Accelerating Lossless Compression with GPU

Dhammika Marasinghe*, Janitha Samarasinghe†, Nathasha Naranpanawa‡ and Roshan Ragel§
Department of Computer Engineering
University of Peradeniya
Peradeniya 20400 Sri Lanka
*dhammikammdb123@gmail.com, †jcsam93@gmail.com, ‡nathashanaranpanawa@gmail.com, §roshanr@pdn.ac.lk

***Abstract*—With the increasing amount of data used for various purposes, maintaining the efficiency of storing and transmitting them are proving to be a challenge. Data compression brings forward a solution to this issue.**

We perform a lossless compression of data on Graphic Processor Units (GPU) for NVIDIAs CUDA Framework, using a parallelized Lempel-Ziv-Storer-Szymanski (LZSS) compression algorithm. Algorithms were implemented using PFAC library and KMP along with LZSS. Performance results were obtained for serial and parallel CPU implementations of the LZSS algorithm as well as CUDA implementations. The improved algorithms were also benchmarked with gzip and bzip2, which are popular compression programs. We achieved a speedup of 5x for the parallelized LZSS CPU implementation compared to the serial LZSS algorithm.

Keywords: LZSS, PFAC, Compression, GPU.

I. INTRODUCTION

The computing community constantly deals with the ever increasing amount of data. The number of applications working with huge amount of data has also grown over the years. To be able to store and transmit these data in an efficient form is a challenge given that the state of resources and facilities does not increase at the same pace as the amount of data. Data compression is one of the popular solutions brought forward for this issue.

Data compression can be lossy or lossless. With lossy compression, some of the data such as redundant information will be eliminated. When the file is decompressed, only a part of the original data will be retrieved. But with lossless compression, no data is omitted and the original data is restored completely at decompression.

But again, using compression is a challenge itself, given that extra overhead and performance issues have to be considered. The execution time and the compression ratio are critical elements of compression. Many compression algorithms are subjected to these issues when performed on CPU. Lossless compression requires a considerable amount of execution time when performed on CPU, which results in a performance degradation.

The opportunity of parallelizing of compression algorithms on graphic processing units (GPUs) is useful in this context. By using NVIDIA GPUs Compute Unified Device Architecture (CUDA) Framework, parallel compression algorithms can be implemented, reducing the overhead and improving performance. Given the architectural strengths of GPUs, certain algorithms used for lossless compression have proven to be more effective on GPUs and compression can gain a very high speed-up compared to CPUs.

But compression algorithms are sequential. Hence, the speedup observed on GPU is not that significant for such sequential algorithms. For a significant performance gain, a parallelized lossless compression algorithm has to be implemented on GPU.

This paper presents an approach for lossless data compression accelaration. Section II discusses work that has been carried out regarding the same problem. Section III presents the background concepts related to the approach we have followed. Implementation details are given in Section IV and is followed by performance evaluation details in Section V. Conclusions are presented in Section VI.

II. RELATED WORK

Ozsoy & Swany (2011) [6] have examined the feasibility to use CUDA framework for LZSS lossless data compression algorithm.They presented two CPU versions and two CUDA implementations of LZSS. The CUDA implementations were optimized to utilize the global memory usage as well as the shared memory usage.Tests show that their work outperforms the serial LZSS by up to 18x. Some of the possible improvements on their work includes improved searching with better search algorithms, data structures suitable for CUDA implementation, etc.

The above work was improved upon by Ozsoy, Swany & Chauhan (2013)[7]. They have substantially improved the basic CULZSS algorithm by reducing control-flow divergence, making better use of the CUDA memory hierarchy, and overlapping CPU and GPU execution. They have also transformed the algorithm to work in a software pipeline across the CPU and the GPU, enabling it to operate on a stream of data.The throughput achieved for the improved algorithm was observed to be 34x better than the serial CPU implementation of LZSS algorithm and 2.21x better than the parallelized version. The paper discusses that in faster and more data generated cases, the step using Huffman algorithm can be a bottleneck among the other pipeline steps.

