picocomputing

now you can

White Paper

Inexact Search Acceleration on FPGAs Using the Burrows-Wheeler Transform

Greg Edvenson, Corey Olson, Paul Draghicescu

Pico Computing, Inc.

www.picocomputing.com · 206-283-2178 · 506 2nd Ave, Suite 1300, Seattle, WA 98104

Inexact Search Acceleration on FPGAs Using the Burrows-Wheeler Transform

Paul Draghicescu, Greg Edvenson, and Corey Olson
Pico Computing, Inc.
Seattle, Washington 98109
{pauld,greg,corey}@picocomputing.com

***Abstract*—Inexact search is a difficult and time-consuming task with widespread application. Acceleration of inexact search could have tremendous impact upon fields such as chemistry, meteorology, and even bioinformatics. Field-programmable gate arrays provide a means by which to accelerate this process. We demonstrate the acceleration of inexact search using the Burrows-Wheeler transform on FPGAs using the short read mapping problem in bioinformatics as an example application. Using 12 FPGAs, we are able to accelerate the search of 100-base short reads against the human genome by 48X.**

I. Introduction

Search is a widespread area, with many more applications than simply Bing or Google. Not only has Google built an empire searching the internet based upon user input, but meteorologists have been searching for weather patterns, chemists have been searching for the weights and types of atoms contained in an unknown substance, and biologists have been searching for patterns in our genetic code.

This application space involves searching through a very large data set, such as the set of webpages on the internet, which we call a *reference*. The item that is being searched for, called the *query*, may or may not match the reference exactly at some point. In fact the most interesting cases tend to be when the query is close to the reference but does not match exactly (e.g. a search on Google for *Super Computer* returns links to pages containing the word *Supercomputer*). Therefore, it is important to support *inexact matching* of the query to the reference. Inexact matching tends to require a brute-force approach, resulting in much longer search times. In many cases, researchers are searching a very large reference database (or many databases) millions or even billions of times; speed is of the utmost importance.

II. Methods

A. Burrows-Wheeler Transform

There are many ways to do inexact searching, but one such way makes use of an algorithm called the *Burrows-Wheeler Transform* (BWT) [3]. The BWT is classically used in data compression, such as bzip2 [7]. The BWT is used for inexact searching because it has a very low memory footprint, and all locations where the query matches the reference can be found in time proportional to the length of the query.

Before we can search for a query within a reference using the BWT, we must first construct a BWT index of the reference. For the purposes of this paper, we assume the reference is a single string of characters. In practice, many types of databases can be converted to this representation for processing.

To begin the construction of the index, we first append a terminating character to the tail of the reference string, creating a reference string with length *N*. This terminating character (denoted as $) should not appear anywhere else in the reference string, and it is lexicographically less than all other characters in the reference string. We then form a square 2D matrix, where each dimension is equal to the length of the reference string (N). We place the reference string in the first row of the matrix. We then fill in every row of the matrix with every possible rotation of the reference. After the matrix has been completely filled, we lexicographically sort the rows of the matrix such that after the sort the row that begins with $ should be in the first row of the matrix. Note that once the matrix is sorted, the ith lexicographically smallest suffix of the reference string is in the ith row of the matrix. After the sort, the last column of the matrix is referred to as the *BWT string*. Figure 1 shows an example of the BWT matrix and BWT string construction for the reference string *MISSISSIPPI*.

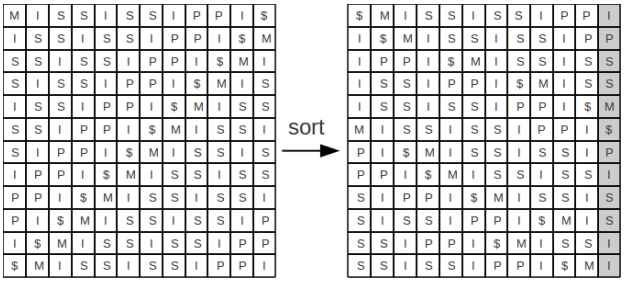


Fig. 1: Construction of the BWT matrix (left). Matrix is sorted (right) and the last column is the BWT string.

