Enhancing Sparse Matrix Operation Efficiency with CSR in Machine Learning

Aneesh Durai
University of California, Berkeley
Berkeley, CA, USA
aneesh.durai@berkeley.edu

Abstract

Efficient data handling is widely used in machine learning especially with the increasing prevalence of large and sparse datasets. In this paper, we explore improving sparse matrix operation efficiency using the Compressed Sparse Row (CSR) format. Focusing on Linear Regression techniques and Singular Value Decomposition (SVD), we investigate methods to optimize these operations for sparse matrices. In the process, we explore an application of the Lanczos algorithm in efficiently computing a low–rank approximation of SVD for sparse matrices. Our approach demonstrates notable improvements and thus can contribute to a wide array of practices done in larger scales by data engineers.

1 Introduction

In larger scale machine learning landscapes, the ability to efficiently process large volumes of data is a challenge. Sparse matrices, which are widely seen across a variety of machine learning applications, present serious computational expense due to their high volume of zero values. Efficiently manipulating these matrices is critical in the performance and scalability of machine learning algorithms. This project analyzes optimizing sparse matrix operations, with a particular focus on the Compressed Sparse Row (CSR) format, a popular method of representing sparse matrices in computational tasks. To analyze the performance, we investigate three fundamental operations central in data science and machine learning: Ordinary Least Squares, Gradient Descent, and Singular Value Decomposition (SVD).

2 Literature Analysis of Related Work

The paper *Iterative Methods for Sparse Linear Systems* by Yousef Saad explores a wide range of sparse matrix representations, specifically the Compressed Sparse Row (CSR) format, and emphasizes its computational efficiency and memory [1]. It also discusses various iterative methods for solving linear systems which are suited to sparse matrices.

The paper *Implementing Sparse Matrix-Vector Multiplication Throughput-Oriented Processors* by Nathan Bell and Michael Garland of NVIDIA explores the optimization of sparse matrix-vector multiplication (SpMV) which highlights the challenges of implementing SpMV in parallel computing architectures, including GPUs [2]. Their work provides valuable insights into how to leverage modern hardware for efficient computation of SpMV, a task that is inherently challenging due to the irregular memory access patterns of sparse matrices.

The textbook *Matrix Computations* by Gene Golub and Charles Van Loan analyzes Lanczos algorithm as well as the range of applications in its application in approximating eigenvalues and eigenvectors of large symmetric matrices [3].

The paper *CUR matrix decompositions for improved data analysis* by Michael Mahooney and Petros Drineas focuses on the intersection of sparse matrix operations and machine learning. They examine the role of sparse matrix techniques such as dimensionality reduction and clustering algorithms [4].

3 Approach

3.1 Ordinary Least Squares in Linear Regression

Linear Regression is a statistical method used to model a relationship between a dependent variable and independent variables. The goal is to generate a line of best fit. In the context of machine learning, we typically extend to multiple linear regression represented as $y = B_0 + B_1 x_1 + B_2 x_2 + ... + B_n x_n + E$. The primary task in linear regression is to estimate the parameters B that minimize the error between the predicted and actual values. In order to accomplish this, we implement Ordinary Least Squares (OLS), which minimizes the sum of the squared difference between the observed and predicted values. Mathematically, it solves the equation $B = (X^T X)^{-1} X^T y$.

3.1.1 Matrix Operation Analysis

In Ordinary Least Squares with the context of linear regression, the matrix operations with primarily occur during the computation of $B = (X^T X)^{-1} X^T y$. First we notice that the term $X^T X$ involves transposing the matrix X and then multiplying it by itself. If X is sparse and stored in CSR format, these operations are going to be performed more efficiently as the multiplications involve fewer computations based on the level of sparsity. The inversion of $X^T X$ can be optimized by again using the sparse result in the inversion process. We once again perform matrix-vector multiplication in the final step $(X^T X)^{-1} X^T y$ to compute the coefficients. The sparse representation again is useful when we multiply the sparse matrix X with y.

