Parallelized Quadtrees for Image Compression in CUDA and MPI

Aidan E. Jones

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2024

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial
fulfillment of the requirements for graduation from the
Honors Program of Liberty University.

____
Mark S. Merry, Ph.D.
Thesis Chair

____
Frank Tuzi, Ph.D.
Committee Member

____
Emily C. Knowles, D.B.A.
Assistant Honors Director

____
Date

Abstract

Quadtrees are a data structure that lend themselves well to image compression due to their ability to recursively decompose 2-dimensional space. Image compression algorithms that use quadtrees should be simple to parallelize; however, current image compression algorithms that use quadtrees rarely use parallel algorithms. An existing program to compress images using quadtrees was upgraded to use GPU acceleration with CUDA but experienced an average slowdown by a factor of 18 to 42. Another parallelization attempt utilized MPI to process contiguous chunks of an image in parallel and experienced an average speedup by a factor of 1.5 to 3.7 compared to the unmodified program.

Keywords: image compression, quadtree, CUDA, MPI, parallelization, Python

Parallelized Quadtrees for Image Compression in CUDA and MPI

In today's age of information, both space and time are resources that are a concern for software developers and users alike. Data compression is an area of software development in which the software developer seeks to reduce the space footprint of data through various means, including removing redundant or unnecessary data. One source of information that can fill significant amounts of space is visual data, or images, which is the natural focus of image compression algorithms. Many such algorithms exist, though most do not use a type of data structure known as a quadtree, despite its usefulness as a method of representing an image for non-compression purposes and its potential utility as a method to compress similarly-colored 2-dimensional regions of an image. There is, however, a computer program that does use quadtrees as a way to reduce local redundancy within image data, although it is not the final step but is rather a precursor to the data being processed by Lempel-Ziv-Markov chain algorithm (LZMA) and it is also quite slow, taking nearly ten seconds to process a moderately-sized image on current consumer hardware. As such, the primary aim of this project was to upgrade the existing quadtree image compression algorithm with modern parallelization and GPU-acceleration techniques.

Literature Review

Substantial work has been conducted in the past regarding the usage of quadtrees for image compression, as has some work on attempting to create parallelized versions of quadtrees, though most of the work concerning quadtrees is focused on using the data structure to facilitate fast querying of sparse point sets. None of the current standard image formats use quadtrees for compression but rather use discrete cosine transform or some variant of zip compression. Sources from the previous 15 years were preferred, although in some cases much older sources were included due to their authority. Specifically concerning modern image formats, technical specifications or other information published by the original designers were selected.

Modern Image Formats

Several image compression methods are used in current image formats. The most common image formats are Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), Portable Network Graphics (PNG), and, more recently, WebP (Google, 2023).

Graphics Interchange Format

According to the technical specifications for GIF files (CompuServe, 1990), images encoded as a GIF file consist of multiple blocks of data, most of which have a predefined size. The format of a GIF file consists of a header, a logical screen descriptor, an optional global color table, any number of data blocks, and a trailer. Each individual data block is either a graphic block or a special purpose block, which includes both comments and application specific extensions. Graphic blocks consist of an optional control extension, then either a plain text extension for displaying text on the contained image, or a table-based image, which consists of an image descriptor, which describes, among other things, the dimensions of an image and whether or not the image uses the global color table or a local color table. The actual image data for table-based image blocks is stored in a series of up to 256-byte blocks which contain a one-byte field to encode length and then up to 255 bytes of data compressed using lossless variable-length-code Lempel-Ziv-Welch (LZW) compression. Multiple graphic blocks in conjunction with timing data included in graphic control extension blocks can be used to create an animated effect (CompuServe Incorporated, 1990).

Joint Photographic Experts Group

The compression algorithm used for JPEG images divides the image data into 8 x 8 blocks which are then processed using the discrete cosine transform (DCT) (Hudson et al., 2017). As large portions of the image data after being processed by DCT will have a value of zero or close to zero, the data can be further compressed by encoding it as a sequence of value pairs, where the first value is the number of data points until the next non-zero data point, and the second value is the number of bits required to represent the value of the next non-zero data point. Each value pair is then followed by the value of the corresponding non-zero data point. Finally, the sequence of value pairs is compressed with 2D Huffman coding for further compression gains (Hudson et al., 2017).

Portable Network Graphics

The PNG image format was originally designed with the intent of completely replacing GIF, due to its many shortcomings (Adler et al., 1996). The most notable shortcomings of GIF at the time of PNG’s inception were the legal troubles surrounding its use of LZW, a patented algorithm, no support for truecolor images, and minimal support for transparency. PNG addressed those shortcomings while also adding further benefits in the form of error checking and other measures to prevent transmission errors as well as allowing sufficient room for future extension of the format while also retaining interchangeability (Adler et al., 1996).

The current specification (Adler et al., 2023) for the PNG image format defines a PNG image as consisting of the PNG signature, which defines the file as a PNG and includes countermeasures against being interpreted as a text file or certain sequences of bytes being changed by the operating system, one IHDR chunk, which defines important information such as image dimensions, color type, and underlying compression method, an optional PLTE chunk, which defines an up to 256 entry color palette, any number of IDAT chunks, which contain the raw image data compressed using DEFLATE, a variation of LZ77 with Huffman coding, and one IEND chunk, which marks the end of the PNG file. The standard also allows for ancillary chunks, chunks that are not strictly necessary for a decoder to display an image, to be included in specific portions of the PNG file depending on the chunk type (Adler et al., 2023).

WebP

The WebP image format was designed to further compress images compared to PNG or JPEG in order to reduce the amount of time for web pages to load, especially in the context of a mobile device (Google, 2023). Furthermore, the WebP format supports both lossy and lossless compression, full transparency, and true-color animations, all of which are existing features or improvements on existing features of the primary image formats: JPEG, PNG, and GIF. The lossy compression algorithm is based on key frame encoding of VP8, a video compression format designed by On2, which is now owned by Google. The underlying compression method for lossy WebP is, DCT, the same as used in JPEG, although further improvements are made by dividing the image into blocks and predicting block content based on prior data before compressing with DCT and then compressing with arithmetic entropy encoding instead of Huffman encoding (Google, 2023).

Lossless WebP, similarly to PNG, uses a variant of LZ77 with Huffman encoding; however, the image data is subjected to a variety of prediction transforms above and beyond the ones specified in the PNG specification (Google, 2023). Lossless WebP is also automatically able to switch the compression to use a local palette of 256 colors if there are that few unique colors in the image, or it can use color cache coding to further compress the image with a reconstructible dynamic palette. WebP also offers a hybrid format with lossy color and lossless transparency, which yields an image sixty to seventy percent smaller than the corresponding PNG (Google, 2023).

Quadtrees

Quadtrees, similarly to the more well-known binary tree, are a hierarchical data structure consisting of nodes that are either internal nodes, that is, nodes that have child nodes, or leaf nodes, and are typically defined such that each node has exactly zero children, in the case of a leaf node, or four children, in the case of an internal node (Hunter & Steiglitz, 1979). Quadtrees whose internal nodes always have exactly four child nodes are able to recursively decompose a dataset that represents a 2-dimensional space (Samet, 1984). This property allows quadtrees to be used for efficient storage and querying data whose key has two distinct portions, such as a point on a 2-dimensional coordinate plane.

