FPGA-Accelerated Zip Compression

CSE 4602 Capstone Design Project Final Report
Submitted to Professor Hall and the Department of Computer Science & Engineering

Engineers:
Elias Dahl[1] - d.elias@wustl.edu
Sivan Elisha[2] - e.sivan@wustl.edu
John Iwenofu[3] - c.iwenofu@wustl.edu

Advisor:
Michael Hall[4] - mhall24@wustl.edu

May 1, 2026

Project Duration:
January 12, 2026 - May 1, 2026

[1] Candidate for B.S. in Computer Engineering with minor in Electrical Engineering in Department of Computer Science and Engineering and Department of Electrical and Systems Engineering

[2] Candidate for B.S. in Computer Engineering in Department of Computer Science and Engineering and Department of Arts and Sciences

[3] Candidate for B.S. in Computer Engineering with secondary major in Computer Science in Department of Computer Science and Engineering

[4] Lecturer, Department of Computer Science and Engineering

Abstract

This capstone project focuses on the design and implementation of an FPGA-accelerated ZIP file compression system. Developed on the PYNQ-Z2 platform, the system offloads compression workloads from the microprocessor to dedicated hardware to improve performance and scalability. The hardware module utilizes the DEFLATE algorithm, while user data is transmitted via a WebSocket interface between the client and server. Incoming files are divided into fixed-size chunks and continuously streamed to the FPGA for processing, with the compressed chunks immediately returned to the client.

This architecture directly addresses the high computational and financial costs associated with relying solely on software-based compression across multiple physical servers. By accelerating the process through a cost-effective hardware module, the system reduces server dependency while achieving performance comparable to standard software tools. Testing with user files ranging from 1 MB to 20 MB demonstrated a substantial reduction in file size, alongside a highly scalable throughput where upload and download speeds scaled linearly with file size.

Introduction

Since the Internet's inception, data processing and sharing between servers and devices have grown exponentially. As society becomes increasingly reliant on digital technology, the volume of data generated has surged. This has created a critical need for infrastructure capable of handling this rapid expansion without compromising performance. The standard method for handling large amounts of data involves compression, which retains the core information within a file while reducing its footprint, making it more affordable and efficient to store and transmit.

The foundation of modern compression is the LZ77 algorithm, developed by Jacob Ziv and Abraham Lempel in 1977[1]. Before then, most compression algorithms, such as Huffman coding, were statistical, requiring the computer to make two time-consuming passes over a file to count character probabilities and then compress. Ziv and Lempel introduced a "Sliding Window" approach containing a search buffer and a look-ahead buffer. By checking if a string of text in the look-ahead buffer had occurred recently in the search buffer, the algorithm could replace repeated text with a simple pointer. This allowed data to be compressed sequentially in a single pass, making the algorithm universal and sequential. Modern standard tools, such as GZIP, utilize the DEFLATE standard, which combines the LZ77 sliding window with a Huffman coder stage to efficiently reduce file sizes.

As data volumes continue to increase over the years, servers are added to address the need of large data handling. However, they face overwhelming demands on disk space and network bandwidth. While software compression tools like GZIP effectively reduce data size, they are computationally expensive. As traffic increases, the CPU cycles required for compression create a significant performance bottleneck. The traditional solution—scaling horizontally by adding

more physical servers—is financially inefficient and fails to address the root computational problem.

To address this server bottleneck, the primary objective of this project is to develop a Hardware Compression Accelerator using an FPGA to speed up file compression. By offloading the compression algorithm to dedicated FPGA hardware pipelines, the main microprocessor is freed to handle other user requests, resulting in a scalable system. However, developing hardware solutions introduces strict physical constraints. Because this project utilizes a TUL PYNQ-Z2 board, the system design is heavily constrained by the board's limited available on-chip memory for the programmable logic.

To achieve the project goals within these physical constraints, the design must meet specific functional and non-functional requirements. A key functional requirement is that the system must feature a user interface allowing clients to upload a file and receive the compressed version in a standard ZIP format, with the microprocessor seamlessly managing data chunks and assembling the output. To fulfill the non-functional requirements, the system must overcome the hardware memory constraint by efficiently dividing incoming files into 512 KB chunks and streaming them continuously over WebSocket without overwhelming the hardware's RAM.

On the hardware side, a custom pipeline will be designed in Vitis HLS to perform the core DEFLATE compression. On the software side, a Direct Memory Access (DMA) interface utilizing the AXI-Stream protocol will be constructed to enable CPU-free data transfer between the microprocessor and FPGA. Finally, we will benchmark throughput and latency against software compression to quantify the accelerator's performance and prove its feasibility as a scalable solution. We also added a more comprehensive benchmark that measures the data analytics for our data compression implementation as part of our changes from our original proposal. Different files will be tested, such as text files, random binary files, and binary pattern files, ranging from 1 MB up to 20 MB.

System Design & Implementation

System Overview

Our compression system is composed of three main components, a client program running on a user’s PC, a server program running on the PYNQ PS, and a hardware DEFLATE compressor implemented on the PYNQ PL. An overview of this system architecture can be seen in Figure 1 below.

