Optimal Packed String Matching *

Oren Ben-Kiki[1], Philip Bille[2], Dany Breslauer[3], Leszek Gąsieniec[4], Roberto Grossi[5], and Oren Weimann[6]

1 **Intel Research and Development Center, Haifa, Israel**
2 **Technical University of Denmark, Copenhagen, Denmark**
3 **Caesarea Rothchild Institute, University of Haifa, Haifa, Israel**
4 **University of Liverpool, Liverpool, United Kindom**
5 **Dipartimento di Informatica, Università di Pisa, Pisa, Italy**
6 **University of Haifa, Haifa, Israel**

Abstract

In the packed string matching problem, each machine word accommodates α characters, thus an n-character text occupies n/α memory words. We extend the Crochemore-Perrin constant-space $O(n)$-time string matching algorithm to run in optimal $O(n/\alpha)$ time and even in real-time, achieving a factor α speedup over traditional algorithms that examine each character individually. Our solution can be efficiently implemented, unlike prior theoretical packed string matching work. We adapt the standard RAM model and only use its AC^0 instructions (i.e., no multiplication) plus two specialized AC^0 packed string instructions. The main *string-matching* instruction is available in commodity processors (i.e., Intel's SSE4.2 and AVX Advanced String Operations); the other *maximal-suffix* instruction is only required during pattern preprocessing. In the absence of these two specialized instructions, we propose theoretically-efficient emulation using integer multiplication (not AC^0) and table lookup.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity. F.2.2 Non-numerical Algorithms and Problems—Pattern Matching.

Keywords and phrases String matching. Bit parallelism. Real time. Space efficiency.

1 Introduction

Hundreds of articles have been published about string matching, exploring the multitude of theoretical and practical facets of this fundamental problem. For an n-character text T and an m-character pattern x, the classical algorithm by Knuth, Morris and Pratt [21] takes $O(n+m)$ time and uses $O(m)$ auxiliary space to find all pattern occurrences in the text, namely, all text positions i, such that $T[i..i+m-1] = x$. Many other algorithms have been published; some are faster on the average, use only constant auxiliary space, operate in real-time, or have other interesting benefits. In an extensive study, Faro and Lecroq [12] offer an experimental comparative evaluation of some 85 string matching algorithms.

Packed strings. In modern computers, the size of a machine word is typically larger than the size of an alphabet character and the machine level instructions operate on whole words, i.e., 64-bit or longer words vs. 8-bit ASCII, 16-bit UCS, 2-bits biological DNA, 5-bits amino acid alphabets, etc. The packed string representation fits multiple characters into one larger word, so that the characters can be compared in bulk rather than individually: if the

* Partially supported by the European Research Council (ERC) project SFEROT, by the Israeli Science Foundation grant 347/09 and by Italian project PRIN AlgoDEEP (2008TFBWL4) of MIUR.

characters of a string are drawn from an alphabet Σ, then a word of $\omega \geq \log_2 n$ bits fits up to α characters, where the packing factor is $\alpha = \frac{\omega}{\log_2 |\Sigma|} \geq \log_{|\Sigma|} n$.[1]

Using the packed string representation in the string matching problem is not a new idea and goes back to early string matching papers by Knuth, Morris and Pratt [21, §4] and Boyer and Moore [6, §8-9], to times when hardware character byte addressing was new and often less efficient than word addressing. Since then, several practical solutions that take advantage of the packed representation have been proposed in the literature [2, 4, 11, 15, 16, 25]. However, none of these algorithms improves over the worst-case $O(n)$ time bounds of the traditional algorithms. On the other hand, any string matching algorithm should take at least $\Omega(n/\alpha)$ time to read a packed text in the worst case, so there remains a gap to fill.

