Parallelization of LZSS Algorithm

E4750_2021Fall_LZSS_report

Zhang, Qixiao(qz2487)   Hou, Baizhou (bh2803)
Columbia University

Abstract: *The Lempel-Ziv-Storer-Szymanski compression algorithm (LZSS) is a derivation of the LZ77 algorithm, and is widely used in archivers such as PKZIP, RAR, etc. [1]. LZSS achieves a generally good compression ratio on a wide variety of files and data types. In this work, we follow an approach similar to a parallel implementation described by Ozsoy[2] and achieve significant speedup over a naive serial implementation. Our GPU-CPU hybrid approach falls short when compared to a highly optimized CPU implementation. We outline challenges and possible future work for further improving this implementation.*

Key Words: *GPU LZSS Compression Encoding*

1. Overview

1.1 Introduction

The origin of data compression is based on information theory. Shannon, the father of information theory, first clarified the relationship between probability and message redundancy in mathematical terms. In his 1948 paper "A Mathematical Theory of Communication," Shannon pointed out that redundancy exists in any message and that the size of redundancy is related to the probability of occurrence of each symbol in the message[3]. Drawing on the concept of thermodynamics, Shannon called the average amount of information in a message after eliminating redundancy "information entropy" and gave a mathematical expression to calculate information entropy. This work was later regarded as the pioneering work of information theory, and information entropy also laid the theoretical foundation of the following works on data compression algorithms.

The primary purpose of data compression is to eliminate redundancy in information. The rapid development of technology and the demands of users lead to an expectation of fast streaming, while limited bandwidths make serving content difficult. Compression allows large multimedia content to be practical. In fact, most media such as video, music, and more are stored and transmitted–and sometimes even directly used–in both lossless and lossy compressed forms. Indeed, efficient use of network bandwidth and server storage management has become an increasingly important driver in the computing community [2].

We believe there is some merit in attempting to parallelize and speed up LZSS computation. By breaking the input data chunk to be compressed in parallel, there should be ample opportunity to parallelize the computation.

1.2 Prior Work

There are several prior works related to the parallelization of the LZSS workload. A particular challenge for these implementations of this specific lossless compression algorithm is to maintain a low compression ratio while keeping computation intensity reasonably low. Our primary prior work reference is the work of Ozsoy and Swany on LZSS lossless data compression on CUDA [3]. Ozsoy describes two main approaches to parallelizing the LZSS algorithm, both of which start by separating the input into chunks and loaded into the shared memory of each thread-block. Then, in "Version 1", Ozsoy describes further separating these chunks to be compressed in parallel by the threads within the chunk by noticing that the compression of characters is independent. In "Version 2", threads within the thread-block iterates through its individual search window to match patterns starting at a specific index of the chunk to be encoded by the block. This approach reaches maximum throughput more readily and generally lends itself better to the memory architecture of NVidia GPUs, so it is the approach we base this work's implementation on.

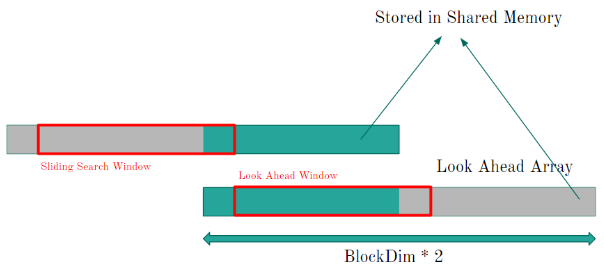

Fig. 1: Thread-Level Example of LZSS Parallel Search

2. Description

2.1. Goal and Objectives

The goal is to construct a GPU-accelerated LZSS compression implementation and compare the performance to both a naive serial implementation and a highly-optimized state-of-the-art CPU implementation. In this case, we use a basic python linear search implementation to act as the naive CPU implementation and use a C-implemented tree-optimized CPU implementation as the state-of-the-art comparison point.

2.2. Problem Formulation and Design

For the implementation, we intuit observe that separating the input into chunks and encoding them in parallel has a negative impact on the compression ratio.

