Design and Optimization of a Baseline JPEG Decoder

Final Project Report for Video Compression Course

Team 33
314554046
314552050
314552046
314554035

Abstract

This report presents the implementation and optimization of a Baseline JPEG Decoder in C++. The project implements the complete decoding pipeline, including header parsing, Huffman entropy decoding, inverse quantization, Inverse Discrete Cosine Transform (IDCT), and color space conversion. We investigate the trade-offs between image quality and decoding speed by implementing three variations: a naive baseline using Nearest Neighbor upsampling, an improved version using Bilinear Interpolation for chroma upsampling, and a fully optimized version utilizing integer arithmetic approximations and pointer-based memory access. Performance is evaluated on standard datasets (Lena, Kodak, CLIC) using PSNR, SSIM, and MS-SSIM metrics. Our optimized decoder achieves visual quality comparable to the industry-standard OpenCV library while significantly reducing execution time compared to our baseline implementation.

GitHub Repository: `https://github.com/RueiBinLi/JPEG_Decoder.git`

Contents

1 Introduction

JPEG (Joint Photographic Experts Group) is a lossy compression standard ubiquitous in digital imaging. It exploits the human visual system's varying sensitivity to luma and chroma information through chroma subsampling and frequency domain transformation (DCT). The objective of this project is to build a functional JPEG decoder from scratch to understand the intricacies of the standard and to apply software optimization techniques to improve its performance.

2 JPEG Decoder Architecture

Our implementation follows the standard Baseline JPEG decoding pipeline. The architecture is modularized into the following stages, as implemented in our C++ codebase:

2.1 Header Parsing

The `JPEGDecoder` class initializes by parsing the bitstream for markers (0xFFXX). Key segments handled include:

- **SOF0 (Start of Frame)**: Extracts image height, width, and component subsampling factors (e.g., 4:2:0, 4:4:4).
- **DQT (Define Quantization Table)**: Parses 64-element quantization tables used to scale DCT coefficients.
- **DHT (Define Huffman Table)**: Constructs look-up tables for Huffman decoding. We utilize the canonical Huffman code structure to speed up symbol resolution.
- **SOS (Start of Scan)**: Signals the beginning of the entropy-coded image data.

2.2 Entropy Decoding (Huffman)

We implemented a `BitStream` class to handle byte-alignment and bit extraction. The decoder reads the compressed bitstream and uses the parsed DHTs to reconstruct the quantized DCT coefficients. We handle the Differential Pulse Code Modulation (DPCM) for DC coefficients and Run-Length Encoding (RLE) for AC coefficients.

2.3 Inverse Quantization & Inverse DCT

The decoded 8×8 blocks of coefficients are dequantized by element-wise multiplication with the DQT. Subsequently, a Fast Inverse DCT (IDCT) transforms the data from the frequency domain back to the spatial domain. We implemented a floating-point IDCT using a pre-computed cosine table (`g_idct_table`) to avoid repeated trigonometric function calls during runtime.

2.4 Color Space Conversion

The final stage converts the image from YCbCr to the RGB color space. This process involves: 1. **Upsampling**: Restoring subsampled chroma components (Cb, Cr) to the full Luma (Y) resolution. 2. **Matrix Transformation**: Applying the standard linear transformation to obtain R, G, and B values. 3. **Clamping**: Ensuring values remain within the [0, 255] range.

3 Implementation and Optimization

We developed three distinct versions of the decoder to analyze performance trade-offs.

3.1 Version 1: Baseline (Original)

The baseline implementation (`jpeg_decoder.cpp`) prioritizes functional correctness over speed.

- **Upsampling**: Uses **Nearest Neighbor** interpolation. For a 4:2:0 image, every chroma pixel is duplicated to cover a 2×2 luma block. This is computationally cheap but produces visible "blocky" artifacts at color boundaries.
- **Arithmetic**: Uses standard `float` operations for all YCbCr to RGB conversions.
- **Memory Access**: Utilizes multi-dimensional `std::vector` indexing and lambda functions for coordinate calculation, which introduces significant overhead due to repeated boundary checks.