Existing work. A significant theoretical step recently taken introduces a few solutions based on either tabulation (a.k.a. "the Four-Russian technique") or word-level parallelism (a.k.a. "bit-parallelism"). Fredriksson [15, 16] used tabulation and obtained an algorithm that uses $O(n^\varepsilon m)$ space and $O(\frac{n}{\log_{|\Sigma|} n} + n^\varepsilon m + occ)$ time, where occ denotes the number of pattern occurrences and $\varepsilon > 0$ denotes an arbitrary small constant. Bille [5] improved these bounds to $O(n^\varepsilon + m)$ space and $O(\frac{n}{\log_{|\Sigma|} n} + m + occ)$ time. Very recently, Belazzougui [3] showed how to use word-level parallelism to obtain $O(m)$ space and $O(\frac{n}{m} + \frac{n}{\alpha} + m + occ)$ time. Belazzougui's algorithm uses a number of succinct data structures as well as hashing: for $\alpha \leq m \leq n/\alpha$, his time bound is optimal while space occupancy is not. As admitted by the above authors, none of these results is practical. A summary of the known bounds and our new result is given in Table 1, where our result uses two instructions described later on.

Time	Space	Reference
$O(\frac{n}{\log_{\|\Sigma\|} n} + n^\varepsilon m + occ)$	$O(n^\varepsilon m)$	Fredriksson [15, 16]
$O(\frac{n}{\log_{\|\Sigma\|} n} + m + occ)$	$O(n^\varepsilon + m)$	Bille [5]
$O(\frac{n}{\alpha} + \frac{n}{m} + m + occ)$	$O(m)$	Belazzougui [3]
$O(\frac{n}{\alpha} + \frac{m}{\alpha} + occ)$	$O(1)$	This paper

Table 1 Comparison of packed string matching algorithms.

Our results. We propose an $O(n/\alpha + m/\alpha)$ time string matching algorithm (where the term m/α is kept for comparison with the other results) that is derived from the elegant Crochemore-Perrin [9] algorithm. The latter takes linear time, uses only constant auxiliary space, and can be implemented in real-time following the recent work by Breslauer, Grossi and Mignosi [7] – benefits that are also enjoyed in our settings. The algorithm has an attractive property that it compares the text characters only moving forward on two wavefronts without ever having to back up, relying on the celebrated Critical Factorization Theorem [8, 22].

We use a *specialized word-size packed string matching instruction* to anchor the pattern in the text and continue with bulk character comparisons that match the remainder of the pattern. Our reliance on a specialized packed string matching instruction is not far fetched, given the recent availability of such instructions in commodity processors, which has been a catalyst for our work. Our algorithm is easily adaptable to situations where the packed string matching instruction and the bulk character comparison instruction operate on different word sizes. The output occurrences are compactly provided in a bit-mask that can be spelled out as an extensive list of text positions in extra $O(occ)$ time.

[1] Assume that $|\Sigma|$ is a power of two, ω is divisible by $\log_2 |\Sigma|$, and the packing factor α is a whole integer.

Unlike the prior theoretical work, our solution has a cache-friendly sequential memory access without using large external tables or succinct data structures, and therefore, can also be efficiently implemented. The same specialized packed string matching instruction could also be used in other string matching algorithms, e.g. the Knuth-Morris-Pratt algorithm [19, §10.3.3], but our algorithm also works in real-time and uses only constant auxiliary space.

Model of computation. We adapt the standard word-RAM model with ω-bit words and with only AC^0 instructions (i.e., arithmetic, bitwise and shift operations but no multiplication) plus two other specialized AC^0 instructions. The main word-size packed string matching instruction is available in the recent *Advanced String Operations* in *Intel's Streaming SIMD Extension (SSE4.2) and Advanced Vector Extension (AVX) Efficient Accelerated String and Text Processing* instruction set [18, 20]. The other instruction, which is only used in the pattern preprocessing, finds the lexicographically maximum suffix. Specifically, adopting the notation $[d] = \{0, 1, \ldots, d-1\}$, the two instructions are the following ones:

Word-Size String Matching (`wssm`): find occurrences of one short pattern x that fits in one word (up to α characters) in a text y that fits in two words (up to $2\alpha-1$ characters). The output is a binary word Z of $2\alpha-1$ bits such that its ith bit $Z[i] = 1$ iff $y[i..i+|x|-1] = x$, for $i \in [2\alpha-1]$. When $i + |x| - 1 \geq \alpha$, this means that only a prefix of x is matched.

Word-Size Lexicographically Maximum Suffix (`wslm`): given a packed string x that fits in one word (up to α characters), return position $i \in [\alpha]$ such that $x[i..\alpha-1]$ is lexicographically maximum among the suffixes in $\{x[j..\alpha-1] \mid j \in [\alpha]\}$.

