15-418 Project Report: LZ Compression

Jack Woodson (jrwoodso), Alex Li (alexli3)

April 30, 2026

URL: https://jrwoodso.github.io/15-418-LZ_Compression/

Summary

We implemented a parallel versions of the LZW (Lempel–Ziv–Welch) and LZ77 (Lempel-Ziv 1977) compression and decompression algorithm using multithreading on a CPU, as well as LZW on GPU. For LZW, Our system can multiple files concurrently or use chunk-based parallelism to accelerate compression while preserving correctness and compression ratio; the GPU-based version significantly accelerates image compression. We evaluated performance on both the GHC and PSC clusters and demonstrate significant speedups over a single-threaded baseline. For LZ77, we looked at both a hybrid CPU+GPU based approach, as well as a pure CUDA GPU approach.

Background of LZ Compression

LZ-family compression algorithms form the foundation of widely-used formats like ZIP, PNG, and GIF. These algorithms take advantage of data repetition. Rather than storing repeated sequences in full, they encode references to prior occurrences, achieving significant size reduction. We will focus on LZW (1984) and LZ77 (1977).

Key Data Structures and Operations

LZW. The central structure is a dynamic dictionary D (hash table) mapping strings to codes. Every input symbol triggers a **lookup** of $(w + c)$, an **insert** on a miss, and an **emit** of $D[w]$. A well-performing dictionary directly determines compression throughput.

LZ77. Two fixed-size buffers are maintained: a *search window* (W bytes of prior input) and a *lookahead buffer* (L bytes ahead of the current position). The sole operation is longest-prefix match: find offset o and length ℓ maximizing agreement between the lookahead prefix and a window substring, naively $O(W \cdot L)$ per position.

Here is pseudocode for both algorithms.

Algorithm 1 LZW Compression

Require: Input sequence S
Ensure: Sequence of output codes
Initialize dictionary D with all single-symbol strings
$w \leftarrow S[0]$
for $i \leftarrow 1$ to $|S| - 1$ **do**
    $c \leftarrow S[i]$
    if $(w + c) \in D$ **then**
        $w \leftarrow (w + c)$
    else
        output $outputD[w]$
        add $(w + c)$ to D
        $w \leftarrow c$
    end if
end for
output $D[w]$

Algorithm 2 LZ77 Compression

Require: Input sequence S, window size W, lookahead buffer size L
Ensure: Sequence of output triples $(\text{offset}, \text{length}, \text{next symbol})$
$i \leftarrow 0$
while $i < |S|$ **do**
    search window $\leftarrow S[\max(0, i - W) \dots i - 1]$
    lookahead buffer $\leftarrow S[i \dots \min(i + L - 1, |S| - 1)]$
    find longest match of a prefix of lookahead buffer in search window
    $(\text{offset}, \text{length}) \leftarrow$ position and length of match
    if no match found **then**
        output $(0, 0, S[i])$
        $i \leftarrow i + 1$
    else
        $c \leftarrow S[i + \text{length}]$ ▷ next symbol after match
        output $(\text{offset}, \text{length}, c)$
        $i \leftarrow i + \text{length} + 1$
    end if
end while

Computational Bottlenecks and Parallelism

- **LZW bottleneck:** Dictionary lookup and insertion, once per symbol. At high fill rates (2^{12}–2^{16} entries), hash collision chains lengthen and cache misses dominate. The dictionary far exceeds L1/L2 capacity, and data-dependent key changes between steps defeat prefetching.
- **LZ77 bottleneck:** Longest-prefix match search, $O(W \cdot L)$ per position in the worst case. The search window (typically 32 KB) fits in L2/L3 cache, making LZ77 significantly more cache-friendly than LZW.

Dependencies. LZW has a true data-dependency chain of length $|S|$: the next string w_{i+1} depends on whether $(w_i + c_i) \in D$, which depends on all prior insertions. This means that the algorithm is inherently sequential on any input data. In contrast, LZ77 has a read-after-write dependency on the window buffer (position i reads $S[i - W \dots i - 1]$, written by prior steps) and a data-dependent stride $\ell + 1$ between successive output positions.

Parallelism. Both algorithms can use coarse-grained *chunk-level parallelism*: partition input into P chunks, compress independently in parallel, and concatenate outputs (Figure 1). The tradeoff is a cold-start penalty at boundaries. Each chunk rebuilds its dictionary or window from scratch, which

can reduce the compression ratio. The parallel LZW approaches we investigated also combined this was folder-level parallelism, as well as GPU parallelism on images.