Once we have the BWT string, we must construct 2 arrays. The first array, *C(a)*, holds the number of characters in the original reference string that are lexicographically smaller than a. The second array, *O(a,i)*, stores a count of the the number of occurrences of a in the first i characters of the BWT string. Figure 2 shows an example of the C() and O(,) arrays for the reference string *MISSISSIPPI*.

After the construction of the matrix, we can use it to search in linear time for substrings of the original reference string.

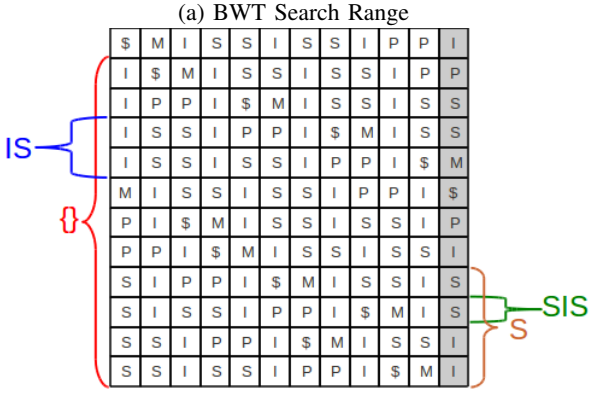
I	M	P	S
0	4	5	7

i	I	M	P	S
0	1	0	0	0
1	1	0	1	0
2	1	0	1	1
3	1	0	1	2
4	1	1	1	2
5	1	1	1	2
6	1	1	2	2
7	2	1	2	2
8	2	1	2	3
9	2	1	2	4
10	3	1	2	4
11	4	1	2	4

Fig. 2: Example of the count array (left) and the occurrence array (right) for the reference string MISSISSIPPI.

S	k	l	| SA Interval |
{}	1	11	11
S	8	11	4
IS	3	4	2
SIS	9	9	1

(a) BWT Search Range



(b) BWT Search Steps

Fig. 3: Example searching for query SIS in the reference MISSISSIPPI. Note that the size of the SA interval is non-increasing with each step in 3a.

To begin, we must realize some properties of the constructed BWT matrix. First, assuming that S is a substring of the reference string R, then all occurrences of S in R will appear in sequential rows in the BWT matrix (because the matrix is sorted lexicographically). We can therefore put a lower and upper bound on the interval of the BWT matrix where the substring S occurs, which is called the *SA interval*. The lower bound of the SA interval is denoted as k and the upper bound is denoted as l.

Burrows and Wheeler described a way to compute this interval in linear time with respect to the length of the substring. For example, suppose we start with the empty substring, $S = \{\}$. That substring can be found anywhere in the BWT matrix, so the lower bound is the first valid row of the matrix ($k = 1$) and the upper bound is the last valid row of the matrix ($l = N - 1$). If we now prepend a single base (a) from the tail of the query to the current substring (S), we can compute the new lower and upper bounds of the BWT matrix, as described by Burrows and Wheeler:

$$k = C(a) + O(a, k-1) + 1 \quad (1)$$

$$l = C(a) + O(a, l). \quad (2)$$

Using this mechanism, we can iteratively prepend a single character from the query to the current substring. As long as the computed lower and upper bounds do not cross, i.e. $k < l$, then the newly formed substring is contained within the original reference string. If we have prepended all bases of the query to the substring and the SA interval is larger than 1, i.e. $l - k > 1$, then the query is found within the reference string.

At this point, we can use another construct, which is not described in this paper, to convert the SA interval to locations in the reference string. For details on the conversion of the SA interval to reference locations, please refer to the original Burrows-Wheeler paper [3]. Figures 3a and 3b show an example of searching the reference *MISSISSIPPI* for the query *SIS*.

Please note that some details on the construction and use of the BWT have been omitted for simplicity. Full details of the construction of the BWT string and search of a substring in the reference can be found in the original Burrows-Wheeler paper [3].

