Burrow Wheeler Transform in Parallel

Ruhua Jiang, Jingwen Pei

***Abstract*—Burrow Wheeler Transform (BWT) of a text is a reversible transformation of the text. It was originally applied to text compression, by observation that the text after BWT can be more compressible than the original text. Recently, BWT based method plays a center role in bioinformatics applications, especially the reads mapping. In reads mapping, a BWT based FM-index of reference genome is constructed first, and millions or hundreds of millions of DNA reads are mapped to the reference by so called backward search. The popularity of BWT based method in reads mapping is mainly due to its excellence in space efficiency, good searching performance and the flexibility of supporting different k while doing k-mer searching, comparing the indexing method like hashing, or even suffix tree or suffix array.**

To achieve high performance in terms of memory consumption and running time in creating BWT is non-trival. Naively, one can build BWT by first build the full suffix array, and then construct the BWT based on suffix array, but it turns out to be very costly in memory and time. Researchers have made extensive effort to improve the performance of building BWT, and there are bunch of works proposed. Most of them are sequential algorithms, however. Here, we propose a parallel approach to build the BWT , more specifically, we use CUDA programming framework which leveraging the power of graphics processing unit, to speedup a sequential BWT construction algorithm that based on block wise suffix sorting. We compared with the traditional sequential BWT building program and the speedup is presented.

***Index Terms*—Burrow Wheeler Transform, GPU, CUDA.**

Code avaliable in https://code.google.com/p/cudabwt/

I. Introduction

BURROW and Wheeler first proposed BWT [1] as a block-sorting, lossless data algorithm in 1994. The method applies a reversible transformation to an input string to form a brand new string that is consisted of the same characters of the original string. It is a technique that makes compression be achieved within a percent, which is at the same level with statistical modeling techniques but significantly efficient. The advantage of BWT is that the modified string is much easier to compress by simple compression algorithm. After reversible transformation, same characters tend to be grouped together. As a result, there is a great chance to find a character close to another instance of the same character. This kind of text can be easily compressed by using simple algorithms. Bzip2 [2] is one of the most well-known program that is based on BWT method to do compression.

Recently, due to its short running time for search, which proportional to the query string length, reasonable memory footprint and flexibility of supporting different k while doing k-mer search, BWT based index and search becomes almost the most popular method in the reads mapping application, the most well known programs are BWA[3] and Bowtie [4], while another one is hashing. The reference genome is usually indexed using BWT based FM-index or the hashing based index first, and tens of thousands of reads are mapped to the reference usually by applying a seed-and-extend heuristic strategy. The seeding procedure here means finding the exact match (sometimes allow certain mismatch) by using the index information, and preliminary locates the position in the reference genome that reads may comes from. After that, a extend procedure, usually a dynamic programming subroutine, will be called to align the rest of the reads other than the seed region. Hashing can achieve almost constant look-up, however, the memory consumption is large, and the length of seed often is fixed, which makes it not so flexible. For human genome sequences indexing, the current method based on CPU can take several hours. One may argue that indexing procedure only take once, after that it can be used over and over. However, recently announced Gnome 10K project aims to sequence and assemble genomes of 10,000 vertebrate organisms[5], and each of them need be indexed before do any downstream analysis. Moreover, study in [6] also shows that BWT based indexing can also be useful for reads indexing in some application, and the most notable one is the string-graph approach to de novo assembly of short reads. Thus, how to construct BWT effectively is a critical research topic that attracts lots of efforts.

To briefly illustrate how BWT works, let X denote the input string with N characters. N rotations are first formed by cyclically shifting the characters of X, and then sort them lexicographically. To better understand, a N by N matrix M can be formed, whose rows are the rotations of X in a lexicographical order of the first letter of each rotation. According to the order, a new string L is formed by extracting the last character of each rotation, whose i-th character in L is the last character of the i-th rotation, that is $L[0]$, $L[1]$, $\cdots$, $L[N-1]$ are $M[0, N-1]$, $M[1, N-1]$, $\cdots$, $M[N-1, N-1]$ respectively. Let I denote the index of the original string X in N rotations. L together with I is used to save the complete information of string X. Here, L is the BWT of original string X.

