1. Introduction

Image compression is a fundamental technique in digital media processing, enabling efficient storage and faster transmission of images without significantly degrading quality. Our project is a Rust-based image compression tool that intelligently detects whether an image is in JPG or PNG format and applies an appropriate compression method.

Unlike conventional image compression tools that apply a uniform approach, our system first determines the image type and then processes it using a format-specific algorithm. The compression works by dividing the image into smaller chunks, applying compression to each segment, and then reconstructing the final compressed image. This modular approach ensures optimized performance and better handling of different image types.

2. Project Overview

Our image compression system consists of four major stages:

2.1 Image Type Detection

Before compression, the system determines whether the image is a JPG or PNG. This is done by analyzing the file's header and metadata. JPG files typically start with the `FFD8` marker, while PNG files have a signature beginning with `89504E47`. The program reads the image's binary data and extracts these signatures to classify the image type accurately.

2.2 Chunk-Based Processing

Once the image type is identified, the image is divided into smaller chunks for processing. This method allows for efficient memory usage and enables parallel processing in future improvements. Each chunk is compressed individually and then combined to reconstruct the final image.

2.3 Compression Algorithm

Depending on the detected format, different compression techniques are applied:

- **JPG Compression:** Since JPG is a lossy format, the system applies Discrete Cosine Transform (DCT) to each chunk, followed by quantization to remove redundant details and achieve significant file size reduction.

- **PNG Compression:** Since PNG contains metadata,colour profiles and other chunks. We remove the data that is not necessary (lossless compression) by using oxipng. It removes any unnecessary data or any duplicated chunk, which in turn would help reduce the size of the PNG.

2.4 Image Reconstruction

After compression, the system reassembles the compressed chunks to reconstruct the final image. This step requires careful handling to prevent visual artifacts and ensure smooth merging of the compressed blocks.

3. Contributions of Team Members

Each team member played a crucial role in the development of the image compression tool.

Kevin was responsible for designing and implementing the **JPG compression algorithm** as well as the **image type detection system**. His work involved developing an efficient method for identifying image formats and optimizing the JPG compression pipeline using DCT and quantization techniques. Debugging the compression output and ensuring smooth chunk integration were key challenges he tackled.

Hanna focused on implementing PNG compression and RLE implementation. However, due to the lack of information and knowledge she couldn't combine the use of the RLE compression towards PNG (out bound issue). Therefore, she change the implementation route to use Oxipng instead. Since PNG use a DEFLATE compression (like a zip), oxipng will re-compresses the image data which is more efficient deflate technique. She then would filter to improve th compression, since png store pixels row by row. Most of the compression is done by Oxipng.

4. Challenges Faced During Development

During development, we encountered multiple technical challenges that required careful problem-solving and debugging.

One of the biggest challenges was ensuring accurate image type detection. Some images contained additional metadata that interfered with our format-checking logic. To solve this, we refined our approach by analyzing multiple header markers and validating them against expected patterns.

Another major challenge was achieving efficient **chunk-based processing**. While breaking the image into chunks improved memory usage, it introduced issues during reconstruction, especially for JPG images. Merging the DCT-processed chunks correctly without introducing visual artifacts required fine-tuning of the compression parameters.

The PNG compression implementation faced issues with the optimization issue. Where the compression would take the image in as a whole since th start , if there's many photo the code will process the images first before compressing, which causes it to not be optimal. However, when i tried to make it into chunks by chunk it would lead to out of bound issue. Which will then turn our code into an infinity loop.

5. Lessons Learned

Throughout this project, we gained extensive experience in image compression techniques and Rust programming.

One key takeaway was the importance of format-specific compression. We realized that lossless techniques like Oxipng work well for PNGs, whereas lossy methods like DCT-based compression are better suited for JPG images.

We also learned a great deal about **efficient memory management** in Rust. Since image processing involves handling large amounts of binary data, optimizing memory usage was crucial to maintaining performance. We used Rust's ownership model to minimize unnecessary allocations and improve processing speed.

Additionally, debugging was a major learning experience. Handling raw binary image data required careful logging and visualization of intermediate outputs. By using debugging tools and testing various images, we improved our ability to identify and fix compression issues.

Finally, working as a team was essential to the success of this project. Clear division of tasks and regular discussions helped us integrate different components smoothly.

6. Future Improvements and Enhancements

While our current implementation successfully compresses JPG and PNG images, several improvements can be made:

1. **Support for additional formats:** Extending the tool to handle GIF and BMP images would make it more versatile.

2. **Parallel Processing:** Implementing multi-threading in Rust would improve performance by allowing simultaneous processing of image chunks.

3. **Advanced PNG Compression:** Figure out the ways to implement RLE into the compression and try to make the code as optimal as much as possible .
4. **Graphical User Interface (GUI):** A user-friendly interface would make the tool more accessible for non-technical users.

7. Conclusion

Our image compression project in Rust successfully demonstrated an efficient approach to reducing image file sizes while maintaining quality. By implementing **format-specific compression** techniques and a **chunk-based processing system**, we achieved significant improvements in compression performance.

Despite challenges related to image type detection, chunk processing, and optimization, we overcame these obstacles through careful debugging and refinement. This project provided valuable insights into **image processing, Rust performance optimization, and teamwork**.

With further improvements, our tool has the potential to become a powerful and efficient image compression solution for a variety of applications