If these instructions are not available, then we can emulate them, but our proposed emulations cause a small slowdown of $\log\log\omega$ as shown in Table 2.

Time	Space	Reference
$O\left(\omega + \frac{n \log\log\omega}{\alpha} + \frac{m}{\alpha} + occ\right)$	$O(1)$	This paper

Table 2 Bounds in the word-RAM when the ω-bit `wssm` and `wslm` instructions are not available.

2 Packed String Matching

In this section we describe how to solve the *packed string matching* problem using the two specialized word-size string matching instructions `wssm` and `wslm`, and standard word-RAM bulk comparisons of packed strings.

▶ **Theorem 1.** *Packed string matching for a length m pattern and a length n text can be solved in $O(\frac{m}{\alpha} + \frac{n}{\alpha})$ time in the word-RAM extended with constant-time `wssm` and `wslm` instructions. Listing explicitly the occ text positions of the pattern occurrences takes an additional $O(occ)$ time. The algorithm can be made real-time, and uses just $O(1)$ auxiliary words of memory besides the read-only $\frac{m}{\alpha} + \frac{n}{\alpha}$ words that store the input.*

The algorithm behind Theorem 1 follows the classical scheme, in which a text scanning phase is run after the pattern preprocessing. In the following, we first present the necessary background and then describe how to perform the text scanning phase using `wssm`, and the pattern preprocessing using `wslm`.

2.1 Background

Critical Factorization. Properties of periodic strings are often used in efficient string algorithms. A string u is *a period* of a string x if x is a prefix of u^k for some integer k, or equivalently if x is a prefix of ux. The shortest period of x is called *the period* of x and its length is denoted by $\pi(x)$. A *substring* or a *factor* of a string x is a contiguous block of symbols u, such that $x = x'ux''$ for two strings x' and x''. A *factorization* of x is a way to break x into a number of factors. We consider factorizations of a string $x = uv$ into two factors: a *prefix* u and a *suffix* v. Such a factorization can be represented by a single integer and is *non-trivial* if neither of the two factors is equal to the empty string.

Given a factorization $x = uv$, a *local period* of the factorization is defined as a non-empty string p that is consistent with both sides u and v. Namely, (i) p is a suffix of u or u is a suffix of p, and (ii) p is a prefix of v or v is a prefix of p. The shortest local period of a factorization is called *the local period* and its length is denoted by $\mu(u, v)$. A non-trivial factorization $x = uv$ is called a *critical factorization* if the local period of the factorization is of the same length as the period of x, i.e., $\mu(u, v) = \pi(uv)$. See Figure 1.

```
  a | b a a a b a        a b | a a a b a        a b a | a a b a
b a   b a              a a a b   a a a b            a   a
      (a)                    (b)                    (c)
```

Figure 1 The local periods at the first three non-trivial factorizations of the string abaaaba. In some cases the local period overflows on either side; this happens when the local period is longer than either of the two factors. The factorization (b) is a critical factorization with local period aaab of the same length as the global period abaa.

Crochemore-Perrin algorithm. Although critical factorizations may look tricky, they allow for a simplification of the text processing phase of string matching algorithms. We assume that the reader is familiar with the Crochemore-Perrin algorithm [9] and its real-time variation Breslauer-Grossi-Mignosi [7]. Observe that Crochemore and Perrin use Theorem 2 to break up the pattern as $x = uv$ for non-empty prefix u and suffix v, such that $|u| \leq \pi(x)$.

▶ **Theorem 2.** *(Critical Factorization Theorem, Cesari and Vincent [8, 22]) Given any* $|\pi(x)| - 1$ *consecutive non-trivial factorizations of a string* x*, at least one is critical.*

Then, they exploit the critical factorization of $x = uv$ by matching the longest prefix z of v against the current text symbols, and using Theorem 3 whenever a mismatch is found.

▶ **Theorem 3.** *(Crochemore and Perrin [9]) Let* $x = uv$ *be a critical factorization of the pattern and let* p *be any local period at this factorization, such that* $|p| \leq \max(|u|, |v|)$*. Then* $|p|$ *is a multiple of* $\pi(x)$*, the period length of the pattern.*

Precisely, if $z = v$, they show how to declare an occurrence of x. Otherwise, the symbol following z in v is mismatching when compared to the corresponding text symbol, and the pattern x can be *safely* shifted by $|z| + 1$ positions to the right (there are other issues for which we refer the reader to [9]).

