Approximate String Searching with AVX2 and AVX-512

Tamanna Chhabra[1], Sukhpal Singh Ghuman[1], and Jorma Tarhio[2]

[1] Faculty of Applied Science and Technology
Sheridan College, Ontario, Canada
firstname.lastname@sheridancollege.ca
[2] Department of Computer Science
Aalto University, Finland
firstname.lastname@aalto.fi

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 experimental results show that our algorithms outperform earlier algorithms for both exact and approximate string matching on various benchmark data sets.

Keywords: Approximate string matching, Hamming distance, exact string matching, SIMD computing, experimental comparison

1 Introduction

String matching [5] 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.

There have been numerous developments in this field in the recent past. Many variations of this problem have appeared, such as exact string matching [5], approximate string matching [10], order preserving matching [2], jumbled pattern matching [7] and many more.

Given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in an 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. For example, if $x = ababb$ and $y = abbab$, then the Hamming distance between x and y is 2.

In our study, we propose algorithms that make use of SIMD (Single Instruction Multiple Data) computing for approximate string matching. 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 [12] that already utilizes SIMD for exact string matching. We extend and modify this algorithm to handle mismatches, thereby allowing approximate string matching.

Our main focus is on demonstrating the practical efficiency of the new algorithms. Our algorithms count the number of occurrences with up to five mismatches. To

Tamanna Chhabra, Sukhpal Singh Ghuman, Jorma Tarhio: Approximate String Searching with AVX2 and AVX-512, pp. 57–67.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6

show their competitiveness, we conduct practical experiments that validate their performance. 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 the reference method.

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.

2 Background

For exact string matching Tarhio et al. [12] presented a naive algorithm (shown as Algorithm 1) which uses the SIMD instruction architecture. The algorithm compares α characters in parallel, where α is 16 or 32. In the following, the names N16 and N32 are used for these variations. N16 uses the SSE2 instruction set and N32 the AVX2 instruction set.

Algorithm 1: SIMD-naive-search
1 construct vector(c) for each $c \in \Sigma$
2 $count \leftarrow 0$; $i \leftarrow 0$
3 while $i \leq n - m$ do
4 $\quad found \leftarrow 2^\alpha - 1$
5 $\quad$ for $j \leftarrow 0$ to $m - 1$ do
6 $\quad\quad found \leftarrow found$ and SIMDcompare(t_{i+j}, vector(p_j), α)
7 $\quad\quad$ if $found = 0$ then goto out
8 $\quad count \leftarrow count$ + popcount($found$)
9 $\quad$ out: $i \leftarrow i + \alpha$

The key idea of Algorithm 1 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 in line 1 for each character of the alphabet. In the case of AVX2, α is 32, and the algorithm first compares the vector of p_0 with $t_0 \ldots t_{31}$, and then it compares the vector of p_1 with $t_1 \ldots t_{32}$ and so on. The bitvector *found* of 32 bits keeps track of active match candidates. The intrinsic function `_mm_popcnt_u32` [9] is used for counting matches in line 8.

The SIMDcompare function for the AVX2 architecture uses three intrinsic functions [9] described below:

- `_mm256_loadu_si256`: The function loads 256 bits of integer data from memory into the destination. The memory address does not need to be aligned on the particular boundary.
- `_mm256_cmpeq_epi8`: The function compares a 32 8-bit integer elements in two 256-bit vectors and sets the corresponding bit in the output vector to 1 if the two elements are equal, and to 0 otherwise. The result is a 256-bit vector where each bit represents the result of a single comparison operation.
- `_mm256_movemask_epi8`: The function creates a 256-bit vector of 32 8-bit integer elements and returns a 32-bit integer value where the ith bit is set to 1 if the ith element of the vector has its most significant bit set, and to 0 otherwise. This

function creates a mask from the most significant bit of the comparison result as a 32-bit integer.

