Hindawi
Journal of Sensors
Volume 2020, Article ID 5726527, 22 pages
https://doi.org/10.1155/2020/5726527

*Research Article*

# Block-Split Array Coding Algorithm for Long-Stream Data Compression

**Qin Jiancheng**[1], **Lu Yiqin**[1] and **Zhong Yu**[2,3]

[1]*School of Electronic and Information Engineering, South China University of Technology, Guangdong, China*
[2]*Zhaoqing Branch, China Telecom Co., Ltd., Guangdong, China*
[3]*School of Software, South China University of Technology, Guangdong, China*

Correspondence should be addressed to Lu Yiqin; eeyqlu@scut.edu.cn

Received 21 April 2019; Accepted 28 January 2020; Published 18 February 2020

Academic Editor: Harith Ahmad



With the advent of IR (Industrial Revolution) 4.0, the spread of sensors in IoT (Internet of Things) may generate massive data, which will challenge the limited sensor storage and network bandwidth. Hence, the study of big data compression is valuable in the field of sensors. A problem is how to compress the long-stream data efficiently with the finite memory of a sensor. To maintain the performance, traditional techniques of compression have to treat the data streams on a small and incompetent scale, which will reduce the compression ratio. To solve this problem, this paper proposes a block-split coding algorithm named "CZ-Array algorithm," and implements it in the shareware named "ComZip." CZ-Array can use a relatively small data window to cover a configurable large scale, which benefits the compression ratio. It is fast with the time complexity O($N$) and fits the big data compression. The experiment results indicate that ComZip with CZ-Array can obtain a better compression ratio than gzip, lz4, bzip2, and p7zip in the multiple stream data compression, and it also has a competent speed among these general data compression software. Besides, CZ-Array is concise and fits the hardware parallel implementation of sensors.

## 1. Introduction

With the advent of IR (Industrial Revolution) 4.0 and the following rapid expanding of IoT (Internet of Things), lots of sensors are available in various fields, which will generate massive data. The soul of IR 4.0 and IoT with intelligent decision and control relies on these valuable data. But the spreading sensors' data also bring problems to the smart systems, especially in the WSN (wireless sensor network) with precious bandwidth. Due to the limited storage capacity and network bandwidth, GBs or TBs of data in IoT make an enormous challenge to the sensors.

Data compression is a desirable way to reduce storage usage and speed up network transportation. In practice, stream data are widely used to support the large data volume which exceeds the maximum storage of a sensor. For example, a sensor with an HD (high-definition) camera can deal with its video data as a long stream, despite its small memory. And in most cases, a lot of sensors in the same zone may generate similar data, which can be gathered and compressed as a stream, and then transmitted to the back-end cloud platform.

This paper focuses on the sensors' compression which has strict demands about low computing consumption, fast encoding, and energy saving. These special demands exclude most of the unfit compression algorithms. And we pay attention to the lossless compression because it is general. Even a lossy compression system usually contains an entropy encoder as the terminal unit, which depends on the lossless compression. For example, in the image compression, DCT (discrete cosine transform) algorithm [1] needs a lossless compressor. Some lossy compression algorithms can avoid the entropy encoder, such as SVD (singular value decomposition) algorithm [2], but they often consume more computation resources and energy than a lossless compressor.

A problem is about the finite memory of each sensor under the long-stream data. The sensors have to compress GBs of data or more, while a sensor has only MBs of RAM (random access memory) or less. In most of the traditional compression algorithms, the compression ratio depends on

the size of the data window, which is limited by the capacity of the RAM. To maintain the performance, traditional techniques have to treat the data streams on a small and incompetent scale, which will reduce the compression ratio.

For example, the 2 MB data window cannot see the stream data out of 2 MB at a time; thus, the compression algorithm cannot merge the data inside and outside this window, even if they are similar and compressible. The window scale restricts the compression ratio. Unfortunately, due to the limited hardware performance and energy consumption of the sensors, it is difficult to enlarge the data window for the long-stream data compression.

Moreover, multiple data streams are common in IoT, such as the dual-camera video data, and these streams may have redundant data that ought to be compressed. But since the small data window can see only a little part of a stream, how can it see more streams so that they can be merged?

In our previous papers, we have designed and upgraded a compression format named "CZ format" which can support the data window up to 1 TB (or larger) [3, 4] and implemented it in our compression software named "ComZip." But the sensor's RAM still limits the data window size. And using flash memory to extend the data window is not good, because the compression speed will fall evidently.

To solve the problems of long-stream data compression with limited data window size, this paper proposes a block-split coding algorithm named "CZ-Array algorithm" and implements it in ComZip. CZ-Array algorithm has the following features:

(1) It splits the data stream into blocks and rebuilds them with the time complexity O($N$) so that the data window can cover a larger scale to fit the big data compression

(2) It builds a set of matching links to speed up the LZ77 compression algorithm [5], depressing the time complexity from O($N^2$) into O($N$)

(3) It is concise for the hardware design in the LZ77 encoder pipeline, so that the sensors may use parallel hardware to accelerate the LZ compression

We do some experiments on both platforms x86/64 and ARM (Advanced RISC Machines), to compare the efficiencies of data compression among ComZip with CZ-Array, gzip, lz4, bzip2, and p7zip. The experiment results indicate that ComZip with CZ-Array can obtain the best compression ratio among these general data compression software in the multiple stream data compression, and it also has competent speed. Besides, the algorithm analysis infers CZ-Array is concise and fits the hardware parallel implementation of sensors.

To make further experiments, we provide 2 versions of ComZip on the website: for Ubuntu Linux (x86/64 platform) and Raspbian (ARM platform). Other researchers may download them from http://www.28x28.com:81/doc/cz_array.html.

The remainder of this paper is structured as follows:

Section 2 expresses the problems of long-stream data compression for sensors. Section 3 introduces the algorithm of CZ-Array coding. Section 4 analyzes the complexities of the CZ-Array algorithm. The experiment results are given in Section 5. The conclusions are given in Section 6.

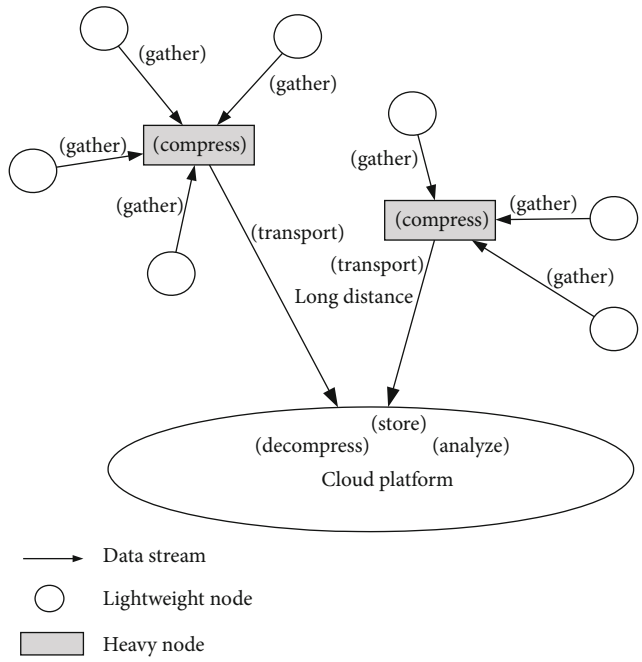



Figure 1: Network of lightweight and heavy sensors.

## 2. Problems of Long-Stream Data Compression for Sensors

Video/audio or other special sensors in IoT can generate long-stream data, but the bottlenecks of data transportation, storage, and computation in the networks of sensors need to be eliminated. Data compression meets this requirement. Figure 1 shows a typical scene in a network with both lightweight and heavy sensors, where long-stream data are generated.

This network has lots of lightweight nodes to sense the situation and generate long-stream data. Since they have limited energy, storing capacity, and computing resources, they can neither keep the data locally nor make strong compression. Meanwhile, a few heavy nodes in the network can gather and compress the data to reduce the long-distance bandwidth, and then transport them to the back-end cloud platform. The cloud platform has plenty of resources to store, decompress, and analyze the data.

In our previous papers, we have discussed the big data compression and encryption in the heavy nodes among a network of sensors [4, 6], but if the heavy nodes compress long-stream data, we still have problems with the finite memory, energy consumption, speed, and compression ratio. This paper focuses on the following problems:

(1) Limited by the data window size; how to cover a larger scale to see more data streams for a better compression ratio?

(2) Keeping the compression ratio roughly unchanged; how to improve the speed of the compression algorithm?

In our previous [3] and [4], we have shown that a larger data window can gain a better compression ratio. In this paper, we still use the same definition of the compression ratio as follows:

$$R = 1 - D_{\text{zip}}/D \quad (1)$$

$D_{\text{zip}}$ and $D$ are the volumes of the compressed and original data, respectively. If the original data are not compressed, $R = 0$. If the compressed data are larger than the original data, $R < 0$. Always $R < 1$.

The compression algorithms can merge similar data fragments within a data window. Thus, a larger data window can see more data and have more merging opportunities. But the data window size is limited by the capacity of RAM. Although in Figure 1 the cloud platforms have plenty of RAM for large data windows, a typical heavy-node sensor has only MBs of RAM at present. Using the VM (virtual memory) to enlarge the data window is not good enough, because the flash memory is much slower than RAM.

Moreover, a heavy node may gather data streams from different lightweight nodes, and the streams may have similar data fragments. But how can a data window see more than one stream to get more merging opportunities?

For the second problem, the compression speed is important for the sensors, especially the heavy nodes. GBs of stream data have to be compressed in time, while the sensors' computing resources are finite. Cutting down the data window size to accelerate the computations is not a good way, because the compression ratio will sink evidently, which runs into the first problem again.

To solve the problems, we need to review the main related works around sensors and stream data compression.

In [3] and [4], we have discussed current mathematic models and methods of lossless compression can be divided into 3 classes:

(1) The compression based on probabilities and statistics: typical algorithms in this class are Huffman and arithmetic coding [7]. The data window size can hardly influence the speed of such a class of compression

To maintain the statistic data for compression, the time complexity of Huffman coding is O(lb$M$) and that of traditional arithmetic coding is O($M$). $M$ is the amount of coding symbols, such as 256 characters and the index code-words in some LZ algorithms. O($M$) is not fast enough, but current arithmetic coding algorithms have been optimized and reached O(lb$M$), including that in ComZip [3].

(2) The compression based on the dictionary indexes: typical algorithms in this class are the LZ series [5], such as LZ77/LZ78/LZSS

To achieve the string matching, the time complexity of traditional LZ77, LZ78, and LZSS coding are O($N^2$), O($N$lb$N$), and O($N$lb$N$). $N$ is the data window size. While in ComZip, we optimize LZ77 and reach O($N$) by the CZ-Array algorithm. This paper focuses on it.

(3) The compression based on the order and repeat of the symbols: typical algorithms in this class are BWT (Burrows-Wheeler transform) [8], MTF (move-to-front) [9], and RLE (run-length encoding).

The time complexity of traditional BWT coding is O($N^2$lb$N$), which is too slow. But current BWT algorithms have been optimized and reached O($N$), including the CZ-BWT algorithm in ComZip [6].

Moreover, we use a new method to split the data stream into blocks and rebuild them with the time complexity O($N$). This is neither BWT nor MTF/RLE, but it is another way to improve the compression ratio by accommodating the data window scale. This paper focuses on it as a part of the CZ-Array algorithm.

MTF and RLE are succinct and easy to be implemented, but their compression ratios are uncompetitive in the current compression field. To achieve better compression ratio and performance, current popular compression software commonly combine different compression models and methods. Table 1 lists the features of popular compression software and ComZip.

To focus on the balance of compression ratio and speed, this paper ignores some existing methods, such as MTF, RLE, PPMd [10], and PAQ, which have too low compression ratio or speed.

In this paper, the term "data window" refers to different data structures in each compression algorithm. If software combines multiple data windows, we take its bottleneck window as the focus. Table 2 shows the data windows in typical compression algorithms.

To enlarge the data window and improve performance, researchers try various ways. Each compression class has a different algorithm optimization. As mentioned above, ComZip also keeps up with the optimal algorithms of the 3 classes, but each time we cannot focus on too many points. In [6], we have compared the time complexities of traditional BWT, BWT with SA-IS [11], BWT with GSACA [12], and CZ-BWT, which are current BWT studies.

