# Burrows - Wheeler Transform in lossless Data compression Problems on hybrid Computing Systems

Mikhail M. Rovnyagin[1], Sergey S. Varykhanov[2], Dmitry M. Sinelnikov[3], Viktor V. Odintsev[4]
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute), Moscow, Russian Federation
mmrovnyagin@mephi.ru[1] masmx64@gmail.com[2] dsinelnikov96@gmail.com[3] zakhams@gmail.com[4]

***Abstract*— Currently, hybrid computing systems and clusters based on them are used to solve an increasing number of various tasks. This article addresses the issue of lossless data compression on hybrid computing systems. There are sections describing the development and implementation of a stack of lossless data compression algorithms based on the Burrows - Wheeler transform (BWT), as well as a section on data sorting on hybrid computing systems as one of the BWT steps. At the end are the test results of the proposed algorithms.**

***Keywords— High performance computing; Сжатие данных без потерь; GPU; CUDA; BWT***

## I. INTRODUCTION

Hybrid computing systems and clusters are very popular in the tasks of processing large amounts of data. They find application in many industries to solve many different problems[4][5]. Data compression on hybrid computing systems is in demand for implementing Virtual Network Functions (VNF)[3]. Unfortunately, not all algorithms can be simply parallelized and ported to the GPU. Therefore, the topic of this article is devoted to the issue of efficient lossless data compression on hybrid computing systems. Most lossless compression algorithms consist of a strict sequence of steps that must be performed sequentially, so the task of parallelizing lossless data compression algorithms is not a trivial task.

This article is devoted to the development and implementation of an algorithm based on Burrows - Wheeler transform (BWT). This transformation, which is widely used in lossless data compression algorithms, is the basis of the bzip2 encoder compression algorithm stack, which takes the place of one of the main compressors in most *NIX like operating systems. Another common gzip encoder[8] uses the DEFLATE[7] algorithm.

BWT allows you to present data in the form of an encoded sequence of the same length as the original. However, in the encoded sequence, unlike the original, there will be a large number of repeated characters, which can then be quickly and efficiently compressed by other encoders.

## II. RELATED WORKS

This article is devoted to the development and implementation of a stack of algorithms for lossless data compression based on Burrows - Wheeler transform (BWT). The developed stack is designed to work on hybrid computing systems running CUDA[10].

The third chapter describes the BWT method and proposes a stack of algorithms based on it, designed to work effectively on the GPU.

The fourth chapter is devoted to a detailed description of the implementation of the modified Burrows - Wheeler transform. Forward and reverse transformation described

A significant part of the work is devoted to GPU sorting. This topic is important because sorting is one of the main steps of BWT and it is its performance that significantly affects the overall speed of the algorithm.

At the end of the article, tests were performed on the compression and performance of the algorithm proposed in this article. Tests performed on CPU and in hybrid CPU-GPU mode.

The topic of lossless data compression on hybrid computing systems was already discussed in [2], where a method was proposed for implementing the LZSS algorithm.

Also, while working on the article, the following sources were studied[1][6][4].

## III. BWT BASED ALGORITHM FOR GPU

Burrows - Wheeler transform (BWT) is an algorithm that does not compress data directly, but converts the original data into a form suitable for compression. All characters of the original sequence retain their values, but the order of their sequence changes.

The main idea of the algorithm is that if the same substrings were found in the source data, then their characters will form repeating sequences of characters after the conversion[9].

The lossless data compression algorithm proposed in this article is a stack of algorithms based on Burrows - Wheeler transform. Each algorithm from this stack either directly compresses the data or converts the data into a form suitable for compression. The algorithm works with sequences of bytes

 978-1-7281-5761-0/20/$31.00 

of arbitrary length. The list of algorithms is presented in table 1.

| # | Algorithm |
|---|---|
| 1 | Run Length Encoding |
| 2 | Burrows - Wheeler transform |
| 3 | Move To Front |
| 4 | Run Length Encoding |
| 5 | Interval encoding |

Tab. 1. Algorithm stack.

The first coding step is to use the Run Length Encoding (RLE) algorithm for the entire input sequence. If the number of repetitions of the same character is more than 3 times, then the first 4 repeating characters are written into the sequence, and a byte is stored behind them that stores the number of remaining repetitions of this character.

The second step is to apply the BWT transform to the sequence obtained in the previous step. In this case, the sequence is divided into fragments of a fixed length. The last fragment may be smaller.

The third step is to convert the sequence using the MTF algorithm (Move To Front). This allows you to replace most of the characters with the value “0”, because after the BWT conversion there will be a large number of identical characters in a string.

The fourth step uses the RLE algorithm (Run Length Encoding) again, but this time the data is converted to view when each character of the encoded sequence stores the character and the number of its repetitions. If the number of consecutive characters is more than 256, then the data is encoded with several code pairs <char, number of repetitions>.