To simplify the matter in the rest of the paper, we discuss how to match the pattern suffix v assuming without loss of generality that $|u| \leq |v|$. Indeed, if $|u| > |v|$, the Crochemore-Perrin approach can be simplified as shown in [7]: use two critical factorizations, $x = uv$ and $x' = u'v'$, for a prefix x' of x such that $|x'| > |u|$ and $|u'| \leq |v'|$. In this way, matching both u' and v' suitably displaced by $|x| - |x'|$ positions from matching v, guarantees that x

occurs. This fact enables us to focus on matching v and v', since the cost of matching u' is always dominated by the cost of matching v', and we do not need to match u. For the sake of discussion, it suffices to consider only one instance, namely, suffix v.

We now give more details on the text processing phase, assuming that the pattern preprocessing phase has correctly found the critical factorization of the pattern x and its period $\pi(x)$, and any additional pattern preprocessing that may be required (Section 2.3).

While other algorithms may be used with the `wssm` instruction, the Crochemore-Perrin algorithm is particularly attractive because of its simple text processing. Therefore, it is convenient to assume that the period length and critical factorization are exactly computed in the pattern preprocessing burying the less elegant parts in that phase.

2.2 Text processing

The text processing has complementary parts that handle short patterns and long patterns. A pattern x is *short* if its length is at most α, namely, the packed pattern fits into a single word, and is *long* otherwise. Processing short patterns is immediate with `wssm` and, as we shall see, the search for long patterns reduces to that for short patterns.

Short patterns. When the pattern is already short, `wssm` is repeatedly used to directly find all occurrences of the pattern in the text.

▶ **Lemma 4.** *There exists an algorithm that finds all occurrences of a short pattern of length $m \leq \alpha$ in a text of length n in $O(\frac{n}{\alpha})$ time using $O(1)$ auxiliary space.*

Proof. Consider the packed text blocks of length $\alpha + m - 1$ that start on word boundaries, where each block overlaps the last $m - 1$ characters of the previous block and the last block might be shorter. Each occurrence of the pattern in the text is contained in exactly one such block. Repeatedly use the `wssm` instruction to search for the pattern of length $m \leq \alpha$ in these text blocks whose length is at most $\alpha + m - 1 \leq 2\alpha - 1$. ◀ ◀

Long patterns. Let x be a long pattern of length $m > \alpha$: occurrences of the pattern in the text must always be spaced at least the period $\pi(x)$ locations apart. We first consider the easier case where the pattern has a long period, namely $m \geq \pi(x) > \alpha$, and so there is at most one occurrence starting within each word.

▶ **Lemma 5.** *There exists an algorithm that finds all occurrences of a long-period long pattern of length $m \geq \pi(x) \geq \alpha$, in a text of length n in $O(\frac{n}{\alpha})$ time using $O(1)$ auxiliary space.*

Proof. The Crochemore-Perrin algorithm can be naturally implemented using the `wssm` instruction and bulk character comparisons. Given the critical factorization $x = uv$, the algorithm repeatedly searches using `wssm` for an occurrence of a prefix of v of length $\min(|v|, \alpha)$ starting in each packed word aligned with v, until such an occurrence is discovered. If more than one occurrence is found starting within the same word, then by Lemma 3, only the first such occurrence is of interest. The algorithm then uses the occurrence of the prefix of v to anchor the pattern within the text and continues to compare the rest of v with the aligned text and then compares the pattern prefix u, both using bulk comparison of words containing α packed characters. Bulk comparisons are done by comparing words; in case of a mismatch the mismatch position can be identified using bitwise `xor` operation, and then finding the most significant set bit.

A mismatch during the attempt to verify the suffix v allows the algorithm to shift the pattern ahead until v is aligned with the text after the mismatch. A mismatch during the

attempt to verify u, or after successfully matching u, causes the algorithm to shift the pattern ahead by $\pi(x)$ location. In either case the time adds up to only $O(\frac{n}{\alpha})$. ◀ ◀

When the period of the pattern is shorter than the word size, that is $\pi(x) \leq \alpha$, there may be several occurrences of the pattern starting within each word. The algorithm is very similar to the long period algorithm above, but with special care to efficiently manipulate the bit-masks representing all the occurrences.

