MASSIVELY PARALLEL LZ77 COMPRESSION AND DECOMPRESSION ON THE GPU

by

Kayla Wesley, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2022

Committee Members:

Martin Burtscher, Chair

Vangelis Metsis

Wuxu Peng

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

To all who made this opportunity possible.

ACKNOWLEDGEMENTS

To Dr. Martin Burtscher, I would like to give my most heartfelt thank you for asking me to join your group and guiding me during my time at Texas State University. Your knowledge, kindness, and leadership has made this experience an amazing opportunity for growth as an engineer, and I am incredibly grateful for the privilege of being included in your group. To the Lossy Compression group members, thank you for the critical support during my research and providing very valuable feedback during the research process. To the ECL group, it has been a privilege to work alongside you and to hear about your research as it happens. I would also like to thank the committee members of my thesis for taking an interest in my work and for providing an excellent education in the respective classes I have taken from them. Lastly, I would like to thank my husband for his tireless support of me and my educational goals. You have all made this possible, and I cannot properly bring to words the gratitude I feel regarding your support.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation	Description
i.e.	That is
e.g.	For example
ET	Encoder Throughput
DT	Decoder Throughput
Ext.	Extended
Len	Length
Dis	Distance
Pfx	Prefix Array

ABSTRACT

Parallelizing data compression algorithms is difficult because algorithms like LZ77 have iterative dependencies that cannot easily be circumvented. Previous computation steps directly impact later steps in the algorithm, and parallelizing those dependent steps can prove difficult. My solution is to express both the encoding and decoding portions of LZ77 in terms of prefix sums, union-find operations, and other parallelizable computations. The results show that this methodology is effective in improving throughput. Compared to codes from the literature, my CUDA implementation is an order of magnitude faster but tends to have a lower compression ratio.

I. INTRODUCTION

LZ77 is a data compression and decompression algorithm created by Abraham Lempel and Jacob Ziv in 1977 [1]. Along with its variations, this powerful algorithm is one of the most widely used lossless compression algorithms. For example, LZ77 is used in LZSS, LZ4, DEFLATE/INFLATE and countless other compression algorithm variations [2][3][4]. One of the most notable types resulting from LZ algorithms today is PNG, and one of the most notable compressors is gzip [5].

LZ77 works by identifying common subsequences in input data and replacing later occurrences of those common subsequences by a reference to a prior occurrence. This is sometimes referred to as a dictionary-based reduction. LZ77 is relatively easy to implement serially. However, decoding and especially encoding are challenging to parallelize due to dependence chains that run from the beginning to the end of the computation. In fact, no particularly efficient GPU versions of LZ77 exist. My work endeavors to improve upon this. Firstly, I created versions of the serial LZ77 compression and decompression algorithms that focus on improving the compression ratio. This was done using existing byte-saving techniques in the field of compression and by combining those changes with the standard LZ77 implementation [14]. Secondly, I made intermediate changes to the serial implementation in order to parallelize the algorithm and then created efficient GPU implementations of my more parallelizable codes.

II. BACKGROUND INFORMATION

This section outlines the definitions and background information pertinent to understanding LZ77 as an algorithm and research area. An example input is used throughout this document to explain precisely what LZ77 does and illustrate how each algorithm I discuss differs from one another.

In the standard LZ77 algorithm, the example input is iterated through serially by using a **search window** of previous input values. This iteration and search is used to find subsequence **matches** of previous input occurrences. A **look-ahead buffer** is used to compare input values that have yet to be processed with values in the search window. The search window consists of input values that have already been processed by the algorithm and may now be used to find matching subsequences. Each round's iteration produces a **triple** in the format: (**distance** to match, **length** of matched bytes, **value** saved). Table 1 contains definitions for the LZ77 algorithm described in this section.

Table 1: LZ77 Definitions

Terms	Definitions
Match	Input data that match previously processed subsequences within the input data.
Sliding Window (or window)	The range of previously processed input data that the algorithm searches through for a match.
Look-ahead Buffer	The range of input data actively being processed when seeking matches in the sliding window.
Rule 1	Copy the farthest match in the sliding window.
Rule 2	Copy the longest match in the sliding window. NOTE: Rule 2 takes precedence over Rule 1
Triple	The encoded format of (**distance**, **length**, **value**) for the compression scheme.
Distance	The distance to where a match was found in the sliding window when counting back from the byte being processed to the start of the match (i.e., right to left).
Length	The length of the match that was found in the sliding window.
Value	The value stored in the last element of the triple for that iteration of encoding.

LZ77 Example

The LZ77 example explained in this section uses a character string of forty bytes. This same character string will be used throughout the document to exemplify the

differences in the algorithms I discuss.

Table 2: LZ77 Encoder – Example

Input: ABCABCABCDABCDEFABCDEFGABCDEFGHABCDEFGHI = 40 bytes				
Step	**Sliding Window Size = 10**	**Look Ahead Size = 10**	**Encoding**	**Matches**
1	[A]	BCABCABCDA	(0, 0, A)	
2	A[B]	CABCABCDAB	(0, 0, B)	
3	AB[C]	ABCABCDABC	(0, 0, C)	
4	ABC[A]	BCABCDABCD	(3, 6, D)	ABCABC
5	ABCABCABCD[A]	BCDEFABCDE	(4, 4, E)	ABCD
6	ABCABCABCDABCDE[F]	ABCDEFGABC	(0, 0, F)	
7	ABCABCABCDABCDEF[A]	BCDEFGABCD	(6, 6, G)	ABCDEF
8	ABCABCABCDABCDEFABCDEFG[A]	BCDEFGHABC	(7, 7, H)	ABCDEFG
9	ABCABCABCDABCDEFABCDEFGABCDEFGH[A]	BCDEFGHI	(8, 8, I)	ABCDEFGH
TOTAL = 9 X 3 Bytes = 27 bytes < original 40 bytes.				

Table 2 shows a step-by-step example of a classically implemented LZ77 algorithm. Each step in the table represents a single iteration of the serial algorithm, and all greyed out text in the sliding window column is out of bounds for the sliding window size, which is set to ten. The look ahead buffer is also set to ten in this example. These are set to ten to simplify and demonstrate how the sliding window works, but in practice, LZ77 algorithms may use different sized sliding windows to process the data.

In step 1, the algorithm starts with an empty search window and nothing to match [A] to. Step 2's search window is not empty, but nothing matches [B]. So, the same encoding scheme for a matchless state occurs. This matchless triple is stored as: (0, 0, value). Both zeros indicate that no match exists for this iteration by saying the distance to the match is 0, the length for number of matched bytes is 0, and the value to be stored in the triple is the one currently being processed. Step 3 also stores a matchless triple for [C]. The first match is found in step 4, where it can match all three stored values within the sliding window and store this iteration's triple as (3, 6, D). The **length** is 6 because the match continues past the index holding [A] and matches the next two bytes. The

number 3 in the **distance** slot of the triple represents the location in the sliding window that is three positions away from [A], counting right to left. Lastly, the **value** slot in the triple stores the subsequent byte in the look ahead buffer. So, with a match of "ABCABC", it stores the byte that is after the second "C" in the look ahead buffer, and that value happens to be "D".

This example uses **Rule 2**, which is described in Table 1. Rule 2 can be seen in step 5. In step 5, "ABC" can be matched at a distance of 10, but seeing as Rule 2 takes precedence over Rule 1, the largest match (i.e., "ABCD") is stored for that triple. If both possible matches were the same size, Rule 1 would have been used as a tie breaker, and the triple would have stored the farthest possible match in the search window.

The compression ratio for this example is: 40 bytes / 27 bytes = ~1.48. Given different input sizes and byte sequences, the compression ratio can change, but this example shows a reduction of 13 bytes in storage cost for the given input.

This example shows why my research is focused on parallelization and why LZ77 is difficult to parallelize. I focus on LZ77 because of its ability to reduce the size of files by limiting byte redundancy, but LZ77 is difficult to parallelize due to the iterative data dependency required to match patterns in the search window.