LZ77 has some fine-grained intra-position parallelism. Comparisons against distinct candidate offsets are fully independent and map naturally to SIMD byte comparisons, though variable match lengths create irregular loop trip counts that reduce lane utilization. LZW's data-dependent branching and variable-length keys make intra-stream SIMD infeasible.

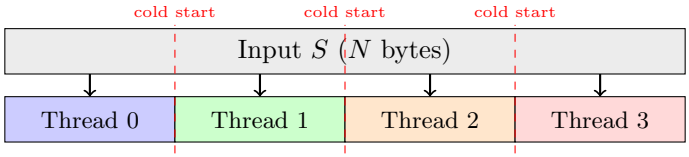


Figure 1: Chunk-level parallelism. Each thread independently compresses a partition, incurring a cold-start penalty at boundaries.

Both algorithms are generally memory-bound with irregular control flow. Data locality favors LZ77, whose fixed-size window stays warm in cache; LZW's unbounded dictionary growth causes increasing cache pressure and TLB misses over time.

LZW Approach

For LZW, we explored a wide variety of hardware and software. On the CPU side, we tested targeted both the GHC machines as well as nodes in the PSC cluster. These solutions used OpenMP for parallelism, as well as mmap for I/O operations, as well as custom BitReader/BitWriter classes for compaction. On the GPU side, we targeted the RTX 2080 GPUs in the GHC cluster, using CUDA and Thrust to generate fast kernel code.

For CPU-based approaches, there were two mapping methods that we found to be fastest through testing. The first was a data-parallel approach using an adjustable BLOCK-SIZE, each with a private hash table, and with worker threads getting dynamic work assignment to deal with imbalance. An additional focus was memory efficiency, where we tried to reduce overall size so data structures could fit in caches, as well as number of costly operations, like using an epoch counter to avoid clearing memory blocks. The second was a file-based data-parallel approach, where if the target was a directory, the program would parallelize the compression of multiple files at once, as well as continuing to use blocking if a single file was large enough.

For GPU-based approaches, we transposed the raw input image so that each row was compressed independently; each row mapped to one CUDA thread on hardware. Hash tables and prefix/suffix arrays were allocated in global memory, with one copy per strip, but dictionaries were in private memory since threads work independently. The output buffers were also one-per-strip which were concatenated for output. And like with the CPU-based approach, the kernel used a timestamp to avoid clearing the whole table every iteration.

As for changes made from the serial implementations to support parallelism, the largest change was resetting the dictionary every block/chunk/row to reduce the inherent dependencies in the LZW algorithm. This slightly reduces compression but allows for massive parallelism. Further, as mentioned above, timestamped versions reduced clearing memory. Finally, parallel threads directly wrote to mmap'ed file to eliminate concatenation and purely memory-constrained steps.

Iteration process: During the process of parallelizing LZW, we tried many approaches. We originally thought about focusing on some smaller files, like the bible, but even our serial code was already super fast and parallel overheads were dominating. We tried blocking on small blocks which caused poor compression and runtime performance. We also tried pipelining (inspired by parallel databases) where one thread would sequentially read the input and maintain the dictionary and pass results to a queue, and a consumer thread would pull from the queue and focus on bitpacking and file output. This failed. The results were not really conclusively better.

At this point, we decided to pivot to large files first. We found a commonly used benchmark, a 1GB dump of Wikipedia data, coined enwik9 [8]. This allowed us to use fairly large blocks so compression ratio could stay high, but also have many blocks to keep dynamically assign to workers using OpenMP. We also added a degree of adaptive block-sizing. We tested up to 128 cores on the PSC, since we wanted to explore the maximum on single CPU/GPU systems.

After performance testing, we added file-level parallelism. Our thinking at this point was the largest time penalties in compression were: 1. A large file, or 2. Many files. Since we had just tacked (1), we were interested in (2). Using the commonly tested Canterbury Corpus dataset [9], which included a mix of data from books to code, we focused on dynamically and concurrently assigning files to worker threads, and continuing to use the chunked approach for larger files, so that we would not encounter the straggler problem in the case of wildly varying file sizes.

Finally, with about a week left, we wanted to explore Funasaka et al. 's paper that used GPUs to speed up image compression [7]. This was quite difficult, since we had to turn a pretty novel approach described in a paper into functional code. Specifically, debugging was quite challenging. It also had some unique elements, like transposing the input and the consideration of what memory to use.

