Graduate School of Arts & Science

GPU Architecture and Programming
Final Project Report

Group No.2

Parallelize two search algorithms with multi-GPU
and compare the overall performance
with the state-of-the-art

December 2020

Group Members: **Jiancheng Wang, Guangsen Yan**

Student IDs: **jw5865, gy649**

Course Code: **CSCI-GA.3033-025**

Introduction

Pattern matching is the act of analyzing continuous text data and searching it with a known pattern to check if the given pattern is in the presence of any constituents of the text data. It has various well-established algorithms where they are used in many fields of applications, ranging from the light-weight search bar in most browsers and text editors to virus signature detection [1]. However, pattern matching is also considered as a relatively computationally intensive operation.

So the project motivation came from the idea of accelerating the pattern matching process by parallelizing two successful pattern matching algorithms - Knuth-Morris-Pratt and Boyer-Moore-Horspool - to make effective use of the GPU's computational power which is excellent for performing parallel computing.

This report describes the implementation of parallelizing the above mentioned pattern matching algorithms and is divided into three parts. The Pattern Matching section explains the background and pseudo code of the two algorithms. The Discussion of Project section provides the experiment methodology of our parallelization approaches. Technical details and advanced CUDA skills adopted will also be presented in this section. The final Conclusions and Future Work section summarizes the experimental results and challenges we met during this project.

Pattern Matching

String pattern matching is an important topic in text processing and it is widely used in applications such as spell checker and spam filter. Here, we are going to compare three different pattern matching algorithms: naive approach, KMP, and BMH.

Naive approach

Let's denote T as the text string, and P as the pattern string. The naive approach compares $T[i]$ with $P[j]$ one at a time. On success, it increments both i and j by 1 until j reaches the end of P. Otherwise, it resets j and set i' to $i+1$. In other words, when the naive approach sees a mismatch, it shifts the window by 1 position to the right, and then performs the same process. The time complexity for this approach in the worst case is *O(m * n)*.

Knuth-Morris-Pratt

The idea of Knuth-Morris-Pratt (KMP) algorithm stems from the fact that when we encounter a mismatch, it is unnecessary to match some of the characters in the window, as we already know they will match anyway. The preprocessing step involves building a LPS table that records the longest length of the prefix which is also a suffix. The algorithm then starts by comparing $T[i]$ with $P[j]$. When it sees a mismatch at $T[i]$, it recursively compares $T[i]$ with $P[j']]$, where $j' = P[LPS[j'-1]$ until $T[i]$ matches $P[j']$ or $j' = 0$, since the LPS table guarantees that the first LPS [j'-1]-character prefix of P matches the same length suffix of T. In this way, we could slide the window much faster in the worst case and improve the time complexity to $O(m+n)$.

Boyer-Moore-Horspool

The Boyer-Moore-Horspool (BMH) algorithm is a simplified variant of the original Boyer-Moore algorithm [2]. Similar to the previously introduced KMP algorithm, BMH also starts by preprocessing a bad-character table that helps record skip information to avoid unnecessary mismatch. Let *m* be the size of the pattern *P*, the pseudo-code of BMH preprocessing phase is:

```
for c ∈ Σ
        do shift[c] ← m
for i ← 0 to m − 2
        do shift[P[i]] ← m − 1 − i
```

This guarantees that no matter what size of the text and pattern are, the length of the bad-character table would constantly be the size of the alphabet, usually 256 for ASCII. The advantage of this feature for GPU computation will be discussed in the following section.

Then for the search phase, the BMH algorithm would use a window of size *m* to match the pattern from right to left within the window, and skip characters with the support of the bad-character table whenever a mismatch happens.

Discussion of Project

The experiments were conducted on an Intel Xeon CPU server (cuda5) with a 2.6GHZ clock speed. The GPU used was the NVIDIA TITAN Z with a maximum of 15 multiprocessors and 1024 threads per block. The GPU supports a maximum of 6GB global memory and 48KB shared memory per block. The CUDA compute capability is 3.5.