Figure 1: Serial LZ77 – Iterative Data Dependency of Step 4

Figure 1 shows how the serial version of LZ77 has an iterative data dependency in step 4 of the example. This dependency occurs for all matched cases, but for the purpose of this figure, I focus on step 4. This fourth iteration of the input

"ABC[A]BCABCD…" would match the letters "[A]BC" with the first, second, and third iteration values. It also continues the match past the first three to include the value being searched for and the two values following in the match triple. This makes up the **length** of 6 as stated previously. Serially, this happens by simply looking back at previous values in the search window and matching them in the look-ahead buffer, but if each letter were to be handled by separate threads at the same time, the dependency becomes problematic. One thread cannot see what another thread is doing in parallel without communicating to that other thread. However, cross-thread communication for every iteration would slow down the parallel process to be worse than serial and eliminate the usefulness of parallelizing the algorithm in the first place.

Table 3: Parallel LZ77 Data Dependency of Step 4 & 5

Step	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5
Encoding Input	A	B	C	A	B
Encoded Output	(0, 0, A)	(0, 0, B)	(0, 0, C)	Dependent on threads 1, 2, & 3	ERROR
Decoding Input	(0, 0, A)	(0, 0, B)	(0, 0, C)	(3, 6, D)	ERROR
Decoded Output	A	B	C	ERROR	ERROR

Table 3 shows that during the encoding process, thread 4 will grab the "ABCABC" match from the search window and store (3, 6, D). If thread 5 operates independently from thread 4, then there is no way for thread 5 to know that thread 4 found and handled a match condition. Therefore, thread five would attempt to match the B that was handled already by thread 4. Had threads 1 – 3 been triples with matches, thread 4 would have the same issue as 5.

During the decoding process, thread 4 needs to read from other threads to fully decode the triple. For thread 4, it will require A from thread 1, B from thread 2, and C

from thread 3. However, there is no way of guaranteeing that threads 1 – 3 will have those values ready for thread 4 when it tries to read those values. Furthermore, given the error during the encoding process, thread 5 will have nothing to decode.

III. PREVIOUS WORK

LZ77 is a lossless data compression algorithm from which many LZ variants have been derived [1]. The approaches of these variants vary, but the core principle of a dictionary-based reduction to compress bytes is common among all of them. When looking for parallelized versions of LZ77, only a few results turn up. This section of my report covers LZ77 and general lossless compression parallelization in the literature.

The main paper found when searching for parallelized LZ77 work is "Massively-Parallel Lossless Data Decompression" [6]. In this work, the authors used a combination of prefix sums, warp-based removal of dependencies, and a high-water mark flag with ballot voting and a bitmap to hold all thread states in the warp. Their approach yields a two-fold speedup when compared with some CPU codes. I corresponded with the authors of this parallel decompression algorithm in the hopes of including their research in my benchmark suite. Unfortunately, they explained that the research is proprietary and cannot be made available for a comparative study.

The paper titled "CUDA Lossless Data Compression Algorithms: A Comparative Study" shows an analysis of parallelized Huffman, LZSS, and Block-Sorting Compression algorithms [7]. This paper is important because LZSS is one of the closest variants of LZ77 in use and only differs in that it makes sure that "the dictionary reference should be shorter than the string it replaces" [8]. The study is directly relevant to my work in that it has a comparison of current GPU implementations and because the parallel LZSS algorithm of the study outperforms the others in compression and decompression time. The other algorithms outperform the parallel LZSS code when the compression ratio is the metric of comparison [7]. This parallelized LZSS algorithm is

called CULZSS and was first published in 2011 [14].

This introduces the first benchmark I used to analyze my work, CULZSS. CULZSS was published with a CPU version and two GPU versions. However, I only tested the GPU version provided by the author's GitHub page [17]. This implementation passes a buffer pointer to the GPU, compresses the data into a provided memory region, and returns a pointer to the data and length [14]. This is done by splitting the data into blocks and threads, outputting the compressed data into buckets, and then finishing up the process on the CPU by merging the compressed chunks of data in the buckets together.

The next algorithm used in the benchmark suite goes by the name PLZ, and covers Lempel-Ziv Factorization, which is functionally equivalent to LZ77 [9]. They use prefix sums and list ranking in their proof. The outcome of their work boasts an $O(\log^2 n)$ runtime under certain conditions. This algorithm covers three different modes to encode, and I selected the fasted of the modes to use as a benchmark in the Analysis & Results section. The PLZ code includes no decoder. So, my decoders could not be compared to this benchmark.

LZ4 is the last of the benchmark suite and is referenced in the first paper discussed in this section [6]. That paper used it and other contemporary algorithms to test how well their work compared. Unlike with LZSS, LZ4 is a much more recent descendant of LZ77 and is actively maintained. Its lossless method of compression incorporates far more optimizations than its distant predecessors. It uses the same kind of byte-oriented approach as LZ77, has a minimum match length of four, and changes the storage format from a triple to a two byte system with "block" and "frame" optimizations [3]. The block and frame optimizations are far more complex in how they are managed

when compared to LZ77 or even my encoding scheme, but it is a contemporary algorithm that is perfect for determining how well my work performs. It is for this reason that I chose LZ4's serial implementation as the third benchmark. There is an adapted CUDA version of LZ4 that I did not test, but I wanted to test how well my code works against one of the best serial and lossless compression algorithms available on the CPU. This allows me to see how the parallelization of an older algorithm measures against an optimized, contemporary CPU version.

IV. APPROACH

This section describes my parallel and serial approaches to LZ77 compression and how they relate to previous work. As a start, the serial LZ77 implementation described in the Background section is modified. Those modifications are: adding a storage condition to the triple and including a match limitation similar to LZSS. Modifications such as these are not new, but they suit my initial goals for improved parallelization and compression.

To improve the compression ratio, the matchless triples store an additional value in the **length** slot of the triple instead of a zero. This added storage condition reduces the wasted space of one of the two zeros stored in matchless cases. For those cases, the empty zeros hold little information, but they do take up space. So, the **distance** slot is kept as zero, but the zero in the **length** slot for the triple is replaced with the byte immediately following the input **value** being saved for that iteration. An example of this would be an input data of "ABC", where "A" is being searched for in the window, but no match is found for "A". So, the storage condition saves: (0, B, A) instead of (0, 0, A). This makes use of one of the zero bytes and reduces the wasted space saved in the triple. In this example, the next byte to be searched for would be "C" and not "B", because "B" was saved in the (0, B, A) triple already.

Next, a match limitation was added. This match limitation guarantees at least a two-byte minimum for both the matched and unmatched cases. This reduces the worst-case **match** condition for LZ77, where the triple saves a match case with a **length** of only one byte. For unmatched cases, the added storage condition ensures that at least two of the input's bytes are saved in the triple. For matched cases, the algorithm only considers

matches greater than or equal to 2 bytes. Matches with only one byte are not considered and are instead saved as unmatched. With this scheme, the worst-case is only one byte extra possible for each triple.

All search window and look-ahead buffer sizes are set to 256 for both serial and parallel implementations. This change is displayed in the example tables for each of the four algorithms described in the Approach section.

Implementation Definitions

Table 4: Definitions of LZ77 Implementation Modifications

Term	Definition
Added Storage Condition	To save space in the encoding scheme, the unmatched triples save an extra byte of the input data in the length position of the triple.
Match Limitation	All matched and unmatched triples save at least two of the input bytes in the triple.

Serial Encoder

The serial application uses the longest match and furthest match rules given by LZ77 and is illustrated by Table 5's walkthrough of the example input. In addition to those classic rules, the serial approach also includes the modifications outlined at the start of the Approach section.

Table 5: Serial Encoder – Example

Input: ABCABCABCDABCDEFABCDEFGABCDEFGHABCDEFGHI = 40 bytes				
	Search Window	**Look Ahead**	**Encoding**	**Matches**
1	[A]	BCABCABCDA	(0, B, A)	
2	AB[C]	ABCABCDABC	(0, A, C)	
3	ABCA[B]	CABCDABCDE	(3, 5, D)	BCABC
4	ABCABCABCD[A]	BCDEFABCDE	(4, 4, E)	ABCD
5	ABCABCABCDABCDE[F]	ABCDEFGABC	(0, A, F)	
6	ABCABCABCDABCDEFA[B]	CDEFGABCDE	(6, 5, G)	BCDEF
7	ABCABCABCDABCDEFABCDEFG[A]	BCDEFGHABC	(7, 7, H)	ABCDEFG
8	ABCABCABCDABCDEFABCDEFGABC DEFGH[A]	BCDEFGHI	(8, 8, I)	ABCDEFGH
TOTAL = 8 X 3 Bytes = 24 bytes < original 40 bytes.				

