TARAS SHEVCHENKO NATIONAL UNIVERSITY OF KYIV
Faculty of Computer Science and Cybernetics
Department of Mathematical Informatics

QUALIFICATION WORK
for obtaining a master's degree
in a training direction 122 Artificial Intelligence
on the topic:

STRING PATTERN MATCHING ALGORITHMS WITH SIMD TECHNOLOGY

Made by 2-nd year student
Anton Zuiev _____(signature)

Academic adviser:
Professor Igor Zavadsky _____(signature)

I certify that in this work there are no borrowings from
the works of other authors without the corresponding
references
Student _____(signature)

The work was considered and allowed to be defended at the session of the
Department of
Mathematical Informatics
«__» _____ 2021y.,
protocol №____
Acting Head of Department
V. Tereshenko _____(signature)

Kyiv – 2021

Table of Content

Table of Content 2

Introduction 4

String pattern matching 5
- Basic description 5
- Time and memory complexity 7
- Search window 8
- Shift table 8
- Multiple search windows 9
- Existing algorithms 10
  - KMP 10
  - BM/BMH 12
  - BNDM 15
  - FJS 17
  - Zk/RZk 18

SIMD 20
- Basic concepts 20
- Used instructions 21

Modifications 23
- SIMD version 1 25
- SIMD version 2 25
- SIMD version 3 25
- SIMD range comparison 25

Test application 27
- Images 28
- Application interface 30
- Data generation 32

Test environment 33

Results 34
- Implementations description 34
- Sigma = 256 35

Up to 10 matches 35
Up to 20 matches 36
Up to 50 matches 37
Sigma = 192 38
Up to 10 matches 38
Up to 20 matches 39
Up to 50 matches 40
Sigma = 128 41
Up to 10 matches 41
Up to 20 matches 42
Up to 50 matches 43
Analysis 45
Future work 46
Conclusion 47
References 48
Implementations 49
Algorithms 49
Testing 54

Introduction

This work is dedicated to development of pattern matching algorithms using SIMD instructions. These instructions, provided by processor creators, can be used to perform parallel data operations of software level . Despite the fact that these instructions can be highly dependent on target machine configuration, they are still viable for server sided applications (search engines, online editors etc.) and might be used in applications distributed for different target machine architectures. The aim of this work is to understand, does development of architecture-dependent implementations provide significant enough performance boost to justify resources spent on them.

Theoretical part of this work provides a description of pattern matching task, short description about state of art in this area (including Zk/RZk algorithms which are used as base for modifications).

Practical part of this work contains a few modifications of mentioned algorithms, testing application and test results.

All source code located here https://github.com/anzue/PatternMatchingAlgorithmsTester,

including runtime results in table form https://github.com/anzue/PatternMatchingAlgorithmsTester/tree/master/results

String pattern matching

Basic description

In this short part of the work we'll describe formally what a pattern matching task is and what this work attempts to achieve in the area.

The formal definition of the task is the following:

Given:

1. Integer value Σ - alphabet (and $|\Sigma|$ - length of the alphabet) on which all operations performed. Usually $|\Sigma|$ is somewhere in range [2,256]. Possible values of characters are often represented as $0, 1, 2, \dots, (|\Sigma| - 1)$.
2. Integer value M - length of the pattern
3. Integer value N - length of the text
4. Pattern P - sequence of characters from Σ with length M ($|P| = M$)
5. Text T - sequence of character from Σ with length N ($|T| = N$)

Let's define indexing on pattern and the text based on character order in them and address *i-th* character from beginning $P[i]$ and $T[i]$ respectively.

Let's define substring of string S with left limit i and right limit j, such that $i <= j$ as new string $(S[i], S[i+1], \dots, S[j])$.

Match - some position i in text, such that $P = T[i,i+m]$ (when talking about string equality we always mean per-element equality)

So the pattern matching task is:

Find total amount/all positions/first position of the matches of pattern P in text T.

The choice usually depends on the specifics of the subtask. Also, algorithms able to solve at least one of them efficiently are able to solve all three. So for the simplicity we'll just focus on the first one.

As we saw earlier, pattern matching has 5 variables (obviously, some of them dependent, but lets work with this). The thing is, the content of the pattern P and text T is proven to have *in general* lesser impact on algorithms performance than *M,N* and $|\Sigma|$. It's still possible to find cases when "amazing" algorithm will be very slow (up to $O(M*N)$ time complexity), but these cases are very rare (read "almost impossible"). That's why we're really interested in performance in a distance over multiple test cases.

Also, it's noticeable that when *N* is significantly larger than *M* (it's usually written as $N >> M$), the size of *N* doesn't really matter in the long run as well. Increasing or reducing *N x* times, while keeping the same structure, will result in roughly *x* times reduced/increased runtimes.

As for M, it's size matters. For example, for $M <= 64$ some tricky bit mask approaches are more easily applicable, for small Ms checking for matches in a naive way seems more doable than for large. Large patterns, on the other hand, greatly benefit algorithms that are able to perform a lot of jumps based on pattern length.

$|\Sigma|$ is significant too. When $|\Sigma|$ is a small probability that text will contain some large pieces of pattern in positions with no match, however these values are more easily packed into variables and better for making some hash-tricks. Big $|\Sigma|$ can provide you confidence that if the first 2-3 characters of pattern are matched in some position in text then this position has a high probability of being an actual match (in other words, some first letters/ rare letters heuristics are more effective).

So, we determined, that parameters, that are really worth looking into are M and $|\Sigma|$. They'll be experimented on during the practical part. N will remain a constant large number. Content of P and T will be generated randomly during runtime of each experiment.

Time and memory complexity

When talking about algorithms people often talk about complexity. And it is also mentioned in this work a few times. But what is this? Let's provide a short description.

For example we have some algorithm that has 1 input number n and does some calculations. Fnd we are able to mathematically prove that the amount of operations he uses is strictly between $2n^4 + 3n^3 + 5n^2 - 7n + 4$ and $6n^4 - 3n^3 + 7n^2 - 1$. Writing something like this 1 time takes half a line and duplicating this every time doesn't make a lot of sense. And if +4 changes to +5 for some reason you need to change a lot of occurrences. We also may notice that for big n values $2n^4$ and $6n^4$ are a lot more significant than other parts of the sum, So we may start by ignoring other parts and saying that the amount of operations is "roughly" between $2n^4$ and $6n^4$. We may go even further. For big n $2n^4$ and $6n^4$ are numbers of the same order, while something like $3n^5$ will have n times more digits. Also calculating constants every time may be a bit tricky.

So to save time on calculations the O-notation was introduced. Let's say that if we able to prove, that execution time of the algorithm is smaller than $C*f(n)$ for each large n (formally speaking, for each $n > n_0$, where n_0 is some constant) where $f(n)$ is any function and C is constant. Then we can say that the execution time of the algorithm is $O(f(n))$. There's similar definitions for complexity lower bound $(o(f(n))$ and for exact complexity $(\Omega(f(n))$ but we'll not dive into this topic here. You may find more about this in source [9]. This definition allows us to ignore small addictive parts of algorithm execution times. For example above we may say that $6n^4 - 3n^3 + 7n^2 - 1 < 6n^4 + 7n^4 = 13n^4 = O(n^4)$. Looks a lot simpler. Definitions while having more than 1 input variable work the same way.

As mentioned earlier, while talking about pattern matching algorithms, 3 variables matter - M,N,Σ. But the sad part is, despite complexity being an important measure for most of the algorithm development field it matters not that much for our task. Most of the pattern matching algorithms have proven worst time complexity $O(M*N)$ and average time complexity like $\Omega(M+N)$ or $\Omega(N)$, which is the same most of the time (because $M <<$

N). However, in pattern matching constants inside O-notation is what matters. And calculation of this constant is generally hard. It is possible to calculate math expectation on a random text, but performance for real cases is hard, if at all possible to calculate theoretically. Therefore practical part of this work will contain no time/memory complexity estimations (it's possible to show all of them need *O(M)* or *O(1)* memory, *O(N + M)* average case time and *O(N*M)* worst case time).

Search window

This work uses the term "Search window" a lot. Let's describe what it means. When searching for a match of the pattern P it text T most (probably all, but let's stick with "most") pattern matching algorithms try to "align" pattern with some position of the text. Most of the time the search window is located in the position of comparison of characters in pattern and text (something like *if(P[m-1] == T[j + m-1]* tells us that search window is *T[j:j+m-1]*) . Also, a good way to locate the search window position is to look into comparison implementation. When talking about "search window movement" this means relative change of positions of the pointers we're determining our search window by.

Shift table

Pattern matching algorithms often use so called shift tables. The aim of these tables is to precalculate some information bases on pattern structure and afterwards be able to shift search window bases of this table content. For example, if our search window ends in position i of string S and we're moving it right, we may want to determine the shift based on last occurrence in the string of character S[i+1], or pair (S[i+1], S[i+2]) or something like this depending on the algorithm. In case of using more than 1 value as key for the shift table it's better to use some kind of hash/combine these values some way (one of these ways described in Zk/RZk algorithms description).

Multiple search windows

Due to the way compilation transforms code written by programmers into machine code, the loops are inherently a very slow part of the program. And, if evading something like functions during pattern matching is completely possible, loop usage is still required, because there is no more efficient way to process the whole text. But, some time during loop execution is still savable. Each time loop iteration ends, some resources are spent to jump to the loop start in code and to loop condition calculation. And there is a way to avoid this sometimes. In source [6] proposed to use 2 search windows moving towards each other. We still have some conditions on text start, but we'll check this 2 times less than before and save 50% of the resources needed to start new iterations of the loop. Of course, this may increase complexity of the inner part and condition too much, but practical results show that it really isn't. Sometimes using 3,4,5 and even 8 windows proves to be efficient. Also, this trick is easily applicable to most (if not all) existing algorithms.

Existing algorithms

This part of the work aims to give a general idea of how some of the pattern matching algorithms work. Pseudocodes from other works have [Source: *] in the end with info about referenced work.

KMP

KMP stands for Knuth–Morris–Pratt algorithm. It is the first string matching algorithm with linear execution time ($O(M+N)$). It may seem that having guaranteed execution time is a good thing, but it really isn't. Most of the common pattern matching algorithms have worst case scenario $O(M*N)$ for pattern $P = 1^M$ and text $P = 1^N$, but these cases are rare (if not impossible) in practice. However, these algorithms usually have some mechanics(skip/jump/shift tables for example), that provide them the ability to sometimes(or quite often, depending on M,N,Σ) skip up to M characters of the text and shift search window ahead (or behind, if moving backwards) them. Therefore, KMP, who always looks at each character of the text exactly once doesn't see use now.

