CUDA-ACCELERATED ODETLAP: A PARALLEL LOSSY COMPRESSION IMPLEMENTATION FOR MULTIDIMENSIONAL DATA

By

Daniel N. Benedetti

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER AND SYSTEMS ENGINEERING

Approved by the
Examining Committee:

W. Randolph Franklin
Thesis Advisor

Richard Radke, Member

Michael Wozny, Member

Peter Fox, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2014
(For Graduation August 2014)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

This research was partially supported by NSF grant IIS-1117277. Support for the Twentieth Century Reanalysis Project data set is provided by the U.S. Department of Energy, Office of Science Innovative and Novel Computational Impact on Theory and Experiment (DOE INCITE) program, and Office of Biological and Environmental Research (BER), and by the National Oceanic and Atmospheric Administration Climate Program Office.

ABSTRACT

This thesis presents an efficient ODETLAP implementation for the compression of gridded, multidimensional data. As vast quantities of data are collected for geographic information systems, compression allows for the storage and transmission of larger data sets. Techniques that utilize autocorrelation in all data dimensions allow for greater levels of compression but are more computationally intensive. ODETLAP, Overdetermined Laplacian Approximation, uses a subset of points from the original data set to accurately reconstruct the data. As it expands into higher dimensions, ODETLAP is capable of using relationships in data across multiple dimensions. Various parallelization techniques are used to improve computation time, utilizing CUDA for general purpose programming on a graphics processing unit (GPGPU). An efficient ODETLAP implementation was created directly in GPU memory, successfully avoiding the overhead associated with the transfer of data between GPU memory and main memory.

1. INTRODUCTION

Large, gridded data sets are a common occurrence in several scientific fields, including geology, oceanography, and meteorology for use in geographic information science (GIS). As vast quantities of data is collected, efficient compression becomes a concern for the purposes of both transmission and storage. Compression can be either lossy or lossless. In lossless compression, the original data values are preserved in their entirety. In lossy compression, the exact values of the original data cannot be obtained, but the goal is to achieve as near a representation of the original data as possible. The benefit of lossy compression over lossless compression is that it is often possible to achieve a much higher level of compression, resulting in a smaller compressed file size than a lossless compression scheme would be capable of creating.

The ODETLAP method can be used as a lossy compression technique for such gridded data sets. A small subset of points from the original data set is strategically selected, with both the value and location of each point being recorded. This set of point values and locations can then be efficiently compressed, the values being compressed lossily and the locations being compressed losslessly. The ODETLAP method can then be used to reconstruct the data from this compressed set of points. The ODETLAP process as a compression technique is discussed in greater detail in Section 3.

For this thesis, the goal was to use ODETLAP to create a compression technique that efficiently compressed multidimensional gridded data. Initial testing was done using a two-dimensional data set. The primary test set utilized terrain data from the Puget Sound provided by the U.S. Geological Survey and The University of Washington [16, 27]. Subsets for testing were created from the 4097×4097 elevation map data set, of sizes 512×512, 1024×1024, and 2048×2048. The original data set is shown in Figure 1.1.

Upon successful compression of two-dimensional data, implementations were created for the compression of three-dimensional and four-dimensional data. Test sets were extracted from the Twentieth Century Reanalysis data set provided by the Phys-

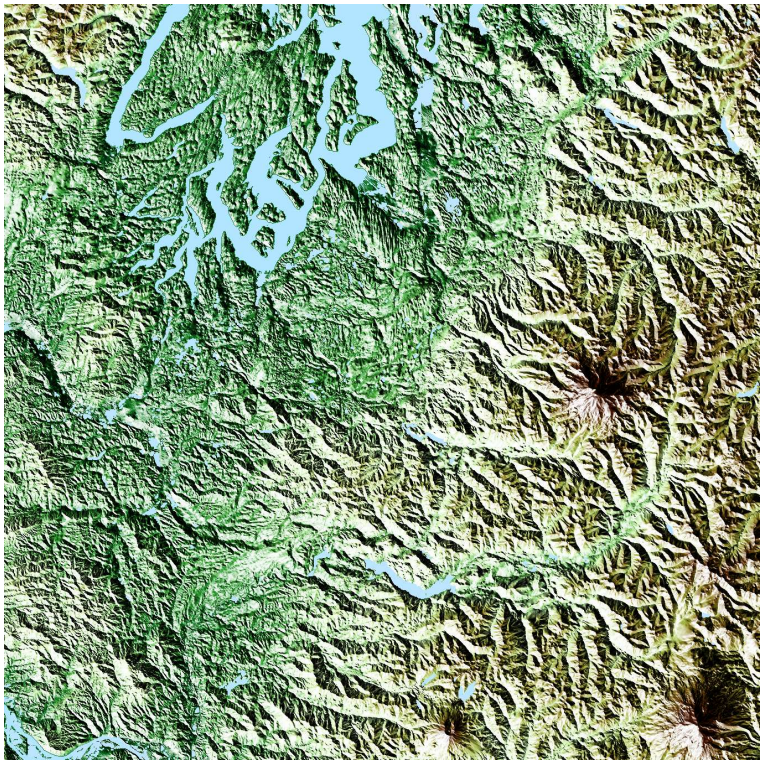


Figure 1.1: Puget Sound data set shaded relief plot.

ical Sciences Division of the Earth System Research Laboratory [7]. In particular, the pressure level monthly mean air temperature data was utilized. Tests in three dimensions viewed temperature against longitude, latitude, and pressure level for a single recorded time slice for a test set of size $144 \times 73 \times 17$. Tests in four dimensions included several time slices, with the largest test set of size $144 \times 73 \times 17 \times 20$.

This thesis will first discuss the ODETLAP method. Specific ODETLAP design decisions will be discussed, such as how points for addition to the set of known points are selected and how boundary points from the original data set are handled. Details on both a MATLAB CPU and CUDA GPU implementation will be discussed, followed by a brief discussion on the selection of a system solver. The thesis concludes with a discussion on results pertaining to computational time requirements, memory usage, and compressed memory size.

2. RELATED WORK

Most compression techniques work well for two-dimensional data, but cannot be used effectively for data in higher dimensions. That is, when applied to data in more than two dimensions, the common compression techniques only utilize relationships in two dimensions. Techniques commonly used for the compression of images, such as JPEG [28] or PNG [10], can be used for such two-dimensional compression. The simple techniques are typically only intended for the compression of two-dimen-sional data, data in higher dimensions is split into two-dimensional slices and laid out as if it was a large two-dimensional data set. These techniques have been used along with terrain simplification for improved compression results [2].

More advanced techniques, such as JPEG2000 [6] and SPIHT [24], utilize wavelets compression and other image processing techniques for improved compression capabilities. Greater levels of compression can be achieved if autocorrelation is utilized in all data dimensions. As these techniques are capable of being extended to account for more than two dimensions, they perform better than simple JPEG and PNG when compressing the same data sets [5, 17, 21].

Although originally designed for use in terrain approximation, the ODETLAP technique was found to be capable as a compression technique [8, 30]. It was then used successfully on the compression of three-dimensional data, showing that ODETLAP is capable of exploiting data dependency in all dimensions to achieve greater levels of compression [15]. Further testing was done to verify ODETLAP's ability to compress four-dimensional and five-dimensional time-varying data [13, 14].

Due to the nature of ODETLAP depending on the solution of a large, overdetermined system, it is quite computationally intensive and parallelization techniques were used to improve performance [26]. A large data set can be split into several smaller, overlapping pieces. Each piece can then be compressed using ODETLAP separately in parallel on multiple CPUs, and can be meshed together upon reconstruction of the original data set [25]. An alternate method is to utilize general-purpose computing on graphics processing units (GPGPU) for the solver, utilizing the larger

number of parallel processors to improve computation time [9, 13]. While these parallelization techniques do improve upon the results of ODETLAP on a CPU, they are limited by memory buses due to transfer times associated with moving the data between multiple CPUs or between the CPU and GPU. To improve overall performance, this thesis instead works directly on the GPU to reduce the overall execution time induced by the data transfer overhead [3].

3. ODETLAP METHOD

ODETLAP, or Overdetermined Laplacian Partial Differentiation, is a technique that is inspired by but is distinctively different than the Laplacian partial differential equation. The Laplacian partial differential equation serves as a means of smoothing a data set, but does not preserve local minima and maxima. In two dimensions, it can be represented as $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$. This relationship utilizes autocorrelation that is found in many data sets, such as terrain and other environmental data sets. Additionally, the Laplacian PDE can be easily expanded into higher dimensions. Likewise, ODETLAP is able to utilize autocorrelation for all dimensions of the data and works well for data of higher dimensions.

For a gridded data set, the ODETLAP algorithm creates an averaging equation for each data point in the data set such that its value is affected by the value of its neighboring points. Each point should have some influence over the final value of each other point in the system. For simplicity, only the values of the immediately adjacent neighbors are considered for each data point [8]. Thus, for the two-dimensional implementation of ODETLAP, averaging equations are generated for each point in the data set in the form of

$$u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4}.$$

Figure 3.1 shows an example of a general point in a 2D data set, highlighting the neighboring points that would be used as part of an averaging equation for that point. Boundary points which do not have a full set of immediately adjacent neighboring points are handled separately. A discussion on boundary points can be found in Section 3.2.

After building the equations for the Laplacian PDE, a small subset of points are selected to be the known points. Equations for the known points take the form $u_{i,j} = z_{i,j}$, where $z_{i,j}$ is the value for the data point from the original data set. Using the Laplacian averaging equations alone applies a smoothing effect to the data set. The inclusion of the known points in the system of equations helps to preserve local

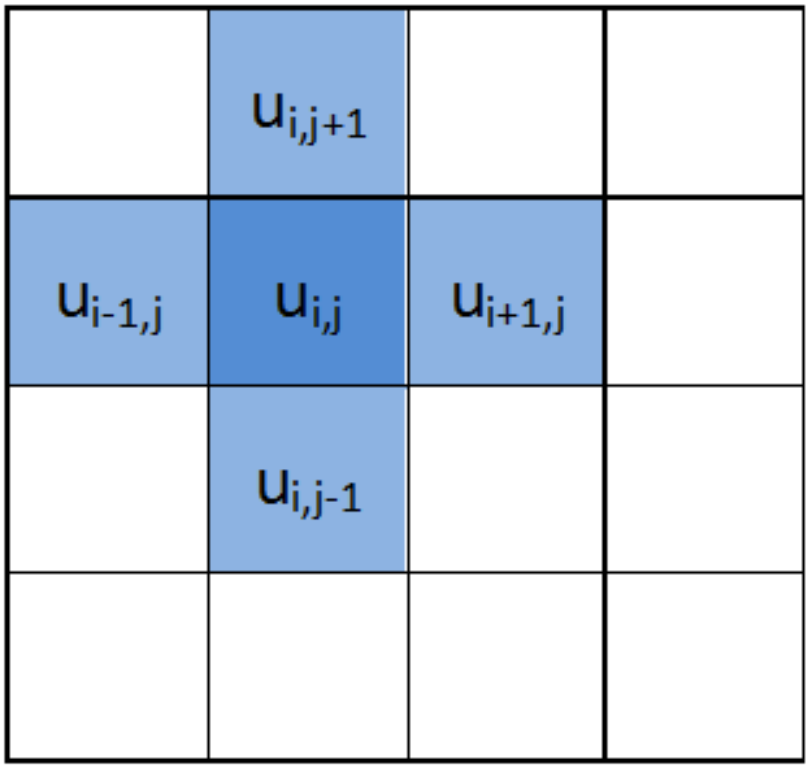


Figure 3.1: Points used for a 2D ODETLAP averaging equation

extrema that are eliminated by the system containing only the averaging equations.