Table 5 shows the step-by-step walkthrough of my serial encoder. Step one shows

that none of the input has been processed yet, and the search window is empty of values to match with [A]. The only difference between this step one and the step one listed in the Background section is the storage saving mechanism used in the **length** slot. In this algorithm, the **length** slot of the triple is used as an added storage condition and provides the first iteration with the encoding scheme of (0, B, A) instead of (0, 0, A).

The rest of the walkthrough operates in the same manner as the original example listed in the Background section. For this input, the compression ratio is improved due to the added storage condition. Instead of the compression resulting in the original example's output of 27 bytes, it is 24 bytes. That means 3 additional bytes are saved by using the modifications for this input.

It is from this serial encoder that I began work on the parallelized version of the encoder. The modifications and classic LZ77 rules are used in the parallel version, but due to the data dependency restrains of the serial application, the way bytes and matches are handled in the parallel version is quite different. I first coded some of the changes in serial to make the algorithm more parallelizable, and then translated the parallelizable code to callable CUDA kernels. These changes are discussed in the Parallel Encoder section of this document.

Serial Decoder

The serial decoder takes the encoded input and serially iterates through each input triple to restore the original, uncompressed input. In Table 6, the **Output** column shows the decoded bytes in red and the match references highlighted in blue. The **Matched Output** column shows the populated bytes that are derived by decoding matches.

Table 6: Serial Decoder – Example

INPUT: (0,B,A) (0,A,C) (3,5,D) (4, 4, E) (0, A, F) (6,5,G) (7,7,H) (8,8,I)			
	Input	**Output**	**Matched Output**
1	(0, B, A)	AB	
2	(0, A, C)	ABCA	
3	(3, 5, D)	ABCA_ _ _ _ _D	ABCABCABCD
4	(4, 4, E)	ABCABCABCD_ _ _ _E	ABCABCABCDABCDE
5	(0, A, F)	ABCABCABCDABCDEFA	
6	(6, 5, G)	ABCABCABCDABCDEFA_ _ _ _ _G	ABCABCABCDABCDEFABCDEFG
7	(7, 7, H)	ABCABCABCDABCDEFABCDEFG_ _ _ _ _ _ _H	ABCABCABCDABCDEFABCDEFGABCDEFGH
8	(8, 8, I)	ABCABCABCDABCDEFABCDEFGABCDEFGH_ _ _ _ _ _ _ _I	ABCABCABCDABCDEFABCDEFGABCDEFGHABCDEFGHI
OUTPUT: ABCABCABCDABCDEFABCDEFGABCDEFGHABCDEFGHI			

Most of Table 6 is straightforward. Each triple is read sequentially, and the matched output is populated accordingly. In step 3, the bytes “BCA” are available due to the second iteration (i.e., step 2). By iterating through the matched bytes, the first three compressed spaces are filled prior to the final two spaces. This means that those final two spaces are filled by “BC” when continuing the byte restoration sequentially.

To parallelize this algorithm, there is still the issue of iterative dependency. As seen in step three, the data required to process and restore the compressed bytes are given in step two. If I were to parallelize this process directly, the same issues as described in the Background section would occur. This is why parallelization is nontrivial for both the encoder and decoder.

Parallel Encoder

The encoder algorithm processes the data in seven steps, as seen in Figure 2. It starts by having each byte of the input data processed in parallel. Each byte is compared to the previous bytes within the search window. The search window consists of the previous 256 bytes that exist within the input data. These independent find operations are done in parallel and will naturally have overlapping matches, but it allows the encoder to

massively parallelize this step in LZ77 compression. From there, the encoder performs a parallel set of iterations to determine and mark the locations of each non-overlapping match and populates the locations array with both that data and the data of bytes without matches. The locations array is the same size as the input array, and therefore each element in the array can also be split across all threads. The process of marking the locations correctly will be described in greater detail later in this section. However, I will refer to the non-overlapping matches and unmatched instances marked in the locations array as **triples**, because that is what each location marked represents. The locations are marked by "1" if a triple exists and "0" if no triple is marked for that position/thread.

After each triple location is marked, the prefix array is populated with precisely the same binary numbers as the locations array. The populated prefix array is then used to perform a parallelized and inclusive prefix sum on the match locations. For each location marked in the prefix array, the sum increases by one, and the resulting total equals the number of triples up to the current location. This allows the encoder to determine the output size of the compressed data and where to output the triple of each marked thread.

The output size is used to create an output array that is populated by the finalized triples of compressed data. The output array of triples is copied from the GPU and saved to a binary file. At the start of the binary file, the original size of the input data is saved for the decoder to use.

These phases cover a high-level flow of the encoder algorithm and are depicted in Figure 2. All definitions used in this section are listed in Table 7, and Table 8 shows an example of this algorithm to better illustrate how the input is compressed.

Table 7: Parallel Encoder & Decoder Definitions

Term	Definition
Maxlen	Maxlen = 256. This is the maximum size of the search window.
ThreadsPerBlock	ThreadsPerBlock = 512. This is the number of threads per block used.
Input array	The file input stored as bytes.
Prefix array or Prefix Sum array	Prefix array is the array that is populated with the values that the prefix sum operation will use.
Match length array	The match length for each processed byte in the file is stored in the match length array.
Match distance array	The match distance for each processed byte in the file is stored in the match distance array.
Locations array	The array of locations saved during phase 2 of encoding (i.e., triple locations).
Inclusive Prefix Sum	Inclusive prefix sum is the cumulative addition of each element in a list. Example: 1, 2, 3, 4, 5, 6 → 1, 3, 6, 10, 15, 21
Parent Array	The array that is populated with the values that the union find operation will use.
Offset	Offset is the 256 indices of padding added to the front of the parent array. Each index in this padding represents the 256 possible bytes any match could have.
Union Find (e.g., disjoint-set)	Navigates a chain of parent array pointers to the offset in order to discover the encoded match characters. All chains of parent array pointers end by pointing to an index in the offset. That offset index represents the byte equivalent of the match.
Output Array	Output array is the array that is populated with the fully decoded or encoded data.
Pre-encoded Data Size	Pre-encoded data is the first few bytes of the encoded file that is provided as input to the decoder. This value is set by the encoder during the encoded file's creation and is then used by the decoder to account for the size correctly.

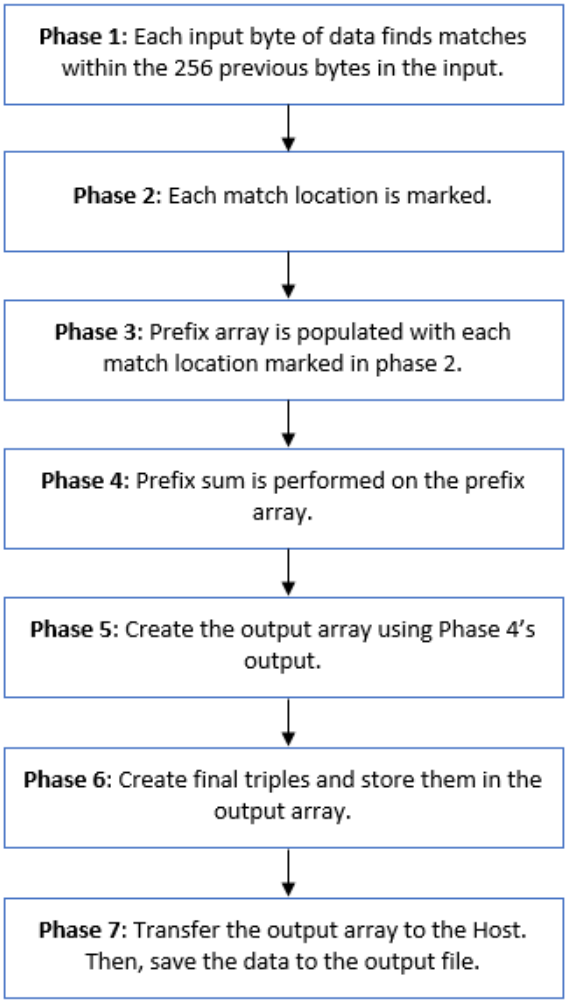

Figure 2: Parallel Phases of the Encoder.

Phase 1 consists of each byte of data being independently processed. As stated previously, this phase finds possible matches irrespective of overlap. In fact, overlap is to be expected due to how this is parallelized. Rules 1 and 2 are both accounted for as my algorithm gives precedence to largest matches over furthest matches. Additionally, all modifications discussed at the beginning of the Approach section are included in this algorithm. Each byte searches the 256 bytes preceding it to find the best match possible. The find procedure of Phase 1 returns both an array of match distances and match lengths corresponding with each byte processed. These arrays are used during Phase 2 to mark

each match location and are used to create the triples in Phase 6.

