Parallel String Matching Algorithms and Implementation in CUDA

Shikun (Jason) Wang

May 11, 2018

1 Introduction

This project explores the parallel algorithms in string matching and implementation in CUDA on GPU. String matching problem, as formally defined following, involves finding the positions where the substring of the text matches with the given pattern.

2 String matching problem

We define $Y[1:n]$ to be a string consisted of n characters, and $Y[i:j]$ to be the substring of Y from i^{th} character in Y to j^{th} character in Y. So, given a string $Y[1:n]$, and a pattern $P[1:m]$, we want to output a *match* vector with size n such that the i^{th} element of *match* is 1 if the pattern appears in the string starting from index i, and 0 otherwise. Figure 1 gives a visual representation of the string matching problem.

3 String matching algorithms

The brute force algorithm is to compare $P[1:m]$ with $Y[i:i+m-1]$ for every position i, where $1 \leq i \leq n-m+1$ where the running time is O(mn)if it is executed serially. Under PRAM (parallel random-access machine) model, we can parallelize the procedure of comparing at each position i in the text, which theoretically make the running time O(1), but we still need O(mn)

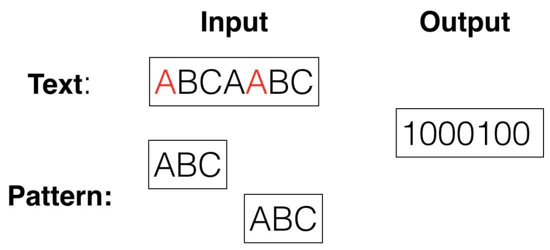

Figure 1: The red character represent the starting position where the pattern matches with the text.

operations. This algorithm has also been implemented on CUDA (See Code 4.1)We need to improve our algorithm to reduce the number of operations.

3.1 Duel Algorithm

Let Y be a string of length m and period p. Let $\pi(Y) = min(p, \lceil m/2 \rceil)$. We first define a function

$$\phi(i) = \begin{cases} 0 & n = 1 \\ k & \text{otherwise} \end{cases}$$

, where $1 \leq i \leq \pi(Y)$ and k is some index for which $Y(k) \neq Y(i+k-1)$. The values of the function associated with each i can be constructed into a WITNESS arry $(1 : \pi(Y))$ where each entry i's value denote the index k of Y that $Y(k) \neq Y(i+k-1)$. Using this WITNESS array, we can efficiently eliminate one out of two position in the text that the pattern might occur. The algorithm described in the book is defined in Algorithm 1[1]:

3.2 Non-periodic case

For a pattern of length m and period p, it is defined as **non-peridoic** if $p > m/2$. So for this pattern string, its WITNESS array is of size $\lceil m/2 \rceil$. We can split our input string into blocks of $\lceil m/2 \rceil$, and since the pattern is non-periodic, there is only one position among all the indexes in one block that is potentially the matching position. All we want to do is to eliminate all the position but one by apply $Duel(i, j)$ in parallel, and then run the

Algorithm 1: Duel algorithm[1]

```
Data: (1) Text string T of length n; (2) WITNESS array of the
      pattern string of length m; (3) two index i, j such that
      $j - i < \pi(Y)$
Result: return one index as potential candicate for matching
begin;
$k := WITNESS(j - i + 1)$;
if $T(j + k - 1) \neq T(k)$ then
    return i
else
    return j
end
```

brute force algorithm on the remaining candidates.

The algorithm given in Jaja's book is based on theoretical PRAM model. The pseudo code of the algorithm is given in algorithm 2 [1]. Proved by Jaja, this algorithm's running time is $O(\log m)$, and number of operations is $O(n)$.

