Lossless Single Pass Image Compression by Interpolation

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

June 28, 2022

Abstract

This algorithm operates by scanning through each pixel in a raster pattern, using already scanned pixels to decompress the next pixel's value. By saving the error between the predicted pixel value and the actual value, we were able to 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 LZW compression with TIFF files.

1. Introduction

1.1 Technical Overview

The idea is based off of 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. 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's more likely that the guess will be correct, or off by a small margin, making the distribution better for compression.

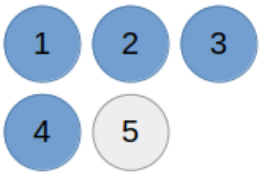

Figure 1: The other 4 pixels are used to find the value of the 5th.

1.2 Background

The images that were used in the development of this paper were all thermal images, with values ranging from 19,197 to 25,935. In the system, total possible values can range from 0 to 32,768. Most images had ranges of at most 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. Everything detailed here can still apply to standard grayscale or RGB images, but for testing, only 16 bit thermal images were used.

2. Related Work

2.1 PNG

PNG is a lossless compression algorithm that also operates using a single pass system[9]. The image is separated into several blocks of arbitrary size, which are then compressed using a combination of LZ77 and Huffman encoding [6]. 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 just contains the location of 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[6]. The Huffman encoding in conjuction with LZ77 helps form "deflate", the algorithm summarized here, and the one used in PNG.

Our algorithm has a similar use of Huffman encoding, 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 [10] 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 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 entire data as it moves along, PNG only operates on a short portion of the text [6].

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 to not use the entire corpus in compressing an image and instead only evaluate it based off of 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 research 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 come close 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" [1]. This method seems to operate with a similar end goal, to save the interpolation, but operates on 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 just saved and one that is not saved but is later interpolated. In this way it is filling in the gaps of an already robust image and saving the finer details. This other method could possibily 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, optimizing over a larger amount of data. This paper is also similar to "Iterative polynomial interpolation and data compression" [5], where the researchers did a similar approach but with different shapes. The error numbers were still saved, but they used specifically polynomial interpretation which we did not see fit to use in ours.

The closest method is "Near-lossless image compression by relaxation-labelled prediction" [2] which has similarity with 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 in order to retrieve the original later. 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 by using more points in the process, but the implementation becomes too complicated and may lose value. The goal for us was to have a simple and efficient encoding operation, and this would have too many things 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. The use of more points could have been implemented into ours too but we chose not to due to the potential additional temporal complexity.

3. The Approach

To begin, the border values are encoded into the system starting with the first value. The values after that are just modifications from the first value. There are not many values here and the algorithm needs a place to start. Alternate things could have been done but they would have raised temporal complexity with marginal gain. Once the middle points are reached, the pixel to the left, top left, directly above, and top right have already been read in. Each of these values is given a point in the x-y plane, with the top left at (-1,1), top pixel at (0,1), top right pixel at (1,1), and the middle left pixel at (-1,0), giving the target (0,0). Using the formula for a plane in 3D ($ax + by + c = z$) we get the system of equations

$$-a + b + c = z_0$$

$$b + c = z_1$$

$$a + b + c = z_2$$

$$-a + c = z_3$$

.

These complete the form $Ax = b$ as

$$A = \begin{bmatrix} -1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

Due to there being 4 equations and 4 unknowns, this is unsolvable.

This can be corrected by making

$$A = \begin{bmatrix} 3 & 0 & -1 \\ 0 & 3 & 3 \\ 1 & -3 & -4 \end{bmatrix}$$

and

$$b = \begin{bmatrix} -z_0 + z_2 - z_3 \\ z_0 + z_1 + z_2 \\ -z_0 - z_1 - z_2 - z_3 \end{bmatrix}$$

.

The new matrix is full rank and can therefore be solved using `numpy.linalg.solve` [7]. The x that results corresponds to two values followed by the original c from the $ax + by + c = z$ form, which is the predicted pixel value.

Huffman encoding performs well on data with varying frequency [8], which makes it a good candidate for saving the error numbers. Most pixels will be off by low numbers since many objects have close to uniform surface temperature or have an almost uniform temperature gradient.