Similar to a binary tree and other related data structures, a quadtree can also be stored succinctly in an array instead of being scattered throughout a computer’s memory (de Bernardo et al., 2023). Assuming that each internal node has exactly four children and that there is only one level of the tree that does not have internal nodes, this is the most efficient way to store a quadtree in memory, as it eliminates the overhead associated with each node having four pointers, one for each child node.

Image Representation

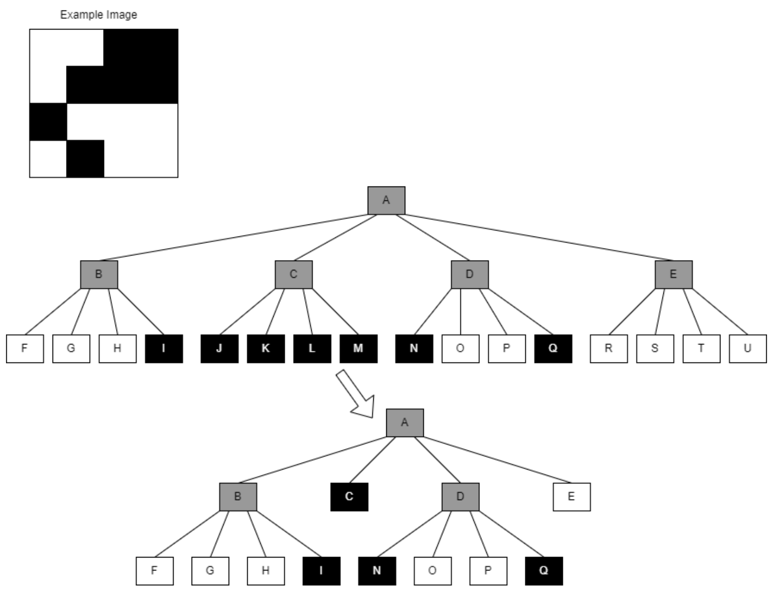
Due to the aforementioned properties, quadtrees are well suited to represent the pixel data of an image for certain tasks in a way that minimizes memory consumption and computational complexity for those tasks (Hunter & Steiglitz, 1979). A quadtree can be used to represent an image by recursively dividing the image into four sections until each section is a single pixel and can no longer be subdivided (Hunter & Steiglitz, 1979; Shusterman & Feder, 1994). If the quadtree is stored succinctly in an array, further memory savings can be made when representing an image, since the size of the image is known before constructing the quadtree and can be used to calculate the amount of data to pre-allocate in a contiguous chunk of memory.

Image Compression

The aforementioned method in which quadtrees represent an image through recursive decomposition of the pixel data lends itself well to image compression, due to the computational simplicity of traversing the tree and the ease of comparing neighboring pixels. One can compress an image either by working from the bottom up, in which case one would compare sibling nodes using a similarity heuristic and then assigning the average value to the parent node, or, alternatively, one can work from the root node downwards, assuming each internal node has already been populated with the average value of its child nodes, and subdivide nodes based on a difference heuristic (Hunter & Steiglitz, 1979; Klinger & Dyer, 1976; Samet, 1984). The choice of heuristic as well as compression direction and number of iterations affect the efficiency of compression and whether the algorithm results in lossy or lossless compression. See Figure 1 and Figure 2 for a visual comparison of bottom-up versus top-down compression.

Figure 1

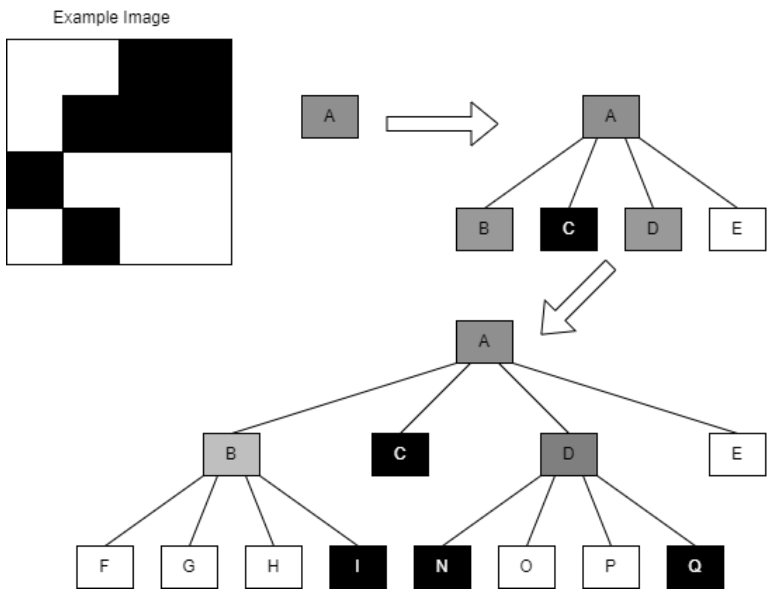
Visualization of Bottom-Up Quadtree Compression



Note. Grey nodes in this figure have no specific value assigned to them.

Figure 2

Visualization of Top-Down Quadtree Image Compression



Note. Grey nodes in this figure are assigned the average value of their child nodes.

Lossless Compression. In lossless compression, the original image can always be perfectly reconstructed from its compressed form. With quadtrees this type of image compression is achieved by combining sibling nodes only when the value of each sibling node is equal to the value of each other sibling node. While lossless compression has some use cases, for images it is typically more useful to use lossy compression due to needing higher compression ratios and the general consideration that many images, especially photographs, frequently have large amounts of minor details that would not significantly affect the perception of the image if they were to be removed (Al Sideiri et al., 2020).

Lossy Compression. While lossy compression methods are typically able to achieve higher compression ratios, a naïve approach, such as one in which sibling nodes are combined only if each of their values is within a predefined threshold of the mean of their values, will typically perform poorly in terms of rate-distortion, a key metric which is typically defined as the ratio between the number of bits per data point in the compressed data and the expected distortion of the data when decoded and is commonly used to compare the results of different image compression algorithms (Blau & Michaeli, 2019; Shukla et al., 2005; Shusterman & Feder, 1994). One way to improve performance is by combining neighboring nodes that do not share a direct parent (Shukla et al., 2005), while another method of improving performance involves using different combination thresholds that are optimized for different levels of a quadtree (Shusterman & Feder, 1994). Yet another method of improving performance, specifically when subdividing instead of combining, involves choosing a set number of iterations to run the algorithm and prioritizing regions of the image with high detail to be subdivided further into more granular detail (Inspiaaa, 2023). Finally, when quadtrees are used in video compression, the performance of the compression algorithm can be further enhanced by updating combination thresholds based on previous frames of the video (Gao et al., 2016; Sullivan & Baker, 1994).

Parallelization

One significant way in which an algorithm can be sped up is through parallelization, which is a technique that increases the efficiency of a program through writing the implementation of an algorithm in such a way that it can be duplicated and executed simultaneously on a cluster of interconnected computers or hardware that is designed to be parallel, such as a graphical processing unit (GPU) or a multicore variety of central processing unit (CPU) (Asanovic et al., 2006). Due to the wide variety in level of support and methods of implementation, fully leveraging parallelism for increasing the efficiency of a program is generally tied to the architecture for which one is writing the program (Asanovic et al., 2006).