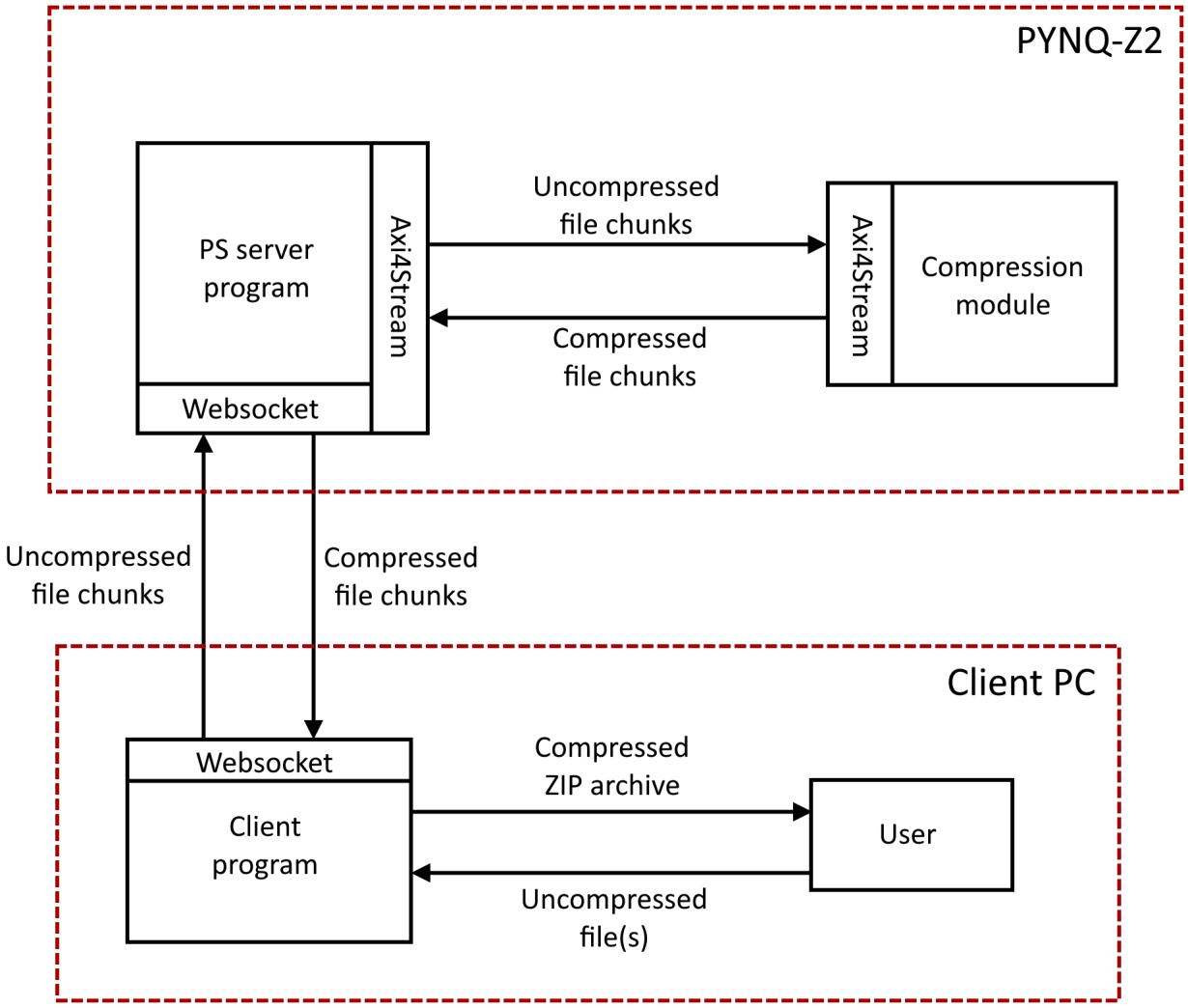

Figure 1: System overview diagram.

As seen in Figure 1, a user can upload uncompressed files to the client program. The client program then sends the files to the server over a WebSocket. Next, the server sends the input data into the compression module, where it is compressed and returned to the server. Finally, the server sends the compressed files back to the client, where they are packed into a ZIP archive and saved on the user's PC. Originally, we intended to send the files in multiple chunks to allow unbounded input file size, but due to DMA issues we were limited to one chunk per input file, which we mitigated by increasing the chunk size to 20 megabytes (MB).

Design Approach

The design of the system was driven by the goal of creating a scalable, end-to-end compression pipeline that could efficiently handle file uploads, offload computation to hardware, and return results in a user-friendly format. To achieve this, several key design decisions were made regarding how data is transmitted, processed, and managed across the system.

Chunking Strategy

One of the primary design decisions was to divide input files into fixed-size chunks before transmission and processing. This approach was motivated by both system scalability and hardware constraints. By breaking files into smaller segments, the system avoids loading entire files into memory, which is particularly important given the limited on-chip memory available on the FPGA platform.

Chunking also enables incremental processing, allowing data to be transmitted and processed continuously rather than waiting for the full file to be received. This improves responsiveness and supports handling larger files in principle. Additionally, chunking simplifies data transfer between system components by providing predictable, bounded units of data.

However, this approach introduces additional complexity. The system must maintain correct ordering of chunks, handle partial chunks at the end of files, and ensure that reconstructed output remains consistent. These requirements increase the complexity of both the communication protocol and the client-side reconstruction logic.

Streaming vs. Buffering

Another key design decision was to adopt a streaming-based architecture rather than a traditional buffering approach. In a buffered system, the entire file would be received and stored before processing begins. In contrast, the streaming approach allows data to be processed as it is received, enabling overlap between communication and computation.

Streaming provides several advantages. It reduces memory usage, as only small portions of the file need to be held in memory at any given time. It also improves system efficiency by allowing different stages of the pipeline (upload, processing, and download) to operate concurrently. This leads to better utilization of both network and hardware resources.

The tradeoff is increased system complexity. Streaming requires careful synchronization between components, as data is continuously in motion. It also complicates error handling and debugging, since failures may occur mid-stream rather than at well-defined boundaries. Additionally, variable output sizes from the compression process require dynamic handling during reconstruction.

WebSockets vs. REST

For communication between the client and server, WebSockets were chosen over a traditional REST-based approach. This decision was primarily driven by the need for real-time, bidirectional communication.

WebSockets provide a persistent connection that allows data to be sent and received continuously without the overhead of repeatedly establishing new connections. This is particularly well-suited for streaming applications, where large amounts of data must be transferred incrementally. The ability to support full-duplex communication also enables the system to simultaneously upload input data and receive processed output, improving overall performance.

In contrast, a REST-based approach would require separate HTTP requests for each chunk, introducing significant overhead and latency. It would also make it more difficult to support concurrent send and receive operations, limiting the efficiency of the pipeline.

The tradeoff of using WebSockets is increased implementation complexity. The system must manage connection state, define a custom messaging protocol for chunk boundaries and control signals, and handle potential connection interruptions. Despite these challenges, WebSockets provide a more suitable foundation for the real-time streaming behavior required by the system.

Design Tradeoffs: Complexity vs. Performance

Across all design decisions, a consistent tradeoff emerged between system complexity and performance. The use of chunking, streaming, asynchronous communication, and WebSockets significantly improves system efficiency, scalability, and responsiveness. These choices enable continuous data flow and better utilization of hardware acceleration.

However, these benefits come at the cost of increased complexity in system design and implementation. The distributed nature of the system requires careful coordination between components, robust error handling, and more sophisticated debugging strategies. Seemingly small design decisions, such as chunk size or message structure, can have a significant impact on overall performance and correctness.

Ultimately, the design prioritizes performance and scalability, accepting additional complexity as a necessary tradeoff. This reflects the realities of building high-performance distributed systems, where efficiency often requires more advanced coordination and engineering effort.

Implementation

Client Implementation

The client component of the system is responsible for handling user interaction, file ingestion, communication with the server, and reconstruction of the final compressed output. It serves as the entry and exit point of the system, allowing users to upload files for compression and receive the resulting ZIP archive. The client was implemented in Python and uses a simple interface that allows users to select one or more files, after which the system extracts relevant metadata such as file size, chunk size, and total number of chunks. This metadata is transmitted to the server prior to sending any file data, enabling the server to properly initialize and coordinate the processing pipeline.

To support efficient streaming and operate within the memory constraints of the FPGA, the client divides each file into fixed-size chunks of 512 KB. Instead of loading the entire file into memory, the file is read incrementally from disk, allowing large files to be processed without excessive memory usage. Each chunk is transmitted sequentially as it is read, enabling continuous data flow through the system. This chunking strategy not only reduces memory overhead but also aligns with the FPGA's requirement for processing data in bounded segments.

Communication between the client and server is implemented using a persistent WebSocket connection, enabling real-time, bidirectional data transfer. Each chunk is sent using a structured message format consisting of a JSON start message, the raw binary payload, and a JSON end message. This framing ensures that the server can correctly identify chunk boundaries and maintain proper ordering. In addition to chunk data, control messages are used to signal the start and end of an upload, as well as to communicate expected chunk counts, allowing the server to validate incoming data and detect errors.

To improve performance and responsiveness, the client leverages Python's asyncio library to enable asynchronous communication. Two concurrent tasks are created: one responsible for transmitting file chunks to the server, and another responsible for receiving processed chunks from the server. This design enables full-duplex communication, allowing upload and download operations to occur simultaneously over a single connection. As a result, the system can overlap communication and processing, reducing idle time and improving overall throughput.

As processed chunks are returned from the server, the client receives them in a streaming manner and immediately begins reconstructing the output file. Because the size of compressed chunks may differ from the original input chunks, the client dynamically handles incoming data without assuming fixed output sizes. Chunk ordering and sequence numbers are verified to ensure correctness, and the data is written directly into a ZIP archive as it is received. The client tracks

CRC32 checksums and total file size to validate data integrity during reconstruction, ensuring that the final output matches the original input.

