Accelerate String Searching Algorithm with GPU Using CUDA

Yutong Han
School of Electronic and
Information ElectricalEngineering
Shanghai Jiaotong University
Email: ivyhan2013@sjtu.edu.cn

***Abstract*—In computer science, string searching algorithms, sometimes called string matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.**
NVIDIA provided a parallel computing platform which can program by CUDA. And using these technique string searching problem can be accelerated.

***Index Terms*—String searching,algorithm,GPU,CUDA,parallel programming**

I. Introduction

In this project, I present an approach for elevating the performance of this algorithm via GPU (Graphic Processing Unit). With the rapid development of Graphics Processing Unit to many-core multiprocessors, it shows great potential in many applications and high performance computing.

Several well-known string matching algorithms, such as Boyer-Moore (BM)algorithm, RK algorithm, and Knuth-Morris-Pratt (KMP) algorithm, have been proposed to improve the performance of traditional algorithm (brute force algorithm). In this project,The Boyer-Moore-Horspool algorithm was chosen since it involves sequential accesses to the global memory, which can cut down the overhead of memory access as well as this algorithm is more effective than some other string match algorithm.[1]

I study the use of parallel computation on GPUs for accelerating string matching. A direct implementation of parallel computation on GPUs is to divide an input stream into multiple segments, each of which is , we study the use of parallel computation on GPUs for accelerating string matching. A direct implementation of parallel computation on GPUs is to divide an input stream into multiple segments, each of which is processed by a parallel thread for string matching. For example in Fig. 1(a), using a single thread to find the pattern AB takes 24 cycles. If we divide an input stream into four segments and allocate each segment a thread to find the pattern AB simultaneously, the fourth thread only takes six cycles to find the same pattern as shown in Fig. 1(b). However, the direct implementation of dividing an input stream on GPUs cannot detect a pattern occurring in the boundary of adjacent segments. We call the new problem as the boundary detection problem. For example, in Fig. 2, the pattern AB occurs in the boundary of segments 3 and 4 and cannot be identified by threads 3 and 4.And the boundary problem can be resovled by having threads to process overlapped computation on the boundary(as shown in Fig 3)[6]

Fig. 1. Single vs. multiple thread approach

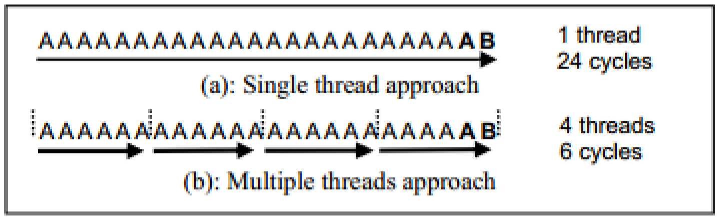


Fig. 2. . Boundary detection problem that the pattern AB cannot be identified by Thread 3 and 4

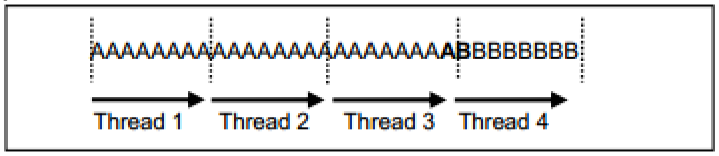
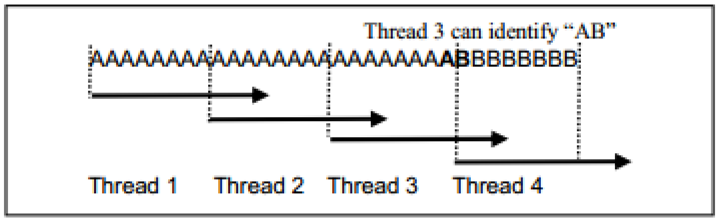


Fig. 3. Every thread scans across the boundary to resolve the boundary detection problem.



II. Related work

A. BMH algorithm

The string matching problem can be defined as: let be an alphabet, given a string array S[N] and a pattern array P[M] where M is the length of the pattern and N is the length of the text, report all locations i in S where is an occurrence of

P, S[i + k]=P[k] for MN. Algorithm 1 illustrates serial BMH string match algorithm. The Boyer-Moore-Horspool algorithm [1] uses a window of size M to search the text from right to left and an array (known as the bad-character shift) of the rightmost character of the window to skip character positions when a mismatch occurs.

Algorithm 1 Boyer Moore Horspool algorithm

```
table[0...255] = N;
for i=0 to N do
  table [P[i]]=N-i;
  while i≤N do
    k =i ;
    for j=m to 1 do
      if P[j]!=T[k] then
        break;
      else
        k–;j–;
      end if
    end for
    if j=-1 then
      match;
      i++;
    else
      mismatch;
      i+=table[T[i]];
    end if
  end while
end for
```

B. *Properties of my GPU*

The properties of my GPU is shown in TABLE 1.

TABLE I
SOME PROPERTIES OF TEST GPU

Name	GeForce GTX 880M
Clock Rate	993000
Maximum Block Dimensions	{1024, 1024, 64}
Maximum Grid Dimensions	{2147483647, 65535, 65535}
Maximum Threads Per Block	1024
Maximum Shared Memory Per Block	49152
Total Constant Memory	65536
Warp Size	32
Maximum Pitch	2147483647
Maximum Registers Per Block	65536
Texture Alignment	512

III. IMPLEMENTATION OF BMH ALGORITHM ON GPU USING CUDA

A. *Kernel of BMH algorithm on GPU*

Since the function with __ *global*__ qualifier must be a *void* function. So the input and the output must transfer with a pointer.