With these intrinsic functions, SIMDcompare for AVX2 is implemented as follows:

```
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))
```

For N16, the corresponding intrinsic functions for loading, comparing, and creating a mask for comparison are used in the SSE2 instruction set architecture.

Tarhio et al. [12] used three different orders for comparing the characters of the pattern: plain order, fixed order, and reverse English frequency order. In the case of the 32 byte version, we use the following names for these variations: N32, N32F, and N32E. The plain order advances from left to right in the pattern. The fixed order applies the following heuristic order: $p_0, p_{m-1}, p_3, p_6, \ldots, p_2, p_5, \ldots, p_1, p_4, \ldots$ excluding space characters which are compared last. The reverse frequency order could be applied to any type of data, but only English has been used in experiments. Algorithm 1 uses the plain order. In the case of other orders, the call of the SIMDcompare function in line 6 is in the form

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

where π is a permutation of pattern positions. The algorithm uses $\alpha \cdot |\Sigma|$ bytes extra space for the vectors.

In addition to the SIMD instructions, loop peeling has a key role in the efficiency of Algorithm 1. 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, we call the number of the moved iterations the peeling factor r. Tarhio et al. [12] used $r = 2$ or 3 for English and $r = 5$ for DNA.

3 Algorithms for approximate matching

Our aim is to develop algorithms for approximate string matching. Algorithm 2 (as shown below) is used to count all the occurrences of a given pattern string P in a text string T, with at most k mismatches. To perform the comparisons efficiently, the algorithm uses SIMD (Single Instruction Multiple Data) instructions, which can compare multiple characters in parallel. The algorithm is a variation of N32 extended with mismatch counting. It works by handling α consecutive starting positions of an occurrence candidate of P in parallel. In the case of AVX2, α is 32. Bitvectors $found[0], \ldots, found[k]$ of α bits are used to keep track of mismatches. Initially, every bit of each $found[i]$ is set. 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 all the bits of $found[k]$ become zero, then none of the α candidates can be an occurrence of the pattern. The comparison vectors for each position of the pattern are computed before search in line 1. Each comparison vector contains α copies of the corresponding character. The popcount function counts the number of set bits in a vector.