B. FPGAs

Field-Programmable Gate Arrays (FPGAs) have been used for decades as accelerators for various data-intensive applications including network routing, signal processing, and even cryptography. They provide a platform for creating parallel hardware without the large cost assocatiated with building an *Application Specific Integrated Circuit* (ASIC). An FPGA comprises a sea of programmable *Lookup Tables* (LUTs), registers, programmable interconnect, and distributed *Random Access Memories* (RAMs).

Unlike other accelerators, the logic within the FPGA can be re-programmed to mirror a parallel hardware circuit. Instead of executing software, as in a *Graphics Processing Unit* (GPU), the lookup tables are programmed to compute any 6-input function, and the interconnect serves to connect resources together. FPGAs are programmed on a bit-level granularity, so they can be much more efficient than traditional compute resources when operating on data widths not equal to that of a CPU. The combination of massively parallel execution and bit-level granularity gives FPGAs a large performance and power advantage as compared to CPUs or GPUs.

Pico Computing produces a series of FPGA boards, called

the *M-Series Modules*, which can be plugged into a backplane, which is a PCI-Express card with the same form-factor as a GPU [2]. This backplane plugs into the *PCI-Express* (PCIe) slot in a motherboard. Multiple modules can be plugged into a single backplane. This system enables a host CPU to execute an application, but to offload the compute-intensive portion of the application to the FPGA. All communication to the FPGA modules from the host is via x8 Gen2 PCIe, and each module has some local DDR3 memory.

C. BWT on FPGAs

The compute-intensive portion of the BWT search application to offload to the FPGA is the traversal of the BWT index for each query. For performance reasons, assume the FPGA can store the entire index in its local DDR3 memory, which is initialized once at the beginning of the search program. For the case of Pico Computing's M505 module, the index must be 8 GB or less.

The CPU sends a query to the FPGA to be processed. The FPGA acts as a depth-first search engine for the BWT index. The FPGA will iteratively prepend a single character from the query to the current search string and compute a new index range, as shown in Section II-A. In this manner, the FPGA performs an *exact search* of the query. If the query is found within index, the final index range is reported back to the CPU, which converts that index range to a set of locations within the target. If the query is not found within the index, the FPGA reports that back to the host. To perform an *inexact search*, the host will then modify one or more characters of the query string before sending it back to the FPGA for processing. Types of modifications include substituting one character for a different one, inserting a character into the query, or deleting a character from the query. The host maintains a stack of the different variations of the query that have been sent to the FPGA.

Offloading the index searching to the FPGA enables multiple levels of parallel computation, which in turn improves the performance. First, multiple queries (or variations of the same query) can be sent to an FPGA for processing at a time. In doing so, the time to compute the upper and lower index bounds is overlapped with the time to access the index. Therefore, it is more efficient to have a single FPGA operating upon many queries at a time. The host system simply must maintain a stack (or set of stacks) for each query that is being processed by the FPGA. Second, multiple FPGAs can be added to the system to improve performance by operating upon different queries in parallel. Since the entire index is contained within the DDR3 memory of an FPGA module, all accesses to the index for a query result in local communication. This allows many FPGAs to be added to the system with a linear performance improvement.

III. Example Application

The invention of *Next-Generation Sequencing* (NGS) machines in 2007 created a form of this search problem. In conjunction with software, NGS machines can determine an individual's DNA sequence of nucleotide bases much faster and for a much lower cost than previous methods. The process involves taking a DNA sample, replicating the sample many times, chopping the DNA randomly into short fragments (about 100 bases in length), and feeding those fragments into the NGS machines. The machine then determines the nucleotide sequence of each fragment, producing a string of characters called a *short read*.

Software is then used to determine the source location of each short read in the original DNA sample. This is done by comparing each short read to a known reference genome, which is approximately 3 billion nucleotide bases and was compiled as part of the Human Genome Project. Note that since the DNA sequence varies from one individual to the next and the reference genome was compiled as an average of a few individuals, the short reads will not necessarily exactly match the reference genome. This inexact search phase is known as the short read mapping problem [6].