The idea behind KMP is relatively hard (compared to most other algorithms). For this we'll introduce so-called prefix-function. Prefix function of a given string S at position i is the greatest $j < i$, such that $S[0:j] = S[i-j:i]$. If for some j $S[0:j] = S[i-j:i]$ then $S[0:j-1] = S[i-j-1:i-1]$ and it is possible to prove that $j-1$ is a prefix function of $i-1$, or prefix function of prefix function etc. So, we have a deterministic algorithm of calculating prefix function for given i: iterate over g = P[i-1], P[P[i-1]], … until you have $S[g+1] = S[i]$. This is our prefix function. This allows us to calculate prefix functions for the whole string. And there is proof (we will not show it here, the work isn't dedicated to KMP after all) that total amount of iterations spend to calculate prefix function is linear of string length ($O(|M|)$). So, to find all matcher for pattern matching algorithm all we have to do is calculate prefix function of P + "#" + T where "#" is some delimiter not present in the text. After this each position with prefix function equal pattern length is a match. Other

versions skip concatenation and keep strings separate, using the same logic. One of these implementations is shown below

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed, m ← j - k  may be
returned here)
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

[Source: https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm]

BM/BMH

BM stands for Boyer–Moore algorithm. It sees (almost) no use, but it’s just a classical example of pattern matching algorithms.

1. Construct a skip table. For each character of the alphabet calculate, how far from the end of the string the first occurrence of this character is located (and length of pattern if character doesn't occur in the pattern).
2. Let's start a loop from the start of the pattern to the end.
3. Check occurrence at the current position.
4. Having the search window end located at some position of the text, we may use shift table values to calculate how much we need to shift the search window with the next letter after the search window. Unless we reach the end of the text, go to paragraph 3.

Ideas for shifts calculations are following:

1. Bad character rule - when you previous comparison fails, you find the next position in your string where this character is located and shift your string for at least position difference. This seems like a good idea, because generally comparison will more often fail on rare characters which can sometimes provide significant shifts
2. Good suffix rule - while comparing pattern and search window from the end, remember the size of the suffix equal to the size of the pattern. When comparison stops, we are able to use the distance to the next occurrence of this suffix in the pattern as possible shift. These values, as well as values for the first rule are precalculated.

Code will look something like this

```
// preprocessing for strong good suffix rule
void preprocess_strong_suffix(int *shift, int
*bpos,
                              char *pat, int m)
{
   // m is the length of pattern
   int i=m, j=m+1;
   bpos[i]=j;

   while(i>0)
   {
      while(j<=m && pat[i-1] != pat[j-1])
      {
          if (shift[j]==0)
             shift[j] = j-i;

         j = bpos[j];
      }
      i--;j--;
      bpos[i] = j;
   }
}

void preprocess_case2(int *shift, int *bpos,
                      char *pat, int m)
{
   int i, j;
   j = bpos[0];
   for(i=0; i<=m; i++)
   {
      if(shift[i]==0)
         shift[i] = j;

     if (i==j)
         j = bpos[j];
   }
}
```

```
void search(char *text, char *pat)
{
   // s is shift of the pattern with respect to text
   int s=0, j;
   int m = strlen(pat);
   int n = strlen(text);

   int bpos[m+1], shift[m+1];

   //initialize all occurrence of shift to 0
   for(int i=0;i<m+1;i++) shift[i]=0;

   //do preprocessing
   preprocess_strong_suffix(shift, bpos, pat, m);
   preprocess_case2(shift, bpos, pat, m);

   while(s <= n-m)
   {

      j = m-1;

     while(j >= 0 && pat[j] == text[s+j])
        j--;

     if (j<0)
      {
         printf("pattern occurs at shift = %d\n", s);
         s += shift[0];
      }
      else
         /*pat[i] != pat[s+j] so shift the pattern
           shift[j+1] times  */
         s += shift[j+1];
   }

}
```

[Source : https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic]

BMH stands for Boyer–Moore–Horspool algorithm. Whatever said about the BM algorithm search phase stands for BMH as well. However it uses only the skip table based on how much we need to shift based on the last character of the pattern to get the same character in the same position. In general this boosts the performance of the algorithm by quite a bit.

Below provided pseudocode from Wikipedia displaying possible implementation

```
function preprocess(pattern)
    T ← new table of 256 integers
    for i from 0 to 256 exclusive
        T[i] ← length(pattern)
    for i from 0 to length(pattern) − 1 exclusive
        T[pattern[i]] ← length(pattern) - 1 - i
    return T

function same(str1, str2, len)              Compares two strings, up to the first len characters.
    i ← len - 1
    while str1[i] = str2[i]                Note: this is equivalent to !memcmp(str1, str2, len).
        if i = 0                             The original algorithm tries to play smart here: it checks for
the
            return true                      last character, and then starts from the first to the second-
last.
        i ← i - 1
    return false

function search(needle, haystack)
    T ← preprocess(needle)
    skip ← 0
    while length(haystack) − skip ≥ length(needle)
        haystack[skip:] -- substring starting with "skip". &haystack[skip] in C.
        if same(haystack[skip:], needle, length(needle))
            return skip
        skip ← skip + T[haystack[skip + length(needle) - 1]]
    return not-found
```

[source: https://en.wikipedia.org/wiki/Boyer-Moore-Horspool_algorithm]

BNDM

BNDM stands for Backward Nondeterministic Dawg Matching algorithm. For each character *c* algorithm saves a bitmask which shows if some position of pattern has character *c* or not. Mathematically speaking, $mask[c][i] = 1$ if $P[i] = c$ and 0 otherwise. The problem with this algorithm already can be seen. It will be limited to patterns with length not greater than the bit size of type used for the mask, usually this is 32 or 64. And even in cases, when it's applicable, having to work with big alphabets will cause us to do a lot of unnecessary bit operations we could've avoided and miss a lot of good skips. But in general I find this interesting as a approach, targeting some specific values of pattern length and alphabet size for optimizing.

Search contains 2 loops. Inner loop iterates on length of text. Outer loop iterates on the window, starting at outer loop position, ending at that position plus M and going backwards. Let's say that outer loop position is *i* and inner loop position is *j* going from $i+M-1$ down to *i*

In the outer loop assign some variable $f = (1 << M) - 1$ and update *f* in the inner loop using formula $f = (f \& mask[T[j]]) << 1$

The next ideas apply for the algorithm:

1. After iterating for M characters, if $f_{M-1}=1$ we have a match
2. If at position $j > i$ before we had $f_{M-1}=1$ we have a match of some prefix of pattern with the window, $T[j : i + M-1] = P[0, i+M - j-1]$
3. If at some moment $f = 0$ there is no need to continue iterating

As we may see, this algo is quite simple of the level of idea.

```
for a ∈ Σ do B[a] ← 0 endfor
for j ← 1..m do
        B[p_j] ← B[p_j ] | (1 << (m − j))
endfor
i ← 0
while i ≤ n − m do
        j ← m;
        last ← m;
        D ← (1  m) − 1
        while D != 0 do
                D ← D & B[t_{i+j} ]
                j ← j − 1
                if D & (1  (m − 1)) != 0 then
                        if j > 0
                                then last ← j
                                else report occurrence at i + 1
                        endif
                endif
                D ← D  << 1
        endwhile
        i ← i + last
endwhile
```

[source : [7]]

BNDM as it isn't used much, however some it's modifications are more effective and practically applicable.

FJS

FJS stands for Franek-Jennings-Smyth. The idea behind this algorithm is to combine other modifications of Boyre-Moore algorithm (Boyer-Moore-Sunday to be precise) with Knuth-Morris-Pratt algorithm. Generally the search phase is performed in a manner of BHM, just moving right to left. But when a character is found in the text, instead of naive comparison we perform a KMP-like matching operation. After that we perform this KMP-like iteration for as long as we have a non-empty segment of the search window as a prefix of the pattern.

This algorithm is probably one of the few (except slow KMP and automata algorithms for small patterns/alphabets) algorithms able to perform well on average cases at the same time as having it's worst case *O(N)* which, in my opinion, makes it remarkable.

```
void makebetap( const CTYPE* p, int m ) {
    int i = 0, j = betap[0] = -1;

    while( i < m ) {
      while( (j > -1) && (p[i] != p[j]) ) {
        j = betap[j];
      }
      if( p[++i] == p[++j] ) {
        betap[i] = betap[j];
      } else {
        betap[i] = j;
      }
    }
  }

  void makeDelta( const CTYPE* p, int m ) {
    int i;

    for( i = 0; i < ALPHA; ++i ) {
      Delta[i] = m + 1;
    }
    for( i = 0; i < m; ++i ) {
      Delta[ p[i] ] = m - i;
    }
  }

  void FJS( const CTYPE* p, int m, const CTYPE* x, int n ) {
    if( m < 1 ) return;
    makebetap( p, m );
    makeDelta( p, m );

    int i = 0, j = 0, mp = m-1, ip = mp;

    while( ip < n ) {
      if( j <= 0 ) {
        while( p[ mp ] != x[ ip ] ) {
          ip += Delta[ x[ ip+1 ] ];
          if( ip >= n ) return;
        }
        j = 0;
        i = ip - mp;
        while( (j < mp) && (x[i] == p[j]) ) {
          ++i; ++j;
        }
        if( j == mp ) {
          output( i-mp );
          ++i; ++j;
        }
        if( j <= 0 ) {
          ++i;
        } else {
          j = betap[j];
        }
      } else {
        while( (j < m) && (x[i] == p[j]) ) {
          ++i; ++j;
        }
        if( j == m ) {
          output( i-m );
        }
        j = betap[j];
      }
      ip = i + mp - j;
    }
  }
```

[Source : [8]]

Zk/RZk

The basic idea behind both Zk and RZk is the same.