The study of compression with AI (artificial intelligence) is a hopeful direction. Current researches mostly surround some special fields, such as image compression by neural networks [13–15]. Yet, it is still a problem for AI to achieve the general-field lossless compression efficiently, especially in the sensors with limited computing resources. This paper discusses the general compression, and some algorithms are more practical than AI. We may continue the study of AI algorithms as our future work.

The performance of compression is important for sensors, so the studies of parallel computing and hardware acceleration such as ASIC (application-specific integrated circuit) and FPGA (field-programmable gate array) [16–18] are valuable. A hotspot of researches is the GPU (graphics processing unit) acceleration [19–21]. But as we mentioned in [4], the problem is the parallel threads split the data window into

Table 1: Features of compression software.

| Software | Format | Basic algorithms | Maximum data window Support | Maximum data window Current | Shortages |
|---|---|---|---|---|---|
| WinZip | Deflate | LZSS & Huffman | 512 KB (LZSS) | 512 KB | Small data window; low compression ratio; weak big data support |
| WinRAR | RAR | LZSS & Huffman | 4 MB (LZSS) | 4 MB | Small data window; low compression ratio; weak big data support |
| | PPMd | PPM | — | — | Good compression ratio for text data only; weak big data support |
| gzip | Deflate | LZ77 & Huffman | 32 KB (LZ77) | 32 KB | Small data window; low compression ratio; weak big data support |
| lz4 | lz4 | LZ77 | 64 KB (LZ77) | 64 KB | Small data window; low compression ratio; limited big data support |
| bzip2 | bz2 | BWT & Huffman | 900 KB (BWT) | 900 KB | Small BWT block; low compression ratio; weak big data support |
| 7-zip/p7zip | LZMA | LZSS & arithmetic | 4 GB (LZSS) | 1.5 GB | Separated data windows for multithreads; limited big data support |
| ComZip | cz | BWT & LZ77 & arithmetic | 1 TB or more (LZ77) | 512 GB | Need larger data window for higher compression ratio |

Table 2: Data windows in different algorithms.

| Algorithm | Data window | Example of window size |
|---|---|---|
| LZ77 | Sliding window | 32 KB (gzip) |
| LZSS | Sliding window binary tree | 1GB (p7zip) |
| LZ78/LZW | Dictionary trie | 4 KB (GIF) |
| BWT | BWT block | 900 KB (bzip2) |
| Huffman<br>Arithmetic<br>PPMd | Statistic table | 64 KB (order-1 context)<br>16 MB (order-2 context) |

smaller slices and then reduce the compression ratio. Exactly, this paper cares about the data window size and scale.

In [6], we have considered the concision of CZ-BWT for hardware design, and this paper also considered the hardware implementation of CZ-Array. To enlarge the data window scale, CZ-Array follows the proposition of RAID (redundant arrays of independent disks) [22], which was previously used for the storage capacity, the performance, and the fault tolerance. To improve the compression speed, CZ-Array draws the reference from lz4 [17], one of the current fastest LZ algorithms based on LZ77. Both RAID and lz4 are concise, but RAID was not used for the compression before, and lz4 has small data windows and a low compression ratio.

We are arranging our work in data coding research. Table 3 shows the relationship within our work. We use the same platform: the ComZip software system. ComZip is a complex platform with unrevealed parts for the research of various coding problems, and it is still developing. To make the paper description clear, each time we can only focus on a few points. So each paper shows different details, and we call them Focus A, B, C, and D.

Figure 2 shows these focuses within the platform ComZip. We can see that the figures in each paper have some different appearances of ComZip because different problems are considered. Typically, this paper focuses on array coding, so the green arrows in Figure 2 points to the ComZip details which are invisible in the figures of [3, 4, 6]. Besides, this paper compares the similar structure in Focus C and D: the Matching Link Builder (MLB). Despite these MLBs in CZ-BWT [6] and CZ-Array have different functions, we cover them with a similar structure, which can simplify the hardware design and save the cost.

## 3. CZ-Array Coding

*3.1. Concepts of CZ-Array.* The compression software ComZip uses the parallel pipeline named "CZ pipeline." We have introduced the framework of the CZ encoding pipeline in [4, 6], and the reverse framework is the CZ decoding pipeline. Figure 3 is the same encoding framework, and the difference is that CZ-Array units are embedded, including

(1) The BAB (block array builder), which can mix multiple data streams into a united array stream and enlarge the data window scale

(2) The MLB (matching link builder), which can improve the speed of LZ77 and BWT encoding

CZ-Array combines the following methods to cover a larger data window scale and speed up the compression:

(1) CZ-Array uses the BAB to mix multiple data streams into a united array stream so that the multiple streams can be seen in the same data window

As shown in Figure 4, no matter how long a data stream is, the compression unit can only see the part in the data window. A traditional compression algorithm in Figure 4(a) has to treat the streams serially, which means the streams in the queue are out of the window scale and invisible. A parallel compression algorithm in Figure 4(b) can treat multiple

Table 3: Relationship of data coding research on ComZip.

| Paper | Similar base | Different focus |
|---|---|---|
| [3] | ComZip platform (shown framework) | A. Compression format for multi-level coding (including its framework) |
| [4] | ComZip platform (shown parallel pipeline) | B. Combined compression & encryption coding (including its parallel pipeline) |
| [6] | ComZip platform (shown BWT filter) | C. Fast BWT coding (including its matching link builder); hardware design |
| This paper | ComZip platform (shown CZ-Array units) | D. Array coding (including block array builder & matching link builder); hardware design (comparing the similar structure with [6]) |

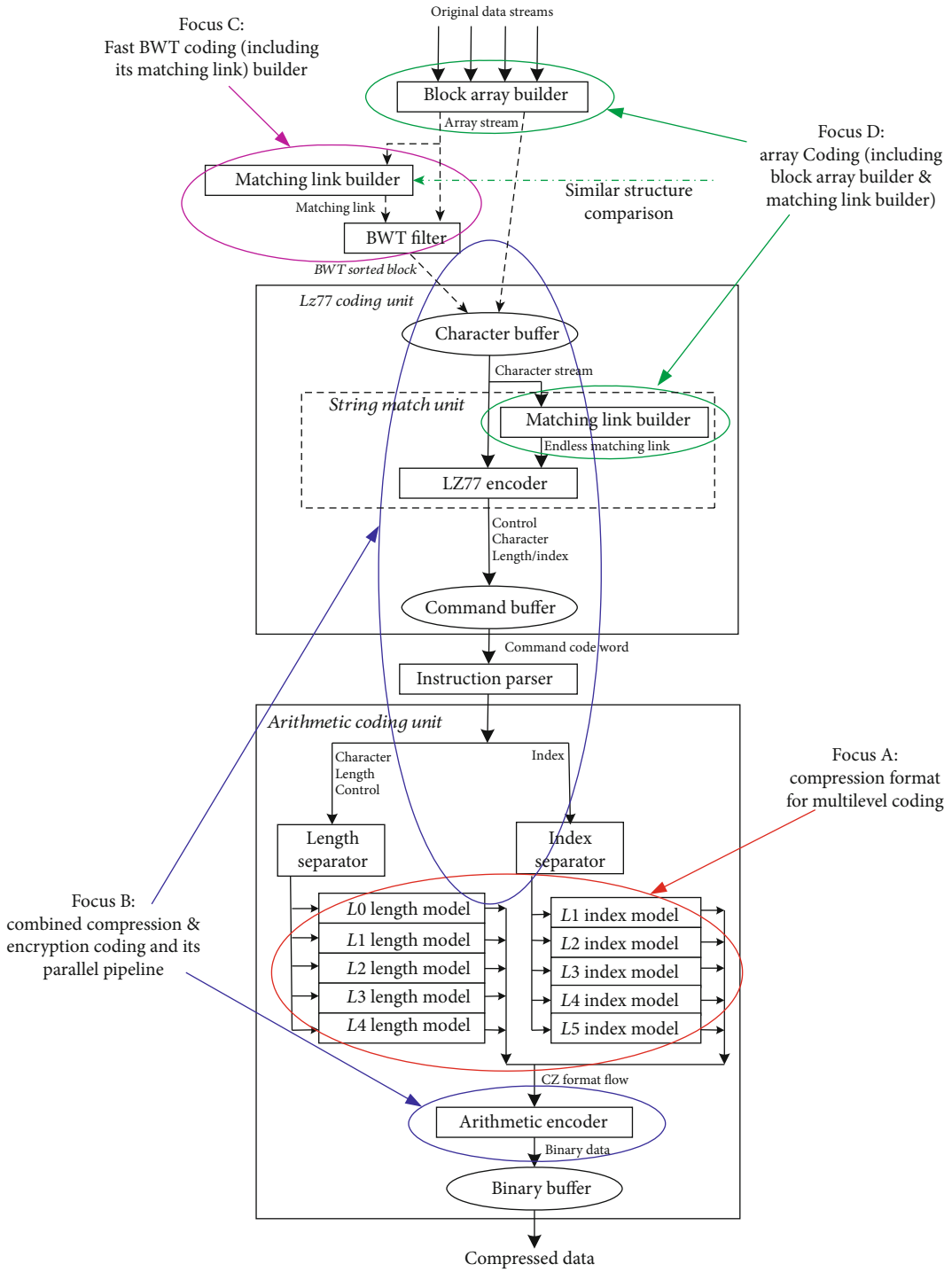



Figure 2: Focuses on research of ComZip.

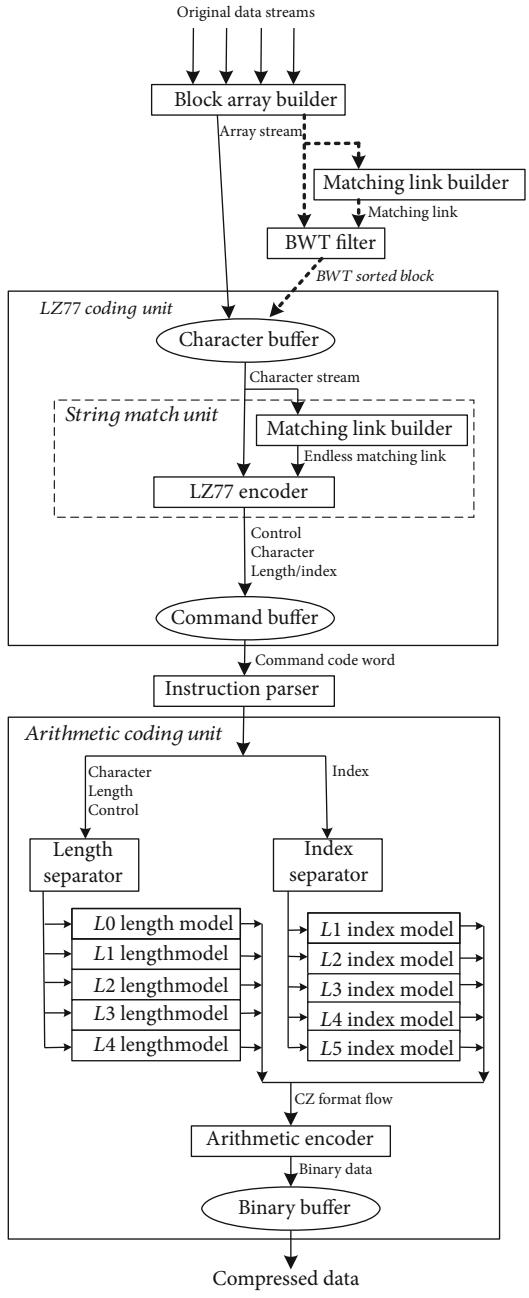



Figure 3: The framework of CZ encoding pipeline with CZ-Array.

streams to accelerate, but the window has to be divided into parts, and different parts cannot see each other; thus, the window scale is shrunk.

The window scale in Figure 4(c) is not shrunk while CZ-Array treats multiple streams in the same window. This case implies a better compression ratio because similar parts of different streams may be seen and matched by chance. Such a chance will not appear in Figure 4(b).

Mixing multiple streams into a united stream by BAB is fast. We got the hint from RAID [22] and implemented the BAB in the field of lossless compression. In BAB, streams are split into blocks, and the blocks are rearranged in a different sequence and then transmitted into the data window as a new stream.