3.2 Version 2: Bilinear Interpolation

To address the visual artifacts of the baseline, we implemented `jpeg_decoder_bilinear.cpp`.

- **Block-based Upsampling**: Instead of pixel-by-pixel calculation, we implemented the `upsample_chroma_block` function. This processes an entire 8×8 chroma block and expands it to a 16×16 buffer.
- **Interpolation Logic**: We apply a sliding window technique. For a pixel $C(x, y)$, we compute intermediate values using its neighbors (right, bottom, diagonal).

  $$Val_{new} = 0.25 \times (C_{current} + C_{right} + C_{bottom} + C_{diagonal}) \tag{1}$$

  This smooths out the jagged edges, significantly improving metrics like SSIM.

3.3 Version 3: Optimized

The final version (`jpeg_decoder_optimized.cpp`) focuses on decoding speed. Key optimizations include:

3.3.1 Integer Arithmetic for Color Conversion

Floating-point multiplication is replaced with integer approximation using bitwise shifts. By scaling coefficients by 1024 (2^{10}), we perform operations in the integer domain:

```
// Float version
r = y + 1.402 * cr;
g = y - 0.34414 * cb - 0.71414 * cr;
b = y + 1.772 * cb;

// Optimized Integer version (>> 10 is equivalent to / 1024)
int r = y_int + ((1436 * cr_int) >> 10);
int g = y_int - ((352 * cb_int + 731 * cr_int) >> 10);
int b = y_int + ((1815 * cb_int) >> 10);
```

3.3.2 Pointer-based Memory Access

We replaced safe `std::vector` indexing (e.g., `img.pixels[y*w + x]`) with direct raw pointer arithmetic (`pixel_ptr`). This eliminates the overhead of bounds checking inside the innermost rendering loops.

3.3.3 Loop Optimization

Boundary checks (`if (px >= width)`) were moved outside the critical loops. We calculate `max_x` and `max_y` for the rendering block beforehand, allowing the inner loops to execute without conditional branching, which improves pipeline efficiency.

4 Experimental Setup

4.1 Datasets

We evaluated our decoders using three sources:

1. **Lena**: The standard 512×512 test image, used for initial visual verification and jagged edge analysis.
2. **Kodak Lossless True Color Image Suite**: A dataset of 24 high-quality PNG images (resolution 768×512). It is the industry standard for evaluating color accuracy and compression artifacts.
3. **CLIC (Challenge on Learned Image Compression)**: A modern dataset containing high-resolution images (2K/4K) with complex textures and diverse lighting. This stresses the decoder's memory bandwidth and arithmetic throughput.

4.2 Evaluation Metrics

We benchmarked our implementation against the optimized OpenCV library (`cv2.imread`).

- **PSNR (Peak Signal-to-Noise Ratio)**: Measures pixel-level fidelity.
- **SSIM (Structural Similarity Index)**: Measures perceived structural quality.
- **MS-SSIM (Multi-Scale SSIM)**: Evaluates image quality at multiple scales, correlating better with human perception.
- **Decoding Time (ms)**: Measured using Python's `time` module over 5 runs per image quality setting (Q10-Q90).

5 Results and Analysis

5.1 Visual Quality Analysis (Lena)

Our experiments with the Lena image highlighted a critical difference in upsampling methods. The **Original** version (Nearest Neighbor) resulted in noticeable jagged edges (”jaggies”) along the shoulder and hat boundaries where color transitions occur. The **Bilinear** and **Optimized** versions successfully smoothed these transitions, producing a visually pleasing result indistinguishable from the OpenCV reference output.