In 2014, Zu & Hua [8] presented a paper on parallelizing the LZSS algorithm using the NVIDIA CUDA framework to improve compression speed. The proposed method eliminates thread serialization by redesigning the algorithm process. Their performance evaluation focuses on compression speed, compression ratio, hash table size and decompression. For their implementation, a speedup of 2x was achieved for

compression over the improved version of CULZSS presented by Ozsoy, Swany & Chauhan (2013) [7].

Cloud et al. (2011) [9] have carried out accelerating lossless data compression with GPUs by using a modification of the Huffman algorithm. The modified implementation permits uncompressed data to be decomposed into independently compressible and decompressible blocks, allowing for concurrent compression and decompression on multiple processors. The paper presents that the GPU based encoder showed superior performance compared to the multi-core CPU encoder and single threaded CPU implementation. They also discuss how total encoding throughputs using the GPU are weighed down by the need to transfer data to and from the card.

WaveAccess, a software development company has done a research in 2011 [10] to explore the performance speedup of compression algorithms that can be achieved by utilizing the multi-core CUDA. A speedup of 4x was achieved on GPU for their implementation. Their implementation has not approached the Huffman encoding, given its sequential nature and requirements for synchronization. As future work, they have discussed implementing parallel BZIP2 for complete comparison of full-power CPU and GPU.

The paper presented by Dhore et al. (2015) [11] proposes an algorithm for video image compression using NVIDIA CUDA on GPU. The proposed framework used less storage space and also gave a better process time compared to the time required by the sequential implementation. A drawback of this implementation is that run length encoding is advantageous only when there is lots of consecutive data but is consumes more space if non-consecutive data is more.

In 2016, Tulsyan et al. [12] introduced an algorithm DKLZSS through their research on lossless compression. The algorithm introduces a dynamic KMP string matching method for parallel LZSS compression on GPGPUs. To improve the performance of the compression, they have split the input data into chunks and used a dynamic KMP algorithm on each chunk, which omits redundant processes in the original KMP algorithm when building the failure table. A main drawback of this method is that after the compression has been performed the compressed data from each thread is combined sequentially, which reduces the performance. However, tests show that this implementation outperforms the original LZSS algorithm in cases of high repetitions in the input data.

In summary, using algorithms such as Huffman, KMP and BWT could be disadvantageous given that they are sequential algorithms and hence much parallelization cannot be achieved. It is also seen that most of the performance issues of parallelized implementations are based on the efficiency of the string matching algorithms. Hence, we attempt to implement an accelerated lossless compression which focuses on the efficiency of string-matching.

III. BACKGROUND

A. Lossless Compression

We focus on lossless compression, which is a data compression technique in which the original data content of the compressed file can be retrieved exactly upon decompression. The file size is reduced, but it does not involve any loss of information. Upon reconstruction, the data will be identical to the original data of the file.

Basically, the data is rewritten in a more efficient manner when lossless compression is performed on a file. But since no content or quality is lost, the compressed file size will be larger than that of files compressed with lossy compression. In a best case scenario, lossless compression can reduce the file size by 50%, whereas lossy compression might be able to reduce it even further. Also, lossless compression is computationally expensive. Because of this, I/O performance of data transmission might not be improved even though the file size is reduced.

For applications and programs that need to process large amounts of data or improve upon them to yield more information, integrity of data needs to be preserved. Also, some programs might not tolerate lossy compression, such as text files or financial data. Lossless compression is useful in such situations.

B. LZSS Algorithm

Lempel-Ziv-Storer-Szymanski (LZSS) [2] is a dictionary encoding technique which is a derivative of LZ77. LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream.

LZSS algorithm replaces a string of symbols with a reference to a dictionary location of the same string. For this, a minimum number matching is required, and a bit flag indicates encoding or not. The difference between LZ77 and LZSS is that while in LZ77 the dictionary reference can be longer than the substring it replaces, the reference has to be shorter than the string in LZSS.