Shift table is some table, that by some character (or sequence of characters) from text defines how much we are able to shift our search window. These tables are the result of precalculations and used in many algorithms. Generally, the more characters in the shift table, the more the shift. But increasing shift table size decreases access time to it. Both algorithms have *k* in their names, this k stands for the amount of bits in the shift table. Basically, given two subsequent characters from the string *T[i], T[i+1]* we'll calculate there position in the shift table as *(T[i] * 256 + T[i+1]) & (1<<k - 1).* Basically this means : write binary codes together, take k less significant bits. This approach is named 1.5-byte read. Choice of k is a separate task on its own, but this work has so many modifications, so to avoid extending the amount of variations to 10+ it was decided to focus on k=13 for this work. Worth nothing, that under some conditions specific versions of algorithms (like Z8) may perform really well. The pseudocode for BMH-like algo with this idea will look like

```
mask = 2^k − 1;
foreach i ∈ [0; 2^k) do Z[i] ← 1;
for i = 0 to m − 2 do
Z[word(P[i], P[i + 1])] ← 0
        pos = m − 2;
while pos < n do
        while Z[word(T[pos], T[pos + 1])&mask] != 0 do
        pos += m − 1;
        check the occurrence at pos − m + 2;
        pos += QS[T[pos + 2]];
```

[Source : [3]]

QS is Quick search shift table based on character locations in pattern. But, what if we may be able to avoid entering a comparison check (which contains an internal loop) every time “1.5” characters matched? We can do this by actually trying to match the next “1.5” characters as well and launch comparison only when we got both matches. Pseudocode will look like this.

```
pos = −1;
while pos < n do
        repeat
                pos += m − 1;
                while Z[word(T[pos], T[pos + 1])&mask] != 0 do
                        pos += m − 1;
                --pos;
        until Z[word(T[pos], T[pos + 1])&mask] != 0 ;
        check the occurrence at pos − m + 3;
        pos += QS[pos + 3] − m + 2;
```

[Source : [3]]

This algorithm is a basic version of the Zk algorithm. The RZk algorithm does basically the same in the opposite order - instead of going from 1 to n they go from n down to 1. This change was forced by architectural reasons and described in work [3].

Also, worth noting, multiple windows idea, mentioned before, works great for both algorithms.

SIMD

Basic concepts

SIMD stands for “single instruction, multiple data”. The idea behind this principle is to provide software-sided operations on multiple data at the same time. Due to supreme speed of software implementations this can give significant improvements on simple tasks. *Machine used for development and testing has an Intel processor, therefore everything stated below might work differently/don’t work at all for other processors.*

SIMD operations are separated into technologies. Technologies are usually created for different processor families, for example, MMX, the oldest one was adder for Pentiums, and AVX512 is available only for server-oriented processor families. This work uses technologies SSE - SSE 4.2 because:

1. We often need to operate on 3 or more pointers of type int (32 bits), 96 bits requires at least a 128 bit variable to store it.
2. Despite the fact that the W5 version of RZk seems promising, best results are shown by W4-W2 versions. Due to an already large number of experimental versions a choice was made to focus on versions able to fit their pointers into 128

   List with instructions used for this work you may visit source [2].

Area of SIMD operation usage for pattern matching algorithms is currently underdeveloped. One of works on this area is source [1].

Let's take a look at some operations used in the algorithm and analyze why using them may be good or bad for performance.

Used instructions

Note: when speaking about 8 bit integers we usually mean char (one character from text or pattern) unless stated otherwise, 32bit integers are ints and used for pointers(search window positions) to positions in text.

Type of int variables used by SIMD is __m128i , when speaking about “128 bits variables” we mean them.

_mm_cvtsi128_si32	get lowest 32 bit integer from 128 bits variable. This operation is our method to unpack.
_mm_srli_si128	shift 128bit variable some amount of bytes right. When shift is 4, 8 or 12 we can use _mm_cvtsi128_si32 afterwards to extract packed variables from 128bits integer
_mm_set_epi8	pack 16 8bits integers into 128bits variable
_mm_sub_epi32	subtract 32bits integers contained in 2 128bits variables
_mm_setr_epi32	pack 4 32bits integers into 128bits integer
_mm_loadu_si128	load 128bits variable from pointer to memory. Useful to cast pointer to some point of text/pattern into pointer of type __m128i and load it into variable afterwards (a lot more efficient than using _mm_set_epi8 after loading 16 chars from memory)
_mm_cmpeq_epi8	compare 8bit integers packed in the __m128i

	and return mask that contains *0xFF* when respective variables are equal and 0 when not
_mm_movemask_epi8	take the most significant bit of each byte and store the result in the int variable. Useful to transform *0xFF* from _mm_cmpeq_epi8 into simple bitmask of match/not match
_mm_set1_epi8	fill 128bits variable with given 8bit value. Useful to compare sequences of values to the same given value using 1 operation
_popcnt32	Calculate amount of 1 bits in the int variable, combine with _mm_movemask_epi8 and _mm_cmpeq_epi8 to get a matcher for 16 characters

Modifications

SIMD version 1

Lets focus on w3/w4 versions of the algorithms. Let's take a quick look at a code fragment for the basic version of RZk W3

```
while (z[word(pos0) & mask] != 0
&& z[word(pos1) & mask] != 0
&& z[word(pos2) & mask] != 0) {
        pos0 -= m;
        pos1 -= m;
        pos2 -= m;
    }
```

As described earlier, this loop provides fast skips for the algorithms and the amount of executions of this part is generally large. Also subtraction operations are always the same - the amount subtracted is not dependent on any parameters and remains constant. Based on what we know about SIMD, we may try to replace 3 subtractions with 1. The result will look like this

```
while (z[word(get0_i128(packed_positions)) & mask] != 0
          && z[word(get1_i128(packed_positions)) & mask] != 0
          && z[word(get2_i128(packed_positions)) & mask] != 0) {
        packed_positions = _mm_sub_epi32(packed_positions, fill_m);
    }
```

Where get0_i128, get1_i128, get2_i128 are some defines to fast fetch int variables from packed __m128i variables. As of now they look like

```
#define get0_i128(i128) _mm_cvtsi128_si32(i128)
#define get1_i128(i128) _mm_cvtsi128_si32(_mm_srli_si128((i128), 4))
#define get2_i128(i128) _mm_cvtsi128_si32(_mm_srli_si128((i128), 8))
#define get3_i128(i128) _mm_cvtsi128_si32(_mm_srli_si128((i128), 12))
```

SIMD version 2

Let's start with the next idea. When in Zk/RZk algorithms 2 pairs of masks are matched we require comparison between whole pattern and whole text. In the basic version of the algorithm we just do this character by character. We can still parse pointers to *long long* to compare 8 characters per operation, but SIMD provides us with the ability to do this for 16 characters at once. So this idea allows us to save roughly 15/16 of the comparison time. Initial attempt was to pack all characters of the string into __m128i manually, however, that approach turned out to be very inefficient, because memory loading still performed for each variable. Using SIMD memory loading operations provided better results.

Even if the last paragraph sounds good, it's not good enough on it's own. Speeding up comparison can have small effects for following reasons:

1. Main part of the algorithm is search window/windows iterations when we shift pointers, having 2-2.5-3, depending on k value in Zk/RZk coincide is relatively rare event (especially for big alphabet sizes)
2. Often naive comparison might've failed on 3rd of 4th character and operations saved by SIMD in this case would be negligible

After that we may wonder, is there any good reason not to perform other loop operations on the packed variable? Mostly not. We still need to extract positions into separate variables to speed up match check but subtraction operations after check can be performed only on packed variable

SIMD 1	SIMD 2

if (z[word(pos0 + 1) & mask] == 0 && pos0 >= 0) {
for (i = 0; i < m && P[i] == T[pos0 + i]; ++i) {
};
if (i == m) {
MATCH(pos0);
}
pos0 -= RQS[T[pos0 - 1]];
} else
pos0 -= m - 1;
	if (z[word(pos0 + 1) & mask] == 0 && pos0 >= 0) {
	check_i128(pos0);
	sub0_i128(RQS[T[pos0 - 1]]);
	} else
	sub0_i128(mm1);

Here *check_i128* represent quite a lot of code to compare 16 characters at a time, *sub0_i128(val)* does

```
packed_positions = _mm_sub_epi32( packed_positions,  _mm_setr_epi32(val, 0, 0, 0));
```

sub1, sub2 and sub2 implemented in the same way for respective positions. After this modification we have quite a good version of an algorithm that already proves itself to be better than the basic version.

SIMD version 3

The idea behind this is relatively simple. While in version 2 we believed that only 3 or 4 variables really deserved to be packed, here we will pack 2 variables as well. More details in the implementation section.

SIMD range comparison

The idea behind this modification is based on source [2]. More detailed description may be found there.

For simplification let's assume that we are using the *RZk* algorithm, this means that the search window is moving backwards. Let's also say we want to compare text T starting in position pos with pattern. We may be able to instead compare each string
T[pos - 15 : pos - 15 + m], T[pos - 14 : pos - 14 + m], …, T[pos : pos + m]
Using roughly the same amount of operations. This will allow us to shift 16 positions left afterwards we compared this. Theoretically, this may be useful for short patterns.
To do this fast comparison we will do the following

1. Lets define the integer mask = 11...11 (16 lowest bits are 1)
2. Iterate positions in pattern (we doing it for 0 to m-1, but order doesn’t really matters)
3. For each position j in pattern mask |= SIMDcompare(T + pos - j - 15, P[j]). What does SIMDcompare do? It compares a range of 16 numbers from text starting from a

given position(T + pos - j - 15) with a given char (P[j]) and returns the result as a 16 bit mask (1-match, 0-not match). So, iteration for position j basically checks for each string, if this string matches pattern at position j, and sets 0 in mask for all that dont.

4. Report match for all strings with 1 in the mask after all iterations

The only question is how to implement SIMDcompare. We use implementation like this

```
_mm_movemask_epi8(_mm_cmpeq_epi8(_mm_loadu_si128(T_pos), _mm_set1_epi8(P_char)))
```

