Optimize string matching algorithm with sse2

Zhenghua Dai, Xingang Yu, Liang You, Ran Wei

ABSTRACT

String matching is widely used in many fields of computer sciences, such as information retrieval, bioinformatics, network intrusion detection and so on. However, for programmers, the strstr function is rather inflexible. It is necessary to design a faster string matching algorithm. In this paper, an implementation of strstr function optimized with SSE2 is presented. (How to implement it, is should be illustrated by a sentence.) To our best knowledge, this is the first work that has been done on the speed up of the exact string matching algorithm with SSE2 instruction exts. The source code can be obtained by checking out from the SVN repository on: http://strstrsse.googlecode.com/svn/trunk/. We evaluate the presented algorithm elaborately. The results show that the presented algorithm is 4 times faster than BM algorithm, and 4 times faster than strstr in glibc when the pattern is not very long. In best case scenario, our algorithm is 10 times faster than BM algorithm.

Categories and Subject

Descriptions

[**Algorithms**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms

Algorithms

Keywords

String matching, strstr, SSE2, SIMD.

1. INTRODUCTION

String matching is to find the first or all the occurrences of a string (called pattern) in a given text. It is widely used in many fields of computer science industry, such as information retrieval, bioinformatics, network intrusion detection and so on. To a programmer, the strstr function is rather a completely strange term. Thus, designing faster string matching algorithm is an exploration of the research this such area. In this paper we will focus on the single pattern string matching algorithm.

In the last several decades, the key point of the development of string matching algorithm is how to minimize the numbers of comparisons. First, let us review the development of single pattern string matching algorithms. The first algorithm that made its appearance is the brute-force algorithm, its complexity is (described in big-O notation) O(mn), where m is the length of the pattern and n is the length of the text. The other two algorithms appeared in 1977, they all claimed that themselves were fast, and actually, they were. They opened the door to speed up the string matching. KMP[1] is an algorithm based on automaton, it's complexity was reduced to O(n). BM[2] is an algorithm based jump table, its complexity was reduced to O(n/m). After 15 years, another automaton based algorithm, shift-or[3] came to its existence. The third type of algorithm we have to mention is the implantation of strstr in glibc. It does not reduce the number of comparisons, but to reduce the overhead of the memory access. In 2009, Intel commit there strstr with sse4.2. But most of the current CPUs only have sse2 instead of SSE4.2.

Now we can see that the string matching algorithm can be speeded up by reducing the number of comparisons with automaton or jump table, or reducing the overhead of memory access. It is obviously that it is hard to speed up the automaton-based algorithm and jump-table-based algorithm with SSE2. And speeding up high-tuning brute-force string matching algorithm is also challenging. To the best of our knowledge, no works have tackled the problem of optimize string matching with SSE2.

Our experiments showed that our algorithm was 2 to 10 times faster than BM algorithm and strstr of glibc.

The remainder of this paper is organized as follows: The second section will be the description of our algorithm. The third section will provide the results of our experiments. And the fourth section would give some conclusions.

2. Optimize string matching with SSE2

In this section, we will present our string matching algorithm, strstrsse. The string matching is to find the first or all the occurrence of a pattern in a given text. In this paper, we mainly discuss how to find all the occurrence of one single pattern.

For readability purposes, we will define some words here; we call a 32 byte long memory block that starts from aligned address a *block*.

For the convenience of describing our algorithm, let us suppose that A stands for the first char of the pattern, B the second, and C the third. The length of pattern is m, and the length of the text is n.

The key thinking of our algorithm is to find out candidates quickly and then check the candidates byte by byte.

The following is the main frame of the algorithm:

1. Process the unaligned data
2. For each block in text
3. If there is candidates in this block
4. Search the candidates' position byte by byte.
5. End if
6. End for

2.1 Unaligned data

The start address of text maybe unaligned, so we should search the unaligned bytes byte by byte for the candidates until the address is aligned by 16 bytes. In the worst case, there is 15 bytes before the aligned address.