Once all chunks have been received and validated, the archive is finalized and made available to the user for download. This streaming reconstruction approach avoids buffering the entire output in memory and allows the system to efficiently handle large files. Overall, the client implementation enables a seamless end-to-end workflow, combining chunk-based processing, asynchronous communication, and real-time reconstruction to support efficient interaction with the FPGA-accelerated compression system.

Server Implementation

The server serves as the centerpiece between the client and the hardware implementation. It continuously receives 512 KB chunks from the client and sends them to the FPGA immediately upon receipt. During that process, the chunks sent to the FPGA are returned to the server as processed data, which is then automatically forwarded back to the client. This creates a parallel and concurrent transfer of data, as the transmission of unprocessed chunks and the return of processed chunks to the client occur at the same time.

To facilitate this transfer, a WebSocket establishes a connection between the client and the server, setting up a protocol that allows continuous file streaming. This enables the seamless transfer of data from the client once a file is uploaded for compression. Chunks of 512 KB are transferred through the WebSocket to the server without interruption, with the chunk size being validated on the server end. This validation ensures that the server receives the correct amount of data from the client before sending the precise chunk size to the FPGA. The server also handles an edge case where the final chunk received from the client is under 512 KB; in this scenario, the server successfully accepts the smaller chunk and forwards it to the FPGA.

The transfer of data chunks from the server to the FPGA follows a process similar to the transfer from the client to the server. The server sends the 512 KB chunks to the FPGA through a compressor pass-through, where the data is processed by the hardware compression implementation via the AXI-Stream protocol. Once a chunk has been processed on the hardware side, it is returned to the server and immediately forwarded to the client. This allows the client to reassemble the compressed chunks into the final file for the user. By offloading the assembly of the compressed chunks to the client side, the server improves its overall performance and easily handles concurrent data transfers.

The transfer of data chunks to the server and subsequently to the FPGA occurs simultaneously. The system manages this concurrent transfer of data between the client and the FPGA through the server in a highly streamlined process. For example, the server can receive new data chunks from the client while simultaneously receiving compressed chunks from the FPGA and sending them back to the client. This parallel streaming allows the file to be compressed much faster and

significantly improves processing efficiency compared to waiting for all chunks to arrive sequentially.

Compressor Implementation

The hardware compressor was designed using Vitis HLS and Vivado, and used a PYNQ overlay to interact with the server. The two stages of DEFLATE, LZ77 and Huffman coding, were each encapsulated into their own module written in Vitis HLS. A simplified FSM of the LZ77 module is shown below in Figure 2.

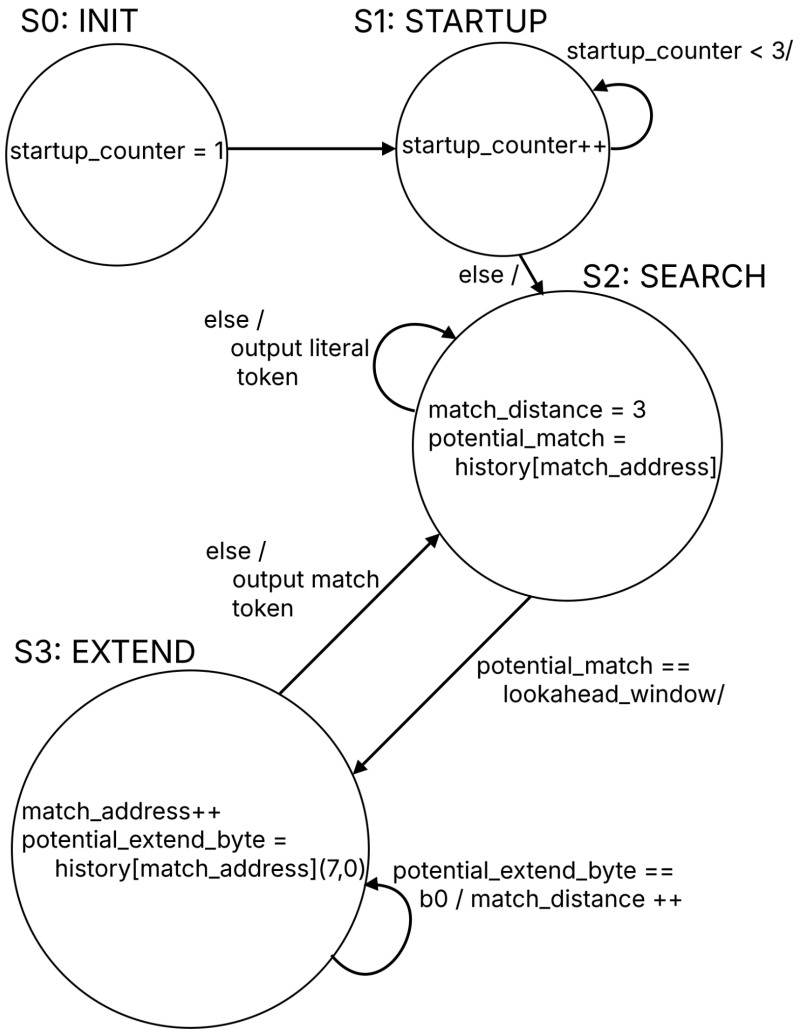

Figure 2: Simplified LZ77 module FSM.

In each SEARCH, EXTEND, or SKIP cycle of the state machine, one byte of the input file is shifted into the 3 byte lookahead window, where b2 is its oldest byte and b0 is its newest. The bytes of the lookahead window are then hashed and used as an index to store the index where this combination can be found in the circular history buffer. In the first three cycles, covered by

the INIT and STARTUP states, there aren't enough bytes in the buffer to make matches yet, so nothing happened besides shifting bytes into the lookahead window. When at least four bytes have been read, on the first SEARCH cycle, the hash of the lookahead bytes begin being used to check the history buffer for three-byte matches. Every cycle that a match isn't found, a token containing the literal byte is sent on the output stream. If one is found, the FSM tries to EXTEND the match, checking each byte following the match in the history buffer against new bytes entering the lookahead window. This continues until the match limit, 258 symbols, is hit, or until the bytes don't match. When this happens, the match is encoded as the length of the match, and the distance from the match back to the original sequence, and sent as a token on the output stream. The next cycle is a SKIP, allowing the bytes compressed by EXTEND to exit the lookahead window so no double compression occurs, before returning to SEARCH. This loop continues until all input bytes have been processed.

The LZ77 module was debugged in simulation using a self checking testbench in Vitis. A test vector of input strings was fed into the simulated module, and the output tokens were decoded and used to reconstruct an output string, which was compared against the original. This was useful for testing and accounting for edge cases early in development.

The second stage of the compressor is a Huffman coder module, which processes the tokens output by the LZ77 stage. A simplified FSM of the Huffman module is shown below in Figure 3.