Parallel Computing Technologies

Options for parallel computing technology primarily consist of either clusters of computing devices connected to each other in some sort of network or specialized hardware, which typically is either a multicore CPU, a GPU, or a general-purpose GPU (GPGPU) (Dilliwar et al., 2013; Hernandez-Lopez & Muñiz-Pérez, 2022; Zhang et al., 2011). In the case of parallelization through clusters of computing devices, some form of synchronization is needed to ensure that the different computing devices in the cluster have the correct data, run the correct portions of the program, and share computational results at the right time, a task that is typically accomplished through implementing a specific library or protocol such as Message Passing Interface (MPI) (Dilliwar et al., 2013; Software in the Public Interest, 2023). Alternatively, in the case of specialized hardware, one would need to use a special software library, such as MPI, in the case of multicore CPUs, or even another programming language entirely, such as CUDA, in the case of certain types of GPUs (Nickolls et al., 2008).

CUDA. CUDA was designed by NVIDIA as an application programming interface (API) that would provide software developers a common platform to fully leverage parallel capabilities offered by GPUs produced by NVIDIA (NVIDIA Corporation, 2023). The CUDA API itself operates as a minimal extension of the C and C++ programming languages and allows the software developer to write a program that otherwise executes normally but is able to call parallel kernels, which can range in complexity from a few lines of a function to a full-blown program, that are then executed by the GPU instead of the CPU (Nickolls et al., 2008). This flexibility allows developers to write code that will utilize any GPU or GPGPU produced by NVIDIA, which consequently has resulted in CUDA being used by many projects that make use of parallelized code to run on many different kinds of GPUs or GPGPUs (Al Sideiri et al., 2020; Đurđević, & Tartalja, 2011; Hernandez-Lopez & Muñiz-Pérez, 2022; Temizel et al., 2011; Zhang et al., 2011).

Message Passing Interface. Message Passing Interface, commonly abbreviated as MPI, is a standardized protocol that is primarily designed for concurrent programming on computer clusters with a distributed memory architecture but is also well-suited for use on a multicore CPU with a shared memory architecture instead (Software in the Public Interest, 2023). The standardization of the MPI protocol, its specific stated design of being used for concurrent programming on a distributed memory system, and the ease with which an implementation of MPI can be set up on a cluster are all properties that make it useful for parallelization projects that are designed to run in a high performance computing cluster environment (Burstedde, 2020; Hernandez-Lopez & Muñiz-Pérez, 2022; Teunissen & Keppens, 2019).

Parallelizing Quadtrees

All of the functionality needed to use quadtrees for image compression is able to be parallelized for either CPU or GPU parallel technologies (Hernandez-Lopez & Muñiz-Pérez, 2022; Morrical & Edwards, 2017; Zhou et al., 2018). Depending on the chosen implementation and underlying technology the code is to be run on, the speedup ranges from six to thirty-seven times faster compared to a typical non-parallel implementation (Al Sideiri et al., 2020; Dilliwar et al., 2013; Hernandez-Lopez & Muñiz-Pérez, 2022; Zhang et al., 2011).

The breakdown of that large range of six to thirty-seven is as follows. Al Sideiri et al. (2020) demonstrated a fractal image compression algorithm a speedup of 1.3 with smaller images but steadily increasing to a peak speed up of 6.4 with larger images on a GeForce GT 660 M using CUDA compared to an Intel Core i5. Dilliwar et al. (2013) demonstrated a speedup of 5.5 to 6.9 for 256 x 256 test images and 6.3 to 7.5 for 512 x 512 test images using Java Parallel Processing Framework to coordinate a fractal image compression algorithm across eight computing nodes. Hernandez-Lopez and Muñiz-Pérez (2022) demonstrated a speedup of 15 on a multicore CPU and a speedup of 25 on a GPU for running a fractal image compression algorithm with quadtrees. Finally, Zhang et al. (2011) demonstrated a speedup of 37 on a GPGPU implementation compared to a single-threaded CPU implementation of a program to encode geographic information systems (GIS) data. Notably, the speedup demonstrated by Zhang et al. (2011) was only six times faster than a multicore CPU version of the same program.

Most of the research about parallelizing quadtrees for use in image compression algorithms is centered primarily around fractal image compression, a compression method that is focused on finding regions of the image to be compressed that are similar to other regions of the image. While fractal image compression is not the same as the image compression method proposed by this researcher, as the method itself is more concerned with finding self-similarity between portions of the image, rather than combining neighboring regions of the same color, the research referenced from this area is still focused on the problem of parallelizing quadtrees, and therefore remains useful for this research project (Al Sideiri et al., 2020; Dilliwar et al., 2013; Hernandez-Lopez & Muñiz-Pérez, 2022). Other research regarding parallelization of quadtrees is primarily centered on using quadtrees for object resolution and simulation of fluid dynamics or magneto-hydrodynamics (Burstedde, 2020; Morrical & Edwards, 2017; Teunissen & Keppens, 2019). One more point of research worth noting is that Zhou et al. (2018) proposed, and provided an implementation of, a general-purpose, parallelized quadtree that would be usable for the image compression method proposed by this researcher as well as many other use cases mentioned in other research.

Method

The purpose of this project was to compare the relative speedup of a parallelized quadtree image compression algorithm compared to the non-parallelized version. The code for this project went through three separate stages of attempted development before achieving the final, functional version. The final version of the code exists as multiple python scripts, some of which include the libraries MPI4Py and CuPy, both used for parallelization, among other libraries such as NumPy and Pillow, which were used for the actual image processing. The code itself was tested on an HP Pavilion gaming laptop running Windows 10 Home on an Intel Core i5 and an NVIDIA GeForce GTX 1650.

Regarding testing and the acquisition of data, the various test cases were separated out into separate python scripts that could each be run individually, though the central procedure of each script was the same. Each script was given a list of file names, those names being the names of the images to be compressed, and would iterate through the list, compressing each image and then decompressing it, timing both operations, and then writing the timing data and other pertinent information, such as the size of the compressed file, to an output CSV file. The test data itself consisted of a mix of photos taken by the researcher.

Tools Used

The core of the final version of the code is a quadtree image compression library written in Python by Inspiaaa (2023). This library provided the backbone for this project to begin compressing test images within a reasonable amount of time, taking twenty seconds or less to compress a 2400 x 3200 photograph. The existing quadtree image compression library also had four main dependencies: Pillow, a library for opening an image as an array of pixels and vice versa, NumPy, a popular library for faster array operations as well as some type information and a simple interface with Pillow, tqdm, which provided a progress bar in the command line output, and Sorted Containers, which, among other things, provided an always sorted list that is used in part of the compression routine to prioritize the addition of detail in different subregions of the image.

Additional libraries that were added in later were MPI4Py, which provided a simple wrapper for MPI functionality to be usable in a Python script, and CuPy, which provided an array library similar to NumPy but instead uses the GPU for operations. Additionally, in order to support the usage of MPI4Py and CuPy, the testing environment was also installed with Microsoft MPI 10.1.3 and CUDA Toolkit 12.3. Finally, the Python environment itself was Python 3.12.1.

Initial Design Attempts

The very first version of this project was written in C++ and started with the base of an earlier project that involved binary trees, which were easily extended to be quadtrees instead. However, that version of the project became a dead end due to the lack of success with integrating an image handling library into the project. Several separate attempts were made to install and use different image handling libraries, notably CImg, ImageMagick, and libpng, with no success. This lack of success with attempts to include any image handling library meant that no progress was made with implementing any form of image compression; therefore, there is no data for this first attempt.

