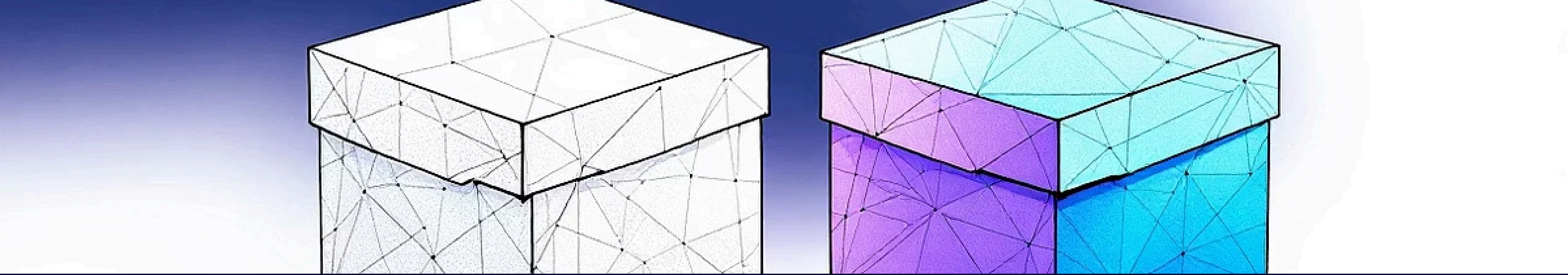
# Huffman Compression Algorithm Optimisation

## Performance Analysis: Version 1 vs. Version 2

Key Achievement Highlight: 86% Faster Encoding | 60% Memory Reduction | 100% Accuracy

Team Members: **Abutali Abusseiit, Nurlan Izbassar, Zhumakhanet Akbota, Syrlybekov Madiyar**

Today we're presenting our optimisation of the Huffman compression algorithm, where we achieved significant performance improvements whilst maintaining perfect accuracy.



## Project Overview

### What We Did

- Implemented two versions of Huffman compression
- Optimised for speed and memory efficiency
- Maintained 100% data accuracy

### Key Results

- 86% faster encoding
- 60% less memory usage
- 625% throughput increase

Test data: 4.4MB file (4.4 million characters)

We built two implementations of Huffman compression. Version 1 is the baseline - it works correctly but is slow. Version 2 is our optimised version with major performance improvements. We tested both on a 4.4MB file and measured speed, memory, and accuracy, demonstrating significant gains.

# Why We Optimised: Version 1 Performance Issues

### String Concatenation

Repeatedly creates new objects, leading to $O(n^2)$ complexity and significant performance overhead, especially with large inputs.

### Hash Map Overhead

Unordered maps for frequency counting introduce non-trivial overhead due to hashing, collision resolution, and dynamic memory allocation.

### Memory Fragmentation

Pointer-based tree structures can cause scattered memory allocations, leading to fragmentation and poor cache utilisation.

These issues become critical with large files, often containing millions of characters, severely impacting processing times.

Version 1, whilst functionally correct, suffered from several critical performance bottlenecks. The most significant was string concatenation, where every character encoded resulted in a new string object creation. This quickly leads to an $O(n^2)$ complexity that is unsustainable for large files. Additionally, the overhead of hash maps for frequency counting and the memory fragmentation from pointer-based tree structures further exacerbated the performance degradation.

## Our Optimisation Strategy

### Array-Based Counting

Replaced `unordered_map` with a fixed-size `array[256]`, guaranteeing a consistent 1KB memory footprint and enabling direct O(1) indexing for character frequency tabulation.

