— Parallel String Matching —

Dengyu Liang
University of Ottawa
dengyuliang@cmail.carleton.ca

December 8, 2022

Abstract

This project explores CPU-parallel and GPU-parallel implementations of 6 different string-matching algorithms, and evaluates their performance and the benefits of parallelizing on natural language text, genomic data, and program-generated data. Most algorithms can benefit from CPU parallelism. achieving 3-10x speedup in 16 threads. GPU performs poorly on small files, and has huge speedups on special data, it has a 2x speedup on the SO algorithms, and 2/3 Cuda cores speedup in the worst case on the brute force algorithms.

1 Introduction

String Matching is one of the most common problems in computer science, and although Single-pattern algorithms work very well on this problem, the algorithm speed is affected by the linear increase in the size of the matching string, especially when solving Bioinformatics DNA Sequencing or search engines or content search in large databases, a large amount of data inevitably takes a lot of time. Modern CPUs tend to have a large number of cores, and it is increasingly common to utilize GPUs for data processing. Parallelization has become an important part of algorithm design. Because of the data structure of String Matching, it can take benefits from Parallel Programming to solve large-scale Matching problems efficiently on large-scale systems.

This project is to explore the parallel implementation of traditional sequence string matching algorithms and implement its parallel version over cilk and cuda, then compare the benefit of different string matching algorithms in different parallel architectures. In this project, I implement six different string-matching algorithms in CPU parallel and three in GPU. For algorithms that are difficult to algorithmically parallelize, I partition the index to perform parallel operations. Some of the algorithm takes also use bit parallel or SSE instruction to speed up. I analyze The speedup obtained by different algorithms, and the speedup obtained in the worst case.

A more effective algorithm can be selected through pattern length analysis and preprocessing. Our results also discuss this point. The data type also has an impact on efficiency. A gene sequence with four letters has more repetitions than natural language, and this repetition will also greatly affect efficiency. We'll also compare different datasets and explore the differences.

2 Literature Review

String Matching is an important problem in computer science, and many people continue to study it. The most recent comparison was Philip Pfaffe, who discussed 7 parallelized string matching algorithms based on SIMD[1]. There are different architectures according to Flynn classification, and for different architectures, there have different methods to minimize latency and improve performance. This study compares the advantages and disadvantages of different architectures in solving string-matching problems and measures the performance of CPU parallelism and GPU parallelism.

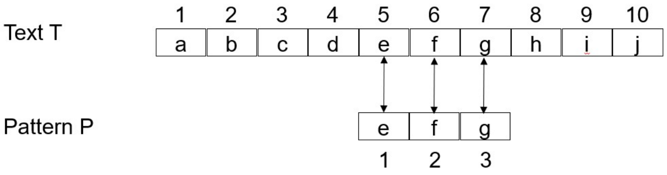


Figure 1: String Matching wants to find the pattern from Text

2.1 Sequential String Matching

String Matching is one of the common fundamental problems, and many algorithms have been proposed to solve it. Faster solutions were proposed as early as 1977, the kmp algorithm and the Boyer–Moore algorithm[1]. Since 1970, more than 80 String Matching algorithms have been proposed. Based on previous research those algorithm deformations, combinations, and expansion, String Matching has been continuously optimized. Now, using parallelized algorithms to improve performance is a more mainstream method[1], nonetheless, can benefit from the ideas of these Simple optimization algorithms. By exploring the details of these algorithms, the efficiency of parallelized algorithms can be further improved.

2.2 Parallel String Matching

Parallel computing has great potential and extremely high scalability. Making good use of parallel computing can greatly speed up the calculation speed. Although not all algorithms can benefit from parallel computing. Specific to the String Matching algorithm, It can be parallelized from data input, It is also possible to parallelize certain operations. For different algorithms, we have to analyze what kind of speedup we achieved in different numbers of threads and calculate the best core utilization.

Parallel String Matching base on MISD
Although there exist MISD architectures dedicated to pattern matching that can be used to solve string match such as Halaas's recursive architecture [6] could match up to 127 patterns at a clock frequency of 100 MHz, there is not much research in this area , it has not shown better performance for single pattern, Its performance is about the same as a 1GHZ single

core CPU. MISD processors also lack practical applications, and research in this area has been shelved.