Algorithm 2: Non-periodic case[1]

```
Data: (1) Text string T of length n and pattern string of size m; (2)
      WITNESS array of $m/2$
Result: MATCH array
begin;
1. Split T into $\lceil \frac{2n}{m} \rceil$ blocks $T_i$;
2. For each blocks, construct a binary tree where each node contains
   one index position in that block, then use $Duel(i, j)$ eliminate all
   index but one;
3. For each candidate index in the block, perform the brute force
   algorithm;
```

4 Implementation

4.1 Brute-Force Algorithm

This is my implementation of the brute-force algorithm, where each thread is responsible for checking each position of the text. Since it is only checking but not modifying the text, we don't need to perform any synchronization. Also, a sightly modified version of this is used to the perform the stage 3 of the algorithm 2.

```
__global__ void brute_force(char *text, char *pattern, int *match, int pattern_size, int text_size){
/*get the absolute pid*/
int pid = threadIdx.x + blockIdx.x*blockDim.x;
if (pid <= text_size - pattern_size){
int flag = 1;
for (int i = 0; i < pattern_size; i++){
if (text[pid+i] != pattern[i]){
//if this is not a match; mark 0
flag = 0;
}
}
match[pid] = flag;
}
}

```

4.2 Implementation of Algorithm 2 using GPU Global Memory

My first implementation of Algorithm 2 involves two kernel calls separated by *cudaDeviceSync*(). The first kernel is shown in the following. In order to perform the binary tree reduction, An array *ouput* is produced to store intermediate result in each round of reduction. Firstly, index of each element in the text is copied to the *output* array to start the reduction.

```
__global__ void nonperiodic_version(char *text, char *pattern, int *output, int *witness_array, int blocksize){
int id = threadIdx.x + blockDim.x*blockIdx.x;
int size = blocksize;
int s = (blocksize + 2 - 1) / 2; //step size
//read index to buffer
```

```
 6  output[id] = id;
 7  __syncthreads();
 8  //perform binary tree reduction until step size is 1
 9
10
```

Next, binary reduction process is performed with consecutive memory access. Line 6 to Line 13 is the implementation of the algorithm 1. Basically, it compares the index j +k position in text with k position in pattern, where index k is obtained from the witness array define above. Then if it matches, index j should go to next round, otherwise, index i should. SynchronizeThread() is called in each round of reduction to make sure every thread is on the same pace. After each round, step size is reduced to $\lceil \frac{s}{2} \rceil$ and block size is reduced to $\lceil \frac{size}{2} \rceil$

```
 1  while(s>=2){
 2  int starting_pos = id;
 3  if (threadIdx.x<size/2){
 4  if (starting_pos + s <size){ //if not, the value in output
   is not change
 5  //get the two index i, j
 6  int j = output[starting_pos+s];
 7  int i = output[starting_pos];
 8  int k = witness_array[j-i]; //get index k from witness array
 9  if (text[output[j + k ]] == pattern[k]){ //duel algorithm
   comparision
10  output[starting_pos] = j;
11  }
12  }
13  }
14  __syncthreads();
15  s = (s+2-1)/2; //decrement step size
16  size = (size + 2 - 1) / 2; //decrement size
17  }
18
```

The last step of the reduction is performed by thread 0 and store the index candidate to the output[0], which is used to perform the brute fore algorithm later on

```
 1  //only two index remaining, thread 0 perform duel algorithm
   and store final value in the output
 2  if (threadIdx.x ==0){
 3  int starting_pos = id;
```

```
    int j = output[starting_pos+s];
    int i = output[starting_pos];
    int k = witness_array[j-i];
    if (text[output[j + k ]] == pattern[k]){
    output[starting_pos] = output[starting_pos+s];
    }
    }

```

One trade-off I had made is the size of thread block. The number of threads in a block is equal to $\lceil \frac{m}{2} \rceil$, where m is the length of the pattern string. Hence the size of block depends of the length of the pattern string, so it is possible that the block size is smaller than the size of a warp, which affects the efficiency.

Lastly, the number of blocks created is potentially large if the pattern is small but the text is large. Even through the number of blocks is not limited by number of cores available in GPU, the scheduling of blocks will makes the system slow if the amount of thread blocks is large.