### Bit-Level Operations

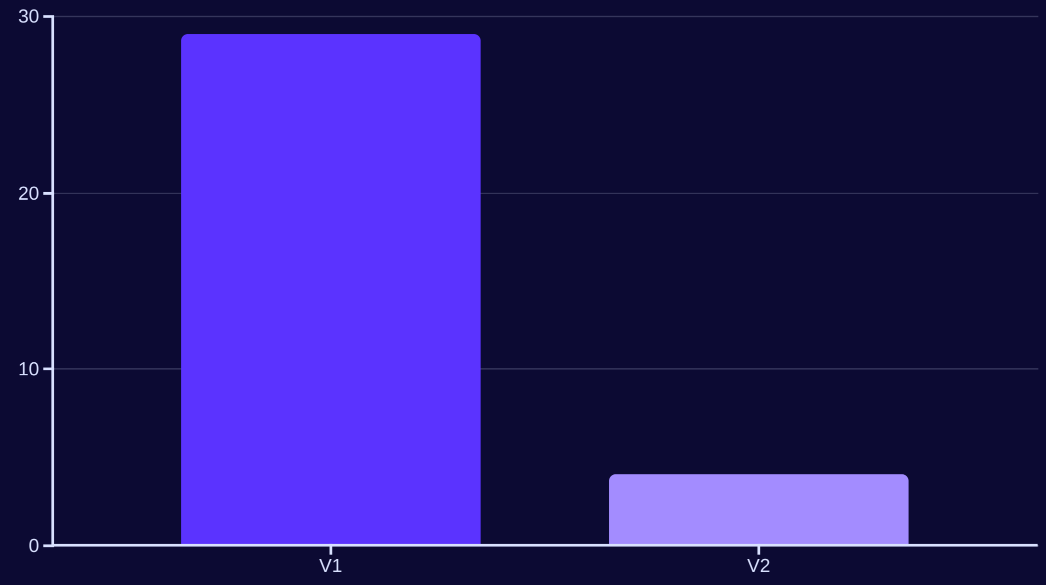
Substituted inefficient string concatenation with a custom `BitWriter` class, allowing direct bit packing into bytes and eliminating $O(n^2)$ overhead for encoded output construction.

### Index-Based Tree

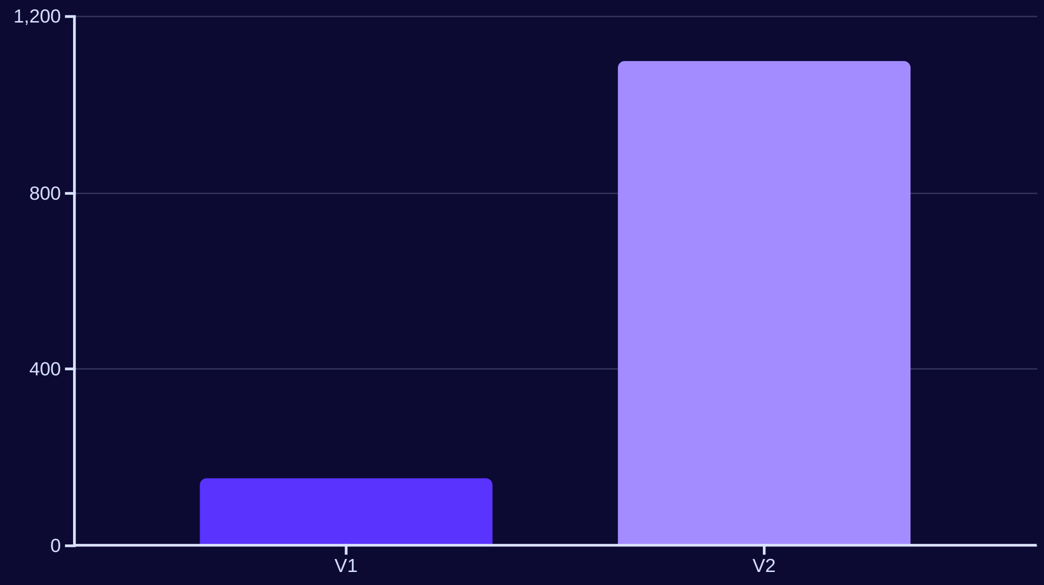
Transitioned from pointer-based tree nodes to a `vector` storing nodes with index references, ensuring a contiguous memory layout for improved CPU cache performance and reduced fragmentation.