Phase 2 has several functions that work to determine the correct location of each match, and if no match exists, it marks the correct location for the unmatched byte. The locations array is the same size as the input array, match distance array, and match length array. Therefore, each thread can be given an element within the array to process in parallel. The location marking process uses these arrays in a series of parallel functions to determine which of the overlapping matches should be marked. Firstly, it initializes supporting variables, determines the reachability of the matches via an atomic add on the matches' length, and builds chains by joining sections with just a single match. Each of these steps is done in parallel by kernel calls, but the next steps require a do-while loop to remove dangling (unreachable) match chains and to merge the remaining chains. The undangling and merge functions are kernel calls within the do-while loop, which iterates until the Boolean flag says that only one chain is left. With the triple positions marked, the encoder will be able to process each position in parallel.

During Phase 3, each thread stores the marked locations of triples to the prefix array. It is a one-to-one transfer of the locations array data to the prefix array. This allows Phase 4 to perform the prefix sum in parallel and on the prefix array without any need to modify the locations array. Phase 4 performs the inclusive prefix sum by using an NVlab's prefix-sum function from the CUB library [16]. The last value of the result of this prefix sum call is the overall size of the compressed output. For each location marked with a "1" and not a "0", the sum increases by one. Therefore, the inclusive prefix sum's total shows precisely the number of triples my compression algorithm is going to produce. Each intermediate sum of the locations provide the position data for Phase 6 to

write the compressed triples in parallel without fear of overlap between threads.

The output array created in Phase 5 is used to store the compression triples and to store the original size of the input. Phase 6 creates the triples and stores them in the output array. These triples are based on the non-overlapping sequences derived from Phase 2. When combined, these non-overlapping sequences cover the entire input and therefore can act as the finalized triples for the algorithm. This is done by looking at the locations array in parallel, one array element per thread, and letting the threads with a marked location for a triple process the data relating to that position. What I mean by this is that if and only if a thread has a triple location marked will it proceed in creating the triple for that position. On the threads that are marked, the match length array holds the corresponding **length** for the triple at the same index position as in the locations array. This is also true for the match distance array being used to acquire the triple's **distance**. The **value** for each triple is easily read from the input array because the triples' position and match length have been accounted for. This parallelized procedure results in the output array being populated by the correct, compressed triples.

Phase 7 is the final stage of the algorithm. It copies the output array back to the host and saves both the compressed data and the original file size to a binary file. This file is compatible with my decoder algorithm and has the necessary information to allow the decoder to restore the original data losslessly.

Parallel Encoder Example

In this section, I will be going through each phase of the parallel encoder and show how they work on the example input used throughout this document. Each column heading with "T" and a number represents a thread on the GPU, and each row represents

a parallelized step on the GPU.

Table 8: Parallel Encoder – Example

Input: ABCABCABCDABCDEFABCDEFGABCDEFGHABCDEFGHI							
	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**
1	[A]	A[B]	AB[C]	ABC[A]	ABCA[B]	ABCAB[C]	ABCABC[A]
2	- Dis = 0 Len = 0	- Dis = 0 Len = 0	- Dis = 0 Len = 0	[ABCABC] Dis = 3 Len = 6	[BCABC] Dis = 3 Len = 5	[CABC] Dis = 3 Len = 4	[ABC] Dis = 6 Len = 3
3	Loc = 1	Loc = 0	Loc = 1	Loc = 0	Loc = 1	Loc = 0	Loc = 0
4	Pfx = 1	Pfx = 0	Pfx = 1	Pfx = 0	Pfx = 1	Pfx = 0	Pfx = 0
5	Sum = 1	Sum = 1	Sum = 2	Sum = 2	Sum = 3	Sum = 3	Sum = 3
6	(0,B,A)		(0,A,C)		(3,5,D)		
Output: (0,B,A)(0,A,C)(3,5,D)(4,4,E)(0,A,F)(6,5,G)(7,7,H)(8,8,I)							

Table 8 shows the input as it is transformed in phases 1 through 6, but for simplicity, the table only shows the first seven threads. It starts with the match finding stage. Each thread (i.e., T1 through T7) represents a single byte in the input. These bytes are managed independently in parallel, and therefore, during the match phase, they are not aware of what the other threads' findings are. T1 through T3 show no matches. T4 shows the first match of ABCABC for the given byte [A]. This match starts at a distance of three from the starting position and finds a match of six bytes in total. Both T4 and T5 can traverse back three positions to find the start of their match (i.e., Dis). Iterating from that match point to the last matching byte in the input gives the length value (i.e., Len). In T7, the furthest match is used because no greater length matches are found.

Row three shows how the locations are marked. For this input, the algorithm performed two parallel iterations to determine which of the matched and unmatched triple locations to use. As expected, T1 holds the first triple and accounts for T2's byte. T2's byte is stored in the **length** portion of the triple, the **distance** remains zero, and the letter "A" is stored in the **value** position of the triple. This procedure of storing the subsequent byte (i.e., T2's B) in the length position is used for all unmatched triples.

T5 shows the first instance of a matched triple being used. Although T4 technically had a match found, that byte is already accounted for by T3's unmatched triple. So, the first match triple saved is T5's. T5's triple accounts for T6 and T7's byte. All of this is shown in rows three through six. Row three shows the locations marked for the triples. Row four shows the location array's values being stored in the prefix array to prepare for the prefix sum, and row five shows how each location marked iterates the sum value stored during the prefix sum. Row six shows the triples created as a result of the previous phases.

The first two triples shown in Table 8 are for unmatched cases, where the subsequent byte is stored in the length position of the triple. The third triple in the example shows a match case that covers T5 through T9 for the match and includes the subsequent byte (i.e., T10's) in the **value** position of the triple.

The final row of the table shows the fully compressed output after the algorithm finishes. This shows a reduction in bytes from the original 40 bytes to the compressed 24 bytes. Each triple represents three bytes, and there are eight triples. This is how the compressed size is calculated, and the resulting compression ratio is rounded to 1.67.

Parallel Decoder

The decoder employs two important techniques to allow for parallelization: prefix sum and union find. It is not uncommon for prefix sum or union find (i.e., disjoint-set data structure) to be used for parallelizing serial algorithms [6][10]. From what my research has indicated thus far, no previous works use these techniques together on LZ77. The implementation is described in detail within Figure 3, and Table 9 exemplifies the parallel decoding process via the same example input used in previous sections.

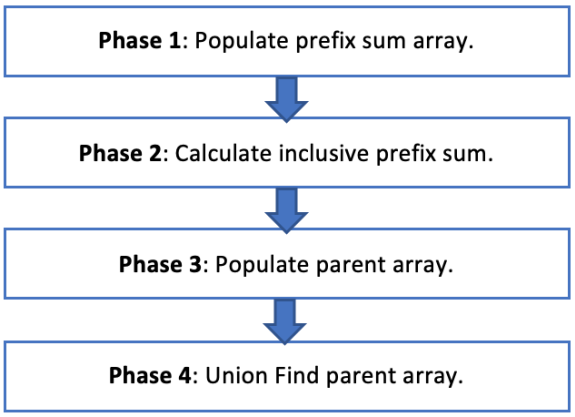

Figure 3: Parallel Phases of the Decoder.

Table 9: Parallel Decoder – Example

Input: (0,B,A) (0,A,C) (3,5,D) (4,4,E) (0,A,F) (6,5,G) (7,7,H) (8,8,I)								
	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**
1	(0, B, A)	(0, A, C)	(3, 5, D)	(4, 4, E)	(0, A, F)	(6, 5, G)	(7, 7, H)	(8, 8, I)
2	AB = 2	CA = 2	5 + D = 6	4 + E = 5	FA = 2	5 + G = 6	7 + H = 8	8 + I = 9
3	AB	CA	_ _ _ _ _ D	_ _ _ _ E	FA	_ _ _ _ _G	_ _ _ _ _ _ _H	_ _ _ _ _ _ _ _ I
4	AB	CA	BCABCD	ABCDE	FA	BCDEFG	ABCDEFGH	ABCDEFGHI
OUTPUT = ABCABCABCDABCDEFABCDEFGABCDEFGHABCDEFGHI								

In Figure 3, Phase 1 pairs with rows 1 and 2 in Table 9. This may seem strange because row 1 just shows the nine triples unchanged. However, the triples themselves hold the data necessary to drive all phases of the algorithm. What row 1 represents is the division of the input among the available threads. In this example, there are a total of 8 threads, one for each triple. Each triple can be processed independently, and given the triple's **length** and **value** data, each thread knows how many bytes it produces for the decoded output.