3.1.2 Implementation

To evaluate this, we implement Ordinary Least Squares in Python. We then run the algorithm using a randomly generated 1000-by-1000 dimension dense matrix with a fixed level of 90% sparse data along with the same matrix's CSR format. We performed this trial 100 times and compared the differences in execution time between using dense matrix and the CSR matrix. After that, we performed a 250-trial simulation where the sparsity levels varied between 10% and 90% in order for us to analyze the impact of the varying sparsity levels on the performance of OLS.

3.1.3 Results

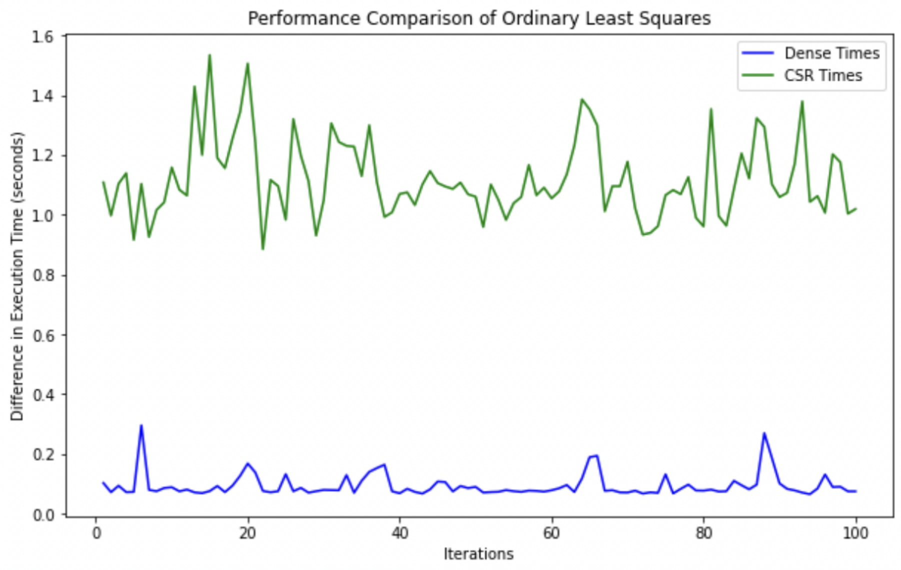

Figure 1: Performance Comparison of Ordinary Least Squares

According to the performance in Figure 1, we can see that despite implementing Ordinary Least Squares using the matrix computation approach, the CSR time does not actually speed up the process. There can be multiple reasons for why this is the case. It is worth noting that there is a matrix inversion involved in Ordinary Least Squares which is a highly expensive task especially if the matrix is sparse. The inverse of a sparse matrix is not necessarily sparse.

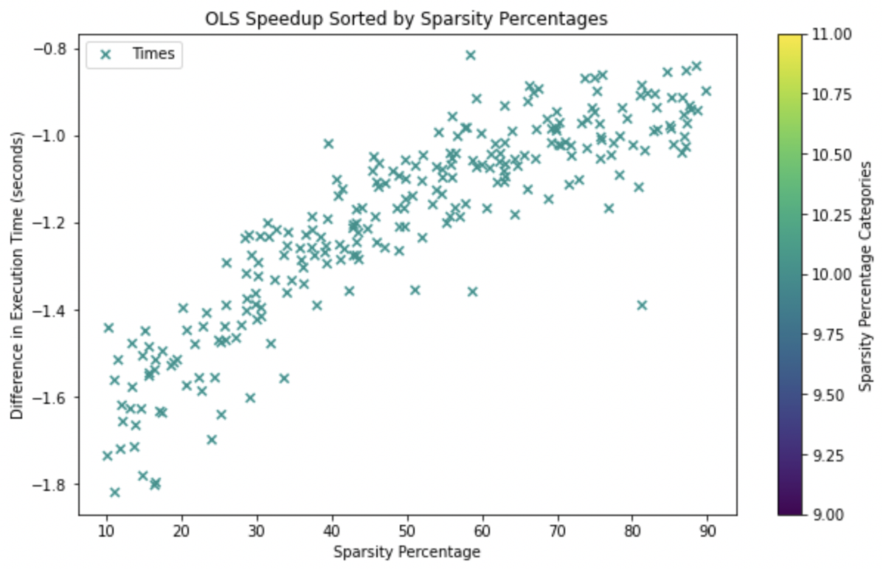

Figure 2: Ordinary Least Squares Algorithm Speedup Sorted by Sparsity Percentages