(2) CZ-Array uses the MLB to make a BWT or LZ77 encoding pipeline so that the encoding speed can be optimized

The MLB can create matching links before BWT or LZ77 encoding. As shown in Figure 5, the matching links are helpful for the fast location without byte-searching in the data stream. Figure 5(a) shows the basic matching links, which are used in CZ-BWT. In [6], we call them "bucket sorting links." Figure 5(b) shows the endless matching links, which are used in LZ77 shown in Figure 5(c).

From Figure 5(c), we can see that the reversed string "ZYXCBA…" is the current target to be encoded. $k$ is the basic match length. The LZ77 encoder may follow the matching link to find the maximum match length and its location, which is much faster than simply searching the data window. In the rest of this section, we will introduce the algorithms of making and following the matching links.

(3) Both BAB and MLB are concise for the hardware design so that the sensors can gain better compression performance by the acceleration of FPGA/ASIC

CZ-Array algorithm is made up of several subalgorithms, such as the block array building algorithm and the matching link building algorithm. These 2 algorithms are concise and fit for the hardware design. We will provide their primary hardware design in Section 4.

*3.2. BAB Coding.* The BAB benefits from the hint of RAID, although they are different in detail. Figure 6 shows the simplest scenes of RAID-0 and BAB coding. In Figure 6(a), we suppose the serial file is written to the empty RAID-0. This file data stream is split into blocks and then written into different disks in the RAID. In Figure 6(b), the blocks are read from these disks and then re-arranged into a serial file.

In Figure 6(c), we assume all data streams are endless or have the same length. They are split into blocks, organized as a block array, and then arranged into a united array stream in the BAB. In Figure 6(d), we see the reversed process: a united array stream is restored into multiple data streams.

As RAID-5 can use $n+1$ blocks ($n$ data blocks and 1 parity-check block) to obtain redundant reliability, the block array can also use $m+1$ blocks ($m$ data blocks and 1 error-correction-code block) for the information security. $m$ and $n$ are different. For example, $m = 1000$ and $n = 4$. If the compressed data stream is changed, the error-correction-code block will be found unfit for the data blocks. But this paper just cares about the compression ratio and speed, so we focus on the simplest block array, like RAID-0.

In Figure 6, the RAID and BAB coding algorithms can easily determine the proper order of data blocks because the cases are simple and the block sizes are the same. But in practice, the data streams are more complex, such as the

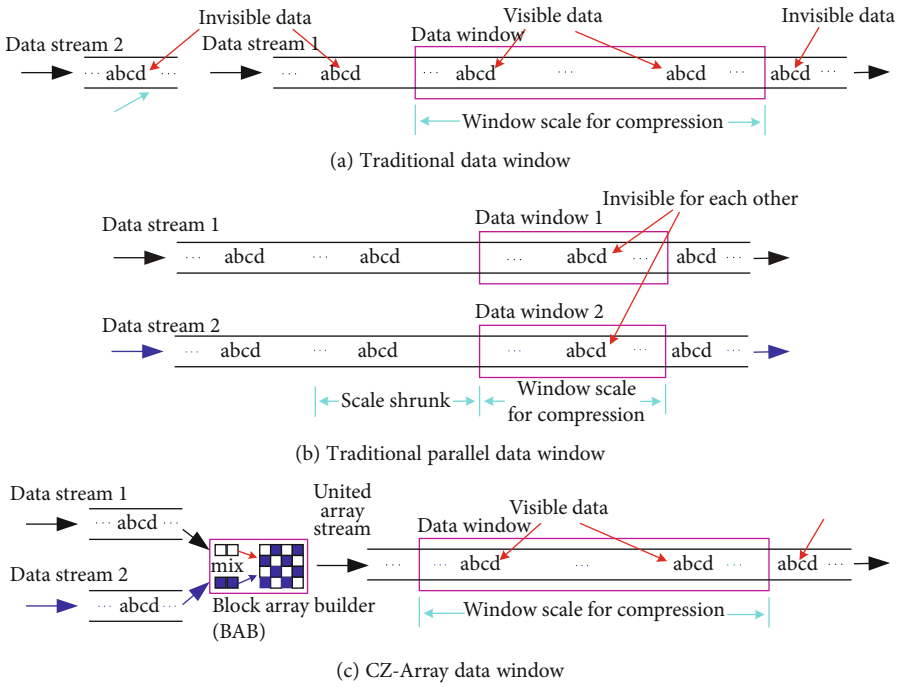



Figure 4: Data window scale for compression.

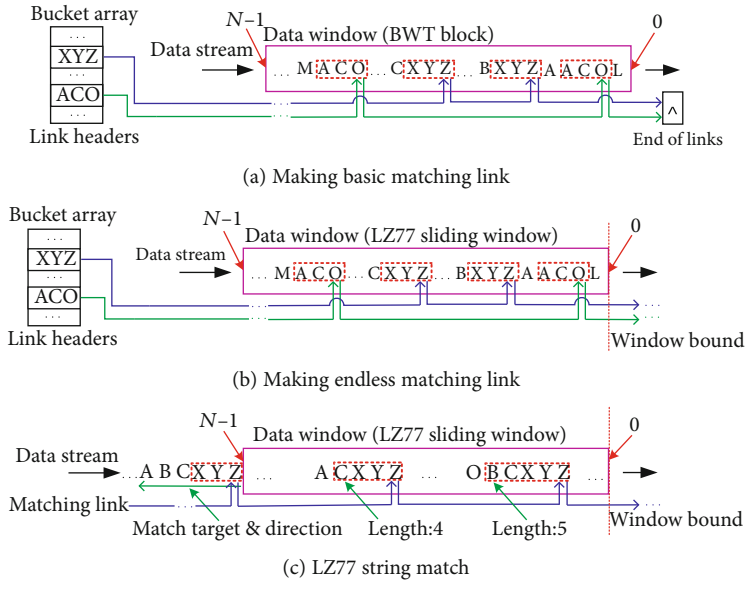



Figure 5: Matching link for compression ($k = 3$).

multiple streams with different lengths, the sudden ending stream, and the newly starting stream. Thus, the BAB coding algorithm has to treat these situations.

In the BAB encoding process, we define $n$ as the row amount of the data block array; $B$ as the size of a block; $A$ as the amount of original data streams (to be encoded); $Stream[i]$ as the data stream with ID $i$ ($i = 0, 1, \cdots, A-1$); $Stream[i].length$ is the length of $Stream[i]$ ($i = 0, 1, \cdots, A-1$).

Figure 6 shows the simplest situation $n = A$. But in practice, the data window size is limited, so this $n$ cannot be too large. Otherwise, the data in the window may be overdispersed and decrease the compression ratio. Hence, $n < A$ is usual, and each stream (to be encoded/decoded) has 2 status: active and inactive. A stream under the BAB coding (especially in the block array rows) is active, while a stream in the queue (outside of the rows) is inactive.

As shown in Figure 7, we divide the situations into 2 cases to simplify the algorithm design:

(1) Case 1: $Stream[i].length$ and $A$ are already known before encoding. For example, ComZip generates an XML (Extensible Markup Language) document at

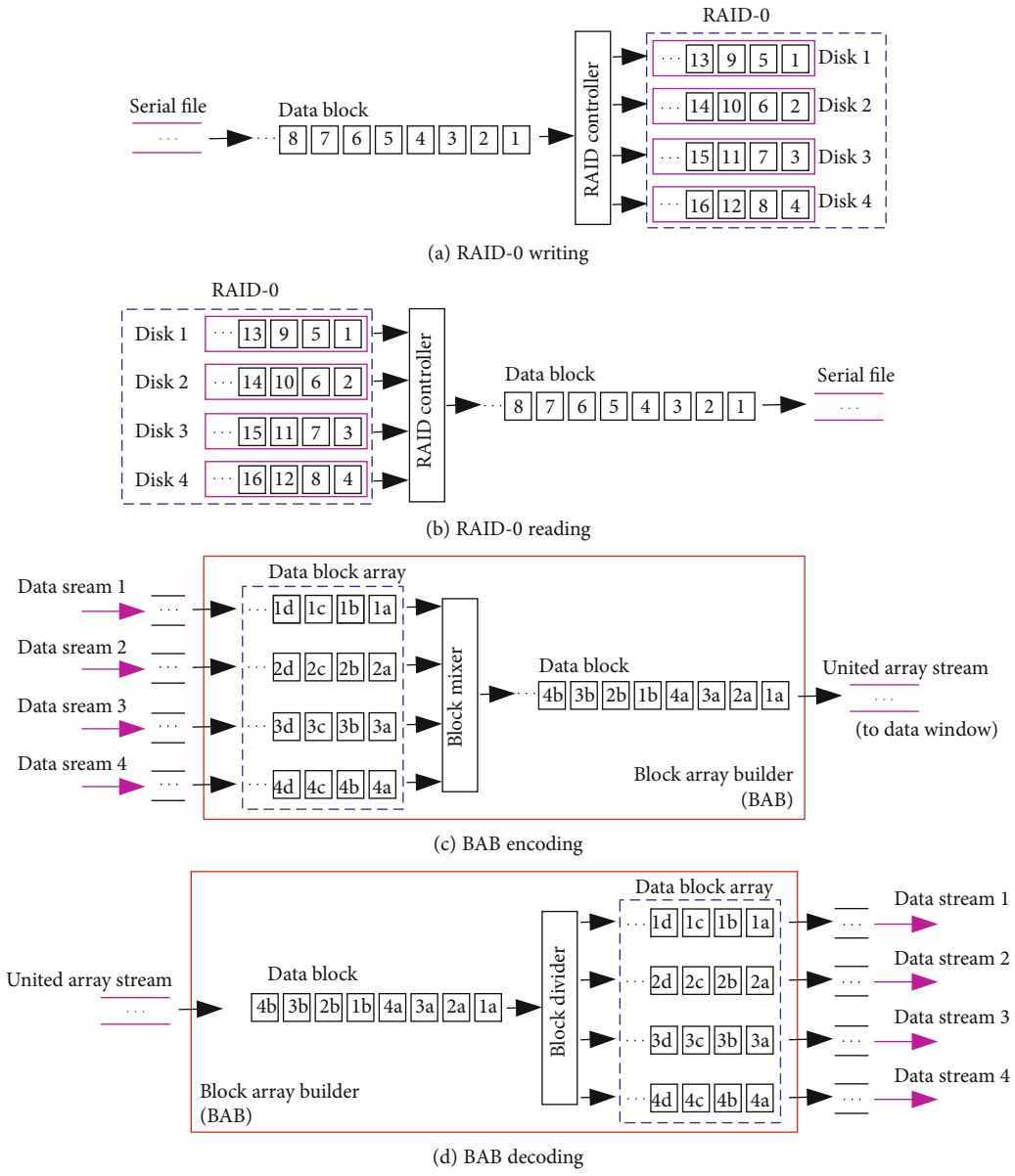



Figure 6: RAID-0 and BAB coding ($n = 4$).

the beginning of the compression. Each file is regarded as a data stream, and the XML document stores the files' catalog

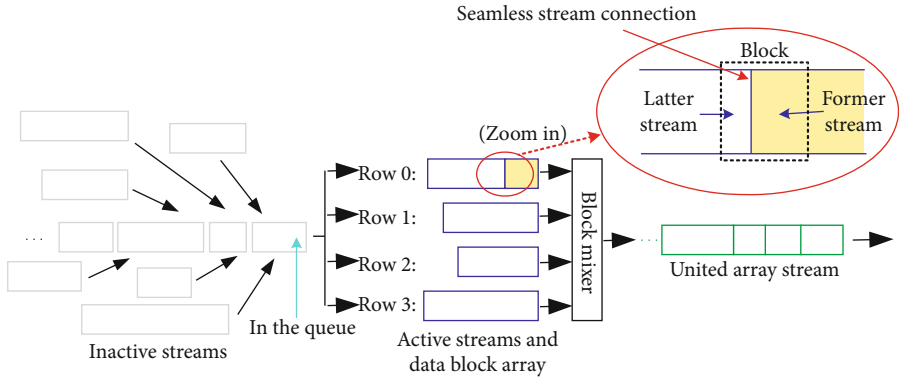
As shown in Figure 7(a), this case indicates all the streams are ready and none is absent during the encoding process. So the algorithm can use a fixed amount $n$ to encode the streams. When an active stream ends, a new inactive stream in the queue will connect this end. It is a seamless connection in the same block. And in the decompression process, streams can be separated by *Stream*[*i*].*length*.