Starting code: The base implementations of LZW were written using the algorithm outline on Wikipedia, along with understanding basic optimizations made in the UNIX compress utility. The parallel versions were written from scratch after determining that data-parallelism was generally considered the only viable way to really scale up LZW compression. Finally, the GPU-based approach was based on the methods described by Funasaka et al. (2015) [7].

LZW Results

The main benchmark used was runtime/speedup compared to single-threaded. In the case of the GPU-based LZW, we compared to CPU performance. In certain cases with high overheads (like OpenMP), we also looked at purely the algorithmic speedups, like we had done in Asst3 and Asst4. A secondary benchmark was compression ratio, measured by $\frac{\text{Compressed Size}}{\text{Input Size}}$.

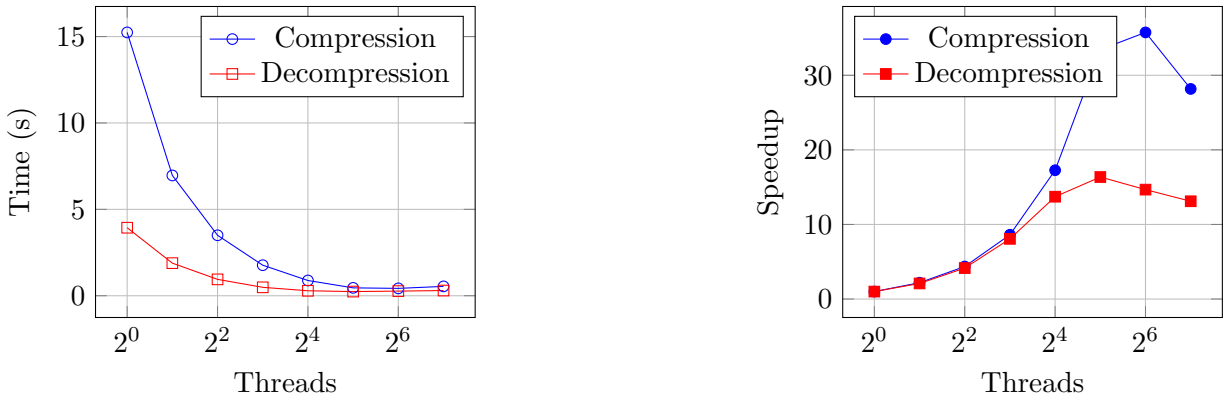
Important to note, we also did work on compression on the CPU side. However, it is a much simpler parallel problem, so we were able to achieve great speedup, until we became constrained by the SSD write speed. Since the output grows compared to the input, our decompression was efficient enough to produce files that began to be constrained by output speed. We included decompression on the block-based LZW on PSC data below in 2.

For experimental setups, see 1.

LZW Version	Hardware	Testing Data
LZW Block Parallel	3.0 GHz Intel Core i7-9700 (8 core), AMD EPYC 7742 CPUs (64 cores, 128 threads)	Enwik9
LZW File Parallel	3.0 GHz Intel Core i7-9700 (8 core)	Canterbury Corpus + Enwik9
GPU LZW	RTX 2080 on GHC	Custom raw images (4096×3072)

Table 1: Experimental Setups

In 2, we ran the block-based LZW code on the 1GB Enwik9 dataset on the PSC machines. We focused on scaling compared to execution with 1 core, which achieves performance in similar to that of UNIX compress. Throughput reached 2.35 GB/s on compression, and 4.16 GB/s on decompression.