Our project are mainly focus on making the algorithm proposed in [7] be parallel. There do exist some parallel version of BWT construction algorithm. Authors of [11] propose to use Mapreduce programming model to speedup the BWT construction. And [12] even did some preliminary study on to how to parallelize the BWT construction on GPU, which is similar to what we are going to do here.

II. Methods

A. Burrow-Wheeler Transform

Let Σ denote an alphabet, X denote the input string with N characters that belong to alphabet Σ. X can be written as $X = x_0x_1 \cdots x_{n-1}$, N possible suffixes can be determined based on X. For example, if $X = BANANA$, then all possible suffixes are $BANANA$, $ANANA$, $NANA$, ANA, NA, and A. Attach each suffixes with symbol \$ and its corresponding prefix to form the rotating strings, and then sort all rotations lexicographically. Suffix array SA is defined as a permutation of the integer 0, 1, $\cdots$, $N-1$, where $SA(i)$ denotes the start position of the i-th smallest suffix, $SA = (6, 5, 3, 1, 0, 4, 2)$ in this case. Here \$ is not an element from alphabet Σ, so the string started with \$ comes first in sorting process. The last column of the rotations is called BWT, it equals $ANNB\$AA$ in this case. After strings sorting, the relationship between suffix array SA and BWT can be easily constructed:

$$BWT[i] = \begin{cases} X[SA[i]-1] & SA[i] > 0 \\ X[n] & SA[i] = 0 \end{cases}$$

If we construct the BWT in the way that we described above, it certainly can not be very efficient. Normally, people construct BWT from suffix array, which is a compact representation of a suffix tree as we describe above, but in a more clever way. There are numerous algorithms for constructing suffix arrays. The theoretically best ones work in linear time [8], [9], [10], there still exists several so-called light weight algorithms that need little space in addition to the text and the suffix array, and some of them are very fast. However, all of them need to have at least the full suffix array in memory. [7]proposed a new way that suffix array is computed in smaller block, and we can compute block of BWT then. Its single processor CPU implementation turn outs to be little slower than the fast algorithm, however, the memory usage is only one third of the previous algorithm, which can be certainly good if we want to apply it to GPU, since memory of GPU is very limited comparing the main memory.

B. GPU-accelerated Burrow-Wheeler Transform

In this project, we studied and implemented the parallel algorithm for BWT proposed by [12]. In BWT, array BWT is obtained by using Suffix array. However, to get sorted suffix array requires a large amount of memory space to store which is an obstacle when designing parallel algorithm. In order to design a efficient parallel algorithm, Fast BWT proposed by [7] is considered as the main structure of the algorithm. The author proposed a method called blockwise suffix sorting. The main structure can be described as follows:

(1) Choose a random sample set from X as splitters.

(2) Sort the set of splitter suffixes. Let j_1, j_2, $\cdots$, j_{r-1} be the elements ordered such that $S_{j_1} < S_{j_2} < \cdots < S_{j_{r-1}}$.

(3) For each $k \in [0, r)$:

(a) for each suffix $S \in X$, determine the position of S $S_{j_k} \leq S < S_{j_{k+1}}$. Store S in suffix block B_k.

(b) Compute the block of suffix array by sorting the suffix block B_k.

(c) Compute the block BWT from the block of SA using the equation mentioned above and discard the block of SA.