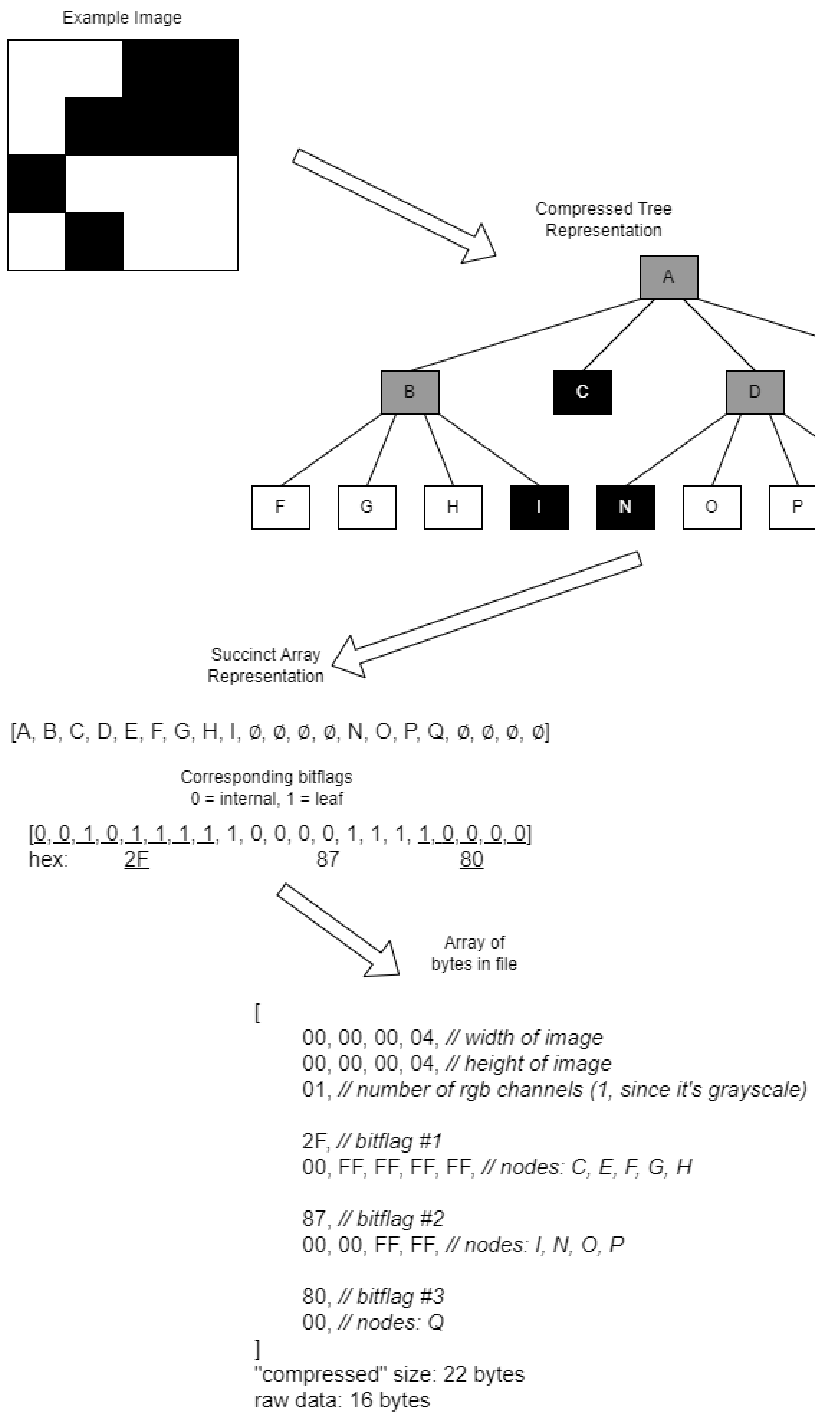
The second attempt to write a quadtree image compression library began significantly more fruitfully, given that it at least compressed some of the test images successfully. This version of the code was written in Python instead of C++ and used both Pillow, for image handling, and NumPy, for faster array operations, some type information, and its ability to interface easily with Pillow. This version of the code was a naïve implementation of a bottom-up quadtree image compression algorithm and an exact similarity heuristic. What this means is that the algorithm begins at the lowest level of the quadtree, the level at which the pixels are, and combines regions of four sibling nodes into their parent node only if all four nodes have exactly the same value, ultimately resulting in lossless compression, at least theoretically. In practice, however, this naïve implementation initially resulted in even the output of a very basic test image to be sixty percent larger than the raw data and ten times larger than the equivalent PNG.

Much of the overage compared to the raw data was due to padding that had been introduced to make the test image have square dimensions of a power of two so that the quadtree could evenly subdivide the image. This was removed by ensuring that the padding was not written to the output file, which ultimately led to some of the smaller test images resulting in an output smaller than the raw data or in some cases outperforming the PNG format. However, larger test images still did not fare well and furthermore, took over five minutes from start to finish to compress the quadtree representation and write the output file.

The output file itself consisted of the dimensions of the image and the number of channels in the image, followed by each channel of the image compressed separately. In order to store the trees in the file in a minimalistic manner, only leaf nodes were stored. As storing just the leaf nodes was not enough information to reconstruct a quadtree, especially since the compression algorithm resulted in some leaf nodes being further up in the tree, bit flags were also stored for the full tree. The bit flags were written as sets of eight bits and were interspersed among the output of the leaf nodes (see Figure 3 for a visualization). Further attempts to reduce the overhead due to the quadtree representation were unsuccessful.

Figure 3

Diagram of Example Image Being Compressed Using Original Algorithm



```
[A, B, C, D, E, F, G, H, I, ∅, ∅, ∅, ∅, N, O, P, Q, ∅, ∅, ∅, ∅]

[0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0]
hex:       2F                87               80

[
    00, 00, 00, 04, // width of image
    00, 00, 00, 04, // height of image
    01, // number of rgb channels (1, since it's grayscale)

    2F, // bitflag #1
    00, FF, FF, FF, FF, // nodes: C, E, F, G, H

    87, // bitflag #2
    00, 00, FF, FF, // nodes: I, N, O, P

    80, // bitflag #3
    00, // nodes: Q
]
"compressed" size: 22 bytes
raw data: 16 bytes
```

Note. The node values 00 and FF are black and white, respectively.

Parallelization of Existing Tool

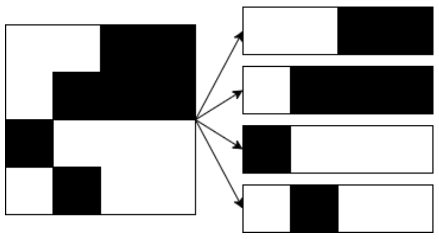
After two unsuccessful attempts to create an original program to compress images with quadtrees, the focus of this project then shifted to using an existing program, upgrading it to use MPI and CUDA, and then comparing the results. The selected program (Inspiaaa, 2023) conformed to the general specifications of this project, meaning that it compressed images using a quadtree, and, conveniently, was able to do so in a reasonable amount of time, which facilitated testing. Regarding the specifics of the program, it used a top-down compression method, as opposed to the bottom-up method used in the prior attempt and ran for a set number of iterations instead of traversing the whole tree. It prioritized which nodes it would subdivide by sorting them by level of detail, as determined by the standard deviation of each child node from the mean value. Finally, it further compressed the output with LZMA, the compression algorithm used by 7-zip. These specific differences in implementation resulted in lossy compression that was both more efficient and significantly faster than the prior attempt.