(2) Case 2: *Stream*[*i*].*length* and $A$ are unknown before encoding. For example, the sensors send video streams occasionally. If a sensor stops the current stream and then continues, we regard the continued stream as a new one

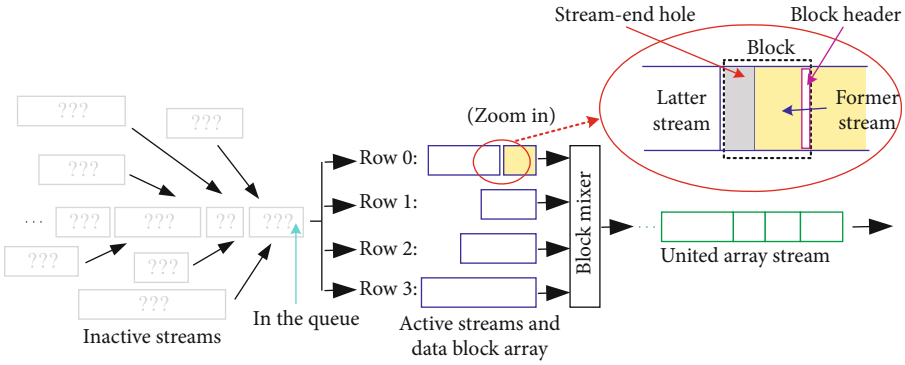
As shown in Figure 7(b), this case indicates the real-time stream amount $A$ is dynamic, and each *length* is unknown until *Stream*[*i*] ends. So the algorithm has a maximum amount $Nx$ and maintains the dynamic amount $n$. When an active stream ends, $n - 1$. When $n < Nx$, the algorithm will try to get a new inactive stream in the queue to add to the empty array row. If it succeeds, $n + 1$.

The different cases in Figure 7 lead to different block structures. A block in Figure 7(a) contains pure data, while a block in Figure 7(b) has the structure shown in Table 4, which has a block header before the payload data. We design this block header to provide information for the BAB decoding algorithm so that it can separate the streams properly.

To focus on the accurate problems around compression ratio and speed, this paper skips the implement for Case 2




(a) Case 1: known stream amount and lengths




(b) Case 2: unknown stream amount lengths

Figure 7: BAB encoding algorithm design ($n = 4$).

Table 4: Block structure for Case 2.

| Type | Content | Length |
|---|---|---|
| Block header | Structure version | 2 B |
| | Payload length ($B$) | 3 B |
| | Stream ending tag | 1 b |
| | Reserved tags | 7 b |
| Payload | Payload data | Fixed length (e.g., 1 MB) |

and discusses the algorithms for Case 1 only. Algorithms 1 and 2 show the BAB encoding and decoding for case 1.

To make the algorithm description clear, the implementing details are omitted. For example, since each stream length are known in Case 1, Algorithms 1 and 2 simply take the length control for granted: If the current stream ends, the input/output procedure will get the actual data length only, even if the fixed block length $B$ is wanted.

*3.3. MLB and LZ77 Encoding.* The MLBs can accelerate BWT and LZ77 encoding. We have introduced CZ-BWT encoding in [6], which actually includes the MLB algorithm in its phase 1. This paper just focuses on the other MLB algorithm and LZ77 encoding with matching links.

As shown in Figure 5(b), the MLB can use a bucket array to make endless matching links, such as the "ACO" link and "XYZ" link, and then provide them to the LZ77 encoder. Figure 5(c) shows the example of string matching in the LZ77 encoder. Following the "XYZ" link, we can see 2 matching points: 4 bytes matching and 5 bytes matching.

More details about these algorithms are shown in Figures 8(a) and 8(b). There are 2 phases in our LZ77 encoding: (Figure 8(a)) building the matching links in the MLB and (Figure 8(b)) each time following a matching link to find the maximum matching length in LZ77 encoder. So, the outputting length/index code-words come from these maximum matching lengths and the corresponding matching point positions. When a matching point cannot be found, a single character code-word is outputted.

In Figure 8(a), the data stream and its matching links can be endless. But in practice, the data window is limited by the RAM of the sensor. While the data stream passes through, the data window slides in the stream buffer, and the parts of data out of the buffer will lose information so that we cannot follow the matching links beyond the data window. Cutting the links is slow, but we may simply distinguish whether the link pointer has exceeded the window bound.

To locate the data window sliding on the stream buffer, we define the "straight" string $s$ in the window as follows:

$$s[i] = \text{buf}[i + \text{pos}] \ (i = 0, 1, \cdots, N - 1) \tag{2}$$

where buf[0...$M - 1$] is the stream buffer ($M = 2N$), pos is the current position of the data window on the stream buffer, $s[i...i + 2]$ is a 24b integer ($i = 0, 1, \cdots, N - 1$,), and buf $[i \cdots i + 2]$ is also a 24b integer ($i = 0, 1, \cdots, M - 3$).

```
Description of CZ-Array Algorithm
/* This encoding algorithm uses the following procedures:
      input_Encoder( stream, length ) : Data inputting from one of the active streams;
      output_Encoder( ) : Data outputting as a block into the united array stream.
*/
function encodeBAB( Stream[ 0..A-1 ] ) { /* Stream is the input stream queue (ready for coding) */
      row = array( 0..n-1 ); /* n is the row amount of the data block array (n<=A) */
      row[ 0..n-1 ] = Stream[ 0..n-1 ]; /* initialize the rows */
      i = 0; count = n;
      repeat do {
         while (buffer.length < B ) do { /* B is the size of a block */
            input_Encoder( row[ i ], B - buffer.length ) ➔ buffer;
            if ( row[ i ] is empty ) do { /* current stream ends */
               if ( count < A ) do { row[ i ] = Stream[ count ]; count = count +1; } /* switch to a new stream */
               else if ( n > 0 ) do { row[ i ] = row[ n-1 ]; n = n-1; } /* delete a row */
               else do { buffer ➔ output_Encoder( ); return; } /* this algorithm ends */
            }
         }
         buffer ➔ output_Encoder( ); /* split and output a block */
         i = (i+1) mod n; /* i = 0, 1, 2, … , n-1, 0, 1, 2, … , n-1, 0, 1, 2, … */
      }
}
```

Algorithm 1: BAB Encoding for Case 1.

In [6], we have built the matching links for CZ-BWT, which are also called "bucket sorting links." Now, we build the matching links for LZ77 on *buf*. Figure 8 shows the example of the "XYZ" link, and we see 3 matching points in it. The bucket array has $256^3$ link headers, and all links are endless. We define the structure of the links and their headers as follows:

$$\text{header}[m] = \begin{cases} \max(i)\,(\text{buf}[i \cdots i+k-1] = m), \\ \text{null}\,(\text{buf}[i \cdots i+k-1] \neq m \text{ for each } i), \\ \quad \left(i = 0, 1, \cdots M-k\,; m = 0, 1, \cdots 256^k - 1\right), \end{cases} \tag{3}$$

$$\text{link}[i] = \begin{cases} i - \max(j)\,(\text{buf}[i..i+k-1] = \text{buf}[j \cdots j+k-1]), \\ \text{expired value}\,(\text{buf}[i..i+k-1] \neq \text{buf}[j \cdots j+k-1] \text{ for each } j), \\ \quad (i = 0, 1, \cdots M-k\,; j = 0, 1, \cdots i-1), \end{cases} \tag{4}$$

$k$ is the fixed match length of each matching point in the links. In the example of Figure 5, $k = 3$ for "ACO" and "XYZ". $i$ is the plain position in Equations (3) and (4). We can see $link[i]$ stores the distance (i.e., relative position) of the next matching point. Be aware that if the next matching point is out of the window bound, $link[i]$ stores a useless value. The algorithm can distinguish this condition: Let the value null $= M$, if $i - link[i] < 0$, $link[i]$ is expired.

Figure 8(b) shows the fast string matching by following the matching links. We can see the reversed string "ZYXCBA…" starting in $s[N]$ gets 2 matching points: reversed "ZYXC" and "ZYXCB," with the matching lengths 4 and 5.

The data stream may be very long or endless, while the stream buffer is finite; thus, the data window sliding on this buffer will reach the edge buf$[M-1]$. We can move the data window back to buf[0], but it is slow. So, we assume this buffer is a "cycle" string buf$[0...M-1]$, which has the following feature:

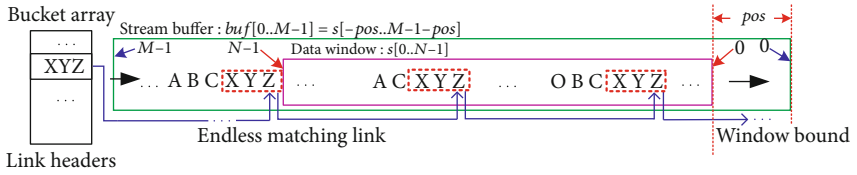
$$\text{buf}[i+M] = \text{buf}[i]\ (i = 0, 1, 2, \cdots) \tag{5}$$

Then, Equation (2) is changed and extended into a congruent modulus equation:

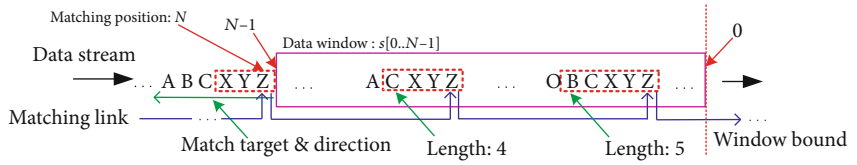
$$s[i] = \text{buf}[(i + \text{pos}) \mod M]\ (i = \cdots, -2, -1, 0, 1, 2, \cdots) \tag{6}$$

```
/* This decoding algorithm uses the following procedures:
      input_Decoder( length ) : Data inputting from the united array stream;
      output_Decoder( stream ) : Data outputting into one of the active streams.
*/
function decodeBAB( Stream[ 0..A-1 ] ) { /* Stream is the output stream queue (ready for coding) */
      row = array( 0..n-1 ); /* n is the row amount of the data block array (n<=A) */
      row[ 0..n-1 ] = Stream[ 0..n-1 ]; /* initialize the rows */
      i = 0; count = n;
      repeat do {
         input_Decoder( B ) ➔ buffer; /* B is the size of a block */
         if ( buffer.length =0 ) do { return; } /* input stream is empty, this algorithm ends */
         while ( buffer.length >0 ) do {
            if ( row[ i ] is full ) do { /* current stream ends */
               if ( count < A ) do { row[ i ] = Stream[ count ]; count = count +1; } /* switch to a new stream */
               else if ( n > 0 ) do { row[ i ] = row[ n-1 ]; n = n-1; } /* delete a row */
            }
            buffer ➔ output_Decoder( row[ i ] );
            i = (i+1) mod n; /* i = 0, 1, 2, … , n-1, 0, 1, 2, … , n-1, 0, 1, 2, … */
         }
      }
}
```

Algorithm 2: BAB Decoding for Case 1.




(a) Data window in stream buffer




(b) LZ77 string match

Figure 8: LZ77 string match with matching link ($k = 3$).

As $M = 2N$, the data window always occupies half of the stream buffer, and the fresh stream data fill the other half. To simplify the algorithm description, this paper hides the implementing detail of Equation (6) but takes for granted that Equation (5) is working.

Following the matching links is not fast enough and needs further acceleration. Seeking all the matching points along a link for the maximum matching length consumes too much time, yet most of the attempts fail to increase the matching length. So, we propose a configurable parameter *sight* to control the matching times and observe that LZ77 encoding speeds up evidently while the compression ratio decreased slightly.

Algorithm 3 shows the MLB coding, including the matching link building and LZ77 encoding. And we can use a common LZ77 decoding algorithm without matching links.

## 4. Analyses of CZ-Array Algorithm

The CZ-Array algorithm includes BAB encoding/decoding and MLB encoding. And MLB decoding is unnecessary because MLB encoding is just creating matching links to accelerate LZ77 encoding. We may refer to RAID-0 in the analyses of BAB coding, and we may compare LZ77 with MLB encoding and the other typical LZ algorithms in some popular compression software such as 7-zip/p7zip, gzip, and lz4.

