Accelerating Lossless Compression Algorithms

Hari Om, PES University
Abhishek Banerji, PES University
Gaurav CG, PES University
Dr. Rahul Nagpal, PES University

Data plays a vital role in today's world. There are millions of data which is generated and the need arises to effectively store these data or to archive these data. Hence compression of data plays a vital role. These compressions are done by a class of compression algorithms which need to be fast in compressing as well as in decompressing data. Hence in this report, we propose to accelerate these algorithms by running these algorithms in parallel with the help of GPUs. This is an extension of the work done by other open-source developers. We developed an OpenCL implementation of some of these algorithms which harness the capabilities of both CPU and GPU which are heterogeneous systems. Our work can be found at https://github.com/BuzzHari/hp-project.

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: heterogeneous parallelism, Lossless compression algorithm, LZSS, burrows-wheeler transform, run length encoding

ACM Reference Format:
Hari Om, Abhishek Banerji, Gaurav C G, Dr.Rahul Nagpal, 2020. Accelerating Lossless Compression Algorithms. *1st CAPS-PSRL Symposium of Heterogeneous Parallelism* 1, 1, Article 3 (May 2020), 18 pages.
DOI: 0000001.0000001

1. Introduction

The class of algorithms which perform compression and decompression without any loss of data are known as lossless compression algorithms. They include Run-length encoding, Burrows-Wheeler transform, Lempel-Ziv, Huffman coding, prediction by partial matching and so on. In this paper, we implemented three of the above-mentioned algorithms. These algorithms are serial algorithms which have their limitations which depend on the performance of a single core and fail to take advantage of modern GPU and multi-core architectures. These algorithms can be used to compress and decompress a lot of data much faster if they were to run in parallel making use of multi-core systems. The algorithms which have been parallelized have been mentioned in the below subsections.

1.1. Run-Length Encoding

Run-length encoding (RLE) is a form of lossless data compression in which runs of data (sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs. It replaces sequences of the same data values within a file by a count number and a single value. Suppose the following string of data (17 bytes) has to be compressed:
ABBBBBBBBBCDEEEEF

This work is supported by Centre for Advanced Parallel Systems - Parallel Systems Research Lab (CAPS-PSRL) PES University, 100 Feet Ring Road, BSK III Stage, Bangalore-560085
Author's addresses: ¡insert names and email ids¿, Department of Computer Science, PES University, 100 Feet Ring Road, BSK III Stage, Bangalore-560085

DOI: 0000001.0000001

Using RLE compression, the compressed file takes up 10 bytes and could look like this: A *8B C D *4E F

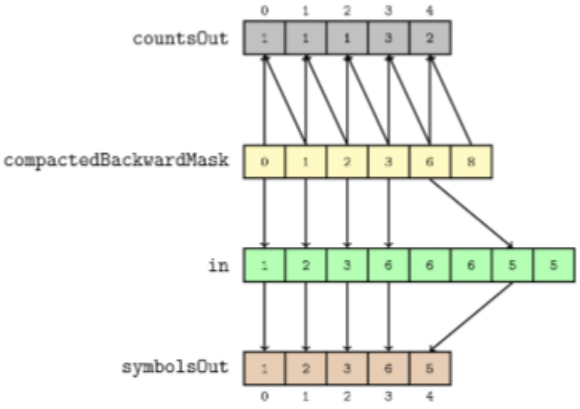

Fig. 1: Run-Length Encoding

1.2. Lempel-Ziv-Storer-Szymanski Algorithm

Lempel–Ziv–Storer–Szymanski (LZSS) is a lossless data compression algorithm, a derivative of LZ77, created in 1982 by James Storer and Thomas Szymanski. it is a dictionary coding technique which attempts to replace strings of symbols with a reference to a dictionary position of the same string. LZSS omits dictionary references if the length is less than the "break-even" point and it uses one-bit flags to show whether the next chunk of data is a literal (byte) or a reference to an offset/length pair. Figure 2 and 3 illustrates a simple example which shows the working of the algorithm.

```
  0: I am Sam
  9:
 10: Sam I am
 19:
 20: That Sam-I-am!
 35: That Sam-I-am!
 50: I do not like
 64: that Sam-I-am!
 79:
 80: Do you like green eggs and ham?
112:
113: I do not like them, Sam-I-am.
143: I do not like green eggs and ham.
```

