An FPGA-based BWT Accelerator for Bzip2 Data Compression

Weikang Qiao*, Zhenman Fang†, Mau-Chung Frank Chang*, Jason Cong*
* Center for Domain-Specific Computing, UCLA † Simon Fraser University
wkqiao2015@ucla.edu, zhenman_fang@sfu.ca, mfchang@ee.ucla.edu, cong@cs.ucla.edu

***Abstract*—The Burrows-Wheeler Transform (BWT) has played an important role in lossless data compression algorithms. To achieve a good compression ratio, the BWT block size needs to be several hundreds of kilobytes, which requires a large amount of on-chip memory resources and limits effective hardware implementations. In this paper, we analyze the bottleneck of the BWT acceleration and present a novel design to map the anti-sequential suffix sorting algorithm to FPGAs. Our design can perform BWT with a block size of up to 500KB (i.e., bzip2 level 5 compression) on the Xilinx Virtex UltraScale+ VCU1525 board, while the state-of-art FPGA implementation can only support 4KB block size. Experiments show our FPGA design can achieve ~2x speedup compared to the best CPU implementation using standard large Corpus benchmarks.**

I. Introduction

The Burrows-Wheeler Transform (BWT) [1], invented by Burrows and Wheeler in 1994, has played an important role in information technology. One of the key applications for BWT is data compression, as BWT tends to place the same characters together for easier compression. The widely used bzip2 [2] compression algorithm, which combines BWT with run-length encoding and huffman encoding [3], is reported to have much higher compression ratio than many other lossless compression standards such as ZLIB [4] and GZIP [5].

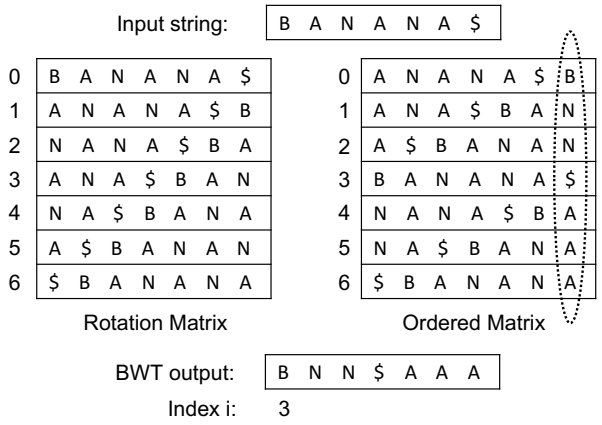

Fig. 1. Example of the BWT on a string "banana"

The basic idea of BWT is illustrated in Figure 1: given a string of length N, we first append it with a unique sentinel symbol $ and assume $ is larger than any input symbols. Then we generate all rotations of the string, sort the N+1 rotations in lexicographical order and store them into the ordered matrix. The last column of the ordered matrix will be the BWT output of the string; and an index i is also recorded, where i is the position of the original string in the ordered matrix.

Here the length N is usually referred to as the BWT *block size* and it determines the compression ratio of the bzip2 compression algorithm. The larger the BWT block size is, the better compression ratio will be achieved. On the other hand, the execution time of BWT also depends on the block size: a direct implementation of this rotation and sorting based algorithm takes $O(N^2 logN)$ time. So the execution of BWT is very time-consuming when N is a large number, i.e., when we try to achieve a high compression ratio. In fact, our profiling results show that the BWT execution takes up to 90% of the whole bzip2 software runtime and has an average compression throughput of 6.7 MB/s when the BWT block size is 300KB.

Implementing BWT with such a huge block size inherently poses great challenges to both software and hardware designs. For software implementations, several optimized sorting algorithms have been proposed [6], [7], [8]. Among them the fastest design is the ternary-split quicksort method [8], which works well for less repetitive text but has a very slow speed when the text has many repetitive symbols. For hardware implementations, the situation becomes even worse. First, FPGA-based sorting network cannot be extremely large, especially in the case where each key to be compared is long. Second, the algorithm requires a lot of on-chip memory resources: for example, the fastest software implementation of BWT needs at least $9 * blocksize$ memory, which is well beyond the limit of current FPGA on-chip memory. In fact, state-of-the-art FPGA implementations can only perform the BWT sorting part on a block size of 4KB and no complete implementation is reported against the fastest CPU solution.

