Reducing Memory Access Latencies using Data Compression in Sparse, Iterative Linear Solvers

An All-College Thesis

College of Saint Benedict/Saint John's University

by Neil Lindquist
April 2018

Project Title: Reducing Memory Access Latencies using Data Compression in Sparse, Iterative Linear Solvers
Approved by:

Mike Heroux
Scientist in Residence

Robert Hesse
Associate Professor of Math

Jeremy Iverson
Assistant Professor of Computer Science

Bret Benesh
Chair, Department of Mathematics

Imad Rahal
Chair, Department of Computer Science

Director, All College Thesis Program

Abstract

Contents

1 Introduction

2 Background

2.1 Conjugate Gradient

Conjugate Gradient is the iterative solver used by HPCG [2]. Symmetric, positive definite matrices will guarantee the converge of Conjugate Gradient to the correct solution within n iterations, where n is the number of dimensions, when using exact algebra [8]. More importantly, Conjugate Gradient can be used as in iterative method, providing a solution, $\vec{x}$, where $\left\|\mathbf{A}\vec{x} - \vec{b}\right\|$ is within some tolerance, ϵ, after significantly fewer than n iterations, allowing it to find solutions to problems where even n iterations is infeasible [9].

To understand the Conjugate Gradient, first consider the quadratic form of $\mathbf{A}\vec{x} = \vec{b}$. The quadratic form is a function $f : \mathbb{R}^n \to \mathbb{R}$ where

$$f(\vec{x}) = \tfrac{1}{2}\vec{x}^T\mathbf{A}\vec{x} - \vec{b}\cdot\vec{x} + c \tag{1}$$

for some $c \in \mathbb{R}$. Note that

$$\nabla f\left(\vec{x}\right) = \tfrac{1}{2}\left(\mathbf{A} + \mathbf{A}^T\right)\vec{x} - \vec{b}$$

Then, when $\mathbf{A}$ is symmetric,

$$\nabla f(\vec{x}) = \mathbf{A}\vec{x} - \vec{b}$$

So, the solution to $\mathbf{A}\vec{x} = \vec{b}$ is the sole critical point of f [7]. Since $\mathbf{A}$ is the Hessian matrix of f at the point, if $\mathbf{A}$ is positive definite, then that critical point is a minimum. Thus, if $\mathbf{A}$ is a symmetric, positive definite matrix, then the minimum of f is the solution to $\mathbf{A}\vec{x} = \vec{b}$ [9].

The method of Steepest Decent is useful for understanding Conjugate Gradient, because they both use a similar approach to minimize Equation 1, and thus solve $\mathbf{A}\vec{x} = \vec{b}$. This shared approach is to take an initial $\vec{x}_0$ and move downwards in the steepest direction, within certain constraints, of the surface defined by Equation 1 [7]. Because the gradient at a point is the direction of maximal increase, $\vec{x}$ should be moved in the opposite direction of the gradient. Thus, to compute the next value of $\vec{x}$, use

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i\vec{r}_i \tag{2}$$

for some $\alpha_i > 0$ and where $\vec{r}_i = -\nabla f\left(\vec{x}_i\right) = \vec{b} - \mathbf{A}\vec{x}_i$ is the residual of $\vec{x}_i$. Since $\mathbf{A}\vec{x} = \vec{b}$ is the only critical point and a minimum of the quadratic function, f, the ideal value of α_i is the one that minimizes $f\left(\vec{x}_{i+1}\right)$. Thus, choose α_i such that

$$\begin{aligned} 0 &= \tfrac{\mathrm{d}}{\mathrm{d}\alpha_i} f\left(\vec{x}_{i+1}\right) \\ &= \tfrac{\mathrm{d}}{\mathrm{d}\alpha_i} f\left(\vec{x}_i + \alpha\vec{r}_i\right) \\ \alpha_i &= \frac{\vec{r}_i\cdot\vec{r}_i}{\vec{r}_i\cdot\mathbf{A}\vec{r}_i}. \end{aligned}$$

Algorithm 1 Steepest Decent [9]

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
for $i = 0, 1, \ldots$ **until** $\|\vec{r}_i\| \leq \epsilon$ **do**
  $\alpha_i \leftarrow \frac{\vec{r}_i \cdot \vec{r}_i}{\vec{r}_i \cdot \mathbf{A}\vec{r}_i}$
  $\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{r}_i$
  $\vec{r}_{i+1} = \vec{r}_i - \alpha \mathbf{A}\vec{r}_i$
end for

Note that by using Equation 2, we can derive

$$\vec{r}_{i+1} = \vec{r}_i - \alpha \mathbf{A}\vec{r}_i. \tag{3}$$

Because $\mathbf{A}\vec{r}_i$ is already computed to find α_i, using Equation 3 to compute the residual results in one less matrix-vector product per iteration. Algorithm 1 shows the resulting algorithm.

Example 1. Consider the linear system

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

and use $c = 0$. Note that the solution is

$$\vec{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

When starting at the origin, the iteration of Method of Steepest Decent becomes

$$\begin{aligned}
\vec{x}_0 &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} & \vec{r}_0 &= \begin{bmatrix} 5 \\ 5 \end{bmatrix} & \alpha_0 &= 2/7 \\
\vec{x}_1 &= \begin{bmatrix} 10/7 \\ 10/7 \end{bmatrix} & \vec{r}_1 &= \begin{bmatrix} 5/7 \\ -5/7 \end{bmatrix} & \alpha_1 &= 2/3 \\
\vec{x}_2 &= \begin{bmatrix} 40/21 \\ 20/21 \end{bmatrix} & \vec{r}_2 &= \begin{bmatrix} 5/21 \\ 5/21 \end{bmatrix} & \alpha_2 &= 2/7 \\
\vec{x}_3 &= \begin{bmatrix} 290/147 \\ 50/49 \end{bmatrix} & \vec{r}_3 &= \begin{bmatrix} 5/147 \\ -5/147 \end{bmatrix} & \alpha_3 &= 2/3 \\
&\vdots & &\vdots & &\vdots
\end{aligned}$$

The $\vec{x}_i$'s are plotted with a contour graph of the quadratic form in Figure 1. □

The Conjugate Directions family of linear solvers, of which Conjugate Gradient is a member of, attempts to improve on the number of iterations needed by Steepest Decent. [9]. Note that, in Example 1, the directions of $\vec{r}_0$ and $\vec{r}_2$ are the same and the directions of $\vec{r}_1$ and $\vec{r}_3$ are the same. Thus, the same direction has to be traversed multiple times. Additionally, note that the two sets of residual directions are perpendicular to each other. Conjugate Directions attempts

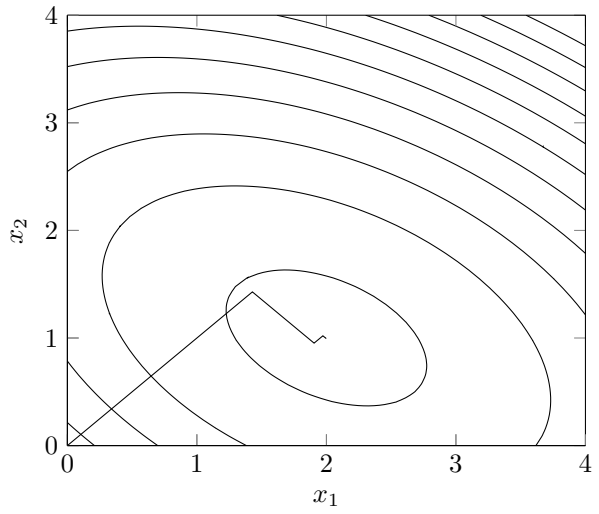

Figure 1: Contour graph of the quadratic function and the first six values of $\vec{x}$ produced by steepest decent for Example 1

to improve on this, by making the search directions, $\vec{d}_0, \vec{d}_1, \ldots$, **A**-orthogonal to each other and only moving $\vec{x}$ once in each search direction. Two vectors, $\vec{u}, \vec{v}$ are **A**-orthogonal, or conjugate, if $\vec{u}^T \mathbf{A} \vec{v} = 0$. The requirement for Conjugate Directions is to make $\vec{e}_{i+1}$ **A**-orthogonal to $\vec{d}_i$, where $\vec{e}_i = \vec{x}_i - \mathbf{A}^{-1}\vec{b}$ is the error of $\vec{x}_i$. The computation of α_i changes to find the minimal value along $\vec{d}_i$ instead of $\vec{r}_i$.

$$\alpha_i = \frac{\vec{d}_i^T \vec{r}_i}{\vec{d}_i^T \mathbf{A} \vec{d}_i}.$$

Conjugate Gradient is a form of Conjugate Directions where the residuals are made to be **A**-orthogonal to each other [9]. This is done using the Conjugate Gram-Schmidt Process. To do this, each search direction, $\vec{d}_i$ is computed by taking $\vec{r}_i$ and removing any components that are not **A**-orthogonal to the previous $\vec{d}$'s. So, let $\vec{d}_0 = \vec{r}_0$ and for $i > 0$ let

$$\vec{d}_i = \vec{r}_i + \sum_{k=0}^{i-1} \beta_{(i,k)} \vec{d}_k$$

with $\beta_{(i,k)}$ defined for $i > k$. Then, solving for $\beta_{(i,k)}$ gives

$$\beta_{(i,k)} = -\frac{\vec{r}_i \cdot \mathbf{A}\vec{d}_i}{\vec{d}_j \cdot \mathbf{A}\vec{d}_j}.$$

Note that each residual is orthogonal to the previous search directions, and thus the previous residuals. So, it can be shown that $\vec{r}_{i+1}$ is **A**-orthogonal to

Algorithm 2 Conjugate Gradient [8]

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
$\vec{d}_0 \leftarrow \vec{r}_0$
for $i = 0, 1, \ldots$ **until** $\|\vec{r}_i\| \leq \epsilon$ **do**
  $\alpha_i \leftarrow \frac{\vec{r}_i \cdot \vec{r}_i}{\vec{d}_i \cdot \mathbf{A}\vec{d}_i}$
  $\vec{x}_{i+1} \leftarrow \vec{x}_i + \alpha_i \vec{d}_i$
  $\vec{r}_{i+1} \leftarrow \vec{r}_i + \alpha_i \mathbf{A}\vec{d}_i$
  $\beta_{i+1} \leftarrow \frac{\vec{r}_{i+1} \cdot \vec{r}_{i+1}}{\vec{r}_i \cdot \vec{r}_i}$
  $\vec{r}_{i+1} + \beta_{i+1}\vec{d}_i$
end for

all previous search directions, except $\vec{d}_i$ [9]. Then, $\beta_{(i,k)} = 0$ for $i - 1 \neq k$. To simplify notation, let $\beta_i = \beta_{(i,i-1)}$. So, each new search direction can then be computed by

$$\vec{d}_i = \vec{r}_i + \beta_i \vec{d}_{i-1}.$$

Algorithm 2 shows the final Conjugate Gradient algorithm.

Example 2. Consider the linear system used in Example 1 where

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}.$$

The result of applying Conjugate Gradient is

$$\vec{x}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \vec{r}_0 = \begin{bmatrix} 5 \\ 5 \end{bmatrix} \quad \vec{d}_0 = \begin{bmatrix} 5 \\ 5 \end{bmatrix} \quad \alpha_0 = 2/7$$
$$\vec{x}_1 = \begin{bmatrix} 10/7 \\ 10/7 \end{bmatrix} \quad \vec{r}_1 = \begin{bmatrix} 5/7 \\ -5/7 \end{bmatrix} \quad \beta_1 = 1/49 \quad \vec{d}_1 = \begin{bmatrix} 40/49 \\ -30/49 \end{bmatrix} \quad \alpha_1 = 7/10$$
$$\vec{x}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad \vec{r}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that after two iterations, $\vec{x}$ reaches the exact solution, compared to the iterations of Steepest Decent in Example 1. Figure 2 shows the values of $\vec{x}$ with the contour graph of the quadratic function. □

One way to improve the Conjugate Gradient method is to precondition the system [8]. Instead of solving the original system, $\mathbf{A}\vec{x} = \vec{b}$, Conjugate Gradient solves $\mathbf{M}^{-1}\left(\mathbf{A}\vec{x} - \vec{b}\right) = 0$ instead, where $\mathbf{M}^{-1}$ is the preconditioner. Note that $\mathbf{M}$ should be similar to $\mathbf{A}$, but $\mathbf{M}^{-1}$ should be easier to compute than $\mathbf{A}^{-1}$. Algorithm 3 shows the preconditioned variant of the Conjugate Gradient.

2.2 Multigrid Preconditioner with Gauss-Seidel Step

Multigrid solvers are a class of methods designed for solving discretized partial differential equations (PDEs) and take advantage of more information that just

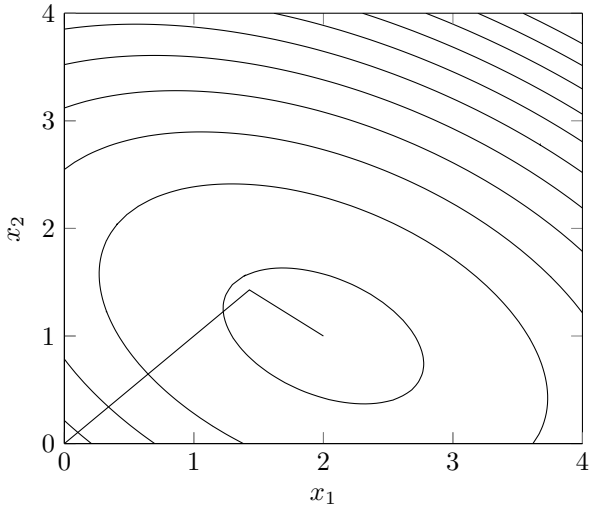

Figure 2: Contour graph of the quadratic function and the each value of $\vec{x}$ produced by Conjugate Gradient for Example 2

the coefficient matrix and the right hand side [8]. In particular, the solvers use discretizations with different mesh sizes to improve performance of relaxation based solvers. In HPCG, a multigrid solver with high tolerance is used as the preconditioner [2]. Because the solver provides an approximation to $\mathbf{A}^{-}1$, the preconditioned matrix is an approximation of $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. This reduces the condition number of the linear system, and so, reduces the number of iterations needed for Conjugate Gradient to converge [8].

The multigrid method uses meshes of different sizes to improve performance of a relaxation style iterative solver [8]. Most relaxation type iterative solvers are able to quickly reduce the components of the residual in the direction of eigenvectors associated with large eigenvalues for the iteration matrix. Such eigenvectors are called high frequency modes. The other components, in the direction of eigenvectors called low frequency modes, are difficult to reduce with standard relaxation. However on a courser mesh, many of these low frequency modes are mapped to high frequency modes [8]. Thus, by applying a relaxation type iterative solver at various mesh sizes, the various components of the residual can be reduced quickly.

In HPCG, a symmetric Gauss-Seidel iteration is used by the multigrid as the relaxation iteration solver at each level of coarseness [2]. The symmetric Gauss-Seidel iteration consists of a forward Gauss-Seidel iteration followed by a backward Gauss-Seidel iteration. Letting $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$ where $\mathbf{L}$ is strictly lower triangular, $\mathbf{D}$ is diagonal and $\mathbf{U}$ is strictly upper triangular, the iteration

Algorithm 3 Preconditioned Conjugate Gradient [8]

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
$\vec{z}_0 \leftarrow \mathbf{M}^{-1}\vec{r}_0$
$\vec{d}_0 \leftarrow \vec{z}_0$
for $i = 0, 1, \ldots$ **until** $\|\vec{r}_i\| \leq \epsilon$ **do**
  $\alpha_i \leftarrow \frac{\vec{r}_i \cdot \vec{z}_i}{\vec{d}_i \cdot \mathbf{A}\vec{d}_i}$
  $\vec{x}_{i+1} \leftarrow \vec{x}_i + \alpha_i \vec{d}_i$
  $\vec{r}_{i+1} \leftarrow \vec{r}_i + \alpha_i \mathbf{A}\vec{d}_i$
  $\vec{z}_{i+1} \leftarrow \mathbf{M}^{-1}\vec{r}_{i+1}$
  $\beta_{i+1} \leftarrow \frac{\vec{r}_{i+1} \cdot \vec{z}_{i+1}}{\vec{r}_i \cdot \vec{z}_i}$
  $\vec{d}_{i+1} \leftarrow \vec{z}_{i+1} + \beta_{i+1}\vec{d}_i$
end for

can be represented by

$$\vec{x}_i^* = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{L}\vec{x}_i^* - \mathbf{U}\vec{x}_i\right)$$
$$\vec{x}_{i+1} = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{U}\vec{x}_{i+1} - \mathbf{L}\vec{x}_i^*\right)$$

with $\vec{x}_i^*$ representing an intermediate vector. Note that while $\vec{x}_i^*$ and $\vec{x}_{i+1}$ are on both sides of the equation where they are respectively computed, they can be computed with this formulation by computing the entries in order as they become available for the product with L and U respectively.

2.3 Problem Setup of High Performance Conjugate Gradient

The problem used to create the linear system used by HPCG, and thus by this project, is a three dimensional partial differential equation (PDE) model [2]. This problem is approximating the function $u(x, y, z)$ over the three dimensional rectangular region $\Omega \in \mathbb{R}^3$ such that

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0,$$

with $u(x, y, z) = 1$ along the boundaries of Ω. Note that the solution is $u(x, y, z) = 1$ over Ω. The linear system is created by using the finite difference method with a 27-point stencil on the PDE over a rectangular grid with nodes of fixed distance. The matrix's diagonal consists of the value 26, and -1's fill the entries for the row's 26 grid neighbors. The right hand side of the equation has a value of 14 for corner points, 12 for edge points, 9 for side points and 0 for interior points [4]. The solution vector consists of all 1's.

HPCG uses an implementation of Conjugate Gradient algorithm with a multigrid preconditioner variant [2]. As HPCG is designed to emulate the performance characteristics of real world problems with out needing to be a robust

solver, it only uses 3 levels of grid coarseness with only a since smoother pass at the coarsest grid level. The smoother used by the multigrid is based on a symmetric Gauss-Seidel step, however each process uses the old value for entries located on other processes. The restriction operation simply samples half the points in dimension, resulting in a reduction of grid size by a factor of eight in each level of coarseness. To prolong the coarse grids, each coarse point is added to the fine point in was sampled from. The zero vector is used as the overall initial guess for x, as well as the initial guess for each grid level in the multigrid cycle.

2.4 Data Access Patterns of High Performance Conjugate Gradient

The way the matrix and vector data is accessed can provide limitations on attempts to compress the data. The Conjugate Gradient and Multigrid implementations in HPCG do not directly access the matrix and vector values, but instead use low level functions to actually manipulate the data structures [2]. These low level functions include copying a vector, setting a vector to zero, the dot product, a scaled vector sum, the matrix vector product, the symmetric Gauss-Seidel step, the multigrid restriction and the multigrid prolongation. Further data accessing functions exist in HPCG, however, they are not part of the timing. So, any additionally restrictions can be overcome by converting to an uncompressed format, applying the function, then recompressing. The low level functions used in the timed section of the code can be viewed together to produce the overall data access requirements, For the matrices, the matrices do not need to be mutable, the rows need to be readable in both a forward and backwards iteration, the data for a given row has no restriction on its read order, and the diagonal for a given row must be accessible. The vectors, on the other hand, need both random read and write access, with the writes being immediately accessible to future reads.

Copying a vector and setting a vector to zero provide the least data access requirements. Note that a vector's content can by copied by transferring the current representation of the values without any processing. Setting a vector to zero merely requires the ability to write vector values. Both of these functions add little to the data access requirements and are both simple to reimplement with alternative vector representation.

The dot product and sum of scaled vectors are both straightforward functions. Each of them iterates over two or three vectors and applies a few arithmetic operations. The dot product accumulates the sum of the product of the pair of vector entries across the iterations. The sum of scaled vectors computes $w_i = \alpha x_i + \beta y_i$ for each set of entries. Note that the only data iteration between rows in either of these operations is the sum in the dot product, however addition is an associative operation. Thus, both of these functions can be arbitrarily parallelized or have their iteration reordered.

The matrix vector product iterates once over the rows and for each row sums the nonzero entries times the vectors corresponding entries [2]. Both the rows

and the sum in each row may be iterated in any order or in parallel. Thus, the matrix information can be compressed for any iteration order of rows and any iteration order for the values in each row. However, the vector information must be able to be read at an arbitrary index. For each iteration, the matrix information is read only once and the vector entries are read for each nonzero value in the corresponding column (8 to 27 times for the matrix described in Section 2.3). So, assuming the problem is too large for the matrix to fit entirely in the memory caches, the matrix data will always need to be read from main memory, while vector data will be able to utilize caches, resulting in up to 27 fold fewer reads than the matrix data. This hints that the compressing matrix information is more likely to provide an increase in performance of the matrix vector product than compressing the vector information.

The symmetric Gauss-Seidel step is similar to the sparse matrix-vector product, with added complications. First, the step has two iterations, one forward and one backward. Instead of simply summing the row-vector product, each row does the following calculation

$$x_i \leftarrow b_i - \frac{1}{a_{ii}} \sum_{j=1}^{n} a_{ij} x_j$$

with the terms containing nonzero matrix entries removed [2]. Note that each x_i is used immediately in the subsequent rows, this means that any deviation from the base row iteration order or any parallelization of the rows may reduce the effectiveness of the step. Because any delay in writing the new values to $\vec{x}$ results in effectively parallelizing the iteration of the rows, the vector values must be written immediately or within a few iterations. Additionally, the Gauss-Seidel step has the additional requirement that the matrix diagonal of the current row must be accessible.

The restriction and prolongation functions used in the multigrid are the last matrix and vector value accessing functions used in the Conjugate Gradient implementation. Restriction samples points from two fine grid vectors and stores the difference in a coarse grid vector. Prolongation takes a the entries in a coarse grid vector and adds them to select fine grid vectors. So, between these two functions, random read and write access is needed by vectors in all but the coarsest mesh.

2.5 Compression Strategies

Numerous compression strategies were considered for this project. Figure 3 lists the compressions tried for each main data structure. Note that most compression methods were only used with one or two of the data types, even if able to be reasonably used within the constraints of additional data.

2.5.1 Restrictions on Compression Strategies

The restrictions on usable compression strategies primarily come from the data access requirements described in Section 2.4. These requirements were that ma-

Strategy	Vector Values	Matrix Values	Matrix Indices
Single Precision	Yes	Yes	Not Able
Mixed Precision	Yes	Not Able	Not Able
1 Bit	Not Able	Yes	Not Able
Squeeze (SZ)	Yes	Yes	Yes
ZFP	Yes	Yes	Not Able
Elias Gamma	Not Able	Not Able	Yes
Elias Delta	Not Able	Not Able	Yes
Huffman	Not Able	No	Yes
Op Code	Not Able	Not Able	Yes

Figure 3: Overview of Compression Strategies

trix rows need to be readable in both a forward and backwards iteration, the diagonal for a given row must be accessible, and the vectors have both random read access and random, immediate write access. Due to the highly regular nature of the particular matrix used and the existence of solvers specially optimized for solving this type of problem, the requirement that all compression techniques are able to handle any sparse matrix was added to increase the usefulness of this work [8]. Although, an exception was made to the requirement to handle general matrices for the 1-bit Compression described in Section 2.5.3 as that compression method is designed to provide an upper bound for improvements from compressing matrix values. Finally, integer compression was limited to lossless compression methods to ensure that the proper vector entries were being acted on, while floating point compression was allowed to be lossy.

Note that some cleverness can be used to work around some restrictions. By compressing the data in small blocks, sequential compression strategies can be used while retaining effectively random access reads and writes [6]. Then, at most, the individual block needs to be decompressed or recompressed for a single read or write. Similarly, a sequential compression method can be used on the matrix information by compressing the data twice, once for forward iteration and once for backwards iteration.

2.5.2 Single and Mixed Precision Floating Point Numbers

The most obvious compression of floating point data is using single precision representation instead of double precision representation. While it only has a compression rate of 1:2, it allows the compression and decompression of values using at most 1 extra hardware operation. Additionally, it provides the same data access properties as the double prevision version. For the matrix values, single precision representation is lossless in the test problem, since each matrix value is an integer. However, for the vector values, using single precision floats resulted in a large increase of Conjugate Gradient iterations due to the loss of precision. So, by making only select vectors single precision, a compromise can be found where vectors that need high precision can keep that precision and vectors that do not need as much precision can get improved performance.

2.5.3 1 bit Compression

To provide a estimated upper bound on improvements in performance from matrix value compression, 1 bit compression was devised. This scheme uses the fact that the matrix values in the test matrix are all either -1 or 26. Note that as implemented, this scheme has very limited number of matrices that can be compressed with in. However, certain compression schemes that modify the compression based on the data being compressed, such as Huffman coding described in Section 2.5.8, can achieve the same compression for the test matrix. Note that the upper bound provided for 1 bit compression is only an upper bound for the particular pair of vector and index compressions that 1 bit compression was used with. The importance of compressing multiple structures, as described in Section 2.5.9, is shown using 1 bit compression.

2.5.4 Squeeze (SZ) Compression

Squeeze (SZ) compression is a group of compression strategies based on using curve fitting and can be used for both integers and floating point values. The compression strategy referred to as SZ compression in this paper deviates from the original description by using a generalization of the core approach of the original implementation of SZ compression [1]. SZ compression allows for string bounds to be placed on the compression error.

The compressed data is stored in two arrays, one storing the curve each value is compressed with and the other storing values that could not be fit by any curve. To compress each value, the error between the prediction made by each curve is compared. If the smallest error is within the user supplied tolerance, the associated curve is stored. Otherwise, the value is appended to the list of uncompressed values and the curve is stored as uncompressed. Because only the compressed value is available at decompression time, those values are used during compression time to compute the value produced by each curve. This allow error requirements to be meet. The compression rate is

$$\frac{ps + \lceil \log_2(n) \rceil}{s}$$

where s be the number of bits used by an uncompressed value, p be the percent of values that are compressed and n be the number of curves available.

The curves available are selected based on the nature of the data being compressed. Note that the word curve is used loosely here to refer to any predictive function. Figure 4 shows all of the curve fitting function that were used. For compressing vector values, the Neighbor, Linear and Quadratic curves were used. Because the vector values represent a value at each grid point, these curves attempted to capture smooth changes and relations in the data. The matrix indices were compressed using only the increment compression mode, since approximately two thirds of the indices fit that pattern. The matrix values were compressed with a few different combinations of curves. These combinations were Neighbor alone, Neighbor and Neighbor's Neighbor, and Neighbor, Neigh-

Uncompressed	$v_i \leftarrow$ original ith value
Neighbor	$v_i \leftarrow v_{i-1}$
Linear	$v_i \leftarrow 2v_{i-1} - v_{i-2}$
Quadratic	$v_i \leftarrow 3v_{i-1} - 3v_{i-2} + v_{i-3}$
Neighbor's Neighbor	$v_i \leftarrow v_{i-2}$
Last Uncompressed	$v_i \leftarrow$ last uncompressed value stored
Increment	$v_i \leftarrow v_{i-1} + 1$

Figure 4: Curve Prediction Functions Used

bor's Neighbor and Last Uncompressed. These curves were chosen to find the best way to compress a series of -1's with occasional 26's.

2.5.5 ZFP Compression

ZFP compression is a lossy floating point compression scheme designed for spacial correlated data [6]. ZFP compression is designed to take advantage of spacial relations for data up to 4 dimensions. Note that the matrix values were compressed with ZFP, in spite of the fact that there is no spacial relation between points. Because the vectors represent points in 3 dimensions, 1 and 3 dimension compression was tried. The matrix values were only compressed with 1 dimension. ZFP compresses its values by grouping the data into blocks of 4^d elements, where d is the number of dimensions compressing with [6]. When random access is required, each block is compressed at a fixed size to allow access to arbitrary blocks. ZFP was implemented using the existing C++ library. Both the high- and low-level interfaces were tried for the vector compression.

2.5.6 Elias Gamma Coding and Delta Coding

Elias Gamma and Delta codings are a pair of similar compression methods that are designed to compress positive integers by not storing extra leading 0's [3]. Because these schemes are able to better compress smaller numbers, the matrix indices were stored as the offset from the preceding value. Then, because these codings are only able to compress positive integers, the indices of each row must be sorted in acceding order. Finally, the first index in each row is stored as the offset from -1, to ensure an index of 0 is properly encoded.

To encode an integer n with Gamma coding, let $N = \lfloor \log_2(n) \rfloor + 1$ be the number of bits needed to store n. Then, n is represented by $N-1$ zeros followed by the N bits of n [3]. Thus, n can be stored with only $2N - 1$ bits. For small values of N this is highly effected, reaching compression ratios of up to 1:32. See Figure 5 for examples of gamma coding.

Delta coding is similar to Gamma coding, except instead of preceding the number with $N-1$ 0's, the number is preceded by $gamma(N)$ and only the last $N - 1$ bits are stored. So, n can be stored with only $N + 2\lfloor \log_2(N) \rfloor$ bits. Figure 6 contains examples of delta coding. Note that delta coding provides better

	Compression Rate
$\text{gamma}(1) = 1_2$	1:32
$\text{gamma}(2) = 0\,10_2$	3:32
$\text{gamma}(3) = 0\,11_2$	3:32
$\text{gamma}(4) = 00\,100_2$	5:32
$\text{gamma}(5) = 00\,101_2$	5:32
$\text{gamma}(6) = 00\,110_2$	5:32
$\text{gamma}(7) = 00\,111_2$	5:32
$\text{gamma}(8) = 000\,1000_2$	7:32
$\text{gamma}(64) = 000000\,1000000_2$	13:32
$\text{gamma}(256) = 00000000\,100000000_2$	17:32
$\text{gamma}(1024) = 0000000000\,10000000000_2$	21:32

Figure 5: Select Examples of Elias Gamma Coding

compression for large numbers, but worse compression for certain smaller numbers. Additionally, because decoding a delta encoded value requires decoding a gamma encoded value, decoding a delta coded value is more expensive than decoding a gamma coded value.

2.5.7 Op-Code Compression

Opcode compression is based on the index compression used in Compressed Column Index (CCI) matrices [5]. Note that this integer compression is never given it's own name in the original description and so is referred to as opcode compression in this paper. Opcode compression is inspired by CPU instruction encodings which are seperated into an "opcode" portion and a data portion (hence the name). To read each value, the first few bits are read to determine the number of bits used for the data portion, which stores the encoded value. Like Gamma and Delta coding, opcode compression reduces the number of leading 0's stored, and similarly is utilized by encoding the difference from the preceding index. If some opcodes are used significantly, that opcode can be shorted to save bits. This shortened opcode can be handled in a lookup table by placing the opcode's information at every location that begins with the opcode. For example, if 0, 10 and 11 are the possible opcodes, then the information for opcode 0 is located at the indices of 00 and 01.

The description of CCI matrix format uses a fixed decode table. However, when using a lookup table, it is possible to use custom decode tables to adjust the compression for the specific matrix's sparsity pattern. Table 1 shows the opcodes used for CCI format.

	Compression Rate
$\text{delta}(1) = 1_2$	1:32
$\text{delta}(2) = 010\,0_2$	4:32
$\text{delta}(3) = 010\,1_2$	4:32
$\text{delta}(4) = 011\,00_2$	5:32
$\text{delta}(5) = 011\,01_2$	5:32
$\text{delta}(6) = 011\,10_2$	5:32
$\text{delta}(7) = 011\,11_2$	5:32
$\text{delta}(8) = 00100\,000_2$	8:32
$\text{delta}(64) = 00111\,000000_2$	11:32
$\text{delta}(256) = 0001001\,00000000_2$	15:32
$\text{delta}(1024) = 0001011\,0000000000_2$	17:32

Figure 6: Select Examples of Elias Delta Coding

Opcode	Length
0	4 bits
100	5 bits
110	15 bits
101	20 bits
111	26 bits

Table 1: CCI Format Opcodes

2.5.8 Huffman Coding

2.5.9 Combined Compression Strategies

3 Test Results

4 Conclusions and Future Work

5 References

[1] S. Di and F. Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 IEEE International Parallel and Distributed Processing Symposium*, pages 730–739, May 2016.

[2] Jack Dongarra, Michael Heroux, and Piotr Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems. Technical Report UT-EECS-15-736, Electrical Engineering and Computer Sciente Department, Knoxville, Tennessee, November 2015.

[3] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[4] David R. Kincaid and E. Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Pure and applied undergraduate texts. American Mathematical Society, 2002.

[5] Orion Sky Lawlor. In-memory data compression for sparse matrices. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[6] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, Dec 2014.

[7] J. Nearing. *Mathematical Tools for Physics*. Dover books on mathematics. Dover Publications, 2010.

[8] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[9] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.