensions, which allow us to compare 64 bytes in parallel. Now we use `_mm_popcnt_u64` for counting matches. Here is the redesigned SIMDcompare function for AVX-512:

```
SIMDcompare(x, y, 64)
   x_ptr = _mm512_loadu_si512(x)
   y_ptr = _mm512_loadu_si512(y)
   return _mm512_cmpeq_epi8_mask(x_ptr, y_ptr)
```

As a result we get algorithms N64A, N64FA, and N64EA. The same adaptation into the AVX-512 platform applies naturally also to N32 for exact matching. Therefore, we also present new experimental results of exact string matching in the next section.

5. Experimental results

We present experimental results in order to compare the behavior of our algorithms against the best known solutions in the literature for approximate and exact string searching.

Setting All the algorithms were coded[1] using the C programming language and compiled with Apple Clang 14.0.0 and run in the testing framework of Hume and Sunday [16]. The processor was i5-1030NG7 with 6 MB cache and 8 GB RAM. The operating system used was MacOS Ventura 13.0.1.

We used three texts: English (the KJV Bible, 12 MB), DNA (the genome of E. Coli, 10 MB), and random binary ($|\Sigma| = 2$, 12 MB) for testing. We chose the length of the text to be at least 1.5 times the cache size (by concatenating the multiples of the text) in order to avoid cache interference with running times [17]. In addition, we made some experiments with texts of 100 MB, but the results were similar to those with the shorter texts. Sets of patterns of lengths 5, 8, 10, 16, and 32 were randomly taken from the texts. Each set contained 200 patterns.

[1] The codes are available at https://users.aalto.fi/tarhio/hamming/.

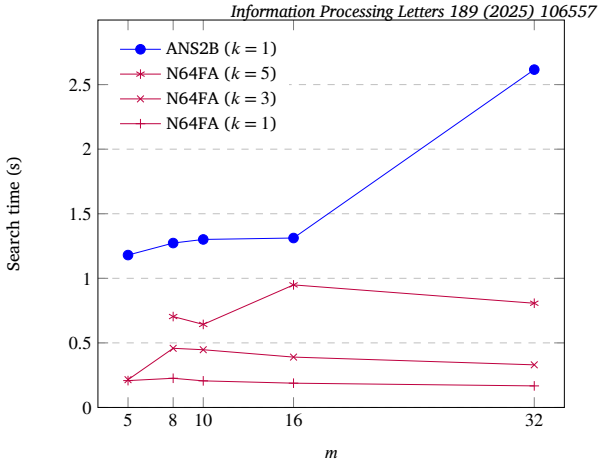

Fig. 1. Search times of ANS2B for $k = 1$ and N64FA for $k = 1$, 3, and 5 in the English dataset.

The tests were made with 99 repeated runs and the execution time of the algorithms for each set is given in seconds. Speedup is reported as a ratio of the running times of the reference algorithm and a new algorithm.

Approximate matching We compared our algorithms (N32A and N64A for the plain order, N32FA and N64FA for the fixed order, N32EA and N64EA for the reverse English frequency order) against ANS2B, BYPSB, and BYPSC [7] for $5 \le m \le 32$. Fiori et al. [7] tested twelve algorithms for the k mismatches problem, and ANS2B, BYPSB, and BYPSC were clearly the best among them. Because all these algorithms apply SIMD, we also present test results of TWSA [18], which is one of the best non-SIMD algorithms for the k mismatches problem.

We carried out the experiments for $k = 1$, 3, and 5. The results for $k = 1$ and $k = 3$ are shown in Table 2 and Table 3, respectively. The best time for each pattern set has been boxed and the peeling factor used for each run has been super-scripted in the tables. From the results, it is clear that the new algorithms outperform earlier algorithms with a wide margin. For the English dataset, the speedup of N64FA over ANS2B is about six for $k = 1$, indicating a significant improvement in performance. Although BYPC is faster than ANS2B for $m = 16$ and 32, N64FA is clearly faster than BYPC. The speedup AVX-512 offers over AVX2 is typically 1.5 or more, i.e. speedup of the variations of N64A over the corresponding versions of N32A.