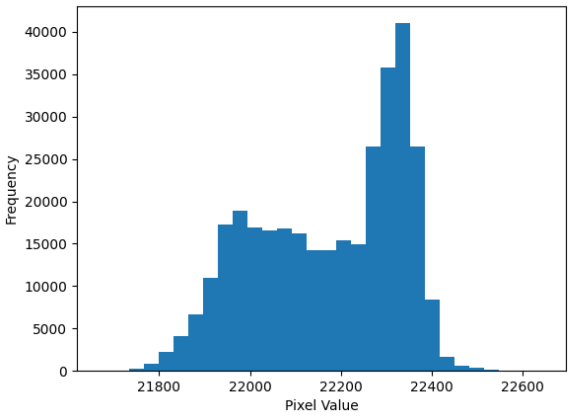

Figure 2: Encoding the Pixel Values

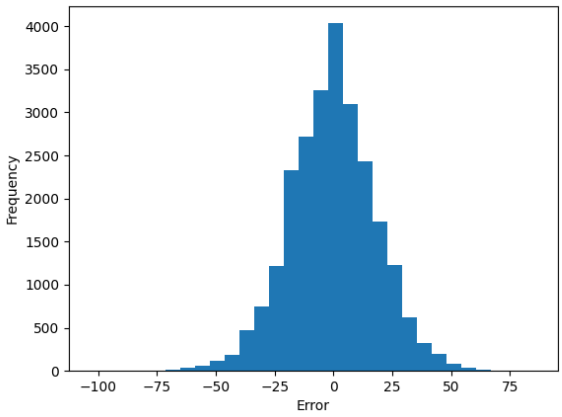

Figure 3: Encoding the Error Values

In order to adjust for objects in images that are known to have an unpredictable temperature (fail the cases before), a bin system is used. The residuals from `numpy.linalg.lstsq` [7] are used to determine the difference across the 4 known points, which is then used to place it in a category. This number is the difference between trying to fit a plane between 4 different points. 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 in this case is likely to be more correct. 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 first category. An average number between all of them was chosen, since using the average versus specific bins had an effect on compression of less than half a percent.

4. Results

We attained an average compression ratio of 0.4057 on a set of 262 images, with compression ratios ranging from 0.3685 to 0.4979. Because the system as it stands 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 large changes in compression ratios depending on how many files are compressed at a time, despite the ability to decompress files individually.

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 just the first 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 detailed 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.

5. Discussion

The files produced through this method are much smaller than the others, but this comes at great computational costs. 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 instead of python will increase the speed substantially, but there are other improvements that can be made.

The issue with `numpy.linalg.solve` was later addressed to fix the potential slowdown, but 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. [4] The least squares method mentioned in this project also has a shortcoming, but this one cannot be solved as easily. The psudoinverse 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 [3], while the psudoinverse when implemented uses more python runtime, adding to temporal complexity.

This compression suffers greatly when it is only used on individual images, which is not a problem for the project it was tested on. 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 would have to be a trigger to not use Huffman encoding anymore and use an alternate system to read in the bits. We did not to do this, but it would be a simple change for someone with a different use case.

References

[1] A. Abrardo, L. Alparone, and F. Bartolini. Encoding-interleaved hierarchical interpolation for lossless image compression. *Signal Processing*, 56(3):321–328, 1997. 2

[2] B. Aiazzi, L. Alparone, and S. Baronti. Near-lossless image compression by relaxation-labelled prediction. *Signal Processing*, 82(11):1619–1631, 2002. 2

[3] J. Alman and V. V. Williams. Algorithm 853: an efficient algorithm for solving rank-deficient least squares problems. *ACM Transactions on Mathematical Software*, Vol. x, No. x,, 20xx. 4

[4] S. Blackford. LAPACK Benchmark. http://www.netlib.org/lapack/lug/node71.html, Oct. 1999. Accessed: 2022-6-23. 4

[5] M. DÆhlen and M. Floater. Iterative polynomial interpolation and data compression. *Numerical Algorithms*, 5(3):165–177, Mar 1993. 2

[6] L. P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. https://www.w3.org/Graphics/PNG/RFC-1951, 1996. Accessed: 6/14/2022. 1, 2

[7] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. 3

[8] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, Sept. 1952. 3

[9] Mark Adler, Thomas Boutell, John Bowler, Christian Brunschen, Adam M. Costello, Lee Daniel Crocker, Andreas Dilger, Oliver Fromme, Jean-loup Gailly, Chris Herborth, Alex Jakulin, Neal Kettler, Tom Lane, Alexander Lehmann, Chris Lilley, Dave Martindale, Owen Mortensen, Keith S. Pickens, Robert P. Poole, Glenn Randers-Pehrson, Greg Roelofs, Willem van Schaik, Guy Schalnat, Paul Schmidt, Michael Stokes, Tim Wegner, Jeremy Wohl. Portable Network Graphics (PNG) Specification (Second Edition). https://www.w3.org/TR/PNG/, Nov. 2003. Accessed: 6/23/2022. 1

[10] Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984. 2