To put this problem into scale, Illumina, which is a manufacturer of one type of NGS machine, projects to sequence 1.2 billion short reads in approximately 27 hours [1]. Software solutions, such as the *Burrows-Wheeler Aligner* (BWA) [4] are traditionally run on CPU clusters to map these reads to the reference genome. Mapping these 1.2 billion short reads using this software solution would take approximately 24 hours to complete on a 16-core CPU cluster [5].

A. BWA Software

Our approach is to tie into the existing BWA software, allowing the CPU to perform tasks it is optimized for, such as file handling and memory management. We use the method described in Section II-C to offload the compute-intensive portion of the algorithm to the FPGA. BWA software will find all alignment locations for a given short read, assuming the short read differs from the reference by no more than d characters.

The standard BWA software first constructs an index for a given reference genome, as demonstrated in Section II-A. Recall that $C(a)$ is the total number of bases in the reference genome lexicographically smaller than a. Also recall that an occurrence array ($O(a,i)$) stores the number of occurrences of a in the BWT string, between base 0 and i. Short reads are then batch read into memory from a file. For each short read, the CPU first computes a lower limit for the number of differences between the short read and the target reference genome. This lower limit is computed for every substring in the short read containing the final base; this set of lower limits is placed into an array called the $D()$ array. For each short read, the CPU maintains a heap, which is implemented as a set of stacks to hold all the current queries, which are variations of the original short read. Stacks are sorted by score, where queries with equivalent scores are pushed onto the same stack, and the stack with the best score is at the top of the heap.

The CPU begins the alignment of a short read by pushing the original short read onto the heap. The software then iteratively pops a single query from the top of the heap. If

the current query already has d variations from the short read, then BWA attempts to exactly match the remaining bases of the short read (those not already searched in the current query) using a method named *bwt_match_exact_alt*. If the current query has less than d differences, the software prepends a single base from the short read to the current query and computes the new BWT index upper and lower bounds, as shown in Section II-A. If these newly computed bounds are legal, the software pushes this new query onto the heap. It also pushes all possible variations of the current query onto the heap (4 insertions, 1 deletion, 3 substitutions). The software continues processing queries until either the maximum number of alignments for a short read has been exceeded or the heap is empty. In this manner, the software finds all alignments for a short read having less than d differences from the reference genome.

B. FPGA Design

The logic within the M-505 FPGA is built upon Pico Computing's framework. Pico Computing provides a streaming communication system, which enables efficient PCIe communication between the CPU and the FPGA module. Pico also provides access to an 8 GB DDR3 SODIMM for storage of the BWT index.

On top of that framework, we have implemented a system to replace the *bwt_match_exact_alt* method from the BWA software, as described in Section III-A. The system accepts a query, the current query base index, and the current lower bound (k) and upper bound (l) of the BWT index on an input stream. The FPGA stores the query in on-chip memory while it is being processed for rapid base retrieval. The current base of the query is used to compute the next memory address to read from the DDR3 memory. The BWT index is stored in a compressed form (only 1 out of every 64 occurrence array counts is stored in the index), such that it fits in approximately 3 GB of DRAM. Two random access reads are required to compute the next BWT index bounds, one for the upper bound ($k = C(a) + O(a, k-1) + 1$) and one for the lower bound ($l = C(a) + O(a, l)$).

After k and l are computed, the FPGA verifies that the bounds are valid, i.e. $k < l$. If not, the query is not found within the reference, and the query is ejected from the on-chip query memory. If the bounds are valid and all bases within the query have been processed, the query, the current base index, k, and l are reported back to the host on an output stream. If all bases of the query have not been processed, then the current query base index is decremented, and the next base of the query is processed in a similar fashion.

The FPGA has plenty of on-chip memory to support many queries at a given time. Currently, 128 queries are maintained and operated upon at a time within the user logic. Also, read requests for consecutive queries are pipelined to the DDR3 memory to enable out-of-order execution and improve random access performance. A diagram of the logic required to implement this BWA system in the FPGA is shown in figure 4.