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

Algorithm 2: 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          c ← SIMDcompare(t_{i+j}, vector(j), α)
7          for s ← k downto 1 do
8              found[s] ← found[s] and (found[s − 1] or c)
9          found[0] ← found[0] and c
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))
```

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.

Algorithm 1 and Algorithm 2 use different approaches for constructing comparison vectors. In Algorithm 1, vectors are constructed for each character of the alphabet, while in Algorithm 2, vectors are constructed for each position of the pattern. Thus Algorithm 2 needs $\alpha \cdot m$ bytes extra space for the vectors. This approach saves space when m is less than $|\Sigma|$.

Algorithm 2 uses the plain order. If other orders are used (see Section 2), the call of the SIMDcompare function in line 6 is in the form

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

where π is a permutation of pattern positions. Algorithm 2 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.

Proof of the counting method

Without losing generality, we can assume that positions of candidates are processed in order from left to right. Let $f_{i,k}$ be the bit of *found*$[k]$ corresponding a candidate starting from t_j after $p_0 \cdots p_i$ has been processed. According to the construction, $f_{i,k} \geq f_{i,k-1}$ holds.

Proposition: $f_{i,k} = 1$ holds if $t_j \cdots t_{j+i}$ contains at most k character mismatches with $p_0 \cdots p_i$, and otherwise $f_{i,k} = 0$ holds.

Proof by induction: If t_j is p_0, then we have $f_{0,0} = f_{0,1} = \cdots = f_{0,k} = 1$. If t_j is not p_0, then we have $f_{0,0} = 0$ and $f_{0,1} = \cdots = f_{0,k} = 1$.

Let us assume that the proposition holds for $f_{i-1,k-1}$. If t_{j+i} is p_i, then we have $f_{i,k} = f_{i-1,k-1}$ and the proposition holds. If t_{j+i} is not p_i, we have two cases. If $f_{i-1,k-1} = 0$ holds, then we have $f_{i,k} = f_{i-1,k-1}$ and the proposition holds. If $f_{i-1,k-1} = 1$ holds, then $f_{i-1,k} = 1$ holds. So $f_{i,k} = 1$ is satisfied. Because $t_j \cdots t_{j+i-1}$ contains at most $k - 1$ mismatches according to the induction assumption, $t_j \cdots t_{j+i}$ contains at most k mismatches.

Let us consider an example. The bolded entry in Table 1 shows the value of $f_{i,3}$ after processing aabbab in the text for P = aaaaaa. Here vectors are shown in the order of the text.

Table 1. An example of computation of $f_{i,3}$. P = aaaaaa, $k = 3$.

	a	a	b	b	a	b
$f_{i,3}$	1	1	1	1	1	**1**
$f_{i,2}$	1	1	1	1	1	0
$f_{i,1}$	1	1	1	0	0	0
$f_{i,0}$	1	1	0	0	0	0

Tuning up

The pseudocode of Algorithm 2 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 $k = 1, 2, \ldots, 5$. By combining these algorithms, we were able to craft a more efficient implementation. First we split the for loop in line 5 of Algorithm 2 into two parts and reduce some unnecessary assignments. The outcome is shown in Algorithm 3.

Algorithm 3 Loop (line 5) of Alg. 2 split.

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

When k is fixed, we can unroll the three loops in lines 1, 3, and 8 of Algorithm 3 and the initialization loop in line 4 of Algorithm 2. After these changes, the code still contains computations that are not essential, but the compiler can fairly efficiently eliminate them.

We call the tuned version N32A. Exactly in the same way as in the case of exact string matching explained in Section 2, we get the variations N32FA and N32EA for handling pattern positions in the fixed heuristic order and the reverse English frequency order.

4 Adaptation to AVX-512

We decided to adapt Algorithm 2 to leverage AVX-512 extensions, which allow us to compare 64 bytes in parallel. The set of AVX-512 intrinsic functions does not contain a 512-bit counterpart for the `_mm256_cmpeq_epi8` intrinsic function which was applied in SIMDcompare for AVX2. Therefore we selected another intrinsic function computing

the mask as well, eliminating the need for an extra intrinsic function. 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)
```

In addition, we use `_mm_popcnt_u64` for counting matches. We describe how these 512-bit intrinsic functions [9] work. The function `_mm512_loadu_si512` works correspondingly to `_mm512_loadu_si512`, which was explained in Section 2.

The intrinsic function `_mm512_cmpeq_epi8_mask`[1] performs an element-wise comparison of two 512-bit registers containing 64 8-bit integer elements each. It returns a 64-bit mask, where each bit represents the result of the comparison of the corresponding 8-bit integer element in the input registers. If the two elements are equal, the corresponding bit in the mask is set to 1, otherwise it is set to 0.

SIMDcompare for AVX-512 is lighter than SIMDcompare for SSE2 or AVX2, because SIMDcompare for AVX-512 has one intrinsic function less than the others.

The same adaptation into the AVX-512 platform applies naturally also for Algorithm 1 for exact matching. Therefore, we present experimental results of exact string matching in the next section in addition to the results of approximate string matching.

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.

5.1 Setting

All the algorithms were implemented[2] using the C programming language and compiled with Apple Clang 14.0.0 and run in the testing framework of Hume and Sunday [8]. The processor used was Intel Core 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 [11]. Sets of patterns of lengths 5, 8, 10, 16, and 32 were randomly taken from the texts. Each set contains 200 patterns. The tests were made with 99 repeated runs. Speedup is reported as a ratio of the running times of the reference algorithm and a new algorithm.

5.2 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)

[1] Note that `_mm256_cmpeq_epi8_mask` is not available in AVX2 but only in AVX-512. Therefore it was not used in Algorithm 1.

