Lossless Single Pass Image Compression with Efficient Error Handling for Thermal Images

Andrey Filippov Nathaniel Callens Jr.
Kelly Chang Bryce Hepner Nikolai Masnev
Elphel, Inc.
1455 W. 2200 S. #205, Salt Lake City, Utah 84119 USA
{andrey,bryce}@elphel.com

August 15, 2022

Abstract

The specific properties of thermal images compared to photographic ones are higher dynamic range (16 bits) and dependence of pixels only on the temperature variations of self-radiating objects. The ambient temperature variations add to the pixel values, not multiply them as in the case of the illuminated scenes. We base our algorithm on the 4-neighbor method and use local context to switch between encoding tables as the expected prediction error depends only on the differences between the known pixels invariant of their average value. This approach allows for building a 2D histogram for the prediction error and the "smoothness" of the known pixels and using it to construct the encoding tables. Table selection only depends on the four-pixel values (so available to the decoder) and does not increase the compressed stream. As a result, we could losslessly compress thermal images to be less than 41% of their original size. The resulting files were approximately 34% smaller than their equivalent PNGs, and 35% smaller than TIFF files compressed with LZW.

1. Introduction

1.1 Overview

The base system is not new, but it will be explained here in order to keep consistent definitions and in case any reader is not familiar with the method.

The idea is based on how images are scanned in originally. Like a cathode-ray tube in a television, the algorithm goes line by line, reading/writing each pixel individually in a raster pattern.

Each pixel, as long as it is not on the top or side boundaries, will have 4 neighbors that have already been read into the machine. Those points can be analyzed and interpolated to find the next pixel's value. A visual demostration of this pattern is given in Figure 1. The goal is to encode the error between that value and the original value, save that, and use that to compress and decompress the image. Even though a possibly larger integer may need to be stored, it is more likely that the guess will be correct or off by a small margin, making the distribution better for compression.

The approach of using the neighboring pixels for compression is not new, as evidenced by its use in ISO/IEC 14495-1:1999 [**?**] and "CALIC-a context based adaptive lossless image codec" [**?**], which were both written more than 20 years before the publication of this paper. Our final implementation differs from these methods, and others, in ways that we found beneficial for thermal images, and in ways others may find to be beneficial as well.

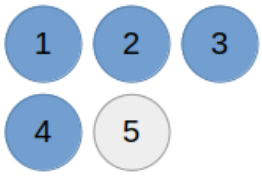

Figure 1: The other 4 pixels are used to find the value of the 5th.

1.2 Background

The images that were used in the development of this paper were all thermal images, with the values in this specific dataset the ranging from 19,197 to 25,935. Total possible values with these sensors can range from 0 to 32,768. Most images had ranges of about 4,096 between the smallest and the largest pixel values. The camera being used has 16 forward-facing thermal sensors creating 16 similar thermal images every frame.

Thermal images are unique in that pixel values will not depend on lighting but solely on the temperature values of the objects they represent. Direct lighting can change these values due to the heat exchange, but the general case is that due to heat conduction, objects will have near uniform temperature across the surface. This creates a need for a different type of compression system, one that is better suited for this different type of data used in the IR spectrum. Thermal images also have large offsets since when the environment heats up, the pixel values increase while the relationship between objects remains almost constant. For example, grass will always be cooler than a similar colored surface due to the different thermal properties, but when the day gets hotter, both surfaces will get hotter. The images are 16-bit because they have to save these larger temperature values, even if they will be shown on a screen in 8-bit format. Normal compression systems work on thermal images, but since they are not optimized for these, we found it necessary to use a different system.