▶ **Lemma 6.** *There exists an algorithm that finds all occurrences of a short-period long pattern of length m, such that $m > \alpha > \pi(x)$, in a text of length n in $O(\frac{n}{\alpha})$ time using $O(1)$ auxiliary space.*

Proof. Let p be the prefix of x of length $\pi(x)$, and write $x = p^r p'$, where p' is a prefix of p. If we can find the maximal runs of consecutive ps inside the text, then it is easy to locate the occurrences of x. To this end, let $k \leq r$ be the maximum positive integer such that $k \cdot \pi(x) \leq \alpha$ while $(k+1) \cdot \pi(x) > \alpha$. Note that there cannot exist two occurrences of p^k that are completely inside the same word.

We examine one word w of the text at a time while maintaining the current run of consecutive ps spanning the text word w' preceding w. We apply `wssm` to p^k and $w'w$, and take the rightmost occurrence of p^k whose matching substring is completely inside $w'w$. We have two cases: either that occurrence exists and is aligned with the current run of ps, and so we extend it, or we close the current run and check whether p' occurs soon after. The latter case arises when there is no such an occurrence of p^k, or it exists but is not aligned with the current run of ps. Once all the maximal runs of consecutive occurrences of ps are found (some of them are terminated by p') for the current word w, we can decide by simple arithmetics whether $x = p^r p'$ occurs on the fly. ◀ ◀

Real-time algorithm. As mentioned in Section 2.1, the Crochemore-Perrin algorithm can be implemented in real time using two instances of the basic algorithm with carefully chosen critical factorizations [7]. Since we are following the same scheme here, our algorithm reports the output bit-mask of pattern occurrences ending in each text word in $O(1)$ time after reading the word. Thus, we can obtain a real-time version as claimed in Theorem 1.

2.3 Pattern preprocessing

Given the pattern x, the pattern preprocessing of Crochemore-Perrin produces the period length $\pi(x)$ and a critical factorization $x = uv$ (Section 2.1): for the latter, they show that v is the lexicographically maximum suffix in the pattern under either the regular alphabet order or its inverse order, and use the algorithm by Duval [10]. The pattern preprocessing of Breslauer, Grossi and Mignosi [7] uses Crochemore-Perrin preprocessing, and it also requires to find the prefix x' of x such that $|x'| > |u|$ and its critical factorization $x' = u'v'$ where $|u'| \leq |v'|$. Our pattern preprocessing requires to find the period π' for the first α characters in v (resp., those in v'), along with the longest prefix of v (resp., v') having that period. We thus end up with only the following two problems:

1. Given a string x, find its lexicographically maximum suffix v (under the regular alphabet order or its inverse order).
2. Given a string $x = uv$, find its period $\pi(x)$ and the period of a prefix of v.

When $m = O(\frac{n}{\alpha})$, which is probably the case in many situations, we can simply run the above algorithms in $O(m)$ time to solve the above two problems. We focus here on the case when $m = \Omega(\frac{n}{\alpha})$, for which we need to give a bound of $O(\frac{m}{\alpha})$ time.

▶ **Lemma 7.** *Given a string x of length m, its lexicographically maximum suffix v can be found in $O(\frac{m}{\alpha})$ time.*

Proof. Duval's algorithm [10] is an elegant and simple linear-time algorithm that can be easily adapted to find the lexicographically maximum suffix. It maintains two positions i and j, one for the currently best suffix and the other for the current candidate. Whenever there is a mismatch after matching k characters ($x[i+k] \neq x[j+k]$), one position is "defeated" and the next candidate is taken. Its implementation in word-RAM is quite straightforward, by comparing α characters at a time, except when the interval $[\min(i,j), \max(i,j)+k]$ contains less than α positions, and so everything stays in a single word: in this case, we can potentially perform $O(\alpha)$ operations for the $O(\alpha)$ characters (contrarily to the rest, where we perform $O(1)$ operations). We show how to deal with this situation in $O(1)$ time. We employ `wslm`, and let w be the suffix thus identified in the word. We set i to the position of w in the original string x, and j to the first occurrence of w in x *after* position i (using `wssm`). If j does not exist, we return i as the position of the lexicographically maximum suffix; otherwise, we set $k = |w|$ and continue by preserving the invariant of Duval's algorithm. ◀ ◀