The parallel algorithm is designed based on the algorithm described above. To accelerate the procedure on GPU, some different technologies are implemented in blockwise suffix sorting method. The parallel algorithm is consisted of two sections: suffix blocking on GPU and parallel suffix sorting on GPU.

a) Suffix blocking on GPU: The choice of splitters is very important for algorithm. A bad set of splitters can make algorithm time-consuming for long sequences. In this algorithm, we adopt a simple strategy for the selection of splitters. For simplicity, we group all suffixes into 4^q subsets according to their p-character prefixes. Here we assume $\Sigma = A, C, G, T$ which is usually used in genomic data. For example, when $p = 1$, it means we group all suffixes into 4 subsets, each of which has prefix A, C, G, and T respectively. If the size of a suffix block exceeds a threshold, then we can simply divide such suffix block into 4 smaller blocks according to $p+1$-character prefixes. This procedure can be repeated when suffix block exceeds until the size of block satisfy our threshold. Given a p-character prefix X and sequence T of length n, we want to output a suffix block of which all suffixes have the same prefix with X. The parallel algorithm of suffix blocking on GPU can be described as the following:

```
1:  Allocate an array $B$ to store the suffix block
2:  Allocate an counting array, $A[m]$, $m$ is the number of threads
3:  $idx \leftarrow threadIdx.x + threadIdx.y \times blockSize$ {compute thread id}
4:  $l = n/m$ {Each thread scan $l$ consecutive suffixes of $T$}
5:  **for** all threads inside thread block **do**
6:      from the position $idx \times l$ to scan $T$, compute the number $p$ of suffixes that have the same prefix with $X$
7:      $A[idx] = p$
8:  **end for**
9:  syncthreads()
10: compute the prefix sum of $A$ using parallel scan primitive
11: syncthreads()
12: **for** all threads inside thread block **do**
13:     from the position $idx \times l$ to scan $T$
14:     **if** a suffix in position $a$ is $b$-th suffix that has the same prefix with $X$ **then**
15:         $B[A[idx] + b] = a + idx \times l$
16:     **end if**
17: **end for**
```

b) Parallel suffix sorting on GPU: After we get a suffix block, the suffix array of such block needs to be calculated. In this part, radix sortin algorithm on GPU is applied to sort suffix blocks on GPU. To achieve that, we assume sequence X has no repetitions longer than v. As a result, we only need to sort their first $v+1$ characters instead of the whole sequence. As the first p characters are used as splitter, they are the same in one block. So the only part that needs to be sorted is $S = s_0 s_1 \cdots s_{v-p}$. Sequence string S can be easily converted to an integer using the following function:

$$f(S) = \sum_{i=0}^{p} g(s_i) \times 4^{v-i}$$

After converting each sequence into an integer, the GPU-based radix sorting algorithm can be used to sort suffix blocks.

Given a sequence X of length n and parameter p, output the BWT of the sequence X. The whole algorithm can be described as following:

```
1: Copy sequence $X$ to the global memory of GPU
2: Enumerate all $p$-character prefixes in ascending order and store to an array $P$
3: **for** $i = 0$ to $4^p$ **do**
4:     Execute parallel suffix blocking on GPU to generate a suffix block $B$, all suffixes in the suffix block have the same prefixes with $P[i]$
5:     Execute parallel suffix sorting on GPU to generate a block of suffix array $SA_i$
6:     Copy $SA_i$ to main memory
7:     Compute $BWT_i$
8: **end for**
9: Output the $BWT$ of $X$
```

III. Implementation

A. GPU architecture and CUDA framework

Traditionally, most of the applications are written in sequential program. So the design of microprocessors based on single central processing unit is optimized to reduce the latency of single thread. Algthough nowdays several CPU have multiple cores, the number of threads available are still limited. While the design philosophy of GPU totally different, it is optimized for throughput, with tens of thousands of threads are available. Here we implement the algorithm based on CUDA, which is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the GPU. Figure 1 gives an overview architecture of a modern CUDA-capable GPU[14]. More specifically, we use CUDA C here. CUDA C is an extension to the popular C programming language with new keywords and application programming interfaces for programmers to take advantage of heterogeneous computing systems that contain both CPUs and massively parallel GPUs [14].