Figure 2 shows us that less sparse matrices had a way worse execution time compared to that of extremely sparse matrices. There can be multiple reasons why. We still performed matrix multiplication in our implementation, and that probably was more computationally efficient for extremely sparse matrices in CSR than for less sparse ones which is supported by the data in the graph. So we do know that the method does improve depending on how sparse the matrix is, but overall it did not achieve a faster result likely due to overhead and matrix inversion. However, we can look at another algorithm in Linear Regression that could produce a faster outcome

3.2 Gradient Descent in Linear Regression

Gradient Descent is an optimization method used for finding the minimum value of a function. Contrary to Ordinary Least Squares, Gradient Descent is an iterative optimization algorithm used to find the minimum of a function. In the context of linear regression, it is used to find the coefficients of the linear model to minimize a cost function, in our scenario the mean squared error.

3.2.1 Matrix Operation Analysis

In Gradient Descent, each iteration will calculate the gradient and that involves multiplying the matrix X with the coefficient vectors and then with the residuals. If we implement this in CSR format, the multiplication step is expected to be more efficient because we reduce the number of operations required for the sparse matrix. And unlike Ordinary Least Squares, we will not need to perform a matrix inversion in our implementation.

3.2.2 Implementation

The implementation of Gradient Descent was done in a comparable manner to what we did in Ordinary Least Squares. We again used a randomly generated 1000-by-1000 dimensional matrix with a fixed sparsity level of 90%. We conducted a 100-trial simulation to compare the execution times of applying Gradient Descent on the dense matrix as well as its CSR formatting. After that we performed a 250-trial simulation where the sparsity levels varied between 10% and 90% in order to evaluate the execution time difference across a wide range of sparsity levels.

3.2.3 Results

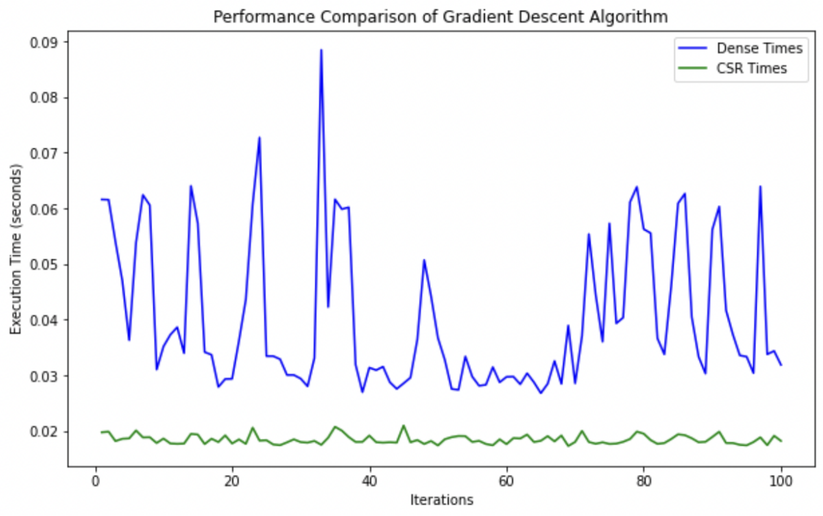

Figure 3: Performance Comparison of Gradient Descent Algorithm

Figure 3 shows that we did in fact achieve a performance speedup with Gradient Descent. We can see that the performance slightly fluctuates due to the fact that we are generating a random linear regression problem in each of the iterations. So we cannot necessarily calculate a consistent average speedup, but it does range between 0.03 seconds to 0.07 seconds. And it remains consistently faster than the performance with a dense matrix.