The system of equations containing both the averaging equations and the known point equations can be represented in the form $Ax = b$, where x is the solution to the system and A is a large, sparse matrix. For a simple $N \times N$ data set, the A matrix will be at least $N^2 \times N^2$ in size. A direct solution for x using a matrix inverse would be inefficient and quite computationally intensive. Additionally, A is overdetermined after the addition of the known point equations, and as such, an exact solution cannot be obtained. For this reason, a Krylov-subspace iterative solver is used. Iterative solvers are discussed in greater detail in Section 4.

After the initial system solution, an average error metric for the system is calculated. For an $N \times N$ system, the average error is determined by

$$E_{avg} = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{|u_{i,j} - z_{i,j}|}{z_{max} - z_{min}},$$

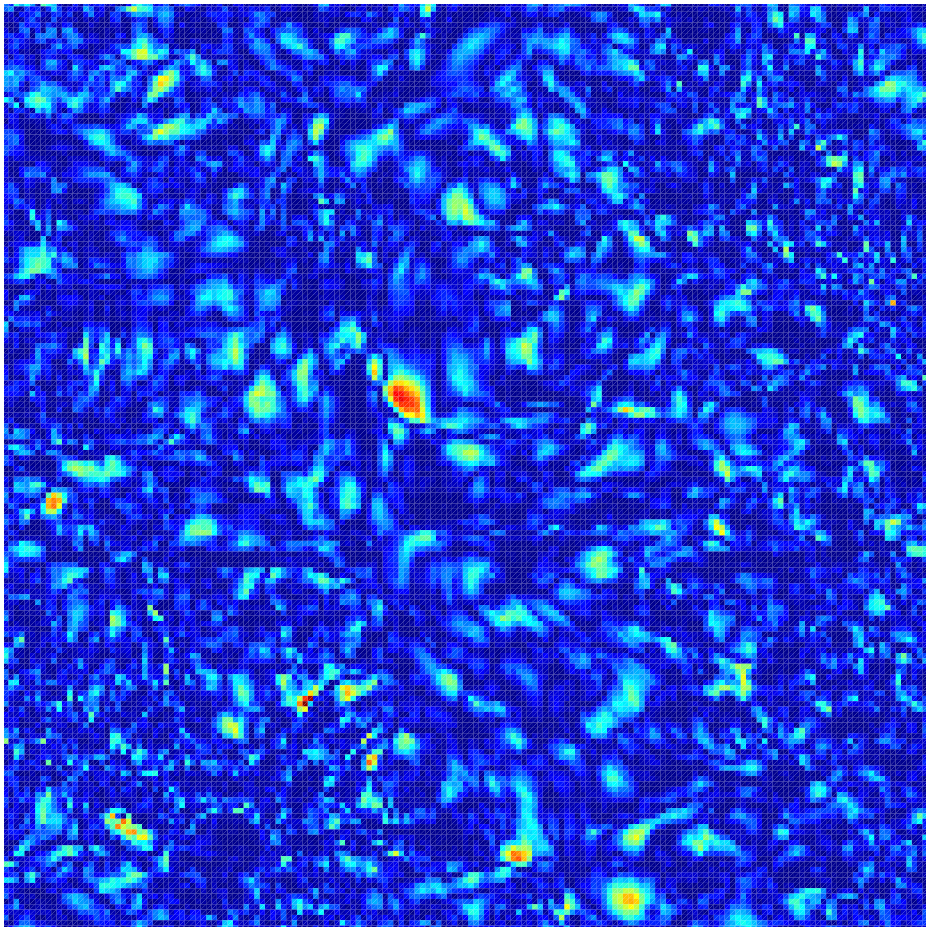


Figure 3.2: ODETLAP absolute error plot.

where z_{max} and z_{min} are the maximum and minimum values from the original data, respectively. The typical stopping condition for ODETLAP is achieved when the average error falls below a specified threshold. If the average error is above this threshold, additional known points are selected and the known point equations are added to the system. An example of an absolute error plot can be seen in Figure 3.2. The warmer colors correspond to the points of greater error, which would be the points selected for the iterative worst error selection. The solution to the updated system is then approximated, and the new average error is obtained. This iterative refinement by addition of new points repeats until the average error falls below the acceptable threshold.

There are cases in which ODETLAP will not converge to a solution that satisfies the average error threshold. In such a scenario, other stopping conditions can be uti-

Iterations	Avg. Err.
Initial	10.0%
1	2.7%
2	1.7%
3	1.5%
4	1.3%
5	1.2%
6	1.0%
7	0.9%
8	0.8%
9	0.7%
10	0.7%

Table 3.1: Average error after ODETLAP iterations.

lized. The ODETLAP iterations will stop after either a specified number of iterations or selected points. This condition exists to serve as a limit for computation time or compressed data size. Table 3.1 shows the average error after running ODETLAP for 10 iterations. The relative improvement in reduction of average error decreases with each iteration, and as such, it is not effective to run ODETLAP for a large number of iterations. This also suggests that change in error between iterations serves as a reasonable candidate for a stopping condition.

After satisfying one of the stopping conditions, the selected known points are then collected and used to compress the original data set. Whereas the original data set only stored the data values (location is inferred by order of data points), ODETLAP needs to store both data locations and values. The locations of the data points must be precise, and as such, are compressed losslessly. Due to the nature of ODETLAP relying on approximation, it is not necessary to losslessly compress the data values.

The total size depends largely on the number of bits of precision used to store the data values. In the case of 8-bit compression, there are $2^8 = 256$ different value levels. The minimum data value from the original set represents level 0 and the maximum data value from the original set represents level 255, with the remaining values linearly scaled between these two points. Each preserved data value is then assigned to the level closest to its actual value. Depending on the original data set,

this may be a sufficiently large number of different levels. To ensure the quality of the results, 16-bit compression is utilized for the results presented in this thesis.

After assigning the point values to levels, the data is then encoded as binary in two parts. The first part is a map for the point locations. For each possible point in the data set, a 1 is specified for locations in which the data value is used for the compression and a 0 is specified for locations in which the data value is not used. The second part is the value level representation for the data values. The values are listed in the order to match the order of locations in the binary map for ease of data reconstruction. After the data is encoded, it is compressed using the open, single stream 7z archive format found in the 7-Zip package [23]. A custom compression technique has been considered, but it is unlikely that such a technique would match the performance of the 7-Zip package.

A simple flowchart describing the ODETLAP process for compression can be seen in Figure 3.3. Initialization requires that the data set to be compressed is read and the compression parameters specified, such as smoothness coefficient, points selected per iteration, and stopping condition. The initial A matrix and b vectors are constructed from the averaging equations, which results in a striped square A matrix and a b vector containing all zeroes. The initial point selection adds known points equations, requiring the A matrix and b vector to be resized. With this system constructed, an initial solution x is determined and the error for that solution is calculated. If the stopping condition is not satisfied, additional points are selected based on the error and added to the system, and the A matrix and b vector are updated accordingly. The solution and errors are then recalculated for this new system. This process repeats until the stopping condition is satisfied, at which point the selected point locations and values are compressed.

3.1 Point Selection

The success of ODETLAP relies on the selection of a set of data points that accurately highlight the features in the original data set. ODETLAP is an iterative method, and as such, new points are selected on each iteration. The selection of points

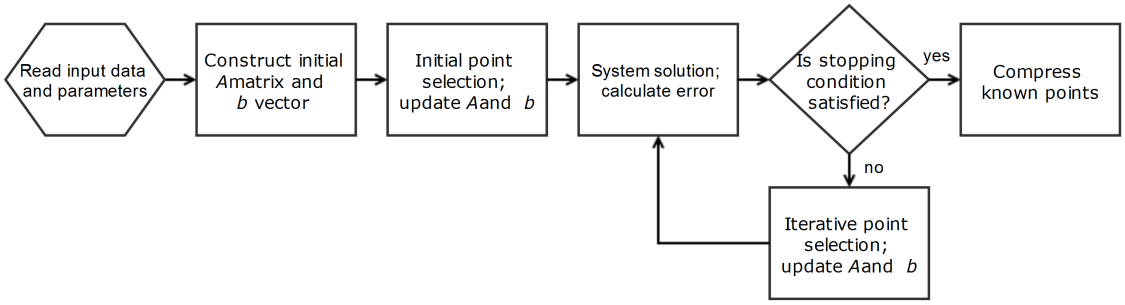


Figure 3.3: ODETLAP as a compression method.

can be separated into two distinct phases. The first phase is the initial point selection. This phase refers to the selection of points for the initial system, which is solved for the first ODETLAP iteration. The second phase is the worst error point selection. This phase refers to the selection of points based on the error for each subsequent ODETLAP iteration after the initial selection. The methods of selection are designed to both minimize the number of total points added to the system (compressed file size) and to minimize the number of ODETLAP iterations (compression time).

Various methods exist for the selection of the initial set of known points. Comparing and contrasting the effectiveness of the methods was not a focus for this thesis, although it has been studied in greater detail in previous work on ODETLAP. It was shown that the difference in technique did not have a great effect on the quality of data compression [29]. Considering the desire for a fully parallelized implementation, an initial point selection technique was chosen such that it would be capable of being implemented in parallel. For these reason, a simple gridded set of data points is selected.

Approximately 1% of points from the original data set are chosen for the initial selection. In the case of a 2-dimensional system, one point will be selected from every ten points in each dimension. This gives the desired number of points selected, as $(\frac{1}{10})^2 = \frac{1}{100} = 1\%$. The grid is evenly spaced in each dimension, and does not differ for systems with differently sized dimensions For a system of dimensions 100×200, ODETLAP will select a grid of dimensions 10×20 for its initial selection from the original data set.

The initial gridded point selection spreads known points evenly throughout the entire data set. The goal of the iterative point selection, then, is to select more points in the areas of greater interest in the data set. These areas of interest are typically locations in which the gradient vectors for each point have the largest magnitudes, and are underrepresented by the initial point selection. The iterative selection ensures that details located between grid points are not smoothed over, preserving local minima and maxima throughout the data set. For each iteration, the absolute difference between the approximated value from the previous iteration and the actual value is calculated for each data point. That is, for each point, $e_{i,j} = |u_{i,j} - z_{i,j}|$. The points with the greatest absolute difference are then selected to be used in additional known point equations.