The process of parallelizing the code was rather straightforward, given that both kinds of parallelization were through existing Python libraries. The libraries themselves were not sufficient on their own, as they depended on existing implementations of MPI and CUDA to be installed on the testing environment. Open MPI had initially been the chosen implementation of MPI to be used for testing, however the original testing environment was not equipped with an NVIDIA GPU, and so the implementation of MPI was instead changed to Microsoft MPI, due to the new testing environment being Windows.

Two different strategies were used for the attempt to parallelize the program with MPI and CUDA. In the version of the code that used MPI, in the form of the MPI4Py Python library, the image data was allocated into equal-sized contiguous chunks across each subprocess and then the outputs of each subprocess were combined into one output file. The initial attempt to implement this strategy did not result in chunks that would have existed had the image been subdivided by the quadtree but rather divided the image into large bands, which resulted in unacceptable compression artifacts (see Figure 4 for visual example).

Figure 4

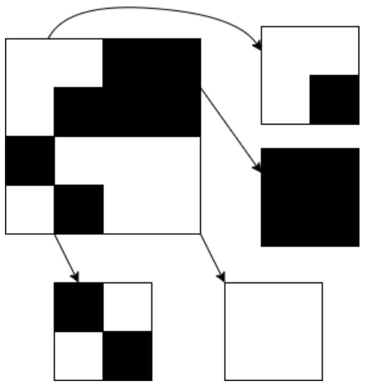
Visual Representation of Initial MPI Attempt Dividing into Four Chunks



This initial attempt also caused the program to terminate if the number of pixels in the image was not evenly divisible by the number of processes running the program. The second attempt at parallelization with MPI instead calculated the indices for an array slice based on the rank of each process relative to the total number of processes and did not result in the same compression artifacts as before (see Figure 5 for visual explanation).

Figure 5

Visual Representation of Proper Chunking with MPI



The method used to parallelize the program with CUDA, using the CuPy Python library, was much simpler and involved replacing the usage of NumPy with CuPy instead. There was also another version of the program that included both MPI4Py and CuPy, but the data from that version was not included in the results because it terminated early without outputting data or leaving an error message.

Results

The results of the MPI version of the image compression program were promising, with an average speedup of 2.44 with four subprocesses compared to the basic implementation, whereas the CUDA version of the program was not faster, but instead was 24.7x slower than the basic version. The image compression program was tested on an HP Pavilion Gaming laptop running Windows 10 Home 22H2. The laptop had an Intel Core i5-9300H CPU and an NVIDIA GeForce GTX 1650 GPU. The code was run on Python 3.12.1 and also depended on Microsoft MPI 10.1.3 and CUDA Toolkit 12.3. The test data consisted of five test images, each in three different square dimensions: 64 x 64, 256 x 256, and 1024 x 1024, for a total of fifteen test images. For each image tested, three relevant data points were collected: the time it took to compress the image, the ratio of the compressed image to the raw data, the ratio of the compressed image to the equivalent PNG, and the amount of time it took to decompress the image.

Original Attempt

The attempt to write an original algorithm for compressing images with quadtrees was initially promising, at least in regard to time. The results of the original algorithm (see Table 1) tended to be around 16% larger than the raw data and ranged from 1.5 to 2.25 times larger than the equivalent PNG. Furthermore, with larger input images, the original algorithm was much slower than even the unparallelized version of the external algorithm.

Table 1

Image Compression Data from Original Algorithm

Dimension	Compression (s)	Reconstruction (s)	Output : Raw	Output : PNG
64 x 64	0.1332	0.0672	1.1662	1.5705
256 x 256	1.7913	1.1168	1.1635	1.8028
1024 x 1024	28.403	17.754	1.1633	2.2672

Baseline Data

The baseline data, which was the base against which the later data was compared in order to calculate the speedup, was gathered from running a mostly unmodified version of the quadtree image compression library. The only modifications made to the program were adding calls to Python's time module in order to time the process and also adding a loop to process each image sequentially and then write the collected data to a CSV file. The baseline data for the four images tested can be seen in Table 2.

Table 2

Single-Threaded Image Compression Data

Dimension	Compression (s)	Reconstruction (s)	Output : Raw	Output : PNG
64 x 64	0.2982	0.0152	0.7937	1.0679
256 x 256	4.3473	0.2518	0.7016	1.0846
1024 x 1024	15.409	0.8690	0.1407	0.2722

CUDA

The version of the program that was upgraded to use CUDA was largely the same as the base version, as far as general order of execution and the steps taken for the algorithm. The only difference, aside from the results, between the CUDA version and the base version is that the

CUDA version uses the CuPy array library for GPU-accelerated Python instead of NumPy. Regarding the results, the CUDA version of the program is significantly slower than the base version, which can be seen in Table 3.

Table 3

Image Compression with GPU-Accelerated Arrays

Dimension	Compression (s)	Reconstruction (s)	Output : Raw	Output : PNG
64 x 64	5.4901	0.4330	0.7937	1.0679
256 x 256	101.49	7.0648	0.7016	1.0846
1024 x 1024	646.18	21.380	0.1407	0.2722

The CUDA version of the program demonstrated an average slowdown by a factor of 29 for compression and 21 for decompression.

MPI

The version of the image compression program that was upgraded to use MPI deviated significantly from the base version, and due to the peculiarities of how the data was combined between processes to output the compressed form produced output that was mutually incompatible with the base version and was partially larger due to extra metadata being necessary to correctly reconstruct the original image. The program was tested on four, nine, and sixteen processor cores, which resulted in the image being split into one separate chunk for each process, and the number of iterations to run the compression algorithm for was also divided by the number of processor cores. Results broken down by number of cores used by MPI can be seen in Table 4, Table 5, and Table 6.

Table 4

Image Compression with MPI on 4 Cores

Dimension	Compression (s)	Reconstruction (s)	Output : Raw	Output : PNG
64 x 64	0.1392	0.0096	0.8369	1.1263
256 x 256	1.7740	0.1052	0.7107	1.0987
1024 x 1024	5.5655	0.2982	0.1398	0.2707

Table 5

Image Compression with MPI on 9 Cores

Dimension	Compression (s)	Reconstruction (s)	Output : Raw	Output : PNG
64 x 64	0.1144	0.0066	0.7161	0.9640
256 x 256	1.1538	0.0724	0.5543	0.8572
1024 x 1024	5.0614	0.3056	0.1397	0.2707

Table 6

Image Compression with MPI on 16 Cores

Dimension	Compression (s)	Reconstruction (s)	Output : Raw	Output : PNG
64 x 64	0.2338	0.0120	0.9721	1.3085
256 x 256	1.4298	0.0966	0.7295	1.1277
1024 x 1024	4.4650	0.2824	0.1399	0.2712

The MPI version of the code exhibited an approximate speedup by a factor of 2.5 to 3.7, depending on the size of the image and number of cores (see Figure A2), although the compressed data was roughly 1% less compressed than the output of the base version of the program.