Using data from the code sequence, statistics for entropy compression are calculated. In total, 3 probability tables are used. Each consists of 256 values. The first table stores the probability of occurrence of each of the characters. The second table contains the probabilities for each possible value of the "number of repetitions" field, for the symbol "0". The maximum length of repetitions can be 256 (see the previous paragraph). The third table also stores data on the probabilities of the value of the number of repetitions field, but for the remaining characters. This approach was chosen because after the BWT-> MTF chain a large number of long chains of zeros will be formed.

The fifth step encodes the data using interval coding[11], which in this case refers to a subset of (Prediction by Partial Matching) PPM (0) compression algorithms. The interval coding algorithm is quite simple, but can be implemented in different ways. In this paper, compression and decompression using integrated coding was implemented on the GPU to increase performance. The implementation of interval coding has many subtleties, non-standard solutions and pitfalls, however, its description is beyond the scope of this article.

The decoding process is performed in reverse order. Each of the algorithms in the compression stack is reversible.

## IV. BWT IMPLEMENTATION FOR GPU

**Forward transformation**

Implementing Burrows - Wheeler transform in its pure form is extremely inefficient. Having a message of length n, you must have at least $n^2$ memory. Therefore, instead of an array of strings, an array of pointers or indexes to the first character from each line is stored. And the line itself is obtained by passing in a circle from the first character to the previous one.

In its pure form, you can sort the lines by comparing them character by character. However, this method is extremely inefficient, since almost all sorting algorithms often compare elements with each other, and each comparison leads to a rather difficult string comparison operation. Especially negatively it affects similar lines, since at some stage of sorting neighboring elements are compared, and such comparisons force the line comparison function to look ahead for a lot of characters.

The situation is aggravated by the fact that just a few fairly long matching lines lead to a situation where the array has a lot of periods that match up to several tens of characters.

**Modified forward transformation**

To solve these problems, a modified version of the string sorting algorithm was developed and implemented. Two key features of this modification are line cache and incomplete sorting.

One of the first problems that the modification solves is random access to RAM. Despite the fact that in RAM theory, memory can access any cell in the same amount of time, in practice this is not so[12]. An effective solution was to create a cache of multiple line characters.

It is much more efficient to compare the whole numbers, instead of comparing the bytes one by one, so each line of the cache stores an unsigned integer of 32 or 64 bits, depending on the settings. The high byte of the number stores the high character.

Since the cache is formed on each sort, its efficient filling is also important. Bytes of the number in most computing systems are stored on a low to high order.

In order not to load the processor with bitwise shifts and logical operations, the data is stored in the reverse order. At the beginning of the data there are several duplicate bytes that are filled with characters from the end of the data. This allows


978-1-7281-5761-0/20/$31.00 ©2020 IEEE

you to populate the cache by copying data as numbers, rather than byte-by-bit.

In addition to the line cache, there is an array of first character positions. Each position is stored in an unsigned 32 bit integer. low bits store the position of the first character of the string, large 8 bits - the last character of the string, and the 23rd bit is reserved for the flag of the last character.

Storing the last character helps to simplify sorting, and the flag of the last character to get rid of the end of line character and solve the problem of uncertainty in the position of the last character in incomplete sorting. The cache of lines and the array of positions are more clearly illustrated in Figure 1.

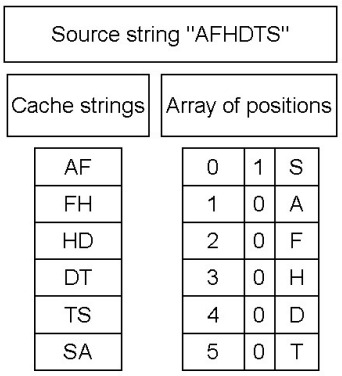


Fig. 1. The contents of the line cache and position array.

**Reverse transformation**

The reverse of the Burroughs - Wheeler transformation is multiple stable sorting of strings by the first character of a string. The conversion algorithm itself was described earlier in the BWT chapter.

However, every time a column is added to the row matrix, sorting by the first character will cause the same permutations, so it does not make sense to sort many times, but just do it once and remember where each row moved.

In the implementation of the inverse transform, only an array is stored that contains the positions and values of the corresponding bytes from the converted sequence. Physically, all this data is stored in exactly the same way as with direct conversion. Each cell consists of a 32 bit unsigned number. The high byte stores the value of the corresponding character after conversion, and the lower 3 bytes are reserved for the position number. The last character bit is not used.

Since it is necessary to store the number of the initial position of the symbol, the need to use stable sorting disappears. Instead, the resulting array of characters and positions is simply sorted by the values of the unsigned numbers contained in it. Characters are in the first places, so sorting will sort by characters anyway, and groups consisting of identical characters will be sorted by position number. As a result, the result is exactly the same as using stable sorting.