Parallel String Matching base on SIMD

SIMD uses one instruction stream to process multiple data streams. A typical example is the units on the GPU. Most Parallel String Matching benefits from SIMD architecture. Such as Chinta Someswararao's Butterfly Model [9], and some string matching algorithms are developed based on the SSE instruction set [1], for example the SSEF algorithm. which is an SIMD instruction set developed by intel.

Parallel String Matching base on GPU(CUDA)

Compared with the CPU, modern GPU has huge advantages in parallel computing. A single GPU integrates thousands of computing units, so thousands of parallel computing can be realized on a single GPU. There are also some ways to use CUDA programming provided by Nvidia GPU for String Matching. Giorgos Vasiliadis implements string search and regular expression matching operations for real-time inspection of network packets through the pattern matching library [2], And in 2011 it reached a data volume close to 30 Gbit/s. Efficient GPU algorithms can be 30 times faster than a single CPU core, and Approximate String Matching with k Mismatches reports an 80 times speedup [3]. This makes GPU computing have certain advantages in cost and speed, but the research in this area is not as much as the traditional string algorithm.

Parallel String Matching base on MIMD

MIMD machines can execute multiple instruction streams at the same time. Many modern CPUs belong to this type. In order to fully mobilize multiple instruction streams, targeted parallel algorithms are inevitable.There are not many studies in this area. Hitoshi developed an algorithm call PZLAST use for MIMD processor PEZY-SC2 and compared the performance of BLAST+, CLAST and PZLAST algorithms[11], which are specially optimized for the biological field.However, they didn't explore the percentage of parallel utilization of the algorithm.

Distributed Memory Parallel String Matching

There are few articles discussing String Matching algorithm on Shared and Distributed-Memory Parallel Architectures, except Antonino Tumeo's Aho-Corasick algorithm in 2012 compared and analyzed the distributed memory architecture and shared memory architecture[10]. Results at the time showed that shared-memory architectures based on multiprocessors were the theoretical best performance, but there is an upper limit to the amount of parallelism, at 80 threads he starts to get degraded speedup, beyond 96 threads the speedup becomes marginal. considering cost constraints, GPUs that did not reach the PCI-Express bandwidth limit had the best price/performance ratio. The performance of gpu has developed several times in the past ten years, and theoretically its performance is far faster than that of cpu by now. Although distributed can provide sufficient space and computing resources, limited by the cost of communication, the performance of distributed computing is not satisfactory. Especially for a single match problem. Nonetheless, this can lead to considerable performance gains for multiple search, as shown by Panagiotis' work[8], In a 16-processor cluster, the matching time can be reduced by half of the original.

3 Problem Statement

A definition of the problem of string matching is to find a pattern P in a text T. we define a pattern that has length p and text that has length t. The pattern and text are based on the same alphabet. The results are the positions of every occurrence of P in T. The input is dynamic, preprocessing of pattern or text have to take place at runtime. Only exact matches are returned, approximate matches or regular expression patterns are not considered.

4 Implementation

The algorithms we explore range from the most basic brute force algorithms, partial optimization algorithms to modern advanced algorithms using SSE and bit parallelism. In the parallel implementation of the algorithm, except for the Brute Force algorithm, which is a direct parallel loop, the CPU of other algorithms uses the method of index segmentation in parallel, so that each thread is allocated to a string of t/c+p, thereby realizing parallel operations. The gpu only implements the parallel implementation of Brute Force and Boyer-Moore. Some of the algorithms use the SSE instruction set to improve bit parallelism. The following subsections provide a brief overview of the implemented algorithms.

4.1 Brute Force

The basic Brute Force algorithm compares the pattern to the text, one character at a time until unmatching characters are found. Brute Force means the worst algorithm in most problems. But that's not exactly the case in String Matching Problem. In the average case, Brute Force takes O (P + T) steps. The result reflected in the experiment is that in some natural language searches, the Brute Force algorithm faster than some algorithms. But there's no way to avoid bad worst-case outcomes. In the worst case, the Brute Force algorithm requires (p*t) steps, which is unacceptable in large searches. Many algorithms benefit from the pattern to avoid the worst case.