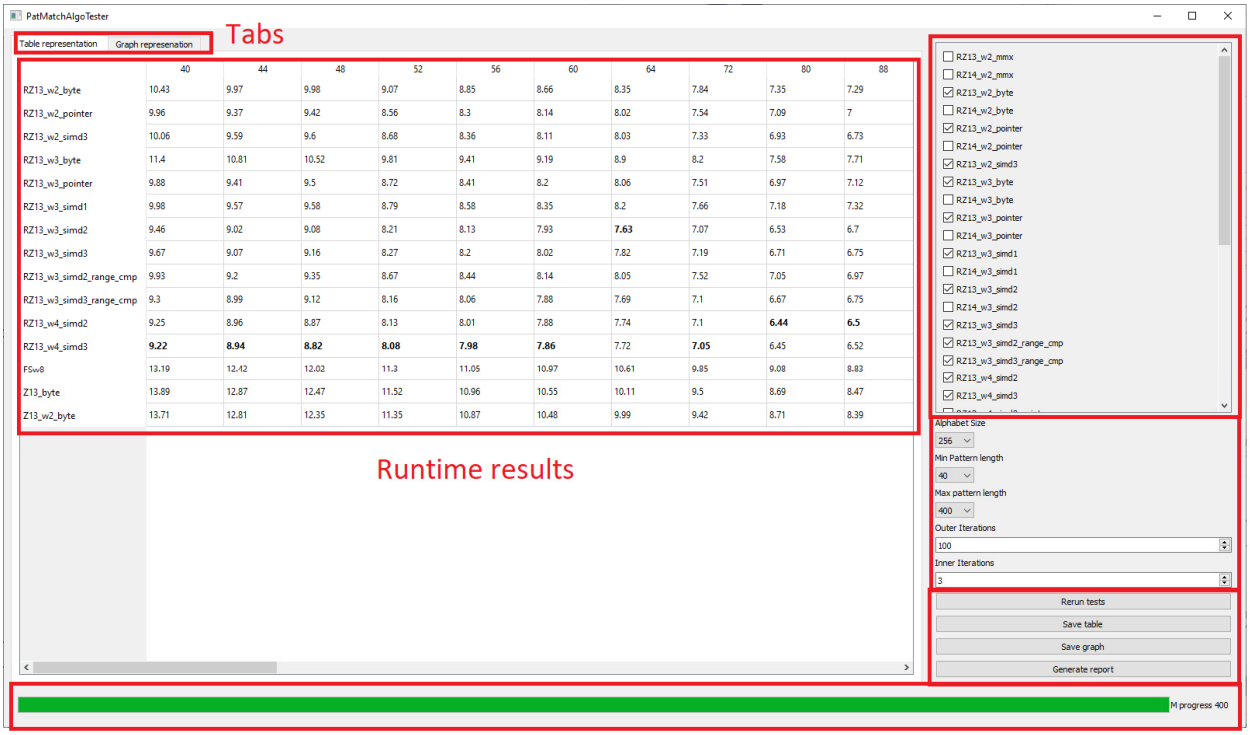
Test application

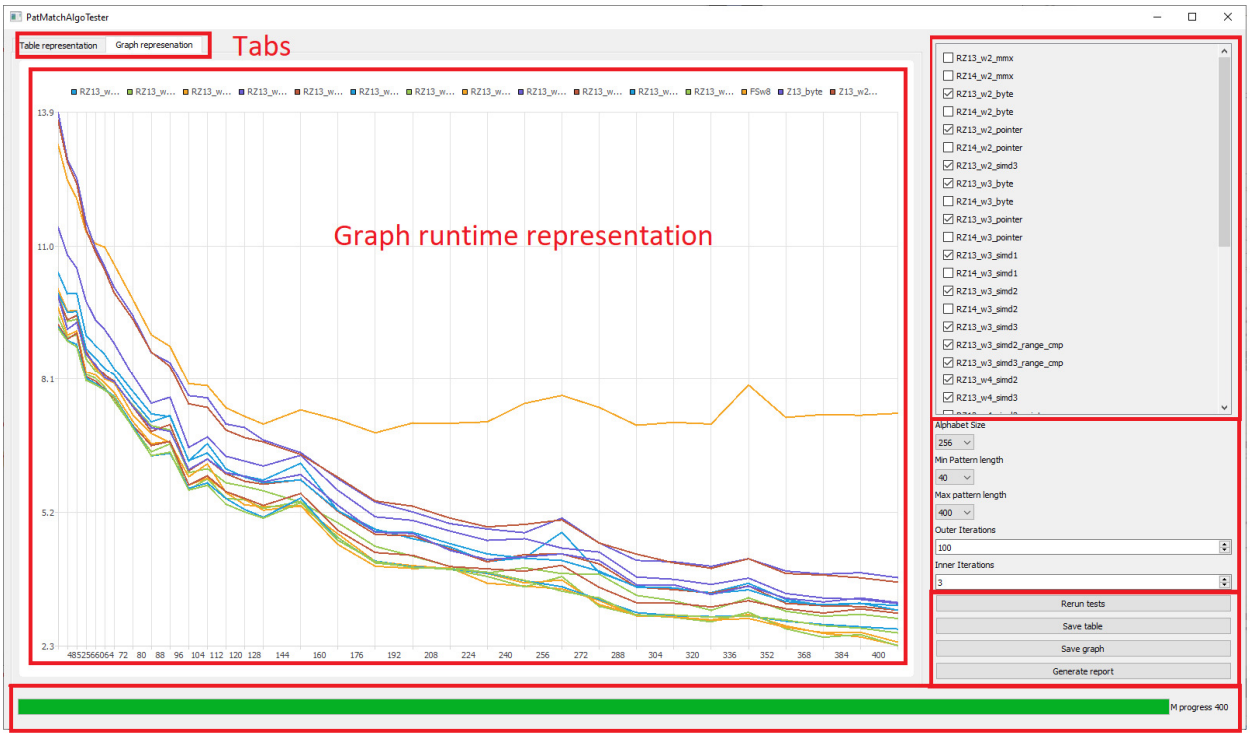
For the purposes of testing newly developed versions of the algorithm special environment was developed. Application was designed having next ideas in mind

1. Simplicity - it probably will be launched a lot of times so making it simpler will save a lot of time
2. Flexibility - C++ rebuild may take a lot of time, so the more parameters are configurable during runtime the better.
3. Understandable results - easy to see and understand performance of the algorithm compared to others
4. Additions - easy to add new algorithms and/or parameters to the flow without major changes
5. Simplicity in development - modifications to application can be performed fast in case they are required.

Considering these thoughts I decided to use the Qt framework for C++ to develop this application. I already had positive experience working with it and it (in my opinion) provides a good UI and easy to work with. It also provides integrated kit for graphs creation (QtChart) and also cross platform (tests were run only on windows for now, may be useful in the future).

Images


PatMatchAlgoTester
Table representation
Graph represenation
Tabs
Runtime results
Algorithm selec
Parameter selec
Control section
Progress bar
Alphabet Size
Min Pattern length
Max pattern length
Outer Iterations
Inner Iterations
Rerun tests
Save table
Save graph
Generate report
M progress 400


PatMatchAlgoTester
Table representation
Graph represenation
Tabs
Graph runtime representation
Algorithm selec
Parameter selec
Control section
Progress bar
Alphabet Size
Min Pattern length
Max pattern length
Outer Iterations
Inner Iterations
Rerun tests
Save table
Save graph
Generate report
M progress 400

Application interface

1.Tabs

Provides choice between table and graph view

2.Runtime results table

Provides average runtimes of algorithms per launch. Rows represent algorithms, columns represent pattern length (m). Fastest algorithms for each pattern length is highlighted.

3.Graph representation

Basically the same data as in the table but in a more visually understandable form. Not very useful when trying to optimize 5-10% of some particular set of pattern length values.

4.Algorithm selection

Allows to select by using dropboxes, which algorithms are launched. Provides a lot of flexibility in terms of ability to save a lot of time on time consuming launches in the form of ability to select algorithms by packs/test some subset of algorithms you are interested in.

5.Parameter selection

Allows to configure next execution parameters for the algorithm

I. Alphabet size

   Maximum value of integer stored in character. In range [2, 256]. Even if the value is small all chars are still stored in separate bytes.

II. Minimum/Maximum pattern lengths

   Sets the limits on how we iterate on pattern length. Allows to save time while investigating algorithm behaviour under some specific circumstances.

III. Outer iterations

   During each outer interaction pattern is recreated, that means all occurence in T are lost

IV. Inner iterations

   During each inner iteration pattern insert into text, therefore the more inner iterations value set, the more occurrences will happen. Allows to simulate

having different number of pattern occurrences in text without really having to have one.

6.Control section

I. Rerun tests

Launch test on selected parameters, update table and graph view.

II. Save table

Save current table into .md file

III. Save graph

Save graph as image

IV. Generate report

Launch algorithms with all parameters, except alpha and for all alphas from preset set and save result into file

7.Progress bar

Pretty straightforward. Shows progress on M during runtime. When running “Generate report” mode also shows progress on Alpha (alphabet size).

Data generation

This part describes how we generate data to test our algorithms. Ideally, we would like to have real datasets, but the need to have datasets for different alphabet sizes, combined with time required to collect real dataset a decision was made to use generated data.

Generation proceeds as follows:

1. Generate text containing 500000 characters using random int generator modulo alphabet size
2. For each of outer iterations generate pattern using same random int generator
3. For each of inner iterations insert one pattern occurrence into the text. Considering the relative sizes of pattern and text it's safe to assume that amount of matches is almost always equal to inner loop iterations number.

This kind of generation provides following benefits

1. Text is pseudorandom
2. Tests have cases with different number of matches, combined with ability to configure max inner loop iterations this gives us good flexibility.
3. Pattern is new each outer loop iteration, therefore launching with a big (I used 100) outer iterations number gives us a quite reliable result that is less likely to be deformed but some specific pattern generation result.

Runtime calculation//todo

Test environment

All coding and testing was done on a single machine with following configuration

OS	Windows 10 Home
Processor	Intel i5 8600k(no overclocking)
RAM	16 GB
Compiler	MinGW 8.1.0 32 bit
IDE	QtCreator 4.14.2
Qt	Qt 5.15.2
Qt additional packages	QtCharts

Results

Implementations description

1. RZ13_byte - basic version of RZk, k=13
2. RZ13_w2_byte - 2 windows version
3. RZ13_w2_pointer - 2 windows version, window positions are pointers instead of integers
4. RZ13_w2_simd3 - 2 windows with packing
5. RZ13_w3_byte - 3 windows version
6. RZ13_w3_pointer - 3 windows version, window positions are pointers instead of integers
7. RZ13_w3_simd1 - 3 windows version, added packing from SIMD 1
8. RZ13_w3_simd2 - 3 windows version, added packing and comparison from SIMD 2
9. RZ13_w3_simd3 - 3 windows version, added packing from SIMD 3
10. RZ13_w3_simd2_range_cmp - packing from SIMD 2, comparison from range cmp
11. RZ13_w3_simd3_range_cmp - packing from SIMD 3, comparison from range cmp
12. RZ13_w4_simd2 - 4 windows version, packing from SIMD 2
13. RZ13_w4_simd3 - 4 windows version, added packing from SIMD 3
14. FSw8 - non-Zk/RZk algorithm, to have general idea of behaviour compared to field algorithms.
15. Z13_byte - basic Zk algorithm, k=13
16. Z13_w2_byte - 2 windows Zk, k=13

The only k we are using is 13. Main reason for this is to save time, having to test 15 modifications for 2-3 different k each would be very inefficient. Best result for each pattern length is marked with the asterisk. Pattern length <32 aren't shown't, RZ13_w2_pointer is the best algorithm there.