Fig. 2: LZSS-Original Data

```
 0: I am Sam
 9:
10: (5,3) (0,4)
16:
17: That(4,4)-I-am!(19,16)I do not like
45: t(21,14)
49: Do you(58,5) green eggs and ham?
78: (49,14) them,(24,9).(112,15)(92,18).
```

Fig. 3: LZSS-Compressed Data

1.3. Burrows-Wheeler Transform

Burrows-Wheeler transform(BWT) is a method to convert data into a form with more runs of similar characters. It rearranges the characters strings following certain algorithmic steps as shown in Fig 4, and Fig 5, which leads to the result as shown in Fig 6. This format of data is highly compressible by algorithms like Run Length encoders. The BWT algorithm with the transformed data also produces some additional data which is used to transform it back to the original data. Invented by Michael Burrows and David Wheeler in 1994, this algorithm is one of the core techniques used in the compression tool bzip2.

Example:
"this is a test." yields the following rotations:

```
S0  = "this is a test."
S1  = "his is a test.t"
S2  = "is is a test.th"
S3  = "s is a test.thi"
S4  = " is a test.this"
S5  = "is a test.this "
S6  = "s a test.this i"
S7  = " a test.this is"
S8  = "a test.this is "
S9  = " test.this is a"
S10 = "test.this is a "
S11 = "est.this is a t"
S12 = "st.this is a te"
S13 = "t.this is a tes"
S14 = ".this is a test"
```

Fig. 4: BWT: Text Rotations

Example:
"this is a test." yields the following sorted rotations:

```
S7  = " a test.this is"
S4  = " is a test.this"
S9  = " test.this is a"
S14 = ".this is a test"
S8  = "a test.this is "
S11 = "est.this is a t"
S1  = "his is a test.t"
S5  = "is a test.this "
S2  = "is is a test.th"
S6  = "s a test.this i"
S3  = "s is a test.thi"
S12 = "st.this is a te"
S13 = "t.this is a tes"
S10 = "test.this is a "
S0  = "this is a test."
```

Fig. 5: BWT: Sorted Rotations

Example:
"this is a test." yields the following output:

```
L = "ssat tt hiies .", I = 14
```

Fig. 6: BWT: Transformed String

2. Initial Benchmark

The benchmark is done by comparing the serial versions of the proposed algorithms with their CUDA parallel counterparts. This is done so that we can see how much speedup can be achieved and set our self a target benchmark which we aim to achieve when implemented using OpenCL.

2.1. Run-Length Encoding

For RLE, we have tested with four different files varying in size and observe the speedup which was gained. Here we can see that there is a huge speedup with identical compression ratios . The comparison of Serial RLE and CUDA RLE is tabulated and shown below.

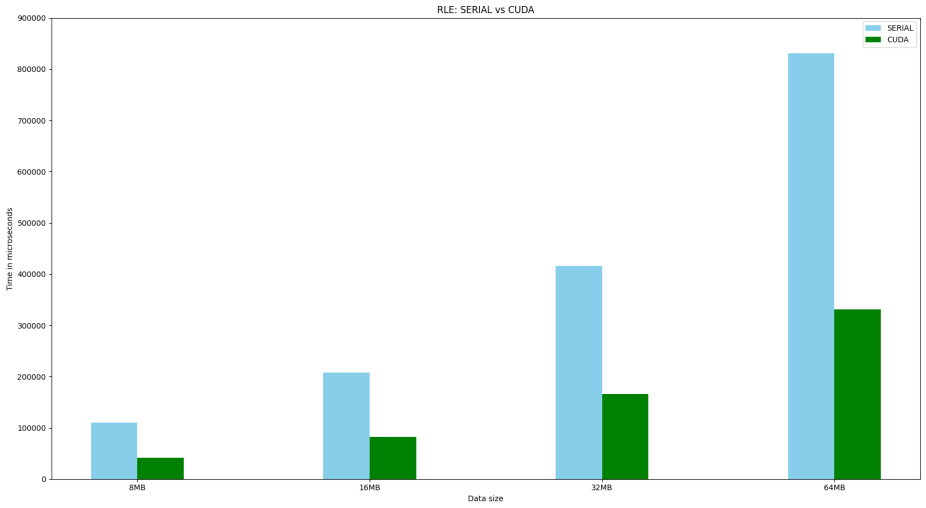

Fig. 7: Benchmark: RLE-COMPRESSION

File Size	Time in us (Serial)	Time in us (CUDA)	Compression Ratio
8MB	110.3	41.3	3.3
16MB	207.6	82.6	3.3
32MB	415.3	165.7	3.3
64MB	830.8	331.4	3.3