LZSS works with two buffers: a sliding window search buffer which searches the recently encoded text and a lookahead buffer with uncoded text.

The basic LZSS algorithm is as following;

Algorithm 1 LZSS algorithm

```
1:  procedure LZSS
2:      while lookahead_buf ≠ empty do
3:          find longest match in search_buf
4:          if match and L >= min_length then
5:              output ← (Position, L)
6:              search_buf ← (move by L chars)
7:              lookahead_buf ← (move by L chars)
8:          else
9:              output ← (1, char)
10:     ▷ char is the character at the head of lookahead_buf
11:             search_buf ← (move by1 char)
12:             lookahead_buf ← (move by1 char)
13:     return output
```

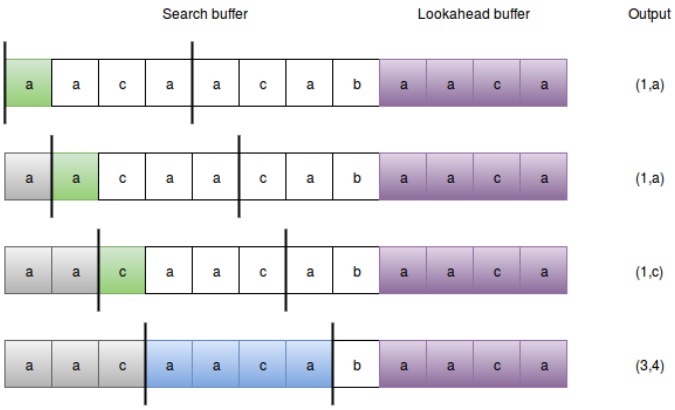

Fig. 1. An example of LZSS applied to a string

C. PFAC LIBRARY FOR AHO-CORASICK ALGORITHM

PFAC library [3] is an open source library with C level API. The abbreviation PFAC stands for the Parallel Failureless Aho-Corasick algorithm proposed in [4]. It is a variant of the well-known Aho-Corasick (AC) algorithm. PFAC accelerates pattern-matching by improving upon Aho-Corasick algorithm and achieves significant performance enhancements compared to traditional Aho-Corasick algorithm.

Aho-Corasick algorithm has two stages.

1) Converts multiple string patterns into a state machine
2) String-matching is done by traversing the state machine

PFAC improves by removing all failure transitions of Aho-Corasick algorithm and the self-loop transition of the initial state. Optimization techniques of PFAC also include reducing memory transactions of global memory and reducing latency of transition table lookup.

PFAC invokes two stages for string-matching.

1) Constructs a PFAC state machine
   The patterns to be matched are put into a state machine, with accepted states ending at each pattern construction.
2) Creates individual thread for each byte of an input stream.
   This is done in order to identify any pattern starting at the thread's starting position. The number of threads created is equal to the length of the input stream.

Performance analysis done by the developers show that PFAC achieves a significant speedup compared to traditional Aho-Corasick algorithm.

IV. IMPLEMENTATIONS

A. LZSS on CPU

1) Serial LZSS on CPU
   As a first step, a serial implementation of LZSS algorithm was executed on the CPU. The algorithm was adapted from the implementation by Okumura [5], which proposes the use of a binary tree. The algorithm works along the following steps;
   (i) A ring buffer is first initialized with only space characters, which sets the buffer at an empty state.
   (ii) Input characters are read into the ring buffer.
   (iii) Search the buffer for the longest string that matches the characters read.
   (iv) Send the length and position to the buffer if match found.
   The size of the ring buffer is set as 4096 bytes. Therefore, the position of the substring can be encoded in 12 bits. If the longest match is not longer than two characters, one character is sent as output without encoding. And the process is restarted with the next character. An extra bit is sent each time to indicate whether the output was encoded or not.
   This implementation uses multiple binary trees. As a result the speedup for the search of the longest match is increased.