Sigma = 256

Up to 10 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	12.89	9.9	9.18	7.77	7.31	6.71	6.26	5.2	4.56	4.12	4.18	3.55	3.42	3.27
RZ13_w2_byte	11.75	9.39	8.66	7.55	6.85	6.41	6.23	5.18	4.63	4.22	4.15	3.67	3.54	3.5
RZ13_w2_pointer	10.72*	8.81	8.39	7.15	6.57	6.26	5.9	4.82	4.32	3.91	3.87	3.38	3.32	3.21
RZ13_w2_simd3	11.58	9.2	8.58	7.39	6.73	6.2	5.76	4.69*	4.2*	3.76*	3.57*	3*	2.8*	2.49*
RZ13_w3_byte	12.72	10.06	9.22	7.89	7.52	6.99	6.5	5.6	5.17	4.54	4.35	3.84	3.55	3.45
RZ13_w3_pointer	11.03	8.96	8.42	7.42	6.78	6.35	6.12	5.08	4.66	4.19	4.05	3.6	3.42	3.33
RZ13_w3_simd1	11.16	8.96	8.56	7.33	6.84	6.6	6.23	5.23	4.8	4.36	4.15	3.7	3.6	3.49
RZ13_w3_simd2	11.31	8.74	8.25*	7.11	6.72	6.25	5.78	4.88	4.33	3.89	3.62	3.04	2.83	2.61
RZ13_w3_simd3	11.4	8.94	8.43	7.14	6.74	6.35	5.9	4.94	4.38	3.92	3.7	3.06	2.84	2.61
RZ13_w3_simd2_range_cmp	11.02	8.82	8.38	7.32	6.98	6.65	6.15	5.26	4.96	4.5	4.39	4.09	4.08	4.11
RZ13_w3_simd3_range_cmp	10.98	8.87	8.46	7.4	6.89	6.62	6.26	5.42	5.1	4.7	4.64	4.38	4.44	4.53
RZ13_w4_simd2	10.83	8.6	8.42	6.96*	6.53	6.15	5.69*	4.9	4.43	3.95	3.71	3.15	2.95	2.71
RZ13_w4_simd3	10.82	8.59*	8.29	7.11	6.49*	6.02*	5.74	4.92	4.43	3.94	3.73	3.13	2.95	2.72
FSw8	15.78	12.11	11.22	9.75	8.78	8.35	7.9	7.59	7.78	7.65	7.97	8.15	8.39	8.6
Z13_byte	16.41	11.94	10.24	8.97	8.1	7.35	6.81	5.79	5.38	4.85	4.77	4.37	4.28	4.16

Z13_w2_byte	15.94	11.78	10.19	8.73	7.97	7.48	6.79	5.92	5.49	4.93	4.85	4.38	4.14	4.02

Up to 20 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	12.56	9.92	8.6	7.63	6.76	6.19	6.27	5.42	4.89	4.6	4.63	4.23	4.41	4.23
RZ13_w2_byte	11.33	9.37	8.23	7.39	6.45	5.97	6.08	5.48	5.05	4.76	4.68	4.47	4.62	4.57
RZ13_w2_pointer	10.45	8.78	7.86	7.02	6.24	5.71	5.76	5.14	4.7	4.44	4.32	4.13	4.3	4.18
RZ13_w2_simd3	11.2	9.2	8.02	7.13	6.13	5.6	5.61	4.88	4.28	3.96	3.71	3.28	3.22	2.85
RZ13_w3_byte	12.52	10.1	8.85	7.58	6.65	6.21	6.37	5.88	5.47	5.09	4.84	4.58	4.57	4.48
RZ13_w3_pointer	10.86	8.87	7.92	6.99	6.24	5.81	5.98	5.44	4.98	4.7	4.53	4.36	4.41	4.28
RZ13_w3_simd1	10.92	9.01	8.15	7.12	6.43	6	6.11	5.6	5.2	4.89	4.71	4.61	4.64	4.62
RZ13_w3_simd2	10.78	8.82	7.77	6.84	5.94	5.47	5.56	4.97	4.47	4.09	3.75	3.38	3.29	3.01
RZ13_w3_simd3	10.91	8.87	7.93	6.91	6	5.55	5.6	5.01	4.47	4.12	3.75	3.41	3.27	3
RZ13_w3_simd2_range_cmp	10.81	8.88	8.08	7.22	6.45	6.12	6.22	5.84	5.54	5.34	5.32	5.41	5.81	5.92
RZ13_w3_simd3_range_cmp	10.83	8.92	8.2	7.35	6.59	6.26	6.45	6.14	5.85	5.76	5.76	6.01	6.6	6.79
RZ13_w4_simd2	10.59	8.65	7.86	6.81	5.91	5.45	5.48	5.01	4.5	4.15	3.79	3.52	3.37	3.16
RZ13_w4_simd3	10.47	8.63	7.78	6.76	5.91	5.48	5.49	4.98	4.49	4.13	3.8	3.49	3.36	3.17
FSw8	15.49	12.0	11.0	9.66	8.51	8.1	8.02	8.07	8.37	8.53	8.75	9.59	10.4	10.8
Z13_byte	15.96	11.8	10.0	8.65	7.63	7.06	6.85	6.1	5.74	5.37	5.35	5.23	5.54	5.45

Z13_w2_byte	15.52	11.7	9.98	8.62	7.59	6.9	6.78	6.2	5.79	5.44	5.3	5.16	5.26	5.28

Up to 50 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	13.69	10.63	9.72	8.93	7.76	7.49	7.53	6.96	6.92	6.33	6.74	6.66	6.93	7.3
RZ13_w2_byte	12.35	10.1	9.35	8.7	7.5	7.29	7.39	7.11	7.17	6.63	6.95	7.1	7.49	7.98
RZ13_w2_pointer	11.43*	9.44	8.88	8.22	7.18	6.92	7.01	6.6	6.59	6.08	6.41	6.46	6.8	7.19
RZ13_w2_simd3	12.14	9.8	8.9	8.11	6.76	6.36	6.33	5.71*	5.47*	4.8*	4.83*	4.42*	4.22*	4.15*
RZ13_w3_byte	13.51	10.86	9.96	9.04	7.71	7.55	7.62	7.42	7.49	6.73	7.01	6.98	7.16	7.51
RZ13_w3_pointer	11.77	9.62	9.05	8.36	7.26	7.07	7.24	6.96	6.95	6.4	6.66	6.69	6.98	7.37
RZ13_w3_simd1	11.99	9.78	9.32	8.63	7.52	7.36	7.5	7.24	7.29	6.77	7.09	7.23	7.57	8.07
RZ13_w3_simd2	11.67	9.34	8.65*	7.95	6.58	6.32	6.33	5.85	5.7	4.93	4.92	4.54	4.39	4.35
RZ13_w3_simd3	11.86	9.44	8.75	8	6.64	6.36	6.42	5.91	5.73	4.96	4.95	4.54	4.38	4.32
RZ13_w3_simd2_range_cmp	11.84	9.82	9.53	8.9	7.96	7.89	8.08	8.18	8.49	8.23	8.81	9.59	10.56	11.61
RZ13_w3_simd3_range_cmp	11.94	10	9.73	9.29	8.37	8.38	8.7	8.91	9.4	9.33	10.07	11.11	12.43	13.74
RZ13_w4_simd2	11.52	9.24	8.75	7.86	6.62	6.29	6.28	5.9	5.79	5.06	5.06	4.74	4.55	4.53

RZ13_w4_simd3	11.48	9.24*	8.72	7.81*	6.57*	6.29*	6.26*	5.88	5.79	5.04	5.11	4.73	4.59	4.57
FSw8	16.64	13.15	12.37	11.2	10.22	9.99	10.11	10.68	11.75	11.75	12.88	14.14	15.47	16.99
Z13_byte	17.25	12.75	11.24	9.99	8.92	8.5	8.42	7.96	8.02	7.63	8.08	8.36	8.86	9.43
Z13_w2_byte	16.61	12.61	11.04	9.91	8.71	8.3	8.15	7.87	7.87	7.39	7.75	7.88	8.24	8.66

Sigma = 192

Up to 10 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	13.42	10.72	9.47	8.21	6.97	6.73	6.62	5.22	4.97	4.38	4.57	3.69	3.63	3.66
RZ13_w2_byte	12.07	9.94	8.92	7.9	6.51	6.48	6.45	5.23	5.13	4.52	4.44	3.83	3.8	3.84
RZ13_w2_pointer	11.17*	9.38	8.46*	7.63	6.38	6.3	6.13	4.96	4.79	4.24	4.15	3.57	3.56	3.61
RZ13_w2_simd3	12.15	9.93	8.86	7.68	6.34	6.34	6.11	4.95*	4.56*	4.07*	4.13	3.17*	3.04*	2.99*
RZ13_w3_byte	13.13	10.78	9.58	8.22	6.85	6.98	6.98	5.71	5.55	4.99	4.74	3.98	3.97	3.82
RZ13_w3_pointer	11.47	9.66	8.8	7.82	6.4	6.37	6.43	5.27	5.07	4.56	4.35	3.79	3.75	3.75
RZ13_w3_simd1	11.6	9.59	8.8	7.81	6.53	6.62	6.61	5.41	5.34	4.71	4.59	4	3.94	3.93
RZ13_w3_simd2	11.52	9.58	8.68	7.44	6.22	6.38	6.23	5.05	4.97	4.21	4*	3.3	3.16	3.05
RZ13_w3_simd3	11.7	9.63	8.63	7.75	6.28	6.41	6.3	5.08	4.87	4.29	4.07	3.34	3.23	3.11
RZ13_w3_simd2_range_cmp	11.44	9.58	8.69	7.8	6.47	6.59	6.58	5.46	5.34	4.88	4.79	4.34	4.51	4.52

RZ13_w3_simd3_range_cmp	11.35	9.4	8.69	7.7	6.47	6.66	6.58	5.57	5.49	5.15	5.06	4.62	4.9	5.06
RZ13_w4_simd2	11.3	9.33*	8.49	7.56	6.15	6.17*	6.11	5.02	4.79	4.25	4.16	3.43	3.26	3.18
RZ13_w4_simd3	11.27	9.37	8.55	7.36*	6.12*	6.22	6.05*	5.01	4.84	4.21	4.18	3.41	3.28	3.19
FSw8	16.35	12.67	11.51	10.16	8.85	8.64	8.32	7.73	8.44	8.24	8.55	8.56	9.13	9.53
Z13_byte	16.49	12.4	10.48	8.97	7.72	7.52	7.06	5.91	5.66	5.15	5.15	4.53	4.52	4.53
Z13_w2_byte	16.13	12.2	10.41	9.23	7.67	7.4	7.08	6.04	5.86	5.35	5.08	4.59	4.49	4.48