2. Related Work

2.1 PNG

PNG is a lossless compression algorithm that also operates using a single pass system [**?**]. The image is separated into several blocks of arbitrary size, which are then compressed using a combination of LZ77 and Huffman encoding [**?**]. LZ77 operates by finding patterns in the data and creating pointers to the original instance of that pattern. For example, if there are two identical blocks of just the color blue, the second one only has to make reference to the first. Instead of saving two full blocks, the second one is saved as a pointer to the first, telling the decoder to use that block. Huffman encoding is then used to save these numbers, optimizing how the location data is stored. If one pattern is more frequent, the algorithm should optimize over this, producing an even smaller file [**?**]. The Huffman encoding in conjunction with LZ77 helps form "deflate", the algorithm summarized here, and the one used in PNG.

Our algorithm uses Huffman encoding similarly, but a completely different algorithm than LZ77. LZ77 seeks patterns between blocks, while ours has no block structure and no explicit pattern functionality. Ours uses the equivalent block size of 1, and instead of encoding the data, it encodes alternate data which is used to compress.

2.2 LZW

LZW operates differently by creating a separate code table that maps every sequence to a code. Although this is used for an image, the original paper by Welch [**?**] explains it through text examples, which will be done here as well. Instead of looking at each character individually, it looks at variable-length string chains and compresses those. Passing through the items to be compressed, if a phrase has already been encountered, it saves the reference to the original phrase along with the next character in the sequence. In this way, the longer repeated phrases are automatically found and can be compressed to be smaller. This system also uses blocks like PNG in order to save patterns in the data, but instead by looking at the whole data set as it moves along, PNG only operates on a short portion of the text [**?**].

Ours, similarly to PNG, only looks at a short portion of the data, which may have an advantage over LZW for images. Images generally do not have the same patterns that text does, so it may be advantageous not to use the entire corpus in compressing an image and instead only evaluate it based on nearby objects. The blue parts of the sky will be next to other blue parts of the sky, and in the realm of thermal images, temperatures will probably be most similar to nearby ones due to how heat flows.

2.3 Similar Methods

Our prior searches did not find any very similar approaches, especially with 16-bit thermal images. There are many papers however that may have influenced ours indirectly or are similar to ours and need to be mentioned for both their similarities and differences. One paper that is close is "Encoding-interleaved hierarchical interpolation for lossless image compression" [**?**]. This method seems to operate with a similar end goal, to save the interpolation, but operates using a different system, including how it interpolates. Instead of using neighboring pixels in a raster format, it uses vertical and horizontal ribbons, and a different way of interpolating. The ribbons alternate, going between a row that is directly saved and one that is not saved but is later interpolated. By doing this, it is filling in the gaps of an already robust image and saving the finer details. This other method could possibly show an increase in speed but not likely in overall compression. This will not have the same benefit as ours since ours uses interpolation on almost the entire image, instead of just parts, helping it optimize over a larger amount of data. This paper is also similar to "Iterative polynomial interpolation and data compression" [**?**], where the researchers did a similar approach but with different shapes. The error numbers were still saved, but they specifically used polynomial interpretation which we did not see fit to use in ours.

The closest method is "Near-lossless image com-

pression by relaxation-labelled prediction" [?], which is similar in the general principles of the interpolation and encoding. The algorithm detailed in the paper uses a clustering algorithm of the nearby points to create the interpolation, saving the errors to be used later in the reconstruction of the original image. This method is much more complex, not using a direct interpolation method but instead using a clustering algorithm to find the next point.

This could potentially have an advantage over what we did by using more points in the process, but in proper implementation it would become too complicated for our purposes. The goal for us was to have a simple and efficient encoding operation, and this would have too much to process. It also has a binning system like ours, with theirs based off of the mean square prediction error. The problem is that which bin it goes into can shift over the classification process adding to the complexity of the algorithm.

3. The Approach

To begin, the border values are encoded into the system, starting with the first value. Once that is saved, the rest of the values are just saved as the difference to the first. This is not the most technical approach, but it reduces complexity, leaving room for the body of the system.