In this paper, we present the first FPGA-based hardware design of the entire BWT algorithm that can perform BWT with a block size of hundreds of kilobytes. The implementation is based on the antisequential suffix sorting algorithm [9], which is originally proposed to reduce the worst-case complexity issue (highly repetitive symbols) of the ternary-split quicksort method. In most cases (less repetitive symbols), it is slower than the ternary-split quicksort method on CPUs due to the high cache miss rate. However, we find FPGAs are good candidates to implement such an algorithm since there are a large amount of distributed on-chip memories. We implement two acceleration architectures—direct suffix list design and two-level suffix list design—on a Xilinx Virtex UltraScale+ VCU1525 board. Our designs have a good scalability as long as there are enough on-chip memory resources: we support up to 500KB block size on this device. Using standard large Corpus benchmarks, we can achieve an average speedup of 2x over the fastest software implementation of BWT [2].

II. Background and Related Work

A. Prior FPGA-based Designs and Their Limitations

Current hardware implementations [10], [11], [12] usually choose the suffix sorting as their core algorithm and focus on building high-performance parallel sorting networks. The state-of-the-art work [12] builds a sorting network that can process up to 4KB block size in parallel in 4,049 cycles. However, those sorting networks can only do one iteration of the sorting. To achieve a complete BWT implementation, the string needs to be stored and fed into the sorting networks multiple times until no equal consecutive characters appear any more, making the total execution still time-consuming. Besides, the block size that the sorting networks can process is very small. In fact, it takes $O(N)$ stages for such a sorting network to process N characters, making the total number of the compare and swap elements $O(N^2)$. Considering the bzip2 standard requires performing BWT on a block size of 100s of KB to achieve good compression ratio, it is impractical to implement BWT with such sorter-based approach for the bzip2 standards.

B. Antisequential Suffix Sorting (ASS) Algorithm

1) Algorithm Overview: An antisequential suffix sorting algorithm is proposed in [9], aiming to reduce the worst case complexity of the ternary-split quicksort approach. It works as below: each rotation can be denoted using a $suffix$, which is the position of its first byte in the original string, and then the corresponding last byte for that rotation is at position $suffix - 1$. We also denote the sequence of $x_i x_{i+1} ... x_j$ as x_i^j. Given a string $x_1 x_2 .. x_N$ and suffix i, we define the suffix string of suffix i as $s_i = x_i^N$ for $1 \leq i \leq N$. Since the last byte of each suffix string is x_N, which is the sentinel symbol \$ and is larger than any input symbol, each suffix string is unique. In other words, sorting each rotation is equivalent to sorting its suffix string.

Now assume input symbols come in antisequentially from x_N to x_1, we can form a sorted list of previously processed suffix strings. When a new symbol x_i comes in, we only need to insert its suffix i into the right position of previously sorted suffix list, based on the relative order of its suffix string $s_i = x_i s_{i+1}$. Consider s_j such that $j > i$, suffix j is already in the sort list. We define a bucket S_i:

$$S_i = \{s_j : j > i, x_j = x_i, s_{j+1} > s_{i+1}\} \quad (1)$$

If S_i is not empty, then its smallest element is the right suffix j in front of whom the input suffix i should be inserted. It can be easily found since the position of suffix $i+1$ in the list is already known and we can scan up the list from suffix $i+1$ to check if such suffix j whose corresponding $x_{j-1} = x_i$ exists. If it exists, we insert the suffix i before $j - 1$. An example illustrating such process is shown in Figure 2.

If no such match found, that means either the input symbol x_i never appeared before or its suffix string s_i is the largest suffix string that starts with symbol x_i. To deal with the two special cases, a dictionary table is established. The table records if each type of symbol appeared before and if appeared, the largest suffix whose suffix string starts with that symbol. In the first case, we find the largest symbol that

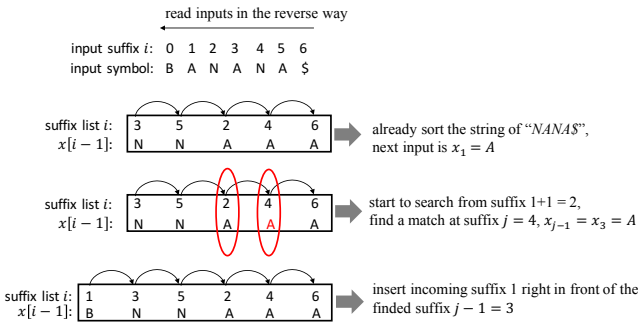

Fig. 2. Example of the BWT by antisequential suffix sorting

is lexicographically smaller than x_i and insert suffix i right after its largest suffix. In the second case, since x_i already appeared, we insert suffix i right after the previously largest suffix starting with x_i. In both cases we also need to update the dictionary table with the latest suffix i for symbol x_i.