Fig. 8: Benchmark: RLE-COMPRESSION

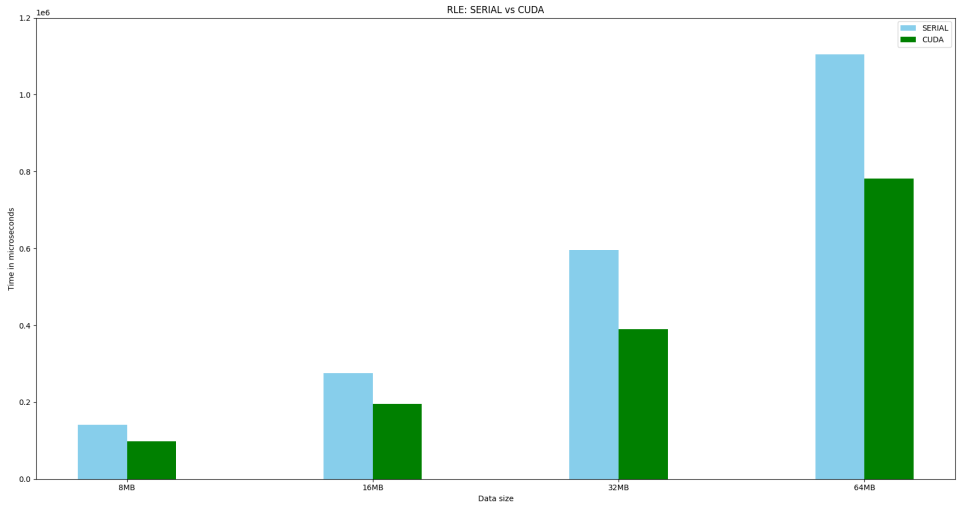

Fig. 9: Benchmark: RLE-DECOMPRESSION

File Size	Time in us (Serial)	Time in us (CUDA)
8MB	141.5	97.6
16MB	276.1	195.1
32MB	595.1	390.4
64MB	1105.2	781.2

Fig. 10: Benchmark: RLE-DECOMPRESSION

2.2. Lempel-Ziv-Storer-Szymanski Algorithm

For LZSS, we have tested with three different files varying in size and observe the speedup which was gained. Here we can see that there is a huge speedup but the compression ratio calculated for the parallel code was not up to the mark.

File Size in mb	Time in s (Serial)	Time in s (Parallel)	Compression Ratio (serial)	Compression Ratio (parallel)
6.2	6.77	0.09	2	1.06
100	112.2	0.34	1.92	1.06
954	855.35	2.18	2.19	1.20

Fig. 11: Benchmark Tabulated: LZSS-Compression

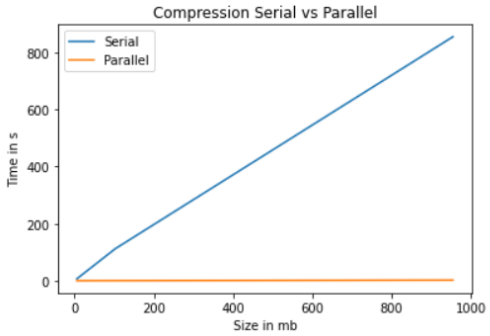

Fig. 12: Linear representation

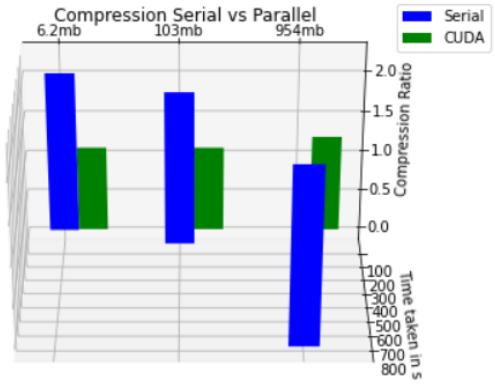

Fig. 13: 3d Representation including compression ratio

We can see the speedup between the serial decompression and parallel CUDA decompression in the below fig.