Boyer-Moore-Horspool

The general concept of parallelizing the BMH algorithm is to define numerous chunks to cover the entire text data. Each chunk is assigned to a thread - the smallest execution unit in a CUDA program [3]. Then each thread would concurrently perform the BMH search within its own chunk to look for whether a pattern exists. If missed, that thread would look up the shift table, make the shift, and continue to search. Otherwise, a result array would document the position of every occurrence of the pattern in the text data whenever there is a match.

Deciding the size of a chunk is a trade-off between the performance of the BMH algorithm itself and the efficiency of GPU memory access. If the chunk size is particularly small, the global memory access frequency per thread would be lower which is advantageous for GPU computing to reduce communication overhead, yet the BMH shift table utilization would equally drop fast. In the extreme case where chunk size equals 1, as the GPU achieves coalesced memory transactions from multiple consecutive global memory locations, the complete BMH algorithm would be downgraded to naive brute-force which is not favorable for the motivation of this project. On the other hand, a large chunk size encourages better BMH shift table utilization at the cost of more global memory accessing overhead for each thread. For the unity of various experiment parameters in this project, the chunk size for the BMH GPU version is set as pattern size plus 32 extra characters to balance the earlier mentioned trade-off.

Preliminary experiment results regarding the early version of our parallelized BMH algorithm showed that there was a significant speedup of the processing time for the GPU kernel to complete the search compared with the sequential CPU version. However, over 95% of the GPU activities were actually communication overhead:

BMH on Single GPU with Text Size 10MB, Pattern Size 4 bytes

GPU activities	Time(%)	Time	Calls	Avg
[CUDA memcpy HtoD]	56.28%	17.549ms	4	4.3873ms
[CUDA memcpy DtoH]	39.05%	12.177ms	1	12.177ms
KERNEL_BMH	4.67%	1.4558ms	1	1.4558ms

Since transfers between the host and device are the slowest link of data movement involved in GPU computing [4], a number of techniques were adopted to optimize the efficiency of data transfer.

For this parallelized BMH algorithm, the components consisting of the host to device data transfer were minimized into 4 essential factors: text data, a pattern to search, a bad-character shift table and an integer type result array at the same length as the text data. The result array is also responsible for the sole source of device to host communication after the search has ended. As it has been mentioned in the previous section, the length of the bad-character table would constantly be the size of the alphabet set, which is usually 256 for ASCII characters. In other words, the size of the bad-character table would not be affected by the length of any input e.g. text data or pattern and remains relatively small. So the communication overhead was primarily dominated by the length of text data (result array) and pattern.

Reducing the size of text data or pattern is certainly a solid manner to narrow down the communication overhead. However, it is against the nature of GPU computation that typically gains computational advantage over its sequential counterpart by large-scale parallelization on large data sets. Pinned host memory was then leveraged to improve data transfer efficiency without compromising the problem size.

The pinned memory is used to be an intermediate of data transfer between pageable host memory and device memory. By directly allocating host memory as pinned memory, the cost of the host-device communication decreased up to 69%:

BMH on Single GPU with Text Size 10MB, Pattern Size 4 bytes, Pinned Memory

GPU activities	Time(%)	Time	Calls	Avg
[CUDA memcpy HtoD]	48.01%	5.1167ms	4	1.2792ms
[CUDA memcpy DtoH]	38.40%	4.0930ms	1	4.0930ms
KERNEL_BMH	13.59%	1.4485ms	1	1.4485ms

This improvement is even more significant with larger data sets:

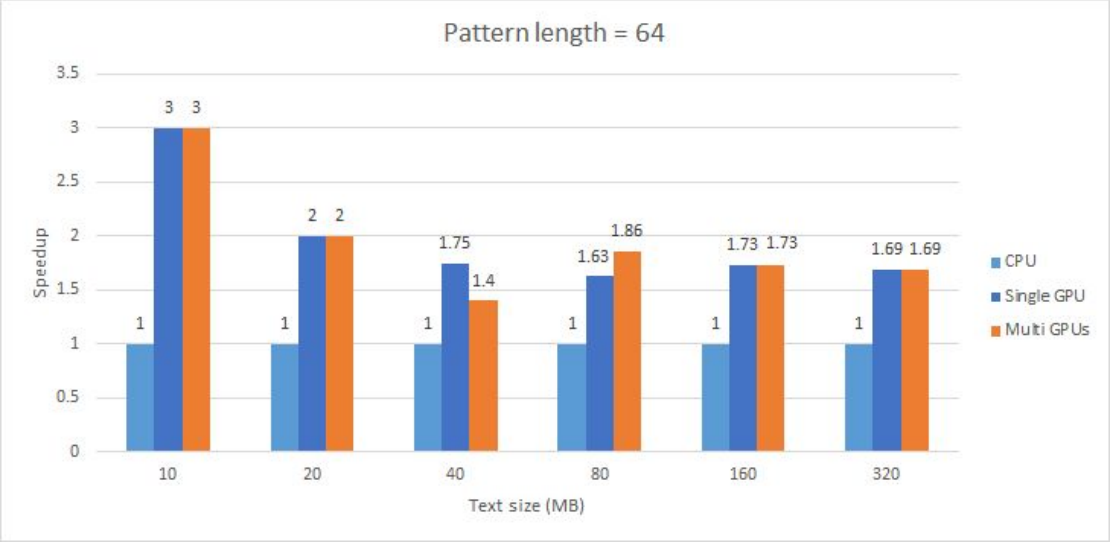
BMH on Single GPU with Text Size 320MB, Pattern Size 64 bytes, Pageable Memory

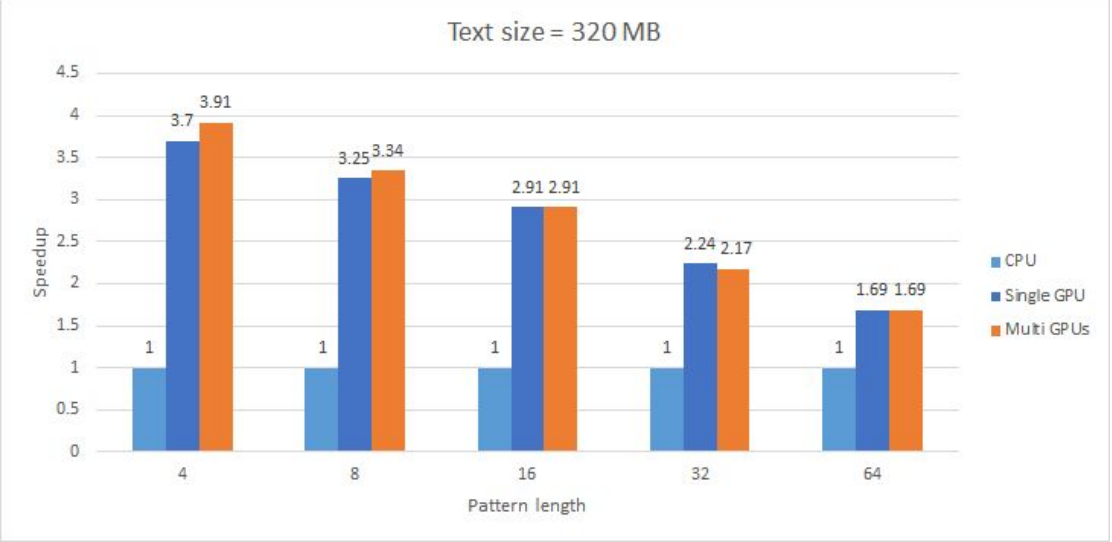
GPU activities	Time(%)	Time	Calls	Avg
[CUDA memcpy HtoD]	70.12%	1.02914s	4	257.28ms
[CUDA memcpy DtoH]	27.34%	401.27ms	1	401.27ms
KERNEL_BMH	2.54%	37.263ms	1	37.263ms

BMH on Single GPU with Text Size 320MB, Pattern Size 64 bytes, Pinned Memory

GPU activities	Time(%)	Time	Calls	Avg
[CUDA memcpy HtoD]	48.30%	137.75ms	4	34.437ms
[CUDA memcpy DtoH]	39.01%	111.25ms	1	111.25ms
KERNEL_BMH	12.70%	36.212ms	1	36.212ms