Since the ASS algorithm greatly reduces the required on-chip memory amount, it makes the acceleration of BWT with large block sizes (i.e., 100s KB) possible on an FPGA.

2) Bottleneck of CPU Implementation: In [9], the authors did some software implementations to compare the performance of ASS algorithm with the ternary-split quicksort based BWT algorithm [8]. Their experiments show the ASS is on average 4x slower. Our profiling on Intel Xeon CPU E5-2680 shows it is mainly because of the high L2 cache miss rate, as the memory access pattern for accessing the suffix list is quite irregular and the 256KB L2 cache cannot hold all the suffix list. On the other hand, FPGAs feature a large amount of distributed on-chip memory resources, which can be customized to ensure the fast on-chip data accesses.

III. Antisequential Suffix Sorting in Hardware

In this section, we first present a straightforward design of ASS and discuss its design trade-offs. Then we introduce a two-level suffix list approach to accelerate ASS by reducing suffix list search time.

A. Direct Suffix List Design

Figure 3 shows the architecture of the direct suffix list design. At each cycle one symbol will be fed into the design. The input symbol will be first compared with the entries in the dictionary table to find if the same symbol appeared before, and the finding results will be sent to the controller. If no such symbol appeared before, the controller will insert the input suffix after the previously largest suffix and update the dictionary table. If the input symbol has appeared, the controller will start searching the suffix list until it finds a match or reaches the tail of the list. After that the input suffix will be inserted into the right position and the dictionary table will get updated correspondingly.

Although the overall procedure is quite sequential, we can exploit some inherent parallelism and design optimizations to increase the performance of the design as below.

1) Dictionary Table Design: The dictionary table can be implemented as a simple hash ram of 128 entries (since regular text files have 128 ascii codes) and be accessed through the

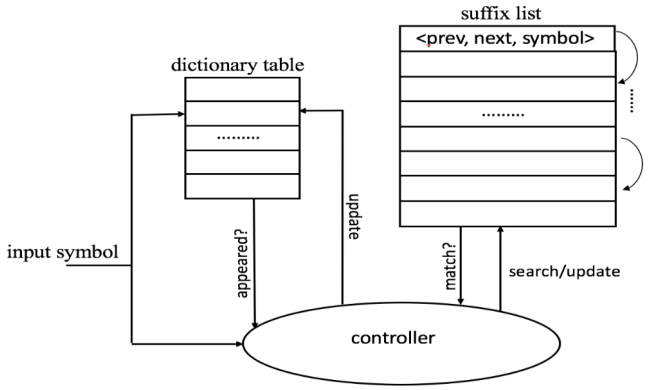

Fig. 3. Block diagram for direct suffix list design

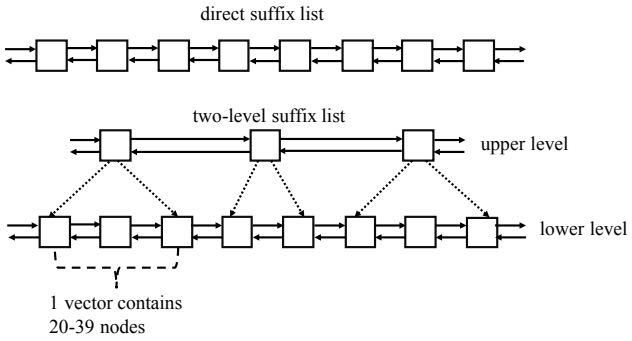

Fig. 4. Two-level suffix list

8-bit input symbol value. This configuration works well when the input symbol already exists in the table. However, when the input symbol never appeared, we need to search through the whole table to find the largest symbol. For example, consider the case where the first two symbols coming in are "A" and "z", for input "z", it takes 57 cycles to find the largest existing symbol (57 is the difference of the two symbols' ascii value). To speedup this process, the dictionary table is optimized as a ram of 32 entries and each entry contains 4 consecutive symbols' index information. Reading 1 entry can get the largest appeared symbol index in the 4-symbol entry in 1 cycle through encoding and a priority multiplexer. Thus, in the previous example the total cycle to find the largest existing symbol is reduced to 15.

2) Suffix List Design: The suffix list is also implemented as a ram whose depth is rounded to the BWT block size. The content of the suffix list entry is $< prev, next, symbol >$. For a block size of 100kB design, each entry takes 42 bits, since the $prev$ and $next$ pointers take 17 bits each and another 8 bits are used to store the corresponding symbol x_{i-1}. The BRAM is configured as a dual-port ram and each cycle one read and one write operations are allowed to perform. There is no read output register so read latency is only 1 cycle. To access the next element of current node, we only need to read the content of current node and use its 17-bit $next$ part as the next read address. In general, it takes 1 cycle to search the next element and 2-3 cycles to insert a new element depending on if the node to be inserted is a head or not.