### 4.1. Time Complexities

*4.1.1. BAB Coding.* In practice, RAID-0 is very fast, and its algorithm is simple. The key factors of its performance are $N$ and $n$. $N$ is the input/output data stream length, and $n$ is the disk amount of RAID-0. We regard $n$ as a constant.

```
/* This encoding algorithm uses the following procedures:
      input( buffer + offset, stream, length ) : Data inputting from the character stream;
      output( length, index ) : Data outputting as a length/index code-word;
      output_Char( character ) : Data outputting as a single character code-word.
*/
function encodeLZ_MLB( stream ) { /* stream is the input stream */
      buf = array( 0..M-1 ); /* buf is the data stream buffer, and Eq (5) is always working */
      pos = N; /* pos traces the current position of the data window (M=2N) */
      define: s = buf + pos; /* s[0..N-1] is the data window, and Eq (6) is always working */
      link = array( 0..M-1 ); /* link stores the matching links for buf */
      bucket = array( 0..256^3-1); /* bucket stores the link headers of Eq (3) */
      link[ 0..M-1 ] = null; bucket[ 0..256^3-1 ] = null; /* initialize the matching links and link headers */
      repeat do { /* Hint: positions of buf form a congruent group */
         remain = input( buf + N - pos, stream, N); /* remain gets the length of current input data */
         if ( remain =0 ) do { return; } /* this algorithm ends */
         for n = 0..remain-1 do { /* Phase (a): build matching links */
            i = N – pos + n; j = buf[ i..i+2 ]; p = bucket[ j ];
            if not ( p = null ) and not ( buf[ p..p+2 ] = j ) do { bucket[ j ] = null; } /* clear the expired header value */
            if ( bucket[ j ] = null ) do { link[ i ] = null; }
            else do { link[ i ] = i - bucket[ j ]; } /* link stores the distance (i.e. relative position) of Eq (4) */
            bucket[ j ] = i; /* bucket stores the plain position of Eq (3) */
         }
         while ( remain >0 ) do { /* Phase (b): LZ77 string match and encode */
            if ( remain <3 ) do {
               for i = 1..remain do { output_Char( s[N] ); pos = pos+1; } /* output the remaining characters */
               break; /* jump out of the "while" cycle */
            }
            n = pos + N; dist =0; /* dist traces the distance between s[N] and current matching point */
            len =1; index =0; /* len and index trace the current maximum matching length and its relative position */
            for i = 1..sight do { /* sight controls the matching times */
               d = link[ n - index ]; /* d is the distance from current matching point */
               if ( d = null ) or ( dist + d >= N ) do { break; } /* jump out of the "for" cycle */
               dist = dist + d; x = N; y = N - dist;
               for ( l = 0..remain-1 ) and ( s[x] = s [y] ) do { x = x+1; y = y+1; } /* get the matching length l */
               if ( l > len ) do { len = l; index = dist; } /* find a better matching point */
            }
            If ( len >2 ) do { output( len, index ); } /* output a length/index code-word */
            else do { output_Char( s[N] ); len =1; } /* output a single character */l
            pos = pos + len; remain = remain - len;
         }
      }
}
```

Algorithm 3: MLB Coding ($k = 3$).

So the time complexity of RAID-0 encoding/decoding is $O(N/n) = O(N)$.

As shown in Algorithms 1 and 2 and Figure 6, BAB coding is similar to RAID-0. The difference is BAB coding has to treat multiple streams with different lengths, but this operation is also fast and does not influence the performance. So the time complexity of BAB coding is also $O(N)$. We can expect the practical BAB coding is as fast as RAID.

*4.1.2. MLB and LZ77 Encoding.* As shown in Algorithm 3 and Figure 8, we can divide the algorithm into 2 parts: Phase (a) is building the matching links, and Phase (b) is LZ77 encoding with matching links.

Phase (a) is similar to Phase 1 of CZ-BWT encoding in [6]. We have discussed the time complexity of bucket sorting is $O(N)$ in [6]. $N$ is the data window (BWT block) size. So Phase (a) also has the time complexity $O(N)$.

With the aid of matching links, Phase (b) is faster than the traditional LZ77 algorithm. According to the principle of LZ77 compression [5], the key computation of LZ77 encoding is the string matching, which determines the encoding speed. The traditional LZ77 encoding has the complexity of time $O(N^2)$ because it has to find the matching points by seeking the data window, a byte after another.

The software gzip also uses matching links to accelerate the LZ77 encoding, but each time it traces a whole link to find all matching points in the data window, so the time complexity is still $O(N^2)$. While 7-zip uses binary trees for LZSS encoding. Seeking a binary tree is faster than tracing a

matching link, so the time complexity of 7-zip (LZSS) encoding is O($N$lb$N$).

We have a faster way in Phase (b) of Algorithm 3. A configurable parameter *sight* is used to control the amount of the applicable matching points in each link. Then we need not trace the whole matching link. Take sight = 10 for example, each time only the first 10 matching points in the link are inspected to find their maximum matching length. In practice, we regard *sight* as a constant, so the time complexity of ComZip (LZ77) encoding is O(sight $N$) = $O(N)$.

As we have investigated, lz4 is one of the fastest parallel compression software of the LZ series. It has the top speed because each time it just inspects a single matching point, and it makes full use of the CPU cache. Then, we may roughly regard lz4 as a fast LZ77 with the short matching links of sight = 1. The time complexity of lz4 encoding is also O($N$).

In [6], we have discussed bzip2 (BWT) encoding has the time complexity O($N^2$lb$N$).

### 4.2. Space Complexities

*4.2.1. BAB Coding.* The memory usage is important for the sensors. As shown in Algorithms 1 and 2, BAB coding needs a block buffer and a set of arrays. The block buffer has space complexity O($B$). $B$ is the block size. The stream information arrays have space complexity O($n$). $n$ is the row amount of the array. But $n$ is very small, so the space complexity of BAB coding is O($B$). $B < N$.

*4.2.2. MLB and LZ77 Encoding.* The software gzip has a 32 KB data window. It needs the RAM 10 N (2 N for dual window buffer, 4 N for matching links, and 4 $N$ for 4*32 KB hash table). So, the space complexity of gzip (LZ77) encoding is O($N$). And lz4 is a multithread parallel compression software. Each thread has a 64 KB data window and needs the RAM 1.25 N ($N$ for window buffer, and 0.25 $N$ for 16 KB hash table). So the space complexity of lz4 encoding is O($pN$). $p$ is the amount of the parallel threads. In [6], we have discussed bzip2 (BWT) encoding needs the RAM 2.5 N. It has the time complexity O($N$).

Both 7-zip and ComZip support multithread parallel compression, and they also support the large data window at the GB level. Each thread of 7-zip (LZSS) encoding needs the RAM 9.5 N (mainly for the binary tree). So, the space complexity of 7-zip (LZSS) encoding is O($pN$).

ComZip needs the RAM for the window buffer, matching links, and link headers. In Figure 8, the stream buffer is a dual window buffer and needs the RAM 2 N, but in ComZip we have optimized the dynamic RAM usage which needs 1 − 2 $N$. The matching links need the RAM 4 N (supporting 2 GB data window) or 5 N (supporting 512 GB). The link headers form a bucket array with $256^k$ elements. If $k = 3$, a bucket array needs the RAM 64 MB (supporting 2 GB data window) or 80 MB (supporting 512 GB), which can be ignored if the data window is larger than 16 MB. In summary, ComZip (LZ77) encoding needs the RAM 5-7 N, so it has the space complexity O($N$).

*4.3. Compression Ratios.* The LZ series compression bases on the string matching. So the compression ratio depends on the matching opportunities and maximum matching length of each matching point. If a matching point goes out of the data window, it cannot contribute to the compression ratio.

BAB coding rebuilds the data stream so that the multiple streams can be seen in the same data window. If the streams are similar, many matching points will benefit the compression ratio. But if the streams are completely different, the compression ratio will slightly decrease, like the effect of reducing the data window size. These results were observed in the data experiments, and we may infer an experimental rule: a closer matching point may have a larger matching length.

The data in practice are complex, so we need to do various data experiments for this rule modeling. This paper just takes a simple example to explain these BAB effects. According to Equation (1) and Figure 4, we assume $D_{zip}$ has the following trend:

$$D_{zip} = O\left(lbN^{-1}\right) \tag{7}$$

$N$ is the data window size. When BAB mixes $n$ streams, the data window size for each stream is $N/n$. Then, Equation (7) becomes

$$D_{zip} = O\left(lbN^{-1}n\right) \tag{8}$$

We regard $N$ as a constant in this example. Then, Equation (8) becomes

$$D_{zip} = O(lb\, n) \tag{9}$$

Equation (9) infers that the compression ratio will slightly decrease if the streams in the block array are completely different. But if the streams are similar, as a simple example we may assume $D_{zip}$ has the following trend:

$$D_{zip} = O\left(n^{-1}\right) \tag{10}$$

The effects of Equations (9) and (10) get together, the results will show the main factor Equation (10), which infers the compression ratio will increase.

We may still use Equation (7) as an example to estimate the compression ratio limited by the data window size, but we should also consider the influence of the compression format. The 32 KB window of gzip is far away from the 1 GB data window of 7-zip, but the length/index code-word of gzip is short, which helps to improve the compression ratio. After all, the size of 32 KB is too small, we may expect the compression ratio of gzip is the lowest in our experiments. But the results show lz4 is the lowest.

Lz4 is designed for high-speed parallel compression. Each compression thread has an independent 64 KB data window. But to reach the time complexity of O($N$), each time in the LZ encoding it tries only one matching point, which will lose the remainder of matching opportunities in the window. As a contrast, gzip tries all the matching points in the 32 KB

Table 5: Comparison of data encoding.

| Compression encoding | Window size support | Time complexity | Space complexity | Compression ratio |
|---|---|---|---|---|
| gzip (LZ77) | 32 KB | $O(N^2)$ | $O(N)$ | Very low; small data window to keep the speed |
| lz4 (LZ77) | 64 KB | $O(N)$ | $O(pN)$ | Very low; small data window and single matching point for fast compression |
| bzip2 (BWT) | 900 KB | $O(N^2 \text{lb} N)$ | $O(N)$ | Low; small data window to keep the speed |
| 7-zip (LZSS) | 4 GB | $O(N \text{lb} N)$ | $O(pN)$ | High; independent data windows for multi-threads limit the window size |
| ComZip (BAB) | 512 GB | $O(N)$ | $O(B)$ | High for similar data streams |
| ComZip (MLB) | 512 GB | $O(N)$ | $O(N)$ | Matching point control for speed slightly decreases the compression ratio |
| ComZip (LZ77) | 512 GB | $O(N)$ | $O(N)$ | High; shared data window for multi-threads |

window, and it also has Huffman coding, so its compression ratio is higher than lz4.

Although bzip2 uses BWT encoding instead of the LZ series, we may still use the example of Equation (7), which is supported by the experiment results. Because the data windows of gzip, lz4, and bzip2 are all small, their compression ratios are uncompetitive. And if they use larger windows, their compression speed will slow down.

7-zip and ComZip have the advantage of large data windows in the level of GB, so their compression ratios are high. And both of them support multithread parallel compression for high speed. But we ought to consider the RAM consumption of the multiple threads. If the RAM of a sensor is limited, can it still support the large window for a high compression ratio?

Like the behavior of lz4, 7-zip also uses independent data windows for the parallel threads. As mentioned above, 7-zip (LZSS) encoding has space complexity $O(pN)$. For example, if it uses $p = 16$ threads, and each thread uses the $N =$ 512 MB window, the total window size is $pN = 8$ GB. On the other hand, if the RAM limits the total window size to $pN = 512$ MB, the independent window for each thread is only $N = 32$ MB. So the parallel 7-zip encoding decreases the compression ratio by $N$.

ComZip shares a whole data window for all parallel threads. If the total window size is $N = 512$ MB, each (LZ77) encoding thread can make full use of the 512 MB window. So the parallel encoding of ComZip can keep the high compression ratio.

Moreover, ComZip (LZ77) encoding has the time complexity $O(N)$, which implies it has the latent capacity to enlarge the data window for a higher compression ratio, without obvious rapid speed loss. As a contrast, 7-zip (LZSS) encoding has the time complexity $O(N \text{lb} N)$.