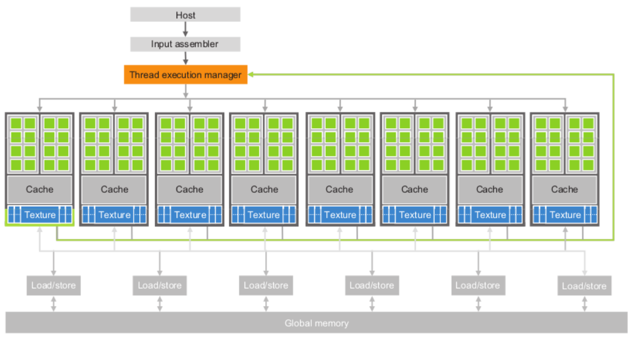

Fig. 1. Architecture of a CUDA-capable GPU

In terms of computation hierarchy, there are three layers in CUDA. Thread, block and grid. A grid consist blocks, and each blocks consist plenty threads, usually maximum number is 1024 threads per block. There are also three layers in terms of memory hierarchy. The GPU itself has a global memory. It can be accessed by all the threads. Since blocks are independent, the communication between threads in different blocks needs go through the global GPU memory. There is a shared memory can be accessed by all the threads in the same block. Shared memory usually much smaller than the global memory, but much faster. For each thread, it also has its own memory called local memory. It is usually extremely limited but extremely fast.

B. Development environment and library

We use a IDE provided by NIVIDA called nsight in this project. It provides powerful debugging and profiling functionality that enable us to fully optimize the performance of the CPU and GPU. An Amazon Elastic Compute Cloud (Amazon EC2) GPU instance is used for developing, testing and running the experiments. For how to configure the instance, refer to [16] for more details. We use a fasta file parser written by Heng Li to parser the fasta file. For characters like N, the parser randomly replace it by one of $\Sigma = A, C, G, T$. We use Thrust [15] library do radix sort that we mentioned above. The current version of radix sort in Thrust library is based on that implementation that can be found in https://code.google.com/p/back40computing/wiki/RadixSorting. It is an implementation dedicated for CUDA. Our code is available in https://code.google.com/p/cudabwt/ . It uses GNU GPL v2 license.

IV. Results

We firstly try to evaluate the speedup against the original sequential algorithm proposed by [7] (It can be downloaded from https://code.google.com/p/dcs-bwt-compressor/), however the code has several issues, and we have try several ways to fix it but still cannot compile it. So we unable to collect the running time for that program. Instead of using program above, we use BWA [3] to index the the DNA sequence file. BWA use the IS program originally proposed by Nong Ge [17] at the Sun Yat-Sen University and implemented by Yuta Mori. It is a linear time algorithm, while the original sequential algorithm our project based on has a worst complexity of O(nlogn) [7], so it is quite unfair for us here.

The GPU is NIDVIDA Tesla S2050, and the CPU is CPU Intel(R) Core(TM) i5-3210M @ 2.50GHz. In our experiment we make p = 2, and v = 16 + p. Table I lists out running time for different text files, whose size ranges from $6M$ to $60M$. Time(CPU) represents the running time of sequential algorithm. Time(GPU) represents the running time of parallel algorithm. To further illustrate relationship between speed-up and text file size, fig 2 is plotted. Table II shows the speed-up value when using different number of threads and blocks. In the table, the size of text size is fixed, T/B represents the number of threads and blocks being used, and speed-up value is computed using the sequential running time in table I.

From table I and figure 2 we can see the speedup increases with the increasing of the data size. However, the speed (or derivative) of speedup are decreasing from data size 12M. It

Size	Time(CPU)	Time(GPU)	Speed up
$6M$	$16.648s$	$14.377s$	1.158
$12M$	$31.990s$	$26.333s$	1.215
$24M$	$72.896s$	$47.611s$	1.531
$47M$	$158.682s$	$94.218s$	1.684
$60M$	$204.646s$	$119.103s$	1.718

TABLE I
SPEED UP FOR DIFFERENT SIZE OF TEXT

	$T/B = 1024$	$T/B = 512$	$T/B = 256$
$60M$	119.103	$118.98s$	$120.784s$
Speed up	1.72	1.72	1.69

TABLE II
SPEED UP FOR DIFFERENT NUMBER OF THREADS

shows that the increasing of data size, we gain more computation reduction than the memory access overhead. However, that gaining will not always keep if we increase the data size to infinity. Table II shows that by decreasing the number of threads used in GPU, the running time increase, however, not significantly. by decreasing the threads number, although the fraction of region of each thread taking care of is increasing proportionally, however, since the data size is not so large,the absolute region do not increase too much. So, it is reasonable that it does not slow down significantly.