- Encoding

The pseudo code for the encoding of the data is as following;
Step 1: Initialize the dictionary to a known value.
Step 2: Read an unencoded string that is the length of the maximum allowable match.
Step 3: Search for the longest matching string in the dictionary.
Step 4: If a match is found greater than or equal to the minimum allowable match length: Write the encoded flag, then the offset and length to the encoded output. Otherwise, write the unencoded flag and the first unencoded symbol to the encoded output.
Step 5: Shift a copy of the symbols written to the encoded output from the unencoded string to the dictionary.
Step 6: Read a number of symbols from the unencoded input equal to the number of symbols written in Step 4.
Step 7: Repeat from Step 3, until all the entire input has been encoded.

- Decoding

The pseudo code for the decoding process is as following;
Step 1: Initialize the dictionary to a known value.
Step 2: Read the encoded/not encoded flag.
Step 3: If the flag indicates an encoded string: Read the encoded length and offset, then copy the specified number of symbols from the dictionary to the decoded output. Otherwise, read the next character and write it to the decoded output.
Step 4: Shift a copy of the symbols written to the decoded output into the dictionary.
Step 5: Repeat from Step 2, until all the entire input has been decoded.

2) Parallel LZSS on CPU
   Given that parallelizing the LZSS algorithm on dictionary-based operations requires research which is

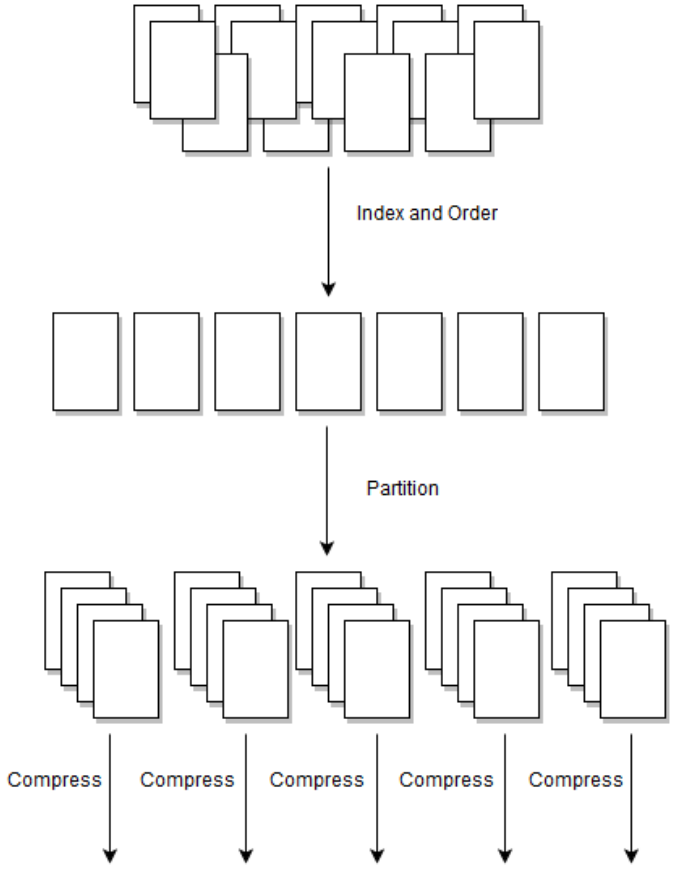

Fig. 2. Partitioning input data

out of the scope of the project, the attention was focused on using the SIMD architecture on the data to be compressed.

Considering a set of data that is very large in size, preferably a set of files for a game, each file is indexed. This will bring all the files into an order. Then, the set of files is partitioned into smaller chunks, in this case 8 sections.

Then, LZSS algorithm is executed on each of these sections of data. This process is parallelized by assigning a separate thread for each chunk of file, ensuring the performance is increased. The process is illustrated in Fig.2.

Each partitioned section of data is encoded separately using the serial LZSS algorithm. At the decoding stage, again each encoded section is decoded separately and appended in order. Since decoding does not undergo any searching process, it is fast and no optimizations are needed.