ComZip uses *sight* to control the amount of the applicable matching points for high speed and skips the matching link remainder. This will decrease the compression ratio slightly, but we can enlarge either $N$ or *sight* to cover this loss.

Table 5 shows the comparison of CZ-Array and other LZ/BWT encoding of these popular compression software.

*4.4. Complexities of Hardware Design.* Hardware acceleration is valuable for the sensors which have limited computing resources. The algorithms of CZ-Array, including BAB and MLB coding, are concise for the hardware implementation.

As shown in Figure 6, BAB and RAID coding are alike. In practice, the hardware of the RAID controller is mature and fast, which may benefit the BAB hardware implementation. Figure 9 shows the primary hardware design of BAB. The encoder uses a multiplexer to mix the data streams, and the decoder uses a demultiplexer to divide the united array stream. The accumulators count the clock. When a block is outputted, the multiplexer/demultiplexer will switch to the next data stream.

This is the primary design that can merely treat the data streams with the same length. The implementation of Algorithms 1 and 2 needs additional logic circuits for the situation control.

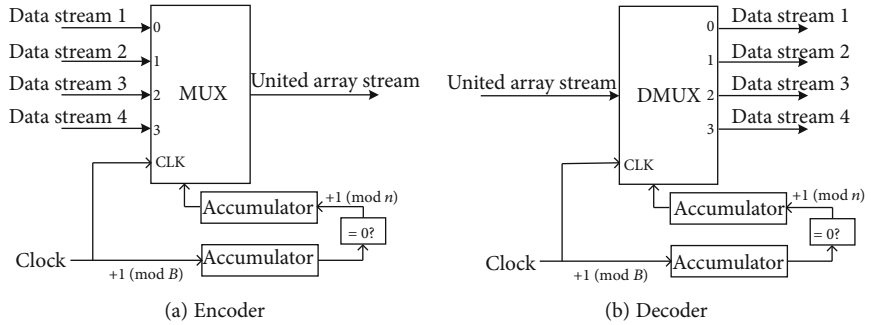
Figure 10 shows the primary hardware design of MLB. To show the design clearly, we use 3 blocks of RAM. But in practice, we can use the same block of RAM and divide it into 3 data areas by the address. This is also the primary design, and the implementation of Algorithm 3 Phase (a) needs additional design in detail. For example, this design cannot stop and reset the accumulator when it goes out of the data window.

In the view of hardware, both Figures 9 and 10 are succinct and easy to optimize the hardware speed.

## 5. Experiment Results

In [3, 4, 6], we have done some experiments to compare ComZip and other compression software, but they are in different versions and different cases. This paper focuses on CZ-Array and compares ComZip with p7zip (the Linux version of 7-zip), bzip2, lz4, and gzip. These compression software are popular, and a lot of other compression software have compared with them. We may just compare ComZip with them because the comparison with others can be estimated.

In this paper, we use the parameter "-9" for gzip and lz4 to gain their maximum compression ratio. For ComZip itself, we use $n = 4$ to test BAB coding and then use $n = 1$ to simulate ComZip without BAB. This is workable by modifying the parameter "column" in the configuration file "cz.ini". We also use sight = 20 and $B = 1$ MB by modifying the parameter "sight" and "cluster."




(a) Encoder (b) Decoder

Figure 9: Primary hardware design of BAB ($n = 4$).

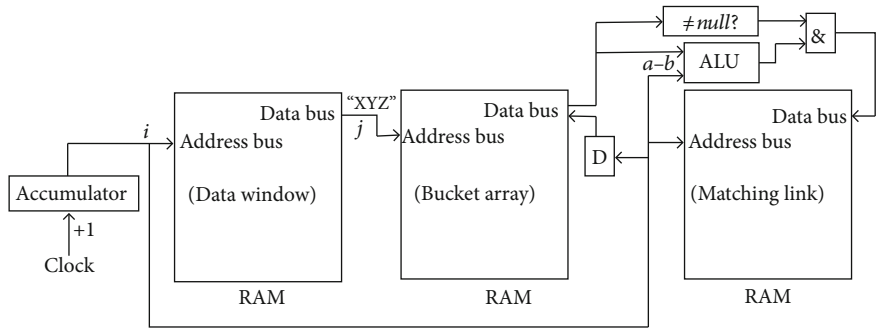



Figure 10: Primary hardware design of MLB ($k = 3$).

Table 6: Experiment data file.

| File name | Data file size (B) | Method of fetch |
|---|---|---|
| lede-imagebuilder-17.01.0-x86-64.Linux-x86_64.tar | 128 225 280 | Download and decompress the xz file. |
| lede-imagebuilder-17.01.1-x86-64.Linux-x86_64.tar | 128 716 800 | |
| lede-imagebuilder-17.01.2-x86-64.Linux-x86_64.tar | 128 788 480 | |
| lede-imagebuilder-17.01.3-x86-64.Linux-x86_64.tar | 128 798 720 | |
| Total | 514 529 280 | |

We find 4 public files to simulate the data streams, and thank the developers of the OpenWRT project. Table 6 shows these files. They can be downloaded from http://downloads.openwrt.org/

The experiments in this paper are on 2 hardware platforms: x86/64 and ARM. Their performances may provide references to the future and current heavy sensor nodes. The operating systems of both experiment platforms are Linux. We still provide ComZip on the website, but the port is changed into 81. Researchers may use it to do more experiments with new data. It can be downloaded from http://www.28x28.com:81/doc/cz_array.html

*5.1. Tests on x86/64 Platform.* This platform is a common desktop computer with the following equipment: AMD Ryzen 2700X 8-core and 16-thread CPU, 64 GB DDR4 RAM, 250 GB SSD (solid state disk) and Ubuntu Linux 16.04 (x64). We regard this computer as a future high-end mobile sensor because the carrier's motor can provide enough energy. The software versions are ComZip v20180421 (64b), p7zip v16.02, bzip2 v1.0.6, lz4 v1.8.3, and gzip v1.6. And to enhance the efficiency of matching links, ComZip uses $k = 3.25$ in this experiment, which means the bucket array in Figure 8 has matching strings with 26b instead of 24b.

Table 7 and Figure 11 show the relationship of the compressed file size $D_{zip}$ and the data window size $N$. From Equation (1), we can find that this relationship is virtually the relationship of the compression ratio and $N$.

Table 8 and Figure 12 show the relationship between the compression/decompression time and $N$. Figure 12 hides the decompression time because we have not optimized the decompression of ComZip yet. The current decompression program is just to verify the compression algorithms of ComZip are correct, so it may be slower than other software. We focus on the compression performance first, and the optimization of decompression for ComZip is our future work.

In Figure 11 and Table 7, we observe that p7zip ($N = 512$ MB) has the highest compression ratio $R$, but we ought to know its total window size. In this experiment, p7zip has 16 parallel compression threads. If Table 7 shows $N = 512$ MB, the total window size is 8 GB. On the other hand, we may compare the compression ratios of ComZip ($N = 512$ MB)

Table 7: x86/64: comparison of compressed file size (B).

| Data window | ComZip with BAB | ComZip without BAB | p7zip | bzip2 | lz4 | Gzip |
|---|---|---|---|---|---|---|
| 0.03 MB | — | — | — | — | — | 234 066 273 |
| 0.06 MB | — | — | — | — | 261 855 600 | — |
| 0.1 MB | — | — | — | 231 491 898 | — | — |
| 0.2 MB | — | — | — | 228 255 698 | — | — |
| 0.3 MB | — | — | — | 226 771 672 | — | — |
| 0.4 MB | — | — | — | 225 478 871 | — | — |
| 0.5 MB | — | — | — | 224 732 694 | — | — |
| 0.6 MB | — | — | — | 223 767 839 | — | — |
| 0.7 MB | — | — | — | 223 097 527 | — | — |
| 0.8 MB | — | — | — | 222 165 703 | — | — |
| 0.9 MB | — | — | — | 221 587 446 | — | — |
| 1 MB | 223 780 016 | 225 163 088 | 191 232 707 | — | — | — |
| 2 MB | 127 784 120 | 220 720 560 | 186 481 526 | — | — | — |
| 4 MB | 126 559 056 | 214 789 848 | 179 617 113 | — | — | — |
| 8 MB | 107 222 416 | 210 012 232 | 175 085 076 | — | — | — |
| 16 MB | 106 406 528 | 174 312 656 | 148 283 229 | — | — | — |
| 32 MB | 104 671 480 | 167 591 040 | 132 927 341 | — | — | — |
| 64 MB | 93 272 224 | 166 164 400 | 130 137 294 | — | — | — |
| 128 MB | 91 735 488 | 92 899 880 | 96 912 630 | — | — | — |
| 256 MB | 91 472 240 | 92 575 008 | 96 763 233 | — | — | — |
| 512 MB | 91 476 872 | 92 516 000 | 79 314 280 | — | — | — |

and p7zip ($N = 16$ MB) because their RAM consumptions are near: ComZip ($5N$ or $6N$) vs p7zip ($16^*9.5N$).

So, we may consider ComZip with BAB has the best compression ratio among these software, and lz4 has the worst $R$ owing to the 64 KB small window and the single matching point algorithm. We also observe the compression ratio of ComZip with BAB is obviously higher than that without BAB. But when $N = 128$ MB or larger, ComZip gains the almost same compression ratio, despite whether it uses BAB. The reason is that the window is large enough to see 2 data streams or more, which has the same effect as BAB coding. Another special point is in $N = 1$ MB, ComZip also gains the almost same compression ratio, because $B = N$ leads to the invisible blocks for each other and useless BAB coding.

According to the compression speed shown in Figure 12 and Table 8, ComZip without BAB ($N = 512$ MB) is the fastest, while ComZip with BAB ($N = 512$ MB) and lz4 are very close to it. Both lz4 and ComZip have the advantages of compression speed because they have the time complexity O($N$). But we observe abnormal curves which show ComZip with a small $N$ is slower than itself with a large $N$. The reason is the large $N$ brings more matching opportunities and decreases the total encoding operations.

The curve of p7zip is complex. When $N < 16$ MB, p7zip compression is also fast. But when $N$ is between 16 and 64 MB, its speed decreases rapidly and fits the time complexity O($N$lb$N$). When $N = 128$ MB, the window can see 2 data streams and brings lots of matching opportunities, which changes the trend of O($N$lb$N$). When $N = 512$ MB, the speed decreases rapidly again because the memory requirement for data windows ($16^*9.5N$) exceeds the RAM (64 GB), and the VM (virtual memory) is used. Fortunately, the VM in SSD is faster than that in HDD (hard disk driver). As a contrast, ComZip can use 64 GB RAM to build a 10 GB data window ($6N$).

The compression ratios and speeds of gzip and bzip2 are all uncompetitive, so this paper simply provides their results.

Above all, the experiment results on this x86/64 platform show that CZ-Array in ComZip has the following advantages to use large data windows: high compression ratio and speed, shared window, and efficient RAM consumption. ComZip with BAB can have the highest compression ratio and speed among these software, which is practical for the long-stream data compression.

Moreover, the experimental data are virtual router image files of OpenWRT, and the results infer that CZ-Array in ComZip has the latent capacity to compress quantities of virtual machine files in the cloud platform with good performance.

*5.2. Tests on ARM Platform.* This platform is a popular Raspberry Pi 3B+ with the following equipment: ARM Cortex-A53 4-core CPU, 1 GB DDR2 RAM, 32 GB Micro SDXC (SD eXtended Capacity), and Raspbian Linux 9. We regard this Raspberry Pi as a current heavy node of mobile sensors which is inexpensive. The software versions are: ComZip v20180421 (32b), p7zip v16.02, bzip2 v1.0.6, lz4 v1.8.3, and gzip v1.6. In this experiment, ComZip uses $k = 3$.

Table 9 and Figure 13 show the relationship of the compressed file size $D_{zip}$ and the data window size $N$. Table 10 and Figure 14 show the relationship of the compression/decompression time and $N$.