3) Tasks Overlap: Since it takes only 1 cycle to update the dictionary table and 3 cycles to insert a new entry into the suffix list, these two parts can be overlapped after searching the list. We also overlap the dictionary table references between even and odd input symbols. The only sequential part left is the suffix list searching.

B. Two-Level Suffix List Design

In the direct suffix list design, the performance is determined by the total search depth of the suffix list, which takes up to 90% of the total execution time on an FPGA. To reduce this searching procedure, a two-level suffix list is suggested in [9] where the lower level is the original suffix list and the upper level is a sparse version of the lower list. Each upper-level node collects the information of what symbols are in its K lower-level nodes; here we call these K nodes as a *vector*. This approach again does not show good performance in the software implementation, due to the high cache miss rate. However, the two-level suffix list design can be very effective for FPGA acceleration due to its rich amount of distributed BRAMs. We choose $K = 20$ as suggested by the theoretical analysis in [9].

1) Lower-Level Suffix List: The content of the lower-level suffix list entry now becomes $< prev, next, upper, bound >$. The $prev$ and $next$ pointers are the same as those in the direct suffix list design; the up pointer is the address of its corresponding node in the upper-level list; the 1-bit $bound$ indicates if the entry is at the boundary of its *vector*. When insert a new node after the boundary node, the $bound$ bit of the new node is copied from the original boundary node and the $bound$ bit of the old boundary node is cleared.

2) Upper-Level List: The organization of each upper-level node is $< prev, next, lower, flag, count, last >$. The $prev$ and $next$ pointers records the positions of its previous and next upper-level nodes; the $lower$ pointer indicates the smallest suffix of its lower-level nodes; the $flag$ is 128 bits and each bit indicates if its corresponding symbol exists in the lower-level vector; the 6-bit $count$ records the size of the vector and the 1-bit $last$ signal indicates if it is the last node in the upper-level. For a block size of 100kB design, the number of upper-level list entries is $100k/20 = 5k$.

3) Maintaining the Upper-Level List: In general, to insert a new entry into the lower-level list, we only need to update the $flag$ bits and add $count$ by 1 in its corresponding upper-level node. If the new entry is inserted before the original smallest suffix, we also update the $lower$ pointer with the new suffix. To ensure the updating flags part not be the critical path, we allocate 1 extra clock cycle for it.

A special case is when the size of *vector* becomes larger. We maintain the size of the vector between 20 and 39. If a new entry is inserted and the size of the vector is about to become 40, we will split the vector into two size-20 vectors. Since the $flag$ information needs to be recorded precisely for each vector, we need to refer to all the lower-level nodes in the original vector to get the updated $flag$ information.

IV. Evaluation

A. Experimental Setup

To better understand the performance difference, we perform experiments using both software and hardware. On the

software side, we implement the basic antisequential suffix sorting algorithm and the fastest BWT algorithm extracted from the bzip2 software on a Intel Xeon E5-2689 CPU with 256KB L2 cache and 35MB L3 cache. On the hardware side, we implement both the direct suffix sorting and the two-level suffix sorting designs and measure the kernel execution time on a Xilinx Virtex UltraScale+ VCU1525 board. The benchmarks we use are selected from the standard Calgory Corpus and Large Corpus benchmarks whose file size is more than 500kB.

B. Resource Utilization

Our hardware implementation uses less than 1% of LUTs slices and Flip-Flops, 0 DSPs, as the ASS algorithm is not computation-intensive. The BRAM usage of both the direct suffix list design and the two-level suffix list design is reported in Table I: it increases with the BWT block size. The two-level suffix list design for 500KB block size only takes 35.5% BRAM resources, indicating this FPGA is even capable of perform BWT with a block size of 900KB to satisfy the highest compression ratio in the bzip2 compression standard. The BRAM usage for the direct suffix list design is even less.

We did not include the data for a block size more than 500KB, as the clock frequency that the designs can run drops as the block size increases, and the expected hardware speedup goes below 1.5x. This is because the critical path sits between the operation of reading the suffix list and send its result to the controller to decide which element to read for the next time.