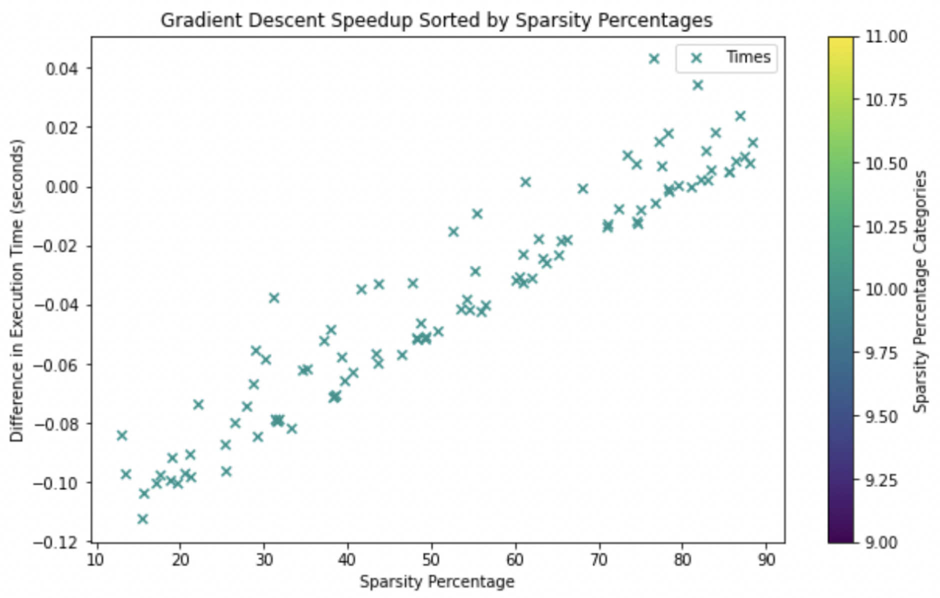

Figure 4: Gradient Descent Algorithm Speedup Sorted by Sparsity Percentages

Figure 4 shows that as our sparsity levels are higher, our computational time improves. However, our matrices need to be extremely sparse in order for Gradient Descent to actually improve in execution time as shown in the graph. This is likely due to the computational savings from processing fewer non-zero elements and also because the overheads in dealing with sparse matrices could potentially outweigh the reduction in arithmetic operations which is what is consistent with what we have seen before. Ultimately, gradient descent can be faster for CSR matrices primarily when these matrices are highly sparse as the computational savings from handling fewer non-zero elements outweigh the overheads and inefficiencies associated with sparse matrix operations. As the sparsity decreases, these benefits diminish and the performance can become worse than that of dense matrix operations.

3.3 Singular Value Decomposition

We will see how we can improve the computation time of Singular Value Decomposition where the matrix is sparse. Singular Value Decomposition itself is a factorization of a matrix into three other matrices: $A = U\Sigma V^T$ where A is the original matrix, U and V are orthogonal matrices containing the left and right singular vectors, respectively. If the original matrix A is sparse, Σ will also be a form of sparse matrix since it is diagonal. U and V are typically dense, even if A is sparse, because the singular vectors usually have non-zero components in all positions. Σ is always a diagonal matrix and can be considered sparse due to its structure. U and V result from orthogonalization and are generally dense. It is worth noting in many practical applications of SVD, especially in dimensionality reduction applications, we want to use low-rank approximations. Therefore, we implement an iterative method, Lanczos algorithm. The point of this is to be able to efficiently approximate the largest eigenvalues and eigenvectors in a low-rank SVD approximation.

3.3.1 Lanczos Algorithm

Lanczos algorithm is an iterative method developed by Cornelius Lanczos and is used to approximate the eigenvalues and eigenvectors of sparse and symmetric matrices. The algorithm starts with a vector and generates a sequence of vectors that form an orthogonal basis for the Krylov subspace. The algorithm applies a process similar to the Gram-Schmidt orthogonalization at each step, ensuring the generated vectors remain orthogonal.