After the conversion, an array is obtained, where each element stores the number of the position this line will go to after sorting. All these transitions form one large cycle for all elements. This cycle is shown in figure 2. The left column shows the index of the array element, the middle column shows the character that corresponds to it, and the right column shows the number of the element to go to. As a result of BWT conversion, the sequence “ABDCE” turns into “BECAD”

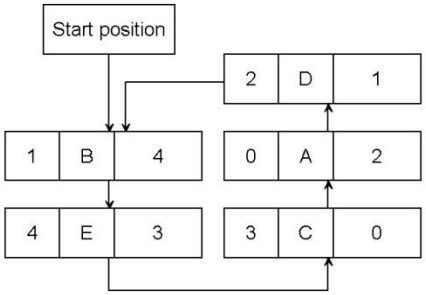


Fig. 2. The formation of the transition cycle.

During the direct conversion, the number of the last character was recorded, now this number is interpreted as the number of the last character, and it is with it that the cycle of transitions through the elements of the array begins.

Having done a full cycle through the elements and writing out all the source symbols from each element, we get the source data that was before the conversion, but only in the reverse order.

## V. GPU-CPU HYBRID SORTING

The most expensive from the point of view of processor resources is sorting in the BWT algorithm, therefore, for coding and decoding, it is necessary to use a fast sorting algorithm that will sort arrays of caches and line positions from several KB to several MB in size.

In total, as part of the research, many sorting methods were tested that were performed on the GPU or in hybrid mode on the CPU and GPU. The most effective method turned out to be when the elements are divided into subarrays of the maximum size, which is placed in the shared memory of the video card, and then sorted by the bucket Sort algorithm. Each CUDA block sorts its subarray with the maximum possible number of threads per block, and then the resulting arrays are merged using merge sort, which combines 4 or 8 subarrays into one at once. Usually you have to do three rounds of sorting. The specific parameters depend on the size of the array. For example, having a string cache size of 8 characters, 4096 elements are placed in shared memory. If the initial block size is 1MB, then you will need to perform three rounds of merge sort, which will combine 8, 8 and 4 arrays each.

One or both rounds of merge sort can be submitted to the CPU. However, taking into account that the CPU is loaded

 978-1-7281-5761-0/20/$31.00 ©2020 IEEE

with processing of other sorting steps, it was decided in the final version to leave merge sort completely on the GPU side.

A visual representation of this method is presented in Figure 3.

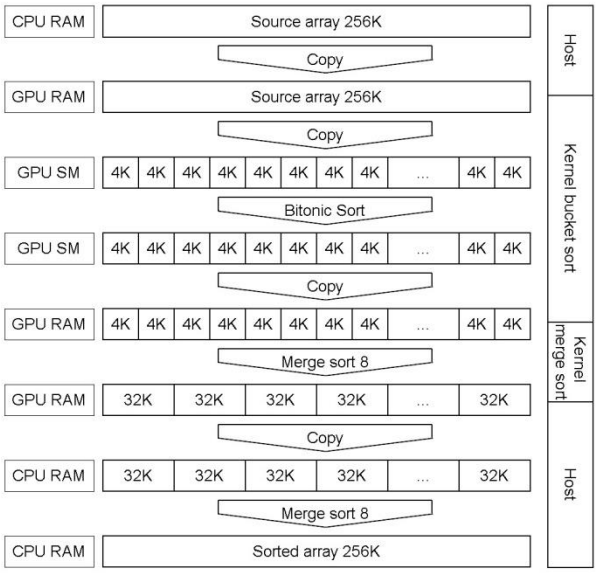


Fig. 3. Hybrid Array Sort Method.

## VI. Test and Results

Below are the results of testing the performance and compression ratio for the developed compression algorithm based on the BWT transform. Since the main parameter that affects the quality of compression and speed is the size of the block for conversion, in the first test we measured the compression efficiency for various types of data and various block sizes. The results are presented in Figure 4.

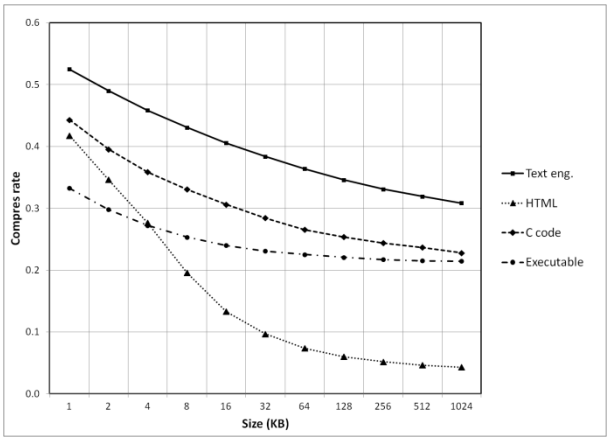


Fig. 4. Compression ratio for various file types.