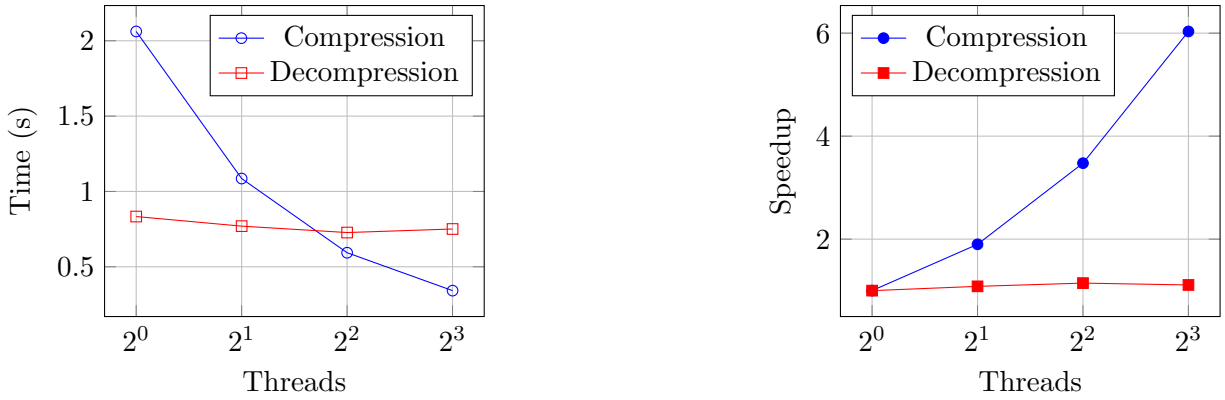


(a) Execution time vs. threads

(b) Speedup vs. threads

Figure 2: Parallel compression and decompression with LZW Blocks on PSC (enwik9).

Further, in 3, we also tested the file-parallel LZW, which we ran on the Canterbury Corpus, augmented with half of the Enwik9 dataset. We were constrained with the AFS limits, but the Enwik9 portion was still substantially larger than the rest of the Corpus, so this still adequately tested the straggler issue. Again, we addressed this by also using the chunked approach when encountering large files in the folder. For this method, we were exclusively focused on compression speedup.



(a) Execution time vs. threads

(b) Speedup vs. threads

Figure 3: Parallel compression and decompression (compression ratio = 0.4484).

Finally, for the GPU-based LZW compression, we tested on several 4096×3072 resolution, raw, grayscale images. Specifically, we chose these images, codenamed 'earth', 'nature', and 'chart'. **??** contains these photos before grayscaling. The capture a mix of synthetic render, real photography, and productivity/work.

Figure 4: Selected images: earth, nature, and chart.

Here is a bar graph (Figure 5 Left) comparing runtimes between 2 CPU configurations (1 core, 8 core on GHC) and the RTX 2080's performance. For better clarity, we use compression ratio', measured by $\frac{\text{Input Size}}{\text{Compressed Size}}$ (note it is the inverse of the measurement used elsewhere in this report).

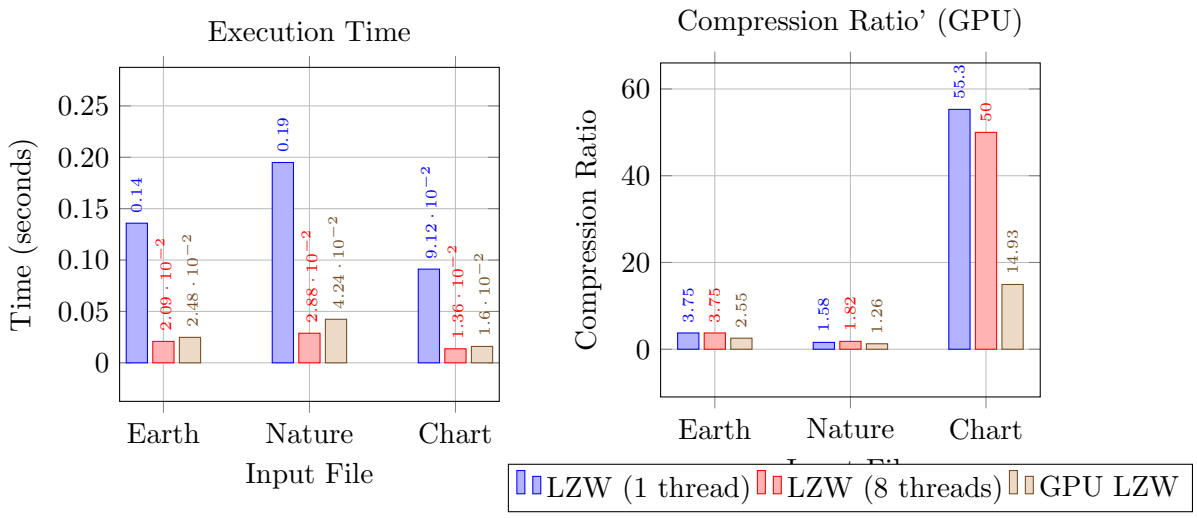


Figure 5: Performance comparison: Execution time (left) and compression ratio' (right)

Finally, we compare the compression ratios (Figure 5 right). The GPU-based approach generally lags behind in compression ratio, though by a small amount. For the chart (3 cluster), the parallel LZW on CPU is really able to take advantage of aggressively compressing the whilespace, while the GPU's row-based approach limits the compression ratio. One option we explored was decreasing the frequency which we cleared the dictionary, but this led to large performance penalties which we will discuss later.

