ABSTRACT

Image processing has numerous applications in the technology field. With the advancements in image capturing technology for example, cameras and video, issues in storage and performance become a hindrance. Therefore, managing image quality and image processing are important. In their original state, images or image streams can take up vast amounts of storage resulting in less efficient image data transfer. However, we propose a different technique in compressing an image. Our image compression technique involves leveraging the processing power of the Graphics Processing Unit (GPU).

INTRODUCTION

Image compression is a way in which an image file is changed so that it takes up less space than its original file. Compression technique work on reducing size without degrading quality. In this project we intend to improve upon compression techniques through using the GPU. Image compression is useful in many fields including medical.

In areas with heavy computation applications, such as MRI, CT Scans and diagnostic examinations. These areas that have large data storage demands, but often times have memory limited challenges. In these cases, utilizing the GPU helps to relieve processing demands of the Central Processing Unit, which in turns makes it more effective

RELATED WORKS

This project is most similar to JPEG compression algorithm. Instead of using a BMP, which is considered to be outdated, a JPEG file is used. In that instance they use Huffman's encoding and decoding algorithm to compress and decompress their images.

PROCESS

First and foremost, our image compression technique requires us to read an image into the CPU as a bmp file. BMP, also known as bitmap is a file format created by microsoft to store pixel color information. A single pixel in a bmp usually can represent up to 256 (2^8) different colors in its 8 bit format, more colors can be represented but comes with the trade off of requiring more storage. Bmps can range from 1 bit to 32 bit colors. Most bmps in their original state are uncompressed and have lossless image quality which make them ideal for standalone applications. This large image size is a result of bmps not having a common compression technique between them.

When the cpu reads the bmp file, we attain the x and y dimensions of the image, along with the size of the file. This information comes from the two headers in a bmp file, which describes the details of the file; such as, file size and picture dimension, and a color table. This is important because from this we can categorize every pixel into a 2-d array. Each pixel in this 2-d array has a dataset associated with it that comes in the form of 4 integer values; red, green, blue(RGB) and count. The values of RGB determine the color of a pixel. Note that the count variable is not an original variable found in bmp files. The value for counter plays a role in our technique of compression and decompression. This is the reason as to why we chose a bmp file format. Because, we can employ parallelization through the

GPU ,in order to improve the performance of the compression.

In the GPU we calculate the number of cores by using the maximum number of cores, and through trial and error we figured out how many cores and threads are optimal for our technique. Each row of the 2-d array is then sent to separate cores in the GPU. A single pixel is then selected in the row and the counter value of that pixel is increment if and only if the next pixel in the sequence holds the exact same RGB values. This process is then repeated on all following pixels until a pixel that has a different RGB value is found. Pixels that have the same RGB as the selected pixel is have their array positions erased. This process is then repeated on all the other cores simultaneously. After this operation the image has become compressed.

To decompress this image in the GPU the reverse of this operation is done. Each separate row of the 2-d array representing the image is sent to individual cores. The array is then read until the iterator comes across a pixel with a counter value. If the counter value is greater than one, new pixels with the same RGB value are added to the right of the current pixel with their counter variables set to zero. The number of pixels added this way are determined by the counter value. The value of counter for the current pixel is then set to zero. If the counter is zero then the iterator does nothing. This process is repeated for all pixels until all pixels have counter zero.

RESULTS

There are no results to present due to errors caused by our lack of knowledge of the C programming language. However, we expect that there is to be a compressed bmp file at the end of our compression. In the worst case there should be no compression due to the all the pixels having different RGB values. However, in the average case, we can expect there to be some compression. In the best case the whole image can be reduced to a single pixel.

CONCLUSIONS

In the end although we have no results, we conclude that the compression can be improved further. To avoid the worst case scenario where each pixel being a different RGB value, the comparison can be improved. The comparison can be altered so that the pixels being compared should be considered ‘similar’ if they are within a reasonable deviation from themselves. This creates more overhead since an additional calculation is needed, calculating a percentage, instead of just a equality check.

REFERENCES:

Cickovski, T. (n.d.). *CUDA Tutorial*. Retrieved April 24, 2017, from http://users.cis.fiu.edu/~tcickovs/CUDA_Tutorial.pdf

Tan, L. (n.d.). *Image File Formats*. Retrieved April 24, 2017, from https://pdfs.semanticscholar.org/9706/ccca02e875331113035bc284937b7db551f1.pdf

Zeller, C. (2011). CUDA C/C Basics. Retrieved April 24, 2017, from http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

Patel, P., & Wong, J. (n.d.), et al. JPEG Compression Algorithm using CUDA. Retrieved April 24, 2017, from http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/JPEG_report.pdf