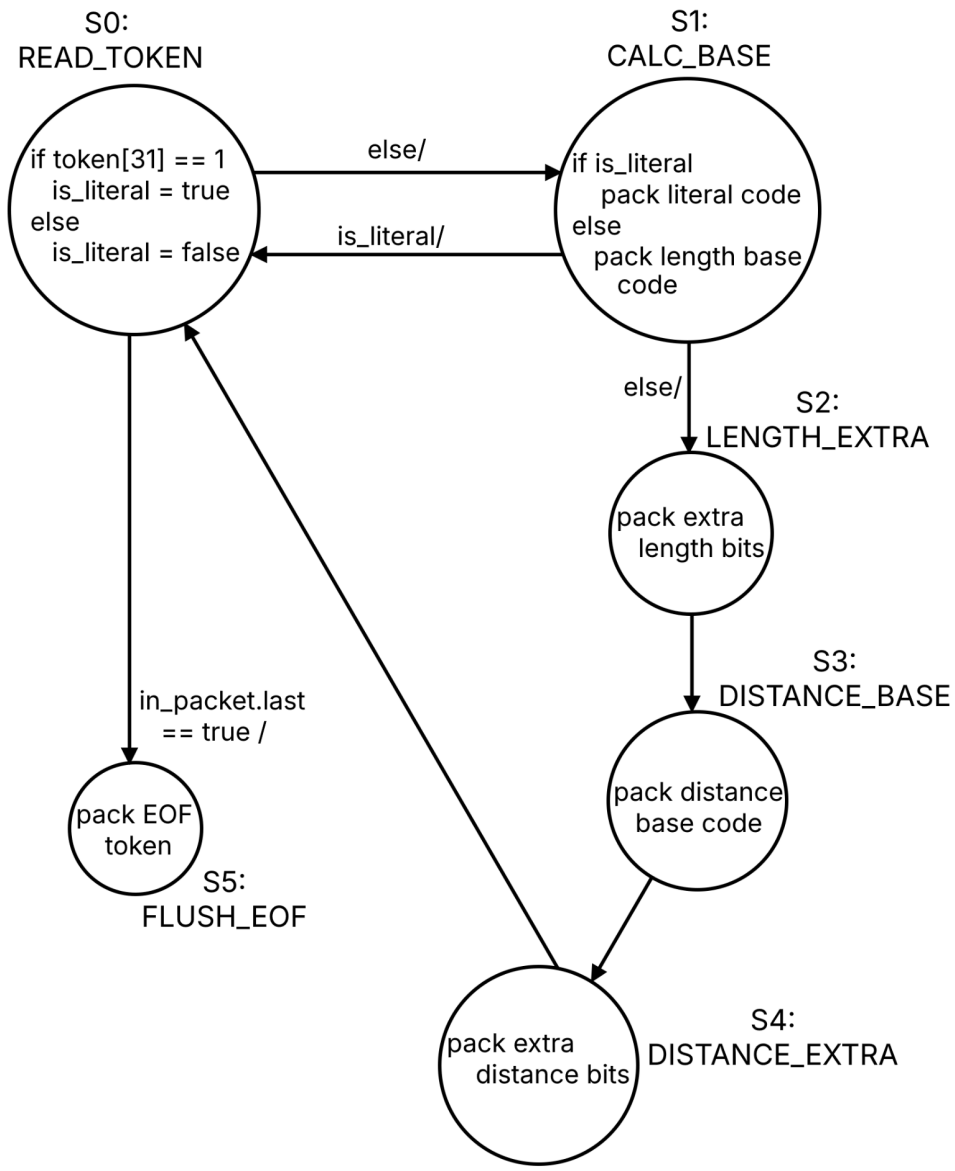

Figure 3: Simplified Huffman coder module FSM.

The Huffman module implements fixed Huffman Coding as described in RFC 1951 [7], which describes the DEFLATE format. Its main role is calculating or looking up final encodings for the literal and length-distance pairs it receives from the LZ77 stream. This is necessary for mixing matches and literal bytes together in the same output file. In READ_TOKEN, the newest input stream token is checked to see if it holds a literal or a match, and extracts the literal or the length and distance to be encoded. Next in CALC_BASE, the literal or length are converted into a base code using the same function, get_base_code, and are packed into the output bit accumulator. Match lengths can have extra bits needed to encode them, which are looked up from the length ROM in this state as well. In LENGTH_EXTRA, these extra bits are shifted into the accumulator. DISTANCE_BASE looks up the distance base code from its ROM, as well as its

extra bits. The base code is packed in this state, while the extra bits are packed in DISTANCE_EXTRA. In conjunction with this FSM loop, the output bit accumulator outputs an encoded word every 32-bits that are packed. When the coder receives the EOF token from the LZ77 stream, it packs a reserved end-of-block code and flushes the final output word(s). The last word sent is marked as LAST, and the valid bytes are marked based on how full the accumulator was. A final note is that before base codes are packed into the output buffer, they are reversed to be big-endian, which is required by DEFLATE.

This module was tested in simulation by hand checking short output strings. A self-checking testbench would have been much more complex to construct for this module, so it was passed over to move on to integrated hardware testing.

Finally, the two modules were integrated with the PYNQ-Z2's ZYNQ Processing System in Vivado. Because the system, LZ77, and Huffman components ran at 100, 77, and 50 MHz respectively, AXI4-Stream FIFO buffers were used to pass data between each component. An AXI DMA was used to send and receive the data to and from the compression modules.

Evaluation & Results

Testing and Data Collection

To test both the correctness and performance of the system three types of input files, with sizes ranging from hundreds of kilobytes(kB) to 20 megabytes(MB). We originally intended to test the system with larger files in the gigabyte(GB) range, but had to limit the max file size to 20MB due to the DMA transfer issues encountered late in the project. The three input file types used were:

- Literature text files sourced from Project Gutenberg[3]. These have high potential for compression due to the repetitiveness of language.
- BMP image files, which don't implement complex compression of their own and thus can be compressed by the system. Both images drawn in Paint.NET[4], with large areas of the same color, and photos, with more varied pixel colors and distributions, were used.
- Random binary/text files generated with the PineTools Random File Generator[5], which contain little to no redundancy and represent worst-case scenarios.

The compression system was tested for correctness by compressing the test files, extracting the returned archives using 7-ZIP[2], and comparing the binary contents of the original and extracted files with our Python comparison script. If the archives were extractable and their contents matched the original files, we interpreted them to be correct. The inability to extract the archive or a discrepancy in the output file indicated an error in the system, and occurrences of this were used to guide debugging of the system.

Performance of the system was evaluated by comparing its throughput and compression ratio on the test files with that of the zlib[6] DEFLATE library, used via command line on the client PC.

For the system to truly be useful, it has to outperform local compression on the same PC that uses the client program, which is why this methodology was used. To compute the throughput of the compressor, the client logs the upload, processing, and total times for each file processed. The local zlib execution times were logged with the unix time command. The timings for each test file were recorded multiple times and averaged, used with the original file size to calculate throughput for each file:

$$Throughput = \frac{Original\ file\ Size}{Avg.\ Total\ Time}$$

The compression ratio for each file was calculated for the system and local zlib methods from the original and compressed file sizes:

$$Compression\ Ratio = \frac{Original\ File\ Size}{Compressed\ File\ Size}$$

Our original goal was to surpass the throughput of local file compression, while sacrificing some compression ratio due to using fixed Huffman codes. If our compressor could be faster than local compression while still achieving measurable compression, we would consider it a success.

A supplementary test performed was a comparison of upload and download times for text(.txt) and binary(.bin) files with random contents. This was performed with 1, 5, 10, and 15 MB files with random contents, with each file being saved as both a text and binary file such that the contents of the file were the same for each extension. Repeated timings were taken for each file size and extension and averaged. The throughput was also calculated from these timings. We performed this test because early testing showed binary files taking longer to process than text files, and we wanted to investigate the phenomenon further.

Results

Correctness

An example of the manual process used to check the correctness of all test files is shown below in Figures 4 and 5. Figure 4 shows the client sending england-under-the-angevin-kings.txt, and receiving the compressed file in a ZIP archive. Figure 5 shows the Python script used to verify that the extracted output files were identical to the original test files.