2.2 What are the candidates?

There are three types of candidates: the first is the one starts with abc, the second is the one that starts with a, and located at the end

of a block, the third is the one that starts with ab and located at the end of a block.

2.3 How to pick up the candidates

Our aim of this stage is to find out if there is abc in the current 32 bytes. That is to say, if there is no abc in current 32 bytes, and the last byte is not a and the last two bytes are not ab, then we can skip these 32 bytes safely. Otherwise, we pick up a candidate.

To find out the first type of candidates, we use a 32-bit unsigned int as flag. Firstly, we compare the block with A and get a 32-bit flag called flagA, if the i-th bit of flagA is 1, the corresponding address is A. For example, if the current block is "UUUUGGGGKKKKABCDDCBAFFFFWWWWRRRR", the flagA would be 0000 0000 0000 1000 0001 0000 0000 0000, Secondly, we can get a 32-bit flag flagB telling us whether there is B or not. Thirdly we can get a 32-bit flag flagC telling us whether there is C or not. Then we can get the flag by the follow formula:

$$flag = (flagA) \mid (flagB \gg 1) \mid (flagC \gg 2)$$

in this example, the flag is 0000 0000 0000 0000 0001 0000 0000 0000. From the flag, we can know whether there is ABC and where .

To find out the later two types of candidates, we first check if the first and the second character of the next block is C. If not, there is no candidate of the later two types in the current block. Otherwise, we check whether the last one is A and whether the last two is AB.

2.4 Search at the candidates position.

We handle the three types of candidates in three different ways.

A 32-bit flag will show if there are any first type candidates in the current block. If the flag is not zero, then there is, the bits in the flag tell you whether the corresponding address in the block is a candidate or not. That is to say, if the i-th bit in the flag is 1, the i-th address in current block is a candidate. We can use bsf instruction to get the 1-bit of the flag. Then we can search for the pattern byte by byte at the candidate position.

When we find out the later two types of candidates, the three characters at the beginning of the pattern have been found. Then we could compare with the fourth character at the candidate position. For example:

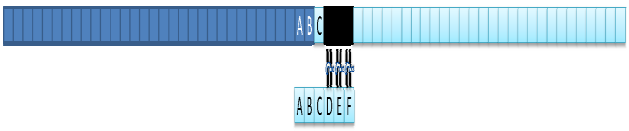

Figure 1. Searching at the second type of candidate position. At the beginning of the next block, there is a C; and on the end of the previous block, there is a AB. So this is the second type of candidate. Then we should compare the left characters of pattern with the text marked in black at the next block.

3. Experiments

The alphabet can be divided into two groups, one is big alphabet set, such as ASCII, the other is small alphabet such as genome. So we will test our algorithm on the two alphabets.

We tested our algorithm on Intel core 2 duo CPU. And we compared our algorithm with the other five famous ones, bm[2], bmh[4], strstr in glibc ,kmp[1] and shiftor[3].

3.1 Test on ASCII alphabet

The text is the novel *Gone with the wind* which is 2.3Mbytes long. To test our algorithm, we should choose some words that have different length, and the words should have different number of occurrences in the text. Finally we choose 7 words: 'the' , 'The', 'love', 'would', 'Atlanta', 'Scarlett' and 'wholeheartedness'. The lengths of them are 3,3,4,5,7, 8 and 11. The numbers of occurrences of them in the text are 18315， 1292， 288，218, 488, 6 and 1.

We can see that, our algorithm is 2 to10 times faster than strstr of glibc.

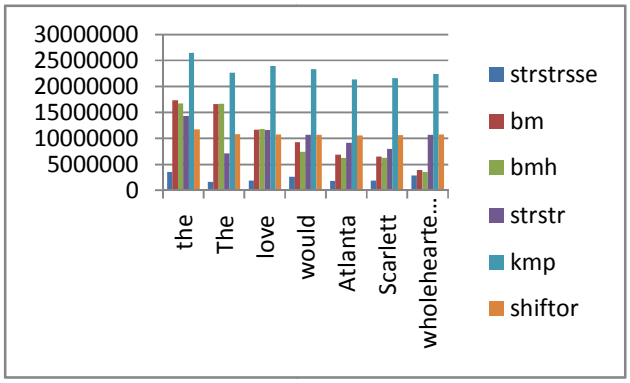

Figure 1. Result on ASCII alphabet, the text is Gone with the wind. The x-axis is different patterns. The y-axis is the number of cycles used when searching the pattern in the text takes.