Huffman encoding performs well on data with varying frequency [?], making it a good candidate for saving the error numbers. Figures 2 and 3 give a representation of why saving the error numbers is better than saving the actual values. This is compounded on the additive nature of thermal images, since temperature values can range greatly, a system is needed that efficiently incorporates that. Most pixels will be off from the predicted values by low numbers since many objects have close to uniform surface temperature or have an almost uniform temperature gradient.

Planar interpolation between the 4 known points is done in order to predict the next pixel value. Because this is an overdetermined system, it will not only output the predicted pixel value, but the square of the residuals (squared euclidean norm of b - Ax) as well [?]. Other than in the title, we use the term "error" to describe the difference between the predicted pixel value and the actual value, and the term "difference" to describe the square root of the residuals. This difference number is a valuable predictor, not of the original pixel value, but of the error that will be outputted. It is not good enough to predict it outright, as there is too much noise and not enough direct correlation, but it can be used to create several different encoding tables that can aid in compression. Another approach was also used in testing, which was using the difference between the maximum pixel value and the minimum. This had similar results, but was not used in the final process since the residuals were already automatically calculated, while the min and max differencing would have to be done in addition to this, further complicating it.

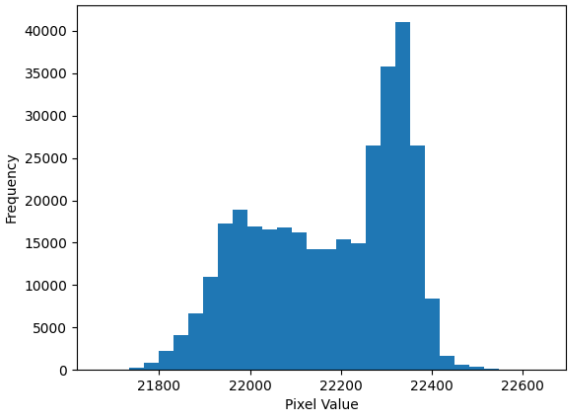

Figure 2: Encoding the Pixel Values

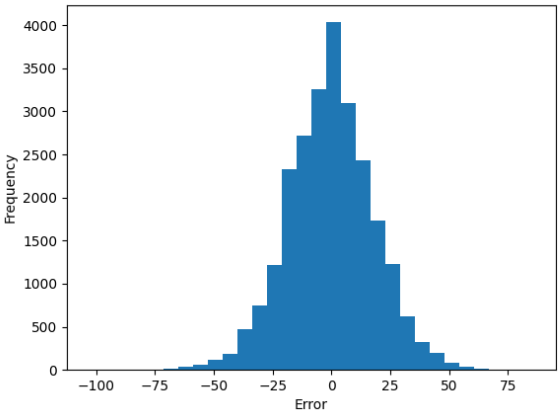

Figure 3: Encoding the Error Values

In order to adjust for objects in images that are known to have an unpredictable temperature (have high difference values), a bin system is used. If a plane is able to be drawn that contains all 4 points, it makes sense that the error will be much smaller than if the best-fitted plane was not very close to any of the points. Something more certain is more likely to be correctly estimated. 5 bins were used with splits chosen by evenly distributing the difference numbers into evenly sized bins. Many of the images had several different bin sizes ranging from 11 in the first category to a difference of 30 as the size of the first category.

An average number between all of them was chosen in order to save space.

This makes the system much better adapted to larger ranges of error values, such as looking at grass or another high frequency surface. The system performs better than a standard system without bins on this data since it is able to optimize better for these larger values. As shown in 4, the median error gets worse as difference increases. By focusing the root of the Huffman tree on the shift in error values, it is possible to get better compression.