When k increases, our algorithms become slower. As an example, Fig. 1 shows the search times of ANS2B for $k = 1$ and N64FA for $k = 1$, 3, and 5 in the English dataset. The 64-byte algorithms stay competitive at least until $k = 5$. The times of ANS2B for $k = 3$ and 5 are not shown, because they were very close to the times for $k = 1$.

In most of the cases N64EA and N64FA are almost equally fast for English data as well as N64A and N64FA for binary and DNA data. There is no upper limit for the pattern size our algorithms can handle, but the speed does not change much when the pattern gets longer.

Effect of peeling factor We analyzed the performance of the N64FA algorithm with various peeling factors for $k = 1$ and $5 \le m \le 16$ as shown in Fig. 2. The optimal choice of the peeling factor r depends on the nature of data. The values $r = 4, 5$ produced the best speed for English data, $r = 8$ for DNA, and $r = m$ for binary data. In general, adjusting the r value may lead to significant savings in search times. In the best scenarios, search speed can be more than double. For instance, when searching for DNA patterns of 10 characters, the search times are 0.639 and 0.262 for $r = 2$ and $r = 8$, respectively.

Choice of comparison vectors Algorithm N32 constructs comparison vectors for each character of the alphabet, whereas Algorithm N32A con-

Table 2
Search times in seconds for 200 patterns, $k = 1$.

		$m = 5$	8	10	16	32
English	ANS2B	1.18	1.27	1.30	1.31	2.62
	BYPSC	—	3.14	3.19	0.807	0.43
	TWSA	4.53	3.25	2.77	1.83	0.886
	N32FA	$0.336^{\{5\}}$	$0.295^{\{4\}}$	$0.283^{\{4\}}$	$0.265^{\{4\}}$	$0.222^{\{4\}}$
	N32EA	$0.357^{\{5\}}$	$0.325^{\{4\}}$	$0.305^{\{4\}}$	$0.245^{\{3\}}$	$0.209^{\{3\}}$
	N64FA	$\boxed{0.208}^{\{5\}}$	$0.220^{\{5\}}$	$0.206^{\{5\}}$	$\boxed{0.188}^{\{4\}}$	$0.167^{\{4\}}$
	N64EA	$0.219^{\{5\}}$	$\boxed{0.217}^{\{4\}}$	$\boxed{0.200}^{\{4\}}$	$\boxed{0.188}^{\{4\}}$	$\boxed{0.154}^{\{4\}}$
DNA	ANS2B	1.02	1.09	1.12	1.12	2.15
	BYPSB	—	3.58	3.22	0.81	0.35
	TWSA	5.71	3.72	3.20	1.90	0.909
	N32A	$0.273^{\{5\}}$	$0.448^{\{8\}}$	$0.443^{\{8\}}$	$0.452^{\{8\}}$	$0.428^{\{8\}}$
	N32FA	$0.277^{\{5\}}$	$0.370^{\{7\}}$	$0.373^{\{7\}}$	$0.398^{\{7\}}$	$0.368^{\{7\}}$
	N64A	$\boxed{0.189}^{\{5\}}$	$0.271^{\{8\}}$	$0.271^{\{8\}}$	$\boxed{0.269}^{\{8\}}$	$0.266^{\{8\}}$
	N64FA	$0.205^{\{5\}}$	$\boxed{0.249}^{\{8\}}$	$\boxed{0.262}^{\{8\}}$	$0.277^{\{8\}}$	$\boxed{0.252}^{\{8\}}$
Binary	ANS2B	1.26	1.34	1.35	1.38	2.75
	BYPSB	—	11.91	8.20	5.01	0.699
	TWSA	—	—	—	3.85	1.91
	N32A	$0.323^{\{5\}}$	$0.520^{\{8\}}$	$0.654^{\{10\}}$	$0.965^{\{8\}}$	$0.930^{\{11\}}$
	N32FA	$0.334^{\{5\}}$	$0.539^{\{8\}}$	$0.704^{\{11\}}$	$0.988^{\{13\}}$	$0.915^{\{13\}}$
	N64A	$\boxed{0.213}^{\{5\}}$	$0.304^{\{8\}}$	$\boxed{0.385}^{\{10\}}$	$\boxed{0.536}^{\{15\}}$	$0.549^{\{15\}}$
	N64FA	$0.221^{\{5\}}$	$\boxed{0.299}^{\{8\}}$	$0.392^{\{10\}}$	$0.539^{\{14\}}$	$\boxed{0.530}^{\{14\}}$

Table 3
Search times in seconds for 200 patterns, $k = 3$.

