Chimera compression

By: Isaac Schmidt Jonsen

As computing evolves, more and more data is being moved and used in wild new ways. One area of computing that is starting to require more data than modern channels can handle is high quality video and images. With 4K and even 8K video on the horizon it is becoming clear that modern compression algorithms might not be able to keep up with the amount of data used to display these high quality images. When rendered in RGB, 4K images can require more than 10 mb of data. This is a ton of data for a single image and when considering the rise of mobile computing, 10 mb is far too large to even consider any lossless algorithms. This is the impetus for modern video and image compression. The focus of this paper is to review a compression scheme introduced early this year.

Chimera Algorithm

The Chimera algorithm is a lossy compression scheme introduced in the paper titled “Chimera: A New Efficient Transform for High Quality Lossy Image Compression” by Walaa Khalaf, et al. Published on March 3rd 2020, this paper suggests comparing parts of the image to a dictionary of masks and reducing the image to three sets of coefficients. The paper reported that this proposed algorithm outperformed JPEG and JPEG 2000 on multiple metrics for eight different images. The paper does not, however, report any real shortcomings of the algorithm and there don’t appear to be any real reviews of the paper. I was unable to find explicit code online and wrote an implementation to the best of my ability.

The Chimera algorithm works by breaking up the image into 4x4 squares and comparing these squares to a library of masks. The paper suggests creating a library of 256 masks by sampling 16 basic shapes and creating 4x1 vectors. Each vector is then transposed and multiplied by all 16 vectors resulting in a library of 256 4x4 matrices. Many of these masks were useless for various reasons so suitable replacements were suggested for each. Every 4x4 block of pixels in the image was then broken down into 9 coefficients, 3 for each color channel. As each channel is worked on in the same way it is easier to simply work through the algorithm in grayscale and only work with the three coefficients. The three coefficients were:

- A = The minimum element of the block
- B = The difference between the maximum element and minimum or max-min
- C = The index of the mask that is most similar to the block

A and B are self explanatory and trivial to find. C however requires the block to be normalized so that entries are floats between 0 and 1. This normalized block can then be compared to each mask. The paper suggests finding the R correlation between the block and the masks to find the mask that gives the least error. This seemed ambiguous so my code included trying to minimize the Frobenius norm between the matrices. These three coefficients use 1.5 bits per pixel which corresponds to a compression ratio of roughly 5.3:1. The data can be decompressed using the simple formula:

$$block = A + (B * mskLib[C]) \tag{6}$$

Where block is the reconstructed 4x4 block, A, B, and C are the respective coefficients, and mskLib is the library of masks indexed at C. My code was written in python and implements the algorithm on grayscale images.

Before going over the results it should be mentioned that I am not a computer science major with extensive knowledge of computer theory. It is obvious that a seasoned programmer could write a program that utilizes multiple cpu cores and reduces runtime by a significant amount. In addition, the program was run on a very low power laptop which inflated runtimes even more.

The following pages contain multiple images of different sizes along with the corresponding Chimera compression, and a 50% quality JPEG compression. In each row the first image is the original, the second the chimera compression and reconstruction, and third is JPEG compression and reconstruction.

Img 1 Runtime: 1,920 seconds

1920x1080 (same size as images in the paper)

Img 2 Runtime: 1,896 seconds

1800x1200

Img 3 Runtime: 495 seconds

924x614

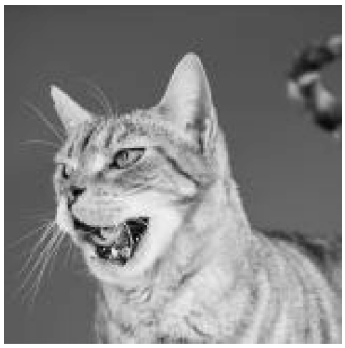
Img 4 Runtime: 232 seconds

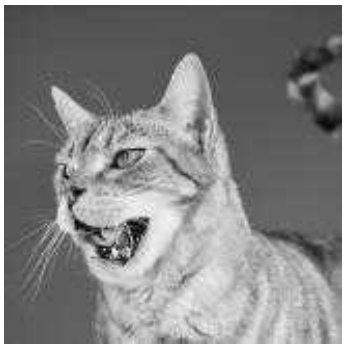
512x512



Img 5 Runtime: 22 seconds

163x163





Ssim of original vs compressed	CHIMERA	JPEG
Img 1	.961	.960
Img 2	.935	.957
Img 3	.918	.975
Img 4	.892	.951
Img 5	.879	.950

The first thing that jumps out are the runtimes on each image. Keeping in mind that this code is unoptimized, if this algorithm were to be professionally implemented, we would see runtimes significantly. That being said, even a significant reduction in runtime would leave a compression algorithm that can take over a minute on a single image. Especially in an age where image processing and compression are expected to be more or less instant, this algorithm is unusable. The majority of the runtime is taken up by the determination of coefficient C. Comparing every 4x4 block in an image to every mask in a 256 maks library will invariably take a long time.

