SPEEDING UP INDEX CONSTRUCTION WITH GPU FOR DNA DATA SEQUENCES

Rahmaddiansyah[1] and Nur'aini Abdul Rashid[2]

[1] *Universiti Sains Malaysia (USM), Malaysia, new_rahmad@yahoo.co.id*
[2] *Universiti Sains Malaysia (USM), Malaysia, nuraini@cs.usm.my*

ABSTRACT. The advancement of technology in scientific community has produced terabytes of biological data. This datum includes DNA sequences. String matching algorithm which is traditionally used to match DNA sequences now takes much longer time to execute because of the large size of DNA data and also the small number of alphabets. To overcome this problem, the indexing methods such as suffix arrays or suffix trees have been introduced. In this study we used suffix arrays as indexing algorithm because it is more applicable, not complex and used less space compared to suffix trees. The parallel method is then introduced to speed up the index construction process. Graphic processor unit (GPU) is used to parallelize a segment of an indexing algorithm. In this research, we used a GPU to parallelize the sorting part of suffix array construction algorithm. Our results show that the GPU is able to accelerate the process of building the index of the suffix array by 1.68 times faster than without GPU.

Keywords: Indexing technique, Graphic Processor Unit (GPU), Speed up, DNA sequences.

INTRODUCTION

String matching is an essential process for some computer applications. String matching is the process of finding the existence and positions of a pattern within a longer string or text. The applications of string matching process include spelling checker, parser, validating of id and password and many others. It is also a basic operation in areas like information retrieval, pattern recognition, data compression, network security and others. Because of the vast application of string matching and the rapid increased of data size, string matching is an active area in even until to date.

There are two common techniques used to string matching process, that are lookup tables such as direct-address tables or hash tables, and text preprocessing or indexing such as suffix tree (ST) or suffix array (SA). The advantage of lookup table is that it is faster to access a number from a list than to compute the number. However, lookup tables need larger memory space because of the extra variable needed to track all numbers and stored unused numbers. Therefore, most fast string matching algorithm pre-process the text to facilitate faster searching.

Developments in the molecular biology techniques lead to the increasing number of genomic and proteomic data. The size of GenBank and its collaborating DNA and protein databases which contain data coded as long strings has reached 100 Giga bases , doubles every 17 months. It has become critical for researchers to develop effective data structure and efficient algorithms and also using sophisticated technology equipment for storing, querying, and analyzing these data.

Recent graphics architectures provide tremendous memory bandwidth and computational horsepower. For example, the NVIDIA GeForce 6800 Ultra can achieve a sustained 35.2

GB/sec of memory bandwidth. Performance of the graphics hardware increases more rapidly than CPUs.

This paper offers a faster suffix array construction algorithm using GPU. To construct index quickly, we adopt the algorithm proposed by (Karkkainen & Sanders et al., 2006) that represents a reliable indexing technique with time complexity of $O(n)$. Sorting technique is important part of this algorithm, effort to increase the speed of sorting on this algorithm is done by replacing sequential radix sort with parallel radix sort using GPU (Satish, Harris et al. 2009).

RELATED WORKS

Several index structures for sequence data has been introduced, including PATRICIA trees (Morrison,1968), inverted files (Weiner,1973), prefix index (Jagadish et al. ,2000), String B-Tree (Ferragina and Grossi, 1999), q-grams (Burkhardt et al.,1999), suffix trees (ST) (Weiner ,1973), and suffix arrays (SA) (Manber and Myers, 1990). Biological data sequence, a special type of string, do not have a proper structure whereby it cannot be segmented into meaningful terms. Some of the existing data structure such as inverted files, prefix index and string B-Tree, which are efficient to natural language strings, are not applicable to biological sequences. That also applies to q-gram, which plays an important role in fast exact string matching algorithms, is not suitable for low similarity search as in pattern search(. Thus, there is an increasing interest on ST and SA as desirable index structures to support a wide range of applications on biological sequence data.

Weiner proposed the idea of suffix trees in 1973 (Weiner,1973). McCreight in 1976 reported an efficient but complex algorithm that builds a suffix tree in a time proportional to the length of the input string (Edward,1976). Ukkonen gave another simpler linear-time algorithm for this purpose in 1995(Ukkonen,1995). Although suffix trees are fundamental to string processing, they are not widely used in practical software programs because they involve extensive space usage. In 1990, Manber and Myers (Manber and Myers,1990) invented suffix arrays, which have been widely accepted as a space-efficient alternative to suffix trees. An experimental comparison of many suffix array construction algorithms are presented in (Puglisi, Smyth et al. ,2007). The best algorithms in the comparison is the algorithm by Maniscalco and Puglisi(Maniscalco and Puglisi, 2006) which is the fastest but has an $\Omega(n^2)$ worst-case complexity, and a variant of the algorithm by Burkhardt and Karkkainen (Burkhardt and K¨arkk¨ainen ,2003). Some other researcher focused on using faster computer and parallelism for example: using 128 processor for a scalable parallel suffix array construction (Kulla and Sanders ,2007), using PC cluster for a parallel construction of large suffix (Chen and Schmidt, 2005), and parallel/distributed external-memory suffix tree construction introduced by (Gao and Zaki, 2008).