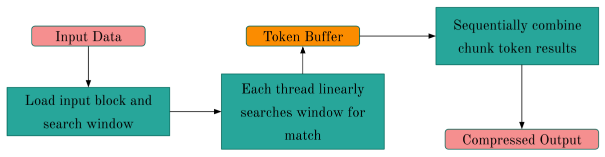
Input Data
Token Buffer
Sequentially combine chunk token results
Load input block and search window
Each thread linearly searches window for match
Compressed Output

Fig. 2: Algorithm Flow

For our implementation, we split the full compression algorithm into 2 separate steps which we call encoding and compression. In the encoding step, the best match tokens generated from a linear search through the search window are stored in a buffer. Then in the compression step, we sequentially scan the token buffer to produce a final compressed output. This results in a GPU-CPU hybrid algorithm that generally has low code complexity and minimizes the negative impact on the compression ratio resulting from parallelization at the borders of the chunks.

2.3. System and Software Design

For the encoding part, we statically allocate a token buffer equal to twice the size of the input. This allows us to store byte-level offset length pairs. While this limits the maximum search window size, it allows us to use a simple byte array as the token buffer. Moreover, it the smaller window size reduces the overall memory requirement and runtime of the algorithm at the cost of compression ratio.

Once the GPU completes the parallel encoding part and populates the token buffer, the data is sequentially stitched together by CPU. By linearly iterating through the token buffer, it is simple to find the best match token to use at each index and discard redundant or non-compressing tokens. This final output from this process is the compressed output with literals where tokens are non-compressing and tokens where a compressing pattern match exists.

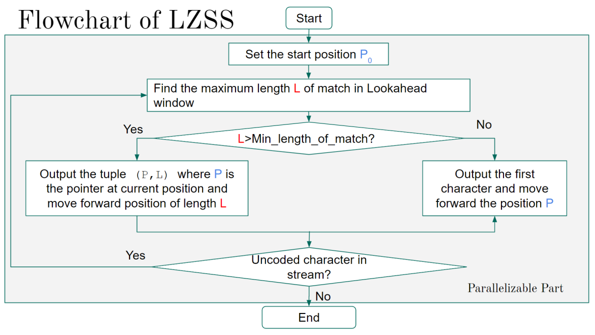
Flowchart of LZSS
Start
Set the start position P0
Find the maximum length L of match in Lookahead window
Yes
L>Min_length_of_match?
No
Output the tuple (P,L) where P is the pointer at current position and move forward position of length L
Output the first character and move forward the position P
Yes
Uncoded character in stream?
No
Parallelizable Part
End

Fig. 3: Flowchart of LZSS algorithm

2.3. Pseudo Code

Thread-level Kernel:

```
//Load chunk to shared memory
...

//Encoding
for i in range(WINDOW_SIZE):
  ii = i+idx;
  ij = i-WINDOW_SIZE+i;
  match = (d[ii] == d[ij])
  if(match):
    matchLen += 1;
  else:
    matchLen = 0;

  if(matchLen > bestMatch):
    bestLen = matchLen;
    bestOffset = offset;

if(bestLen > MIN_MATCH):
  token_buf[2*idx] = bestOffset;
  token_buf[2*idx+1] = bestLen;
else:
  token_buf[2*idx] = 0;
  token_buf[2*idx+1] = 0;
```

Sequential Token Buffer Scan:

```
for i in range(INPUT_LEN):
  if(token_buf[2*i] > 0):
    output.append(token)
  else:
    output.append(input[i])
```

3. Results

3.1 Similar Structure Files

We tested two groups of files in our experiments. First, we tested the running time of our parallel implementation. Since the naive CPU serial LZSS algorithm is way too slow compared to our implementation, we tried a highly optimized CPU implementation. This CPU algorithm used a tree data structure in order to accelerate the matching function. This greatly reduce the search time for pattern matching to O(log(n)) complexity.

From our result in *Fig. 4*, we can see that the CPU results seem always around 3x faster compared to our GPU implementation. This shows that highly optimized CPU LZSS has its own edge over computing with small-size files. However, we explored the running time results more (*Table 1*) and we found that if the test file is big enough, the GPU running time will catch the CPU running time in the end. This result is concluded by the Grow Rate per MB for GPU since the growth rate for GPU is always smaller than CPU.