File Size in mb	Time in s (serial)	Time in s (parallel)
6.2	0.07	0.07
100	1.49	0.25
954	9.37	1.71

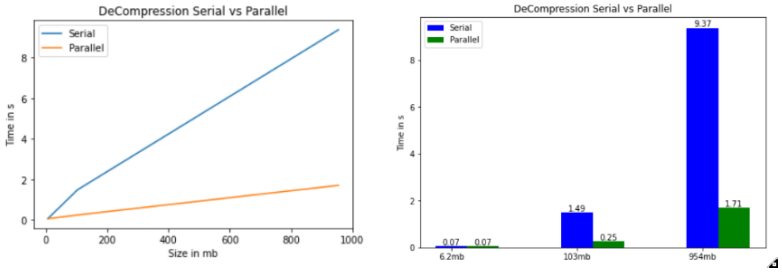

Fig. 14: Benchmark: LZSS-Decompression

2.3. Burrows-Wheeler Transform

The Cuda-BWT we bench-marked didn't perform well compared to the serial implementation. This maybe due to hardware reasons, as the Cuda code base was quite old and was last tested using Cuda-5 toolkit. Also different test data may produce different results, as seen in Fig 16.

File Size in MB	Serial (Time in secs)	Cuda (Time in secs)
2.4 (world192)	0.146	0.202
3.1 (dictionary)	0.180	0.057
3.9 (bible)	0.275	0.269
4.5 (E.coli)	0.454	1.098
6.3 (big)	0.393	4.187

Fig. 15: Benchmark: Burrows-Wheeler Transform

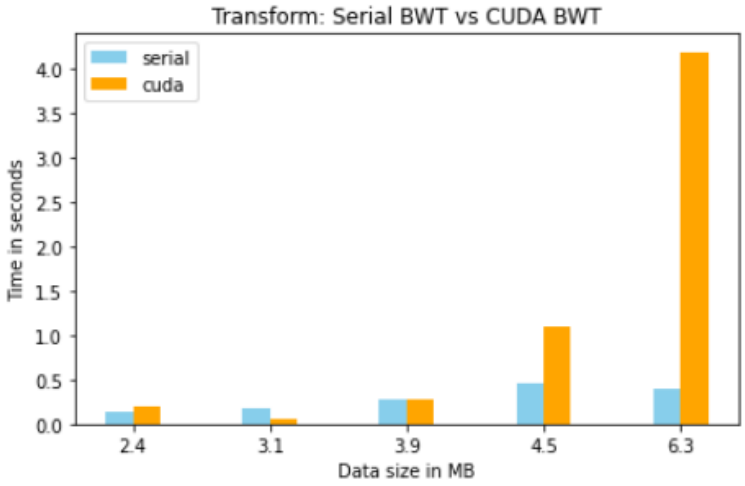

Fig. 16: Benchmark: Burrows-Wheeler Transform

3. Concurrency In Proposed Algorithms

In this section we discuss the parts of the algorithms that can be made parallel in order to achieve faster compression/decompression.

3.1. Run-Length Encoding

In RLE algorithm there are four stages of the compression.
1. generating Backward Mask.
2. generating Scanned Backward Mask.
3. generating Compacted Mask.
4. generating the symbolsOut and countsOut.

Stages 1,3 and 4 can be made parallel.

3.2. Lempel-Ziv-Storer-Szymanski Algorithm

In LZSS algorithm, we can see that the compression of characters is not dependent if the two starting characters of the sub-strings are not in the range of the same search buffer length. Because of these characteristics, we can divide the input data into several blocks, and compress them in parallel.
We can also observe that there is an independent computation in every single matching of the characters inside each chunk. The search for a sub-string match starts with reading from the unencoded buffer and checks one by one with the window items. If there is a match, it checks the consequent items from each buffer to find a sub-string match. This process traces the window and it returns the longest match with the encoding information, length of sub-string and match offset. Item matching for each character does not depend on each other. Therefore, the matching computation runs in parallel for each character in the unencoded look-ahead buffer.

3.3. Burrows-Wheeler Transform

(1) The data can be divided into blocks, and could be transformed in parallel.
(2) The rotations of the block, could be done using suffix arrays.Constructing parallel suffix arrays may further increase speed up.
(3) Sorting is a major part of the algorithm. Sorting on the GPU or multi-threaded CPU sort would contribute further to the speed up.

4. OpenCL Implementation

4.1. Run-Length Encoding

In RLE algorithm there are four stages of the compression.