Another approach to increase parallelization performance is to introduce support for multi-GPU capability. The concept for the multi-GPU version of BMH algorithm is quite straightforward. The text data is evenly split and assigned to each device as slices which are used as local text data for each device and then executing the BMH algorithm discretely. In order to avoid the case where a matched pattern happens to be splitted over the text data, all slices must be overlapped to a certain extent for cross-checking. With additional overlapping space, the minimal length for every slice is $N_s = \frac{N}{Total_{DeviceNumber}} + M - 1, M = pattern\_size, N = text\_size$





The result diagrams illustrated the speedup gained by both single and multi GPUs when executing the parallelized version of BMH algorithm with various text sizes and pattern length settings. The input text data used in this project was an iterated DNA sequence of ChOR-seq [5] and the strings to be matched were randomly chosen DNA patterns composed of the “ACGT” characters.

From the barcharts we can learn that, even with the costly bilateral communication between the host side and the device side taken into account, our paralleled versions of both single and multi GPUs still had considerable performance improvements.

Contrary to popular belief, the paralleled version seemed to perform better with compact text data and patterns in average-cases. This is due to the nature of the BMH algorithm that larger patterns are more likely to help the sequential version build

a more effective bad-character table assisting it to make larger skip when a mismatch happens. With limited chunk size on GPU versions as discussed earlier, such large skip would be generally wasted since the thread could easily reach its chunk boundary. In addition, larger text data or patterns would conventionally result in more expensive communication overhead that significantly slows down GPU performance.

The above experiment did not suggest tremendous superiority to utilize multi-GPU in most average-cases. Thus, another experiment focusing on the worst-case performance was carried out.

In the worst-case scenario for the BMH algorithm, the text data and the pattern are literally repeated single identical characters, making the shift values of the bad-character table being only as little as *1*. This introduces a time complexity of *O(mn)*.

BMH on Single GPU with Worst-case Size 16MB, Pattern Size 512 bytes

GPU activities	Time(%)	Time	Calls	Avg
KERNEL_BMH	92.25%	353.12ms	2	176.56ms
[CUDA memcpy DtoH]	4.17%	15.948ms	2	7.9742ms
[CUDA memcpy HtoD]	3.59%	13.738ms	8	1.7173ms

BMH on Multi GPUs with Worst-case Size 16MB, Pattern Size 512 bytes

GPU activities	Time(%)	Time	Calls	Avg
KERNEL_BMH	94.23%	406.16ms	1	406.16ms
[CUDA memcpy HtoD]	3.00%	12.937ms	4	3.2344ms
[CUDA memcpy DtoH]	2.77%	11.950ms	1	11.950ms

Compared with the average-case, the multi-GPU performance is approximately twice the one of the single GPU version. The above tables showed the worst-case utilizations of both GPU versions are much higher than the average-cases. This indicates that the communication overhead between host and device in the worst case can be overcome by the greater advantage for a higher kernel utilization over the long haul given the larger problem size.