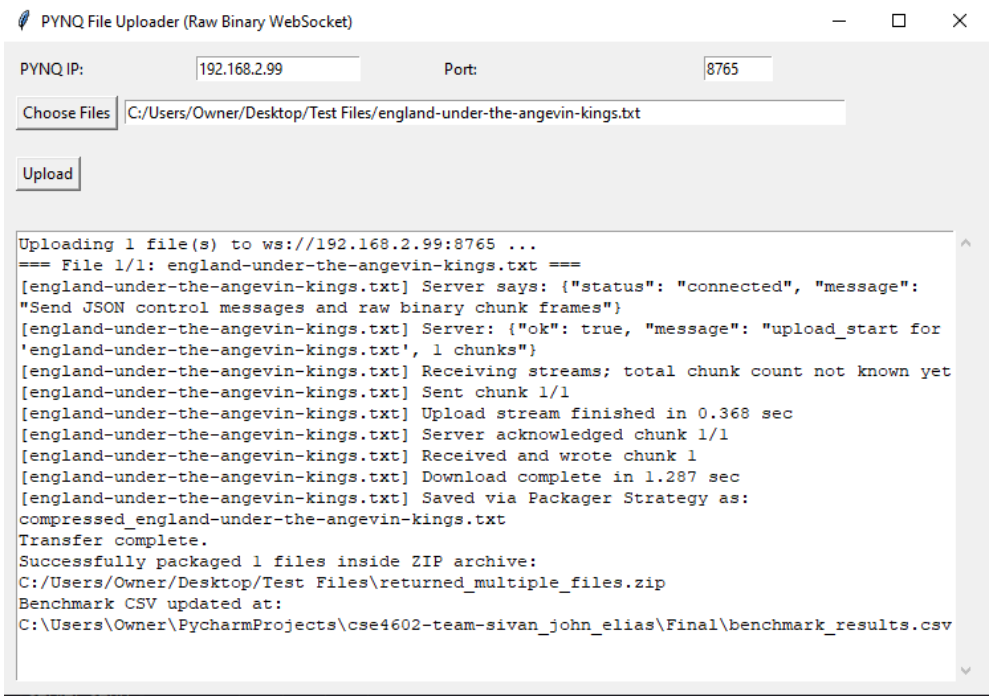

Figure 4: Client program sending an input file and saving the compressed file as a ZIP archive.

```python
def check_if_files_are_identical(first_file_path: Union[str, Path],  1 usage  Elias Dahl *
                                 second_file_path: Union[str, Path]) -> bool:
    """
    Compares two files to see if their contents are exactly identical
    using filecmp
    """
    are_files_identical: bool = filecmp.cmp(
        first_file_path,
        second_file_path,
        shallow=False
    )

    return are_files_identical

if __name__ == "__main__":
    #test files
    ORIGINAL_FILE = "england-under-the-angevin-kings.txt"
    DECOMPRESSED_FILE = "compressed_" + ORIGINAL_FILE

    are_identical: bool=check_if_files_are_identical(ORIGINAL_FILE,DECOMPRESSED_FILE)

    print("Are ", ORIGINAL_FILE, " and ", DECOMPRESSED_FILE ,"identical?:",are_identical)
```

Figure 5: Python script used to check if files are identical.

The output from running the script in Figure 5 was:
Are england-under-the-angevin-kings.txt and compressed_england-under-the-angevin-kings.txt identical?: True

Throughput

The compression throughput achieved by the FPGA compressor and local zlib compression is shown below in Figure 6 and Table 1.

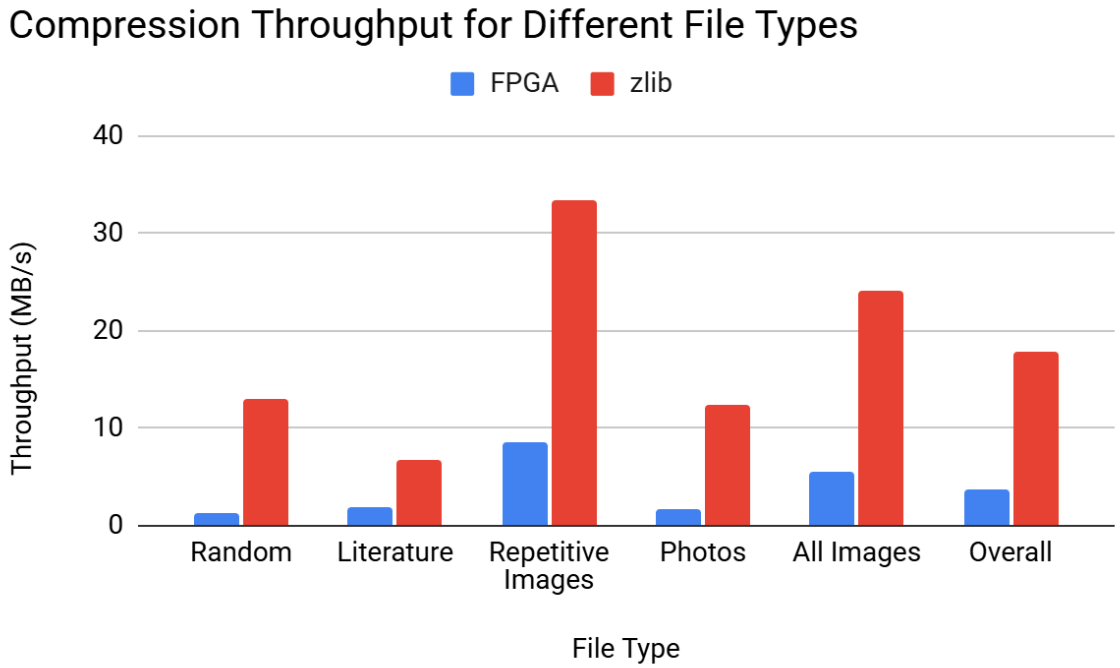

Figure 6: Compression throughput in MB/s for different file types on the FPGA and zlib.

File Type	Random	Literature	Repetitive Images	Photos	All Images	Overall
FPGA Throughput (MB/s)	1.3804	1.9481	8.5069	1.6761	5.4710	3.7965
zlib Throughput (MB/s)	12.938	6.6638	33.339	12.452	24.056	17.890

Table 1: Compression throughput in MB/s for different file types on the FPGA and zlib.

As seen from Figure 6 and Table 1, our system did not meet its goal of surpassing the throughput of local compression on the client PC. An interesting outcome is that the ranking of throughputs across file types was different for the FPGA compressor and zlib, particularly with random files. zlib achieves a higher relative throughput with random files than literature, whereas the FPGA compressor processed literature faster.

Compression Ratio

The compression ratio achieved by our compression system and local zlib compression is shown below in Figure 7 and Table 2.