— generating Backward Mask:
In this stage we take the input array and create a mask array which marks the indexes where a new run starts with 1 and subsequent indexes with 0's. This give us an array which tells us where a run starts (when we encounter a '1' we know a new run start at that index).
— generating Scanned Backward Mask:
In this stage we take the mask array and perform prefix sum on it. This will give us an array which will tell how long each run is.
— generating Compacted Mask:
At this point we know all the locations of the output pairs (encoded through Scanned Backward Mask) and we know which symbols are being repeated (encoded through Backward Mask). We just need to find out how long each run is. We can obtain an array of indexes of the beginnings of every run. Then we could calculate the run-lengths by subtracting the current element and the previous element from each other.
— generating the symbolsOut and countsOut:
In this stage we get the run-lengths of all the runs and store the count in countsOut and the respective symbol in symbolsOut.

4.2. Lempel-Ziv-Storer-Szymanski

The LZSS depends on the buffer size/sliding window used. Hence we can run characters which appear after the buffer size in another thread with no dependencies. To achieve this, we use an array of structures which contains the string and its corresponding length.
Every thread will access its string and length using their global_id.
The algorithm runs on these threads and upon completion; it concatenates the output array of structures using the lengths of each array in the structure. We append this length before the string which is also first read from the decompress function and reads the 'length' number of characters and so on.
A kernel of 256 work items per workgroup is used and the number of workgroups depends upon the number of blocks which is the quotient of total file size and buffer size.
Each thread takes 100KB of string data which was found experimentally testing with various file sizes. This paves way for the creation of more threads and each thread gets access to as little data as possible. Each thread is successful in compressing the given 100KB into half or with compression ratio around 2.0.
The serial implementation used a bit to indicate encoded or unencoded data and used 'bitfilestream'. But in our OpenCL implementation, we used a 'flags' character variable which is attached to the output string to indicate whether the next character is encoded or unencoded.

4.3. Burrows-Wheeler Transform

The data which is read from a file, is divided into blocks of certain size. These blocks are basically the work-items which the kernel will act on. The kernel is designed to transform these blocks in parallel. Each block is taken and Burrows-Wheeler Transform is applied on it. The Main Ideas of the algorithm are:

(1) Divide text file into blocks.

(2) Send blocks to GPU.
(3) Apply BWT on each block.
(4) Send the transformed blocks back to Host device.
(5) Copy the blocks in a file.

The text file was divide into blocks of 4KB. This was found to be the optimal size by trial and error. These blocks all at once are sent to the GPU, where it is transformed and Radix sort is used to sort all rotations by their first two characters. This pre-sorted data is then sent to the Host device, where quick-sort is run over this pre-sorted data. This happens serially on the CPU, and it has a scope for improvement, such as by using a multi-threaded sort. Finally this sorted data is sent back to the GPU, to extract the last characters from the blocks and to create the transformation. This transformation is finally sent back and written to a file.

5. OpenCL Benchmark

5.1. Run Length Encoding

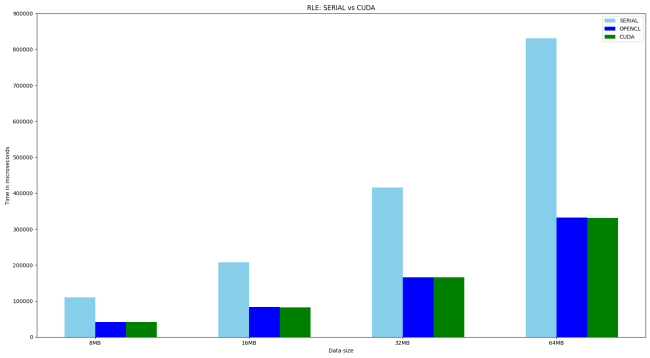

Fig. 17: Benchmark: RLE-COMPRESSION

File Size	Time in us (Serial)	Time in us (OpenCL)	Time in us (CUDA)	Compression Ratio
8MB	110.3	41.9	41.3	3.3
16MB	207.6	83.3	82.6	3.3
32MB	415.3	166.4	165.7	3.3
64MB	830.8	331.9	331.4	3.3