As for the problem size, it is definitely important to see how our code performs across input types. However, we can restrict the scope in testing somewhat, since we have specialized and tailored versions for large files, folders/directories, and images.

Limitations and Deeper Analysis

LZW-Block and LZW-File Block:

Both methods share the same underlying LZW compression and decompression algorithms and data structures.

To validate our hypothesis, we ran perf stat on the single-threaded compression of enwik9. The results are conclusive: the CPU achieved an IPC of only 0.59 — well below the 2–4 IPC typical of compute-bound workloads — indicating the pipeline spends more time stalled on memory than executing instructions. We observed 25M LLC load misses (2.60% miss rate), which at 100ns DRAM latency per miss accounts for approximately 2.5 seconds of accumulated stall time, consistent with our observed runtime. Critically, actual DRAM bandwidth consumed was approximately 1 GB/s — less than 1% of the EPYC 7742's 204 GB/s theoretical peak — definitively ruling out bandwidth saturation. The bottleneck is therefore memory access latency from irregular hash table lookups that thrash the LLC, not bus throughput. At high thread counts, each additional thread adds its own private hash table working set, further stressing LLC capacity and increasing miss rates, explaining the plateau and regression beyond 32 cores.

LZW-File-Block:

Compression scales to around 6× at 8 threads, which is reasonable but not ideal. The stragglers are the likely culprit in comparison to the single file blocking. The Enwik9 chunk in the combined corpus dominates runtime, so even with dynamic and blocked assignment, the last thread holding that chunk serializes the tail.

GPU-LZW:

The GPU is actually slower than 8 CPU threads on all three images (e.g., Nature: 0.042s GPU vs. 0.029s 8-thread CPU). The bottleneck here is insufficient parallelism per row combined with PCIe transfer overhead. Each row of a 4096×3072 image maps to one CUDA thread, giving you 3072 threads. The work per thread is high-latency and serial-dependent (each symbol depends on the previous), which is not ideal for GPU execution. Global memory hash table accesses are the inner-loop bottleneck, with one hash table per strip in global memory and no spatial locality between successive accesses, we hit high-latency global memory on nearly every symbol. Additionally, we tried a blocked approach on GPU too, but even changing from 1 row per dictionary refresh to 2 rows resulted in doubling the runtime. For earth, nature, and graph, the runtimes (ms) were 20.28, 26.89, and 8.85, approximately 135%, 58%, and 85% slower, making the GPUs now much slower than 8-core CPU. This was likely due to increased memory requirements for the dictionaries, and resulting in bad memory patterns, hash conflicts, and even worse spatial locality.

Runtime Analysis

For compression, we analyzed the LZW-blocked approach.

Table 2: Cycle breakdown of `compressLZW_Block` (30.24B cycles total)

Region	**Cycles**	**%**
Epoch load stall	15,784,742,540	52.2
Linear probe loop	5,445,274,079	18.0
Miss: bit output	3,165,424,881	10.5
Hit: val read	3,018,906,966	10.0
Key hash + prefetch	2,609,173,660	8.6
Dict insert	216,262,987	0.7
Total	**30,239,785,113**	**100.0**

The profiling tells a very clear story. Roughly 80% of all cycles are spent in just three consecutive operations that represent a single conceptual action — looking up one entry in the hash table: Epoch load stall (52%) is the single largest bottleneck. The instruction at 0x35ec accounts for more cycles than everything else combined. This is a nearly-pure DRAM latency stall: ht_epochs is a separate uint32_t[] array of 131,072 entries (512 KB), and on each new symbol, the first probe hits a cold cache line.

Linear probe loop (18%) compounds the above. Every collision resolves by stepping to the next bucket — which is a different cache line again. In the combined bytecode, the jne 3600 back-edge at $>$ 1B samples means the average probe depth is substantial. This will only worsen at high load factors. Val read on hit (10%) is the final miss in the chain. Even after successfully finding the entry through epoch + key comparison, ht_vals lives in a third separate array at a third independent memory address. Three pointer chases for one logical record. Even after the many optimizations we made, more changes in BitWriter likely could further speed things up. Perhaps pulling BitWriter

fields into local variables at the top of the loop would help the compiler keep them register-resident instead of reloading through on every iteration.

GPU-LZW: For the GPU-LZW, we used NVidia Nsight while we worked on the solution, as well as at the end, to understand a performance breakdown of the code. We chose to benchmark earth since it is a mix between the extremes of graph and nature. There are 3 distinct phases in the runtime: Initialization, Main Algorithm, and Output+Cleanup.