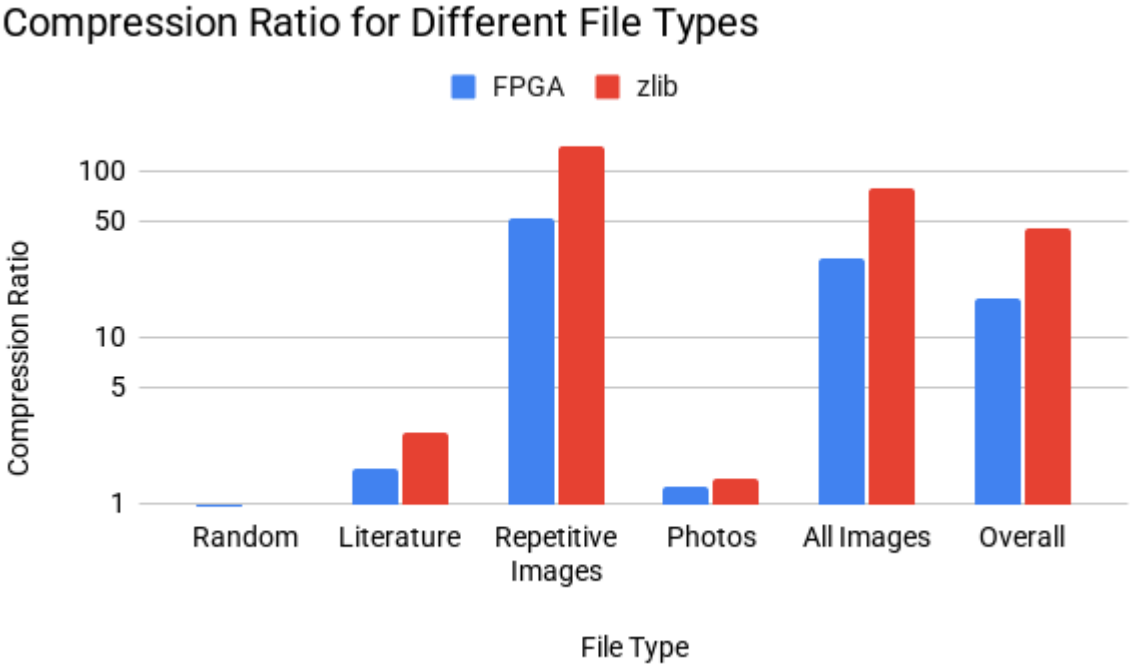

Figure 7: Compression ratio for different file types from FPGA and zlib compression.

File Type	Random	Literature	Repetitive Images	Photos	All Images	Overall
System Ratio	0.94848	1.6092	52.893	1.2567	29.944	17.330
zlib Ratio	0.99952	2.6637	143.36	1.4180	80.276	45.744

Table 2: Compression ratio for different file types from FPGA and zlib compression.

As seen in Figure 7 and Table 2, the compression ratio of the FPGA compressor was worse than the zlib software implementation, as expected due to the differences in the hardware and software algorithms. It is notable that the ranking of compression ratios between files types is the same for both the FPGA compressor and zlib.

Binary vs Text File Processing Time

A comparison of the upload time, download time, and throughput for binary and text files is shown below in Figures 8-10.

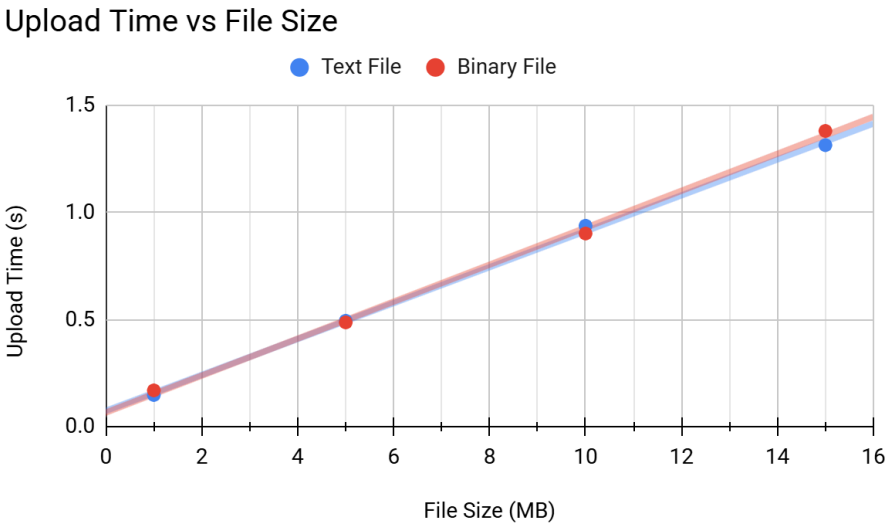

Figure 8: Upload time vs file size for text and binary files.

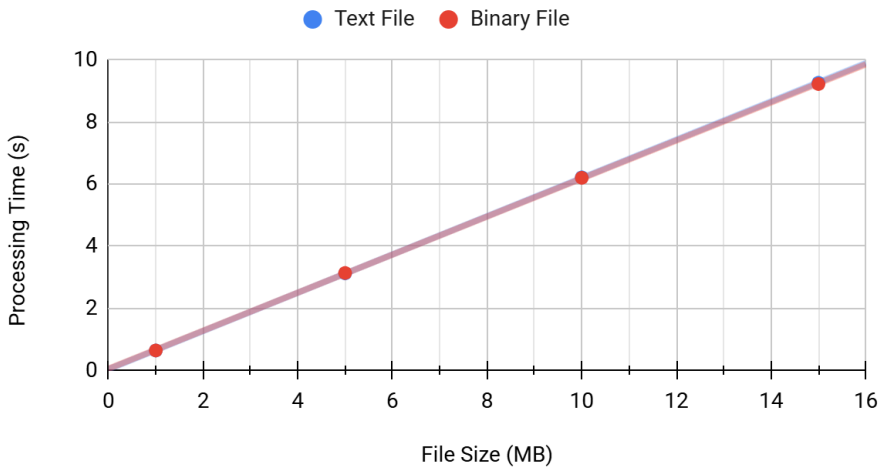

Figure 9: Processing time vs file size for text and binary files.

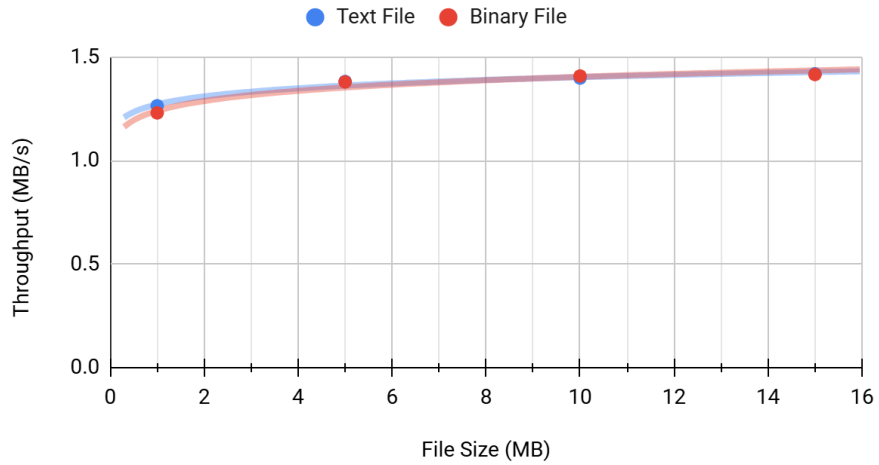

Figure 10: Throughput vs file size for text and binary files.

As seen from Figures 8-10, there was no significant difference in metrics between text and binary files when using a larger sample size. This disproves the early impression that binary files took longer to upload and process than text files. Figure 10 also shows that throughput is lower for smaller file sizes, before leveling off as the file size increases, indicating overhead that is amortized when processing larger files.