Fig. 18: Benchmark: RLE-COMPRESSION

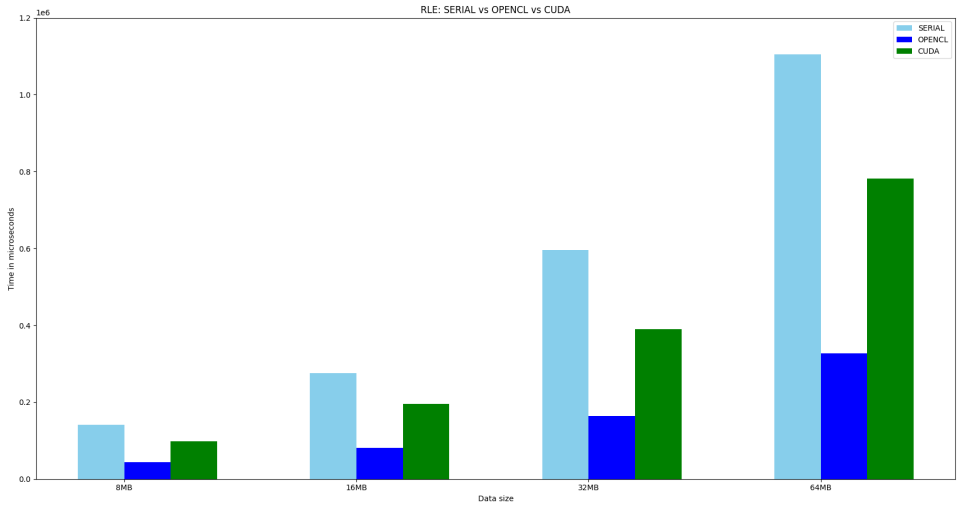

Fig. 19: Benchmark: RLE-DECOMPRESSION

File Size	Time in us (Serial)	Time in us (OpenCL)	Time in us (CUDA)
8MB	141.5	44.4	97.6
16MB	276.1	81.8	195.1
32MB	595.1	163.4	390.4
64MB	1105.2	327.5	781.2

Fig. 20: Benchmark: RLE-DECOMPRESSION

5.2. Lempel-Ziv-Storer-Szymanski

From the below graph we can see that the speed up achieved by the OpenCL version is faster than the serial code but is behind the CUDA version. The reason for this difference is due to the buffersize which is used in CUDA version which provides incredible speedup but with the trade off of compression. We felt that the entire purpose is to compress faster and not just to run faster.

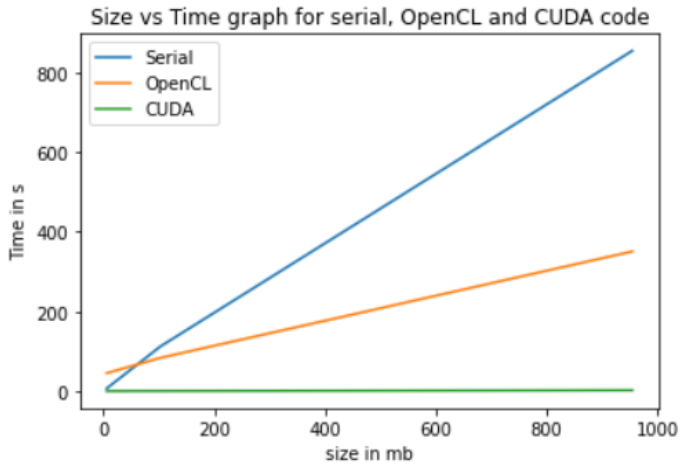

Fig. 21: Result: LZSS

The above mentioned point can be further strengthened by the below 3d graph which is plotted against size, compression ratio and time.
We can see that the compression ratio of our algorithm was on par with the serial implementation while also being faster than it.

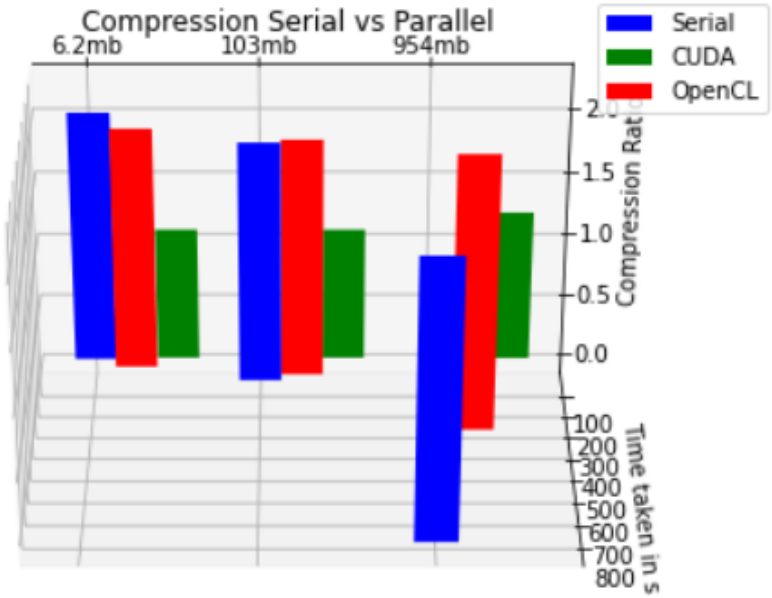

Fig. 22: Result: LZSS

These graphs are tabulated in the table below which summarizes the performance of our OpenCL version of LZSS.