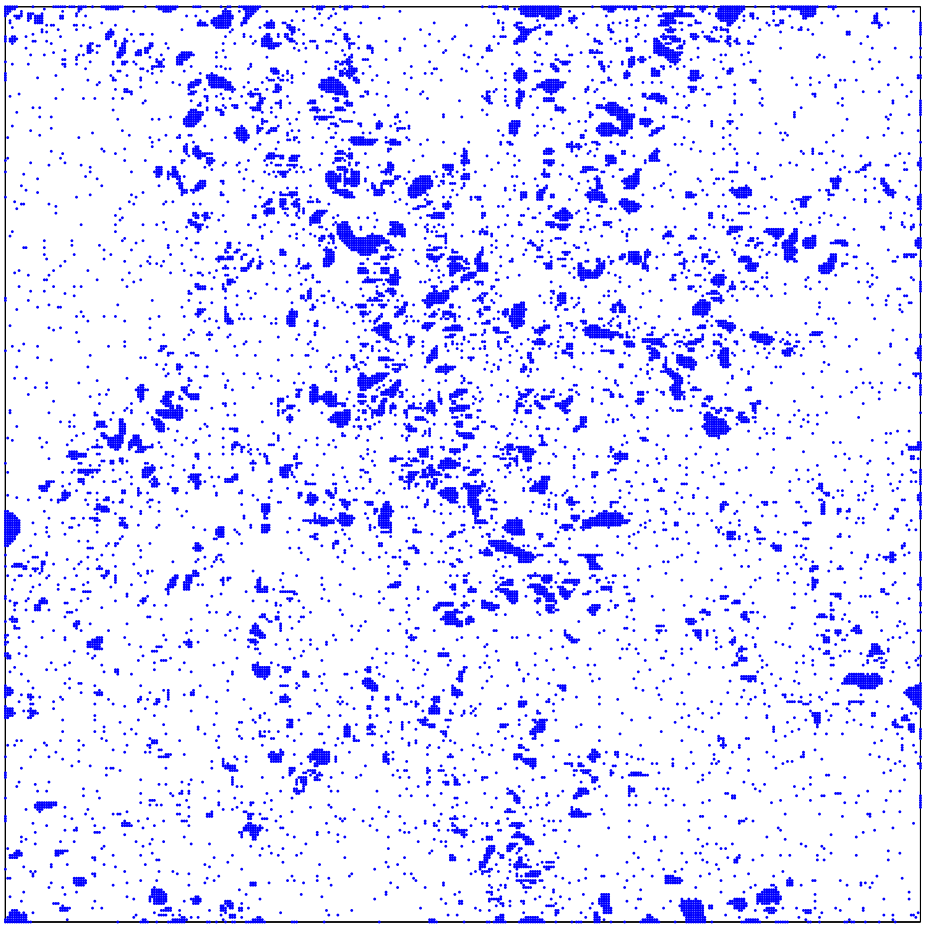


Figure 3.4: Plot of points selected as known points.

The number of points selected on each iteration should be a sufficiently small

number, typically in the range of 0.1% to 1.0% of the total number of data points. The points of greatest error tend to be clustered together, typically around edges and extrema. Selecting too many points in each iteration would therefore potentially select a large number of points in the same area. Selecting a very small number of points on each iteration would prevent such an occurrence, but would lead to a longer total compression time for ODETLAP. A technique known as "forbidden zone" has been utilized to alleviate this issue, which prevents points being selected if other points are already being added within a certain vicinity on the same selection round. This technique requires the points be selected sequentially, as each additional point to be added must first examine the locations of the previously selected points on that iteration. Such a technique is nontrivial to implement in parallel and is not included in this implementation. Figure 3.4 shows the clumping of points around areas of greatest error that forbidden zone seeks to remove. Note that random initial point selection was used in this example.

3.2 Boundary Equations

Another important aspect to the success of ODETLAP is the handling of points located on the boundaries of the data set. In the case of a two-dimensional data set with dimensions of equal size, the data can be represented as a 2-dimensional square. Geometrically, this 2-dimensional square is bounded by four 1-dimensional lines on its edges, known as 1-faces. Additionally, it has four 0-dimensional corners, known as 0-faces. The averaging equations cannot be directly applied to points located on the 1-faces and 0-faces of the data set due to the lack of a full complement of four immediate neighbors, and as such, these points must be handled separately.

The simplest approach to handling the 1-faces and 0-faces would be to exclude these points from averaging equations. This approach may be permissible for large data sets, but its effect is more noticeable on smaller data sets. Of note, it is important to ensure that removing a large number of equations does not leave the system underdetermined. A sufficiently large number of points must be selected in the initial point selection to make up for the removed boundary equations.

The approach used in the parallel ODETLAP implementation is to include a special set of averaging equations for the 1-faces. For each point along the 1-face, averaging equations are created and added to the system using only the two immediate neighbors along the edge. That is, for points on the top or bottom 1-face (constant value in the j dimension), equations are added in the form $u_{i,j} = \frac{1}{2}(u_{i+1,j} + u_{i-1,j})$. For points on the left or right 1-face (constant value in the i dimension), equations are added in the form $u_{i,j} = \frac{1}{2}(u_{i,j+1} + u_{i,j-1})$. The 0-face equations are still removed. This is permissible, however, as there are only four removed equations.

The error along edges and in corners tends to be greater than the error in the central parts of the data set. As such, the worst error selection phases will add a larger number of points to the edges. This will make up for the loss of order of accuracy due to the drop in dimension on the special-case boundary equations. In particular, the four 0-face corner points are almost always included by the iterative point selection, unless the area surrounding the corner is particularly flat.

3.3 Smoothness

A factor to consider when building the system of equations is the relative weight between the two types of equations. There are a substantially larger number of averaging equations than known point equations in the overdetermined system. If the equations were all weighted evenly, the averaging equations tend to smooth the entire system. While this may be desirable in some scenarios, a more accurate reconstruction of the data set is necessary in the case of data compression. A weighting factor, known as the smoothness coefficient or 'R', is introduced to address this need.

The smoothness coefficient is applied as a multiplier to the values in the rows in the A matrix corresponding to the averaging equations. In the case of the aforementioned 2-dimensional data set with a smoothness coefficient $R = 1$, the row elements in A will be set as -4.0 and 1.0 for the point location and the neighbor locations, respectively. In the case of $R = 0.5$, the row elements in A will then be set as -2.0 and 0.5 for the point location and neighbor locations, respectively. A relatively small value for R must be used for an accurate data reconstruction. Typically, a value of $R = 0.1$

seems to produce the best results. Using even smaller values for R results in averaging equations having negligible impact on the final results.

3.4 Higher Dimensions

ODETLAP is easily expandable into higher dimensions, with minimal change being required to the implementation. The overall flow remains unchanged, as the process of averaging equations, initial point selection, and iterative solution with iterative point selection remains the same. There are, however, some important changes that occur with regard to the averaging equations, the boundary equations, and the initial point selection.

A key reason for utilizing the Laplacian ODE is its ability to expand to higher dimensions. In the case of a 3-dimensional data set, the ODE is represented by $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$. For the discrete form of this, it can be said that each point will have six immediately adjacent neighbors, with two on each dimension. This lends to the construction of averaging equations in the form $u_{i,j,k} = \frac{1}{6}(u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1})$. This pattern continues into any number of higher dimensions. The highest dimension in this ODETLAP implementation is 4-dimensions, which utilizes averaging equations in the form $u_{h,i,j,k} = \frac{1}{8}(u_{h+1,i,j,k} + u_{h-1,i,j,k} + u_{h,i+1,j,k} + u_{h,i-1,j,k} + u_{h,i,j+1,k} + u_{h,i,j-1,k} + u_{h,i,j,k+1} + u_{h,i,j,k-1})$.

The special-case boundary equations become more complicated in higher dimensions. For a 3-dimensional data set with evenly sized dimensions, which can be represented as a cube, the boundaries consist of six 2-faces (squares), twelve 1-faces (lines), and eight 0-faces (vertices). For the 2-dimensional data set, only two sets of special equations were required (for constant i and constant j 1-faces). For the 3-dimensional data set, three sets of special equations will be required for the 2-faces (constant i, constant j, and constant k). Additional special equations are required for the 1-faces. The complexity continues to increase moving into higher dimensions, as shown in Table 3.2. It would therefore be unwieldy to directly specify special equations for each possible boundary location. To account for higher dimensions, then, the implementation must use an approach that dynamically constructs the equations

	1-faces	2-faces	3-faces	4-faces
2D Data	4			
3D Data	12	6		
4D Data	32	24	8	
5D Data	80	80	40	10

Table 3.2: Number of each type of n-face for data sets of multiple dimensions.

for each point based on its position within the data set.

Lastly, initial point selection must be expanded to select points in each dimension. Due to the uniform nature of the selection, the grid must simply be expanded into the additional dimensions. The target of approximately 1% of points being selected still holds. In the case of a 3-dimensional data set, one out of every five points in each dimension may be selected, giving $(\frac{1}{5})^3 = \frac{1}{128} = 0.8\%$. In the case of a 4-dimensional data set, one out of every three points in each dimension may be selected, giving $(\frac{1}{3})^4 = \frac{1}{81} \approx 1.23\%$. Moving into even higher dimensions may require a more sophisticated gridding technique, but this method is sufficient for the purposes of 2-, 3-, and 4-dimensional data sets.

4. IMPLEMENTATION

The ODETLAP algorithm is quite computationally intensive. In order to improve the usefulness of ODETLAP for use as a method of data compression, the implementation must be efficient. This section discusses two different implementations, one using MATLAB [18] and one using the CUDA [20] library for C++. A two-dimensional MATLAB implementation was created first to serve primarily as a baseline to which the two-dimensional CUDA implemention could be compared. Afterwards, ODETLAP implementations were created for three-dimensional and four-dimensional data sets using CUDA only.

The ODETLAP algorithm can be broken into several distinct components, as previously shown in Figure 3.3. The most computationally intensive section is the solution of the overdetermined system. The computation time can be greatly improved through the use of parallelization. The initial matrix construction, initial point selection, and iterative point selection portions also benefit from a parallel implementation. MATLAB operates in parallel on CPUs for certain functions, while CUDA relies on Nvidia-based GPUs or accelerators for parallelization. GPUs have a significantly larger number of processing cores and threads than CPUs, and as such, the CUDA implementation is able to achieve a greater level of parallelization.

Due to the parallel nature of the implementation, traditional loops are not appropriate to use. Loop-based code iterates over an array or matrix one element at a time, utilizing only a single thread for the computation. The solution to this problem is the use of vectorized code. Each element of a vector, if it is independent of other elements in the vector, can be operated on simultaneously. ODETLAP relies primarily on common vector operations, such as sequencing, sorting, and reductions, and matrix operations including transposition and multiplication. MATLAB and CUDA both support these vector and matrix operations, and operate most effectively on vectorized code.