In the Main algorithm, there are 3 key kernels, dominated by the heaviest part of the algorithm. 1. lzw_compress_strip, taking 8.92ms. (91%). 2. concatenate_kernel, taking 0.82ms. (8.3%). 3. transpose_kernel, taking 0.09 ms. ($< 1\%$).

In terms of memory management, which happens during Initialization and writing output, Nsight says a figure of around 100 ms. This *dominates* the runtime of the algorithm. Thus, we note that in a single-file compression test, it would appear that CPU is a much more logical choice. However, there are workflows where if the data is produced on GPU, then compressing there before copying to system memory could be much faster, and reduces the bottleneck of the GPU-CPU transfer bandwidth. Also, we hypothesize that much of this latency can be hidden in a pipeline, where many images are continuously being compressed. This would be room for further improvement and investigation.

Machine Choices

Luckily, we explored both approaches (though with more time, we could have gone into further depth). For large text files (enwik9, Canterbury Corpus): yes, CPU was the right call. Text LZW benefits heavily from large block sizes that preserve cross-block dictionary patterns, and our 32-core PSC results show exceptionally strong scaling. We think a GPU would struggle here because there's no natural 2D structure to exploit, and we'd have to either use very short rows (poor compression) or very few threads (poor parallelism).

For image compression: the GPU choice was reasonable but ultimately not a win. Our data shows 8 CPU threads matching or beating the RTX 2080 on all three images. The fundamental issue is that LZW is a serial, pointer-chasing algorithm, and each output code depends on all prior codes in the current block. And when we tried to use larger blocks in the GPU, execution times sky rocked (as analyzed above). We know that GPUs excel at data-parallel workloads with independent, regular memory access patterns. Still, for a use case with many small images processed in a batch (thousands of images), GPU parallelism across images (not rows) could be compelling. The Funasaka et al. approach we implemented is essentially row-level SIMD.

LZ77 Approach

Similar to our LZW implementation, we tested a wide range of hardware and software. For LZ77, we started on the GHC machines, making use of the 8 cores with a data chunking approach, assigning threads to chunks of data, allowing each thread to perform sliding windows on the data.[1] This was a very naive approach and led to a loss of compression around the boundaries of each data chunk. As expected, this implementation scaled well but had sub-optimal runtime and compression ratios. A huge bottleneck at the time was the time spent looking for matches within each window.

This led to the idea of implementing LZ77 using CUDA to identify matches within a window. Additionally, the cheap cost of thread creation, in addition to the significant increase in the number of threads accessible to us, seemed like a great solution to the problems we identified early on with

the very sequential data-dependent LZ77 algorithm. Especially in the context of data compression and working with massive datasets, using data parallelism rather than task parallelism seemed the most intuitive approach.

This new implementation was done in CUDA with GHC RTX 2080. An immediate issue that was discovered upon running this on our different datasets was the memory divergence. To fix this issue we used share memory tiling. By allowing the first few threads to load the window size into local memory we resolved this issue. Which led to the next problem that we were concerned about when working with CUDA threads was warp divergence. To tackle this problem we implemented CUDA's anysync allowing for the stopping of work if no productive work was being done on a warp thread.

Now that our GPU-focused LZ77 implementation began performing well in terms of runtime, it was time for us to turn our attention to the compression ratio. At the time, we were using a list of nodes with a byte size of 5 to hold our length, offset, and string information. This yielded beneficial data compression on highly repetitive datasets but sometimes resulted in a compressed file larger than the original,, as our implementation would encode single characters with no matches or matches smaller than 5 bytes into this list. To solve this issue, we used bit packing, as seen in the LZSS flagging.[3] This resulted in a significant improvement in our compression ratio.

An identified bottleneck during this time was the single CPU used by the CUDA threads. A possible solution that we came up with was to create a hybrid approach, making use of multiple cores on the CPU as well as the GPU's threads.[4] Unfortunately, we weren't able to achieve a working solution that yielded any promising results. Our runtime was faster yet the compression ratio seriously suffered in a way that it didn't make sense to continue down this road with the time that we had left with the project. Instead, we turned to a different solution, which was Huffman coding[5]. This was our final implementation.