These two drawbacks are caused by the limitation of the hardware, but theoretically, if the system is PRAM, the algorithm would be good.

4.3 Implementation of Algorithm 2 using GPU Shared Memory

Then, I tried to implemented the same Algorithm 2 with shared memory in each block. Instead of creating a huge *output* array to hold the intermediate results during the binary tree reduction, one dimensional share memory space is allocated in each block. Since there is no interaction between blocks, there is no synchronization problem with using the shared memory. The kernel code is shown as following:

```
    __global__ void
   nonperiodic_version_binary_tree_shared_memory(char *text,
   char *pattern, int *blockoutput, int *witness_array, int
   blocksize){
    int pid = threadIdx.x;
    int id = threadIdx.x + blockDim.x*blockIdx.x;
    //int size = blocksize;
    int s = (blocksize + 2 - 1) / 2;
    int size = blocksize;
    __shared__ int buffer[1024];
    buffer[pid] = id;
    __syncthreads();
    while(s >= 2){
    if (threadIdx.x <= size/2){
    int starting_pos = pid;
    if (starting_pos + s >= size){
    buffer[pid] = id;
    }else{
    int j = buffer[starting_pos+s];
    int i = buffer[starting_pos];
    int k = witness_array[j-i];
    if (text[j+k] != pattern[k]){
    buffer[starting_pos] = i;
    }else{
    buffer[starting_pos] = j;
    }
    }
    }
    __syncthreads();
    s = (s+2-1)/2;
    size = (size + 2 - 1) / 2;
    }
    }

```

Space of size 1024 is allocated in shared memory to first store each index. Next, *pid* is used as starting position to perform the binary reduction through shared memory. The implementation is similar to the one with global memory.

5 Serial String Matching Algorithm (KMP) and implementation

The commonly used serial string matching is Knuth Morris Pratt (KMP) algorithm. It uses similar idea of skipping some positions in the text when comparing the text array with pattern array. The algorithm can be illustrated by Jaja in his book(reference). There are two stage in KMP algorithm. The first one is pre-process the pattern and build the failure function, which has length of the pattern and is defined as $F(j) = j - D(j)$, where $D(j)$ is the period of P(1:j-1). The code of building failure function is shown below:

```
 void failure_function_cpu(char *pattern, int *
failure_function, int pattern_size){
 //first term is always zero
 failure_function[0] = 0;
 int k = 1;//index of failure functino
 int j = 0;//size of period
 while ( k < pattern_size){
 if (pattern[k] == pattern[j]){
 j ++; //increment
 failure_function[k] = j; set to be j
 k ++; //increment k
 }else{
 if (j !=0){
 k = failure_function[k-1];
 }else{//at the begining of the period
 failure_function[k] =0;
 k++;
 }
 }
 }
 }

```

A process of linear looping through the pattern is used to keep track of the period of P(1:j-1), and to set the corresponding entry of failure function according to the Jaja's algorithm [1]. The running time of this would be O(m).

The next stage is to use this failure function and match pattern with the text by skipping some index position as indicated in Jaja's algorithm [1]. The idea is to skip some length of prefix of the pattern that we have checked matched with the text previously. The code is shown following.

```
 void serial_string_matching_KMP(char *text, char *pattern,
int pattern_size, int text_size, int *failure_function){
 //index for text
 int i = 0;
 //index for pattern
 int j = 0;
 while (i < text_size){
 //if matching increment both index
 if (pattern[j] == text[i]){
 j++;
 i++;
 }
```

```
    //if match the pattern print message
    if (j == pattern_size){
    //printf("found at index %d \n", i-j);
    j = failure_function[j-1];
    }
    else if ( i < text_size && pattern[j] != text[i]){
    if (j != 0){
    j = failure_function[j-1];
    }else{
    i+=1;
    }
    }

```

Since there is a linear looping through the text, the running time of this implementation would be O(n).