In addition to these common operations, ODETLAP depends on the construction of a large, sparse matrix and the solution of an overdetermined system. MATLAB's

core package provides the capabilities for processing sparse matrices and a solver capable of approximating a solution to such an overdetermined system. The base library for CUDA does not support sparse matrices, but a package cuSPARSE is available. For this implementation, however, the Thrust template library is utilized. With that, the CUSP sparse template library is available and provides several sparse matrix formats and system solvers.

4.1 MATLAB

The first ODETLAP implementation created for this thesis uses MATLAB to compress a two-dimensional data set. The focus of the initial implementation was functionality, with a goal of achieving basic compression with minimal code complexity. Certain aspects of this implementation were therefore selected for their simplicity instead of their performance. Optimization was then performed in several stages upon completion of the initial implementation. The following section will discuss each portion of the ODETLAP method as it was implemented in MATLAB.

The first stage for ODETLAP is the initialization. Parameters for input data file name, smoothness coefficient R, and number of iterations must be assigned values. ODETLAP parameters can be specified in two different ways, written directly as constants within the MATLAB code or assigned as command line arguments. To ease initialization time, the data sets for use with MATLAB were converted to MAT-File format [19]. This allowed the matrix containing the original data set to be saved and quickly loaded in the format required for use by ODETLAP without further processing. For simplicity, the MATLAB implementation requires a matrix with even dimensions.

After the input file is read, information about the data set is extracted. The total number of points read is specified as n. The initial A matrix A matrix will be of size $n \times n$, and the number of nonzero matrix values is $5n^2$. Additionally, the number of points selected on each iteration is specified as 2% of n^2. The final information obtained during initialization are the max and min values of the data set, which are used as part of the error calculation.

The initial A matrix and b vector are constructed for the averaging equations. The b vector is a zero vector, and is trivial to construct. The A matrix is sparse, and the row index, column index, and value must be specified for each nonzero element. For each element in the k-dimensional original data set, there are $2k + 1$ nonzero values. That is, for a 2-dimensional matrix, each element has five nonzero values; for a 3-dimensional matrix, each element has seven nonzero values; etc. Iterating through a k level loop to populate these nonzero values is simple, and was done for the initial MATLAB implementation. This, however, is extremely inefficient as it is only able to utilize a single thread.

As each nonzero row index, column index, and value is independent, vectorization techniques can be utilized to construct the A matrix. MATLAB does not require sparse matrices to be sorted by either row index or column index, which allows for the use of vector sequencing and concatenation for construction. In the 2-dimensional scenario where each equation contains five nonzero values, one can consider the positions of these values in the original data set as center ($u_{i,j}$), left ($u_{i-1,j}$), right ($u_{i+1,j}$), bottom ($u_{i,j-1}$), and top ($u_{i,j+1}$). If each nonzero element is said to be in these positions, it is simpler to construct all elements in that matching position together as a group before moving onto the next position than it is to construct each equation as a row in the matrix. These positions relative to the center value expand into higher dimensions, but become more difficult to specify by name.

The row indices are simple to specify through the use of sequences. For the k-dimensional original data set, there should be $2k + 1$ nonzero values per row for the general case, with the exception being the boundary points. Each row corresponds to one of the averaging equations, which means each of the nonzero values per row will correspond to one of the aforementioned positions. For each position, we then need one row index for each row. Therefore, sequences from 1 to n^2 are created to serve as the row indices for each position.

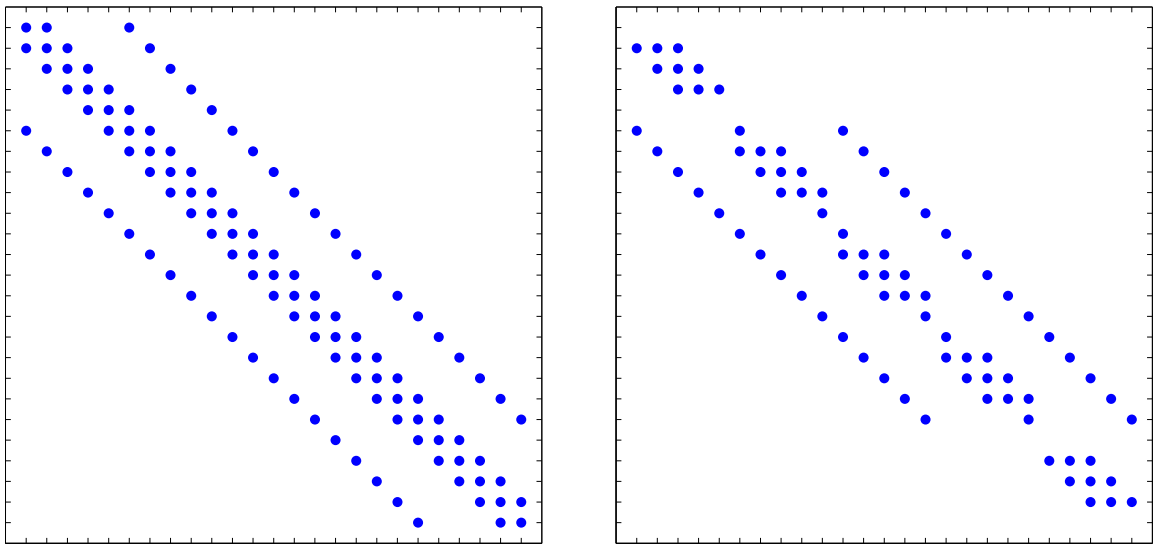
The column indices can also be specified by utilizing sequences, but they are not as easily constructed. The center position can be specified as a simple sequence from 1 to n^2. All other positions are offset according to their difference from the

center using one-dimensional indexing, with the unused offset indices being padded to preserve the expected number of nonzero column indices. In the case of the left and right positions, one dimensional indexing of the matrix places these values with offsets -1 and $+1$ from the center position, giving a sequence of 1 to $n^2 - 1$ and 2 to n^2, respectively. In the case of the bottom and top positions, one dimensional indexing of the matrix places these values with offsets $-n$ and $+n$ from the center position, giving a sequence of 1 to $n^2 - n$ and $n + 1$ to n^2, respectively. This pattern expands into higher dimensions. For a 3-dimensional data set, the offset for the third dimension would be $-n^2$ and $+n^2$, respectively. If the dimensions are not uniform in size, as will be discussed in the ODETLAP implementations, the sizes of the lower order dimensions are used to replace the general n in these examples, as per the position based on one dimensional indexing of the original input data.

The values are simple to specify through the use of a constant vector. The center position nonzero values for k-dimensional input data are set to $2kR$ and the values for all other positions are set to $-R$, where R is the specified smoothness coefficient. In MATLAB, this can be done by simply using the `ones()` function to create a constant vector with each value set to 1, and performing an elementwise multiplication with $2kR$ or $-R$. The elements with padded zero column indices have their value also set to zero.

Through matrix concatenation, the matrices for each position can be combined. With the row indices, column indices, and values specified for each nonzero element, then, the initial A matrix can be constructed. At this stage in the construction, the nature of the offsets and sequences result in a striped matrix. An example of a 5×5 data set's system matrix can be seen in Figure 4.1a. The padded column index offset creates rows with fewer than the maximum number of elements, all of which would result in invalid equations. These rows, among others, correspond to boundary points for which special equations must be adapted. As such, these rows must be modified.

While it is possible to locate and adjust the boundary equations in parallel (as is done in the CUDA implementation), it is quite complex. For the purpose of the MATLAB implementation, it was deemed a low priority. In the sequential implementation,



(a) Initial construction

(b) Corrected for boundaries

Figure 4.1: Initial A matrix construction plot of nonzero values.

each point in the original data set is iterated over. If the point falls along one of the boundaries, the values for the appropriate non-center positions are set to 0, and the center position's value is reduced by $2R$. If the point falls along multiple boundaries, then multiple pairs of values can be removed and the center value can be reduced to zero. Figure 4.1b shows the resulting A matrix after updating the equations to account for the boundaries. For this two-dimensional example, the rows corresponding to point locations along the edges only have three nonzero values and the rows corresponding to points located in the corners have no nonzero values. This reflects the intention for the boundary equations to use only one dimension for their averaging equations and the corner points to have no averaging equations. Sample code for the initial construction of the A matrix and b vector can be found in Appendix A.1.

With the initial A matrix and b vector constructed, it is now necessary to add known point equations to the system before an initial solve attempt. The current system is underdetermined, as a result of the removed equations for all corner points. Additionally, the b vector contains only 0 values and an approximated solution would likely return 0 values for all points. To address these issues, an initial set of known points are added. For the MATLAB implementation, a pseudo-random set of points

are selected. Rows are added to A for each added equation, with the only nonzero being a 1 in the column corresponding to the points one-dimensional index. The value for the point from the original data set is then used in the equivalent row in the b vector. The resulting A matrix is no longer square as the system is overdetermined.

The initial system is now properly expressed for both A and b, and the next step is to find an initial solution. As the system is overdetermined, the solution may not be exact and instead must be approximated. The proper solution to $Ax = b$ would require a matrix inverse, such that $x = A^{-1}b$. As this is extremely inefficient, a numerical solution is used. MATLAB provides a default system solver, called as $x = A \setminus b$. The default solver will then choose the best numerical system solver based on the provided system. In the case of the rectangular A matrix used by ODETLAP, the only solver available in MATLAB is the LSQR method [22].

While the LSQR method provides a valid solution to the overdetermined system, the method is quite slow compared to several other numerical solvers. As all other sparse numerical system solvers provided by MATLAB (all all solvers provided by CUSP) require a square A matrix, the system must be first modified. The equation $A^T Ax = A^T b$ will provide an equivalent normalized system $A' = xb'$, where $A' = A^T A$ and $b' = A^T b$. As A' is a square matrix, this modified system can be used with all solvers provided by both MATLAB and CUSP. Due to its simplicity and efficiency, the Conjugate Gradient (CG) solver is used. System solvers are discussed in greater detail in Section 4.3.

After the initial system solution, ODETLAP now enters into its iterative phase. The initial solution, x, is compared to the original input data values, and the error can now be calculated for each point. While the stopping conditions are not yet satisfied, the points with the greatest error magnitudes are added to the A matrix and b vector as additional known point equations. The system is again normalized and a new solution is found. After one of the stopping conditions is satisfied, the ODETLAP process is complete. The full list of known point locations and values, from both the initial and iterative point selection phases, are collected. Lossy value levels are defined based on the minimum and maximum of the original data, and

assigned to each point. The locations for all selected points are preserved losslessly. This final collection of point locations and values are then compressed losslessly using 7zip.

The MATLAB functions are typically optimized to work on vectorized code, but do not operate in parallel. There are several functions that do however utilize parallelism available in MATLAB. Of the functions used by ODETLAP, `mldivide` (the system solver function), `multiply`, and `transpose` all utilize multiple CPU threads. Most significant of these is the system solver, as it easily requires the most computation time of any component of the ODETLAP algorithm. MATLAB is unfortunately restrictive on the maximum number of workers through licensing. As such, all tests using MATLAB ran using only 16 CPU worker threads.