4.2 KMP algorithm

KMP algorithm, a well-known early optimization algorithm, KMP calculates the number of times it can skip each time it reads a string, It uses a preprocessing phase on the pattern to build a partial match table. Although this avoids the worst result, In theory, KMP has a time cost of O(t), but due to the calculations at the same time, KMP can't speed up too much. Excellent performance in some cases, but doesn't take high scores in most cases. relatively, due to more calculations, it gets more benefit from parallelism.

4.3 Boyer-Moore

Boyer-Moore algorithm calculates the bad character rule and the good suffix rule, and matches from the last index to the front. Therefore, it can skip some unmatched results and thus exceed linear time in algorithm analysis for most natural languages and large pattern capacity matching problems with better speedup than KMP. it has (p) Preprocessing time and (t/p) at best, O(p*t) at the worst Matching time, and require (p) space to store the good and bad rule.

4.4 Rabin-Karp

The Rabin-Karp algorithm gets inspiration from the hash function. When scanning strings for comparison, it uses the hash value generated by Rolling hash comparison instead of directly comparing strings. This avoids directly scanning the entire pattern. Only strings with the same hash can be compared. A full scan will be performed. Because the hash is fixed, the speed of the algorithm depends on the hash function and the string alphabet. The worst result is still mn, but as a heuristic algorithm, it is faster than the expected value of the brute force algorithm.

4.5 Shift-Or

The Shift Or algorithm uses bitwise techniques[1], it keeps an array of bits, R, showing if prefixes of the pattern don't match at the current place. Before searching, mismatch arrays are computed for each character in the alphabet and saved in an array, S. For the next position, with the character c, R = shift(R) or S[c]. If the last bit of R is 0, the pattern matches. The runtime is deterministic and in O(n*m). Nevertheless, it is fast operation due to bit parallelism.

4.6 SSEF

The SSEF algorithm precomputes 65536 filter lists based on the kth bit of each character on the pattern. These filters are then applied efficiently, utilizing SSE instructions, on shifting alignments of pattern and text. SSEF is restricted to patterns with a minimum length of m greater than 32. The worst-case runtime is in O(n*m). If we consider the probability to filter possible matches, SSEF achieves an average runtime in O(n*m/65536). The SSEF algorithm doesn't exactly cover all implementations, but it's a good comparison since Philip Pfaffe claims it has the best performance on their evaluation.

4.7 Exact-Packed-String-Matching

Exact-Packed-String-Matching (EPSM) was presented in 2013 by Faro and Külekci. EPSM makes use of bit-parallelism by packing several characters into a bit-word and partitioning text T into chunks Ti. These bit-word-sized chunks are compared with a packed pattern bit-word. Shift and bitwise-and operations are used to efficiently compare text chunks with the pattern. Our implementation uses SSE registers as 128 bit words. The asymptotic runtime for the algorithm remains O(n*m).

5 Experimental Evaluation

In this section, I present the performance evaluation of parallel string matching algorithms. The sequential implementation of some codes comes from String Matching Algorithms Research Tool. Here are the datasets I used for comparison:

A sample of natural language (the text of the Bible)
A set of genome sequences (consisting of four letters)
Extreme data generated by some programs (lots of repetitive text)

In order to evaluate the benefits of parallelization, the input file is directly read into memory completely to reduce I/O latencies, and then my algorithm is run on a number of 1, 2, 4,

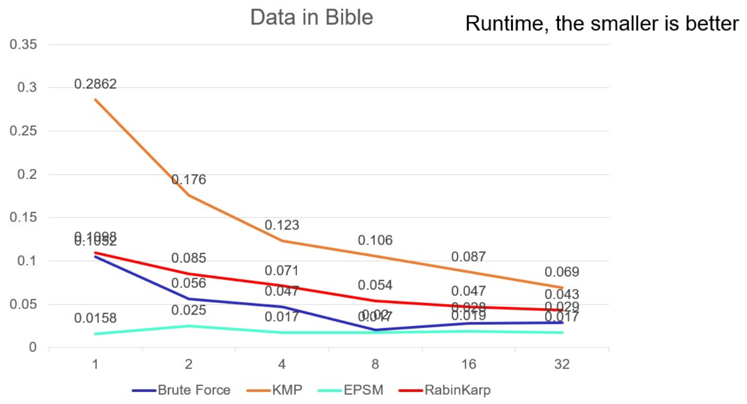


Figure 2: Runtime of seven algorithms for 1,2,4, 8, 16, 32, and 64 patterns. natural language text (bible).