		$m = 5$	8	10	16	32
English	ANS2B	1.25	1.325	1.23	1.27	2.70
	BYPSC	—	—	—	4.24	1.17
	TWSA	6.23	5.18	4.65	2.83	—
	N32FA	$0.322^{\{5\}}$	$0.706^{\{6\}}$	$0.693^{\{6\}}$	$0.589^{\{6\}}$	$0.832^{\{6\}}$
	N32EA	$0.335^{\{5\}}$	$0.771^{\{6\}}$	$0.652^{\{6\}}$	$0.526^{\{6\}}$	$0.427^{\{6\}}$
	N64FA	$0.215^{\{5\}}$	$0.458^{\{8\}}$	$0.447^{\{7\}}$	$0.390^{\{6\}}$	$0.330^{\{6\}}$
	N64EA	$\boxed{0.213}^{\{5\}}$	$\boxed{0.415}^{\{8\}}$	$\boxed{0.445}^{\{7\}}$	$\boxed{0.349}^{\{6\}}$	$\boxed{0.260}^{\{6\}}$
DNA	ANS2B	1.09	1.08	1.22	1.12	2.18
	BYPSB	—	—	—	4.37	1.09
	TWSA	3.87	5.32	4.62	2.92	—
	N32A	$0.245^{\{5\}}$	$0.656^{\{8\}}$	$0.945^{\{10\}}$	$1.06^{\{10\}}$	$1.07^{\{10\}}$
	N32FA	$0.267^{\{5\}}$	$0.663^{\{8\}}$	$1.07^{\{10\}}$	$1.12^{\{10\}}$	$1.04^{\{10\}}$
	N64A	$\boxed{0.170}^{\{5\}}$	$\boxed{0.343}^{\{8\}}$	$\boxed{0.531}^{\{10\}}$	$\boxed{0.615}^{\{10\}}$	$0.615^{\{10\}}$
	N64FA	$0.189^{\{5\}}$	$\boxed{0.343}^{\{8\}}$	$0.542^{\{10\}}$	$0.629^{\{10\}}$	$\boxed{0.600}^{\{10\}}$
Binary	ANS2B	1.38	1.30	1.32	1.38	3.62
	BYPSB	—	—	—	17.19	6.86
	TWSA	4.91	5.16	5.25	5.81	—
	N32A	$0.316^{\{5\}}$	$0.744^{\{8\}}$	$1.16^{\{10\}}$	$2.67^{\{8\}}$	$3.09^{\{8\}}$
	N32FA	$0.321^{\{5\}}$	$0.823^{\{8\}}$	$1.26^{\{10\}}$	$2.93^{\{12\}}$	$3.60^{\{12\}}$
	N64A	$\boxed{0.202}^{\{5\}}$	$\boxed{0.419}^{\{8\}}$	$\boxed{0.581}^{\{10\}}$	$\boxed{1.22}^{\{12\}}$	$\boxed{1.57}^{\{12\}}$
	N64FA	$0.202^{\{5\}}$	$0.429^{\{8\}}$	$0.598^{\{10\}}$	$1.38^{\{12\}}$	$1.70^{\{12\}}$

structs vectors for each position of the pattern. The latter approach offers computational advantage during the search process, potentially resulting in faster running times. To verify this, we tested N64FA and N32FA on our main test processor equipped with AVX-512. Surprisingly, both approaches performed equally well for both algorithms on this processor.