It can be noted that most data formats come to a point where an increase in block size no longer affects the compression ratio, when the block size reaches about 512 KB.

After that, the results of performance testing for CPU and CUDA implementations are presented. The results are presented in Figure 5.

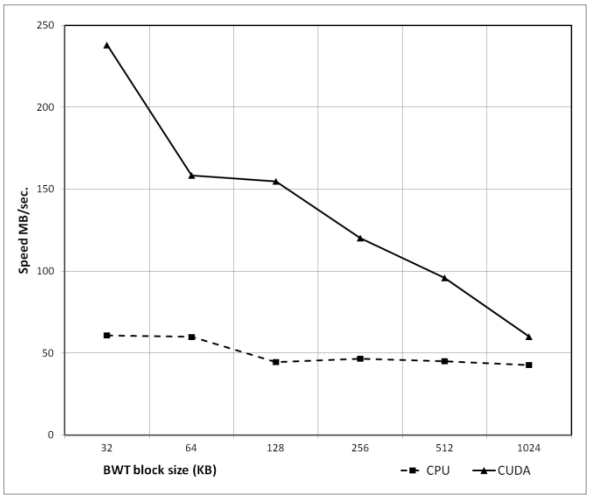


Fig. 5. CPU and CUDA implementation coding speed.

GPU utilization reaches its highest value with a small block size. This is because the first sorting stage is the most efficient, and the subsequent merge sort cannot be parallelized to a sufficiently large number of fragments.

## Conclusion

The lossless data compression algorithm proposed in this article demonstrates good results in terms of compression level and speed. The CPU version of the BWT-based algorithm showed multithreaded performance: 50 MB/s, which is significantly higher than the existing lossless data compression tools, while the compression quality remains at a high level. The CPU-GPU version reaches a speed of 240 MB/s.

As a result of a large number of tests, it was concluded that the main factors that affect performance are the speed of the data transfer interface between the CPU and GPU and the speed of the GPU memory subsystem.

However, despite this, the authors of this article believe that the introduction of lossless data compression algorithms in hybrid computing systems today can be more effective than using standard lossless data compression tools for the CPU.

## References


[1] Donald E. Knuth The Art of Computer Programming, vol.3. Sorting and Searching - M.: Williams, 2017. – 824p.

[2] Mikhail M. Rovnyagin, Sergey S. Varykhanov, Yury V. Maslov, Iuliia S. Riakhovskaia, Oleg V. Myltsyn, "NFV chaining technology in hybrid computing clouds", Young Researchers in Electrical and Electronic Engineering (EIConRus) 2018 IEEE Conference of Russian, pp. 109-113, 2018.

[3] Mikhail M. Rovnyagin, Alexey A. Kuznetsov, "Application of hybrid computing technologies for high-performance distributed NFV systems", Young Researchers in Electrical and Electronic Engineering (EIConRus) 2017 IEEE Conference of Russian, pp. 540-543, 2017.

[4] Pavel V. Sukharev, Nikolay P. Vasilyev, Mikhail M. Rovnyagin, Maxim A. Durnov, "Benchmarking of high performance computing


 978-1-7281-5761-0/20/$31.00 ©2020 IEEE

clusters with heterogeneous CPU/GPU architecture", Young Researchers in Electrical and Electronic Engineering (EIConRus) 2017 IEEE Conference of Russian, pp. 574-577, 2017.
[5] A. A. Kozlov, A. A. Aleshina, I. S. Kamenskikh, M. M. Rovnyagin, D. M. Sinelnikov, D. A. Shulga, "Increasing the functionality of the modern NoSQL-systems with GPGPU-technology", Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW) 2016 IEEE NW Russia, pp. 242-246, 2016.
[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms - M.: The MIT Press, 2009. – 1292p.
[7] "RFC 1951 DEFLATE Compressed Data Format Specification version 1.3" May 1996.
[8] Gailly, J.-L., and Adler, M., GZIP documentation and sources, available as gzip-*.tar in ftp://prep.ai.mit.edu/pub/gnu/.
[9] Burrows–Wheeler Transform / URL: https://www.dcode.fr/burrows-wheeler-transform (Access date: 12.11.2019)
[10] CUDA C Programming Guide / URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide (Access date: 12.11.2019)
[11] Interval encoding / URL: https://www.intuit.ru/studies/courses/1069/206/lecture/5324?page=5 (Access date: 12.11.2019)
[12] Encyclopedia of Parallel Computing — M.: Springer, 2011. – 146p.

 978-1-7281-5761-0/20/$31.00 ©2020 IEEE