8, 16, 32 threads, and their respective running times are recorded. And test patterns with lengths of 2, 4, 8, 16, 32, 64 and 1000.
All CPU parallel experiments were performed on AMD ryzen9 3950x and compiled by opencilk. The CPU has 16 CPU cores (32 hardware threads) clocked at 4.1GHz. The GPU is based on nVidia RTX 2070 8GB running on the latest CUDA 22.04.
In the following subsection, I discuss an excerpt of our result data.

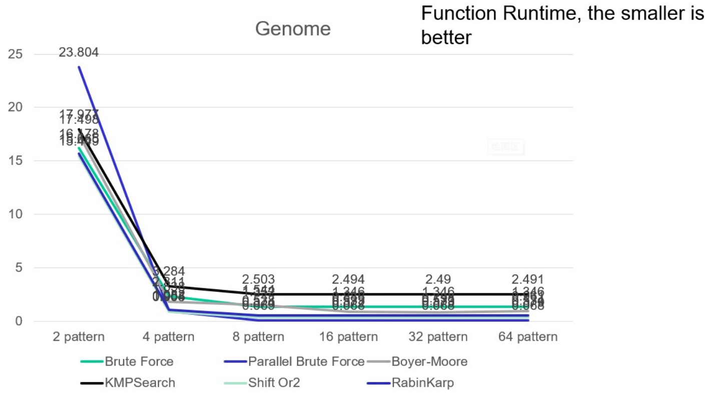


Figure 3: Runtime of seven algorithms for 1,2,4, 8, 16, 32, and 64 patterns. genome data set.

5.1 Pattern

This is about the impact of pattern on parallel performance. When I want to analyze the benefits of threads, I want to find a relatively stable pattern as an input. Figure 2,3,4 is

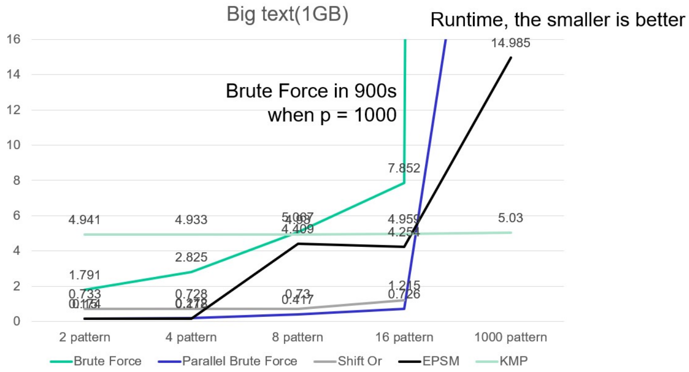


Figure 4: Runtime of seven algorithms for 1,2,4, 8, 16, and 1000 patterns.complex generated data set.

our test result, we can see that in our test, it is indeed affected Very short pattern (pattern less than 4) does have a certain impact on the algorithm, but when pattern greater than 8, only the Boyer-Moore algorithm benefits from it, and the impact on other algorithms is marginal.

However, in the worst case, brute force and epsm increase the running time significantly with the increase of patterns, while the KMP algorithm remains stable. This is due to the fact that epsm partitions the text T into blocks to use bit parallelism compared to packed mode bitwords. Due to the characteristics of memory comparison, the length of each comparison using bit parallelism is fixed, and the advantages cannot be utilized for large patterns., a disadvantage of use bit parallelism. However, KMP does not increase the running time because it can jump enough distance each time. Of course, in this case, the running time of the brute force algorithm reached 15 minutes, which is why we cannot directly use the brute force algorithm, the running time in this case is unacceptable.