To further investigate the performance on other processors, we tested N32FA with both approaches on three processors without AVX-512 but with AVX2. In this scenario, the latter approach (used originally in N32A) exhibited a speed improvement of approximately 5–10 percent compared to the former approach. The same improvement is naturally applicable to exact string matching with N32.

Exact matching We compared the 64 byte variations of N32 against EPSM [19], EPSMA [20], and the corresponding variations of N32 [2]. EPSM and EPSMA were clearly the best for $m \leq 16$ in the extensive experimental comparison of [20]. Table 4 demonstrates that the N64 variations are clearly faster than the N32 and EPSM variations for $5 \leq m \leq 16$ on English. However, the gain of the AVX-512 technology is smaller than in the case of approximate matching.

6. Conclusions

We have introduced new algorithms for the k mismatches problem. N32A and its variations utilize SIMD instructions based on the AVX2 technology. We adapted N32A into N64A which applies the AVX-512 technology. As a secondary outcome, we have developed N64, which is a refined version of a previous algorithm, tailored for exact string matching using AVX-512.

We have presented an experimental analysis of variations of N32A, N64A, and N64. By comparing them with previous algorithms, we have demonstrated their excellent performance for short patterns and small values of k. Notably, we have observed a substantial speed improvement by executing instructions that process 64 bytes simultaneously.

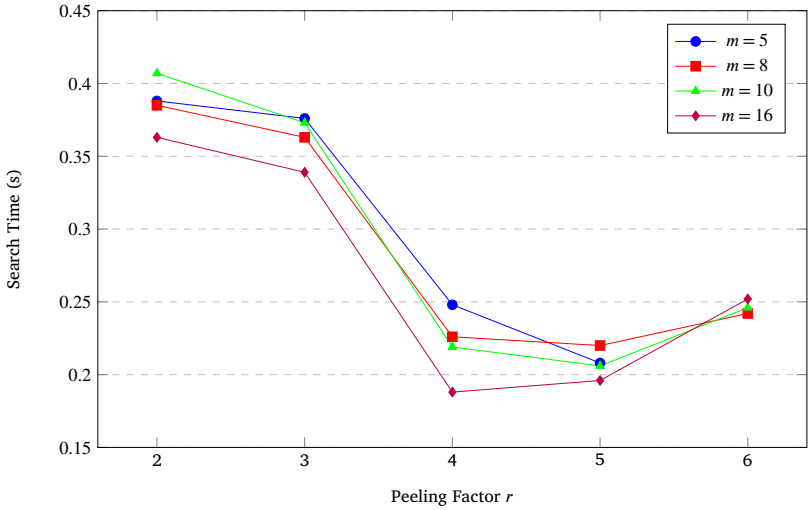

Fig. 2. Search times of N64FA in seconds for 200 patterns for $k = 1$ in the English dataset with various peeling factors.

Table 4
Search times in seconds for 200 patterns, exact matching.

		$m = 5$	8	10	16
English	EPSM	0.429	0.409	0.451	0.386
	EPSMA	0.280	0.306	0.330	0.239
	N32F	$0.180^{\{3\}}$	$0.163^{\{3\}}$	$0.170^{\{3\}}$	$0.164^{\{3\}}$
	N64F	$\boxed{0.164}^{\{4\}}$	$\boxed{0.148}^{\{3\}}$	$\boxed{0.148}^{\{3\}}$	$\boxed{0.141}^{\{3\}}$

CRediT authorship contribution statement

Tamanna Chhabra: Writing – review & editing, Writing – original draft, Software, Formal analysis, Conceptualization. **Sukhpal Singh Ghuman:** Writing – review & editing, Writing – original draft, Visualization, Conceptualization. **Jorma Tarhio:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] S. Faro, T. Lecroq, The exact online string matching problem: a review of the most recent results, ACM Comput. Surv. 45 (2) (2013) 13, https://doi.org/10.1145/2431211.2431212.

[2] J. Tarhio, J. Holub, E. Giaquinta, Technology beats algorithms (in exact string matching), Softw. Pract. Exp. 47 (12) (2017) 1877–1885, https://doi.org/10.1002/spe.2511.