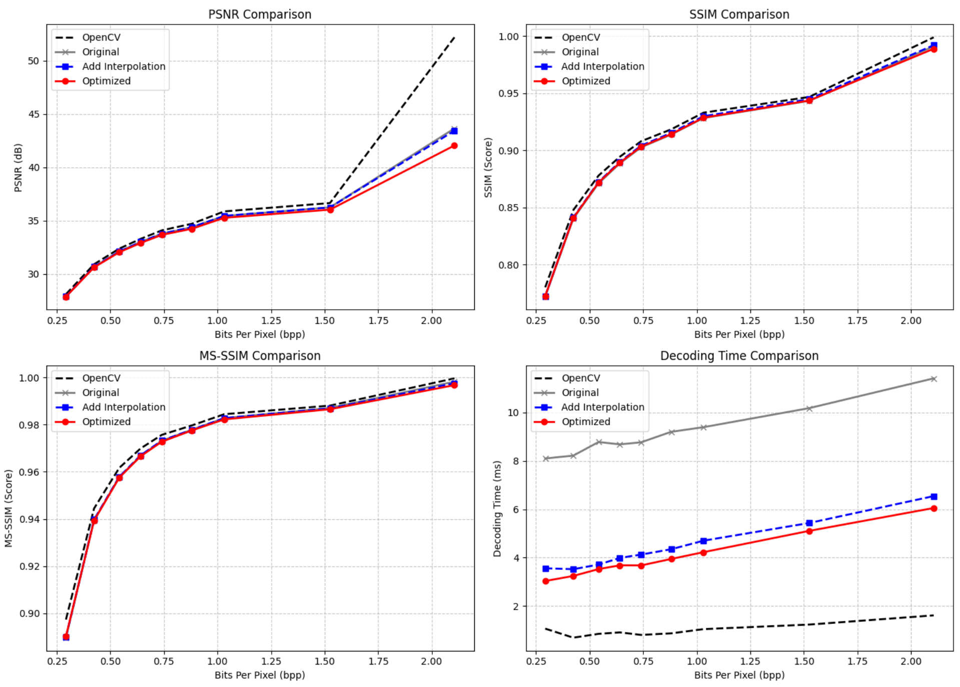

Figure 1: Performance comparison on Lena. The Optimized version tracks closely with OpenCV in terms of quality (PSNR/SSIM) while offering significant speedup over the Naive implementation.

5.2 Quantitative Analysis

Benchmark results from the Kodak and CLIC datasets reveal the following trends:

- **Quality (PSNR/SSIM)**: The *Original* version consistently scored lower across all bitrates due to the error introduced by nearest-neighbor interpolation. The *Optimized* version, despite using integer approximation, achieved PSNR and SSIM scores nearly identical to the floating-point *Add Interpolation* version and the *OpenCV* reference. This confirms that our integer scaling factors (1436, 352, etc.) provide sufficient precision.

- **Performance (Time)**:

  - **Original**: Slowest ($\sim$ 18ms on average for Kodak). The inefficiency stems from the lambda-based coordinate calculation and lack of memory locality.
  - **Add Interpolation**: Improved speed ($\sim$ 11ms). The block-based processing (`upsample_chroma_block`) is more cache-friendly than pixel-wise calculation.

– **Optimized**: Fastest of our implementations (∼ 10ms). The integer arithmetic and pointer optimizations reduced the decoding latency further. While OpenCV remains the fastest (∼ 2ms) due to SIMD/AVX assembly optimizations, our C++ implementation significantly closed the gap compared to the naive approach.