First, before doing pattern searching, BMH algorithm led us to make *BadCharTable* , and I do this in host. Then I copy this table together with pattern and string from host to device.
After that I want to divide string into several segment, and each thread deal with one segment.

The length of each segment is $stringlen/N + patternlen$,which N is number of the thread, except last segment which is $stringlen/N + stringlen\%N$.

Then using the BMH algorithm for each thread, using an integer array to record the locations where an occurrence of pattern happens. Then using a pointer to transfer this array back to the host.

B. *Strategy of setting block number and thread number*

I have already give my GPU propertites at introduce section.
There are 1024 threads per blocks and the maximum block dimensions is equal to *1024,1024,64*.And we can use several blocks and threads to implement parallel.So the question is how we choose the number of the threads and blocks? I have an experiment on this(shown in table 2).

TABLE II
RUNNING TIME OF DIFFERENT TEST CASE WITH DIFFERENT THREADS AND BLOCKS CHOICE

	test 1	test 2	test 3
string length	34873628	34873628	34873628
pattern length	1751	3	3999
256*16	167	177	367
512*4	170	195	390
256*8	173	189	391
128*16	178	178	383
64*32	171	179	586
32*64	168	179	588
256*4	172	189	391
128*8	173	191	405
64*16	171	218	609
32*32	171	207	922
16*64	170	200	964
16*32	171	261	2746

After having look at this table, I find that when *number of blocks*number of threads* increase the running decrease. When the number of threads more than 16, the running time increase roughly at test case 3.

So I use 256 blocks and each blocks contains 16 threads.

The reason why *when the number of threads more than 16, the running time increase roughly at test case 3*,I think is the *warp size*of my GPU is 32,the half warp size is 16.

The warp is the basic unit of scheduling and branch divergence,the half warp is the basic unit of memory access.[3][4]And if there are too many threads in a block with no dectection and protection, if-branch and memory access may bring heavy latency then cripple the performance of the parallel implementation.[5]

IV. Analysis of the algorithm complexity

The preprocessing time of the BMH algorithm is O(M+K),which K is the size of an alphabet.And the preprocessing is done in CPU.

The algorithm performs best with long pattern, when it consistently hits a bad character at or near the final byte of the current position in the string and the final byte of the pattern does not occur elsewhere within the pattern.And the best case of the BMH algorithm is O(N/M),which is same as the best case of BM algorithm.

The worst case behavior happens when the bad character skip is consistently low (with the lower limit of 1 byte movement) and a large portion of the pattern matches the pattern. The bad character skip is only low, on a partial match, when the final character of the pattern also occurs elsewhere within the needle, with 1 byte movement happening when the same byte is in both of the last two positions.

The worst case is significantly higher than for the BoyerMoore string search algorithm, which is even same as navie algorithm(O(M*N)).But this case is hard to achieve, and this BMH algorithm is more cache friendly than BM algorithm. Since the preprocessing work is done in CPU then copy to GPU device, the memory copy and the memory access in device will bring heavy latency. This is the reason why I choose BMH algorithm rather than BM algorithm.[8]

The pattern matching work is done in parallel, so the searching time should be divided by the number of threads running at same time. However, since the *overlapping* issue and another GPU performance, algorithm can not achieve at that time.

Space complexity is arised by auxiliary memory, *BadCharTable*,which is O(K).

V. Conclusion

In this paper, parallel implementation of Boyer-Moore-Horspool algorithm is presented using the CUDA toolkit. Comparing serial implementation, parallel implementation is much faster.By having this experiment,I have learn the knowledge of GPU more deeply.I also find some futher optimization of the parallel implementation.

Shared memory can be used to decrease the latency of memory access. Some dectection of *bank conflict* can aviod conflict accelerating the algorithm.[7]

In order to resolve the *boundary detection* problem ,we can use AhoCorasick algorithm rather than BMH algorithm. [6]

Acknowledgment

The authors would like to thank the teaching assistant of this course for reading and checking this report.

References

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

[2] H. Kopka and P. W. Daly, *String Matching on a multicore GPU using CUDA*.

[3] NVIDIA CUDA manual reference, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[4] cuda-the-basics, http://supercomputingblog.com/cuda/cuda-the-basics/

[5] Junrui Zhou* and Hong An and Xiaomei Li and Min Xu and Wei Zhou *Implementation of String Match Algorithm BMH on GPU Using CUDA*

[6] Cheng-Hung Lin* and Sheng-Yu Tsai** and Chen-Hsiung Liu** and Shih-Chieh Chang** and Jyuo-Min Shyu** *Accelerating String Matching Using Multi-threaded Algorithm on GPU*

[7] JCharalampos S. Kouzinopoulos and Konstantinos G. Margaritis *String Matching on a multicore GPU using CUDA*

[8] BoyerMooreHorspool algorithm, https://en.wikipedia.org/wiki/Boyer\%E2\%80\%93Moore\%E2\%80\%93Horspool_algorithm