Our optimisation strategy focused on three core areas to mitigate the identified performance issues. Firstly, we moved to an array-based approach for character frequency counting, which offers constant-time access and a predictable memory footprint. Secondly, we implemented bit-level operations to replace costly string concatenations, drastically improving encoding speed. Lastly, we redesigned the Huffman tree to use indices within a vector instead of pointers, leading to better memory locality and cache efficiency.

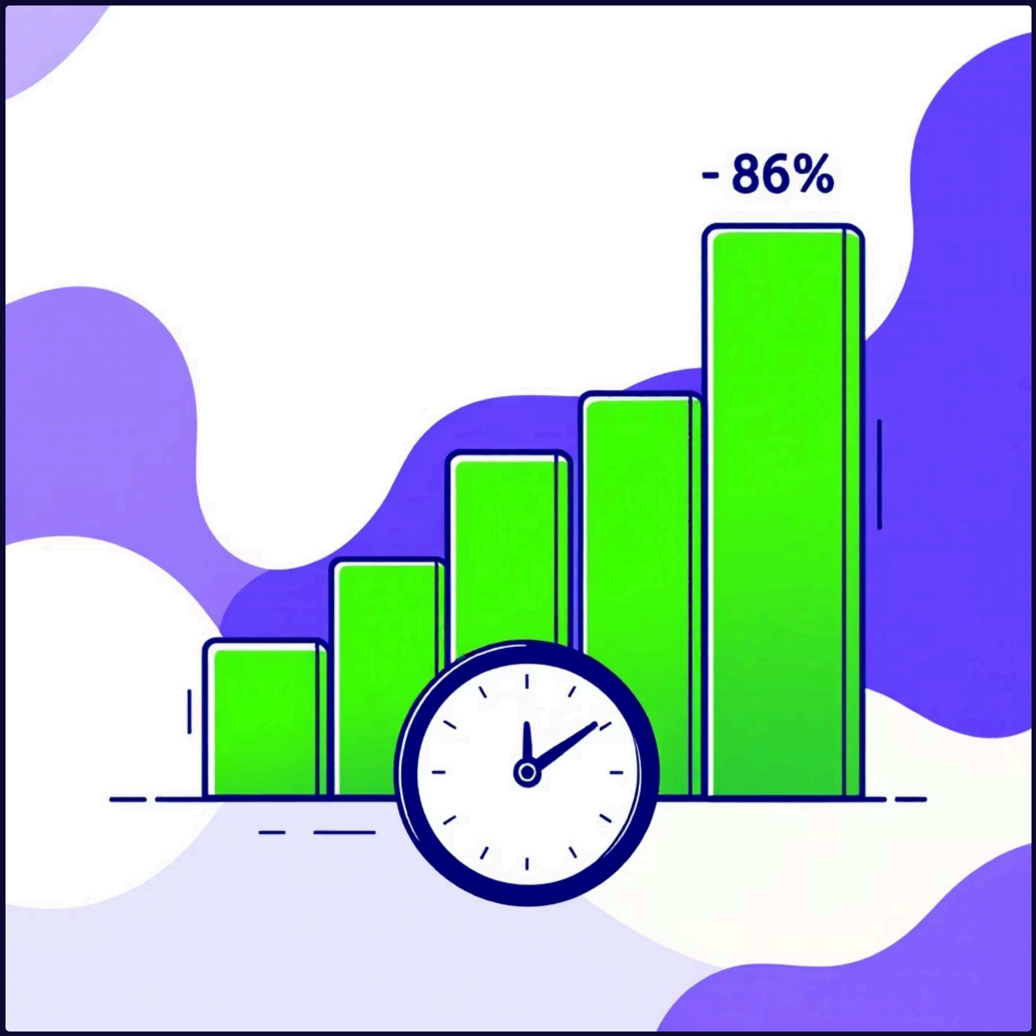
# Speed Performance

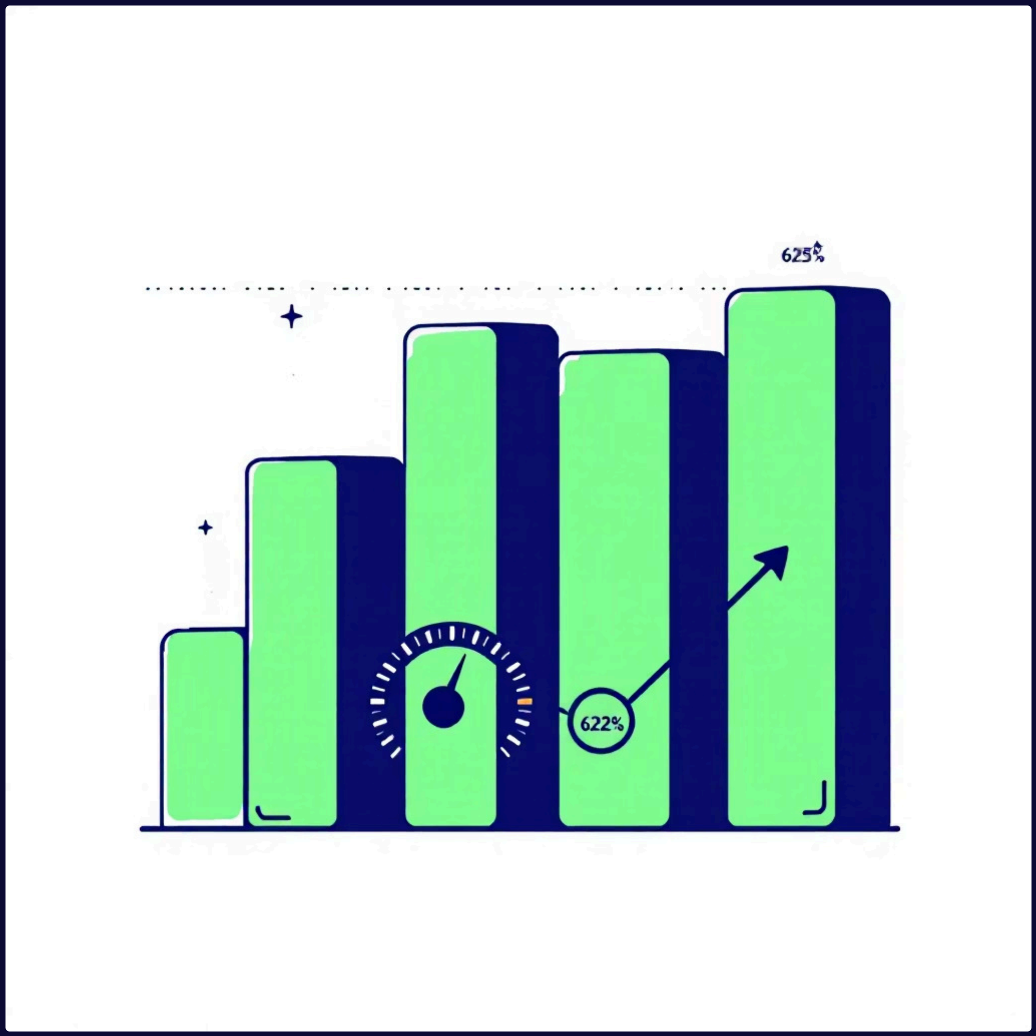


## Encoding Time: 86% Reduction



## Throughput: 625% Increase





Total processing time: 59ms → 30ms (49% reduction)

The speed improvements observed are nothing short of dramatic. Our optimised Version 2 reduced encoding time from 29 milliseconds to a mere 4 milliseconds, representing an 86% reduction. Concurrently, throughput soared from 152 megabytes per second to an impressive 1,100 megabytes per second, marking a 625% increase. Overall, the total processing time was almost halved, moving from 59ms down to 30ms, significantly enhancing efficiency.

## Memory Performance

68%

### Encoding Memory Reduction

From 22.0 MB down to 6.9 MB, a substantial saving in resources during the compression phase.

51%

### Decoding Memory Reduction

Reduced from 29.5 MB to 14.4 MB, demonstrating efficiency across the entire compression-decompression cycle.