Up to 20 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	13.62	10.81	9.48	8.32	7.74	7.21	6.85	5.84	5.43	5.21	4.95	4.64	4.71	4.59
RZ13_w2_byte	12.37	10.25	9.12	7.97	7.33	7.04	6.74	5.86	5.54	5.42	5.02	4.82	5.03	4.96
RZ13_w2_pointer	11.33*	9.58	8.67*	7.63	6.95	6.63	6.33	5.62	5.2	5	4.68	4.58	4.58	4.58
RZ13_w2_simd3	12.15	10.04	8.83	7.71	6.94	6.56	6.21	5.26*	4.82*	4.54*	4.12*	3.73*	3.49*	3.36*
RZ13_w3_byte	13.4	10.98	9.63	8.32	7.7	7.34	7.07	6.33	6.05	5.69	5.16	4.98	4.97	4.87
RZ13_w3_pointer	11.8	9.8	8.84	7.8	7.26	6.83	6.62	5.88	5.59	5.35	4.95	4.76	4.74	4.77
RZ13_w3_simd1	11.98	9.91	9.09	7.94	7.39	7.02	6.81	6.03	5.79	5.55	5.24	4.99	5.11	5.09
RZ13_w3_simd2	11.75	9.59	8.73	7.64	6.94	6.63	6.24	5.46	5.05	4.74	4.22	3.82	3.69	3.48
RZ13_w3_simd3	12.05	9.8	8.8	7.74	7.09	6.66	6.29	5.52	5.12	4.8	4.28	3.81	3.69	3.51

RZ13_w3_simd2_range_cmp	11.78	9.74	8.92	7.9	7.48	7.31	6.89	6.36	6.15	6.12	5.76	5.88	6.25	6.49
RZ13_w3_simd3_range_cmp	11.73	9.71	9.06	8.06	7.54	7.44	7.12	6.6	6.46	6.41	6.25	6.6	7.04	7.35
RZ13_w4_simd2	11.48	9.55*	8.7	7.49	6.75*	6.47*	6.13	5.46	5.2	4.76	4.28	3.96	3.87	3.64
RZ13_w4_simd3	11.49	9.68	8.77	7.41*	6.86	6.49	6.08*	5.42	5.15	4.88	4.3	4.02	3.84	3.63
FSw8	16.74	13.18	11.93	10.62	9.66	9.26	8.91	8.73	9.23	9.66	9.57	10.21	11.17	11.51
Z13_byte	16.85	12.87	10.75	9.36	8.59	8	7.56	6.56	6.22	6.04	5.78	5.7	5.9	5.8
Z13_w2_byte	16.48	12.74	10.67	9.38	8.4	7.99	7.41	6.69	6.37	6.2	5.76	5.63	5.71	5.6

Up to 50 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	14.52	11.56	10.32	9.18	8.76	8.35	8.61	7.83	7.46	7.17	7.52	7.18	7.63	8.05
RZ13_w2_byte	13.09	10.97	9.84	8.86	8.47	8.52	8.43	7.76	7.51	7.57	7.76	7.62	8.23	8.83
RZ13_w2_pointer	12.13*	10.29	9.37	8.42	8.07	8.04	7.96	7.33	7.07	6.88	7.1	6.95	7.56	7.98
RZ13_w2_simd3	12.96	10.7	9.52	8.39	7.76	7.23*	7.23*	6.45*	6.04*	5.63*	5.67*	4.96*	4.96*	5.06*
RZ13_w3_byte	14.3	11.76	10.49	9.33	8.91	8.49	8.64	8.25	7.86	8.5	7.76	7.48	7.91	8.44
RZ13_w3_pointer	12.57	13.85	9.57	8.64	8.36	7.91	8.45	7.73	7.44	7.16	7.44	7.27	7.87	8.33
RZ13_w3_simd1	12.83	10.79	9.84	8.98	8.55	8.46	8.76	8.1	7.99	7.77	7.88	7.94	8.47	8.96
RZ13_w3_simd2	12.58	10.3	9.34	8.26	7.65	7.35	7.57	6.6	6.19	5.86	5.78	5.21	5.19	5.18

RZ13_w3_simd3	12.84	10.42	9.4	8.35	7.72	7.31	7.41	6.81	6.26	6.06	5.76	5.26	5.21	5.25
RZ13_w3_simd2_range_cmp	12.77	10.77	10.01	9.27	9.1	8.82	9.48	9.13	9.16	9.37	10.08	10.5	11.61	12.89
RZ13_w3_simd3_range_cmp	12.69	10.84	10.34	9.59	9.47	9.56	9.91	10	10.18	10.38	11.19	12.07	13.59	15.12
RZ13_w4_simd2	12.51	10.16	9.36	8.16*	7.71	7.26	7.32	6.77	6.37	6.05	5.91	5.31	5.36	5.46
RZ13_w4_simd3	12.33	10.16*	9.27*	8.18	7.61*	7.34	7.45	6.83	6.34	9.68	6.05	5.38	5.4	5.51
FSw8	17.86	14.2	13.15	12.15	11.4	11.4	14.17	11.84	12.57	13.07	14.41	14.96	16.69	18.43
Z13_byte	17.97	13.67	11.86	10.51	10.03	9.37	9.46	8.78	8.79	8.53	9.02	8.98	9.69	10.37
Z13_w2_byte	17.43	13.54	11.58	10.32	9.68	9.15	9.17	8.75	8.53	8.29	8.72	8.47	8.99	9.54

Sigma = 128

Up to 10 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	14.14	10.47	9.11	7.93	7.35	6.72	6.28	5.25	4.9	4.87	4.34	3.96	3.72	3.67
RZ13_w2_byte	12.83	9.97	8.79	7.5	6.93	6.26	6.11	5.16	4.78	4.73	4.28	3.91	3.8	3.72
RZ13_w2_pointer	11.8*	9.26*	8.23*	7.13*	6.51*	5.96*	5.9*	5*	4.41*	4.94	4.06*	3.63*	3.53	3.5
RZ13_w2_simd3	13.03	10.05	8.89	7.64	7.22	6.21	6.18	5.09	4.84	4.68*	4.12	3.76	3.52*	3.37*
RZ13_w3_byte	13.75	10.76	9.23	7.95	7.26	6.71	6.46	5.77	5.09	5.17	4.73	4.12	3.98	3.91
RZ13_w3_pointer	12.52	9.98	8.68	7.3	6.84	6.16	6.12	5.29	4.94	5.14	4.43	3.98	3.9	3.83

RZ13_w3_simd1	12.38	9.82	8.99	7.64	7.51	6.44	6.47	5.58	5.07	5.05	4.66	4.05	4	3.92
RZ13_w3_simd2	12.35	9.8	8.76	7.28	6.82	6.14	6.06	5.23	4.92	4.84	4.22	3.86	3.65	3.5
RZ13_w3_simd3	12.6	9.84	9.08	7.5	7.05	6.33	6.13	5.41	5.09	4.86	4.28	3.93	3.75	3.59
RZ13_w3_simd2_range_cmp	12.32	9.57	8.68	7.41	6.91	6.28	6.17	5.44	5.05	5.1	4.54	4.1	4.08	4.04
RZ13_w3_simd3_range_cmp	12.05	9.56	8.59	7.31	6.99	6.22	6.22	5.51	4.98	5.03	4.6	4.18	4.18	4.26
RZ13_w4_simd2	12.11	9.49	8.71	7.26	6.76	6.1	6.02	5.26	5	4.95	4.37	3.99	3.83	3.75
RZ13_w4_simd3	12.16	9.31	8.69	7.23	6.82	6.07	6.05	5.26	4.97	4.88	4.33	3.97	3.8	3.75
FSw8	18.02	13.56	12.13	10.64	9.88	8.92	8.48	8.05	8.01	8.55	8.58	8.45	8.96	9.43
Z13_byte	16.98	12.29	10.33	8.94	8.02	7.16	6.92	6.02	5.44	5.21	4.98	4.38	4.27	4.2
Z13_w2_byte	16.48	12.2	10.39	9.05	8.13	7.12	7	6.03	5.58	5.58	5.02	4.54	4.26	4.14

Up to 20 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	14.29	11.53	9.87	9.11	8.04	7.38	7.32	6.34	5.78	5.71	5.65	5.46	5.3	5.35
RZ13_w2_byte	12.82	10.85	9.43	8.62	7.73	7.03	7.03	6.28	5.87	5.77	5.63	5.63	5.68	5.74
RZ13_w2_pointer	11.87 *	10.26*	9*	8.17*	7.35*	6.83	6.74	5.96	5.6	5.38	5.54	5.32	5.32	5.49
RZ13_w2_simd3	12.91	10.81	9.41	8.47	7.5	6.77*	6.68	5.86*	5.29*	5.11*	4.92*	4.68*	4.47*	4.24*
RZ13_w3_byte	13.99	11.56	10.06	9.19	8.25	7.5	7.47	6.77	6.32	6.24	6.08	5.99	5.69	5.75

RZ13_w3_pointer	12.51	10.49	9.28	8.51	7.69	7.14	7.07	6.4	6	5.84	5.71	5.75	5.6	5.74
RZ13_w3_simd1	12.73	10.72	9.6	8.82	7.94	7.32	7.27	6.76	6.27	6.2	6.03	6.02	6.07	6.11
RZ13_w3_simd2	12.44	10.44	9.36	8.4	7.48	6.82	6.8	6	5.51	5.31	5.04	4.82	4.56	4.52
RZ13_w3_simd3	12.81	10.7	9.46	8.55	7.52	7.01	6.85	6.11	5.65	5.41	5.15	4.9	4.71	4.53
RZ13_w3_simd2_range_cmp	12.45	10.68	9.52	8.71	8.12	7.54	7.5	6.94	6.65	6.72	6.79	7.04	7.27	7.72
RZ13_w3_simd3_range_cmp	12.47	10.57	9.5	8.82	8.09	7.6	7.62	7.2	7.02	7.18	7.29	7.8	8.06	8.56
RZ13_w4_simd2	12.46	10.37	9.26	8.4	7.37	6.84	6.76	6.05	5.62	5.47	5.21	5	4.94	4.68
RZ13_w4_simd3	12.31	10.29	9.21	8.31	7.41	6.83	6.62*	6.04	5.6	5.51	5.16	4.96	4.77	4.71
FSw8	17.87	14.32	12.94	11.7	10.82	10.09	9.88	9.52	9.79	10.34	10.93	11.98	12.53	13.26
Z13_byte	17.11	13.21	11.04	9.85	8.88	8.21	7.9	7.14	6.65	6.49	6.48	6.61	6.51	6.76
Z13_w2_byte	16.79	13.12	11.03	9.84	9	8.11	7.8	7.17	6.68	6.55	6.53	6.52	6.34	6.42