▶ **Lemma 8.** *The preprocessing of a pattern of length m takes $O(\frac{m}{\alpha})$ time.*

3 Word-Size Instruction Emulation

Our algorithm uses two specialized word-size packed string matching instructions, `wssm` and `wslm`, that are assumed to take $O(1)$ time. In the circuit complexity sense both are AC^0 instructions, which are easier than integer multiplication that is not AC^0, since integer multiplication can be used to compute the parity [17]. Recall that the class AC^0 consist of problems that admit polynomial size circuits of depth $O(1)$, with Boolean `and`/`or` gates of unbounded fan-in and `not` gates only at the inputs.

While either instruction can be emulated using the four Russians' technique, table lookup limits the packing factor and has limited practical value for two reasons: it sacrifices the constant auxiliary space and has no more cache friendly access. We focus here on the easier and more useful main instruction `wssm` and propose efficient bit parallel emulations in the word-RAM, relying on integer multiplication for fast Boolean convolutions.

▶ **Lemma 9.** *After a preprocessing of $O(\omega)$ time, the $\omega/\log\log W$-bit* `wssm` *and* `wslm` *instructions can be emulated in $O(1)$ time on a ω-bit word RAM.*

3.1 Bit-parallel emulation of `wssm`

String matching problems under *general matching relations* were classified in [23, 24] into easy and hard problems, where easy problems are equivalent to string matching and are solvable in $O(n+m)$ time, and hard problems are at least as hard as one or more Boolean convolutions, that are solved using FFT and integer convolutions in $O(n \log m)$ time [1, 14]. To efficiently emulate the `wssm` instruction we introduce *two layers* of increased complexity: first, we observe that the problem can also be solved using Boolean convolutions, and then, we use the powerful, yet standard, integer multiplication operation, that resembles integer convolutions, to emulate Boolean convolutions. In the circuit complexity sense Boolean convolution is AC^0, and therefore, is easier than integer multiplication.

String matching and bitwise convolution via integer multiplication. Consider the Boolean vectors $t_0 \cdots t_{n-1}$ and $p_0 \cdots p_{m-1}$: we need to identify those positions k, such that $t_{k+i} = p_i$, for all $i \in [m]$. Given a text and a pattern, where each of their characters is

encoded in $\log_2 |\Sigma|$ bits, we can see them as Boolean vectors of length $\log_2 |\Sigma|$ times the original one. We can therefore focus on binary text and pattern. We want to compute the occurrence vector c, such that c_k indicates if there is a pattern occurrence starting at text position $k \in [n]$ (so we then have to select only those c_k that are on $\log_2 |\Sigma|$ bit boundaries in c). In general, we have

$$c_k = \bigwedge_{i=0,\ldots,m-1} (t_{k+i} = p_i) = \overline{\left(\bigvee_{i=0,\ldots,m-1} (t_{k+i} \wedge \overline{p_i}) \right) \vee \left(\bigvee_{i=0,\ldots,m-1} (\overline{t_{k+i}} \wedge p_i) \right)}.$$

Define the *OR-AND Boolean convolution operator* $\hat{c} = a \bigtriangledown b$ for the Boolean vectors $a = a_{n-1} \cdots a_0$, $b = b_{m-1} \cdots b_0$, and $\hat{c} = \hat{c}_{n+m-1} \cdots \hat{c}_0$, to be

$$\hat{c}_k = \bigvee_{i=\max\{0,k-(n-1)\},\ldots,\min\{m-1,k\}} (a_{k-i} \wedge b_{m-i-1}).$$

Then, the occurrence vector c can be computed by taking the least n significant bits from the outcome of two convolutions, $\hat{c} = \overline{(t \bigtriangledown \overline{p}) \vee (\overline{t} \bigtriangledown p)}$. Treating the Boolean vectors as binary integers with the left shift operator $\ll$, we can compute $a \bigtriangledown b$ using standard integer multiplication $a \times b$, but the sum has to be replaced by the OR operation:

$$a \bigtriangledown b = \bigvee_{i=0,\ldots,m-1} [(a \ll i) \times b_i] = a \times b \quad \text{(where } + \text{ is replaced by } \vee\text{)}.$$