4.2 CUDA

After completion of the MATLAB implementation for two dimensions, the goal was to improve upon the computation time through the use of parallelization on the GPU. This section will detail the use of the CUDA library in C++ for parallelization on Nvidia GPUs for an efficient ODETLAP implementation. The template libraries Thrust and CUSP and their constructs will be discussed. Implementation details will be given for ODETLAP in two-dimensions, three-dimensions, and four-dimensions.

Prior work has been done to try to improve the computation time of ODETLAP through the use of GPU parallelism. The key distinction in this work is that ODETLAP is implemented directly on the GPU. While GPU acceleration is useful for many large, computational intensive applications, the cost of data transfer between CPU and GPU memory is quite expensive. If an input data set is not sufficiently large, the overhead associated with data transfer to and from the GPU will outweigh the potential computational time improvement associated with utilizing the GPU.

To avoid data transfer, the data can be constructed directly on the GPU. Two libraries provided by CUDA are used for this purpose, Thrust [11] and CUSP [1]. Thrust is a template library that provides many functions for vector manipulation in CUDA. It additionally allows for construction of dense vectors directly on the GPU.

Thrust contains all of the functionality needed for the initial matrix construction, initial point selection, and iterative point selection portions of ODETLAP. The CUSP library is a sparse extension to the Thrust library. It contains various formats for representing sparse matrices. Most importantly, CUSP contains an assortment of sparse system solvers, needed for the solve portion of ODETLAP. Specific functionality for both Thrust and CUSP will be discussed as the ODETLAP process in CUDA and C++ is described.

The general process for ODETLAP is unchanged from the MATLAB implementation discussed in Section 4.1. There are, however, several important changes that result from the limited nature of functions available when working on the GPU. Thrust requires the use of a functional programming methodology. Starting from either a vector of constant values or a sequence of values, the vector undergoes a series of transformations until it is converted to the desired vector. This general process occurs throughout the ODETLAP CUDA implementation. While these operations are all quite efficient, the limited number of functions available forces several stages of transformation in order to obtain the desired vector. This leads to code that is quite unintuitive, and considerably more complicated than the comparable code from the MATLAB implementation.

The Matrix Market file format is used by CUSP for reading sparse and dense matrices from files. This format supports dense matrices in one or two dimensions. As the CUDA ODETLAP implementation is intended to use inputs in higher dimensions, the input data file is converted to a one dimensional vector before being saved in the Matrix Market format [4]. Unlike the MATLAB implementation, the CUDA implementation allows for input data with varying data dimensions. As the input data is saved as a one dimensional vector, the size of the data dimensions must now be specified as additional parameters.

All other initialization steps are unchanged from the MATLAB implementation. Finding the minimum and maximum of the set is done using the Thrust `min_element` and `max_element` functions, respectively. Like most Thrust functions, the parameters are iterators for the start and end of the vector being operated on. In this case, the

beginning and end iterators are pointers to the first and last elements of the input data vector.

Memory is allocated for the A matrix using the Coordinate (COO) matrix format. A COO matrix operates in an equivalent manner to the sparse matrices of MATLAB. That is, row indices, column indices, and values are specified for each nonzero value. CUSP provides several other formats that are more efficient for different types of matrices. For example, the Diagonal (DIA) format is most optimal for diagonal matrices, which are a common occurrence. Most of these formats, however, are not intended for use in dynamically constructed matrices. Despite this, the A matrix could conceivably be constructed in any of these other formats. The COO matrix was chosen primarily for its similarity to MATLAB.

The initial matrix construction for the averaging equations using Thrust and CUSP is considerably more verbose than the construction in MATLAB. This is due primarily to the lack of vector concatenation found in MATLAB. Despite this, the same techniques are used. The Thrust `sequence` function is used extensively to populate the row indices and column indices, with the start value for the sequence being specified as an argument. A step value can optionally be specified, but for the purposes of ODETLAP, the default step size of 1 is always used. The Thrust `fill` function is used to populate the values and column index padding, with the value to pad with being specified as an argument. In place of vector concatenation, the CUDA implementation instead starts with one large vector and operates only on pieces. Both the `sequence` and `fill` functions take the start and end iterators as arguments for the desired vector to operate on, allowing for each position to be operated on independently.

The boundary equations are also accounted for in parallel for the CUDA implementation of ODETLAP. The general Thrust `transform` function is utilized here, and heavily throughout the CUDA implementation. For each point or pair of points in a vector, a specified unary or binary operation is applied. Several predefined operations are provided, such as the unary operation `negate` or the binary operation `add`. Alternatively, custom operations can be defined, as is the case for the boundary

point adjustments.

For each dimension, a separate transformation is created. These transformations, applied to each index of the original data set using one-dimensional indexing, test if a point is located along one of the boundaries for that dimension. If the point is located on a boundary, the transformation correctly modifies the corresponding values for the previously specified nonzero values in the A matrix. That is, the values for the appropriate non-center positions are set to 0, and the center position's value is reduced by $2 * R$. As the operations are applied independently to each point, all transformation operations utilize the parallel capabilities of the GPU.

After applying all transformations for both the general purpose averaging equations and boundary equations,the row indices, column indices, and value vectors should be equivalent to their MATLAB counterparts. Unlike MATLAB, however, CUSP requires that the sparse matrix nonzero values be sorted by row. As such, it provides a `sort_by_row` function which is called upon completion of the initial matrix construction. Sample code showing initial point selection for CUDA in 2D can be found in Appendix A.2.

The initial point selection is also performed fully in parallel for the CUDA implementation. The original intention was to use a random shuffle to select points randomly in the same manner as MATLAB, a simple grid was chosen instead for ease of parallelization. The initial point selection phase requires several operation in order to add the equations for known points to the A matrix and b vector. Starting with a sequence from 1 to the total number of points, with each value corresponding to some location in the original set using one-dimensional indexing, a `transform` operation is applied to each point. For the points that will be selected based on the grid size of the initial selection, the values of those points are unchanged. All other points in the sequence vector have their values changed to 0. A Thrust `sort` operation is used to bring the nonzero indices to the front of the vector. The indices corresponding to the known point equations are now located at the front of the initial point selection vector.

Before the new equations can be added, the A matrix and b vector must be resized.

A Thrust `count_if` operation is used to count the nonzero values to determine the precise number of new equations that will be added to the system. The A matrix must be resized both in number of rows and in number of nonzero values. A sequence starting from the current highest row number in A is used to specify the row indices for the new equations. The indices fetched via transformation and sorting previously correspond to the points to add and are used as the column indices. The values for each of these new nonzero values in A are filled as 1. To assign values to the b vector, a Thrust `gather` operation is performed on the point values from the original data set using the indices of the newly added known point equations as keys. Finally, the values brought to the front of the vector after the `gather` are assigned as the values in the b vector.

The iterative point selection phases are quite comparable in their construction to the initial point selection. For the iterative selection, only the Thrust operations to select points to add differ. That is, the points selected come from sorting the error for each point and selecting the indices corresponding to the points of highest error. For each case, several Thrust operations are required to simply select a set of known points and add them to the A and b vector. As each of these functions are optimized for parallel computation on the GPU, the overall time requirement for these point selection sections is still quite low. It is the reliance of these parallel vector operations that forces the use of a functional programming style in place of loop-based code.

With the parallel efficiency of the stages used for matrix construction and point selection, a large percentage of computation time goes towards the solve section. All solvers in CUSP require a square A matrix, so the normalized system $A^TAx = A^Tb$ is solved. The major distinction here is that the transpose and multiplication operations cannot be performed in a single step. Instead, the CUSP `transpose` and `multiply` functions are used, respectively. Like the MATLAB implementation, the Conjugate Gradient (CG) method is again utilized. Discussion and performance comparison of the CG and other system solvers can be found in Section 4.3.

The primary intention of this implementation is to avoid the large time penalties that result from data transfer between main memory and GPU memory. That is, the

A matrix and b vector are constructed directly in GPU memory, and all computation is performed on the GPU. During the solving stage, intermediate matrices and vectors used in the normalized system solution must be stored. So in addition to A and b, memory must be allocated for A^T, A^TA, and A^Tb. As memory on the GPU is quite limited compared to the memory available to a CPU, this code section serves as a memory limit to restrict the maximum size of the input data set.

The error computation depends on Thrust transformations and a reduction. The first transformation applied utilizes the Thrust built-in binary operator for subtraction, `minus`, using the original data set and the solution to the system x as parameters. A custom transformation is then applied to the result of this to find the absolute value for the difference between these two vectors. This absolute difference vector is stored and used for the iterative point selection phases. Lastly, the average error is obtained by a reduction, which finds the sum of all of the individual absolute errors. The sum is then divided by the number of points and the data range to find the average percentage of error for each point in the data set.

The final step is compression of the data. The data on the GPU must be copied to the host before a file can be written. Additionally, file input and output cannot be done in parallel. The writing and compression of the output data are not included as part of the timing calculations.

Different implementations were created for each size of input data set. Moving into higher dimensions, the initial construction changes minimally. The number of nonzero values grows with increasing sizes of k for a k-dimensional data set. Let n_x, n_y, n_z, and n_t represent the size of each of the up to four dimensions x, y, z, and t, respectively. The initial A matrix for a two-dimensional data set is $n_2 \times n_2$, where $n_2 = n_x n_y$, and the number of nonzero values before adjustment is $5n_2$. The initial A matrix for a three-dimensional data set is $n_3 \times n_3$, where $n_3 = n_x n_y n_z$ and the number of nonzero values before adjustment is $7n_3$. The initial A matrix for a four-dimensional data set is $n_4 \times n_4$, where $n_4 = n_x n_y n_z n_t$ and the number of nonzero values before adjustment is $9n_4$. This trend can continue into higher dimensions.

The averaging equations are also updated to reflect the number of data dimensions.

The number of nonzero values in each row for the k-dimensional data input is $2k+1$, and the value assigned to the nonzero corresponding to the center value is $2kR$, where R is the smoothness coefficient. The only other change for higher dimensional data inputs is the initial point selection. As the number of dimensions increases, the distance between grid points in each dimension will have to decrease in order to achieve a comparable percentage of initial points for the selection.

4.3 System Solvers

Both MATLAB and CUSP offer an abundance of system solvers, most falling in the Krylov subspace of solvers [12]. Among these, the Conjugate Gradient (CG) solver was first tested, due to its simplicity. As this solver worked without issue, the other more complicated solvers were left untested for some time. During final optimization, testing on other solvers available in CUSP as they were compared to the results and performance of the CG solver. The Biconjugate Gradient Stabilized (BiCGstab), Conjugate Residual (CR), and Generalized Minimum Residual (GMRES) methods were tested. The GMRES method has an extra parameter known as a restart value. Decreasing the restart value improves computation time at the expense of accuracy, while increasing it has the opposite effect.