Discussion

The overall results of this project are promising, although some portions of the project appeared to be closed off to further innovation. The base version of the project resulted in compressed images with a significantly smaller data footprint than the raw data would be while also being somewhat smaller than the equivalent PNG or JPG file. The primary area with room for improvement was the execution time, which was moderately improved by a factor of 2.4 using MPI on 4 cores, although a similar attempt at improvement with CUDA resulted in a significant increase in execution time by a factor of 28. This slowdown is likely due to a combination of two different factors. Firstly, CuPy has a large overhead for creating arrays, which is likely exacerbated by the program using many small arrays instead of one large array (Castillo, 2021; CuPy, n.d.; Maehashi, 2019). Secondly, three CuPy functions, sum, mean, and std, that are used four times throughout the process are reduction operations, which are known to be much slower operations (CuPy, n.d.; Entschev, 2019).

Challenges and Limitations

Both the original algorithm and the pre-existing algorithm use the same data structure, a quadtree, to reduce redundant data within the image. However, the original implementation used a bottom-up approach, with lossless compression, whereas the borrowed implementation used a top-down approach with lossy compression instead. One other consequential difference between the two implementations is that the original implementation ran until it had checked every single node, whereas the borrowed implementation ran for a set number of iterations and prioritized nodes based on the level of detail, as defined by the deviation between the value of each child node and the value of the parent node.

The inefficiency in execution time for the original implementation likely stems from two different, though connected, sources. One significant design decision for the original implementation involved storing the tree succinctly in an array rather than as a typical collection of nodes with pointers to each other in memory. This approach was chosen in order to maximize memory efficiency during execution, but likely caused a significant slowdown due to how many memory accesses were needed for a large image. However, a significant number of memory accesses is not necessarily significant on its own. Due to the tree being stored in an array, with all of its levels stored contiguously with each other, a function was written to calculate the indices of a node's four children, which, because of how it was implemented, is likely the major source of the significant slowdown for larger images. This specific method returned a 4-tuple containing the indices. In order to properly index into the array storing the tree, the returned tuple also had to be indexed into, effectively doubling the number of memory accesses. Furthermore, this function also performed five exponentiation operations, which are somewhat expensive, not to mention that this function was also executed at least once for every single node in the tree, which, because the algorithm processed each channel separately, was three times more than the number of pixels, which was yet more inefficient compared to the borrowed implementation.

Significance

While the image compression algorithm demonstrated in this project showed moderate gains in compression ratio compared to modern image formats such as PNG or JPG, the execution time certainly could be improved. Even though the MPI version of the program demonstrated a moderate speedup, the program still took six seconds to compress a larger image, which is still orders of magnitude slower than most implementations of other image formats. It would be beneficial to further test this algorithm against other image compression algorithms on a level playing field, rather than as a pure Python script against library implementations of the other algorithms.

Similar studies in parallelizing quadtree image compression yielded an average speedup ranging from 5 to 15 with processor-based parallelization (Dilliwar et al., 2013; Hernandez-

Lopez & Muñiz-Pérez, 2022; Zhang et al., 2011) which is much higher than the average speedup of 1.5 to 3.7 in this project. The exact reason for the discrepancy is uncertain, although it is likely due to the fact that most of the code used for this project was written in Python, which is known to be slower than equivalent code in a compiled language, hence the suggestion that further research be done to test the algorithm against other image compression algorithms on a level playing field.

Conclusion

There are many image compression algorithms currently in common usage, but none of them use parallelized quadtrees as the central part of their procedure. The unmodified version of a Python script to compress images using a quadtree took approximately fifteen seconds to compress a moderately large image, but also exhibited a moderate increase in compression efficiency compared to the equivalent PNG or JPG file. Two separate attempts were made to improve the execution time of the compression algorithm. One modification attempted to leverage GPU acceleration with CUDA through using the CuPy Python library but resulted in a net slowdown by a factor of 28. Another modification attempted to use Microsoft MPI through the MPI4Py Python library and resulted in a net speedup ranging from 1.5 to 3.7. In conclusion, further work would be beneficial in exploring the efficiency of this implementation on more equal ground compared to contemporary image compression algorithms.

References

Adler, M., Boutell, T., Bowler, J., Brunschen, C., Costello, A. M., Crocker, L. D., Dilger, A., Fromme, O., Gailly, J., Herborth, C., Jakulin, A., Kettler, N., Lane, T., Lehmann, A., Lilley, C., Martindale, D., Mortenson, O., Parmenter, S., Pickens, K. S., … Wohl, J. (2023). *Portable network graphics (PNG) specification* (3rd ed.). World Wide Web Consortium. https://www.w3.org/TR/png/

Adler, M., Boutell, T., Brunschen, C., Costello, A. M., Crocker, L. D., Dilger, A., Fromme, O., Gailly, J., Herborth, C., Jakulin, A., Kettler, N., Lane, T., Lehmann, A., Lilley, C., Martindale, D., Mortenson, O., Pickens, K. S., Poole, R. P., Randers-Pehrson, G., … Wohl, J. (1996). PNG (portable network graphics) specification: Version 1.0. World Wide Web Consortium. https://www.w3.org/TR/REC-png-961001

Al Sideiri, A., Alzeidi, N., Al Hammoshi, M., Chauhan, M. S., & AlFharsi, G. (2020). CUDA implementation of fractal image compression. *Journal of Real-Time Image Processing, 17*(1), 1375-1387. https://doi.org/10.1007/s11554-019-00894-7

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A. Plishker, W. L., Shalf, J., Williams, S. W., & Yelick, K. A. (2006). The landscape of parallel computing research: A view from Berkeley. *Berkeley EECS Technical Report Series*. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

Blau, Y., & Michaeli, T. (2019). Rethinking lossy compression: The rate-distortion-perception tradeoff. *Proceedings of the 36th International Conference on Machine Learning*. https://proceedings.mlr.press/v97/blau19a/blau19a.pdf

Burstedde, C. (2020). Parallel tree algorithms for AMR and non-standard data access. *ACM Transactions on Mathematical Software, 46*(4), 1-31. https://doi.org/10.1145/3401990

Castillo, E. M. [@emcastillo]. (2021, February 28). *This is expected, array creation with CPU data is pretty heavy since you have to create the array first in* [Comment]. GitHub. https://github.com/cupy/cupy/issues/4767

CompuServe Incorporated. (1990). *Graphics interchange format(sm): Version 89a*. W3. https://www.w3.org/Graphics/GIF/spec-gif89a.txt

CuPy. (n.d.). *Performance best practices*. https://docs.cupy.dev/en/v12.3.0/user_guide/performance.html

de Bernardo, G., Gagie, T., Ladra, S., Navarro, G., & Seco, D. (2023). Faster compressed quadtrees. *Journal of Computer and System Sciences, 131*(1), 86-104. https://doi.org/10.1016/j.jcss.2022.09.001

Dilliwar, V., Sinha, G. R., & Verma, S. (2013). Efficient fractal image compression using parallel architecture. *i-Manager's Journal on Communication and Engineering Systems, 2*(3), 13-22. https://go.openathens.net/redirector/liberty.edu?url=https://www.proquest.com/scholarly-journals/efficient-fractal-image-compression-using/docview/1476284294/se-2

Đurđević, M. Đ & Tartalja, I. I. (2011). Domino tiling: A new method of real-time conforming mesh construction for rendering changeable height fields. *Journal of Computer Science and Technology, 26*, 971-987. https://doi.org/10.1007/s11390-011-1194-8