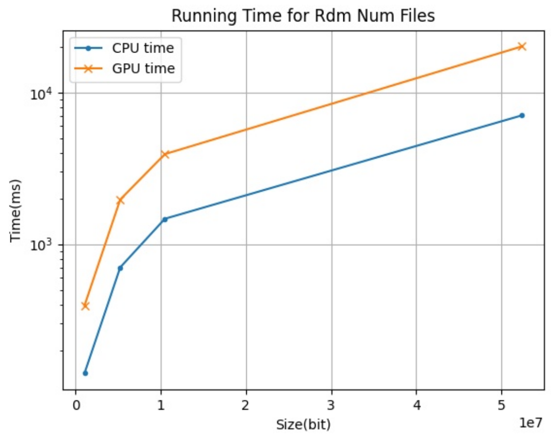

Fig. 4: Running Time for random number text files with different sizes (1 MB,5 MB,10 MB,50 MB)

For the compression ratio comparisons (*Fig. 5*), our result is not as good as the CPU version. We think this could be the reason for our small window size. We set our window size to 128 to limit our offset value. If the offset value is more than two bytes, the encoded message will be longer and the threshold of encoding will be bigger. We will have to sacrifice our efficiency for this. Specifically, the runtime memory requirement that needs to be allocated to the buffer would be large.

Besides the exact value of these two versions, we can also see our result is consistent in that it has the same compression ratio for files with the same structure.

Table 1:Running time growth rate per MB

File Size(MB)	**GPU Running Time**	**CPU Running Time**	**Growth Rate per MB for GPU**	**Growth Rate per MB for CPU**
1	411.6897	136.707	N/A	N/A
5	1970.061	683.837	0.9571	1.0004
10	3951.62	1373.635	1.0029	1.0043
50	19728.9	6992.727	0.9985	1.0215

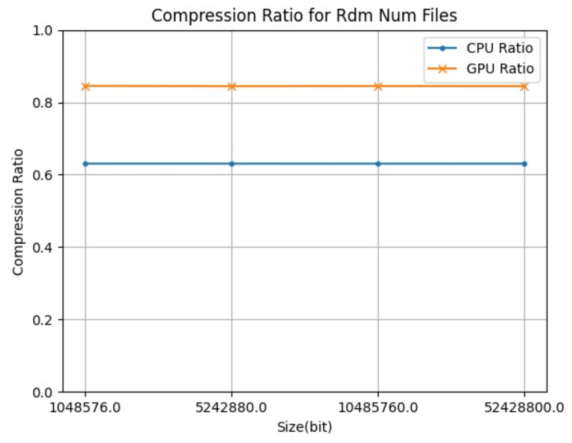

Fig. 5: Compression Ratio for random number text files with different sizes (1 MB,5 MB,10 MB,50 MB)

3.1 Different Structure Files

We tried another set of files called Canterbury Corpus dataset. The Canterbury Corpus serves as a standard against which academics can compare different lossless compression techniques. For several research compression approaches, this website offers test files and the results of compression tests. This dataset contains different kinds of files such as `XML`, `C`, `xargs.1`, `grammar.lsp`, `asyoulik.txt` and `SHA1SUM`.

We also added the naive CPU serial LZSS algorithm to make comparisons. From *Fig. 3*, we figured out the running time of the naive version is 10x slower than the highly optimized version and our parallel version.

For the compression ratio, we have some different findings. Both the CPU compression methods don’t have a good compression ratio for the SHA1SUM file. The tree-optimized CPU version even has a compression ratio bigger than 1. That is to say, it didn’t successfully compress the file. Our implementation has an edge over these two kinds of algorithms when processing SHA1SUM. Nevertheless, the compression ratios of other files are just relatively acceptable. It has better results than the naive version but is interior to the tree-optimized one.