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

against ANS2B, BYPSB, and BYPSC [6] for $5 \leq m \leq 32$. Fiori et al. [6] 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 [3], 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 as shown in Tables 2, 3, and 4. The best time for each pattern set has been boxed and the used peeling factor 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. 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, Figure 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 $k = 5$.

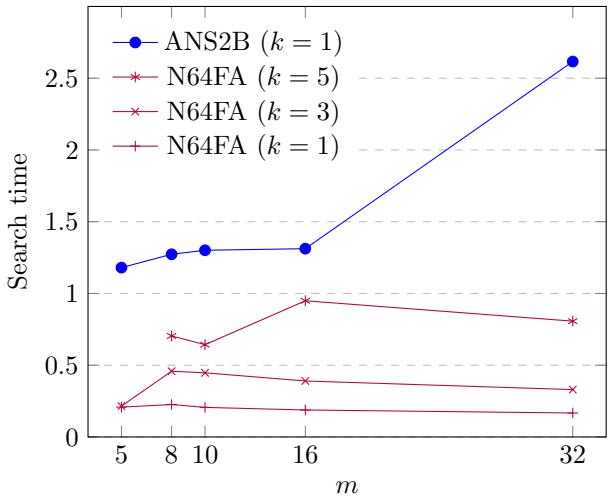

Figure 1. Search times of ANS2B for $k = 1$ and N64FA for $k = 1$, 3, and 5 in the English dataset.

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.

One noteworthy finding is that the advantage of N64A over N32A increases when patterns do not appear in the text. This observation is useful for scenarios checking for pattern absence.

5.3 Effect of peeling factor

We analyzed the performance of the N64FA algorithm with various peeling factors for $k = 1$. The results are presented in Table 5. The optimal choice of the peeling factor r depends on the nature of the dataset. For English data, lower r values produce better speed, while for DNA and binary data, higher r values yield improved performance.

Table 2. Approximate search times, $k = 1$.

		$m = 5$	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	$\boxed{0.208}^{\{5\}}$	$0.226^{\{4\}}$	$0.206^{\{5\}}$	$\boxed{0.188}^{\{4\}}$	$0.167^{\{4\}}$
	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. Approximate search times, $k = 3$.

		$m = 5$	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	$0.202^{\{5\}}$	$0.429^{\{8\}}$	$0.598^{\{10\}}$	$1.38^{\{12\}}$	$1.70^{\{12\}}$

In general, adjusting the r value may lead to significant savings in search times—by doubling the search speed in many cases.

Table 4. Approximate search times, $k = 5$.

		$m = 8$	10	16	32
English	ANS2B	1.35	1.37	1.29	2.72
	BYPSC	—	—	—	3.86
	TWSA	6.07	5.36	3.50	—
	N32FA	$1.14^{\{8\}}$	$1.44^{\{10\}}$	$1.15^{\{9\}}$	$0.955^{\{9\}}$
	N32EA	$1.02^{\{8\}}$	$1.40^{\{10\}}$	$0.983^{\{9\}}$	$0.983^{\{9\}}$
	N64FA	$0.419^{\{8\}}$	$0.686^{\{10\}}$	$0.949^{\{7\}}$	$0.807^{\{7\}}$
	N64EA	$0.381^{\{8\}}$	$0.653^{\{10\}}$	$0.833^{\{7\}}$	$0.664^{\{7\}}$
DNA	ANS2B	1.33	1.10	1.13	2.23
	BYPSB	—	—	—	4.144
	TWSA	3.85	4.67	3.60	—
	N32A	$0.587^{\{8\}}$	$1.74^{\{10\}}$	$2.52^{\{8\}}$	$2.57^{\{8\}}$
	N32FA	$0.611^{\{8\}}$	$1.12^{\{10\}}$	$2.45^{\{8\}}$	$2.70^{\{8\}}$
	N64A	$0.301^{\{8\}}$	$0.493^{\{10\}}$	$1.28^{\{9\}}$	$1.28^{\{9\}}$
	N64FA	$0.332^{\{8\}}$	$0.548^{\{10\}}$	$1.28^{\{9\}}$	$1.37^{\{9\}}$
Binary	ANS2B	1.27	1.21	1.26	6.62
	BYPSB	—	—	—	18.78
	TWSA	4.74	4.84	5.31	—
	N32A	$0.747^{\{8\}}$	$1.30^{\{10\}}$	$2.93^{\{4\}}$	$4.66^{\{3\}}$
	N32FA	$0.704^{\{8\}}$	$1.26^{\{10\}}$	$3.07^{\{4\}}$	$5.07^{\{3\}}$
	N64A	$0.375^{\{8\}}$	$0.649^{\{10\}}$	$1.57^{\{7\}}$	$2.71^{\{7\}}$
	N64FA	$0.379^{\{8\}}$	$0.651^{\{10\}}$	$1.62^{\{7\}}$	$2.95^{\{7\}}$