The next important takeaway is the reduction in quality as the image gets smaller. This may seem obvious after the fact, but given that the masks are all based on basic shapes and slopes, the masks become less relevant to each block as the image starts to try to describe greater detail in each 4x4 block. In other words, if we look at the most extreme case of a 4x4 image, it is very unlikely that it would share any similarity to one of the masks.

As a side note, the paper claims that it compares compression to JPEG using a 5.3:1 ratio (the same as Chimera). This ratio would correspond to much higher quality than what I used for the JPEG comparison. Even using a much lower quality JPEG image, all of my JPEG Ssim values were higher than in the paper. I was unable to recreate the low values seen in the paper. I decided to compare the Chimera algorithm to a lower quality JPEG image that matched the Chimera quality rather than the compression ratio. If I were to compare to 3.5:1 JPEG it would be about as close to lossless as JPEG gets and would be a worthless comparison.

Extension

There are a few minor improvements that I came up with. The first and easiest to implement had to do with the scaling factor in reconstructing the image. Due to some of the ambiguity in the algorithm, the only way I could properly compute the reconstructed image was to divide the mask by 24 in reconstruction. This wasn't specifically mentioned in the description of the algorithm. However, in setting up the code this way I noticed that changing this scaling factor could "smooth" or "sharpen" the final image. I found that dividing by 30 rather than 24 produced a higher Ssim on some images. For example, when using 29 for the smallest image the Ssim went from .879 up to .891 and the image is shown below:

Using 29 Using 24

If the runtime of the algorithm could be sufficiently reduced, it would make sense to compare maybe 10 to 20 different values and use the value that maximizes the Ssim when reconstructing the image. Obviously this method wouldn't work given only the compressed data, so possibly appending another coefficient "D" that gives the best scaling factor. It is also worth mentioning that using a larger "D" value corresponds to a smoother image within each block. This is a result of the difference between each pixel inside of the block being reduced. This scaling does

breakdown if the "D" value gets too large as it becomes essentially the same as setting the whole block equal to coefficient A.

Another improvement that seems obviously and is briefly mentioned in the paper is expanding the library. The library could be expanded to contain masks with finer detail. As it stands, the masks only describe basic details and as we've seen, this breaks down as the resolution of the image decreases. This again relies on finding a way to reduce the runtime of the algorithm as adding more masks would add significant runtime to an already taxing algorithm. In addition to adding time to the algorithm, expanding the library would result in coefficient C requiring at least 9 bits per block which is a weird number of bits. Although a single additional bit may seem insignificant it would correspond to roughly 65 kb more for a 4k image. Implementing this improvement would require extensive research to develop another 256 quality masks that would ideally be individually created rather than generated by vectors.

The last improvement that could perhaps help would be to implement variable block sizes. If the algorithm were able to split the image into different size squares while still comparing them to the general pattern of the masks, one could reduce runtime at the cost of potentially introducing more error. In theory the code could immediately assign the first mask to all blocks that are reasonably uniform. This would mean that large sections of white space could be instantly assigned to the correct mask without requiring comparison to the whole library. This would reduce the amount of time spent calculating coefficient C. This kind of scheme, while increasing efficiency, would introduce error along the edges of the image and would add a significant amount of complexity to the code. If a clever enough programmer could effectively implement a variable block size, it would also perhaps allow the reconstruction to work similar

to JPEG 2000 with the ability to show a low quality image that improves as more data is read. This improvement seems the least likely to work as it requires some clever programming and might actually worsen the runtime due to its complexity. It would also require a more complex encoding scheme as Huffman encoding would no longer be achievable. All things considered, adding variable block size is more experimental than the other suggested improvements and is also more related to the code than the algorithm itself.

Conclusion

The paper as a whole seems very intentionally misleading. The fact that it does not provide code to review helps to hide the runtime of the algorithm. In addition, the deliberate choice to only show the results of high definition images ensures that the reader won't notice that the algorithm is much worse on lower quality images.

When we think about how the future of image compression might look it seems natural to think that new and exciting algorithms might overtake the old in popularity. It seems to be the exact opposite of this in fact. Over the past 20 years we have seen algorithms that make moderate improvements over JPEG but as time goes on, JPEG is becoming more and more synonymous with image compression just as google has become with search engines. We are also seeing more and more algorithms being developed that claim to be the next big thing in compression. It is my belief that JPEG is here to stay and when trying to implement lossy image compression into software or websites, a developer need not know much more than how to work with JPEGs.