3.3.2 Applying The Algorithm To SVD

So when we perform Lanczos algorithm to compute SVD, we do not directly apply it to the original matrix A but to A^TA (or AA^T), which are symmetric positive semi-definite matrices. Lanczos algorithm then generates a tridiagonal matrix whose eigenvalues approximate those of A^TA (or AA^T).

Our process is as follows:

1. We apply the Lanczos algorithm to the symmetric matrix A^TA. This gives us a tridiagonal matrix T and an orthogonal matrix Q such that $Q^T(A^TA)Q = T$.
2. We solve the tridiagonal eigenvalue problem for T in order to find the eigenvalues and eigenvectors. This allows us to approximate the eigenvalues and eigenvectors of A^TA.
3. We compute the singular of A by taking the square roots of the non-negative eigenvalues of A^TA, and the corresponding eigenvectors of A^TA are the right singular vectors, V.
4. We then compute $U = AV\Sigma^{-1}$ in order to compute the left singular vectors U.

3.3.3 Matrix Operation Analysis

In this algorithm, we deal with a lot of matrix operations and like we saw before, CSR formatting is an efficient storage method and plays a crucial role in optimizing these operations. The first step of SVD is forming A^TA or AA^T, depending on the dimensions of A. This allows us to create a symmetric matrix. By using CSR formatting, our multiplication speed is projected to improve from the sparsity of A which will reduce computational complexity.

To implement Lanczos algorithm, we progressively build a smaller tridiagonal matrix. So for each iteration, we multiply the sparse matrix A^TA (or AA^T) by a vector. If we did this using CSR formatting, these multiplications can be done faster because we significantly reduce the number of operations required. It is also worth noting that Lanczos algorithm maintains orthogonality between the vectors and that means we may have additional vector operations like dot products which could add to the time complexity.

We are then given a tridiagonal matrix which will have a shorter computing time than if we were to perform the algorithm on the original A matrix. Finally, we are left with computing the singular values along with vectors. The only step that is going to be optimized here using sparse matrices is calculating the left singular vectors U. We already would have computed the singular value and right singular vectors. In order to find U, we compute $U = AV\Sigma^{-1}$. If we compute AV with A being sparse, we can improve the multiplication time done in CSR format as it involves fewer operations. The rest of the operations are purely scaling operations that do not involve any optimization.

3.3.4 Implementation

To assess this algorithm, we can compare the runtime of a standard SVD as a standard implementation with a dense matrix to that of our refined SVD that implements Lanczos algorithm with the matrix in CSR format. We also performed a sanity check to compare applying a dense matrix versus a CSR matrix to Lanczos algorithm. We first perform a 100-trial simulation that assesses the time differences between applying Lanczos algorithm with a 1000-by-1000 dimensional dense matrix with 90% sparse data along with its CSR format. We then perform a 250-trial simulation that again assesses the time differences but this time, we also randomize what the level of sparsity is between 10% and 90%. After that, we perform a comparison of the execution time of our standard SVD with a randomly generated 1000-by-1000 dimensional dense matrix and our approximated SVD with Lanczos done with k = 10 iterations using that dense matrix's CSR format.