File Size in mb	Time in s (Serial)	Time in s (OpenCL)	Time in s (CUDA)	Compression Ratio (serial)	Compression Ratio (OpenCL)	Compression Ratio (CUDA)
6.2	6.77	45.1	0.09	2	1.93	1.06
100	112.2	83.5	0.34	1.92	1.90	1.06
954	855.35	351.1	2.18	2.19	2.18	1.20

Fig. 23: Tabulated Result: LZSS

The results for decompression are also provided below.

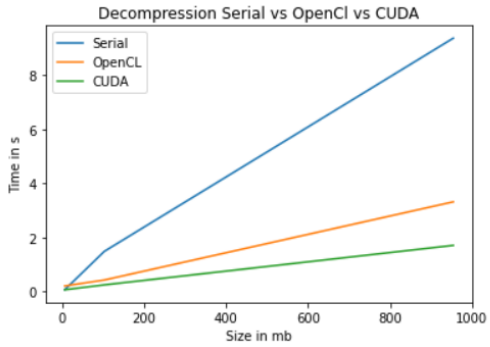

Fig. 24: Line Graph

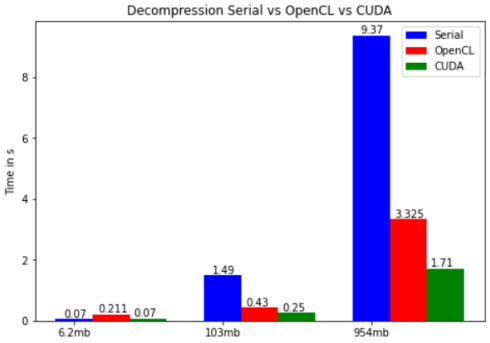

Fig. 25: Bar Graph

The tabulated result is provided as reference in the below figure.

File Size in mb	Time in s (serial)	Time in s (CUDA)	Time in s (OpenCL)
6.2	0.07	0.07	0.211
100	1.49	0.25	0.430
954	9.37	1.71	3.325

Fig. 26: Tabulated Result: LZSS

5.3. Burrows-Wheeler Transform

The OpenCL-BWT still requires enhancements as it performs slower than Serial-BWT. But in compared to CUDA-BWT, it beats it as the file size increases. The results are tabulated and shown below. The speed up of reverse transformation of Serial-BWT vs. OpenCL-BWT is also tabulated and shown below.

File Size in MB	Serial (Time in secs)	Cuda (Time in secs)	OpenCL (Time in secs)
2.4 (world192)	0.146	0.202	0.300
3.1 (dictionary)	0.180	0.057	0.340
3.9 (bible)	0.275	0.269	0.458
4.5 (E.coli)	0.454	1.098	0.642
6.3 (big)	0.393	4.187	0.600