5.2 Thread

We focus on the impact of threads on algorithms or different speedup ratios. Figure x shows the variation of the running time of the matching algorithm in the Bible text under different numbers of threads, most algorithms are speedup under multi-core, except EPSM, but the speed of EPSM on the parallel has over the sequence. The running speed on CUDA in some algorithm is slightly better than sequence, but it is difficult to exceed the parallel speedup of 16 threads. Figure 2 is the data under the genome file, due to the nature of the genes, The large number of repeat in the small dataset ATCG, the running time of the brute force algorithm is increased, while the other algorithms remain stable. Figure 3 shows the poor data being generated. In this case the brute force algorithm takes 15 minutes, most algorithms take seconds, the fastest algorithm are shift or and KR, less than a second. In this case, we can see a significant speedup, with all algorithms achieving a speedup of more than 5X speed up in 32-thread parallel. The time spent by cuda is much faster than that of cpu parallel in this case. In particular, the brute force algorithm is actually 0.3 seconds, it is over 1500x speedup. This speedup ratio has way higher than all sequential algorithms,

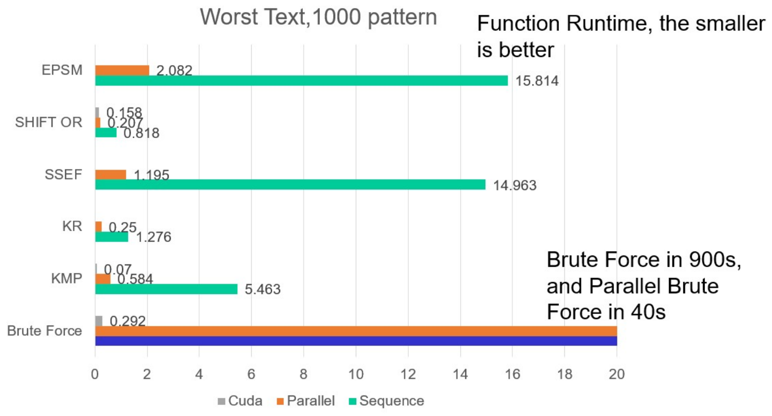


Figure 5: Runtime of seven algorithms for 1,2,4, 8, 16, and 32 threads. Genome data set, complex generated data sett, pattern length of 32.

means that under normal text, the GPU can hardly exert its performance, and everything is done.

5.3 Speed up

Now let's look at the speedup. The speedup ratio in normal text is not high. On average, the speedup ratio can only reach less than half of the number of cores. When there are 32 threads, the performance of the RabinKrap algorithm even drops slightly, while the KMP algorithm achieves a good speedup ratio. And the brute force algorithm speeds up poorly. But in the worst text, the brute force algorithm has achieved a good speedup ratio, which is not without cost, and the time spent dropping from 10 minutes to 40 seconds is still unacceptable. In contrast, the speedup achieved by KMP and RabinKrap is not high, but it is far more than other algorithms, KMP even came to the third place in speed in this case

6 Conclusions

I sliced the index according to the number of threads to do parallel versions of six different string-matching algorithms, and compared to the version on the GPU, I tested on different types and sizes of datasets. Most algorithms can benefit from CPU parallelism except for EPSM. achieving 3-10x speedup in 16 threads. Brute force algorithms benefit most from parallelism, and parallel brute force algorithms perform best on most natural texts, Followed by EPSM. While sequential EPSM has the best performance in genetics and generated text, it cannot benefit from parallelism except for the worst case. In the worst case, the GPU best achieves a speedup that matches the number of Cuda cores in the brute-force algorithm, on the average normal text has a 4x speedup. In the general case, the algorithm only has a 2x speedup on the GPU. I observed the effect of the pattern size on the string matching algorithm. For short patterns, EPSM has far better performance than parallel performance, In my case only the worst case is seen to speed up the algorithm by parallelism. while for 1000 patterns SSEF is better than EPSM. This is because the EPSM using

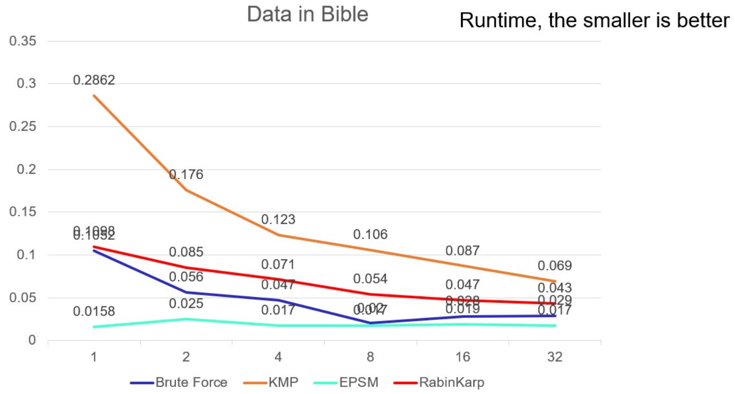


Figure 6: Runtime of seven algorithms for 1,2,4, 8, 16, and 32 threads. natural language text, pattern length of 32.

bit parallelism only needs one instruction to compare within the range allowed by the memory, and the excess part still needs to be searched by the loop. This shortcoming is made up for in SSEF, but due to the existence of the filter list, SSEF cannot guarantee to find all the solutions. Even in this case, EPSM still has good performance and is the second best.