**Average: 60% Memory Reduction**

Memory usage was a critical focus, and the results are compelling. During the encoding process, we achieved a 68% reduction in memory footprint, bringing it down from 22.0 MB to just 6.9 MB. For decoding, memory consumption was cut by 51%, from 29.5 MB to 14.4 MB. Cumulatively, Version 2 achieves an average of 60% less memory usage compared to its predecessor, a significant advantage for resource-constrained environments.

## Accuracy: 100% Maintained

### Version 1: 100% Accurate

### Version 2: 100% Accurate

#### Bit-Identical Restoration Verified

Each compressed file was perfectly restored to its original, uncompressed state, bit for bit.

#### Checksum Comparison Passed

Cryptographic checksums of original and restored data matched precisely, confirming data integrity.

#### No Data Loss or Corruption

Rigorous testing confirmed the absence of any data degradation or loss during the compression and decompression process.

All performance gains achieved without sacrificing correctness, a cornerstone of reliable data processing.

Crucially, all the performance enhancements implemented in Version 2 have been achieved without compromising the fundamental requirement of 100% data accuracy. We meticulously verified bit-identical restoration, performed checksum comparisons that passed without exception, and confirmed absolutely no data loss or corruption. This ensures that the optimised algorithm is not only fast and efficient but also entirely trustworthy for any application requiring high fidelity data preservation.

# Performance Comparison

| | | | |
|---|---|---|---|
| Encode Time | 29.0 ms | 4.0 ms | -86.2% |
| Decode Time | 14.0 ms | 25.0 ms | +78.6% |
| Encode Memory | 22.0 MB | 6.9 MB | -68.5% |
| Decode Memory | 29.5 MB | 14.4 MB | -51.3% |
| Encode Throughput | 151.7 MB/s | 1100.0 MB/s | +625.0% |
| Decode Throughput | 314.3 MB/s | 176.0 MB/s | -44.0% |
| Accuracy | 100% | 100% | Maintained |

This comprehensive table summarises the performance comparison between Version 1 and Version 2. As evident, encoding metrics—time, memory, and throughput—show substantial improvements. However, it's notable that decode performance is currently slower, an area identified for future optimisation. Despite this, the overall gains in critical encoding aspects represent a significant leap forward, making Version 2 highly advantageous for many applications.

## Real-World Applications

### Data Archival Systems

Compress data once, then store millions of files efficiently. The 60% memory reduction translates directly into significant storage cost savings for large-scale archives.

### Embedded/IoT Devices

Many embedded and IoT systems operate with severely limited memory (often only a few MB of RAM). A 60% memory reduction is critical, potentially making the difference between feasibility and impracticality.

### Real-Time Compression

In high-throughput processing pipelines and streaming data scenarios, the 86% speed improvement directly enables greater data volume processing and reduced latency.

The optimisations in our Huffman compression algorithm have tangible benefits across various real-world applications. For large-scale data archival, the 60% memory reduction dramatically cuts storage costs. In resource-constrained embedded and IoT devices, this memory efficiency is often a prerequisite for deployment. Furthermore, the 86% increase in encoding speed opens up new possibilities for real-time compression in high-volume data streams, enhancing system responsiveness and throughput.

# Conclusion & Next Steps

## Achievements

- ✓ 86% faster encoding
- ✓ 60% memory reduction
- ✓ 625% throughput increase
- ✓ 100% accuracy maintained

## Future Work

- Optimise decode performance (currently slower)
- Apply batch bit reading techniques
- Implement lookup tables for common codes

Status: Production-ready for encode-heavy workloads.

To summarise, our Huffman compression algorithm optimisation successfully delivered an 86% faster encoding, a 60% reduction in memory usage, and a substantial 625% increase in throughput, all whilst preserving 100% data accuracy. Version 2 is now production-ready for applications with encode-heavy workloads. Our immediate future work will focus on optimising the decode performance, exploring batch bit reading techniques, and implementing lookup tables for frequently occurring codes to further enhance overall efficiency. Thank you for your attention. Are there any questions?