Table 5. Search times of N64FA with varied peeling factor, $k = 1$.

	r	$m = 5$	8	10	16		r	$m = 5$	8	10	16
English	2	0.388	0.385	0.407	0.363	Binary	4	0.233	0.422	0.760	1.10
	3	0.376	0.363	0.373	0.339		5	0.208	0.412	0.732	1.05
	4	0.248	0.226	0.219	0.188		6	—	0.400	0.704	1.054
	5	0.208	0.220	0.206	0.196		7	—	0.376	0.637	0.995
	6	—	0.242	0.246	0.252		8	—	0.319	0.647	0.997
DNA	2	0.371	0.624	0.639	0.648		9	—	—	0.625	0.988
	3	0.376	0.684	0.703	0.738		10	—	—	0.384	1.03
	4	0.346	0.639	0.653	0.663		11	—	—	—	0.785
	5	0.205	0.539	0.559	0.559		12	—	—	—	0.657
	6	—	0.379	0.384	0.385		13	—	—	—	0.594
	7	—	0.280	0.278	0.283		14	—	—	—	0.539
	8	—	0.249	0.262	0.277		15	—	—	—	0.543
	9	—	—	0.279	0.290		16	—	—	—	0.581

5.4 Choice of comparison vectors

Algorithm 1 constructs comparison vectors for each character of the alphabet, whereas Algorithm 2 constructs 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 different processors, we tested N32FA with both approaches on three other processors without AVX-512 but with AVX2. In

this scenario, the latter approach (used in Algorithm 2) exhibited a speed improvement of approximately 5–10 percent compared to the former approach.

5.5 Exact matching

We compared the 64 byte variations of Algorithm 1 (N64 for the plain order, N64F for the fixed order, and N64E for the reverse English frequency order) against EPSM [4], EPSMA [1], and the corresponding variations of N32 [12]. EPSM and EPSMA were clearly the best for $m \leq 16$ in the extensive experimental comparison of [1].

Table 6 demonstrates that the N64 variations are clearly faster than the N32 and EPSM variations for $5 \leq m \leq 16$. The speedup becomes particularly noticeable in the case of binary data. However, the gain of the AVX-512 technology is smaller than in the case of approximate matching. For example, the speedup of N64F over N32F is 1.17 but the speedup of N64FA over N32FA is 1.41 for $k = 1$ both in the case of English data for $m = 16$.

Table 6. Exact search times.