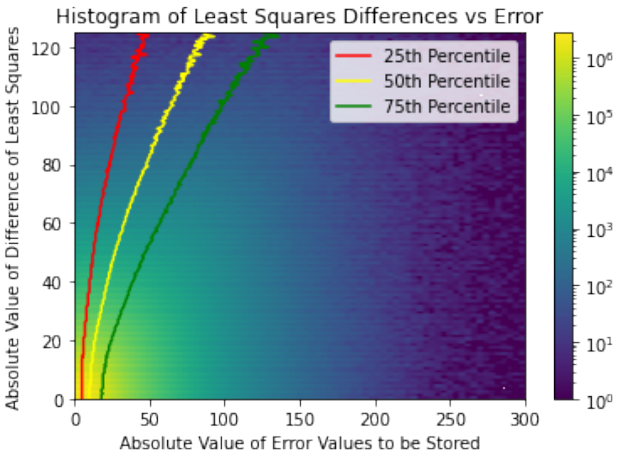

Figure 4: Encoding the Error Values

4. Results

We attained an average compression ratio of 0.4057 on a set of 262 images, with compression ratios on individual images ranging from 0.3685 to 0.4979. Because the system runs off of a saved dictionary, it is better to think of the system as a cross between an individual compression system and a larger archival tool. This means that there are significant changes in compression ratios depending on how many files are compressed at a time, despite the ability to decompress files individually and independently.

When the size of the saved dictionary was included, the compression ratio on the entire set only changed from 0.4043 to 0.4057. However, when tested on a random image in the set, it went from 0.3981 to 0.7508. This is not a permanent issue, as changes to the method can be made to fix this. These are outlined in the discussion section below.

This was tested on a set of a least 16 images, so this does not affect us as much. When tested on a random set of 16 images, the ratio only changed from 0.3973 to 0.4193.

Compression Rates			
Original	LZW	PNG	Ours
100%	61.94%	61.21%	40.57%

Our method created files that are on average 33.7% smaller than PNG and 34.5% smaller than LWZ compression on TIFF.

For estimation of limits of compression, we use information theory. Expected length of source code: $L = \sum_x p(x) l(x)$
$l(x)$ is a length of codeword, corresponding to state x. $p(x) = \frac{n_x}{N}$, n_x - number of elements for state x, N - total number states. From information theory we have: $L \geq H$ Where $H = -\sum_x p(x) \log_2 p(x)$ - Shannon entropy. From the previous formula: $L = \frac{\sum_x n_x l(x)}{N} \geq -\sum_x p(x) \log_2 p(x)$.

From that we have: compression rate $\geq \frac{NH}{\text{uncompr size}}$

After averaging over 224 images, compression limit is 0.37.

5. Discussion

The files produced through this method are much smaller than the ones produced by the others, but this comes at great computational costs in its current implementation. PNG compression was several orders of magnitude faster on the local machine than the method that was used in this project. Using a compiled language or integrated system instead of python will increase the speed, but there are other improvements that can be made.

The issue with `numpy.linalg.solve` was later addressed to fix the potential slowdown. Calculating the inverse beforehand and using that in the system had marginal temporal benefit. `numpy.linalg.solve` runs in $O(N^3)$ for an $N \times N$ matrix, while the multiplication runs in a similar time. [**?**] The least squares method mentioned in this project also has a shortcoming, but this one cannot be solved as easily. The pseudoinverse can be calculated beforehand, but the largest problem is that it is solving the system for every pixel individually and calculating the norm. `numpy.linalg.lstsq` itself runs in $O(N^3)$ for an $N \times N$ matrix [**?**], while the pseudoinverse, when directly implemented, uses more python runtime, adding to temporal complexity.

This compression suffers when it is only used on individual images, which is not a problem for the use cases of this project. The test images came from a camera that has 16 image sensors that work simultaneously. The camera works in multiple image increments and therefore creates large packets that can be saved together, while still having the functionality of decompressing individually. This saves greatly on the memory that is required to view an image. It was therefore not seen necessary to create a different system to

compress individual files as individual images are not created.

A potential workaround for this problem would be to code extraneous values into the image directly instead of adding them to the full dictionary. This has the downside of not being able to integrate perfectly with Huffman encoding. A leaf of the tree could be a trigger to switch from Huffman encoding, and instead use an alternate system to read in the bits. We did not do this, but it would be a simple change for someone with a different use case.