Solver	Time	Avg. Err.
CG	10.7 s	0.055%
BiCGstab	16.3 s	0.055%
CR	11.6 s	0.054%
GMRES		
restart: 20	13.3 s	0.063%
restart: 50	17.2 s	0.054%

Table 4.1: System solver comparison for time and error.

Testing was done using the CUDA 2D ODETLAP implementation. The solver was changed for each test, while all other factors remained unchanged. The smoothness coefficient was set to $R = 0.1$, with 1% of points selected on each of five solver

iterations. The results of this test on the 2048×2048 test set can be found in Table 4.1. GMRES was tested for restart values of both 20 and 50. The CG method is optimal, as it provides the quickest computation while still providing the best accuracy. The other solver methods are more complicated, requiring more time for completion, and are typically intended to handle cases where the system is not easily solved by the simple CG method.

Each solver has the option of accepting a preconditioner. The intention of a preconditioner is to transform the system into a form that allows the solver to converge in a shorter amount of time. By default, if no preconditioner is specified, A is multiplied by a simple identity matrix. Another simple preconditioner is a diagonal matrix where the values for the diagonal come from the values along the diagonal in the A matrix. The last preconditioner tested was the smoothed aggregation, available in CUSP. The results of this test can be seen in Table 4.2. The identity and diagonal preconditioners are comparable, while the smoothed aggregation preconditioners took more time to compute and nearly doubled the memory usage. The default identity matrix was used for all results in this thesis. More testing would be required to determine if there are scenarios in which use of the diagonal preconditioner is beneficial to use.

Preconditioner	Time	Memory
Identity	10.7 s	1527 MB
Diagonal	10.9 s	1543 MB
Smoothed Aggregation	33.5 s	2977 MB

Table 4.2: Preconditioner comparison for time and peak memory usage.

5. RESULTS

All data collected for the presented results was obtained while running various ODETLAP implementations on the GPU server GeoXeon. Significant hardware include dual Intel Xeon 8 core, 16 thread 3.1 GHz CPUs, 128 GB of 1600 MHz DDR3 main memory, and an Nvidia Tesla Kepler K20x Computing Processor. The K20x has 2688 processing cores and 6 GB of memory. The CPU implementation runs on MATLAB version R2013a. Due to licensing restrictions, only a maximum of 16 CPU threads are utilized. The GPU implementation uses CUDA version 6.0, Thrust version 1.7, and CUSP version 0.4.

For practical use as a compression method, ODETLAP must be able to compress a data set in a timely manner. Reconstruction of data takes a fraction of the time that compression takes, as it requires only a single system solution. A timing comparison for ODETLAP in two dimensions was done, comparing the results attained using the MATLAB implementation to the CUDA implementation. The results can be found in Table 5.1. All tests were performed using five iterations with smoothness coefficient $R = 0.1$ and 1% of the total points selected on each iteration.

Data Set	MATLAB	CUDA
512x512	7.2 s	1.7 s
1024x1024	32.6 s	3.4 s
2048x2048	154.8 s	10.8 s

Table 5.1: MATLAB and CUDA elapsed time comparison for 2D ODETLAP.

The test verifies that the GPU CUDA implementation is capable of faster computation of ODETLAP for the purpose of compression. As the size of the data set increases, the parallelism of the large number of GPU cores is better utilized and the time improvement is greater. For the largest test, the CUDA implementation is more than fourteen times faster than the MATLAB implementation. A larger 4096×4096 test set was created, but there is not enough memory available on the GPU to test it. This set took more than thirty minutes to compress using MATLAB.

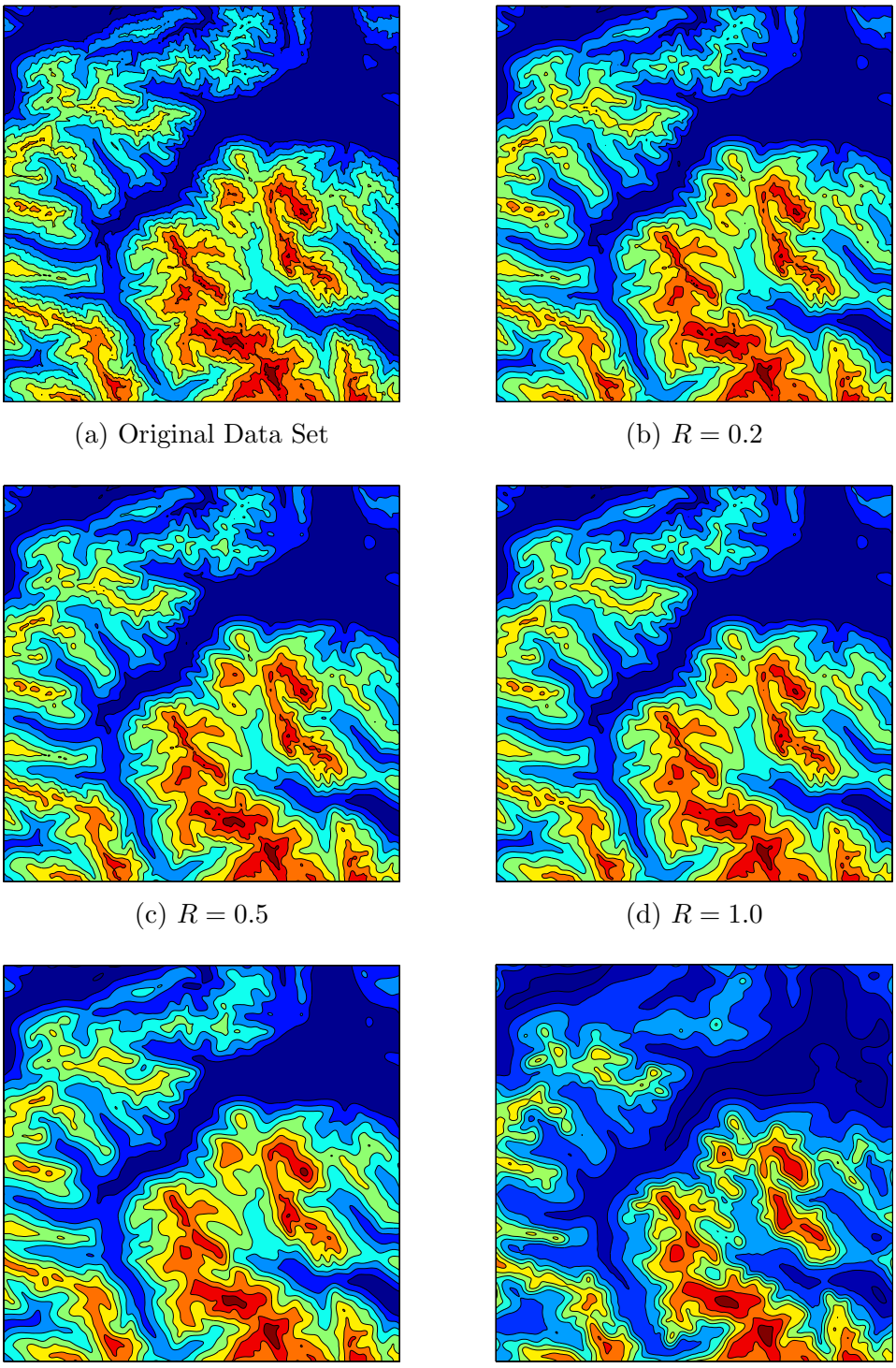
In order for the compressed data to yield accurate results upon reconstruction, a good smoothness coefficient value must be selected. A value of $R = 1.0$ corresponds to equal weighting between the averaging equations and the known point equations. If R is greater than 1.0, the averaging equations have a greater weighting, resulting in a smoother reconstruction. If R is less than 1.0, the known point equations have a greater weighting, resulting in a reconstruction with sharper value changes. The goal is then to select a value for R such that the reconstruction is not overly smoothed while minimizing the sharpness of gradients. Typically, the best value will be directly related to the inherent smoothness of the original data.

R	Avg. Err.
0.2	0.8%
0.5	0.8%
1.0	0.9%
2.0	1.4%
5.0	7.9%
10.0	21.6%

Table 5.2: Smoothness comparison for 2D ODETLAP, 2048×2048 test set.

Table 5.2 shows the testing of various smoothness values (R) for a 2D test set when reconstructed using ODETLAP. The original input set is 512×512. For each test, there are five ODETLAP solver iterations, with 1% of points added for each iteration. This particular test performs quite well at $R = 1.0$ and even at $R = 2.0$. The error increases quite rapidly after $R = 5.0$. Values below $R = 0.5$ do not yield better results. In most cases, a value in the range of $R = 0.1$ to $R = 0.2$ will be ideal. The success of the high values of R is the result of the smoothness of the input data set. A comparison of the reconstructed data using the tested smoothness values can be seen in Figure 5.1.

Smoothness tests were also performed for three-dimensional and four-dimensional test sets. As with the test in two dimensions, there are five ODETLAP iterations and approximately 1% of points are selected for each iteration. As the data used for the



(a) Original Data Set (b) $R = 0.2$

(c) $R = 0.5$ (d) $R = 1.0$

(e) $R = 2.0$ (f) $R = 5.0$

Figure 5.1: Smoothness comparison of reconstructed data.

R	Avg. Err.
0.05	1.7%
0.1	1.7%
0.2	1.9%
0.5	2.4%
1.0	33.5%

Table 5.3: Smoothness comparison for 3D ODETLAP, $144 \times 73 \times 17$ test set.

three-dimensional test is a subset of the four-dimensional test set, the ideal smoothness values are comparable for both tests. The results for the three-dimensional test can be seen in Table 5.3 and the results for the four-dimensional test can be seen in Table 5.4. In both cases, the error becomes quite large moving to $R = 1.0$ and greater, as is to be expected. For such test sets, the ideal value is likely within the range of $R = 0.1$ and $R = 0.2$.

R	Avg. Err.
0.05	1.2%
0.1	1.2%
0.2	1.4%
0.5	2.7%
1.0	43.4%

Table 5.4: Smoothness comparison for 4D ODETLAP, $144 \times 73 \times 17 \times 20$ test set.

Availability of memory on the GPU is the primary limiting factor with regard to the maximum size of the data set that can be tested. Table 5.5 shows the peak memory usage for test sets of various sizes. As the total number of data points increases, the total memory usage increases linearly. In addition to the total number of points in the system, the number of data dimensions of the input set also affects the peak memory usage. The data point for the 2048×2048 test set, while having more total points than the $144 \times 73 \times 17 \times 20$ set, uses just more than half the total memory. As the number of dimensions increases, the number of neighbors considered

Test Set	Data Points	Memory	Time	Avg. Err
$18 \times 73 \times 17 \times 20$	446760	444 MB	5.1 s	1.1%
$36 \times 73 \times 17 \times 20$	893520	802 MB	9.8 s	1.1%
$72 \times 73 \times 17 \times 20$	1787040	1520 MB	20.1 s	1.1%
$144 \times 73 \times 17 \times 20$	3574080	2954 MB	41.7 s	1.1%
2048×2048	4194304	1527 MB	10.6 s	0.2%

Table 5.5: ODETLAP peak GPU memory usage and time comparison.

in the averaging equations increases. This suggests that the initial A matrix requires a larger number of nonzero values, increasing the total memory size of A.