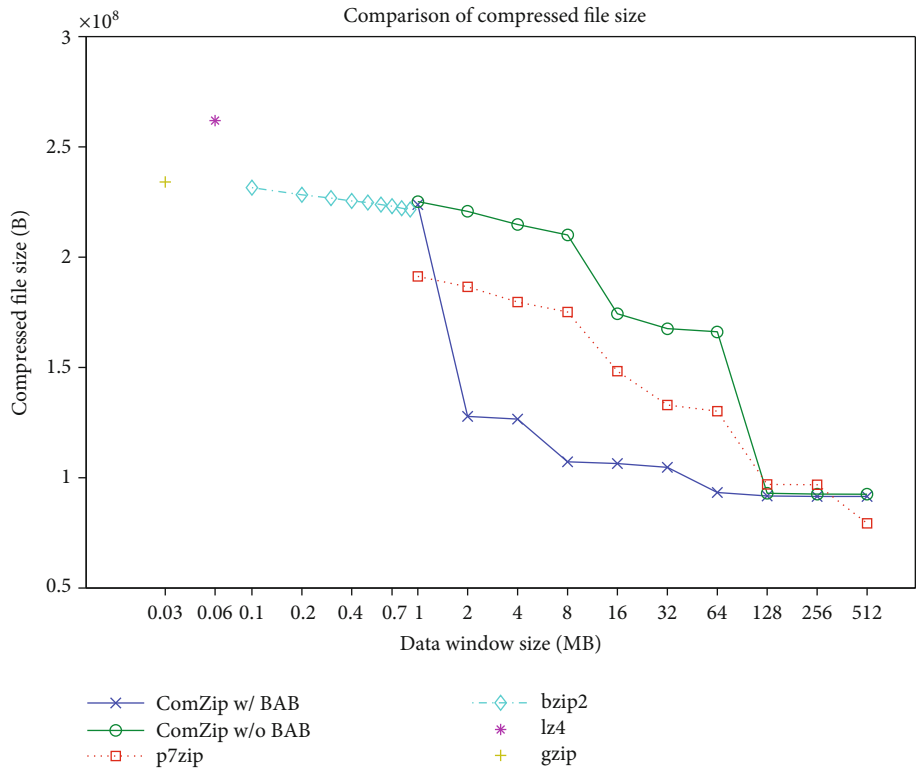



Figure 11: Compressed file size and data window size in Table 7.

Table 8: x86/64: comparison of file compression/decompression time (seconds).

| Data window | ComZip with BAB | | ComZip without BAB | | p7zip | | bzip2 | | lz4 | | gzip | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Encode | Decode | Encode | Decode | Encode | Decode | Encode | Decode | Encode | Decode | Encode | Decode |
| 0.03 MB | — | — | — | — | — | — | — | — | — | — | 60 | 3 |
| 0.06 MB | — | — | — | — | — | — | — | — | 15 | 1 | — | — |
| 0.1 MB | — | — | — | — | — | — | 41 | 18 | — | — | — | — |
| 0.2 MB | — | — | — | — | — | — | 39 | 19 | — | — | — | — |
| 0.3 MB | — | — | — | — | — | — | 39 | 19 | — | — | — | — |
| 0.4 MB | — | — | — | — | — | — | 40 | 19 | — | — | — | — |
| 0.5 MB | — | — | — | — | — | — | 40 | 20 | — | — | — | — |
| 0.6 MB | — | — | — | — | — | — | 40 | 19 | — | — | — | — |
| 0.7 MB | — | — | — | — | — | — | 42 | 19 | — | — | — | — |
| 0.8 MB | — | — | — | — | — | — | 42 | 18 | — | — | — | — |
| 0.9 MB | — | — | — | — | — | — | 42 | 19 | — | — | — | — |
| 1 MB | 18 | 26 | 19 | 26 | 18 | 10 | — | — | — | — | — | — |
| 2 MB | 18 | 16 | 19 | 25 | 18 | 10 | — | — | — | — | — | — |
| 4 MB | 20 | 15 | 22 | 24 | 20 | 9 | — | — | — | — | — | — |
| 8 MB | 18 | 14 | 19 | 24 | 21 | 9 | — | — | — | — | — | — |
| 16 MB | 18 | 14 | 19 | 20 | 21 | 8 | — | — | — | — | — | — |
| 32 MB | 18 | 13 | 19 | 19 | 32 | 7 | — | — | — | — | — | — |
| 64 MB | 19 | 12 | 20 | 19 | 59 | 7 | — | — | — | — | — | — |
| 128 MB | 19 | 11 | 16 | 12 | 65 | 5 | — | — | — | — | — | — |
| 256 MB | 18 | 12 | 16 | 11 | 64 | 5 | — | — | — | — | — | — |
| 512 MB | 15 | 12 | 14 | 11 | 128 | 4 | — | — | — | — | — | — |

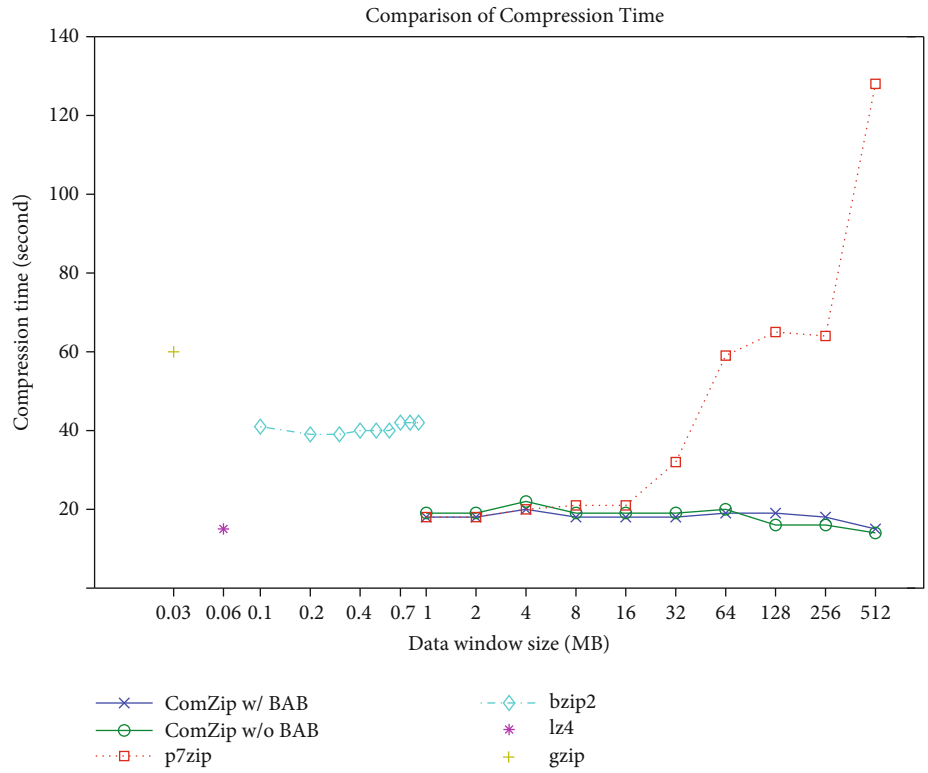



Figure 12: Compression time and data window size in Table 8.

Table 9: ARM: comparison of compressed file size (B).

| Data window | ComZip with BAB | ComZip without BAB | p7zip | bzip2 | lz4 | gzip |
|---|---|---|---|---|---|---|
| 0.03 MB | — | — | — | — | — | 234 066 467 |
| 0.06 MB | — | — | — | — | 261 738 789 | — |
| 0.1 MB | — | — | — | 231 491 802 | — | — |
| 0.2 MB | — | — | — | 228 255 442 | — | — |
| 0.3 MB | — | — | — | 226 773 089 | — | — |
| 0.4 MB | — | — | — | 225 478 012 | — | — |
| 0.5 MB | — | — | — | 224 733 757 | — | — |
| 0.6 MB | — | — | — | 223 766 291 | — | — |
| 0.7 MB | — | — | — | 223 090 167 | — | — |
| 0.8 MB | — | — | — | 222 267 771 | — | — |
| 0.9 MB | — | — | — | 221 587 957 | — | — |
| 1 MB | 227 955 840 | 230 038 056 | 191 232 707 | — | — | — |
| 2 MB | 130 044 680 | 224 420 768 | 186 481 526 | — | — | — |
| 4 MB | 128 748 280 | 218 482 480 | 179 617 112 | — | — | — |
| 8 MB | 109 182 888 | 213 749 592 | 175 085 076 | — | — | — |
| 16 MB | 108 328 480 | 178 654 224 | 148 283 229 | — | — | — |
| 32 MB | 106 662 856 | 172 251 192 | 131 328 609 | — | — | — |
| 48 MB | 97 264 744 | 171 096 888 | — | — | — | — |
| 64 MB | 95 014 848 | 170 786 384 | 129 993 993 | — | — | — |
| 80 MB | 94 946 696 | 165 405 752 | — | — | — | — |
| 96 MB | Insufficient RAM | Insufficient RAM | — | — | — | — |

This experiment is limited by the platform hardware, especially the 1 GB RAM. When $N = 96$ MB, ComZip aborts for insufficient RAM. But other software are worse. Only p7zip can use $N = 64$ MB ($p = 1$). $p$ is the number of parallel compression threads. And p7zip can only support $N = 32$ MB ($p = 2$) and $N = 16$ MB ($p = 4$), and the larger $N$ cannot work.

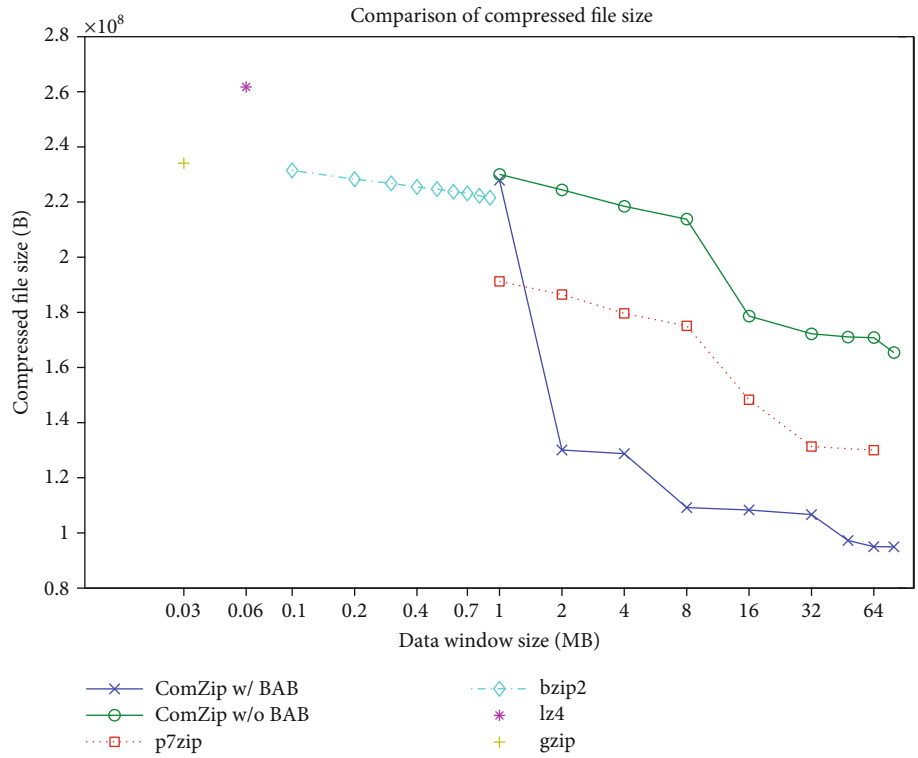



Figure 13: Compressed file size and data window size in Table 9.