Observe the following to actually use the plain standard integer multiplication $a \times b$. Since the sum of up to m Boolean values is at most m, it can be represented by $L = \lceil \log m + 1 \rceil$ bits. If we pad each digit of a and b with L zeros, and think of each group of $L + 1$ bits as a *field*, by adding up at most m numbers the fields would not overflow. Thus, performing the integer multiplication on the padded a and b gives fields with zero or non-zero values (where each field actually counts the number of mismatches). Adding the two convolutions together we get the overall number of mismatches, and we need to identify the fields with no mismatches, corresponding to occurrences and compact them. In other words, if we use padded vectors $t', \overline{t'}, p'$, and $\overline{p'}$, we can compute $r = (t' \times \overline{p'}) + (\overline{t'} \times p')$ and set $\hat{c}_k = 0$ if and only if the the corresponding field in r is non-zero.

We use the constant time word-RAM bit techniques in Fich [13] to pad and compact. Note that in each field with value f we have that $0 - f$ is either 0, or borrows from the next field 1s on the left side. Take a mask with 1 in each field at the least significant bit, and subtract our integer m from this mask. We get that only zero fields have 0 in their most significant bit. Boolean AND with the mask to keep the most significant bit in each field, then shift right to the least significant bit in the field. The only caveat in the above "string matching via integer multiplication" is its need for padding, thus extending the involved vectors by a factor of $L = \Theta(\log m) = O(\log w)$ since they fit into one or two words. We now have to use L machine words, incurs a slowdown of $\Omega(L)$. We next show how to reduce the required padding from L to $\log \log \alpha$.

Sparse convolutions via deterministic samples. A *deterministic sample (DS)* for a pattern with period length π is a collection of at most $\lceil \log \pi \rceil$ pattern positions, such that any two occurrence candidate text locations that match the pattern at the DS must be at least π locations apart [26]. To see that a DS exists, take π consecutive occurrence candidates. Any two candidates must have at least one mismatch position; add one such position to the DS and keep only the remaining minority candidates, removing at least half of the remaining candidates. After at most $\lceil \log \pi \rceil$ iterations, there remains only one candidate and its DS. Moreover, if the input characters are expanded into $\log_2 |\Sigma|$ bits, then the DS specifies only

$\lceil \log \pi \rceil$ bits, rather than characters. Candidates can be eliminated via Boolean convolutions with the two bit vectors representing the 0s and 1s in the DS, that is, sparse Boolean vectors with at most $\lceil \log \pi \rceil$ set bits. The period π, the DS, and the other required masks and indices are precomputed in $O(\omega)$ time.

Consider now how we performed string matching via integer multiplication in the previous paragraph. Then, the padding in the bitwise convolution construction can be now reduced to only $L' = \lceil \log \log \pi + 1 \rceil$ bits instead of L bits, leading to convolutions of shorter $O(\omega \log \log \pi) = O(\omega \log \log \omega)$ bit words and slowdown of only $O(\log \log \omega)$ time. Using ω-bit words and $O(\omega)$-time preprocessing, we can treat $O(\omega / \log \log \omega)$ bits in $O(1)$ time using multiplication, thus proving Lemma 9.

3.2 `wssm` on contemporary commodity processors

Benchmarks of packed string matching instructions in "Efficient Accelerated String and Text Processing: Advanced String Operations" *Streaming SIMD Extension (SSE4.2) and Advanced Vector Extension (AVX)* on Intel Sandy Bridge processors [18, 20] and Intel's Optimization Reference Manual [19] indicate remarkable performance. The instruction *Packed Compare Explicit Length Strings Return Mask (PCMPESTRM)* produces a bit mask that is suitable for short patterns and the similar instruction *Packed Compare Explicit Length Strings Return Index (PCMPESTRI)* produces only the index of the first occurrence, which is suitable for our longer pattern algorithm.