There are two possible cases: unmatched and matched triples. In the unmatched cases, the triple's **length** and **value** positions each hold a byte for the output and are always going to account for two bytes of the output. This is due to the added storage condition described at the beginning of the Approach section. In match cases, the triple's **length** position holds the size of the match, and the triple's **value** holds one byte for the

output. This accounts for **length** + 1 number of bytes for the output, which is handled by the triple's thread. This addition can be seen in row 2 of Table 9. It is with these numbers that the prefix sum array is populated. Each number represents one value per index of the prefix sum array and will be used in Phase 2's operation. The populated data for this example can be seen in the **Prefix Array Populated** column of Table 10, where each "T" value represents a thread.

For Phase 2, the values populated in the previous phase must now undergo the same inclusive prefix sum operation I referenced in the Parallel Encoder section [16]. This provides the precise index range each thread can safely operate within. Without this range, any attempt to do the subsequent phases would result in data race conditions. The result of this prefix sum can be seen in the **Prefix Sum** column of Table 10. Each number in said column will be used to determine what output index to start and stop populating data within. In Table 10, thread 1 starts at 0 and can submit write changes to all output indices less than 2. Thread 8 can submit write changes to all output indices starting from thread 7's **Prefix Sum** value to indices less than 40. In the later phases, both the parent and output arrays use the prefix sum values to write in a thread safe manner.

Table 10: Phase Data – Example

T	Encoding	Prefix Sum Array Populated	Prefix Sum	Populated Parent Array	Find Operation	Matches
1	(0, B, A)	2	2	AB		
2	(0, A, C)	2	4	CA		
3	(3, 5, D)	6	10	_ _ _ _ _ D	BCABCD	BCABC
4	(4, 4, E)	5	15	_ _ _ _ E	ABCDE	ABCD
5	(0, A, F)	2	17	FA		
6	(6, 5, G)	6	23	_ _ _ _ _ G	BCDEFG	BCDEF
7	(7, 7, H)	8	31	_ _ _ _ _ _ _ H	ABCDEFGH	ABCDEFG
8	(8, 8, I)	9	40	_ _ _ _ _ _ _ _ I	ABCDEFGHI	ABCDEFGH
Output = ABCABCABCDABCDEFABCDEFGABCDEFGHABCDEFGHI						

Phase 3 can be seen in row 3 of Table 9 or in the **Populated Pared Array** column

of Table 10. This stage of the algorithm is used to prepare the parent array for Phase 4's union find operation. In order to populate this correctly, an array that is the size of the output plus 256 is used. This padding of 256 indices is referred to as the **offset**, and the offset is used specifically because there are 256 values possible for any byte in the input. In other words, each index of the offset can be viewed as a value for every possible byte being decoded. The parent array keeps the offset unpopulated and uses the 0 to 255 indices themselves as parent references for matched bytes.

The parent array is populated starting from 256 onward and uses the prefix sum values found in the previous phase to be sure all writes are done in a thread safe manner. All unmatched values are decoded without conflict and can be used to populate the indices after the offset. However, this is not done by inserting the value into the array. Instead, the parent array value is set to be equal to the offset's index matching it. This is called a parent pointer because it is literally pointing to its numeric equivalent, and it is used for both unmatched and matched cases. The unmatched cases are self-explanatory as they are stored directly in the **length** and **value** positions of the triple and can point to their parent directly. Matched cases are handled differently. Instead of pointing to an index within the offset, they point to the position known to be where their match is located. This allows for a chain of references to be resolved in a near constant work time complexity, during Phase 4.

In Phase 4, each thread performs a find operation for every value they have that has yet to be accounted for (i.e., the match cases). Union find is a known means by which this operation can happen in a thread safe manner [12]. Essentially, each parent pointer chains with the next and results in populating all remaining parent array indices that are

outside of the offset's range. The multi-threaded chains each end when a parent pointer referencing an index from the offset because that index is the equivalent of the matched byte. Table 9's row 4 matches this phase, and Table 10's **Find Operation** column does as well. That covers the decoder implementation on the GPU.

V. EXPERIMENTAL METHODOLOGY

My work includes the modified serial versions of the LZ77 encoder and decoder, the parallelized version of the decoder, and the parallelized version of the encoder. To test each approach, I compare the serial implementations to the parallel implementations and do a comparative study with other parallelized attempts at lossless encoding and decoding in literature. The test sets include images, spreadsheets, text documents, and large graph files. Both the serial and parallel approaches successfully encode and decode all of the test data. Additionally, the parallel and serial versions are interchangeable in that both serial and parallel versions produce the same result, but each algorithm is tested and verified independently.

The following table includes the current list of files used in the test set, their file type, and the size of the file.

Table 11: Test Set Files

File Name	Size in MB	File Type
Large Canterbury Corpus: [11]		
Bible.txt	4.05	Text
E.coli	4.64	Text
world192.txt	2.47	Text
Graph Dataset: [13]		
rgg_n_2_22_s0.egr	259.65	Graph
cit-Patents.egr	147.25	Graph
coPapersDBLP.egr	124.13	Graph
in-2004.egr	114.26	Graph
as-skitter.egr	95.55	Graph
2d-2e20.sym.egr	37.72	Graph
amazon0601.egr	21.16	Graph
internet.egr	3.60	Graph
Generated Dataset:		
Test4.bmp	9.36	Image
Canterbury Corpus: [11]		
Kennedy.xlsx	1.03	Spreadsheet

These varied file types and sizes are used to test validity, provide a wide scope of analysis, and ensure that there are no corner cases for the encoder or decoder versions. Sources for the files used are listed in the table. All but the image file are used in other parallelization and compression focused publications and provide a great baseline from which to interpret test results.

I used an NVIDIA GeForce RTX 3090 GPU and an Intel Xeon Gold 6226R CPU for all my tests.

VI. ANALYSIS & RESULTS

There are two metrics by which I analyze the algorithms: compression ratio and throughput. The compression ratio is the size of the original data divided by the size of the compressed data. The timers used to measure the runtime, from which I calculate the throughput, are captured on the same section of code in my parallel and serial codes.

In addition to the serial versus parallel comparison, I also compare each version with existing algorithms from the literature. When comparing my algorithms to other compression approaches, great effort was expended in being sure that the timed sections in the code were comparable. In some cases, different timed sections within my code are used to ensure fairness in analysis.

Serial vs. Parallel Results

For the test set, two separate measurements of time were taken: (1) the time for both data being copied to the GPU and the compression/decompression portion, (2) just the time of the compression/decompression on the GPU. The reason the time measurement with the data copying was included is due to how one of the benchmarks measured their time. It was important to have my time measurements be comparable with the benchmarks to be sure my analysis was as accurate as possible. So, the following tables represent the two different throughput measurement sets. The time measurement without copying data to the GPU will be referred to as “default” and the time measurement including the data copying will be referred to as “extended”. All table references to encoder throughput will be labeled “ET”, and all table references to decoder throughput will be labeled “DT”.

Table 12: Serial and Default Parallel Throughput Results.

Test File	Serial ET (MB/s)	Parallel ET (MB/s)	Serial DT (MB/s)	Parallel DT (MB/s)	Encoder Speedup	Decoder Speedup
rgg_n_2_22_s0.egr	20.54	1,071.06	147.27	10,258.22	52.14	69.65
cit-Patents.egr	13.29	1,124.12	117.69	37,540.00	84.59	318.97
coPapersDBLP.egr	19.19	990.98	146.31	20,313.85	51.65	138.84
in-2004.egr	20.46	697.35	161.04	10,829.63	34.08	67.25
as-skitter.egr	17.98	873.59	120.51	15,698.36	48.58	130.27
2d-2e20.sym.egr	13.72	818.13	119.69	29,396.71	59.65	245.61
amazon0601.egr	15.28	802.96	102.93	28,480.29	52.56	276.68
test4.bmp	15.92	654.86	106.03	21,288.53	41.14	200.78
E.coli	6.03	551.34	106.46	11,258.96	91.44	105.75
bible.txt	11.17	658.70	85.17	9,981.24	58.97	117.19
internet.egr	14.35	489.49	129.60	6,623.45	34.12	51.11
world192.txt	12.60	585.77	81.26	4,546.69	46.50	55.95
kennedy.xls	23.59	550.81	182.94	2,954.79	23.35	16.15
Geometric Mean	14.94	733.95	120.45	12,757.16	49.13	105.91

Table 12 shows the throughputs of the serial version and **default** parallel version of the encoder. Using these throughputs, the speedup values for the parallel code can be derived. The bottom row shows the geometric mean of the column values. For example, Table 12 shows a geometric mean speedup of 49.13 for the parallel encoder and a geometric mean speedup of 105.91 for the parallel decoder. The is a substantial improvement on throughput for both the parallel encoder and decoder compared to the serial code and shows how massively parallelizing the algorithms improves the overall efficiency of the lossless compression process. Table 13 shows the **extended** throughputs, which include preparation items like copying data to the GPU before processing the data. The **default** throughputs focus on how much time the actual compression and decompression sections of the code actually take.