		$m = 5$	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	$0.164^{\{4\}}$	$0.148^{\{3\}}$	$\boxed{0.148}^{\{3\}}$	$\boxed{0.141}^{\{3\}}$
	N64E	$\boxed{0.154}^{\{4\}}$	$\boxed{0.145}^{\{4\}}$	$0.150^{\{4\}}$	$0.143^{\{4\}}$
DNA	EPSM	0.469	0.476	0.495	0.314
	EPSMA	0.234	0.456	0.463	0.265
	N32	$0.168^{\{5\}}$	$0.205^{\{5\}}$	$0.205^{\{5\}}$	$0.212^{\{5\}}$
	N32F	$0.168^{\{5\}}$	$0.196^{\{5\}}$	$0.203^{\{5\}}$	$0.204^{\{5\}}$
	N64	$\boxed{0.152}^{\{5\}}$	$\boxed{0.165}^{\{6\}}$	$\boxed{0.165}^{\{6\}}$	$\boxed{0.159}^{\{6\}}$
	N64F	$0.153^{\{5\}}$	$0.167^{\{6\}}$	$0.169^{\{6\}}$	$0.173^{\{6\}}$
Binary	EPSM	4.92	4.85	5.15	0.550
	EPSMA	0.280	4.87	4.91	1.35
	N32	$0.177^{\{5\}}$	$0.237^{\{8\}}$	$0.323^{\{10\}}$	$0.344^{\{11\}}$
	N32F	$0.191^{\{5\}}$	$0.277^{\{8\}}$	$0.327^{\{10\}}$	$0.460^{\{11\}}$
	N64	$0.162^{\{5\}}$	$\boxed{0.198}^{\{8\}}$	$\boxed{0.230}^{\{10\}}$	$\boxed{0.259}^{\{16\}}$
	N64F	$\boxed{0.160}^{\{5\}}$	$0.202^{\{8\}}$	$0.236^{\{10\}}$	$0.282^{\{11\}}$

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 side result, we got N64, an adaptation of an earlier algorithm, for exact string matching with AVX-512.

We have presented an experimental analysis of variations of N32A, N64A, and N64. Through comparisons with earlier algorithms, we have demonstrated their excellent performance for short patterns. Notably, we have observed a substantial speed improvement by executing instructions that process 64 bytes simultaneously. Additionally, our experiment underscores the critical role of loop peeling in enhancing the performance of these new algorithms.

References

1. M. A. Aydogmus and M. O. Külekci: *Optimizing packed string matching on AVX2 platform*, in High Performance Computing for Computational Science — VECPAR 2018 — 13th International Conference, São Pedro, Brazil, September 17-19, 2018, Revised Selected Papers, H. Senger, O. Marques, R. E. Garcia, T. P. de Brito, R. Iope, S. L. Stanzani, and V. Gil-Costa, eds., vol. 11333 of Lecture Notes in Computer Science, Springer, 2018, pp. 45–61.
2. T. Chhabra, S. Faro, M. O. Külekci, and J. Tarhio: *Engineering order-preserving pattern matching with SIMD parallelism.* Software: Practice and Experience, 47(5) 2017, pp. 731–739.
3. B. Durian, T. Chhabra, S. S. Ghuman, T. Hirvola, H. Peltola, and J. Tarhio: *Improved two-way bit-parallel search*, in Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 71–83.
4. S. Faro and 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.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
6. F. J. Fiori, W. Pakalén, and J. Tarhio: *Approximate string matching with SIMD.* Comput. J., 65(6) 2022, pp. 1472–1488.
7. S. S. Ghuman, J. Tarhio, and T. Chhabra: *Improved online algorithms for jumbled matching.* Discret. Appl. Math., 274 2020, pp. 54–66.
8. A. Hume and D. Sunday: *Fast string searching.* Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
9. Intel: *Intel intrinsics guide*, https://www.intel.com/content/www/us/en/docs/intrinsics-guide, Accessed: 2023-05-20.
10. G. Navarro: *A guided tour to approximate string matching.* ACM Comput. Surv., 33(1) 2001, pp. 31–88.
11. W. Pakalén, H. Peltola, J. Tarhio, and B. W. Watson: *Pitfalls of algorithm comparison*, in Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021, pp. 16–29.
12. J. Tarhio, J. Holub, and E. Giaquinta: *Technology beats algorithms (in exact string matching).* Software: Practice and Experience, 47(12) 2017, pp. 1877–1885.