ODETLAP is able to achieve good levels of compression due to its use of only a small subset of points from the original data set. There is some added expense due to the need to store the point locations in addition to the point values. The results for compression of this set using ODETLAP can be found in Table 5.6. Using the same test set of size $144 \times 73 \times 17 \times 20$, the uncompressed file size corresponds to the 3574080 each requiring 4 bytes of memory as they are stored as floats. Compressing this data directly using 7-Zip does yield some reduction in file size. ODETLAP is run for five iterations with smoothness $R = 0.2$, selecting approximately 1% of points per iteration. The resulting data values are assigned 16-bit levels to correspond to the precision of the original floating point data, though a smaller number of bits would likely be sufficient. The resulting data values and locations are then compressed using 7-Zip, and the final file size is then less than 2% of the size of the original input data set.

	File Size	Avg. Err.
Uncompressed	13962 KB	0.0%
7-Zip	4035 KB	0.0%
ODETLAP	1760 KB	1.1%
ODETLAP + 7-Zip	266 KB	1.1%

Table 5.6: ODETLAP compressed file size.

6. FUTURE WORK

While computation time is vastly improved through the use of CUDA, the ODETLAP implementation is restricted by the memory available on the GPU. As technology improves, the amount of memory available will surely increase. Despite this, there may perhaps exist opportunities to optimize the memory usage of this implementation. If data transfer with main memory could be avoided, one possible solution would be to solve smaller chunks on the GPU and reconstruct the final product [25]. As peak memory usage occurs during the solution step, if the size of the A matrix is reduced, the size of the input data can be increased.

An area of improvement with regard to the final compressed data size is addressing the forbidden zone problem. As no clear solution exists for the determination of forbidden zone directly in parallel, a modified forbidden zone technique could be utilized. One possibility would be to break the selection into multiple steps during each worst error selection, first performing the selection in parallel and then removing points that would fall into a forbidden zone afterwards. Alternatively, a method for removing unnecessary points from the final set could be considered.

As ODETLAP is expandable into any number of dimensions, it would be worth creating a dynamic implementation capable of receiving a data set of any known size. In effect, if the number of data dimensions can be determined during the initialization stage, the initial matrix construction can dynamically determine the number of neighbors and averaging equations required. A more dynamic approach would be required for the initial point selection, as gridded selection becomes poor for a system with many dimensions. System solution, iterative worst error selection, and compression are unchanged for higher dimensions.

7. CONCLUSION

Compression allows for the transmission and storage of larger sets of data. To effectively compress multidimensional data sets, autocorrelation should be utilized in all dimensions of the data. ODETLAP can be used as a compression technique that is capable of scaling into any number of dimensions. Utilizing Laplace's equation, ODETLAP is capable of approximating a data set from a small set of points from that data set. As a compression method, then, ODETLAP is used by selecting a small set of points of interest from the original data set and lossily compressing them. These points are selected iteratively, ensuring that the points of greatest error are selected in an effort to reduce overall error on subsequent iterations and the error in the final approximation. The data can then accurately reconstructed from this small set of points.

Parallel programming using CUDA on the GPU provides a means for the efficient compression of multidimensional gridded data. The ODETLAP method is computationally intensive, as it requires that a large, overdetermined system be constructed and solved numerically. The Thrust and CUSP libraries provided by CUDA are used for both the construction and solving of the large, sparse overdetermined system required by ODETLAP. Computational time is vastly improved through the use of the GPU when compared to an implementation utilizing minimal parallel programming on the CPU. While the computation time is greatly improved through the use of CUDA, the GPU restricts the maximum size of data that can be compressed due to the limited memory available on the GPU.

REFERENCES

[1] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2012, version 0.4.0. [Online]. Available: http://cusp-library.googlecode.com [Accessed: 02/19/2013]

[2] B. Ben-Moshe, L. Serruya, and A. Shamir, "Image compression terrain simplification," in *Proceedings of the 19th Annual Canadian Conference on Computational Geometry, 2007*, Carleton University, Ottawa, Canada, Aug. 2007.

[3] D. N. Benedetti, W. R. Franklin, and W. Li, "CUDA-accelerated ODETLAP: A parallel lossy compression implementation," in *23rd Fall Workshop on Computational Geometry*, City College, New York City, USA, Oct. 2013, extended abstract.

[4] R. Boisvert, R. Pozo, and K. Remington, "The matrix market exchange formats: Initial design," *Technical Report NISTIR-5935*, Dec. 1996.

[5] E. Christophe and W. A. Pearlman, "Three-dimensional SPIHT coding of volume images with random access and resolution scalability," in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, PA, USA, Oct. 2006, pp. 1897 – 1901.

[6] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The JPEG2000 still image coding system: an overview," *IEEE Transactions on Consumer Electronics*, vol. 46, no. 4, pp. 1103 – 1127, Nov. 2000.

[7] G. P. Compo. 20th century reanalysis v2. NOAA/OAR/ESRL PSD, Boulder, CO, USA. [Online]. Available: http://www.esrl.noaa.gov/psd/ [Accessed: 11/11/2013]

[8] W. R. Franklin, M. Inanc, and Z. Xie, "Two novel surface representation techniques," in *Proceedings of the AutoCarto Research Symposium, 2006*, Vancouver, WA, USA, Jun. 2006.

[9] W. R. Franklin, T.-Y. Lau, and P. Fox, "CUDA-accelerated HD-ODETLAP: Lossy high dimensional gridded data compression," in *2012 International Workshop on Modern Accelerator Technologies for GIScience (MAT4GIScience 2012)*, Columbus, OH, USA, Sep. 2012.

[10] E. A. Hakkennes and S. Vassiliadis, "Hardwired Paeth codec for portable network graphics (PNG)," in *Proceedings on the 25th EUROMICRO Conference, 1999*, vol. 2, Milan, Sep. 1999, pp. 318 – 325.

[11] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.7.0. [Online]. Available: http://thrust.github.io/ [Accessed: 02/19/2013]

[12] I. M. Jaimoukha and E. M. Kasenally, "Krylov subspace methods for solving large lyapunov equations," *SIAM Journal on Numerical Analysis*, vol. 31, no. 1, Feb. 1994.

[13] Y. Li, "CUDA-accelerated HD-ODETLAP: A high dimensional geospatial data compression framework," Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, NY, USA, Jul. 2011. [Online]. Available: http://www.ecse.rpi.edu/Homepages/wrf/theses/li-phd.pdf [Accessed: 09/30/2012]

[14] Y. Li and W. R. Franklin, "4D-ODETLAP: A novel high-dimensional compression method on time-varying geospatial data," in *Geospatial Data and Geovisualization: Environment, Security, and Society, a special joint symposium of ISPRS Technical Commission IV & AutoCarto 2010*, Orlando, FL, USA, Nov. 2010.

[15] Y. Li, T.-Y. Lau, C. Stuetzle, P. Fox, and W. R. Franklin, "3D oceanographic data compression using 3D-ODETLAP," in *18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS 2010)*, San Jose, CA, USA, Nov. 2010.

[16] P. Lindstrom and V. Pascucci, "Visualization of large terrains made easy," in *IEEE Proceedings on Visualization, 2001*, San Diego, CA, USA, Oct. 2001, pp. 363 – 574.

[17] Y. Liu and W. A. Pearlman, "Four-dimensional wavelet compression of 4-D medical images using scalable 4-D SBHP," in *Proceedings of the Data Compression Conference*, Snowbird, UT, USA, Mar. 2007, pp. 233 – 242.

[18] MATLAB, *version 8.1.0 (R2013a)*. Natick, Massachusetts: The MathWorks Inc., 2013.

[19] MATLAB. (2014) Mat-file format (r2014a). Natick, Massachusetts. [Online]. Available: http://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf [Accessed: 06/16/14]

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40 – 53, March/April 2008.

[21] T. N. Ospina, M. Iregui, J. Victorino, and E. Romero, "Efficient access to compressed 3D and 4D MRI using JPEG2000," in *SPIE Proceedings on Medical Imaging 2011: Advanced PACS-based Imaging Informatics and Therapeutic Applications*, vol. 7967, Lake Buena Vista, Florida, USA, Feb. 2011.

[22] C. C. Paige and M. A. Saunders, "LSQR: An algorithm for sparse linear equations and sparse least squares," *ACM Transactions on Mathematical Software*, vol. 8, no. 1, pp. 43 – 71, Mar. 1982.

[23] I. Pavlov. (2014, Jun.) 7-zip. [Online]. Available: http://www.7-zip.org/ [Accessed: 06/12/2014]

[24] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 243 – 250, Jun. 1996.

[25] J. Stookey, "Parallel terrain compression and reconstruction," Master's thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, Feb. 2008. [Online]. Available: http://www.ecse.rpi.edu/Homepages/wrf/theses/stookey-ms.pdf [Accessed: 09/24/2012]

[26] J. Stookey, Z. Xie, B. Cutler, W. R. Franklin, D. M. Tracy, and M. V. A. Andrade, "Parallel ODETLAP for terrain compression and reconstruction," in *16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2008)*, W. G. Aref *et al.*, Eds., Irvine, CA, USA, Nov. 2008.

[27] G. Turk and B. Mullins. (2007, Sep.) Puget sound terrain. [Online]. Available: http://www.cc.gatech.edu/projects/large_models/ps.html [Accessed: 02/08/2013]

[28] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii – xxxiv, Feb. 1992.

[29] Z. Xie, "Representation, compression and progressive transmission of digital terrain data using over-determined Laplacian partial differential equations," Master's thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, Mar. 2008. [Online]. Available: http://www.ecse.rpi.edu/Homepages/wrf/theses/xie-ms.pdf [Accessed: 09/24/2012]

[30] Z. Xie, W. R. Franklin, B. Cutler, M. A. Andrade, M. Inanc, and D. M. Tracy, "Surface compression using over-determined Laplacian approximation," in *SPIE Proceedings on Advanced Signal Processing Algorithms, Architectures, and Implementations XVII*, vol. 6697-15. San Diego, CA, USA: International Society for Optical Engineering, Aug. 2007.

A. SAMPLE CODE

There are four distinct implementations created as part of this thesis: A MATLAB implementation in two dimensions and CUDA implementations for two dimensions, three dimensions, and four dimensions. The full code for each implementation, so small samples are included to showcase major sections of the ODETLAP process in both MATLAB and CUDA.