B. LZSS on GPU

1) Based on Okumura's LZSS

- Implementation Architecture

The CUDA implementation follows the same architecture of the parallel implementation of LZSS on the CPU.

As former research has been heavily based on parallelizing string comparisons, our implementation was focused on processing the file in smaller chunks thereby using the large memory and many cores of the Tesla K40 GPU in order to compress the file in parallel with minimal loss of compression ratio.

Similar to the CPU implementation, the file is read into a buffer and partitioned to a number of parts and each part is compressed in parallel in a SIMD manner. The compressed output is written to files from the output buffer once the compression is complete.

- I/O Improvements

As seen from profiling, it was found that a large amount of time was spent in reading from the file and writing to output files while compressing. The fact that we were implementing on CUDA meant that reading from a file would need CPU intervention. Due to this there was a major slowdown to be anticipated, therefore it was decided to read the entire file into a buffer then copy that buffer to the GPU. Instead of file writes as the output buffer overflows, memory was invested in a larger output buffer to store the entire encoded code.

Upon completion of all kernels, the memory was copied back to the CPU in order to be written to the output files.

- String Matching Optimizations

The implementation of LZSS that we were working on improving was implemented for optimal string matching performance with the utilization of 256 individual Binary Search Trees which hold the strings that were to be matched. Optimization of the string matching algorithm by implementing it parallely failed due to the dependency of the match in the history of matches. Therefore the BST implementation was favored over parallelization attempts as the research was focused on accelerating compression by SIMD approach.

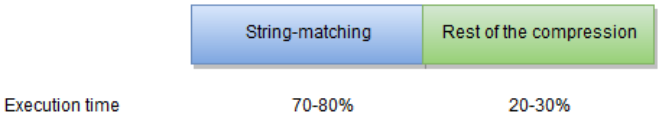

Fig. 3. Time taken for string-matching operations

Most of the performance issues of parallelized implementations are based on the efficiency of the string matching algorithm. Majority of the time is taken for string-matching operations during compression as shown in Fig.3. For a significant acceleration of compression, string-matching operations have to be optimized. Therefore, we focused on parallelizing the LZSS implementation by Michael Dipperstein [1], which provides clear separation of methods.

2) Using PFAC library
For LZSS to work, the longest match of the lookahead buffer must be found in the search buffer. But, since PFAC is a perfect pattern matching algorithm, patterns of minimum to maximum length must be passed in order to find the longest match rather than the perfect match. For this purpose the lookahead was compiled into all possible patterns as shown in Fig.4. The patterns start with 3 characters since the minimum reasonable match for compression is 3 characters. PFAC will then return an array with the match locations and corresponding pattern. Since we compile the patterns in increasing order of size, ergo match length, the largest pattern number will correspond to the longest match. We then return the longest match length and position on the search buffer back to the LZSS encoding algorithm.

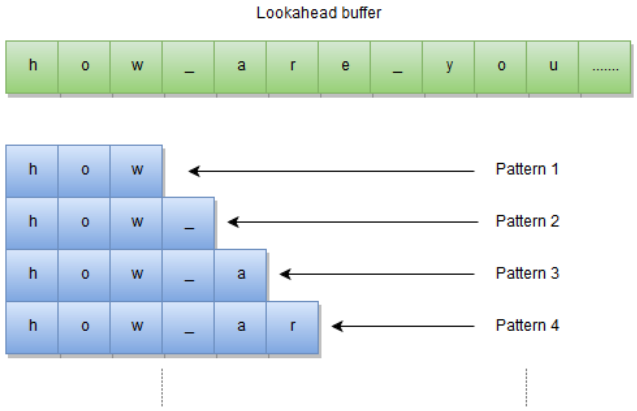

Fig. 4. Pattern compilation

3) Using CUDA threads
In this implementation, we created threads to search for a match parallely in the search buffer as shown in Fig.5.