Faro and Lecroq kindly made their *String Matching Algorithms Research Tool (SMART)* available [12]. Benchmarks show that for up to 8-character patterns, the raw packed string matching instructions outperformed all existing algorithms in SMART. The Crochemore-Perrin algorithm with packed string matching instructions performed very well on longer patterns. These preliminary experimental results must be interpreted cautiously, since on one hand we have implemented the benchmarks very quickly, while on the other hand the existing SMART algorithms could benefit as well from packed string matching instructions and from other handcrafted machine specific optimization; in fact, a handful of the existing SMART algorithms already use other Streaming SIMD Extension instructions.

4 Conclusions

We demonstrated how to employ string matching instructions to design optimal packed string matching algorithms in the word-RAM, which are fast both in theory and in practice. There is an array of interesting questions that arise from our investigation. *(1)* Compare the performance of our algorithm using the hardware packed string matching instructions to existing implementations (e.g. Faro and Lecroq [12] and platform specific *strstr* in *glibc*). *(2)* Derive Boyer-Moore style algorithms that may be faster on average and skip parts of the text [6, 27] using packed string matching instructions. *(3)* Extend our results to dictionary matching with multiple patterns [3]. *(4)* Improve our emulation towards constant time with ω-bit words and AC^0 operations. *(5)* Find critical factorizations in linear-time using only equality pairwise symbol comparisons: such algorithms could also have applications in our packed string model, possibly eliminating our reliance on the `wslm` instruction.

References

1 A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

2 R. A. Baeza-Yates. Improved string searching. *Softw. Pract. Exper.*, 19(3):257–271, 1989.

3 D. Belazzougui. Worst Case Efficient Single and Multiple String Matching in the RAM Model. In *Proceedings of the 21st International Workshop On Combinatorial Algorithms (IWOCA)*, pages 90–102, 2010.

4 M. Ben-Nissan and S. Tomi Klein. Accelerating Boyer Moore searches on binary texts. In *Proceedings of the 12th International Conference on implementation and Application of Automata (CIAA)*, pages 130–143, 2007.

5 P. Bille. Fast searching in packed strings. *J. Discrete Algorithms*, 9(1):49–56, 2011.

6 R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20:762–772, 1977.

7 Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple Real-Time Constant-Space String Matching. In Raffaele Giancarlo and Giovanni Manzini, editors, *CPM*, volume 6661 of *Lecture Notes in Computer Science*, pages 173–183. Springer, 2011.

8 Y. Césari and M. Vincent. Une caractérisation des mots périodiques. *C.R. Acad. Sci. Paris*, 286(A):1175–1177, 1978.

9 M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):651–675, 1991.

10 J.P. Duval. Factorizing Words over an Ordered Alphabet. *J. Algorithms*, 4:363–381, 1983.

11 S. Faro and T. Lecroq. Efficient pattern matching on binary strings. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, 2009.

12 S. Faro and T. Lecroq. The exact string matching problem: a comprehensive experimental evaluation report. Technical Report 0810.2390, arXiv, Cornell University Library, 2011. `http://arxiv.org/abs/1012.2547`.

13 F. E. Fich. Constant time operations for words of length w. Technical report, University of Toronto, 1999. `http://www.cs.toronto.edu/~faith/algs.ps`.

14 M.J. Fischer and M.S. Paterson. String matching and other products. In *Complexity of Computation*, pages 113–125. American Mathematical Society, 1974.

15 K. Fredriksson. Faster string matching with super-alphabets. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 44–57, 2002.

16 K. Fredriksson. Shift-or string matching with super-alphabets. *IPL*, 87(4):201–204, 2003.

17 M. L. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

18 Intel. *Intel® SSE4 Programming Reference*. Intel Corporation, 2007.

19 Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Co., 2011.

20 Intel. *Intel® Advanced Vector Extensions Programming Reference*. Intel Corporation, 2011.

21 D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322–350, 1977.

22 M. Lothaire. *Combinatorics on Words*. Addison-Wesley, Reading, MA, U.S.A., 1983.

23 S. Muthukrishnan and K. V. Palem. Non-standard stringology: algorithms and complexity. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 770–779, 1994.

24 S. Muthukrishnan and H. Ramesh. String Matching Under a General Matching Relation. *Inf. Comput.*, 122(1):140–148, 1995.

25 J. Tarhio and H. Peltola. String matching in the DNA alphabet. *Software Practice Experience*, 27:851–861, 1997.

26 U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. *SIAM J. Comput.*, 20(1):22–40, 1990.

27 Andrew Chi-Chih Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.