[3] R.A. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Commun. ACM 35 (10) (1992) 74–82, https://doi.org/10.1145/135239.135243.

[4] J. Tarhio, E. Ukkonen, Approximate Boyer-Moore string matching, SIAM J. Comput. 22 (2) (1993) 243–260, https://doi.org/10.1137/0222018.

[5] Z. Liu, X. Chen, J. Borneman, T. Jiang, A fast algorithm for approximate string matching on gene sequences, in: A. Apostolico, M. Crochemore, K. Park (Eds.), Proceedings of Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19–22, 2005, in: Lecture Notes in Computer Science, vol. 3537, Springer, Berlin, 2005, pp. 79–90.

[6] L. Salmela, J. Tarhio, P. Kalsi, Approximate Boyer-Moore string matching for small alphabets, Algorithmica 58 (3) (2010) 591–609, https://doi.org/10.1007/s00453-009-9286-3.

[7] F.J. Fiori, W. Pakalén, J. Tarhio, Approximate string matching with SIMD, Comput. J. 65 (6) (2022) 1472–1488, https://doi.org/10.1093/comjnl/bxaa193.

[8] R.A. Baeza-Yates, C.H. Perleberg, Fast and practical approximate string matching, Inf. Process. Lett. 59 (1) (1996) 21–27, https://doi.org/10.1016/0020-0190(96)00083-X.

[9] G.M. Landau, U. Vishkin, Efficient string matching with k mismatches, Theor. Comput. Sci. 43 (1986) 239–249, https://doi.org/10.1016/0304-3975(86)90178-7.

[10] Z. Galil, R. Giancarlo, Improved string matching with k mismatches, SIGACT News 17 (4) (1986) 52–54, https://doi.org/10.1145/8307.8309.

[11] R. Clifford, A. Fontaine, E. Porat, B. Sach, T. Starikovskaya, The k-mismatch problem revisited, in: R. Krauthgamer (Ed.), Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016, SIAM, 2016, pp. 2039–2052.

[12] K.R. Abrahamson, Generalized string matching, SIAM J. Comput. 16 (6) (1987) 1039–1051, https://doi.org/10.1137/0216067.

[13] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, J. Algorithms 50 (2) (2004) 257–275, https://doi.org/10.1016/S0196-6774(03)00097-X.

[14] K. Fredriksson, S. Grabowski, Exploiting word-level parallelism for fast convolutions and their applications in approximate string matching, Eur. J. Comb. 34 (1) (2013) 38–51, https://doi.org/10.1016/j.ejc.2012.07.013.

[15] Intel, Intel intrinsics guide, https://www.intel.com/content/www/us/en/docs/intrinsics-guide. (Accessed 20 May 2023).

[16] A. Hume, D. Sunday, Fast string searching, Softw. Pract. Exp. 21 (11) (1991) 1221–1248, https://doi.org/10.1002/spe.4380211105.

[17] W. Pakalén, H. Peltola, J. Tarhio, B.W. Watson, Pitfalls of algorithm comparison, in: J. Holub, J. Zdárek (Eds.), Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021, pp. 16–29, http://www.stringology.org/event/2021/p02.html.

[18] B. Durian, T. Chhabra, S.S. Ghuman, T. Hirvola, H. Peltola, J. Tarhio, Improved two-way bit-parallel search, in: J. Holub, J. Zdárek (Eds.), Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1–3, 2014, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 71–83, http://www.stringology.org/event/2014/p08.html.

[19] S. Faro, M.O. Külekci, Fast packed string matching for short patterns, in: Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013, 2013, pp. 113–121.

[20] M.A. Aydogmus, M.O. Külekci, Optimizing packed string matching on AVX2 platform, in: H. Senger, O. Marques, R.E. Garcia, T.P. de Brito, R. Iope, S.L. Stanzani, V. Gil-Costa (Eds.), High Performance Computing for Computational Science — VECPAR 2018 — 13th International Conference, São Pedro, Brazil, September 17–19, 2018, in: Lecture Notes in Computer Science, vol. 11333, Springer, 2018, pp. 45–61, Revised Selected Papers.