Discussion & Reflection

Based on the results gathered from testing, we met some, but not all of the goals for our system. We were able to build a file compression system that produced correct output with significant compression, but fell short of our goal of a high throughput and unbounded file size. Having correct output is the only strength our compressor has at this time, although there is large potential for improvement in the future. An expected limitation was the reduced compression ratio compared to a software implementation. Our hardware compressor uses fixed Huffman coding, where encodings are preset based on RFC 1951[7], whereas zlib uses dynamic Huffman coding, building a Huffman tree based on the frequencies of the tokens within a block, allowing for more compression. An unexpected limitation was our throughput being much lower than the software implementation, which was likely due to multiple factors. The biggest was the other unexpected limitation of being limited to one file chunk, due to incorrect handling of repeated DMA transfers. Not only does this limit our max file size due to limited continuous memory available on the PYNQ board, it also forces the compressor to wait for the server to receive the whole file before it can begin compressing. Similarly, it forces the server to wait for the whole file to be compressed before returning it to the client, in total eliminating any chance of overlapping execution and streaming. As this issue was unable to be resolved before the end of the project timeline, we were unable to thoroughly investigate other potential slowdowns, such as WebSocket overhead creating a network bottleneck, an issue encountered by other groups in the class.

It was seen from the throughput results that the FPGA compressor had higher throughput for literature files than random files, whereas zlib had the opposite result. One explanation for this is that with random files, the FPGA is making little to no matches, resulting in all literal tokens being sent to the Huffman stage. With literature files, where much more matches are made, less tokens are being processed by the Huffman coder, resulting in a shorter processing time. As for why zlib is able to process the random files faster than literature, it could be that its LZ77 match making is relatively slow compared to its Huffman coder, causing files with low matches to process quickly.

One quirk seen in early testing, binary files supposedly taking longer for our system to process than text files, turned out to be an artifact of a small sample size of timing. When compressing binary and text files of various sizes over repeated runs, there was no significant difference in upload or processing time, as well as throughput. This result reinforces the value of thorough

testing before making conclusions, as initial testing with a small sample size made us believe in a side effect that didn't actually exist.

Ethical Considerations

The design and deployment of a file compression system that transfers user data across a network introduces several important ethical considerations related to privacy, security, and responsible system design. Because our system operates as a networked service that accepts user-provided files and processes them remotely, it must be designed with careful attention to how data is handled throughout the pipeline.

One of the primary concerns is data privacy during transmission. Files submitted by users may contain sensitive or personal information, and in the current implementation, data is transmitted over a WebSocket connection without encryption. This creates a potential risk of interception or unauthorized access if the system were deployed in an untrusted network environment. To address this concern in a production setting, the system would need to incorporate secure communication protocols such as TLS to ensure that all data is encrypted in transit.

Another important consideration is the security of stored and processed data. Although the system is designed to stream data rather than persist it long-term, temporary buffering occurs at both the client and server levels. Without proper safeguards, this intermediate data could be exposed to unauthorized processes or users. Future improvements should include stricter memory management policies, secure deletion of buffers after use, and access controls to prevent unintended data exposure.

The system must also account for the risk of malicious or malformed inputs. Since users can upload arbitrary files, there is potential for abuse through corrupted data, oversized payloads, or intentionally crafted inputs designed to disrupt processing. While basic validation is performed (such as checking chunk sizes and ordering), a production-ready system would require more robust input validation, rate limiting, and error handling mechanisms to ensure stability and prevent misuse.

In addition, there are broader considerations related to fair and responsible resource usage. Hardware-accelerated systems such as FPGA-based services can provide significant performance benefits, but they also introduce constraints on shared computational resources. In multi-user environments, mechanisms would be needed to ensure fair allocation of processing time and prevent any single user from monopolizing system resources.

Finally, this project highlights the importance of secure system design in distributed architectures. Because the system spans multiple components, including a client, server, and FPGA, ensuring correctness and security across all layers is non-trivial. Errors or vulnerabilities

in any part of the pipeline can compromise the entire system. This reinforces the need for comprehensive testing, validation, and security considerations early in the design process.

Overall, while our implementation demonstrates the feasibility and performance benefits of FPGA-accelerated file compression, deploying such a system in real-world environments would require additional safeguards to ensure user data is handled securely, responsibly, and ethically.

Project Reflection

Although our system fell short of our desired metrics, there were still some things it did well. In initial system testing with the compressor passthrough, files were able to be streamed in chunks to and from the server and client, allowing overlap between sending, receiving, and processing the files in the passthrough. Although this pipeline broke during final integration, the framework is there for this flow to be reimplemented with improvements to the compressor design. Alongside this pipeline, both the client and server can handle concurrent tasks, interleaving communicating with I/O operations. Again, while the full performance benefits of this weren't able to be seen in the final product, the groundwork is laid for a high performance system after future improvements. Finally, the compressor was able to correctly compress files up to 20MB, and the same fixes that would resolve the DMA transfer pipeline stalls would remove this limit as well.

In final integration, major bugs occurred that broke various aspects of the system, like the file chunk pipeline that worked with the passthrough module. This was due to the Huffman coder incorrectly handling DMA transfers, only being able to trigger one at the end of the file. Unfortunately, there was less time than needed to resolve this issue alongside other correctness issues that occurred during final testing and debugging. We were able to partially sidestep this issue by increasing the chunk size from 512kB to 20MB, allowing us to test the correctness of compression on bigger files without streaming multiple chunks.

Even though we weren't fully successful, we learned valuable lessons from our project. As previously stated, we had much more difficulty with final integration than expected. Although we did have earlier successful integration tests with the compressor passthrough, we should have repeated this incrementally throughout the semester. To this end, it became clear that the order in which components are developed has a big impact on your ability to incrementally test. Although the LZ77 stage is the first of the two sequential stages, it would have been advantageous to build the Huffman encoder first. The Huffman coder is what produces the final output, so a passthrough LZ77 module could have been used to produce correct but uncompressed output. This would have allowed us to experience several integration challenges way earlier, including ZIP archive packing of real DEFLATE streams, and variable length output DMA transfers.

Project Management

Deliverables

At the beginning of the project, we set out to design and implement a complete hardware-accelerated compression system that could accept user input, offload computation to an FPGA, and return compressed results through a networked interface. The original plan included building a fully integrated pipeline with real-time streaming, a functional compression module implemented on the FPGA, and a benchmarking framework to evaluate performance against software-based solutions. A key design goal was to support large file sizes through a chunk-based streaming architecture, allowing the system to process data incrementally without being limited by memory constraints.

Over the course of the project, several adjustments were made due to the complexity of hardware-software integration and debugging challenges. While some aspects of the original plan required refinement, the team successfully delivered a functional end-to-end system that demonstrates the core objectives of the project. However, our project only supports files of up to 20 MB in size, meaning that the compression implementation could not take files over that size as explained previously.

The final system included a working pipeline that allows users to upload files, stream data in fixed-size chunks, process those chunks through the FPGA, and receive the output in a reconstructed ZIP archive. The system supports asynchronous communication between the client and server, enabling overlapping upload and download operations and demonstrating the feasibility of real-time data streaming.

The project also achieved successful FPGA integration, with the server coordinating data transfer between the client and the hardware using AXI-based communication. However, as discussed in the project reflection, incorrect handling of DMA transfers by the Huffman coder module caused a stall on the server when attempting to process more than one chunk. With no time to fully debug the issue, we increased the chunk size to 20MB to allow for larger file transfers. This caused the file to need to be processed in full in each stage of the pipeline, preventing the system from leveraging the chunk-based streaming approach originally implemented with the compressor passthrough.

In addition, a benchmarking and evaluation framework was developed to measure system performance. This includes collecting metrics such as upload time, processing time, total runtime, and throughput across different file types and sizes. The results were used to analyze system behavior and highlight both the benefits and limitations of the current implementation.