BASIC SUFFIX ARRAY CONSTRUCTION (Kasahara and Morishita, 2006)

Definition 1: Let S denote the target string $b_0 b_1 ... b_{n-1}$ of length n. The *i-th* element of S is described by $S[i]$. The substring of S that ranges from the left position l*-th* to the right position r*-th*, $b_l ... b_r$, is denoted by $S[l,r]$, where $l, r \in [0, n-1]$. A *prefix* is a substring starting from the 0-th position, $S[0,r]$, while a *suffix* is a substring ending at the last position, $S[l, n-1]$.

Prefix and suffix are proper if they are shorter than original string.

Example 1 Let S denote ATAATACGATAATAA. In the following table, the left half shows proper prefixes of S, while the right presents proper suffixes of S.

S [0,0] = **A** S [10,14]= **AATAA**
S [0,1] = **AT** S [11,14]= **ATAA**
S [0,2] = **ATA** S [12,14]= **TAA**

Definition 2: Let S be a string of length *n*. A *suffix array* SA of S is an array of lexicographically sorted suffixes of S such that SA [i] = k if and only if the *i-th* suffix in the lexicographic order starts at position k in S. An *inverse suffix array* ISA is such an array that ISA [k] = i if and only if SA [i]=k. In other words, the suffix starting at k position in S has rank ISA [k] in lexicographic order. We assume that both SA and ISA have zero-origin indexing.

SUFFIX ARRAY CONSTRUCTION ALGORITHM

Linear-Time Suffix Array Construction (LSAC)

Linear-Time Suffix Array Construction (LSAC) algorithm adopt the idea of the divide-and-conquer approach (Karkkainen, Sanders et al. 2006) . Given an input string of length n, this algorithm builds the suffix array of 2n/3 suffixes. Let T (n) denote the computation time of the overall execution. The recursive call takes T (2n/3), and hence we have the recurrence T (n) = T (2n/3) + cn. Solving this gives T (n) ≈3cn, and therefore the time complexity of the algorithm is $O(n)$.

Figure 1 represents operation of Karkkainen-Sanders (LSAC) algorithm. S12 [j] describes the starting position i= f (j) for the suffix in the input S . The first half stores such indexes that i mod 3 = 1, e.g., 1, 4, 7, and the latter half contains i mod 3 = 2, e.g., 2, 5, 8… We then radix sort first triplets of individual suffixes so that all the triplets are ranked and put into rankl2. Then we replace the starting positions of suffixes in S12 with ranks of the first triplets of suffixes. For example, the first triplet TAA in the suffix starting at position 1 of S is ranked 7 among all the triplets, and hence 7 is assigned to S12 [0]. This replacement transforms S12 into the list of ranks of triplets, 7863234551.

If all triplets are ranked differently, it is almost straightforward to order suffixes in S12 according to the ranks of triplets; however, more than one triplet can have the same rank. In this case, we call the algorithm recursively to generate the suffix array 9435687201 for string 7863234551. Since the length of 7863234551 is 10, which is two-thirds the length of the input S , the original problem is partitioned into a smaller problem.

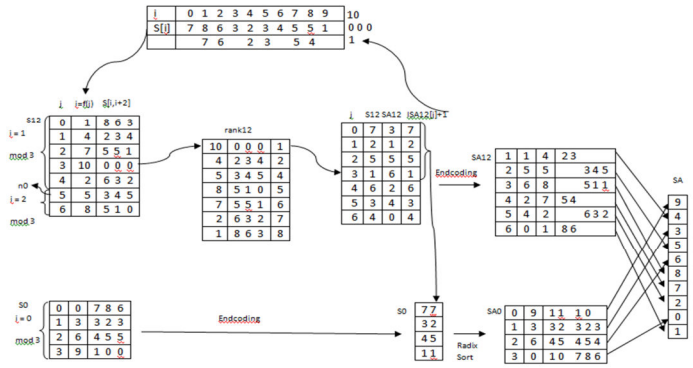

Figure 1. Operation kärkkäinen-Sanders algorithm (Karkkainen, Sanders et al. 2006).