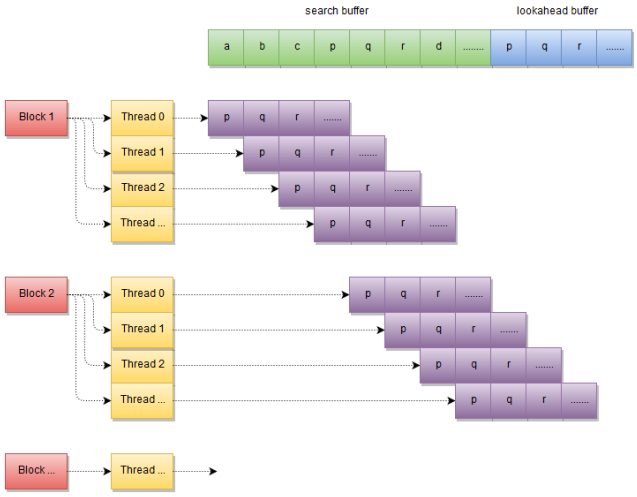

Fig. 5. Parallelized string-matching with Cuda threads

Since the search buffer is 4096 characters in length and the lookahead buffer is 18 characters in length, matching 18 characters in each thread would yeild a fast result. To attain this, we created 4096-18 = 4078 (approximated to 4080) threads in 4 blocks comprising of 1020 threads each. These threads match the entire lookahead buffer with 18 characters of the search buffer, each thread starting with an offset (blockIdx.x*blockDim.x+threadIdx.x) which is the starting point of the search on the search buffer. Once the result is returned which is an array of match length corresponding to the match position, the maximum match length is found and returned.

V. PERFORMANCE EVALUATION

A. Setup

- Testbed configurations

To evaluate how well the CUDA implementations work, we used a Tesla K40 card with CUDA version 7.5, installed on a machine with 6th gen Intel i7(6700k) CPU running at 4.0GHz and 32 GB DDR3 RAM at 2133 MHz. The CPU LZSS implementations were also tested on the same testbed.

- Data sets

The executions were benchmarked with 3 data files. The first file is a small data file of 581KB. The second is a larger file with a size of 3.2MB. The third file is much larger one with a file size of 100MB. All files are textual.

B. Results and Analysis

- Compression speed

To evaluate the performance of each technique regarding the compression speed all implementations along with gzip and bzip2 programs were executed on the benchmark files. Given below are the results obtained for time taken to compress the files;

TABLE I
EXECUTION TIME OF IMPLEMENTATIONS

Implementation	Time taken(s)		
	500kb file	3.2mb file	100mb file
Serial LZSS	0.32	0.71	20.47
Parallel LZSS (CPU)	0.04	0.15	4.06
Parallel LZSS (GPU-CUDA 512)	2.88	3.90	37.10
PFAC	128.21	770.94	inf
Serial LZSS with KMP	0.88	5.09	200.37
Parallel-matching LZSS (CUDA)	86.11	436.70	inf
gzip	0.07	0.26	5.40
bzip2	0.62	0.38	8.07

Even though we achieved a speedup of 5x for the parallelized LZSS CPU implementation compared to the serial LZSS algorithm, none of the CUDA implementations showed any favorable speedup.