3.3.5 Results

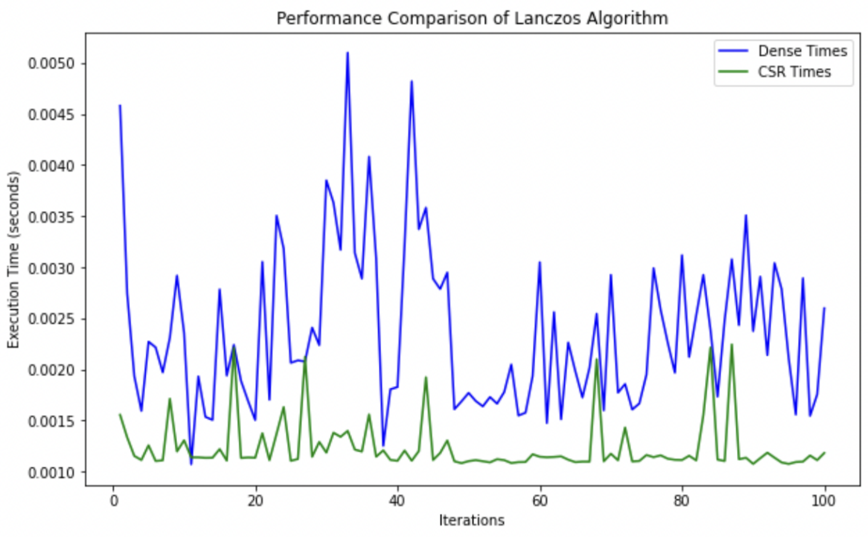

Figure 5: Performance Comparison of Lanczos Algorithm

According to Figure 5, we can see that the Lanczos iterative method clearly has a faster runtime with a CSR matrix to that of its dense matrix representation. We tested this matrix using a 10% density, meaning that 90% of the data is sparse. Of the 100 simulations of Lanczos Algorithm we ran, we saw 98% of the execution time is faster for a CSR representation. While the speedup is not necessarily high, we are only working with only a 1000-by-1000 dimension matrix.

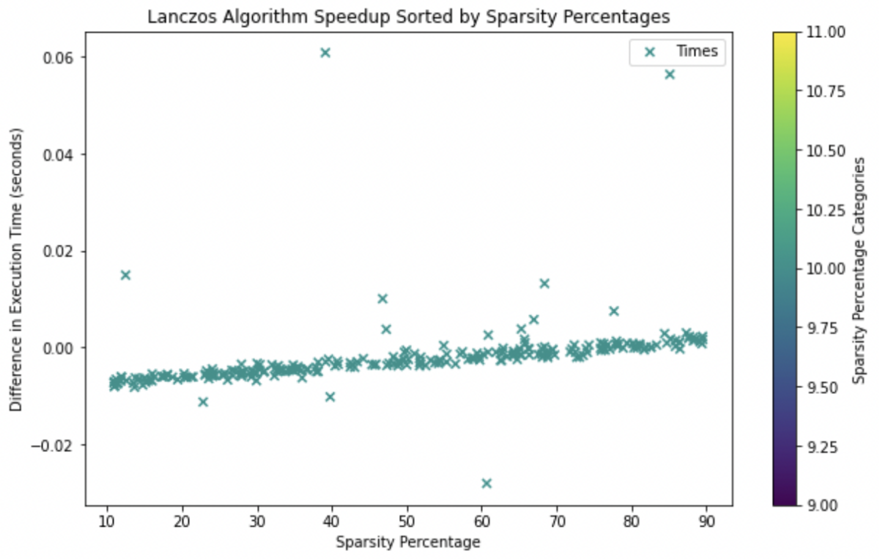

Figure 6: Lanczos Algorithm Speedup Sorted by Sparsity Percentages

According to Figure 6, Lanczos algorithm is shown to speed up only in the case of highly sparse matrices. There is a bit of overhead involved in our computation and if a matrix is only moderately sparse, this overhead diminishes the performance gains from not dealing with sparse data. We can see that in highly sparse matrices, the reduction in numerical operations far outweighs that overhead. It is also possible that the condition number of the matrix gives this matrix the property that it converges faster for highly sparse matrices.