Knuth-Morris-Pratt

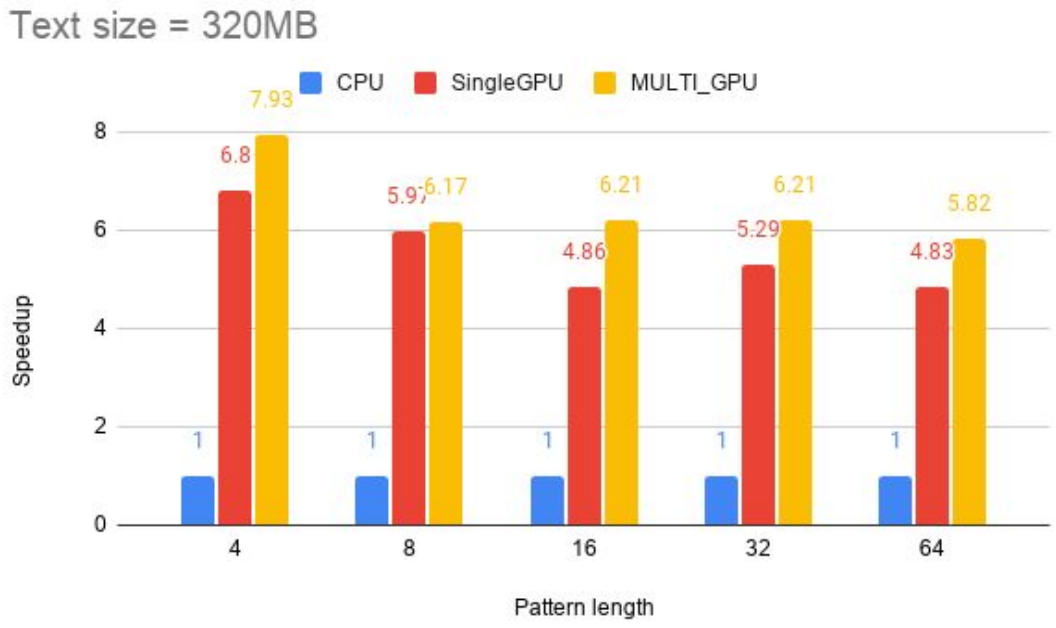
In order to maximize the parallelism, the algorithm splits the text into several chunks and then performs pattern matching simultaneously. Each thread performs KMP on a specific chunk of the text data to check if the pattern exists. The result array is first initialized to 0. When a pattern has been found, the algorithm would mark the position at the start of the matching pattern of the result array as 1. As KMP involves lots of accesses to the LPS table, the algorithm uses shared memory to reduce the latency in accessing global memory. This is done by having each thread first load a chunk of the LPS table $N_m = \frac{M}{Threa\_per\_block}, M = pattern\_size$ to the shared memory, and then use the barrier to synchronize among all threads within a block. In order to avoid splitting pattern over 2 threads, the number of characters assigned to each thread $N_n = \frac{N}{Total_{ThreadNumber}} + M - 1, M = pattern\_size, N = text\_size$

Since the parallelization of KMP is similar to BMH, we ran the experiment in the same setting as mentioned above, including tiling, using pinned memory and taking advantage of multiple GPUs. The very implementation is also fixed in terms of the chunk size and thread per block to better compare the two algorithms.

KMP on Single GPU with Text Size 320MB, Pattern Size 64 bytes, Pinned Memory

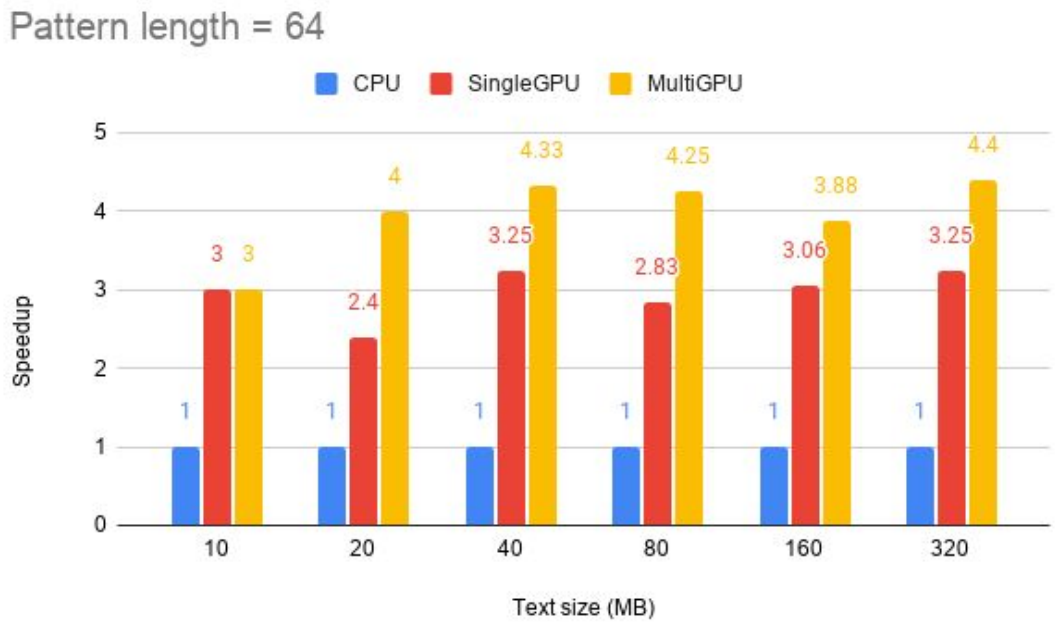
GPU activities	Time(%)	Time	Calls	Avg
KERNEL_KMP	42.76%	199.64ms	1	199.64ms
[CUDA memcpy HtoD]	34.46%	160.90ms	4	40.224ms
[CUDA memcpy DtoH]	22.78%	106.33ms	1	106.33ms