Next, we consider how to build the suffix array of suffixes that start at position i mod 3 = 0. First, the starting positions of these suffixes are put into S0 . We then treat each suffix as the pair of its first element and the rank of the suffix next to the first element according to the

order defined by $SA12$. For instance, the suffix starting at position 0 is represented by A9, because its first element is A and its next suffix is ranked 9. This encoding allows us to radix sort pairs for suffixes in SO to build the suffix array $SA0$ for $S0$, which takes time proportional to the size of $S0$. Having the two sorted lists of suffixes in $SA12$ and $SA0$, the final step is to merge them.

Having the two sorted lists of suffixes in SA12 and SA0, the final step is to merge them. Care must be taken to compare the two elements in these two lists. A suffix in SA12 that starts at x can be represented in one of the following two forms:

- If x mod 3 = 1, the suffix starting at x+1 is ranked, making it possible to represent the suffix as the pair of the first element and the rank of the following suffix, e.g., Al for x=13.
- If x mod 3 = 2, the suffix starting at x+1 is not ranked because x+1 mod 3 = 0. We herefore denote the suffix by the triplet of the first and second elements and the rank of the following suffix, e.g., 128 for x=5.

In the former case, a suffix cannot be represented by the triplet because the rank of the suffix starting from x+2 is missing; e.g., consider the case when x=l. Similarly, in the latter case, we cannot express a suffix by a pair. However, a suffix in SA0 can be denoted by both representations. For example, the suffix starting at 0 can be expressed by either A9 or AT4, making it possible to compare it to any suffix in SA12. Merging the two sorted lists yields the suffix array of the original input string in S. This step is straightforward and takes time proportional to the sum of the lengths of the two lists.

GPU Radix Sort

Satish, Harris and Garland (Satish, Harris et al. 2009) parallelized the radix sort on CUDA by dividing the sequences into p blocks for each thread. Their method focuses on utilizing the memory bandwidth into two ways, minimizing communication with global memory and maximizing synchronizing of scatters. The first goal was accomplished by partitioning the data and limits the size of digit b > 1. The second goal was achieved by locally sort the partitioned data using on-chip shared memory.

Our propose algorithm.

The input to our algorithm is the DNA text file. The steps are given as follows:

1) Convert DNA alphabet as number (A=1, C=2, G=3, T=4).
2) Give three digit values 0 at the end of sequence.
3) Divide suffixes into two classes, e.g., suffixes starting at positions i mod 3 ≠ 0 and the others
4) Construct the suffix array of the first class by using GPU radix sort of the first triplets of individual suffixes, so that all the triplets are ranked and put into histogram. If more than one triplet have the same rank, repeat steps 3 and 4.
5) Use the result in step 4 to radix sort pairs for suffixes in second class to build the suffix array
6) Merge the two suffix arrays into one, and return the result.

The above algorithm is designed as heterogeneous computing. Construction suffix array is done in sequential, whereas the sorting process, step 4 and 5, that is the most compute intensive parts are done in parallel using GPU. In sorting process, we adopt the method by (Satish, Harris et al. 2009) whereby CUDA kernels are executed by blocks of t = 256 threads each, processing 4 elements per thread or 1024 elements per block. Since each block will process a tile of 1024 elements, we use $P = [n/1024]$ blocks in our computations.

IMPLEMENTATION AND RESULT

In this section, we describe the experiments carried to evaluate performance of the indexing techniques, and compare them based on their construction times. We used a standard desktop computer with AMD Phenom-II810 2.6GHz Quad-Core processor, Dual Channel 4GB (2x2GB) DDR2-800 Memory, Tesla C2050 Cards (GPU), Ubuntu 8.04 64-bit Operating System. We used the C++ source code to implement sequential radix sort and LSAC algorithm, CUDA source code and Thrust libraries (Seward 2000) to implement GPU radix sort and our algorithm.

Sort Performance

Before implementation of our algorithm, we test performance of sequential radix sort and of GPU radix sort. To conduct this experiments, we use some input data which randomly chosen consisting of four alphabet.

Table 1. Construction time and Speedup data.

N	Gpu Radix Sort Second)	Sequential sort (second)	Speedup
2^{19}	0.08	0.02	0.25
2^{20}	0.09	0.04	0.44
2^{21}	0.09	0.08	0.89
2^{22}	0.1	0.14	1.40
2^{23}	0.12	0.29	2.42
2^{24}	0.15	0.57	3.80
2^{25}	0.22	1.16	5.27
2^{26}	0.35	2.32	6.63
2^{27}	0.63	4.62	7.33
2^{28}	1.16	9.24	7.97

For the number of data that is smaller than 2^{21}, sequential radix sort is faster than the GPU radix sort. GPU radix sort outperforms sequential radix sort when the size of data is larger than 2^{21}.