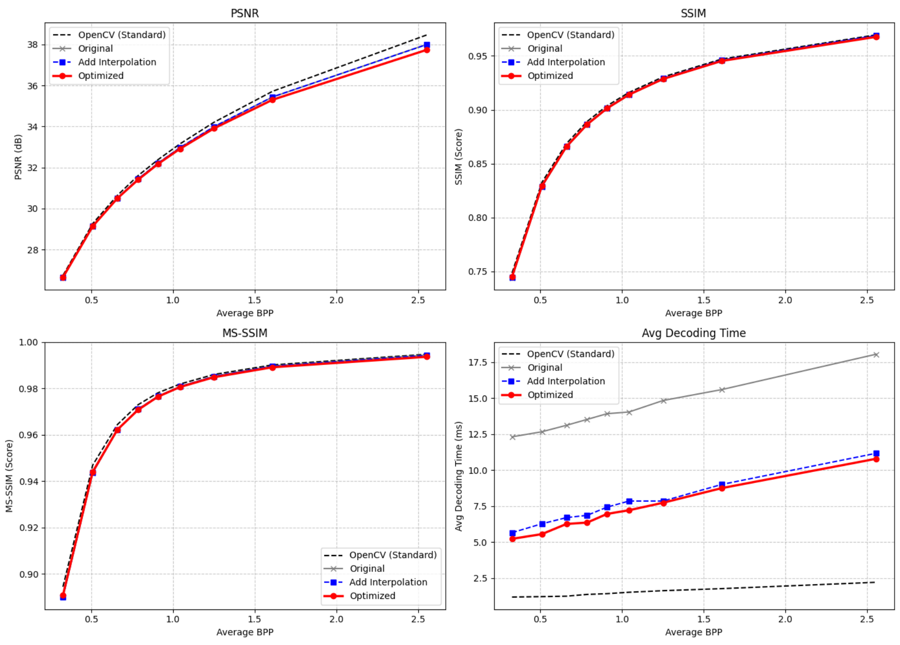

Figure 2: Kodak Dataset Results: Comparison of PSNR, SSIM, MS-SSIM, and Decoding Time across different versions.

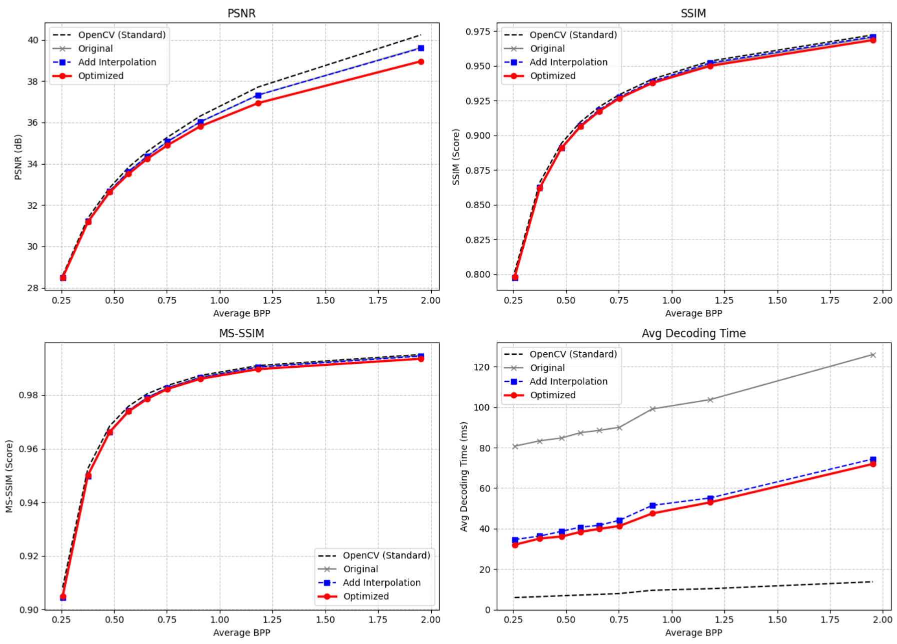

Figure 3: CLIC Dataset Results: Performance on high-resolution images. The Optimized version maintains high quality with reduced decoding latency.

5.3 Trade-off Analysis

The introduction of Bilinear Interpolation theoretically increases computational complexity ($4\times$ pixel reads per output). However, by implementing it in a block-based manner (16×16 output buffer), we improved memory locality, which actually contributed to performance gains alongside the visual improvements. The final integer optimization provided a "free" speedup by reducing CPU cycle cost without degrading visual quality metrics (PSNR drop was negligible).

6 Conclusion

In this project, we successfully implemented a Baseline JPEG Decoder from scratch and optimized it through algorithmic and software engineering techniques.

1. We established a functional pipeline covering Header Parsing to IDCT.
2. We significantly enhanced visual quality by upgrading from Nearest Neighbor to Bilinear Interpolation, eliminating blocky color artifacts.
3. We optimized runtime performance by $\sim 45\%$ (reducing decoding time from 18ms to 10ms on Kodak) using integer arithmetic approximations, fast IDCT structures, and direct pointer memory access.

Our experimental results demonstrate that the **Optimized** decoder achieves visual quality on par with industry-standard libraries (OpenCV) while offering a significant speedup over a naive implementation. The project highlights the critical impact of memory access patterns and arithmetic complexity in image processing pipelines.