Table 13: Serial and Extended Parallel Throughput Results

Test File	Serial ET (MB/s)	Extended ET (MB/s)	Serial DT (MB/s)	Extended DT (MB/s)	Extended Encoder Speedup	Extended Decoder Speedup
rgg_n_2_22_s0.egr	20.54	931.85	147.27	10,258.22	45.36	8.06
cit-Patents.egr	13.29	936.01	117.69	37,540.00	70.43	9.70
coPapersDBLP.egr	19.19	876.16	146.31	20,313.85	45.66	8.89
in-2004.egr	20.46	643.66	161.04	10,829.63	31.46	7.94

as-skitter.egr	17.98	778.83	120.51	15,698.36	43.31	10.36
2d-2e20.sym.egr	13.72	721.35	119.69	29,396.71	52.59	10.24
amazon0601.egr	15.28	715.84	102.93	28,480.29	46.86	12.29
test4.bmp	15.92	595.41	106.03	21,288.53	37.41	11.22
E.coli	6.03	513.16	106.46	11,258.96	85.11	11.07
bible.txt	11.17	599.39	85.17	9,981.24	53.66	12.79
internet.egr	14.35	458.95	129.60	6,623.45	31.99	8.40
world192.txt	12.60	534.38	81.26	4,546.69	42.43	11.63
kennedy.xls	23.59	511.67	182.94	2,954.79	21.69	4.88
Geometric Mean	14.94	660.42	120.45	1,149.42	44.21	9.54

The thing to note in Table 13 is that the geometric mean change for speedup is more pronounced in the extended decoder than it is in the extended encoder. Timing the data copying portion of the code with the compression section of the code can significantly change the time, but even with this inclusion, the extended decoder speedup is 9.54. The encoder speedup is 44.21. The point of distinguishing these differences is that the CULZSS benchmark comparison uses the **extended** throughputs and not the **default**. CULZSS includes portions of their setup in their timed sections, and in order to be fair, I made the **extended** timers to accommodate that difference.

Next, it is important to cover the compression ratio of the encoder. As stated previously, the serial and parallel compression ratios are identical due to the fact that the exact same triple scheme is used in both cases. Table 14 shows how the original size compares to the compressed size provided by my encoders and shows the resulting compression ratios.

Table 14: Compression Output Comparison

Test Files	Original Size (MB)	Compressed Size (MB)	Compression Ratio
rgg_n_2_22_s0.egr	259.65	199.58	1.30
cit-Patents.egr	147.25	189.27	0.78
coPapersDBLP.egr	124.13	83.25	1.49
in-2004.egr	114.26	57.45	1.99
as-skitter.egr	95.55	72.77	1.31
2d-2e20.sym.egr	37.72	40.69	0.93

amazon0601.egr	21.16	19.05	1.11
test4.bmp	9.36	8.25	1.13
E.coli	4.64	2.74	1.69
bible.txt	4.05	3.20	1.26
internet.egr	3.60	1.98	1.82
world192.txt	2.47	2.21	1.12
kennedy.xls	1.03	0.29	3.49
Geometric Mean			1.39

There are only two instances in the test set where the encoder fails to compress the data down to a smaller size. Those instances are “cit-Patents.egr” and “2d-2e20.sym.egr”. All other files were compressed to a smaller size, and given the throughput data, all of the files were processed faster with the Parallel implementation. So, only two out of eight graph files failed to reduce in size, and all of the other file types tested were successfully reduced. That does not mean that all files of those types will successfully reduce in every case, but it does show that the algorithm generally compresses data of differing types well. Take into account the speedup, and one can also conclude that the parallel implementation is a significant improvement in throughput compared to serial LZ77 compression.

Benchmark Tests

There were very few offerings of preexisting parallel LZ77 implementations available to test my results against. Many emails were sent to professors of related publications and only a few responded or had publicly accessible code for me to test. However, there were three different compression algorithms that I was able to utilize as my benchmark suite, and they cover different approaches of both parallel and serial lossless compression.

PLZ

The first benchmark used was an encoder-only, CPU parallelized compression algorithm that is functionally equivalent to LZ77 [9]. This algorithm is called PLZ and was covered in the Previous Work section. PLZ has three different modes of compression. Each mode represents a different throughput capability of PLZ. Therefore, I chose to use the fastest mode to compare my code against. The throughput comparison can be seen in Table 15. The second column shows the throughputs for the fastest PLZ mode, and the third and fourth columns show my serial and parallel throughput compared to PLZ.

Table 15: PLZ Throughput Comparison

Test Files	PLZ Throughput (MB/s)	Serial ET / PLZ Throughput	Parallel ET / PLZ Throughput
rgg_n_2_22_s0.egr	2.51	8.19	426.94
cit-Patents.egr	1.93	6.89	582.48
coPapersDBLP.egr	2.54	7.56	390.39
in-2004.egr	2.78	7.35	250.53
as-skitter.egr	2.19	8.22	399.54
2d-2e20.sym.egr	2.67	5.15	306.94
amazon0601.egr	2.56	5.96	313.05
test4.bmp	3.84	4.14	170.43
E.coli	3.29	1.83	167.59
bible.txt	3.58	3.12	183.90
internet.egr	4.39	3.27	111.60
world192.txt	4.21	2.99	139.02
kennedy.xls	9.11	2.59	60.44
Geometric Mean	3.21	4.65	228.39

As seen in Table 15, the PLZ throughput geometric mean is 3.21 MB/s, and when compared to both my serial and parallel encoders, it is far slower than my implementations for every file tested. Bearing in mind that PLZ is a parallelized LZ77 equivalent, it might seem surprising that even the serial implementation has a greater throughput, but there happens to be a very good reason why it is slower. The compression ratio it achieves is much greater than mine. This is a common trade off in compression

algorithms [18]. Depending on the use case for an algorithm, people may choose one aspect over the other. PLZ would be the better choice if compression ratio was the highest priority. However, if throughput was more important for the use case, then both my serial and parallel implementations would be better choices. Table 16 shows how my compression ratio compares with PLZ's.

Table 16: PLZ Compression Ratio Comparison

Test Files	Original Size (MB)	PLZ Compressed Size (MB)	PLZ Compression Ratio	Encoder Ratio / PLZ Ratio
rgg_n_2_22_s0.egr	259.65	30.76	8.44	0.15
cit-Patents.egr	147.25	32.78	4.49	0.17
coPapersDBLP.egr	124.13	3.79	32.72	0.05
in-2004.egr	114.26	6.52	17.52	0.11
as-skitter.egr	95.55	13.42	7.12	0.18
2d-2e20.sym.egr	37.72	8.65	4.36	0.21
amazon0601.egr	21.16	3.85	5.49	0.20
test4.bmp	9.36	1.67	5.59	0.20
E.coli	4.64	0.43	10.72	0.16
bible.txt	4.05	0.34	11.99	0.11
internet.egr	3.60	0.65	5.54	0.33
world192.txt	2.47	0.19	12.80	0.09
kennedy.xls	1.03	0.15	6.76	0.52
Geometric Mean			8.54	0.16

Referencing Table 14, my encoder ratio geometric mean is 1.39. If that is compared with the PLZ ratio geometric mean, it is clear why their algorithm took so long to complete. It is able to achieve an 8.54 compression ratio geometric mean, which is excellent. Every single file shows a red ratio in the **Encoder Ratio / PLZ Ratio** column because none of my ratios beat theirs. However, the time cost to achieve this great ratio is significant. This illustrates the common tradeoff between compression ratio and throughput. The other benchmarks used in this section show the same tradeoff to a degree but not as clearly as this PLZ example.