LZ77 Results

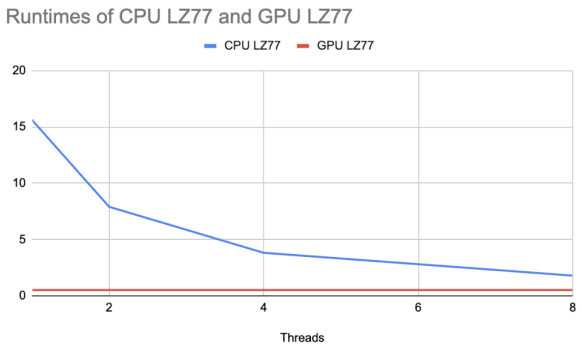
The analysis of LZ77 will be slightly different from the LZW algorithm above as we computed LZW on a CPU and its cores while LZ77 was primarily GPU based. We will still compare the runtimes of the previous CPU implementation of LZ77 to our final implementation of LZ77 done on the GHC GPU. In addition to the compression ratio, and throughput of the compression algorithm.

For experimental setup see 3.

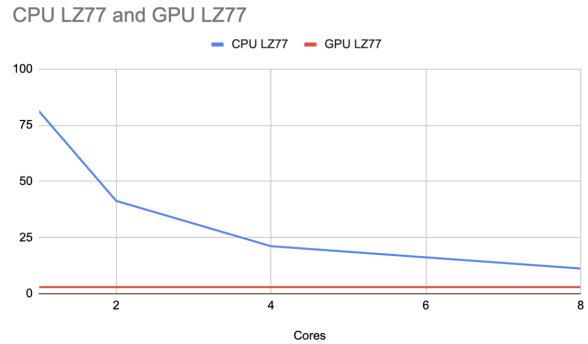
LZ77 Versions	**Hardware**	**Testing Data**
LZ77 Block Parallel	3.0 GHz Intel Core i7-9700 (8 core)	Canterbury Corpus + Enwik9
GPU LZ77	RTX 2080 on GHC	Canterbury Corpus + Enwik9

Table 3: Experimental Setups

In the figures below we ran the LZ77 CPU implementation alongside the LZ77 GPU implementation to compare their results.



In the graph above is the comparison between the CPU and GPU based implementations of LZ77 on the Canterbury Corpus. It should also be noted that the average compression of the CPU LZ77 algorithm averaged around 0.5 while the GPU implementation averaged around 0.35 compression ratio on the Canterbury Corpus. In which the GPU implementation performed better on both total runtime as well as compression ratio. Additionally the total file size of the 2.68 MB.



The graph above is the comparison between the two different implementations of LZ77 that we have on the enwik9 dataset (1 GB). Once again the GPU implementation significantly outperformed the CPU version. The average compression ratio of CPU LZ77 was 0.74 while the GPU compression ratio was 0.42.
The next two graphs focus on the GPU implementation of the LZ77 algorithm, where we look at the compression ratio when we change the main unique feature of the LZ77 the window size.

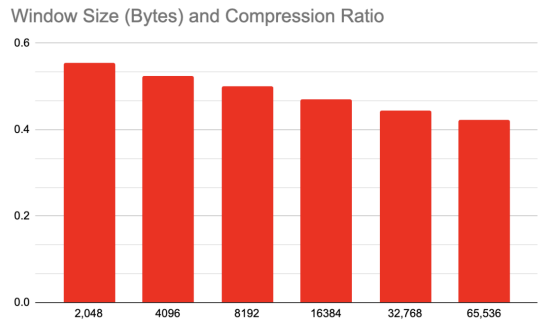


Figure 6: Compression Ratio vs Dataset

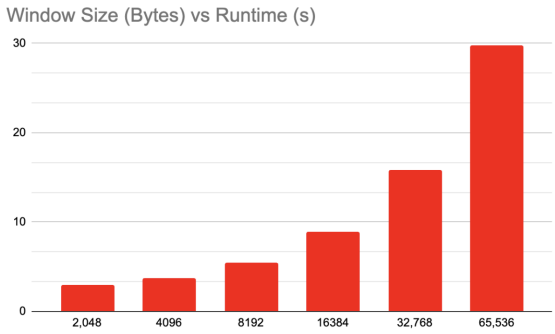
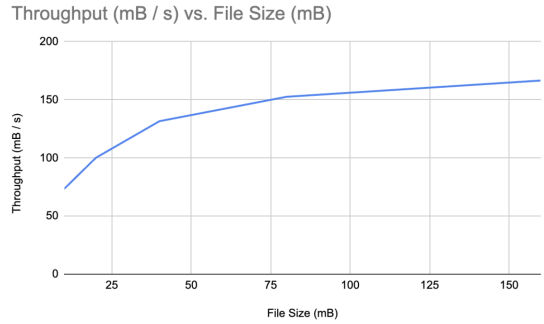


Figure 7: Effect of Window Size on Ratio

There is a slight decrease in compression ratio when the window size is adjusted on LZ77; however, this decrease in compression ratio is not worth it as you look to the graph on the right noticing the

exponential increase in runtime.