6 Experiments

These algorithms are implemented with CUDA and run on GeForce GTX TITAN X graphic card. The input text file and pattern string are generated. The ratio of size of text to the size of pattern matters in my implementation, it decides number of blocks per grid and the number of threads per block in the kernel call. Hence, in my experiment, pattern size is fixed to be approximately one thousand and run both improved algorithm 2 and parallelized brute force algorithm. Then, the running time of each stage of the algorithms is recored.

7 Results and Discussion

$\log_2$ **of input size**	20	22	24
Copy input to GPU	0.560384	1.069824	3.014336
Brute Force Kernel	17.264000	67.444992	310.652466
Copy output to CPU	0.708128	2.559744	10.067680
Total running time	18.532428	71.0735	323.7345

Table 1: Run time of parallelized brute force algorithm for string matching with pattern length approximately 10^3. All times are in milliseconds.

$\log_2$ **of input size**	20	22	24
Build WITNESS Array	0.03	0.03	0.03
Copy input to GPU	0.785664	1.352672	3.696800
Index Elimination Kernel	0.123680	0.404608	1.461184
Brute Force Kernel	0.063232	0.081984	0.806720
Copy output to CPU	0.794976	3.004512	7.773664
Total running time	1.7674	4.873072	13.7682

Table 2: Run time of algorithm 2 for string matching with pattern length approximately 10^3. All times are in milliseconds.

Table 2 shows the running time of each stage of algorithm 2 while Table 1 shows the running time of each stage of the parallel brute force algorithm. The input size ranges from 2^{20} to 2^{24} and the pattern size is 1000.

Even though in theory, the parallel brute force algorithm should perform $O(1)$ running time under PRAM, as you can see, the actual implementation of the brute force algorithm's running time is approximately O(n), since the number of operations in parallel brute force algorithm is O(mn) in theory, which is large. Therefore, in the actual CUDA implementation, the scheduling delay between each WARP of threads makes it slower.

However, in algorithm 2, we improved the number of operation to O(n), and also we optimized the system with consecutive memory access. Therefore, the actual running time of the GPU kernel is fast compared to the brute force algorithm.

We can further optimized the system by replacing the global memory buffer *output* in the process of binary reduction of index elimination with shared memory in each block. Since we need to constantly read data from the memory, my conjecture is that using shared memory would improve the performance. Table 3 shows the result of running algorithm with shared memory on the same input:

However, the result actually shows no difference between method using global memory and method using shared memory. One possible reason is that the frequency of accessing memory is not large enough that changing global memory to shared memory would make a significant different on the performance.

Lastly, we run the same input on our serial string matching algorithm

$\log_2$ **of input size**	20	22	24
Build WITNESS Array	0.03	0.03	0.03
Copy input to GPU	0.736640	1.2390	2.719616
Elimination in shared memory	0.174368	0.676064	2.120864
Brute Force Kernel	0.620896	0.711264	2.389024
Copy output to CPU	0.736224	2.658560	8.388256
Total running time	2.582128	5.559896	15.886760

Table 3: Run time of algorithm 2 in shared memory for string matching with pattern length approximately 10^3. All times are in milliseconds.

(KMP) and the result is recorded in table 4:

$\log_2$ **of input size**	20	22	24
Build Failure Function	0.002	0.002	0.003
KMP matching	3.705	14.676	46.746
Total running time	3.707	14.678	46.749

Table 4: Run time of KMP algorithm with pattern length approximately 10^3. All times are in milliseconds.

KMP algorithm with running time as O(n+m) is unsurprisingly performing worse than algorihtm 2 working on GPU, but it is surprisingly better than the brute algorithm running on GPU. One conjecture might be that KMP algorithm does the similar optimization as algorithm 2 which eliminate some index positions that it will not need to check again. This optimization reduce number of CPU operations, especially with periodic pattern.

References

[1] Jaja, Joseph, *Introduction to Parallel Algorithm*, Addison-Wesley Publishing Company, 1992. p312-339. Print.