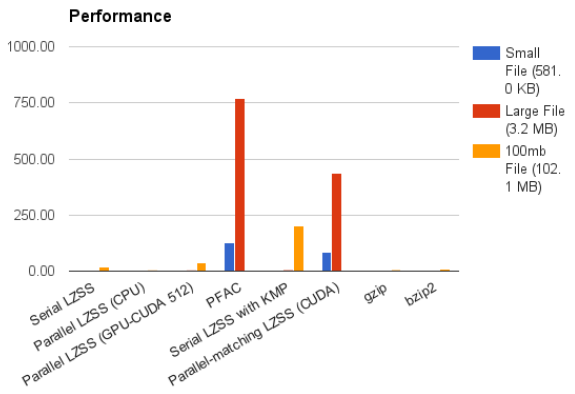

Fig. 6. Performance comparison

VI. CONCLUSIONS

Though the majority of the time is spent on string matching in the case of LZSS based compression, optimizing that using CUDA is not a viable option due to several reasons. Though string matching can be attained faster with many threads working together to find the longest match, the overhead of initializing the device, creating threads, copying data to Device and back to Host and freeing data is too high in comparison to the length of string to be matched. This is because of the sliding window principle used in LZSS. Meaning only a small part (4096 characters - known as the search buffer) of the whole file is matched with the 18 characters of the lookahead buffer. Running the entire algorithm on CUDA would also not profit in the situation seeing that a single thread on CUDA has much less resources when compared to a CPU yet the rest of the algorithm needs to run serially. In conclusion, the best method to speed up LZSS compression for massively parallel environments would be partitioning the dataset as the compression speed increases with minimal loss of compression ratio.

REFERENCES

[1] M. Dipperstein, ”LZSS (LZ77) Discussion and Implementation”, Michael.dipperstein.com, 2015. [Online]. Available: http://michael.dipperstein.com/lzss/. [Accessed: 20- Aug- 2016].

[2] ”Lempel-Ziv-Storer-Szymanski”, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Lempel-Ziv-Storer-Szymanski. [Accessed: 20- Aug- 2016].

[3] C. Liu, L. Chien, C. Lin and S. Chang, ”PFAC Library GPU-based string matching algorithm”, 2011. [Online]. Available: https://github.com/pfac-lib/pfac/blob/master/PFAC_userGuide_r1.1.pdf. [Accessed: 20- Aug- 2016].

[4] C. Lin, C. Liu, L. Chien and S. Chang, ”Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs”, IEEE Transactions on Computers, vol. 62, no. 10, pp. 1906-1916, 2013.

[5] H. Okumura, ”Data Compression Algorithms of LARC and LHarc”, Oku.edu.mie-u.ac.jp, 1989. [Online]. Available: https://oku.edu.mie-u.ac.jp/ okumura/compression/ar002/compr2.txt. [Accessed: 20- Aug- 2016].

[6] A. Ozsoy and M. Swany, ”CULZSS: LZSS Lossless Data Compression on CUDA”, 2011 IEEE International Conference on Cluster Computing, 2011.

[7] A. Ozsoy, M. Swany and A. Chauhan, ”Optimizing LZSS compression on GPGPUs”, Future Generation Computer Systems, vol. 30, pp. 170-178, 2014.

[8] Y. Zu and B. Hua, ”GLZSS: LZSS Lossless Data Compression Can Be Faster.”, 2014. [Online]. Available: http://staff.ustc.edu.cn/ bhua/publications/GLZSS-2014.pdf. [Accessed: 20- Aug- 2016].

[9] R. Cloud, M. Curry, H. Ward, A. Skjellum and P. Bangalore, ”Accelerating Lossless Data Compression with GPUs”, Arxiv.org, 2011. [Online]. Available: http://arxiv.org/abs/1107.1525. [Accessed: 21- Aug- 2016].

[10] ”Breakthrough in CUDA data compression”, Wave-access.com, 2011. [Online]. Available: http://www.wave-access.com/public_en/blog/2011/april/22/breakthrough-in-cuda-data-compression.aspx#.Vv4qqvknJwj. [Accessed: 20- Aug- 2016].

[11] R. Dhore, D. Sapkal, P. Jadhav, S. Borkar and A. Gogawale, ”Framework for Video Image Compression Using CUDA and NVIDIAs GPU”, International Journal of Emerging Engineering R esearch and Technology, vol. 3, no. 3, 2015.

[12] V. Tulsyan, A. Sarode, A. Vasishta, T. Notani and A. Sharma, ”DKLZSS - A Dynamic KMP String Matching Method for Parallel LZSS Compression on GPGPUs”, International Journal of Computer Applications, 2016.