Of course a project on compression algorithms would not be complete without a graph on the throughout $FileSize/runtime$. To be able to gain meaningful results to see how the LZ77 algorithms' throughput we ran the code on different subset size of the enwik9 dataset. The compression ratio is logarithmic as the file size increases. In which we saw a throughput of 170 mb/s.

The problem size is very important when we're looking at compression algorithms. We tried our best to be able to test the different sizes as seen on the throughput graph above. However, as we were testing the performance of these algorithms on the GHC machines, there were limitations to exactly how big our input sizes could be.

Limitations and Deeper Analysis

The biggest limitation in this implementation is that the Load/Store unit pipeline is being over utilized by transferring memory from global to local. In which the threads are being stalled as their utilization is only roughly 60%. Memory access was where majority of the sliding windows spent its time roughly 60%. This statistic makes sense in regards to a compression algorithm accessing the data and then compressing into a smaller form. 30% of the time the algorithm spent stalled waiting for memory. 10% of the time was arithmetic computation. These metrics were computed from running Nsight Compute on the LZ77 implementation. Clearly, the next steps of improvement on this implementation of LZ77 would be to reduce the utilization of the Load/Store Unit pipeline. A possible way to fix this issue would be to send vectorized loads over the pipeline. Additionally, since the threads were stalled for a point of the algorithm would be to find different ways to utilize them in the compression of a file.

Machine Choice Yes, utilizing the GPU for the LZ77 implementation was sound. The LZ77 is a very general algorithm and there are dozens of variations of the LZ sliding window algorithms that are both CPU and GPU focused. In terms of our project since we chose to do LZW in CPU we decided to for more meaningful results in the comparison betwene the two would be to have LZ77 be performed on the GPU.

Division of Work

Alex worked on the LZW portions, while Jack worked on the LZ77 approaches. Of course, there were frequent points where we met and discussed issues, and brainstormed solutions. The reports and website were also written together. We think a 50-50 split in work is a fairly resonable assumption.

Acknowledgements

We would like to thank Professors Mowry and Railing for support during the semester, as well as Professor Mowry's suggestions and feedback during our initial and milestone meetings.

References

[1] [Unknown], "[LZ77 Compression Algorithm]," *[Microsoft]*, [2026]. Available: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb

[2] [J. Shun and F. Zhao], "[Practical Parallel Lempel-Ziv Factorization]," *[2013 Data Compression Conference]*, [2013]. Available: https://ieeexplore.ieee.org/document/6543048

[3] [C. M. Stein, D. Griebler, M. Danelutto and L. G. Fernande], "[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]*, [2019]. Available: https://ieeexplore.ieee.org/document/8671624

[4] [Yue Zhang, Yuki Hara, Oliver Sinnen, Shinichi Yamagiwa], "[Parallelising Stream-Based Lossless Data Compression on GPUs and CPUs]," *[Journal or Conference Name]*, [2026]. Available:https://dl.acm.org/doi/10.1145/3784828.3784829

[5] [Hiya Mavani], "[The Internet's Shrink Ray: Data Compression with Huffman, LZ77, and Deflate]," *[Rutgers USACS]*, [2026]. Available:https://medium.com/@rutgersusacs/the-internets-shrink-ray-data-compression-with-huffman-lz77-and-deflate-04ab37f01819

[6] [M. Safieh and J. Freudenberger], "[Address space partitioning for the parallel dictionary LZW data compression algorithm]," *[2019 16th Canadian Workshop on Information Theory]*, [2019]. Available: https://ieeexplore.ieee.org/document/8929928

[7] S. Funasaka, K. Nakano, and Y. Ito, "A High-Speed LZW Compression Algorithm using GPUs," *Hiroshima University*, 2015. Available: https://www.cs.hiroshima-u.ac.jp/cs/_media/cp14.pdf

[8] M. Mahoney, "Large Text Compression Benchmark," 2006. Available: http://mattmahoney.net/dc/text.html

[9] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proceedings DCC '97. Data Compression Conference*, 1997, pp. 201–210. Available: https://ieeexplore.ieee.org/document/582019