Overall, the final deliverables reflect a functional and integrated system that meets the primary project goals of demonstrating FPGA acceleration and system-level integration, while also revealing important limitations in scalability that provide direction for future improvements.

The full code for our client and server programs, and compressor HLS can be found in our github repo[8].

Timeline

Phase	Timeframe	Planned Work	Actual Outcome
Project Planning & Design	Jan 12 - Jan 31	Define system architecture, choose communication method, outline chunking strategy	Completed on time; finalized WebSocket-based streaming architecture and overall system design
Client Development	Feb 1 - Feb 20	Implement file upload interface, chunking, and communication logic	Completed on time; client supports chunking, async communication, and reconstruction
Server Development	Feb 10 - Mar 5	Build WebSocket server, manage chunk routing, integrate FPGA interface	Mostly on track; basic pipeline completed but required later debugging
Hardware (FPGA) Development	Feb 15 - Mar 20	Implement LZ77 + Huffman modules and test individually	Completed on time; modules worked correctly in simulation
System Integration	Mar 20 - Apr 10	Integrate client, server, and FPGA into full pipeline	Delayed due to hardware-software communication issues (DMA, chunk handling)
Debugging & Optimization	Apr 5 - Apr 20	Fix integration bugs, improve streaming performance	Took longer than planned due to complex debugging and synchronization issues

Benchmarking & Testing	Apr 15 - Apr 25	Run performance tests and collect metrics	Completed on time once system stabilized
Final Demo & Report	Apr 25 - May 1	Prepare demo and finalize report	Completed on time

What Stayed on Track

The early stages of the project progressed largely as planned. System architecture decisions, including the use of chunking, streaming, and WebSocket communication, were defined early and remained consistent throughout development. The client-side implementation was completed on schedule and successfully supported asynchronous file upload, chunk-based transmission, and real-time reconstruction of compressed outputs.

Similarly, the hardware modules for compression (LZ77 and Huffman coding) were implemented and validated in simulation without major delays. This allowed the team to establish confidence in the correctness of the core compression logic before integration.

What Slipped (Integration & Debugging)

The primary delays in the project occurred during system integration and debugging. While individual components (client, server, and hardware) functioned correctly in isolation, combining them into a fully operational pipeline proved significantly more complex than anticipated.

The most critical issue was related to hardware-software communication, particularly DMA handling. Instead of supporting true streaming across multiple chunks, the system was forced to process entire files at once. This eliminated the intended overlap between upload, processing, and download stages and introduced a strict file size limitation of approximately 20 MB.

These integration challenges led to extended debugging time, especially in coordinating asynchronous communication, ensuring correct data ordering, and handling edge cases across system boundaries. As a result, optimization efforts and deeper performance improvements were limited by time constraints.

Overall, while the system achieved a functional end-to-end pipeline, the integration phase required significantly more effort than initially planned, highlighting the complexity of hardware-software co-design and the importance of early validation of communication interfaces.

Team Contributions

System Integration & Front-End Lead - *Sivan Elisha*
Sivan was responsible for the design and implementation of the client-side system and overall end-to-end integration of the pipeline. This included building the file upload interface, implementing chunk-based file streaming, and developing the asynchronous WebSocket communication layer using Python's asyncio to support full-duplex data transfer. She designed and implemented the client-side logic for sending structured chunk messages, receiving processed data, and reconstructing the final ZIP archive with proper ordering and data integrity validation. She also developed the benchmarking and data collection framework used to measure upload time, processing time, and overall system throughput, and was responsible for generating and visualizing performance results for evaluation.

Software & Communication Lead (Microprocessor Control) - *John Iwenofu*
John was responsible for the backend communication between the user interface and the hardware compressor. He developed the server-side architecture that facilitated the overall file compression process. He established WebSocket communication between the client (frontend) and the server (backend) to enable data streaming, while also managing the data routing to the FPGA (hardware) for compression. Finally, he implemented the transfer of processed chunks from the FPGA back to the client through the server, enabling concurrent data transfer between the client and the hardware. Overall, he ensured that the system would not be overwhelmed by large files by managing the data flow in strict 512 KB chunks across the client, server, and hardware.

Hardware Lead - *Elias Dahl*
Elias was responsible for designing the hardware compressor and integrating it with the server program. He designed the LZ77 and Huffman coder modules in Vitis HLS, and integrated them with the ZYNQ processing system in Vivado. He also wrote overlay drivers and wrapper classes in Python for the compressor, allowing for simplified use in the server.

Conclusion & Future Work

Conclusion

This project successfully achieved its primary goal of designing and implementing a complete FPGA-accelerated file compression system. The team built a fully functional end-to-end pipeline that allows users to upload files, process them through a hardware compression module, and receive the resulting compressed output in a standard ZIP format.

Through this implementation, we demonstrated the feasibility of integrating hardware acceleration into a distributed system. The project validated that computational workloads, such

as compression, can be offloaded from the CPU to an FPGA, enabling a scalable architecture where the processor is freed to handle other tasks. The system also showed that a streaming-based communication model using WebSockets can support real-time data transfer between a client, server, and hardware module.

Although performance goals were not fully met, the project provides a strong proof of concept for combining streaming data pipelines with FPGA acceleration. The successful integration of client-side processing, server coordination, and hardware computation highlights the practicality of hardware-software co-design in modern systems.

Future Work

While the system is functional, several areas for improvement remain and provide clear direction for future development.

The most critical improvement is resolving the DMA multi-chunk limitation. Currently, the system processes entire files at once instead of streaming multiple chunks through the FPGA. Fixing this issue would restore true streaming behavior, allow overlapping of upload, processing, and download stages, and significantly improve both scalability and throughput.

Another key area is improving compression speed. The current implementation does not outperform software-based compression, largely due to integration overhead and the lack of true concurrent streaming. Optimizing data transfer, reducing communication latency, and improving hardware pipeline efficiency would help close this gap.

Improving the compression ratio is also an important next step. The current system uses fixed Huffman coding, which is less efficient than the dynamic Huffman coding used in software implementations like zlib. Extending the hardware design to support dynamic coding or hybrid approaches could lead to significantly better compression performance.

Finally, the system would benefit from stronger fault tolerance and robustness. Adding support for resumable uploads, better handling of connection failures, and more comprehensive error recovery mechanisms would make the system more reliable and suitable for real-world deployment.

Overall, these improvements would transform the current prototype into a more scalable, efficient, and production-ready system.

References

[1] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337-343, May 1977. https://ieeexplore.ieee.org/document/1055714 (accessed Feb. 11, 2026).

[2] “7-Zip.” Accessed: Feb. 10, 2026. [Online]. Available: https://www.7-zip.org/

[3] “Project Gutenberg,” Project Gutenberg. Accessed: May 01, 2026. [Online]. Available: https://www.gutenberg.org/

[4] “Paint.NET - Free Software for Digital Photo Editing.” Accessed: Feb. 10, 2026. [Online]. Available: https://www.getpaint.net/

[5] “Online Random file generator,” PineTools. Accessed: May 01, 2026. [Online]. Available: https://pinetools.com/random-file-generator

[6] “zlib Home Site.” Accessed: May 01, 2026. [Online]. Available: https://zlib.net/

[7] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, May 1996.

[8] E. Dahl, S. Elisha, J. Iwenofu, “Team Project Repository”, 2025 [Online] Available: https://github.com/WashU-CSE4602-SP26/cse4602-team-sivan_john_elias