Therefore, through the analysis of the pattern, it's able to select a better method for the String Matching problem. In the future, I plan to analyze the pattern and provide the optimal solution by choosing a better method for string comparison.

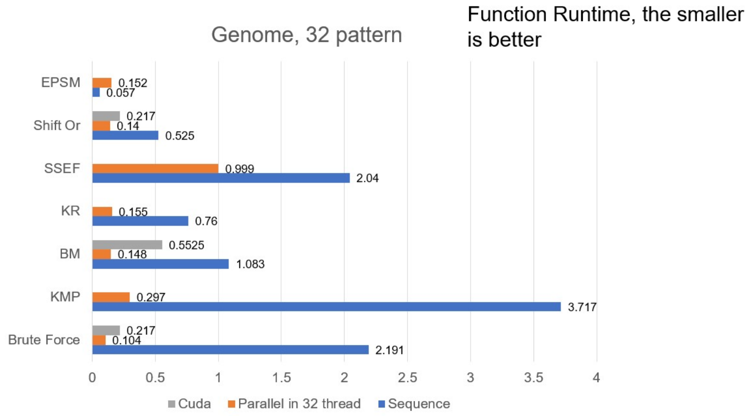
References

[1] Pfaffe, P., Tillmann, M., Lutteropp, S., Scheirle, B., Zerr, K. (2017). Parallel String Matching. In: , et al. Euro-Par 2016: Parallel Processing Workshops. Euro-Par 2016. Lecture Notes in Computer Science(), vol 10104. Springer, Cham. https://doi.org/10.1007/978-3-319-58943-5 15

[2] C. -L. Hung, T. -H. Hsu, H. -H. Wang and C. -Y. Lin, "A GPU-based Bit-Parallel Multiple Pattern Matching Algorithm," 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2018, pp. 1219-1222, doi: 10.1109/HPCC/SmartCity/DSS.2018.00205.

[3] G. Vasiliadis, M. Polychronakis and S. Ioannidis, "Parallelization and characterization of pattern matching using GPUs," 2011 IEEE International Symposium on Workload Characterization (IISWC), 2011, pp. 216-225, doi: 10.1109/IISWC.2011.6114181.

[4] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. SIGPLAN Not. 47, 8 (August 2012), 129–140. https://doi.org/10.1145/2370036.2145833



Genome, 32 pattern

Function Runtime, the smaller is better

Figure 7: Runtime of seven algorithms for 1,2,4, 8, 16, and 32 threads. Genome data set, pattern length of 32.

[5] Y. Liu, L. Guo, J. Li, M. Ren and K. Li, "Parallel Algorithms for Approximate String Matching with k Mismatches on CUDA," 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, 2012, pp. 2414-2422, doi: 10.1109/IPDPSW.2012.298.

[6] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snove and O. R. Birkeland, "A recursive MISD architecture for pattern matching," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 12, no. 7, pp. 727-734, July 2004, doi: 10.1109/TVLSI.2004.830918.

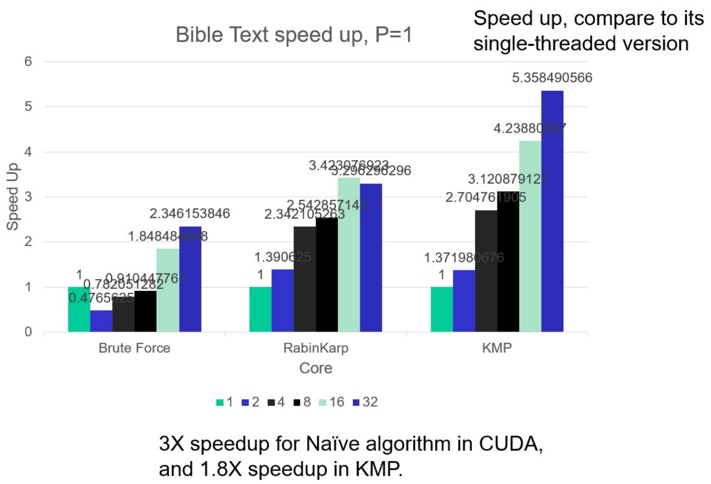
[7] Knuth, D. E., Morris, Jr, J. H., Pratt, V. R. (1977). Fast pattern matching in strings. SIAM journal on computing, 6(2), 323-350.