Table 10: ARM: comparison of file compression/decompression time (seconds).

| Data window | ComZip with BAB | | ComZip without BAB | | p7zip | | bzip2 | | lz4 | | gzip | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Encode | Decode | Encode | Decode | Encode | Decode | Encode | Decode | Encode | Decode | Encode | Decode |
| 0.03 MB | — | — | — | — | — | — | — | — | — | — | 385 | 31 |
| 0.06 MB | — | — | — | — | — | — | — | — | 117 | 24 | — | — |
| 0.1 MB | — | — | — | — | — | — | 261 | 98 | — | — | — | — |
| 0.2 MB | — | — | — | — | — | — | 269 | 114 | — | — | — | — |
| 0.3 MB | — | — | — | — | — | — | 281 | 119 | — | — | — | — |
| 0.4 MB | — | — | — | — | — | — | 297 | 127 | — | — | — | — |
| 0.5 MB | — | — | — | — | — | — | 312 | 125 | — | — | — | — |
| 0.6 MB | — | — | — | — | — | — | 324 | 128 | — | — | — | — |
| 0.7 MB | — | — | — | — | — | — | 338 | 130 | — | — | — | — |
| 0.8 MB | — | — | — | — | — | — | 356 | 133 | — | — | — | — |
| 0.9 MB | — | — | — | — | — | — | 365 | 132 | — | — | — | — |
| 1 MB | 374 | 466 | 375 | 472 | 255 | 33 | — | — | — | — | — | — |
| 2 MB | 217 | 269 | 363 | 455 | 469 | 47 | — | — | — | — | — | — |
| 4 MB | 216 | 268 | 352 | 443 | 384 | 37 | — | — | — | — | — | — |
| 8 MB | 221 | 229 | 340 | 431 | 295 | 42 | — | — | — | — | — | — |
| 16 MB | 186 | 222 | 282 | 353 | 318 | 25 | — | — | — | — | — | — |
| 32 MB | 191 | 224 | 280 | 343 | 591 | 39 | — | — | — | — | — | — |
| 48 MB | 183 | 206 | 284 | 339 | — | — | — | — | — | — | — | — |
| 64 MB | 186 | 197 | 286 | 348 | 943 | 38 | — | — | — | — | — | — |
| 80 MB | 186 | 200 | 273 | 335 | — | — | — | — | — | — | — | — |
| 96 MB | Insufficient RAM | | Insufficient RAM | | — | — | — | — | — | — | — | — |

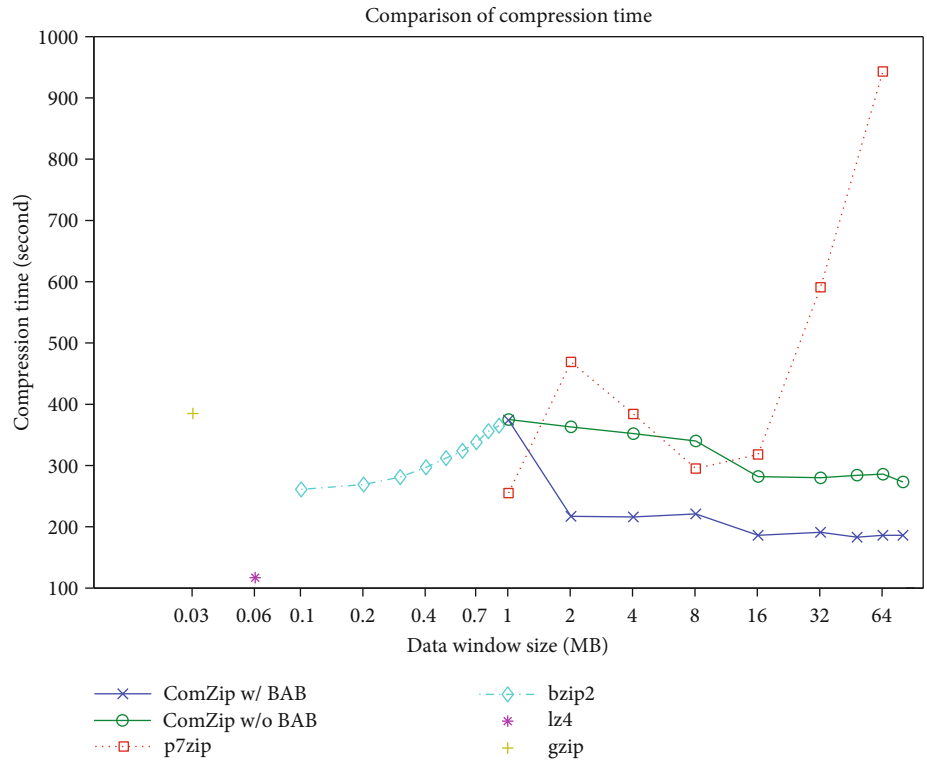



Figure 14: Compression time and data window size in Table 10.

Figure 12 and Table 9 show that in this experiment p7zip cannot reach $N = 128$ MB, so its compression ratio does not keep up with that of ComZip with BAB. Then, ComZip with BAB has the best compression ratio, and lz4 has the worst.

In Figure 14 and Table 10, we observe that lz4 is the fastest, and ComZip with BAB is the second fastest. We estimate the reason may be that the CPU cache of ARM is small, which fits the small 64 KB data window of lz4 to exhibit the performance. And to ComZip, the 4-core CPU limits the performance of the parallel CZ encoding pipeline.

The curve of p7zip is complex. When $N$ is between 2 and 8 MB, the larger $N$ brings more matching opportunities and benefits the speed, and the RAM is sufficient to support $p = 4$. When $N = 32$ MB ($p = 2$) and $N = 64$ MB ($p = 1$), the speed falls seriously because the compression threads are reduced. Again, this paper simply provides the results of gzip and bzip2 as the comparison.

Above all, the experiment results on this ARM platform also show that CZ-Array in ComZip has a high compression ratio and speed. Especially in the limited RAM of a sensor, in these software, only ComZip with BAB can see 2 streams or more in the data window, and its shared window brings efficient RAM consumption. On this platform, ComZip with BAB can have the largest data window, highest compression ratio, and the second fastest speed among these software, which is practical for the long-stream data compression.

From all of the above experiment results, we can get some support about the advantages of CZ-Array: large and shared window and efficient RAM consumption, high compression ratio and speed contrasting to the other popular compression software. And these results provide some references to the performance of CZ-Array running on x86/64 and ARM platforms, which may infer the feasibilities and practicalities of using CZ-Array in the future and current sensors.

Based on the current results, we may continue the research in the following points as our future works:

(1) Analyzing the relationship between the compression ratio and the matching opportunities

(2) Enhancing the MLB and LZ77 coding to obtain a better compression ratio

(3) Optimizing the decompression algorithms of ComZip for better speed

(4) Accelerating the encoding/decoding with GPU, ASIC or other hardware equipment

## 6. Conclusions

The rapid expansion of IoT leads to numerous sensors, which generate massive data and bring the challenges of data transmission and storage. A desirable way for this requirement is stream data compression, which can treat long-stream data in the limited memory of a sensor.

But the problems of long-stream compression in the sensors still exist. Owing to the limited computation resources of each sensor, enlarging the data window without a rapid decrease of the encoding/decoding speed is a problem. Compressing multiple data streams and seeing them in the same window limited by a sensor's RAM is another problem. If

the sensor needs hardware acceleration, simplifying the compression algorithms for the hardware design is necessary.

To solve these problems, this paper presents the CZ-Array algorithm, a block-split coding algorithm including BAB and MLB coding. CZ-Array is implemented in the shareware ComZip. It supports the data window up to 512 GB currently, which meets the requirements of long-stream data compression.

The analyses indicate that CZ-Array encoding has the time complexity O($N$), which exceeds 7-zip and keeps up with lz4, one of the currently fastest compression software. The space complexity of CZ-Array encoding is also O($N$), which exceeds 7-zip and lz4 in the multithread parallel compression. So the compression ratio of CZ-Array with the large data window can be high. The primary hardware design of BAB and MLB infers that the hardware acceleration for CZ-Array is relatively easy to realize.

The experiment results support that if CZ-Array in ComZip treats multiple streams with similar data it has the ability to obtain the best compression ratio and the fastest or second fastest compression speed, in the comparison with the popular compression software: p7zip, bzip2, lz4, and gzip. And these results provide references to the performance of CZ-Array running on x86/64 and ARM platforms, which may infer that it is feasible and practical to use CZ-Array in the future and current sensors.

On the other hand, these experiment results also provide some proof of the weakness in CZ-Array. Comparing to p7zip, the compression ratio of ComZip without BAB is not high enough. The reason is the parameter *sight* limits the matching opportunities.

## Data Availability

The data used to support the findings of this study are included within the article.

## Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

## Acknowledgments

This paper is supported by the R&D Program in Key Areas of Guangdong Province (2018B010113001, 2019B010137001), Guangzhou Science and Technology Foundation of China (201802010023, 201902010061), and Fundamental Research Funds for the Central Universities.

## References

[1] G. Y. Lee, S. H. Lee, and H. J. Kwon, "DCT-based HDR exposure fusion using multiexposed image sensors," *Journal of Sensors*, vol. 2017, Article ID 2837970, 14 pages, 2017.

[2] C. M. Li, K. Z. Deng, J. Y. Sun, and H. Wang, "Compressed sensing, pseudodictionary-based, superresolution reconstruction," *Journal of Sensors*, vol. 2016, Article ID 1250538, 9 pages, 2016.

[3] J. C. Qin and Z. Y. Bai, "Design of new format for mass data compression," *The Journal of China Universities of Posts and Telecommunications*, vol. 18, no. 1, pp. 121–128, 2011.

[4] J. C. Qin, Y. Q. Lu, and Y. Zhong, "Parallel algorithm for wireless data compression and encryption," *Journal of Sensors*, vol. 2017, Article ID 4209397, 11 pages, 2017.

[5] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," in *IEEE Transactions on Information Theory*, vol. 23no. 3, pp. 337–343, 1977.

[6] J. C. Qin, Y. Q. Lu, and Y. Zhong, "Fast algorithm of truncated Burrows-Wheeler transform coding for data compression of sensors," *Journal of Sensors*, vol. 2018, Article ID 6908760, 17 pages, 2018.

[7] M. Alistair, M. N. Radford, and H. W. Ian, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, 1998.

[8] M. Burrows and D. J. Wheeler, *A Block-Sorting lossless Data Compression Algorithm*, DIGITAL System Research Center, 1994.

[9] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[10] M. Alistair, "Implementing the PPM Data Compression Scheme," in *IEEE Transactions on Communications*, vol. 38no. 11, pp. 1917–1921, 1990.

[11] N. Timoshevskaya and W. C. Feng, "SAIS-OPT: on the characterization and optimization of the SA-IS algorithm for suffix array construction," in *4th IEEE International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pp. 1–6, Florida, USA, 2014.

[12] U. Baier, *Linear-time Suffix Sorting - a New Approach for Suffix Array Construction*, Master Thesis at Ulm University, 2015.

[13] F. Hussain and J. Jeong, "Efficient deep neural network for digital image compression employing rectified linear neurons," *Journal of Sensors*, vol. 2016, Article ID 3184840, 7 pages, 2016.

[14] G. L. Sicuranza, G. Ramponi, and S. Marsi, "Artificial neural network for image compression," *Electronics Letters*, vol. 26, no. 7, pp. 477–479, 1990.

[15] J. Jiang, "Image compression with neural networks - a survey," *Signal Processing: Image Communication*, vol. 14, no. 9, pp. 737–760, 1999.

[16] P. C. Tseng, Y. C. Chang, Y. W. Huang, H. C. Fang, C. T. Huang, and L. G. Chen, "Advances in hardware architectures for image and video coding – a survey," in *Proceedings of the IEEE*, vol. 93no. 1, pp. 184–197, 2005.

[17] S. M. Lee, J. H. Jang, J. H. Oh, J. K. Kim, and S. E. Lee, "Design of hardware accelerator for Lempel-Ziv 4 (LZ4) compression," *IEICE Electronics Express*, vol. 14, no. 11, pp. 1–6, 2017.

[18] U. I. Cheema and A. A. Khokhar, "High performance architecture for computing Burrows-Wheeler transform on FPGAs," in *Proceedings of the International Conference on Reconfigurable and FPGAs*, pp. 1–6, Cancun, Mexico, 2013.

[19] S. Funasaka, K. Nakano, and Y. Ito, "Adaptive loss-less data compression method optimized for GPU decompression," *Concurrency and Computation-practice & Experience*, vol. 29, no. 24, 2017.

[20] A. Ozsoy and M. Swany, "CULZSS: LZSS lossless data compression on CUDA," in *2011 IEEE International Conference on Cluster Computing*, pp. 403–411, Austin, TX, USA, 2011.

[21] M. Deo and S. Keely, "Parallel suffix array and least common prefix for the GPU," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 197–206, 2013.

[22] D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz, "Introduction to redundant arrays of inexpensive disks (RAID)," in *IEEE Computer Society International Conference: Intellectual Leverage*, pp. 112–117, San Francisco, CA, USA, 1989.