Entschev, P. A. (2019, July 23). *Single-GPU CuPy speedups*. Medium. https://medium.com/rapids-ai/single-gpu-cupy-speedups-ea99cbbb0cbb

Gao, Y., Liu, P., Wu, Y., & Jia, K. (2016). Quadtree degeneration for HEVC. *IEEE Transactions on Multimedia, 18*(12), 2321-2330. https://doi.org/10.1109/TMM.2016.2598481

Google. (2023). *Compression techniques*. https://developers.google.com/speed/webp/docs/compression

Hernandez-Lopez, F. J., & Muñiz-Pérez, O. (2022). Parallel fractal image compression using quadtree partition with task and dynamic parallelism. *Journal of Real-Time Image Processing*, *19*(1), 391-402. https://link.springer.com/article/10.1007/s11554-021-01193-w

Hudson, G., Léger, A., Niss, B., & Sebestyén, I. (2017). JPEG at 25: Still going strong, *IEEE MultiMedia*, *24*(2), 96-103. https://doi.org/10.1109/MMUL.2017.38

Hunter, G. M., & Steiglitz, K. (1979). Operations on images using quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *1*(2), 145-153. https://doi.org/10.1109/TPAMI.1979.4766900

Inspiaaa. (2023, January 9). *QuadTreeImageCompression*. GitHub. https://github.com/Inspiaaa/QuadTreeImageCompression

Klinger, A., & Dyer, C. R. (1976). Experiments on picture representation using regular decomposition. *Computer Graphics and Image Processing*, *5*(1), 68-105. https://doi.org/10.1016/S0146-664X(76)80006-8

Maehashi, K. (2019, February 18). *GPU computations are not always guaranteed to be faster than CPU. In your use-case I guess your dataset is too* [Comment]. GitHub. https://github.com/cupy/cupy/issues/2025

Morrical, N., & Edwards, J. (2017). Parallel quadtree construction on collections of objects. *Computers & Graphics*, *66*(1), 162-168. https://doi.org/10.1016/j.cag.2017.05.024

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue, 6*(2), 40-53. https://dl.acm.org/doi/10.1145/1365490.1365500

NVIDIA Corporation. (2023). *About CUDA*. NVIDIA Developer. https://developer.nvidia.com/about-cuda

Samet, H. (1984). The quadtree and related hierarchical data structures. *Computing Surveys, 16*(2), 187-260. https://doi.org/10.1145/356924.356930

Shukla, R., Dragotti, P. L., Do, M. N., & Vetterli, M. (2005). Rate-distortion optimized tree-structured compression algorithms for piecewise polynomial images. *IEEE Transactions on Image Processing, 14*(3), 343-359. https://doi.org/10.1109/TIP.2004.840710

Shusterman, E., & Feder, M. (1994). Image compression via improved quadtree decomposition algorithms. *IEEE Transactions on Image Processing, 3*(2), 207-215. https://doi.org/10.1109/83.277901

Software in the Public Interest. (2023, February 23). *Open MPI v4.1.5 documentation*. Open-MPI. https://www.open-mpi.org/doc/v4.1/

Sullivan, G. J., & Baker, R. L. (1994). Efficient quadtree coding of images and video. *IEEE Transactions on Image Processing, 3*(3), 327-331. https://doi.org/10.1109/83.287030

Temizel, A., Halici, T., Logoglu, B., Temizel, T. T., & Omruuzun, F. (2011). Chapter 34 – Experiences on image and video processing with CUDA and OpenCL. In W. W. Hwu (Eds.), *GPU computing gems: Emerald edition* (pp. 547-567). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-384988-5.00034-6

Teunissen, J., & Keppens, R. (2019). A geometric multigrid library for quadtree/octree AMR grids coupled to MPI-AMRVAC. *Computer Physics Communications, 245*. https://doi.org/10.1016/j.cpc.2019.106866

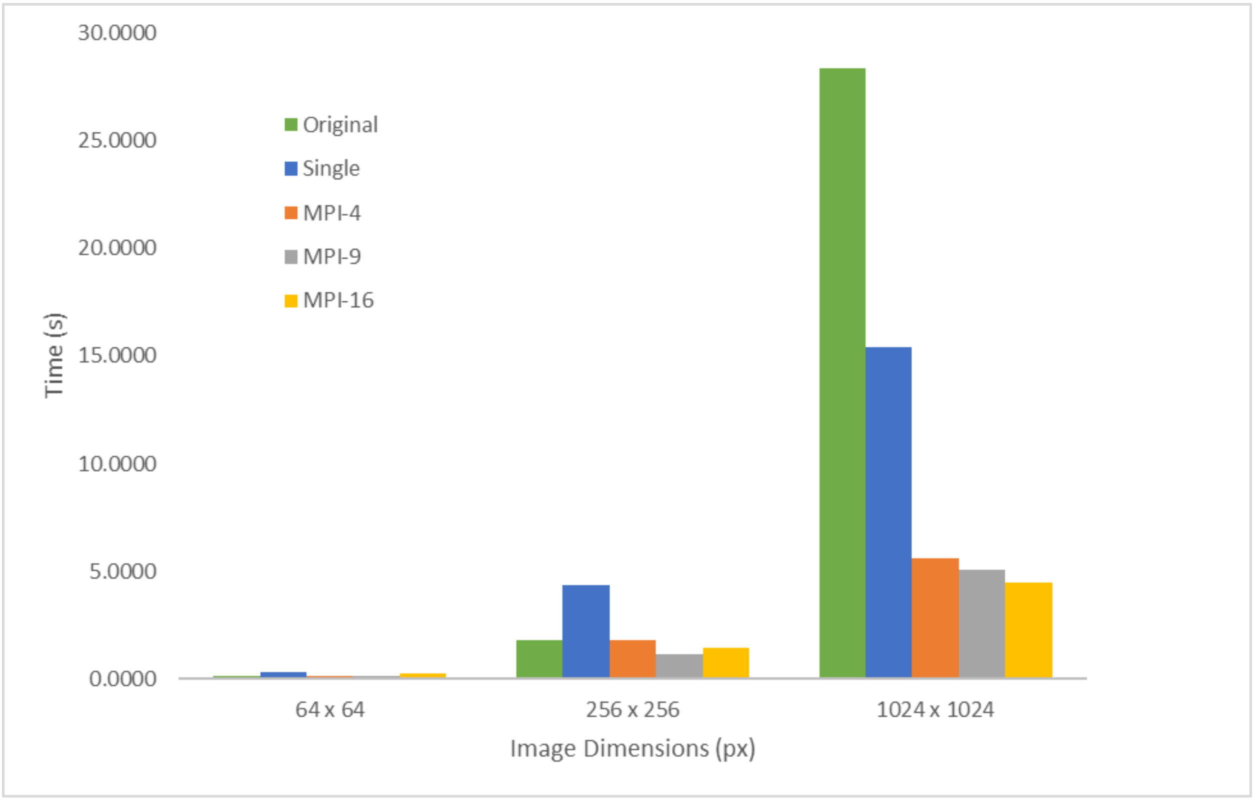
Zhang, J., You, S., & Gruenwald, L. (2011). Parallel quadtree coding of large-scale raster geospatial data on GPGPUs. *GIS ’11: Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information*, 457-460. https://doi.org/10.1145/2093973.2094047

Zhou, K., Tan, G., & Zhou, W. (2018). Quadboost: A scalable concurrent quadtree. *IEEE Transactions on Parallel and Distributed Systems, 29*(3), 673-686. https://doi.org/10.1109/TPDS.2017.2762298

Appendix A

Figure A1

Comparison of Compression Times for Original, Single-Threaded, and MPI



Note. CUDA results were excluded from the graph because they would have increased the scale such that there would be no observable difference between the other data points.