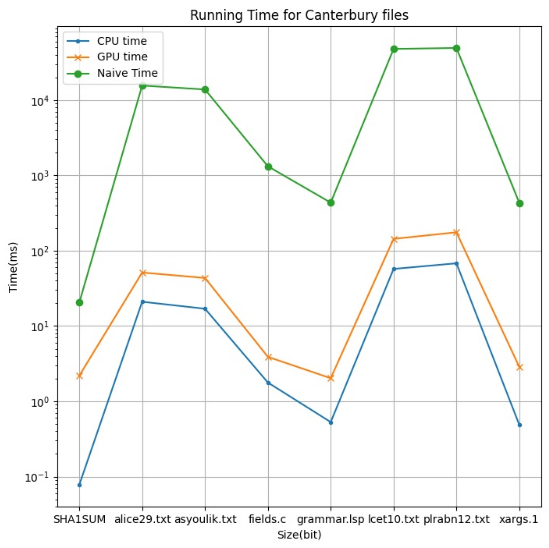

Fig. 6: Running Time for Canterbury files

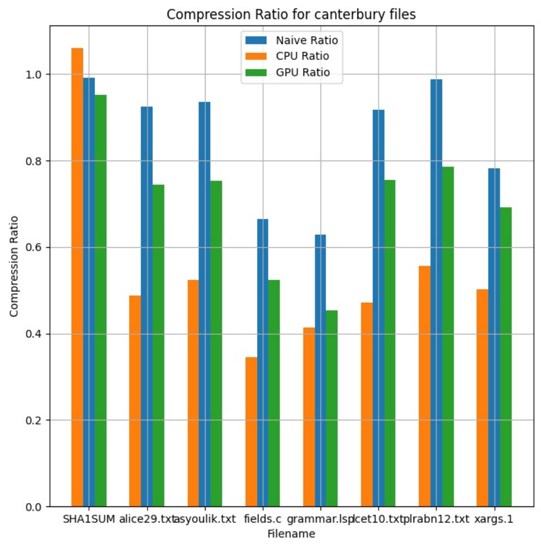

Fig. 7: Compression Ratio for Canterbury dataset

4. Discussion and Further Work

Generally, we observe that this implementation performs poorly compared to state-of-the-art CPU implementation, which has highly optimized pattern match searching algorithm and large search window. When making a more fair comparison to a naive sequential implementation of LZSS, we can see a respectable 10x speedup. This result can be seen in *Fig. 6.*

We believe the execution time can be reduced significantly by parallelizing the part where the token buffer is scanned to form the final output. This would come at a small cost to compression ratio due to optimal tokens potentially being discarded at the boundaries of the chunks. Alternatively, a more sophisticated data structure could be employed for the token buffer to allow for large search windows without excessively bloating the runtime memory for storing large offset-length pairs. Furthermore, a larger window size would allow for a better compression ratio.

5. Conclusion

All in all, our implementation is a success. We verified the feasibility of the paper from Ozsoy A. Swany in that we got a similar speedup by comparing the running time of the parallelized algorithm and serial CPU implementation. As we know, another standard to evaluate a compression algorithm is the compression ratio. Our result for compression ratio is acceptable, but could be improved with above outlined changes to enable larger search window sizes. The compression ratio of our CUDA program stays in the middle between the naive serial version and the highly-optimized version. Moreover, we outlined possible further work that could bring overall execution time close to a highly optimized state-of-the-art CPU implementation while achieving similarly comparable compression ratios. This implementation already outperforms a naive CPU implementation.

6. Acknowledgments

First, we would like to express our sincere gratitude to Professor Kostic for his dedication to teaching. We learned a lot from your class and thank you for your instruction this whole semester. In addition, we are grateful to our course assistant Manuel Jenkin Jerome who has devoted a large part of his personal time to this course.

Second, we would like to thank all the authors of the papers. They have done a great job before us.

7. References

[1] *Lempel–Ziv–Storer–Szymanski* (2022) *Wikipedia*. Wikimedia Foundation. Available at:

https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Storer%E2%80%93Szymanski#cite_note-3.
[2] Ozsoy, A., Swany, M. and Chauhan, A. (2012) "Pipelined parallel LZSS for streaming data compression on gpgpus," *2012 IEEE 18th International Conference on Parallel and Distributed Systems* [Preprint]. Available at: https://doi.org/10.1109/icpads.2012.16.
[3] Ozsoy, A. and Swany, M. (2011) "CULZSS: LZSS lossless data compression on Cuda," *2011 IEEE International Conference on Cluster Computing* [Preprint]. Available at: https://doi.org/10.1109/cluster.2011.52.
[4] Stein, C.M. *et al.* (2019) "Stream parallelism on the LZSS data compression application for multi-cores with gpus," *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* [Preprint]. Available at: https://doi.org/10.1109/empdp.2019.8671624.

8. Appendices

Presentation video link:

https://drive.google.com/file/d/1-MmPTQ5Xs2Pes64MC1lQxyCVhnJJX5Lo/view?usp=share_link

Individual Student Contributions (in %)

Task	**% Student 1**	**% Student 2**	
Overall	**50**	**50**	
Coding	**50**	**50**	
Slides	**50**	**50**	
Report	**50**	**50**	