The above experiment runs on a 320MB dataset, produced by replicating the DNA sequence of ChOR-seq [5], with a random DNA sequence composed of “ACGT” characters as the search pattern. Compared to the result of parallelized BMH in the same setting, parallelized KMP underperforms in terms of the kernel processing speed, given the data transfer between GPU memory and system memory is roughly the same.



This experiment is conducted to explore how pattern length could affect the performance of parallelized KMP. By fixing the size of the text, we can clearly see a decline in the speedup when pattern size increases. Theoretically, the speedup should stay the same, as long as the chunk size for each thread is static. However, in order to cover the splitting over two patterns, we manually assign additional pattern_length characters to each thread and thus causes more computation to

happen. Besides, unlike the BMH algorithm, the LPS table of the KMP algorithm also grows with the increase of pattern length and therefore leads to more memory latency when a mismatch happens.



The above illustrated above showcases the performance of sequential, single GPU, and multi GPU versions of KMP. Although the absolute runtime is longer than BMH, KMP parallelizes better when compared horizontally -- the speedup is about 3 times of the sequential version.

KMP on Single GPU with Text Size 160MB, Pattern Size 64 bytes, Pinned Memory

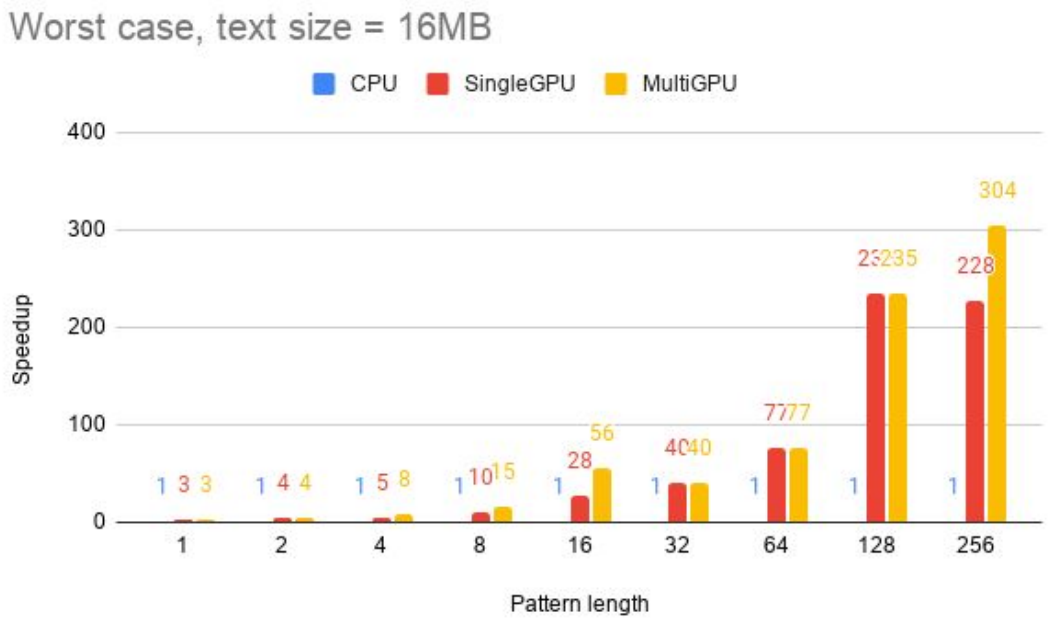
GPU activities	Time(%)	Time	Calls	Avg
KERNEL_KMP	62.93%	207.39ms	1	207.39ms
[CUDA memcpy HtoD]	21.17%	69.754ms	4	17.439ms
[CUDA memcpy DtoH]	15.91%	52.419ms	1	52.419ms