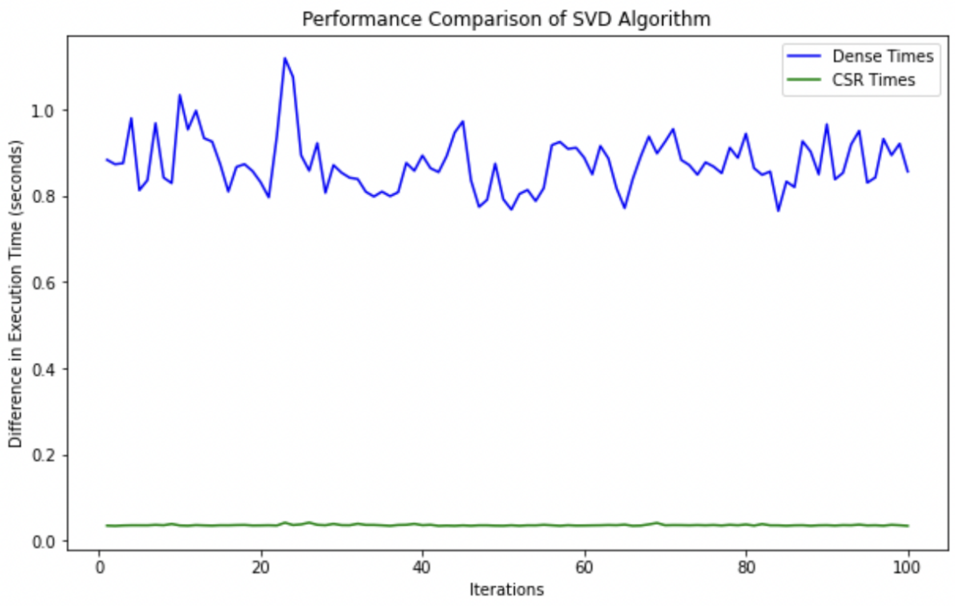

Figure 7: Performance Comparison of SVD with Lanczos Algorithm

According to Figure 7, we can see that applying SVD using Python's built-in SVD method is far slower compared to applying SVD with the Lanczos iterative method. We achieve an average speedup of nearly 0.85 seconds including any computation overhead.