A.1 Initial A Matrix Construction: MATLAB

```
% Specify dimension size and number of nonzero elements
nnz = 5*n*n;

% Build the initial A matrix, for the general purpose equations.
row_ind(1:nnz) = [1:n^2, 1:n^2, 1:n^2, 1:n^2, 1:n^2];
col_ind(1:nnz) = [1:n^2, 1, 1:n^2-1, 2:n^2, 1, 1:n, 1:(n-1)*n, ...
                  (n+1):n^2, 1:n];
value(1:nnz) = [ones(1,n^2)*4*R, 0, ones(1,(n^2-1)*2)*-1*R, ...
                zeros(1,n+1), ones(1,(n-1)*2*n)*-1*R, zeros(1,n)];

% Add special equations for edges, iterating with 1D indexing
for ind = 1:n^2
    % Index falls on left/right edge
    x = mod(ind,n);
    if (x == 1 || x == 0)
        value(ind) = value(ind) - 2*R;
        value(ind+n^2) = 0;
        value(ind+2*n^2) = 0;
    end

    % Index falls on top/bottom edge
    y = ind;
    if (y <= n || y > n^2 - n)
        value(ind) = value(ind) - 2*R;
        value(ind+3*n^2) = 0;
        value(ind+4*n^2) = 0;
    end
end

% Build sparse A matrix and zero b vector
A = sparse(row_ind, col_ind, value);
b = zeros(1,n^2);
```

A.2 Initial A Matrix Construction: CUDA 2D

```
// Device function to convert from 1D indexing to 2D indexing
__device__ void get_index(int i, int *p) {
  p[0] = i % d_nx;
  p[1] = (i / d_nx) % d_ny;
}

// Thrust Remove Left/Right Faces Template
template<typename T>
struct lr_faces : public std::binary_function<T,T,T> {
  __device__ T operator() (const T &v, const T &i) const {
    int p[2];  // location of point in 2 dimensions
    get_index(i, p);

    if (p[0] == 0 || p[0] == d_nx - 1) // Point belongs on left or right face
      return T(v) > 0 ? T(v-2*d_r) : T(0);
    else
      return T(v);
  }
};

// Thrust Remove Top/Bottom Faces Template
template<typename T>
struct tb_faces : public std::binary_function<T,T,T> {
  __device__ T operator() (const T &v, const T &i) const {
    int p[2];  // location of point in 2 dimensions
    get_index(i, p);

    if (p[1] == 0 || p[1] == d_ny - 1) // Point belongs on top or bottom face
      return T(v) > 0 ? T(v-2*d_r) : T(0);
    else
      return T(v);
  }
};

// Initial A matrix construction
void init_a(coo_matrix<I,D,M> &a) {
  // ==== CENTRAL EQUATIONS ====

  // Row Indices -- Center, Left, Right, Up, Down
  for (int i = 0; i < 5; i++)
    thrust::sequence(a.row_indices.begin() + i*n,
                     a.row_indices.begin() + i*n + n);

  // Column Indices
  thrust::sequence(a.column_indices.begin(),
                   a.column_indices.begin() + n);                // Center
  thrust::fill(  a.column_indices.begin() + n,
                   a.column_indices.begin() + n + 1, 0);         // Buffer
  thrust::sequence(a.column_indices.begin() + n + 1,
                   a.column_indices.begin() + 2*n);              // Left
  thrust::sequence(a.column_indices.begin() + 2*n,
                   a.column_indices.begin() + 3*n - 1,
                   1);                                           // Right
```

```
  thrust::fill(  a.column_indices.begin() + 3*n - 1,
                 a.column_indices.begin() + 3*n + nx,
                 0);                                          // Buffer
  thrust::sequence(a.column_indices.begin() + 3*n + nx,
                 a.column_indices.begin() + 4*n);             // Up
  thrust::sequence(a.column_indices.begin() + 4*n,
                 a.column_indices.begin() + 5*n - nx,
                 nx);                                         // Down
  thrust::fill(  a.column_indices.begin() + 5*n - nx,
                 a.column_indices.begin() + 5*n,
                 0);                                          // Buffer

  // Fill values (4 center, -1 edges)
  thrust::fill(a.values.begin(), a.values.begin() + n, 4*r);   // Center
  thrust::fill(a.values.begin() + n, a.values.begin() + 5*n, -1*r); // Boundaries

  // ==== SPECIAL CASE BOUNDARY EQUATIONS ====
  counting_iterator<I> seq(0);

  // Remove left/right faces
  thrust::transform(a.values.begin()  , a.values.begin() + n, seq,
      a.values.begin()  , lr_faces<D>());
  thrust::transform(a.values.begin() + n, a.values.begin() + 2*n, seq,
      a.values.begin() + n, lr_faces<D>());
  thrust::transform(a.values.begin() + 2*n, a.values.begin() + 3*n, seq,
      a.values.begin() + 2*n, lr_faces<D>());

  // Remove top/bottom faces
  thrust::transform(a.values.begin()  , a.values.begin() + n, seq,
      a.values.begin()  , tb_faces<D>());
  thrust::transform(a.values.begin() + 3*n, a.values.begin() + 4*n, seq,
      a.values.begin() + 3*n, tb_faces<D>());
  thrust::transform(a.values.begin() + 4*n, a.values.begin() + 5*n, seq,
      a.values.begin() + 4*n, tb_faces<D>());

  // COO format requires that its entries be sorted by row
  a.sort_by_row();
}
```

A.3 Initial Point Selection: CUDA 4D

```
// Test for nonzero value
template<typename T>
struct is_nonzero : public unary_function<T,bool> {
  __host__ __device__ bool operator()(const T &i) {
    return i > 0;
  }
};

// Find absolute value
template<typename T>
struct absolute_value : public unary_function<T,T> {
```

```
  __host__ __device__ T operator()(const T &i) const {
    return i < T(0) ? -i : i;
  }
};

// Initial Point Selector (Uniform)
template<typename T>
struct initial_points : public std::unary_function<T,T> {
  __device__ T operator() (const T &i) const {
    int p[4];  // location of point in 4 dimensions
    get_index(i, p);

    // Select 1 out of every 81 points (uniform spacing of 3 in all 4 dims)
    if ((p[0]%3 == 1 && p[1]%3 == 1 && p[2]%3 == 1 && p[3]%3 == 1))
      return T(i);
    else
      return T(0);
  }
};

// Add initial (random) selected points)
void add_initial(coo_matrix<I,D,M> &a, array1d<D,M> &b, array1d<D,M> vi) {
  device_vector<I> points(n);

  // Select points
  thrust::sequence(points.begin(), points.end());
  thrust::transform(points.begin(), points.end(), points.begin(),
      initial_points<I>());

  // Sort and count nonzero points
  thrust::sort(points.begin(), points.end(), thrust::greater<I>());
  nadd = thrust::count_if(points.begin(), points.end(), is_nonzero<I>());

  // Resize A matrix and b vector
  a.resize(n+nadd, n, nnz+nadd);
  b.resize(n+nadd);

  // Add point locations to A matrix
  thrust::sequence(a.row_indices.begin() + nnz, a.row_indices.begin() + nnz +
      nadd, n);
  thrust::copy(points.begin(), points.begin() + nadd, a.column_indices.begin() +
      nnz);
  thrust::fill(a.values.begin() + nnz, a.values.begin() + nnz + nadd, 1);

  // Add point values to b vector
  device_vector<D> val(n);
  thrust::gather(points.begin(), points.end(), vi.begin(), val.begin());
  thrust::copy(val.begin(), val.begin() + nadd, b.begin() + n);
}
```

A.4 Solver and Main: CUDA 4D

```
// Approximate solution to Ax = b
void solve(coo_matrix<I,D,M> &a, array1d<D,M> &b, array1d<D,M> &vo) {
  // Normalize --> A'A and A'b
  coo_matrix<I,D,M> at;          // transpose of A
  transpose(a, at);

  array1d<D,M> atb(n);           // A'b
  multiply(at, b, atb);

  coo_matrix<I,D,M> ata;         // A'A
  multiply(at, a, ata);

  // Stabilized Bicongugate Gradient solution to Ax = b
  default_monitor<D> monitor(atb, 100, 1e-6);
  identity_operator<D,M> precond(n,n); // set preconditioner
  gmres(ata, vo, atb, restart, monitor, precond);

  // Get total memory usage
  cudaMemGetInfo(&mfree, &mtotal);
}

// Main function
int main(int argc, char *argv[]) {
  // Print welcome message
  welcome();

  // Initial time measurement
  double start = get_time();
  double time = start;

  // Load input matrix
  std::cout << "\n Reading file for input" << std::flush;
  array1d<D,M> vi;
  read_matrix_market_file(vi, infile);
  find_extrema(vi);
  update_time(time);

  // Construct initial matrix for Ax = b
  std::cout << "\n Construct initial A matrix" << std::flush;
  array1d<D,M> b(n,0);
  coo_matrix<I,D,M> a(n, n, nnz);
  init_a(a);            // populate A matrix
  update_time(time);

  // Initial (random) point selection
  std::cout << " Initial point selection" << std::flush;
  add_initial(a, b, vi);
  update_time(time);

  // Initial solve
  std::cout << " Initial system solution" << std::flush;
  array1d<D,M> vo(n,0); // Solution matrix
  solve(a, b, vo);
```

```
  // Calculate error
  array1d<D,M> e(n,0);  // Error matrix
  double err = total_err(vi, vo, e);
  update_time(time);
  std::cout << "  " << nadd << " pts ("
          << float(nadd)/float(n)*100 << "%)" << std::endl;
  std::cout << " Avg. residual: " << err << "\n" << std::endl;

  // Iteratively add new points
  int iteration = 0;
  while (iteration < iter_stop) {
    iteration++;

    // Point selection (greatest error)
    std::cout << " Iteration " << iteration << "\t\t" << std::flush;
    add_worst(a, b, vi, e);

    // Solve system and calculate error
    solve(a, b, vo);
    err = total_err(vi, vo, e);

    // Finish printing
    update_time(time);
    std::cout << "  " << nadd << " pts ("
           << float(nadd)/float(n)*100 << "%)" << std::endl;
    std::cout << " Avg. residual: " << err << "\n" << std::endl;
  }

  // Encode and compress
  std::cout << " Encode and compress\t" << std::flush;
  encode(a, b);
  update_time(time);

  // Print results
  results(err, start);
}

// Approximate solution to Ax = b
void solve(coo_matrix<I,D,M> &a, array1d<D,M> &b, array1d<D,M> &vo) {
  // Normalize --> A'A and A'b
  coo_matrix<I,D,M> at;          // transpose of A
  transpose(a, at);

  array1d<D,M> atb(n);           // A'b
  multiply(at, b, atb);

  coo_matrix<I,D,M> ata;         // A'A
  multiply(at, a, ata);

  // Stabilized Bicongugate Gradient solution to Ax = b
  default_monitor<D> monitor(atb, 100, 1e-6);
  identity_operator<D,M> precond(n,n); // set preconditioner
  gmres(ata, vo, atb, restart, monitor, precond);
```

```
  // Get total memory usage
  cudaMemGetInfo(&mfree, &mtotal);
}
```