CULZSS

For the second benchmark, I chose a well-cited algorithm by the name of CULZSS [14]. It is a parallelized approach to LZSS, which is closely related to LZ77 [8]. Their approach includes both CPU and GPU parallelization techniques, as described in the Previous Work section. It is because of their unique approach that it was prudent to use the extended throughput timers instead of the default timers I use for the other two benchmarks. To do otherwise would have been an unfair comparison. This comparison can be found in Table 17. As stated before, ET stands for Encoder Throughput, and DT stands for Decoder Throughput. Additionally, Table 17 reduces the word "Extended" to Ext. in order to make the headers more readable.

Table 17: CULZSS Throughput Comparison

Test Files	CULZSS ET (MB/s)	CULZSS DT (MB/s)	Serial ET / CULZSS ET	Ext. ET / CULZSS ET	Serial DT / CULZSS DT	Ext. DT / CULZSS DT
rgg_n_2_22_s0.egr	101.99	214.90	0.20	9.14	0.69	5.53
cit-Patents.egr	105.73	232.22	0.13	8.85	0.51	4.91
coPapersDBLP.egr	144.42	213.94	0.13	6.07	0.68	6.08
in-2004.egr	159.54	244.50	0.13	4.03	0.66	5.23
as-skitter.egr	125.47	233.97	0.14	6.21	0.52	5.34
2d-2e20.sym.egr	122.97	289.54	0.11	5.87	0.41	4.23
amazon0601.egr	86.66	205.25	0.18	8.26	0.50	6.16
test4.bmp	56.12	112.57	0.28	10.61	0.94	10.57
E.coli	32.19	67.14	0.19	15.94	1.59	17.55
bible.txt	31.49	58.80	0.35	19.03	1.45	18.53
internet.egr	29.08	57.74	0.49	15.78	2.24	18.85
world192.txt	20.25	38.56	0.62	26.39	2.11	24.51
Geometric Mean	68.74	134.77	0.21	9.81	0.86	8.71

Table 17's first two columns show CULZSS' encoder throughput and decoder throughput, respectively. The subsequent four columns include the serial encoder comparison, the parallel encoder comparison, the serial decoder comparison, and the parallel decoder comparison. Unsurprisingly, CULZSS beats my serial code considerably, but this is a good thing. It further shows that this algorithm is a good

comparator with which my parallel algorithms can be analyzed. Where PLZ effectively illustrated the tradeoff between time and compression, CULZSS balances its compression ratio cost similarly to mine. This is illustrated in both Table 17 and Table 18's comparison columns.

Looking specifically at the parallel comparison of throughputs in Table 17's fifth and seventh columns, it is clear that my algorithm's throughput exceeds CULZSS' encoder and decoder by a geometric mean of 9.81 and 8.71, respectively. In fact, every file processed by the parallel encoder and decoder reported a greater throughput than what CULZSS did for those same files.

Table 18: CULZSS Compression Ratio Comparison

Test Files	Original Size (MB)	CULZSS Compressed Size (MB)	CULZSS Compression Ratio	Encoder Ratio / CULZSS Ratio
rgg_n_2_22_s0.egr	259.65	222.39	1.17	1.11
cit-Patents.egr	147.25	145.34	1.01	0.77
coPapersDBLP.egr	124.13	98.86	1.26	1.19
in-2004.egr	114.26	69.74	1.64	1.21
as-skitter.egr	95.55	74.98	1.27	1.03
2d-2e20.sym.egr	37.72	36.00	1.05	0.88
amazon0601.egr	21.16	18.39	1.15	0.97
test4.bmp	9.36	7.91	1.18	0.96
E.coli	4.64	3.42	1.36	1.25
bible.txt	4.05	3.17	1.28	0.99
internet.egr	3.60	1.57	2.30	0.79
world192.txt	2.47	1.53	1.62	0.69
Geometric Mean			1.32	0.97

Table 18 shows the compression ratio of CULZSS in the **CULZSS Compression Ratio** column. Unlike with my algorithm, CULZSS successfully reduces every file. This includes the two files that my algorithm failed to compress to a smaller size. However, when CULZSS' ratios are compared to my encoder ratios in the **Encoder Ratio / CULZSS Ratio** column, the geometric mean result is nearly one-to-one. As an additional

test, I ran the algorithms on a few more files to see if the trend would remain. Depending on the files I added, all of which were large, the geometric mean consistently hovered either above or below 1. Meaning, CULZSS and my encoder have nearly a one-to-one ratio trend. Additionally, if you look at Table 14, the geometric mean for my encoder is 1.39. That is larger than CULZSS' 1.32 geometric mean of the same files. However, when calculating the comparison, the result is 0.97. This shows how the ratio variance plays a role in the resulting geometric mean.

It is also important to note that my serial decoder performs better on the smaller files of the test set than CULZSS' decoder, but this was expected. The Kennedy.xlsx file was removed for this precise reason. CULZSS specifies the file sizes must exceed 1MB, and Kennedy.xlsx was too close to that that size for CULZSS to process it. CULZSS' design specifications require larger files in order to be truly efficient, but this is quite reasonable given the costs of exchanging data from host to device in GPU programming. My design allows for smaller files than CULZSS to still run efficiently. However, mine also loses out to the serial version if the files get small enough. This is an excellent example of the overhead associated with parallelizing algorithms. In both algorithms there is a lower bound where the overhead exceeds the benefit.

To summarize this benchmark, my parallel encoder consistently runs faster on all files of the data set by a substantial margin, and although CULZSS successfully reduces two files that my algorithm cannot, it has nearly the same geometric-mean compression ratio.

LZ4

For the final benchmark, I used LZ4. The LZ4 algorithm being tested for this

benchmark is serial, but an adapted parallel version of it does exist. The reason for choosing this algorithm is explained further in the Previous Work section. However, it is a descendant of LZ77 that is highly optimized to compress on the CPU [19]. LZ4's compression is far more complex in how it works than my algorithm and the classic LZ77 algorithm. To learn more about how it works, please refer to the LZ4 portion of the Previous Work section of this document. It has a GitHub repository that is well contributed to and a package maintained by Microsoft [15]. The reason this algorithm has been chosen is that it is a contemporary, lossless, and CPU based compression algorithm that can demonstrate how my parallelization of LZ77 might compete with modern descendants of LZ77.

There are two different modes available for LZ4 compression. The mode that is optimized for compression ratio is selected by passing the flag "9" as a command line argument, and the mode that is optimized for throughput is selected by passing the flag "1". I ran both modes, and decided to focus the comparison on the throughput optimized option. This was chosen due to the fact that the ratio optimized version was unanimously slower and had a better compression ratio for all files except for Kendedy.xlsx. The better compression ratio is obviously achieved at the cost of time like with PLZ. So, it made more sense to compare with the mode similar in focus to my own. The comparison of mode "1" can be seen in Table 19.

Table 19: LZ4 Throughput Comparison

Test Files	LZ4 ET (MB/s)	LZ4 DT (MB/s)	Serial ET / LZ4 ET	Parallel ET / LZ4 ET	Serial DT / LZ4 DT	Parallel DT / LZ4 DT
rgg_n_2_22_s0.egr	347.65	800.02	0.06	3.08	0.18	12.82
cit-Patents.egr	773.81	849.92	0.02	1.45	0.14	44.17
coPapersDBLP.egr	749.01	887.59	0.03	1.32	0.16	22.89
in-2004.egr	802.46	892.46	0.03	0.87	0.18	12.13
as-skitter.egr	428.72	757.09	0.04	2.04	0.16	20.74
2d-2e20.sym.egr	494.63	706.15	0.03	1.65	0.17	41.63

amazon0601.egr	375.64	679.10	0.04	2.14	0.15	41.94
test4.bmp	366.86	660.03	0.04	1.79	0.16	32.25
E.coli	325.80	638.72	0.02	1.69	0.17	17.63
bible.txt	253.92	626.10	0.04	2.59	0.14	15.94
internet.egr	280.54	613.59	0.05	1.74	0.21	10.79
world192.txt	284.68	620.52	0.04	2.06	0.13	7.33
kennedy.xls	380.89	659.46	0.06	1.45	0.28	4.48
Geometric Mean	417.99	715.84	0.04	1.76	0.17	17.82

In Table 19, the serial codes run significantly slower than LZ4. This is not surprising because I was essential testing LZ77 against its highly optimized descendant. What is interesting is that the parallel encoder and decoder comparison to LZ4 has a geometric mean speedup of 1.76 and 17.82, respectively. This is obviously an improvement to the serial LZ4 algorithm mode that focuses on throughput, especially for the decoder throughput comparison. The encoder throughput comparison makes sense when taking into account the compression ratio. Similar to PLZ, there is a clear tradeoff between throughput and ratio. This can be seen in Table 20.