Up to 50 matches

PatLength	32	48	64	80	96	112	128	160	192	224	256	320	384	448
RZ13_byte	15.06	12.12	10.74	9.6	9.01	8.64	8.63	8.05	7.78	7.67	7.67	7.79	8.17	8.61
RZ13_w2_byte	13.5	11.47	10.3	9.24	8.58	8.32	8.68	8.18	7.93	7.95	7.88	8.32	8.78	9.34
RZ13_w2_pointer	12.61*	10.8*	9.77*	8.79	8.19	7.93	8.01	7.74	7.47	7.47	7.38	7.66	8.13	8.65
RZ13_w2_simd3	13.58	11.33	10	8.84	8.01	7.58*	7.5*	6.9*	6.51*	6.33*	6.05*	5.83*	5.71*	5.72*
RZ13_w3_byte	14.84	12.24	11.04	9.7	9	8.77	8.76	8.54	8.34	8.18	8.01	8.23	8.61	9.05

RZ13_w3_pointer	13.24	11.2	10.15	9.14	8.51	8.28	9.12	8.15	7.91	7.9	7.79	8.05	8.45	8.94
RZ13_w3_simd1	13.54	11.5	10.64	9.57	8.94	8.79	9.14	8.55	8.5	8.42	8.43	8.77	9.27	9.82
RZ13_w3_simd2	13.15	10.97	9.85	8.77	8	7.6	7.76	7.11	6.8	6.57	6.23	6.07	6.01	6.03
RZ13_w3_simd3	13.48	11.14	10.1	8.9	8.16	7.75	8.07	7.23	6.89	6.66	6.31	6.12	6.09	6.09
RZ13_w3_simd2_range_cmp	13.36	11.4	10.73	9.8	9.36	9.24	9.48	9.62	9.82	10.09	10.41	11.58	12.61	13.78
RZ13_w3_simd3_range_cmp	13.37	11.53	10.9	10.17	9.8	9.8	10.3	10.44	10.78	11.25	11.71	13.18	14.56	15.98
RZ13_w4_simd2	13.22	10.91	10.05	8.79	8.01	7.66	7.58	7.31	7.01	6.71	6.45	6.3	6.24	6.31
RZ13_w4_simd3	13.08	10.9	9.94	8.71*	7.98*	7.66	7.61	7.33	7	6.77	6.48	6.32	6.34	6.36
FSw8	18.82	15.39	14.19	13.13	12.24	12.03	12.55	12.45	13.15	13.81	14.47	16.08	17.72	19.44
Z13_byte	18.04	14.01	12.05	10.81	10.1	9.68	9.95	9.14	9.03	9.01	9.14	9.52	10.19	10.81
Z13_w2_byte	17.7	13.9	12	10.77	9.98	9.45	9.61	9.11	8.89	8.81	8.77	9.11	9.5	10.04

Analysis

These results are in fact quite good. Modifications are able to beat basic algorithms on different values of input parameters and overperform more efficient pointer versions for big pattern lengths combined with large enough alphabet sizes. Overall efficiency of the pointer versions on smaller patterns seems interesting. Even if it's not the target of the work, future investigation of the subject may lead to some pointer-SIMD hybrid able to perform well on small patterns/alphabets, which seems to be an issue as for now.

In general, performance of SIMD algorithms is good. Trying to improve results for small patterns seems reasonable, however this will probably result in losing efficiency for long patterns. And current 20-40% speed increase compared to basic versions already seems good enough to call this experiment successful.

Is SIMD 2 or SIMD 3 better? RZ13_w2_simd3 performs really well but it can't have the SIMD 2 option. For other cases it’s pretty similar. Probably future investigation and more test data will help to distinguish results a bit.

Range comparison is the disappointment of the work. It wasn’t expected to work extremely well for the long patterns, but for short patterns it’s just beaten by the RZ13_w2_pointer. It still outperforms the basic version but now fast enough. Future modification is required to make it a viable option.

Future work

SIMD technology usage allowed us to implement a huge amount of modifications, but it still has some potential for modifying these algorithms. Some of possible ideas

1. Pointer algorithms seem to be efficient. Packing pointers might be faster as well
2. Use AVX (256 bit instructions) for even more efficient comparison
3. Use AVX for more than 4 pointers (w5 version performed well in source [3]).
4. Use MMX (64 bit instructions) for possibly better w2 versions.
5. Possibly use a mix of AVX/SSE/MMX for even better performance.
6. Modify algorithms to make them more efficient on short patterns when using range comparison
7. Investigate mechanics of pointer manipulation on SIMD types for petencial of better extraction.

Conclusion

During this work multiple implementations of different ideas were developed. Most of them were able to beat the runtime of basic algorithms. *RZ13_w4_simd2, RZ13_w4_simd3 and RZ13_w2_simd3* are quite efficient and beat other versions for large patterns and sigmas. Increasing the amount of matches also boosts performance relative to basic algorithms due to efficient comparison methods. We may conclude that using SIMD instructions in these algorithms was very efficient, besides the fact that we require a lot of unpack operations (1 for each window each inner loop iteration to be precise). Algorithms are still yet to be tested on real data, but for now results they show are very convincing. If they prove themself to have comparable performance for practical cases they'll probably have a chance to see real use one day.

As might be seen through this work, using SIMD makes your applications faster at a price of code readability and limitations of the target machine processor. And sometimes this both might be an issue. But in cases when execution time is the main criteria, using SIMD is definitely a way to go.

References

[1] Technology Beats Algorithms (in Exact String Matching)*

Jorma Tarhio, Jan Holub, and Emanuele Giaquinta

[2] The Intel Intrinsics Guide

https://software.intel.com/sites/landingpage/IntrinsicsGuide

[3] Fast exact pattern matching in a bitstream and 256-ary strings

Igor O. Zavadskyi

[4] Backward Nondeterministic Dawg Matching algorithm

Christian Charras - Thierry Lecroq

https://www-igm.univ-mlv.fr/~lecroq/string/bndm.html

[5] NAVARRO G., RAFFINOT M., 1998. A Bit-Parallel Approach to Suffix Automata: Fast Extended String Matching, In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science

[6] 13. A. Hudaib, R. Al-khalid, D. Suleiman, M. A. A. Itriq, and A. Al-Anani: A fast pattern matching algorithm with two sliding windows (tsw). Journal of Computer Science, 4(5), p. 393–401.

[7] Tuning BNDM with q-Grams Branislav Durian

Jan Holub Hannu Peltola Jorma Tarhio

[8] The FJS string matching algorithm

Christopher G.Jennings https://cgjennings.ca/articles/fjs/

[9] The Art of Computer Programming

Donald Knuth

Implementations

Algorithms

Let's take a look on the principal parts of the implementations.

First, common SIMD operations. All of them are defined as defines, because these are

```
// get from packed variables
#define get0_i128(i128) _mm_cvtsi128_si32(i128)
#define get1_i128(i128) _mm_cvtsi128_si32(_mm_srli_si128((i128), 4))
#define get2_i128(i128) _mm_cvtsi128_si32(_mm_srli_si128((i128), 8))
#define get3_i128(i128) _mm_cvtsi128_si32(_mm_srli_si128((i128), 12))

// subtract inside packed variables
#define sub0_i128(val)              \
  packed_positions = _mm_sub_epi32( packed_positions,  _mm_setr_epi32(val, 0, 0, 0));
#define sub1_i128(val)              \
  packed_positions = _mm_sub_epi32( packed_positions,  _mm_setr_epi32(0, val, 0, 0));
#define sub2_i128(val)              \
  packed_positions = _mm_sub_epi32(  packed_positions,   _mm_setr_epi32(0, 0, val, 0));
#define sub3_i128(val)              \
  packed_positions = _mm_sub_epi32( packed_positions,   _mm_setr_epi32(0, 0, 0, val));

#define pack_last_i128(P, mod)    \
  _mm_set_epi8(               \
     mod > 0 ? *(P) : 0,  mod > 1 ? *(P + 1) : 0,    mod > 2 ? *(P + 2) : 0,   mod > 3 ? *(P + 3) : 0,   \
     mod > 4 ? *(P + 4) : 0, mod > 5 ? *(P + 5) : 0,   mod > 6 ? *(P + 6) : 0, mod > 7 ? *(P + 7) : 0,   \
     mod > 8 ? *(P + 8) : 0, mod > 9 ? *(P + 9) : 0,   mod > 10 ? *(P + 10) : 0, mod > 11 ? *(P + 11) : 0, \
     mod > 12 ? *(P + 12) : 0, mod > 13 ? *(P + 13) : 0, mod > 14 ? *(P + 14) : 0,mod > 14 ? *(P + 14) : 0);

// pack char array
#define pack_pattern_i128()                                  \
  for (i = 15; i < m; i += 16) {                             \
     packed_pattern[i / 16] = _mm_loadu_si128((__m128i*)(P + i - 15)); \
  }                                                          \
  packed_pattern[m / 16] = pack_last_i128(P + m - m % 16, m % 16);
```

```
// simple simd check
#define check_i128(pos)                                                                 \
  eq = true;                                                                            \
  for (i = pos + 15; eq && i < pos + m; i += 16) {                                      \
    packed_text = _mm_loadu_si128((__m128i*)&T[i - 15]);                                \
    eq &= (_popcnt32(_mm_movemask_epi8(_mm_cmpeq_epi8(packed_text, packed_pattern[(i - pos) / 16]))) == 16); \
  }                                                                                     \
  if (eq) {                                                                             \
    packed_text = pack_last_i128(T + pos + m - m % 16, m % 16);                         \
    eq &= (_popcnt32(_mm_movemask_epi8(_mm_cmpeq_epi8(packed_text, packed_pattern[(m) / 16]))) == 16);      \
    if (eq) {                                                                           \
      MATCH(pos);                                                                       \
    }                                                                                   \
  }

#define alpha_i128 16
// simd range check
#define SIMDcompare_i128(x, y) \
  _mm_movemask_epi8(_mm_cmpeq_epi8(_mm_loadu_si128(x), _mm_set1_epi8(y)))

#define check_cmp_i128(pos, lim)                                        \
  found = UINT_MAX;                                                     \
  for (_j = 0; _j < m; ++_j) {                                          \
    found = found & SIMDcompare_i128((__m128i*)(T + pos + _j - 15), P[_j]); \
    if (found == 0)                                                     \
      break;                                                            \
  }                                                                     \
  MULTIMATCH(pos - 15, found, lim);


// match report
#define MATCH(pos) ++count
#define MULTIMATCH(pos, key, lim) \
  count += (lim <= pos) ? __builtin_popcount(key) : (lim - (pos) <= 16 ? __builtin_popcount(key >> ((lim) - (pos))) : 0)
```