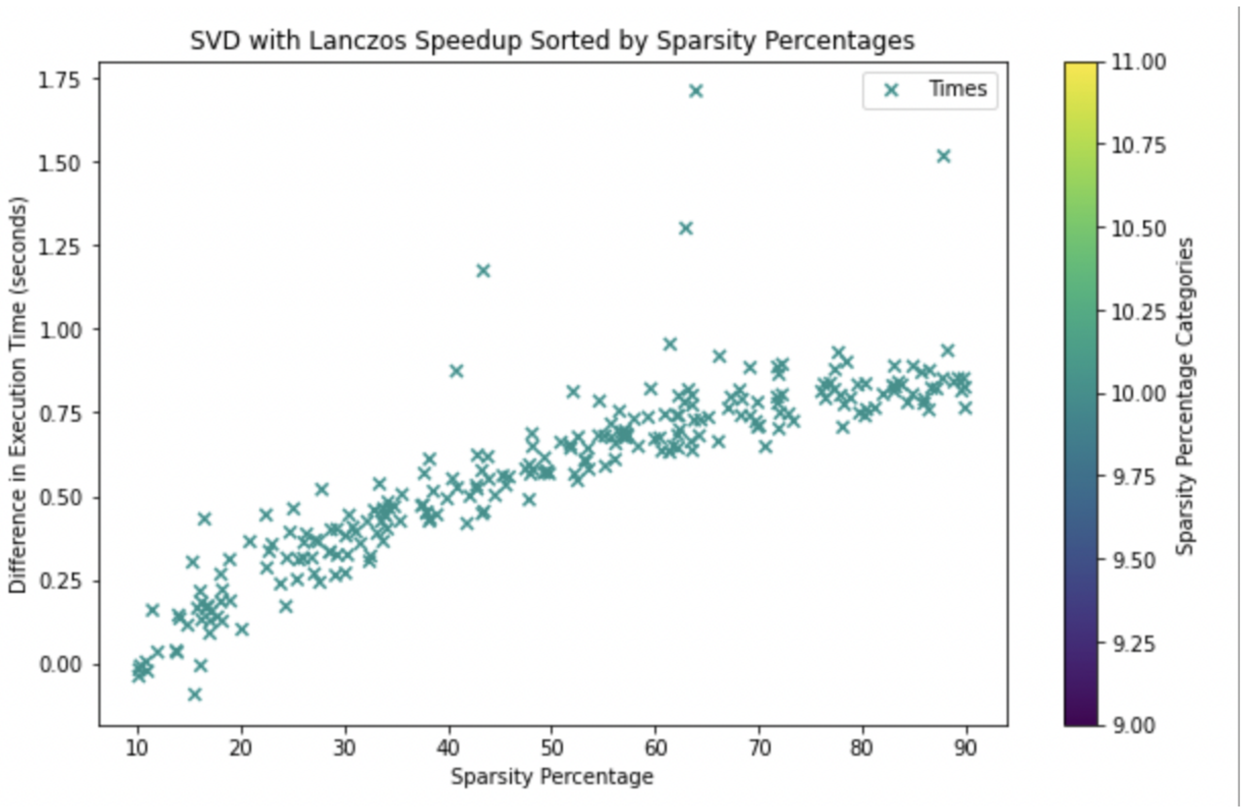

Figure 8: SVD with Lanczos Speedup Sorted By Sparsity Percentages

According to Figure 8, we can see that our simulation of 250 trials ranging from 10% to 90% sparsity demonstrates a speedup time that gradually increases as the level of our sparsity increases. As we have seen in the previous algorithms, this is likely due to the fact that Lanczos algorithm heavily relies on matrix-vector multiplications. In highly sparse matrices, these multiplications are more efficient because they significantly reduce the computational workload. Lower sparse matrices increase the number of calculations required, thereby reducing the computational efficiency.

4 Conclusion

This project attempts to enhance the computational efficiency of matrix operations in a variety of machine learning applications by leveraging the CSR format. By implementing Ordinary Least Squares (OLS), Gradient Descent, and Singular Value Decomposition (SVD) with the Lanczos algorithm, we wanted to see the impact of sparsity on our performance.

The results from our simulations with OLS demonstrated that while matrix operations in CSR format can be optimized based on how sparse the matrix is because of the matrix multiplication, we did not necessarily achieve a speedup due to the overhead computations as well as the fact that matrix inversion on CSR was computationally intensive.

Nevertheless, we can still improve the time of Linear Regression as we saw in Gradient Descent which showed an improvement in execution time when applied to highly sparse matrices. This demonstrates the advantage of sparse matrix operations, particularly iterative algorithms that are heavily using matrix-vector multiplications. We saw that the efficiency gains in Gradient Descent were more evident with increasing sparsity, validating the hypothesis that sparse representations do in fact have computational benefits.

Our most significant finding was seen by implementing Lanczos algorithm for SVD. We saw that it consistently outperformed a standard built-in SVD implementation. The algorithm demonstrated substantial improvement in speedup when our matrices were extremely sparse. That highlights the potential for significant improvements in computational performance when applying sparse matrix techniques to machine learning algorithms.

These findings can serve as the framework for how sparse matrix representations can offer a more efficient advantage compared to practices done by everyday machine learning engineers and data scientists. However, algorithms involving operations like matrix inversion may not necessarily have those benefits.

Nevertheless, our study provides insight in the improvement of performance time and highlights what the future possibilities are and what the future limitations are. In the future, we hope to perform an analysis of how these representations affect accuracy as well as how these representations work for larger and more complex data sets.

5 Code

The simulation can be found here: https://github.com/aneeshdurai/sparse_data_final_project/

6 Credits

I would like to sincerely thank Professor Demmel for allowing me to enroll in this class and the support. I would also like to thank Lewis Pan for the invaluable guidance and support.

7 References

[1] Saad, Y. (2003). Iterative Methods for Sparse Linear Systems.
[2] Bell, N., Garland, M. (2008). Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors.
[3] Golub, G. H., Van Loan, C. F. (2013). Matrix Computations.
[4] Mahoney, M. W., Drineas, P. (2009). CUR matrix decompositions for improved data analysis.