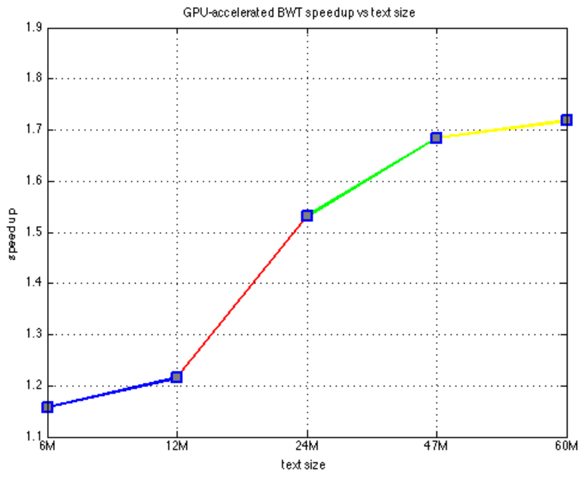

Fig. 2. Speed up vs text size

V. Conclusion and Future Work

We implement a parallel BWT construction algorithm for DNA sequence, and achieve resonable speedup against the state-of-the-art BWT construction algorithm. Our method assume that suffixs are distinguishable using its first v letters. In our experiment, $v = 18$. For human genome single chromosome, around 90% of 18mer are unique. To make our program perfectly applicable to all kinds of human genome data, we could add a refinement procedure in the future.

References

[1] Burrows, Michael; Wheeler, David J. (1994), *A block sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation

[2] http://www.bzip.org/

[3] Li H. and Durbin R. (2009) *Fast and accurate short read alignment with Burrows-Wheeler Transform*. Bioinformatics, 25:1754-60. [PMID: 19451168]

[4] Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 2009;10(3):R25.

[5] http:genome10k.org

[6] Chi-Man Liu, Ruibang Luo, Tak-Wah Lam, *GPU-Accelerated BWT Construction for Large Collection of Short Reads*, arXiv:1401.7457

[7] Juha Karkkainen, *Fast BWT in Small Space by Blockwise Sux Sorting*, Theoretical Computer Science 387(3)(2007)249-257

[8] J. Karkkainen, P. Sanders, S. Burkhardt, *Linear work sux array construction*, J. ACM 53 (6) (2006) 918936.

[9] D. K. Kim, J. S. Sim, H. Park, K. Park, *Constructing sux arrays in linear time*, J. Discrete Algorithms 3 (24) (2005) 126142.

[10] P. Ko, S. Aluru, *Space ecient linear time construction of sux arrays*, J. Discrete Algorithms 3 (24) (2005) 143156

[11] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz. 2011. *Rapid parallel genome indexing with MapReduce.* In Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11). ACM, New York, NY, USA, 51-58.

[12] Zhiheng Zhao,Jianping Yin, Wei Xiong*GPU-accelerated Burrow-Wheeler Transform for genomic data*,Biomedical Engineering and Informatics (BMEI), 2012 5th International Conference

[13] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, *Scan primitives for GPU computing*, in Graphics Hardware 2007, Aug. 2007, pp. 97-106.

[14] David Kirkand and Wen-Mei Hwu. 2012. *Programming Massively Parallel Processors: A Hands-on Approach*, Second Edition, Elsevier Science Technology. (ISBN: 978-0-12-415992-1)

[15] https://code.google.com/p/thrust

[16] https://aws.amazon.com/articles/7249489223918169

[17] G. Nong, S. Zhang, W. H. Chan, *Linear suffix array construction by almost pure inducedsorting* In: Proceedings of DCC, 2009.