Figure A2

Relative Speedup of MPI across 4, 9, and 16 Cores Compared to Single-Threaded

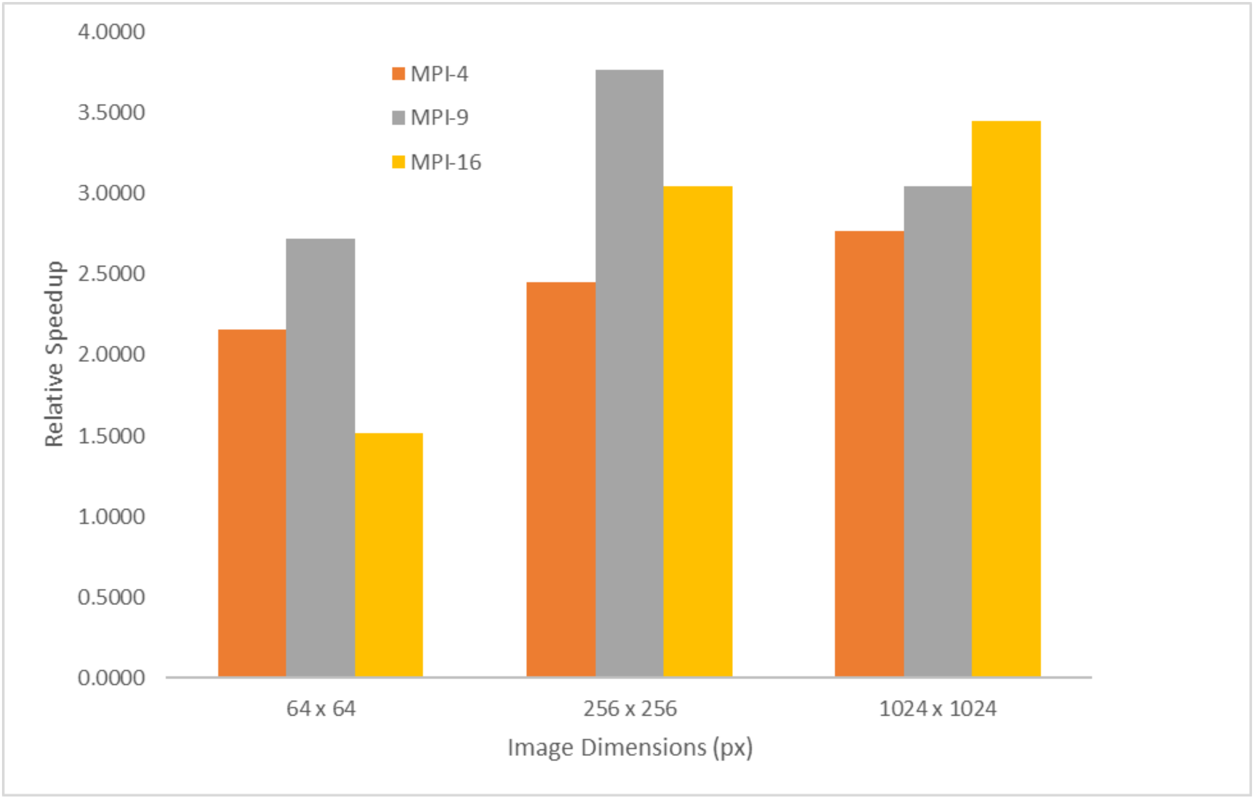
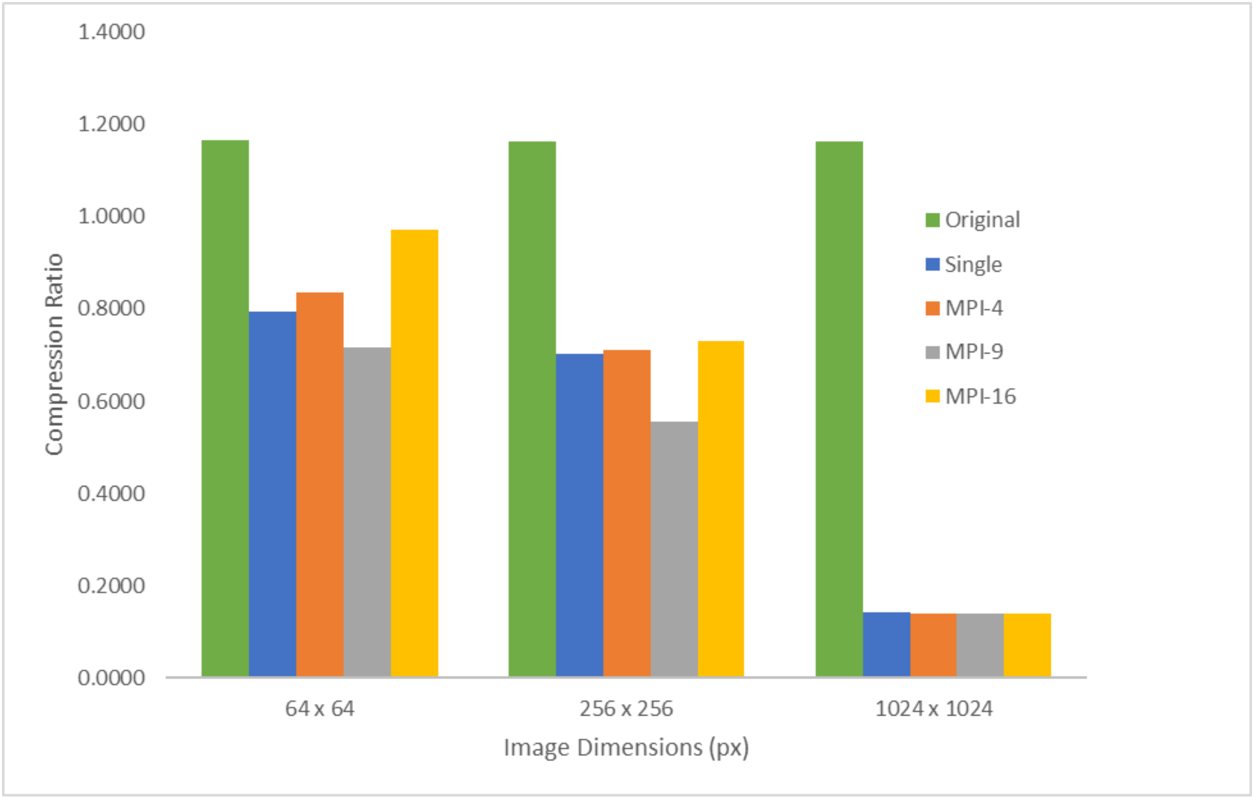


Figure A3

Comparison of Compression Ratio Compared to Raw Data



Note. CUDA is not portrayed on this graph because the compression ratio is the same as for single-threaded.

Appendix B

The code for this project can be found at https://github.com/FyreByrd/honors-qt-compression.