TABLE I
RESOURCES UTILIZATION

Block Size	BRAM Utilization	Frequency (MHz)
two-level(100KB)	141 (6.53%)	222
two-level(300KB)	462 (21.4%)	187
two-level(500KB)	768 (35.5%)	156
direct (100KB)	120.5 (5.58%)	231
direct (300KB)	385 (17.82%)	192
direct (500KB)	642 (29.7%)	157

C. Performance Evaluation

A performance comparison of various designs with a block size of 300KB is shown in Figure 5. Although the software implementation of the antisequential suffix sorting algorithm is the slowest in the four cases, our FPGA implementations of the two-level suffix list achieves an average speedup of 2x over the fastest bzip2 BWT kernel on the CPU. The average speedups for 100KB and 500KB block sizes are 2.3x and 1.6x, and detailed breakdown is omitted due to space constraints.

As shown in Figure 5, in general, the direct suffix list design is slower than the CPU bzip2 BWT kernel. However, an interesting observation is for benchmark E.coli, the direct suffix list method works the best. This is because E.coli is a genomic sequence file whose input symbols have only 4 types and in most cases only a few searches are needed to get a match. For the rest four benchmarks whose input symbols are more diverse, the two-level suffix list design is the best and outperforms the direct suffix list design by around 10x. This gives us the options to select between the two designs depending on the input file characteristics

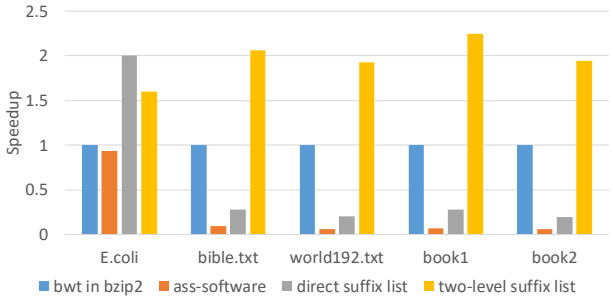

Fig. 5. Performance comparison when the BWT block size = 300kB

V. CONCLUSION

In this paper, we designed and implemented the first complete BWT solution on FPGAs with a block size up to 500KB, which supports high quality compression on FPGAs. Although the original antisequential suffix sorting algorithm slows down in CPUs due to its high cache miss rate, we find it is a good fit for FPGA acceleration since FPGAs feature a rich set of distributed on-chip BRAMs. Experiments show that our FPGA implementations achieve around 2x speedup over the fastest CPU solution. In future work, we will optimize the clock frequency of our BTW accelerator and accelerate the entire bzip2 compression standard on FPGAs.

VI. ACKNOWLEDGEMENT

This work is partially supported by Mentor Graphics under the Center for Domain-Specific Computing Industrial Partnership Program. We would like to thank Fedor Pikus for his valuable support.

REFERENCES

[1] M. Burrows *et al.*, "A block-sorting lossless data compression algorithm," *SRC Research Report*, 1994.
[2] "Bzip2 Compression Library," http://www.bzip.org, [Online; accessed 13-Jan-2019].
[3] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sept 1952.
[4] "Zlib Compression Library," http://www.zlib.net/, [Online; accessed 13-Jan-2019].
[5] "Gzip file format specification version 4.3," https://tools.ietf.org/html/rfc1952.
[6] J. L. Bentley *et al.*, "Fast Algorithms for Sorting and Searching Strings," in *Proceeding of Data Compression Conference*. ACM, 1997, pp. 360–369.
[7] N. J. Larsson *et al.*, "Fast Suffix Sorting," *technical report LU-CS-TR*, 1999.
[8] J. Seward, "On the Performance of BWT Sorting Algorithms," in *Proceeding of Data Compression Conference*, ser. DCC '00. IEEE, 2000, pp. 173–182.
[9] D. Baron *et al.*, "Antisequential Suffix Sorting For BWT-Based Data Compression," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 385–397, April 2005.
[10] U. Cheema *et al.*, "A High Performance Architecture for Computing Burrows-Wheeler Transform on FPGAs," in *International Conference on Reconfigurable Computing and FPGAs*, ser. ReConFig '13. IEEE, 2013.
[11] J. A. Perez-Celis *et al.*, "An FPGA Architecture to Accelerate the Burrows Wheeler Transform by Using a Linear Sorter," in *Parallel and Distributed Processing Symposium Workshops*, ser. IPDPS '16. IEEE, 2016, pp. 156–161.
[12] B. Zhao *et al.*, "Streaming Sorting Network Based BWT Acceleration on FPGA for Lossless Compression," in *Proceedings of the 2017 International Conference on Field-Programmable technology*, ser. FPT '17, 2017, pp. 247–250.