Fig. 27: Transform: Serial-BWT vs CUDA-BWT vs OpenCL-BWT

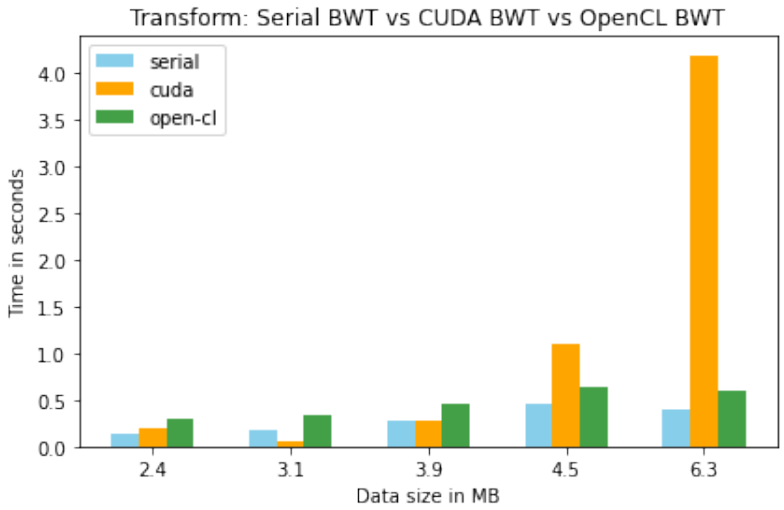

Fig. 28: Transform: Serial-BWT vs CUDA-BWT vs OpenCL-BWT

File Size in MB	Serial (Time in secs)	OpenCL (Time in secs)
2.4 (world192)	0.011	0.093
3.1 (dictionary)	0.015	0.095
3.9 (bible)	0.017	0.103
4.5 (E.coli)	0.019	0.105
6.3 (big)	0.025	0.119

Fig. 29: Reverse: Serial-BWT vs OpenCL-BWT

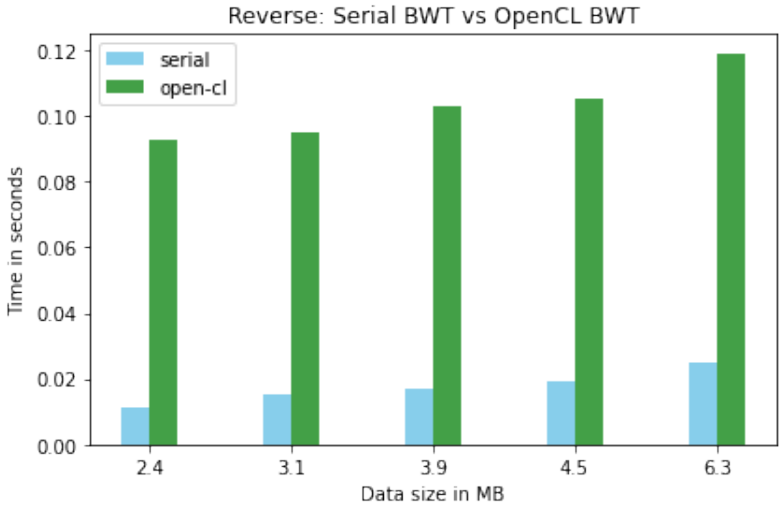

Fig. 30: Reverse: Serial-BWT vs OpenCL-BWT

6. Conclusions

In conclusion, we see decent speed-up in case of OpenCL-RLE and OpenCL-LZSS. Even though CUDA implementation of these algorithms are faster, but they only support Nvida GPUs, which is not the case of OpenCL, as it supports a wide range of GPUs. OpenCL-BWT is slower than serial, but it performs much better than the CUDA-BWT.

7. Future Work

OpenCL-RLE can be further optimized, if we break down the data and pass it to the GPU, as in it's current implementation, it fails to compress big files. OpenCL-LZSS can also be further optimized, by finding out the optimal block size. A Producer-Consumer mechanism to send blocks of data to the GPU, can further speed the algorithm up. OpenCL-BWT will see more speed-up when the sorting phase is done in parallel.

8. Acknowledgements

We express our gratitude to our guide Dr. Rahul Nagpal for providing us the opportunity to work on an exciting niche and progress in development experience through this project. We also thank immensely our mentor Rahul S for his continued support and guidance throughout the course of this project.

REFERENCES

Eric Arnebäck. 2020. Implementing Run-length encoding in CUDA. (2020). https://erkaman.github.io/posts/cuda_rle.html

Ana Balevic. 2020. Fine-Grain Parallelization of Entropy Coding on GPGPUs. (05 2020).

M. Burrows and D.J. Wheeler. 1994. A Block-sorting LosslessData Compression Algorithm. (1994). https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

Michael Dipperstein. 2014a. Burrows-Wheeler Transform Discussion and Implementation. (2014). http://michael.dipperstein.com/bwt/

Michael Dipperstein. 2014b. LZSS (LZ77) Discussion and Implementation. (2014). http://michael.dipperstein.com/lzss/index.html

Aaftab Munshi. 2020. TheOpenCL Specification. (2020). https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

A. Ozsoy and M. Swany. 2011. CULZSS: LZSS Lossless Data Compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*. 403–411.

A. Ozsoy, M. Swany, and A. Chauhan. 2012. Pipelined Parallel LZSS for Streaming Data Compression on GPGPUs. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 37–44.