3.2 Test on genome

To test the performance on small alphabet, we use the chr1 as the text. Chr1 is the first human genome. The character set is {A G C T}. We search 6 different promoter in the chr1 genome.

The Figure 2 shows us that our algorithm is faster than the other 5.

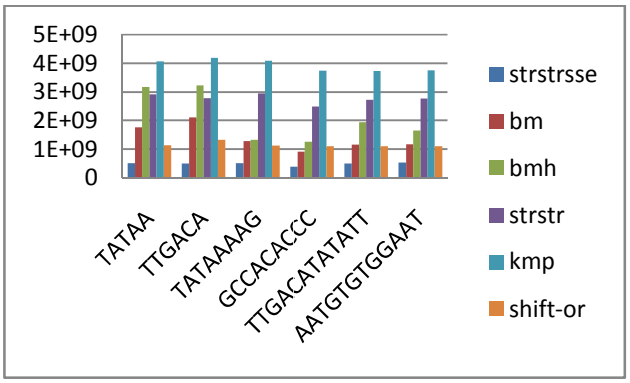

Figure 2. Result on genome. The text is the first human genome chr1. The pattern is 6 different promoters.

3.3 Tests on very long patterns

The average complexity of bm is O(n/m), so the performance get better and better with the increase of the length of the pattern. The automaton based algorithms perform stably whatever the length of the pattern is. In this test, we want to know when the bm will beat the strstrsse. The text is the same with test 3.1.

From Figure 3, we can see that our algorithm is much better than the others when the length of pattern is smaller than 16. When the length of pattern is greater than 24, the performance of bm and bmh converge to our algorithm.

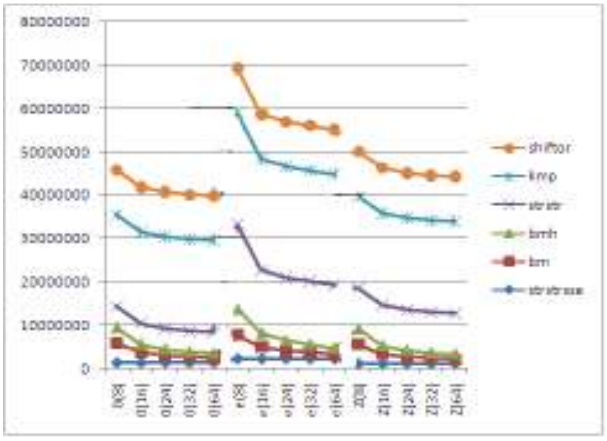

Figure 3. Test on very long patterns. 0(8) stands for 8 continuous 0s, 00000000. And other pattern is the same. The character 0 does not exist in the text. The frequency of 'e' is the highest, the frequency of 'z' is the lowest except for characters having 0 frequency. The text is '*Gone with the wind'*.

4. Conclusion

In this paper, we introduce our string matching algorithm strstrsse, which is optimized with SSE2. It is always faster than strstr in glibc. And when the pattern length is smaller than 16 bytes, it is faster than bm algorithm and bm varieties, that is to say, our algorithm is the fastest. Most of the current CPUs have SSE2, so our algorithm could be widely used.

5. REFERENCES

[1] KNUTH D.E., MORRIS (Jr) J.H., PRATT V.R., 1977, Fast pattern matching in strings, SIAM Journal on Computing 6(1):323-350.

[2] BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. *Communications of the ACM*. 20:762-772.

[3] BAEZA-YATES, R.A., GONNET, G.H., 1992, A new approach to text searching, Communications of the ACM . 35(10):74-82.

[4] HORSPOOL R.N., 1980, Practical fast searching in strings, *Software - Practice & Experience*, 10(6):501-506.