Now let’s take a look at modifications. Parts changed from previous versions are bald

RZk-w3-SIMD1

```
 for (i = 0; i < m - 1; ++i) {
    z[(*(unsigned short*)(P + i)) & mask] = 0;
  }
  for (i = 0; i < (1 << (k - b)); ++i) {
    z[(i << b) | P[m - 1]] = 0;
  }

  for (i = 0; i < SIGMA; ++i)
    RQS[i] = m + 1;
  for (i = m - 1; i >= 0; --i)
    RQS[P[i]] = i + 1;

  int ndiv3 = n / 3;
  int twondiv3 = 2 * n / 3;

  int pos0 = ndiv3;
  int pos1 = twondiv3;
  int pos2 = n - m;

  __m128i fill_m = _mm_set1_epi32(m - 1);
  __m128i packed_positions = _mm_setr_epi32(pos0,
pos1, pos2, 0);

  while (get0_i128(packed_positions) + m >= 0) {
    packed_positions = _mm_setr_epi32(pos0, pos1,
pos2, 0);
    while (z[word(get0_i128(packed_positions)) &
mask] != 0
           && z[word(get1_i128(packed_positions)) &
mask] != 0
           && z[word(get2_i128(packed_positions)) &
mask] != 0) {
      packed_positions =
_mm_sub_epi32(packed_positions, fill_m);
    }

    pos0 = get0_i128(packed_positions);
    pos1 = get1_i128(packed_positions);
    pos2 = get2_i128(packed_positions);

    if (z[word(pos0 + 1) & mask] == 0 && pos0 >= 0) {
      for (i = 0; i < m && P[i] == T[pos0 + i]; ++i) {
      };
      if (i == m) {
        MATCH(pos0);
      }
      pos0 -= RQS[T[pos0 - 1]];
    } else
      pos0 -= m - 1;

    if (z[word(pos1 + 1) & mask] == 0 && pos1 > ndiv3)
{
      for (i = 0; i < m && P[i] == T[pos1 + i]; ++i) {
      };
      if (i == m) {
        MATCH(pos1);
      }
      pos1 -= RQS[T[pos1 - 1]];
    } else
      pos1 -= m - 1;

    if (z[word(pos2 + 1) & mask] == 0 && pos2 >
twondiv3) {
      for (i = 0; i < m && P[i] == T[pos2 + i]; ++i) {
      };
      if (i == m) {
        MATCH(pos2);
      }
      pos2 -= RQS[T[pos2 - 1]];
    } else
      pos2 -= m - 1;
  }

  while (pos1 >= ndiv3) {
    while (z[word(pos1) & mask] != 0 && z[word(pos2)
& mask] != 0) {
      pos1 -= m;
      pos2 -= m;
    }

    if (z[word(pos1 + 1) & mask] == 0 && pos1 > ndiv3)
{
      for (i = 0; i < m && P[i] == T[pos1 + i]; ++i) {
      };
      if (i == m) {
        MATCH(pos1);
      }
      pos1 -= RQS[T[pos1 - 1]];
    } else
      pos1 -= m - 1;

    if (z[word(pos2 + 1) & mask] == 0 && pos2 >
twondiv3) {
      for (i = 0; i < m && P[i] == T[pos2 + i]; ++i) {
      };
      if (i == m) {
        MATCH(pos2);
      }
      pos2 -= RQS[T[pos2 - 1]];
    } else
      pos2 -= m - 1;
  }

  while (pos2 >= twondiv3) {
    while (z[word(pos2) & mask] != 0) {
      pos2 -= m;
    }
    if (z[word(pos2 + 1) & mask] == 0 && pos2 >
twondiv3) {
      for (i = 0; i < m && P[i] == T[pos2 + i]; ++i) {
      };
      if (i == m) {
        MATCH(pos2);
      }
      pos2 -= RQS[T[pos2 - 1]];
    } else
      pos2 -= m - 1;
  }
```

RZk-w3-SIMD2

For the next implementation only the main loop is present to avoid duplication.

Here most of the checks moved to SIMD instructions

```
 while (get0_i128(packed_positions) + m >= 0) {
     while (z[word(get0_i128(packed_positions)) & mask]
!= 0
         && z[word(get1_i128(packed_positions)) & mask]
!= 0
         && z[word(get2_i128(packed_positions)) & mask]
!= 0) {
         packed_positions =
_mm_sub_epi32(packed_positions, fill_m);
     }

     pos0 = get0_i128(packed_positions);
     pos1 = get1_i128(packed_positions);
     pos2 = get2_i128(packed_positions);

     if (z[word(pos0 + 1) & mask] == 0 && pos0 >= 0) {
        check_i128(pos0);
        sub0_i128(RQS[T[pos0 - 1]]);
     } else
        sub0_i128(mm1);

     if (z[word(pos1 + 1) & mask] == 0 && pos1 > ndiv3)
{
        check_i128(pos1);
        sub1_i128(RQS[T[pos1 - 1]]);
     } else
        sub1_i128(mm1);

     if (z[word(pos2 + 1) & mask] == 0 && pos2 >
twondiv3) {
        check_i128(pos2);
        sub2_i128(RQS[T[pos2 - 1]]);
     } else
        sub2_i128(mm1);
  }
```

```
  pos1 = get1_i128(packed_positions);
  pos2 = get2_i128(packed_positions);

  while (pos1 >= ndiv3) {
     while (z[word(pos1) & mask] != 0 && z[word(pos2)
& mask] != 0) {
        pos1 -= m;
        pos2 -= m;
     }

     if (z[word(pos1 + 1) & mask] == 0 && pos1 > ndiv3)
{
        check_i128(pos1);
        pos1 -= RQS[T[pos1 - 1]];
     } else
        pos1 -= mm1;

     if (z[word(pos2 + 1) & mask] == 0 && pos2 >
twondiv3) {
        check_i128(pos2);
        pos2 -= RQS[T[pos2 - 1]];
     } else
        pos2 -= mm1;
  }

  while (pos2 >= twondiv3) {
     while (z[word(pos2) & mask] != 0) {
        pos2 -= m;
     }
     if (z[word(pos2 + 1) & mask] == 0 && pos2 >
twondiv3) {
        check_i128(pos2);
        pos2 -= RQS[T[pos2 - 1]];
     } else
        pos2 -= mm1;
```

SIMD 3. Almost everything is moved to use SIMD. Also the idea of permanently shifting packed positions when the first variable is no longer usable. Huge change, but more noticeable for w4 versions (it's code even bigger, so w3 will do here).

```
while (get0_i128(packed_positions) + m >= 0) {
    while (z[word(get0_i128(packed_positions)) & mask] != 0
      && z[word(get1_i128(packed_positions)) & mask] != 0
      && z[word(get2_i128(packed_positions)) & mask] != 0) {
      packed_positions = _mm_sub_epi32(packed_positions,
fill_m);
    }

    pos0 = get0_i128(packed_positions);
    pos1 = get1_i128(packed_positions);
    pos2 = get2_i128(packed_positions);

    if (z[word(pos0 + 1) & mask] == 0 && pos0 >= 0) {
      check_i128(pos0);
      sub0_i128(RQS[T[pos0 - 1]]);
    } else
      sub0_i128(mm1);

    if (z[word(pos1 + 1) & mask] == 0 && pos1 > ndiv3) {
      check_i128(pos1);
      sub1_i128(RQS[T[pos1 - 1]]);
    } else
      sub1_i128(mm1);

    if (z[word(pos2 + 1) & mask] == 0 && pos2 > twondiv3) {
      check_i128(pos2);
      sub2_i128(RQS[T[pos2 - 1]]);
    } else
      sub2_i128(mm1);
  }

  packed_positions = _mm_srli_si128(packed_positions, 4);

  while (get0_i128(packed_positions) >= ndiv3) {
    while (z[word(get0_i128(packed_positions)) & mask] != 0
&& z[word(get1_i128(packed_positions)) & mask] != 0) {
      packed_positions = _mm_sub_epi32(packed_positions,
fill_m);
    }

    pos1 = get0_i128(packed_positions);
    pos2 = get1_i128(packed_positions);

    if (z[word(pos1 + 1) & mask] == 0 && pos1 > ndiv3) {
      check_i128(pos1);
      sub0_i128(RQS[T[pos1 - 1]]);
    } else
      sub0_i128(mm1);

    if (z[word(pos2 + 1) & mask] == 0 && pos2 > twondiv3) {
      check_i128(pos2);
      sub1_i128(RQS[T[pos2 - 1]]);
    } else
      sub1_i128(mm1);
  }

  pos2 = get1_i128(packed_positions);

  while (pos2 >= twondiv3) {
    while (z[word(pos2) & mask] != 0) {
      pos2 -= m;
    }
    if (z[word(pos2 + 1) & mask] == 0 && pos2 > twondiv3) {
      check_i128(pos2);
      pos2 -= RQS[T[pos2 - 1]];
    } else
      pos2 -= mm1;
  }
```

Range cmp only difference is that each expression Is replaced with

```
check_i128(pos0);
sub0_i128(RQS[T[pos0 - 1]]);
```

```
check_cmp_i128(pos0, 0);
sub0_i128(alpha_i128)
```

Testing

```
for (QString& mstr : patternLengths) {
      m = mstr.toInt();

      progressBar->mProgress->setValue(progressBar->mProgress->value() + 1);
      progressBar->mProgress->setFormat("M progress " + QString::number(m));

      memset(execTime, 0, sizeof(float) * algorithms.size());

      for (ig = 0; ig < OUTER_ITER; ig++) {

         for (int i = 0; i < m; i++) {
            P[i] = (rand() + glob) % SIGMA;
            glob = (glob * 123 + 3157) % 893;
         }

         memcpy(TN - m, P, m);
         memcpy(TN + N, P, m);

         for (ii = 0; ii < INNER_ITER; ii++) {

            int patpos = get_rand_int(N - m - 2);
            memcpy(TN + patpos, P, m);

            CLEAR_MATCHES;

            std::random_shuffle(ids, ids + algorithms.size());

            for (i = 0; i < algorithms.size(); ++i) {
               int id_i = ids[i];
               int id_0 = ids[0];

               matches[id_i] = algorithms[id_i]->search(P, (int)m, TN, (int)N, &execTime[id_i]);

               if (matches[id_i] != matches[id_0]) {
                  cout << "Result for algo " << algorithms[id_i]->name << " is " << matches[id_i]
                      << " while for algo " + algorithms[id_0]->name + " is " + to_string(matches[id_0])
                      << "\nM = " << m << ", N = " << N << ", SIGMA = " << SIGMA << std::endl;

                  FIND_DIFF_MATCHES
               }

            }
         }
      }
   }
```