[8] P. D. Michailidis and K. G. Margaritis, "Performance Evaluation of Multiple Approximate String Matching Algorithms Implemented with MPI Paradigm in an Experimental Cluster Environment," 2008 Panhellenic Conference on Informatics, 2008, pp. 168-172, doi: 10.1109/PCI.2008.13.

[9] Someswararao, Chinta. (2012). Parallel Algorithms for String Matching Problem based on Butterfly Model. International Journal of Computer Science and Technology.

[10] A. Tumeo, O. Villa and D. G. Chavarria-Miranda, "Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 3, pp. 436-443, March 2012, doi: 10.1109/TPDS.2011.181.

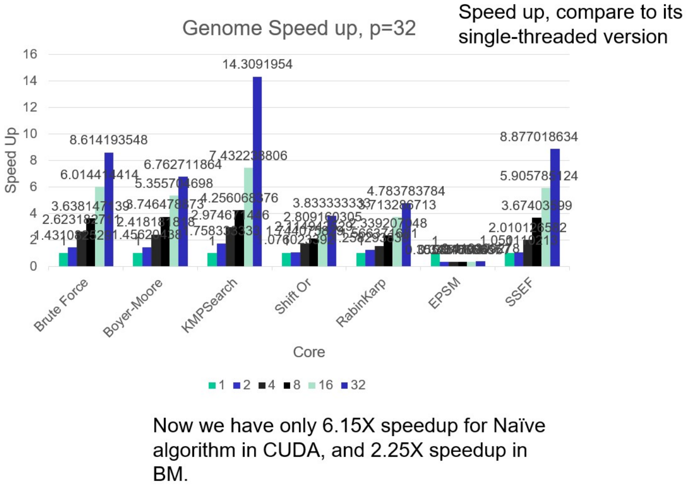
[11] H. Ishikawa et al., "PZLAST: an ultra-fast sequence similarity search tool implemented on a MIMD processor," 2021 Ninth International Symposium on Computing and Networking (CANDAR), 2021, pp. 102-107, doi: 10.1109/CANDAR53791.2021.00021.



Speed up, compare to its single-threaded version

3X speedup for Naïve algorithm in CUDA, and 1.8X speedup in KMP.

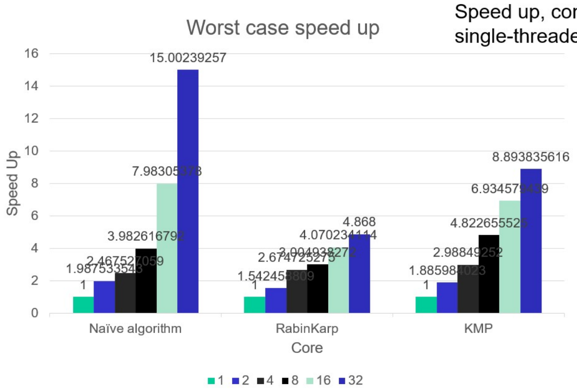
Figure 8: Runtime of seven algorithms for 1,2,4, 8, 16, and 32 threads. natural language text, pattern length of 1.



Speed up, compare to its single-threaded version

Now we have only 6.15X speedup for Naïve algorithm in CUDA, and 2.25X speedup in BM.

Figure 9: Runtime of seven algorithms for 1,2,4, 8, 16, and 32 threads. Genome data set, complex generated data sett, pattern length of 32.



Speed up, compare to its single-threaded version

There’s 2500X speedup for Naïve algorithm in CUDA, and 30X of KMP

Figure 10: Runtime of seven algorithms for 1,2,4, 8, 16, and 32 threads. Genome data set, pattern length of 32.