Table 20: LZ4 Compression Ratio Comparison

Test Files	Original Size (MB)	LZ4 Compressed Size (MB)	LZ4 Compression Ratio	Encoder Ratio / LZ4 Ratio
rgg_n_2_22_s0.egr	259.65	151.00	1.72	0.76
cit-Patents.egr	147.25	147.25	1.00	0.78
coPapersDBLP.egr	124.13	25.52	4.86	0.31
in-2004.egr	114.26	24.91	4.59	0.43
as-skitter.egr	95.55	66.23	1.44	0.91
2d-2e20.sym.egr	37.72	36.30	1.04	0.89
amazon0601.egr	21.16	17.20	1.23	0.90
test4.bmp	9.36	6.43	1.46	0.78
E.coli	4.64	2.60	1.79	0.95
bible.txt	4.05	1.98	2.05	0.62
internet.egr	3.60	2.18	1.65	1.10
world192.txt	2.47	1.23	2.02	0.55
kennedy.xls	1.03	0.37	2.75	1.27
Geometric Mean			1.87	0.74

It is clear that the improvement to throughput, which my encoder shows in this dataset, is balanced by LZ4's better compression ratio. The column **Encoder Ratio / LZ4**

Ratio shows this. Only two of the files have a better compression ratio when comparing my encoder to LZ4's, and they are among the smaller files tested. What this further illustrates is the costs associated with optimizing for throughput or for compression ratios. The LZ4 mode tested was the mode optimized for throughput, and a parallelized version of its predecessor (i.e., my encoder) outpaced that mode's throughput. Even with the better ratios, this is a good example of how an algorithm that is exceptionally better in serial can be outpaced by a simpler solution in parallel. This does not prove that LZ4 would not outmatch my algorithm when parallelized, but it does show how parallelism is invaluable when it comes to time-based optimizations.

VII. SUMMARY AND CONCLUSION

My work started by creating a serial version of LZ77 and included some byte saving techniques to improve the compression ratio. By preventing single byte matches and adding a storage condition to unmatched triples, I was able to improve the compression ratio. I then converted the serial codes into massively parallelized versions of the same encoding and decoding scheme. This created a foundation to compare the serial throughput against the parallel throughput. The parallel throughput was much improved due to how the input bytes were split among threads on the GPU, processed in phases on the GPU to limit the need for data synchronization, and by using techniques from the literature to prevent data races. For the parallel encoder, I divided the bytes to be individually processed by threads, marked each overlapping location of match and unmatched triples, reduced the overlapping triple locations to a non-overlapping set that covered the entire input, used a prefix sum to determine the number of compressed triples, and then created said triples in parallel. For the decoder, I divided the triples to be independently processed on GPU threads, calculated an inclusive prefix sum using the provided triple data, populated a parent array with a padding of 256 bytes, and performed a union-find operation on the parent array to restore the original input values. Each phase runs in parallel.

This approach to massively parallel compression and decompression made the difficult task of parallelizing LZ77's dependencies feasible. By utilizing storage saving techniques that resemble LZSS', I was able to improve upon the compression ratio and approximately match more contemporary algorithms like CULZSS in compression ratio. This was done with a faster throughput than CULZSS. That already shows that there is

merit in my approach. Although both LZ4 and especially PLZ provide better compression ratios, my algorithm's throughput outpaced theirs consistently. When looking at the parallel decoder throughput alone, it always produced faster throughputs than both LZ4 and CULZSS, and it exceeded LZ4 in geometric mean by nearly 18 times.

In conclusion, the goal of massively parallelizing LZ77 on the GPU was met, and the results show that my parallelization approach could be a better option than some contemporary algorithms if the use case emphasizes throughput over compression ratio. There is room to improve upon the compression ratio, but it is unclear whether those optimizations would be at the cost of throughput. Regardless, the benefits of this massively parallelized approach used for both the decoder and encoder are clear.

REFERENCES

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," in IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, May 1977, doi: 10.1109/TIT.1977.1055714.

[2] James A. Storer and Thomas G. Szymanski. 1982. *Data compression via textual substitution*. J. ACM 29, 4 (Oct. 1982), 928–951. https://doi-org.libproxy.txstate.edu/10.1145/322344.322346

[3] Y. Collet, "LZ4/lz4_block_format.MD at dev • LZ4/LZ4," *GitHub*, 02-Feb-2022. [Online]. Available: https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md. [Accessed: 27-Jul-2022].

[4] P. Deutsch, "RFC 1951 - DEFLATE compressed data format specification version 1.3," *Document search and retrieval page*, May-1966. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1951. [Accessed: 27-Jun-2022].

[5] P. Deutsch, "RFC 1952 - gzip file format specification version 4.3," *Document search and retrieval page*, May-1966. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1952. [Accessed: 26-Jun-2022].

[6] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman and K. A. Ross, "Massively-Parallel Lossless Data Decompression," 2016 45th International Conference on Parallel Processing (ICPP), 2016, pp. 242-247, doi: 10.1109/ICPP.2016.35.

[7] K. K. Yong, M. W. Chua and W. K. Ho, "CUDA lossless data compression algorithms: A comparative study," 2016 IEEE Conference on Open Systems (ICOS), 2016, pp. 7-12, doi: 10.1109/ICOS.2016.7881980.

[8] M. Dipperstein, “LZSS (LZ77) Discussion and Implementation,” *Index O'Stuff*, 11-Mar-2015. [Online]. Available: http://michael.dipperstein.com/lzss/index.html. [Accessed: 14-Nov-2021].

[9] J. Shun and F. Zhao, "Practical Parallel Lempel-Ziv Factorization," 2013 Data Compression Conference, 2013, pp. 123-132, doi: 10.1109/DCC.2013.20.

[10] L. Yang, Y. Yang, G. B. Mgaya, B. Zhang, L. Chen and H. Liu, "Novel Fast Networking Approaches Mining Underlying Structures From Investment Big Data," in IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 51, no. 10, pp. 6319-6329, Oct. 2021, doi: 10.1109/TSMC.2019.2961378.

[11] J. Abel, “Data Compression Corpora,” *The Data Compression Resource on the Internet*, 2002. [Online]. Available: http://data-compression.info/Corpora/. [Accessed: 27-Jun-2022].

[12] Jayadharini Jaiganesh and Martin Burtscher. 2018. A high-performance connected components implementation for GPUs. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18). Association for Computing Machinery, New York, NY, USA, 92–104. https://doi-org.libproxy.txstate.edu/10.1145/3208040.3208041

[13] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software 38, 1, Article 1 (December 2011), 25 pages. DOI: https://doi.org/10.1145/2049662.2049663

[14] A. Ozsoy and M. Swany, "CULZSS: LZSS Lossless Data Compression on CUDA," 2011 IEEE International Conference on Cluster Computing, 2011, pp. 403-411, doi: 10.1109/CLUSTER.2011.52.

[15] Y. Collet, “LZ4/LZ4: Extremely fast compression algorithm,” *GitHub*. [Online]. Available: https://github.com/lz4/lz4. [Accessed: 26-Jun-2022].

[16] NVIDIA Corporation, “cub::DeviceScan Struct Reference,” *Cub: Cub::devicescan struct reference*. [Online]. Available: https://nvlabs.github.io/cub/structcub_1_1_device_scan.html. [Accessed: 26-Jun-2022].

[17] A. Ozsoy, “adnanozsoy / CUDA_Compression,” *GitHub*, 2011. [Online]. Available: https://github.com/adnanozsoy/CUDA_Compression. [Accessed: 27-Jun-2022].

[18] C. Spackman, “Compression/decompression tradeoffs for data networking and storage,” *EETimes*, 09-May-2007. [Online]. Available: https://www.eetimes.com/compression-decompression-tradeoffs-for-data-networking-and-storage/. [Accessed: 26-Jun-2022].

[19] T. Matsuoka , *LZ4*. [Online]. Available: https://lz4.github.io/lz4/. [Accessed: 26-Jun-2022].