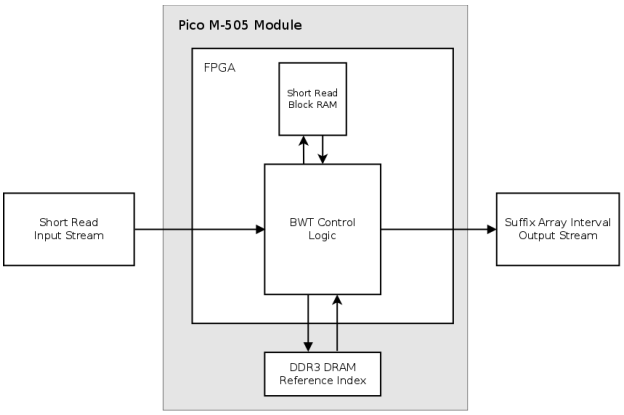


Fig. 4: The FPGA logic required to match queries to the reference. Note that the reference BWT index is stored in off-chip DRAM.

C. Software Modifications

To make the best use of the FPGA, instead of searching each short read one at a time, we allocate an array to keep track of multiple software heaps to search in parallel. We initialize the first heap in our array with a short read and begin our search in software as described in Section III-A. During our search, when no more differences are allowed for the current query, instead of calling bwt_match_exact_alt, we add the current query to a queue for the FPGA. Two threads are dedicated for streaming communication with the FPGA: 1 thread sends blocks of queries from the queue to the FPGA for processing and 1 thread receives blocks of results from the FPGA and places them in a queue.

The software continues filling the array of heaps with new short reads until all heaps are occupied. When the array is full, we dequeue a result from the FPGA, and restore the heap belonging to that short read. The search then continues in software with the new read until we have found an alignment, the heap is empty, or until we queue the read to the FPGA again. Figure 5 shows the interaction of the BWA software with the FPGAs.

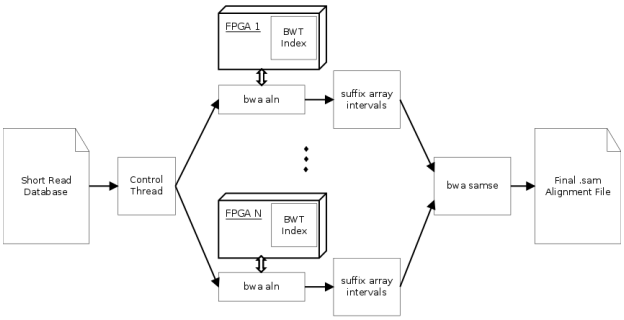


Fig. 5: Interaction of the BWA software with the FPGAs.

D. Results

The described system comprises a host chassis with 2 Intel Xeon CPUs and 12 M-505 FPGAs residing on 3 EX-

500 backplanes. This system is capable at aligning reads 48 times faster than the software version alone. This speedup enables researchers to receive their results much faster, thereby becoming much more efficient. Moreover, the FPGA design comes with tremendous power and area benefits. The FPGAs in the BWA system consume a mere 120 Watts total (10 Watts each) and the entire system fits inside a single 4U chassis. The power savings of an FPGA system drastically reduces energy consumption costs over the lifetime of a traditional CPU cluster.

About Pico Computing

Based in Seattle, Washington, Pico Computing specializes in highly integrated development and deployment platforms based on Field Programmable Gate Array (FPGA) technologies. Applications for Pico Computing technologies include cryptography, networking, signal processing, bioinformatics, and scientific computing. Pico Computing products are used in embedded systems as well as in military, national security, and high performance computing applications. For more information about Pico products and services, visit www.picocomputing.com.

References

[1] Hiseq systems comparison. [Online]. Available: http://www.illumina.com/systems/hiseq_comparison.ilmn, 2012.

[2] M-501. [Online]. Available: http://picocomputing.com/m-series/m-501/, 2012.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[4] Heng Li. Ngs mapper roc curves. [Online]. Available: http://lh3lh3.users.sourceforge.net/alnROC.shtml, November 2009.

[5] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrowswheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[6] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 2010.

[7] Julian Seward. bzip2 : Home. [Online]. Available: http://www.bzip.org/, 1996.