KMP on Single GPU with Text Size 80MB, Pattern Size 64 bytes, Pinned Memory

GPU activities	Time(%)	Time	Calls	Avg
KERNEL_KMP	64.30%	110.71ms	1	110.71ms

[CUDA memcpy HtoD]	20.22%	34.805ms	4	8.7013ms
[CUDA memcpy DtoH]	15.48%	26.654ms	1	26.654ms

By looking into the result of the profiling tool, we found out that the processing time for all three stages is proportional to the text size. From the formula we used to assign the total number of threads and the total number of blocks: $N_{TotalThread} = \frac{N}{chunk\_size}$, and $N_{TotalBlock} = \frac{N_{TotalThread}}{thread\_per\_block}$, the only determinant of the processing time, excluding any system overhead, is the text size N. In other words, the speedup should always stay the same, since the ratio between text size and processing time is only determined by $thread\_per\_block$ and $chunk\_size$, which are constant.



The worst-case scenario for KMP happens when the text is composed of replicated versions of the pattern followed by a different character. Assume the pattern is “aaaaa”, the “ideal” worst case is “aaaaabaaaaab….”. In this example, when $T[i]$ is the first ‘b’, the algorithm will see a mismatch. It then utilizes the LPS table to re-align the pattern with the target text. In the pattern composed of duplicate single characters, $LPS[i] = i - 1$. This yields the pattern to iteratively shift one position to the right until the first ‘a’ just past the first ‘b’. In other words, the larger the $LPS[i]$, the slower the pattern window shifts. The overall time complexity in this case would

be O(m* n). From the figure shown above, KMP parallelized very well in the worst case.

Conclusions and Future Work

This project parallelized two successful search algorithms Knuth-Morris-Pratt and Boyer-Moore-Horspool in both single and multi-GPU versions. Each parallelized implementation has shown considerable performance improvements over the original sequential CPU programs, especially in the worst-cases. It is also worth noting that due to the fact of communication overhead and the nature of certain algorithms, the stereotype of larger problem sets guaranteed to be beneficial to the parallel computation on the GPU is not consistently correct. The BMH experiment of the performance over average-cases is a case in this point.

There are absolutely challenges and limitations in this project. For example, some related works [6] mentioned that by loading the lookup table and the pattern into the shared memory, string matching algorithms on the GPU could receive up to 25 times performance increase. However, it does not comply with our experimental results as leveraging shared memory by storing frequently used pattern and lookup table in each block for both KMP and BMH algorithms would also cause serious bank conflict issues since different threads in the same block are very likely to randomly access the data on the same bank of the shared memory simultaneously.

The future works of this project can be conducted in several directions. First, the optimization of the communication overhead still has room for improvement. If this time-consuming process can be further refined, then it would greatly improve the overall performance of our paralleled programs. Second, further research should investigate if there exists an approach to better utilize the shared memory that detects and solves the bank conflict issue. Last but not least, more diverse patterns and text data can be introduced to explore if there are any interesting behaviors regarding our paralleled search algorithms.

References

[1] X. Zhou et al (2008). *MRSI: A Fast Pattern Matching Algorithm for Anti-virus Applications.*

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

[3] Cornell Virtual Workshop. *Introduction to GPGPU and CUDA Programming: CUDA Thread.*

Available at: https://cvw.cac.cornell.edu/gpu/thread

(Last accessed: December 8 2020)

[4] M. Harris (2012). *How to Optimize Data Transfers in CUDA C/C++*

Available at: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

(Last accessed: December 8 2020)

[5] HGNC. *C7orf26 chromosome 7 open reading frame 26 [Homo sapiens (human)]*

Available at:

https://www.ncbi.nlm.nih.gov/gene?Db=gene&Cmd=DetailsSearch&Term=79034

(Last accessed: December 8 2020)

[6] C. S. Kouzinopoulos et al (2009).*String Matching on a multicore GPU using CUDA*