CMPE 297 HW#2

Due: Friday, Oct 7, 11:59pm
Total Score: /100

Instructor: Hyeran Jeon

Computer Engineering Department, San Jose State University

Homework is not a group work. Each student should submit his/her own homework solution.

In this Homework, you will parallelize the sequential version of Rabin-Karp string matching algorithm.

Rabin-Karp algorithm: The Karp-Rabin string matching algorithm was designed in 1987 by R. Karp and M.Rabin. It uses hash values to find patterns inside given input text. It is widely used to search multiple patterns in the given string. The popular fields that Rabin-Karp is used are plagiarism checking and malware inspection. For example, Rabin-Karp is an efficient algorithm to detect if there is any identical sentence in two papers and check if there is any well-known malware signature in network packets. Rabin-Karp inspects the input text in a smaller search window unit (a sub-string of the input text that has the same length with the pattern). For a quick comparison, it uses hashing mechanism. The algorithm calculates a hash value for the pattern to detect. Then, it calculates another hash value of the current search window in the input text. If the two hash values match match, it compares the pure text of the pattern and the search window. One the pure texts also match, the algorithm notifies that there is one occurrence of the pattern. If the hash value doesn't match, the algorithm moves on to the next search window without checking the pure text. The hashing mechanism enhances the performance as non-matched part of the input text can be quickly explored by comparing short hash values.

As we do not know where the pattern would start in the given input text, the search window moves by one character at a time. For example, when you check the existence of a pattern "BC" in an input text, "ABCDE", the hash value of the pattern and the hash value of the first two characters, AB, is compared. If the hash values do not match, the search is moved to the next search window, BC. The hash values of "BC" in the input text and the pattern should identical. Then, the pure text "BC" and the pattern are compared. As the pattern matches to the current search window, the algorithm notifies that it found the pattern in the input text.

The sequential Rabin-Karp implementation is shown in Figure 1. In this homework, you will parallelize the code in CUDA. Applications like string matching can be well parallelized as a long input text should be dealt with. Please read the requirements and guidelines carefully to design a CUDA kernel.

```
void RabinKarp(char* input,                    // input text
            int   input_len,                   // input text length
            char* pattern,                     // pattern
            int   pattern_len,                 // pattern length
            bool* result)                      // result is a boolean array
{
      int inph, path, i;
      for(path=i=0;i<pattern_len;++i)
      {                                          //pattern hash
            path=(path * 256 + pattern[i]) % 997; //any hashing function can be
                                                  //used
      }
      int j=0;
      while(j<=input_len-pattern_len)
      {
            int k;
            for(inph=0,k=j; k<pattern_len+j; k++)
                  inph= (inph * 256 + input[k]) % 997;//input hash
                                                      //the same hashing
                                                      //function should be used

            // Check hashes first and then compare the current search window
            // and the pattern
            if(inph==path && memcmp(input,j,pattern,patLen)==0)
                  result[j]=1; // mark that the pattern is found in position j
            ++j;
      }
}
```

Figure 1 Sequential Rabin-Karp

Requirements and guidelines:

We will design two versions of Rabin-Karp string matching: Single pattern matching and multiple-pattern matching. You can use any kind of parallelism in the code. One possible parallelization approach is to assign one thread per search window and have individual thread handle the pattern and one search window comparison. To do that you should assign (input_length – pattern_length) threads to a kernel. Assume that the input text is short enough to be handled in one thread block. Please check the detailed guideline below:

Single pattern matching: In this version, one pattern is searched in an input text.

- The pattern and its hash value are passed as inputs to kernel
- The pattern's hash value will be calculated by host CPU
- The input text is passed without hashing to the kernel.
- In the kernel, the hash value of a search window of the input text should be calculated and compared with the pattern hash value.
- Once hash values match, you should compare the pure text in the search window and the pattern by using your own memory compare function.
- Once the pattern is found in the input text, the position of the pattern in the input text is

marked in a Boolean array, result. The array, “result”, has (input_length – pattern_length) elements. Once the pattern is found in the input text, the element in the matching position is marked as TRUE (or 1). If the pattern appears multiple times in the input text, mark all the positions. When the kernel is terminated, the host code should print all the locations that the pattern is found.

- In the single pattern matching code, use the skeleton code as a baseline. In the skeleton code, fill the sections marked as “ADD CODE HERE”. The single pattern matching code should print like below:

```
Total cycles: 2300
Searching for a single pattern in a single string
Print at which position the pattern was found
Input string = HEABAL
pattern=AB
Pos:0 Result: 0
Pos:1 Result: 0
Pos:2 Result: 1
Pos:3 Result: 0
Pos:4 Result: 0
```

Multiple pattern matching: In this version, one or more patterns are searched in an input text.

- Like in single pattern matching, the hashed value of the patterns will be passed to the kernel and compared with input text.
- In this version, you don’t need to indicate the position of the pattern in the input text; it is sufficient to print if the pattern appears in the input text or not. Design the result array accordingly.
- In the multiple pattern matching code, use the single pattern matching code as a baseline.
- Change the pattern input to represent an array of multiple patterns and copy them from the CPU to the GPU. You need to declare and initialize the pattern (array of strings), as it is not provided in the skeleton code.
- A sample pattern list is like below:

```
Char *temp[3];   // 3 strings of any lengths
temp[0] = “eello”;
temp[1]=”oqw frge”;
temp[2] = “acde fgha”;
```

The array temp in the above example should be transferred to the GPU.

- In the kernel, modify the code to check if each of the pattern exists in the input string.
- Feel free to modify the kernel invocation statement if needed.
- The multiple-pattern matching code should print like below:

```
Sample Output for Part2:
Searching for multiple patterns in the input sequence.
```

```
     Pattern: "eello" was found.
     Pattern: "oqw frge" was found.
     Pattern: "acde fgha" was found.
```

Your code should be successfully compiled and run.

Submit the code to Canvas before the deadline.