Table 1 present the time of sorting. The speed up of GPU radix sort is 7.93 times faster compared to the sequential sort. These results suggest that adopting GPU sort algorithm with GPU to construct DNA data using the LSAC algorithm can significantly increase the speed indexing construction. To prove this hypothesis, we carried out our algorithm in constructing SA using actual DNA data as input data.

We use 25 Homosapiens (DNA) data, which is presented in fasta. We removed the header lines, new line symbols, and blanks from original data.

Figure 4 show that our algorithm works well in constructing SA for all the DNA data that we took as sample. It also shows the speed up of the construction algorithm of SA is 0.98 to 1.68 times compared to the original LSAC algorithm.

Implementation Sequential and Linear-Time Suffix Array Construction with GPU for DNA data sequences

Figure 4. the graph shows the performance of LSAC and LSAC

CONCLUSION

Base on the results, we conclude that the GPU is a device that can potentially to improve the performance of suffix array constructing algorithm. Selection sorting task can be processed in parallel with GPU, because it provides a significant effect on construction time, where it can accelerate the process to 1.68 times. Indeed, acceleration of suffix array construction process is much slower than acceleration of GPU sorting which reached 7.93 times. This is because to construct SA of a DNA data that has characteristics of the loop alphabet; require sorting process in multiple times to form an unique ranking. Plus in the process of formation of ranking the number of data in sorting process shrinks if looping is common. Therefore, it reduces performance because the GPU radix sort is low performance when sorting small data as shown in Fig.3 In other words if the GPUs are involved in the processing of data, problems handling large bioinformatics data quickly can be resolved by using the GPU.

ACKNOWLEDGEMENT

We would like to acknowledge the School of Computer Science APEX GRANT for supporting the research.

REFERENCES

. "GenBank." from http://www.ncbi.nlm.nih.gov/Genbank/index.html.

. "NCBI: National Center for Biotechnology Information." from http://www.ncbi.nlm.nih.gov.

Burkhardt, S., A. Crauser, et al. (1999). Q-gram based database searching using a suffix array (QUASAR). Proceedings of the third annual international conference on Computational molecular biology. Lyon, France, ACM.

Burkhardt, S. and J. K¨arkk¨ainen (2003). "Fast Lightweight Suffix Array Construction and Checking." Proc. 14th Annual Symposium on Combinatorial Pattern Matching. LNCS 2676: 55–69.

Chen, C. and B. Schmidt (2005). "Parallel Construction of Large Suffix Trees on a PC Cluster." Euro-Par 2005 Parallel Processing: 1227-1236.

Edward, M. M. (1976). "A Space-Economical Suffix Tree Construction Algorithm." J. ACM 23(2): 262-272.

Ferragina, P. and R. Grossi (1999). "The string B-tree: a new data structure for string search in external memory and its applications." J. ACM 46(2): 236-280.

Gao, F. and M. J. Zaki (2008). "PSIST: A scalable approach to indexing protein structures using suffix trees." Journal of Parallel and Distributed Computing 68(1): 54-63.

Jagadish, H. V., N. Koudas, et al. (2000). On effective multi-dimensional indexing for strings. Proceedings of the 2000 ACM SIGMOD international conference on Management of data. Dallas, Texas, United States, ACM.

Karkkainen, J., P. Sanders, et al. (2006). "Linear work suffix array construction." J. ACM 53(6): 918-936.

Kasahara, M. and S. Morishita (2006). Large-Scale Genome Sequences Processing, Imperial College Press 57 Shelton Street Covent Garden London WC2H 9HE.

Kulla, F. and P. Sanders (2007). "Scalable parallel suffix array construction." Parallel Computing 33(9): 605-612.

Manber, U. and G. Myers (1990). Suffix arrays: a new method for on-line string searches. Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms. San Francisco, California, United States, Society for Industrial and Applied Mathematics.

Maniscalco, M. A. and S. J. Puglisi (2006). "Faster lightweight suffix array construction." In: Proc. 17th Australasian Workshop on Combinatorial Algorithms, **16–29**.

Morrison, D. R. (1968). "PATRICIA - Practical Algorithm to Retrieve Information Coded in

Alphanumeric."

Navarro, G. (2001), *A guided tour to approximate string matching.* ACM Comput. Surv.,. 33(1): p. 31-88.

Puglisi, S. J., W. F. Smyth, et al. (2007). "A taxonomy of suffix array construction algorithms." ACM Comput. Surv. 39(2): 4.

Satish, N., M. Harris, et al. (2009). Designing efficient sorting algorithms for manycore GPUs. Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.

Seward, J. (2000). On the performance of BWT sorting algorithms. Data Compression Conference, 2000. Proceedings. DCC 2000.

Ukkonen, E. (1995). "On-line construction of suffix trees." Algorithmica **14**(3): 249-260

Weiner, P. (1973). Linear pattern matching algorithms. Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory.