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

Solving large, sparse linear systems of equations plays an important role in certain scientific computations. For example, the finite element method uses the solution to a system of linear equations to solve partial differential equations [16]. These problems can be large, with possibly millions of variables [4]. So, solving these problems efficiently requires a fast linear solver.

The particular class of linear systems being looked at are sparse, meaning that they have a high proportion of zero coefficients. Iterative solvers are often used to solve these large, sparse systems. These solvers take an initial guess then improve it until it is within some tolerance [16]. On modern computers, these solvers often spend significantly most of their time fetching data from main memory to the processor where the actual computation is done [13]. This work tries to improve the performance of these types of solvers by compressing the data to reduce the time spent accessing main memory.

1.1 Previous Work

Much work has been done on various aspects of utilizing single precision floating point numbers, while retaining the accuracy of double precision numbers. One such approach uses the fact that iterative solvers can take an initial guess of the solution to jump start progress. So, many iterations can be done at the cheaper single precision, then double precision iterations refine the solution to sufficient accuracy [1, 3]. Another main approach is to applying the preconditioner using single precision, while otherwise using double precision, which result in similar accuracy unless the matrix is poorly conditioned [2, 9].

Another effort at compressing large, sparse Linear Systems is Compressed Column Index (CCI) format to store matrices [13]. This format is based of Compressed Sparse Row (CSR) matrix format except uses a compressed representation of the column indices. This project generalizes CCI matrix format by trying compression with additional data structures and by trying additional compression schemes.

2 Background

2.1 Conjugate Gradient

Conjugate Gradient is the iterative solver used by HPCG [6]. 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 [16]. 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 [17].

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

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

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 [15]. 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 $\mathbf{A}\vec{x} = \vec{b}$ [17].

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 [15]. 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 $\vec{r}_i = -\nabla f(\vec{x}_i) = \vec{b} - \mathbf{A}\vec{x}_i$ 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 $f(\vec{x}_{i+1})$. Thus, choose α_i such that

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

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

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

Algorithm 1 Steepest Decent [17].

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
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

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. [17]. 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 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 [17]. This is done using the Conjugate Gram-Schmidt Process. To do this, each search direction, $\vec{d}_i$ is computed

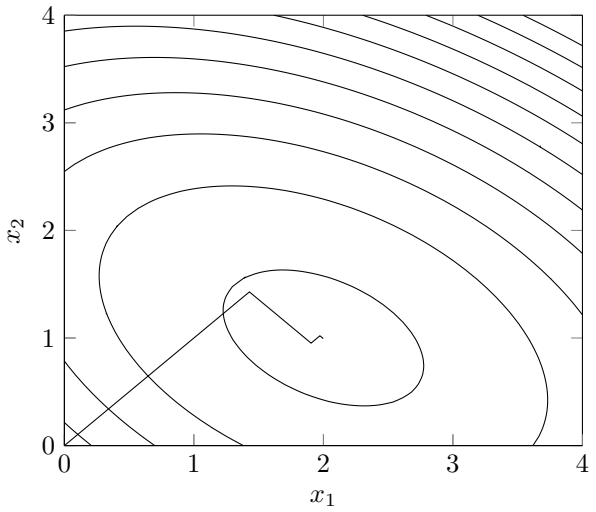


Figure 1: Contour graph of the quadratic function and the first six values of $\vec{x}$ produced by steepest decent for Example 1.

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 all previous search directions, except $\vec{d}_i$ [17]. 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

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

Algorithm 2 Conjugate Gradient [16].

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
$\vec{d}_0 \leftarrow \vec{r}_0$
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

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 [16]. 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 the coefficient matrix and the right hand side [16]. 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 [6]. 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 [16].

The multigrid method uses meshes of different sizes to improve performance of a relaxation style iterative solver [16]. Most relaxation type iterative solvers are able to quickly reduce the components of the residual in the direction of

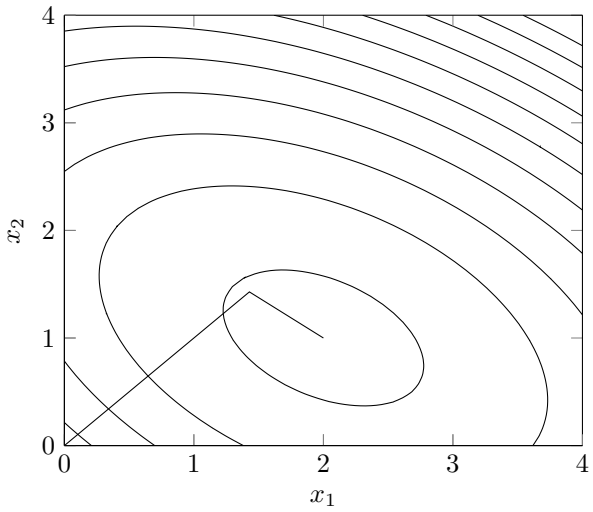


Figure 2: Contour graph of the quadratic function and the each value of $\vec{x}$ produced by Conjugate Gradient for Example 2.

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 [16]. 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 [6]. 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 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.

Algorithm 3 Preconditioned Conjugate Gradient [16].

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
$\vec{z}_0 \leftarrow \mathbf{M}^{-1}\vec{r}_0$
$\vec{d}_0 \leftarrow \vec{z}_0$
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

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 [6]. 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 [12]. The solution vector consists of all 1's.

HPCG uses an implementation of Conjugate Gradient algorithm with a multigrid preconditioner variant [6]. 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 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 [6]. 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.

In addition to the restrictions on usable compression schemes imposed by the data access patterns, they have an effect on the effectiveness of compression schemes. Note that in the inner loop of both the sparse matrix vector product and the symmetric Gauss-Seidel step do not have a data dependency between matrix values and the vector values; however the vector values are dependent on the matrix indices. So, the matrix values can be fetched in parallel to the matrix indices and vector values [8]. This will result in ineffective compression when just compressing one part of the problem, as discussed in Section 2.5.9.

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 [6]. 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 [6]. 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 other types of 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 matrix 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

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.

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 [16]. 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 [14]. 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 [5]. 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. Note that due to the granularity of the matrix values and indices, bounding the error to be less than one results in an effective error bound of 0. Thus, when compressing those data structure, only an error bound of 0 is used.

The curves available are selected based on the nature of the data being compressed. Note that the term "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

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.

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, Neighbor'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 [14]. 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 [14]. 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 [7]. 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 [7]. 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

	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.

$N-1$ 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 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 [13]. 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,

	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 [13].

when using a lookup table, using custom decode tables to adjust the compression for the specific matrix's sparsity pattern will not have a significant performance penalty to decoding. Table 1 shows the opcodes used for CCI format.

2.5.8 Huffman Coding

Huffman coding is an optimal prefix code usable for lossless compression [10]. A prefix code is a coding where each representable value is assigned a unique coding such that no code is the beginning of another code. However, Huffman coding does not take advantage of local patterns in the data, just the overall frequencies of each value. Additionally, Huffman coding can only be decoded sequentially, due to the variable length of storage for each value. So, while it can compress matrix values and indices, it is unable to meet the requirements to compress vector values. Note that the Huffman coding of the matrix values in the test problem is equivalent to the 1-bit coding described in Section 2.5.3. Thus, only matrix indices were tested with Huffman coding.

2.5.9 Combined Compression Strategies

In addition to compressing a single data structure at a time, compression strategies which compress multiple data structures were tried. This provided the opportunity to achieve an overall reduction in data that could not be achieved by compressing a single data structure. Additionally, as discussed in Section 2.4 and as shown in Section 4.1, compressing matrix values alone cannot provide performance improvement.

3 Performance Models

To help understand the requirements to seen an improvement in overall performance, models were constructed to estimate the amount of time spent fetching and decoding information. Two models were constructed, an analytical model that did not take into account processor level parallelism and a simulation based model one that took into account certain processor level parallelism. Both models are based on the sparse matrix-vector product, but, due to the similarity of the data access for the symmetric Gauss-Seidel step, should provide an estimate on both kernals. Additionally, the models only provide estimates for rows with 27 elements, because there are $O(n^3)$ of those rows and only $O(n^2)$ of other rows where the matrix has $O(n^3)$ rows. Finally, the models assume that the compression does not reduce the rate of convergence. The models were analyzed using the memory access latencies of the head node of the testing cluster. These latencies are shown in Table 2. The models were primarily used to find the minimum compression performance to outperform the baseline implementation. The following variables represent the relevant compression characteristics in this section

$$\begin{aligned}
\text{vectDecode} &= \text{time to decode one vector value} \\
\text{vectEncode} &= \text{time to encode one vector value} \\
\text{vectBytes} &= \text{the number of bytes per vector value} \\
\text{matIndDecode} &= \text{time to decode one matrix index} \\
\text{matIndBytes} &= \text{the number of bytes per matrix index} \\
\text{matValDecode} &= \text{time to decode one matrix value} \\
\text{matValBytes} &= \text{the number of bytes per matrix value}
\end{aligned}$$

3.1 Analytical Model

The analytical model is a set of equations that computes the amount of time spent serially fetching and decoding the information. This model is implemented

L1 Cache Latency	4-5 cycles
L2 Cache Latency	12 cycles
L3 Cache Latency	38 cycles
Main Memory Latency	38 cycles + 58 ns
Clock Rate	2.2GHz

Table 2: Estimate Cluster Performance [11].

using the following system of equations

$$
\begin{aligned}
&27 \cdot \text{vectDecode} + \text{vectEncode} + 18 \cdot \text{L1Time} \\
&+ \left(\frac{64 - \text{vectBytes}}{64} \cdot 9 \cdot \text{L1Time} + \frac{\text{vectBytes}}{64} \cdot (6 \cdot \text{L2Time} + 3 \cdot \text{RAMTime}) \right) \\
&+27 \cdot \text{matIndDecode} \\
&+27 \cdot \left(\frac{\text{matIndBytes}}{64} \cdot \text{RAMTime} + \frac{64 - \text{matIndBytes}}{64} \cdot \text{L1Time} \right) \\
&+27 \cdot \text{matValDecode} \\
&+27 \cdot \left(\frac{\text{matValBytes}}{64} \cdot \text{RAMTime} + \frac{64 - \text{matValBytes}}{64} \cdot \text{L1Time} \right)
\end{aligned}
$$

where

$$
\begin{aligned}
\text{L1Time} &= \text{the access latency for L1 cache} \\
\text{L2Time} &= \text{the access latency for L2 cache} \\
\text{RAMTime} &= \text{the access latency for main memory.}
\end{aligned}
$$

This model utilizes a few facts. Firstly, the number of bytes per value divided by 64 provides the percent of values that will require fetching a cacheline from main memory or higher caches, while 1 minus this values is the percent of values that will be able to only need to access L1 cache. Secondly, due the matrix sparsity pattern, two thirds of vector values will always be in L1 cache and two thirds of the remaining values will be in L2 cache. The model was only studied using the performance characteristics shown in Table 2. Outperforming the baseline

implementation in this model requires meeting the approximate bounds

$$\begin{aligned} \text{decode} &= \text{vectDecode} + \text{matValDecode} + \text{matIndDecode} \\ \text{matBytes} &= \text{matValBytes} + \text{matIndBytes} \end{aligned}$$

$$\begin{aligned} \text{vectEncode} &< 878.513 \\ \text{decode} &< 32.5375 - 0.037037 \cdot \text{vectEncode} \\ \text{matBytes} &< 12.9664 - 0.398506 \cdot \text{decode} - 0.0147595 \cdot \text{vectEncode} \\ \text{vectBytes} &< 107.34 - 3.29897 \cdot \text{decode} - 0.122184 \cdot \text{vectEncode} \\ &\quad - 8.27835 \cdot \text{matBytes} \end{aligned}$$

where all encode and decode times are in clocks. Note that the upper bound on the number of bytes per vector value is reduced by 3.29897 per 1 clock increase in decode time. This indicates that only highly effective compression techniques will be effective for vector values. Matrix compression on the other hand, appears to be able to achieve a performance improvement with a slower decompression than required by the vector values.

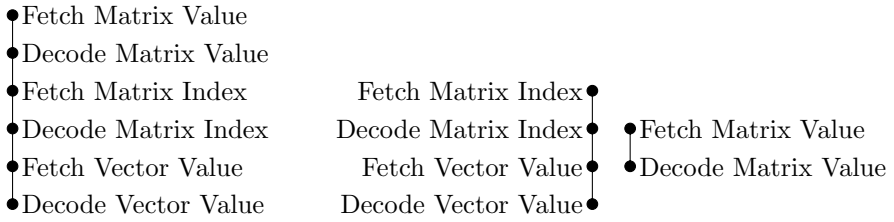
3.2 Simulation Based Model

The parallel model attempts to estimate the performance required to outperform the baseline implementation while taking into account processor level parallelism. Figure 7 shows the dependencies used by each model. Note that this model assumes that the compiler and processor can fully parallelize any operations without data dependencies, while, at the absolute minimum, the instructions must be read serially [8]. Additionally, the model assumes that the bytes for each compressed value is constant, which is not true for most matrix compression and for some vector compression. Lastly, the model was only worked with using integral values for the bytes and decode/encode times. The model then takes the same compression properties as the first model and computes the time to fetch and decode the values and encode the result value over 10 matrix rows with 27 entries per row. Appendix A contains source code for this model.

The bounds on outperforming the baseline implementation were hard to determine for the parallel model due to the nature of the model as a sum of maximizations. However, some boundaries were computed. When only compressing vectors, the compression needs to perform such that

$$8 \geq \text{vectBytes} + \tfrac{7}{6} \cdot \text{vectDecode} + \tfrac{2}{3} \cdot \text{vectEncode}.$$

This model allows for a slightly less efficient decoding step than the analytical model, however encoding vectors must be more efficient for this model. The bounds on matrix compression are less intuitive. Table 3 shows the maximum decode times for matrix indices and values respectively. These compression bounds appear to be significantly related to the frequency at which multiple values are fetched from main memory.



(a) Serial Model. (b) Parallel Model.

Figure 7: Comparison of the data dependency graphs used by each model, where lower nodes are dependent on higher nodes.

Bytes	Max Index Decode Time	Max Value Decode Time
1	61	13
2	46	6
3	61	12
4	0	0
5	-	12
6	-	5
7	-	12
8	-	0

Table 3: Maximum Decode Times for Just Compressing Matrix Indices or Values.

Compression	GFLOP Rating	Iterations	Compression Rate
Baseline	15.3654	50	1:1
Single Precision	7.023 01	115	1:2
Mixed Precision			
$\vec{d}$	5.184 76	150	11:12
$\vec{b}, \vec{x}$	15.3701	50	5:6
$\vec{b}, \vec{x}, \mathbf{A}\vec{d}$	15.0428	51	3:4
$\vec{b}, \vec{x}, \vec{d}$	5.208 32	150	3:4
$\vec{b}, \vec{x}, \vec{d}, \mathbf{A}\vec{d}$	5.232 08	150	2:3
$\vec{b}, \vec{x}, \vec{d}, \mathbf{A}\vec{d}, \vec{z}$	6.958 94	115	7:12
$\vec{b}, \vec{x}, \vec{d}, \vec{z}$	6.913 43	115	2:3
$\vec{b}, \vec{x}, \vec{z}$	12.2146	64	3:4
ZFP (High Level API)			
1d			
16 bits/value	0.690 138	51	1:4
32 bits/value	0.393 97	50	1:2
3d			
16 bits/value	2.720 78	51	1:4
32 bits/value	2.176 39	50	1:2
SZ			
7 values/block	5.891 38	57	8:7
8 values/block	5.787 11	57	1:1 to 2:1
12 values/block	4.985 36	57	2:3 to 4:3
15 values/block	4.515 94	57	8:15 to 16:15
16 values/block	4.535 36	57	1:2 to 3:2
24 values/block	3.677 48	57	1:3 to 4:3
32 values/block	3.1735	57	1:4 to 5:4

Table 4: Results of Compressing Vector Values.

4 Test Results

Tables 4, 5 and 6 show the compression results for compressing just the vector values, matrix values and matrix indices respectively. These tables contain the rating measured by HPCG, the number of iterations needed for convergence and the compression rate based on the number of cache lines fetched, which may be different then the memory allocated. Note that some compression strategies had multiple variations that were tested. Section 4.2 contains information on variations used in vector compression. The compression of just one data structure fails to outperform the baseline implementation; Section 4.1 discusses this further.

Next, combined compression schemes were tried, using SZ and single precision compression for the matrix values and using SZ, gamma and delta compression for the matrix indices. Table 7 shows the results of these combined

Compression	GFLOPs	Iterations	Compression Rate
Baseline	15.3654	50	1:1
Single Precision	12.6331	50	1:2
1-bit	15.1743	50	1:64
SZ			
1 mode	13.5037	50	
2 modes	13.8195	50	
ZFP			
High Level API	0.817 469	50	
Low Level API	0.960 338	53	

Table 5: Results of Compressing Matrix Values.

Compression	GFLOPs	Iterations	Compression Rate
Baseline	15.3654	50	1:1
SZ	14.9322	50	
Gamma	14.6553	50	
Delta	14.3036	51	
Huffman			
Uncompressed First Index			
4 bit window	10.654	51	
6 bit window	10.7666	51	
8 bit window	10.8941	51	
10 bit window	10.8156		
12 bit window	10.8158	51	
Compressed First Index		51	
4 bit window	10.9359	51	
8 bit window	11.1134	51	
16 bit window	10.3323	51	
Op Code			

Table 6: Results of Compressing Matrix Indices.

Compression			
Value	Index	GFLOPs	Iterations
Baseline		15.3654	50
SZ	SZ	18.9702	50
SZ	Gamma	13.661	51
SZ	Delta	10.9903	50
32 bit	SZ	14.1796	51
32 bit	Gamma	17.6676	51
32 bit	Delta	12.56	51

Table 7: Results of Combined Matrix Value and Index Compression Schemes.

Compression				
32 bit Vectors	Value	Index	GFLOPs	Iterations
Baseline			15.3654	50
None	SZ	SZ	18.9702	50
$\vec{b}, \vec{x}$	SZ	SZ	23.967	50
$\vec{b}, \vec{x}, \mathbf{A}\vec{d}$	SZ	SZ	27.5974	50
None	32 bit	Gamma	17.6676	51
$\vec{b}, \vec{x}$	32 bit	Gamma	16.6048	50
$\vec{b}, \vec{x}, \mathbf{A}\vec{d}$	32 bit	Gamma	16.5665	50

Table 8: Results of Combined Vector, Matrix Value and Matrix Index Compression Schemes.

schemes. Like the single compression tables, this table contains the GFLOP rating measure by HPCG and the number of iterations needed for convergence. The combined scheme with the best performance used SZ compression for both values and indices. The only other approach that outperformed the baseline implementation used 32 bit compression for the values and gamma compression for the indices.

Finally, vector compression was combined with the successful combined matrix compression. Due to the poor performance of SZ and ZFP vector compression, only the better versions of mixed precision vector compression were used. Table 8 shows the results for these compression strategies. The first column indicates which vectors were stored in 32bit; the rest of the columns correspond to their counter parts in Table 7. Note that vector compression improved the performance of the SZ compressed matrices, but reduced the performance of the gamma compressed matrices. So, the best implementation for the test problem uses mixed precision vectors with $\vec{b}, \vec{x}$ and $\mathbf{A}\vec{d}$ stored in single precision, and sz compressed matrix values and indices.

4.1 Performance Improvement Bounds

Note that Table 5 shows 1 bit compression under performing the baseline implementation, even though it has a significant compression rate. This demonstrates that compressing the matrix values alone in unable to improve performance. For the vector values, note that the single precision implementation has a 2.3 times increase in iterations to convergence over the baseline implementation and that the GFLOPs rating of the single precision implementation is reduced by a factor of approximately 2.19 from the baseline implementation. This hints that, even without increasing the number of Conjugate Gradient iterations, compressing the vectors requires a compression rate better than 1:2 to provide much of an improvement in performance. This analysis is supported by the fact that none of the compression strategies tried that only compressed a single strategy where able to out perform the baseline implementation.

4.2 Vector Compression

As shown in Table 4, vector compression was not successfully used to improve performance. Section 4.1 discusses why performance improvement is likely limited. However, vector compression is able to make improvements when combined with other compressions, as shown in Table 8.

ZFP had poor performance when compressing vector information. Note that 1 dimensional ZFP compression has a 16 bit granularity, and 3 dimensional ZFP compression has a 1 bit granularity [14]. These granularity restrictions and the resulting iterations needed were used to select the tested compression rates.

SZ compression has two main configurable settings, the number of values in each block and the error bound. There were two measures of error that were considered, absolute error and pointwise relative error. The performance was tested with both a single error being bounded and both errors being bounded. Absolute error is the absolute value of the difference between predicted and actual. The pointwise relative error is the absolute error divided by the actual value. Table 4 contains results for various block sizes with both an absolute error bound of 10^{-10} and a pointwise relative error bound of 10^{-10}. Table 9 contains an comparison of various error bounds for a block size of 8 values per block. Note that an absolute bound of 10^{-2} was unable to converge within 500 iterations.

4.3 Compiler Settings Analysis

4.4 Consistency of GFLOP Ratings

4.5 Testing Environment

Timings measured with a problem of size 96^3 with 60 processes on the walbert cluster.

Error Bound	GFLOP Rating	Iterations
10^{-2} relative	3.668 59	69
10^{-6} relative	5.7806	57
10^{-10} relative	5.787 11	57
10^{-14} relative	5.813 57	57
10^{-18} relative	5.732 77	57
10^{-2} absolute	NA	≥ 500
10^{-6} absolute	4.518 27	57
10^{-10} absolute	5.140 58	57
10^{-14} absolute	5.643 38	57
10^{-18} absolute	5.816 42	57
10^{-2} absolute and 10^{-10} relative	5.755 38	57
10^{-10} absolute and 10^{-2} relative	5.225 92	57
10^{-10} absolute and 10^{-10} relative	5.825 27	57

Table 9: Results of Compressing Vector Values with SZ Compression using Various Error Bounds.

5 Conclusions and Future Work

6 References

[1] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *CoRR*, abs/0808.2794, 2008.

[2] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4):17:1–17:22, July 2008.

[3] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.*, 21(4):457–466, November 2007.

[4] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[5] 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.

[6] Jack Dongarra, Michael Heroux, and Piotr Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems. Techni-

cal Report UT-EECS-15-736, Electrical Engineering and Computer Sciente Department, Knoxville, Tennessee, November 2015.

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

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, sixth edition, 2017.

[9] J. D. Hogg and J. A. Scott. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 37(2):17:1–17:24, April 2010.

[10] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[11] Intel bradwell. `https://www.7-cpu.com/cpu/Broadwell.html`. Accessed: 2018-11-06.

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

[13] 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.

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

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

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

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

A Simulation Performance Model Source Code

Below is the implementation of the simulation performance model. The model was implemented using Common Lisp, being used with Steel Bank Common Lisp version 1.4.

```
;;; Global Variables

; Cluster properties
(defparameter *l1-time* 5)
(defparameter *l2-time* 12)
(defparameter *main-mem-time* 1656/10)

; Model parameters
(defparameter *rows-to-check* 10)

; Default compression settings
(defparameter *bytes-per-mat-ind* 4)
(defparameter *bytes-per-mat-val* 8)
(defparameter *bytes-per-vect* 8)
(defparameter *inds-decode-time* 0)
(defparameter *vals-decode-time* 0)
(defparameter *vect-decode-time* 0)
(defparameter *vect-encode-time* 0)


;;; Model Implementation

(defmethod fetch ((obj (eql :mat-inds)) (i integer))
  "Computes the cost of fetching the ith matrix index"
  (if (/= (floor (* (1- i) *bytes-per-mat-ind*) 64)
          (floor (* i *bytes-per-mat-ind*) 64))
    *main-mem-time*
    *l1-time*))
(defmethod fetch ((obj (eql :mat-vals)) (i integer))
  "Computes the cost of fetching the ith matrix value"
  (if (/= (floor (* (1- i) *bytes-per-mat-val*) 64)
          (floor (* i *bytes-per-mat-val*) 64))
    *main-mem-time*
    *l1-time*))
(defmethod fetch ((obj (eql :vect)) (i integer))
  "Computes the code of fetching the ith vector value"
  (cond
    ((> (mod i 3) 0) *l1-time*)
    ((< (/ i 3) 6) *l2-time*)
    (t (+ (* (/ *bytes-per-vect* 64) *main-mem-time*)
          *l1-time*
          (* (/ *bytes-per-vect* -64) *l1-time*)))))

(defun 1-row ()
  "Computings the average cost to load a row.
   *rows-to-check* provides the number of rows to use"
```

```
47    (+
48      (/ (loop :for i :from 0 :below (* *rows-to-check* 27)
49               :for inds-fetch-time = (fetch :mat-inds i)
50               :for vals-fetch-time = (fetch :mat-vals i)
51               :for vect-fetch-time = (fetch :vect i)
52               :for vals-time = (+ vals-fetch-time
53                                   *vals-decode-time*)
54               :for vect-time = (+ inds-fetch-time
55                                   *inds-decode-time*
56                                   vect-fetch-time
57                                   *vect-decode-time*)
58            :summing (min vals-time vect-time))
59         *rows-to-check*)
60      *vect-encode-time*))
61
62  (defun 1-row-with-props (ind-size val-size vect-size
63                           ind-decode val-decode
64                           vect-decode vect-encode)
65    "Like 1-row, but sets the compression properties"
66    (let ((*bytes-per-mat-ind* ind-size)
67          (*bytes-per-mat-val* val-size)
68          (*bytes-per-vect* vect-size)
69          (*inds-decode-time* ind-decode)
70          (*vals-decode-time* val-decode)
71          (*vect-decode-time* vect-decode)
72          (*vect-encode-time* vect-encode))
73      (1-row)))
```